RAIRO. RECHERCHE OPÉRATIONNELLE

WILLIAM W. AGRESTI

Nonserial dynamic programming for optimal register assignment

RAIRO. Recherche opérationnelle, tome 17, n° 1 (1983), p. 63-97

<http://www.numdam.org/item?id=RO_1983__17_1_63_0>

R.A.I.R.O. Recherche opérationnelle/Operations Research
(vol. 17, n° 1, février 1983, p. 63 à 97)

NONSERIAL DYNAMIC PROGRAMMING FOR OPTIMAL REGISTER ASSIGNMENT (*)

by William W. AGRESTI (1)

Abstract. — *Nonserial dynamic programming is used to model a decision process arising from the processing of computer programs. The specific probelm involves the optimal assignment of quantities to index registers during the execution of the program. The program may exhibit highly nonlinear flow of control with loops and branches. The dynamic-programming model represents the system as a multi-stage decision process and leads to a solution which specifies the optimal decisions to be made at each stage. An example featuring a feedforward loop is solved by the method and the recursion equations are given for a much more complex example.*

Keywords: dynamic programming, assignment problem.

Résumé. — Un problème décision se posant lors de la compilation d'un programme est modelisé par la programmation dynamique non séquentielle. Il s'agit du problème de l'affectation des indices aux registres d'index dans le cas général d'une structure de contrôle quelconque (branchements, boucles). La programmation dynamique représente le système comme un processus décisionnel à plusieurs étapes et fournit les décisions optimales pour chacune d'elles. Un premier exemple est développé complètement tandis que sont formulées les équations de récurrence associées à un second exemple plus complexe.

1. INTRODUCTION

We are interested in exploring one aspect of the interface between operations research and computer science – namely, applying an operations research model to a computer science problem. It is important that we don't take too seriously the designation "operations research model". What we mean is simply that the model and corresponding solution method are mathematical approaches which have been associated with the practice of operations research.

The motivation for the present study arises from the situation which exists every time a computer program is executed. A program written in a high-level language like ALGOL or FORTRAN is first translated into machine language. Then, this machine-language version (the object code) of the user's program is executed. How good is this machine-language version? The answer

(*) Received in October 1981.
(1) Department of Industrial and Systems Engineering. The University of Michigan-Dearborn, 4901 Evergreen Rd. Dearborn, Michigan, U.S.A. 48128.

R.A.I.R.O. Recherche opérationnelle/Operations Research, 0399-0559/1983/63/$ 5.00

depends on the compiler or translator which performed the translation. For the identical user program, one compiler may produce object code which runs twice as fast and yet takes up only half the space of the code generated by a second compiler. The conventional term for this is that the first compiler performed more "optimization" to produce the resulting code. Despite the use of the term "optimization", the object code is not the "best" that could be produced. Nevertheless, chiefly for historical reasons, the word "optimization" has stuck. What really happens inside the compiler is the application of transformations which improve (but don't optimize) the code.

Finally, we arrive at the more specific motivation for this study:

– to what extent is the object code really optimal?

– for particular transformations, can we specify what is truly the optimal representation?

– how can the traditional mathematical theory of optimization be applied?

The transformation that is singled out for study is register allocation. From the earliest FORTRAN compilers until the present, this has been among the best methods for making substantial improvements in the running time of the object code. Register allocation involves determining *how many* of a computer's hardware registers should be reserved for use with each program when it is being translated. To be correct, our problem is better described as register assignment – a much more difficult problem – which seeks to specify precisely *what should be the contents* of each register.

The underlying economic consideration is that a program will execute faster if the values it needs are in high-speed registers rather than in memory. If the value of variable X is needed for a computation, more time is expended if the value of X must be fetched from memory than if the value is already conveniently in a register. Most computers for many years have had a number of registers – for example, the IBM 360/370 machines have 16 general-purpose registers. The compiler that can use these registers wisely can produce object code which executes faster. Ideally, we would like to have quantities, which will be needed in the next computation, available in registers, instead of in memory. The source of the difficulty is simply that there are (usually) many more candidates for storage in the registers than there are registers to accommodate them. We have sketched, then, a decision problem: what quantities should occupy registers at each step in the program so that program runs as fast as possible?

Many operations-research approaches to such a decision problem present themselves immediately. One candidate solution technique would be *mathematical programming*, formulating some of the discussion above into an

objective function with constraints. Possibly, *stochastic programming* is indicated, because there seems to be some notion of the probability that a given quantity will be needed at a given time in the execution. A second candidate solution technique would be *network theory*, because the flow of control in the program will certainly be an issue in determining which values are needed based on different paths having been traversed in the program. A third approach would be to consider it to be a *sequential decision process.* The state of the system might be the contents of the registers, with some stochastic process representing the changes in state over time. The fourth technique, the *assignment method* is mentioned because it immediately comes to mind with something called "the register assignment problem". However, in operations research we know that the technique is appropriate only for a restricted case of linear programming. Accordingly, it seems wise not to consider it further as a separate entity. If it does have value then it should arise from the consideration of our first alternative of mathematical programming in general.

Each of the three approaches above were used by the author to try to formulate the decision problem for optimal register assignment. The model that was chosen had strong intuitive appeal as well as possessing aspects of all three methods. *Nonserial dynamic programming* [4] allows for the mathematical programming approach, and also has some visual appeal as a two-dimensional representation (like a graph) and provides naturally for a staged decision process.

The familiar use of dynamic programming is to transform a sequential, multi-stage decision problem into a series of single-stage problems. The domain is the serial or "straight-line" structure, in which each stage looks like (*fig.* 2). Dynamic programming was first used to model nonserial systems in connection with the chemical process industry. A diverging branch at a stage in the model represented the flow of product passing through a separator, splitting the pure product from the unrecovered raw material [3]. In analogous ways, other material flows defined a combining stage (converging branch), bypass (feedforward loop), and countercurrent flow (feedback loop). The dynamic programming models of such nonserial systems have been generalized and adapted to other nonserial structures [4, 8].

Naturally, the nonserial structure influences the solution procedure. In a diverging branch, for example, there is one stage which has two outputs, each of these serving as the input to a separate serial system. The two serial systems are analyzed separately and combined at the diverging stage. With loops the computations become more complex because, for some stages, the optimal returns must be given as functions of two variables instead of one. The details are provided elsewhere [4, 8].

2. FORMULATION OF THE PROBLEM

Questions relating to the efficient use of a computer's registers have been investigated in many different forms. The first study of interest [5] used a graphical model but, significantly, the programs involved only straight-line flow – no branches or loops. The procedure was given some technical improvements by other investigators [6, 7]. In [6], Kennedy also suggested how the straight-line procedure might be extended to handle a simple loop. Agresti presented a method for handling branching (tree-structured) programs [1]. The interest in the present paper is with very general branching and looping structures that are representative of real computer programs.

Registers which can be used for indexing are the special concern here. Indexing is a valuable programming technique for operating on data which are arranged in storage in some systematic way. It is a standard way, for example, to access the entries in a table in succession. For such purposes, indexing is widely used by assembly-language programmers, and it is found extensively in the compiler-generated object code. Two types of instructions are of interest. One type simply *refers* to the contents of an index register. For example:

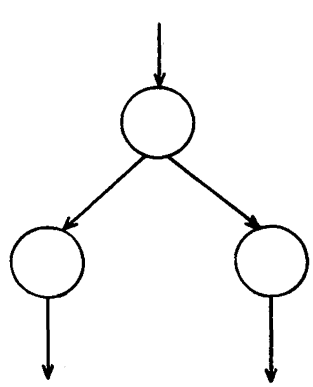
$$\text{ADD } 1, K(2)$$

means that one operand is in register 1 and the second operand is in memory at location K plus the contents of register 2. Here register 2 is used for indexing. A second type of instruction *modifies* the contents of the index register. For example, if we added one to the contents of index register, then the instruction would be of the second type. Where there are more indices than registers, the problem is to devise a plan which specifies which indices should occupy index registers at each step in the program.

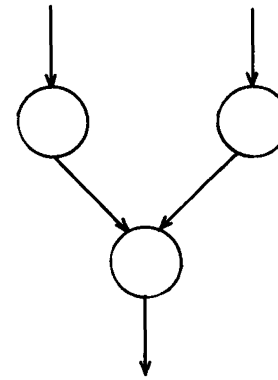
We will assume that for a machine with N index registers, the program calls for M indices ($M > N$). Also there are M core locations reserved for the M indices to keep track of their current values. This means that if the present contents of an index register have not been modified, a new index can be loaded immediately since there is a copy of the present index in memory. If the present contents have been modified, the updated value of the index must be stored in memory before a new index can be loaded. The principal cost that we will attempt to minimize is the number of memory references required. This cost function is consistent with that used by other investigators [1, 5, 6].

The interest here is with programs that exhibit very general flow of control. Accordingly, a control flow graph will be a useful medium to represent the complex flow. In such a directed graph, the nodes are basic blocks; that is,

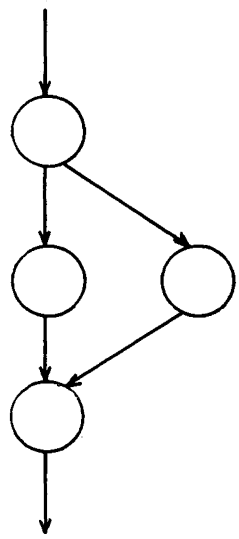
straight-line sequences of instructions in which the only entry to the sequence is at the first instruction and the only exit is at the last instruction. The arcs denote possible transfers of control. Four elementary nonserial structures are represented by the control flow graphs in figure 1.



