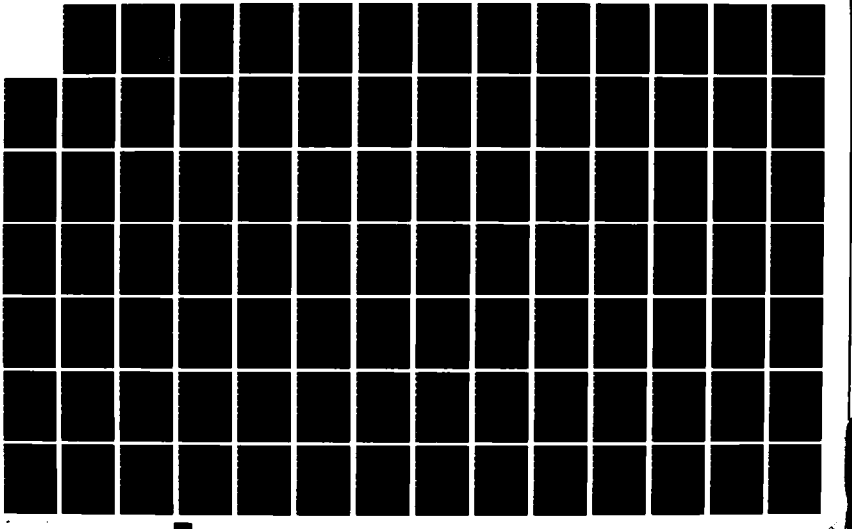
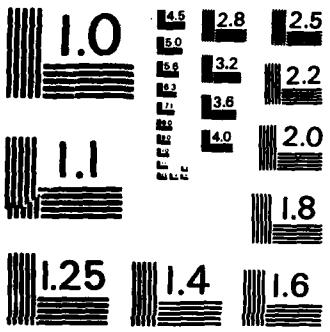
AD-A166 763 SOLVING MULTI-STATE VARIABLE DYNAMIC PROGRAMMING MODELS 1/
USING VECTOR PROCESSING(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH S W STOPKEY DEC 85
UNCLASSIFIED AFIT/CI/NR-86-45T F/G 9/2 NL





SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 86-45T	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Solving Multi-State Variable Dynamic Programming Models Using Vector Processing	5. TYPE OF REPORT & PERIOD COVERED: THESIS/~~DISSERTATION~~
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Stuart Waldemar Stopkey	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: The University of Texas at Austin	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR, WPAFB OH 45433-6583	12. REPORT DATE: 1985
	13. NUMBER OF PAGES: 102
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASS
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 18 Apr 86
Dean for Research and
Professional Development
AFIT/NR, WPAFB OH 45433-6583

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

SOLVING MULTI-STATE VARIABLE DYNAMIC PROGRAMMING MODELS USING VECTOR PROCESSING

by

STUART WALDEMAR STOPKEY, B.S.

MASTER'S REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

DECEMBER, 1985

ABSTRACT

One of the classical allocation models is known as the pallet loading problem. It consists of optimally loading parcels with various weight and volume parameters on a pallet. This pallet has weight and volume limits which constrain the amount of parcels that can be loaded. An optimal load is that combination of parcels which maximizes the summed utilities of all parcels that are loaded.

The pallet loading problem can be generalized to a 2-state variable (weight and volume) Dynamic Programming model. However, the problem can be solved by other methods. In the past, these other methods were preferred due to the difficulty of Dynamic Programming in solving problems of more than one state variable. With the advent of new computer technology in the form of vector processing, that difficulty can be overcome. The efficient column manipulating nature of vector processing is a perfect match for the column nature of the Dynamic Programming problem formulation.

Based upon the algorithm developed using the ideas of vector processing, Dynamic Programming can be considered a very efficient solution technique to the 2-state variable problem. Using the vector processing capability of the ETA Systems supplied CDC Cyber 205, 2-state variable problems too large to be solved previously using DP can now be solved with small computational times. Also, an analysis of the results show that computation time increases linearly as the problem parameters are increased, not exponentially as is the case with other solution methods.

Table of Contents

Page

Chapter

List of Tables

List of Figures

CHAPTER 1

The problem of finding the optimal combination of parcels with varying characteristics to be loaded optimally in a limited cargo area is one that confronts many organizations. At the Air Freight Transport Office at Kelly AFB, San Antonio, this problem is of particular interest. On a given day, an average of 260,000 lbs of material are air-freighted to various destinations. The standard cargo platform is the L436 pallet which can carry up to 10,000 lbs, and has dimensions of 9 ft. x 9 ft. x 9 ft.

Cargo arrives at the terminal in all shapes and sizes, however most of the material consists of boxes with various heighths, widths, lengths, and weights. The number of boxes that are shipped out is on the average over 100 per day based on previous flight manifests. Most flights out carry four loaded pallets, however some aircraft such as the Lockheed C5A can carry up to 27 pallets.

Each box has associated with it a priority ranging from 1 to 3 in integers with "1" being the highest and "3" being the lowest priority.

Based on this priority, combinations of parcels are assembled for shipment on the pallets to the same location. A priority class is defined as the group of all parcels that have the same priority number. A parcel class is defined as all parcels that have the same heighth, width, length, and weight characteristics.

The Transport Office management has determined that the present loading technique is inefficient and that as little as a 5% increase in loading efficiency could lead to savings of nearly one million dollars at this facility alone.

The present method for determining the loads is strictly manual. Parcels set for delivery build up at the loading terminal until such time as the warehouse manager determines that there is enough to justify a load. The priority loading policy is that no lower priority parcel may be loaded until all possible higher priority parcels have been eliminated as loadable.

This means that any possible priority one parcel is shipped before any lower priority parcels are considered. Within a priority class, workers attempt to load parcels on the basis of which ones have been waiting in the parcel loading queue the longest. For example, if a parcel

was rejected from an earlier load, it moves to the head of the parcel queue and becomes the first parcel from the particular priority class to be loaded next time.

For loading purposes, only parcels from a particular priority class are considered at the same time. Only once the given priority class is depleted of loading candidates are the lower priority class parcels considered. The procedure can be described as folllows: all possible priority 1 parcels are loaded, and if there is any weight or volume of the pallet unused, given this available weight and volume, all possible priority 2 parcels are loaded, if there is any weight or volume unused after this pass, then given this available weight and volume all possible priority three parcels are loaded. At each priority level, the problem is to find the optimal combination from a group of parcels to load given available weight and volume. When viewed in this light, at each priority level a solution to what is called the "general pallet loading problem" is being found.

The general pallet loading problem has roughly the same characteristics as the Kelly AFB individual priority group loading problem. However, there are several differences. In the Kelly AFB problem, all

parcels are considered to have different weight and volume characteristics than any other parcel. Based on the definition of a parcel class, the maximum number of boxes in a particular parcel class is one. In the general loading problem, there is no set limit on the number of boxes in any given parcel class. Also, in the general loading problem, the utility of a given parcel can be any value. In the Kelly AFB problem, box utilities are equal in a given priority class until a box has been passed over for a load. At this point, that boxes' utility or "value" is increased to ensure its selection for the next load. Finally, and as stated earlier, the actual Kelly AFB problem would consist of three separate general loading problems, one for each priority class.

At this time, there are several ways to solve the general loading problem, many of which will be discussed later in the report. However, none of the present techniques of solving the problem take advantage of the newest in computer processing technology. To keep up with the increasing need for scientific computing power, a computer processing system based on vector operations as opposed to scalar operations has been developed. Vector operations consist of manipulating whole vectors

of data instead of singular scalar data to produce other vectors. The newly developed computer architecture allows for these vector operations to be done much more quickly and efficiently than a corresponding amount of scalar data.

It is the purpose of this report to develop a new algorithm for the efficient solution of the general pallet loading problem utilizing the new computer technology known as vector processing. Since this technology is relatively new, only a few computers have had the vector processing architecture incorporated into their design. The supercomputer that will be utilized in this research effort is the CDC Cyber 205,one of which is located at ETA Systems, St. Paul, Minn., and another that is located at the University of Minnesota. After the algorithm and computer program have been developed, a sample problem will be formulated and CPU time sensitivities will be performed as part of the research effort. The Kelly AFB problem will be solved as a specialized example case for the computer code testing.

FORMAL PROBLEM STATEMENT

The formal problem statement for the generalized pallet loading problem can be described as follows:

Given a number of diferent parcel classes each with different weight, volume, and utility characteristics, a loading platform with limits on the amount of weight and volume available, and a stipulation that there may be more than one parcel to load from a given parcel class, maximize the utility of the total pallet load by choice of the number of items from each parcel class given their repective utilities. The choice is subject to constraints on the overall maximum weight and volume of the loading platform and on the number of parcels from each parcel class to load.

The formal problem statement for the Kelly AFB cargo loading problem is as follows:

Given up to 100 parcels for each cargo load, each with different weight and volume characteristics, and the L436 pallet with its maximum limits of 10,000 lbs. and 729 cubic ft., maximize the utility of the pallet load by choice of the parcels to be loaded on the pallet given the

respective utilities. This choice is subject to constraints on the overall maximum weight and volume of the pallet and is subject to the restriction that the choice of higher priority parcels must be exhausted before any lower priority parcels can be loaded.

CHAPTER 2

POSSIBLE METHODS OF SOLUTION

Several solution techniques have been proposed and implemented for the solving of the general pallet loading problem. It is appropriate at this time to discuss these techniques.

TRIAL and ERROR -

This is a heuristic approach to the problem that will normally always result in a feasible solution given there is one present. The pallet loading method used at Kelly AFB Freight Terminal is essentially a trial and error algorithm implemented by the parcel loaders. They try different combinations of parcels given the set of loading rules until they achieve a feasible load. This technique very seldom results in the optimal solution. If the loading procedures are correctly followed, the loaders attempt to load the remaining boxes into a decreasing amount of space and weight. Therefore, if there is any remaining space and weight, it is conceivable

that an attempt to load every parcel in the loading queue down to the last priority 3 parcel must be made in this trial and error fashion. To give the reader an appreciation for the diversity of parcels in the loading queue that the parcel loaders must deal with, the following partial list of parcels and associated load parameters is provided:

PARCEL #	WEIGHT (lbs)	VOLUME(cu. ft.)
1	16	2
2	4	6
3	27	3
4	2	1
5	47	4
6	36	4
7	38	7
8	7	1
9	14	2
10	1	1
11	45	3
12	95	42
13	239	57
14	686	75
15	1125	86
16	1135	85
17	16	4
18	12	4
19	13	3
20	36	4
21	16	2
22	2200	63
23	276	85
24	9	1
25	1000	40
26	330	16
27	166	33
28	370	51
29	18	4
30	33	6

TABLE #1 - SAMPLING OF LOADING QUEUE

At this point it may be asked of the reader, how would you handle the problem?

COMPLETE ENUMERATION -

The complete enumeration technique entails comparing the results of all possible combinations of parcels that can be loaded onto the pallet. This solution technique will always find an optimal feasible solution given there is a feasible solution to the problem. This technique can work well for problems in which there are not many possible combinations of parcels. However, as the number of different parcel types increases the number of possible combinations can increase in a dramatic and disqualifying rate.

INTEGER PROGRAMMING -

Integer programming(IP) is a special case of linear programming in which the decision variables are restricted to integer values. In the pallet loading problem where the idea of loading half a box is not feasible, integer programming would be appropriate. The most commonly

implemented integer programming techniques is the "branch and bound" method whereby successive linear programs of the model are run, with constraints added at each step to ultimately force the result into integer form. This method should always find the optimal feasible solution to the problem given there is one. Integer programming is very sensitive to the number of decision variables of the model, additional ones adding immensely to the computational difficulty of the problem. Most integer programming algorithms result in decision variable answers that are in the form of a binary nature, i.e. the decision variable takes on a value of either 0 or 1. For the Kelly AFB problem which has but one parcel to load from each parcel class, IP would be sufficient for small problems. However, for the general pallet loading problem that can have more than one parcel in a parcel class, basic integer programming is an inefficient and in many cases infeasible method of solution(12).

DYNAMIC PROGRAMMING -

Dynamic programming is the proposed method for solving the general pallet loading problem. Dynamic programming consists of breaking

down a large scale problem into sequential parts or "stages" that can be solved separately. Associated with each stage is a utility return generated by a utility function that can be linear or non-linear.

Associated with dynamic programming models are "state variables". State variables can be thought of in an allocation problem as the amount of a given resource left to allocate to the rest of the stages or parcel classes. For example, in the general pallet loading problem , the two state variables are the amount of weight and volume left to allocate to the parcel classes that haven't yet been evaluated. Associated with each parcel class or "stage", are "state variable parameters". State variable parameters are the amounts of state variable resources used up when selecting an item from the given stage. For our pallet loading example, a parcel class of boxes with weight and volume of 2 and 3 respectively would have values of state variable 1 parameter and state variable 2 parameter being 2 and 3 respectively.

The stages of the problem are related to each other by what is known as a "transition function". The transition function defines the relationship between the values of the state variables before they enter a

given stage to their values after they leave the given stage. In most allocation problems, this relationship is linear and is based on the idea that if any items from a stage are selected, then a certain amount of state variable resource will be used up based on the number loaded. For example, if the values of state variables 1 and 2 are 12 and 13 respectively, and two parcels of weight=3 and volume=2 are then loaded, the new state variable 1 and 2 values after that stage are 12-6=6 and 13-4=9 respectively. The dynamic programming breakdown of a problem is shown graphically in the following figure, with each stage representing a different parcel class:

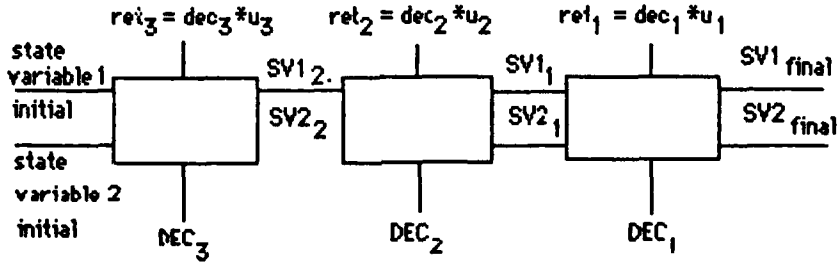


FIGURE #1 - DYNAMIC PROGRAMMING STAGES

In terms of integer programming, the state variable parameters represent constraint coefficients, and the state variables would be the amount of right hand side still available for allocation. Dynamic programming seems to be an ideal theoretical solution for the general pallet loading problem. The stages are set as being each separate class of parcels, the returns are the linear utility functions, and the state variables are the amounts of weight and volume still available after each stage has been processed(12).

