VIRTUAL MEMORY SUPPORT FOR BRANCH AND BOUND ALGORITHMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Chee Fen Yu

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

May 1983

this is dedicated

to my mom, dad, sister and Wui Foong

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Professor Benjamin W. Wah for his guidance, advice and support. I also wish to thank my committee members, Professor K. Hwang and Professor K. S. Fu.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Chee Fen Yu, Purdue University. May 1983. Virtual Memory Support for Branch and Bound Algorithms. Major Professor: Professor Benjamin W. Wah.

The branch and bound algorithm is an organized and intelligently structured search of solutions for enumerative type problems such as NP-complete problems. The best first search criterion always expands the subproblem with the smallest lower bound. This minimizes the time expended by the algorithm but has a worst case exponential space requirement. Since the size of main memory is usually limited some of the subproblems must be moved to secondary storage and staged into main memory when referenced.This staging process may be maintained by a virtual memory system.

The use of a locality based virtual memory system is inadequate for branch and bound algorithms as the algorithms do not exhibit strong locality. It is not apparent how the amount of locality exhibited may be enhanced. The alternative is to adapt the virtual memory system to the algorithms. Simulations indicate that for branch and bound algorithms such a system may achieve a seven to nine-fold improvement in performance over a general purpose virtual memory system with the LRU replacement algorithm.

The work reported herein was partially performed by CIDMAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation and TRW and partially under the support of NSF Grant ECS81-05968

CHAPTER 1
INTRODUCTION

Current virtual memory systems are general purpose systems which are based upon the concept of locality. Locality refers to the tendency of a program to reference in the near future neighboring pages and those pages referenced in the recent past. This has the advantage of not requiring any knowledge about the characteristics of the program being run, and is extremely robust since a program which does not exhibit any locality will still run on the system, albeit inefficiently.

An efficient virtual memory system will be one whereby the characteristics of programs match the model of program behavior assumed by the virtual memory system. For a general purpose virtual memory system, it is likely that a mismatch exists between the behavior of the program being executed and the locality model used.

The problem of reducing this mismatch has been studied with respect to certain specific applications such as matrix manipulations, sorting and database applications. We feel that there is a need to address this problem for a number of other applications such as image database, artificial intelligence and optimization problems. Most of the programs used in these applications require large amounts of memory resources and CPU time, so any improvements realized will bring about substantial benefits.

In this thesis, we propose to study ways of improving the virtual memory environment for branch and bound algorithms which are used in optimization problems.

1.1 Branch and Bound Algorithms

Many of the deterministic problems in computer science, operations research, and other application areas are NP-complete. This class of problems is characterized by a deterministic algorithm that computes a function from a countable domain into a countable range, and it generally involves the optimization of an objective function. The computation time for all known optimal algorithms for this class of problems increases exponentially with the problem size.

The common approach to solving NP-complete problems is to solve optimally for small problems and to solve sub-optimally using heuristics for large problems. The most general technique for solving a wide variety of these problems is the branch and bound algorithm

A branch and bound algorithm is an efficient algorithm to solve for problems that can be put into the form of a constrained optimization.

$$\begin{array}{ll} \text{Minimize} & C_0(x) \\ \text{subject to} & g_1(x) \geq 0 \\ & g_2(x) \geq 0 \\ & \bullet \\ & \bullet \\ & \bullet \\ & g_m(x) \geq 0 \\ \text{and} & x \in X \end{array}$$

in which X represents the domain of optimization defined by the m constraints, normally an euclidean n-space, and x denotes a vector $(x_1, x_2, \ldots, x_n)$. (Problems that are NP-complete can be put into this form. There exists problems that are not NP-complete, but are put into this form as well.) A solution vector that lies in x is called a feasible solution, and a feasible solution for which $C_0(x)$ is minimal is called an optimal solution.

The branch and bound algorithm is an organized and intelligently structured search of the space of all feasible solutions. It has been extensively studied in areas such as artificial intelligence and operations research [9, 19, 21, 22]. It has been applied to solve problems in scheduling [17, 20], knapsack [13, 14], traveling salesman [1, 5], facility allocation [2], integer programming [4, 6], and many others. Dominance relations similar to those used in dynamic programming have been used to prune search tree nodes. Theoretical properties of branch and bound algorithms have been developed in several studies [10, 12, 16, 22].

In branch and bound algorithms [19, 21], the space of all feasible solutions is repeatedly partitioned into smaller and smaller subsets, and both the lower and upper bounds are calculated for solutions within each subset. After each partitioning, subsets with lower bounds (in the case of minimization) that exceed either the value of a known feasible solution or the least upper bound of all subsets are excluded from further consideration. The partitioning process continues until a feasible solution is found such that the value is no greater than the lower bound of any subset.

The state of the partitioning process at any time can be represented as a partial tree (Figure 1). Each node in the tree represents a partition and is called a subproblem. The partitioning process selects a partition and breaks up this partition into smaller partitions. This extends the node in the partial tree representing this partition by one level and uses the sons to denote the smaller partitions. In Figure 1, node j is expanded in the partitioning process into k other partitions, which are represented as sons of node j in the partial tree.

There are two essential features of a branch and bound algorithm: the branching rule and the bounding rule. With respect to the partial tree in Figure 1, each node in the tree has two numbers associated with it - the upper bound and the lower bound of the subproblem. The leaf nodes in the partial tree are candidates for partitioning. A leaf node of the partial tree whose lower bound is less than both the value of a known feasible solution and the greatest upper bound of all leaf nodes is active; otherwise, it is terminated and need not be considered in any further computation.

The branching algorithm examines the set of active leaf nodes and, based on some predefined criterion, selects one for expansion. If the set of active nodes is maintained in a first-in first-out (FIFO) list, the algorithm is called a breadth-first search. If the set is maintained in a last-in, first-out list, the algorithm is called a depth-first search. Lastly, if the node selected for expansion is one with the minimum lower bound, the search algorithm is called a best-first search.

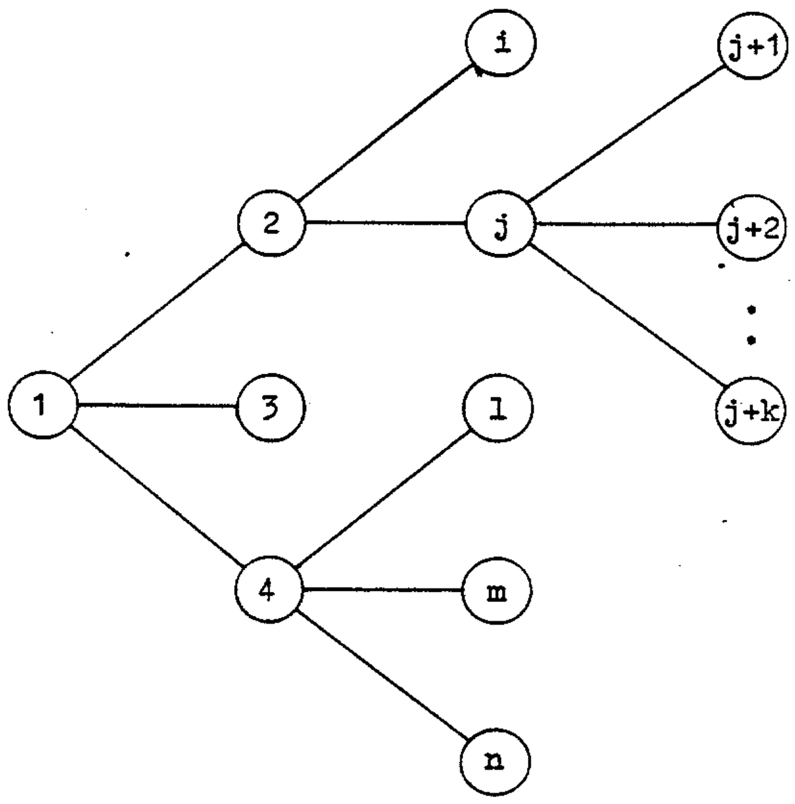


Figure 1 : A branch and bound tree

In a breadth-first search, the nodes of the tree will always be examined in levels. That is, a node at a lower level will always be examined before a node at a higher level. This search will always find a goal node nearest to the root; however, the sequence of nodes examined is predetermined, so the search is "blind." Depth-first search has a similar blind behavior except that a subtree is generated completely before the other subtrees are examined. In both algorithms, the next node to be examined is known, so the state of the parent node leading to the next node from the root node is easily found and is unique. Furthermore, the memory space required for storing the state is very small. These two algorithms are, therefore, space-saving.

In contrast, the best-first search is space-consuming because all active subproblems must be stored as intermediate data in the computer. The total number of nodes expanded, however, is minimum in the sense that any branching operation performed under this policy must also be performed under other policies, provided that all the bounds are unique [19]. Since time is a more critical factor in evaluating large optimization problems, the behavior of the best-first search merits further study.

One study [11] shows that depth-first, breadth-first and best-first searches are special cases of heuristic search. In a heuristic search, an evaluation function f(n) for a subproblem n is computed as the sum of cost of an optimal path from a given start node to n and cost of an optimal path from n to a goal. An ordered search algorithm picks up a subproblem with the minimum value of f for expansion each time. Any general heuristic functions can be included in the computation and the choice of a heuristic function depends on the application.

Once the subproblem has been selected for partitioning, some undetermined parameters in the subproblem must be selected so that alternatives for these parameters can be defined and multiple subproblems created. For example, in the traveling salesman problem, the undetermined alternatives are the set of untraversed edges. In expanding a subproblem, an untraversed edge (i,j) is selected, and two alternatives can be created: (1) the edge is traversed and the salesman goes directly from city i to city j, and (2) vice versa. The parameter chosen to be expanded is usually done in an ad hoc manner.

After new subproblems are created, the bounding algorithm is applied to evaluate the upper and lower bounds of a subproblem. Generally, only the lower bound is evaluated, and the upper bound is updated when feasible solutions are found. The bounding algorithm that is designed is highly dependent on the problem. For example, in an integer programming problem, a linear program with relaxed integer constraints can be used as a lower bound [18]; in the traveling salesman problem, an assignment algorithm [1] or a spanning tree algorithm can be used as the bounding algorithm.

As an example to illustrate the use of branch and bound algorithms, the evaluation of an integer programming problem is shown here. The integer programming problem may be expressed as

$$\begin{array}{ll} \text{Minimize} & CX \\ \text{subject to} & AX \geq B \\ & X^T = (x_1, x_2, \ldots, x_n) \\ & x_i\text{: non-negative integer, } i=1,2,\ldots n. \end{array}$$

These problems differ from ordinary linear programming problems in that the variables are restricted to non-negative integer values.

One approach to the problem is the following. Apply the dual simplex method to a subproblem. If the optimal solution is integral, a feasible solution has been generated; otherwise, create two new subproblems as follows. Choose a variable that has a non-integer value (say $x_i = 4.4$) and restrict that variable to the next lower integral value for one problem ($x_i \leq 4$) and to the next higher integral value ($x_i \geq 5$) for the other. The variable chosen is the one with the greatest up or down penalty. The up penalty for a variable x_i having a value of a_i is the estimate of the amount by which the solution to the current subproblem would increase if the integral constraint $x_i \geq \lceil a_i \rceil$ was introduced. The down penalty is similar, except that it is associated with the constraint $x_i \leq \lfloor a_i \rfloor$. The lower bound of a new subproblem is the sum of the optimal simplex solution and the associated penalty. This process is repeated on the new subproblems.

Figure 2(b) shows the branch and bound tree for the problem in Figure 2(a). The dual simplex method gives an optimal solution of 14.2 for the original problem. Since the variables are not integral, a feasible solution has not been generated. Up and down penalties are calculated for the variables and x_1 has the greatest penalty ($U = 1.8$). Two new subproblems are then created, one with $x_1=0$ and the other with $x_1 \geq 1$. The lower bounds are calculated as in Figure 2(b). The dual simplex method is then applied to the subproblem with the smaller lower bound again, and a feasible solution is generated with all variables having integral values. This constitutes an optimal solution since the lower bound of the remaining subproblem is greater.

$$\min x_0 = 7x_1 + 3x_2 + 4x_3$$
$$x_1 + 2x_2 + 3x_3 \geq 8$$
$$3x_1 + 2x_2 + x_3 \geq 5$$
$$x_1, x_2, x_3 \geq 0 \text{ , integer}$$

(a)

Optimal dual simplex solution

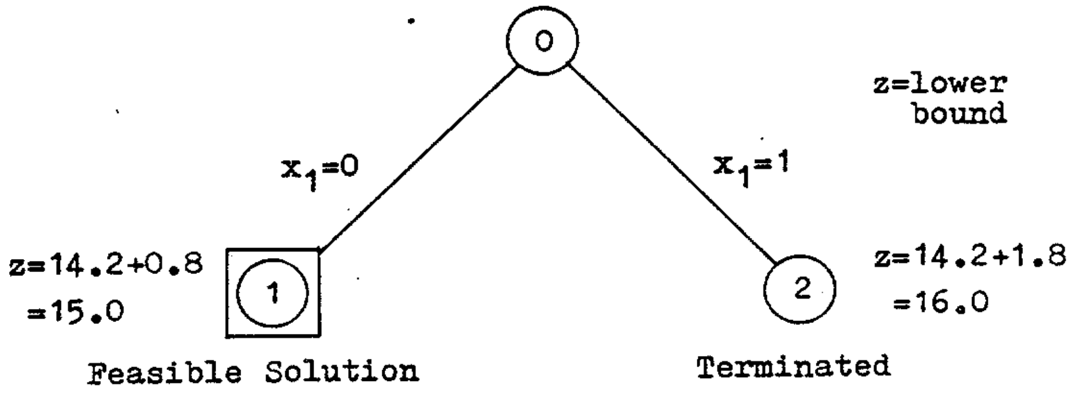
$x_0 = 14.2$

$x_1 = 0.4$

$x_2 = 3.8$

$x_3 = 0$

Variable	Down Penalty	Up Penalty
x_1	0.8	1.8
x_2	0.3	0.13



Optimal dual simplex solution

$x_0 = 15.0$

$x_1 = 0$

$x_2 = 5$

$x_3 = 0$

(b)

Figure 2 : (a) An example of an integer programming problem and (b) the corresponding branch and bound tree

1.2 Virtual Memory Systems

An enormous amount of literature has been published concerning virtual memory systems [24-29]. The set of addresses that a task can reference is called the (virtual) address space V. The set of physical locations in main memory allocated to the task is the memory space M. During execution, reference to addresses in the object code will be to virtual addresses. The virtual memory system automatically maps the address space into the memory space via an address mapping function defined as follows:

$$f : V \rightarrow M \text{ such that if } x \in V \text{ , } f(x) = \begin{cases} y \text{ if } x \text{ is in M at location } y \\ \text{undefined otherwise} \end{cases}$$

An exception or missing-item fault occurs when f(x) is undefined. The item that the task wishes to reference has to be brought into the main memory. The type of allowable missing item or block defines the type of virtual memory system. Depending on the size of the blocks, the virtual memory system may be classified as:

- segmentation where the blocks are of unequal length
- paging where the blocks are of equal length
- segmentation with paging where blocks (segments) are multiples of pages.

The block size chosen is a compromise between constraints imposed by the efficiency of the secondary memory device and the average sizes of the logical program entities. The latency of a paging drum is a device such that a block size of at least 1024 bytes should be used. A moving arm disk requires a block size at least 4 times as big. On the other hand, measurements on programs show that logical blocks are much smaller than 1000 words. Most procedures occupy less than 100 words. Table 1 lists the page sizes of some virtual memory systems.

Three rules govern the transfer of information between the main memory and secondary storage.

- replacement rule which indicates which block is to be displaced from the main memory;
- loading rule which decides when the missing block should be brought into the main memory;
- placement rule which dictates where the missing block is to be placed in the main memory.

Common replacement algorithms include

- *First-in-First-Out (FIFO) :* The page which has been in the main memory for the longest time is removed;
- *Least-Recently-Used (LRU) :* The page which has not been referenced for the longest time is removed;
- *Clock or First-In-Not-Used-First-Out (FINUFO) :* The pages are kept in a circular FIFO queue. Associated with each entry in the queue is a use bit which will be turned on when the corresponding page is referenced after its initial loading. A pointer points to the next page to be replaced. At page fault time, the use bit of the page pointed to by the pointer is examined. If it is off then the page is replaced. If it is on, the use bit is turned off and the pointer is advanced one position. This continues until the pointer points to a page whose use bit is off;

Table 1 : Page sizes used in some computer systems

Computer	Page size
DEC VAX 11/780	512 bytes
IBM 370/168	2048 or 4096 bytes
CDC STAR-100	4096 bytes
DEC PDP-10,20	512 36 bit words
MULTICS	1024 36 bit words

- *Working Set (WS) :* The working set at time t for a window of size T is the set of pages which have been referenced in the interval (t − T + 1 , t). A page is replaced at time t if it does not belong to the working set.

The loading algorithm usually implemented in production virtual memory systems is demand paging in which pages are loaded into the main memory when they are accessed at page fault time. Another approach is prepaging in which future exceptions are predicted and the corresponding pages preloadeded. Clustered loading is sometimes applied because bringing several pages into main memory at a time incurs less overhead than bringing them in individually.

The main memory is normally divided into page frames, each of which has the same size as a page. When the missing page is loaded into main memory, the placement rule will normally place the missing page into the page frame vacated by the page which has been replaced. If no page has been replaced then an empty page frame is arbitrarily chosen.

Central to the successful operation of paging systems is the principle of locality. This principle states that, during execution, a process will favor a subset of V, i.e. , only a subset of its pages need to be resident in main memory to allow seldom interrupted execution intervals. There are two components to this locality of reference: temporal locality which is the tendency for a program to reference in the near future those pages referenced in the recent past; and spatial locality which is the tendency for a program to reference neighboring pages. Loops, constants, temporary variables and working stacks are constructs which lead to temporal locality. Sequential portions of code and traversals of arrays give rise to spatial locality.

These have led to investigations into the behavior of programs in order to determine methods for enhancing the locality of programs. Some rules to be followed have been proposed [30], and some studies on programs implementing specific algorithms such as matrix manipulation [31, 32], sorting [32] and database operations [34] have been made. Investigations have also been performed on automatic restructuring of programs in order to tailor the programs better suited to the locality-based virtual memory environment [35-39]

1.3 Branch and Bound Algorithms in a Locality Based Virtual Memory Environment

In this section, the execution of the branch and bound algorithms in a locality based virtual memory environment is considered. Sub-section 1.3.1 selects the data structures with which the algorithms may be implemented and sub-section 1.3.2 considers the amount of locality exhibited by those implementations.

1.3.1 Data Structures for the Branch and Bound Algorithms

The following three operations on the subproblem list may be defined

(i) insert(x)	insert a subproblem with lower bound x into the subproblem list ;
(ii) delete–smallest(y)	delete the subproblem with the smallest lower bound y from the subproblem list ;

(iii) delete–not–less–than(z)	delete all subproblems with lower bounds not less than z from the subproblem list.

Each iteration of the branch and bound algorithm commences with a delete-smallest operation. Let v be the lower bound of this subproblem. If a feasible solution w is generated, then a delete-not-less-than (w) operation must be carried out. Otherwise s new subproblems with lower bounds $x_1, x_2, \cdots, x_s$ are generated and inserted into the subproblem list. Since $x_1, x_2, \cdots, x_s$ are random variables, it will not be possible to predict where each subproblem will be inserted.

Subproblems are ordered by increasing lower bounds into an ordered list under best-first search. Let the average size of this ordered list be m.

If a sequential mapping into an array is used for the ordered list, both the insert and delete-smallest operations are expensive. The location into which a subproblem is to be inserted may be found efficiently by binary search. However, the use of a sequential mapping forces some of the remaining subproblems to be moved so the sequential mapping is preserved in its proper form, resulting in a time complexity of O(m). The subproblem with the smallest lower bound is the first subproblem in the list. Once this subproblem has been deleted, the remaining subproblems have to be moved in order to preserve the sequential mapping. The delete-smallest operation also has a time complexity of O(m). In contrast, the delete-not-less-than operation is very efficient when a sequential mapping is used. Binary search determines the index beyond which all subproblems are to be deleted. These subproblems are then deleted by updating the size variable of the ordered list. The time complexity will only be O(log m).

An alternative that avoids the excessive data movement of the sequential representation is the linked list. Here successive items of the ordered list may be placed anywhere in memory. To ensure that elements in the list are accessed in the correct order, each item has a pointer to the next element in the list. Insertion and deletion are simple once the desired item has been located. However, finding the correct location may be difficult as sequential search has to be used. This leads to a time complexity of O(m) for an insertion. In contrast, the delete-smallest operation has a time complexity of O(1) since the position of the subproblem with the smallest lower bound is known. Sequential search has to be used in the delete-not-less-than operation. In addition, the deleted nodes will have to be returned to the storage pool. The time complexity will be O(m).

A balanced tree will avoid the excessive cost of insertion exhibited by both the linked list and the sequential allocation. We shall consider two variants of the tree structure namely the B-tree (Appendix A) and the heap (Appendix B).

For a B-tree, both the insert and delete-smallest operations have a time complexity of O(log m). Suppose the delete-not-less-than operation deletes tm subproblems , $0<t<1$. Then it must be carried out as a sequence of tm deletions, resulting in a time complexity of O(m log m).

Insertion and deletion of the smallest subproblem also have a time complexity of O(m log m) in a heap. There is no easy way of accessing the subproblems in a heap in the order of increasing lower bounds. The heap must first be sorted and only then the tm subproblems can be removed. This leads to a time complexity of O(m log m) for the delete-not-less-than operation.

The above discussion shows that no single data structure is optimal for all the three operations. The suitable choice will depend upon the relative frequencies of the various operations. Recalling our assumption that each expansion produces s son subproblems, insertions should be approximately s as frequent as deletions. Simulations show that the delete-not-less-than operation is very infrequent. This implies that the B-tree and the heap are the best data structures for implementing the branch and bound algorithms.

1.3.2 Does the Branch and Bound Algorithm Exhibits Locality?

Suppose that the heap is implemented as an array as described in Appendix B. Then the insert operation exhibits both temporal and spatial locality. Spatial locality arises from the fact that an insert operation always inserts the subproblem at the end of the heap. Temporal locality is exhibited during the restoration of the heap property since the sets of ancestor nodes examined during successive insertions intersect.

On the other hand, the delete-smallest operation exhibits little locality, as the set of nodes examined during the restoration of the heap property for successive delete-smallest operations have few nodes in common. Some locality exists as the root and the node at the end of the heap are always accessed.

The same considerations apply to the sorting phase of the delete-not-less-than operation.

The B-tree is normally implemented by a linked representation. Under these conditions, the insert operation exhibits little locality. The random nature of the insertions imply that the set of nodes accessed during successive insertions will have little in common. In contrast, the delete-smallest and delete-not-less-than operations exhibit both temporal and spatial locality. Each node in a B-tree of order d contains at least d consecutive subproblems. Subproblems removed by successive deletions will either be from the same node or a neighboring node.

For each implementation, the algorithm exhibits locality for either insertion or deletion, but not both. This implies that the branch and bound algorithm will be unable to run efficiently in a locality based virtual memory environment. Modification of the algorithm to enhance the amount of locality exhibited does not appear to be a feasible approach.

Thus, it may be useful to consider an alternative approach to this problem, namely the adaptation of the virtual memory system to the algorithm. This will be the approach discussed in the remainder of this thesis.

1.4 The Perspective and Organization of this Thesis

Before the adaptation of the virtual memory system to the branch and bound algorithms may be carried out it is necessary to investigate the access characteristics of the branch and bound process. Towards this end, a stochastic model of the branch and bound algorithm is presented in Chapter 2. Among the results derived from this model is the distribution function of the active subproblems in the system. Together with the assumptions on the distribution function of the newly generated subproblems, this characterizes the access pattern of the branch and bound algorithm.

Chapter 3 considers the design of a proposed virtual memory system that is based upon these access characteristics. Both analytical techniques and simulations were used to determine suitable values for the system parameters. Simulations indicate that a seven to nine fold improvement in performance may be achieved by utilizing the proposed modified virtual memory system instead of a general purpose virtual memory system with the LRU replacement algorithm for branch and bound algorithms.

There are still many unresolved questions and Chapter 4 examines some of these.

CHAPTER 2
PROBABILISTIC MODELING OF BRANCH AND BOUND ALGORITHMS

In this chapter, a probabilistic model for the branch and bound algorithms is proposed. The problems studied include : (1) finding the memory size requirement for the best first branch and bound algorithm; (2) predicting the number of subproblems evaluated before the branch and bound process terminates; and (3) determining the distribution of active subproblems by their lower bounds

2.1 The Model of the Branch and Bound Process with Best First Search

The branch and bound process can be modeled as two walls moving towards each other. The front wall indicates the value of the lower bound for the subproblems currently expanded. The back wall represents the minimum of all the feasible solutions.

Initially, the front wall is undefined and the back wall is at infinity. The lower bound for the problem is evaluated and this is taken to be the position of the front wall. The problem is then branched into two or more subproblems and a lower bound is calculated for each subproblem. Since the lower bounds of descendent subproblems are always greater than the lower bound of ancestor subproblems, the front wall always moves to the right (Figure 3). Once the current subproblem has been expanded, the front wall moves to the position of the minimum of the set of active subproblems. This subproblem is then expanded and the process repeats.

