81-006

OPTIMAL STORAGE MANAGEMENT IN A CELLULAR COMPUTER

Donald F. Stanat
Gyula A. Mago

Index Terms: Cellular computers, divide and conquer algorithm, functional programming, network of microprocessors, optimal algorithms, parallel processing, reduction languages, storage management.

This work was supported by NSF Grant MCS78-02778. Final preparation of the manuscript was done while the first author was a Visiting Scientist at the IBM San Jose Research Laboratory.

The authors are with the Department of Computer Science, University of North Carolina at Chapel Hill, NC 27514.

ABSTRACT

Optimal Storage Management in a Cellular Computer

Donald F. Stanat and Gyula A. Magó
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

We consider a class of machines in which a program is an expression stored in a linear array of cells, one expression symbol per cell. Empty cells can occur within and at the ends of the expression. Program execution requires shifting the symbols within the array (without changing their relative order) to provide specified numbers of contiguous empty cells adjacent to each symbol of the expression. Because symbols move simultaneously along the linear array of cells, an optimal pattern of movement is one that minimizes the maximum distance moved by any symbol. In the machine considered here, the expression array consists of the (connected) leaves of a full binary tree of cells. We describe an algorithm that uses the tree network to compute an optimal pattern of symbol movement and does so in time proportional to the height of the tree. The algorithm is asymptotically optimal for the given network topology.

I. Introduction

In recent years, Backus [1,2] has introduced several classes of programming languages designed to overcome some of the objectionable features of contemporary high level programming languages such as FORTRAN and ALGOL. The interested reader is referred to the 1977 Turing Lecture [2] for a description of the state of Backus's work as of that time. Although Backus has proposed a number of language classes in the publications cited, their distinctions do not concern us here and we will refer to them simply as _reduction languages_.

Magó recently proposed a basic machine architecture capable of executing reduction language programs efficiently; we will call these _reduction language machines_. The architecture of these machines is well-suited to reduction languages and exploits current electronic technology in a manner unavailable to von Neumann computers. The interested reader is referred to Magó [3] for a definitive description. The work described here grew out of the storage management problem in the reduction language machines described in that paper.

II. Reduction Languages and Reduction Language Machines

A reduction language program is an expression, similar to an algebraic expression, but with more powerful operations. A reduction language program is executed by evaluating the expression. This is done by locating _innermost applications_,

which are subexpressions of the program that can be evaluated independently of the remainder of the program. Evaluating one innermost application will, in general, create others, just as replacing the innermost expression of an algebraic expression by its value will create new innermost expressions. When no unevaluated innermost applications remain, execution of the program is complete. Because reduction languages admit a variety of powerful operators, the value of an innermost application may be either longer or shorter than the application itself and the process of evaluating a single innermost application may result in the creation and evaluation of many more applications. Thus, a program can expand and contract during execution.

One of the most attractive aspects of reduction languages is that all innermost applications are disjoint and are evaluated independently of the remainder of the program. Moreover, these languages have the Church-Rosser property, that is, the result of program execution does not depend on the order in which innermost applications are evaluated. This implies that all innermost applications of a reduction language program can be evaluated concurrently. Because reduction language machines are capable of evaluating arbitrarily many innermost applications concurrently, they exploit the potential parallelism of reduction languages in a natural way.

A reduction language machine is essentially a full binary tree of microprocessors with connections between adjacent leaf cells as shown in Figure 1. The leaf cells form a linear array

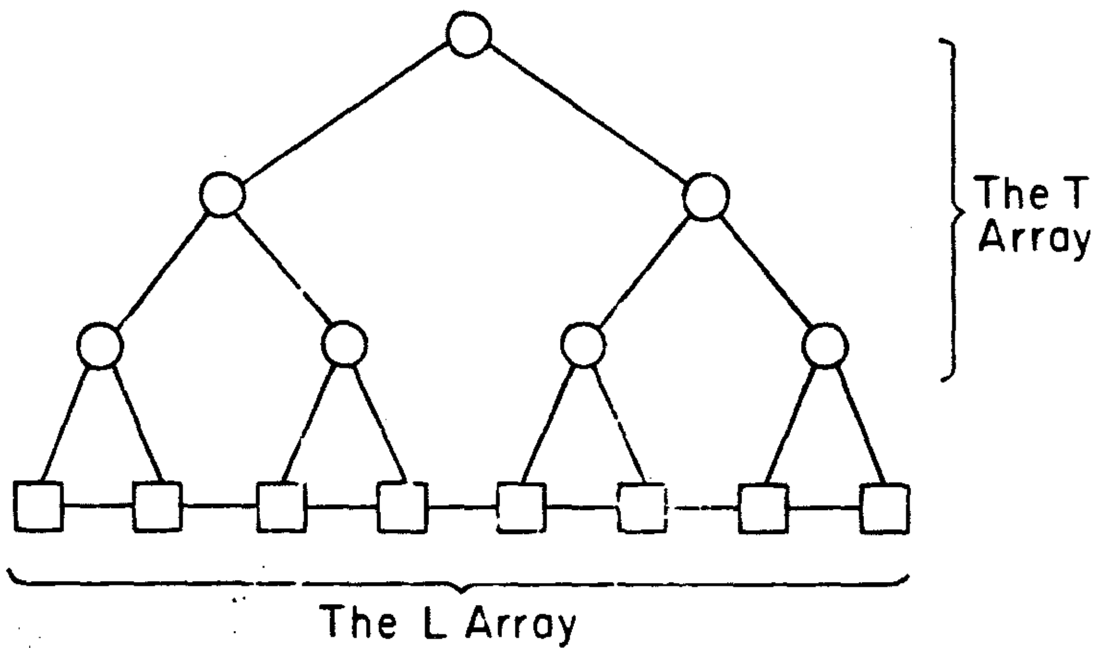


Figure 1: Interconnection pattern of a reduction language machine. Circular nodes denote T cells; square nodes denote L cells.

** Figure 1 goes here **

called the L array (for <u>Leaf</u> or <u>Linear</u>). The internal cells form a binary tree called the T array (for <u>Tree</u>). All the L cells are identical except for those on the right and left ends; all the T cells are identical except for the root cell which handles a variety of special tasks.

A program for a reduction language machine resides in the cells of the L array, one symbol per cell, stored in a left-to-right order. The text is not required to occupy contiguous cells of the L array; empty L cells may be interspersed among the occupied ones. Execution can be described briefly as follows: By communicating among themselves, the cells of the T and L networks determine the locations of the innermost applications. Then the processing capabilities of the tree of T cells are partitioned so that each innermost application has a subtree of the T network dedicated to it. The subtree then evaluates the innermost application. Evaluation of an innermost application requires a number of steps, including determining the operation to be performed, finding the operands, and replacing the expression by its value. Because the value of the innermost application may require more space than the expression itself, processing an innermost application often requires that additional space (in the form of empty cells of the L array) be provided in specific locations within and adjacent to the innermost application. The required space is provided by shifting contents of occupied cells

to the left or right within the L array. This movement, of course, must occur without overwriting any program text symbols, and must not usurp space required for the evaluation of other innermost applications.

III. The Storage Management Problem

Storage management is the repositioning of the contents of occupied cells of the L array to allow the evaluation of all innermost applications. For our purposes, the storage management problem can be described as follows: The L array contains a sequence of symbols, those of a reduction language program, in some of its cells. These symbols are stored in a left-to-right order, one symbol per cell, but not necessarily in contiguous cells. Requests for storage are made only by symbols of the program text. Each symbol requests a specific number of empty L cells to be provided on its immediate right and immediate left. A symbol may request a different number of cells on its right and left, and the number of cells requested may be zero. Space (in the form of empty cells) is to be provided only by shifting symbols (that is, the contents of occupied cells) so that empty cells are positioned in a way which satisfies all requests.

Definition 1: A storage management problem consists of a finite sequence of cells, some of which are occupied, such that the symbol of each occupied cell has two associated nonnegative integers denoting its requests for empty cells on the left and