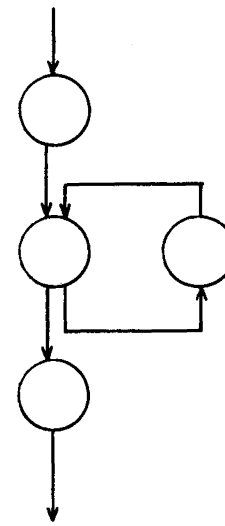
1. The Diverging Branch



2. The Converging Branch



3. The Feedforward Loop



4. The Feedback Loop

Figure 1. — Four Elementary Nonserial Structures

Referring to figure 1 provides a convenient way to characterize the previous results and the present paper. The study by Horwitz, Karp, Miller, and Winograd [5] considered only straight-line code, which is a single basic block with no arcs. Agresti [1] studied tree-structure programs, which are replications

of the diverging-branch structure in figure 1. The present paper permits arbitrarily complex flow, represented by combinations of the four structures. As an illustration of the method, the dynamic-programming procedure will be applied to an example consisting of a feedforward loop (which itself can be viewed as a diverging branch followed by a converging branch). Next, the recursive equations will be given for a second example involving more complex flow and including a feedback loop.

From the control flow graph, the information which is needed for index register assignment is extracted in an *index map*. The structure is identical to the flow graph; but, instead of instructions, the nodes contain only the indices which are called for. For an instruction which modifies the index, an asterisk is placed next to the index. This distinction is necessary because the costs are greater: an additional memory reference is required to re-store a modified index.

Let $X=\{x_1, x_2, \ldots, x_m\}$ be a finite set of indices, and let $S=\langle s_1, s_2 \rangle$ be an ordered set of states of indices. An index in state s_1 is unmodified. When an asterisk has been placed next to an index, that index is in the modified state s_2. An index in state s_2 has had its value changed since it was last loaded from memory.

Associated with each state s_i is a non-negative cost $C(s_i)$, the number of memory references involved in replacing an index which is in state s_i. Therefore, $C(s_1)=1$ and $C(s_2)=2$. The cost – number of memory references – of changing the state of an index from s_i to s_j is given by $c(s_i, s_j)$. For the index register assignment problem, $c(s_1, s_2)=0$; that is, there are no memory references required when an index is modified. Alternatively, $c(s_2, s_1)=1$ because the modified index must be re-stored in memory.

DEFINITION 1: An *index map* I is a triple (B, A, τ) in which:

(i) B is a non-empty set of nodes;

(ii) A, disjoint from B, is a set of directed arcs;

(iii) $\tau: A \to B \times B$ is a mapping such that if $a \in A$ and $\tau(a)=(b, c)$ then, arc a is said to have b as its initial node and c as its terminal node.

Because it corresponds to a basic block in a program flow graph, a node $b \in B$ will be called simply a "block" of the index map. A block is a sequence of symbols from $X \times S$. Within each block the flow of control is linear so that the j-th symbol is called the j-th step in that block Between blocks, however, the flow depends on τ.

DEFINITION 2: A *configuration* on N registers is an N-tuple $(q_1, q_2, \ldots, q_N)$, where $q_i \in X \times S$. Let π be a permutation of the set $\{1, 2, \ldots, N\}$ so that the

configurations $(q_1, q_2, \ldots, q_N)$ and $(q_{\pi 1}, q_{\pi 2}, \ldots, q_{\pi N})$ are identified. If $Q_1=(q_1^1, q_2^1, \ldots, q_N^1)$ and $Q_2=(q_1^2, q_2^2, \ldots, q_N^2)$ are two configurations, then $w^*(Q_1, Q_2)$ is the minimum cost of changing from configuration Q_1 to configuration Q_2:

$$w^*(Q_1, Q_2)=\min_{\pi} \sum_{i=1}^{N} w(q_i^1, q_{\pi i}^2),$$

where:

$$w[(x, s), (x', s')]=\begin{cases} c(s, s') & \text{if } x=x', \\ C(s) & \text{if } x \neq x', \end{cases}$$

and π ranges over all permutations of $\{1, 2, \ldots, N\}$.

The developments thus far will be used to describe the index register assignment problem as a multi-stage decision process. The methods of discrete dynamic programming will be used to obtain solutions. The formulation begins with any low-level language program. An index map is created by the techniques of this section. The program calls for M indices, and the computer has N index registers ($M > N$). There are M memory locations in which to store the M indices. Each step j in the index map is a stage in the dynamic programming model. The state of stage j is given by the configuration:

$$Q_j=(q_1, q_2, \ldots, q_N) \text{ where } q_i \in X \times S.$$

This state vector provides the important data at each stage in the process: namely, what are the current contents of the index registers. The only other necessary information is the current symbol $(x, s)\, x \in X,\ s \in S$ from the index map, because we must insure that the current symbol required at a given step is indeed present in one of the index registers at that step.

At any stage j, a decision is made based on Q_j and (x, s), the j-th symbol in the index map. Six decisions are possible:

1. NO ACTION. This decision may be made if the required symbol is present in the current configuration. That is, the j-th symbol is (x, s) and $(x, s') \in Q_j$ for some $s' \geqq s$.

2. MODIFY. If the j-th symbol is (x_i, s_2) and $(x_i, s_1) \in Q_j$, then the state of x_i is changed from s_1 to s_2 so that $(x_i, s_2) \in Q_j$.

3. LOAD. If the j-th symbol (x_i, s_1) is not in any register, the current value of x_i is loaded into index register k. The contents of index register k, q_k must be in the unmodified state s_1. Because q_k hasn't been changed since it was last loaded into register k, a copy exists in memory so that no "store" operation is necessary.

4. LOAD AND MODIFY. The j-th symbol is (x_i, s_2) and x_i is not present in the current configuration. The value of index x_i is loaded from memory into some index register k whose contents q_k are in state s_1, so that, as in the previous case, a "store" operation is unnecessary. Now the current contents of index register k are $q_k=(x_i, s_1)$. But a modified index, (x_i, s_2), is required so the state is changed and $q_k=(x_i, s_2)$.

5. STORE AND LOAD. The symbol (x_i, s_1) is required but not found in the present configuration. The value of x_i is to be loaded from memory into some index register k whose contents have been modified (state s_2). Because of this, the contents first must be stored in memory before (x_i, s_1) can be loaded into index register k.

6. STORE, LOAD, AND MODIFY. Here the requirement is (x_i, s_2) and index x_i is not present in any index register. The value of x_i is to be loaded into an index register k whose contents have been modified. The necessary action is to store the present contents of k into memory; load the value of x_i into k; and change the state of x_i from s_1 to s_2.

This decision information is expressed as an ordered N-tuple $D=(d_1, d_2, \ldots, d_N)$. Treating as a vector the configuration $Q=(q_1, q_2, \ldots, q_N)$ where $q_i \in X \times S$, state transition is accomplished by vector addition $Q+D$. As an example, consider any two consecutive steps j and $j+1$ in the program flow. Assume that the current configuration at j is:

$$Q_j=(x_3, x_5^*, x_2, x_9).$$

This notation indicates that, in a 4-register problem, the current contents of the index registers are the indices x_3, x_5, x_2, and x_9 respectively. Further, x_5 has been modified since it was last loaded so that an asterisk appears. Suppose that the symbol at $j+1$ in the index map is x_4^*. That index is not present in Q_j; so it must be loaded into an index register. Selecting that index register, the central problem of register assignment, will be solved by algorithms of the next section. Two cases arise, depending on whether the index to be replaced is in the modified or unmodified state:

1. If the algorithm calls for loading x_4^* into either register 1, 3 or 4, then the decision is "LOAD AND MODIFY". The indices x_3, x_2, and x_9 which are currently in these registers are in the unmodified state. Their values have not been changed since they were last loaded from memory into index registers. Consequently, a copy of their current values exists in memory, so the quantity in the index registers can be written-over safely. The current value of x_4 would be loaded from memory into the appropriate index register 1, 3, or 4. Because the actual $j+1$ symbol is x_4^*, this indicates that the corresponding machine

language instruction changes the value of the index. Accordingly, an asterisk is introduced so that the new configuration will exhibit x_4^*, showing that index x_4 has been altered. The decision vector is constructed in such a way that the new configuration Q_{j+1} will be formed from the addition of Q_j and D_{j+1}. Depending on which index – x_3, x_2, or x_9 – is eliminated, D_{j+1} and Q_{j+1} would appear as follows:

(i) if x_3 is eliminated:

$$D_{j+1}=(x_4^*-x_3,\ 0,\ 0,\ 0)$$

$$Q_{j+1}=Q_j+D_{j+1}$$

$$=(x_3,\ x_3^*,\ x_2,\ x_9)+(x_4^*-x_3,\ 0,\ 0,\ 0)$$

$$=(x_4^*,\ x_3^*,\ x_2,\ x_9);$$

(ii) if x_2 is eliminated:

$$D_{j+1}=(0,\ 0,\ x_4^*-x_2,\ 0)$$

$$Q_{j+1}=Q_j+D_{j+1}$$

$$=(x_3,\ x_3^*,\ x_2,\ x_9)+(0,\ 0,\ x_4^*-x_2,\ 0)$$

$$=(x_3,\ x_3^*,\ x_4^*,\ x_9);$$

(iii) if x_9 is eliminated:

$$D_{j+1}=(0,\ 0,\ 0,\ x_4^*-x_9),$$

$$Q_{j+1}=Q_j+D_{j+1}$$

$$=(x_3,\ x_3^*,\ x_2,\ x_9)+(0,\ 0,\ 0,\ x_4^*-x_9)$$

$$=(x_3,\ x_3^*,\ x_2,\ x_4^*).$$

In any case, the cost of this "load and modify" decision is the same – one. Only one memory reference is required: to fetch x_4.

2. If the algorithm calls for loading x_4^* into index register 2, then the correct decision is "STORE, LOAD, AND MODIFY". The index x_5^* which occupies register 2 is in the modified state s_2. The value of index x_5 has been changed since it was last loaded from memory into the register. The copy of x_5 in memory contains the value of x_5 *before* modification. Consequently, the core

copy of x_5 must be updated to the current value in the index register before that value is erased. The first operation is to store the contents of index register 2 into memory. Then load x_4 and modify as in the case above. The decision vector and new configuration look like this:

$$D_{j+1} = (0,\ x_4^* - x_5^*,\ 0,\ 0)$$

$$Q_{j+1} = Q_j + D_{j+1}$$

$$= (x_3,\ x_5^*,\ x_2,\ x_9) + (0,\ x_4^* - x_5^*,\ 0,\ 0)$$

$$= (x_3,\ x_4^*,\ x_2,\ x_9).$$

The cost of this decision is two: one memory reference to re-store x_5^* and one to fetch x_4.