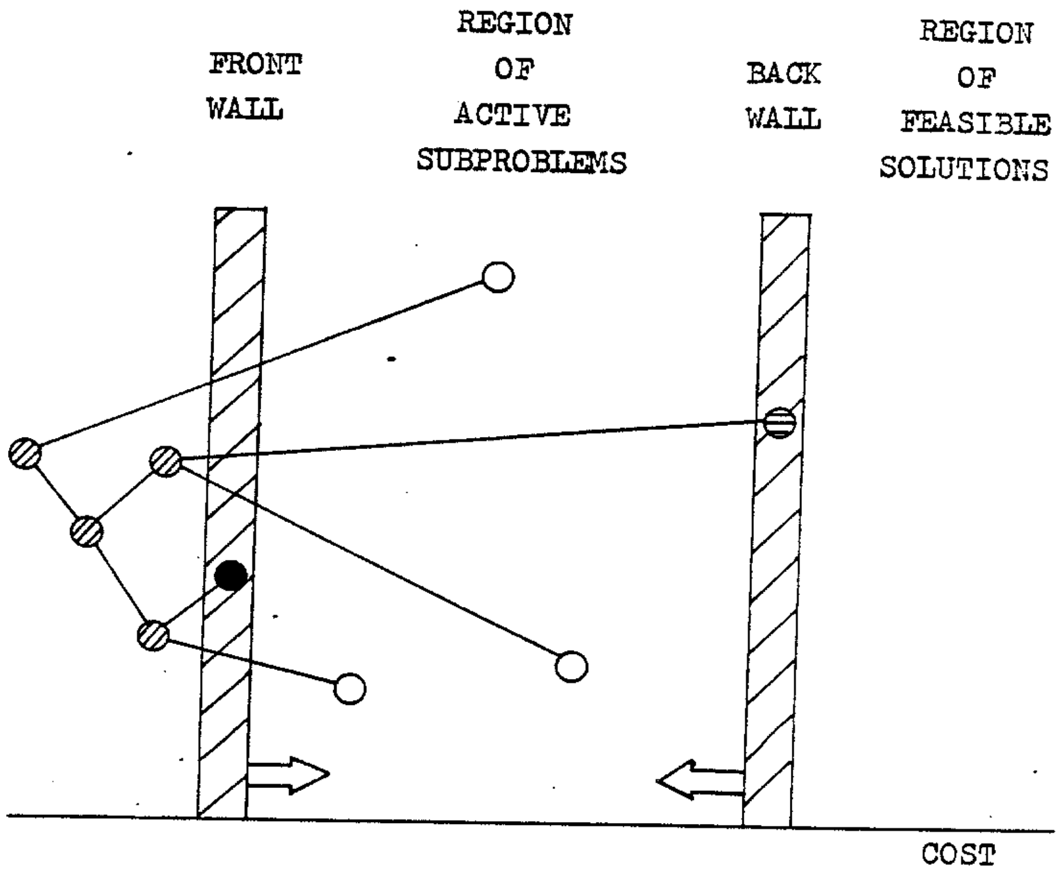
When a subproblem generated becomes a feasible solution, the value of the solution is compared with the position of the back wall. If the position of the back wall is greater than the value of the new feasible solution, the back wall is set to this value; otherwise the feasible solution is ignored. Successive expansion of the subproblems cause the front and back walls to approach each other and the process is terminated when the two walls coincide.

In the following sections, the positions of the front and back walls are calculated. Some simplifying assumptions are made in order for the calculations to be tractable.

2.1.1 The Position of the Front Wall

The solution of the following problem is desired: given the position of the front wall, what is the expected number of subproblems examined; or inversely, given the number of subproblems examined, what is the expected position of the front wall. The set of examined subproblems consists of subproblems that have been processed and no longer belong to the set of active subproblems. The following assumptions are made in the derivation:

(A1) The differences between the lower bounds of the expanded subproblems and the parent subproblem are independent, identically distributed random variables satisfying the gamma density function [8]



○ ACTIVE SUBPROBLEMS

● CURRENTLY EXAMINED SUBPROBLEM

▨ EXAMINED SUBPROBLEM

⊜ FEASIBLE SOLUTION

Figure 3 : The model of the branch and bound process under best first search

$$f_G(y;\alpha,\lambda) = \begin{cases} \frac{\lambda^\alpha}{\Gamma(\alpha)} y^{\alpha-1}e^{-\lambda y} & y > 0 \\ 0 & y \leq 0 \end{cases} \quad (1)$$

The density function is monotonic if $\alpha \leq 1$, and unbounded near the origin when $\alpha < 1$. For $\alpha > 1$, the graph is bell-shaped and as $\alpha \rightarrow \infty$, the density function becomes normal [3]. A gamma density function is chosen because it represents a very general class of density functions. As shown in the next section, this assumption is valid for integer programming.

(A2) Each parent subproblem is expanded into s smaller subproblems. This assumption is valid for a class of NP-complete problems.

Let I be the lower bound of the first subproblem. Let N(x) be the number of subproblems examined when the front wall is at position x and E(N(x)) be the expected value of N(x). When the parent subproblem is expanded, y' and y'' are the differences between the lower bounds of the expanded subproblems and the parent subproblem. E(N(x)) can be written in the form of a renewal equation [23].

$$E(N(x)) = 1 + \int_0^\infty E_1(N(x-y_1)dF_G(y_1) + \int_0^\infty E_1(N(x-y_2)dF_G(y_2)$$

$$+ \cdots + \int_0^\infty E_1(N(x-y_s)dF_G(y_s)$$

or

$$E(N(x)) = 1 + s \int_0^\infty E_1(N(x-y)dF_G(y) \quad (2)$$

where

$$E_1(N(x-y)) = \begin{cases} E(N(x-y)) & \text{if } y < x \\ 0 & \text{if } y \geq x \end{cases} \tag{3}$$

The evaluation of the above renewal equation would result in an incomplete gamma function that cannot be solved analytically. Since $f_G(y) \to 0$, as $y \to \infty$, the assumption that x is reasonably large implies that for any $y > x$, $f_G(y) \simeq 0$. This leads to an approximate renewal equation which can be written as:

$$E'(N(x)) = 1 + s \int_0^\infty E'(N(x-y))dF_G(y) \tag{4}$$

To solve Eqn. (4), a solution is guessed and is substituted into Eqn. (4) in order to verify it. Assume that $E'(N(x)) = k\ e^{mx} - \frac{1}{s-1}$. Substituting into Eqn. (4), we obtain an identity

$$k\ e^{mx} - \frac{1}{s-1} = 1 + s \int_0^\infty (k\ e^{m(x-y)} - \frac{1}{s-1})dF_G(y)$$

$$= 1 + s \int_0^\infty k\ e^{m(x-y)}\ dF_G(y) - \frac{s}{s-1} \int_0^\infty dF_G(y)$$

$$= 1 - \frac{s}{s-1} + s \int_0^\infty k\ e^{m(x-y)}\ dF_G(y)$$

or $$1 = s \int_0^\infty e^{-my} dF_G(y) \tag{5}$$

Using the density function of Eqn. (1) and substituting it into Eqn. (5), m can be solved,

$$m = \lambda(s^{1/\alpha} - 1) \tag{6}$$

To solve for the constant k, we use the boundary condition $E'(I) = 1$. Substituting for $x = I$ in the assumed solution, we obtain

$$1 = k\,e^{mI} - \frac{1}{s-1}$$

or

$$k = \frac{s}{s-1}\,e^{-mI}$$

Therefore,

$$E'(N(x)) = \left(\frac{s}{s-1}\right) e^{\lambda(s^{1/\alpha}-1)(x-I)} - \frac{1}{s-1} \tag{7}$$

In particular when $s=2$ as for the case of the integer programming problem Eqn. (7) becomes

$$E'(N(x)) = 2\,e^{\lambda(2^{1/\alpha}-1)(x-I)} - 1$$

As similar to problems in general renewal theory, the derivation of the distribution function of N(x) is difficult. The expected value of N(x) will, therefore, be used in the calculation of the position of the back wall.

At this time, it is important to know the total number of nodes in the branch and bound tree. All the subproblems to the left of the front wall must have been examined (non-terminal nodes) and all the subproblems to the right of the front wall are active and not examined (terminal nodes). Assuming the degree of every non-terminal node is s, the expected total number of nodes in the branch and bound tree is $E(N_T(x))$ and using Eqn. (7),

$$E(N_T(x)) \simeq s\, E'(N(x)) + 1 \tag{8}$$

2.1.2 The Position of the Back Wall

To determine the position of the back wall, the mechanism involved in generating a feasible solution must be understood. Let n be the number of input parameters. n can be the number of variables in an integer programming problem; n can be the number of cities that a traveling salesman wishes to visit; n can also be the number of nodes in a graph of the vertex covering problem. Before a feasible solution can be obtained, a chain of subproblem expansions must be generated. The number of subproblems in a chain can be less than n (vertex covering problem), equal to n (integer programming problem) or greater than n (traveling salesman problem). To evaluate the position of the back wall, the following additional assumptions are made:

(A3) Every chain that results in a feasible solution is made up of n subproblem evaluations. Each chain starts at the origin and has a length equal to the sum of n independent gamma distributed random variables. Variable length chains will be considered in the future.

(A4) The chains leading to feasible solutions are independent. This assumption is not true in general but is necessary for mathematical tractability.

The number of chains due to $E(N_T(x))$ nodes in the branch and bound tree is $C(n,x)$, and the maximum is $C_{max}(n,x)$ which arises when the nodes in the branch and bound tree are generated in a depth-first fashion. Let the root be at level 1. Then, the number of nodes in level i of the branch and bound tree is,

$$\left\lceil \frac{C_{max}\,(n,x)}{2^{n-i}} \right\rceil$$

The total number of nodes in the branch and bound tree is $E[N_T(x)]$. Thus

$$E(N_T(x)) = \sum_{j=0}^{n-1} \left\lceil \frac{C_{max}(n,x)}{2^j} \right\rceil \tag{9}$$

$C_{max}(n,x)$ can be solved by first calculating its approximate value without the ceiling in Eqn. (9) and searching for the solution in the vicinity of the approximate value.

The actual number of chains formed is, of course, less than $C_{max}(n,x)$. The position of the back wall estimated using $C_{max}(n,x)$ will, therefore, be a lower bound of the actual position.

By assumption (A3), the length of each chain is also gamma distributed with a density function $f_c(y)$ since the family of gamma densities is closed under convolution.

$$f_c(y) = f_G(y;\ n * \alpha,\ \lambda) \tag{10}$$

Since all the chains are assumed independent (assumption (A4)), the position of the back wall is given by the minimum value of all the chains. The distribution function of the minimum of $C(n,x)$ independent, identically distributed random variables is $F_B(y)$ where,

$$F_B(y) = 1 - [1 - F_c(y)]^{C(n,x)} \tag{11}$$

The expected position of the back wall is $E(b)$

$$E(b) = \int_0^\infty y \, dF_B(y) \tag{12}$$

$$= \int_0^\infty y \, C((n,x) \, [\, 1 - F_c(y) \,]^{C(n,x)-1} f_c(y) \, dy$$

$$= y \int_0^\infty C(n,x) \cdot [\, 1 - F_C(y) \,]^{C(n,x)-1} f_C(y) \, dy \; -$$

$$\int_0^\infty \int_0^\infty C(n,x) \, [\, 1 - F_C(y) \,]^{C(n,x)-1} f_C(y) \, dy \, \frac{dy}{dy} \, dy$$

$$= \Big[-y \, [\, 1 - F_C(y) \,]^{C(n,x)} \Big]_0^\infty + \int_0^\infty [\, 1 - F_C(y) \,]^{C(n,x)} \, dy$$

2.1.3 The Distribution of Subproblems Behind the Front Wall

In this section, the distribution function of the difference between the lower bounds of active subproblems and the front wall is calculated. The generation of active subproblems is depicted in Figure 4. The distribution functions of z_1 and z_2 are sought. Since y_1 and y_2 are gamma distributed, an assumption which simplify the calculation is the following.

(A5) The parent subproblem and the corresponding active subproblems it generates can be at any position as long as they lie on opposite sides of the front wall.

From the above assumption, it implies that x and z_2 have the same distribution function. Since y_1 is gamma distributed, x and z_2 are also gamma distributed.

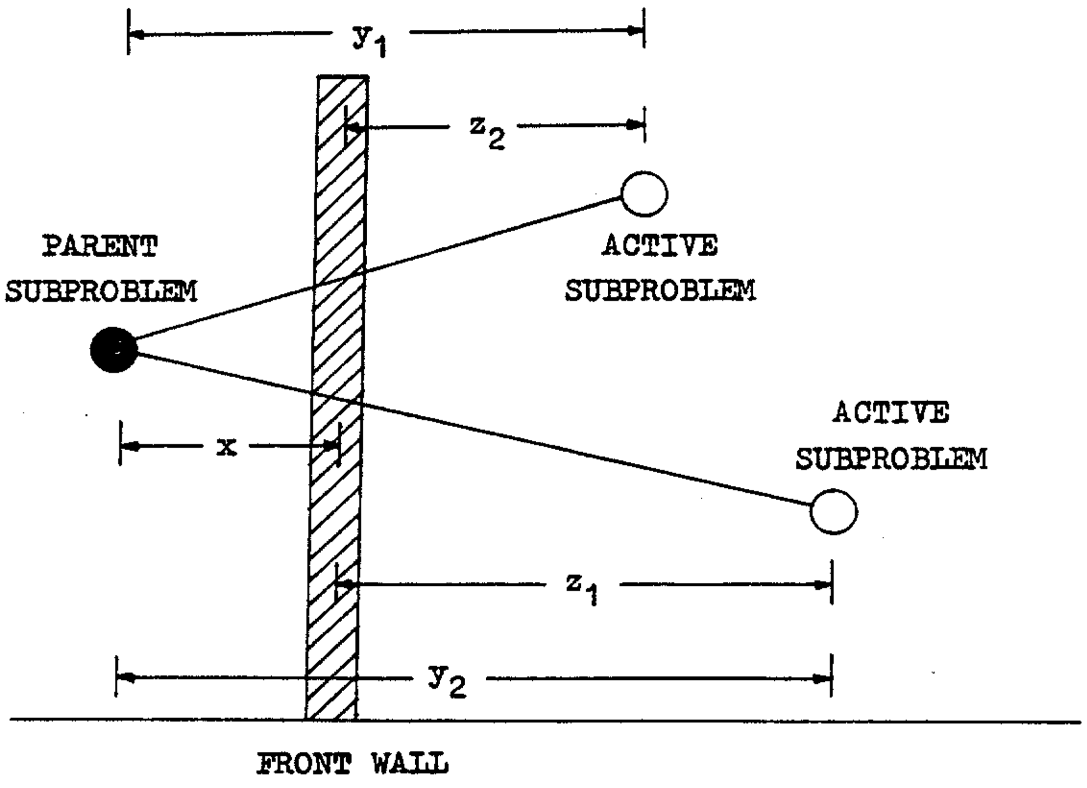


Figure 4 : The generation of a pair of active subproblems

$$f_x(y) = 2 * f_{z_2}(y) = f_G(y;\ \alpha/2,\ \lambda) \tag{13}$$

Similarly, z_1 is gamma distributed with an additional constraint that $z_1 > z_2$. Therefore,

$$f_{z_1}(y) = f_G(y;\ \alpha/2,\ \lambda)[1 - F_G(y;\ \alpha/2,\ \lambda)] \tag{14}$$

The set of active subproblems is a combination of the sets of z_1 and z_2.

2.2. Comparison of the Analytical Model with Simulations

A program to solve the integer programming problem was written in the c language and run on a VAX 11/780 computer at Purdue. The cumulative statistics on the increase in the lower bounds of son subproblems with respect to the parent subproblem was collected over the duration of the solution process. An exponential distribution was fitted on the collected statistics and the results are shown in Table 2. The .20 critical value for the Kolmogorov-Smirnov test is 0.076. Thus the hypothesis that the density function of the increase in lower bounds of son subproblems is exponential for the integer programming problem may be accepted. (An exponential distribution implies $\alpha = 1$ in Eqn. (1))

The cumulative statistics on the distribution of the difference between the lower bounds of active subproblems and the front wall was also collected. Gamma distributions with $\alpha = 0.5, 0.6, 0.7, 0.8, 0.9$ and 1.0 were fitted on the collected statistics and the results are shown in Table 3. The 0.20 critical value for the Kolmogorov-Smirnov test is 0.11. These results show that the exponential distribution ($\alpha = 1$) is a good approximation to the actual distribution.

Table 2 : Testing of the hypothesis that the increase in lower bounds of son subproblems is exponentially distributed for 20 variables, 20 constraints integer programming problems (sample size = 200).

Problem Number	Sample Mean	Kolmogorov-Smirnov Variable, D_n
1	0.184	0.057
2	0.322	0.063
3	0.170	0.062
4	0.230	0.076
5	0.312	0.049

Table 3 : Testing of the hypothesis that the difference between the lower bounds of the active subproblems and the front wall is gamma distributed for four 20 variables, 20 constraints integer programming problems (sample size = 100).

Problem	Sample	Kolmogorov-Smirnov Variable D_n					
Number	Mean	α=0.5	α=0.6	α=0.7	α=0.8	α=0.9	α=1.0
1	0.17	0.21	0.16	0.12	0.09	0.06	0.06
2	0.30	0.23	0.19	0.16	0.13	0.10	0.09
3	0.16	0.20	0.16	0.12	0.09	0.06	0.06
4	0.29	0.24	0.20	0.17	0.14	0.11	0.09

Using the analytical expressions derived in Section 2, we have plotted in Figures 5 to 7 the performance of the branch and bound algorithm using best-first search. In Figures 5 and 6, the position of the front and back walls are plotted for two runs of the integer program using the measured mean of the exponential distribution. It is seen that the expected number of subproblems examined increases exponentially with the position of the front wall. On the other hand, the position of the back wall approaches the front wall as the number of subproblems examined is increased. However, the approach is rather slow and the slope of the graph for the back wall is steep. This implies that as the problem size becomes larger, the number of subproblems that have to be examined before the process terminates increases exponentially.

In spite of the various assumptions that we have made, the estimated positions of the front wall match to within two percent of the simulated position. The results concerning the simulation of the back wall position is not plotted because an initial feasible solution is not generated in our runs and the first feasible solution obtained usually becomes the optimal solution. Nonetheless, assuming 2% error in the estimated positions of the back wall which are plotted in Figures 5 and 6, the number of iterations at termination is predicted correctly. Due to the steepness of the curves and the exponential scale used, the predicted number of iterations may lie in a range of several orders of magnitude.

In Figure 7, the analytical number of active subproblems as a function of the number of subproblems examined is plotted. It indicates that the number of active subproblems first grows to a maximum and decreases to zero at the termination of the process. Furthermore, as n is doubled, the number of active subproblems grows by a factor of 10^4. These indicate the need of an

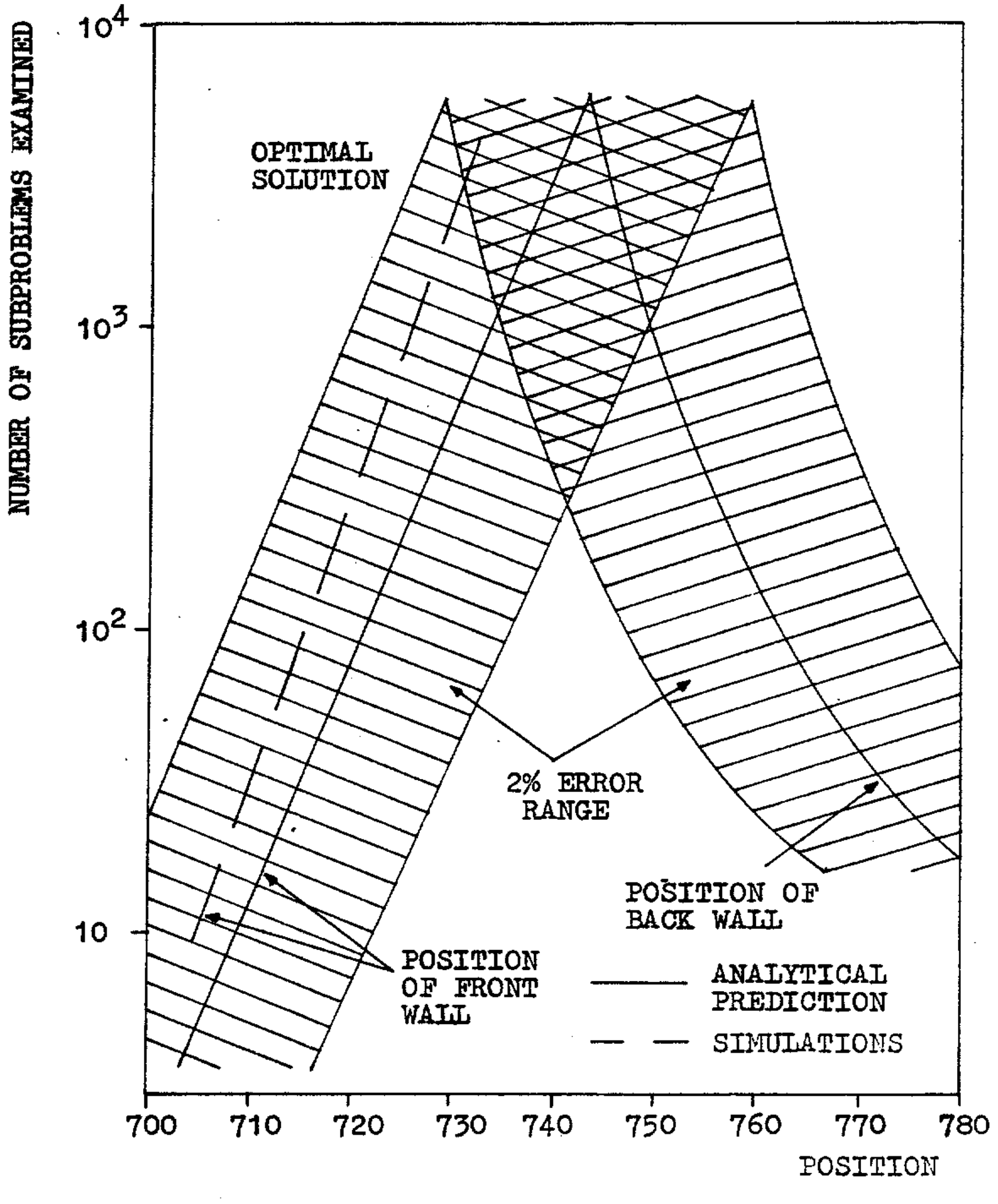


Figure 5 : The positions of the front and back walls for a 20 variables, 20 constraints integer programming problem (problem 1)

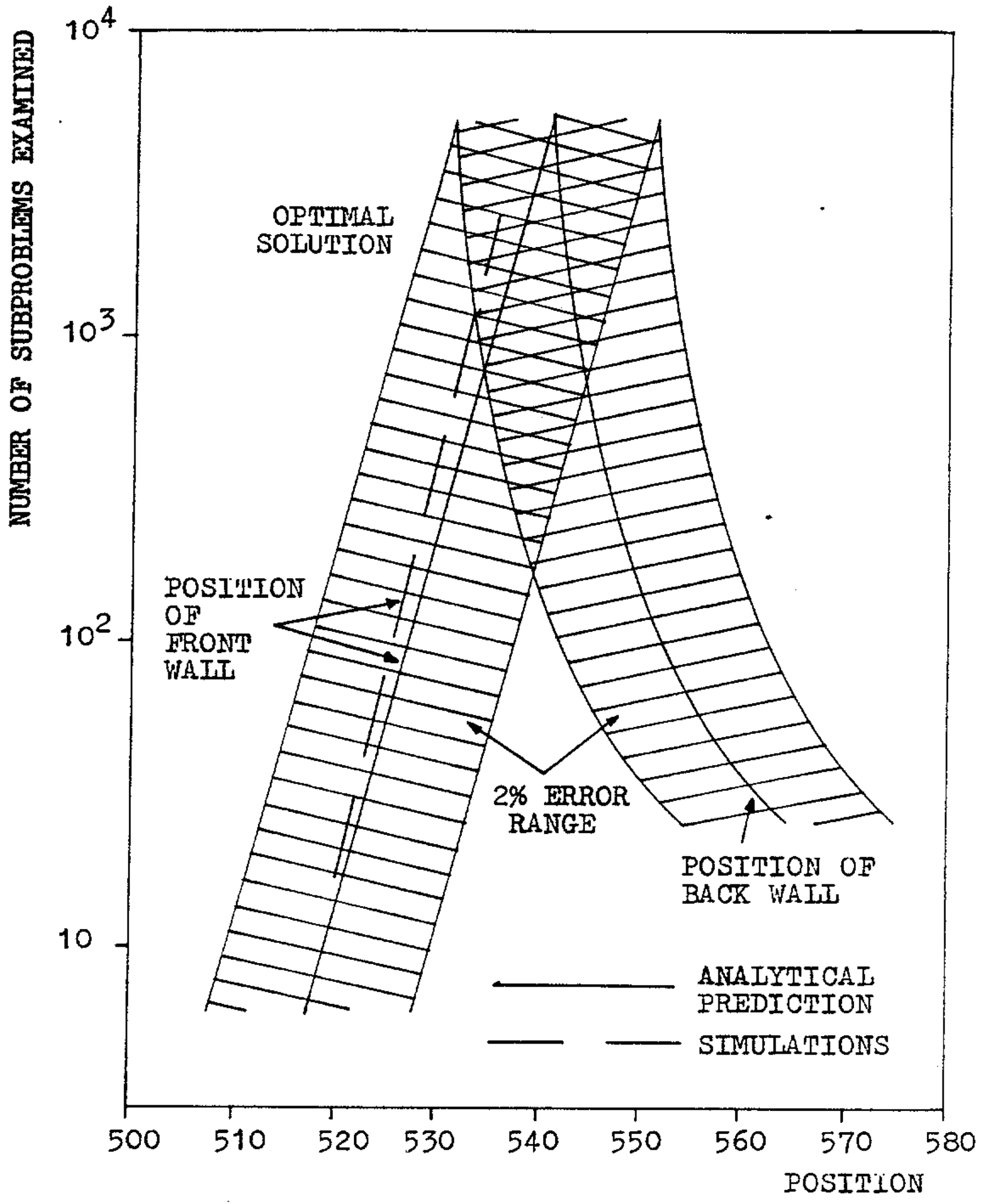


Figure 6 : The positions of the front and back walls for a 20 variables, 20 constraints integer programming problem (problem 2)

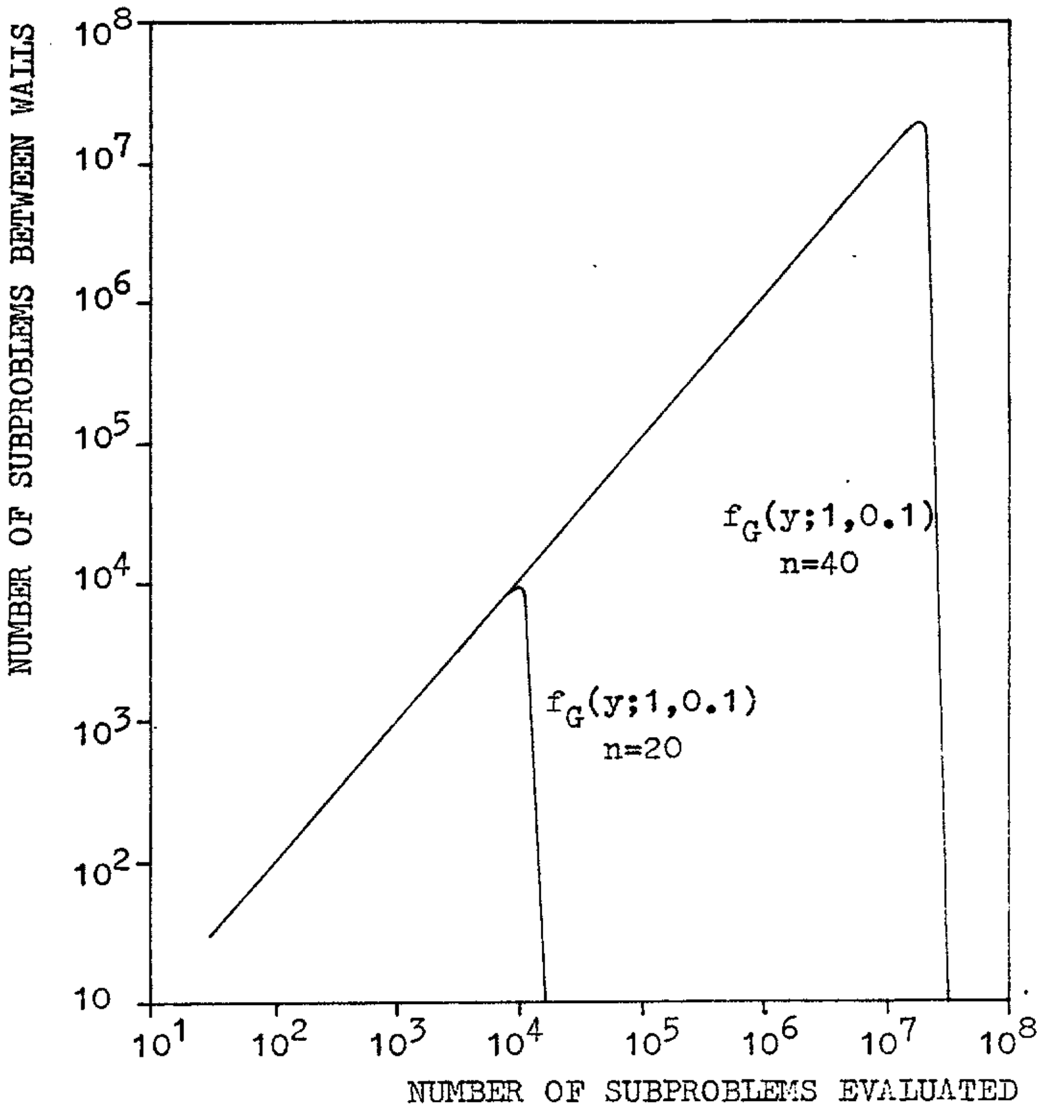


Figure 7 : The number of active subproblems between the two walls as a function of the number of subproblems evaluated

efficient memory management scheme for storing the active subproblems.