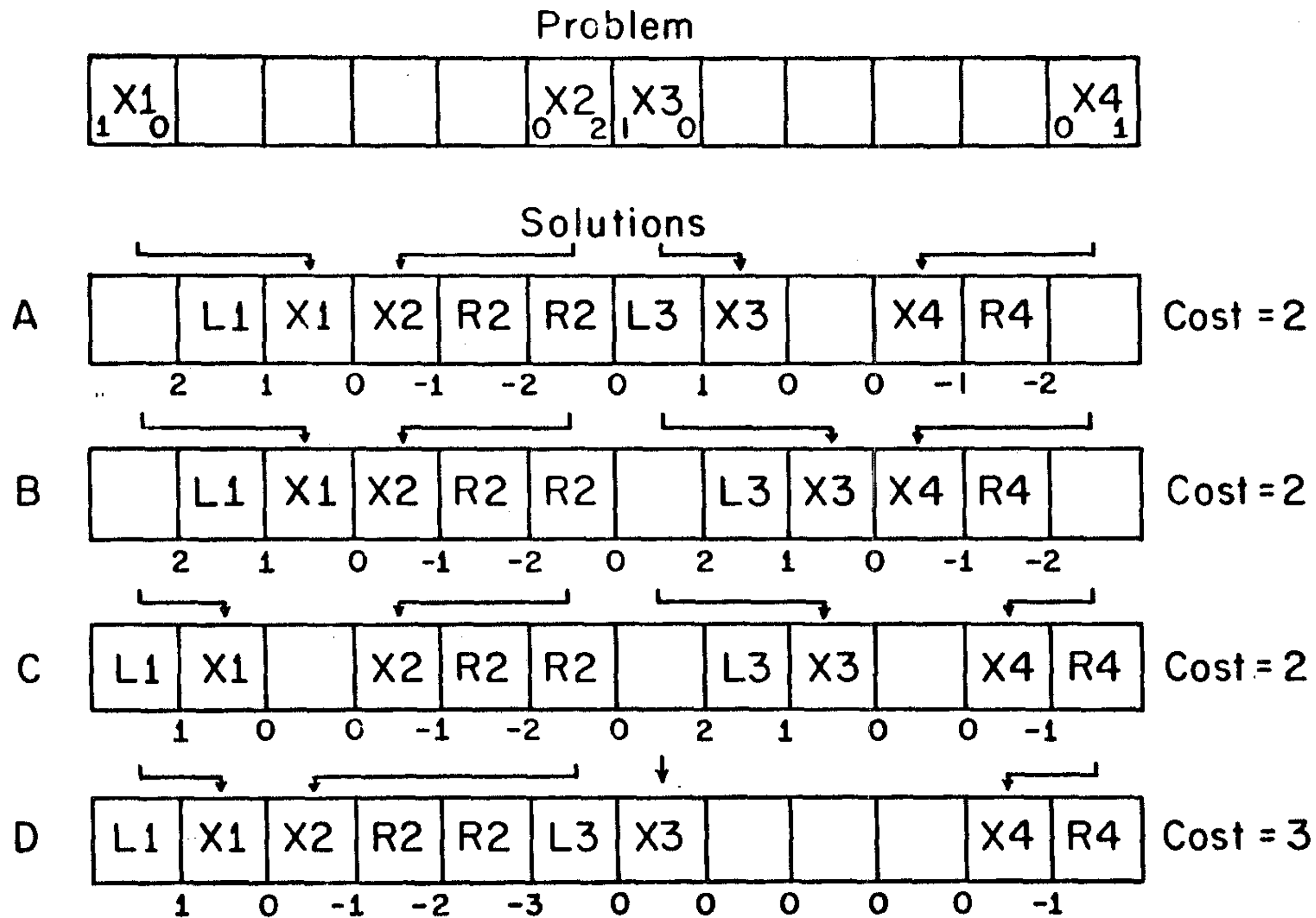


Figure 2: A storage management problem and four solutions, three of which are minimal cost. In the problem, given at the top, blank squares represent empty cells in the L array, Xi denotes the ith symbol of the program text, and the left and right subscripts of Xi denote the *number* of cells requested by Xi on its left and right respectively. In the solutions, Li and Ri denote cells reserved for fulfilling the requests of Xi on the left and right respectively. Arrows denote movement of text symbols. Integers below cell dividers are equal to the flow over the edge between the cells; see Section IV.

right. A solution to a storage management problem consists of a set of magnitude-direction pairs, one pair for each occupied cell, such that if the symbol of each occupied cell is moved according to its pair, then the left-to-right order of the symbols is unchanged and the resulting sequence of occupied and unoccupied cells will be such that all requests are satisfied. Moreover, no empty cell is used to satisfy more than one request.

Figure 2 illustrates a storage management problem and some possible solutions. The contents of a cell of the L array is moved by a succession of shifts to the right or left,

** Figure 2 goes here **

i.e., the movement is contained within the L array itself. Because each L cell can contain only one symbol of the program text, text movement must be coordinated so that data are not overwritten. We assume that buffers are used in shifting cell contents, so that, for example, the contents of a set of contiguous cells can be shifted simultaneously either to the right or left. However, the contents of one occupied cell cannot be shifted into an adjacent occupied cell unless that cell's contents is being shifted in the same direction simultaneously; such an action would cause overwriting. Shifting the contents of a cell to its neighbor is assumed to take one time unit.

Text movement will require minimal time, and no overwriting will occur, if the contents of all occupied cells begin to move at the same time and move continuously until they reach their destination. Thus each occupied cell in L must be told in which direction and how far its symbol is to move. In the following, for the sake of simplicity, we often speak of moving or shifting an L cell, when we mean, in fact, moving the contents of the cell.

The storage management process can be divided into two phases: preparation and text movement. Preparation consists of determining and communicating to each occupied cell the magnitude-direction pair that specifies how the symbol contained in the cell should move. Text movement is the actual movement of the cell contents. The time required for the second phase, text movement, is the maximum distance moved by the contents of any cell along the L array.

Definition 2: The cost of a solution S to a storage management problem P is the largest magnitude of the set of magnitude-direction pairs comprised by the solution S, i.e., the cost of S is the maximum distance moved by any symbol in the sequence of cells. A solution S to a storage management problem P is said to be minimal cost (or optimal) if the cost of S is no greater than that of any other solution to P.

In Fig. 2, solutions A, B and C are minimal cost; D is not. The following proposition is easily proved.

Proposition 1: If P and P' are storage management problems over the same linear array of cells such that the set of occupied cells and the set of requests of P' are subsets of those of P, then a minimal cost solution for P' will cost no more than a minimal cost solution for P.

The problem we address in this paper is the design of an algorithm to compute minimal time solutions to storage management problems. Moreover, we would like the preparation phase of this algorithm to be as small as possible. It is clear from the interconnection diagram of Figure 1 that the only paths for information flow which can involve the entire L array are along the L array itself and through the root node. Hence, a solution can be computed entirely within the L array, or by using information paths through the root cell. The time required by a preparation phase which occurs entirely within the L array is prohibitive. Thus, an asymptotically optimal preparation phase for the given interconnection pattern involves

a) passing information and performing computations in an upward wave through the tree, starting with the L cells and proceeding to the root, with the time and space required for the computations at each level, including the root, bounded by constants, and then,

b) passing information and performing computations in a downward wave from the root to the L cells so that each L cell receives its part of the solution information, with the time and space requirements for the computations at each level bounded by constants.

If a minimal time solution can be found in this way, then for an L array consisting of n cells, the preparation phase would require O(log n) time, and the text movement phase would require minimal time. In this paper we develop algorithms for execution by the L and T cells to compute a minimal time solution with one upward wave of information going from the L array to the root of the tree followed by a downward wave from the root to the cells of the L array. The time and space requirements for the computations performed by any cell are bounded by constants; thus the solution described here is an O(log n) (and therefore asymptotically optimal) algorithm for finding an optimal solution to any storage management problem.

The storage management algorithm described by Magó [3] is similar to the one we develop here in that the preparation phase requires a single wave up and down through the tree. The solution computed by Magó's algorithm, however, is not optimal.

IV. The Strategy of the Algorithm

In order to tell each occupied cell of the L array which direction and how far it should move, we consider the contents of each occupied cell and each cell request to be units which move along the L array. A flow of these units will be calculated by the T array for each edge of the L array. The magnitude of the flow over an edge represents the number of non-blank symbols plus cell requests that will be moved over the edge, where cell

requests are considered to originate in the cell of the requesting symbol. The direction of the flow specifies the direction in which symbols and requests are to be shifted over the edge. Figure 2 gives examples of flow values. With the algorithm described here, the flow calculated for an edge is communicated to the L cell immediately to the right of the edge; thus, each L cell (except the one on the extreme left of the L array) is told the magnitude and direction of the flow across the edge to its left. The flow along the edge adjacent to an occupied cell can be used to calculate the movement for the contents of that cell in the following way. Consider any occupied L cell C and suppose that the flow over the edge to its left is an integer f, where a negative value of f denotes a flow to the left and a positive value a flow to the right. (The cell on the extreme left end of the L array is told that there is a zero flow over the nonexistent edge on its left.) If C has requested p cells on its left, then the contents of C should move f+p, where a positive value indicates a move to the right and a negative value a move to the left.

Each cell C of the T array is responsible for computing the flow over a single edge of the L array, namely the edge which lies between its left descendant leaves and its right descendant leaves; we say each T cell is <u>associated</u> with this edge of the L array which, in the diagram of Figure 1, lies directly beneath it. During the upward wave, each T cell computes parameters that determine a tentative flow for its associated edge. During the downward wave, each T cell computes the actual flow over its

associated edge. This information is then carried to the proper L cell by the downward wave.

The upward wave begins with each L cell computing the parameters of a tentative partial solution to the storage management problem. Each L cell constructs this solution solely on the basis of whether it contains a symbol, and if so, the requests of the symbol. During the upward wave each T cell receives parameters of tentative partial solutions computed by its two sons; these parameters are used by the T cell to compute the parameters of a larger tentative partial solution by marrying the partial solutions of its sons. The upward wave ends when the root node has computed the parameters of its tentative solution.

To begin the downward wave, the root node incorporates the constraint that there can be no flow in or out of the endpoints of the L array. Each T cell other than the root receives from its parent node comparable constraints in the form of restrictions on the flow in and out of its leftmost and rightmost descendant leaves. After a T cell has received its constraints, it modifies its solution to satisfy the constraints, computes the flow over its associated edge, and communicates constraints to each of its sons. The computed flows are carried along with the downward wave until they reach the appropriate L cell, where they are used to determine the movement required of the symbol in each occupied cell as described earlier in this section.

The success of the algorithm depends on the fact that no cell is required to find an entire solution to the problem posed

in its leaves; instead, each cell finds only part of the solution, including the flow over its associated edge. Thus the solution for the entire machine is computed by the conglomeration of T and L cells, and the solution for the leaves of any T cell C is computed by C together with its descendants. (If C is any L or T cell, we mean by "the leaves of C" the set of L cells that are descendants of C.) With this made explicit, we now begin to speak simply of a cell computing a solution when we mean, in fact, that the cell is computing its share of a solution, where this share is represented by a set of parameter values.

V. Compact Minimal Cost Solutions

The set of parameters computed by each cell C during the upward wave characterizes a set of minimal cost tentative solutions of the storage management problem posed by the leaves of C. To find these parameters, each cell C assumes that immediately to the right and left of its leaves there is an arbitrary number of empty L cells that can be used to satisfy requests made by its leaves. If C is a T cell, the set of its leaves is the union of the leaves of its two sons; thus C can construct a tentative solution for its leaves by marrying the tentative solutions of its sons. Marriage of the two subsolutions may require resolving conflicting demands for some L cells. The cell C resolves conflicts by constructing a solution which involves moving some occupied leaves of its left son further to the left, those of its right son further to the right,

or both. Even if the two subsolutions do not have conflicting requirements, the tentative solution computed by C will not generally consist simply of the combined solutions of its sons. Instead, C will shift both occupied and requested cells in order to make its own tentative solutions as compact as possible (to minimize conflict with other partial solutions) without compromising their minimal cost. After computing the parameters of its own tentative solutions, each cell passes the parameters to its parent cell. To make these notions precise, we now give a number of definitions and state some preliminary results.

