PACKING VIEWS OF A DATABASE INTO STORAGE
(PRELIMINARY VERSION)

by

James Bitner
Department of Computer Science
University of Texas at Austin
Austin, Texas 78712

TR-186

November 1981

Abstract: In a database with several different views, we can explicitly store the indices for the records in a given view or recalculate them each time the view is requested. We study the problem of deciding which views should be explicitly stored given that the space for storing the indices is limited. We show that the problem is NP-complete if either the set of sizes of the views or the relationship between the views is "too complex". We also develop an algorithm for selecting the views to store if both are not "too complex" and indicate when it is truly practical.

Acknowledgements: I am grateful to Don Simon for his help in preparing this paper and (in advance!) Umesh Dayal for reviewing a preliminary version. The basic idea of using lists to implement the vectors in Section 3 was originated by Larry Davis.

1. Introduction

Many database systems (such as ZETA [MYL75], System R [AST76], and INGRESS [STO76]) allow the specification of views. This mechanism allows a user to restrict his/her attention and transactions to a limited part of a (much) larger data base. See, for example, Figure 1.1 where our data base consists of one relation containing information about students, and another containing information about people who are employed. Several views of this data base are shown: for example, one user might only be concerned with employed female graduate students and another only with male students. The relationship between views can be specified by a directed acyclic graph (DAG) where the view corresponding to node x can be constructed by performing some operation(s) on views corresponding to the set of nodes {y|there is an arc from x to y}, i.e. x's immediate successors. Some nodes have no successors; these are the base relations and cannot be calculated.

Since a given user's transactions are restricted to a given subset of the database, we are faced with the unappealing task of recomputing this view on every transaction. An alternative is to store the indices of the records in that view so that it need not be recomputed. This, of course, incurs an overhead in storage. Given enough storage, we could store all the views explicitly. However if this is not the case, we must choose which views to store and which to calculate. This is the question addressed by this paper.

A DISCUSSION OF PREVIOUS WORK GOES HERE

We assume the user supplies us with the following information:

1. A DAG indicating which views are needed for the construction of each view.

2. An estimate of the cost c(v) of constructing each view given that its sons have already been constructed. This cost gives the difference in cost (i.e., the relative advantage) of storing the view explicitly versus calculating it and should include time required to maintain the indices when and if updates occur. (If updates are very frequent, the cost could even be negative.)

3. An estimate of the size s(v) of each view.

4. An estimate of the probability p(v) a given view will be required by a transaction.

5. The maximum amount of space (M) available for the storage of indices for views.

We are then to determine the subset of views that can be stored in the given amount of space and minimize the expected cost.

In order to state the problem formally, we need to develop some notation.

Definition: Given a DAG with vertex set V, v in V, and V' a subset of V, let

reach(v,V') = {x|there is a path from v to x containing no vertices in V'} and

<u>tail(v,V')</u> = {x|there is a path from x to v containing no vertices in V'}

The cost of choosing a set V' can be calculated by examining how much it costs to construct each node. Node v will be requested with probability p(v). If v is not in V', it must be constructed from its immediate successors. If any of them is not in V', it must be constructed from its immediate successors, and so on. Clearly, the cost of constructing v is the sum of the cost of constructing all vertices in reach(v,V'). (Note that if v is in V', reach(v,V') is empty, and hence v contributes nothing to the cost.) Therefore, we have

<u>Definition</u>: Given a digraph G with vertex set V and functions p and c and V' a subset of V

$$\mathrm{COST}(V') = \sum_{u \text{ in } V} p(u) * \sum_{v \text{ in reach}(u,V')} c(v)$$

The cost, of course, also depends on G, V, p and c, but we omit this to simplify notation. We can now formally state the problem.

Problem 1

Given

1. A finite directed acyclic graph (DAG) G with vertex set V
2. A probability p(v) for each v in V
3. A construction cost c(v) for each v in V, with c(v) = infinity for leaves
4. A size $s(v) \geq 0$ for each v in V
5. An amount of storage $M \geq 0$

Find a subset V' of V such that

$$\sum_{v \text{ in } V'} s(v) \leq M \quad \text{and} \quad \text{COST}(V') \text{ is minimized}$$

<u>Notes on Problem 1</u>: Though we could require the p(v)'s to sum to one, it is not necessary and later will be convenient not to do so. In this case, the p(v)'s can be thought of as "weights". There is no harm in this, because the "true" cost can easily be obtained by dividing the cost by the sum of the p(v)'s. The cost of constructing a leaf is set to infinity to force all leaves to be stored explicitly. Where necessary, we define infinity * 0 = 0. V' will be the set of views that are stored explicitly. The first requirement of V' is that all the views will fit in the space provided, and the second requirement is that the expected cost of constructing a view is minimized.