Still needed in the representation of the index register allocation problem as a multi-stage decision process is a measure of the utility of a configuration — a return function. The cost function $w^*(Q_1, Q_2)$, which was defined earlier, will be used. Consider step $j+1$. Let Q_j be the configuration before step $j+1$ and Q_{j+1} be the configuration which results from Q_j and decision D_{j+1}. The return r_{j+1} is defined by $w^*(Q_j, Q_{j+1})$ where $Q_{j+1} = Q_j + D_{j+1}$.

The basic elements of the dynamic-programming model — state variables, decision variables, stage transformations, and return function — have been specified. If j and $j+1$ are two consecutive steps in the index map, stage $j+1$ in the decision process can be represented as in figure 2.

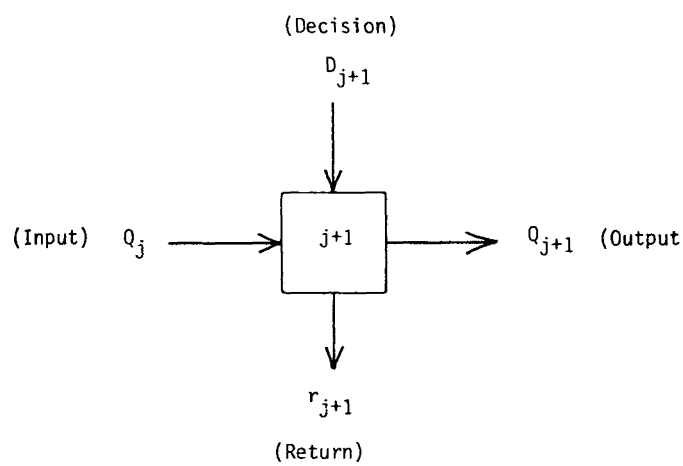


Figure 2. — Typical stage $j+1$ in decision process

3. STATE REDUCTION

Although dynamic programming could be applied now, there is some question about its practicality. There are $2^N M^{N-1}$ configurations possible at each step; which means that the same number of states must be evaluated at each stage in the recursion analysis. Fortunately, this number can be significantly reduced by applying Rule 1 of [5], which proves that only minimal change states need to be considered at each stage:

Rule 1: Only minimal change states must be analyzed by dynamic programming. Let Q_{j-1} be a state associated with step $j-1$, and let Q_j be a state associated with step j. Minimal change states are generated from the following algorithm: call Q_0 the initial state of the index registers. Q_j is a minimal change state if:

(i) Q_j is identical with Q_{j-1}. If $Q_{j-1}=(q_1^{j-1}, q_2^{j-1}, \ldots, q_N^{j-1})$ and $Q_j=(q_1^j, q_2^j, \ldots, q_N^j)$, we say Q_j is identical with Q_{j-1} if there exists some permutation π of the set $\{1, 2, \ldots, N\}$ such that:

$$q_i^j = q_{\pi i}^{j-1}, \qquad i=1, 2, \ldots, N;$$

(ii) Q_j differs from Q_{j-1} by exactly one element, the index required at step j. Call the j-th element (x, s), $x \in X$, $s \in S$. Now $(x, s) \notin Q_{j-1}$ and $\exists k$, $1 \leqq k \leqq N$, such that for some permutation π of $\{1, 2, \ldots, N\}$:

$$q_i^j = q_{\pi i}^{j-1}, \qquad i=1, \ldots, k-1, \quad k+1, \ldots, N,$$

$$q_i^j \neq q_{\pi i}^{j-1}, \qquad i=k,$$

implies that $q_k^j=(x, s)$.

(iii) Q_j differs from Q_{j-1} only in that a modified index in Q_j is unmodified in Q_{j-1}. The j-th element (x, s_2), $x \in X$; and $\exists k$, $1 \leqq k \leqq N$, such that for some permutation π of $\{1, 2, \ldots, N\}$:

$$q_i^j = q_{\pi i}^{j-1}, \qquad i=1, \ldots, k-1, \quad k+1, \ldots, N,$$

$$q_{\pi k}^{j-1} = (x, s_1),$$

$$q_k^j = (x, s_2).$$

The example of the next section will demonstrate that Rule 1 provides a major computational improvement.

4. THE FEEDFORWARD LOOP

The index map in figure 3 contains a feedforward loop. This familiar control structure can be viewed as a combination of a diverging branch followed by a converging branch. There is a program statement (like IF-THEN-ELSE) in which control can be transferred along either of two paths. These two paths have a common termination point, a statement further ahead in the flow. In [3], this structure was studied as the bypass of chemical engineering processes.

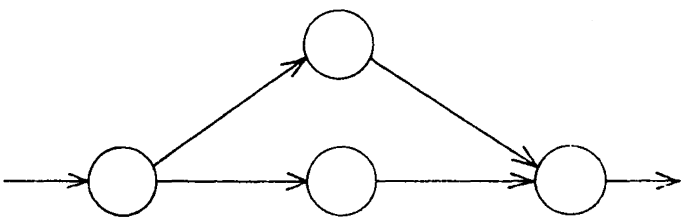


Figure 3. – Index Map of Feedforward Loop

The feedforward loop as a decision process is presented in figure 4. The dynamic programming is more difficult with such nonserial structures. The difficulty is manifested in the increased dimensionality of the state description. Consider the lower path in figure 4. The total return at any stage in that serial system must include the effects of both the input Q_{k_1} and the output Q_{k_m} from the upper path, which we will call the bypass system. The key to the technique is optimizing the serial system k_1 through k_m separately, but describing its total return as a function of two state variables, Q_{k_1} and Q_{k_m}. Carrying the additional state variable increases the computational load. This expression for the total bypass return can be combined with the serial system (stages 1 through n) at either stage j or stage $j+p+1$. Because of state inversion, combining at either stage involves roughly the same number of calculations.

We arbitrarily decide to absorb the k_1 through k_m return at stage j instead of $j+p+1$. Then the solution procedure will take this form:

1. Backward recursion on stages n, $n-1, \ldots, j+p+1$ to obtain $f(Q_{j+p+1})$.

2. Backward recursion on stages $k_1, \ldots, k_m$ to obtain $f(Q_{k_1}, Q_{k_m})$, the optimal return as a function of the bypass input and output.

3. Backward recursion on stages $j+1$, $j+2$, $\ldots$, $j+p$ to find $f(Q_{j+1}, Q_{k_m})$, the optimal *return still based on the output from the bypass.*

4. Find optimal return $f(Q_j)$ by optimizing, over Q_{k_m}, the returns at k_1 and $j+1$, $f(Q_{k_1}, Q_{k_m})$, and $f(Q_{j+1}, Q_{k_m})$.

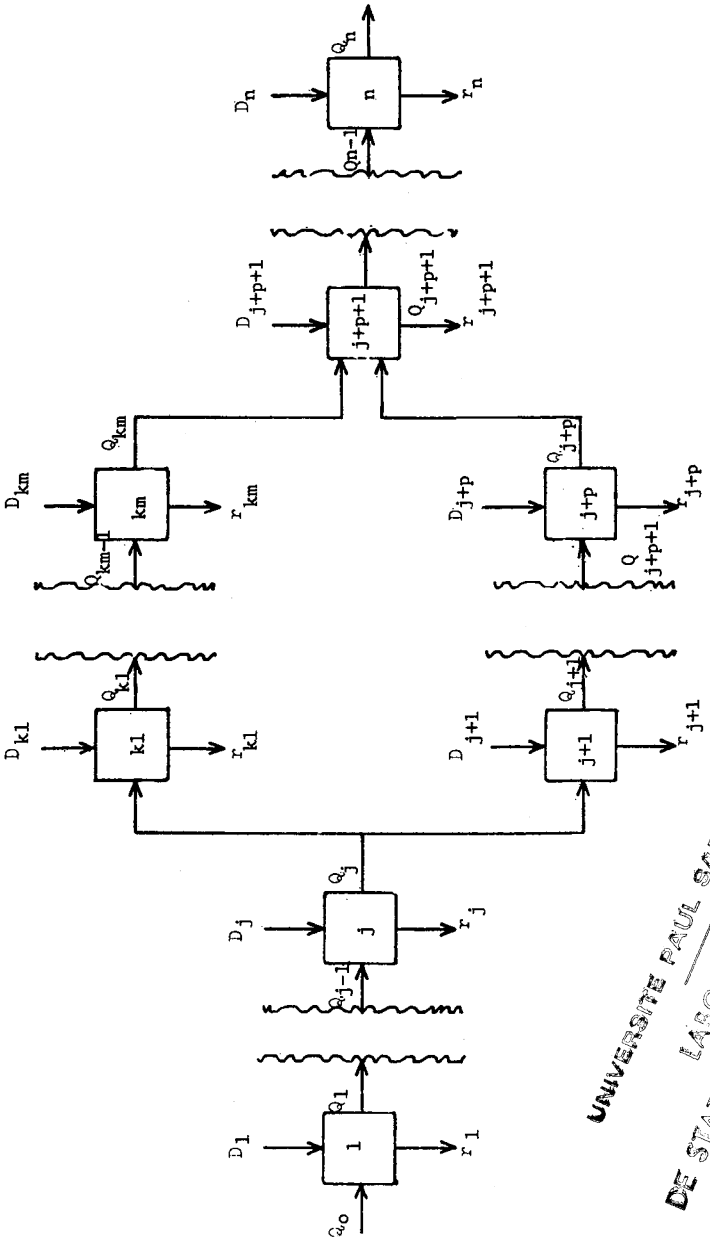


Figure 4. — The Feedforward Loop as a Decision Process

5. Backward recursion on stages $j-1, j-2, \ldots, 1$ to find $f(Q_0)$.

We are solving the initial state optimization problem, so our solution will be $f(Q_0)$, the minimal cost as a function of an initial state.