Definition 3: The specific set of cells which would be either occupied or requested by implementing a solution S to a storage management problem is called the configuration of S. The leftmost cell of a configuration is the left end of the configuration, similarly for the right end. The span of a configuration is zero if the configuration is empty, otherwise it is the distance from the left end of the configuration to the right end plus one. A configuration is said to have no internal blanks if it is nonempty and every cell between the left end and the right end of the configuration is either occupied or requested. We will often refer informally to the "right end of a solution," the "span of a solution" or "moving a solution" by which we mean respectively the right end of a configuration, the span of a configuration or moving the configuration of the solution referred to.

During the upward wave, each L and T cell C computes a set of parameters that characterizes the set of minimal cost solutions with the smallest spans for the storage management problem in the leaves of C.

Definition 4: Let P be a storage management problem whose requests sum to r, and let P' be the storage management problem obtained by adding r unoccupied cells to each end of the array of P, thus ensuring adequate space for text movement to either the right or left. A compact minimal cost (cmc) solution S to P is a set of magnitude-direction pairs such that S is a minimal cost solution to P' and the span of every other minimal cost solution to P' is at least as large as the span of S.

In Fig. 2, solutions A, B and C are all minimal cost, but only A and B are compact minimal cost. Note that a cmc solution to P is not a solution to P but rather a solution to P' where P' is obtained by adding blank cells to both ends of the array which contains P. During the upward wave, each cell C will find the parameters of all cmc solutions to the storage management problem in its leaves. These parameters include the cost, the location of the left and right ends, and the number of internal blanks. The following propositions characterize some important properties of cmc solutions.

Proposition 2: For any storage management problem P, all cmc solutions to P have the same cost, the same span and the same number of internal blanks.

Problem

	$_0X1_1$	$_0X2_0$	

Solutions

A	X1	R1	X2	
B		X1	R1	X2

Figure 3: A storage management problem for which there are two cmc solutions with different endpoints.

Proposition 3: For any storage management problem P, exactly one of the following two assertions holds:

a) The positions of the left ends of all cmc solutions to P are the same, as are the positions of the right ends of all cmc solutions.

b) There are exactly two cmc solutions, neither of which has internal blanks, and the left ends of the two solutions are adjacent, as are the right ends of the two solutions.

Definition 5: A cmc solution S to a storage management problem P is leftmost (rightmost) if the left end of S is at least as far left (right) as the left end of any other cmc solution S' to P.

If assertion a) of Proposition 3 holds for some problem P, then all cmc solutions to P are both leftmost and rightmost; if assertion b) of Proposition 3 holds, then there is a single leftmost cmc solution and a single rightmost cmc solution and they are distinct. Figure 3 illustrates a storage management problem for which assertion b) holds.

** Figure 3 goes here **

A cell constructs a cmc solution for its leaves by modifying and combining the cmc solutions of its sons. Conceptually this is done by moving the subsolution configurations to the left or right while making them as compact as possible. The next lemma

characterizes the cost and span that results when a cmc solution is moved.

<u>Proposition 4</u>: Let P be a storage management problem, and let S be a solution to P with b internal blanks and cost c. For any integer $k \geq 0$, if there are at least max(0,b-k) blanks available to the right of S, then there is a (not necessarily unique) solution S' to P such that the following hold:

a) the left end of S' is k cells to the right of the left end of S,

b) the cost of S' is c + k, and

c) the number of internal blanks in S' is b - min(2k,b).

We denote any such solution S' by R(S,k). The analogous assertion obtained by replacing <u>right</u> with <u>left</u> and <u>left</u> with <u>right</u> also holds, and the resulting solution is denoted by L(S,k). If S is a cmc solution, any solution of the form L(S,k) or R(S,k) will be called a <u>modified cmc solution</u>.

The solution R(S,k) is obtained by pushing the left end of S to the right by k cells, which eliminates up to k internal blanks. If S has more than k blanks, the right end of S is simultaneously pushed to the left until either all the internal blanks are eliminated or the right end has been moved k cells; thus R(S,k) has up to 2k fewer internal blanks than S.

<u>Proposition 5</u>: The class of modified cmc solutions is closed under the operators L and R. In particular, if S is any cmc solution and k and m are nonnegative integers, then

$L(L(S,k),m)=L(S,k+m)$ and

$L(R(S,k),m)=X(S,n)$,

where X is either L or R and $\min(k,m) \le n \le \max(k,m)$. Moreover, if the number of internal blanks in a solution S is no less than 2k, then $L(S,k)=R(S,k)$, and therefore, $L(R(S,k),m)=L(S,k+m)$. The corresponding assertions obtained by replacing all occurrences of L and R by R and L respectively also hold.

Proposition 6: If S is a cmc solution to a storage management problem P, then R(S,k) is compact and minimal cost in the sense that if S' is a solution to P and the left end of S' is k cells to the right of the left end of S, then S' is at least as costly as R(S,k) and the span of S' is at least as great as that of R(S,k). The assertion remains true when all occurrences of R, left and right are replaced by L, right and left respectively.

It follows from Proposition 6 that if S is a cmc solution to a subproblem P, and it is necessary to accommodate other subsolutions by, for example, moving the left end of S to the right by k cells, then the least expensive and most compact solution available is R(S,k). We will often impose two constraints, one at each end. This will only be done, however, when the solution has a sufficient number of internal blanks so that both constraints can be satisfied. Proposition 5 assures us that the successively imposed constraints will have the same effect as a single constraint and the result, again by Proposition 6, is assured to be as economical as possible.

VI. The Upward Wave: Computing the cmc Solutions

To be able to compute the parameters of its cmc solutions during the upward wave and to participate in computing the global solution during the downward wave, each cell C computes the following integer-valued parameters during the upward wave.

a) #CELLS: the number of L cells which are descendants of C.

b) COST: the cost of any cmc solution to the storage management problem in the leaves of C.

c) LEFT: specifies location of the left end of the leftmost cmc solutions relative to the leftmost L cell descendant of C. If the value of LEFT is negative, it is equal to the number of additional blank L cells to the left of the leaves of C which are used by the leftmost cmc solutions computed by C. If the value of LEFT is positive, then it is equal to the number of unoccupied and unrequested L cells that can be used to satisfy requests of cells to the left of the leaves of C without disturbing any cmc solution to the problem in the leaves of C.

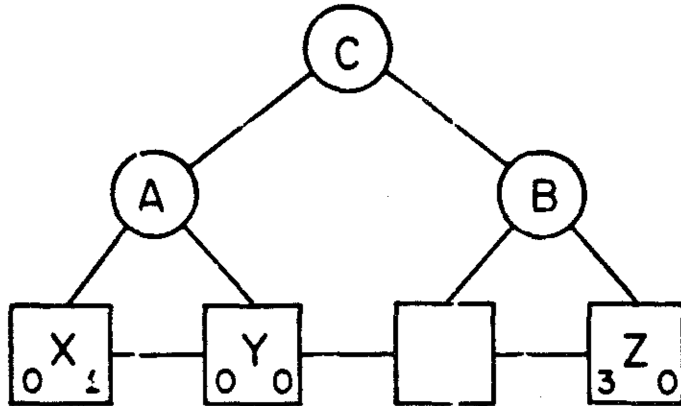
d) RIGHT: this parameter is analogous to LEFT; it specifies the right end of the rightmost cmc solutions.

e) BLANKS: the number of internal blanks (unoccupied and unrequested cells) between the left end and right end of any cmc solution to the storage management problem in the leaves of C. This is a measure of how much the solution can be "squeezed."

f) ARB: a single bit of information to indicate whether the

positions of the left and right ends of the cmc solutions found by C are unique. If the endpoints of all cmc solutions are the same, then ARB=0. If there are two different positions for cmc solutions, as in Fig. 3, then ARB=1. Note that the positions of the endpoints of all leftmost cmc solutions are LEFT and RIGHT + ARB; the positions of the endpoints of all rightmost cmc solutions are LEFT + ARB and RIGHT. The span of any cmc solution is equal to #CELLS - LEFT - RIGHT - ARB.

The six parameters listed above are computed by each L and T cell during the upward wave. Each L cell computes the values directly, based only on whether it is occupied, and if so, the number of cells requested on its right and left. Each T cell computes its parameters based on the parameters of its sons. Specifically, each T cell will receive the parameter values of its left son as the input parameters #CELLS1, COST1, LEFT1, RIGHT1, BLANKS1 and ARB1. The corresponding parameter values of the right son will be received as the input parameters #CELLS2, COST2, LEFT2, RIGHT2, BLANKS2 and ARB2. After computing its own values of these parameters, each T cell other than the root will pass the values to its father cell. (We take parameter passing to be by value between cells; thus although LEFT1 of each T cell is initialized to the value of LEFT in its left son, assigning a new value to LEFT1 does not change the value of LEFT in the son node.) In computing the values of its parameters, a T cell C will generally change the values of its input parameters to



	#CELLS	COST	LEFT	RIGHT	BLANKS	ARB
A	2	1	-1	-1	0	1
B	2	0	-2	0	0	0
C	4	2	-2	-2	0	1

Figure 4: Parameter values computed by three T cells during the upward wave.

I At least one subtree of C has no occupied L cells.

- **A** Neither subtree of C has any occupied L cells. (C constructs the empty solution.)
- **B** The right subtree of C has occupied L cells but the left subtree does not. (C adopts the solution of its right son.)
- **C** The left subtree of C has occupied L cells but the right subtree does not. (C adopts the solution of its left son.)

II Both subtrees of C have occupied L cells.

- **A** The cmc solutions of the sons of C do not overlap.
  - **1** The cost of the cmc solution of the left son is no greater than that of the right son. (The cmc solution of the left son is moved right zero or more cells and compacted.)
  - **2** The cost of the cmc solution of the right son is less than that of the left son. (The cmc solution of the right son is moved left and compacted.)
- **B** The cmc solutions of the sons of C overlap
  - **1** The solutions overlap by a single cell and the position of neither solution is unique, i.e., ARB = 1 for both solutions. (The overlap is actually nonexistent; the two solutions combine into a single solution with ARB = 1.)
  - **2** The solutions of the sons of C overlap by more than one cell or the endpoints of the solution of at least one son are unique.
    - **a** The left solution costs at least as much as the right and the difference in cost is at least as great as the overlap remaining after use of the ARB spaces. (The solution of the left son is moved to the left zero or more cells and compacted.)
    - **b** The right solution costs less than the left solution and the difference in cost is at least as great as the overlap remaining after use of the ARB spaces. (The solution of the right son is moved to the right and compacted.)
    - **c** The overlap after use of the ARB spaces is greater than the difference in cost between the two solutions. (Both solutions are be compacted and moved outward until overlap is eliminated.)