<u>Definition</u>: If V' denotes the subset of views which are stored

explicitly, we call V' a *marking* of G. Nodes in V' are called *marked* (or *closed*) nodes and nodes not in V' are called *unmarked* (or *open*) nodes.

The following reformulation of the cost will be convenient later:

Lemma 1.1:

$$COST(V') = \sum_{u \text{ in } V} c(u) * \sum_{v \text{ in } tail(u,V')} p(v)$$

Proof: We look at how often each node must be constructed. A node v must be constructed each time an ancestor of v is requested, provided that there is a path from the ancestor to v consisting solely of nodes not in V'. (Note that if v itself is explicitly stored, then it will never have to be constructed and contributes nothing to the total.) Therefore the contribution of a node to the total cost is the cost of the node multiplied by the probability of any node in tail(v,V') being requested. []

As stated before, we will study the problem of minimizing COST(V'). We show that, in general, this is a very hard problem if either the set of sizes or the DAG is "too complex". In Section 2, we show that if the sizes are arbitrary integers, the problem is NP-complete, even if the DAG is only a tree. (Though the sizes may, of course, be real, the problem is NP-complete even if they are restricted to being integers.) In Section 4, we show that if the DAG is unrestricted, the problem is NP-complete even if every node is equally likely and all costs and sizes equal 1. Though the problem where the DAG is constrained to be a tree is, in general, intractable, we develop a dynamic programming algorithm to solve this problem and indicate some cases where it is truly practical.

2. Section 2

In this section, we show Problem 1 is NP-complete when the costs and sizes are arbitrary integers even if the graph is extremely simple. We first reformulate Problem 1 as a decision problem:

Problem 2: Given G, V, p, c, s and M as in Problem 1 and $F \geq 0$ where c and s are integer functions, is there a subset V' of V such that:

(1) $\sum_{x \text{ in } V'} s(x) \leq M$ and

(2) $\mathrm{COST}(V') \leq F$?

We show Problem 2 is NP-complete by reducing the Partition Problem to it. This problem was proven NP-complete in [KAR 72] and is stated as follows:

Partition Problem: Given positive integers $a_1, \ldots, a_n$ summing to S, is there a subset P of $\{1, \ldots, n\}$ such that

$$\sum_{i \text{ in } P} a_i = S/2?$$

We prove that Problem 2 is NP-complete for a variety of simple classes of graphs. Thus, Problem 2 for a general graph must also be NP-complete.

Theorem 2.1: Problem 2 is NP-complete even if the graph is of the form shown in Figure 2.1.

Proof: Problem 2 is obviously in NP since we can non-deterministically select a subset V' of V and check the two conditions in polynomial time.

We now reduce the Partitioning Problem to Problem 2. Given an instance of the Partitioning Problem, define an instance of Problem 2 where $V = \{v_1, \ldots, v_n, w_1, \ldots, w_n\}$ and for all i there is an edge from v_i to w_i. Let $M = S/2$, $F = S/2$, and

$$c(v_i) = a_i \qquad s(v_i) = a_i \qquad p(v_i) = 1$$

$$c(w_i) = \infty \qquad s(w_i) = 0 \qquad p(w_i) = 1$$

We show this instance of the Partition Problem has a solution iff this instance of Problem 2 has a solution.

Suppose P is a solution to this instance of the Partition Problem. Define $V' = \{v_i \mid i \text{ not in } P\} \cup \{w_i, \ldots, w_n\}$. To calculate COST(V') note that reach(v) is empty unless v is not in V' in which case reach(v) = {v}. Hence

$$\mathrm{COST}(V') = \sum_{x \text{ not in } V'} c(x)*p(x) = \sum_{\{i \mid v_i \text{ not in } V'\}} a_i = \sum_{i \text{ in } P} a_i = S/2$$

and the size required to store V' is

$$\sum_{x \text{ in } V'} s(x) = \sum_{\{i \mid v_i \text{ in } V'\}} a_i = \sum_{i \text{ not in } P} a_i = S - S/2 = S/2$$

Therefore V' meets both the cost and space requirements and is a solution to this instance of Problem 2.