The return at $j+p+1$ is the typical return at a converging stage:

$$r_{j+p+1}(Q_{k_m}, Q_{j+p}, D_{j+p+1}) = w^*(Q_{k_m}, Q_{j+p+1}) + w^*(Q_{j+p}, Q_{j+p+1}).$$

In the same way, the divergence in stages k_1 and $j+1$ has the return:

$$r_{k_1\, j+1}(Q_j, D_{k_1}, D_{j+1}) = w^*(Q_j, Q_{k_1}) + w^*(Q_j, Q_{j+1})$$

The recursive equations for the feedforward loop are:

1. For stages $j+p+2, \ldots, n$:

$$f(Q_{n-1}) = \min_{D_n} r_n(Q_{n-1}, D_n),$$

subject to:
$Q_n = Q_{n-1} + D_n$:

$$f(Q_{i-1}) = \min_{D_i} \{r_i(Q_{i-1}, D_i) + f(Q_i)\},$$

subject to:

$$Q_i = Q_{i-1} + D_i, \qquad i = j+p+2, \ldots, n-1.$$

2. For stage $j+p+1$:

$$f(Q_{k_m}, Q_{j+p}) = \min_{D_{j+p+1}} \{r_{j+p+1}(Q_{k_m}, Q_{j+p}, D_{j+p+1}) + f(Q_{j+p+1})\}.$$

3. For stages $j+2, j+3, \ldots, j+p$:

$$f(Q_{k_m}, Q_{j+p-1}) = \min_{D_{j+p}} \{r_{j+p}(Q_{j+p-1}, D_{j+p}) + f(Q_{k_m}, Q_{j+p})\},$$

subject to:

$$Q_{j+p} = Q_{j+p-1} + D_{j+p},$$

$$f(Q_{k_m}, Q_{i-1}) = \min_{D_i} \{r_i(Q_{i-1}, D_i) + f(Q_{k_m}, Q_i)\},$$

subject to:

$$Q_i = Q_{i-1} + D_i, \qquad i = j+2, j+3, \ldots, j+p-1.$$

4. For the bypass, stages $k_2, k_3, \ldots, k_m$:

$$f(Q_{k_m-1}, Q_{k_m}) = \min_{D_{k_m}} r_{k_m}(Q_{k_m-1}, D_{k_m}),$$

subject to:

$$Q_{k_m} = Q_{k_m - 1} + D_{k_m}$$

$$f(Q_{i-1}, Q_{k_m}) = \min_{D_i} \{r_i(Q_{i-1}, D_i) + f(Q_i, Q_{k_m})\},$$

subject to:

$$Q_i = Q_{i-1} + D_i, \qquad i = k_2, k_3, \ldots, k_m - 1.$$

5. For stages $j+1$ and k_1:

$$f(Q_j) = \min_{Q_{k_m}, D_{k_1}, D_{j+1}} \{r_{k_1, j+1}(Q_j, D_{k_1}, D_{j+1}) + f(Q_{k_1}, Q_{k_m}) + f(Q_{j+1}, Q_{k_m})\},$$

subject to:

$$Q_{k_1} = Q_j + D_{k_1} \qquad \text{and} \qquad Q_{j+1} = Q_j + D_{j+1}$$

6. For the remaining serial stages 1, 2, ..., j:

$$f(Q_{j-1}) = \min_{D_j} \{r_j(Q_{j-1}, D_j) + f(Q_j)\},$$

subject to:

$$Q_j = Q_{j-1} + D_j$$

$$f(Q_{i-1}) = \min_{D_i} \{r_i(Q_{i-1}, D_i) + f(Q_i)\},$$

subject to:

$$Q_i = Q_{i-1} + D_i, \qquad i - 1, 2, \ldots, j-1.$$

The recursion equations make explicit what was discussed earlier concerning the solution technique. The additional state variable Q_{k_m} is maintained in computations for the optimal return at the following stages: $j+p+1$; $j+2$, $j+3$, ..., $j+p$; and k_2, k_3, ..., k_m.

When the branches are combined in the recursion at stages k_1 and $j+1$, the optimal return is determined over all values of Q_{k_m}. After this analysis, only the customary single state variable is needed during stages 1 through j until the process terminates.

There is an alternative to minimizing over values of Q_{k_m}. The advantage would be that Q_{k_m} need not be retained as a state variable in the calculations of optimal returns at the stages listed above. The technique is to find Q_{k_m} initially and use a sequential search procedure (like Fibonacci search) to identify new values when $f(j)$ is found [8], p. 197. As the recursive equations indicate, this method is not being applied to the feedforward loop.

5. AN EXAMPLE

Figure 5 depicts a sample index map containing a feedforward loop. The index map was extracted from a program which used five indices on a computer with two index registers. To solve the example, Rule 1 will identify the minimal change states at each step. Then the recursion equations will be applied to find the initial state optimal return, $f(Q_0)$.

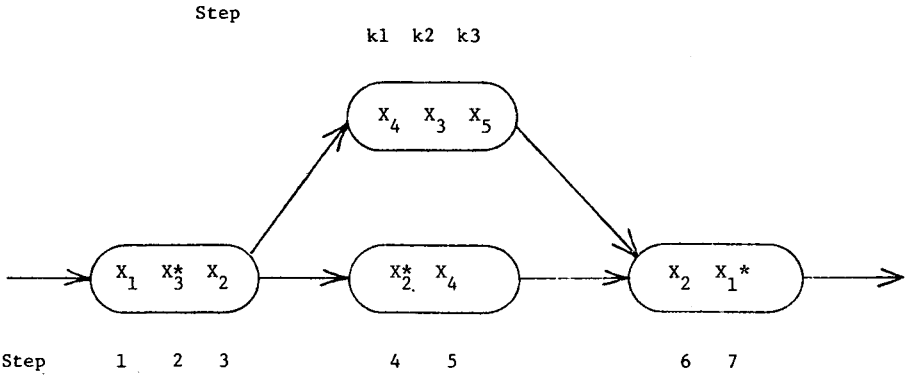


Figure 5. — Example of Index Map with Feedforward Loop

Directed by the solution procedure, we begin with stage 7. Six minimal change states are associated with the final stage:

$$Q_7^1 = x_1^* x_2,$$

$$Q_7^2 = x_1^* x_5,$$

$$Q_7^3 = x_1^* x_3,$$

$$Q_7^4 = x_1^* x_4,$$

$$Q_7^5 = x_1^* x_3^*,$$

$$Q_7^6 = x_1^* x_2^*.$$

The recursive analysis at this final stage involves only one state variable. The results are summarzied in table I in the usual fashion, listing the quantities $f(Q_6)$, $D_7(Q_6)$, and $Q_7(Q_6)$:

We can proceed no further until the optimal return for the bypass is determined as a function of the bypass input and output. We need values for $f(Q_{k_1}, Q_{k_3})$.

Our first requirement is:

$$f(Q_{k_2}, Q_{k_3}) = \min_{D_{k_3}} r_{k_3}(Q_{k_2}, Q_{k_3})$$

and the values are presented in table II.

Now it may be more clear just what effect the additional state variable has on the computation. Instead of a list of values for $f(Q_{k_2})$, we must maintain a table of results for $f(Q_{k_2}, Q_{k_3})$. This is one illustration of an earlier observation that an extra state variable increases the calculations exponentially.

TABLE I

Stage 7 recursion analysis

	$f_7(Q_6)$	$D_7(Q_6)$	$Q_7(Q_6)$
$Q_6^1 = X_1 X_2$	0	1	1
$Q_6^2 = X_1 X_5$	0	1	2
$Q_6^3 = X_2 X_3$	1	2	3
$Q_6^4 = X_2 X_4$	1	2	4
$Q_6^5 = X_2 X_3^*$	1	2	5
$Q_6^6 = X_2^* X_4$	1	2	6

TABLE II

Optimal stage k_3 return $f(Q_{k^2}, Q_{k^3})$

	$Q_{k_3}^1 = X_1 X_5$	$Q_{k_3}^2 = X_2 X_5$	$Q_{k_3}^3 = X_3 X_5$	$Q_{k_3}^4 = X_4 X_5$	$Q_{k_3}^5 = X_3^* X_5$
$Q_{k_2}^1 = X_1 X_3$	1	2	1	2	1
$Q_{k_2}^2 = X_2 X_3$	2	1	1	2	1
$Q_{k_2}^3 = X_3 X_4$	2	2	1	1	1
$Q_{k_2}^4 = X_3^* X_4$	3	3	2	2	1

Continuing with the bypass stages, we proceed backward to stage k_2, seeking values (displayed in table III) for:

$$f(Q_{k_1}, Q_{k_3}) = \min_{D_{k_2}} \{r_{k_2}(Q_{k_1}, D_{k_2}) + f_{k_3}(Q_{k_2}, Q_{k_3})\},$$

subject to:

$$Q_{k_3} = Q_{k_2} + D_{k_3}.$$

Explaining one entry from table III, consider:

$$f(Q^1_{k_1}, Q^1_{k_3}) = 2.$$

The following discussion will explain how this value was determined. The recursion analysis at k_2 involved input configurations Q_{k_1} and output configurations Q_{k_2}:

$$Q^1_{k_1} = x_1 x_4, \qquad Q^1_{k_2} = x_1 x_3,$$

$$Q^2_{k_1} = x_2 x_4, \qquad Q^2_{k_2} = x_2 x_3,$$

$$Q^3_{k_1} = x^*_3 x_4, \qquad Q^3_{k_2} = x_3 x_4,$$

$$Q^4_{k_2} = x^*_3 x_4.$$

TABLE III

Optimal stage k_2 return $f(Q_{k_1}, Q_{k_3})$

	$Q^1_{k_3} = X_1 X_5$	$Q^2_{k_3} = X_2 X_5$	$Q^3_{k_3} = X_3 X_5$	$Q^4_{k_3} = X_4 X_5$	$Q^5_{k_3} = X^*_3 X_5$
$Q^1_{k_1} = X_1 X_4$	2	3	2	2	2
$Q^2_{k_1} = X_2 X_4$	3	2	2	2	2
$Q^3_{k_1} = X^*_3 X_4$	3	3	2	2	2

Four decisions are used to change $Q^1_{k_1} = x_1 x_4$ into the output states of k_2:

$D^1_{k_2}$: LOAD x_3 (in place of x_4);