Figure 5: Case structure of the algorithm UP_TCELLS executed by each cell C of the T network. The way in which C constructs its cmc solution is described parenthetically for each of the cases.

describe the way the solutions for its sons are to be changed to obtain the cmc solution to the storage management problem of the cell C. Fig. 4 shows the parameter values computed by a small tree during the upward wave.

** Figure 4 goes here **

The algorithms for computing the cmc solutions by the L cells and T cells during the upward wave are UP_LCELLS and UP_TCELLS respectively. These algorithms are given in the Appendix. UP_LCELLS is quite straightforward, but UP_TCELLS is not. A very high level description that ignores some important details of UP_TCELLS is given in Figure 5.

** Figure 5 goes here **

VII. The Downward Wave: Computing the Flows

During the upward wave, each T cell computes parameters for all cmc solutions to the storage management problem in its leaves. Each cell resolves conflicts between the cmc solutions of its sons, but assumes an unlimited supply of empty cells beyond its leftmost and rightmost leaves. During the downward wave, the finite extent of the L array is taken into account. This is done by using input parameters LEFTC and RIGHTC of each T

cell to specify the left and right ends within which the T cell must find a solution for its storage management problem.

Passing a negative value of LEFTC to a cell C means that C may use a solution which extends beyond its leftmost leaf by -LEFTC cells; a positive value of LEFTC means that satisfying the requests to the left of the leaves of C will utilize LEFTC L cells extending to the right from the leftmost L descendant of C. Therefore, if LEFTC is positive, some of the leftmost L descendants of C are not available for solving the storage management problem in the leaves of C. RIGHTC specifies the right constraint in the same way. The parameters of the cmc solution computed in the upward wave by each T cell include LEFT, RIGHT, LEFT1, BLANKS1, RIGHT1, LEFT2, BLANKS2, RIGHT2 and ARB; these values are used during the downward wave. If the constraints LEFTC and RIGHTC lie outside the bounds LEFT and RIGHT of its cmc solution, then the cell can implement one of its cmc solutions. Otherwise, it must compute a new solution, by "shifting and squeezing" the cmc solution computed in the upward wave. While finding the parameters of a solution within the limits specified by LEFTC and RIGHTC, each T cell will compute new values of LEFTC1 and RIGHTC1 and communicate them as constraints to its left son. New values of LEFTC1 and RIGHTC1 will specify constraints for the right son.

Each T cell also calculates the flow of occupied and requested cells across its associated edge and communicates this value to its right son via the output parameter FLOW. Moreover,

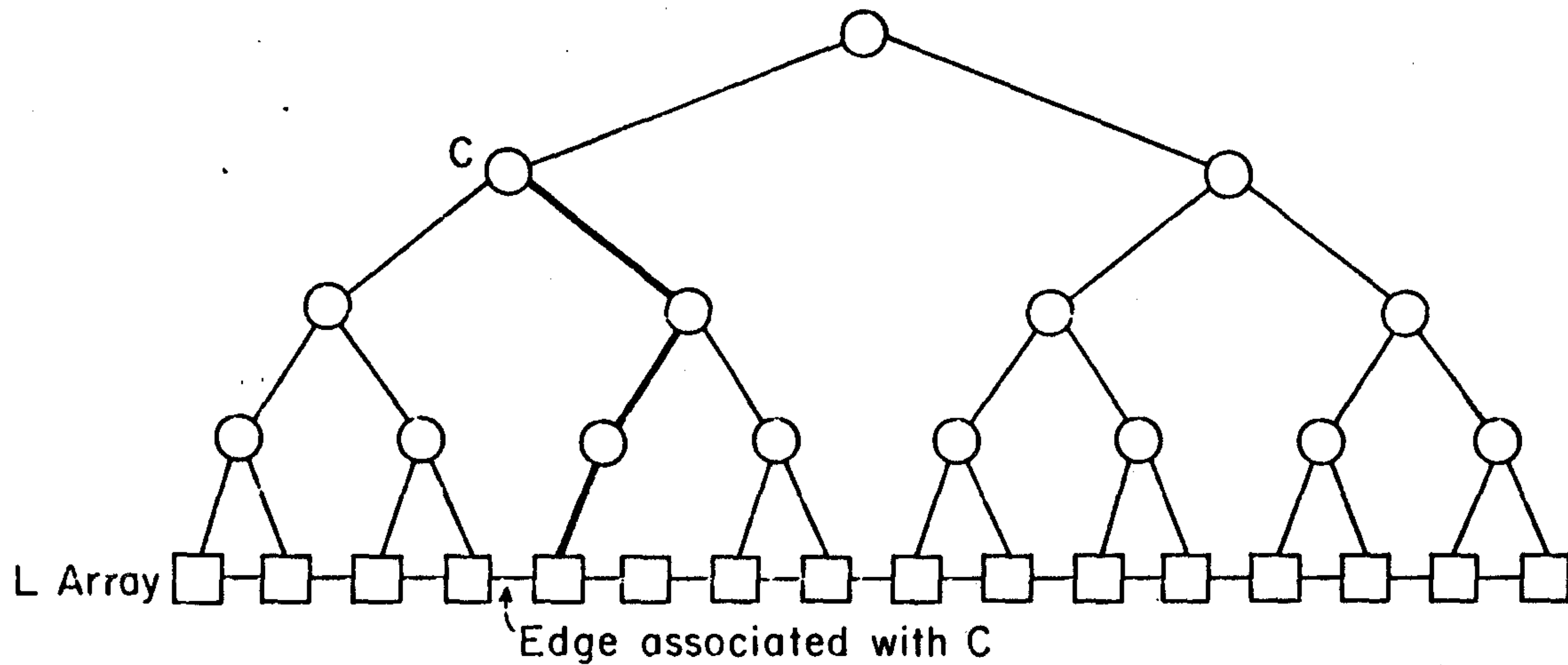


Figure 6: Cell C computes the flow along its associated edge. The value of this flow travels with the downward wave along the bold edges from cell C to the L cell to the right of the edge associated with C.

every cell except the root will receive a flow value DFLOW from its father; this value will be transmitted unchanged to its left son. Figure 6 shows the path of a flow value from the T cell C which computed it to the L cell to the right of the edge associated with C.

** Figure 6 goes here **

In summary, each T cell except the root will receive parameters LEFTC, RIGHTC and DFLOW from its father; the root cell will assign the value of 0 to these variables. Every T cell (including the root) will transmit parameters LEFTC1, RIGHTC1 and DFLOW to its left son and LEFTC2, RIGHTC2 and FLOW to its right son.

At the end of the downward wave, each L cell receives the parameter DFLOW from its father; this parameter specifies the flow over the edge to the immediate left of the cell (where the edge is imaginary in the case of the leftmost L cell). From this value, each occupied L cell determines in the manner described earlier the distance and direction that its contents is to move.

VIII. Proofs of Correctness and Optimality

In this section we give informal proofs that the principal algorithms presented in the Appendix are correct, i.e., if all requests of a storage management problem can be satisfied, then

these algorithms find a minimal cost solution as characterized in Definition 2. Proofs of the lemmas are omitted to conserve space. Note that parameters are passed between procedures by reference; thus the same parameter name may be used as both an input and output parameter.

The algorithm UP_TCELLS uses a pair of procedures MOVEL and MOVER to move cmc solutions to the left or right both to resolve conflicts and to make solutions more compact. When MOVEL is given a positive integer k and the parameters of a cmc solution S, it computes the parameters of the modified cmc solution L(S,k) as defined in Proposition 4. Similarly, MOVER finds the parameters of R(S,k). (The applicability of the procedures MOVEL and MOVER is not restricted to cmc solutions. During the downward wave, these procedures are applied to modified cmc solutions to produce the parameters of other modified cmc solutions.) The first lemma asserts the correctness of MOVEL and MOVER.

Lemma 1: If P is a storage management problem and S is a modified cmc solution to P with parameters LEFT, RIGHT and BLANKS, then the procedure MOVEL(k, RIGHT, BLANKS, LEFT) finds the parameters of a modified cmc solution whose right end is k cells to the left of the right end of S. Similarly, MOVER(k, LEFT, BLANKS, RIGHT) finds the parameters of a modified solution whose left end is k cells to the right of the left end of S.

Theorem 1: During the upward wave, each L cell (using the

procedure UP_LCELLS) and T cell (using the procedure UP_TCELLS) computes the correct values of LEFT, BLANKS, RIGHT, ARB, and COST for all cmc solutions of its storage management problem. Moreover, every T cell computes parameters of modified cmc solutions for its sons that are consistent with its own cmc solution. These parameters are LEFT1, BLANKS1, RIGHT1, LEFT2, BLANKS2, and RIGHT2. The parameters ARB1 and ARB2 of the modified cmc solutions are equal to ARB of the T cell itself.

Proof: The proof is by induction on the height h of the cell, where the height of each L cell is zero and the height of each T cell is one greater than the height of its sons. For the basis step, the cell is an L cell and h=0. The parameters of the cmc solution are computed by the L cell algorithm UP_LCELL. A cmc solution for a single cell never requires any movement of an occupied cell, hence its cost is zero and therefore the solution is minimal cost. If the L cell is occupied with left request p and right request q, then any solution must occupy one cell and request p+q cells. The solution found by the L cell algorithm has a span of p+q+1 cells; it follows that the solution is cmc. Since the solution has no internal blanks and is unique, BLANKS and ARB are set to zero. This completes the basis step.

