RAYMOND DOUGLAS BROWNRIGG.

DYNAMIC PROGRAMMING AS A SCHEDULING TOOL IN MULTIPROGRAMMED COMPUTING SYSTEMS

Submitted for the degree of DOCTOR OF PHILOSOPHY in INFORMATION SCIENCE at the Victoria University of Wellington, WELLINGTON, NEW ZEALAND.

DECEMBER 1978

ACKNOWLEDGEMENT.

I would like to acknowledge my original supervisor, Dr. B.A. Murtagh, for the inspiration resulting in the research described in Chapter 2 of this thesis, and his successor, Dr. J.H. Hine, for his encouragement and support given during the preparation of this thesis.

DYNAMIC PROGRAMMING AS A SCHEDULING TOOL IN MULTIPROGRAMMED COMPUTING SYSTEMS.

ABSTRACT

A potentially parallel iterative algorithm for the solution of the unconstrained N-stage decision problem of Dynamic Programming is developed. This new solution method, known as Variable Metric Dynamic Programming, is based on the use of variable metric minimisation techniques to develop quadratic approximations to the optimal cost function for each stage. The algorithm is applied to various test problems, and a comparison with an existing similar algorithm proves favourable. The Variable Metric Dynamic Programming solution method is used in the implementation of an adaptive high-level scheduling mechanism on a multiprogrammed computer in a university environment. This demonstrates a practical application of the new algorithm. More importantly, the application of Variable Metric Dynamic Programming to a scheduling problem illustrates how Mathematical Programming may be used in complex computer scheduling problems to provide in a natural way the required dynamic feedback mechanisms.

KEYWORDS

Dynamic Programming, parallelism, variable metric minimisation, high-level scheduling, adaptive scheduling, multiprogramming, dynamic feedback.

CONTENTS

CONTENTS (continued)

LIST OF FIGURES

LIST OF TABLES

SECTION 1.

INTRODUCTION AND OVERVIEW.

Operations Research is a relatively young science which has nevertheless produced a wealth of results and useful applications in the modern world. Computer Science is an even younger science which has grown rapidly since its inception, and shows no signs of slowing down its growth rate. This thesis discusses research involving a blend of disciplines from these two sciences, namely Dynamic Programming from Operations Research, and Multiprogrammed Scheduling from Computer Science.

Dynamic Programming is an Operations Research technique which has a number of significant applications. However a limiting factor in the practical application of Dynamic Programming to problems of a realistic size has been the large amounts of computing resources required for the implementation, the well-known 'curse of dimensionality' of Dynamic Programming. This research is directed towards the development of an iterative algorithm which, by using Variable Metric minimisation techniques to solve the unconstrained N-stage decision problem of Dynamic Programming, promises to achieve savings in both computation time and high-speed storage compared with the traditional solution algorithm. This new algorithm thus helps to widen the size range of real problems for which Dynamic Programming may be applied as a general solution method.

The new algorithm is demonstrated by applying it to a problem in controlling a multiprogrammed computing system. The use of Dynamic Programming, and in general any form of Mathematical Programming, in computer operating systems is a research direction which shows great potential. This is particularly true for the investigation of dynamic feedback algorithms, since Mathematical Programming techniques provide a means of formalising the feedback mechanisms, which have previously been characterised by ad hoc arguments. The application chosen involves the dynamic control of those scheduling parameters which affect the relative levels of service provided to different classes of batch customers with the goal of providing 'equitable' service, at the same time controlling the degree of multiprogramming in an attempt to help optimise overall performance. These parameters were previously set and modified by the computer operators on a much longer time scale, but with essentially the same goals in mind. This application is an excellent demonstration of how Mathematical Programming may be used for scheduling multiprogrammed computing systems, and of how Dynamic Programming in particular may be used for optimising a composite of two or more performance criteria using dynamic feedback.

The following section begins by presenting the unconstrained discrete-time Dynamic Programming problem, the standard Dynamic Programming solution method, and a brief resume of an iterative solution method, the Differential Dynamic Programming (DDP) solution method. Then the new Variable Metric Dynamic Programming (VMDP) iterative solution method is developed in detail. The algorithm uses quadratic approximations to functions as a method of storing information between iterations, with Variable Metric minimisation techniques being used to generate these approximations. Implementation details are discussed, and then the new solution method is compared with the DDP solution method, which turns out in some respects to be a special case of the former. This means that the new VMDP solution algorithm is able to cope with more complex problems than the DDP solution algorithm. The new algorithm is then proven to be quadratically convergent with one-step convergence for the problem with linear constraints and quadratic criteria (the LQP problem). Results from solving some simple non-LQP problems show that the VMDP algorithm converges faster than other existing algorithms. Further, an analysis of the VMDP and the DDP algorithms shows that computationally the new algorithm is no worse than the DDP algorithm.

As a lead in to a practical application of this new solution algorithm, Section 3 presents a brief survey of Computer Scheduling. The survey concentrates on multiprogrammed scheduling in a uniprocessor environment, although monoprogrammed scheduling and multiprocessor scheduling are both mentioned. Uniprocessor multiprogrammed scheduling is divided into low-level scheduling and high-level scheduling, then each of these is further subdivided into processor scheduling and more general resource scheduling. Adaptive scheduling is given special mention, since this is a relatively new but potentially fruitful discipline. Finally, performance criteria are discussed, since any scheduling implementation must be based on attempting to optimise some performance criterion.

Section 4 then discusses the application of the VMDP algorithm, as proposed in Section 2, to a problem in scheduling a multiprogrammed computing system, and the implementation of this on a batch and interactive computing system in a university environment. The problem studied is a high-level (job-scheduling) problem in which the decisions made affect when batch jobs are started, and how many jobs from each of the different job classes are to be active together.

This is proposed not as a replacement for any part of the existing scheduling mechanism, but as an extension to it. The existing high-level scheduling mechanism, which consists essentially of a set of static, operator settable, scheduling parameters, is described and the proposed extensions, which provide a mechanism for modifying some of these parameters dynamically, are outlined. Then the problem to be solved is formulated in Dynamic Programming terms, and the functions and variables used, both inputs to and outputs from the solution process, are defined in terms of information available from or required by the existing scheduling mechanism. Further details of the specification of the Dynamic Programming problem are then discussed, along with details of how the results of the solution process are applied on a dynamic basis, and how all this is incorporated into the existing operating system.

Experimental data collected to test the effectiveness of using the extended scheduling mechanism are presented and analysed. The analysis concludes that the implementation has been successful in providing improvements in performance in those areas with which the chosen composite optimality criterion is concerned. In particular, a small but significant improvement in processor utilisation is achieved as well as larger improvements in the predictability of the relative service delivered to the different classes of batch jobs.

Section 5 consists of a summary of the main results and findings of the research performed, and a discussion on the implications of these for further research. This is followed by a bibliography, consisting mainly of references for the survey of section 3.

SECTION 2.

A PARALLEL VARIABLE METRIC DYNAMIC PROGRAMMING ALGORITHM.

2.1 INTRODUCTION.

The N-stage decision problem of Dynamic Programming is concerned with a system which at any instant may be described by a vector, known as the state vector, and a set of N decisions, each of which is specified by a vector, known as the control, or decision, vector. The problem is to determine the optimal sequence of N decisions which transforms the system from an initial given state (at time 0), to a generally unknown final state at time N. The way in which a decision affects the state of the system during a transition from one stage to the next is exactly determined by a transformation function, which specifies, for each stage, the new state of the system, as a function of the current state and the decision applied at that stage. The problem may also have further constraints imposed, in the form of limits on the values of the state and control vectors. The optimality of the solution is based on a cost function which is a sum of functions, one for each stage, each being a function of the state of the system and the decision applied, at that stage.

The standard solution method for this type of Dynamic Programming problem involves, at each stage, selecting a number of discrete values for each component of the state and control vectors, and, for each different value of the state vector, calculating the cost of every possible decision which could be taken from that state. This results in an algorithm whose computational requirements vary in proportion to d^{n+m}, where d is the number of different values of each component of the state and control vector, and n, m are respectively the dimensionalities of the state and control vectors. This gives rise to the so-called 'curse of dimensionality' of Dynamic Programming, whereby problems that are solvable in theory may be just too large to be handled by the available computing resources. This failing of the standard solution method has resulted in a search for other solution methods, usually iterative, which are not as prone to the dimensionality problem.

The use of Variable Metric minimisation techniques in solving the unconstrained version of this problem promises to achieve savings in both computation time and high-speed storage compared with the standard algorithm, at the same time alleviating the 'curse of dimensionality'. The algorithm proposed, which is essentially an iterative second-order gradient method, has the property of finite convergence for the LQP problem, and involves the generation of a quadratic approximation to the optimal cost function as a function of the state vector at each stage. Variable Metric minimisation techniques [107, 108] are used to generate the information necessary to make this quadratic approximation to the cost function in a region around a nominal (non-optimal) trajectory. The quadratic information is then used to update the trajectory in such a way that an overall reduction in the cost function is achieved. The particular Variable Metric method used is that which involves a symmetric rank-1 update formula, which allows the generation of quadratic information without actually performing a minimisation at each step. Furthermore, the implementation of this particular method results in an inherently parallel algorithm which is therefore all the more powerful.

A similar iterative second-order gradient method, known as Differential Dynamic Programming, has been proposed by Jacobson and Mayne [87, 102], and this turns out in some senses to be a special case of the new algorithm, for the discrete time version. It must be noted here however that the Differential Dynamic Programming solution method has been extended to the continuous time problem, whereas the new algorithm is at present considered only in the context of discrete time decisions. In this section, the differences between and the similarities of the two algorithms are outlined, as well as possible variations for the new algorithm.

The subscript and superscript notation used for this section is defined as follows

$V_k(\underline{x}_k, \underline{u}_k)$ is a function of two variables, defined at time k.

$V_x^k(\underline{x}_k, \underline{u}_k)$ is the first partial derivative of this function with respect to the variable $\underline{x}_k$.

$V_{xu}^k(\underline{x}_k, \underline{u}_k)$ is the second partial derivative of the function with respect to $\underline{x}_k$ and $\underline{u}_k$.

2.2 BASIS OF THE METHOD.

2.2.1 The Problem and the Dynamic Programming Formulation.

The unconstrained N-stage decision problem is presented as follows

$$\text{find} \quad \hat{V}_0(\underline{x}_0) = \underset{\{\underline{u}_0,\ldots,\underline{u}_{N-1}\}}{\text{minimum}} \left\{ \sum_{k=0}^{N-1} L_k(\underline{x}_k, \underline{u}_k) + F(\underline{x}_N) \right\} , \qquad (2.1)$$

the corresponding sequence of controls $\{\underline{\hat{u}}_0, \underline{\hat{u}}_1, \ldots, \underline{\hat{u}}_{N-1}\}$,

and the corresponding trajectory $\{\underline{\hat{x}}_0, \underline{\hat{x}}_1, \ldots, \underline{\hat{x}}_N\}$,

where $\underline{\hat{x}}_0 = \underline{c}$ and $\underline{\hat{x}}_{k+1} = \underline{f}_k(\underline{\hat{x}}_k, \underline{\hat{u}}_k)$,

with $\underline{x}_k = (x_k^1, x_k^2, \ldots, x_k^n)$ and $\underline{u}_k = (u_k^1, u_k^2, \ldots, u_k^m)$,

the circumflex '^' denoting optimal values.

Application of the Principle of Optimality [87] results in the Dynamic Programming iterative equation

$$\hat{V}_k(\underline{x}_k) = \underset{\underline{u}_k}{\text{minimum}} \; L_k(\underline{x}_k, \underline{u}_k) + \hat{V}_{k+1}(\underline{f}_k(\underline{x}_k, \underline{u}_k)) ,$$

for $k = 0, 1, \ldots, N-1$, (2.2)

with the boundary condition

$$\hat{V}_N(\underline{x}_N) = F(\underline{x}_N) .$$

2.2.2 The Standard Solution Method.

The first step of the standard solution method for the above problem involves the discretisation of each component of the control vector, and each component of the state vector (if these are not already discrete-valued). The iterative equation is then solved for $k = N-1, N-2, \ldots, 0$, evaluating and storing $\hat{V}_k(\underline{x}_k)$ for each of the quantised values of $\underline{x}_k$. Each of these evaluations involves the computation of the expression

$$V_k(\underline{x}_k, \underline{u}_k) = L_k(\underline{x}_k, \underline{u}_k) + \hat{V}_{k+1}(\underline{f}_k(\underline{x}_k, \underline{u}_k)) \tag{2.3}$$

for each of the quantised values of $\underline{u}_k$, and determining the minimum. The values of $\hat{V}_{k+1}(.)$ are determined by interpolating between the stored values of $\hat{V}_{k+1}(\underline{x}_{k+1})$ from the calculations for the previous value of k. For each value of $\hat{V}_k(\underline{x}_k)$ stored, the corresponding minimising control (denoted by $\hat{\underline{u}}_k(\underline{x}_k)$) must also be stored. The optimal cost is then simply $\hat{V}_0(\underline{c})$, and the corresponding sequence of controls and trajectory are found from the equations

$$\begin{aligned} \hat{\underline{u}}_k &= \hat{\underline{u}}_k(\hat{\underline{x}}_k) , \\ \hat{\underline{x}}_{k+1} &= \underline{f}_k(\hat{\underline{x}}_k, \hat{\underline{u}}_k) , \\ \hat{\underline{x}}_0 &= \underline{c} , \end{aligned} \tag{2.4}$$

where the evaluation of $\hat{\underline{u}}_k(\hat{\underline{x}}_k)$ may involve interpolation between the stored values of $\hat{\underline{u}}_k(\underline{x}_k)$.

2.2.3 The Differential Dynamic Programming Solution Method.

The Differential Dynamic Programming method of solution is iterative and hence requires a nominal sequence of controls, denoted by $\{\bar{\underline{u}}_0, \bar{\underline{u}}_1, \ldots, \bar{\underline{u}}_{N-1}\}$, from which is calculated a nominal trajectory denoted by $\{\bar{\underline{x}}_0, \bar{\underline{x}}_1, \ldots, \bar{\underline{x}}_N\}$, using the equations

$$\begin{aligned} \bar{\underline{x}}_0 &= \underline{c} , \\ \bar{\underline{x}}_{k+1} &= \underline{f}_k(\bar{\underline{x}}_k, \bar{\underline{u}}_k) . \end{aligned} \tag{2.5}$$

The nominal cost for this sequence of controls is calculated from the expression

$$\bar{V}_0(\bar{\underline{x}}_0) = \sum_{k=0}^{N-1} L_k(\bar{\underline{x}}_k, \bar{\underline{u}}_k) + F(\bar{\underline{x}}_N) \; . \qquad (2.6)$$

The next step, the first of the iterative process, involves the calculation of the parameters $\underline{\alpha}_k$ and β_k for $k = N-1, N-2, \ldots, 0$ from the recursive set of equations

$$\underline{\alpha}_k = -C_k^{-1}.H_u^k \quad , \quad \beta_k = -C_k^{-1}.B_k \quad , \qquad (2.7)$$

where $H_k(\underline{x}_k, \underline{u}_k, \underline{\lambda}) = L_k(\underline{x}_k, \underline{u}_k) + \underline{\lambda}^T.\underline{f}_k(\underline{x}_k, \underline{u}_k)$,

$$A_k = H_{xx}^k(\bar{\underline{x}}_k, \bar{\underline{u}}_k, V_x^{k+1}(\bar{\underline{x}}_{k+1})) + (f_x^k)^T.V_{xx}^{k+1}(\bar{\underline{x}}_{k+1}).f_x^k \; ,$$

$$B_k = H_{ux}^k(\bar{\underline{x}}_k, \bar{\underline{u}}_k, V_x^{k+1}(\bar{\underline{x}}_{k+1})) + (f_u^k)^T.V_{xx}^{k+1}(\bar{\underline{x}}_{k+1}).f_x^k \; ,$$

$$C_k = H_{uu}^k(\bar{\underline{x}}_k, \bar{\underline{u}}_k, V_x^{k+1}(\bar{\underline{x}}_{k+1})) + (f_u^k)^T.V_{xx}^{k+1}(\bar{\underline{x}}_{k+1}).f_u^k \; ,$$

$$V_x^k = H_x^k(\bar{\underline{x}}_k, \bar{\underline{u}}_k, V_x^{k+1}(\bar{\underline{x}}_{k+1})) + \beta_k^T.H_u^k(\bar{\underline{x}}_k, \bar{\underline{u}}_k, V_x^{k+1}(\bar{\underline{x}}_{k+1})) \; ,$$

and $V_{xx}^k = A_k - \beta_k^T.C_k.\beta_k$.

with the boundary conditions

$$V_x^N(\bar{\underline{x}}_N) = F_x(\bar{\underline{x}}_N) \; ,$$

and $V_{xx}^N(\bar{\underline{x}}_N) = F_{xx}(\bar{\underline{x}}_N)$,

all unspecified arguments being $\bar{\underline{x}}_k$, $\bar{\underline{u}}_k$.

The second step of the iterative process involves calculating the new trajectory and sequence of controls from the equations

$$\begin{aligned} \delta\underline{u}_0 &= \varepsilon\underline{\alpha}_0 \; , \\ \delta\underline{u}_k &= \varepsilon\underline{\alpha}_k + \beta_k\delta\underline{x}_k \; , \\ \delta\underline{x}_{k+1} &= \underline{f}_k(\bar{\underline{x}}_k + \delta\underline{x}_k, \bar{\underline{u}}_k + \delta\underline{u}_k) - \bar{\underline{x}}_{k+1} \; . \end{aligned} \qquad (2.8)$$

where $\varepsilon > 0$ is a scalar required to ensure that the quadratic information inherent in V_x^k and V_{xx}^k is accurate enough. The scalar ε, where $0<\varepsilon\leq 1$, limits the magnitude of the departure of the new trajectory from the nominal trajectory (for which V_x^k and V_{xx}^k are calculated). The newly generated sequence of controls is then taken as a nominal sequence, and the iterative process is repeated.

2.2.4 The Variable Metric Solution Method.

The Variable Metric method of solution is also an iterative process, requiring a nominal sequence of controls from which a nominal trajectory and nominal cost are calculated as in equations 2.5 and 2.6 for the Differential Dynamic Programming method. Then for each iteration of the process, a new cost function is developed and saved in the form of a quadratic approximation for each stage k, where $k = 0, 1, \ldots, N$. These are then used to generate a new nominal sequence of controls and a new nominal trajectory for the next iteration. This process is repeated until some criterion for convergence is satisfied.

Given the nominal sequence of controls $\{\underline{\bar{u}}_0, \underline{\bar{u}}_1, \ldots, \underline{\bar{u}}_{N-1}\}$, the nominal trajectory $\{\underline{\bar{x}}_0, \underline{\bar{x}}_1, \ldots, \underline{\bar{x}}_N\}$, and the nominal cost $\bar{V}_0(\underline{\bar{x}}_0)$, a new cost function, namely

$$I_k^*(\underline{x}_k) = L_k(\underline{x}_k, \underline{u}_k^*(\underline{x}_k)) + I_{k+1}^*(\underline{f}_k(\underline{x}_k, \underline{u}_k^*(\underline{x}_k))) \quad , \tag{2.9}$$

with the boundary condition

$$I_N^*(\underline{x}_N) = F(\underline{x}_N) \tag{2.10}$$

is determined as a quadratic approximation around the point $\underline{\bar{x}}_k$. Note that since $\underline{u}_k^*$ is a function of $\underline{x}_k$, the function I_k^* is in fact a function of $\underline{x}_k$ only. To define the function $\underline{u}_k^*(\underline{x}_k)$, consider the similar optimal cost function which is generated for the standard solution method, namely

$$\hat{V}_k(\underline{x}_k) = \min_{\underline{u}_k} \{L_k(\underline{x}_k, \underline{u}_k) + \hat{V}_{k+1}(\underline{f}_k(\underline{x}_k, \underline{u}_k))\} \quad , \qquad (2.11)$$

which could also be written as

$$\hat{V}_k(\underline{x}_k) = L_k(\underline{x}_k, \underline{\hat{u}}_k(\underline{x}_k)) + \hat{V}_{k+1}(\underline{f}_k(\underline{x}_k, \underline{\hat{u}}_k(\underline{x}_k))) \quad , \qquad (2.12)$$

where $\underline{\hat{u}}_k(\underline{x}_k)$ denotes the minimising $\underline{u}_k$, which is implicitly a function of $\underline{x}_k$. The differences between the two cost functions are that $\underline{u}_k^*(\underline{x}_k)$ is not a minimising control, but rather a control which tends to minimise the cost function I_k^*, and that the values for $I_{k+1}^*(.)$ are obtained from a quadratic approximation, rather than from interpolation between grid points, as are the values for $\hat{V}_{k+1}(.)$. Thus there are two major parts at each stage k for each iteration of the Variable Metric method, namely the determination of the function $\underline{u}_k^*(\underline{x}_k)$, and the determination of the quadratic approximation to $I_k^*(\underline{x}_k)$ around the point $\underline{\bar{x}}_k$.