$D^2_{k_2}$: LOAD x_2; LOAD x_3;

$D^3_{k_2}$: LOAD x_3 (in place of x_1);

$D^4_{k_2}$: LOAD x_3 (in place of x_1); MODIFY x_3;

The four output states are produced by these transformations:

$$Q^1_{k_2} = Q^1_{k_1} + D^1_{k_2}$$

$$= (x_1, x_4) + (0, x_3 - x_4)$$

$$= (x_1, x_3),$$

$$Q^2_{k_2} = Q^1_{k_1} + D^2_{k_2}$$

$$= (x_1, x_4) + (x_2 - x_1, x_3 - x_4)$$

$$= (x_2, x_3);$$

$$Q^3_{k_2} = Q^1_{k_1} + D^3_{k_2}$$

$$= (x_1,\ x_4) + (x_3 - x_1,\ 0)$$

$$= (x_3,\ x_4)$$

$$Q^4_{k_2} = Q^1_{k_1} + D^4_{k_2}$$

$$= (x_1,\ x_4) + (x^*_3 - x_1,\ 0)$$

$$= (x^*_3,\ x_4)$$

The first component of $f(Q^1_{k_1},\ Q^1_{k_3})$ is the single-stage return $r_{k_2}(Q^1_{k_1},\ D_{k_2})$:

$$r_{k_2}(Q^1_{k_1},\ D^1_{k_1}) = w^*(Q^1_{k_1},\ Q^1_{k_2}) = 1,$$

$$r_{k_2}(Q^1_{k_1},\ D^2_{k_1}) = w^*(Q^1_{k_1},\ Q^2_{k_2}) = 2,$$

$$r_{k_2}(Q^1_{k_1},\ D^3_{k_1}) = w^*(Q^1_{k_1},\ Q^3_{k_2}) = 1,$$

$$r_{k_2}(Q^1_{k_1},\ D^4_{k_1}) = w^*(Q^1_{k_1},\ Q^4_{k_2}) = 1.$$

The second component of $f(Q^1_{k_1},\ Q^1_{k_3})$ is the optimal return at the last stage. These values were presented earlier in the results of the stage k_3 analysis:

$$f(Q^1_{k_2},\ Q^1_{k_3}) = 1,$$

$$f(Q^2_{k_2},\ Q^1_{k_3}) = 2,$$

$$f(Q^3_{k_2},\ Q^1_{k_3}) = 2,$$

$$f(Q^4_{k_2},\ Q^1_{k_3}) = 3.$$

Combining these two return components yields "2" as the cost in $f(Q^1_{k_1},\ Q^1_{k_3})$:

$$f(Q^1_{k_1},\ Q^1_{k_3}) = \min\{[r_{k_2}(Q^1_{k_1},\ D^1_{k_2}) + f(Q^1_{k_2},\ Q^1_{k_3})],$$

$$[r_{k_2}(Q^1_{k_1},\ D^2_{k_2}) + f(Q^2_{k_2},\ Q^1_{k_3})],$$

$$[r_{k_2}(Q^1_{k_1},\ D^3_{k_2}) + f(Q^3_{k_2},\ Q^1_{k_3})],$$

$$[r_{k_2}(Q^1_{k_1},\ D^4_{k_2}) + f(Q^4_{k_2},\ Q^1_{k_3})]\}$$

subject to:

$$Q^1_{k_2} = Q^1_{k_1} + D^1_{k_2},$$

$$Q^2_{k_2} = Q^1_{k_1} + D^2_{k_2},$$

$$Q^3_{k_2} = Q^1_{k_1} + D^3_{k_2},$$

$$Q^4_{k_2} = Q^1_{k_1} + D^4_{k_2},$$

$$f(Q^1_{k_1}, Q^1_{k_3}) = \min\{1+1,\ 2+2,\ 1+2,\ 1+3\},$$

$$= 2.$$

Continuing to follow the solution process outlined earlier, we are now able to analyze stage (*b*) and then, stages 4 and 5. The method at stage 6 is similar to that which was used in the converging stage in [3]. The only difference is that the optimal return is a function of two states, Q_{k_3} and Q_5. Because we have worked through the calculations necessary with two state variables and because the technique at a converging stage is known, we will summarize the results for stage 6. Table IV gives the values for $f(Q_5, Q_{k_3})$ with the output

TABLE IV

Optimal stage 6 return $f(Q_5, Q_{k_3})$ and corresponding output state Q_6 (a)

	$Q^1_{k_3} = X_1 X_5$	$Q^2_{k_3} = X_2 X_5$	$Q^3_{k_3} = X_3 X_5$	$Q^4_{k_3} = X_4 X_5$	$Q^5_{k_3} = X^*_3 X_5$
$Q^1_5 = X_1 X_4$	$2(Q^1_6)$	–	–	$3(Q^4_6)$	–
$Q^2_5 = X^*_2 X_4$	$4(Q^4_6)$	$3(Q^2_6)$	$4(Q^3_6)$	$2(Q^6_6)$	–
$Q^3_5 = X^*_3 X_4$	–	–	–	–	$3(Q^5_6)$

(a) where $Q^1_6 = X_1 X_2$; $Q^2_6 = X_2 X_5$; $Q^3_6 = X_2 X_3$; $Q^4_6 = X_2 X_4$; $Q^5_6 = X_2 X^*_3$; $Q^6_6 = X^*_2 X_4$.

state Q_6 in parentheses. At the next stage, results for $f(Q_4, Q_{k_3})$ are obtained. The values are given in table V in the same format.

We have reached that point in the solution process where the returns for the two branches are combined. Using the recursive equations, we minimize over values of the extra state variable Q_{k_3} thus eliminating it from any future calculations. The optimal return can be expressed now as a function of one variable Q_3, which takes on two values:

$$Q^1_3 = x_1 x_2 \qquad \text{or} \qquad Q^2_3 = x_2 x^*_3.$$

TABLE V

Optimal stage 5 return $f(Q_4, Q_{k_3})$ and corresponding output state Q_5 ([a])

	$Q^1_{k_3}=X_1X_5$	$Q^2_{k_3}=X_2X_5$	$Q^3_{k_3}=X_3X_5$	$Q^4_{k_3}=X_4X_5$	$Q^5_{k_3}=X^*_3X_5$
$Q^1_4=X_1X^*_2$	$4(Q^1_5)$	$4(Q^2_5)$	$5(Q^2_5)$	$3(Q^2_5)$	–
$Q^2_4=X^*_2X^*_3$	$6(Q^2_5)$	$5(Q^2_5)$	$6(Q^2_5)$	$4(Q^2_5)$	$5(Q^3_5)$

([a]) where $Q^1_5=X_1X_4$; $Q^2_5=X^*_2X_4$; $Q^3_5=X^*_3X_4$.

Table VI gives the results for the optimal return $f(Q_3)$, where:

$$f(Q_3)=\min_{Q_{k_3},\,D_{k_1},\,D_4}\{r_{k_1;4}(Q_3,\,D_{k_1},\,D_4)+f(Q_{k_1},\,Q_{k_3})+f(Q_5,\,Q_{k_3})\}$$

subject to:

$$Q_{k_1}=Q_3+D_{k_1} \qquad \text{and} \qquad Q_4=Q_3+D_4.$$

The pairs of states in parentheses beneath each return entry in table VII indicate the resultant states, Q_{k_1} and Q_4, such that:

$$Q_{k_1}=Q_3+D_{k_1} \qquad \text{and} \qquad Q_4=Q_3+D_4.$$

TABLE VI

Recursion analysis at stages 4 and k_1, $f(Q_3)$ and corresponding output states Q_4 ([a]) and Q_{k^1} ([b])

	$Q^1_{k_3}=X_1X_5$	$Q^2_{k_3}=X_2X_5$	$Q^3_{k_3}=X_3X_5$	$Q^4_{k_3}=X_4X_5$	$Q^5_{k_3}=X^*_3X_5$	$f(Q_3)$
$Q^1_3=X_1X_2$	7	7	8	6	10	6
	$(Q^1_{k_1};\,Q^1_4)$	$(Q^1_{k_1};\,Q^1_4)$	$(Q^1_{k_1};\,Q^1_4)$	$(Q^1_{k_1};\,Q^1_4)$	$(Q^1_{k_1};\,Q^1_4)$	
			$(Q^2_{k_1};\,Q^1_4)$	$(Q^2_{k_1};\,Q^1_4)$	$(Q^2_{k_1};\,Q^2_4)$	
$Q^2_3=X_2X^*_3$	10	9	9	7	8	7
–	$(Q^1_{k_1};\,Q^1_4)$	$(Q^2_{k_1};\,Q^2_4)$	$(Q^3_{k_1};\,Q^2_4)$	$(Q^3_{k_1};\,Q^2_4)$	$(Q^3_{k_1};\,Q^2_4)$	–
–	$(Q^1_{k_1};\,Q^2_4)$	$(Q^3_k;\,Q^2_4)$	–	–	–	–
–	$(Q^3_{k_1};\,Q^1_4)$	–	–	–	–	–
–	$(Q^3_{k_1};\,Q^2_4)$	–	–	–	–	–

([a]) where $Q^1_4=X_1X^*_2$; $Q^2_4=X^*_2X^*_3$.
([b]) where $Q^1_{k1}=X_1X_4$; $Q^2_{k1}=X_2X_4$; $Q^3_{k1}=X^*_3X_4$.

The listing of more than one pair of states acknowledges the existence of alternative optima. For reference, the output states are repeated below.

$$Q^1_{k_1} = x_1 x_4, \qquad Q^1_4 = x_1 x^*_2,$$

$$Q^2_{k1} = x_2 x_4, \qquad Q^2_4 = x^*_2 x^*_3,$$

$$Q^3_{k_1} = x^*_3 x_4.$$

The branching portion of the index map has been completed. All that remains is the serial dynamic programming stages 1, 2 and 3. Results of that analysis are presented in table VII.