Discrete state level dynamic programming methods employ matrices to store information used in solving the problem. The fundamentals of this data storage will be discussed in greater detail in chapter 3, however, the problem of matrix dimensionality that arises from this storage will be touched on now. For a one state variable problem, at every stage, all feasible state variable levels must be evaluated. In DP, this involves crossing all the stages with all the state variable levels creating a two dimensional matrix. For a problem of two state variables, for every stage, every possible combination of feasible state variables must be evaluated. This implies that a three dimensional matrix is needed

to hold all state variable 1 values crossed with state variable 2 values crossed with every stage.

These matrices can be viewed as a collection of vectors and when viewed in this light, lend themselves to possible vector processing solution. The classical dynamic programming algorithm must be viewed as to how it can be manipulated into a vector processing format.

There is one major drawback to discrete state level dynamic programming and that is the size and dimensionality of the matrices created. As discussed previously, for the general pallet loading problem of 2 state variables, a three dimensional array is needed to hold all necessary solution information. For a problem of 40 different parcel classes, 200 possible different weight levels, and 300 possible different volume levels, a matrix with 2.4 million data elements would be required. If another 100 possible weight levels are added, the matrix size jumps to 3.6 million data elements. On standard computers, this problem would be practically unsolvable. However, with new computers such as the Cyber 205, a matrix of 2.4 million words uses up only a quarter of the core memory of 8 million words(8).

Since vector processing on the Cyber 205 is the computational means by which the new algorithm will be developed, it is appropriate to discuss the fundamentals of the vector processing computational system.

VECTOR PROCESSING

Vector processing is the next logical computer processing development after the scalar computer operation. Where as a scalar operation involves only individual data points being operated upon, vector processing involves operating on whole vector streams of data to do the same operation. For purposes of vector computing, a vector can be described as a contiguous set of elements that have in common the fact that the same operation will be applied to all of them. It is very important to note that the members of the vector are all stored sequentially in the computer memory, not scattered about randomly. While it is possible to vectorize randomly distributed data points, that operation involves the extra expense of searching for and then storing the data points sequentially in memory and then performing the vector function.

Vector processing takes advantage of the fact that the same

operation will be applied to a large amount of data. In scalar computing, the instructions are loaded and data is retrieved, loaded, and then returned for each operation. In vector processing, the computer knows that the same instruction will be used for a fixed amount of operations and therefore doesn't need to retrieve it every time. The same is true of the data, since the computer knows that the next data point is in the next sequential memory location, the amount of time needed to locate and retrieve is cut drastically.

It has been determined that vector processing is much more efficient and advantagous when applied to vectors of 50 or more(7). When a call for a vector operation is made, an actual restructuring of the computer processor is accomplished. The processor realigns itself based on the instruction so as to facilitate the vector processing. This initial restructuring involves a certain amount of computer time "overhead". When short vectors, normally considered below 50, are operated upon, not enough operations are performed to amortize this initial overhead expense. However, as vectors get longer, this overhead is spread out so thinly as to allow the computation time to approach one clock cycle per

result. With this in mind, any algorithm based on vector processing should stress the use of long vector streams of data to achieve the most efficient solutions. A vector processor is able to approach the rate of one result per clock cycle due to a pipelining of instruction execution within the processor itself.

The restructuring within the processor that was discussed earlier ensures that all the necessary computational components within the processor are prepared for execution. Implicit in this action is the set-up of an assembly line of computation actions that each result must pass through. For purposes of illustration, a sample operation that consists of an instruction fetch (IF), instruction decoding (ID), operand fetch (OR), and instruction execution (EX) will be analyzed. In a computer that is not able to pipeline instructions, each of these four steps must be completed before the next operation can be started, a situation shown schematically in figure #2. This figure shows each instruction completing the four part processing before the next instruction can be started(13).

soon as the previous instruction cycle has been acted upon by the IF. Once this start-up is over, a completed instruction cycle will be generated for every clock cycle of the computer. With this capability, not only are time savings incurred from the greater speed of the newer computers, but also from the more efficient instruction execution of instruction pipelining.

Rudolfo Garza's 1981 dissertation on vector processing applications to linear programming bears this point out(8). He was able to achieve computer time reductions by a factor of 100 on large vector stream operations as compared to scalar computing. However, on very short vector lengths, the scalar processor was as fast as or in some cases faster that the vector processed data.

Another interesting aspect of vector processing is the use of dummy calculations to keep a vector operations going. Based on vector length, the total CPU time difference to operate on a vector of length N and a vector of length N+1 is negligible. However, the amount of time needed to stop and restart a vector operation is large since the restructuring overhead is incurred again. In some cases, it is actually efficient to create dummy operations to keep a vector operation going as opposed to

stopping and restarting the vector function later. This situation may arise in cases where a trade-off between memory use and execution speed is made. The storage of extra or "dummy" data can many times remove the necessity of closing a vector operation down and then restarting it. These dummy results are stored as are the good results, however, only the good results need be retrieved for further manipulation.

The Cyber 205 has both implicit and explicit vectorization. In the implicit mode, if the CPU of the Cyber 205 locates a control function, such as a DO loop that it recognizes as vectorizable, the vectorization will occur automatically. In this case, the vectorization may occur without the programmer's knowledge, or the programmer formulates the regular Fortran code in such a manner that the Cyber 205 recognizes the structure as vectorizable. An example of formulating for implicit vectorization is setting up a DO loop in which the right most subscript of an array is held constant. This is equivalent to moving down the column elements of the array, an action that the processor recognizes as vectorizable. There are however, special vector Fortran statements that explicitly tell the computer what to do, where to go in memory to find the data, and how

much of the data to operate on.

An example vector Fortran statement is

Array1(1;10) = Array2(1;10) + Array3(10;10)

This statement instructs the computer to add together ten elements from each vector Array2 and Array3 starting at array subscript locations 1 and 10 respectivley and store the ten results in Array1 starting at subscript location 1 given all arrays are one dimensional. If the arrays where two dimensional, the subscript starting location would be a two number address, for three dimensions, a three number subscript location.

The actual sequential storage of data in memory can be easily visualized, and a grasp of this aspect of vectorization is necessary to an understanding of the technology. There is an identifiable relationship between array subscript and memory storage. The following graph shows that relationship between array subscripts of an array A(3,2,2) and the ordinality sequence that the vector processor recognizes(4):

ORDINALITY	ELEMENT OF A(3,2,2)
1	A(1,1,1)
2	A(2,1,1)
3	A(3,1,1)
4	A(1,2,1)
5	A(2,2,1)
6	A(3,2,1)
7	A(1,1,2)
8	A(2,2,2)
9	A(3,1,2)
10	A(1,2,2)
11	A(2,2,2)
12	A(3,2,2)

FIGURE #4 - ORDINALITY OF ARRAY SUBSCRIPTS

VECTOR PROCESSOR PIPELINING

The CDC Cyber 205 can be described as a double pipe vector processing machine. This implies that within the Central Processing Unit of the computer, two completely different and independent vector operations can be accomplished at the same time, each by a different vector processor. The vector processors are sub-elements of the CPU much as an Arithmetic Logic Unit is a sub-element of a serial computer.

The double pipe allows operations on long vectors to be accomplished in roughly one-half the time that would be required if only a single vector processor element was used(4). The actual double pipeline division of

labor is as shown in figure #5.

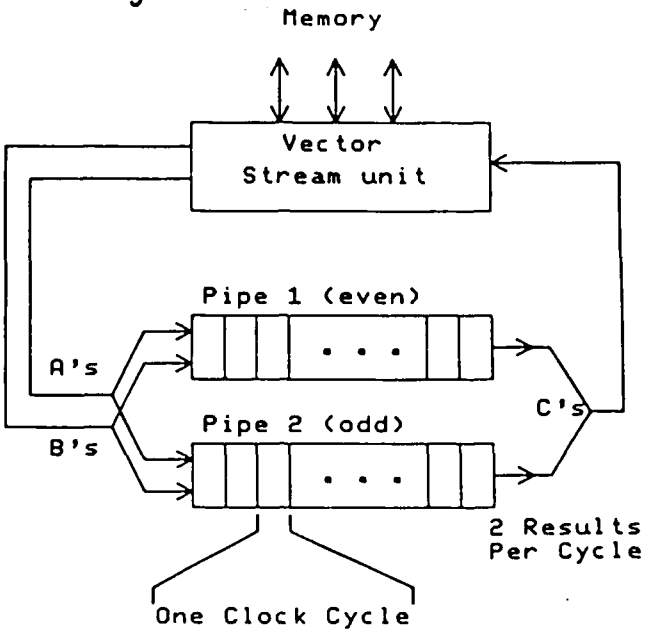


FIGURE #5 - DOUBLE VECTOR PIPELINE OPERATION

As is shown, every other operation is sent to the same processor. For this description, an individual vector processor would consist of one complete sequence of execution units such as the IF, ID, OF, and EX described in the instruction pipelining discussion. Both vector processors have been reconfigured electronically or "micro-coded" to perform the same computer instruction. Once the input vectors have been split up by alternating elements to be pipelined, the resultant vector is automatically re-integrated for memory storage. The reason the time factor reduction from going from one pipeline to two pipelines is not exactly one-half is

that more time is required to initially "micro-code" two vector processors than one vector processor.

Other control structures of vector Fortran make even more efficient use of the double pipeline. These structures make use of the fact that both vector processors can be "micro-coded" to do different tasks at the same time. The result is a two sequence operation that has the result of one pipeline becoming the input to the second pipeline. An excellent example of this is called the "linked triad". The "linked triad" instruction is of the form

$$D(i) = A(i) + B(i) * C(i)$$

Vectors B and C are input into the first vector processor and then the result of that operation and vector A is input into the second processor the output of which is stored in vector D. Figure #4 show the pipeline division of labor as well as the sequencing idea that the "linked triad" entails. The advantage of the "linked traid" can be found in the "micro-coding savings". Using the "linked triad" set-up, both vector processors need be micro-coded only once, while performing the same

operation using the previously discussed vector pipelining requires both vector processors to be micro-coded twice(5).

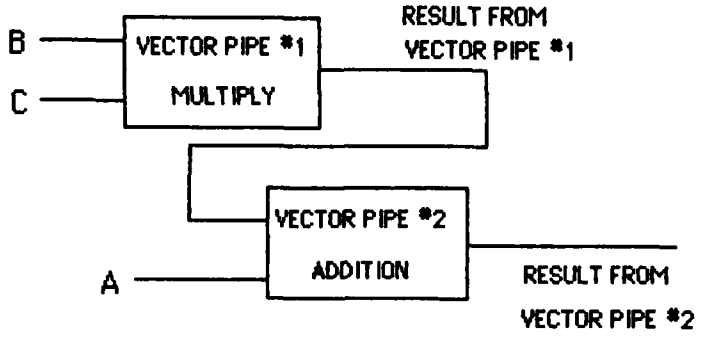


FIGURE #6 - A(i)+B(i) *C(i) PIPELINING

MEMORY AVAILABILITY

One of the major advantages of the CDC Cyber 205 over older serial computers is that being a much newer computer it had state-of-the-art computer engineering incorporated into its design. This is very evident and important in the area of memory availability. The Cyber 205 is known as a virtual memory machine. Every computer has a certain amount of memory that is really nothing more than the computers immediate working

space or "core". For example, on the University of Texas Cyber 170/750, the core memory is roughly 131,000 words of information, while the University of Minnesota Cyber 205 has roughly 8 million words of core memory. The added advantage of virtual memory implies that as long as enough disk and off-core memory is available, the computer will automatically swap information in and out of core as needed for processing, and in this fashion, a computer with almost limitless memory has been created.

LITERATURE REVIEW

In historical Operations Research literature, a single state variable Dynamic Programming model that is very similiar to the Pallet Loading problem is known as the "Knapsack" problem. The Knapsack problem can be considered as that facing a traveller loading a knapsack or suitcase. There are a number of items to be taken along, each with a certain value or utility. The knapsack must be loaded keeping the total weight in mind(This is the state variable.) If the suitcase is used instead of the knapsack, it must be loaded keeping the total volume in mind. In the pallet loading problem, both weight and volume must be considered - a two state variable problem. While not as complex as the two-state pallet loading problem, the knapsack problem nonetheless can provide some insight into the workings of dynamic programming allocation. The literature review will therefore concentrate on areas related to vector processing as applied to Operations Research problems, Dynamic Programming in general, and on solution techniques to either the

knapsack or the pallet loading problem.

In 1961, P.C. Gilmore and R.E. Gomory began the first real analysis of a "knapsack" type problem. Their methodology at first concentrated on a straight column-generating Linear Programming solution approach. This was an attempt to apply the latest solution technique, Linear Programming, to the problem(8). In 1963, they further revised the column-generating Linear Programming approach to the problem by attempting to pre-screen candidates for loading based on a ratio of utility divided by state variable parameter(9). In 1965, Gilmore and Gomory attempted the use of guillotine cuts on the solution space of the problem in an attempt to find and alternate solution technique. The guillotine cuts approach was a pre-cursor to the cutting plane algorithms applied in integer programming situations today(10). The major problem that they were not able to overcome was the enormous number of columns generated by a linear programming formulation of the problem. For one state variable knapsack problems, they solved the knapsack problem at every stage as classical DP prescribes, however at any more than one state variable, they were not able to overcome the

difficulty created by the extra dimensionality.