Using the boundary condition 2.10, and given the quadratic approximation to $I_{k+1}^*(\underline{x}_{k+1})$ around $\underline{\bar{x}}_{k+1}$, namely

$$\begin{aligned} I^*_{k+1}(\underline{x}_{k+1}) = a_{k+1} + (\underline{x}_{k+1} - \underline{\bar{x}}_{k+1})^T . \underline{g}_{k+1} + \\ \tfrac{1}{2}(\underline{x}_{k+1} - \underline{\bar{x}}_{k+1})^T . H_{k+1} . (\underline{x}_{k+1} - \underline{\bar{x}}_{k+1}) \end{aligned} \qquad (2.13)$$

the function $\underline{u}_k^*(\underline{x}_k)$ is determined as follows. Define a new function $I_k(., .)$ as

$$I_k(\underline{x}_k, \underline{u}_k) = L_k(\underline{x}_k, \underline{u}_k) + I_{k+1}^*(\underline{f}_k(\underline{x}_k, \underline{u}_k)) \quad , \qquad (2.14)$$

noting the similarity between this and the cost function 2.3 in the standard solution method. Now given a fixed value, say $\underline{x}_k^i$, of the state vector at stage k, the function I_k and its derivative

$dI_k/d\underline{u}_k$ are calculated for m+1 different values of $\underline{u}_k$ in a suitable neighbourhood of $\bar{\underline{u}}_k$ (m being the dimensionality of $\underline{u}_k$). Variable Metric techniques are then used to build up an approximation to the inverse hessian of I_k in a neighbourhood of the point $\bar{\underline{u}}_k$, with $\underline{x}_k$ being fixed at $\underline{x}_k^i$. From this, the 'variable metric direction', $\delta\underline{u}_k(\underline{x}_k^i)$ is calculated from the expression

$$\delta\underline{u}_k(\underline{x}_k^i) = -I_{uu}^{-1}(\underline{x}_k^i).I_u(\underline{x}_k^i, \bar{\underline{u}}_k) \quad , \tag{2.15}$$

where I_{uu}^{-1} denotes the inverse hessian. This is the direction that

a Variable Metric minimisation would calculate in attempting to find the minimum of $I_k(\underline{x}_k^i, \underline{u}_k)$ as a function of $\underline{u}_k$, given that the present value of $\underline{u}_k$ is $\bar{\underline{u}}_k$. This then determines one value for the function $\underline{u}_k^*(\underline{x}_k)$, from the expression

$$\underline{u}_k^*(\underline{x}_k^i) = \bar{\underline{u}}_k + \alpha\delta\underline{u}_k(\underline{x}_k^i) \quad , \tag{2.16}$$

where α, $0<\alpha\leq 1$, is required to ensure that the point $\underline{f}_k(\underline{x}_k^i, \underline{u}_k^*(\underline{x}_k^i))$, at which I_{k+1}^* would be evaluated, is close enough to the point $\underline{x}_{k+1}$, around which the quadratic approximation to I_{k+1}^* has been made, for the approximation to be valid. The use of this scalar α has a similar effect to the 'region limiting strategy', reported by Arora and Pierre [9]. This evaluation of $\delta\underline{u}_k(\underline{x}_k^i)$ is repeated for a total of n+1 different values of $\underline{x}_k$, such as $\underline{x}_k^i$, in a suitable neighbourhood of $\bar{\underline{x}}_k$ (n being the dimensionality of $\underline{x}_k$), to generate a linear approximation to the function as

$$\delta\underline{u}_k(\underline{x}_k) = \underline{\alpha}_k + \beta_k^T.(\underline{x}_k - \bar{\underline{x}}_k) + o\|\underline{x}_k - \bar{\underline{x}}_k\|^2 \ . \tag{2.17}$$

This serves to provide a linear approximation to $\underline{u}_k^*(\underline{x}_k)$ from equation

2.16. Note that the vector $\underline{\alpha}_k$ is not related to the scalar α.

We now have the situation where the function $I^*_k(\underline{x}_k)$ in equation 2.9, which is merely the function $I_k(\underline{x}_k, \underline{u}_k)$ evaluated at $\underline{u}_k = \underline{u}^*_k(\underline{x}_k)$, can be evaluated approximately for any value of $\underline{x}_k$. Also, since $I^*_{k+1}(.)$ is a quadratic expression and $\underline{u}^*_k(\underline{x}_k)$ is linear, $dI^*_k/d\underline{x}_k$ may also be evaluated. This is detailed in section 2.3.2.1. The algorithm now evaluates $I^*_k(\underline{x}_k)$ and its derivative $dI^*_k/d\underline{x}_k$ for n+1 different values of $\underline{x}_k$ in a suitable neighbourhood of $\bar{\underline{x}}_k$, and uses Variable Metric techniques to build up gradient and hessian information which will serve to approximate $I^*_k(\underline{x}_k)$ to second order about $\bar{\underline{x}}_k$. The values of $\underline{x}_k$ chosen need not necessarily be the same as those chosen for generating the linear approximation to $\delta\underline{u}_k(\underline{x}_k)$, but some computation time is saved and some accuracy retained if they are the same.

This whole process of generating a quadratic approximation to $I^*_k(\underline{x}_k)$ about $\bar{\underline{x}}_k$ is repeated for k = N-1, N-2, ..., 0, to complete the first step of each iteration. In the second step of the iteration, a new trajectory and sequence of controls are calculated as follows

$$\begin{aligned} \delta\underline{u}_0 &= \varepsilon\underline{\alpha}_0 \quad , \\ \delta\underline{u}_k &= \varepsilon(\underline{\alpha}_k + \beta_k^T\delta\underline{x}_k) \quad , \qquad (2.18) \\ \delta\underline{x}_{k+1} &= \underline{f}_k(\bar{\underline{x}}_k + \delta\underline{x}_k, \bar{\underline{u}}_k + \delta\underline{u}_k) - \bar{\underline{x}}_{k+1} \quad . \end{aligned}$$

Again the scalar ε, $0<\varepsilon\leq 1$, is used to limit the magnitude of departure of the new trajectory from the nominal trajectory so as to ensure the accuracy of the quadratic approximation, thereby resulting in a decrease in the value of the cost function. Note the expression used for $\delta \underline{u}_k$ is slightly different from that used in the Differential Dynamic Programming method, equations 2.8, this particular expression being chosen for its similarity to expression 2.16 for evaluating the function $\underline{u}_k^*(\underline{x}_k)$, in which the scalar α is used to limit the deviation from the nominal control. The generation of a new trajectory and sequence of controls completes one iteration of the algorithm, at which point a test for convergence is performed if necessary.

The particular Variable Metric update formula used in the algorithm is the symmetric rank-1 update formula as discussed by Murtagh and Sargent [108]. Starting with the identity matrix, a sequence of inverse hessian matrices is generated from the expression

$$S_{i+1} = S_i + (\underline{p}_i - S_i\underline{q}_i).(\underline{p}_i - S_i\underline{q}_i)^T/(\underline{q}_i^T.(\underline{p}_i - S_i\underline{q}_i)) \qquad (2.19)$$

where $\underline{p}_i = \underline{x}_{i+1} - \underline{x}_i$,

and $\underline{q}_i = \underline{g}_{i+1} - \underline{g}_i$,

$\underline{g}_i$ being the gradient of the function of $\underline{x}$ at the point $\underline{x}_i$.

Similarly, a sequence of hessian matrices, resulting in a quadratic approximation, may be generated from the expression

$$H_{j+1} = H_j + (\underline{q}_j - H_j\underline{p}_j).(\underline{q}_j - H_j\underline{p}_j)^T/(\underline{p}_j^T.(\underline{q}_j - H_j\underline{p}_j)) \qquad (2.20)$$

where $\underline{p}_j$ and $\underline{q}_j$ are defined in the same way as $\underline{p}_i$ and $\underline{q}_i$ for 2.19.

It is the use of this symmetric rank-1 update formula that allows the quadratic information to be developed from an arbitrary set of grid points in a neighbourhood of the point of interest. This is in contrast to most Variable Metric update methods, in which each new point considered must be a point which has been generated from the existing quadratic information, and must be some point which is closer to the minimum of the function than all previous points generated, closer being in the sense that the function value is less.

2.3 PROPERTIES OF THE VARIABLE METRIC ALGORITHM.

2.3.1 Flowchart of the Algorithm.

Figure 2.1 indicates the method of the Variable Metric algorithm in flowchart form.

2.3.2 Implementation Details.

2.3.2.1 Calculation of Gradients.

Variable Metric minimisation techniques require that whenever an evaluation of the function to be minimised is carried out, gradient information must also be determined. The Variable Metric Dynamic Programming algorithm, although it does not minimise the functions which are treated with Variable Metric techniques (and hence does not use the actual function values) does require this gradient information in order to construct the hessian for the quadratic approximation. A property of the algorithm is that provided that the necessary functions $L_k(\underline{x}_k, \underline{u}_k)$, $\underline{f}_k(\underline{x}_k, \underline{u}_k)$, and $F(\underline{x}_N)$ have analytic first derivatives, no extra computation in the form of numerical differentiation is necessary to determine the required gradients. It must also be noted that no second derivatives are used in the computations.

In the case of finding the quadratic information for $I_k(\underline{x}_k, \underline{u}_k)$ as a function of $\underline{u}_k$ (in order to determine the direction $\delta\underline{u}_k(\underline{x}_k)$), we have

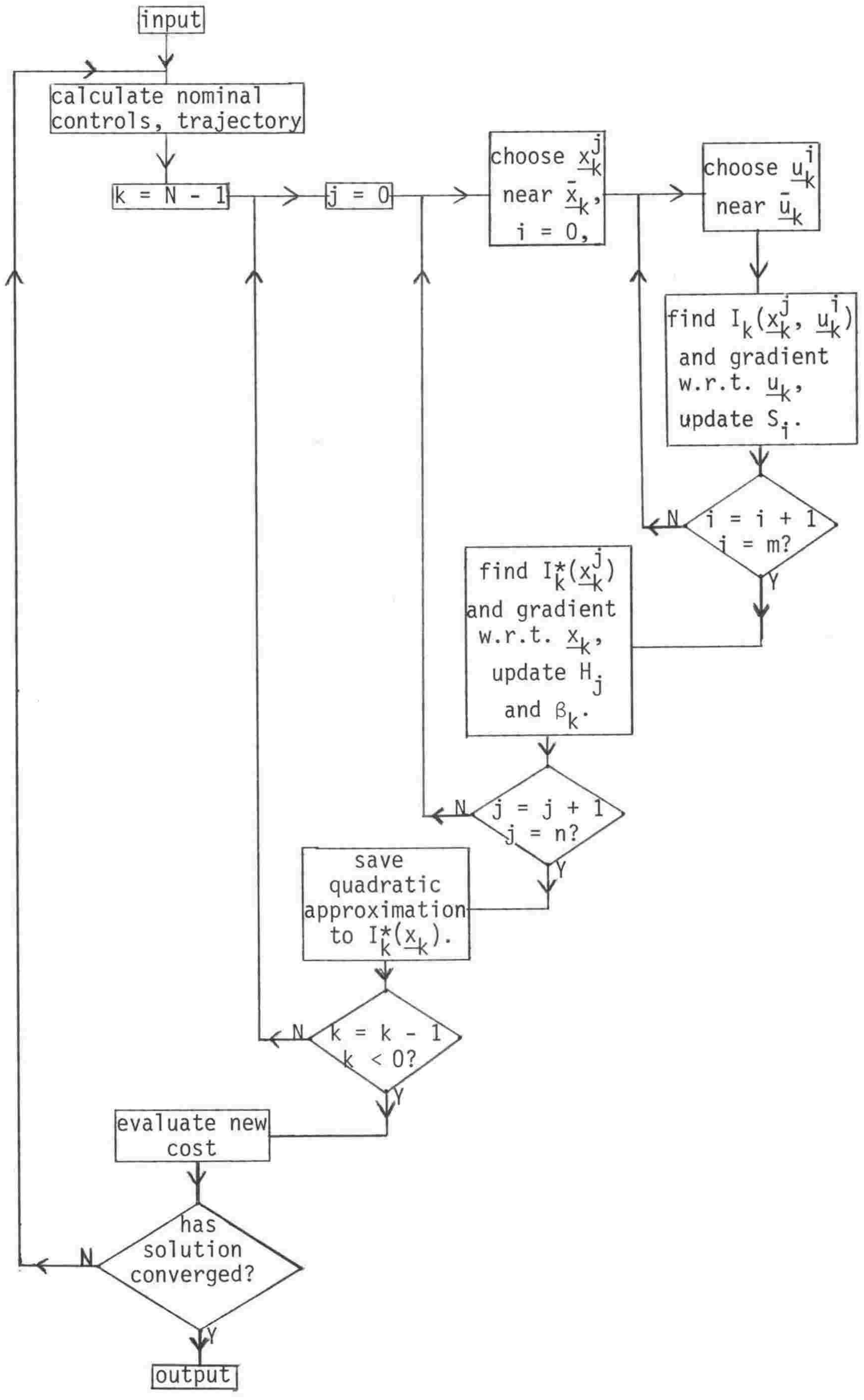

input
calculate nominal controls, trajectory
k = N - 1
j = 0
choose $\underline{x}_k^j$ near $\bar{\underline{x}}_k$, i = 0,
choose $\underline{u}_k^i$ near $\bar{\underline{u}}_k$
find $I_k(\underline{x}_k^j, \underline{u}_k^i)$ and gradient w.r.t. $\underline{u}_k$, update S_i.
i = i + 1 i = m?
N
Y
find $I_k^*(\underline{x}_k^j)$ and gradient w.r.t. $\underline{x}_k$, update H_j and β_k.
j = j + 1 j = n?
N
Y
save quadratic approximation to $I_k^*(\underline{x}_k)$.
k = k - 1 k < 0?
N
Y
evaluate new cost
has solution converged?
N
Y
output

Figure 2.1 Flowchart of the Variable Metric Algorithm.

$$I_k(\underline{x}_k, \underline{u}_k) = L_k(\underline{x}_k, \underline{u}_k) + a_{k+1} + (\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1})^T.\underline{g}_{k+1} +$$
$$\tfrac{1}{2}(\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1})^T.H_{k+1}.(\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1}) +$$
$$o\|\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1}\|^3, \qquad (2.21)$$

resulting in

$$dI_k/d\underline{u}_k = L_u(\underline{x}_k, \underline{u}_k) +$$
$$f_u(\underline{x}_k, \underline{u}_k).(\underline{g}_{k+1} + H_{k+1}.(\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1})) +$$
$$o\|\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1}\|^2, \qquad (2.22)$$

which may be calculated analytically to the required accuracy.

In the case of finding the quadratic information for $I_k^*(\underline{x}_k)$ as a function of $\underline{x}_k$, in order to build the hessian matrix, we have

$$dI_k^*/d\underline{x}_k = I_x^*(\underline{x}_k) + (d\underline{u}_k^*/d\underline{x}_k).I_u^*(\underline{x}_k) \quad . \qquad (2.23)$$

Also, for the linear approximation to $\delta\underline{u}_k$ as a function of $\underline{x}_k$, as in 2.17, we have

$$\delta\underline{u}_k(\underline{x}_k) = \delta\underline{u}_k(\bar{\underline{x}}_k) + (\underline{x}_k - \bar{\underline{x}}_k)^T.\beta_k + o\|\underline{x}_k - \bar{\underline{x}}_k\|^2 \qquad (2.24)$$

where $\beta_k = d\delta\underline{u}_k/d\underline{x}_k(\bar{\underline{x}}_k)$.

Now

$$d\underline{u}_k^*/d\underline{x}_k = \alpha d\delta\underline{u}_k/d\underline{x}_k(\underline{x}_k)$$
$$= \alpha(d\delta\underline{u}_k/d\underline{x}_k(\bar{\underline{x}}_k) + o\|\underline{x}_k - \bar{\underline{x}}_k\|)$$
$$= \alpha.\beta_k + o(\alpha\delta) \quad ,$$

where δ is the length of the steps taken to generate the n+1 different values of $\underline{x}_k$ in a neighbourhood of $\bar{\underline{x}}_k$. Thus we have

$$dI_k^*/d\underline{x}_k(\underline{x}_k) = I_k^*(\underline{x}_k) + \alpha\beta_k.I_u^*(\underline{x}_k) + o(\alpha\delta) \quad , \qquad (2.25)$$

which may be calculated analytically. Note that for the LQP problem, $I_u^*(\underline{x}_k)$ is zero, as is the error term since $\delta\underline{u}_k(\underline{x}_k)$ is linear and hence

$d\delta \underline{u}_k / d\underline{x}_k(\underline{x}_k)$ is in fact constant, that is, it is independent of $\underline{x}_k$.

2.3.2.2 Parameter Settings.

In the Variable Metric algorithm, there are several parameters which must be initialised, and may be altered during the calculations. The most important of these is the parameter α, $0<\alpha\leq 1$, which is used in expression 2.16 to limit the difference between the nominal controls and the newly calculated controls. As required in section 2.3.3.1, this variable must be set to, and remain constant at, the value 1.0 for the one-iteration convergence of an LQP problem. For other problems, other values may be used for the initialisation and further, the parameter may be varied between iterations, and even within a single iteration if desired. A reasonable value for this parameter is $\alpha = 0.5$, although values closer to $\alpha = 1.0$ may be used successfully for problems which are 'almost' LQP problems, such as that discussed in section 2.3.3.2. If α is not kept constant at unity, then there must be some mechanism whereby the value assigned to α tends to 1.0 as the iterative process converges. The reason for this is that when the nominal trajectory is near to the optimal trajectory, then the changes in the trajectory and in the controls from one iteration to the next will be small, provided that the functions used are differentiable. This allows the size of the neighbourhoods chosen around the points on the trajectory and around the nominal controls to be small, resulting in a more accurate quadratic approximation. Now as the approximation becomes more accurate, it becomes more desirable to treat the problem as an LQP problem, and hence the value assigned to α should approach unity. One such mechanism for letting α tend towards unity as the iterative process converges is to give to α the value of the scalar variable ε which results from the previous iteration, since the value of ε needed for a function decrease does in some way reflect the accuracy of the quadratic approximations used. Furthermore, the two scalars α and ε perform similar tasks, both being used to restrict the deviation from the nominal controls, as may be seen by comparing equations 2.16 and 2.18.

The other parameters used are those which determine the sizes of the neighbourhoods around the points $\bar{\underline{u}}_k$ and $\bar{\underline{x}}_k$, which are used respectively for the generation of the direction $\delta\underline{u}_k(\underline{x}_k)$ and the quadratic approximation to the function $I_k^*(\underline{x}_k)$. For an LQP problem, the values used for these parameters are of no analytical consequence, since exact expressions are generated irrespective of the sizes of the neighbourhoods, although numerical accuracy does need to be taken into account when assigning these values. For non-LQP problems, some benefit may be gained from varying the sizes of the neighbourhoods used. The best values that could be used are those which result in neighbourhoods which just contain, at each stage, the new values of the trajectory and control sequence resulting from the current iteration, so that the region in which the quadratic information is appropriate contains the new trajectory. The main difficulty in achieving this lies in not being able to predict future deviations from the current nominal trajectory and sequence of controls. However it would generally be the case that these deviations become smaller as the iterative process converges, which means that the observed deviations resulting from the previous iteration could be used as estimates for the deviations resulting from the current iteration. This still leaves the problem of choosing initial values for these parameters, although since it is not necessary that the neighbourhoods used do contain the new trajectory and sequence of controls, any small value, say 0.01, is likely to be a reasonable choice. In any case, the values of these parameters would tend to be self-regulating if the above method for modifying the parameters is used. If the initial values chosen are too small, then the quadratic approximations would be more accurate than necessary, permitting deviations to occur outside the neighbourhoods, thereby increasing the sizes of the neighbourhoods for the next iteration. Similarly, if the initial choice is too large, then the lack of accuracy in the quadratic approximations would necessitate smaller deviations to obtain an overall cost function decrease, thereby decreasing the sizes of the neighbourhoods for the next iteration. However it is possible that when the initial choices are too large, they may be so much too large that the quadratic information is too inaccurate to result in any cost function decrease, no matter how small ε is chosen. If this does occur, then the parameters must be reduced in size and the iteration repeated. Also it is possible that the use of inaccurate approximations may lead to a non-optimal solution. For this reason it would be better to err on the small side when supplying the initial neighbourhood parameters, since at worst this would tend to involve quadratic information at a point, as does the Differential Dynamic Programming algorithm, rather than in a region around a point.

2.3.3 A Comparison with the Differential Dynamic Programming Algorithm.

The major difference between the two algorithms is that the Variable Metric algorithm uses the scalar α, $0<\alpha\leq1$, in building up the quadratic information for the cost function at each stage, as well as the scalar ε in the second part of each iteration, where the new nominal controls are generated. In addition to, but partly as a result of this, more pertinent gradient information is available for the generation of the hessian matrix for the cost function at each stage. This means that the Variable Metric algorithm is likely to be able to cope with more complex problems than the Differential Dynamic Programming algorithm. However, for the LQP problem, the two algorithms generate and use identical information; in fact the two algorithms are theoretically equivalent for the LQP problem, provided that $\alpha = 1.0$ in the Variable Metric algorithm.

The generation by the Variable Metric algorithm of more stable quadratic approximations in the sense that information is gathered over a region rather than at a point, does occur at the expense of a larger number of function evaluations and floating point multiplications (either of these being useful as a measure of the computer time required to solve a problem), although storage requirements are essentially the same. Also, the inherent parallelism of the new algorithm allows for faster real-time solutions to be obtained.

2.3.3.1 One-Step Quadratic Convergence.

The unconstrained LQP problem has the property that the functions $\underline{f}_k(\underline{x}_k, \underline{u}_k)$, $L_k(\underline{x}_k, \underline{u}_k)$, and $F(\underline{x}_N)$ are respectively linear in $\underline{x}_k$ and $\underline{u}_k$, quadratic in $\underline{x}_k$ and $\underline{u}_k$, and quadratic in $\underline{x}_N$. This results in the existence of a general quadratic recursion formula for the hessian of second derivatives of the cost function as a function of the state vector at each stage. This in turn leads to the one-step convergence of the Variable Metric algorithm, since the cost function may be determined exactly for each stage, provided that the scalar α of expression 2.16 remains constant at unity.

2.3.3.1.1 General Quadratic Recursion Formula for the Hessian of $V_k(\underline{x}_k)$.

Theorem 2.1 Given the properties of the LQP problem, the cost function at each stage is quadratic in the state vector.

The proof is by induction.

Given $\hat{V}_N(\underline{x}_N)$ is quadratic in $\underline{x}_N$.

Assume $\hat{V}_{k+1}(\underline{x}_{k+1})$ is quadratic in $\underline{x}_{k+1}$, that is, assume

$$\hat{V}_{k+1}(\underline{x}_{k+1}) = a_{k+1} + (\underline{x}_{k+1} - \bar{\underline{x}}_{k+1})^T \cdot \underline{g}_{k+1} + \tfrac{1}{2}(\underline{x}_{k+1} - \bar{\underline{x}}_{k+1})^T \cdot H_{k+1} \cdot (\underline{x}_{k+1} - \bar{\underline{x}}_{k+1}) \quad , \tag{2.26}$$

where $a_{k+1} = \hat{V}_{k+1}(\bar{\underline{x}}_{k+1})$,

$\underline{g}_{k+1} = \hat{V}_x^{k+1}(\bar{\underline{x}}_{k+1})$,

and $H_{k+1} = \hat{V}_{xx}^{k+1}$, which is the constant hessian matrix.