For the induction step, we assume that C is a T cell, which is provided with the parameter values of all cmc solutions for its two sons; C is to find the cmc solution for its own storage management problem by finding compatible modified cmc solutions for its two sons. Note that the set

between S1 and S2, is greater than BLANKS1 + AVAIL. If these conditions are met, S1 is moved to the right by an amount large enough to eliminate all its internal blanks and all the blanks between S1 and the leftmost solution of S2; then the value of RIGHT1 is adjusted to represent the fact that its position (as a subsolution of S) will be one cell more to the right if rightmost solutions are used. In the second subcase of Case II.A.1, the conditions for ARB to be 1 do not hold. S1 is moved to the right until either all internal blanks of S1 and all blanks between S1 and S2 are eliminated, or the cost of the repositioned S1 is equal to the cost of S2. The leftmost solution of S2 is chosen by adding ARB2 to RIGHT2; this guarantees that S will be as compact as possible. The resulting component solutions of S consist of the leftmost solution of S2 plus the result of applying MOVER to the rightmost solution of S1. To summarize Case II.A.1, the value of LEFT2 is unchanged; the value of RIGHT2 is either unchanged or changed by the addition of ARB2. The values of LEFT1 and RIGHT1 are set by MOVER, except that RIGHT1 is further modified by subtracting 1 if the position of S (and hence the repositioned S1) is arbitrary. The cost of S is equal to that of S2 and therefore minimal if S2 is cmc. Furthermore, since S1 was moved as far as possible to the right and S2 was moved as far as possible to the left without increasing the cost of S, S is compact and therefore cmc. This completes the treatment of Case II.A.1. Case II.A.2 is analogous, except

that S1 is more expensive than S2; thus the roles of S1 and S2 are reversed and the costs of S1 and S2 are not equal.

Case II.B treats those situations in which the solutions S1 and S2 overlap. The algorithm first sets the parameter NEED equal to the number of cells by which the solutions overlap; Case II.B is then broken into two subcases.

Case II.B.1 applies only when NEED=1 and ARB1=ARB2=1; this represents two arbitrary solutions positioned so that the conflict is resolved if either the rightmost or leftmost solutions of both S1 and S2 are used. Thus the conflict is only a manifestation of the representation of the arbitrary solutions S1 and S2. The two solutions S1 and S2 fit together without adjustment, giving a solution S with ARB=1. LEFT1, RIGHT1, LEFT2 and RIGHT2 are unchanged. Since S is essentially the union of S1 and S2, the cost of S is the larger of the costs of S1 and S2; hence S is minimal cost. Since S has no internal blanks, it is clearly compact. This completes the discussion of Case II.B.1. (If the test for Case II.B.1 fails, it is still possible that there will be an overlap of only one cell between the solutions S1 and S2, and that adequate space will be provided by the ARB spaces. Such cases are handled in Case II.B.2.)

The test for Case II.B.2 is the negation of that for II.B.1. This case first tries to reduce the overlap of S1 and S2 by choosing the leftmost solution of S1 and the

rightmost solution of S2; this is done simply by adding ARB1 to RIGHT1 and ARB2 to LEFT2. The value of NEED is then recomputed; it will have decreased from the original value by 0, 1 or 2, and its new value can be any nonnegative integer. In the following discussion of Case II.B.2, S1 and S2 denote the results of this modification.

Cases II.B.2.a and II.B.2.b apply when the solutions S1 and S2 differ so much in cost that the cheaper one can be moved far enough to eliminate the overlap without increasing its cost beyond that of the more costly solution. Case II.B.2.a treats the case in which the cost of S1 is no greater than the cost of S2 and the difference in the costs of S1 and S2 is at least as great as the value of NEED. The algorithm first uses MOVEL to move S1 to the left sufficiently far to eliminate the overlap between the component solutions. The resulting solution S1' for the left son, however, may still contain internal blanks and have a cost less than that of S2; in that case, the solution can be made more compact by applying MOVER to push S1' to the right far enough either to eliminate all the internal blanks or to increase further the cost of the left subsolution until it is equal to that of S2; we call the result S1". Thus S1 is first moved far enough left to eliminate all the overlap, giving S1'; then S1' is made as compact as possible giving S1". The resulting composite solution S, may, in fact, be one for which ARB should be set to 1. This will be the case if ARB2 is 1, S1" has no

internal blanks, and if S1" could be moved left one more cell without its cost surpassing that of S2. If all these conditions are met, then the parameters LEFT1 and LEFT2 are decremented by 1 to give the positions of the leftmost component subsolutions, and ARB is set to 1. Otherwise, the positions of the left and right ends of all solutions for C are the same and ARB is set to 0. Note that in Case II.B.2.a, S consists essentially of S2 together with the result of repositioning S1. Since the cost of the left component solution is never allowed to exceed that of S2, the cost of S is equal to that of S2, hence S is minimal cost if S2 is. Moreover, within the cost constraint, the left subsolution is made as compact as possible while eliminating the overlap; hence S is also compact and therefore cmc.

Case II.B.2.b is analogous to II.B.2.a with the roles of S1 and S2 reversed except that the cost of S1 is properly greater than that of S2.

The last subcase, Case II.B.2.c, applies when the solutions S1 and S2 overlap and the difference in cost is sufficiently small that both solutions must be moved outward. At this point, ARB1 and ARB2 have already been added to RIGHT1 and LEFT2 respectively, giving leftmost and rightmost solutions for S1 and S2. The first action in this case is to move the less costly of S1 and S2 away from the overlap until the two subsolutions have the same cost. NEED

is then recomputed.

If NEED is even, then it suffices to push each of the now equally costly subsolutions away from the overlap by a distance of NEED/2 to obtain the cmc solution for S; in this case, ARB is set equal to 0.

Now suppose NEED is odd and the component solutions S1 and S2 are equally costly. Both solutions must be moved away from the overlap by at least $\lfloor NEED/2 \rfloor$, and at least one of the solutions must be moved by $\lceil NEED/2 \rceil$. Since moving solutions away from the overlap by k cells makes the cost of S at least k greater than the cost of the subsolution being moved, it follows that neither subsolution should be moved by more than $\lceil NEED/2 \rceil$, and the cost of S will be the cost of S1 (which is equal to the cost of S2) plus $\lceil NEED/2 \rceil$. The question, therefore, is how to make the span of S as small as possible without increasing the cost by more than $\lceil NEED/2 \rceil$.

There are essentially three distinct actions that can be performed on each subsolution. Since they are symmetric we consider only the left subsolution S1. Suppose the number of internal blanks in S1 is greater than NEED, i.e., BLANKS1 > NEED. Since NEED is odd, applying MOVEL to S1 for $\lceil NEED/2 \rceil$ cells causes the left end of the new solution to be _further_ _right_ than when MOVEL is applied to S1 with only $\lfloor NEED/2 \rfloor$ cells. Since the left end of the resulting component solution S1' directly affects the span, and since

we know that applying MOVEL to S1 for $\lceil NEED/2 \rceil$ cells will not increase the cost beyond a minimal value, it follows that if BLANKS1 > NEED then we should apply MOVEL to S1 for $\lceil NEED/2 \rceil$ cells. Moreover, ARB=0 since moving the left end of S1 a smaller distance to the right would leave internal blanks in the left subsolution and moving it further to the right would increase the cost over the minimal value. Note that if BLANKS1 > NEED, then the action taken on S1 is not affected by the action taken on S2, and the resulting solution S has ARB=0.

Now suppose NEED is odd and BLANKS1 ≤ NEED. We first consider the various ways that S1 can be moved to the left by either $\lceil NEED/2 \rceil$ or $\lfloor NEED/2 \rfloor$ cells and all internal blanks eliminated from the left subsolution without increasing its cost by more than $\lceil NEED/2 \rceil$. Most simple is to apply MOVEL to S1 for $\lceil NEED/2 \rceil$ cells: the left subsolution is moved to the left $\lceil NEED/2 \rceil$ cells, all internal blanks are eliminated, and the cost of S1 is increased by $\lceil NEED/2 \rceil$. Similarly, if BLANKS1 < NEED and MOVEL is applied to S1 for $\lfloor NEED/2 \rfloor$ cells, then S1 is moved to the left $\lfloor NEED/2 \rfloor$ cells, all internal blanks are eliminated, and the cost is increased by $\lfloor NEED/2 \rfloor$. Finally, if BLANKS1=NEED, then applying MOVEL to S1 for $\lfloor NEED/2 \rfloor$ cells moves S1 to the left for $\lfloor NEED/2 \rfloor$ cells, leaves a single internal blank, and increases the cost by $\lfloor NEED/2 \rfloor$. Applying MOVER to this resulting solution for a single cell does not change the right end of the solution and eliminates the single remaining blank at a

determine at this point if a solution exists to the global storage management problem. If a solution does not exist, an error is signalled; otherwise the root computes the flow over its associated edge and the constraints for its sons. For T cells other than the root, the values of LEFTC and RIGHTC are received from the parent node. In proving that a node C at a distance k from the root, k > 0, computes the flow and constraints for its sons, the induction hypothesis asserts that the T cell under consideration, as well as all its brother nodes, received correct values of LEFTC and RIGHTC from its parent node, and that there exists a modified cmc solution that satisfies these constraints, i.e., for each T cell with a trivial solution,

LEFTC + RIGHTC < #CELLS

and for a cell with a nontrivial solution,

LEFTC - LEFT + RIGHTC - RIGHT < BLANKS.

We now argue that any T cell, given correct values of LEFTC and RIGHTC, will compute a correct value for the flow over its associated edge. This computation is performed in Step 2 of DOWN. While keeping track of all possible sets of solution endpoints was important during the upward flow, during the downward flow each T cell need only work with a single arbitrary cmc solution. For this reason, the first action in Step 2 is for C to choose (arbitrarily) the set of parameters of the rightmost cmc solutions for subsequent use. Step 2 then treats four mutually exclusive cases. In

Case I, C has no L cell descendants that are occupied and therefore C computed the trivial cmc solution during the upward wave. Any flow over the associated edge of C will result from other subsolutions extending into the L cells of C. The movement of symbols or requested cells into L cells of C from the left will be signalled by a positive value of LEFTC. If this value is greater than #CELLS/2, then there is a nonzero flow to the right over the associated edge of C. Since there are #CELLS/2 cells of L that are left descendants of C, the value of the flow is LEFTC - #CELLS/2. If the value of LEFTC is less than or equal to #CELLS/2, there is no flow to the right over the associated edge of C. The argument is similar for RIGHTC. In each case, these flow values are of the smallest possible magnitude consistent with the constraints LEFTC and RIGHTC. This completes the treatment of Case I of Step 2.