In 1973, J. Casti, M. Richardson, and R. Larson(3) evaluated a single state variable dynamic programming model in terms of parallel programming. The parallel programming they used referred to the parallel positioning of whole processors to perform operations at the same time. Their approach was to evaluate different state level values from the same stage at the same time using the ideas of parallel processing. The restrictions on this approach are the number of processors that can be parralleled, and computer memory. Their work involved a routing of operations to multiple processors, not a new Dynamic programming algorithm or an application of vector processing.

George Nemhauser(14) best explains how Lagrange Multipliers can be used to reduce the dimensionality of the matrices involved in the solution of dynamic programming models. As explained earlier, the greater the number the state variables of the problem, the higher dimension the matrix must be, accompanied by the ensueing computer storage problems. Lagrange Multipliers allowed for the reduction of one state variable per multiplier. As the problem was reformulated in terms

of one extra multiplier instead of a state variable, the dimensionality of the problem was reduced. However, another set of terms incorporating the multiplier was added to the objective function. Using this approach, the optimal solution could be reached only by estimating the Lagrange Multiplier as an approximation to the optimum solution. The use of Lagrange multipliers was successful in reducing the dimensionality of the problem, however, this came at the expense of the complexity of the objective function evaluation. The approximation of the Lagrange Multipliers also added much extra computation to the problem. The lagrange multiplier approach given enough computational time would normally converge to the optimum.

In 1975, H.M. Salkin(15) published a survey of the knapsack problem history which confirmed the common beliefs about dynamic programming and the knapsack problem. He determined empirically that for small problems where computer storage is not a constraint, dynamic programming was as efficient as any other technique for solution. However, for large problems, Sakin found that optimum approximation techniques worked more efficiently as a solution technique.

In 1979, Prabhakant Sinha(16) attempted to solve what he called a "multiple choice knapsack problem". This is the same knapsack problem os one state variable and many stages except with the added option of multiple choices from a given stage. In all previous work, the decision at any stage was a 0-1 choice. Sinha used a branch and bound technique that amounted to a non 0-1 integer programming algorithm. Sinha's approach to the problem worked, but was only efficient for problems with small numbers of decision variables. At problem sizes approaching anything practical, the implicit problems of integer programming surfaced.

In all the literature reviewed, the closest approximation to the problem stated in this paper is R.E. Bellman's "fly away kit" problem(2). Bellman introduced this problem that involved two state variables, weight and volume, and a utility return based on a probablility distribution based on expected values of stage utilities. Bellman solved this problem using classical dynamic programming techniques and the serial computation that it entailed. In his small example, a 30x30x10 matrix (representing initial state variable 1 and 2 values of 30, and 10

distinct parcel classes), was created and every individual component of the matrix had to be solved for in a serial fashion, moving from stage to stage. Bellman was able to solve the problem using the classical dynamic programming techniques that he fathered. His method involved finding the maximum value for every element of the 30x30x10 individually. This amounted to 9000 suboptimizations one right after another. Bellman's classical DP algorithm has been the standard for discrete state level DP employed to this very day.

This paper attempts to solve the same basic problem as Bellman's fly away kit problem by applying vector processing to create an algorithm that will solve much more efficiently for the elements of the main array.

CHAPTER 3

To illustrate the algorithm developed for the multi-state dynamic programming model, a sample problem will be used. Consider the following scenario: There is a small loading platform with a volume limit of 7 cubic ft. and a weight limit of 7 lbs. There are several boxes to be loaded onto this loading platform each with various load parameters of utility, weight, and volume. There are three boxes with utilities of 4, and weight and volume of 2 lbs and 1 cu. ft. respectively. There are 2 boxes with a utilities of 5, and weight and volume of 3 lbs. and 2 cu. ft. respectively. Finally, there are 2 boxes with utilities of 7, and weight and volume of 2 lbs and 3 cu. ft. respectively. While the solution to this small sample problem is trivial, the value of solving this problem is in the better understanding of the algorithm itself.

It is obvious that not all these boxes can be loaded onto the platform. An algorithm is needed to help select that combination of the above mentioned boxes that while remaining within the limits of the load platform, maximizes the sum of the utilities of the various boxes that

will be loaded. In this chapter, an algorithm based upon vector processing that will determine that optimal load combinations will be explained.

The above mentioned scenario has specific load parameters. For the general pallet loading problem of N state variables and M parcel classes, the formal mathematical model is of the following form:

Given

x_i = the number of parcels from parcel class i contained in the optimal solution,

v_i = the utility value of including a parcel from class i in the optimal solution,

A, . . ., G, . . . = the initial values of the state variables 1 to N of the problem. This corresponds to the maximum amount of state variable resource available for allocation,

a_i , . . ., g_i , . . . = the respective amounts of state variable resources 1 to N that will be used if a parcel from parcel class i is loaded. This amounts to an allocation of system

resources to include a parcel in the optimal load,

MN(i) - the maximum number of parcels from parcel class i that can be loaded. This number is determined by counting all parcels with the same load parameters and setting it as an upper load limit for that parcel class,

$$\max \quad \left(\sum_{i=1}^{N} X_i * V_i \right)$$

subject to

$$\sum_{i=1}^{N} a_i * x_i <= A$$

. . .

$$\sum_{i=1}^{N} g_i * x_i <= G$$

$X_i = 0, 1, 2, \ldots, MN$ for all i

$A, \ldots, G, \ldots > 0$

$a_i, \ldots, g_i, \ldots > 0$ for all i = 1 to N

$v_i >= 0$ for all i = 1 to M

It is appropriate at this time to define several terms that relate to the explanation of the algorithm. The definitions will be specific for the two state variable problem that the algorithm is designed to solve.

NX - number of discrete levels of state variable 1, beginning at level 1, the levels of which ascend in increments of 1

PY - number of discrete levels of state variable 2, beginning at level 1, the levels of which ascend in increments of 1

SV1 - the level of state variable 1 at any given point, can be thought of as the amount of state variable 1 left to allocate

SV2 - the level of state variable 2 at any given point, can be thought of as the amount of state variable 2 left to allocate

PARCEL CLASS - this corresponds to a Dynamic Programming stage, and represents the group of all parcels with the same state variable and utility parameters

NUMB - the number of parcel classes that the load will be selected from, equal to the number of DP stages

X - the actual number of parcels from a given parcel class being loaded at a given time

S1(i) - the amount of state variable 1 resource used up by selecting a parcel from class i. This corresponds to "a " in the formal mathematical model

S2(i) - the amount of state variable 2 resource used up by selecting a parcel from class i. This corresponds to "b " in the for mathematical model

Both S1(i) and S2(i) represent what are called the state variable parameters of a given parcel class. For example, if in a problem the maximum amount of weight available was 50 lbs, and 20 lbs had already been allocated then NX would equal 50(1 value per lb), SV1 would equal 30.

Along the same lines, if for every parcel from a parcel class i 5 lbs and 3 cu. ft. were used, then S1(i) and S2(i) would equal 5 and 3 respectively.

Since this is a 2 state variable Dynamic Programming formulation, as mentioned earlier in this report, a 3-dimensional array must be created to hold all the necessary information for solution. Held within this 3-D array will be both the optimal load policy for every parcel class given all possible combinations of SV1 abd SV2, and also the maximized utility of the system based on the number of parcel classes that the load is being selected from at that moment. While looking very much like a complete enumeration, as will be shown later, Dynamic Programming actually bounds the solution set at each stage.

Literally, Dynamic Programming performs the following events: The algorithm moves sequentially from stage to stage, checking to see if by loading elements from the last parcel class evaluated the utility of the entire load will increase. If by loading parcels from parcel class i at any state variable combinations the system utility will increase, then those parcels are incorporated into the optimal load. At stage i, the algorithm is selecting the best load possible from i parcel classes, at stage i+1, the

algorithm is selecting the best load from the first i+1 parcel classes. These loads will be different based on the various load parameters of parcel class i+1.

A major foundation of Dynamic Programming theory is know as Bellman's Principle of Optimality. It states:

"An optimal policy has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from that initial decision."

This is a very powerful statement indeed. It implies that given we know what the optimal policy is for all state variable combinations of stages 1 through i, all that is necessary is to check every combination of loads from parcel class i+1, determine which of any of these will increase the system utility in combination with the optimal from stage i, and then for every combination of state variables at stage i+1 choose the number to load from parcel class i+1 that maximizes system utility. At this point, the optimal load policy and system utility for all state variable combinations from the first i+1 parcel classes has been determined. To actually determine what the optimal load for a system is given initial

state variable values employs a method known as backcasting.

Remembering Bellman's Principal, backcasting is the technique of moving sequentially from the last stage evaluated to the first, and selecting the load from parcel class i that corresponds to the present levels of SV1 and SV2. Moving to stage i-1, the values of SV1 and SV2 are adjusted based on the number loaded from parcel class i. The transition functions are

$$SV1(\text{stage } i-1) = SV1(\text{stage } i) - X * S1(i)$$

$$SV2(\text{stage } i-1) = SV2(\text{stage } i) - X * s2(i)$$

where X is the optimal number loaded from stage i.

Bellman's Principle implies that the number to load at stage i-1, given the new SV1 and SV2, will have already been determined to be optimal. This procedure is then repeated until all the parcel stages have been evaluated. A numerical example of the entire process will be given very shortly.

For each parcel class, the only information saved is the optimal number to load from the stage for all state level combinations. The maximized utility at each stage is not retained because Bellman's

Principle says that at every succeeding stage we are storing the maximized system utility and that is the only useful piece of information for determining what the optimal load from parcel class i +1 would be. At every stage i, the maximized utility for all state variable combinations is held in a portion of the 3-D array discussed earlier. This information is then used to determine what the optimal load from parcel class i+1 should be, and once the new system utilities generated by selecting from i+1 parcel classes are determined, the old utility values in the array are replaced by these new utility values. To appreciate the workings of the algorithm, an understanding of the main 3-D array is in order.

For purposes of the algorithm, the array will be entitled MAIN, and have the dimensions (0:NX+1,1:PY,0:NUMB+2), where the first term of a subscript refers to the beginning value of the index, and the term following the colon refers to how many levels of the index there are starting at the beginning value of the index. The interpretation of these subscripts are as follows:

0:NX+1 - one subscript element for every value SV1 can take on, with the 0 starting point implying no more resource available

1:PY - one subscript element for every value that SV2 can take on

0:NUMB+2 - this is the third dimension of the matrix MAIN and has 2 subscript values, 0 and NUMB+1 that hold the system utilities for parcel classes i+1 and i respectively, the NUMB subscripts between 0 and NUMB+1 hold the optimal policy for each parcel class over all state variable combinations. For purposes of the explanation, a Slice(i) will refer to a final subscript of the main array taken across all of its elements at that specific level. Graphically a Slice(i) represents the following:

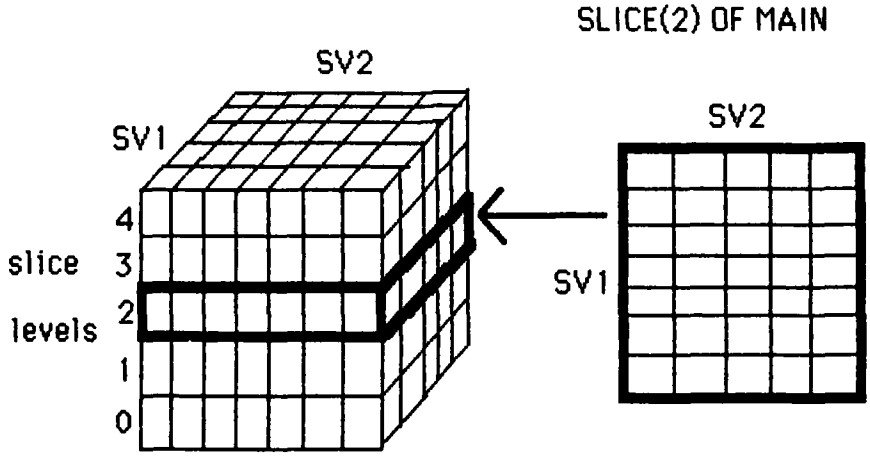


FIGURE #7 - SLICE BREAKDOWN OF ARRAY MAIN

The slices of main all hold different information as discussed earlier.

Slice(NUMB+1) - this slice holds the present maximized system utilities up through stage i

Slice(0) - this slice holds the newly generated system utilities for stage i+1 before they replace the old values in slice(NUMB+1)

Slice(1) to Slice(NUMB) - each of these slices holds for a respective parcel class, the optimal load for all state variable combinations.

The actions specified by the algorithm so far are:

(a) collect all pertinent load information parameters

(b) set up array MAIN with appropriate subscripts based on data

(c) initialize elements of the utility holding Slices to zero. This is done because before any pallets are loaded, the utility at all state variable combinations must equal zero. At this point, and example problem is in order. The example problem consists of 3 parcel classes with the following load parameters:

	NX = 7		PY = 7	
parcel class #	U(i)	MN(i)	S1(i)	S2(i)
1	4	3	2	1
2	5	2	3	2
3	7	2	2	3

TABLE #2 - SAMPLE PROBLEM PARAMETERS

See page 39 of chapter 3 for formal parameter definitions.

Literally, the DP algorithm, for all state variable combinations, selects the optimal load from stage 1, then selects the optimal load from stages 1 and 2, then selects the best load given all three stages to choose from. This first stage will be solved for using the classical DP techniques without any vector processing influence. Based on techniques learned solving for the first stage, the second stage will be solved showing how the new vector processing techniques are applied. The 3 stage, 2 state DP problem is shown graphically in figure #1 in chapter 1.