Now, from equation 2.2

$$\hat{V}_k(\underline{x}_k) = \underset{\underline{u}_k}{\text{minimum}} \{L_k(\underline{x}_k, \underline{u}_k) + a_{k+1} + (\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1})^T \cdot \underline{g}_{k+1} + \tfrac{1}{2}(\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1})^T \cdot H_{k+1} \cdot (\underline{f}_k(\underline{x}_k, \underline{u}_k) - \bar{\underline{x}}_{k+1})\} \ . \qquad (2.27)$$

Performing the minimisation with respect to $\underline{u}_k$, and dropping the subscript k for convenience of notation,

$$\hat{V}(\underline{x}) = L(\underline{x}, \underline{\hat{u}}) + a_{k+1} + (\underline{f}(\underline{x}, \underline{\hat{u}}) - \bar{\underline{x}}_{k+1})^T \cdot \underline{g}_{k+1} + \tfrac{1}{2}(\underline{f}(\underline{x}, \underline{\hat{u}}) - \bar{\underline{x}}_{k+1})^T \cdot H_{k+1} \cdot (\underline{f}(\underline{x}, \underline{\hat{u}}) - \bar{\underline{x}}_{k+1}) \ , \qquad (2.28)$$

where $\underline{\hat{u}} = \underline{\hat{u}}(\underline{x})$ is a function of $\underline{x}$, which satisfies

$$L_u(\underline{x}, \underline{\hat{u}}) + f_u(\underline{x}, \underline{\hat{u}}) \cdot (\underline{g}_{k+1} + H_{k+1} \cdot (\underline{f}(\underline{x}, \underline{\hat{u}}) - \bar{\underline{x}}_{k+1})) = 0 \ . \qquad (2.29)$$

Differentiating with respect to $\underline{x}$

$$d\hat{V}/d\underline{x} = L_x(\underline{x}, \underline{\hat{u}}) + f_x \cdot (\underline{g}_{k+1} + H_{k+1} \cdot (\underline{f}(\underline{x}, \underline{\hat{u}}) - \bar{\underline{x}}_{k+1})) + d\underline{\hat{u}}/d\underline{x} \cdot (L_u(\underline{x}, \underline{\hat{u}}) + f_u \cdot (\underline{g}_{k+1} + H_{k+1} \cdot (\underline{f}(\underline{x}, \underline{\hat{u}}) - \bar{\underline{x}}_{k+1}))) \ . \qquad (2.30)$$

Substituting from 2.29

$$d\hat{V}/d\underline{x} = L_x(\underline{x}, \underline{\hat{u}}) + f_x \cdot (\underline{g}_{k+1} + H_{k+1} \cdot (\underline{f}(\underline{x}, \underline{\hat{u}}) - \bar{\underline{x}}_{k+1})) \ . \qquad (2.31)$$

Differentiating again, noting that f_x is constant since $\underline{f}$ is linear

$$d^2\hat{V}/d\underline{x}^2 = L_{xx} + f_x \cdot H_{k+1} \cdot f_x^T + d\underline{\hat{u}}/d\underline{x} \cdot (L_{xu} + f_u \cdot H_{k+1} \cdot f_x^T) \ . \qquad (2.32)$$

Differentiating 2.29 with respect to $\underline{x}$, noting that f_u is constant

$$d\underline{\hat{u}}/d\underline{x} = -(L_{xu} + f_u \cdot H_{k+1} \cdot f_x^T)^T \cdot (L_{uu} + f_u \cdot H_{k+1} \cdot f_u^T)^{-1} \ . \qquad (2.33)$$

Thus 2.32 becomes

$$d^2\hat{V}/d\underline{x}^2 = L_{xx} + f_x \cdot H_{k+1} \cdot f_x^T - (L_{xu} + f_u \cdot H_{k+1} \cdot f_x^T)^T \cdot (L_{uu} + f_u \cdot H_{k+1} \cdot f_u^T)^{-1} \cdot (L_{xu} + f_u \cdot H_{k+1} \cdot f_x^T) \qquad (2.34)$$

Now since L is quadratic in $\underline{x}$ and $\underline{u}$, L_{xu}, L_{xx}, and L_{uu} are all constant, hence $d^2\hat{V}/d\underline{x}^2$ is constant, that is, $\hat{V}_k(\underline{x}_k)$ is quadratic in $\underline{x}_k$ with constant hessian matrix

$$H_k = L^k_{xx} + f^k_x . H_{k+1} . f^{k\mathsf{T}}_x - (L^k_{xu} + f^k_u . H_{k+1} . f^{k\mathsf{T}}_x)^{\mathsf{T}} . (L^k_{uu} + f^k_u . H_{k+1} . f^{k\mathsf{T}}_u)^{-1} . (L^k_{xu} + f^k_u . H_{k+1} . f^{k\mathsf{T}}_x) \quad . \tag{2.35}$$

This completes the proof by induction, having developed the recursion formula 2.35 for the hessian of $\hat{V}_k(\underline{x}_k)$.

Note however that for a solution to the problem to exist, the sequence of H_k must be positive semi-definite. A necessary condition for this to occur is that the expression

$$L^k_{uu} + f^k_u . H_{k+1} . f^{k\mathsf{T}}_u$$

be non-singular for each stage k.

2.3.3.1.2 Optimality of $\underline{u}^*_k$ when $\alpha = 1.0$ for the LQP problem.

Theorem 2.2 Given the properties of the LQP problem, and that the direction $\delta\underline{u}_k$ is found from equation 2.15, then the control $\underline{u}^*_k = \bar{\underline{u}}_k + \alpha\delta\underline{u}_k$ is the optimal control for the given $\underline{x}_k$ when $\alpha = 1.0$. Further, the linear function $\delta\underline{u}_k(\underline{x}_k)$ of equation 2.17 produces the optimal policy $\underline{u}^*_k(\underline{x}_k) = \bar{\underline{u}}_k + \delta\underline{u}_k(\underline{x}_k)$.

The proof once again relies strongly on the properties of the functions $L_k(\underline{x}_k, \underline{u}_k)$ and $\underline{f}_k(\underline{x}_k, \underline{u}_k)$.

If $L_k(\underline{x}_k, \underline{u}_k)$ is quadratic in $\underline{u}_k$, then $L_u^k(\underline{x}_k, \underline{u}_k)$ is linear in $\underline{u}_k$. Dropping the subscript k for convenience again, this implies

$$L_u(\underline{x}, \underline{u}) = L_u(\underline{x}, \bar{\underline{u}}) + L_{uu}.(\underline{u} - \bar{\underline{u}}) \quad , \tag{2.36}$$

where L_{uu} is constant.

Similarly,

$$\underline{f}(\underline{x}, \underline{u}) = \underline{f}(\underline{x}, \bar{\underline{u}}) + f_u^T.(\underline{u} - \bar{\underline{u}}) \quad , \tag{2.37}$$

where f_u is constant.

Substituting into equation 2.29

$$L_u(\underline{x}, \underline{u}) + f_u.(\underline{g}_{k+1} + H_{k+1}.(\underline{f}(\underline{x}, \underline{u}) - \bar{\underline{x}}_{k+1})) + (L_{uu} + f_u.H_{k+1}.f_u^T).(\underline{u} - \bar{\underline{u}}) = 0 \quad , \tag{2.38}$$

that is

$$\underline{u} - \bar{\underline{u}} = -(L_{uu} + f_u.H_{k+1}.f_u^T)^{-1}. (L_u(\underline{x}, \underline{u}) + f_u.(\underline{g}_{k+1} + H_{k+1}.(\underline{f}(\underline{x}, \underline{u}) - \bar{\underline{x}}_{k+1}))) \quad . \tag{2.39}$$

This is expression 2.15 for $\delta\underline{u}_k(\underline{x}_k)$ (restoring the subscript k), that is

$$\delta\underline{u}_k(\underline{x}_k) = \hat{\underline{u}}_k(\underline{x}_k) - \bar{\underline{u}}_k \quad . \tag{2.40}$$

Hence $\underline{u}_k^*(\underline{x}_k) = \bar{\underline{u}}_k + \delta\underline{u}_k(\underline{x}_k)$ is the optimal control.

From 2.38, $(\hat{\underline{u}} - \bar{\underline{u}})$ is linear in $\underline{x}$, since L_u and $\underline{f}$ are linear in $\underline{x}$, and L_{uu} and f_u are constant. Thus the function $\delta\hat{\underline{u}}_k(\underline{x}_k)$, defined as

$$\delta\hat{\underline{u}}_k(\underline{x}_k) = \hat{\underline{u}}_k(\underline{x}_k) - \bar{\underline{u}}_k \quad , \tag{2.41}$$

is linear in $\underline{x}_k$. From 2.40, $\delta\underline{u}_k(\underline{x}_k) = \delta\hat{\underline{u}}_k(\underline{x}_k)$ for n+1 values of $\underline{x}_k$, and hence the two functions are identical (since a linear function of a vector of dimension n is uniquely determined by n+1 values). This leads to the required result that $\underline{u}_k^*(\underline{x}_k) = \bar{\underline{u}}_k + \delta\underline{u}_k(\underline{x}_k)$ is the optimal policy function at each stage.

The results of theorems 2.1 and 2.2 combine to prove that the Variable Metric algorithm will converge to the optimal solution in one iteration for the LQP problem, provided that $\alpha = 1.0$. From theorem 2.1 the cost function is determined exactly for each stage (providing a solution exists), and from theorem 2.2 the algorithm will generate optimal controls for the given cost function, and will also build up the optimal policy function for each stage. The application of the optimal policy function will lead to the optimal trajectory after the first iteration of the algorithm.

2.3.3.2 Results from a Simple Non-LQP problem.

To demonstrate the differences between the two algorithms, a simple non-LQP problem was solved using both algorithms, and a comparison made of the results. The problem solved was the following

$$\text{minimise} \sum_{k=0}^{10} \|\underline{x}_k\|^2 + u_k^4 + \|\underline{x}_{10} - \begin{bmatrix}9\\9\end{bmatrix}\|^2 \quad ,$$

$$\text{subject to } \underline{x}_{k+1} = \underline{x}_k + u_k . \begin{bmatrix}1\\1\end{bmatrix} \quad ,$$

$$\text{and} \qquad \underline{x}_0 = \underline{0} \quad .$$

Note that the problem is formulated as two-dimensional in the state variable, although because of symmetry it is essentially a scalar problem. For both algorithms, the nominal control sequence was taken to be

$$\bar{u}_k = 0 \quad , \quad k = 0, 1, \ldots, 9 \quad ,$$

this leading to the nominal trajectory

$$\bar{\underline{x}}_k = \underline{0} \quad , \quad k = 0, 1, \ldots, 10 \quad ,$$

with a nominal cost

$\bar{V}_0 = 162$.

Table 1 shows the convergence of the two algorithms, through the values of the nominal cost $\bar{V}_0$, and the scalar ε producing it.

	Differential Dynamic Programming		Variable Metric Dynamic Programming	
iteration	$\bar{V}_0$	ε	$\bar{V}_0$	ε
nominal	162.000 000		162.000 000	
1	116.753 906	0.25	78.070 810	1.00
2	87.150 405	0.50	73.184 224	1.00
3	77.465 526	1.00	73.174 405 86	1.00
4	73.511 723	1.00	73.174 405 133	1.00
5	73.181 220	1.00	73.174 405 133	
6	73.174 410	1.00		
7	73.174 405 135			
8	73.174 405 133			

Table 2.1 Comparison of the basic algorithms for a simple non-LQP problem.

The Differential Dynamic Programming method of solution was programmed by the author following the algorithm proposed by Jacobson and Mayne (in [87], page 112), with the exception that computation was halted when the relative change in the nominal cost, $\Delta\bar{V}_0/\bar{V}_0$, was less than 10^{-9}, this being the convergence test for the Variable Metric algorithm.

For the Variable Metric algorithm, the parameter α was initialised to the value 0.75, and updated to the value of ε resulting from the previous iteration. The neighbourhood parameters were initialised to $(1 + k)/5$, $k = 0, 1, \ldots, 9$, where k denotes the stage, for each component of the state vector, and the control variable. These were updated to the absolute values of the deviations occuring at the previous iteration, with a minimum of 10^{-6} imposed, this being necessary to ensure a non-zero radius for each of the neighbourhoods, since in particular the deviations occuring at stage $k = 0$ are always zero for the fixed initial value problem. Both algorithms were then varied so that at each iteration, the cost was minimised with respect to the variable ε (see section 2.3.5). Table 2.2 shows the results of applying the two modified algorithms to the

	Differential Dynamic Programming		Variable Metric Dynamic Programming	
iteration	$\bar{V}_0$	ε	$\bar{V}_0$	ε
nominal	162.000 000		162.000 000	
1	113.842 344	0.213 353	78.070 810	1.0
2	84.881 758	0.431 117	73.184 224	1.0
3	74.619 551	0.685 409	73.174 405 74	0.997 010
4	73.204 919	0.945 010	73.174 405 133	1.0
5	73.174 427	1.0	73.174 405 133	1.0
6	73.174 405 135	1.0		
7	73.174 405 133	1.0		

Table 2.2 Comparison of the modified algorithms for a simple non-LQP problem.

same simple non-LQP problem. Tables 2.1 and 2.2 show that in both cases the Variable Metric algorithm shows faster convergence than the Differential Dynamic Programming algorithm.

2.3.3.3 Storage Requirements.

Basic high-speed memory requirements for the Differential Dynamic Programming algorithm and the Variable Metric algorithm are much the same. Given the problem where there are N stages, the dimension of the control vector is m, and the dimension of the state vector is n, the gradient and hessian information for the Variable Metric algorithm requires $2(n + 1)^2$ storage locations, this being the same as for the storage of V_x and V_{xx} for the Differential Dynamic Programming algorithm. In addition, the linear approximation to $\delta \underline{u}_k(\underline{x}_k)$ requires $m(n + 1)$ storage locations for each stage k, which is the same as the combined requirements of $\underline{\alpha}_k$ and β_k for the Differential Dynamic Programming algorithm. Thus the total basic storage requirements for each algorithm is $Nm(n + 1) + 2(n + 1)^2$ locations. Further temporary storage is required for both algorithms, for the temporary vectors used in the Variable Metric minimisation techniques, and for the storage of the matrices A_k, B_k, C_k, and C_k^{-1} for the Differential Dynamic Programming algorithm.

2.3.3.4 Computation Requirements.

Computation requirements may be assessed in two different ways, namely by the number of floating point multiplications, or by the number of function evaluations. The most frequently occurring multiplications in the Variable Metric algorithm occur in the evaluation of the gradient of the function $I_k(\underline{x}_k, \underline{u}_k)$, equation 2.14, which involves the evaluation of the gradient of $I^*_{k+1}(\underline{f}_k(\underline{x}_k, \underline{u}_k))$, this requiring $nm + n^2$ multiplications. The number of floating point multiplications for each stage of each iteration is of order $n(n^2m + nm^2)$ since the evaluation of this gradient is performed m times for each of the n + 1 values of $\underline{x}_k$ in a neighbourhood of $\bar{\underline{x}}_k$. As a comparison, the number of floating point multiplications for the Differential Dynamic Programming algorithm is of order $n^3 + n^2m + nm^2$.

The most frequent function evaluations in the Variable Metric algorithm also occur in finding the gradient of the function $I^*_{k+1}(\underline{f}_k(\underline{x}_k, \underline{u}_k))$, for which the function $f^k_u(\underline{x}_k, \underline{u}_k)$ must be determined. This is a matrix function of order m x n and hence is equivalent to mn scalar function evaluations. If all the functions used are considered as either scalar, vector, or matrix functions, and the number of function evaluations is modified accordingly, then the number of 'scalar equivalent' function evaluations required by the Variable Metric algorithm is of order n^2m^2 for each stage of each iteration. As a comparison, the number of function evaluations required by the Differential Dynamic Programming algorithm is of order $n^3 + n^2m + nm^2$ for each stage of each iteration.

Thus using either measure of computation requirements, the Variable Metric algorithm tends to be a factor of n greater in its requirements than the Differential Dynamic Programming algorithm. This is not as bad as it may seem at first since discrete decision processes tend to be characterised by a low dimensionality in the state vector and a high dimensionality in the control vector. Furthermore there is the possibility of modifying the basic Variable Metric algorithm in such a way that the factor of n in the computational requirements is removed, except for the first iteration. This would occur at the expense of an increase in the high-speed memory requirements (see section 2.3.5) but this is a less critical factor in the light of widespread use of computers with virtual memory systems, wherein the apparent amount of high-speed memory is almost limitless.

A further consideration which is gaining importance in the comparison of algorithms in the light of real time applications is that of parallelism within an algorithm [25]. Basically, if an algorithm can be constructed such that certain parts can be performed independently of others then the execution time can be reduced by the use of a computer which has more than one arithmetic processor, and is capable of parallel processing. Now the Variable Metric Dynamic Programming algorithm is inherently parallel in nature at two levels, firstly in the finding of the gradient of $I_k(\underline{x}_k, \underline{u}_k)$ for m + 1 different values of $\underline{u}_k$, and secondly in the finding of the gradient of $I_k^*(\underline{x}_k)$ for n + 1 different values of $\underline{x}_k$. Thus there is the potential for removing a factor of nm from the computation time requirements of the Variable Metric algorithm, although in practice the reduction would probably be determined by the parallel capacity of the computer itself. It must be appreciated also that the factor of nm would only apply to those parts of the computation which are performed in parallel, and so it would be the case that some other part of the algorithm would dominate the computational requirements. Thus the savings gained by the use of parallel processing are probably in the order of a factor of n in the computational requirements. Further research is needed to determine these savings more accurately.

2.3.4 Other Computational Experience.

As well as a comparison with the Differential Dynamic Programming method of solution, the new algorithm was applied to the solution of another continuous control problem in a discretised form. The problem and its solution using the Sequential Conjugate-Gradient-Restoration algorithm is reported in the papers by Heideman and Levy [77, 78]. The discretised form of the problem is as follows

$$\text{minimise } V_0(\underline{x}_0) = \sum_{k=0}^{N-1} \{1/(N+1) \, \|\underline{x}_k\|^2 + 1/N \, \|\underline{u}_k\|^2\} + 1/(N+1) \, \|\underline{x}_N\|^2 \quad .$$

$$\text{subject to } y_{k+1} = y_k + (v_k - z_k^2 + w_k^2)/N \quad ,$$

$$\text{and} \qquad z_{k+1} = z_k + (w_k - y_k z_k + v_k^2)/N \quad ,$$

$$\text{where } \underline{x}_k = (y_k, z_k) \quad ,$$

$$\underline{u}_k = (v_k, w_k) \quad ,$$

$$\text{and} \quad \underline{x}_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad .$$

The nominal controls were chosen as $\bar{\underline{u}}_k = \underline{0}$, $k = 0, 1, \ldots, N-1$.

The problem was solved for six different values of N, namely 5, 10, 20, 30, 50, and 100, representing six levels of discretisation. In each case, the scalar α was initialised to 0.5 and updated to the value of ε resulting from the previous iteration. The neighbourhood parameters were initialised to 0.01, and updated to the absolute values of the deviations occuring at the previous iteration, with a minimum of 10^{-6} imposed. The convergence condition was taken as when the relative change in cost was less than 10^{-5}. Tables 2.3 and 2.4 show the convergence of the solutions for the six problems in terms of the value of the cost function at each iteration, and the value of ε producing it. The fact that the numbers of iterations required for the six different problems were respectively 7, 7, 6, 5, 5, and 5 suggests that the continuous solution may be approximated as closely as desired by choosing an appropriate value for the number of stages N, with convergence occurring after approximately five iterations.

	N = 5		N = 10		N = 20	
iteration	$\bar{V}_0$	ε	$\bar{V}_0$	ε	$\bar{V}_0$	ε
nominal	2.811 854		2.933 303		3.013 719	
1	2.387 643	0.5	2.566 520	1.0	2.319 798	0.25
2	2.266 878	0.5	2.542 881	0.0625	1.857 106	0.5
3	2.011 984	1.0	2.272 605	0.25	1.803 690	0.5
4	1.874 444	0.5	1.904 407	0.5	1.802 247	1.0
5	1.847 300	1.0	1.818 281	1.0	1.802 209	1.0
6	1.846 903	1.0	1.817 339	1.0	1.802 209	1.0
7	1.846 900	1.0	1.817 328	1.0		

Table 2.3 Convergence of the Variable Metric algorithm for N = 5, N = 10, and N = 20.

	N = 30		N = 50		N = 100	
iteration	$\bar{V}_0$	ε	$\bar{V}_0$	ε	$\bar{V}_0$	ε
nominal	3.044 703		3.071 332		3.092 506	
1	2.236 377	0.25	2.191 443	0.25	2.108 423	0.25
2	1.821 961	0.5	1.805 858	1.0	1.800 551	1.0
3	1.798 459	0.5	1.793 429	1.0	1.791 470	1.0
4	1.797 155	1.0	1.793 110	1.0	1.790 086	1.0
5	1.797 153	1.0	1.793 110	1.0	1.790 081	1.0

Table 2.4 Convergence of the Variable Metric algorithm for N = 30, N = 50, and N = 100.

Figure 2.2 shows, by comparing the converged trajectories obtained as solutions to three of the problems in the discretised form, that increasing the number of stages N does indeed result in a closer approximation to the continuous solution, the trajectory shown for N = 100 being the same as that for the solution of the continuous problem as obtained by Heideman and Levy, within the accuracy of the diagram. As reported by Heideman and Levy, the solution to the continuous problem obtained from the Sequential Conjugate-Gradient-Restoration algorithm requires a total of 13 gradient iterations and (within these) 17 restoration iterations. Thus the Variable Metric Dynamic Programming algorithm compares favourably with respect to the number of iterations required.

2.3.5 Possible Extensions to the Algorithm.

One possible extension to the algorithm is that which is

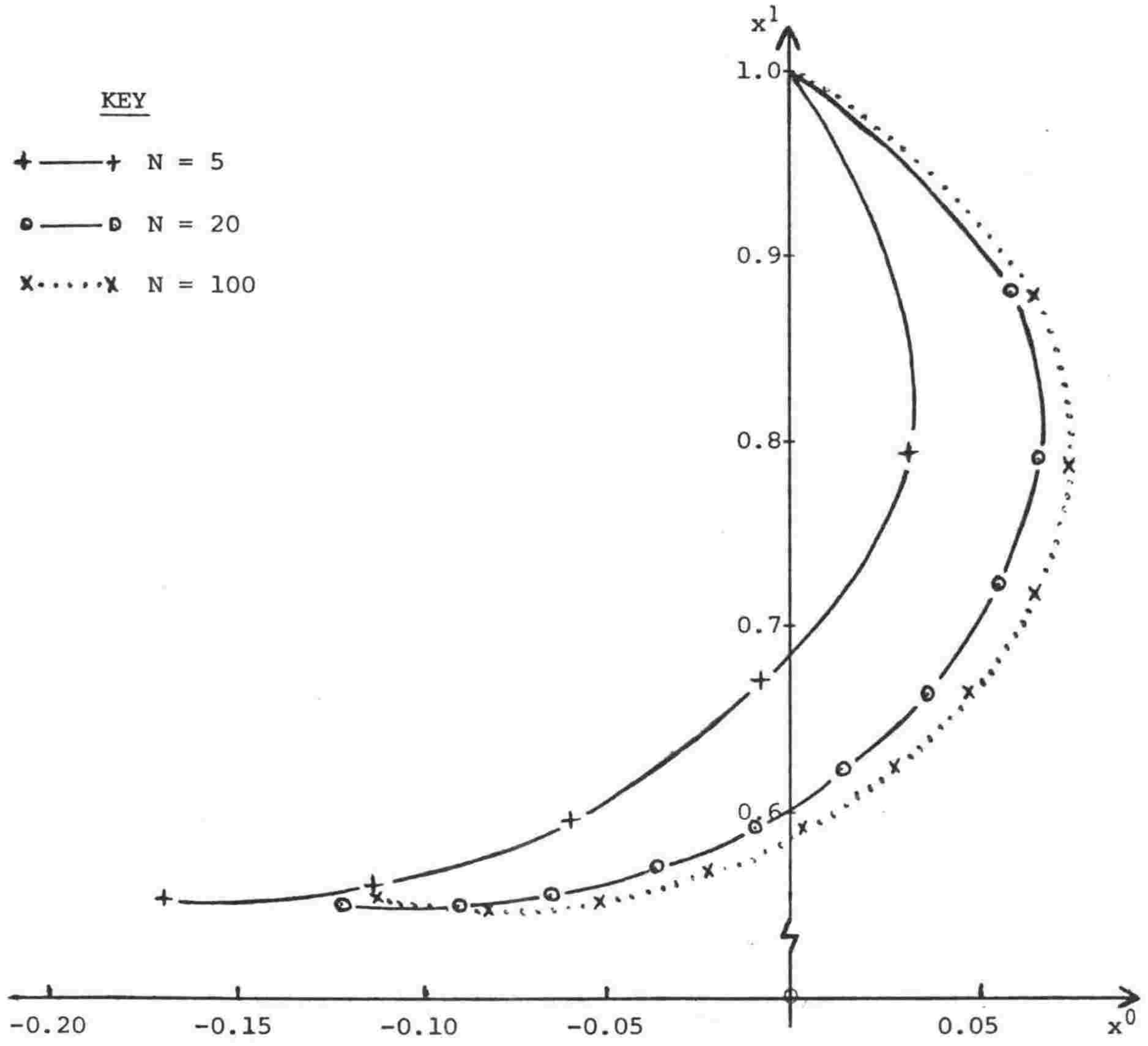


Figure 2.2 Converged trajectories for three of the solutions.

mentioned by Jacobson and Mayne as an extension to the Differential Dynamic Programming algorithm. This involves the minimisation of the actual cost function with respect to the scalar ε in the second part of the iteration where the new sequence of controls is calculated using equations 2.18. This extension is used in the problem referred to in section 2.3.3.2.