Suppose, on the other hand, that V' is a solution to this instance of Problem 2. Let $P = \{i \mid v_i \text{ not in } V'\}$. Note that $w_i, \ldots, w_n$ must be in V' or else the cost would be infinite. Let SIZE(V') be the space requirement of V'. Then COST(V') + SIZE(V')

$$= \sum_{x \text{ not in } V'} c(x) * p(x) + \sum_{x \text{ in } V'-W} s(x)$$

$$= \sum_{\{i \mid v_i \text{ not in } V'\}} a_i + \sum_{\{i \mid v_i \text{ in } V'-W\}} a_i$$

$$= S$$

where $W = \{w_1, \ldots, w_n\}$ can be disregarded in the second sum because $s(w_i) = 0$. Since V' meets the space requirement, $\mathrm{SIZE}(V') \leq S/2$. Since $\mathrm{COST}(V') + \mathrm{SIZE}(V') = S$, we have $\mathrm{COST}(V') \geq S/2$. Since V' meets the space requirement we have $\mathrm{COST}(V') \leq S/2$. Hence $\mathrm{COST}(V') = S/2$. This gives

$$S/2 = \mathrm{COST}(V') = \sum_{\{i \mid v_i \text{ not in } V'\}} a_i = \sum_{a_i \text{ in } P} a_i$$

proving P is a solution to this instance of the Partitioning Problem. []

Theorem 2.2: Problem 2 is NP-complete even if the graph is a tree of height 2 or a binary tree.

Proof: Adding another node to the construction in Theorem 2.1 with size $> M$ as the father of all v_i shows the case of a tree of height 2 is NP-complete. Adding log n levels of nodes with size $> M$ and identical cost proves the other case. Both proofs are similar to Theorem 2.1. []

Section 3: A Dynamic Programming Solution

In this section, we restrict the DAG describing the relationship between the views to be a tree and the size of each node to be an integer. Stated formally,

Problem 3

Given G, V, p, c, s, and M, as in Problem 1. If G is a tree, and s(v) is integer valued for each v in V, find a subset V' of V satisfying the conditions of Problem 1.

In Section 3.1 we develop a high-level algorithm to solve Problem 3. In Section 3.2 we use the idea of a "convolution" of vectors to refine the high level algorithm into a PASCAL-like implementation. We analyze this algorithm and find it takes time $O(n^2 M^2)$ and space $O(n^2 M)$ in the worst case. Section 3.3 discusses a specific method of representing vectors using lists, and Section 3.4 analyzes the savings that occur using this representation instead of a straightforward one. Though the algorithm is not practical for all cases, Section 3.4 indicates and analyzes some cases where it is truly practical. Section 3.5 extends the algorithm to the case of inverted trees (i.e. a tree with the direction of each edge reversed.) Finally, Section 3.6 gives some sample execution statistics for the program.

Section 3.1: A High Level Algorithm

We will choose to use the reformulation of the cost function given by Lemma 1.1 to calculate the cost since the tail of a node in a tree is especially simple. The obvious way of determining the minimum cost of the subtree rooted at node x, given m units of storage, is to first assume x will be closed, split the remaining m-s(x) units among node x's subtrees, recursively calculate the minimal cost for each subtree with its given allocation, and minimize over all partitions of m-s(x). Then second, to repeat the procedure, assuming x is open and split m units among x's subtrees. However, this simple method does not quite work, as we cannot calculate a node's cost unless we know its tail. Therefore we repeat the above procedure at each step assuming a tail of length "tlen" for tlen = 0,...,depth(x).

This recursive solution is inefficient because the cost of a subtree will be calculated many times. To avoid this wasted effort, we will use dynamic programming. That is, we will find optimal marking for all of a node's subtrees and then merge these solutions together in a bottom-up manner (i.e. starting with the leaves) obtaining solutions to larger and larger subtrees until we have a solution for the entire tree. The savings come from the fact that once the cost for a given subtree is computed, it is stored in an array and subsequently can be simply looked up instead of completely recomputed.

A high-level algorithm is given in Figure 3.1. To simplify the program, we assume the sum of the sizes of the leaves has been subtracted from M and that all the leaves have been deleted. This is done because the leaves must be stored. Also, the details of recording the set of closed nodes for achieving a given cost are omitted. This is easily done (at least in the final version of the algorithm).

In this algorithm, we sequence through the nodes in postorder. At each node x, we first calculate the vector minclosed, where minclosed[m] gives the minimum cost for the subtree rooted at x using m units of storage. To calculate this, we distribute m-s(x) units among x's subtree in all possible ways. Note that each son will have a tail of size 0 since x is closed. To calculate the vector minopen, we assume x is open and has a tail of length tlen. We distribute our m units in all possible ways among x's subtrees. In this case each son of x will have a tail of length tlen+1. Finally, mincost[x,tlen,m] is determined by taking the minimum of minclosed[m] and minopen[m].