To initially solve the problem, the array MAIN with appropriate

form (0:8,1:7,0:5) is created and can be represented graphically as follows:

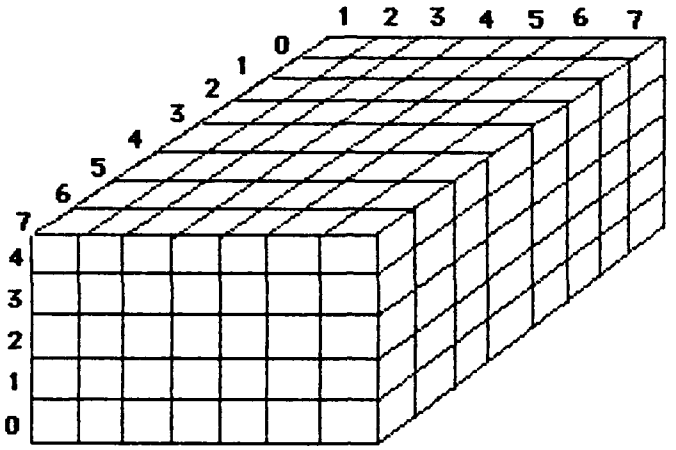


FIGURE #8 - ARRAY MAIN

Slice(0) which holds utilities for stage i+1, and slice(4) which holds utilities for stage i are initialized to zero. Slice(1) which holds the optimal load policies for parcel class one is represented as follows:

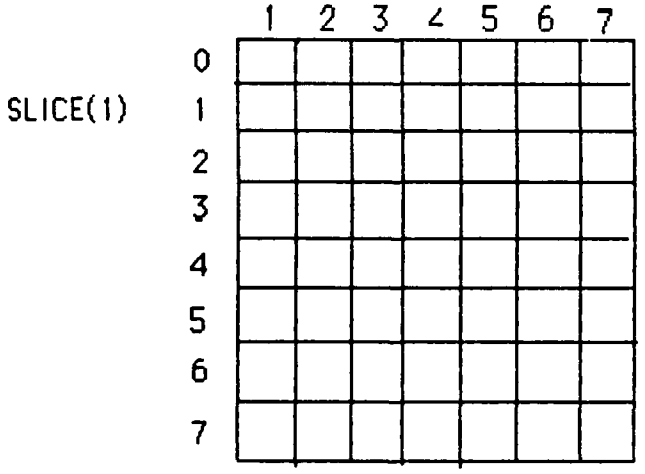


FIGURE #9 - SLICE(1)

Each element of this Slice represents a single state variable combination. Further, at every one of these state variable combinations an attempt must be made to load up to MN(1) parcels. Physically, this may be impossible. If the amounts of SV1 and SV2 are less than S1(1) and S2(1), then a return of NL(No Load) is assigned. A return of NL implies a situation that cannot exist and therefore cannot be incorporated into the solution. Using classical DP, at every state variable combination, a one dimensional array is created to hold temporary utility values. In the case that MN(1) is set at three, the array has the following form:

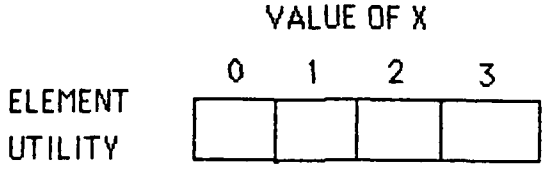


FIGURE #10 - ELEMENT EVALUATOR

For every element of Slice(1), an evaluation of all four of the elements of the holder array will be attempted. The maximum value in the holder array will be selected, and the corresponding X will be entered into slice(1) at the appropriate SV1 and SV2 subscript. The maximized utility will be placed in the same subscript of Slice(0) temporarily. For one element, the process can be represented as

- 0*U(1) + utility previous given adjusted SV1 and SV2

-1*U(1) + utility previous given adjusted SV1 and SV2

- 2*U(1) + utility previous given adjusted SV1 and SV2

- 3*U(1) + utility previous given adjusted SV1 and SV2

The SV1 and SV2 adjusted represent subscripts in Slice(4) which holds the previous stages utilities. The situation where SV1=7 and SV2=7

will be evaluated as an example. It must be remembered that for stage 1, all the previous utilities will of course be zero.

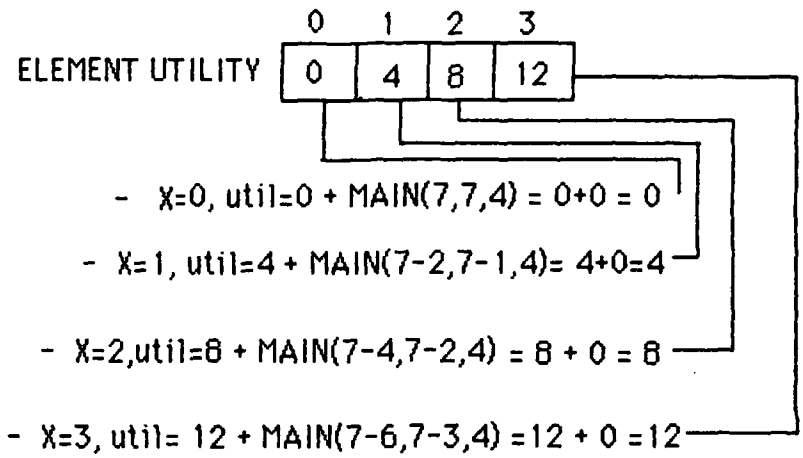


FIGURE #11 - SV1=7, SV2=7 ELEMENTS

At this point we would choose the maximized utility of 12 at x=3. The changing subscripts in MAIN account for the state variable resource being used up by loading elements of parcel class 1. At this point, every possible load from parcel class 1 for the state variable combination SV1=7 and SV2=7 has been checked. The situation where SV1=5 and SV2=6 will be solved as a final example.

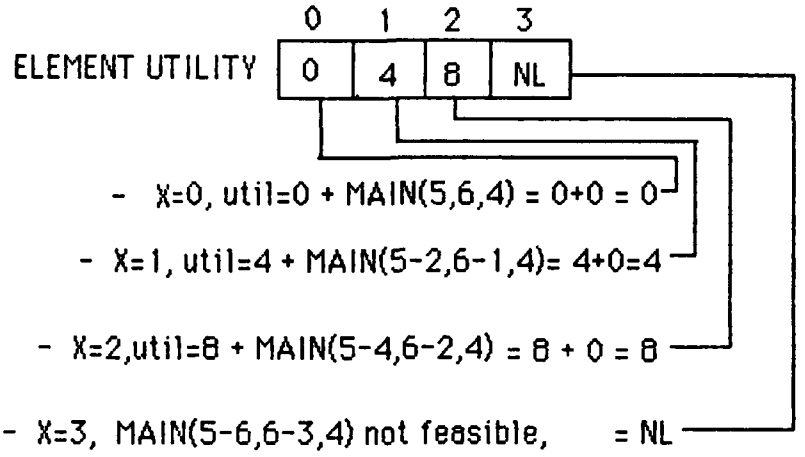


FIGURE #12 - SV1=5, SV2=6 ELEMENTS

At this point, a utility of 8 is chosen at x=2. After these two operations, Slice(0) and Slice(1) have the following forms:

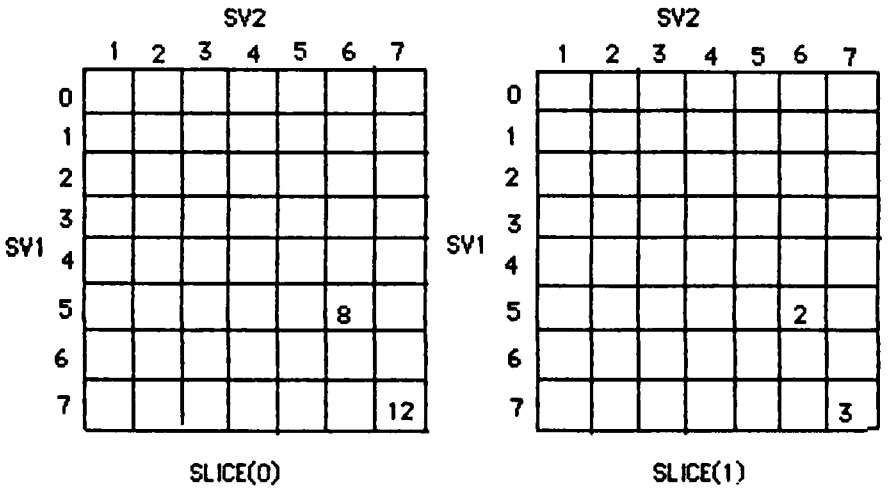


FIGURE #13 - SLICES 0 and 1 after 2 OPERATIONS

It is left as an exercise to complete the process for every element of Slice(1) and complete Slice(0) and Slice(1) into the final form of:

SLICE(0)

SV1 \ SV2	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	4	4	4	4	4	4	4
3	4	4	4	4	4	4	4
4	4	8	8	8	8	8	8
5	4	8	8	8	8	8	8
6	4	8	12	12	12	12	12
7	4	8	12	12	12	12	12

SLICE(1)

SV1 \ SV2	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1
4	1	2	2	2	2	2	2
5	1	2	2	2	2	2	2
6	1	2	3	3	3	3	3
7	1	2	3	3	3	3	3

FIGURE #14 - SLICES 0 and 1 COMPLETED

Now that all calculations for stage 1 are completed, the old utility elements of Slice(4) are replaced with the new utility elements of Slice(0). Slice(0) is re-initialized to zero. It is now time to move onto analyzing the second parcel class. This stage will be analyzed using the vector approach to Dynamic Programming as opposed to classical DP. At this stage, instead of a single element of Slice(2) being analyzed, a whole column at a time will be evaluated. For example, the entire column 7 of Slice(2) will be checked first. This corresponds to the state variable,

SV2 =7, i.e., 7 lbs of loadable weight crossed with all volume levels.

SLICE(2)

	1	2	3	4	5	6	7
0							
1							
2							
3							
4							
5							
6							
7							

FIGURE #15 - SLICE(2)

For every element of column 7, the classical DP technique will be the same, however, when these smaller operations are viewed as a whole, the vector approach to the situation becomes clear. Setting the problem up into a column of smaller classical DP formulations, the information contained in table #3 is derived. The table has the same elements as the classic DP such as X value, parcel class utility, previous system utility, Slice(4) subscript values, and total utility. The maximum utility for each column element is darkened. The optimal X value and utility for each column element are summarized at the bottom of the table.(NL = No load)

ELEMENT	# LOADED	UTILITY FROM STAGE 2 LOAD	MAIN SUBSCRIPTS	UTILITY ADDITION	TOTAL UTILITY
	x=0	**0**	**(1,7,4)**	**0 + 0**	0
1	x=1	NL	-	-	-
	X=2	NL	-	-	-
	X=0	**0**	**(2,7,4)**	**4 + 0**	**4**
2	X=1	NL	-	-	-
	X=2	NL	-	-	-
	X=0	0	(3,7,4)	4 + 0	4
3	**X=1**	**5**	**(5,5,4)**	**0 + 5**	**5**
	X=2	NL	-	-	-
	X=0	**0**	**(4,7,4)**	**8 + 0**	**8**
4	X=1	5	(1,5,4)	0 + 5	5
	X=2	-	-	-	-
	X=0	0	(5,7,4)	8 + 0	8
5	**X=1**	**5**	**(2,5,4)**	**4 + 5**	**9**
	X=2	NL	-	-	-
	X=0	**0**	**(6,7,4)**	**12 + 0**	**12**
6	X=1	5	(3,5,4)	4 + 5	9
	X=2	10	(0,3,4)	0 + 10	10
	X=0	0	(7,7,4)	12 + 0	12
7	**X=1**	**5**	**(4,5,4)**	**8 + 5**	**13**
	X=2	10	(1,3,4)	0 + 10	10

TABLE #3A - ELEMENTS OF COLUMN 7

SV1	LOAD POLICY	MAXIMIZED UTILITY
1	0	0
2	0	4
3	1	5
4	0	8
5	1	9
6	0	12
7	1	13

TABLE 3B - LOAD POLICY AND UTILITY OF COLUMN 7

If the elements of the table are reorganized, and the entries are grouped based on the same X values, the formulation is reduced to adding the same X*U(2) value to certain previous utilities, that are represented as elements of Slice(4). Mathematically, this process can be shown as

at x=0, UTOT(1->7) = 0 + (0,7),(1,7),(2,7),(3,7),(4,7),(5,7),(6,7),(7,7)

at x=1, UTOT(3->7) = 5 + (0,5),(1,5),(2,5),(3,5),(4,5)

at x=2, UTOT(6->7) =10 + (0,3),(1,3)

Each one of the subscripts represents a utility stored in Slice(4) of MAIN. Graphically, those elements are located as follows:

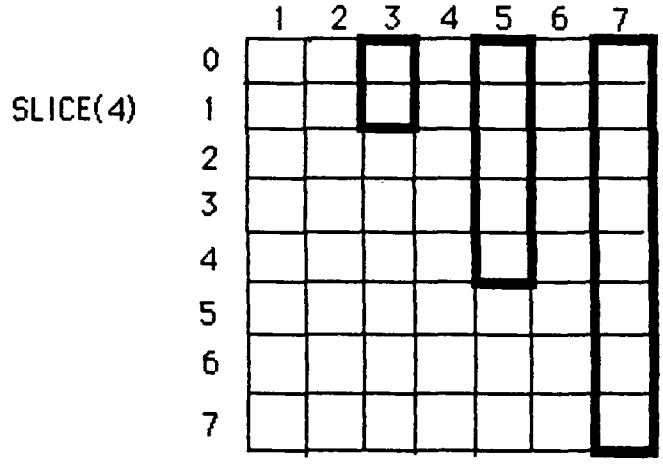


FIGURE #16 - SLICE(4)

These elements represent data stored contiguously within the computer. When viewed in this light, the formulation seems perfect for a vector processing approach which operates on contiguous data elements at the great speed and efficiency discussed earlier. Previously, the classical DP technique stored the temporary utilities generated for every state variable combinaiton in a one dimensional array. In a vector processing mode, the temporary utility values are stored in a two dimensional array, where each record of the array corresponds to a classic DP formulation for the Slice element. This array called BUFF has the dimensions

(1:NX,0:MN+1). For every column of every Slice(i), array BUFF is initialized, filled, and evaluated. Array BUFF is shown graphically as:

		VALUES OF X		
		0	1	2
	1	0	0	0
	2	0	0	0
SV1	3	0	0	0
	4	0	0	0
VALUES	5	0	0	0
	6	0	0	0
	7	0	0	0

FIGURE #17 - ARRAY BUFF INITIALIZATION

In this algorithm, the bulk of the vector processing work is done in the filling and evaluating of BUFF. The operations performed on BUFF at each column can be generalized to the following:

-initilize the elements of BUFF to zero, this can be done at vector speed by setting the starting point as the first element of the array, and then setting the vector operation length as the number of

elements in the entire array.