A second, potentially more fruitful, extension is that mentioned in section 2.3.3.4, whereby the quadratic approximation to $I_k^*(\underline{x}_k)$ is saved at each iteration for each value of the stage k, and each hessian matrix is updated according to the change in trajectory caused by changes in the sequence of controls generated from the previous iteration. It is likely however that for the first iteration the complete process would have to be carried out in order to generate a good first approximation to the hessian matrix, in contrast to the traditional Variable Metric minimisation method of initialising the hessian matrix to the identity matrix. Thus in the first iteration, the gradient of $I_k^*(\underline{x}_k)$ would be calculated for n + 1 different values of $\underline{x}_k$ in a neighbourhood of $\bar{\underline{x}}_k$, whereas in subsequent iterations, the gradient needs only to be calculated for the new $\bar{\underline{x}}_k$, with the change in trajectory from the previous iteration being used as the step in $\underline{x}_k$ needed to update the hessian. This extension would remove a factor of n from the computational requirements of the algorithm, as well as the parallelism at the outer level. However the inherent parallelism of order m would still remain at the inner level, that of finding the gradient of $I_k(\bar{\underline{x}}_k, \underline{u}_k)$ for m + 1 different values of $\underline{u}_k$ in a neighbourhood of $\bar{\underline{u}}_k$.

A third possibility is the extension of the algorithm to include allowance for constraints on the control variables. The implementation of this into the algorithm would follow the method by which constraints are introduced into a Variable Metric minimisation in which a rank-1 update formula is used. Finally, it may be possible to extend the algorithm to cater for the continuous Dynamic Programming problem, but as yet this has not been investigated at all.

2.4 CONCLUSIONS.

The use of Variable Metric minimisation techniques in an algorithm for solving the N-stage decision problem of Dynamic Programming results in a powerful solution method, which is capable of taking advantage of a parallel processing computing system. The algorithm presented compares favourably, in terms of rate of convergence and range of applications, with the Differential Dynamic Programming algorithm for the discrete time problem, which in some senses turns out to be a special case of the former. This favourable comparison is partly a result of the generalisation itself, and partly due to considerations which are analogous to the result that Variable Metric minimisation of functions compares favourably with function minimisation using Newton's method of second derivatives. Possible extensions to the algorithm promise an even better comparison, although the level of parallelism would be reduced.

SECTION 3.

SURVEY ON COMPUTER SCHEDULING.

3.1 INTRODUCTION.

The theory and practice of scheduling is a wide and diverse field which seems to have its origins in the early 1950's with the development of the study of the theory of 'job-shop scheduling' in the realm of manufacturing. It was the advent of multiprogramming and multiprocessing capabilities in computers that was responsible for the upsurge of interest and diversification in the field of computer scheduling which occurred in the early 1960's.

The field of computer scheduling can be subdivided into three broad areas of research, namely Sequencing, Monoprogrammed Scheduling, and Multiprogrammed Scheduling. A fourth area, Performance Analysis, could be regarded as an integral part of each of the other three, although it is often studied in its own right. Sequencing, also known as Deterministic Scheduling, is the study of that class of problems which requires the determination of the order of processing of a predetermined set of jobs for which all necessary characteristics are known in advance. Monoprogrammed Scheduling is concerned with scheduling jobs on the basis of only one job being active at any one time, with each job, once activated, running to completion before any other job may be activated. This is in contrast to Multiprogrammed Scheduling, in which several jobs may be active, and thus partially completed, at any one time. Multiprogrammed Scheduling can be subdivided into Uniprocessor Scheduling and Multiprocessor Scheduling, this being determined by whether the computing system being considered has only one or more than one Arithmetic Processor, or Central Processing Unit (CPU). Each of these may be further subdivided into High-level and Low-level scheduling. High-Level Scheduling is concerned with choosing from all jobs which are waiting to be processed that job which should be activated next, and when. Low-Level Scheduling, on the other hand, is concerned with choosing, from the set of active jobs, which should be using the processor(s) at any given instant. Performance Analysis has received an upsurge in interest in the past few years, being not only concerned with measurement and evaluation of the effects of scheduling strategies, but with all aspects of system performance. Finally, there is a research discipline which applies equally to Uniprocessor and Multiprocessor, High-Level and Low-Level scheduling, and therefore could be considered as a further subfield of Multiprogrammed Scheduling itself. This is Adaptive Scheduling, which is concerned with scheduling strategies which are able to adapt to dynamically changing conditions, such as amount of work waiting, structure of the workload, structure of individual jobs, and hardware availability. In general, Adaptive Scheduling is characterised by the use of ad hoc techniques for providing the feedback information. However the use of Mathematical Programming ideas offers a way of formalising the feedback mechanisms of Adaptive Scheduling. This offers a strong challenge, and new hopes for the future for practitioners in this field.

3.1.1 Definitions.

To avoid misunderstandings, it is useful to define several terms which are used frequently in this survey.

A task is a piece of sequential code which is the smallest unit of work which may compete for resources.

A process is made up of several sequential tasks, designed to produce some specified result.

A job is a collection of parallel and/or sequential processes which represents a unit of customer work, and is the largest unit of work which may be considered by a high-level scheduler.

A mix of processes is that set of processes which have at any time had some processing done on them, but are not completed. This includes those processes currently assigned to the processor(s), those waiting for a processor, and those waiting for some other event, such as the completion of an I/O operation.

The following definitions all refer to different states of processes in a multiprogrammed system. The relationship between these states and example reasons for transitions between states are shown in Figure 3.1.

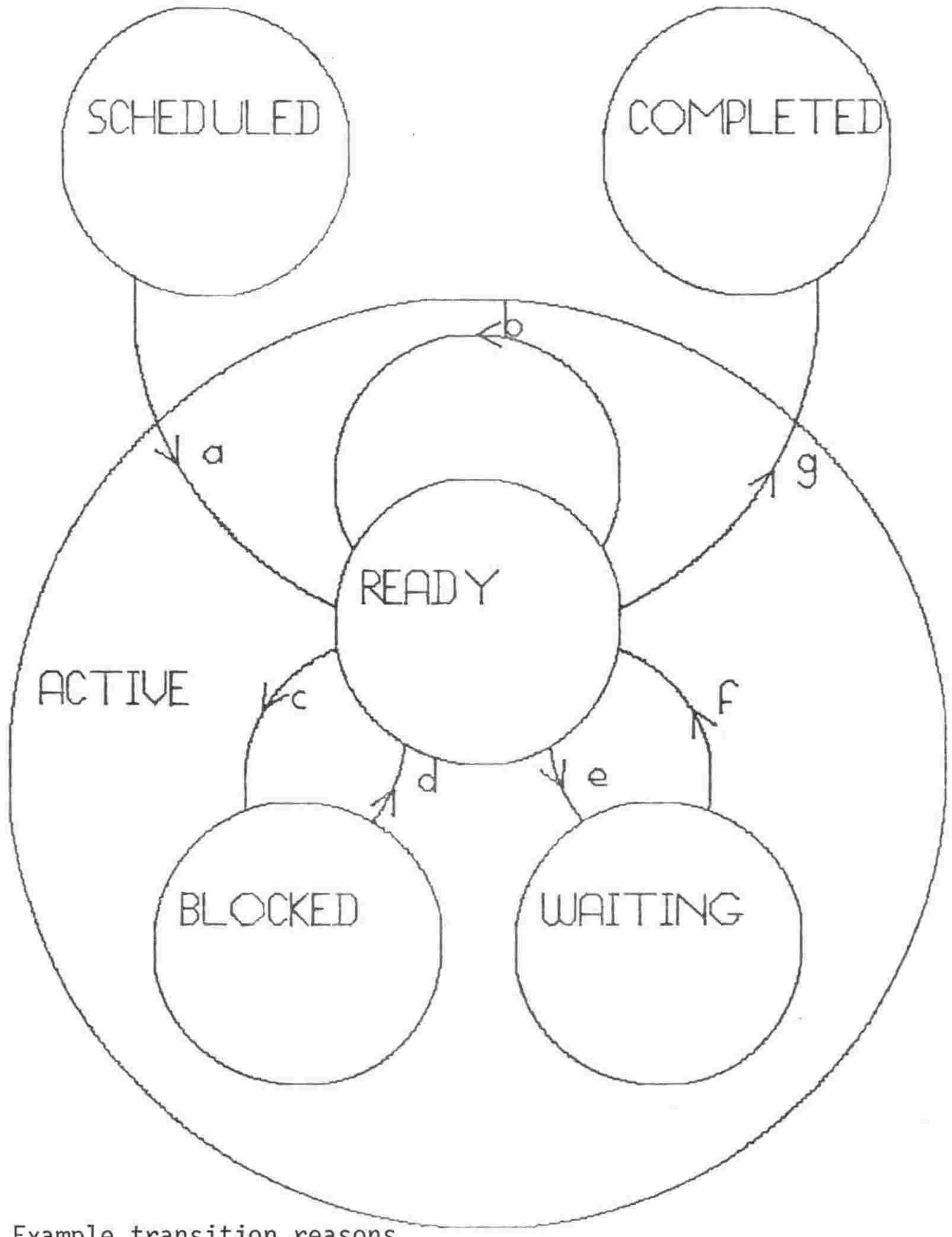
A scheduled process is a process which has got past the high-level scheduler, but cannot enter the mix because of unavailability of initial resources.

An active process is a process which is in the mix.

A ready process is an active process which is currently able to use a processor, including those processes which actually are using a processor.

A waiting process is a non-ready, active process which is waiting for some resource to become available.

A blocked process is a non-ready, active process which is waiting for some event other than resource availability.



Example transition reasons.

a) Initial resources found.

b) Preempted by a higher priority process.

c) Coroutine passes control to its partner.

d) Control passed back from partner.

e) Page fault occurs.

f) Page arrives in main memory.

g) All processing completed.

Figure 3.1 PROCESS STATES IN A MULTIPROGRAMMED SYSTEM

3.2 MONOPROGRAMMED SCHEDULING.

Research into the scheduling of jobs on monoprogrammed systems appears in general to be buried in more general work done in the field of Multiprogrammed Scheduling. In the situation where processor time is the only scarce resource considered, then the problem is the same as that of the single server queueing system, which is analysed extensively by Conway, Maxwell, and Miller [50], who also include a brief history of the work accomplished in this field.

When two or more resources are considered, then the problem is slightly more complex. Often a monoprogrammed system with several resources can be considered to have the single resource of processor time. However, there do occur situations where other resources may 'become available' in some way. Examples of this occur when a resource is a mountable storage device of some kind, such as a magnetic tape reel. In this situation the problem of resource scheduling becomes the problem of premounting mountable resources and, in effect, multiprogramming is introduced, since system resources may be allocated to more than one job at any one time. In a paper by Austin, Hanlon, and Russell [12] this problem is discussed, and the implementation of an algorithm for a monoprogrammed machine is described. However, as is very common with scheduling algorithms, the algorithm is a heuristic one, although the 'shortest job first' discipline forms a basis for it.

3.3 MULTIPROGRAMMED SCHEDULING.

For the purposes of this survey, only the uniprocessor case of Multiprogrammed Scheduling will be considered, although it should be noted that some of the research reported applies equally or partly to Multiprocessor Scheduling also. Research into Multiprogrammed Scheduling can be directed at establishing general laws and results which are expected to hold true universally, or at analysing the situation for a specific system, or at a combination of these. The most widely used research tool for establishing the general laws and results is Analytic Modeling, in which a mathematical model is developed and analysed. For studying specific systems, two main approaches are available. Experimental Measurement involves the design and execution of experiments on a real system, and the analysis of the data collected. Model Simulation differs from this in that it is a simulated system on which the experiments are performed, the advantages of this being that results are generally available more quickly, experiments may be duplicated and reproduced, and the experimental process has a lesser disruptive influence on the normal running of the system.

Coffman and Kleinrock [45] present a wide variety of priority scheduling algorithms and classify these according to various attributes, also providing some ideas on how users could attempt to 'outwit' each of the algorithms. Other papers of a survey type have been published by M^{C}Kinney [103], Conway, Maxwell, and Miller [50], Hellerman [81], Lorin [98], Sayers [135], Anderson and Sargent [7], Coffman and Denning [43], and Bunt [29].

An integral part of any scheduling algorithm is the choice of a criterion for performance evaluation on which to base the decisions that must be made. However it is often the case that in the design of a computing system the objective of the scheduling is not formulated explicitly, but eventually appears implicitly within the operating system. It also frequently occurs that in theoretical discussions, only simple performance criteria, such as processor utilisation, or job throughput, are used, whereas to keep in line with the growing emphasis on multiple resource allocation and resource scheduling, research is needed into more complex performance criteria. It turns out that this need is being fulfilled, with a wide range of performance criteria of varying complexity and versatility having been studied since the early 1970's.

3.3.1 Uniprocessor Multiprogrammed Scheduling.

The study of Multiprogrammed Scheduling can be subdivided into High-Level Scheduling and Low-Level Scheduling, with most research being directed at either one or the other of these. However, Clark and Rourke [39] have studied interactions between high-level and low-level strategies in an attempt to determine 'universally better' algorithms.

3.3.1.1 Low-Level Scheduling.

Low-Level Scheduling can be further subdivided into the two fields of low-level processor allocation, which is also known as task dispatching, and low-level resource allocation. The first of these is a special case of the second, in which the processor is the only resource considered, however there is sufficient research devoted to this discipline for it to be considered separately.

The study of task dispatching in multiprogrammed systems is essentially concerned with ways of creating many logical processors by multiplexing the physical processor(s), this usually being performed by software within the operating system rather than by hardware. There were several task dispatching algorithms in existence by the late 1960's, the most basic of these being the Round Robin, in which the active processes are considered in a fixed sequence, and the processor is allocated to the next process in the sequence which is also ready. A modification of this is the Ready Queue Round Robin, in which the ready processes form a separate queue and the processor is allocated on a First Come First Served basis. Other algorithms superimpose the notion of priority on the basic ready queue structure by linking a newly ready process into the ready queue at some position which reflects the calculated priority of the process. This calculated priority could depend on many factors, such as external priority, expected service time of the process, or type of process (whether operating system function or not for example).

New developments since this time have generally involved methods of improving performance according to some objective, by altering the method of determining these calculated priorities. Chua and Bernstein [38] introduce the concept of level of attained service for a process, and use this to determine priority. This serves to model several new disciplines, known as Late Arrival Round Robin, Early Arrival Round Robin, and Partial Round Robin, each of these having different performance characteristics. Kleinrock and Muntz [92]also use the concept of attained service in an algorithm which varies the discipline used according to the level of service attained by a process. Another algorithm involving variation of the discipline used is that proposed by Blevins and Ramamoorthy [22], in which dynamic feedback is used to determine the best discipline to use. Sherman, Baskett, and Browne [130] use microscopic level trace data to allow the definition of BEST and WORST disciplines using the performance criterion of processor utilisation. This allows absolute comparisons of previously defined disciplines and also allows the ability to test disciplines which attempt to approximate the BEST discipline.

In a slightly different approach to the problem, Bernstein and Sharpe [21] present an algorithm which is based on the assumption that process switching involves overheads of some kind, and hence the amount of process switching should be kept to a minimum. A similar approach is adopted by Potier, Gelenbe, and L'Enfant [117], who present an adaptive algorithm which attempts to reduce process switching overheads at times of overload by allocating extra CPU quanta to the running process on the basis of the number of arrivals during the current quantum. This is a generalisation of the algorithm proposed by Coffman [42] and further analysed by Heacox and Purdom [76].

The study of Low-Level Resource Allocation is concerned with the allocation of a wide variety of resources to the requesting processes. Typical resources dealt with are the processor, memory space, data transfer channels, and peripheral devices. Sometimes a further resource, namely data sets, such as program code files, may also be considered. Generally the resources considered are the processor, main memory, and I/O channels. There has been, however, a great deal of research directed at the study of Memory Management, wherein the two resources of central processor and main memory are the only ones considered.

The study of Memory Management originated with the advent of multiprogramming, with research being directed at fitting fixed sized programs into memory so as to waste as little as possible. With the advent of virtual memory and paging, the field of Memory Management has diversified to encompass the dynamic allocation of memory to active processes. Various methods for supplying the required code and data to active processes have been studied under the headings of paging strategies for general multiprogramming systems, and swapping strategies for time-sharing systems.

Paging strategies are generally composed of three sub-strategies, namely page fetch, which determines when pages are brought into main memory, page placement, which determines where the new page is to reside, and page replacement, which determines which pages, if any, are to be removed from main memory, and when this removal takes place. Early strategies used page demand as the fetch sub-strategy, First In First Out or Least Recently Used as the replacement sub-strategy, and the simple placement sub-strategy of replacing the old by the new. Since then, a variety of paging strategies and sub-strategies have been developed, including Denning's Working Set demand paging strategy [55, 56, 57], Belady and Kuehner's Biassed Page Replacement sub-strategy [19], a modification to the WS strategy proposed by Rodriguez-Rosell [122] which incorporates foreground-background ideas, and the Page Fault Frequency replacement sub-strategy proposed by Chu and Opderbeck [37].

Swapping strategies for time-sharing systems include that reported by Abell, Rosen, and Wagner [1] in which the decision to swap a process into the mix is based on priority considerations, but the choice of processes to be swapped out involves memory management considerations. Nielsen [111] investigates, through simulation, the desirability of including into swapping algorithms various features such as program relocation and memory 'krunching'. The effects of using bulk memory as a swap medium, faster transmission rates for disk storage, and hardware disk optimisers are also investigated. Finally, Anderson and Sargent [6] perform a statistical evaluation of swap scheduling algorithms of the FB_N type, in which there is one high-priority queue for service requests which have not received any service, and N-1 lower-priority queues for those service requests which have been started but not completed.

In the wider area of more general resource allocation, it seems that advances have been made only in recent years. One possible explanation for this is that the problem has only been recognised as being important for this short time, the reason being that the importance of resource scheduling in general has grown mainly for economic reasons, as ever-increasing amounts of time and money are being spent on computing and computing hardware. Dahm, Gerbstadt, and Pacelli [53] introduce a series of ideas on system organisation necessary for resource allocation to be feasible. This seems to mark the beginnings of the direction of endeavour towards the problem, although at this stage no attempt is made to give serious suggestions for the implementation of these ideas. Pass and Gwynn [116] present an adaptive resource allocation algorithm which uses predictor-corrector methods to optimise local and global measures of system performance. Hamlet [73] discusses the implications of the choice of resource allocation algorithm for accounting procedures, in terms of efficiency, reproducibility, and fairness to users. Finally, Lynch and Page [100] describe an implementation of an algorithm which controls resource allocation through task swapping. A generalisation of this algorithm, analysed by Kameda [88], involves a dynamic resource load balancing strategy, with light users of a congested resource being able to bid for a higher overall priority without increasing the total cost of service.

3.3.1.2 High-Level Scheduling.

As is the case with Low-Level Scheduling, High-Level Scheduling may be subdivided into the two fields of High-Level Processor Scheduling and High-Level Resource Scheduling, the former being a special case of the latter. A great deal of background research and useful results for High-Level Processor Scheduling have come from the fields of production scheduling, sequencing, and queueing theory. Excellent surveys of this background material may be found in Conway, Maxwell, and Miller [50] and Sevcik [129]. At that time, by the early 1970's, there was a wide variety of algorithms available for high-level scheduling of the processor. Some of them, such as First Come First Served and Random Selection, were used primarily for comparison purposes. Other algorithms which had been studied are Last Come First Served (LCFS), Shortest Processing Time (SPT), Round Robin, Feedback, Foreground-Background, and pre-emptive versions of

LCFS and SPT, this latter being named Shortest Remaining Processing Time. In the specialised field of time-sharing systems, Coffman and Muntz [46] discuss also the Shortest Elapsed Time Sharing, Basic Pure Time Sharing, and Shortest Expected Remaining Time disciplines.

There have been since some developments which have particular reference to High-Level Processor Scheduling, the most notable of these being Kleinrock's parametric model for a continuum of priority based algorithms [91]. This model encompasses all previously considered priority algorithms and defines three others, the Selfish Round Robin, Last Come First Served with pickup, and Last Come First Served with seizure. It turns out that artificial as it may seem, the Selfish Round Robin discipline is readily amenable to analysis. Sevcik [129] later introduces an algorithm based on service time distributions, known as the Smallest Rank algorithm, and proves this to be optimal under certain conditions. Under more general conditions, he also shows that the Shortest Remaining Processing Time discipline is optimal within a broad class disciplines which, it is argued, contains the globally optimal discipline.

More recently, Bunt [29] has introduced a new scheduling discipline known as the Single Queue (SQ) discipline. This is based on Kleinrock's parametric model in which processes in service gain priority linearly at rate β, and queued jobs gain priority at rate α. One important difference in the basic SQ algorithm is that a fixed number of the highest priority processes are considered to be processed simultaneously, that is, a fixed degree of multiprogramming is always in force. A further modification is introduced in which the parameter β is dynamically regulated by a feedback mechanism, resulting in improved service during short periods of overload.

High-Level Resource Scheduling is probably the most important area of Multiprogrammed Scheduling in the sense that more is to be gained, in terms of any criterion used, by the use of a 'better' high-level resource scheduling algorithm. However there are several problems associated with resource scheduling, under the general heading of determination of unknowns, that must be taken into consideration before such an algorithm may be constructed and implemented. A general job-shop computing system is characterised by a variety of jobs and processes exhibiting wide variations in resource requirements, and in particular a given process may vary widely in its overall resource requirements from one run to the next if for example a different set of data is supplied for input.

Two possible ways of dealing with this problem are the use of the concept of 'average job', and the use of user-stated maximum resource requirements for each resource being considered. The first of these methods is only suitable when the degree of multiprogramming is quite high (between twenty and thirty is proposed by Needham in [82], p213) so that the total requirement for each resource tends to remain stable. The second, more common, method of user-stated maximum resource requirements is a method which is used to some extent in many high-level scheduling algorithms. A third, more basic method of resource scheduling can be achieved without any fore-knowledge of job and process characteristics. This method involves the modification of the degree of multiprogramming on the basis of feedback information obtained from the system on resource loadings. If one or more of the resources being considered is overloaded, then the degree of multiprogramming is reduced, either by suspending or swapping out one of the active processes (a function of the low-level scheduler), or by allowing the degree of multiprogramming to drop of its own accord as processes finish. This method can be modified in many ways, by determining the best process to swap out (at the low level) or by determining the best process to activate next when it is desired to increase the degree of multiprogramming. An algorithm using the simple form of this method is proposed by Bard [16].

Codd [40, 41] presents a static, non-priority algorithm for resource scheduling when exact resource requirements are known. Abell, Rosen, and Wagner [1] describe a dynamic, priority driven resource scheduler in which low priority processes are rolled out to free resources for a higher priority process, this necessitating accurate prior knowledge of resource requirements. Thesen [136] describes a dynamic resource scheduling algorithm which solves a knapsack type linear programming problem formulated in terms of a heuristic utility function to determine the best mix of processes. Once again, resource requirements need to be known accurately in advance, although user-supplied estimates may be sufficient, and more realistic, requirements for the algorithm. Austin, Hanlon, and Russell [12] describe one particular aspect of a resource scheduling algorithm, which is implemented in a monoprogramming environment but could be extended to a multiprogramming environment, in which a job is not started until all reusable resources have been allocated. Larmouth [94] describes another, priority driven algorithm in which all resources must be allocated before a job will start. This is achieved by reserving resources for the highest priority job until all of its resource requirements can be met.