TABLE VII

Recursion analysis for stages 3, 2, *and* 1

Stage 3	$f(Q_2)$	$D_3(Q_2)$	$Q_3(Q_2)$
$Q^1_2 = X_1 X^*_3$	8	1	1
Stage 2	$f(Q_1)$	$D_2(Q_1)$	$Q_2(Q_1)$
$Q^1_1 = X_2$	9	1	1
Stage 1	$f(Q_0)$	$D_1(Q_0)$	$Q_1(Q_0)$
$Q^1_0 = yz$	10	1	1

The initial state optimization problem has been solved with the calculation of:

$$f(Q^1_0 = yz) = 10,$$

in which the configuration "*yz*" indicates two arbitrary initial quantities which are unrelated to the problem. Only ten memory references are needed for the indices in a program corresponding to the index map in figure 5.

One of the advantages of dynamic programming is the simple recognition of alternative optima. In the present example, re-tracing through the recursive analysis identifies four optimal solutions, each requiring only ten memory references. The four solutions are displayed in figure 6 by listing the register configuration which must exist at each program step. Counting the total number of memory references in each solution will verify the figure of 10 provided by the recursion analysis.

To place the solution in perspective, we are saying that if the compiler makes use of results in figure 6, the resulting machine language program will be truly optimal with respect to references to memory for quantities that are used for indexing. In other words, the compiler should include, in the object code, instructions which load and store indices as implied by figure 6. In this

way, the program will execute the fastest that it possibly can (regarding our restricted problem domain) by having the "right" quantities in registers during each portion of the program.

Program Steps:

$$0 \quad 1 \quad 2 \quad 3 \quad \begin{matrix} k_1 & k_2 & k_3 \\ 4 & 5 \end{matrix} \quad 6 \quad 7$$

Optimal Configurations:

Solution # 1:

$$yz \quad x_1 z \quad x_1 x_3^* \quad x_1 x_2 \quad \begin{matrix} x_1 x_4 & x_3 x_4 & x_4 x_5 \\ x_1 x_2^* & x_2^* x_4 \end{matrix} \quad x_2^* x_4 \quad x_1^* x_2^*$$

Solution # 2:

$$yz \quad x_1 z \quad x_1 x_3^* \quad x_1 x_2 \quad \begin{matrix} x_2 x_4 & x_3 x_4 & x_4 x_5 \\ x_1 x_2^* & x_2^* x_4 \end{matrix} \quad x_2^* x_4 \quad x_1^* x_2^*$$

Solution # 3:

$$yz \quad x_1 z \quad x_1 x_3^* \quad x_2 x_3^* \quad \begin{matrix} x_3^* x_4 & x_3 x_4 & x_4 x_5 \\ x_2^* x_3^* & x_2^* x_4 \end{matrix} \quad x_2^* x_4 \quad x_1^* x_2^*$$

Solution # 4:

$$yz \quad x_1 z \quad x_1 x_3^* \quad x_2 x_3^* \quad \begin{matrix} x_3^* x_4 & x_3^* x_4 & x_4 x_5 \\ x_2^* x_3^* & x_2^* x_4 \end{matrix} \quad x_2^* x_4 \quad x_1^* x_2^*$$

Figure 6. — The Four Optimal Solutions to Feedforward Loop Example

6. APPLICATION TO AN ACTUAL PROGRAM

The nonserial dynamic programming method is now applied to an actual program. Keep in mind the way in which the method will be implemented. We would expect the method to be incorporated in an optimizing compiler. The procedure determines when load and store instructions would be generated by the compiler to accomplish the optimal packing of index registers.

As an example of a familiar program which exhibits the feedforward flow, consider (*fig.* 7). This FORTRAN program constitutes the inner loop of a

```
      :
      IF (A (I) . GT . B(J)) GO TO 20
      C(K) = A (I)
      I = I + 1
      D(L) = C(K)
      GO TO 30
20    C(K) = B(J)
      J = J + 1
30    K = K + 1
      L = L + 1
      RETURN
      END
```

Figure 7. — Example FORTRAN Program Segment

mergesort algorithm. The same logic is also found in file update procedures. The only difference here is producing an extra copy of one of the files in array D, to make the indexing more interesting. The index map, corresponding to this program, is presented in figure 8.

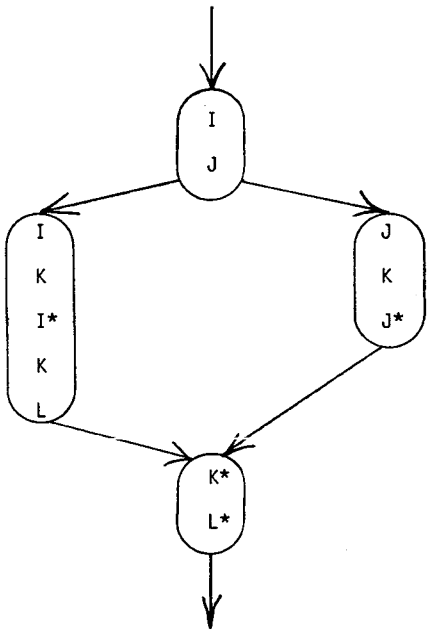


Figure 8. – Index Map Associated with FORTRAN Program

The program was run on the Amdahl 470 V/8 computer at The University of Michigan which uses the Michigan Terminal System (MTS) as its operating system. A listing of the machine language produced by the IBM FORTRAN G compiler was obtained. Presenting a literal copy of this object code would introduce unnecessary detail like operation codes and machine addresses. So, the machine language was re-written in a more accessible form as a simple assembly language-level program.

The organization of the Amdahl computer is similar to the IBM 370, having 16 general-purpose registers. The object program produced by the FORTRAN G compiler used the same register repeatedly for arithmetic operations, so that register has been represented as "ACC" for accumulator. Likewise, the object program used two of the registers for indexing, so we denote them by IR_1 and IR_2 for index register one and two, respectively.

Now the re-written object program in figure 9 should be understandable. The instructions are given in a generic one-address format where, for example:

$$\text{LOAD } IR_1, I$$

means that index register one is loaded with the contents of symbolic address I. Here we make the obvious associations that address I contains the value of variable I; address ONE contains the value one; and so on.

	Instruction		*Comment*
	LOAD	IR_1, I	
	LOAD	ACC, A(IR_1)	ACC CONTAINS A_i
	LOAD	IR_2, J	
	COMPARE	ACC, B(IR_2)	COMPARE A_i, B_j
	BRANCH	>, L_1	if $A_i > B_j$, GO TO L_1
	LOAD	ACC, A(IR_1)	ACC CONTAINS A_i
	LOAD	IR_2, K	
	STORE	ACC, C(IR_2)	STORE A_i INTO C_k
	LOAD	ACC, I	ACC CONTAINS I
	ADD	ACC, ONE	$I \leftarrow I+1$
	STORE	ACC, I	
	LOAD	ACC, C(IR_2)	ACC CONTAINS C_k
	LOAD	IR_1, L	
	STORE	ACC, D(IR_1)	STORE C_k INTO D_l
	BRANCH	ALWAYS, L_2	
L_1	LOAD	IR_1, J	
	LOAD	ACC, B(IR_1)	ACC CONTAINS B_j
	LOAD	IR_2, K	
	STORE	ACC, C(IR_2)	STORE B_j INTO C_k
	LOAD	ACC, J	
	ADD	ACC, ONE	$J \leftarrow J+1$
	STORE	ACC, J	
L_2	LOAD	ACC, K	
	ADD	ACC, ONE	$K \leftarrow K+1$
	STORE	ACC, K	
	LOAD	ACC, L	
	ADD	ACC, ONE	$L \leftarrow L+1$
	STORE	ACC, L	

Figure 9. — Rewritten Object Program Produced by FORTRAN Compiler (28 instructions)

In our past discussion, we distinguished between two types of instructions. In this program, the instruction:

$$\text{ADD } IR_1, \text{ ONE}$$

is an example of one which *changes* an index, while

$$\text{STORE ACC, C}(IR_2)$$

is an example of one which *refers* to an index. This latter instruction means that we are storing the contents of the accumulator into memory at symbolic address C plus the contents of index register two (that is, indexed by *K*, which is the current value of IR_2).

The object program in figure 9 consists of 28 instructions. On the Amdahl 470 V/8 computer, all of the instructions we use here are the same length (4 bytes) and the differences in execution times are not great—for example, there are no multiply or divide instructions which take significantly longer to execute. Accordingly, the size measure (28 instructions) can be used, in a relative sense, as a rough measure of the execution time as well. Size and speed are the two primary resources that an optimizing compiler seeks to conserve.

The index map (*fig.* 8), which corresponds to the program, is solved using the dynamic programming methods of section 4 for a feedforward loop. The problem involves four indices (*I*, *J*, *K*, *L*) and two registers (IR_1, IR_2). The solution, given in figure 10, specifies the optimal assignment of indices to registers.

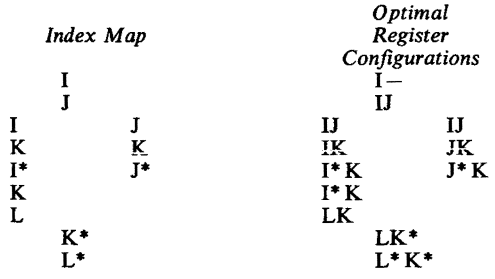


Figure 10. — Solution to FORTRAN Program Example

If the dynamic programming methods were incorporated as part of an optimizing compiler, then the object program in figure 11 would result. Only 22 instructions are needed. Based on our assumptions about cost above, the improvement in both size and speed is 21%. The significance of this improvement grows when we recall that the example represents the *inner loop* of a mergesort or file-update program which easily might be traversed thousands of times because of the sizes of the files.

In fairness, other considerations should be mentioned. The time to compile the program will grow in order to perform the dynamic programming analysis. Also, it is not always straightforward to assess properly the effect of a single potential improvement like index register assignment. Many times in an optimizing compiler, a particular transformation is effective only because other transformations have created opportunities for it to be applied.

The dynamic programming procedures would be able to make good use of some of the typical information which optimizing compilers collect about the program. Use-definition chaining keeps track of the occurrences of a definition or a use of each quantity. Such information is regularly gathered by an

optimizing compiler [2], p. 429. From this information, the dynamic programming procedure could identify when it was able to leave modified indices in registers (as was done in the last three lines of figure 11) or when it must restore the current value into memory for later use.