-add the X*U(i) utility product to every element of the appropriate column of BUFF. Referring back to table #3 , this is simply the scalar constant that must be added to a previous utility value. For vector processing, the beginning subscript and length of oepration are needed. In this case, the beginning subscript is simply the level of state variable 1 at which it is physically possible to load the given number of parcels from the parcel class. For example, in this problem, to load x=0 no state variable1 resource is used and the starting subscript is BUFF(x=0,1). For x=1, the minimum SV1 value to load is 3, so the vector starting point is BUFF(x=1,3). The length of the vector operation is simply (NX+1) - S1(i). This accounts for the starting element and all the other column elements that follow. For the final x=2, the minimum SV1 value is 6, giving the starting point BUFF(x=2,6) and a length of (7+1)-6 = 2.

Looking at figure #16 and the contiguous elements of Slice(4), it can be seen that the lengths just calculated for entering the utility constants are correspondingly the same as the number of previous utility look-ups for a given x. For example, at x=1, the starting point is BUFF(1,3)

with a length of 5. The utility look-up corresponding to x=1 consists of 5 contiguous elements of Slice(4) beginning at (0,5). The problem reduces to adding 5 X*U(i) utility constants to appropriate 5 contiguous elements of Slice(4) and placing them in the correct elements of BUFF. Referring back to figure #16 it can be seen that the columns of Slice(4) that are needed have a constant interval between them. This interval is the value S2(i). This is due to the fact that the second subscript represents state variable 2 resource. At a given x value, an amount of state variable 2 resource equal to X*S2(i) will be used up. As x increases, the column of Slice(4) that is needed for evaluating the previous utility moves down at an increment of S2(i).

The actual filling of the columns of BUFF has been reduced to (a) adding a utility constant of x*U(i) to the (NX+1)-S1(i) starting at location BUFF(x,x*S1(i)), and (b) given the same x, adding to the same starting point and for the same length the (NX+1)-S1(i) elements of Slice(4) beginning at subscript location (0,SV2-x*S2(i)) where SV2 represents the column number of Slice(2) that is presently being evaluted. The memory location corresponding in MAIN is MAIN(0,SV2-x*S2(i),NUMB+1). For every

column being evaluated, the above two steps are completed for x equals zero up to x equals MN(i).

At this point, it is a simple matter of scanning across each row of BUFF to determine which element is the maximum value, and determining in which column of BUFF that it occurred. That column number would correspond to the optimal load for the given state variable combination. These searches can be accomplished using pre-defined VECTOR FORTRAN functions that compare elements in two vectors of the same length. The columns of BUFF are all the same length and can be compared one against another. The procedure is to compare the x=0 column to the x=1 column and storing the maximum in a one dimensional holder array of the same length. The appropriate load number is stored in a separate array of the same length. The holder array which contains the maximized utility from the x=0 and x=1 columns is then compared to the x=2 column, maximums and indices are stored, and the process is then repeated up until the x=MN column has been compared. The two holder arrays now store both the maximized utility for the state variable combinations, as well as the x value for the optimal load. These columns of data are then stored in the

appropriate columns of Slice(0) and Slice(2) respectively. This process is then completed for every feasible column of Slice(2) at which time the old utility values of Slice(4) are replaced by the newly calculated utility values stored in Slice(0).

Getting back to our example problem, and column seven that was being analyzed, figure #18 demonstrates the relationship between the contiguous elements of Slice(4) of MAIN, the x*U(i) utility values, and the data elements of BUFF.

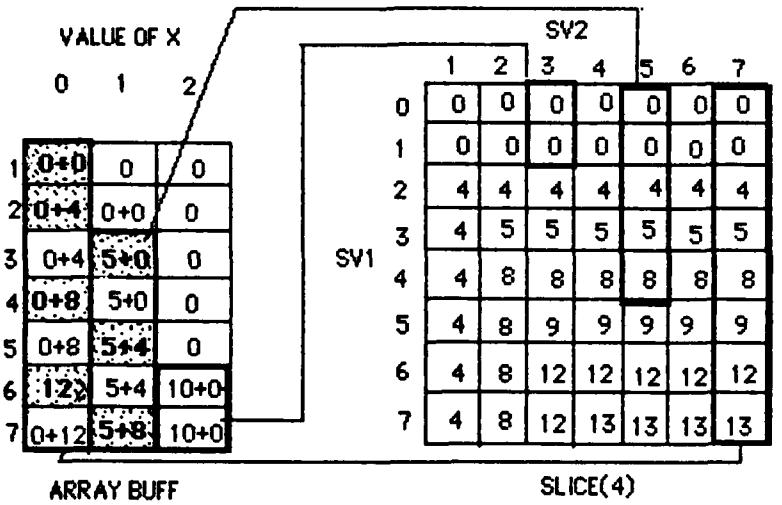


FIGURE #18 - ARRAY BUFF AND MAIN VECTORS

The maximum values that would be determined by a vector search are darkened. The optimal value holder arrays relationship to the Slices of MAIN are shown in the following graph:

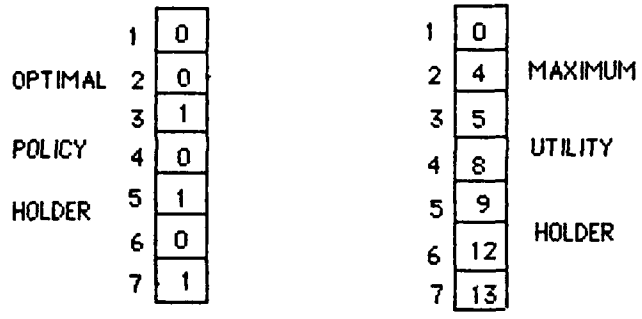


FIGURE #19 - POLICY AND UTILITY HOLDERS

For column numbers that are less than S2(i) for a given parcel class, it is obvious that no parcels can be loaded. In this case the utility from stage i is equal to the utility from stage i+1, since if no parcels can be loaded, the optimal from the previous stages will still be the optimal from i+1 stages. If this process is completed for all columns of Slice(2), then slice(2) and Slice(4) will appear as the following:

SLICE(2)

SV1 \ SV2	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	1	1	1	1	1	1
4	0	0	0	0	0	0	0
5	0	0	1	1	1	1	1
6	0	0	0	0	0	0	0
7	0	0	0	1	1	1	1

SLICE(4)

SV1 \ SV2	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	4	4	4	4	4	4	4
3	4	5	5	5	5	5	5
4	4	8	8	8	8	8	8
5	4	8	9	9	9	9	9
6	4	8	12	12	12	12	12
7	4	8	12	13	13	13	13

FIGURE # 20 - SLICES 2 AND 4 COMPLETED

For the final parcel class, there is only one state variable combination of interest. That is the state variable combination that corresponds to the initial values of the state variables. These values are NX and PY repectively for state variable 1 and state variable 2. For this final stage, the classical DP approach is the easiest to use. The vector approach is used on all stages up until the last one. The classical DP approach evaluates the last stage in the following manner:

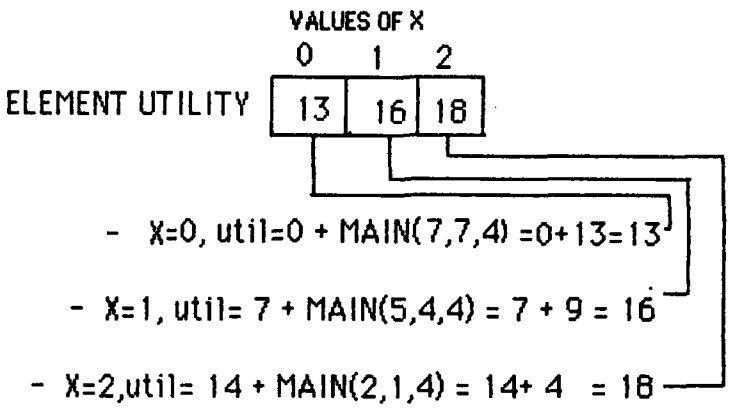


FIGURE #21 - SV1=5, SV2=6 ELEMENTS

The amount to load from parcel class 3 that maximizes the system utility is 2 parcels at a system utility of18. At this point Bellman's Principle of Optimality allows for the backcasting technique to quickly retrieve the optimal load for every parcel class. As Bellman's Principle states, no matter what the initial decision, an optimal policy has the property that the remaining decisions must constitute an optimal policy given the resulting state variable values. From the analysis, it is known that to load two parcels from parcel class three is the optimal load from that parcel class. Loading those two from parcel class 3 uses up 2*2=4 and 2*3=6 amounts of state variable 1 resource and state variable 2

resource respectively. Given that the initial values of NX and PY were 7, this implies that there are 7-4=3 units of state variable 1 resource left to allocate, and 7-6=1 unit of state variable 2 resouce left to allocate over both parcel classes 1 and 2. Contained in Slice(2) of MAIN are the optimal load values from parcel class two given that only parcel classes 1 and 2 are being selected from. Since parcel class two was evaluated over all possible combinations of SV1 and SV2, the entry at MAIN(3,1,2) will be the optimal amount from parcel class 2 to load given their SV1=3 and SV2=1. As shown in figure #20, this value is zero. Since no parcels from parcel class two were loaded, no state variable resource was used up. The entire value of SV1 and SV2 is carried over to be allocated to parcel class 1. As shown in figure #14, the optimal amount to load from parcel class 1 is located at MAIN(3,1,1). The only subscript that changes is the slice identifier. The optimal value at MAIN(3,1,1) is 1. Contained in table #4 is a listing of the optimal load policy as well as the contribution to the total system utility per parcel class.

PARCEL CLASS	OPTIMAL LOAD	UTILITY CONTRIBUTION
1	1	4
2	0	0
3	2	14

TOTAL SYSTEM UTILITY = 18

TABLE #4 - OPTIMAL SOLUTION

In summation, it can be seen that the basis for the vector processing approach is the ability to perform successive vector operations on the array BUFF. Vector Processing is most efficient when applied to column vectors within arrays that represent contiguous computer elements. The algorithm was able to find contiguous elements of Slice(4) that were added in a vector fashion to a constant utility value and stored in contiguous array elements. It is the carrying over of vector structure to all aspects of the algorithm that allows for the application of vector processing to solve the pallet loading problem more efficiently.

CHAPTER 4

Based upon the algorithm developed in chapter 3, a computer program was written which incorporates the vector processing approach to Dynamic Programming. A general sequence of actions performed by the program is shown in figure #22. For a more detailed flowchart, please reference Appendix B.

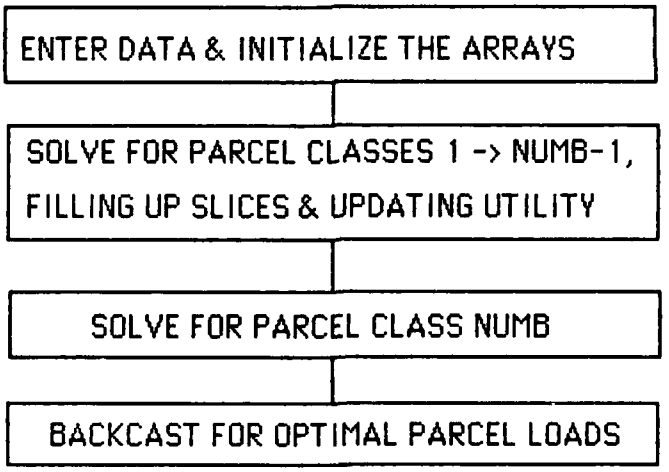


FIGURE #22 - GENERAL ALGORITHM FLOWCHART

A small sample problem will be solved using the computer program based on the algorithm developed in chapter 3. The algorithm will then be

applied to the specific Kelly AFB pallet loading problem of differing priorities. Also, an analysis of the sensitivity of the CPU running time of the program to changes in the following load parameters will be conducted: changes in the maximum load parameters of the pallet, changes in the maximum number of parcels that can be loaded from a parcel class(MN), and changes in the total number of parcel classes from which parcels will be loaded(NUMB). This sensitivity analysis will be very helpful in making projections into the future of the usefulness and extendibility of the program. Finally, a breakdown of which sections of the computer code account for various percentages of the total program CPU time will show where future effort should be spent in streamlining the program. Conclusions based on the experimental results presented in this chapter will be summarized and presented in chapter 5.

The small sample problem consists of a pallet of weight and volume limits of 35 lbs and 35 cu. ft. respectively. There are boxes from 5 different parcel classes to be loaded onto the pallet. Referring back to figure #8 in chapter 3, an array with Fortran dimensions (0:36,1:35,0:7) will be created by the program. This corresponds to an array with 8,820

elements to hold all the necessary dynamic programming information.