Combinations of these methods may be used. One implementation is detailed by Northouse and Fu [114] in which a job is classified into one of several different classes of resource requirements on the basis of user supplied information. A selection algorithm then determines the mix by considering the dynamically varying average resource requirements of the different classes. An evaluation of the choices made by the selection algorithm, on the basis of some performance criterion, is used to provide feedback to the algorithm.

3.3.2 Adaptive Scheduling.

The introduction of dynamic feedback algorithms into the field of High-Level Resource Scheduling is possibly the most important advance which has been made in recent years. As computing systems have become more advanced, their operating systems have become more and more complex, and the interactions between system parameters (those aspects of a system which may be measured or set, including hardware) have become more and more subtle, particularly in multiprocessing-multiprogramming systems. This has meant that the development of accurate system models has become increasingly difficult, and that the production of a system model has become less useful since the use of a complex model consumes large quantities of the computing resources that it is designed to help conserve. However the use of feedback in an adaptive algorithm allows a simple model to be used, since the subtleties of the real system can be accounted for by the feedback mechanism, provided that all independent parameters are taken into account. Another useful feature of adaptive algorithms is that drifts in the characteristics of the input stream can be accounted for on a much smaller time scale than they could be otherwise. Further, the use of Mathematical Programming techniques from Operations Research promises to provide a way of formalising the dynamic feedback mechanisms, thereby improving the theoretical foundations on which the research stands. However, as Chandy and Yeh [32] have mentioned recently, practitioners seem slow to familiarise themselves with some of the wide variety of Mathematical Programming techniques available, resulting in a dearth of research publications utilising this potentially fruitful blend of disciplines.

The development of adaptive techniques for High-Level Resource Scheduling appears to have had its origins in the early 1970's, with the algorithm proposed by Northouse and Fu [114], in which feedback is used both to provide dynamic estimates of the workload, and to modify job selection procedures on the basis of the effectiveness of previous decisions. The algorithm proposed by Bard [16] is a further adaptive high-level resource scheduling algorithm, although only the two resources of CPU and main memory are considered. In this algorithm, feedback is used to determine the effect on the system of the currently specified degree of multiprogramming, and to modify this if the system is overloaded or underutilised.

The introduction of dynamic feedback mechanisms has not been restricted to the field of High-Level Resource Scheduling, examples of its use having been reported in the other fields of High-Level Processor Scheduling, Low-Level Processor Scheduling, Memory Management, and more general Low-Level Resource Allocation. Blevins and Ramamoorthy [22] propose an algorithm in which the actual processor scheduling discipline used is based on feedback information in terms of the distribution of service times of service requests, and the effects of earlier decisions on system performance. Badel et al [13] propose an adaptive algorithm for controlling the degree of multiprogramming in a virtual memory system. This is essentially a low-level processor scheduling algorithm, even though one of the goals is the prevention of thrashing, a memory management problem. Potier, Gelenbe, and L'Enfant [117] also propose an adaptive low-level processor scheduling algorithm in which adaptive techniques are used to reduce task-switching overheads when traffic intensity is high. More recently, Gelenbe and Kurinckx [70] have proposed a dynamic feedback algorithm for controlling the degree of multiprogramming in a virtual memory system. This is known as Random Injection Control, and operates by artificially limiting the set of ready processes through the creation of a further state, known as the impeded state, into which a process moves after having acquired a certain amount of CPU time. The time spent in the impeded state is then determined by a random variable, whose distribution is a function of the throughput of the system.

Denning [57] introduces the use of feedback into Memory Management with the Working Set strategy, which uses implicit feedback to control the effective degree of multiprogramming. This in a sense acts as a buffer between the high-level scheduling algorithm and the resource allocation algorithm, since it restricts the processes which are to be considered by the low-level scheduler. Pass and Gwynn [116] propose a low-level resource allocation algorithm in which feedback information in the form of deviation from expected global system performance is used to modify the parameters of a local performance measure on which is based the scheduling decisions in the form of resource request fulfillment. Another low-level resource allocation algorithm, proposed by Kameda [88], uses implicit feedback in the form of an 'invisible hand' to rectify imbalances in resource utilisations. This involves a bidding mechanism whereby all users bid for priority of the resources they use, on a dynamic basis, when the processes are active. Since a user is constrained by the total cost to him for the whole job, he cannot bid very highly for a resource of which he is a heavy user, whereas light users of a congested resource can afford to bid highly for that resource, thereby dissolving the imbalance.

In the field of High-Level Processor Scheduling, Bunt [29] describes the use of dynamic feedback in an algorithm which uses feedback information, in terms of the arrival rate of jobs and the load on the system, to alter the scheduling strategy dynamically to cope with peak periods of overload. Finally, Larmouth [94, 95] describes the implementation of a high-level resource scheduling algorithm in which information on long-term resource usages is used as feedback information for the more general function of long-term resource management, which rations the system resources over relatively long time periods of the order of days and weeks, rather than milliseconds and seconds.

3.3.3 Performance Criteria.

In a general job-shop, batch and/or remote, computing system, it is frequently the case that all the user is interested in is getting his best 'value for money', that is, the fastest turnaround for the lowest cost, whereas the installation management must consider such items as income, machine utilisation, 'user satisfaction', and job throughput, as well as turnaround. Further, the cost to the user is not necessarily a financial cost; it may be in units of number of jobs submitted, cards keypunched, or lines of code written for example, the actual criterion used depending entirely on the whims of the user.

If all these factors are to be taken into account in the design of a scheduling algorithm, then the use of a single measure to be optimised is often not an effective way of managing a computer installation. What is required is some criterion which makes a trade-off between the divergent goals of making efficient use of the available computing resources, and providing acceptable or better service to all users of those resources. One such algorithm, presented by Aggarwal and M^{C}Carl [4], optimises a composite of the four different items of in-process inventory, facilities utilisation, lateness, and mean setup time, representing respectively the criteria of waiting time, utilisation, turnaround, and overheads. A slightly different approach is adopted by Bunt [29] who describes a scheduling algorithm in which the performance criterion, that of maintaining 'an acceptable level of service', varies with the workload. The particular criterion used, that of throughput measured as a percentage of work submitted, enhances the dynamic, self-regulating nature of the algorithm used. Lynch and Page [100] describe a scheduler in which independent components evaluate decisions and make recommendations on the basis of different performance criteria, such as response time, turnaround time, and resource utilisation. A third component then combines these recommendations to make the best use of the resources under the existing conditions. Thesen [136] presents a heuristic performance criterion which takes account of resource utilisation, job priorities, and job deadlines, to maximise machine utilisation as well as to avoid excessive tardiness.

Other criteria which appear in the literature include Bernstein and Sharpe's 'deviation from promised service' [21], in which the difference between actual and promised rates of completion of processes is minimised, Clarke and Rourke's 'elapsed time multiplication factor' [39] which compares the rate of processing of processes with the rate which would occur if the process was alone in the mix, and which is similar to Kleinrock's 'wasted time' [91], and a group of six criteria for use in time-sharing systems, as described by Stimler [134]. Various other criteria appearing in the literature are throughput, resource utilisation, mean waiting time, response time, and, in the field of comparison of computing systems, quality of performance, such as hardware and software reliability.

As far as the implementation of a performance criterion is concerned, it is suggested by Hellerman [81] that the procedure for the design of a scheduling mechanism should consist of the following four steps:

1. Define an objective function in terms of the criterion chosen, assuming that all necessary information is known in advance.
2. Devise a 'best' scheduling strategy to optimise this objective function.
3. Devise an algorithm for extracting, estimating, or ranking the variables required in 1 from the observed variables.
4. Devise a mechanism which imbeds algorithm 3 into strategy 2.

Further, since the scheduling algorithm of any operating system is generally imbedded within the system, and hence difficult to modify after release of the operating system, it is also desirable that the actual implementation be flexible enough to allow any particular installation to determine the exact nature of the performance objective in terms of the parameters which may be set by the installation manager or modified on a routine basis, for example when the shift changes from day to night, or when the operating mode changes from batch to interactive. This would also cater for long-term changes in the work load encountered. An example of the provision of such installation modifiable parameters occurs in the scheduler described by Lynch and Page [100], for the IBM OS/VS2 Release 2 operating system.

3.4 CONCLUSIONS.

This has been a brief survey of computer scheduling with particular emphasis on the scheduling of a uniprocessor multiprogrammed computing system. One conclusion to be drawn from this survey is that the study of adaptive algorithms for scheduling multiprogrammed systems is gathering momentum as a useful research field. Further, it is to be noted that there are distinct advantages to be gained from the use of Mathematical Programming techniques within dynamic feedback algorithms, although it would appear that the challenge is yet to be actioned by the majority of practitioners in this field. The next section presents just such a scheduling algorithm, in which the Mathematical Programming technique used is Dynamic Programming, in particular the Variable Metric Dynamic Programming algorithm as developed in section 2.

SECTION 4.

APPLICATION OF VARIABLE METRIC DYNAMIC PROGRAMMING TO HIGH-LEVEL SCHEDULING.

4.1 INTRODUCTION.

The use of Mathematical Programming techniques in Computer Science research is suitably demonstrated with the application of Variable Metric Dynamic Programming to the solution of a job scheduling problem, and the implementation of this in the operating system of a batch and interactive computing system running in a university environment. Essentially the application consists of extending the existing job scheduling mechanism by modifying on a dynamic basis some of the job scheduling parameters that were previously set by the operators. These parameters affect the overall degree of multiprogramming and the relative service provided to different classes of batch jobs.

The Dynamic Programming approach allows for two distinct levels of feedback, one being characterised by short-term or internal variations, such as variations in the current workload, and the other being characterised by long-term or external variations, such as the change from day-shift to night-shift. The short-term variations are taken into account by applying a single solution of the problem to different starting points, or states of the system. This is possible because the Dynamic Programming solution may be presented in the form of a policy, which provides an optimal decision for any one of a large number, or even a continuum, of starting states. In contrast, the long-term variations are taken into account by the re-specification of some of the fixed inputs to the solution process, such as the exact nature of the cost functions which represent the optimality criterion, and then the re-solving of the problem to provide a new policy.

4.2 THE PROBLEM.

The problem to which the Dynamic Programming solution method is to be applied is a fairly simple problem in the context of the high-level scheduling of a multiprogrammed computing system. The reason for this is that the application is intended to demonstrate the feasibility of the Dynamic Programming approach and its wide range of applicability rather than to provide a solution to a difficult scheduling problem. Thus the problem has been chosen bearing in mind the desirability of a simple approach to the solution.

In a batch and interactive computing system, there is usually a clear-cut distinction between the batch work and the interactive work. The batch work consists of jobs which arrive mainly from local or remote card readers, but may also be submitted as non-interactive jobs from interactive terminals. All such jobs are queued in some way for consideration by the high-level scheduler. The interactive work, however, consists of a stream of processes which interact with or are invoked from the remote terminals. Generally these interactive processes are implicitly given higher priority than processes which form part of batch jobs because they bypass some, if not all, of the controls imposed by the high-level scheduling mechanism.

Considering now the non-interactive work, these jobs are generally divided into several resource classes on the basis of some externally declarable variables, such as resource requirements or requested priority, this information being provided by the user for the high-level scheduler. The primary function of the high-level scheduler is to determine when batch jobs are to be activated. These decisions are made on the basis of knowledge of the characteristics of the jobs and of instant resource availabilities. A secondary function of the high-level scheduler is to determine which job is to be activated, given that a decision has been made to activate a job. However it is often the case, in more complex high-level scheduling mechanisms, that the 'which' decisions can influence the 'when' decision, and thus this should be viewed more as a joint function than a secondary function. These 'which' decisions are often made on the basis of external considerations, such as requested turnaround times, or current operating environment.

This brings us to the statement of the problem to be approached. Given a high-level scheduling mechanism which categorises each incoming batch job into one of several resource classes, we want to devise some dynamic method of determining how many batch jobs should be active at any one time, and how this total is to be divided among the different resource classes. The specification of the total number of batch jobs which should be active, which is determined on the basis of attempting to improve overall system performance, performs the function of specifying when a new job should be activated. Similarly, the specification of how this total is divided among the different resource classes, which is to be determined on the basis of the relative service to be provided to the different classes of batch customer, specifies which class of job should be activated next.

4.2.1 The Approach Adopted.

As has already been mentioned, the problem to be solved has been chosen with a reasonably simple solution approach in mind. This approach basically involves extending the existing high-level scheduling mechanism, which already allows the specification, by parameters, of the total number of batch jobs which should be active as well as how this total should be divided among the different classes of jobs. The extensions then take the form of modifying these parameters dynamically, in order to achieve some prespecified goal.

One important aspect of the approach is that not all of the total work processed can be controlled using these parameters. In particular, interactive work cannot be controlled in this way since in general this work bypasses the queueing mechanisms of the high-level scheduler. Further, there usually exists some class of 'special' jobs, such as the few jobs which do not fit into the normal resource classes because of special resource requirements, which it is desirable to schedule 'by hand', and hence may not be controlled by the proposed extensions. This leaves us with the problem of controlling the non-interactive, non-special portion of the total work load. Hereafter, this work will be known as the normal batch work, and all other work will be known as the uncontrolled work.*

The extent of this uncontrolled work may vary significantly, either because of variations in the number of interactive users, or

*During this implementation, the portion of the total work load being controlled varied between approximately 20% and 60%.

because of the existence of one of the special batch jobs. In either case, these dynamic variations are of a long-term nature, on a time scale of the same order of magnitude as job completions, although some short-term variation may be introduced by the dynamic nature of requests for service from each interactive user. Thus it is not inappropriate to perform a high-level scheduling function on the basis of using only those resources which are not being used by the uncontrolled work. Even though some minor resources contention will occur because of the short-term variations, this is acceptable because there is always some resources contention in a multiprogrammed system resulting from the short-term dynamic nature of resource requests themselves, this being a problem for the low-level scheduler to resolve.

4.2.2 The Existing System.

The computing system available at Victoria University of Wellington is a Burroughs B6700 computer with 196,608 words of main memory. This is used for batch and interactive work, the majority of all work being student work, either for teaching or research purposes. The operating system being used currently is the Burroughs B6700 Master Control Program (MCP) Version II.9. This provides for a number of installation defined batch job queues, and a series of parameters to be used for the high-level scheduling of this batch work. The incoming interactive work, on the other hand, is controlled directly by one or a number of supervisory programs, known as Message Control Systems (MCS's). A low-level scheduling mechanism is also provided, along with some parameters which may be used to modify its behaviour.

On the current system there are five job queues, each for a different class of batch jobs. Three of these queues are reserved for normal batch jobs, with the classifications being based on the maximum requirements for the three resources of CPU time, I/O time, and lines printed. These are labeled as Queue 3, Queue 5, and Queue 7. The other two queues are used for 'special' jobs, Queue 0 being reserved for high-priority operator entered jobs, such as jobs to assist with error recovery after a failure, and Queue 9 being used for customer jobs which do not fit into any of the three normal classes, because of excessive or exceptional resource requirements. Each queue has associated with it a maximum declared priority. A job's declared priority performs the dual function of specifying where in the queue the incoming job is to be inserted, as well as being used by the low-level scheduler for allocating resources.

The high-level scheduling mechanism is a parameter driven algorithm which selects jobs from the queues and passes them to the low-level scheduling mechanism for further consideration. This function is performed by the MCP procedure SELECTION, which removes a job from one of the job queues, and changes its state from queued to a state known as scheduled, or to the ready state. At this stage, a job is considered to be a process in its own right, this process containing code to fire up the processes which make up the job proper, and code to perform certain housekeeping functions which do not require a process to be fired up, such as the removal of files.

The parameters used by SELECTION are known as mixlimits, consisting of one QUEUE MIXLIMIT for each job queue and an overall batch MIXLIMIT. These are limits on the number of processes currently active, decisions being made by comparing these limits with the individual queue mixcounts, and the overall batch mixcount, which is merely the sum of the individual queue mixcounts.

The decisions made by SELECTION consider first the overall mixcount and then the queue mixcounts. Considering the queues in some well-defined order, the overall mixcount is compared with the overall mixlimit. If the overall mixlimit is higher, the the mixcount for the queue being considered is compared with the corresponding queue mixlimit. If this mixlimit is higher, then jobs will be started from the head of this queue until either the queue mixlimit is equalled, or the overall mixlimit is equalled, or the queue is empty. The job at the head of the queue is determined by the declared priority, with First Come First Served being used to eliminate ties. This process of looking at each queue in turn is repeated every time that a job arrives into any queue, or a job is completed, or any one of the mixlimit parameters is changed.

Setting of the high-level scheduling parameters is done entirely by the computer operators, without any fixed time schedule, and for a variety of reasons. In general the changes are made when the need becomes apparent, that is, when it is noticed that an undesirable situation exists. The reason for making a change usually involves the implicit goal of remedying the undesirable situation. Another commonly used method of effecting a change is for an operator to override the parameter settings by entering a command to activate a specific queued job, which might be done if, for instance, there was a temporary lull in the amount of interactive work.

Since operating conditions may vary dramatically during the course of a single day's processing, it is difficult to specify a typical set of high-level scheduling parameters. However, assuming a light interactive load, and a moderate influx of batch jobs, the parameters might be set as follows. The individual queue mixlimits would be set to 4 for Queue 0, 3 for Queue 3, 2 for Queue 5, 1 for Queue 7, and 0 for Queue 9. The Queue 0 limit is set to a relatively large number because of the high priority of operator entered jobs. However it is very rare that this limit is in force, as it is very infrequent that there are any Queue 0 jobs running at all. The Queue 9 limit is set to zero because these jobs are the special jobs which are activated manually at all times. The other queue mixlimits reflect to some extent that Queue 3 is a high-priority queue for short jobs, Queue 7 is a low-priority queue for long jobs, and Queue 5 is somewhere in between. Finally, the overall batch mixlimit would be set to 8, this being higher than the sum of the individual queue mixlimits for Queues 3, 5, and 7 so that as soon as a job is entered into Queue 0, it would begin executing.

4.2.3 Proposed Extensions.

The aim of the proposed extensions is to control the scheduling of work from Queues 3, 5, and 7 on a dynamic basis, by modifying the individual mixlimit parameters for these queues. This is to be done in such a way that some prespecified goal is always aimed for. This goal is to reflect in some way a desire to deliver different levels of service to the different classes of batch jobs, as well as a general desire to maintain, and improve if possible, the overall system performance.

The level of overall system performance can be controlled by specifying the individual queue mixlimits only insofar as these parameters can be used to control the degree of multiprogramming within the system. Given then that this attempt to control the degree of multiprogramming results in a decision as to how many jobs should be active from Queues 3, 5, and 7, the goal of providing different levels of service to the different classes of batch jobs then involves specifying how this total is to be divided among the three queues, thereby resulting in values for the individual mixlimit parameters.

4.3 THE APPLICATION.

The application of discrete Dynamic Programming to the problem as defined involves firstly the formulation of the problem in Dynamic Programming terms, then the specification of all the variables and functions required for this formulation, and finally the determination of how this is to be incorporated into the existing operating system. For the Dynamic Programming formulation, the problem must be specified as an N-stage decision problem, where the decisions are based on an attempt to optimise a specified cost function, and rely on knowledge of how they will affect the state of the system, in the form of a transformation function. The specification then involves deciding exactly which variables are to be used to describe the state of the system and the decisions taken, and the exact nature of the cost function and the transformation function. Finally the details of incorporating the solution process into the existing operating system are concerned mainly with how the state of the system is determined, and how the decisions are applied.

4.3.1 The Dynamic Programming Formulation.

The unconstrained N-stage decision problem of Dynamic Programming, which is the class of problem to which the Variable Metric Dynamic Programming solution method is addressed, involves a set of N decisions, corresponding to N time intervals. These decisions are made on the basis of controlling the state of the system at the start of each time interval to minimise a cost function, which is a function of the states and the decisions. The control imposed by a decision is defined by a transformation function, which specifies how a decision modifies the state of the system during the time interval in question. In mathematical terms, this is expressed as

$$\begin{aligned} &\underset{\{\underline{u}_0,\underline{u}_1,\ldots,\underline{u}_{N-1}\}}{\text{minimise}} \quad \{ \sum_{k=0}^{N-1} L_k(\underline{x}_k, \underline{u}_k) + F(\underline{x}_N) \} \\ &\text{subject to } \underline{x}_{k+1} = \underline{f}_k(\underline{x}_k, \underline{u}_k) \ , \quad k = 0, 1, \ldots, N-1 \end{aligned} \qquad (4.1)$$

The vector $\underline{x}_k$ represents the state of the system at time k, the vector $\underline{u}_k$ represents the decision taken at time k, the scalar functions L_k and F together form the cost function, and the vector functions $\underline{f}_k$ are the transformation functions for each time interval.

The Dynamic Programming formulation now involves the specification of exactly how the state vector $\underline{x}_k$ represents the state of the system, exactly what the components of the control vector represent, and what the transformation functions $\underline{f}_k$ are. The other quantities as yet undefined are the time horizon, N, and the functions making up the cost function. These need not be specified here since they are variables which may be modified to take into account long-term variations in the operating environment, and thus are more appropriately specified in the discussion of the implementation. Further, the cost function is the mechanism by which management decisions are incorporated into the extended scheduling mechanism and thus should be considered to be a parameter rather than an integral part of the problem formulation.

The specification of the components of the state vector $\underline{x}_k$ is essentially a problem of selecting from the large number of items of available information those which are relevant to the problem in hand. The mixcounts for the normal batch queues must be part of the state of the system since these are what we are trying to control. Other information relevant to the control of the level of service provided to the normal batch customers takes the form of the numbers queued of each class of job. Finally there must be a variable which measures in some way the degree of multiprogramming, since this is the other variable we are trying to control. Because the problem under consideration involves controlling only the numbers of batch jobs active, the absolute degree of multiprogramming is of no special interest. What is required is some variable which reflects in some way the difference between the actual and desired degrees of multiprogramming, so that a decision may be made to increase or decrease the total number of normal batch jobs active. The actual variable used is a count of scheduled and suspended processes, with an added consideration of available memory to account for when there are no processes either scheduled or suspended. A suspended process is a process which has been temporarily removed from the ready state by the low-level scheduler, which has considered the degree of multiprogramming to be too high. For ease of description, this variable will henceforth be known as the scheduled count.

The obvious choice of control or decision variables is the desired values of the individual queue mixcounts for the normal batch queues. This choice allows a decision to involve simply setting the individual queue mixlimits to the computed desired values, and letting the existing high-level scheduling mechanism effect the required changes. Further the control over the degree of multiprogramming is contained implicitly within this information, in that the total number of normal batch jobs which should be active is simply the sum of the specified individual mixlimits. In fact what has been chosen for the control vector is the set of differences between the actual and desired queue mixcounts. This allows the same simple implementation of a decision but has other subtle benefits which will become evident later.

The transformation functions reflect how a certain decision will influence the state of the system, by predicting the value of the state vector at the next stage as a function of the current state and the decision applied. For this particular implementation, the transformation does not depend on which time interval is being considered, thus we have to specify only one function which, for each time interval, determines just how the state variables, namely the queue lengths, the queue mixcounts, and the scheduled count, are affected by a decision to modify the normal batch mixcounts by given amounts. Further, some consideration must be given to the tradeoff between the simplicity and the accuracy of this function.