Case II applies when the left subtree of C contains no occupied L cells but the right subtree does. In this case, the cmc solution of C is essentially the cmc solution of its right son, modified by the availability of additional empty L cells on the left. SQUEEZE1, called with the parameters of the cmc solution of C and the constraints LEFTC and RIGHTC, is used to modify the solution of C. By Lemma 2, SQUEEZE1 finds the cheapest solution that satisfies the constraints LEFTC and RIGHTC. The flow over the associated edge is of the smallest possible magnitude consistent with the modified

cmc solution found by SQUEEZE1. If LEFTC > #CELLS/2, then (as in case I) there is a flow of magnitude LEFTC - #CELLS/2 to the right over the associated edge of C. On the other hand, if the modified cmc solution of the right son of C extends into the L cells of the left son, then LEFT2 < 0 and there is a flow to the left whose value is LEFT2. Case III of Step 2 is analogous to Case II.

Case IV applies when both the left and right subtrees of C have occupied L cells. The procedure SQUEEZE2 is used to modify the component subsolutions of C if this is necessary. SQUEEZE2 is passed the parameters of both subsolutions and the constraints LEFTC and RIGHTC. If the cmc solution computed by C does not violate the constraints, the subsolution parameters are unchanged; otherwise one or both subsolutions are modified to satisfy the constraints. In either case, the resulting subsolutions are guaranteed by Lemma 3 to be the cheapest modified cmc solutions consistent with the given constraints, and these subsolutions are used to determine the flow over the associated edge of C. To be precise, if after execution of SQUEEZE2 the value of RIGHT1 (LEFT2) is negative, then the subsolution of the left (right) son extends beyond its L cells and causes a flow over the associated edge of C into the L cells of the right (left) son. The value of the flow is equal to -RIGHT1 (LEFT2).

Finally, we show that in Step 3 of DOWN, each T cell computes correct values of LEFTC and RIGHTC for each of its sons. The values for the left son are named LEFTC1 and RIGHTC1; for the right son they are LEFTC2 and RIGHTC2. These values must be consistent in that subsolutions are not allowed to overlap. Moreover, the constraints given to each T cell C must admit a solution to the storage management problem of C that is part of a minimal cost global solution. By the induction hypothesis, we assume LEFTC and RIGHTC have the desired properties. The outer constraints transmitted to its sons by a cell C, LEFTC1 and RIGHTC2, are assigned the values of the received or computed constraints LEFTC and RIGHTC respectively; this assures that the solutions for the sons of C will be contained in the space alloted to the solution for C and thus will not overlap with solutions to their right or left. It also assures that the inner constraints, RIGHTC1 and LEFTC2, can be assigned to satisfy the feasibility and minimal cost requirements. The values of the inner constraints RIGHTC1 and LEFTC2 are determined using the flow across the edge associated with C. In every case, this flow is of the smallest possible magnitude consistent with the component modified cmc solutions, and the magnitudes of RIGHTC1 and LEFTC2 are equal to the magnitude of the flow. The values of RIGHTC1 and LEFTC2 are chosen to allow room for movement of occupied cells and cell requests across the associated edge of C when such movement is required, and otherwise are set to 0; this guarantees the

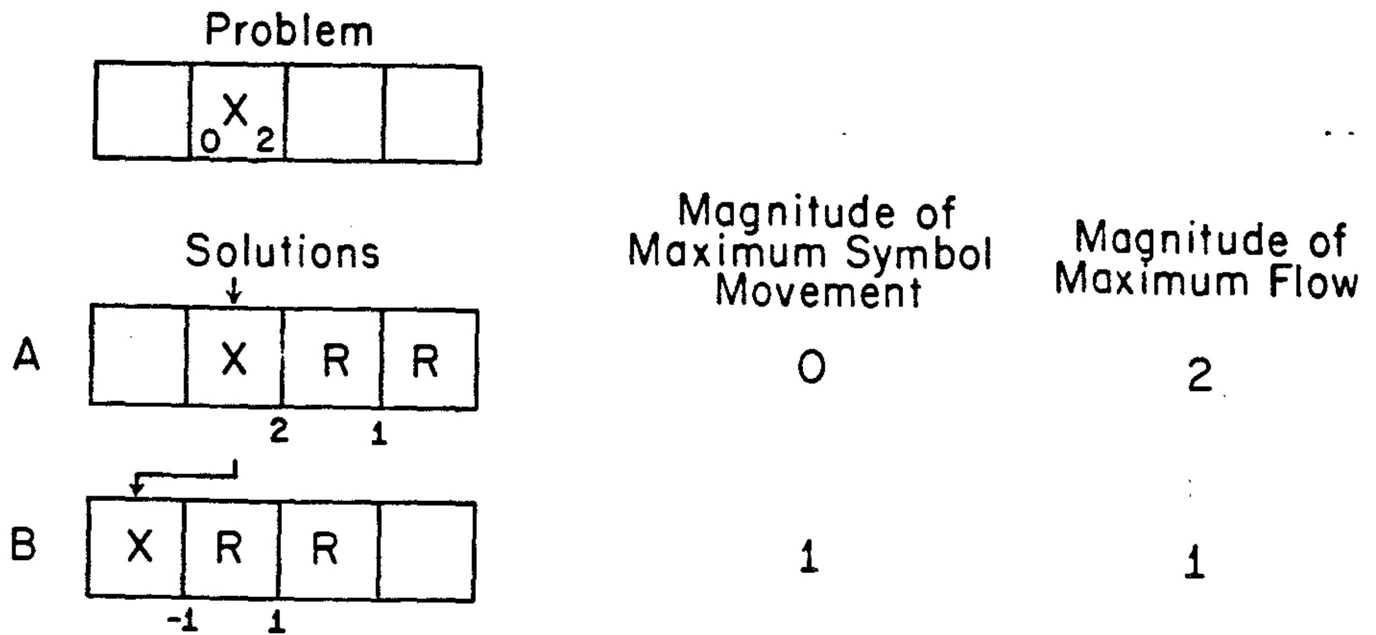


Figure 7: A storage management problem for which minimizing maximum symbol movement and minimizing maximum flow result in different solutions. In the solutions, the label R denotes cells reserved to satisfy the requests of X. Solution A minimizes maximum symbol movement; solution B minimizes maximum flow.

two measures of cost result in different optimal solutions.

** Figure 7 goes here **

The algorithms given for the original problem handle this variant without change except for UP_LCELLS which computes the cmc solution for a single cell of the L array. The algorithm for the L cells for the case in which the flow is to be minimized is given as the last algorithm of the Appendix as UP_LCELLS#2. This algorithm is similar to UP_LCELLS except that each L cell "centers" its cmc solution around itself, and computes a cost equal to the maximum distance that the cmc solution extends beyond the cell itself. Thus only the definition of cost and the algorithm executed by the L cells need be changed to minimize maximum flow; the remaining definitions, propositions, lemmas, theorems and procedures go through unaltered.

X. Conclusion

Efficient resource management is important in any computing system. In a machine of the type described by Magó [3], storage management is performed repeatedly during program execution and therefore has a sizeable impact on execution speed. We have described algorithms to be executed by the cells of the network to produce an optimal solution in minimal time. The algorithms make it possible

for the conglomeration of cells to solve a global problem even though each cell has access to only a small part of the problem specification.

As Magó has observed in [3], the machine size can easily be changed; for example, two machines, each with k L cells, can be used to construct a machine with 2k L cells by adding a root node (whose sons are the root nodes of the original machines) and connecting a single pair of newly adjacent L cells. This robustness of the design is supported by the algorithms we have described; with the exception of the root cell, the algorithm to be executed is determined only by the cell type and is unaffected by the position of the cell in the array or the total number of cells.

The algorithm is also interesting in its own right. The strategy of the algorithm bears some resemblence both to dynamic programming and recursive divide and conquer techniques, but differs in that no agent ever possesses a global solution to the problem. The algorithm exploits the network topology in a way similar to the techniques used by Bentley and Kung [4] ; in fact, the results we describe can be viewed as specifying a network which produces a solution to the flow problem described in Stanat and Magó [5] in O(log n) time and using a network with 2n - 1 cells.

Bibliography

1. J. Backus: Programming Language Semantics and Closed Applicative Languages. Proceedings of the First ACM Symposium on Principles of Programming Languages, pp. 77-86, 1973.

2. J. Backus: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Communications of the ACM, Vol. 21, No. 8, pp. 613-641, 1978.

3. G. A. Magó: A Network of Microprocessors to Execute Reduction Languages. (In two parts.) International Journal of Computer and Information Sciences, Vol. 8, No. 5, pp. 349-385, October 1979, and Vol. 8, No. 6, pp. 435-471, December 1979.

4. J. L. Bentley and H. T. Kung: A Tree Machine for Searching Problems. Proceedings of the 1979 International Conference on Parallel Processing, Oscar N. Garcia, editor, pp. 257 - 266.