2.3 Conclusion

In this chapter we have studied the probabilistic modeling of branch and bound algorithms. With the aid of this model, we can characterize the access behavior of the algorithm. Each iteration of the algorithm commences with the deletion of the subproblem at the head of the subproblem list. This subproblem is expanded into s son subproblems. The differences between the lower bounds of the son subproblems and the parent subproblem are independent identically distributed random variables satisfying the gamma distribution. For the integer programming problem, s is 2 and the distribution is exponential. Also, we find that the differences between the lower bounds of the subproblems in the list and the lower bound of the subproblem at the head of the list is approximately exponential for the integer programming problem.

CHAPTER 3
VIRTUAL MEMORY FOR THE BRANCH AND BOUND ALGORITHMS

In this chapter, the adaptation of the virtual memory system to the branch and bound algorithms is considered.

3.1 A Virtual Memory System for the Branch and Bound Algorithms (VM1)

One desirable feature of such a system is that the execution period between successive replacements should be as long as possible. Another is the batching of disk accesses. In addition, overlap between computation and input/output should be maximized. These suggests that the subproblem list should be split into two disjoint sub-lists, a primary list which resides in the main memory and a secondary list which resides in the secondary storage. Newly generated subproblems are inserted into the primary list. When the primary list grows to its maximum size, subproblems are removed from the primary list and inserted into the secondary list. During this time, computation can still be carried out. The replacement rule selects the subproblems which are to be removed. This use of a primary list in this manner is equivalent to batching the insertions into the virtual space, which reduces the cost of insertions.

The subproblem selected for partitioning is the smaller of the smallest subproblems in the primary and the secondary lists. Thus it is necessary to keep a number of the smallest subproblems of the secondary list in a portion of the main memory, known as the p-buffer. The loading rule governs the movement of subproblems from the secondary list into the p-buffer.

In the following sections, the design of VM1 will be considered by the appropriate choice of the following parameters

- organization of the secondary list
- organization of the main list
- replacement rule
- loading rule
- page size

3.1.1 Organization of the Secondary List

The same considerations that govern the choice of the data structure in Section 1.3 holds in the choice of the organization of the secondary list. The B^+-tree is particularly suitable. Let each leaf correspond to a page of the virtual memory system. The terms page and leaf will be used interchangeably in this chapter. Since the internal nodes of a B^+-tree are very much smaller in size than the leaf nodes, it is possible to keep the index portion of the B^+-tree in main memory. This reduces the number of secondary accesses required by an insertion or a deletion to O(1).

Some modifications to the B^+-tree are necessary. It is more efficient to insert i subproblems into a leaf j as a single operation rather than performing i insertions into the leaf j. The only modification required is to check for overflow only when all i subproblems have been inserted into leaf j. Now, instead of splitting when there are 2m+1 subproblems, there may be 2m+i subproblems. $m+\left\lceil \frac{i}{2} \right\rceil$ subproblems will go into one leaf and $m+\left\lfloor \frac{i}{2} \right\rfloor$ subproblems will go into the other.

It is also necessary to be able to delete more than one subproblem from a leaf, including the entire leaf. This is needed for efficient deletion of terminated subproblems. Deleting the subproblems individually causes an intolerable delay. The modification required is to delete i subproblems from the leaf before checking for underflow. An entire leaf may be deleted by deleting the corresponding separator key and pointer from the parent node.

3.1.2 Organization of the Primary List

The organization chosen for the primary list has to allow efficient insertion and retrieval of the subproblem with the smallest lower bound. Since each subproblem may be rather large, it is desirable not to manipulate the subproblems. This suggests the use of a random organization for the subproblems and a dense index. Since it is only necessary to retrieve the subproblem with the smallest lower bound, it is not necessary to keep the dense index sorted. A more efficient method is to use a heap.

3.1.3 An Analysis of the Expected Number of Page Accesses

In this section, we derive an expression for a quantity η which is the expected number of page accesses normalized by the expected number of iterations under VM1. By minimizing this expression, the suitable page size and replacement rule may be determined.

The two principal causes of page accesses in VM1 are the transfer of subproblems from the main memory to the secondary storage during replacement and the transfer of subproblems from secondary storage to main memory during the loading of the p-buffer.

The expected number of page accesses during replacement is considered in sub-section 3.1.3.1. Sub-section 3.1.3.2 analyses the expected number of iterations between successive replacements while sub-section 3.1.3.3 analyzes the expected number of page accesses due to the loading of the p-buffer. The results of sub-sections 3.1.3.1, 3.1.3.2 and 3.1.3.3 are combined in sub-section 3.1.3.4 to yield an expression for η. This expression is compared with simulation results in sub-section 3.1.3.5.

3.1.3.1 The Expected Number of Page Accesses due to Replacement

Let the maximum size of the primary list be n_p subproblems, that is, the replacement algorithm is called when the primary list grows to n_p subproblems. Suppose ϕ subproblems are then removed from the primary list and inserted into the pages of B^+-tree (secondary list). Also, let the number of subproblems in the secondary list, just before the invocation of the replacement algorithm, be N_s subproblems.

Let $C_{p,i}$, $i = 1,\ldots,n_p$ be the lower bounds of subproblems in the primary list such that $C_{p,1} \leq C_{p,2} \cdots \leq C_{p,n_p}$ Similarly, let $C_{s,i}$, $i = 1,\ldots,N_s$ be lower bounds of the subproblems in the secondary list with $C_{s,1} \leq C_{s,2} \cdots \leq C_{s,N_s}$. We can then define

$$X_i = C_{p,i} - C_{p,1} \qquad i = 1, \cdots, n_p \tag{15}$$

$$Y_j = C_{s,j} - C_{s,1} \qquad j = 1, \cdots, N_s \tag{16}$$

It is assumed that

(i) $C_{p,1} = C_{s,1}$ (17)

(ii) X_i, $i = 1,\ldots,n_p$ are independent, identically distributed random variables having the exponential density function

$$f_X(x) = \lambda_p e^{-\lambda_p x} \tag{18}$$

(iii) Y_j, $j = 1,\ldots,N_s$ are independent, identical exponentially distributed random variables with density function

$$f_Y(y) = \lambda_s e^{-\lambda_s y} \tag{19}$$

Assumptions (ii) and (iii) are reasonable since the distribution of the difference in lower bounds of the active subproblems and the front wall has been shown to be exponential in the previous chapter.

Suppose the ϕ subproblems selected for removal are the subproblems with lower bounds $C_{p,\theta+1}, C_{p,\theta+2}, \cdots, C_{p,\theta+\phi}$, where $X_{\theta+1}$ has the value v and $X_{\theta+\phi}$ has the value w. Let M_s be the number of subproblems in the secondary list with $v < Y_i \leq w$. The probability of $v < Y_i \leq w$ for any subproblem in the secondary list is given by

$$\Pr\{v < Y_i \leq w\} = \int_{x=v}^{w} \lambda_s e^{-\lambda_s x}\, dx$$

$$= e^{-\lambda_s v} - e^{-\lambda_s w}$$

The probability that $M_s = m$ constitutes a binomial distribution

$$\Pr\{M_s = m\} = \begin{pmatrix} N_s \\ m \end{pmatrix} \Big[\Pr\{v < Y_i \le w\}\Big]^m \Big[1 - \Pr\{v < Y_i \le w\}\Big]^{N_s - m}$$

The expected value of M_s given that $X_{\theta+1} = v$, $X_{\theta+\phi} = w$ and $N_s = n$

$$E[M_s | X_\theta = v, X_{\theta+\phi} = w, N_s = n] = n \Pr\{v < Y_i \le w\} \tag{20}$$

Since N_s and $P\{v < Y_i \le w\}$ are independent, Eqn. (20) may be evaluated as

$$E[M_s] = E[N_s] \int_{v=0}^{\infty} \int_{w=0}^{\infty} \Pr\{v < Y_i \le w\}\, f_V(v)\, f_W(w)\, dv\, dw$$

$$= E[N_s] \int_{v=0}^{\infty} \int_{w=0}^{\infty} (e^{-\lambda_s v} - e^{-\lambda_s w})\, f_V(v)\, f_W(w)\, dv\, dw$$

$$= E[N_s] \left[\int_{v=0}^{\infty} e^{-\lambda_s v} f_V(v)\, dv - \int_{w=0}^{\infty} e^{-\lambda_s w} f_W(w)\, dw \right] \tag{21}$$

It can be shown (Appendix C) that

$$\int_{v=0}^{\infty} e^{-\lambda_s v} f_V(v)\, dv = \frac{n_p!}{(n_p - \theta)!} \frac{\Gamma(n_p - \theta + 1 + \varsigma)}{\Gamma(n_p + 1 + \varsigma)} \tag{22}$$

$$\int_{w=0}^{\infty} e^{-\lambda_s w} f_W(w)\, dw = \frac{n_p!}{(n_p - \theta - \phi)!} \frac{\Gamma(n_p - \theta - \phi + 1 + \varsigma)}{\Gamma(n_p + 1 + \varsigma)} \tag{23}$$

where

$$\varsigma = \frac{\lambda_s}{\lambda_p}$$

Inserting Eqns. (22) and (23) into Eqn. (21) yields

$$E[M_s] = E[N_s] \left(\frac{n_p!}{(n_p-\theta)!} \frac{\Gamma(n_p-\theta+1+\varsigma)}{\Gamma(n_p+1+\varsigma)} - \frac{n_p!}{(n_p-\theta-\phi)} \frac{\Gamma(n_p-\theta-\phi+1+\varsigma)}{\Gamma(n_p+1+\varsigma)} \right) \tag{24}$$

Let the M_s subproblems reside in R pages and let $Q_1, Q_2, \cdots, Q_R$ be the number of subproblems in these R pages. Then

$$Q_1 + Q_2 + \cdots + Q_R \geq M_s \tag{25}$$

Assuming the equality in Eqn. (25) and taking expectations yields

$$E[Q_1 + Q_2 + \cdots + Q_R] = E[M_s] \tag{26}$$

By assuming that $Q, Q_1, Q_2, \cdots$ and Q_R are independent and identically distributed, and letting R be a stopping time, renewal theory and Wald's equation gives

$$E[Q_1 + Q_2 + \cdots + Q_R] = E[R]\,E[Q] \tag{27}$$

Combining Eqns. (26) and (27) yields

$$E[M_s] = E[R]\,E[Q] \tag{28}$$

Let h_s subproblems be the size of each page. Assume that the number of subproblems in a page is uniformly distributed between $1/2\,h_s + 1$ and h_s. Thus

$$\Pr\{Q = q\} = \frac{2}{h_s} \qquad 1/2h_s + 1 \leq q \leq h_s \tag{29}$$

and the expected number of subproblems in a page, E[Q], will be given by

$$E[Q] = \frac{3h_s + 2}{4} \tag{30}$$

By substituting Eqn. (30) into Eqn. (28) we obtain

$$E[M_s] = \frac{3h_s + 2}{4} E[R]$$

and

$$E[R] = \frac{4}{3h_s + 2} E[M_s]$$

$$= \frac{4}{3h_s + 2} E[N_s] \left(\frac{n_p!}{(n_p - \theta)!} \frac{\Gamma(n_p - \theta + 1 + \varsigma)}{\Gamma(n_p + 1 + \varsigma)} - \frac{n_p!}{(n_p - \theta - \phi)!} \frac{\Gamma(n_p - \theta - \phi + 1 + \varsigma)}{\Gamma(n_p + 1 + \varsigma)} \right) \tag{31}$$

The derivation of the distribution function of R is extremely difficult. The expected value of R will be used for mathematical tractability. Of the E[R] pages, let subproblems be actually inserted into G pages. Assuming that the ϕ insertions are uniformly distributed among the E[R] pages, the number of subproblems inserted into any page, K, satisfies the binomial distribution

$$\Pr\{K = j\} = \binom{\phi}{j} \left(\frac{1}{E[R]} \right)^{j} \left(1 - \frac{1}{E[R]} \right)^{\phi - j}$$

The probability that at least one subproblem is inserted into a page

$$\Pr\{K \geq 1\} = 1 - \Pr\{K = 0\}$$

$$= 1 - \left(1 - \frac{1}{E[R]}\right)^{\phi}$$

and the expected number of pages into which subproblems are inserted

$$E[G] = E[R]\ Pr\{K \geq 1\}$$

$$= E[R]\left[1 - \left(1 - \frac{1}{E[R]}\right)^{\phi}\right] \tag{32}$$

Out of the G pages, H pages have to be split due to overflow. Suppose a page has q subproblems in it. Then the page has to be split only if more than ($h_s - q$) subproblems are inserted into the page. Thus

$$Pr\{\text{ split page} \mid Q=q\ \} = Pr\{\ K \geq h_s - q + 1\}$$

$$= 1 - Pr\{\ K < h_s - q - 1\}$$

$$= 1 - Pr\{\ K \leq h_s - q\ \} \tag{33}$$

and the probability that a page has to be split, f_r, will be given by

$$f_r = \sum_{q=\delta}^{h_s} Pr\{Q = q\}\ Pr\{\text{split page} \mid Q = q\} \tag{34}$$

where

$$\delta = \max(1/2h_s + 1\ ,\ h_s - \phi + 1)$$

Using Eqns. (29) and (33), Eqn. (34) becomes

$$f_r = \sum_{q=\delta}^{h_s} \frac{2}{h_s}\ (1 - Pr\{\ K \leq h_s - q\})$$

$$= \frac{2}{h_s}(h_s + - \delta) - \frac{2}{h_s}\sum_{q=\delta}^{h_s} \Pr\{ j \le h_s - q\}$$

$$= \frac{2}{h_s}(h_s + - \delta) - \frac{2}{h_s}\sum_{q=\delta}^{h_s}\sum_{j=0}^{h_s-q}\binom{\phi}{j}\left(\frac{1}{E[R]}\right)^{j}\left(1-\frac{1}{E[R]}\right)^{\phi-j}$$

$$= \frac{2}{h_s}(h_s + - \delta) - \frac{2}{h_s}\sum_{j=0}^{h_s-\delta}\left[\left(h_s + 1 - \delta - j\right) \times \right.$$

$$\left. \binom{\phi}{j}\left(\frac{1}{E[R]}\right)^{j}\left(1-\frac{1}{E[R]}\right)^{\phi-j}\right] \tag{35}$$

For $\phi >> 1$, $\frac{1}{E[R]} << 1$ and $\frac{\phi}{E[R]} \simeq \frac{\phi}{E[R]}(1-E[R])$, the Poisson approximation to the binomial distribution may be used as a simplification.

$$f_r \simeq \frac{2}{h_s}(h_s - \delta) - \frac{2}{h_s}\exp\left(-\frac{\phi}{E[R]}\right) \times$$

$$\sum_{j=0}^{h_s-\delta}\left(h_s + 1 - \delta - j\right)\frac{1}{j!}\left(\frac{\phi}{E[R]}\right)^{j}$$

Since any of the G pages is either split or not split, this is a binomial experiment with f_r as the probability of success and $1 - f_r$ the probability of failure. The expected number of pages split is

$$E[H] = f_r G$$

To insert subproblems into a page, the page must be read from the secondary storage into main memory. The subproblems are inserted into the image of the page in main memory and then the image of the page is written back onto secondary storage. This implies that G pages are read in from the disk and

(G+H) pages written back to the disk. Thus the number of page accesses incurred during a replacement, T_s, is given by

$$T_s = G + (G+H)$$

Taking expectations and using Eqn. (32) yields

$$\begin{aligned} E[T_s] &= 2\,E[G] + E[H] \\ &= (\,2 + f_r\,)\,E[G] \\ &= \left\{ 2 + \frac{2}{h_s}(\,h_s + -\delta) - \frac{2}{h_s}\sum_{j=0}^{h_s-\delta}(\,h_s + 1 - \delta - j\,)\binom{\phi}{j}\left(\frac{1}{E[R]}\right)^{j} \times \right. \\ &\qquad \left. \left(1 - \frac{1}{E[R]}\right)^{\phi-j} \right\} E[R]\left\{1 - \left(1 - \frac{1}{E[R]}\right)^{\phi}\right\} \end{aligned} \tag{36}$$

3.1.3.2 The Expected Number of Iterations Between Successive Replacements

In our proposed virtual memory system, the main memory contains the primary list and a p-buffer which holds the leftmost page of the B^+-tree. At present, only the case where the space set aside for the primary list and the p-buffer are fixed will be considered. The case where the space allocated is allowed to vary dynamically will be considered in the future. The loading rule ensures that the p-buffer is never empty whenever the secondary list exists.

When a subproblem is chosen for partioning, let f_s be the probability that the subproblem is from the p-buffer. Also, let the number of subproblems partitioned between two consecutive replacements be D. Of these D subproblems, let D_s subproblems be from the p-buffer and D_p subproblems be from the heap (primary list). Thus,

$$D = D_s + D_p$$

Furthermore, the value of D_s given D satisfies a binomial distribution

$$\Pr\{D_s = k \mid D\} = \binom{D}{k} (f_s)^k \, (1 - f_s)^{D-k}$$

The expected number of subproblems from the p-buffer partitioned between two consecutive replacements,

$$\begin{aligned} E[D_s] &= E[E[D_s \mid D]] \\ &= E[f_s D] \\ &= f_s E[D] \end{aligned} \tag{37}$$

Similarly,the expected number of subproblems from the heap partitioned between two consecutive replacements

$$E[D_p] = (1 - f_s)E[D] \tag{38}$$

In Chapter 2, we have made the assumption that each partitioning of a subproblem generates s new subproblems. These newly generated subproblems are inserted into the heap. The size of the heap will increase by s whenever a subproblem from the p-buffer is partitioned. When a subproblem from the heap is partitioned, the increase in the size of the heap is only $s - 1$ since the subproblem being partitioned has been deleted from the heap. Equating the increase in the size of the primary list between two replacements to the fixed number of subproblems removed during a replacement

$$\phi = s D_s + (s-1) D_p \tag{39}$$

Taking expectations of Eqn. (39) gives

$$\phi = s\,E[D_s] + (s-1)E[D_p]$$

$$= s\,f_s\,E[D] + (s-1)(1-f_s)\,E[D]$$

$$= (s-1+f_s)\,E[D]$$

and

$$E[D] = \frac{\phi}{s-1+f_s} \tag{40}$$

3.1.3.3 The Expected Number of Page Accesses Due to Loading

Let the D_s subproblems reside in F pages of the secondary memory, and $Q_1', Q_2', \cdots, Q_F'$ subproblems be the number of subproblems in each of these F pages. $Q, Q_1', Q_2', \cdots, Q_F'$ are assumed to be independent identically distributed random variables. Let F be a stopping time. The use of renewal theory and Wald's equation gives

$$E[D_s] = E[F]\,E[Q]$$

and

$$E[F] = \frac{E[D_s]}{E[Q]}$$

$$= \frac{4}{3h_s + 2} E[D_s]$$

$$= \frac{4}{3h_s + 2} \frac{f_s \phi}{s-1+f_s}$$

Let the number of page accesses due to the loading of the p-buffer between two successive replacements be T_t. Then

$$E[T_t] = E[F]$$

$$= \frac{4}{3h_s + 2} \frac{f_s \phi}{s-1+f_s} \tag{41}$$

3.1.3.4 An Expression for η

From its definition, η may be written as

$$\eta = \frac{E[T_s] + E[T_t]}{E[D]} \tag{42}$$

Substituting Eqns. (36), (40) and (41) into Eqn. (42) yields

$$\eta = \frac{s-1+f_s}{\phi} \left[\frac{4}{3h_s+2} \frac{f_s \phi}{s-1+f_s} + \left\{ 2 + \frac{2}{h_s}(h_s+1-\delta) - \frac{2}{h_s} \sum_{j=0}^{h_s-\delta} \left(h_s-\delta+1-j\right) \right.\right.$$

$$\left.\left. \times \binom{\phi}{j} \left(\frac{1}{E[R]}\right)^{j} \left(1-\frac{1}{E[R]}\right)^{\phi-j} \right\} E[R] \left(1-\left(1-\frac{1}{E[R]}\right)^{\phi}\right) \right] \tag{43}$$

3.1.3.5 Comparison of η with Simulation Results

To obtain an indication of the validity of Eqn. (43), simulations were performed using integer programming problems, for which $s=2$. Using the values of f_s, λ_p and λ_s obtained from the simulations, the corresponding values of η were computed and compared with the actual number of page accesses, ρ, observed (Table 4). To simplify this comparison, the values for ρ are normalized so that the mean of the normalized values coincide with the mean of η values. The results show that Eqn. (43) is a reasonably good measure of the number of page accesses for integer programming problems.

3.1.4 Replacement Rule

Using the notations introduced in the previous section, let the subproblems in the primary list when the replacement algorithm is called have lower bounds C_i , $i = 1, \cdots ,n_p$ such that $C_1 \leq C_2 \cdots \leq C_{n_p}$.

One of the aims in the design of VM1 is to batch the insertion of subproblems. The lower bound of a subproblem determines the page into which the subproblem is inserted. The subproblems that are most likely to be inserted into the same page as the subproblem with a lower bound C_i are the subproblems with lower bounds C_{i-1} and C_{i+1}. This suggests that these subproblems should be removed as a set. The proposed replacement rule $S(\theta,\phi)$ will select the subproblems with lower bounds $C_{\theta+1},C_{\theta+2}, \cdots ,C_{\theta+\phi}$ for removal. Two special cases of this rule may be identified: $S(0,\phi)$ or front replacement where the ϕ subproblems with the smallest lower bounds are replaced and $S(n_p-\phi,\phi)$ or back replacement where the ϕ subproblems with the largest lower bounds are replaced.

Thus the problem of selecting the proper replacement rule reduces to the selection of the proper values of :

Table 4 : Comparison of η with the actual number of page accesses, ρ, for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $h_s = 33$, $\phi = 60$).

θ	Problem 1			Problem 2		
	Actual		Normalized	Actual		Normalized
	η	ρ	η	ρ	η	ρ
10	7021	3.47	3.443	2899	3.46	3.43
20	6741	3.33	3.29	2768	3.30	3.26
30	6546	3.23	3.20	2657	3.17	3.15
40	6214	3.07	3.12	2521	3.01	3.05
50	6210	3.07	3.07	2475	2.96	2.99
52	6058	3.00	3.06	2474	2.96	2.97

θ	Problem 3			Problem 4		
	Actual		Normalized	Actual		Normalized
	η	ρ	η	ρ	η	ρ
10	2423	3.47	3.38	3155	3.40	3.40
20	2277	3.26	3.21	3019	3.23	3.24
30	2165	3.10	3.09	2930	3.13	3.12
40	2111	3.02	3.00	2887	3.09	3.02
50	1988	2.85	2.94	2763	2.96	2.97
52	1987	2.85	2.91	2728	2.93	2.95

(i) θ or the subproblems which are to be removed from main memory

(ii) ϕ or the number of subproblems to be removed from the main memory during a replacement

For simplicity, only replacement of a fixed number ϕ of subproblems is considered.