The values of the new queue lengths resulting from a decision are taken as the old values from which have been subtracted the corresponding values of the decisions. Thus if it is decided that it is desirable to increase the mixcount for a particular queue by one, then the transformation will predict that the length of the queue will decrease by one, since a job must be removed from the queue to increase the mixcount for that queue. The values of the new queue mixcounts resulting from a decision are taken as the old queue mixcounts to which have been added the decision variables. Finally the scheduled count is modified by adding to it the sum of the decision variables, since this sum represents the desired overall change in the number of active processes. The use of the decision variables directly, instead of having to compute differences, is one of the advantages of using the differences as the control vector over using the actual desired queue mixcounts. This argument applies to the transformation of the queue lengths also.

Now expressing these ideas mathematically, the transformation function has the following form

$$\underline{x}_{k+1} = \underline{x}_k + A.\underline{u}_k \, , \quad k = 0, 1, \ldots, N-1 \qquad (4.2)$$

where $\underline{x}_{k+1}$ is the predicted state at the next stage,

$\underline{x}_k$ is the current state of the system,

$\underline{u}_k$ is the decision made,

and A is a 7x3 matrix with the following values

$$A = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

The ability to use this simple general form of the transformation function is a further result of the decision to use differences for the control vector in preference to the new desired mixcounts. An important point to note about this transformation function is that it is a linear equation, which has implications for the robustness of the solution policy provided by the Variable Metric Dynamic Programming algorithm. These implications will be discussed later.

Considering now the tradeoff between simplicity and accuracy, it could be argued that accuracy in the specification of the transformation function is not as important as the establishment of trends, since there is little hope of producing a specification which is accurate enough not to need any other information in the application of the solution. That is, it is very likely, no matter how accurately the transformation function is specified, that there would be some effects that would not be accounted for. Moreover, the establishment of trends is sufficient to provide a basis for feedback mechanisms to take control of the situation. For example in the transformation of the queue lengths, the prediction that a decrease in a queue mixcount will increase the corresponding queue length specifies that if the queue mixcount is reduced, then the queue length will increase by the same amount, which is always true only if steady state conditions apply. In the real situation, it might really mean that the queue length will not decrease as fast as it was decreasing, or that it will increase faster than it was increasing, but the trend is still the same. These trends provided by the transformation function are reflected in the policy produced by the solution process. The policy then provides decisions which will modify the state of the system in the desired direction.

Now although these decisions are not optimal, because the transformation function is not accurate, continued application of the solution policy with feedback will result in near optimal states, provided that uncontrollable influences do not produce large short-term variations, and provided that the lack of accuracy does not result in oscillations about the optimal state. The former of these potential problems has been mentioned already in connection with the time scale of large scale resource demands made by the uncontrolled work, and the latter is addressed later in the discussion of the cost function used. Given that these potential problems are not serious, this leads us to the conclusion that simple transformation functions do have a use, and may even be preferable if the simplicity provides other benefits.

To complete the problem formulation, it now remains to discuss just how the policy is to be applied to the physical system in order to achieve the goal implicit in the definitions of the cost function. This involves determining just how the input information, in the form of a measurement of the state of the system, results in a decision, in the form of the specification of the new queue mixlimits for the normal batch queues.

The Variable Metric Dynamic Programming solution algorithm produces the solution policy in the form of a vector and a matrix for each stage k. The vector, known as $\underline{\bar{u}}_k$, is the decision proposed for some nominal state, and the matrix, known as β_k, indicates just how this decision should be modified to account for variations in the state of the system. Mathematically, if $\underline{\bar{x}}_k$ represents the nominal state at stage k, and $\underline{x}_k$ is the current state, then the decision $\underline{u}_k$ corresponding to this state is found from the expression

$$\underline{u}_k = \underline{\bar{u}}_k + \beta_k^T . (\underline{x}_k - \underline{\bar{x}}_k) \qquad (4.3)$$

Bearing in mind that this decision vector represents the desired changes in the normal batch queue mixcounts, the application of a decision now involves adding this vector to the vector of current mixcounts, and outputting these as the new queue mixlimits.

An important point to note here is the assumption of the robustness of the policy, this assumption being that expression 4.3 holds true for all values of $\underline{x}_k$. As it happens, this assumption is valid when the problem being solved by the Variable Metric Dynamic Programming solution method is an LQP problem, this being characterised by a linear transformation function and a quadratic cost function. This possibility of guaranteed robustness is another reason for choosing a simple transformation function in preference to an accurate but complex one. However this does not mean that the problem must be chosen to be an LQP problem, since it is possible to include in the implementation some mechanism for re-solving the problem for new values of the nominal state whenever the actual state is not close to the nominal state for expression 4.3 to be valid.

4.3.2 The Use of Feedback.

As has already been mentioned, the use of Dynamic Programming provides two distinct levels of feedback. The first of these, to take care of short-term variations, involves the application of the policy at each time interval, in particular, in the observation of the state of the system, and in the way the solution policy is actually used. The other level, to deal with long-term variations, involves modifying the Dynamic Programming problem itself to provide a new policy.

Given the Dynamic Programming formulation as discussed, there are two different ways in which short-term feedback information is used. Firstly the calculation of the current state of the system may involve feedback items. For example, the calculation of the number of scheduled processes takes into account the amount of available memory, and converts this into a 'deficiency' of scheduled processes. This conversion must use some value for the amount of memory that an 'average' process would require, and it is this value which could be determined dynamically by feeding back, at some predefined interval, the mean memory requirements of the active processes. Further, it is easy to visualise slightly different problem formulations which could use more feedback information in the determination of the state of the system. For example, if the state vector was concerned with units of work, rather than numbers of jobs, for the lengths of the job queues, then feedback information regarding the predicted size of the jobs in each queue would be useful.

The second form of short-term feedback is inherent in the way the solution policy is used in this implementation. The Variable Metric Dynamic Programming solution method provides a solution policy for each stage of the problem, with each policy normally being used once to calculate a sequence of decisions, given a starting state and the transformation function for each stage. This however assumes that the transformation is exact, and further, that the problem has been solved using sufficient time stages to cover the whole period over which control is to be applied. Since neither of these assumptions is practicable in this implementation, we are forced to use some other method of calculating the decisions at each stage. The method chosen is to solve the problem for a given number of stages, N, and to use the policy produced for the first stage for making all decisions. Thus each decision is considered to be the first decision of an N-stage decision sequence. This method overcomes the earlier discussed problem of inaccuracy in the transformation function because the state of the system is reobserved every time a decision is made, instead of being calculated from the previous state using the transformation function. This constitutes the other form of short-term feedback. Instead of accepting the prediction of the new state as made by the model, which is represented by the transformation function, the physical system is used to determine the transformation for making the next decision. This feedback is provided for each variable represented in the state vector, and hence we have a situation involving multiple feedback, which is in contrast to the majority of feedback scheduling mechanisms in which only one variable is modified by feedback.

An example of the other level of feedback is that discussed at the end of the previous section, whereby information is gathered to determine the validity of the solution policy in the case of a non-LQP problem formulation. This could be achieved by maintaining as a feedback item some vector which represents a current average state of the system. This vector could then be compared with the nominal state to decide whether or not the problem needs to be re-solved for a new nominal state. When a decision is made to re-solve the problem, this new vector would be used as the best value for the new nominal state.

4.3.3 Implementation.

A discussion of the implementation of the proposed problem formulation now requires that the remaining variables of the Dynamic Programming problem be specified, along with some details of how the whole process is imbedded into the existing operating system. The most important variable yet to be specified is the exact nature of the cost function, but also the expression for calculating the number of scheduled processes, the length of the time interval, and the number of stages still have to be specified.

As has already been indicated, the aim of the cost function for this implementation is to provide well defined different levels of service to the normal batch customers by controlling the numbers of active batch jobs from the different classes, and to improve overall system performance by controlling the degree of multiprogramming. Now the information available concerning the degree of multiprogramming is the state variable which represents the number of scheduled processes. Thus to control the degree of multiprogramming we can specify a desired value for the number of scheduled processes, and attempt to keep the actual number as close as possible to this desired value. Considering now the provision of different levels of service, this may be achieved by attaching different degrees of importance to the need to process jobs in the different queues, specifically by weighting the queue lengths. Further, some cognisance may be taken of the actual queue mixcounts by aiming to have a balanced mixture of jobs active from the batch queues at all times, but too much importance cannot be attached to this aim because the total number of batch jobs must be determined by the scheduled count.

To make it easier to use the cost function, it is convenient to generalise its form with respect to each component of the state vector. This is done by considering each component to have a target value, and by using the weighted squared difference between the observed value and the target to be the contribution of that component to the cost function. The determination of the target vector, and the weights, is how the cost function can be influenced by management policy decisions. Mathematically then, the contribution of the whole state vector, $\underline{x}$, to the cost function at any stage is given by the expression

$$\sum_{i=1}^{7} w_i(x_i - t_i)^2 \tag{4.4}$$

where $\underline{w}$ is the vector of weights,

and $\underline{t}$ is the vector of target values.

The control or decision vector may also make some contribution to the cost function at all stages except the last (since there is no decision taken after the last stage has been reached). For this implementation, the expression chosen is the sum of squares of the components of the control vector. This reflects a desire to make decisions which are small in magnitude, remembering that the decisions are the desired changes in mixcounts, so that the potential problem of over-reacting to an undesirable state, thereby resulting in oscillations, is reduced. This contribution to the cost function is expressed mathematically as the expression

$$\sum_{i=1}^{3} u_i^2 \tag{4.5}$$

The fact that the cost function we are using is separable into the state and control contributions is not a requirement of the formulation, nor is it a significant simplification as far as the solution process is concerned. However what is significant is that the cost function as specified is quadratic, which means that the problem to be solved is an LQP problem, since the transformation function has already been specified as a linear function. Finally, it should be noted that the cost function is identical for each stage k, resulting in a final form as follows

$$L_k(\underline{x}_k, \underline{u}_k) = \sum_{i=1}^{7} w_i(x_i - t_i)^2 + \sum_{i=1}^{3} u_i^2 \quad , \quad k = 0, 1, \ldots, N-1. \qquad (4.6)$$

$$F(\underline{x}) = \sum_{i=1}^{7} w_i(x_i - t_i)^2$$

For the purposes of an initial implementation, and in the absence of any prespecified management policy, the weights were chosen as 1, 2, and 3 respectively for the queue lengths of Queues 7, 5, and 3, 1 for each of the queue mixcounts, and 100 for the number of scheduled processes. Similarly, the target values were chosen as zero for each of the queue lengths, reflecting a desire to complete all the queued work, 1, 2, and 3 respectively for the queue mixcounts for Queues 7, 5, and 3, reflecting a desire to have if possible a 'good' mix of jobs, and 4 for the number of scheduled processes. The large weight selected for the number of scheduled processes reflects that the attempt to control the degree of multiprogramming results in an equality constraint, in contrast to the minimisation of a weighted sum of squares which results from the attempt to provide different levels of service to the normal batch queues. Similarly, the small weights chosen for the queue mixcounts reflect that not as much importance is attached to these requirements as there is to the others, remembering that the output from the policy is a set of desired mixcounts.

As has been mentioned, the formula for calculating the number of scheduled processes involves the sum of the scheduled and suspended processes, with some consideration given to the amount of available memory in case this sum is zero. The actual formula used is the expression

$$s + u - (a - r)/m \tag{4.7}$$

where s is the observed number of scheduled processes,

u is the observed number of suspended processes,

a is the observed amount of available memory in words,

r is the amount of memory in words that the low-level scheduler attempts to keep free (by suspending processes),

and m is the estimated mean amount of memory required for a process.

Both r and m have been taken as 16000 words. Finally the number of stages for the initial problem solution has been chosen as 6, with a time interval of 60 seconds.

The Burroughs B6700 MCP provides several useful mechanisms for allowing programs to interact with it, mostly taking the form of MCP procedures which are externally callable by a certain class of programs. Firstly, the procedure SYSTEMSTATUS provides the caller with a wide range of information concerning the instantaneous state of the system. This is used by our implementation for determining the amount of available memory, the number of scheduled processes, and the number of suspended processes. Secondly, the procedure DCKEYIN allows the calling program to behave as if it were an operator, by entering commands and receiving responses, this being used to determine the mixcounts of the normal batch queues and to set the new queue mixlimits. The existence of these mechanisms means that the operating system itself does not need to be modified, since an ordinary program, given the necessary 'security clearance', may interact with it. Finally the use of an ordinary program to enhance the high-level scheduling mechanism is further expedited by the provision within the MCP for what is known as a SUPERVISOR. Once a program has been nominated as the SUPERVISOR, then it is automatically initiated whenever the operating system is restarted, such as after a system failure.

Considering briefly the practicality of the implementation, the overheads involved in running the program, to observe the state of the system and apply the policy to determine a new set of queue mixlimits every 60 seconds, turned out to be in the order of 0.1% of CPU time. Given that this could probably be reduced by an order of magnitude from this initial implementation by incorporating the policy application function into the operating system itself, the practicality of implementing these extensions to the high-level scheduling mechanism is assured.

4.4 RESULTS.

To test the performance of the application, data obtained from an independent source have been used to compare various aspects of system performance with and without the controlling program present. This data collection is part of an earlier implemented performance measurement and reporting system [24], which essentially collects all data which is available from the operating system. The data collected during twelve days of running without the controlling program have been analysed in conjunction with a similar amount of data collected after the program was running in its final form. Some teething problems with the implementation have meant that there is a delay of several months between the two sets of data, which may have some implications for the analysis. The analysis itself consists of two parts, reflecting the composite nature of the optimality criterion. The first of these consists of the analysis of overall system performance to determine the effect of attempting to control the degree of multiprogramming, and the second consists of an analysis of the relative service given to the different classes of normal batch customer.

In an attempt to reduce the inherent variance in the data, which have been collected at approximately one minute intervals, the analyses have been performed on the averages of these data over twelve minute intervals. Further, the data have been selected in an attempt to reduce the influence of external variations. For instance, only those twelve minute intervals during which at least some batch work was queued have been considered. Also, the time period 12 midday to 1 pm is not considered because that time slot is reserved for, and frequently used by, software maintenance staff. Similarly the whole of Monday morning is not considered since that time is often used by hardware maintenance engineers. Finally, the time periods 11 am to 12 midday and after 4 pm are not considered either because the operating environment changed between the two sets of data for these time periods. In particular, when the implementation was not running, these times used to be reserved for batch work only, but before the final implementation was running these times were changed to include interactive work as well. Thus to summarise, the data have been analysed for all twelve minute intervals in the time slots 9 am to 11 am (except Mondays), and 1 pm to 4 pm, during which there was at least one batch job queued.

4.4.1 Analysis of Overall Performance.

In analysing whether or not the attempt to control the degree of multiprogramming has resulted in improved overall system performance, it first must be decided just how this performance is to be measured. The aim of using the high-level scheduling mechanism to control the number of scheduled processes is to provide the low-level scheduling mechanism with a continuous selection of processes from which to chose when deciding to allocate resources. The reason for doing this is the assertion that if the low-level scheduler always has a choice, then it can make better decisions. Thus what should be measured to determine a change in performance is that variable which the low-level scheduler is trying to optimise, which in general is resources utilisation. One of the ways this can be measured is through the measurement of the utilisation of the central processor. As it happens, the data collected include processor idle time, the complement of processor utilisation, and it is this variable which has been analysed.

Figures 4.1 and 4.2 show the distributions of idle time, expressed as a percentage of elapsed time, for the two different situations, namely when the high-level scheduling parameters were operator controlled, and when they were program controlled. Because of the non-normal nature of the distributions, which is to be expected, normal tests of significant differences cannot be used with any confidence. However, non-parametric tests may be used in this situation, and in particular the Wilcoxon U-test may be used to test for differences in the means. This test involves ranking the joint data in a specified way, and adding the ranks of one of the subsets. A statistic involving this sum may then be tested against a normal distribution.

To test for a difference in the means, the null hypothesis is taken that the two sets of data come from the same distribution, against the alternative that the second set of data comes from a distribution which has a lower mean. This results in a z-value of 2.16, which indicates a significant result at the 2% level for a one-sided test. That is, at the 2% level of significance, the null hypothesis is rejected on the basis of differences in the means, with the mean for the second set of data being lower. This means that the application involving dynamic control over the degree of multiprogramming, through the queue mixlimits, has shown a small but statistically significant increase in CPU utilisation, as measured by idle time, over the situation involving operator setting of the high-level scheduling parameters. In interpreting this result,

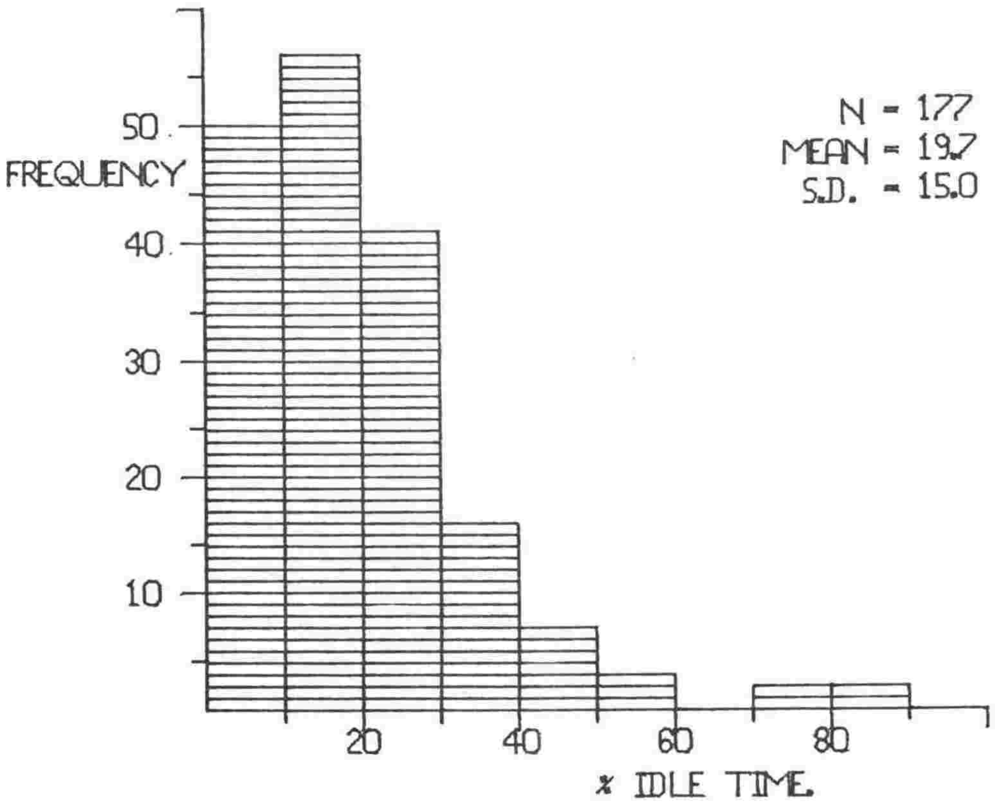


Figure 4.1 DISTRIBUTION OF IDLE TIME - MANUAL CONTROL.

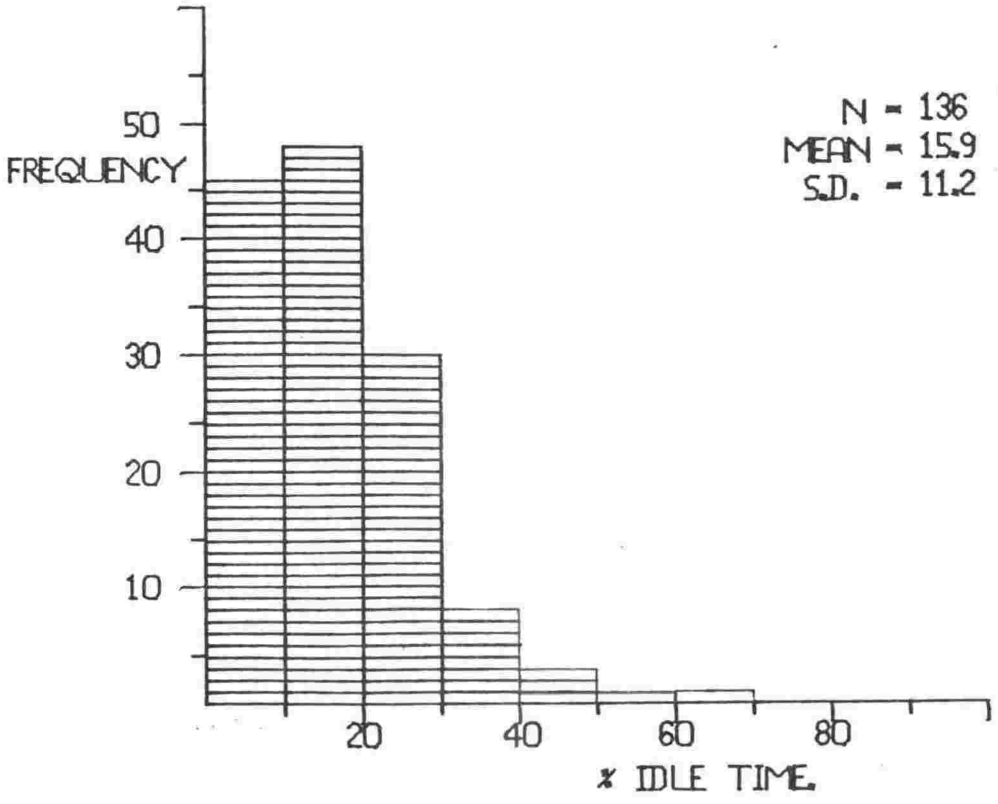


Figure 4.2 DISTRIBUTION OF IDLE TIME - PROGRAM CONTROL.

cognisance must be taken of the possible variation caused by uncontrollable differences between the two samples, such as differences in work load present. Further, it must be remembered that the amount of control that can be imposed by the application is limited to that portion of the workload that is not interactive or special jobs. Thus we cannot state for certain that an improvement in overall performance has resulted from the application of dynamic control of the normal batch work. However, what we can state, and this is possibly a more important observation, is that this application certainly has not resulted in a reduction in CPU utilisation, which means that there are no overall performance losses which could offset gains made in the consideration of the other performance factors.

4.4.2 Analysis of Batch Queue Service.

The provision of different levels of service to batch customers is incorporated into the Dynamic Programming formulation by specifying the weighted squared queue lengths as part of the cost function, the weights themselves reflecting the desired different levels of service. In attempting to minimise this function, the solution will attempt to keep the actual squared queue lengths in inverse proportion to the weights associated with them. Thus to test the effectiveness of the implementation, some measure of how well the queue lengths adhere to this relationship is required. The method chosen for this is to perform a least squares regression on the pairs of queue lengths, and to use the variance of the residuals as an inverse measure of the goodness of the relationship. However it must be pointed out that the goal provided for by the cost function is not in conflict with the sort of goals that the operators were aiming for when the parameter setting was done by hand. This fact is necessary for any significant differences in the variances of the residuals to be meaningful.

Figures 4.3 through 4.8 show the regressions of the queue lengths taken pairwise, before and after the implementation of program control of the high-level scheduling parameters. The slope of the regression line and the variance of the residuals for each regression have been specified on the diagrams.