5. D. F. Stanat and G. A. Magó: Minimizing Maximum Flows in Linear Graphs. Networks, Vol. 9, pp. 333-361, 1979.

Appendix

The following algorithms are executed by the L and T cells during the upward and downward waves. Parameter passing between procedures is by reference; the same parameters can serve for both input and output. See Sections VI and VII for documentation.

```
procedure UP_LCELLS
/¢ Executed by each L cell at the beginning of the upward
/¢ wave.  This procedure has no input other than the cond-
/¢ ition of the L cell.

#CELLS ← 1
if the cell is unoccupied then
   LEFT ← 1
   RIGHT ← 1
else    /¢ The cell is occupied.
   let p and q be the number of cells requested on the left
         and right respectively by the symbol in the L cell.
   LEFT ← -p
   RIGHT ← -q
   end if
BLANKS, ARB, COST ← 0

send LEFT, RIGHT, BLANKS, ARB, COST and #CELLS to parent node
end UP_LCELLS
```

The algorithm UP_TCELLS must often find L(S,k) or R(S,k) for a cmc solution S of one of its sons. When passed the appropriate subset of the parameters of a cmc or modified cmc solution S, the procedures MOVEL and MOVER find the corresponding parameters of the modified cmc solutions L(S,k) and R(S,k) respectively. The procedures MOVEL and MOVER are used in an analogous way during the downward wave by the procedures SQUEEZE1 and SQUEEZE2.

```
procedure MOVER (k, LEFT, BLANKS, RIGHT)
/¢ The left end of a modified cmc solution is to be moved
/¢ right by k cells. The parameters LEFT, BLANKS and RIGHT
/¢ initially contain values of the original solutions.
/¢ They are to be assigned values for new solutions which
/¢ are as compact as possible without costing more than
/¢ necessary for accomodation of the shift to the right.
/¢ Called by UP_TCELLS, SQUEEZE1 and SQUEEZE2.

LEFT ← LEFT + k
OLDBLANKS ← BLANKS
BLANKS ← max {0, OLDBLANKS - 2*k}
/¢ The value of LEFT + RIGHT + BLANKS must remain unchanged.
RIGHT ← RIGHT + OLDBLANKS - BLANKS - k
end MOVER
```

```
/¢ Case I.B: The right subtree has occupied L cells
/¢ but the left subtree does not.
case I.B: LEFT1 = #CELLS1 and RIGHT2 ≠ #CELLS2
   LEFT ← LEFT2 + #CELLS1
   RIGHT ← RIGHT2
   BLANKS ← BLANKS2
   ARB ← ARB2
   COST ← COST2
   end case I.B

/¢ Case I.C: The left subtree has occupied L cells
/¢ but the right subtree does not.
case I.C: LEFT1 ≠ #CELLS1 and RIGHT2 = #CELLS2
   LEFT ← LEFT1
   RIGHT ← RIGHT1 + #CELLS2
   BLANKS ← BLANKS1
   ARB ← ARB1
   COST ← COST1
   end case I.C
end case I
```

```
/¢ Case II: Both subtrees have occupied L cells.
case II: LEFT1 ≠ #CELLS1 and RIGHT2 ≠ #CELLS2
   /¢ Find difference between costs of solutions.
   DCOST ← |COST1 - COST2|

   /¢ Case II.A: The cmc solutions of the sons do not overlap.
   case II.A: RIGHT1 + LEFT2 ≥ 0
      /¢ Find number of empty cells between solutions.
      AVAIL ← RIGHT1 + LEFT2

      /¢ Case II.A.1: The cost of the cmc solution of the
      /¢ left son is no greater than that of the right son.
      case II.A.1: COST1 ≤ COST2
         /¢ Use rightmost cmc solution of left son.
         LEFT1 ← LEFT1 + ARB1
         /¢ First subcase of Case II.A.1: New solution will
         /¢ have ARB = 1.  This will occur only if expensive
         /¢ solution already has an ARB space and all blanks
         /¢ in cheap solution and between solutions can be
         /¢ squeezed out.
         if ARB2 = 1 and DCOST > BLANKS1 + AVAIL then
            MOVER (BLANKS1 + AVAIL, LEFT1, BLANKS1, RIGHT1)
            /¢ Modify left solution with ARB space.
            RIGHT1 ← RIGHT1 - 1
            BLANKS ← 0
            ARB ← 1
```

```
/¢ Second subcase of Case II.A.1: New solution
/¢ will have ARB = 0.
else       /¢ ARB2 = 0 or DCOST ≤ BLANKS1 + AVAIL
   MOVER (min {DCOST, BLANKS1 + AVAIL},
                               LEFT1, BLANKS1, RIGHT1)
   RIGHT2 ← RIGHT2 + ARB2
   BLANKS ← BLANKS1 + BLANKS2 + RIGHT1 + LEFT2
   ARB ← 0
   end if
/¢ Set remaining parameters for Case II.A.1
LEFT ← LEFT1
RIGHT ← RIGHT2
COST ← COST2
end case II.A.1
```

```
/¢ Case II.A.2: The cost of the cmc solution of the
/¢ left son is greater than that of the right son.
case II.A.2: COST1 > COST2
   RIGHT2 ← RIGHT2 + ARB2
   /¢ First subcase of Case II.A.2: New solution will
   /¢ have ARB = 1.  This will occur only if expensive
   /¢ solution already has an ARB space and all blanks
   /¢ in cheap solution and between solutions can be
   /¢ squeezed out.
   if ARB1 = 1 and DCOST > BLANKS2 + AVAIL then
      MOVEL (BLANKS2 + AVAIL, RIGHT2, BLANKS2, LEFT2)
      /¢ Modify right solution with ARB space.
      LEFT2 ← LEFT2 - 1
      BLANKS ← 0
      ARB ← 1
   /¢ Second subcase of Case II.A.2: New solution
   /¢ will have ARB = 0.
   else
      MOVEL (min {DCOST, BLANKS2 + AVAIL},
                                RIGHT2, BLANKS2, LEFT2)
      LEFT1 ← LEFT1 + ARB1
      BLANKS ← BLANKS1 + BLANKS2 + RIGHT1 + LEFT2
      ARB ← 0
      end if
   /¢ Set remaining parameters for Case II.A.2
   LEFT ← LEFT1
   RIGHT ← RIGHT2
```

```
        COST ← COST1
        end case II.A.2
     end case II.A

  /¢ Case II.B: The cmc solutions of the sons overlap.
  case II.B: RIGHT1 + LEFT2 < 0
     NEED ← -(RIGHT1 + LEFT2)  /¢ Size of overlap.

     /¢ Case II.B.1: The solutions overlap by a single cell
     /¢ and neither solution has unique endpoint positions.
     case II.B.1: NEED = 1 and ARB1 = 1 and ARB2 = 1
        LEFT ← LEFT1
        RIGHT ← RIGHT2
        BLANKS ← 0
        ARB ← 1
        COST ← max {COST1,COST2}
        end case II.B.1

     /¢ Case II.B.2: The solutions of the sons overlap by
     /¢ more than one cell or the endpoints of at least
     /¢ one solution are unique.
     case II.B.2: NEED > 1 or ARB1 = 0 or ARB2 = 0
        /¢ Reduce overlap with ARB if possible.
        RIGHT1 ← RIGHT1 + ARB1
        LEFT2 ← LEFT2 + ARB2
        NEED ← NEED - ARB1 - ARB2
```

```
/¢ Case II.B.2.a: The left solution costs no more
/¢ than the right and the difference in cost is at
/¢ least as great as the overlap after use of the
/¢ ARB spaces.
case II.B.2.a: COST1 ≤ COST2 and NEED ≤ DCOST
   /¢ Eliminate overlap by moving left solution.
   MOVEL (NEED, RIGHT1, BLANKS1, LEFT1)
   /¢ Make left solution as compact as possible.
   MOVER (min {DCOST - NEED, BLANKS1},
         LEFT1, BLANKS1, RIGHT1)
   /¢ Determine whether new solution has ARB = 1
   /¢ and set remaining parameters.
   if DCOST > NEED and ARB2 = 1
         and RIGHT1 + LEFT2 = 0 then
      ARB ← 1
      BLANKS ← 0
      /¢ Adjust component solutions for ARB.
      LEFT1 ← LEFT1 - 1
      LEFT2 ← LEFT2 - 1
   else
      ARB ← 0
      BLANKS ← BLANKS1 + BLANKS2
      end if
   LEFT ← LEFT1
   RIGHT ← RIGHT2
   COST ← COST2
   end case II.B.2.a
```

```
/¢ Case II.B.2.b: The left solution costs more than
/¢ the right and the difference in cost is at least
/¢ as great as the overlap after use of the ARB spaces.
case II.B.2.b: COST1 > COST2 and NEED ≤ DCOST
   /¢ Eliminate overlap by moving right solution.
   MOVER (NEED, LEFT2, BLANKS2, RIGHT2)
   /¢ Make right solution as compact as possible.
   MOVEL (min {DCOST - NEED, BLANKS2},
         RIGHT2, BLANKS2, LEFT2)
   /¢ Determine whether new solution has ARB = 1
   /¢ and set remaining parameters.
   if DCOST > NEED and ARB1 = 1
         and RIGHT1 + LEFT2 = 0 then
      ARB ← 1
      BLANKS ← 0
      /¢ Adjust component solutions for ARB.
      RIGHT1 ← RIGHT1 - 1
      RIGHT2 ← RIGHT2 - 1
   else
      ARB ← 0
      BLANKS ← BLANKS1 + BLANKS2
      end if
   LEFT ← LEFT1
   RIGHT ← RIGHT2
   COST ← COST1
   end case II.B.2.b
```

```
/¢ Case II.B.2.c: The overlap after use of the
/¢ ARB spaces is greater than the difference in
/¢ cost between the right and left solutions.
case II.B.2.c: NEED > DCOST
   /¢ Move cheaper solution until costs are equal.
   if COST1 < COST2 then
      MOVEL (DCOST, RIGHT1, BLANKS1, LEFT1)
   else
      MOVER (DCOST, LEFT2, BLANKS2, RIGHT2)
      end if
   /¢ The two solutions still overlap but they
   /¢ now have equal costs.  Recompute overlap.
   NEED ← NEED - DCOST
   /¢ If the overlap is even the solutions can
   /¢ be moved outward by the same amount.
   if NEED is even then
      MOVEL (NEED/2, RIGHT1, BLANKS1, LEFT1)
      MOVER (NEED/2, LEFT2, BLANKS2, RIGHT2)
      BLANKS ← BLANKS1 + BLANKS2
      ARB ← 0
   /¢ In the remaining cases the overlap is
   /¢ odd.  The new solution will have ARB = 1
   /¢ iff all blanks are squeezed out by
   /¢ movement of the subsolutions.
```

```
else if BLANKS1 > NEED and BLANKS2 > NEED then
   /¢ New solution will have blanks
   MOVEL (⌈NEED/2⌉, RIGHT1, BLANKS1, LEFT1)
   MOVER (⌈NEED/2⌉, LEFT2, BLANKS2, RIGHT2)
   BLANKS ← BLANKS1 + BLANKS2 + 1
   ARB ← 0
   end if

else if BLANKS1 > NEED and BLANKS2 ≤ NEED then
   MOVEL (⌈NEED/2⌉, RIGHT1, BLANKS1, LEFT1)
   MOVER (⌊NEED/2⌋, LEFT2, BLANKS2, RIGHT2)
   if BLANKS2 > 0 then
      MOVEL (1, RIGHT2, BLANKS2, LEFT2)
      end if
   BLANKS ← BLANKS1 + BLANKS2 + RIGHT1 + LEFT2
   ARB ← 0
   end if
else if BLANKS1 ≤ NEED and BLANKS2 > NEED then
   MOVEL (⌊NEED/2⌋, RIGHT1, BLANKS1, LEFT1)
   if BLANKS 2 > 0 then
      MOVER (1, LEFT1, BLANKS1, RIGHT1)
      end if
   MOVER (⌈NEED/2⌉, LEFT2, BLANKS2, RIGHT2)
   BLANKS ← BLANKS1 + BLANKS2 + RIGHT1 + LEFT2
   ARB ← 0
   end if
else   /¢ BLANKS1 ≤ NEED and BLANKS2 ≤ NEED
   /¢ All blanks will be eliminated
   MOVEL ( NEED/2 , RIGHT1, BLANKS1, LEFT1)
   MOVER (⌈NEED/2⌉, LEFT2, BLANKS2, RIGHT2)
   BLANKS ← 0
```

```
            ARB ← 1
            RIGHT1 ← RIGHT1 - 1
            LEFT2 ← LEFT2 - 1
            end if
         LEFT ← LEFT1
         RIGHT ← RIGHT2
         COST ← max {COST1,COST2} + ⌈NEED/2⌉
         end case II.B.2.c
      end case II.B.2
   end case II.B
end case II

if the cell is not the root cell then
   send LEFT, RIGHT, BLANKS, ARB, COST and #CELLS to
         parent node.
   end if
end UP_TCELLS
```