	Instruction		*Comment*
	LOAD	IR_1, I	
	LOAD	ACC, A(IR_1)	ACC CONTAINS A_i
	LOAD	IR_2, J	
	COMPARE	ACC, B(IR_2)	COMPARE A_i, B_j
	BRANCH	>, L_1	IF $A_i > B_j$, GO TO L_1
	LOAD	IR_2, K	
	STORE	ACC, C(IR_2)	STORE A_i INTO C_k
	ADD	IR_1, ONE	$I \leftarrow I+1$
	STORE	IR_1, I	
	LOAD	ACC, C(IR_2)	ACC CONTAINS C_k
	LOAD	IR_1, L	
	STORE	ACC, D(IR_1)	STORE C_k INTO D_l
	BRANCH	ALWAYS, L_2	
L_1	LOAD	IR_1, IR_2	$IR_1 \leftarrow J$
	LOAD	ACC, B(IR_1)	ACC CONTAINS B_j
	LOAD	IR_2, K	
	STORE	ACC, C(IR_2)	STORE B_j INTO C_K
	ADD	IR_1, ONE	$J \leftarrow J+1$
	STORE	IR_1, J	
L_2	ADD	IR_2, ONE	$K \leftarrow K+1$
	LOAD	IR_1, L	
	ADD	IR_1, ONE	$L \leftarrow L+1$

Figure 11. — Object Program Using Dynamic Programming Procedures (22 instructions)

7. ARBITRARILY COMPLEX FLOW

The incidence of both feedback loops and branches in the same index map must be regarded as the most realistic situation in terms of actual computer programs. Those programs for which efficient register assignment is important would most likely possess a flow that is more intricate than the feedforward loop examined thus far. The reason for studying an elementary system, however, is to make use of the results in the solution of complex index maps. The complex problems, which are admittedly more common, will be solved by effectively decomposing them into the familiar elementary cases.

There is an important economic trade-off between recursive optimization and optimizing over all variables simultaneously. When the straight-line portions of the system are relatively long, the recursive approach is usually preferable. The technique may be augmented by sequential search procedures to help eliminate extra state variables. When the flow is characterized by relatively short serial segments with very elaborate nonserial structure, optimizing simultaneously over all variables may be more efficient at certain

times during the analysis. The reason is that the recursive optimization would involve increased state variable dimensionality. The essential point is that arbitrarily complex index maps can be solved within the present methodology of nonserial dynamic programming.

Central to the solution process is the identification of elementary structures in the complex index maps. Where the elementary structures are disjoint, occurring sequentially in the program flow, recognition is easiest. Analyzing such a case would involve merely the successive application of the recursion equations for elementary systems. More difficult is the nondisjunctive case in which the basic graphs are nested, interlocked, and overlapped in the flow. Such flow of control is not at all uncommon in computer programs; and, therefore, it should be examined. Accordingly, a sample index map with nontrivial flow is offered in figure 12. The purpose here is to illustrate the

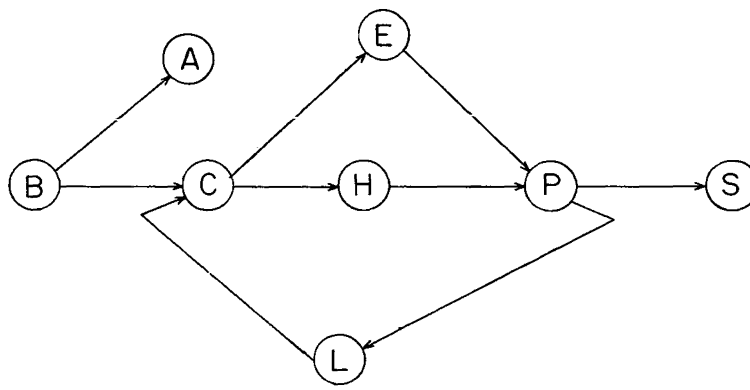


Figure 12. – Complex Example. As an Index Map

approach of decomposing such a map into its elementary structures. In this way, the results for the basic loops and branches may be applied—with appropriate modifications—to provide optimal index register assignment.

In terms of its elementary structures, the index map of figure 12 contains a diverging branch and a feedforward loop embedded in a feedback loop. Fig. 13 presents this same system as a staged decision process. Because the flow is more complex, our notation has been augmented slightly from that which was used with the feedforward loop. First, we label each block in figure 12 with a letter. Second, in figure 13, we denote the stages by a pair of values—one giving the block name and the other giving the stage within the block. Further, we use the upper case letter to indicate the number of stages in the block. For example, stages $(e, 1)$, $(e, 2)$, ..., (e, E) comprise block E. As before, each stage in the decision process represents a step in the index

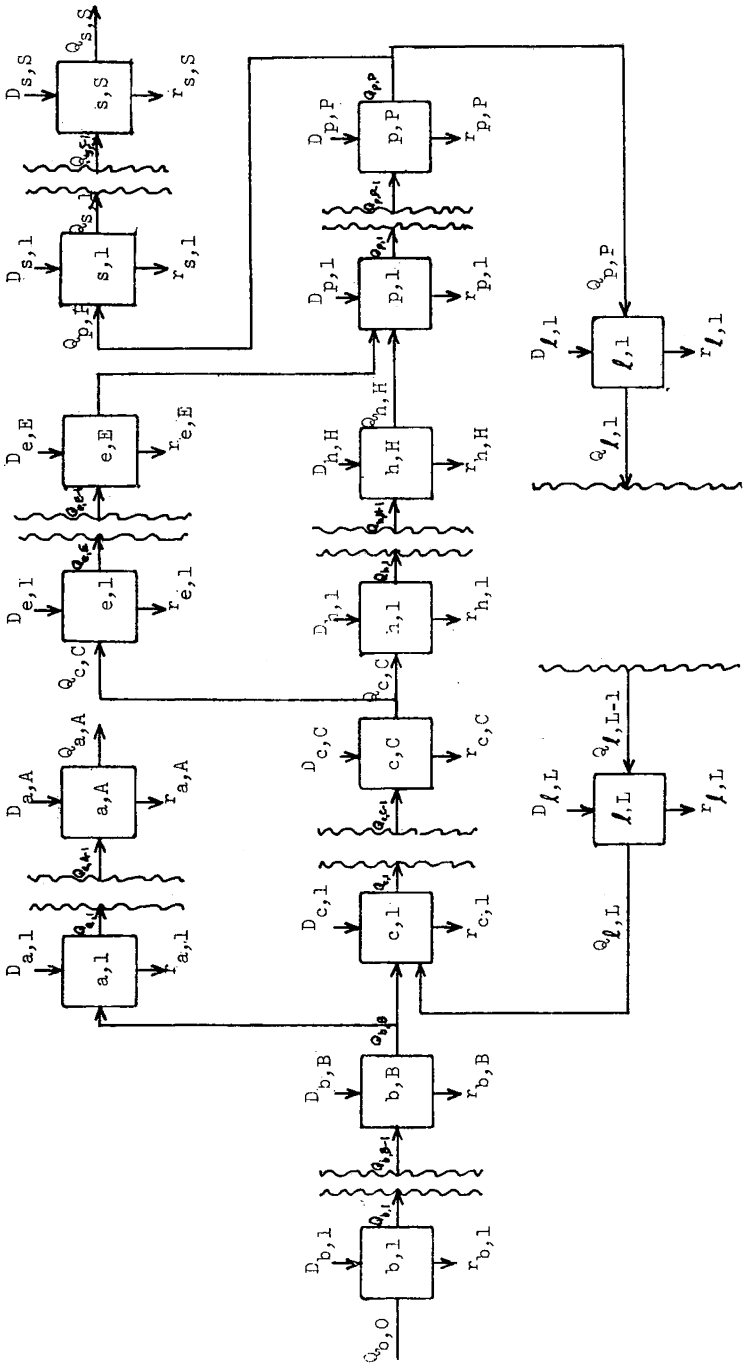


Figure 13. – Complex Example. As a Decision Process

map. The designation of decisions, returns, and states in figure 13 is consistent with our earlier usage except for the stages now being represented by a pair of values.

For the complex system of figure 13, no actual indices or configurations will be specified, but the recursive equations will be given in full. Following these equations for any M-index, N-register problem would lead to the optimal index register assignment decisions that should be followed.

Possible program flow paths are understandably more numerous in such a system. Represented by the sequence of blocks encountered in each path, some flows of control from figure 12 are:

[1] BA
[2] BCEPS
[3] BCHPS
[4] BCEPLCEPS
[5] BCHPLCHPLCEPS

We will solve the initial state optimization problem, obtaining values for $f(Q_{b,0})$, for some arbitary initial configuration $Q_{b,0}$. The general solution procedure is composed of the following sequence of events:

1. Backward recursion on stages of block S, yielding $f(Q_{s,1})$.
2. Backward recursion on stages of block L, yielding $f(Q_{l,1}, Q_{l,L})$.
3. Recursion analysis on stages $(l, 1)$ and $(s, 1)$, to obtain $f(Q_{p,P}, Q_{l,L})$.
4. Backward recursion on stages of block P, resulting in $f(Q_{p,1}, Q_{l,L})$.
5. Backward recursion on stages of block E, yielding $f(Q_{e,1}, Q_{e,E})$.
6. Recursion analysis at the converging stage $(p, 1)$ to obtain $f(Q_{h,H}, Q_{e,E}, Q_{l,L})$.
7. Backward recursion on stages of block H to find $f(Q_{h,1}, Q_{e,E}, Q_{l,L})$.
8. Recursion analysis on diverging stages $(e, 1)$ and $(h, 1)$, producing $f(Q_{c,C}, Q_{l,L})$.
9. Backward recursion on stages of block C, finding $f(Q_{c,1}, Q_{l,L})$.
10. Backward recursion for the diverging branch, block A, obtaining $f(Q_{a,1})$.
11. Recursion analysis on stages $(a, 1)$ and $(c, 1)$, yielding $f(Q_{b,B})$.
12. Finally, backward recursion on stages of block B, to find the solution to the initial state optimization problem, $f(Q_{b,0})$.