The parcel classes to be loaded have the following parameters:

PARCEL CLASS	WEIGHT	VOLUME	UTILITY	NUMBER TO LOAD
1	3	4	5	3
2	5	3	8	3
3	7	7	12	3
4	4	5	3	3
5	2	2	3	1

TABLE #5 - SAMPLE PROBLEM LOAD PARAMETERS

The parameters were input into the computer program and the following solution was found at a running time of 1.508 CPU seconds, with a maximized system utility of 73:

PARCEL CLASS	1	2	3	4	5
OPTIMAL # TO LOAD	2	3	0	3	1
PART OF MAX UTILITY	10	24	0	36	3

TABLE #6 - SOLUTION TO SAMPLE PROBLEM

To verify that the computer program and associated algorithm did indeed work and obtain the optimal load combination, several sample problems were run using the program. The same load parameters were then input into an integer programming code, and the results were compared. While in some cases, different load combinations were derived due to alternative forms of the optimum, in all cases the total system utilities for all sample problems were the same whether solved using the vector DP program or the integer code. For example, in our specific sample case, the maximized utility of the problem using an integer programming code was 73, with the same parcel load being selected. Once that the computer program has been verified as to solution deriving capability, it can be used to solve the large Kelly AFB problem.

Referring back to chapter 1, the Kelly AFB problem can be described as attempting to optimally load a pallet with maximum load capabilities of 729 cu. ft. and 10,000 lbs. In the sample data supplied by the Freight office, there were 50 parcels to be loaded each with a corresponding weight, volume, utility, and priority. Since initially no parcel is any more desirable than any other, the utility for all parcels is 1.

There are 30 parcels of priority 1 and 20 parcels of priority 2 in the queue. With this case, the problem can be broken down into attempting to load any and all priority one boxes first, then attempting to load priority 2 parcels in any remaining weight and volume capacity.

To solve the problem as is, the computer program would set up an array of Fortran dimensions (0:10,001,1:729,0:32). This would correspond to an array with 233,303,328 elements. Computers with this much memory available are only know being perfected and are infeasible for the solution of this problem. In an attempt to pare down the memory required to the feasible range, the parcel weight levels will be grouped in 10 pound increments. Each ten pound increment represents an interval into which an actual weight value could fall. The state variable 1 parameter for a given parcel will be determined by the 10 lb weight increment which encompasses it. In this application, the intervals start at five and ascend in the prescribed 10 lb increments, i.e., 5->15,15->25 etc. For any parcels of weight less than 5 lbs, the value of 1 will be assigned. The new values for weight assigned to each parcel represent 10 lbs per level, not 1 lb per level as before. This process of aggregating the weights into intervals is

known as "discretization".

Given this problem formulation, the Fortran dimensions of the program array will be (0:1001,1:729,0:32). The array created will have 23,351,328 elements. Even at this much smaller array size, the problem size is far from trivial. Consider the fact that the main University of Texas computer, a CDC Cyber 170/750, has a memory capability of 200,000 words. Even though most computers are not capable at this time of storing arrays of this size, the unique memory paging capabilities of the CDC Cyber 205 allow for the easy and efficient use of its virtual memory capability. The following table lists the weight(each integer actually represents 10 lbs), volume, priority, and load decision.

PARCEL	WEIGHT(10LBS)	VOLUME(cu. ft.)	OPTIMAL LOAD VALUE
1	1	1	1
2	1	1	1
3	2	4	1
4	2	2	1
5	5	5	1
6	3	5	1
7	2	1	1
8	4	6	1
9	1	1	1

10	1	1	1
11	1	1	1
12	2	4	1
13	1	2	1
14	1	6	1
15	5	4	1
16	4	4	1
17	4	7	1
18	2	4	1
19	1	1	1
20	2	2	1
21	220	63	1
22	28	85	1
23	37	51	1
24	17	33	1
25	100	40	1
26	32	28	1
27	100	41	1
28	33	16	1
29	24	38	1
30	20	37	1

TABLE #7 - KELLY AFB PRIORITY 1 LOAD LIST AND SOLUTION

For the data supplied by the transport office, the computer program arrived at the solution shown in table #7 in 4.493 CPU seconds. The solution is obvious in that the summed weight and volume for all priority 1 parcels are less than the maximum load limits of the pallet. This implies that all priority 1 parcels are capable of being loaded. Even though the solution result was obvious, the fact that a problem of 30

variables and array size of over 23 million words was solvable is very important indeed. Given this initial result, the second stage is to attempt to load any priority 2 parcels into the remaining weight of 1,000 - 656 = 344 and remaining volume of 729 - 493 = 236. Since the summed weights and volumes of all priority 2 parcels are 635@10 lbs and 670 cu. ft. respectively, the solution of the optimal priority 2 load will not be as simple. To solve for the 20 priority two parcels, the computer program would set up an array with Fortran dimensions of (0:345,1:236,0:22) containing 1,791,240 elements. At this point, the utility for each parcel is set at 1 because no parcels are more valuable than any other. Since there are not enough state variable resources available to load all priority 2 parcels, some will be left for possible inclusion in the next attempted load. These remaining parcels will have an upward adjustment of their parcel utility to ensure their timely inclusion into following load shipment. Table #8 list the priority 2 parcel weights, volumes, and optimal load number.

PARCEL #	WEIGHT(10 lbs)	VOLUME(cu. ft.)	OPTIMAL LOAD NUMBER
1	2	2	1
2	1	6	1
3	3	3	1
4	1	1	1
5	1	1	1
6	5	3	1
7	1	1	1
8	2	4	1
9	2	5	1
10	2	3	1
11	77	60	1
12	93	35	1
13	38	75	0
14	53	57	1
15	25	70	0
16	9	42	1
17	24	57	0
18	69	57	0
19	113	86	0
20	114	85	0

TABLE #8 - PRIORITY 2 LOAD LIST AND OPTIMAL LOAD

The numbers in the right hand column show which parcels from parcel class 2 would be loaded. To reverify the program for this solution is an unnecessary task. However, the reader may find it interesting to swap in and out unloaded and loaded parcels and observe that no higher utility can be obtained. The solution was found in 1.781 CPU seconds. The amount of weight and volume available for allocation after the priority 2 parcels were loaded is 94 @10 lbs and 15 cu.ft. If there had been any priority 3 parcels in the queue, an attempt to optimally load them into this

remaining space would be made. At this point, an optimal load has been determined for shipment. Since this problem solved for an initial queue, assuming no waiting parcels, all the parcel utilities were the same, set at a level of 1. However, now that some parcels will have been passed over, the upward utility adjustment differentiates passed over parcels from newly arrived parcels. In this manner, they will have special preference for inclusion into the next load. This process of determining the optimal load from a given parcel queue of priority 1, 2 , and 3 parcels and the subsequent utility update constitute the Kelly AFB loading procedure.

Now that the Kelly AFB problem has been solved, it is important to check the sensitivity of the program to changes in the various load parameters. In particular, CPU time changes will be checked in response to changes in the maximum load parameters of the loading platforms, and changes in the maximum number to load from each class. The sample problem that will be used as a test case will consist of thirty six parcels classes with load parameters that were specificaly designed to incorporate all possible types of parcels. For example, high utility coupled with low weight and volume, low utility coupled with high weight and

volume, etc. The following table shows associated load parameters except maximum number to load which will be set separately for each run:

PARCEL #	S1 VALUE	S2 VALUE	UTILITY
1	3	3	1
2	3	6	6
3	3	9	2
4	3	12	5
5	3	15	3
6	3	18	4
7	6	3	6
8	6	6	1
9	6	9	5
10	6	12	2
11	6	15	4
12	6	18	3
13	9	3	1
14	9	6	6
15	9	9	2
16	9	12	5
17	9	15	3
18	9	18	4
19	12	3	6
20	12	6	1
21	12	9	5
22	12	12	2
23	12	15	4
24	12	18	3
25	15	3	1
26	15	6	6
27	15	9	2
28	15	12	5
29	15	15	3
30	15	18	4
31	18	3	6
32	18	6	1
33	18	9	5
34	18	12	2
35	18	15	4
36	18	18	3

TABLE #9 - TEST PROBLEM LOAD PARAMETERS

CPU TIME vs. CHANGES IN MAXIMUM LOAD PARAMETERS

A change in a load parameter corresponds to a change in either the maximum weight or volume allowed on the pallet. With regards to the program, changes in these parameters change the amount of computer memory necessary for solution. For example, if the problem has 36 parcel classes, a weight max of 100 lbs and a volume limit of 300 cu. ft. , then the array with dimensions (0:101,1:300,0:38) would use up 1,154,400 words of memory. If the weight limit were increased to 200 lbs., the new memory use would be 201*300*38 = 2,291,400 words. Therefore, changes in maximum pallet parameter will be described as changes in the amount of memory used up by a problem. For the sample problem where the maximum number to load for all parcel classes is 4 items, the following graph shows the relationship between CPU time and memory used in the array MAIN.

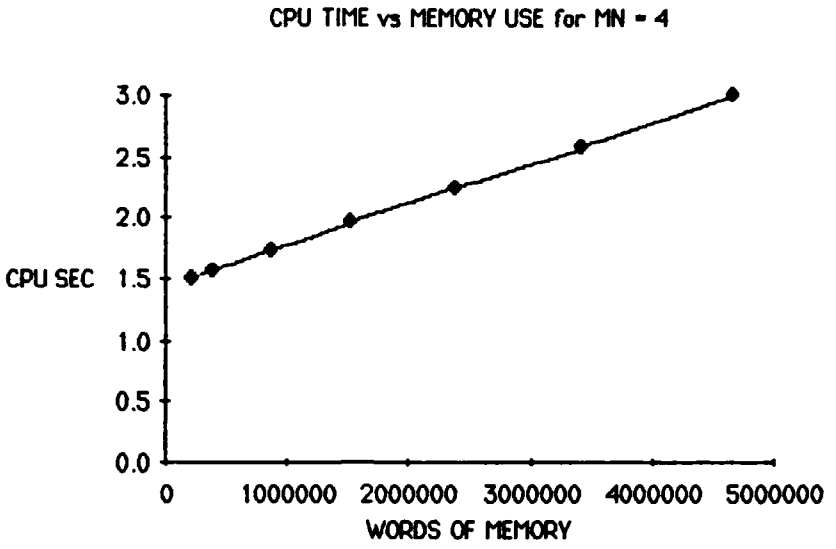


FIGURE #23 - CPU TIME vs. CHANGES IN PALLET PARAMETERS

As can be seen visually in the graph, the relationship is linear, and slightly increasing. A simple regression(see Appendix A) performed on the data points finds an almost perfect linear relationship between the two variables. This implies that as memory used goes up, there is a very slight linear increase in the amount of CPU time to run the problem. For example, according to the linear regression analysis, if the array size used was 20,000,000 words, this would imply a CPU time of only 8.1567 seconds.

CPU TIME vs. CHANGES IN MAXIMUM NUMBER TO LOAD

Within each parcel class, there is a parameter that specifies how

many parcels there are within each parcel class available for loading. Changes in this parameter dramatically effects the amount of calculations to be made, as can be remembered from the analysis in chapter 3. Using the sample problem of 36 parcel classes, the maximum number to load for each parcel class will be changed in increments of one starting at 1 and ending at 8. The pallet load parameters of NX=250 and PY = 250 will be held constant. The following graph shows the relationship between CPU time used to solve the problem and the maximum number to load specified:

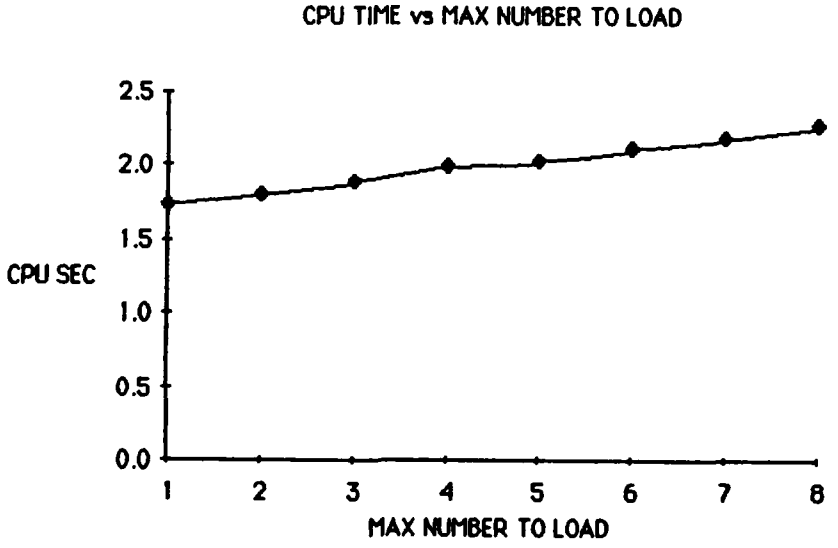


FIGURE #24 - CPU TIME vs. CHANGES IN MAX NUMBER TO LOAD

As was the case for the previous sensititvity, the relationship between the two parameters is of a linear nature. This imples that even

as the maximum number to load is increased to a large number, the corresponding amount of CPU time used will be small based on the linear relationship. A simple regression performed on the data(see Appendix A) shows an almost perfect linear relationship between the two variables. According to the regression, if the maximum number to load from each parcel class was raised to 30, this would imply a CPU time of only 3.94616 seconds. Integer programming on the other hand, experiences an almost exponential increase in CPU time if the maximum number to load were increased(13).