The page access measure, η, calculated in section 3.1.3 allows the value of θ and ϕ to be selected. We desire the values of θ and ϕ which will minimize η as given by equation (43) for a given n_p, h_s , and $E[N_s]$. However, this is difficult as f_s, λ_p and λ_s are complex functions of both θ and ϕ. The approximation of optimizing θ and ϕ independently of each other will be used here. Assume that $E[N_s] >> 1$. This implies that $E[R] >> 1$. Thus

$$\eta \simeq \frac{1+f_s}{\phi}\left\{\left[2 + \frac{2}{h_s}(h_s+1-\delta) - \frac{2}{h_s}\left((h_s+1-\delta)\left[1-\frac{1}{E[R]}\right]^{\phi}\right.\right.\right.$$

$$\left.\left.\left.\cdots + \cdots\right)\right] E[R]\left[1-\left(1-\frac{1}{E[R]}\right)^{\phi}\right]\right\}$$

$$= \frac{1+f_s}{\phi}\left(2 + \frac{2}{h_s}(h_s+1-\delta) - \frac{2}{h_s}(h_s+1-\delta) \times \right.$$

$$\left.\left[1-\frac{1}{E[R]}\right]^{\phi}\right) E[R]\left[1-\left(1-\frac{1}{E[R]}\right)^{\phi}\right]$$

$$\simeq \frac{1+f_s}{\phi}\, 2\, E[R]\left[1-\left(1-\frac{1}{E[R]}\right)^{\phi}\right] \qquad (44)$$

E[R] may be simplified to

$$E[R] = \frac{4\,E[N_s]}{3\,h_s + 2}\left[\frac{n_p!}{(n_p-\theta)!}\,\frac{\Gamma(n_p-\theta+1+\varsigma)}{\Gamma(n_p+1+\varsigma)} - \frac{n_p!}{(n_p-\theta-\phi)!}\,\frac{\Gamma(n_p-\theta-\phi+1+\varsigma)}{\Gamma(n_p+1+\varsigma)}\right]$$

$$= \frac{4\,E[N_s]}{3\,h_s+2}\,\frac{n_p!}{\Gamma(n_p+1+\varsigma)}\left[\frac{\Gamma(n_p-\theta+1+\varsigma)}{(n_p-\theta)!} - \frac{\Gamma(n_p-\theta-\phi+1+\varsigma)}{(n_p-\theta-\phi)!}\right]$$

$$= \frac{4\,E[N_s]}{3\,h_s+2}\left(\prod_{i=1}^{n_p}\frac{i}{i+\varsigma}\right)\frac{1}{\Gamma(1+\varsigma)}\left[\Gamma(1+\varsigma)\prod_{i=1}^{n_p-\theta}\frac{i+\varsigma}{i} - \Gamma(1+\varsigma)\prod_{i=1}^{n_p-\theta-\phi}\frac{i+\varsigma}{i}\right]$$

$$= \frac{4\,E[N_s]}{3\,h_s+2}\left(\prod_{i=n_p-\theta-\phi+1}^{n_p}\frac{i}{i+\varsigma}\right)\times \left\{\left(\prod_{i=n_p-\theta-\phi+1}^{n_p-\theta}\frac{i+\varsigma}{i}\right)-1\right\} \qquad (45)$$

Let θ^* be the value of θ which minimizes Eqn. (44) for a given n_p, h_s, $E[N_s]$ and ϕ. Simulations based on integer programming problems show that the parameters f_s, λ_p and λ_s varies as θ varies (Table 5). Quadratic approximation of these variations was used, whereby

$$f_s = \alpha_{11} + \alpha_{12}\,\theta + \alpha_{13}\,\theta^2$$

$$\lambda_p = \alpha_{21} + \alpha_{22}\,\theta + \alpha_{23}\,\theta^2 \tag{46}$$

$$\lambda_s = \alpha_{31} + \alpha_{32}\,\theta + \alpha_{33}\,\theta^2$$

The optimization of θ was then performed using non-linear programming. Table 6 shows the α_{ij}'s estimated for the integer programming problems shown in Table 5 and the corresponding values of θ^*. It can be seen that there is a significant variation in the values of the α_{11}, α_{12}, α_{21}, α_{22}, α_{31} and α_{32} for the different problems. The values of α_{13}, α_{23} and α_{33} remain reasonably close for all four problems. The effect of slight perturbations in the values of the α_{11}, α_{12}, α_{21}, α_{22}, α_{31} and α_{32} upon the value of θ^* are shown in Table 7. It appears that slight perturbations do not affect the value of θ^*.

The use of Back Replacement is also suggested by statistics on the number of page accesses (Figure 8) collected during the simulations for the same four integer programming problems.

Under Back Replacement, Eqn. (45) reduces to

$$E[R] = \frac{4\,E[N_s]}{3\,h_s + 2} \prod_{j=1}^{n_s} \left(\frac{i}{i + \varsigma}\right) \left(\prod_{i=1}^{\phi} \left(\frac{(i + \varsigma)}{i}\right) - 1\right) \tag{47}$$

Let ϕ^* be the value of ϕ which minimizes Eqn. (44). Once again, complications arise from the variations of f_s, λ_p and λ_s with ϕ(Table 8). Quadratic approximation was again used.

$$f_s = \beta_{11} + \beta_{12}\,\phi + \beta_{13}\,\phi^2$$

$$\lambda_p = \beta_{21} + \beta_{22}\,\phi + \beta_{23}\,\phi^2 \tag{48}$$

Table 5 : Variation of λ_p, λ_s and f_s with θ for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $h_s = 33$, $\phi = 60$).

θ	Problem1			Problem2		
	f_s	λ_p	λ_s	f_s	λ_p	λ_s
10	0.866	0.138	0.196	0.854	0.169	0.356
20	0.802	0.169	0.195	0.783	0.192	0.341
30	0.760	0.203	0.194	0.728	0.262	0.331
40	0.717	0.223	0.193	0.680	0.294	0.323
50	0.684	0.238	0.192	0.642	0.353	0.313
52	0.674	0.240	0.192	0.627	0.364	0.313

θ	Problem3			Problem4		
	f_s	λ_p	λ_s	f_s	λ_p	λ_s
10	0.837	0.111	0.174	0.839	0.177	0.343
20	0.757	0.133	0.166	0.772	0.205	0.336
30	0.703	0.159	0.161	0.714	0.247	0.324
40	0.650	0.186	0.158	0.669	0.303	0.313
50	0.607	0.207	0.155	0.625	0.348	0.306
52	0.591	0.210	0.154	0.620	0.355	0.304

Table 6 : α_{ij} , i,j = 1,...,3 and θ^* for the four integer programming problems shown in Table 5

	Problem 1	Problem 2	Problem 3	Problem 4
α_{11}	0.9287	0.9280	0.9140	0.9130
α_{12}	-0.0068	-0.0079	-0.0084	-0.0078
α_{13}	3.872e-05	4.382e-05	4.434e-05	4.309e-05
α_{21}	0.0946	0.1298	0.0839	0.1447
α_{22}	0.0045	0.0032	0.0026	0.0026
α_{23}	-3.352e-05	2.402e-05	-3.315e-06	2.752e-05
α_{31}	0.1970	0.3702	0.1821	0.3543
α_{32}	-0.0001	-0.0015	-0.0009	-0.0010
α_{33}	1.794e-07	9.151e-06	7.571e-06	2.752e-05
θ^*	52	52	52	52

Table 7 : Variation of θ^* with small perturbations in the values of α_{11}, α_{12}, α_{21}, α_{22}, α_{31} and α_{32} for problem 2 of Table 5

γ	θ^*					
	$\alpha_{11}^- + \gamma < \alpha_{11} \leq \alpha_{11}^+ + \gamma$	$\alpha_{12}^- + \gamma < \alpha_{12} \leq \alpha_{12}^+ + \gamma$	$\alpha_{21}^- + \gamma < \alpha_{21} \leq \alpha_{21}^+ + \gamma$	$\alpha_{22}^- + \gamma < \alpha_{22} \leq \alpha_{22}^+ + \gamma$	$\alpha_{31}^- + \gamma < \alpha_{31} \leq \alpha_{31}^+ + \gamma$	$\alpha_{32}^- + \gamma < \alpha_{32} \leq \alpha_{32}^+ + \gamma$
-7ϵ	52	52	52	52	52	52
-6ϵ	52	52	52	52	52	52
-5ϵ	52	52	52	52	52	52
-4ϵ	52	52	52	52	52	52
-3ϵ	52	52	52	52	52	52
-2ϵ	52	52	52	52	52	52
-1ϵ	52	52	52	52	52	52
$+0\epsilon$	52	52	52	52	52	52
$+1\epsilon$	52	52	52	52	52	52
$+2\epsilon$	52	52	52	52	52	52
$+3\epsilon$	52	52	52	52	52	52
$+4\epsilon$	52	52	52	52	52	52
$+5\epsilon$	52	52	52	52	52	52
$+6\epsilon$	52	52	52	52	52	52
$+7\epsilon$	52	52	52	52	52	52

$\alpha_{11}^- = 0.9200$	$\alpha_{11}^+ = 0.9300$
$\alpha_{12}^- = -0.0079$	$\alpha_{12}^+ = -0.0078$
$\alpha_{21}^- = 0.1200$	$\alpha_{21}^+ = 0.1300$
$\alpha_{22}^- = 0.0032$	$\alpha_{22}^+ = -0.0033$
$\alpha_{31}^- = 0.3700$	$\alpha_{31}^+ = 0.3800$
$\alpha_{32}^- = -0.0015$	$\alpha_{32}^+ = -0.0014$

$$\epsilon = \begin{cases} 0.01 & \text{for } \alpha_{i1} \\ 0.0001 & \text{for } \alpha_{i2} \end{cases}$$

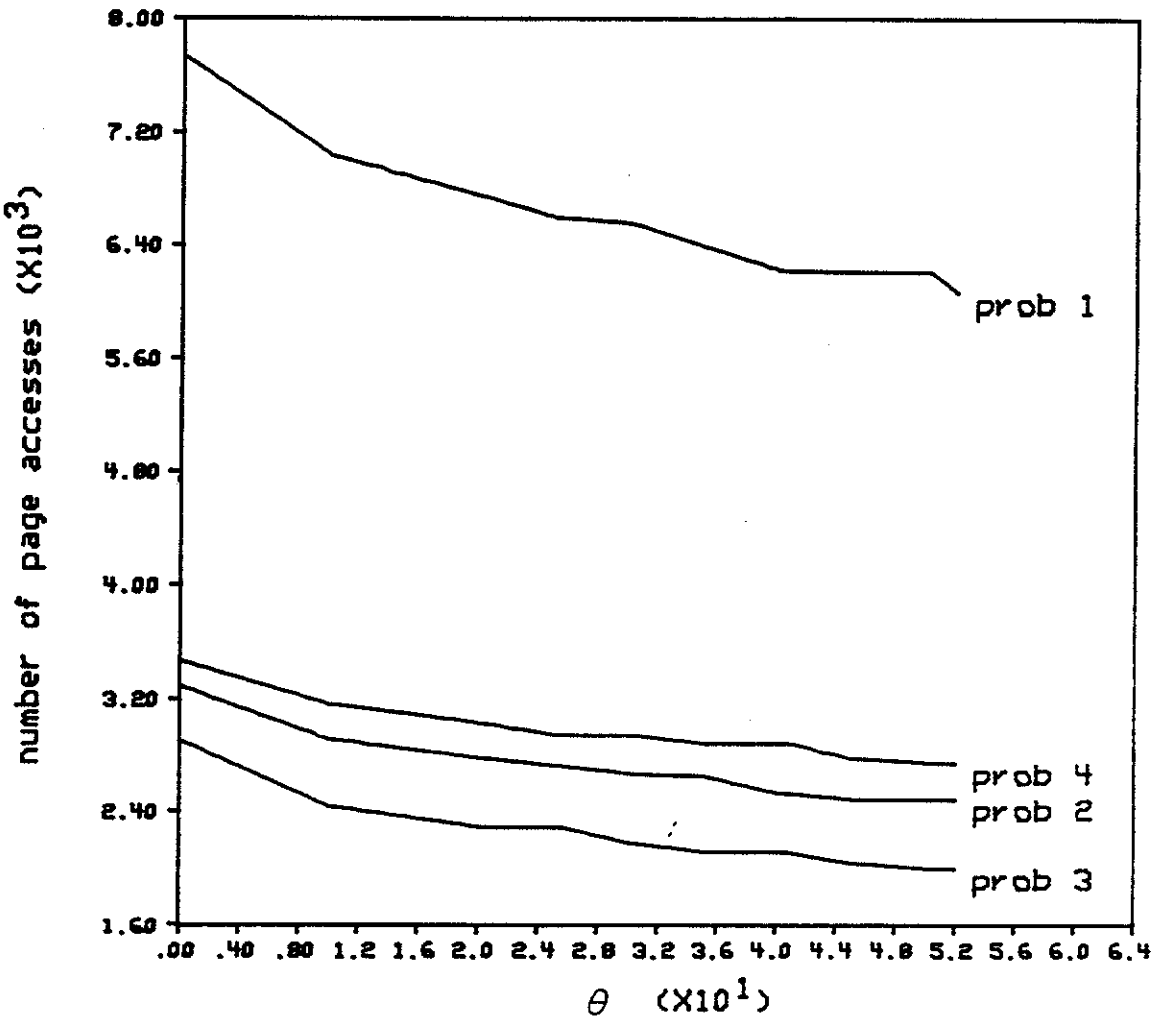


Figure 8 : Variation of the number of page accesses with θ for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $h_s = 33$, $\phi = 60$)

$$\lambda_s = \beta_{31} + \beta_{32}\,\phi + \beta_{33}\,\phi^2$$

The values of the β_{ij}'s for the four problems of Table 8 are shown in Table 9. It is noticed that there is a sharp fluctuation in the value of the parameters in the region of $\phi = 40$. Quadratic approximation will not be able to handle these sharp variations. Non-linear programming was used to solve for ϕ^* and the results are shown in Table 9. It can be seen that the values of ϕ^* were between 0.7 n_p and 0.9 n_p. The results obtained also show that the values for β_{11}, β_{12}, β_{21}, β_{22}, β_{31} and β_{32} differ significantly among the various problems. In contrast, the values of β_{13}, β_{23} and β_{33} remain reasonably constant. The effect of slight perturbations on the values of β_{11}, β_{12}, β_{21}, β_{22}, β_{31} and β_{32} are shown in Table 10. The results show that this has a significant effect on ϕ^*.

Statistics on page accesses (Figure 9) collected by simulations on the same four integer programming problems also suggest such a result. It is noticed that the number of page accesses is minimum at a ϕ^* which is problem dependent. These values are problem dependent and are difficult to predict. A safer approach is to select ϕ^* in the range of $0.7n_p$ to $0.9n_p$. Here the number of page accesses is consistently close to the minima and differs from it by at most 10%.

3.1.5 Page Size

The cost of a page access varies with the size of the page. Thus to select the correct page size it is necessary to consider the total cost of the page accesses and not the number of page accesses.

The characteristics of disk drives will first be considered. There are two types of media, removable media (diskpacks) and non-removable media. Each

Table 8 : Variation of the parameters f_s, λ_p and λ_s with ϕ for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $h_s = 33$, $\theta = n_p - \phi$)

ϕ	Problem 1			Problem 2		
	f_s	λ_p	λ_s	f_s	λ_p	λ_s
10	0.207	0.477	0.119	0.272	0.682	0.203
20	0.215	0.409	0.123	0.298	0.728	0.266
30	0.227	0.369	0.130	0.286	0.534	0.223
40	0.349	0.374	0.191	0.330	0.504	0.228
50	0.384	0.333	0.190	0.360	0.504	0.266
60	0.409	0.305	0.197	0.386	0.439	0.261
70	0.447	0.284	0.194	0.417	0.421	0.273
80	0.482	0.263	0.202	0.460	0.392	0.276
100	0.564	0.234	0.201	0.548	0.358	0.284
110	0.627	0.228	0.206	0.608	0.350	0.285
120	0.687	0.215	0.202	0.670	0.337	0.280

ϕ	Problem 3			Problem 4		
	f_s	λ_p	λ_s	f_s	λ_p	λ_s
10	0.178	0.511	0.115	0.143	0.506	0.084
20	0.189	0.425	0.122	0.380	0.517	0.176
30	0.152	0.352	0.083	0.143	0.340	0.089
40	0.379	0.383	0.193	0.428	0.383	0.183
50	0.411	0.349	0.197	0.447	0.337	0.184
60	0.438	0.316	0.202	0.485	0.308	0.191
70	0.472	0.299	0.199	0.508	0.274	0.190
80	0.507	0.268	0.203	0.545	0.258	0.191
100	0.590	0.242	0.206	0.628	0.226	0.191
110	0.650	0.233	0.208	0.689	0.219	0.197
120	0.697	0.230	0.213	0.752	0.207	0.196

Table 9 : β_{ij} , i,j = 1,2,3 and ϕ^* for the four integer programming problems shown in Table 8

	Problem 1	Problem 2	Problem 3	Problem 4
β_{11}	0.3002	0.2940	0.3182	0.4042
β_{12}	0.0004	-0.0001	0.0008	-0.00046
β_{13}	2.351e-05	2.764e-05	2.011e-05	2.874e-05
β_{21}	0.5410	0.6966	0.5640	0.5821
β_{22}	-0.0050	-0.0052	-0.0053	-0.0060
β_{23}	2.012e-05	1.945e-05	2.231e-05	2.572e-05
β_{31}	0.1738	0.1492	0.1867	0.1715
β_{32}	0.0004	0.0029	0.0001	0.0003
β_{33}	-2.398e-06	-1.780e-06	8.421e-07	-8.018e-07
ϕ^*	110	113	123	108

Table 10 : Variation of ϕ^* with small perturbations in the values of β_{11}, β_{12}, β_{21}, β_{22}, β_{31} and β_{32} for problem 2 of Table 8

γ	ϕ^*					
	$\beta_{11}^- + \gamma < \beta_{11} \le \beta_{11}^+ + \gamma$	$\beta_{12}^- + \gamma < \beta_{12} \le \beta_{12}^+ + \gamma$	$\beta_{21}^- + \gamma < \beta_{21} \le \beta_{21}^+ + \gamma$	$\beta_{22}^- + \gamma < \beta_{22} \le \beta_{22}^+ + \gamma$	$\beta_{31}^- + \gamma < \beta_{31} \le \beta_{31}^+ + \gamma$	$\beta_{32}^- + \gamma < \beta_{32} \le \beta_{32}^+ + \gamma$
-7ϵ	111	113	97	116	109	109
-6ϵ	111	113	97	101	110	110
-5ϵ	112	113	96	115	110	110
-4ϵ	112	113	96	96	110	111
-3ϵ	112	113	95	114	111	111
-2ϵ	112	113	95	114	112	112
-1ϵ	114	113	113	113	112	112
$+0\epsilon$	113	113	113	113	114	113
$+1\epsilon$	113	113	113	112	113	113
$+2\epsilon$	113	113	112	112	113	95
$+3\epsilon$	114	94	112	112	95	95
$+4\epsilon$	114	94	111	111	95	96
$+5\epsilon$	114	94	111	111	96	96
$+6\epsilon$	114	94	111	111	96	100
$+7\epsilon$	115	94	110	110	96	97

$\beta_{11}^- = 0.29 \qquad \beta_{11}^+ = 0.30$

$\beta_{12}^- = -0.0001 \qquad \beta_{12}^+ = 0.0000$

$\beta_{21}^- = 0.69 \qquad \beta_{21}^+ = 0.70$

$\beta_{22}^- = -0.0053 \qquad \beta_{22}^+ = -0.0052$

$\beta_{31}^- = 0.14 \qquad \beta_{31}^+ = 0.15$

$\beta_{32}^- = 0.0029 \qquad \beta_{32}^+ = 0.0030$

$$\epsilon = \begin{cases} 0.01 & \text{for } \beta_{i1} \\ 0.0001 & \text{for } \beta_{i2} \end{cases}$$

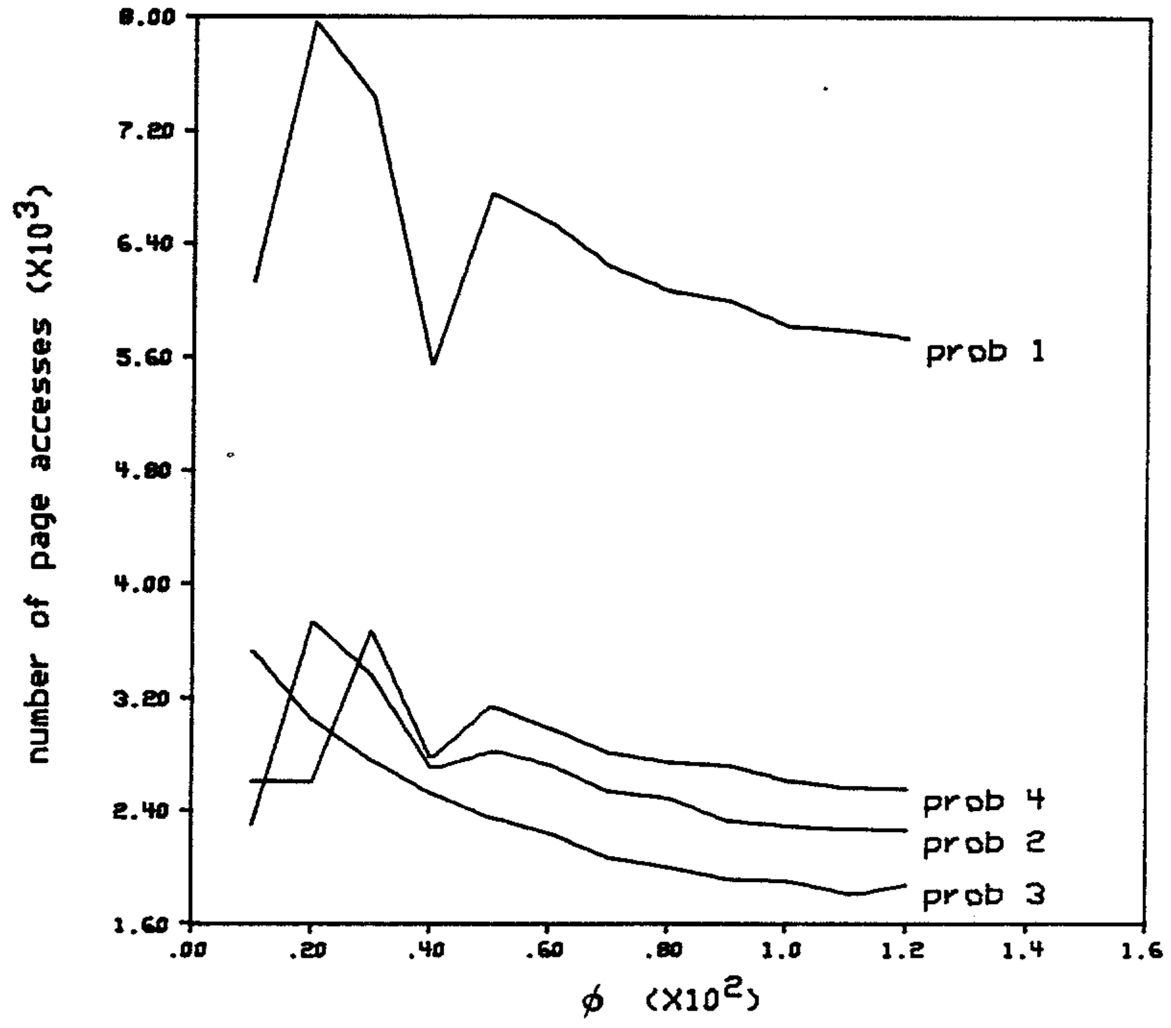


Figure 9 : Variation of the number of page accesses with ϕ for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $h_s = 33$, $\theta = n_p - \phi$)

may have movable access mechanisms or fixed access mechanisms. The most popular is the diskpack drive. Here, a set of disks are mounted together in a stack. Multiple disks are separated by spacers from other disks in the stack. The pack of disks is placed in a spindle and rotate continuously. Each surface can be read by a recording head mounted on one of a set of arms that move in or out, relative to the center of the disk. All recording heads in a disk drive move in unison, but only one head may actually transfer information at any time. No further physical movements of the arms is required to reach any of the set of tracks that are radially equidistant from the center but on different surfaces. For these tracks, only electronic switching delays are incurred to switch from track to track. These delays are relatively negligible. Such a set of tracks is known as a cylinder.

In another type of disk design, the disk units have one head per track which entirely avoids the access arm movement. Geometrically different but logically identical are magnetic drums which has stationary heads mounted on a rotating cylindrical magnetic surface. These head-per-track devices provide less storage.

A track is often subdivided in smaller areas called sectors since the entire track tends to be unmanageably large. The division of a track into sectors may be implemented completely by hardware. There will be a fixed number of sectors, each capable of holding a fixed number of bytes. In some disk units, the track is divided into sectors by a software-controlled formatting operation, which writes markers onto the disk to be recognized by the hardware during normal read and write operations. If the size of a hardware sector is inconveniently small, it is possible routinely to use a number of sectors together.

The effective working unit, be it a sector, a number of sectors, a formated track sector, a track or a software defined portion of the track, will be referred to as a block. A block, then, is a collection of data of a fixed size within a computer system which is moved as a unit between the secondary storage devices and main memory. The fixed size reduces hardware and software complexity at the lower level and defers some decisions to a level closer to the user programs.

The average time to reach a specific known position which contains the data item to be accessed, from an undetermined previous position is known as the random access time. It is frequently broken down into two constituents, seek time and rotational delay. The seek time is the time required to position the access mechanism over the proper track. Seek times can be approximated by a linear relationship $a + bi$, where i is the distance traveled in terms of intertrack spaces and a and b are constants. Often only an average value t_s is used. After the head is positioned on the correct track, a further delay is incurred to reach the desired block on the track. The delay between the completion of the seek and the actual transfer of data constitutes the rotational latency. Part of the delay occurs because the reading cannot commence at an arbitrary point. A track begin point may have to be reached first before the location of the desired block. In many devices, the track begin point is sensed by a separate sensing device. The average value of the rotational latency, t_r, is half the time required for one rotation of the disk.

When the proper track and position is reached the actual data still has to be accessed. On disks the transfer rate is a function of rotational speed and track capacity. Since the transfer of a block is such a frequent operation, the block transfer time, t_b, is often used.

The average cost of a page access for a page of size e blocks will be

$$t_c = t_s + t_r + et_b \tag{47}$$

The average total paging cost normalized by the expected number of iterations will be given by

$$\kappa = \eta \, t_c \tag{48}$$

However, the parameters f, m_p and m_s again show variations with page size (Table 11). An approximation which may be used is a step function. At present, floating point truncation problems prevents the use of nonlinear programming to determine the value of the page size which minimizes κ. Simulation results (Figures 10 - 14) based on the integer programming problem suggest the use of a page size, h_s, in the range of $0.65\,n_p$ and $0.85\,n_p$. Each 20 variables, 20 constraints integer programming problem occupies 1974 bytes.