The recursion equations which are needed at each step in the solution process are given below. They will solve the index register assignment problem for the system of figure 13.

1. For stages $(s, 2), (s, 3), \ldots, (s, S)$ of block S:

$$f(Q_{s,S-1}) = \min_{D_{s,S}} r_{s,S}(Q_{s,S-1}, D_{s,S}),$$

$$f(Q_{s,j-1}) = \min_{D_{s,j}} \{r_{s,j}(Q_{s,j-1}, D_{s,j}) + f(Q_{s,j})\},$$

subject to:

$$Q_{s,j} = Q_{s,j-1} + D_{s,j}, \qquad j = 2, 3, \ldots, S-1.$$

2. For stages $(l, 2), (l, 3), \ldots, (l, L)$ of block L:

$$f(Q_{l,L-1}, Q_{l,L}) = \min_{D_{l,L}} r_{l,L}(Q_{l,L-1}, D_{l,L}),$$

$$f(Q_{l,j-1}, Q_{l,L}) = \min_{D_{l,j}} \{r_{l,j}(Q_{l,j-1}, D_{l,j}) + f(Q_{l,j}, Q_{l,L})\},$$

subject to:

$$Q_{l,j} = Q_{l,j-1} + D_{l,j}, \qquad j = 2, 3, \ldots, L-1.$$

3. For stages $(l, 1)$ and $(s, 1)$:

$$f(Q_{p,P}, Q_{l,L}) = \min_{D_{s,1}, D_{l,1}} \{r_{s,1;l,1}(Q_{p,P}, D_{s,1}, D_{l,1}) + f(Q_{l,1}, Q_{l,L}) + f(Q_{s,1})\},$$

subject to:

$$Q_{s,1} = Q_{p,P} + D_{s,1} \qquad \text{and} \qquad Q_{l,1} = Q_{p,P} + D_{l,1}.$$

4. For stages $(p, 2), (p, 3), \ldots, (p, P)$ of block P:

$$f(Q_{p,P-1}, Q_{l,L}) = \min_{D_{p,P}} \{r_{p,P}(Q_{p,P-1}, D_{p,p}) + f(Q_{p,p}, Q_{l,L})\},$$

subject to:

$$Q_{p,P} = Q_{p,P-1} + D_{p,P},$$

$$f(Q_{p,j-1}, Q_{l,L}) = \min_{D_{p,j}} \{r_{p,j}(Q_{p,j-1}, D_{p,j}) + f(Q_{p,j})\},$$

subject to:

$$Q_{p,j} = Q_{p,j-1} + D_{p,j}, \qquad j = 2, 3, \ldots, p-1.$$

5. For stages $(e, 2), (e, 3), \ldots, (e, E)$ of block E:

$$f(Q_{e,E-1}, Q_{e,E}) = \min_{D_{e,E}} r_{e,E}(Q_{e,E-1}, D_{e,E}),$$

subject to:

$$Q_{e,E} = Q_{e,E-1} + D_{e,E},$$

$$f(Q_{e,j-1}, Q_{e,E}) = \min_{D_{e,j}} \{r_{e,j}(Q_{e,j-1}, D_{e,j}) + f(Q_{e,j})\},$$

subject to:

$$Q_{e,j} = Q_{e,j-1} + D_{e,j}, \qquad j = 2, 3, \ldots, E-1.$$

6. For stage $(p, 1)$:

$$f(Q_{h,H}, Q_{e,E}, Q_{l,L}) = \min_{D_{p,1}} \{r_{p,1}(Q_{h,H}, Q_{e,E}, D_{p,1}) + f(Q_{p,1}, Q_{l,L})\}$$

7. For stages $(h, 2), (h, 3), \ldots, (h, H)$ of block H:

$$f(Q_{h,H-1}, Q_{e,E}, Q_{l,L}) = \min_{D_{h,H}} \{r_{h,H}(Q_{h,H-1}, D_{h,H}) + f(Q_{h,H}, Q_{e,E}, Q_{l,L})\},$$

subject to:

$$Q_{h,H} = Q_{h,H-1} + D_{h,H},$$

$$f(Q_{h,j-1}, Q_{e,E}, Q_{l\,L}) = \min_{D_{h,j}} \{r_{h,j}(Q_{h,j-1}, D_{h,j}) + f(Q_{h,j}, Q_{e,E}, Q_{l,L})\},$$

subject to:

$$Q_{h,j} = Q_{h,j-1} + D_{h,j}, \qquad j = 2, 3, \ldots, H-1.$$

8. For stages $(e, 1)$ and $(h, 1)$:

$$f(Q_{c,C}, Q_{l,L}) = \min_{Q_{e,E}, D_{e,1}, D_{h,1}} \{r_{e,1;h,1}(Q_{c,C}, D_{e,1}, D_{h,1}) + f(Q_{h,1}, Q_{e,E}, Q_{l,L}) + f(Q_{e,1}, Q_{e,E})\}$$

subject to:

$$Q_{e,1} = Q_{c,C} + D_{e,1} \qquad \text{and} \qquad Q_{h,1} = Q_{c,C} + D_{h,1}.$$

9. For stages $(c, 2), (c, 3), \ldots, (c, C)$ of block C:

$$f(Q_{c,C-1}, Q_{l,L}) = \min_{D_{c,C}} \{r_{c,C}(Q_{c,C-1}, D_{c,C}) + f(Q_{c,C}, Q_{l,L})\},$$

subject to:

$$Q_{c,C} = Q_{c,C-1} + D_{c,C},$$

$$f(Q_{c,j-1}, Q_{l,L}) = \min_{D_{c,j}} \{r_{c,j}(Q_{c,j-1}, D_{c,j}) + f(Q_{c,j}, Q_{l,L})\},$$

subject to:

$$Q_{c,j} = Q_{c,j-1} + D_{c,j}, \qquad j = 2, 3, \ldots, C-1.$$

10. For stages $(a, 2)$, $(a, 3)$, ..., (a, A) of block A:

$$f(Q_{a,A-1}) = \min_{D_{a,A}} r_{a,A}(Q_{a,A-1}, D_{a,A}),$$

$$f(Q_{a,j-1}) = \min_{D_{a,j}} \{r_{a,j}(Q_{a,j-1}, D_{a,j}) + f(Q_{a,j})\},$$

subject to:

$$Q_{a,j} = Q_{a,j-1} + D_{a,j}, \qquad j = 2, 3, \ldots, A-1.$$

11. For stages $(a, 1)$ and $(c, 1)$:

$$f(Q_{b,B}) = \min_{Q_{l,L}, D_{a,1}, D_{c,1}} \{r_{a,1;c,1}(Q_{b,B}, Q_{l,L}, D_{a,1}, D_{c,1}) + f(Q_{a,1}) + f(Q_{c,1}, Q_{l,})\},$$

subject to:

$$Q_{a,1} = Q_{b,B} + D_{a,1} \qquad \text{and} \qquad Q_{c,1} = Q_{b,B} + D_{c,1}.$$

12. For stages of block B:

$$f(Q_{b,B-1}) = \min_{D_{b,B}} \{r_{b,B}(Q_{b,B-1}, D_{b,B}) + f(Q_{b,B})\},$$

subject to:

$$Q_{b,B} = Q_{b,B-1} + D_{b,B},$$

$$f(Q_{b,j-1}) = \min_{D_{b,j}} \{r_{b,j}(Q_{b,j-1}, D_{b,j}) + f(Q_{b,j})\},$$

subject to:

$$Q_{b,j} = Q_{b,j-1} + D_{b,j}, \qquad j = 1, 2, \ldots, B-1.$$

The precise recursive equations will differ with each complex program graph. Careful examination of the relationships above will reveal that familiar equations from the feedforward loop examples are present. The key element

in defining the recursive optimization of a complex system is appreciating the dimensionality of the state space. Two state variables are required at many stages above, just as in the bypass of the feedforward loop. New in the present structure is the maintenance of three state variables in the recursion analysis at stages $(p, 1)$ and $(h, 2)$, $(h, 3)$, ..., (h, H). The output state from blocks E and L must be kept variable throughout these steps. Of course, as in the last example, search techniques could be introduced to reduce the dimensionality.

Use of the equations above would find the minimum number of memory references required for index register assignment in programs with index maps like (*fig.* 12). The corresponding register configurations which must exist and decisions which must be made at each program step also would be identified.

The solution of additional examples would follow the same approach. The index map would be examined for occurrences of basic structures. Results for serial and elementary nonserial systems would be carefully applied, being especially aware of the particular state variables which must be present at each stage.

8. CONCLUSIONS

Nonserial dynamic programming has been used as a model to represent a decision problem arising from the translation of programs by compilers. The result is a method which can be automated as part of an optimizing compiler to improve the execution speed of compiled programs.

ACKNOWLEDGEMENT

The contributions of an unnamed referee are sincerely appreciated. The careful work of this person resulted in several corrections and suggestions which have improved the paper.

REFERENCES

1. W. W. AGRESTI, *Register Assignment in Tree-Structured Programs*, Information Sciences, Vol. 18, No. 1, 1979, pp. 83-94.
2. A. V. AHO and J. D. ULLMAN, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.
3. R. ARIS, G. L. NEMHAUSER and D. J. WILDE, *Optimization of Multistage Cyclic and Branching Systems by Serial Procedures*, A. I. Ch. E. Journal, Vol. 10, 1964, pp. 913-919.
4. U. BERTELE and F. BRIOSCHI, *Nonserial Dynamic Programming*, Academic Press, New York, 1972.

5. L. P. HORWITZ, R. M. KARP, R. E. MILLER and S. WINOGRAD, *Index Register Allocation*, Journal of the A.C.M., Vol. 13, No. 1, 1966, pp. 43-61.
6. K. KENNEDY, *Index Register Allocation in Straight Line Code and Simple Loops*, in *Design and Optimization of Compilers*, R. RUSTIN, Ed., pp. 51-63.
7. F. LUCCIO, *A Comment on Index Register Allocation*, Communications of the A.C.M., Vol. 10, pp. 572-574.
8. G. L. NEMHAUSER, *Introduction to Dynamic Programming*, John Wiley and Sons, New York, 1966.