Taking these pairwise, the ratios of the residual variances can be used to test for differences in the goodness of fit of the regression equations. The null hypothesis in each case is that the residual variances are the same, with the alternative that the variance of the second set of data (from when the queue mixcounts were controlled dynamically) is lower. Firstly, for the regression of Queue 5 against Queue 3, the F-ratio is 11.2. This is a very strong result, which rejects the null hypothesis at the 2% level of significance, and even at the 0.1% level. We can infer from this that the dynamic control of the queue mixcounts does result in a significantly better fit of the length of Queue 5 to the length of Queue 3. Similarly for the regression of Queue 7 against Queue 3, the F-ratio is 3.07, which also indicates that the null hypothesis would be rejected at the 2% level, and also as low as the 0.1% level. Once again this indicates a significantly better fit of the length of Queue 7 to the length of Queue 3 when the queue mixcounts are being controlled dynamically. Finally, for the regression of Queue 7 against Queue 5, the F-ratio is 1.54, which is significant

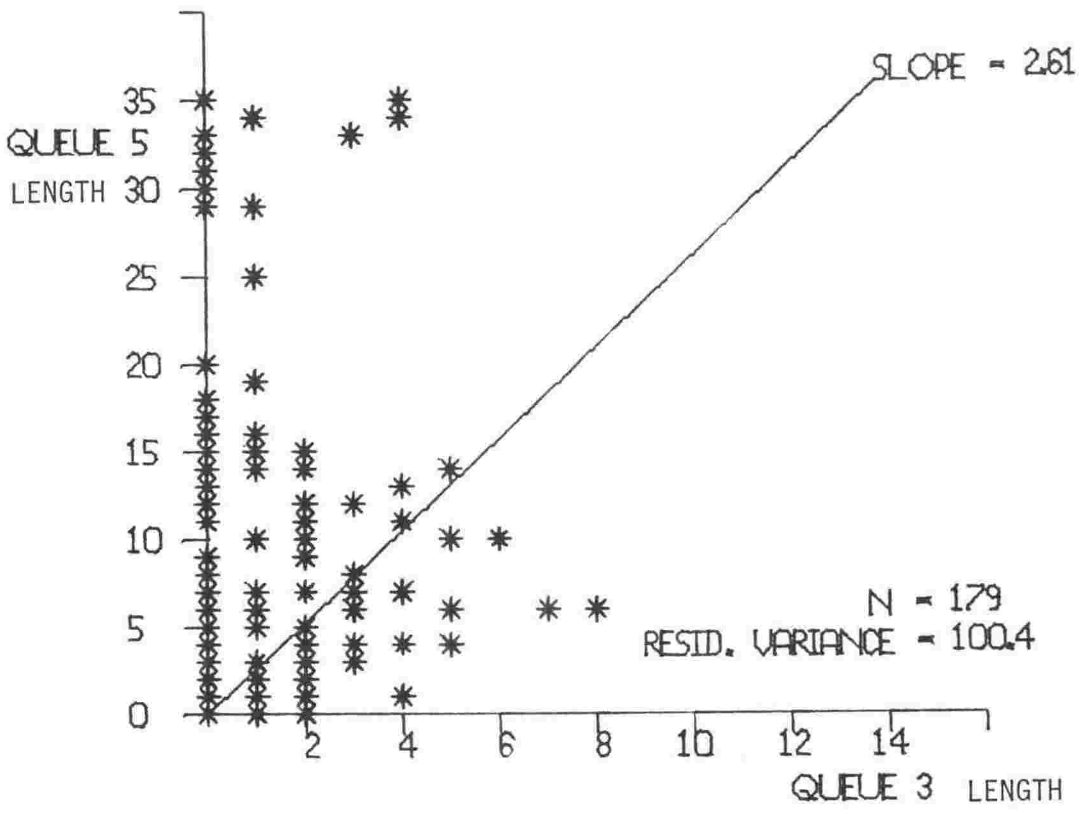


Figure 4.3 QUEUE 5 VS. QUEUE 3 - MANUAL CONTROL.

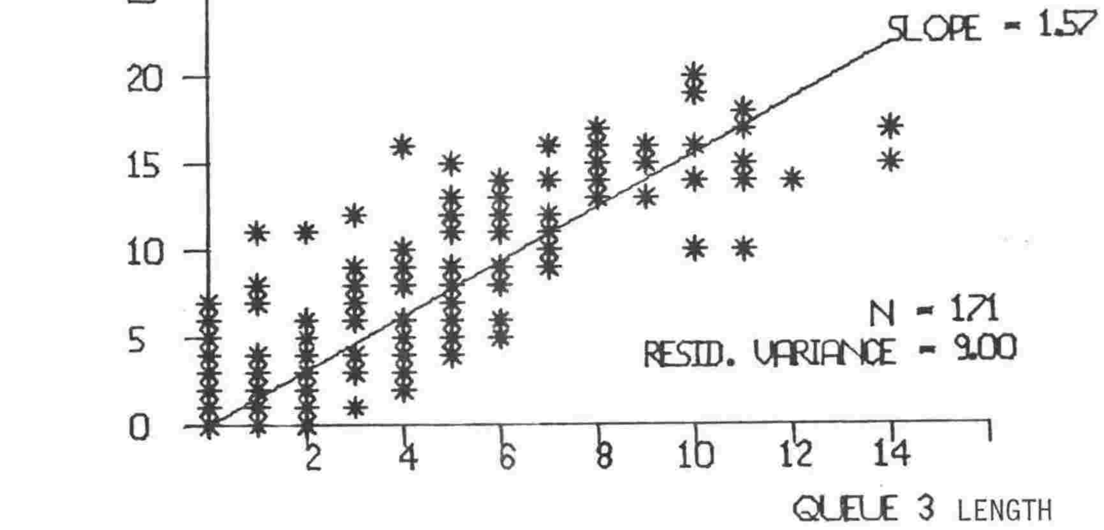


Figure 4.4 QUEUE 5 VS. QUEUE 3 - PROGRAM CONTROL.

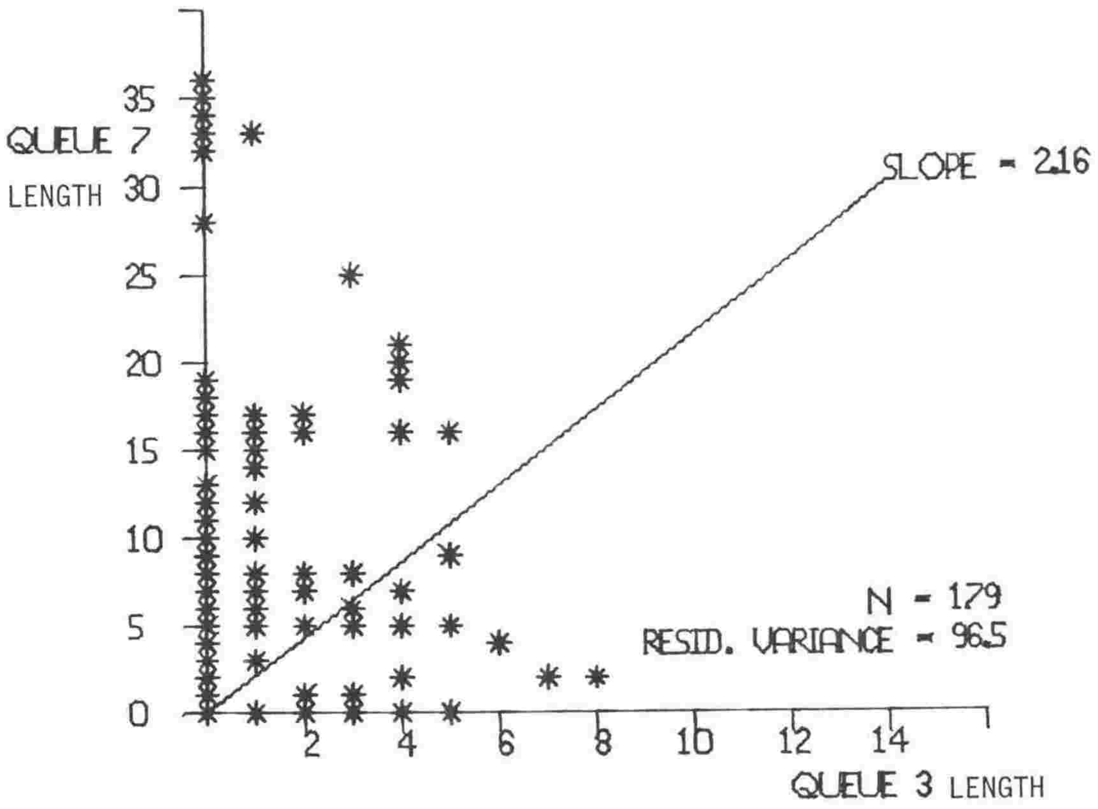


Figure 4.5 QUEUE 7 VS. QUEUE 3 - MANUAL CONTROL.

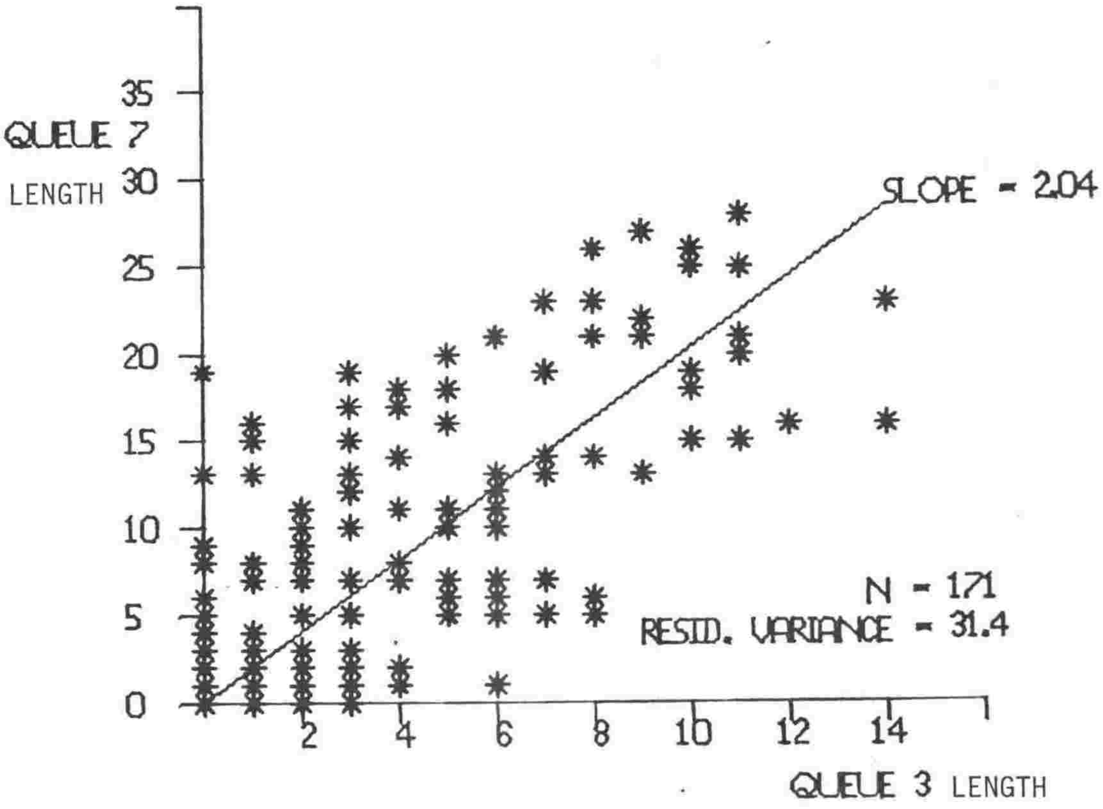


Figure 4.6 QUEUE 7 VS. QUEUE 3 - PROGRAM CONTROL.

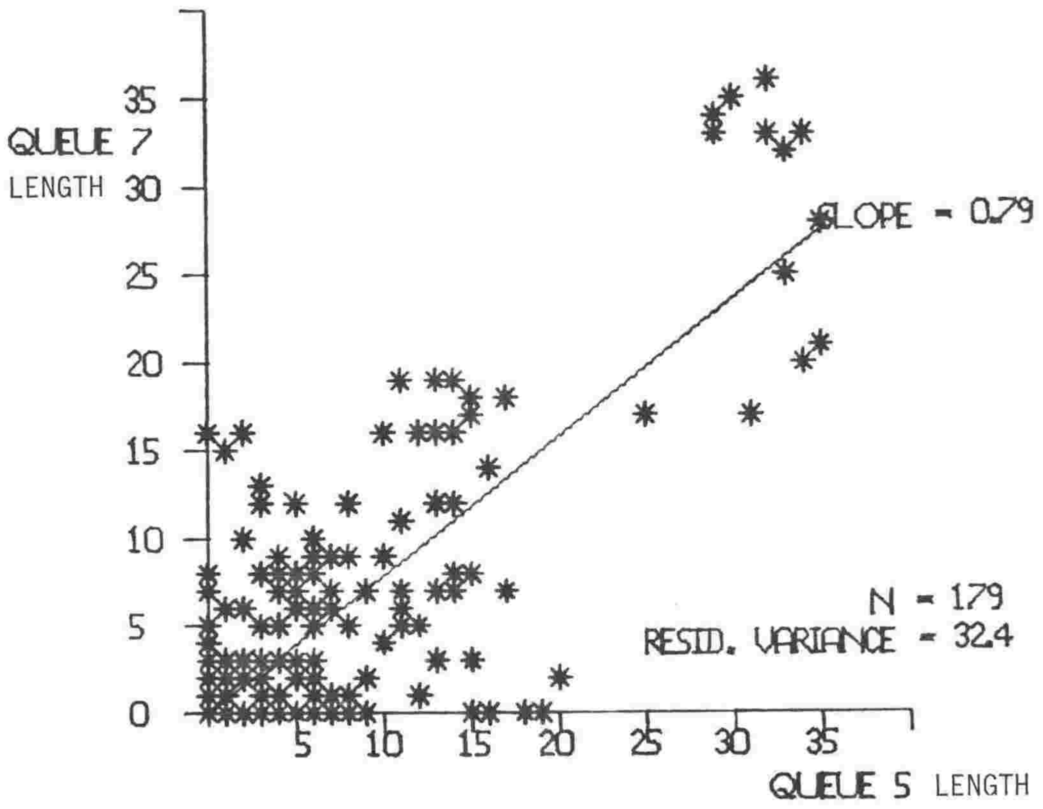


Figure 4.7 QUEUE 7 VS. QUEUE 5 – MANUAL CONTROL.

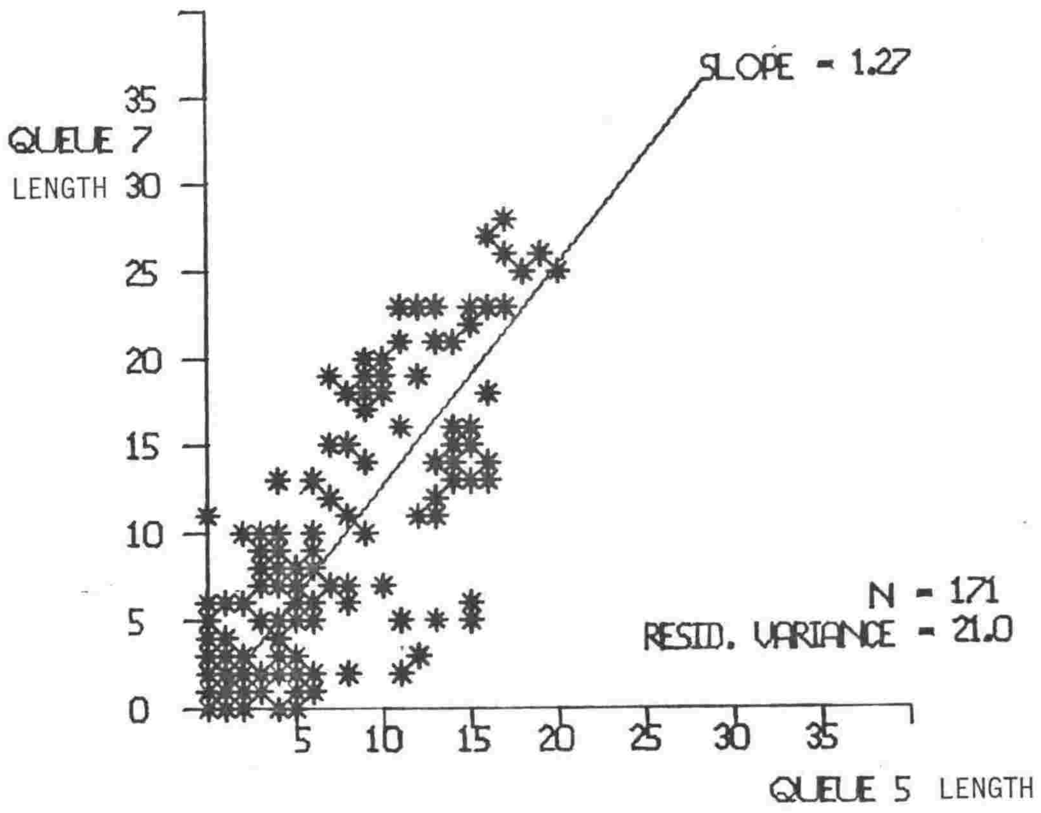


Figure 4.8 QUEUE 7 VS. QUEUE 5 – PROGRAM CONTROL.

at the 2% level, but is not significant at the 0.1% level. Thus on the basis of the data analysed, and bearing in mind the unknown effects of different workloads and, more importantly, different arrival rates, we cannot confidently reject the null hypothesis that the variances of the residuals are the same. In other words the fit of the length of Queue 7 to the length of Queue 5 may not have been significantly improved by the dynamic control of the queue mixcounts.

Considering these three results together, it would appear that the major effect of the dynamic control of the queue mixcounts has been to relate the lengths of Queues 5 and 7, which were already tied together somewhat, to the length of Queue 3. Whether or not this has resulted in a degradation of service for Queue 3 is an unanswered question which is beyond the scope of this application since the cost function used for the Dynamic Programming problem reflects a 'management decision' which is more concerned with relative service to the different queues than with absolute service to any one queue. Thus what can be stated is that in terms of the criterion reflected in the choice of cost function, the application has succeded in achieving improvements in performance.

4.5 CONCLUSIONS.

The application of Variable Metric Dynamic Programming to the problem of dynamically modifying some of the high-level scheduling parameters in a batch and interactive computing system has been successful. This success is a result of the fact that on the basis of each component of a composite criterion, significant improvements in performance have been obtained. Further, since the implementation is an initial one, for which many variables have been assigned approximate or nominal values, experimentation with some of these variables is likely to lead to further improvements. Not only does this demonstrate the applicability of Variable Metric Dynamic Programming as a solution method, but also it demonstrates the beneficial use of a Mathematical Programming technique in a Computer Science application. These techniques often lend themselves naturally to the inclusion of feedback mechanisms into the application, thereby reducing the need for an accurate system model. This is an advantage in operating systems research, since accurate models are difficult to obtain. A final contribution to the success of the application is the practicality of the implementation, which involves reasonably low, and potentially negligible overheads.

Finally, this application has left unanswered a number of questions which may be used as inspiration for future research topics. These will be discussed in the following section on overall conclusions and implications for future research.

SECTION 5.

CONCLUSIONS.

The successful use of Dynamic Programming as a scheduling tool in a multiprogrammed computing system is a good example of how Mathematical Programming in general may be applied to Computer Science problems in which dynamic feedback is desirable, or even necessary, to take account of unforseeable variations. The success of the application has further implications for the viability in a practical situation of Variable Metric Dynamic Programming, being the particular Dynamic Programming solution method used.

Considering firstly the Variable Metric Dynamic Programming algorithm itself, the main conclusion to be reached from the research directed towards the development of this algorithm is that the use of Variable Metric minimisation techniques for generating quadratic approximations to functions results in a Dynamic Programming solution algorithm which has advantages over existing algorithms. The new iterative algorithm, for the unconstrained N-stage decision problem, compares favourably with the existing Differential Dynamic Programming algorithm. This has been demonstrated by applying both algorithms to the solution of a simple theoretical problem. Further, by analysing the two algorithms themselves, the DDP algorithm is shown to be in some respects a special case of the new VMDP algorithm, which has implications for the range of applicability of the new algorithm.

This initial proposal of the algorithm has left a number of unexplored openings for future research. Of those directly concerning the new algorithm as it stands, the most important is the extension of the algorithm to be able to handle the inclusion of constraints in the problem definition. One further possibility which has not been mentioned is the application of Variable Metric minimisation techniques to the iterative solution of the continuous time problem. This would entail the development of a new algorithm using a similar approach, rather than involving the extension of the existing algorithm.

As a lead in to the application of the new algorithm, a brief literature survey has considered current research on the scheduling of multiprogrammed computing systems. The main conclusions reached by this survey are that the use of feedback mechanisms has been recognised as a potentially powerful scheduling tool, and that research directed towards this is steadily gaining momentum. Further, however, it is concluded that there is not yet a great deal of research directed towards the use of Mathematical Programming techniques, which provide the feedback mechanisms in a natural way.

Finally, considering the application itself, what has been demonstrated is a mechanism for controlling dynamically some of the high-level scheduling parameters in a multiprogrammed computing system. The implementation of this has resulted in the attainment of significant improvements in terms of the actual criteria represented by the cost function being used. Some further research into the 'fine-tuning' of some of the variables being used as parameters is likely to lead to further improvements. The use of Variable Metric Dynamic Programming for the application has resulted in a high-level scheduling mechanism which uses dynamic feedback for a relatively large number of variables. This is because of the natural way the solution policy provided by the VMDP algorithm is applied at each stage to a newly observed value of the state of the system.

A further benefit from using Dynamic Programming is that the solution process operates in such a way as to optimise a cost function which reflects a management decision. In the current implementation, this cost function may be modified to reflect changing management desires, although this cannot be done on a dynamic basis. However, the structure of the Dynamic Programming formulation provides the ability to parameterise a generalised cost function in such a way that dynamic modifications are possible. This involves considering the scalar variables which constitute the cost function parameters to be part of the state of the system, remaining constant unless altered by operator input. A further extension to this would then be to have these cost function parameters modified by some independent feedback mechanism which takes into account a more general form of the management policy requirements.

To summarise, a new iterative Dynamic Programming algorithm, called Variable Metric Dynamic Programming, has been developed, and shown to have advantages over existing algorithms. A brief literature survey concerning the scheduling of multiprogrammed computing systems concludes that the use of Mathematical Programming techniques in computer scheduling is a relatively unexplored but potentially fruitful research direction. Finally the new Dynamic Programming algorithm is applied to a high-level scheduling problem, resulting in a successful demonstration of both the capabilities of the new solution algorithm in a pratical situation and, more generally, the potential power of Mathematical Programming techniques for providing feedback mechanisms.

BIBLIOGRAPHY.

The following abbreviations are used to denote sources of some of the references.

ACM	Association for Computing Machinery
AFIPS	American Federation of Information Processing Societies
FJCC	Fall Joint Computer Conference
IEEE	Institute of Electrical and Electronics Engineers
NRLQ	Naval Research Logistics Quarterly
SIAM	Society for Industrial and Applied Mathematics
SJCC	Spring Joint Computer Conference

1. ABELL, V.A., ROSEN, S., and WAGNER, R.E., *Scheduling in a General Purpose Operating System*, AFIPS FJCC, Vol. 37, 1970.

2. ABRAMS, M.D. and TREU, S., *A Methodology for Interactive Computer Service Measurement*, Communications of the ACM, Vol. 20, No. 12, 1977.