3.1.6 Loading Rule

The large page size, h_s, selected means that it is inefficient to use a p-buffer of size h_s. If a separate portion of main memory is set aside for the p-buffer, the value of n_p will be reduced significantly. If the p-buffer shares the same main memory space as the heap, a replacement may be needed to create space for the subproblems being loaded into the p-buffer.

A reasonable compromise is to set aside a p-buffer for k subproblems. Instead of loading the entire page, the k subproblems with the smallest lower bounds are deleted from the left-most page of the B^+-tree and loaded into the p-buffer. This scheme will eventually cause an underflow of the left-most page and precipitate either a redistribution or a concatenation which results in further page accesses.

Table 11 : Variation of λ_p, λ_s and f_s with page size for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $\phi = 120$, $\theta = n_p - \phi$).

Page size	Problem 1			Problem 2		
(kbytes)	f_s	λ_p	λ_s	f_s	λ_p	λ_s
32	0.842	0.181	0.160	0.822	0.297	0.281
64	0.841	0.181	0.160	0.819	0.294	0.281
96	0.845	0.181	0.160	0.819	0.292	0.281
128	0.836	0.180	0.160	0.819	0.291	0.280
160	0.785	0.266	0.286	0.662	0.272	0.191
192	0.785	0.266	0.286	0.662	0.272	0.191
224	0.785	0.266	0.286	0.662	0.272	0.193
256	0.785	0.266	0.286	0.662	0.272	0.192

Page Size	Problem 3			Problem 4		
(kbytes)	f_s	λ_p	λ_s	f_s	λ_p	λ_s
32	0.803	0.167	0.150	0.814	0.278	0.279
64	0.794	0.166	0.150	0.809	0.279	0.280
96	0.806	0.165	0.150	0.811	0.281	0.281
128	0.799	0.164	0.150	0.818	0.277	0.277
160	0.666	0.174	0.150	0.681	0.272	0.243
192	0.664	0.173	0.150	0.681	0.272	0.243
224	0.667	0.173	0.149	0.679	0.272	0.240
256	0.669	0.173	0.150	0.660	0.276	0.244

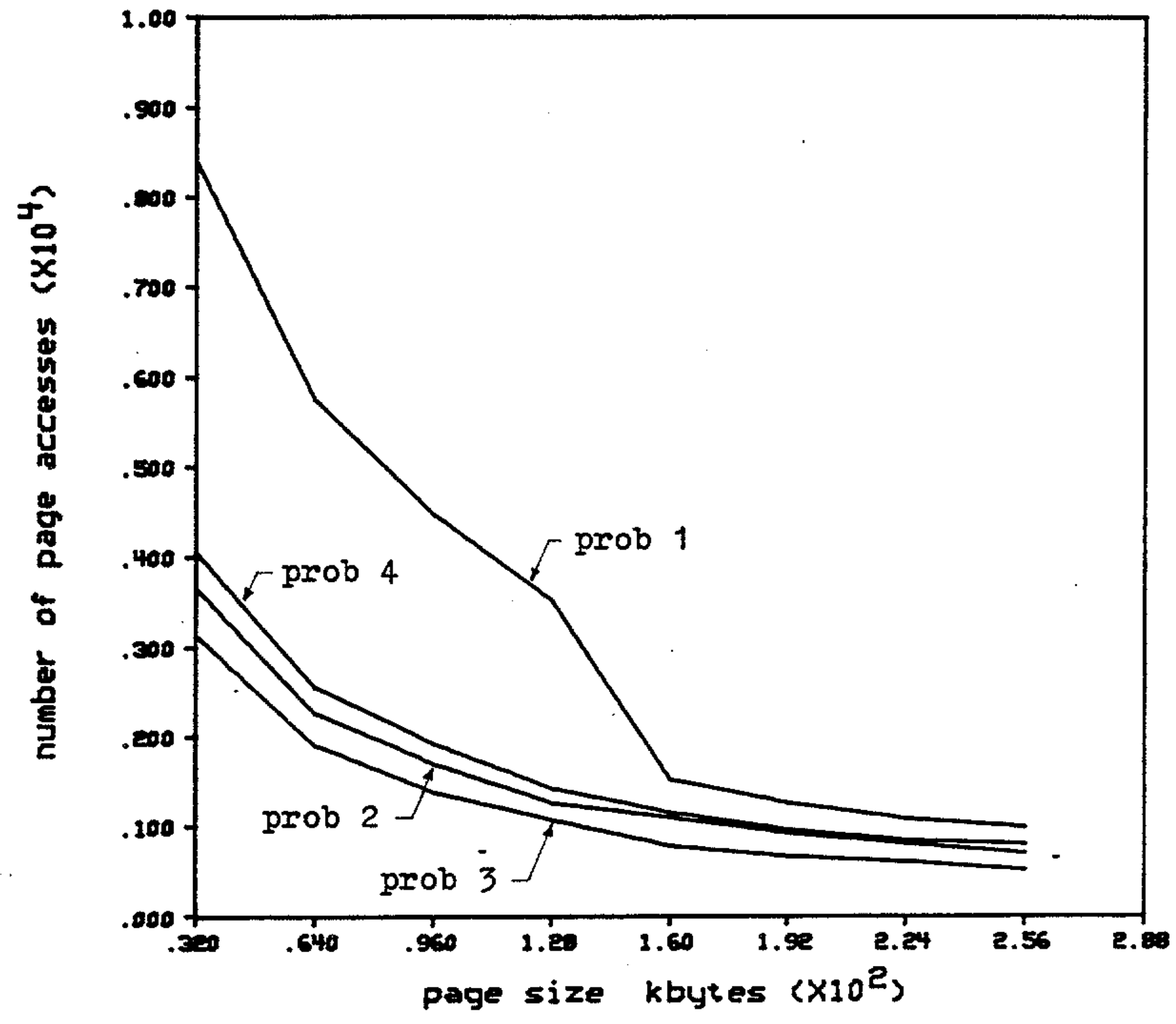


Figure 10 : Variation of the number of page accesses with page size for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $\phi = 120$, $\theta = n_p - \phi$)

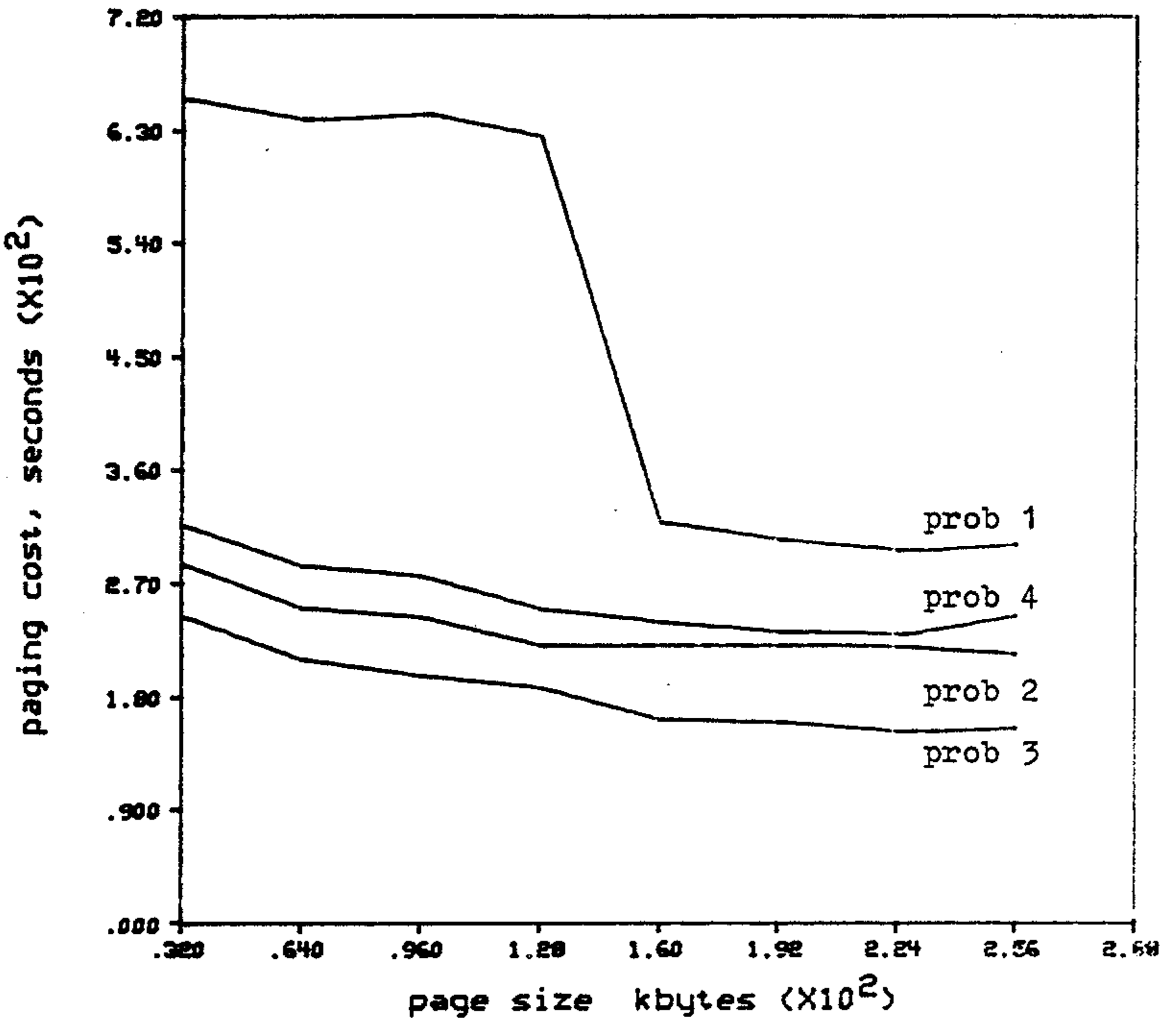


Figure 11 : Variation of the total paging cost with page size on a moving head disk (t_s = 28ms., t_r = 16.67ms.) with a disk density of 16 kbytes/track for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $\phi = 120$, $\theta = n_p - \phi$)

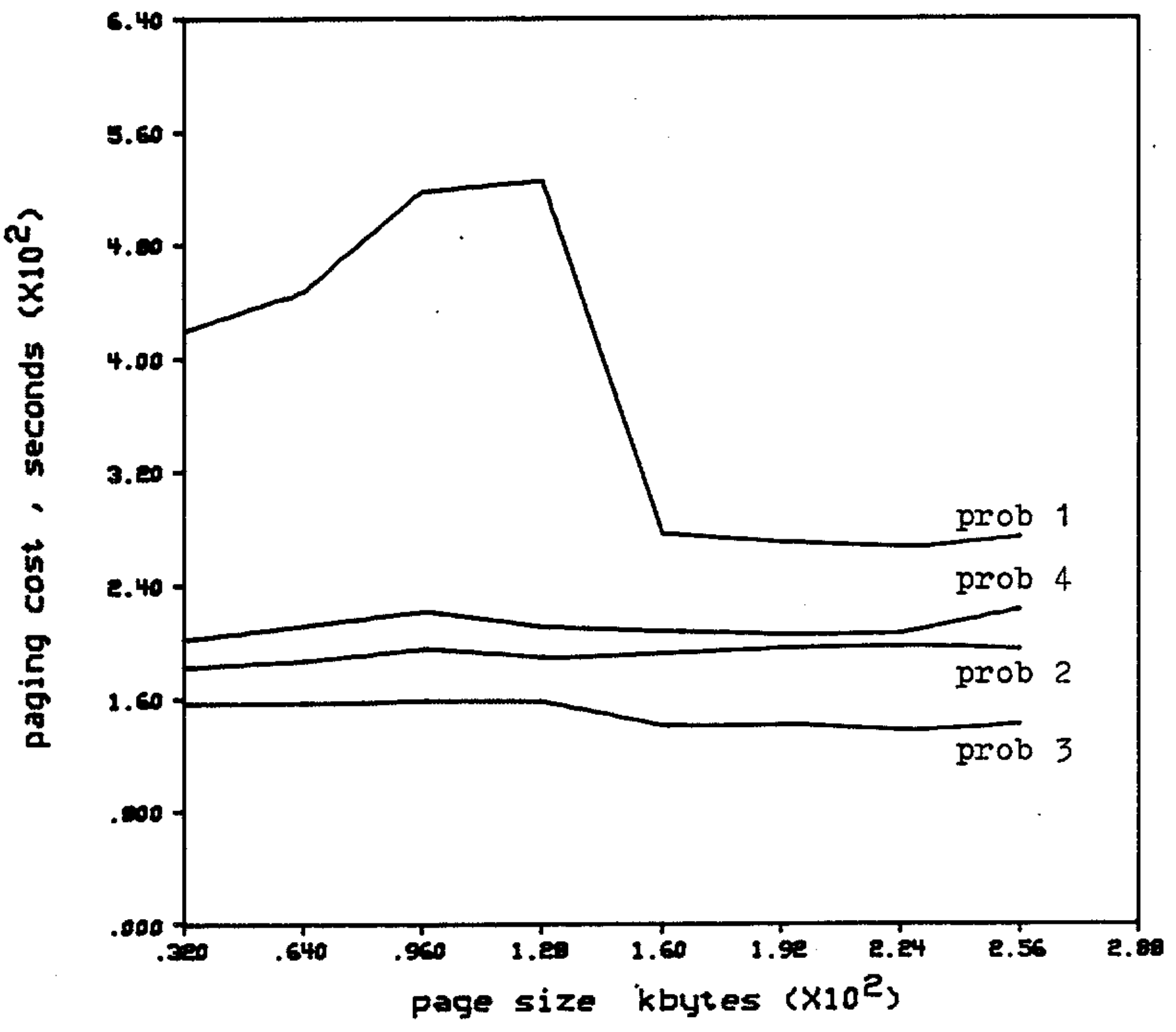


Figure 12 : Variation of the total paging cost with page size on a head per track disk (t_r = 16.67ms.) with a disk density of 16 kbytes/track for four 20 variables, 20 constraints integer programming problems (n_p = 132, ϕ = 120, $\theta = n_p - \phi$)

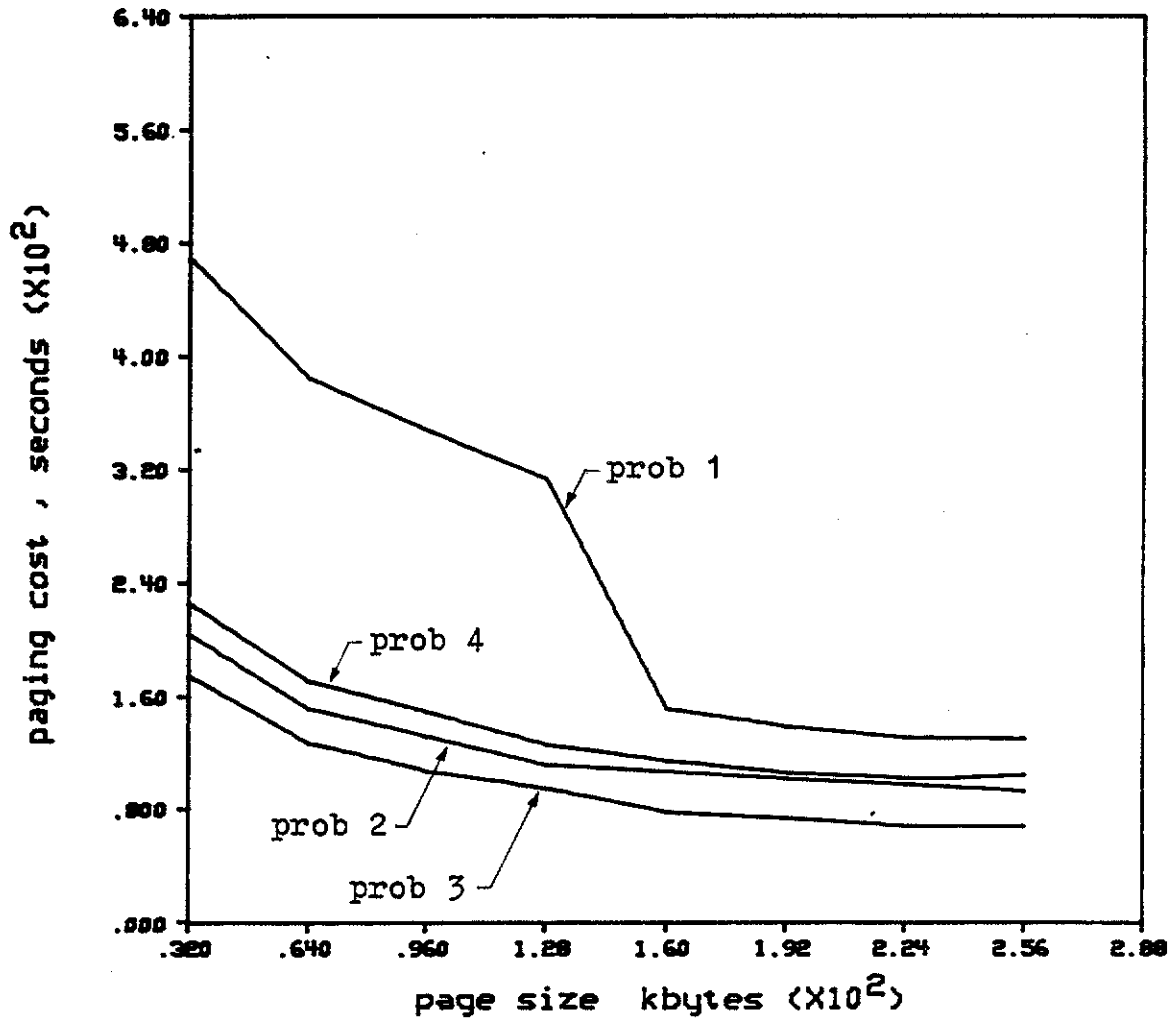


Figure 13 : Variation of the total paging cost with page size on a moving head disk (t_s = 28ms., t_r = 16.67ms.) with a disk density of 64 kbytes/track for four 20 variables, 20 constraints integer programming problems (n_p = 132, ϕ = 120, $\theta = n_p - \phi$)

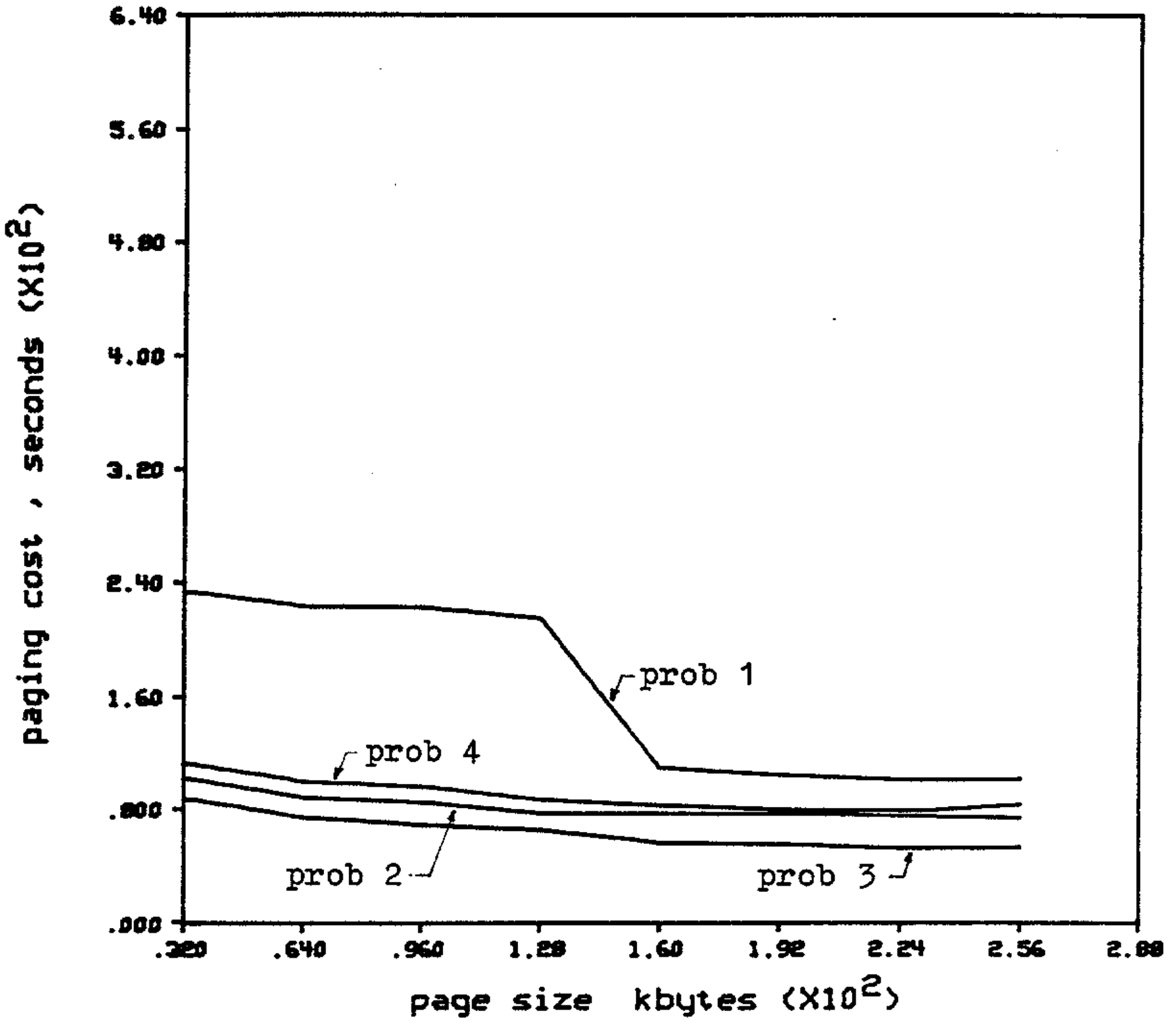


Figure 14 : Variation of the total paging cost with page size on a head per track disk (t_r = 16.67ms.) with a disk density of 64 kbytes/track for four 20 variables, 20 constraints integer programming problems ($n_p = 132$, $\phi = 120$, $\theta = n_p - \phi$)

An alternative is to delete the entire left-most page but load only the k subproblems with the smallest lower bounds. The remaining subproblems are left in a s-buffer in secondary storage. When the p-buffer is empty, an additional k subproblems are moved from the s-buffer to the p-buffer. Only when the s-buffer is empty, is the new left-most page deleted from the B^+-tree. When a replacement occurs while the s-buffer is not empty, the B^+-tree may have subproblems with lower bounds that are smaller than those in the s-buffer. The chance of this occurring is small, since the replacement rule will leave the subproblems with the smallest lower bounds in the main memory heap. Even if this occurs its effect should be minimal since the subproblems in the s-buffer will have to examined by the branch and bound process in the near future. This effect is only visible when the final solution is close to being generated. At worst, it will delay the solution by h_s iterations.

3.1.7 Performance of VM1

To compare the performances of VM1 with that of a general purpose virtual memory system using the LRU replacement algorithm, simulations were performed on a VAX 11/780 using integer programming problems.