CPU TIME vs. CHANGE IN THE NUMBER OF PARCEL CLASSES

A parcel class represents a stage of the dynamic programming problem formulation. By raising the number of parcel classes, the number of stages of the dynamic program is raised. For comparison purposes, each parcel class represents a variable of the integer programming formulation. The sensitivity analysis for changing the number of parcel classes was performed using NX = 250, PY = 250, MN =4, and the load parameters of the 36 parcels of sample problem 2. The number of parcel classes attempted

to load was varied from 4 to 36 and the following graph was derived:

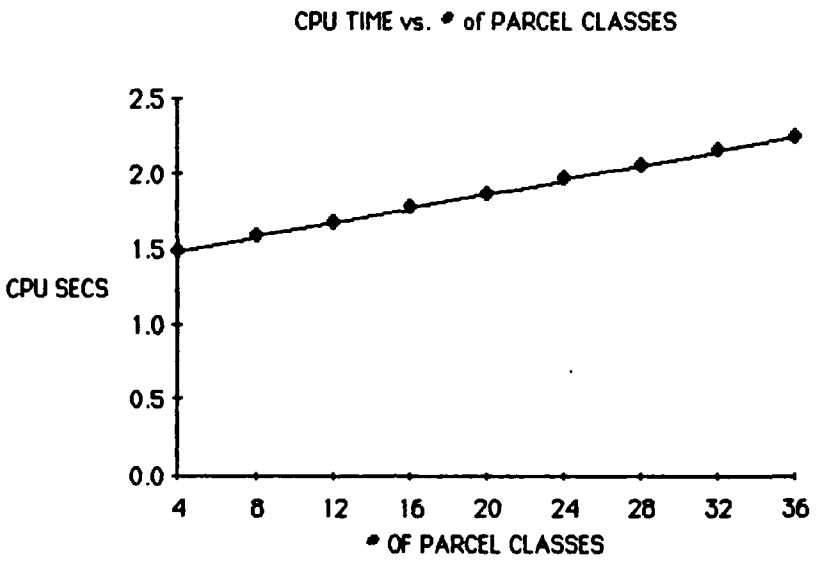


FIGURE #25 - CPU TIME vs. CHANGE IN # OF PARCEL CLASSES

As can be seen from the graph, the relationship between the two variables is linear. This fact is corroborated by the results of a simple linear regression(see Appendix A) performed on the data. This analysis implies that as the number of parcel classes is increased, there will only be a linear increase in the amount of CPU time used. For example, if a problem of 100 parcel classes were solved, according to the linear regression, the CPU time required would be 3.7538 seconds. This linearly increasing computational time is compared to an exponentially increasing

computational time that the integer programming method shows as the number of variables is increased(13).

CPU TIME BREAKDOWN OF COMPUTER PROGRAM

As shown in appendix B, the computer program is broken down into roughly 4 segments which are: the initialization segment, the solution of parcel classes 1 to Numb-1, the solution of parcel class Numb, and the backcasting and output of the solution. Based on CPU timers placed in between segments of the code, the following computation time breakdown has been determined (units of time are "system time units"):

	TIME FOR COMPLETION OF	% OF TOTAL TIME
SEGMENT 1 (initialization)	.0107	4.85
SEGMENT 2 (PC 1 to numb-1)	.2014	91.38
SEGMENT 3 (solve class numb)	.0002	.09
SEGMENT 4 (solution, output)	.0082	3.72
TOTAL TIME	= .2204 STU	100.04%

TABLE #10 - CPU TIME BREAKDOWN

As the graph shows, the majority of the computational time is spent in segment 2 of the program. In segment 2, the computer is filling the optimal policy tableaus for parcel classes 1 to numb-1. The time is so high because at each one of these stages, the computer is solving for every possible state variable combination element. In segment 3, which uses almost no time, the computer is only solving for one column of elements.

In the final chapter of this report, all the computational results and findings will be summarized as to the extendibility and usefulness of the vector programming approach to dynamic programming.

CHAPTER 5

CONCLUSIONS

Based on the results and findings from chapter 4, it is shown that the vector processing approach to dynamic programming and the pallet loading problem is a success. The column nature of the dynamic programming matrices are a perfect match for the column manipulating power of vector processing. In fact, based on the CPU run times for the problems solved in chapter 4, the vector approach to dynamic programming is a very efficient solution technique indeed. The power of the CDC Cyber 205 vector processor is such that approximately 140,000 vector operations of length 200 elements can be performed in 2.268 CPU seconds.

The main drawback of this technique, the large amount of memory required to solve large problems is in the process of being solved at this time. New supercomputers such as the ETA 10 which will have a shared memory capability of 296 million words of memory are on order at several universities at the present time. In the near future, gains in computer memory technology will be such that for almost all applications, memory

limit will no longer be a constraint.

The vector processing algorithm trades off the use of large chunks of memory for a low CPU running time. This is the most efficient means of computing. The memory use for that small amount of time the program is running is very inexpensive compared to the cost of an additional second of CPU time used on a supercomputer. In many cases, the memory use is virtually free given that the disk space is available.

The sensitivity analysis from chapter 4 showed that for all possible changes in problem parameters, the resulting change in CPU running time will be linear, a characteristic that not many optimization techniques can boast. The linear regression analysis of the data presented for changes in CPU time verses changes in load parameters shows that the slope of the relationship lines are very small. This implies that large increases in the load parameters will result in a relatively small increase in expensive CPU time.

As to the Kelly AFB problem, the method specified in chapter 4 shows a feasible way of optimally loading the pallets while maintaining a check on the amount of loads that have passed a given parcel by. With the

advent of larger memory machines, the pared down problem can be solved in full. To actually implement a feasible loading algorithm for the Kelly AFB freight office, an extension of the developed two state variable problem is necessary. The new algorithm needs to take into account the individual heighth, length, width, and weight of each parcel. This implies looking at a 5-dimension solution method. Based on interviews with the management of the Kelly AFB Transport Terminal, their ultimate desire is a computer program that given the weight, length, width, heighth, and priority of a parcel, determine whether or not that parcel belongs in an optimal load, and if so, what is the exact positioning of that parcel on the pallet with respect to the other loaded parcels. This goal leads to the extensions portion of this chapter.

EXTENSIONS

The most difficult extension to this vector processed dynamic programming algorithm will be the addition of more state variables. The Fortran 200 in which the algorithm was programmed allows for arrays of

up to seven dimensions. Given that one dimension is required to keep track of the stages, theoretically up to six state variables could be considered. An extension from the two state variables of the present algorithm to three state variables would be a major step.

The utility function specified by the general pallet loading problem is of a linear nature consisting of a parcel utility parameter multiplied by the number of that parcel class to load. Not all problems have linear utility functions. The algorithm developed solves the general dynamic programming problem. Given this basis, the utility function can be changed to accomodate the utility function of any specific problem whether linear or non-linear as long as it falls in the general DP format. This implies that the program can be extended to solve the whole class of general Dynamic Programming problems as they arise.

A final extension of the program is in the form of streamlining. The CPU time breakdown presented in chapter 4 shows that most of the CPU time is spent in segment 2 of the program(see Appendix B). In fact, according to the analysis, over 90% of the CPU time is spent solving for the slices of MAIN that contain the load information for parcel classes 1

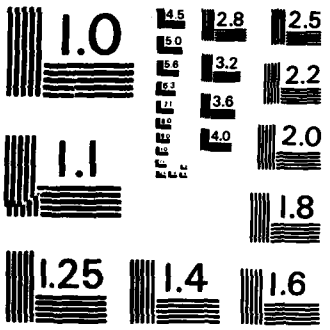
to NUMB-1. If any significant reductions in the CPU time consumed in this segment can be made based on new and better programming, this would contribute to a significant reduction in overall program CPU time. In fact, since segment 2 accounts for such a high percentage of the overall CPU time, a 50% reduction in segment 2 CPU time would result in a 46% decrease in total program time.

In summary, it is shown that vector processing is a feasible and efficient method to solve the general pallet loading problem, and dynamic programming models in general. It solves the problems in the most efficient computational sense and with the changing utility function is applicable to a wide range of applications. The most significant extension will be the addition of an extra state variable as the first step toward realizing the ultimate goal of the Kelly AFB Freight Transport Office.

APPENDIX A

SENSITIVITIES DATA REGRESSION ANALYSIS

AD-A166 763 SOLVING MULTI-STATE VARIABLE DYNAMIC PROGRAMMING MODELS 2/2
USING VECTOR PROCESSING(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH S W STOPKEY DEC 85
UNCLASSIFIED AFIT/CI/NR-86-45T F/G 9/2 NL



DATA AND REGRESSION FOR CPU TIME vs. MAX LOAD PARAMETERS

Contained in table 11 is computer run information used to derive figure #22. The test problem of 36 parcels and a value of MN = 4 for every parcel were the data inputs. The NX and PY parameters were varied and this produced the changing memory requirements.

NX	PY	MAIN ARRAY SIZE(words)	CPU TIME(seconds)
75	75	213,750	1.493
100	100	380,000	1.557
150	150	855,000	1.734
200	200	1,520,000	1.963
250	250	2,375,000	2.245
300	300	3,420,000	2.582
350	350	4,655,000	2.995

TABLE #11 - FIGURE #22 DATA

A regression analysis of CPU time verses array size provided the following results:

(a) Correlation Coefficient = .999788

(This high level indicates an almost perfect linear relationship)

(b) Regression equation developed is:

CPU(seconds) = 1.43671 + .0000003362 * ARRAY SIZE(words)

DATA AND REGRESSION FOR CPU TIME vs. CHANGES IN "MN" VALUE

Table #12 contains the computer run data that was used to derive figure #23. The 36 parcel classes of the test problem from chapter 4 were used with NX = PY = 200(array size of 1,520,000 words).

MN	CPU TIME(seconds)
1	1.728
2	1.805
3	1.880
4	1.995
5	2.028
6	2.114
7	2.184
8	2.268

TABLE #12 - FIGURE #23 DATA

A regression analysis of CPU time verses level of MN provided the following result:

(a) Correlation Coefficient = .9973

(This high level indicates an almost perfect linear relationship)

(b) The regression equation is as follows:

CPU TIME(seconds) = 1.65685 + .07631 * (MN)

DATA AND REGRESSION FOR CPU TIME vs. # OF PARCEL CLASSES

Table #13 shows the data that was used to dereive figure #24. The parcel classes from the Test problem were used for the analysis, starting with 4 and adding 4 more per run up to a total of 36 parcel classes. Other parameters were NX = PY =250, MN =4.

# OF PARCEL CLASSES	CPU TIME(seconds)
4	1.494
8	1.588
12	1.682
16	1.779
20	1.870
24	1.965
28	2.057
32	2.155
36	2.247

TABLE #13 - FIGURE # 24 DATA

A regression analysis of the CPU time verses the number of parcel classes evaluated had the following results:

(a) Correlation Coefficient = .9999866

(This high level implies an almost perfect linear relationship)

(b) The regression equation is as follows:

CPU TIME(seconds) = 1.4000278 + .0235375 * PARCEL CLASSES

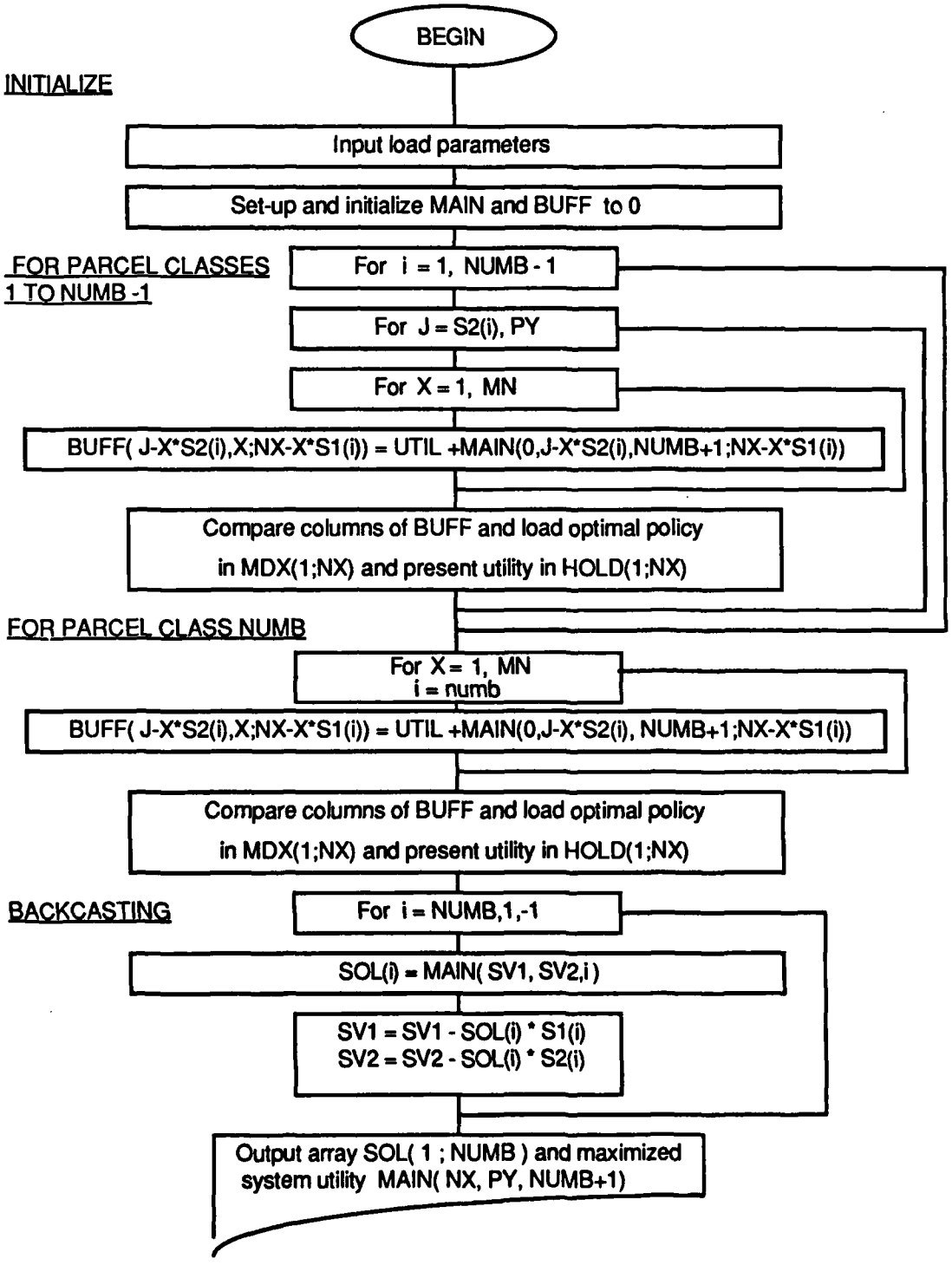
APPENDIX B

GENERAL ALGORITHM FLOWCHART

FORTRAN COMPUTER CODE

SAMPLE OUTPUT FOR PRIORITY 2 PROBLEM

GENERAL ALGORITHM FLOWCHART



```
      PROGRAM  VAP(INPUT,OUTPUT,TAPE6)
*****************************************************************
**     THIS PROGRAM SOLVES THE M-STAGE,N-STATE FLY-AWAY KIT  *)
**  PROBLEM USING VECTOR ARRAY PROCESSING.  ALL VALUES ARE   *)
**  OF TYPE INTEGER.  THE DATA INPUT IS HANDLED THROUGHT THE *)
**  USE OF DATA STATEMENTS SINCE THIS IS THE MOST EFFICIENT  *)
**  MODE OF INPUT FOR THE CYBER 205.                         *)
*****************************************************************
**
**     THIS IS WHERE SEGMENT ONE BEGINS - INITIALIZATION
**
*****************************************************************

      INTEGER   PC,P2,P1,UTIL,P3,PY,BF
      INTEGER MAT
      PARAMETER  (NX=344,PY=236,NUMB=20,P3=NUMB+1,MNT=3)
      PARAMETER  (NX1=NX+1,P31=P3+1,K65=65535)
      PARAMETER  (MF=NX1*PY,BF=MNT*NX)
      INTEGER   HOLD(NX),SOL(NX),MDX(NX),S1(NUMB),S2(NUMB),U(NUMB),
     1  MN(NUMB),BUFF(1:NX,0:MNT+1)
      COMMON/BLKM/ MAT(0:NX+1,PY,0:P3+1)
      INTEGER   MATL(MF)
      EQUIVALENCE (MATL(1),MAT(0,1,P3))

**  THESE ARE THE DATA STATEMENTS THROUGH WHICH
**  NEW PROBLEM PARAMETERS ARE LOADED


**  THESE PARTICULAR PARAMETERS ARE FOR THE PRIORITY TWO LOAD
**  FROM STUART W. STOPKEY'S MASTERS REPORT.

      DATA S1/2,1,3,1,1,5,1,2,2,2,77,93,38,53,25,9,24,69,113,114/
      DATA S2/2,6,3,1,1,3,1,4,5,3,60,35,75,57,70,42,57,57,86,85/
      DATA U/20*1/
      DATA MN/20*1/

      WRITE(6,*) 'OUTPUT FOR DP GENERAL LOADING PROBLEM'
      WRITE(6,*) '    '
      WRITE(6,*)  'AMOUNT OF STATE VARIABLE 1 IS ',NX
      WRITE(6,*)  'AMOUNT OF STATE VARIABLE 2 IS ',PY
      WRITE(6,*)  'NUMBER OF STAGES IS ',NUMB
      WRITE(6,*)  '    '
```

```
*********************************************************************
**       THIS SECTION OF CODE INITIALIZES SLICE(NUMB+1) OF
**  ARRAY MAIN TO ZERO.  IT IS FORMULATED AS SUCH TO ALLOW FOR
**  VECTOR LENGTHS OF OVER 65,535 ELEMENTS TO BE FILLED.
*********************************************************************

      ICHX= (MF-1)/K65
      IF (ICHX.NE.0) THEN
        ID = MOD(MF,K65)
        ISTAR = 1
        DO 2 I=1,ICHX
          ISTAR = 1 + (I-1)*K65
 2      MATL(ISTAR;K65) = 0
        ISTAR = ISTAR +K65
         IF (ID.NE.0) THEN
          MATL(ISTAR;ID) = 0
       ELSE
         MATL(ISTAR;K65) = 0
        ENDIF
      ELSE
        MFF=MF
       MAT(0,1,P3;MFF) = 0
      ENDIF

*********************************************************************
**     MAIN PART OF PROGRAM THAT CALCULATES THE PRESENT UTILITY *)
**  AND THE OPTIMAL POLICY AT EACH STAGE.  USES STOPKEY'S METHOD *)
**  OF VECTOR ANALYSIS.                                          *)
*********************************************************************
**     THIS LOOP CALCULATES THE OPTIMAL POLICY AND UTILITY FOR
**  EVERY PARCEL CLASS FROM 1 TO NUMB-1
*********************************************************************
**
**       THIS IS WHERE SEGMENT TWO BEGINS
**
*********************************************************************

      DO  30  I = 1,NUMB-1
*********************************************************************
**     THIS LOOP FILLS IN THE COLUMNS OF THE MATRIX WITH MAXIMUM
**  UTILITIES MOVING IN ASCENDING ORDER ALONG THE STATE VARIABLE
**  TWO INCREMENTS.  EACH COLUMN STARTS AT MAT(0,J,1)
*********************************************************************
```

```
      DO 20 J = S2(I),PY
          BUFF(1,1;BF) = 0
          BUFF(1,0;NX) = MAT(1,J,P3;NX)

          DO 15 K=1,MN(I)
          PC= K*S1(I)
          P2= J - K*S2(I)
          P1= NX - PC
          UTIL= K*U(I)

          IF (P2.EQ.0) THEN
             BUFF(PC,K;P1+1) = UTIL
           ELSE IF (P2.GT.0) THEN
             BUFF(PC,K;P1+1) = UTIL+ MAT(0,P2,P3;P1+1)
          END IF

15        CONTINUE

        MDX(1;NX) = 0
        HOLD(1;NX) = BUFF(1,0;NX)

         DO 18 K = 1,MN(I)

**********************************************************************
**     AT THIS POINT, VALUES IN THE COLUMNS OF ARRAY "BUFF" ARE
**  COMPARED, WITH THE MAXIMUM VALUE FOR EACH ROW AND THE COLUMN
**  IN WHICH IT WAS FOUND STORED FOR LATER TRANSFER INTO "MAT"
**********************************************************************
            WHERE (BUFF(1,K;NX).GT.HOLD(1;NX))

              HOLD(1;NX) = BUFF(1,K;NX)
              MDX(1;NX) = K
          END WHERE
18       CONTINUE

       MAT(1,J,I;NX) = MDX(1;NX)
20      MAT(1,J,0;NX) = HOLD(1;NX)

       DO 22 J = 1,S2(I)-1
22        MAT(1,J,I;NX) = 0

     DO 25 J= S2(I),PY
25   MAT(1,J,P3;NX) = MAT(1,J,0;NX)

30     CONTINUE
```

```
**********************************************************************
**     THIS SECTION IS THE SAME CODE AS THE PREVIOUS SECTION BUT
**  PERFORMS THE CALCULATION FOR ONLY THE LAST COLUMN OF THE LAST
**  PARCEL CLASS.
**********************************************************************
**
**       THIS IS WHERE SEGMENT THREE BEGINS
**
**********************************************************************

      BUFF(1,0;NX) = MAT(1,PY,P3;NX)
      BUFF(1,1;BF) = 0
      DO 40 K= 1,MN(NUMB)

        PC= K*S1(I)
        P2= PY - K*S2(I)
        P1= NX - PC
        UTIL= K*U(I)

        IF (P2.EQ.0) THEN
           BUFF(PC,K;P1+1) = UTIL
         ELSE IF (P2.GT.0) THEN
           BUFF(PC,K;P1+1) = UTIL + MAT(0,P2,P3;P1+1)
        END IF

 40     CONTINUE
        MDX(1;NX) = 0
        HOLD(1;NX) = BUFF(1,0;NX)

           DO 35 L= 1,MN(NUMB)

             WHERE (BUFF(1,L;NX).GT.HOLD(1;NX))

               HOLD(1;NX) = BUFF(1,L;NX)
               MDX(1;NX) = L
             END WHERE
 35        CONTINUE

        MAT(1,PY,NUMB;NX) = MDX(1;NX)
        MAT(1,PY,P3;NX) = HOLD(1;NX)
      M = PY
      N = NX
```

```
*****************************************************************
**     THIS IS THE SECTION OF CODE WHERE THE BACKCASTING FOR
**  THE SOLUTION AND SOLUTION OUTPUT TAKES PLACE.  A TRACEBACK
**  THROUGH EACH SLICE OF "MAT" IS DONE WITH THE INDICES BEING
**  THE AMOUNT OF EACH STATE VARIABLE REMAINING AND THE PARCEL
**  NUMBER TO DETERMINE THE OPTIMAL # OF EACH PARCEL TO LOAD.
*****************************************************************
**
**      THIS IS WHERE SEGMENT FOUR BEGINS
**
*****************************************************************

      WRITE(6,*) 'SOLUTION STATEMENT FOR PALLET LOADING PROBLEM'
      WRITE(6,*) 'STATE 1 VAL IS ',NX,'  STATE 2 VAL IS ',PY
      WRITE(6,*) '    '
      WRITE(6,*)  'THE MAXIMIZED UTILITY IS  ',MAT(NX,PY,P3)
      WRITE(6,*) 'PARCEL SOL #  S1 LEFT  S2 LEFT'

      SOL(NUMB) = MAT(NX,PY,NUMB)

        DO 50 I = NUMB,2,-1
          N = N - SOL(I)*S1(I)
          M = M - SOL(I)*S2(I)
          WRITE(6,*) I,'       ',SOL(I),'       ',N,'       ',M

       IF (N.LE.0.OR.M.LE.0) THEN
           SOL(I-1)=0
       ELSE
          SOL(I-1) = MAT(N,M,I-1)
        END IF
50      CONTINUE

      WRITE(6,*) I,'       ',SOL(1),'       ',N-SOL(1)*S1(1),
     1    '       ',M-SOL(1)*S2(1)



         END
```

OUTPUT FOR DP GENERAL LOADING PROBLEM

AMOUNT OF STATE VARIABLE 1 IS 344
AMOUNT OF STATE VARIABLE 2 IS 236
NUMBER OF STAGES IS 20

SOLUTION STATEMENT FOR PALLET LOADING PROBLEM
STATE 1 VAL IS 344 STATE 2 VAL IS 236

THE MAXIMIZED UTILITY IS 14

PARCEL	SOL #	S1 LEFT	S2 LEFT
20	0	344	236
19	0	344	236
18	0	344	236
17	0	344	236
16	1	335	194
15	0	335	194
14	1	282	137
13	0	282	137
12	1	189	102
11	1	112	42
10	1	110	39
9	1	108	34
8	1	106	30
7	1	105	29
6	1	100	26
5	1	99	25
4	1	98	24
3	1	95	21
2	1	94	15
1	1	92	13

BIBLIOGRAPHY

1 Beightler, C.S., Foundations of Optimization, Prentice-Hall, 1979.

2 Bellman, R.E., & Dreyfus, Stuart E., Applied Dynamic Programming, Princeton University Press, 1962.

3 Casti, J.R., "Dynamic Programming and Parallel Computation", Journal of Optimization Theory, vol 12, 1973, pp 423-438.

4 Control Data, Cybernet Services Cyber 205 Service Efficient Fortran Technique Users Guide, Pacific-Sierra, 1982.

5 ETA Systems, Symposium on Cyber 205, University of Texas - Austin, June 12-14, 1985.

6 Fisk, John C., "A Heuristic for Solving Large Loading Problems", Naval Research Logistics Quarterly, vol 26, 1979, pp. 643-650.

7 Garza, Rudolpho, The Use of Vector Processing to Solve a Class of Linear Programming Problems, Phd. Dissertation, M. E. Dept., University of Texas at Austin, 1981.

8 Gilmore, P.C., & Gomory, R.E., "A Linear Programming Approach to the Cutting Stock Problem", Operations Research, vol 11, 1963, p. 863.

9 Gilmore, P.C., & Gomory, R.E., "Multi-stage Cutting Stock Problems of 2 or More Dimensions", Operations Research, vol 13, 1965, p.94.

10 Gilmore, P.C., & Gomory, R.E., "Theory and Computation of Knapsack Functions", Operations Research, vol 14, 1966, p. 1045.

11 Gilmore, P.A., "Structuring of Parallel Algorithms", Journal of the Association of Computer Machinery, vol 15, 1968, pp. 176-192.

12 Hillier, F.S., & Lieberman, G.J., Introduction to Operations Research, Holden-Day, 1980.

13 Hwang, k, & Briggs, F.A., Computer Architecture and Parallel Processing, McGraw-Hill, 1984.

14 Nemhauser, G.L., Introduction to Dynamic Programming, Wiley, 1966.

15 Salkin, H.M., "The Knapsack Problem: A Survey", Naval Research Logistics Quarterly, vol 22, 1975, 127-144.

16 Sinha, Prabhakant, " The Multiple Choice Knapsack Problem", Operations Research, vol 15, 1967, p. 319.

VITA

Stuart W. Stopkey was born in Oak Park, Illinois, on May 2, 1962. Growing up in New Orleans, Louisiana, he graduated in 1980 from East Jefferson High School. That same year he enrolled at the United States Air Force Academy in Colorado Springs, Colorado from which he graduated on May 30, 1984. Upon graduation, Stuart was commissioned a Second Lieutenant in the United States Air Force, and received a Bachelor of Science degree with majors in Operations Research and Economics. Stuart was awarded a Fellowship to study at the University of Texas at Austin Operations Research branch of the Mechanical Engineering department which he accepted. He began his studies there in September of 1984, and graduated in December of 1985 with a Master's of Science in Engineering. In February of 1986, Stuart began USAF jet pilot training at Williams AFB, Pheonix, Az.

Permanent Address: 1405 Moss lane

River Ridge, La. 70123

END

DTIC

5-86