3. AGAJANIAN, A.H., *A Bibliography on System Performance Evaluation*, Computer, Vol. 8, No. 11, 1975.

4. AGGARWAL, S.C. and MCCARL, B.A., *The Development and Evaluation of a Cost-Based Composite Scheduling Rule*, NRLQ, Vol. 21, No. 1, 1974.

5. ALDERSON, A., LYNCH, W.C., and RANDELL, B., *Thrashing in a Multiprogrammed Paging System*, in Operating Systems Techniques, Edited by C.A.R. Hoare and R.H. Perrott, Academic Press, London, 1972.

6. ANDERSON, H.A. and SARGENT, R.G., *Investigation into Scheduling for an Interactive Computing System,* IBM Journal of Research and Development, Vol. 18, No. 2, 1974.

7. ANDERSON, H.A. and SARGENT, R.G., *Modeling, Evaluation, and Performance Measurements of Time-Sharing Computer Systems,* Computing Reviews, Vol. 13, No. 12, 1972.

8. ARBUCKLE, R.A., *Computer Analysis and Throughput Evaluation,* Computers and Automation, Vol. 15, No. 1, 1966.

9. ARORA, J.K. and PIERRE, D.A., *Optimal Trajectories for Multidimensional Non-Linear Processes by Iterated Dynamic Programming,* IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, No. 1, 1973.

10. ARORA, S.A. and KACHHAL, S.K., *Optimisation of Design Parameters in a Virtual Memory System,* Proceedings of the Computer Science and Statistics Seventh Annual Symposium on the Interface, pp. 92-99, 1973.

11. ARTHANARI, T.S. and RAMAMURTHY, K.G., *A Branch and Bound Algorithm for Sequencing N jobs on M Parallel Processors,* Opsearch, Vol. 17, pp. 147-156, 1971.

12. AUSTIN, B.J., HANLON, P.P., and RUSSELL, J.J., *Job Scheduling in a Monoprogrammed Environment,* Australian Computer Journal, Vol. 6, No. 1, 1974.

13. BADEL, M., GELENBE, E., LEROUDIER, J., and POTIER, D., *A Model of Performance for Virtual Memory Systems*, Proceedings of the IEEE, Vol. 63, pp. 958-965, 1975.

14. BAKER, K.R. and MARTIN, J.B., *An Experimental Comparison of Solution Algorithms for the Single Machine Tardiness Problem*, NRLQ, Vol. 21, No. 1, 1974.

15. BALUT, S.J., *Scheduling to Minimise the Number of Late Jobs When Set-Up and Processing Times are Uncertain*, Management Science, Vol. 9, No. 11, 1973.

16. BARD, Y., *Application of the Page Survival Index (PSI) to Virtual Memory System Performance*, IBM Journal of Research and Development, Vol. 19, No. 3, 1975.

17. BARD, Y., *Experimental Evaluation of System Performance*, IBM System Journal, Vol. 12, No. 3, 1973.

18. BASS, L.J., *On Optimal Processor Scheduling for Multiprogramming*, SIAM Journal on Computing, Vol. 2, No. 4, 1973.

19. BELADY, L.A. and KUEHNER, C.J., *Dynamic Space Sharing in Computer Systems*, Communications of the ACM, Vol. 12, No. 5, 1969.

20. BEN-BASSAT, M. and BOROVITS, I., *Computer Network Scheduling*, OMEGA, Vol. 3, No. 1, 1975.

21. BERNSTEIN, A.J. and SHARPE, J.C., *A Policy-Driven Scheduler for a Time-Sharing System,* Communications of the ACM, Vol.14, No. 2, 1971.

22. BLEVINS, P.R. and RAMAMOORTHY, C.V., *Aspects of a Dynamically Adaptive Operating System,* IEEE Transactions on Computers, Vol. C-25, No. 7, 1976.

23. BONNER, A.J., *Using System Monitor Output to Improve Performance,* IBM Systems Journal, Vol. 8, No. 4, 1969.

24. BROWNRIGG, R.D., *B6700 Performance Reporting,* Technical Report No. 60, Applied Mathematics Division, Department of Scientific and Industrial Research, New Zealand, 1977.

25. BROWNRIGG, R.D., *Parallel Processing and Dynamic Programming,* M.Sc. Thesis, Victoria University of Wellington, New Zealand, 1974.

26. BRUNO, J., COFFMAN, E.G. JR., and SETHI, R., *Scheduling Independent Tasks to Reduce Mean Finishing Time,* Communications of the ACM, Vol. 17, No. 7, 1974.

27. BUCHHOLZ, W., *A Selected Bibliography on Computer System Performance Evaluation,* Computer Group News, Vol. 2, No. 8, 1969.

28. BUNT, R.B., *Scheduling Techniques for Operating Systems,* Computer, Vol. 9, No. 10, 1976.

29. BUNT, R.B., *Self-Regulating Schedulers for Operating Systems,* Technical Report No. 76, University of Toronto, 1975.

30. BUNT, R. and TSICHRITZIS, D., *An Annotated Bibliography for Operating Systems,* Computing Reviews, Vol. 13, No. 8, 1972.

31. CALINGAERT, P., *System Performance Evaluation: Survey and Appraisal,* Communications of the ACM, Vol. 10, No. 1, 1967.

32. CHANDY, K.M. and YEH, R.T., (Editors), *Current Trends in Programming Methodology. Volume III, Software Modeling,* Prentice Hall, Englewood Cliffs, 1978.

33. CHANSON, S.T. and BISHOP, C.D., *A Simulation Study of Adaptive Scheduling Policies in Interactive Computer Systems,* Performance Evaluation Review, Vol. 6, No. 3, 1977.

34. CHEN, P.P.S. and FRANKLIN, M., (Editors), *International Symposium on Computer Performance Modeling, Measurement, and Evaluation,* ACM, New York, 1976.

35. CHENG, P.S., *Trace Driven System Modeling,* IBM Systems Journal, Vol. 8, No. 4, 1969.

36. CHESTER, K.W., *Computer Resource Allocation,* Datamation, Vol. 20, No. 9, 1974.

37. CHU, W.W. and OPDERBECK, H., *Analysis of the PFF Replacement Algorithm via a Semi-Markov Model,* CACM, Vol. 19, No. 5, 1976.

38. CHUA, Y.S. and BERNSTEIN, A.J., *Analysis of a Feedback Scheduler,* SIAM Journal on Computing, Vol. 3, No. 3, 1974.

39. CLARKE, S.R. and ROURKE, T.A., *A Simulation Study of the Effects of Various Job-Scheduling Algorithms in Computer Systems,* INFOR, Vol. 10, No. 3, 1972.

40. CODD, E.F., *Multiprogram Scheduling: Parts 1 and 2, Introduction and Theory,* Communications of the ACM, Vol. 3, No. 6, 1960.

41. CODD, E.F., *Multiprogram Scheduling: Parts 3 and 4, Scheduling Algorithm and External Constraints,* Communications of the ACM, Vol. 3, No. 7, 1960.

42. COFFMAN, E.G. JR., *Analysis of Two Time-Sharing Algorithms Designed for Limited Swapping,* Journal of the ACM, Vol. 15, No. 3, 1968.

43. COFFMAN, E.G. JR. and DENNING, P.J., *Operating Systems Theory,* Prentice Hall Series in Automatic Computation, 1973.

44. COFFMAN, E.G. JR. and GRAHAM, R.L., *Optimal Scheduling for Two-Processor Systems,* Acta Informatica, Vol. 1, No. 3, 1972.

45. COFFMAN, E.G. JR. and KLEINROCK, L., *Computer Scheduling Methods and Their Countermeasures,* AFIPS SJCC, Vol. 32, 1968.

46. COFFMAN, E.G. JR. and MUNTZ, R.R., *Models of Pure Time-Sharing Disciplines for Resource Allocation,* Proceedings of the ACM National Conference, Vol. 24, pp. 217-228, 1969.

47. COFFMAN, E.G. JR., *Deadlocks in Computer Systems,* in Operating Systems: International Computer State of the Art Report, edited by C. Boon, 1972.

48. COFFMAN, E.G. JR., ELPHICK, M.J., and SHOSHANI, A., *System Deadlocks,* ACM Computing Surveys, Vol. 3, No. 2, 1971.

49. COLIN, A.J.T., *Introduction to Operating Systems,* Macdonald and American Elsevier, 1971.

50. CONWAY, R.W., MAXWELL, W.L., and MILLER, L.W., *Theory of Scheduling,* Addison Wesley, Reading, Massachusetts, 1967.

51. CRITCHLOW, A.J., *Generalised Multiprocessing and Multiprogramming Systems,* AFIPS FJCC, 1963.

52. CROOKE, S., MINKER, J., and YEH, J., *Key Word in Context and Bibliography on Computer Systems Evaluation Techniques,* Technical Report No. TR-146, University of Maryland Computer Science Centre, 1971.

53. DAHM, G.M., GERBSTADT, F.H., and PACELLI, M.M., *A System Organisation for Resource Allocation,* Communications of the ACM, Vol. 10, No. 12, 1967.

54. DE MEIS, W.M. and WEIZER, N., *Measurement and Analysis of a Demand Paging Time-Sharing System,* Proceedings of the ACM National Conference, Vol. 33, pp. 201-216, 1969.

55. DENNING, P.J., *Resource Allocation in Multiprocess Computer Systems,* Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1968.

56. DENNING, P.J., *The Working Set Model for Program Behaviour,* Communications of the ACM, Vol. 11, No. 5, 1968.

57. DENNING, P.J., *Thrashing: Its Causes and Prevention,* AFIPS FJCC, Vol. 33, 1968.

58. DOELLING, N., *Computer Resources Sharing - Some Comments and Predictions,* Computers and Automation, Vol. 17, No. 10, 1968.

59. DRUMMOND, M.E. JR., *A Perspective on System Performance Evaluation,* IBM Systems Journal, Vol. 8, No. 4, 1969.

60. EASTMAN, W.L., EVEN, S., and ISAACS, J.M., *Bounds for the Optimal Scheduling of N jobs on M Processors,* Management Science, Vol. 11, No. 2, 1964.

61. EILON, S. and CHOWDHURY, I.G., *A Note on Steady-State Results in Queueing and Job-Shop Scheduling,* Simulation, Vol. 23, No. 3, 1974.

62. ELMAGHRABY, S.E., *The Machine Sequencing Problem - Review and Extensions,* NRLQ, Vol. 15, No. 2, 1968.

63. ESTRIN, G., *Review of 'System Performance Evaluation: Survey and Appraisal', by P. Calingaert,* Computing Reviews, Vol. 8, p. 160, 1967.

64. FERNANDEZ, E.B. and BUSSELL, B., *Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules,* IEEE Transactions on Computers, Vol. C-22, No. 8, 1973.

65. FISHER, M.L., *Optimal Solution of Scheduling Problems Using Lagrange Multipliers: Part 1,* Operations Research, Vol. 21, No. 5, 1973.

66. GANNON, J.D., (Editor), *An Annotated Bibliography on Computer Program Engineering (2nd Edition),* Report No. CSRG-31, Computer Science Research Group, University of Toronto, 1974.

67. GAVER, D.P. and SCHEDLER, G.S., *Approximate Models for Processor Utilisation in Multiprogrammed Computer Systems,* SIAM Journal on Computing, Vol. 2, No. 3, 1973.

68. GAVER, D.P. and SCHEDLER, G.S., *Processor Utilisation in Multiprogramming Systems via Diffusion Approximations,* Operations Research, Vol. 21, No. 2, 1973.

69. GELDERS, L. and KLEINDORFER, P.R., *Coordinating Aggregate and Detailed Scheduling Decisions in the One-Machine Job-Shop: Part 1, Theory,* Operations Research, Vol. 22, No. 1, 1974.

70. GELENBE, E. and KURINCKX, A., *Random Injection Control of Multiprogramming in Virtual Memory,* IEEE Transactions on Software Engineering, Vol. SE-4, No. 1, 1978.

71. GUTTAG, J., *An Annotated Bibliography on Computer Program Engineering (3rd Edition)*, Report No. CSRG-54, Computer Science Research Group, University of Toronto, 1975.

72. HABERMANN, A.N., *Prevention of System Deadlocks*, Communications of the ACM, Vol. 12, No. 7, 1969.

73. HAMLET, R.G., *Efficient Multiprogramming Resource Allocation and Accounting*, Communications of the ACM, Vol. 16, No. 6, 1973.

74. HARALAMBOPOULOS, G. and NAGY, G., *Profile of a University Computer User Community*, International Journal of Man-Machine Studies, Vol. 9, No. 3, 1977.

75. HAVENDER, J.W., *Avoiding Deadlocks in Multitasking Systems*, IBM Systems Journal, Vol. 7, No. 2, 1968.

76. HEACOX, H.C. and PURDOM, P.W., *Analysis of Two Time-Sharing Queueing Models*, Journal of the ACM, Vol. 19, No. 1, 1972.

77. HEIDEMAN, J.C. and LEVY, A.V., *Sequential Conjugate-Gradient-Restoration Algorithm for Optimal Control Problems, Part 1, Theory*, Journal of Optimisation Theory and Applications, Vol. 15, No. 2, 1975.

78. HEIDEMAN, J.C. and LEVY, A.V., *Sequential Conjugate-Gradient-Restoration Algorithm for Optimal Control Problems, Part 2, Examples*, Journal of Optimisation Theory and Applications, Vol. 15, No. 2, 1975.

79. HELD, M. and KARP, R.M., *A Dynamic Programming Approach to Sequencing Problems,* SIAM Journal, Vol. 10, No. 1, 1962.

80. HELLER, J., *Sequencing Aspects of Multiprogramming,* Journal of the ACM, Vol. 8, No. 3, 1961.

81. HELLERMAN, H., *Some Principles of Time-Sharing Scheduler Strategies,* IBM Systems Journal, Vol. 8, No. 2, 1969.

82. HOARE, C.A.R. and PERROTT, R.H., (Editors), *Operating Systems Techniques,* Academic Press, London, 1972.

83. HOLLOWAY, C.A. and NELSON, R.T., *A Procedure for Job Shop Scheduling with Due Dates, Variable Processing Times, and a Dynamic Arrival Process,* Research Paper No. 123, Graduate School of Business, Stanford University, 1972.

84. HOLLOWAY, C.A. and NELSON, R.T., *Job Shop Scheduling with Due Dates and Variable Processing Times,* Management Science, Vol. 20, No. 9, 1974.

85. HOLT, R.C., *Some Deadlock Properties of Computer Systems,* ACM Computing Surveys, Vol. 4, No. 3, 1972.

86. HORN, W.A., *Some Simple Scheduling Algorithms,* NRLQ, Vol. 21, No. 1, 1974.

87. JACOBSON, D.H. and MAYNE, D.Q., *Differential Dynamic Programming,* American Elsevier, New York, 1970.

88. KAMEDA, H., *The Analysis of an Adaptive Workload Balancing Strategy in Computing Systems Resource Management,* International Journal of Computer and Information Sciences, Vol. 4, No. 4, 1975.

89. KIMBLETON, S.R. and BAKER, K.R., *A Heuristically Oriented Approach to Scheduling Batch Production Oriented Computer Systems,* Proceedings of the 7th Hawaii International Conference on System Sciences, pp. 28-30, 1974.

90. KLEINROCK, L., *A Conservation Law for a Wide Class of Queueing Disciplines,* NRLQ, Vol. 12, No. 2, 1965.

91. KLEINROCK, L., *A Continuum of Time-Sharing Algorithms,* AFIPS SJCC, Vol. 36, 1970.

92. KLEINROCK, L. and MUNTZ, R.R., *Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time-Shared Systems,* Journal of the ACM, Vol 19, No. 3, 1972.

93. LAMPSON, B.W., *A Scheduling Philosophy for Multiprocessing Systems,* Communications of the ACM, Vol. 11, No. 5, 1968.

94. LARMOUTH, J., *Scheduling for a Share of the Machine,* Software Practice and Experience, Vol. 5, No. 1, 1975.

95. LARMOUTH, J., *Scheduling for Immediate Turnround,* University of Salford Computing Laboratory, 1977.

96. LAWLER, E.L., *On Scheduling Problems with Deferral Costs,* Management Science, Vol 11, No. 2, 1964.

97. LIU, C.L. and LAYLAND, J.W., *Scheduling Algorithms for Multiprocessing in a Hard-Real-Time Environment,* Journal of the ACM, Vol. 20, No. 1, 1973.

98. LORIN, H., *Parallelism in Hardware and Software: Real and Apparent Concurrency,* Prentice Hall, 1971.

99. LUCAS, H.C., *Performance Evaluation and Monitoring,* ACM Computing Surveys, Vol. 3, No. 3, 1971.

100. LYNCH, H.W. and PAGE, J.B., *The OS/VS2 Release 2 System Resource Manager,* IBM Systems Journal, Vol. 13, No. 4, 1974.

101. MARSHALL, B.S., *Dynamic Calculation of Dispatching Priorities Under OS/360 MVT,* Datamation, Vol. 15, No. 8, 1969.

102. MAYNE, D.Q., *A Second-Order Gradient Method for Determining Optimal Trajectories of Non-Linear Discrete-Time Systems,* International Journal of Control, Vol. 3, No. 1, 1966.

103. MCKINNEY, J.M., *A Survey of Analytic Time-Sharing Models,* Computing Surveys, Vol. 1, No. 2, 1969.

104. MCNAUGHTON, R., *Scheduling with Deadlines and Loss Functions,* Management Science, Vol. 6, No. 1, 1959.

105. MUNTZ, R.R. and COFFMAN, E.G. JR., *Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems,* Journal of the ACM, Vol. 17, No. 2, 1970.

106. MURPHY, J.E., *Resource Allocation with Interlock Detection in a Multitask System*, AFIPS FJCC, Vol. 33, Part 2, 1968.

107. MURTAGH, B.A. and SARGENT, R.W.H., *A Constrained Minimisation Method with Quadratic Convergence*, in Optimisation, Edited by R. Fletcher, Academic Press, London, 1969.

108. MURTAGH, B.A. and SARGENT, R.W.H., *Computational Experience with Quadratically Convergent Minimisation Methods*, Computer Journal, Vol. 13, No. 2, 1970.

109. NABESHIMA, I, *General Scheduling Algorithms with Applications to Parallel Scheduling and Multiprogramming Scheduling*, Journal of the Operations Research Society of Japan, Vol. 14, No. 2, 1971.

110. NEW, C., *Job-Shop Scheduling*, Data Processing, Vol. 16, No. 2, 1974.

111. NIELSEN, N.R., *An Analysis of Some Time-Sharing Techniques*, Communications of the ACM, Vol. 14, No. 2, 1971.

112. NIELSEN, N.R., *Flexible Pricing: An Approach to the Allocation of Computer Resources*, AFIPS FJCC, Vol. 33, Part 1, 1968.

113. NIELSEN, N.R., *The Allocation of Computer Resources - Is Pricing the Answer?*, Communications of the ACM, Vol. 13, No. 8, 1970.

114. NORTHOUSE, R.A. and FU, K.S., *Dynamic Scheduling of Large Digital Computer Systems Using Adaptive Control and Clustering Techniques*,

IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-3, No. 3, 1973.

115. OPPENHEIMER, G. and WEIZER, N., *Resource Management for a Medium-Scale Time-Sharing Operating System,* Communications of the ACM, Vol. 11, No. 5, 1968.

116. PASS, E.M. and GWYNN, J., *An Adaptive Microscheduler for a Multiprogrammed Computer System,* Proceedings of the ACM National Conference, 1973.

117. POTIER, D., GELENBE, E., and L'ENFANT, J., *Adaptive Allocation of Central Processing Unit Quanta,* Journal of the ACM, Vol 23, No. 1, 1976.

118. PRICE, T.G., *An Analysis of Central Processor Scheduling in Multiprogrammed Computer Systems,* Technical Report No. 57, Digital Systems Laboratory, Stanford Electronics Laboratory, 1972.

119. PULLAN, W.J., *A Simulation of a B6700 Computer System,* M.Sc. Thesis, Otago University, Dunedin, New Zealand, 1976.

120. RAMSAY, K. and STRAUSS, J.C., *A Real Time Priority Scheduler,* Proceedings of the ACM National Conference, Vol. 21, 1966.

121. REITER, A., *A Resource Allocation Scheme for Multi-User On-Line Operation of a Small Computer,* AFIPS SJCC, Vol. 30, 1967.

122. RODRIGUEZ-ROSELL, J. and DUPUY, J-P., *The Design, Implementation, and Evaluation of a Working Set Dispatcher,* Communications of the ACM, Vol. 16, No. 4, 1973.

123. ROSEN, S., *Lectures on the Measurement and Evaluation of the Performance of Computing Systems,* SIAM Regional Conference Series in Applied Mathematics, No. 23, 1976.

124. ROTHKOPF, M.H., *Scheduling Independent Tasks on Parallel Processors,* Management Science, Vol. 12, No. 5, 1966.

125. SCHNECK, P.B., *The Myth of Multiprogramming,* Software Practice and Experience, Vol. 4, No. 1, 1974.

126. SEAMAN, P.H. and SOUCY, R.C., *Simulating Operating Systems,* IBM Systems Journal, Vol. 8, No. 4, 1969.

127. SEVCIK, K.C., *Optimal Processor Scheduling When Service-Times are Hyperexponentially Distributed and Preemption Overhead is Not Negligible,* Technical Report No. CSRG-16, Computer Systems Research Group, University of Toronto, 1972.

128. SEVCIK, K.C., *Scheduling for Minimum Total Loss Using Service-Time Distributions,* Journal of the ACM, Vol. 21, No. 1, 1974.

129. SEVCIK, K.C., *The Use of Service-Time Distributions in Scheduling,* Technical Report No. CSRG-14, Computer Systems Research Group, University of Toronto, 1972.

130. SHERMAN, S., BASKETT, F., and BROWNE, J.C., *Trace-Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System*, Communications of the ACM, Vol. 15, No. 12, 1972.

131. SHIRLEY, S., *Evaluating Benchmark Tests*, Data Systems, September 1969, pp. 31-33.

132. SPINNER, A.H., *Sequencing Theory - Development to Date*, NRLQ, Vol. 15, No. 2, 1968.

133. STEVENS, D.F., *On Overcoming High-Priority Paralysis in Multiprogramming Systems: A Case History*, Communications of the ACM, Vol. 11, No. 8, 1968.

134. STIMLER, S., *Some Criteria for Time-Sharing System Performance*, Communications of the ACM, Vol. 12, No. 1, 1969.

135. THE COMPTRE CORPORATION, *Operating Systems Survey*, Edited by A.P. Sayers, Auerbach, 1971.

136. THESEN, A., *Scheduling of Computer Programs for Optimal Machine Utilisation*, BIT, Vol. 13, No. 2, 1973.

137. TSICHRITZIS, D.C. and BERNSTEIN, P.A., *Operating Systems*, Academic Press, New York, 1974.

138. VARIAN, L.C. and COFFMAN, E.G. JR., *An Empirical Study of the Behaviour of Programs in a Paging Environment*, Proceedings of the ACM Symposium on Operating System Principles, 1967.

139. WALKE, B., *Program Replacement for Better Throughput,* IEEE Transactions on Software Engineering, Vol. SE-3, No. 5, 1977.

140. WICKENS, R.F., *A Brief Review of Computer Assessment Methods,* Radio Electronics Engineering, Vol. 26, No. 5, 1968.