The results for four 20 variables, 20 constraints integer programming problems are shown in Figures 15 - 18. Let the solution of a problem incur an average paging cost P_m on VM1 and P_l on the LRU virtual memory system. The relative performance, R_c, may be defined as

$$R_c = \frac{P_l}{P_m}$$

For a head per disk device, P_l is smaller than P_m at low disk densities. As the disk density increases, P_m decreases faster than P_l, so that P_m is smaller

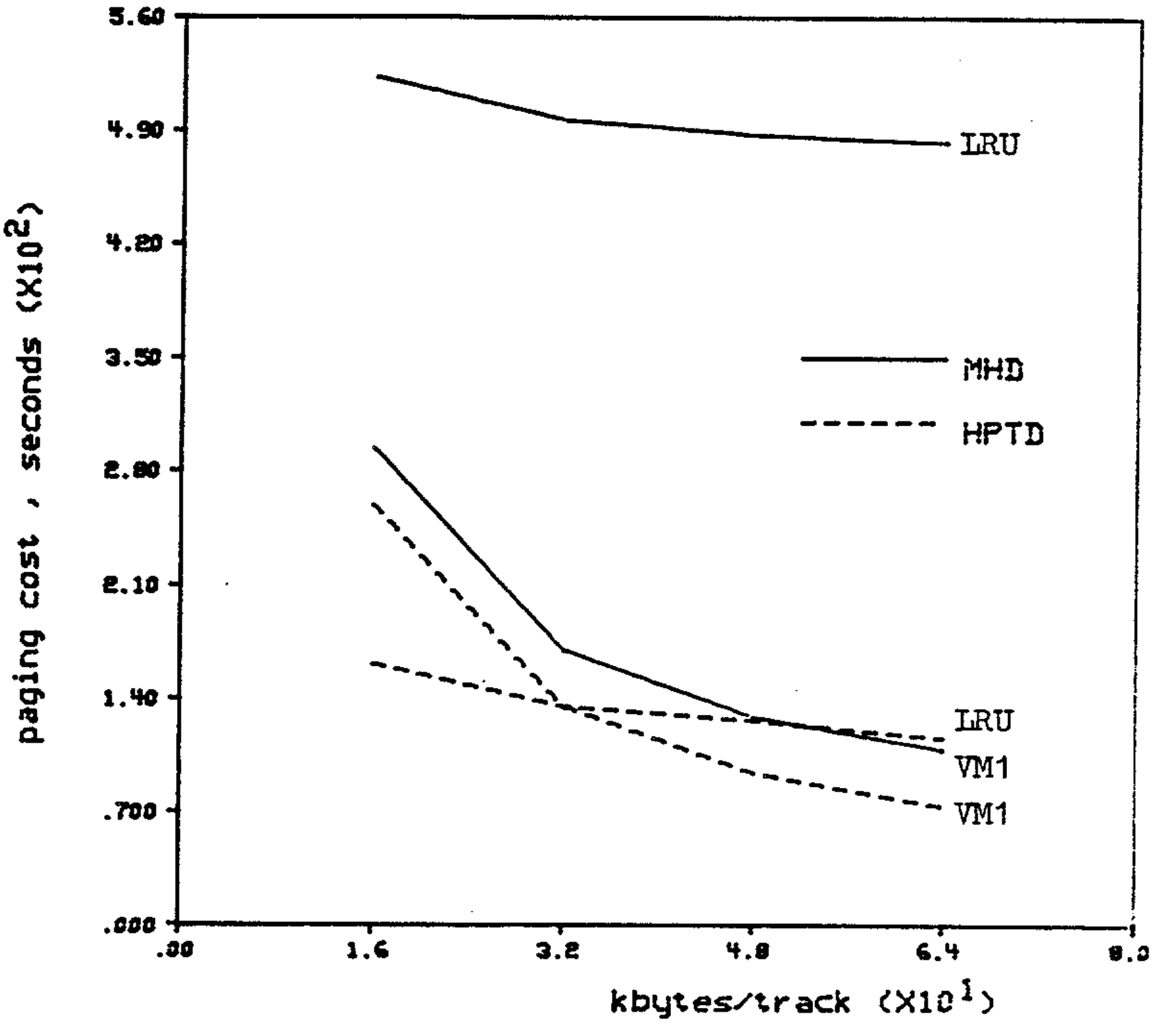


Figure 15 : Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 1) with disk density for a LRU virtual memory system and VM1 (n_p = 132, h_s = 100, ϕ = 120, $\theta = n_p - \phi$)

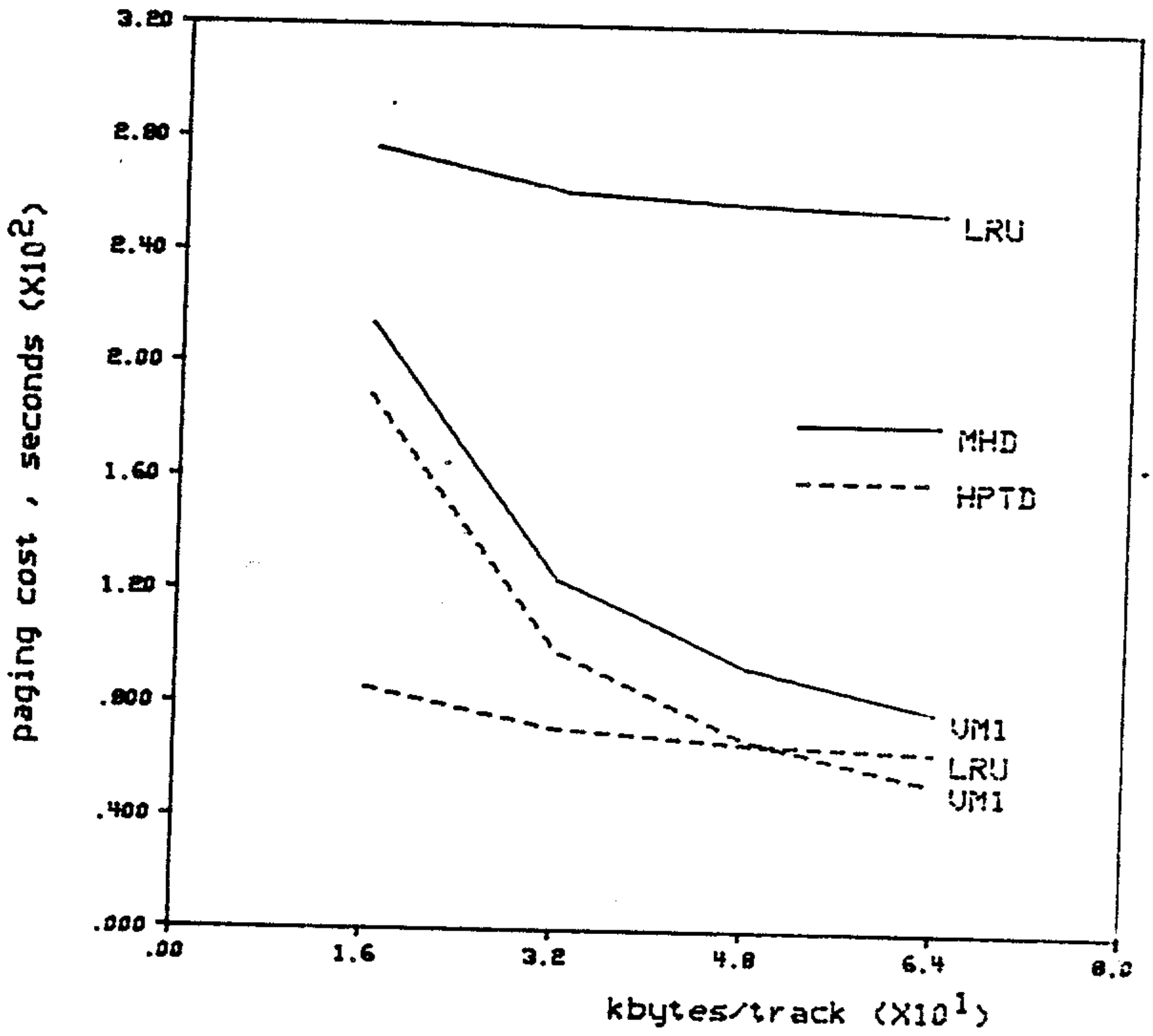


Figure 16 : Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 2) with disk density for a LRU virtual memory system and VM1 ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$)

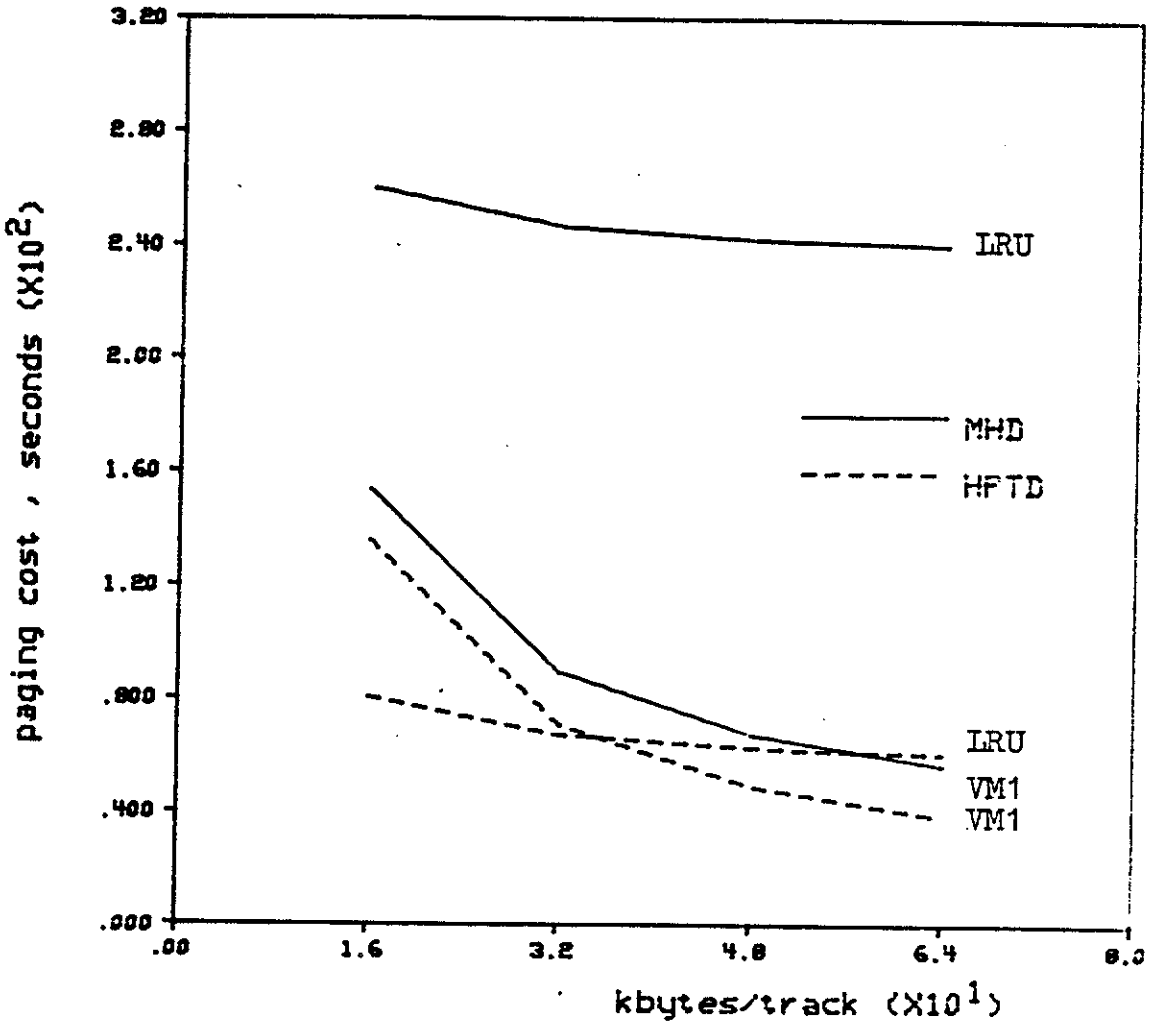


Figure 17 : Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 3) with disk density for a LRU virtual memory system and VM1 ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$)

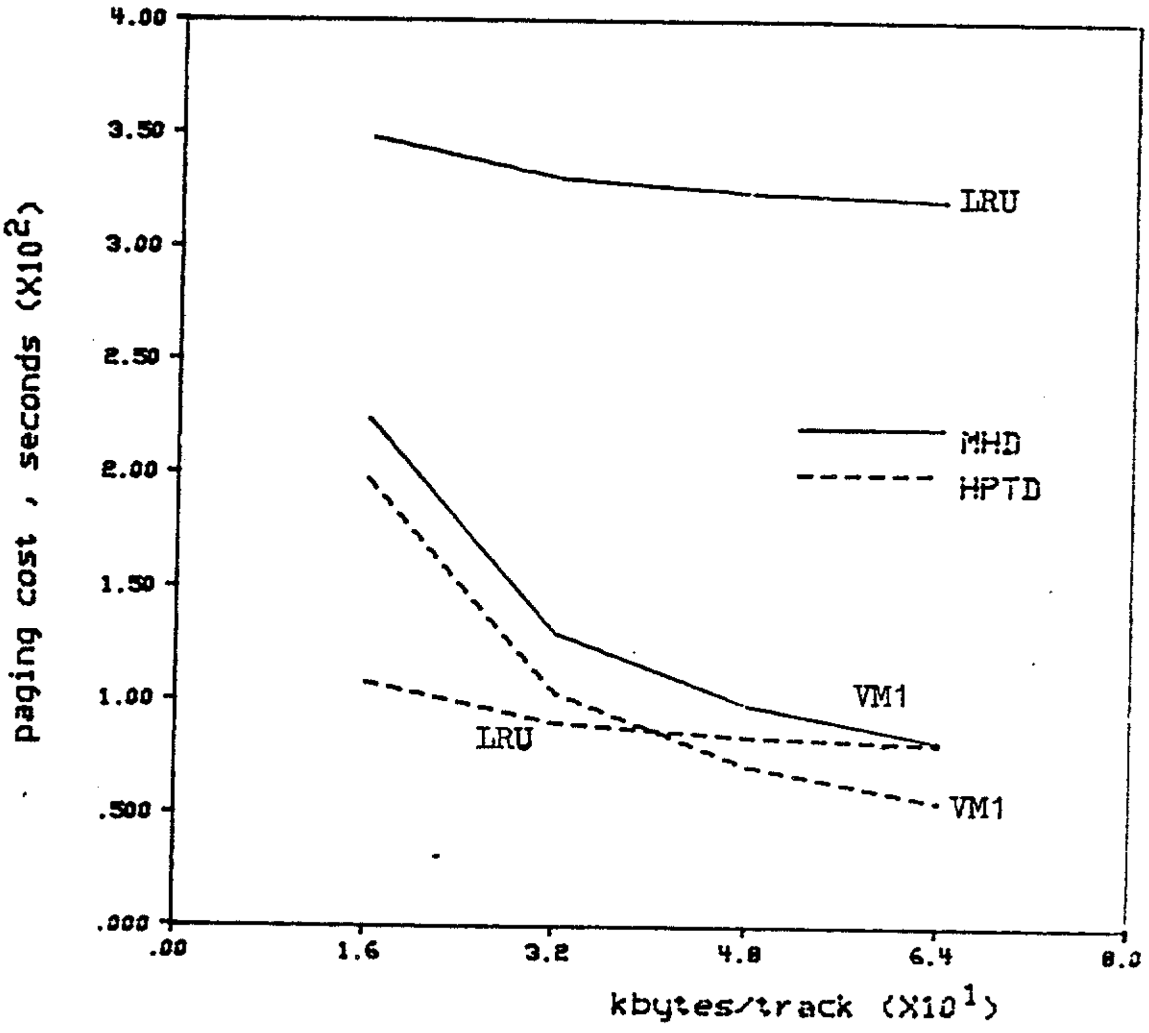


Figure 18 : Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 4) with disk density for a LRU virtual memory system and VM1 ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$)

than P_l at high disk densities. For a head per track device rotating at 3600 rpm, R_c increases from between 0.45 to 0.67 at 16 kbytes/track to between 1.18 and 1.28 at 64 kbytes/track. P_m becomes less than P_l at disk densities in the range 32 to 48 kbytes/track.

For a moving head disk, P_m is always less than P_l. For example, on a moving head disk with an average seek time of 28 ms and a rotational speed of 3600 rpm, R_c is the range 1.33 to 1.75 at 16 kbytes/track and between 3.23 to 4.35 at 64 kbytes/track.

These results indicate that VM1 is better than an LRU virtual memory system only for some disk characteristics. Even then, the improvement is much less than what was hoped for. The culprit is the high cost of each page access due to the large page size used. Statistics gathered during the simulations (Table 12) show that the number of page accesses have been reduced significantly. They also show that an average of only 8.5 subproblems are inserted into a page of size $h_s = 100$ during each replacement. This suggests that it is wasteful to perform insertions by reading a page into main memory, inserting the subproblems into the image of the page in main memory and writing the image back onto the disk. Modifications to reduce the cost of a page access will be considered in the next section.

3.2 A Modification of VM1 with One Subproblem Per Block (VM2)

The block size is a parameter that can be set by the operating system, subject to constraints like sector size and overall disk capacity. The block size is chosen to be the smallest number of sectors equal to or greater than a subproblem. A block is allowed to hold only one subproblem. Thus, each page consists of h_s successive blocks.

Table 12 : p_l, p_v and g_v for four 20 variables, 20 constraints integer programming problems

	Problem 1	Problem 2	Problem 3	Problem 4
p_l	12901	6820	6413	8579
p_v	1241	905	651	945
g_v	8.4	8.5	10.6	8.5

p_l =	number of page accesses for LRU virtual memory system.
p_v =	number of page accesses for VM1 ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$).
g_v =	average number of sub-problems inserted into a page during one replacement for VM1 ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$).

Next, the B^+-tree is further modified by removing the restriction that subproblems in a page have to be stored sequentially by increasing lower bounds. A subproblem is allowed to reside in any block as long as it is in the correct page.

Each page is assigned a unique number between 0 and $l_s - 1$, where l_s is the number of pages in the B^+-tree. A set of l_s status vectors is maintained in main memory. Status vector m is assigned to page m. Each status vector contains h_s bits. Bit n of status vector m will be set whenever block n of page m contains a subproblem, and cleared otherwise.

To insert j subproblems into page m the operating system uses the status vector m to select the j empty blocks into which the j subproblems are to be written. As soon as the write head is positioned over each of the chosen blocks, the corresponding subproblem is written onto the disk, and the status vector updated. Two approaches to the selection of the blocks have been considered:

- *Sequential allocation.* The j vacant blocks with the smallest block numbers are selected. This scheme ensures that the subproblems are kept in successive blocks. Since each subproblem must be written into a specified block, the average time required to insert j subproblems is $t_s + t_r + j\,t_b$.
- *First available allocation.* The read/write head is positioned over the cylinder that contains the page. Assuming that the operating system knows where it is on the disk, the status vector is scanned for the first j vacant blocks that will come under the write heads. Under this scheme, subproblems will occupy clusters (due to the batching of the insertions) scattered throughout the page. The time

required to insert j subproblems will depend on the distribution of subproblems in the page. The best performance is obtained when the j vacant blocks are the first j blocks that come under the head. Here the average time required is $t_s + j\,t_b$. The worst performance occurs when the j vacant blocks have the same angular position on different surfaces. In this case, $t_s + (j-1)t_r + t_b$ will be the average time required.

Since subproblems are not stored in sorted order in a page, loading of the p-buffer requires the selection of the k subproblems with the smallest lower bounds in the page. This may be done by building a selection tree of lower bounds of size k in main memory. The largest lower bound is removed from the selection tree when a new lower bound is inserted, and the corresponding subproblem is deleted from the p-buffer. On completion of this process, the selection tree will contain the k smallest lower bounds in the page and the sub-page buffer will contain the desired subproblems. The selection tree is then removed and the status vector is updated to reflect the subproblems which have been loaded into main memory. The average time required is $t_s + t_r + h_s t_b$.

An alternative is to modify the best-first search process of the branch and bound algorithm. The k subproblems residing in blocks with the smallest block numbers are loaded. Now, instead of partitioning the subproblem with the smallest lower bound, a subproblem with a lower bound that is at least the q smallest is partitioned, where q is the number of subproblems remaining in the page. The effect of this, at worst, is to increase the number of iterations by q. The mean time to load k subproblems into the p-buffer will be reduced to

$t_s + t_r + i\,t_b$ where i is the number of blocks (occupied or vacant) read before the k subproblems are loaded.

When an overflow occurs, before the overflowing page can be split, the median lower bound has to be determined. All subproblems with lower bounds less than the median lower bound remain in this page while those with lower bounds greater than the median lower bound are moved to a new page. Two disk-drives are needed to carry out this operation efficiently. Double buffering in main storage enables the reading of subproblems from the full page and writing of subproblems onto the new page to proceed simultaneously. Let the input and output disk buffers each have a size of g blocks. The splitting operation will have an average cost of $2\,t_s + 2\,t_r + 3\,h_s\,t_b$. On a single disk drive, the overhead is extensive. It will be necessary to read subproblems into the buffer, move the head to the cylinder containing the new page, write the subproblems in the buffer and move the heads back to the original cylinder.

Similarly, when an underflow occurs, before the actual redistribution of subproblems between two pages can be carried out, it is necessary to find the median lower bound which will divide the subproblems evenly among them. If the underflowing and the neighboring pages are on different disk drives, the use of double buffering in main memory allows the reading of subproblems from the neighboring page and the writing of subproblems into the underflowing pages to proceed concurrently. The average cost will then be $2\,t_s + 2\,t_r + 3\,h_s\,t_b$. When the underflowing page and the neighboring page reside on the same disk drive, two possibilities exist :

- The neighboring page is moved to another disk drive. The median lower bound is determined while this is being carried out. Ignoring the overhead of setting up a new page, this requires an average cost of $4\,t_s + 4\,t_r + 4\,h_s\,t_b$.
- Neither the underflowing page nor the neighboring page is moved. Since only one disk is used, there is no need to use double buffering. Thus, up to 2g subproblems can be in main memory buffers when the disk head is moved. Let the average number of subproblems moved during a redistribution be u subproblems. The the average time required will be

$$2\left\lceil\frac{u}{2g}\right\rceil(t_s + t_r) + 2\,h_s\,t_b + u\,t_b$$

For concatenation, subproblems are simply read from the underflowing page into the disk buffers in main memory and written into the neighboring page. If the pages are on different disk drives, an average cost of $2\,t_s + 2\,t_r + 2\,h_s\,t_b$ will be required. If they are on the same disk drive, and one page is moved to another disk drive the average time required is $4\,t_s + 4\,t_r + 4\,h_s\,t_b$ If no page is moved the average cost will be $2\left\lceil\frac{p}{2g}\right\rceil(s + r) + 2\,h_s\,t_b$ where p is the average number of subproblems in the underflowing page.

3.2.1 Performance of VM2

Figures 19 - 22 compares the paging costs incurred in VM2 with those incurred in a general purpose virtual memory system using the LRU replacement algorithm for four 20 variables, 20 constraints integer

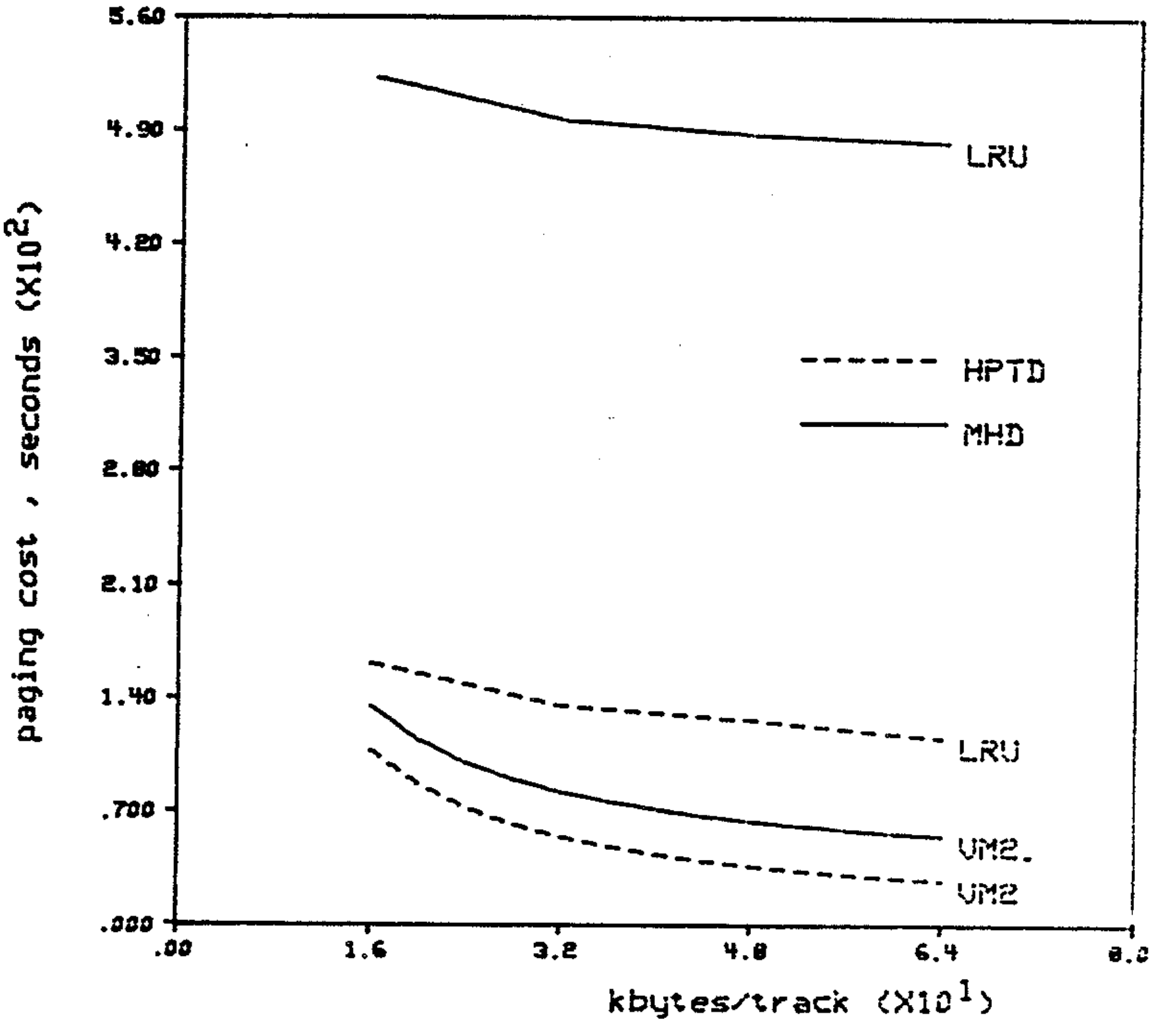


Figure 19 : Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 1) with disk density for a LRU virtual memory system and VM2 (n_p = 132, h_s = 100, ϕ = 120, $\theta = n_p - \phi$)

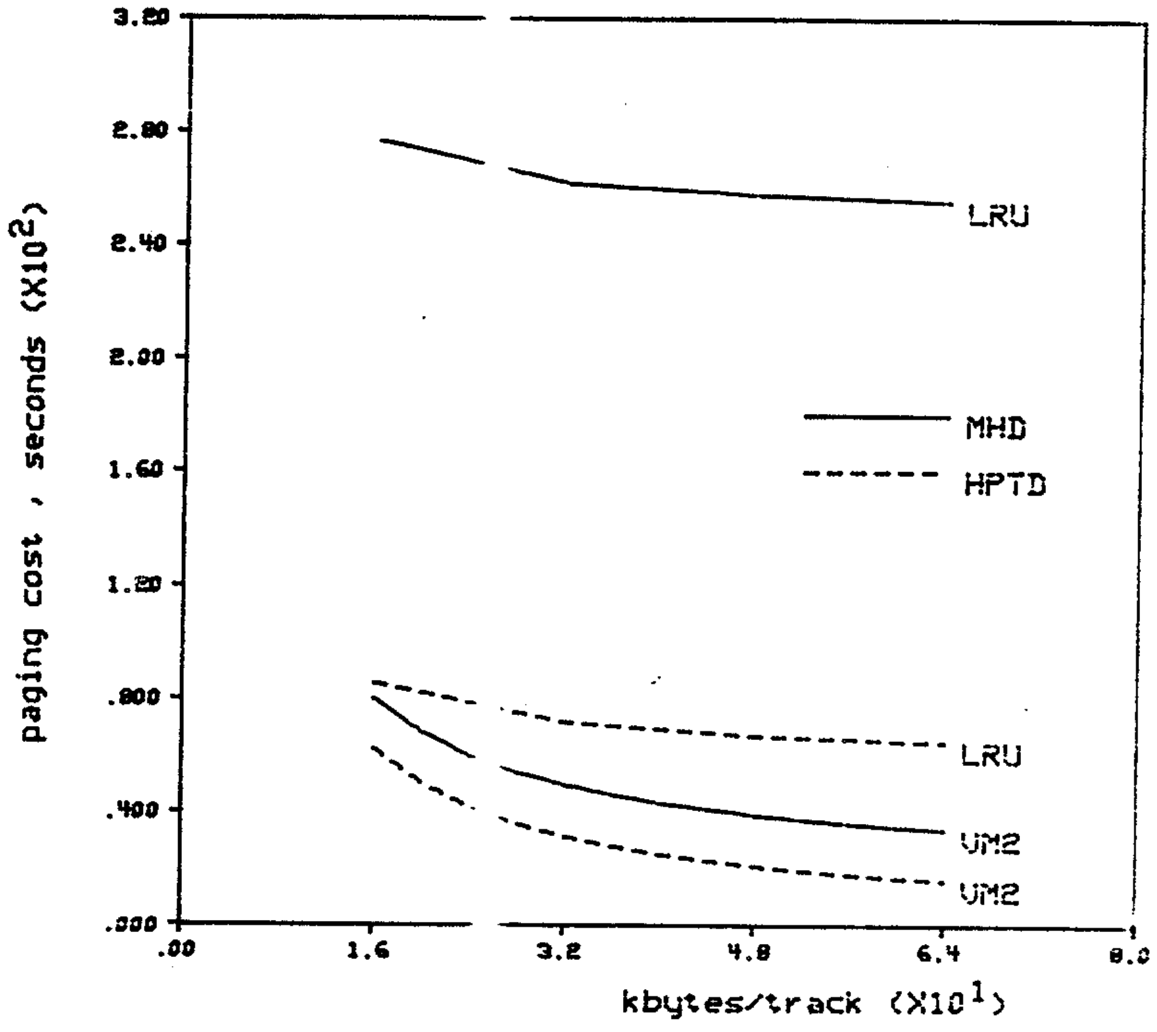


Figure 20 : Variation of the pa ng costs for a 20 variables, 20 constraints integer programming problem (pr blem 2) with disk density for a LRU virtual memory system and VM2 (n_p = 1 2, h_s = 100, ϕ = 120, $\theta = n_p - \phi$)