The procedure DOWN is executed by each T cell C during the downward wave; it computes both the flow over the associated edge of C and constraints on the modified cmc solutions of the sons of C. These computations use the parameters computed by C during the upward wave. Satisfying the constraints imposed on the cmc solution of C is done by shifting and squeezing the modified cmc solutions of its sons. If only one son has a nontrivial solution, shifting and squeezing is done by the procedure SQUEEZE1; if both sons have nontrivial solutions, it is done by SQUEEZE2.

```
procedure SQUEEZE1 (LEFT, BLANKS, RIGHT, LEFTC, RIGHTC)
/¢ Reposition a cmc solution whose parameters are LEFT,
/¢ BLANKS, and RIGHT so that it lies within the boundaries
/¢ given by LEFTC AND RIGHTC.  Replace LEFT, BLANKS and
/¢ RIGHT by the values for the new solution.  This
/¢ procedure assumes a solution is possible, i.e.,
/¢    LEFTC - LEFT + RIGHTC - RIGHT < BLANKS
/¢ SQUEEZE1 is used by DOWN.

/¢ Calculate the number of cells that overlap
/¢ on the left and right.
LDEMAND ← LEFTC - LEFT
RDEMAND ← RIGHTC - RIGHT

/¢ If both LDEMAND and RDEMAND are less than or equal
/¢ to zero, the present solution is adequate.  If
/¢ either demand is positive, the solution must be
/¢ moved.  It suffices to move it in the direction of
/¢ the largest demand.

if LDEMAND > RDEMAND and LDEMAND > 0 then
   MOVER (LDEMAND, LEFT, BLANKS, RIGHT)
else if RDEMAND > LDEMAND and RDEMAND > 0 then
   MOVEL (RDEMAND, RIGHT, BLANKS, LEFT)
   end if
end SQUEEZE1
```

```
procedure SQUEEZE2 (LEFT1, BLANKS1, RIGHT1, LEFT2, BLANKS2,
      RIGHT2, LEFTC, RIGHTC)
/¢ Alter two subsolutions of a rightmost cmc solution so
/¢ that the two new subsolutions do not conflict and lie
/¢ within the boundary constraints LEFTC and RIGHTC.  This
/¢ procedure assumes a solution is possible, i.e.,
/¢   LEFTC - LEFT1 + RIGHTC - RIGHT2 <
/¢                  BLANKS1 + BLANKS2 + RIGHT1 + LEFT2
/¢ SQUEEZE2 is used by DOWN.

/¢ Calculate the number of cells that overlap
/¢ on the left and right.
LDEMAND ← LEFTC - LEFT1
RDEMAND ← RIGHTC - RIGHT2

/¢ Satisfy the left constraint.
if LDEMAND > 0 then MOVER (LDEMAND, LEFT1, BLANKS1, RIGHT1)
   end if
/¢ Satisfy the right constraint.
if RDEMAND > 0 then MOVEL (RDEMAND, RIGHT2, BLANKS2, LEFT2)
   end if
```

```
/¢ Measure the overlap of the two subsolutions and if a
/¢ conflict has arisen, resolve it.  Note that no overlap
/¢ can occur unless MOVEL or MOVER has caused the number of
/¢ available end cells and internal blanks of one subsolution
/¢ to go to zero.  In this case, the other subsolution must
/¢ have space available to absorb the overlap.
OVERLAP ← -(RIGHT1 + LEFT2)
if OVERLAP > 0 then
   /¢ Find subsolution with space available and move it
   /¢ to eliminate the overlap.  The number of available
   /¢ end spaces on the left is LEFT1 - LEFTC.
   if BLANKS1 + LEFT1 - LEFTC > 0 then
      MOVEL (OVERLAP, RIGHT1, BLANKS1, LEFT1)
   else
      MOVER (OVERLAP, LEFT2, BLANKS2, RIGHT2)
      end if
   end if
end SQUEEZE2
```

```
procedure DOWN
/¢ Executed by each T cell during the downward wave.

/¢ Step 1**********************************************
/¢ Obtain constraints on solution to storage management
/¢ problem for L array descendants and a flow value into
/¢ the leftmost L cell descendant.

case I: The T cell C is the root of the T array.
   /¢ Determine if a solution exists.
   if LEFT + RIGHT + BLANKS + ARB < 0 then
      no solution exists; print message and return
   else  /¢ A solution exists.
      /¢ Impose constraints on solution boundaries
      /¢ and set flow into the leftmost L cell to 0.
      LEFTC, RIGHTC, DFLOW ← 0
      end if
   end case I

case II: The T cell C is not the root of the T array
   receive LEFTC, RIGHTC and DFLOW from parent T cell
   end case II
/¢ End of Step 1
```

```
/¢ Step 2**********************************************

/¢ Compute flow for the associated edge of T cell C.

/¢ Choose rightmost cmc solutions arbitrarily

LEFT ← LEFT + ARB

LEFT1 ← LEFT1 + ARB

LEFT2 ← LEFT2 + ARB

/¢  Case I: No L cell descendants of C are occupied.

case I: LEFT1 = #CELLS/2 and RIGHT2 = #CELLS/2

   if LEFTC > #CELLS/2 then

      FLOW ← (LEFTC - #CELLS/2)

   else if RIGHT C > #CELLS/2 then

      FLOW ← -(RIGHTC - #CELLS/2)

      end if

   else FLOW ← 0

      end if

   end case I

/¢ Case II: All left descendant L cells of C are unoccupied

/¢ but some right descendant L cells of C are occupied.

case II:  LEFT1 = #CELLS/2 and RIGHT2 ≠ #CELLS/2

   SQUEEZE1 (LEFT, BLANKS, RIGHT, LEFTC, RIGHTC)

   if LEFTC > #CELLS/2 then FLOW ← (LEFTC - #CELLS/2)

   else if LEFT < #CELLS/2 then

      FLOW ← LEFT - #CELLS/2

      end if

   else FLOW ← 0

      end if

   end case II
```

```
/¢ Case III: Some left descendant L cells of C are occupied
/¢ but all right descendant L cells of C are unoccupied.
case III:  LEFT1 ≠ #CELLS/2 and RIGHT2 = #CELLS/2
   SQUEEZE1 (LEFT, BLANKS, RIGHT, LEFTC, RIGHTC)
   if RIGHTC > #CELLS/2 then FLOW ← -(RIGHTC - #CELLS/2)
   else if RIGHT < #CELLS/2 then
      FLOW ← #CELLS/2 - RIGHT
      end if
   else FLOW ← 0
      end if
   end case III

/¢ Case IV: There are both left and right descendants of
/¢ C which are occupied L cells.
case IV:  LEFT 1 ≠ #CELLS/2 and RIGHT2 ≠ #CELLS/2
   SQUEEZE2 (LEFT1, BLANKS1, RIGHT1, LEFT2, BLANKS2,
         RIGHT2, LEFTC, RIGHTC)
   if RIGHT1 < 0 then FLOW ← -RIGHT1
   else if LEFT2 < 0 then FLOW ← LEFT2  end if
   else FLOW ← 0
      end if
   end case IV
/¢ End of Step 2
```

```
/¢ Step 3*********************************************

/¢ Compute constraints for the solutions of left and right sons.


LEFTC1 ← LEFTC

RIGHTC1 ← -FLOW

LEFTC2 ← FLOW

RIGHTC2 ← RIGHTC

/¢ End of Step 3


send LEFTC1, RIGHTC1 and DFLOW to the left son of C.

send LEFTC2, RIGHTC2 and FLOW to the right son of C.


end DOWN
```

```
procedure UP_LCELLS#2
/¢ Modified L cell procedure to minimize maximum flow during
/¢ storage management; see Section IX.
/¢
/¢ Executed by each L cell at the beginning of the upward
/¢ wave.  This procedure has no input other than the cond-
/¢ ition of the L cell.

#CELLS ← 1
if the cell is unoccupied then
   LEFT ← 1
   RIGHT ← 1
   BLANKS, ARB, COST ← 0
else
   let p and q be the number of cells requested on the left
         and right respectively by the symbol in the L cell.
   LEFT ← -⌈(p+q)/2⌉
   RIGHT ← -⌈(p+q)/2⌉
   if p + q is even then ARB ← 0 else ARB ← 1
      end if
   COST ← ⌈(p+q)/2⌉
   BLANKS ← 0
   end if

send LEFT, RIGHT, BLANKS, ARB, COST and #CELLS to parent node
end UP_LCELLS#2
```