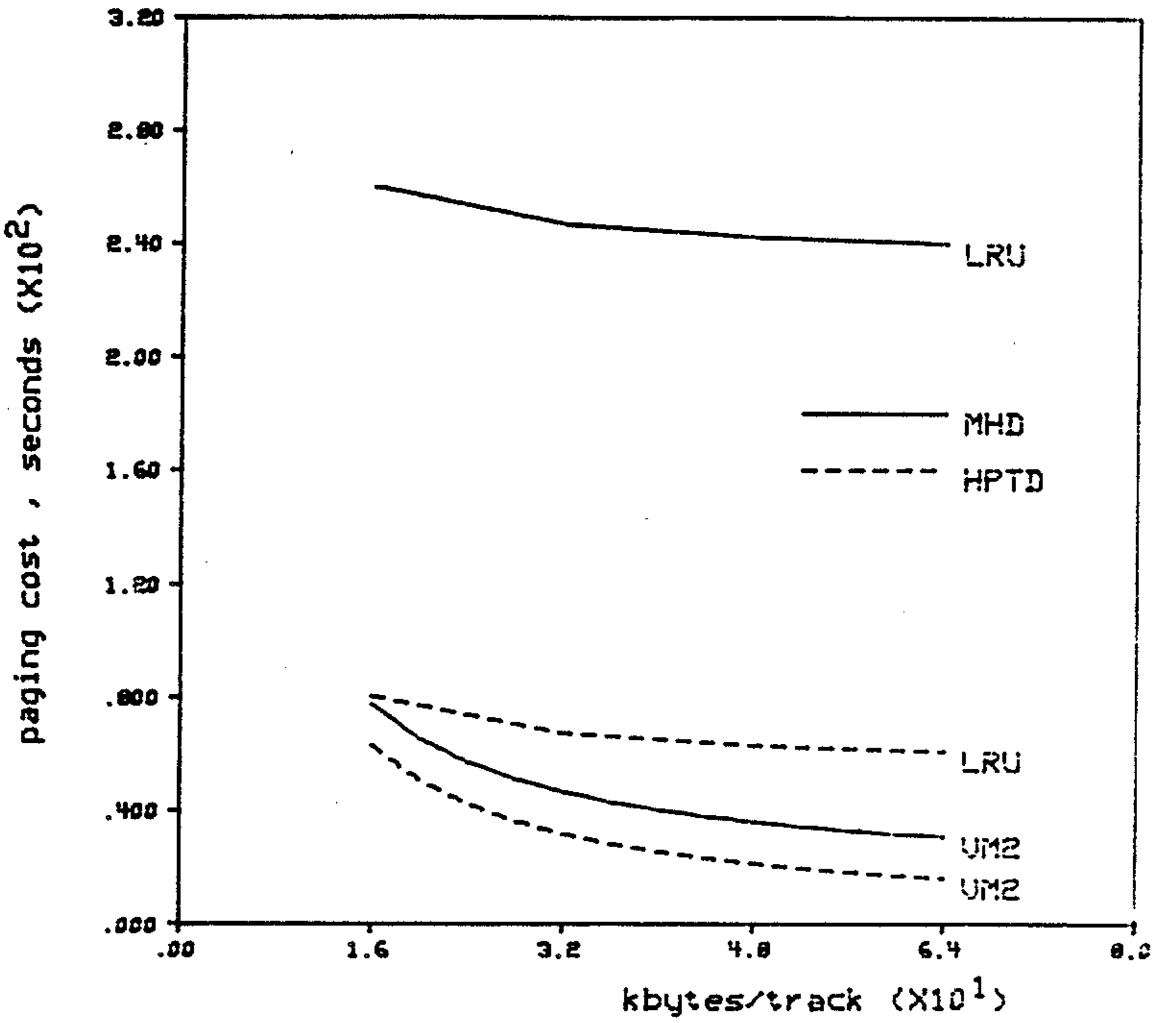


Figure 21 : Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 3) with disk density for a LRU virtual memory system and VM2 ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$)

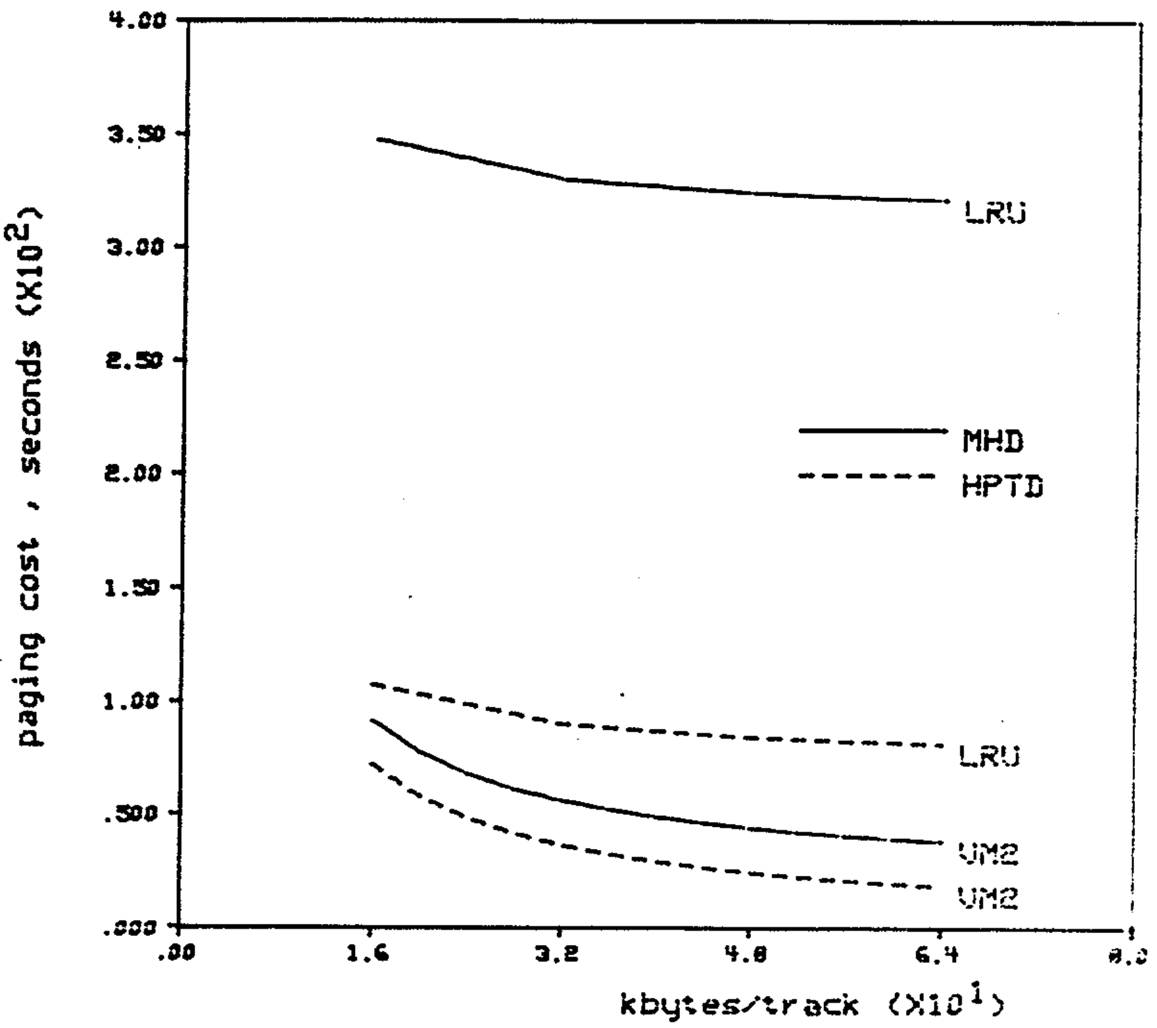


Figure 22 : Variation of the paging costs for a 20 variables, 20 constraints integer programming problem (problem 4) with disk density for a LRU virtual memory system and VM2 ($n_p = 132$, $h_s = 100$, $\phi = 120$, $\theta = n_p - \phi$)

programming problems. It can be seen that VM2 has a better performance than the LRU virtual memory system. The performance of VM2 also shows a greater improvement with increasing disk density than the LRU virtual memory system.

For a integer programming problem, let P'_m be the paging cost on VM2 and P_l the paging on a LRU virtual memory system. As before, the improvement R'_c is defined as

$$R'_c = \frac{P_l}{P'_m}$$

For a head per track device, R'_c increases from between 1.3 to 1.4 at 16 kbytes/track to between 3.7 and 4.0 at 64 kbytes/track. For a moving head disk the increase in R'_c is from between 3.2 and 3.9 at 16 kbytes/track to between 7.7 and 9.1 at 64 kbytes/track.

3.3 Conclusion

In this chapter,we have proposed a virtual memory system that is based upon the access characteristics of the branch and bound process. Simulations indicate that this virtual memory system is more effective than a general purpose virtual memory system with LRU replacement algorithm for branch and bound algorithms. The amount of improvement observed increases as disk density increases.

CHAPTER 4
CONCLUDING REMARKS

A general purpose locality based virtual memory system will not be able to provide an efficient virtual memory environment for the branch and bound algorithms as the algorithms do not exhibit strong locality. The conventional approach of modifying the algorithm so as to increase the amount of locality is not feasible.

The philosophically opposite approach of adapting the virtual memory system to the algorithm has to be used. Towards this end, a probabilistic model for the branch and bound process was proposed. The model consists of two walls approaching each other. The front wall represents the value of the lower bound of the subproblem currently being expanded. The back wall represents the minimum of all feasible solutions. These two walls approach each other and eventually coincide at the termination of the process.

Using this model, the position of the front and back walls and the distribution and number of active subproblems were derived. The model clearly shows that the subproblems constitute a dynamically varying list ordered by lower bounds. The access characteristics of the branch and bound algorithms call for the access of items at the head of this list and random insertion into the list.

The proposed virtual memory system partitions the subproblem list into two disjoint sublists, the primary list and the secondary list. The primary list resides in main memory and is implemented as a heap. The secondary list is organized as a B^+-tree Each leaf of the B^+-tree corresponds to a page. The index portion of the B^+-tree is kept in main memory and the leaves on secondary storage, except for the leaf containing the subproblems with the smallest lower bounds. A portion of this leaf is kept in main memory, since under best first search the subproblem with the smallest lower bound is always expanded.

Newly expanded subproblems are inserted into the primary list. When the primary list grows to its maximum size, n_p, subproblems are moved from the primary list to the secondary list. A suitable replacement algorithm is one which replaces the ϕ subproblems with the largest lower bounds. For the integer programming problem ϕ should lie between 0.7 n_p and 0.9 n_p and the page size between 0.65 n_p to 0.85 n_p. Simulations with integer programming problems show that this virtual memory system is more effective for branch and bound algorithms than a general purpose virtual memory system using the LRU replacement algorithm.

There remain several areas in which further work may be carried out. The variation of λ_p, λ_s and f_s with θ and ϕ needs to be analyzed. This will eliminate the dependence of the analysis of η upon values estimated from simulations and allow θ and ϕ to be optimized simultaneously.

This study has been based considerably upon the integer programming problem. Further investigation into other branch and bound algorithms is necessary. One of these is the vertex covering problem. Here, the lower bounds are integers and the subproblem size is significantly smaller. In addition, the manner in which feasible solutions are generated is different.

The branch and bound algorithms have an average time complexity that varies as the exponential of the input size. Two possible ways of reducing the time required are the parallel branch and bound algorithms and approximate branch and bound algorithms. Virtual memory support for these variants are also possible areas for future study.

LIST OF REFERENCES

LIST OF REFERENCES

[1] W. L. Eastman, "A Solution to the Traveling Salesman Problem", presented at the *American Summer Metting of the Econometric Society,* Cambridge, Mass., Aug. 1958.

[2] M. A. Efroymson and T. C. Ray, "A Branch and Bound Algorithm for Plant Location", *Operations Research,* Vol. 14, pp. 361-368, 1966.

[3] W. Feller, *An Introduction to Probability Theory and its Applications,* Vol. II, 2nd edition, John Wiley & Sons, Inc., 1971.

[4] R. S. Garfinkel and G. L. Nemhauser, *Integer Programming,* John Wiley and Sons, Inc., New York, 1972.

[5] R. Garfinkel, "On Partitioning the Feasible Set in a Branch and Bound Algorithm for the Asymmetric Travelling Salesman Problem", *Operations Research,* Vol. 21, No. 1, pp. 340-342, 1973.

[6] A. M. Geoffrion and R. E. Marsten, "Integer Programming Algorithms: A Framework and State-of-the-Art Survey", *Management Science,* Vol. 18, No. 9, pp. 465-491, May 1972.

[7] L. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees", *Proc. 19'th Symp. Foundations of Computer Science,* pp. 8-21, 1978.

[8] P. G. Hoel, S. C. Port and C. J. Stone, *Introduction to Probability Theory,* Houghton Mifflin Co., 1971.

[9] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Maryland, 1978.

[10] T. Ibaraki, "Computational Efficiency of Approximate Branch and Bound Algorithms", *Math. of Oper. Research,* Vol. 1, No. 3, pp. 287-298, 1976.

[11] T. Ibaraki, "Theoretical Comparisons of Search Strategies in Branch and Bound Algorithms", *Int. Jr. of Comp. and Info. Sci.,* Vol. 5, No. 4, pp. 315-344, 1976.

[12] T. Ibaraki, "Depth-m Search in Branch-and-Bound Algorithms", *Int. Jr. of Comp. and Inf. Sci.,* Vol. 7, No. 4, pp. 315-343, 1978.

[13] G. Ingargiola and J. Korsh, "A Reduction Algorithm for Zero-one Single Knapsack Problems", *Management Science,* Vol. 20, No. 4, pp. 460-663, 1973.

[14] G. Ingargiola and J. Korsh, "A General Algorithm for One Dimensional Knapsack Problems", *Operations Research,* Vol. 25, No. 5, pp. 752-759, 1977.

[15] D. E. Knuth, *The Art of Computer Programming, Sorting, and Searching,* Vol. 3, Addison-Wesley, 1973.

[16] W. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch and Bound Algorithms for Permutation Problems", *JACM,* Vol. 21, No. 1, pp. 140-156, 1974.

[17] B. Lageweg, J. Lenstra and A. Rinnooy Kan, "Job-shop Scheduling by Implicit Enumeration", *Management Science,* Vol. 24, No. 4, pp. 441-400, 1977.

[18] A. H. Land and A. Doig, "An Automatic Method for Solving Discrete Programming Problems", *Econometrica,* Vol. 28, pp. 497-520, 1960.

[19] Lawler, E. L. and Wood, D. W., "Branch and Bound Methods: A Survey", *Operations Research,* Vol. 14, pp. 699-719, 1966.

[20] J. Lenstra, "Sequencing by Enumerative Methods", *Math. Centre. Tract 69,* Mathematisch Centrum, Amsterdam, 1976.

[21] L. Mitten, "Branch and Bound Methods: General Formulation and Properties", *Operations Research,* Vol. 18, pp. 24-34, 1970.

[22] N. J. Nilsson, *Principles of Artificial Intelligence,* Tioga, Palo Alto, 1980.

[23] S. M. Ross, *Applied Probability Models with Optimization Applications,* Holden-Day, San Francisco, 1970.

[24] J. L. Baer, *Computer Systems Architecture,* Computer Science Press, Rockville 1980.

[25] P. J. Denning, "Virtual Memory", *Computing Surveys,* Vol. 2, No. 3, pp. 153-187, Sept. 1970.

[26] R. W. Doran, "Virtual Memory", *Computer,* pp. 27-37, Oct. 1976.

[27] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer", *IBM Systems Journal,* Vol. 5, No. 2, pp. 78-101, Jan 1966.

[28] P. J. Denning, "The Working Set Model for Program Behavior", *CACM,* Vol. 11, No. 5, pp. 323-33, May 1968.

[29] W. W. Chu and H. Opderdeck, "Program Behavior and the Page-Fault-Frequency Replacement Algorithm", *Computer,* pp. 29-38, Nov. 1976.

[30] J. E. Morrison, "User Program Performance in Virtual Storage Systems", *IBM Systems Journal,* Vol. 5, No. 2, pp. 216-237, 1966.

[31] A. C. McKellar and E. G. Coffman, Jr., "Organizing Matrices and Matrix Operations for Paged Memory Systems", *CACM,* Vol. 12, No. 3, pp. 153-164, 1969.

[32] P. C. Fisher and R. L. Probert, "Storage Reorganization Techniques for Matrix Computation in a Paging Environment", *CACM,* Vol. 22, No. 7, pp. 405-415, July 1979.

[33] B. S. Brown, F. G. Gustavson and E. S. Mankin, "Sorting in a Paging Environment", *CACM,* Vol. 13, No. 8, pp. 438-494, August 1970.

[34] A. J. Smith, "Sequentially and Prefetching in Database Systems", *ACM Trans. on Database Systems,* Vol. 3, No. 3, pp. 223-247, Sept. 1978.

[35] J. L. Baer and G. R. Sager, "Dynamic Improvement of Locality in Virtual Memory Systems", *IEEE Trans. on Software Engineering,* Vol. SE-2, No. 1, pp. 54-62, March 1976.

[36] D. J. Hatfield and J. Gerald, "Program Restructuring for Virtual Memory", *IBM Systems Journal,* Vol. 10, No. 3, pp. 168-192, 1971.

[37] D. Ferrari, "Improving Locality by Critical Working Sets", *CACM,* Vol. 17, No. 11, pp. 614-620, Nov. 1974.

[38] D. Ferrari, "Tailoring Programs to Models of Program Behavior", *IBM Journal of Research and Development,* Vol. 19, No. 3, pp. 244-251, May 1975.

[39] D. Ferrari, "The Improvement of Program Behavior", *Computer,* pp. 39-47, Nov. 1976.

[40] D. Comer, "The Ubiquitous B-Tree", *Computing Surveys,* Vol. 11, No. 2, pp. 121-137, June 1979.

APPENDICES

Appendix A
The B-Tree

A B-tree of order m is a search tree which is either empty or of height greater than or equal to one and satisfies the following properties:

(i) the root node has at least two children;

(ii) each node contains at most 2m keys and 2m+1 pointers;

(iii) each node contains at least m keys and m+1 pointers;

The beauty of a B-tree (and indeed all B-tree variants) lies in the methods for inserting and deleting records that always leave the tree balanced. A balanced tree is one which has all its leaves at the same depth. Then, the longest path in a B-tree of n records contains at most O(log n) nodes.

Insertion of a new record is a two step process. First, a find proceeds from the root to locate the proper node for insertion. The new record is inserted into the node and balance is restored. Referring to Figure 23(a), when a record with the key 57 is inserted into the B-tree, the find terminates unsuccessfully at the fourth leaf. Since the leaf can accept another record, the new record is simply inserted yielding the B-tree of Figure 23(b). If a record with key 72 is now inserted, there is no space for it in the appropriate leaf. Space has to be created for it by a split of the 2m+1 records, the m records with the smallest keys remain in the node, the m records with the largest keys are placed in another node and the remaining record is promoted to the parent node where it serves as a separator as shown in Figure 23(c). Usually the parent node can accommodate an additional record and the insertion process terminates. If the parent node happens to be full; then it has to be split. In the worst case, splitting propagates all the way to the root and the B-tree increases in height by one level.

Deletion also requires a find operation to locate the proper node. Once the record has been deleted, a check must be made to see if at least m records remain otherwise an underflow occurs. To restore balance, at least one record is needed and this may be obtained from a neighboring leaf. Since this operation requires at least two accesses to secondary storage, a better redistribution would evenly divide the remaining records between the two neighboring nodes. The deletion of the record with key 84 from the B-tree of Figure 24(a) does not cause an underflow and results in the B-tree of Figure 24(b). However, a further deletion of the record with key 64 causes an underflow and requires redistribution, resulting in the B-tree of Figure 24(c).

Redistribution is feasible only when there are at least 2m records to distribute. When less than 2m records remain, concatenation is performed whereby the records are combined into one of the nodes and the other is discarded. Since only one node remains, the record separating the two nodes in the parent is no longer necessary, it too is added to the single remaining leaf. This is turn may cause underflow and require redistribution from one of its neighbors. Concatenation occurs when the record with key 54 is deleted from the B-tree of Figure 24(c) resulting in the B-tree of Figure 24(d).

The process of concatenation may, thus, force a concatenation at the next higher level and so on to the root level. If the descendants of the root are

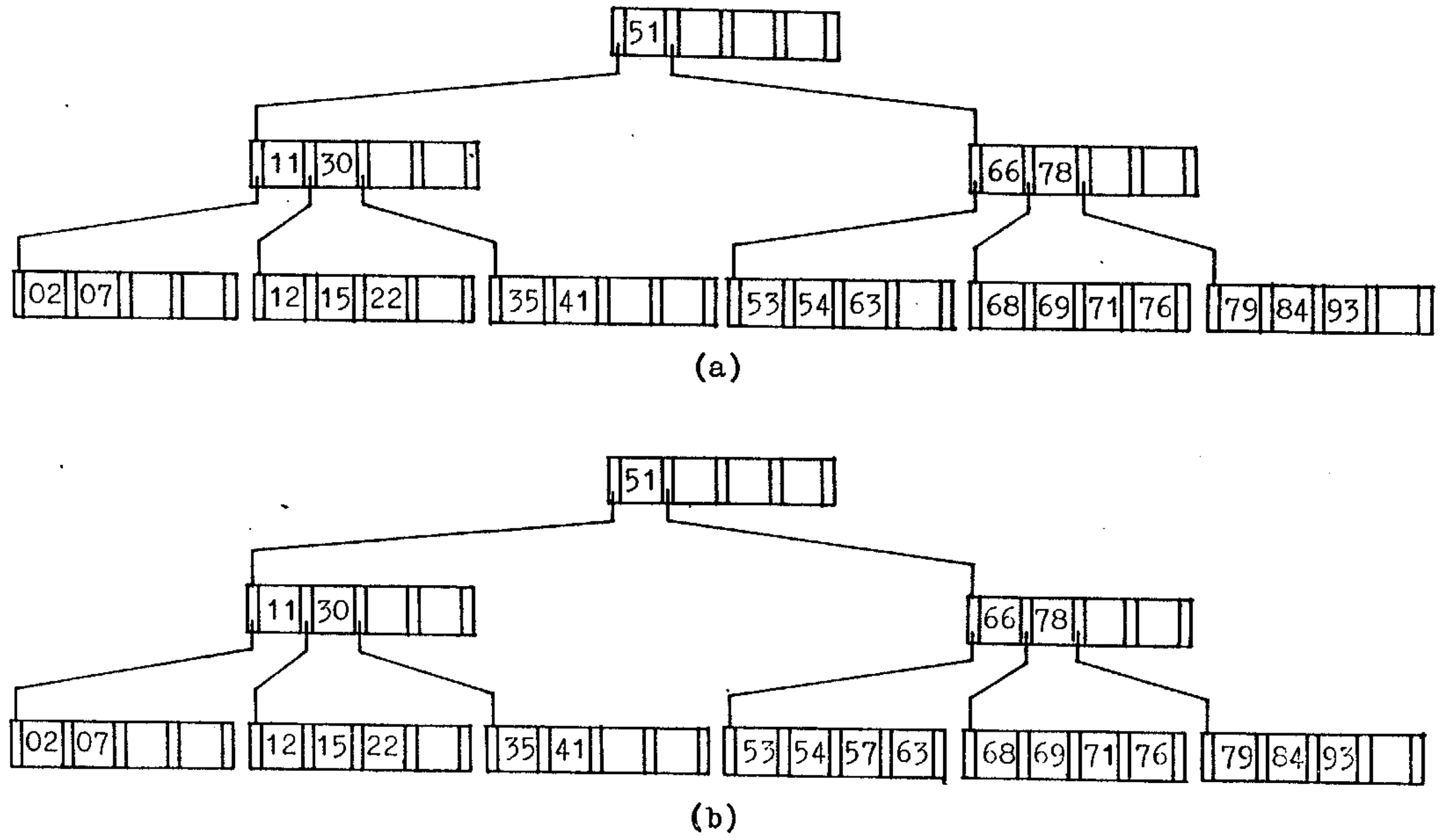
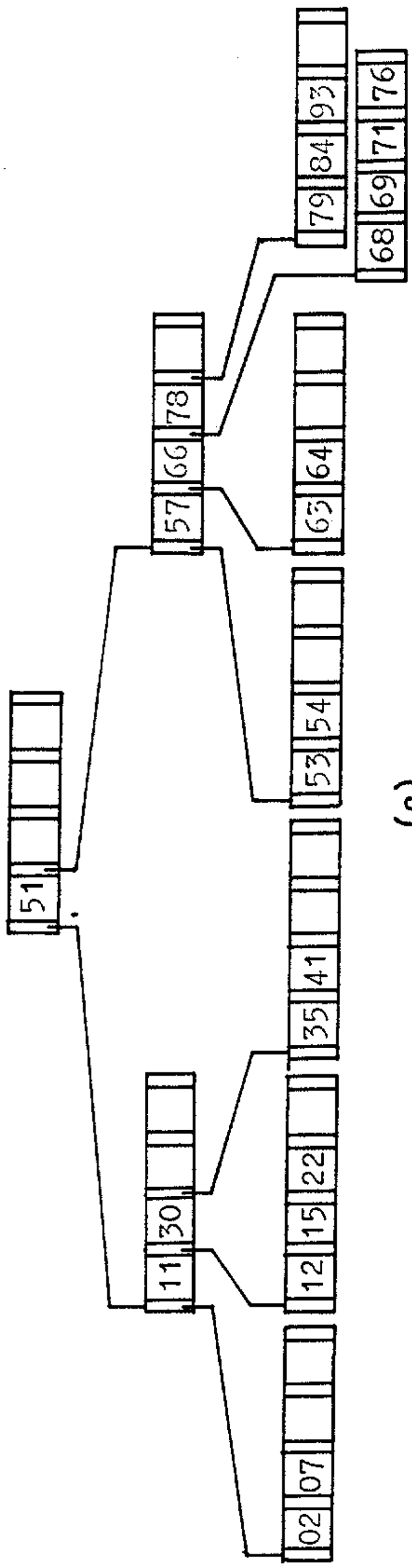
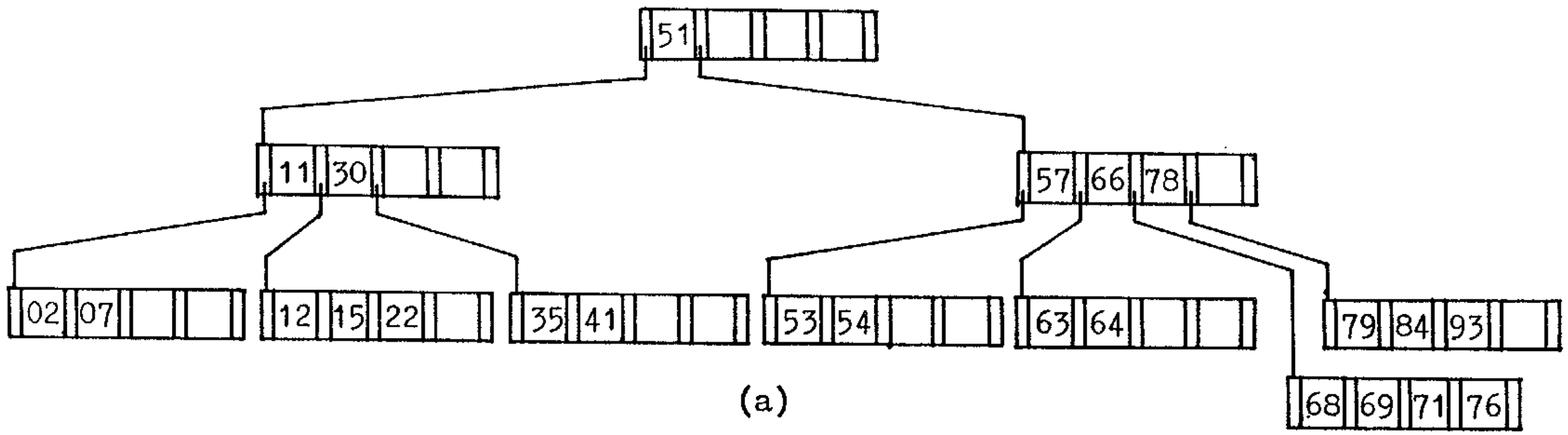


Figure 23 : (a) A B-tree, (b) the B-tree after the insertion of a record with key 57 and (c) the B-tree after the insertion of the record with key 64

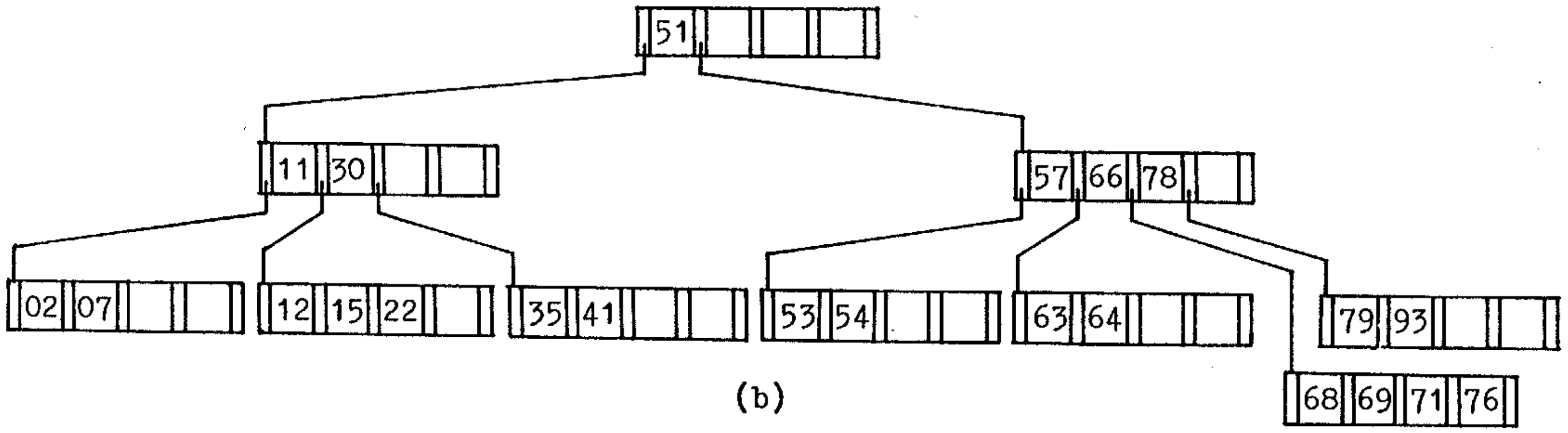


(c)

Figure 23, continued

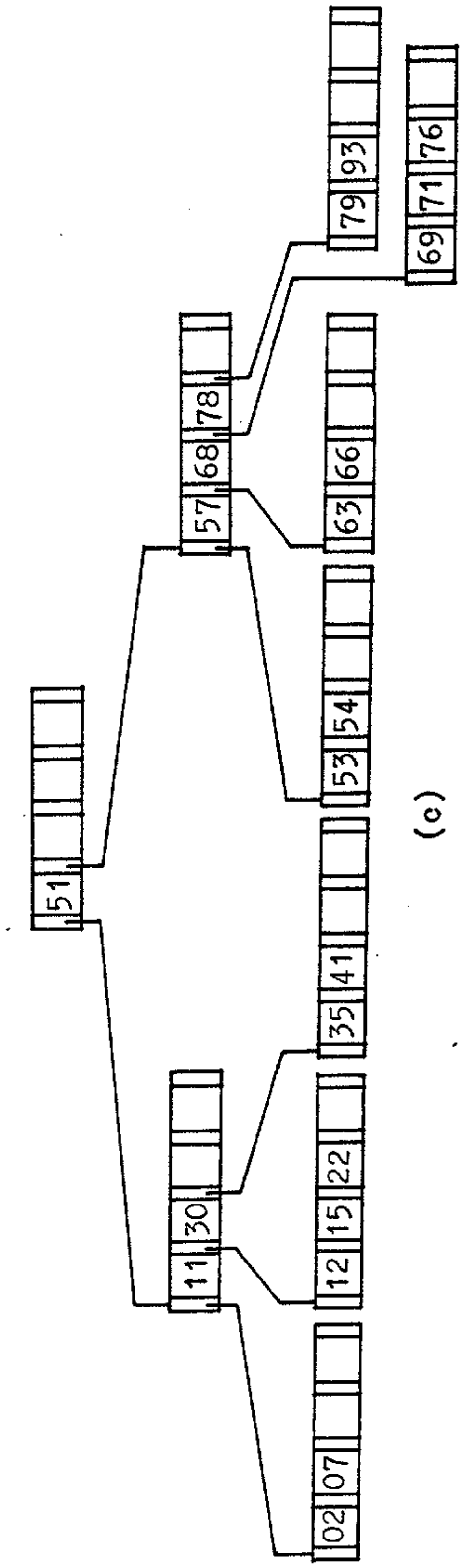


(a)

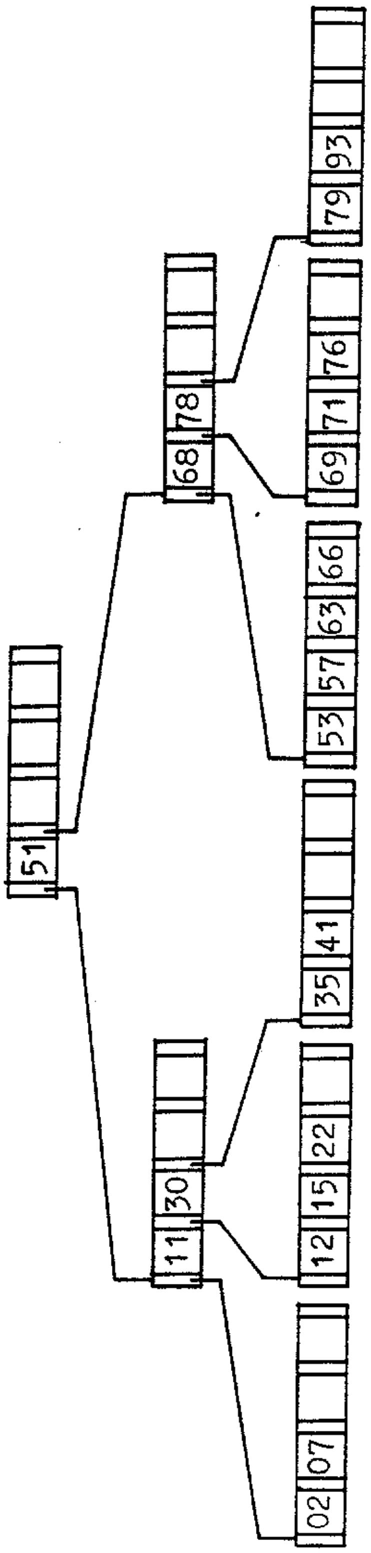


(b)

Figure 24 : (a) A B-tree and the B-trees resulting from the successive deletions of the records with keys (b) 84, (c) 64 and (d) 54



(c)



(d)

Figure 24, continued

concatenated they form a new root decreasing the height of the B-tree by one.

The cost of processing a find operation grows as the logarithm of the number of records in the B-tree, n. An insert or delete operation may require additional secondary storage accesses beyond the cost of the find operation as it progresses back up the B-tree. Overall the costs are at most doubled. Therefore, in a B-tree of order m, insertion and deletion take time proportional to $\log_2 n$ in the worst case.

As with most file organizations, variations of B-trees abound. One very popular variant is the B^+-tree. In a B^+-tree, all records reside in the leaves. The upper levels, which are organized as a B-tree, consist only of an index to enable rapid location of the records. Naturally, index nodes and leaf nodes may have different formats, or even different sizes. In particular leaf nodes are usually linked together left-to-right. This allows for easy sequential processing.

The record to be deleted must always reside in a leaf so its removal is simple. As long as the leaf remains half full, the index need not be changed. Of course, if an underflow condition arises, the redistribution or concatenation procedures may require adjusting values in the index as well as in the leaves.

Insertion and find operations in a B^+-tree are processed almost identically to insertion and find operations in a B-tree. When a leaf splits into two, instead of promoting the middle record, the algorithm promotes a copy of the key of the record, retaining the actual record in the right leaf. Find operations differ from those in a B-tree in that searching does not stop if a key in the index equals the query value. Instead, the nearest right pointer is followed and the search proceeds all the way to a leaf.

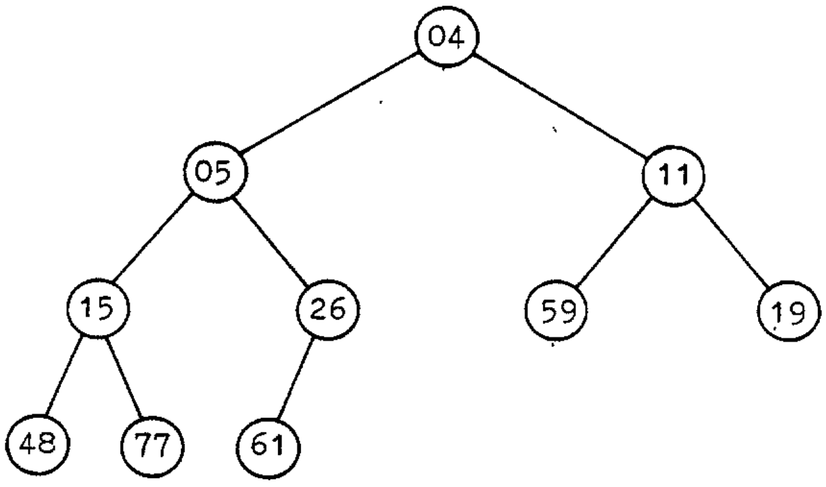
The B^+-tree retains the logarithmic cost properties for operations by key but gains the advantage of requiring at most O(1) access to satisfy a next operation. Thus, B^+-trees are well suited to applications which entail both random and sequential processing.

Appendix B
The Heap

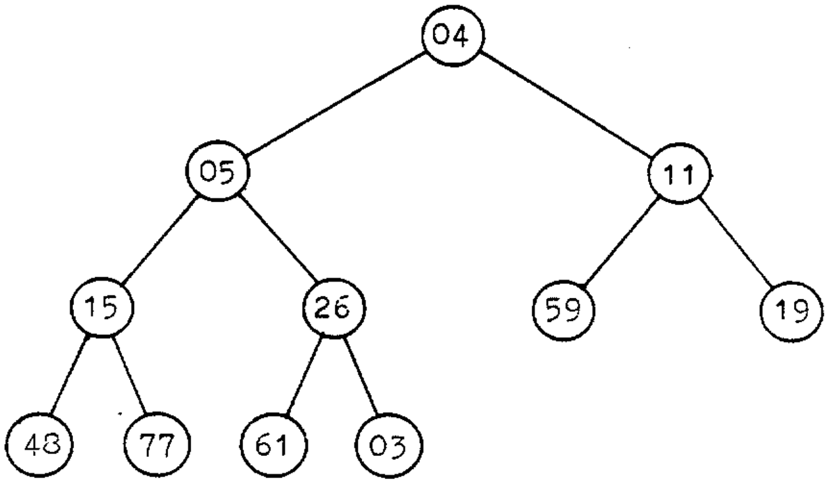
A heap is a complete binary tree with the property that the value of each node is at least as small as the value of its children nodes. The implies that the root of the heap has the smallest key in the tree. Insertion consists of adding a new node to the tree as a leaf and then restoring the heap property, which requires at most 0(log n) comparisons and record movements where n is the number of nodes in the heap. Figure 25 illustrates the insertion of a node with value 3 into a heap.

Retrieval of the subproblem with the smallest lower bound consists of copying the last node onto the root and then restoring the heap property. Figure 26 illustrates this process.

Furthermore, since the heap is a complete binary tree, it may be implemented as a sequential list as shown in Figure 27. This may be done by sequentially numbering the nodes, starting with nodes on level 1, then on level 2 and so on. Nodes on any level are numbered from left to right. These nodes may then be stored in a one dimensional array, with the node numbered i being stored in element i of the array.

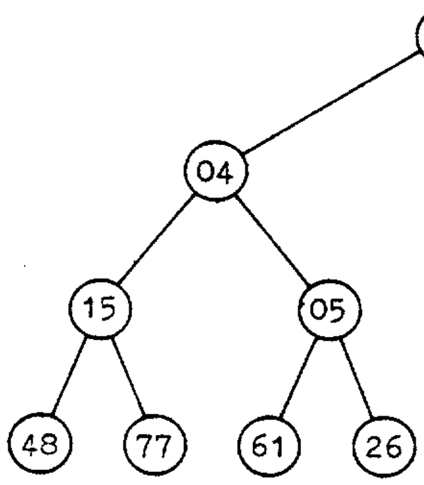


(a)



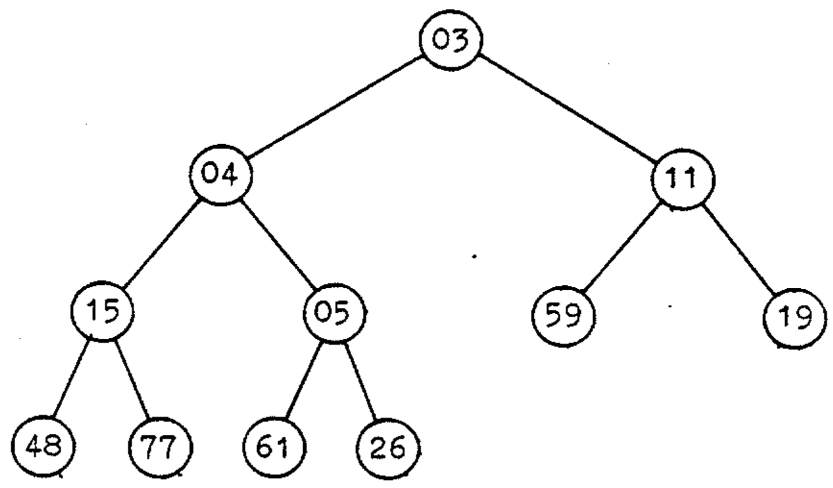
(b)

Figure 25 : (a) A heap, (b) the binary tree resulting from the insertion of a node with value 3 and (c) the resulting heap after restoring the heap property

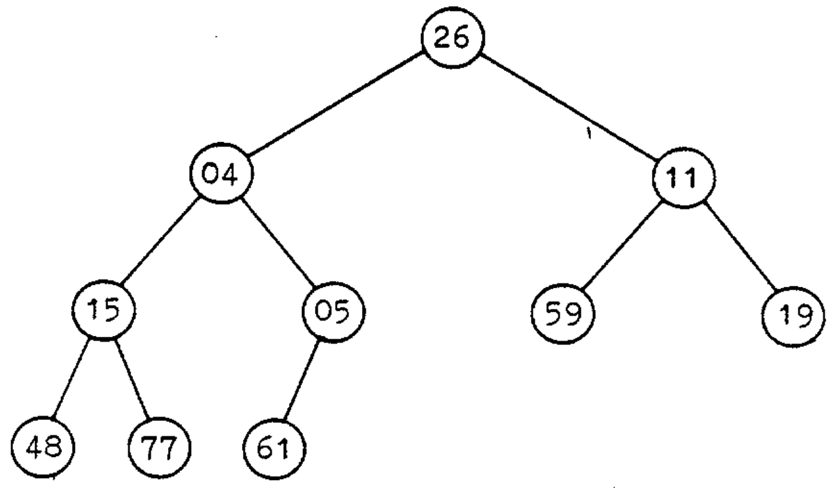


(c)

Figure 25, continued

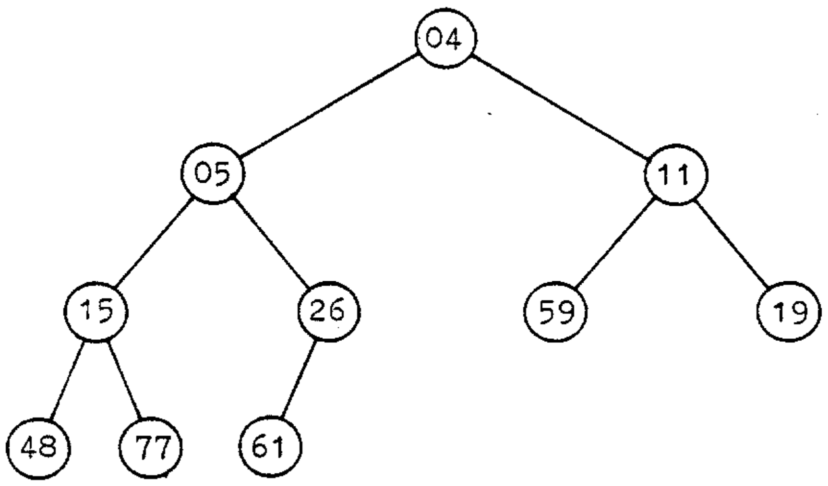


(a)



(b)

Figure 26 : (a) A heap, (b) the binary tree resulting from the deletion of the node with the smallest value and (c) the resulting heap after restoring the heap property



(c)

Figure 26, continued

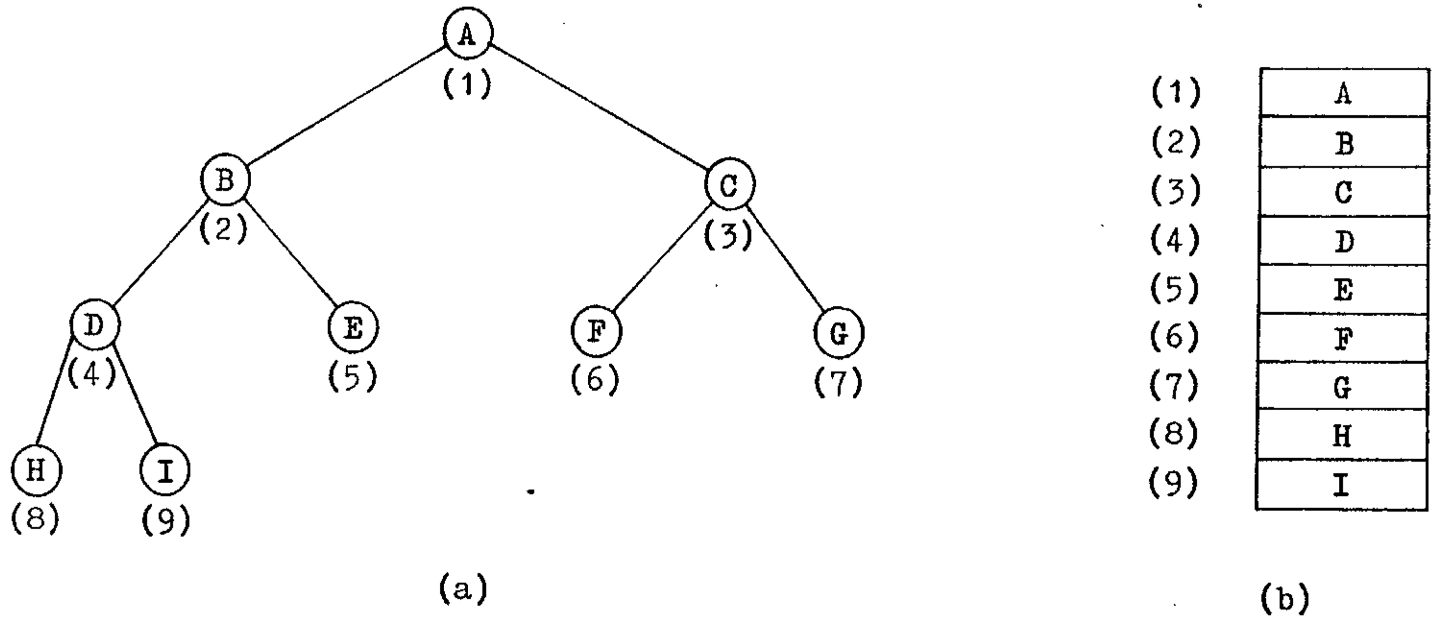


Figure 27 : (a) A heap and (b) its implementation as a sequential list

Appendix C
Maximum Number of Non-Leaf Nodes in a B^+-Tree

Let the order of the non-leaf nodes be m_1 and the order of the leaf nodes be m_2. Also let the number of records in the B^+-tree be n.

The B^+-tree has the maximum number of nodes when each non-leaf node contains only m_1 keys and (m_1+1) pointers and each leaf node contains m_2 records, with the exception of those nodes needed to maintain the balance of the B^+-tree. Let k be the height of the B^+-tree. Then, the maximum number of nodes in each level of the B^+-tree will be given by

Level	Maximum number of nodes
k	$\left\lfloor \frac{n}{m_2} \right\rfloor$
k - 1	$\left\lfloor \frac{1}{m_1+1} \left\lfloor \frac{n}{m_2} \right\rfloor \right\rfloor$
●	●
●	●
1	1

Allowing the simplification of removing the floor operators, the number of non-leaf nodes, g, will be given by

$$g = \frac{n}{m_2(m_1+1)} + \frac{n}{m_2(m_1+1)^2} + \cdots + \frac{n}{m_2(m_1+1)^{k-1}}$$

$$= \frac{n}{m_2(m_1+1)} \left(1 + \frac{1}{m_1+1} + \cdots + \frac{1}{(m_1+1)^k} \right)$$

$$= \frac{n}{m_2(m_1+1)} \frac{1 - \frac{1}{(m_1+1)^{k-1}}}{1 - \frac{1}{m_1+1}}$$

For $m_1 >> 1$ and $k >> 1$ the above simplifies to

$$g \simeq \frac{n}{m_2(m_1+1)} \frac{1}{1 - \frac{1}{m_1+1}}$$

$$= \frac{n}{m_1 m_2}$$

Appendix D

Expected Value of exp (- mV)

Let X_i, i = 1, 2, ..., N be N independent, identical, exponentially distributed random variables with density function $f(x) = me^{-mx}$ and $X_1 \leq X_2 \cdots \leq X_N$. For an arbitrary constant k, let V be the random variable such that $X_k \leq V$ and $X_{k+1} > V$. The expected value of e^{-mV} i.e. $E[e^{-mV}]$ will be derived as follows.

Suppose (k-1) random variables reside in the region $0 < x \leq v$ and 1 random variable in the region $v < x \leq v + \Delta v$. Denote the region $0 < x \leq v$ by γ and the region $v < x \leq v + \Delta v$ by $\Delta\gamma$. Let $u = Pr(0 < X_i \leq v)$ and $\Delta u = Pr(v < X_i \leq v + \Delta v)$. Then

$Pr\{(k-1) \text{ r.v.'s} \in \gamma \text{ and } 1 \text{ r.v.} \in \Delta\,\gamma\}$

$$= P\{(k-1) \text{ r.v.'s} \in \gamma\}\, P\{1 \text{ r.v.} \in \gamma \,|\, (k-1)\text{r.v.'s} \in \gamma\}$$

$$= \begin{pmatrix} N \\ k-1 \end{pmatrix} u^{k-1}[1-u]^{N-k+1} \begin{pmatrix} N-k+1 \\ 1 \end{pmatrix} \frac{\Delta u}{1-u} \left[1 - \frac{\Delta u}{1-u}\right]^{N-k}$$

$$\simeq \begin{pmatrix} N \\ k-1 \end{pmatrix} u^{k-1}[1-u]^{N-k+1} \begin{pmatrix} N-k+1 \\ 1 \end{pmatrix} \frac{\Delta u}{1-u}$$