Section 3.2: A PASCAL Algorithm and its Analysis

Clearly, the dominant cost in the algorithm is computing

$$\min_{m_1+\dots+m_k=m-s(x)} \sum_{i=1}^{k} \text{mincost}[y_i, 0, m_i]$$

and

$$c(x) * \text{tailprob} + \min_{m_1+\dots+m_k=m} \sum_{i=1}^{k} \text{mincost}[y_i, \text{tlen}+1, m_i]$$

A subprogram common to these two calculations is: Given k vectors $A_1,\dots,A_k$, calculate the vector D defined by:

$$D[j] = \min_{m_1+\dots+m_k=j} \sum_{i=1}^{k} A_i[m_i]$$

This operation can be neatly defined in terms of _convolutions_. Given two vectors A and B define the _convolution of A and B_ to be the vector C defined by

$$C[k] = \min_{0\le i\le k} (A[i] + B[k-i])$$

We write C = A * B. This is the standard convolution operation with "+" replacing "*" and "min" replacing "+". The vector D can easily be expressed in terms of convolutions as $A_1 * A_2 * \dots * A_k$ (note that * is associative).

Modifying the program to use convolutions is straightforward, and the result is given in Figure 3.2. To prove it is correct we need to show two relations. (Define for i = 1,...,k, $Y_i[m]$ = mincost[y_i,0,m] and $Z_i[m]$ = mincost[y_i,tlen+1,m].) First, that the vector minclosed can be expressed as $C_1 * Y_1 * \dots * Y_k$ where C_1 is a vector such that $C_1[j] = \infty$ for $j < s(x)$ and $C_1[j] = 0$ for $j \geq s(x)$. Second, that the vector minopen can be expressed as $C_2 * Z_1 * \dots * Z_k$ where C_2 is a vector such that $C_2[j]$ = c(x) * tailprob for all j. Both relations are easily verified and their proofs are omitted.

We now analyze the algorithm. The space requirement is

$$M * E_v[\text{depth}(v) + 1] = O(n^2 M)$$

because each node v requires depth(v) + 1 vectors. To analyze the time requirement note that [depth(v) + 1] * [degree(v)] convolutions are done at node v. The obvious way of performing a convolution requires time $O(M^2)$. The total cost for the algorithm is then

$$E_v[\text{depth}(v) + 1] * \text{degree}(v) * O(M^2)$$

This can be trivially bounded by $O(n^3 M^2)$. The actual bound, of course, depends on the tree; for example, a complete binary tree has a bound of $O(M^2 n \log n)$. A surprisingly bad case is a "linear" tree where each node has one son (except for the one leaf). It has a bound of $O(n^2 M^2)$. However, this is due to the fact that we (for simplicity) perform two convolutions (using the "C_1" and "C_2" vectors just described) where simpler and faster operations could be done. (For example, a convolution, requiring time $O(M^2)$, is not required to simply add a constant to every component of a vector!) We will see later that these degenerate convolutions can be performed very quickly; however, it would be unfortunate for the worst case behavior to be determined by this degenerate case which could easily be removed.

Therefore, we assume the two trivial convolutions are replaced by a more efficient operation. The cost drops to

$$\sum_v [\text{depth}(v) + 1] * [\max(\text{degree}(v) - 1, 0) * O(M^2) + O(M)]$$

since degree(v) - 1 convolutions are required unless degree(v) = 1 in which case we perform an O(M) operation on the vector.

The following theorem shows a bound of $O(n^2 M^2)$ is still achievable.

<u>Theorem 3.1</u>: $f(T) = \sum_v [\text{depth}(v) + 1] * \max(\text{degree}(v) - 1, 0)$

is maximized by the tree shown in Figure 3.3. For that tree, $f(T) = O(n^2)$.

<u>Proof</u>: We begin by showing that if a tree, T, has a node x, which has a non-leaf y and another node z among its sons then there exists another tree with the same number of nodes giving a larger value of f. Consider the tree, T', where z has been removed as a son of x and been made a son of y. How has this change affected f? Every node in z's subtree has had its depth increased by one. This will not decrease f.

Further, y has had its degree increased by one. Since y had at least one son originally, the contribution it makes has been increased by depth(y). Similarly, the contribution x makes has been decreased by depth(x). Since depth(y) > depth(x), f(T) < f(T').

This means the tree must have the form shown in Figure 3.5 where k satisfies $0 \le k \le n$. The only node making a non-zero contribution to f is node x. Clearly, the value of f is k * (n-k) which is maximized when k=n/2, giving a value of $n^2/4$ for f. []

Section 3.3: Using a List to Represent a Vector

The performance of the algorithm is not very acceptable due to the M^2 term in the time requirement. This is especially unfortunate because most of the vectors contain very few distinct elements. This forms the idea for an important improvement. Instead of storing the entire vector A explicitly we represent it as a sequence of ordered pairs $(a_1, b_1), \ldots, (a_k, b_k)$ where the pair (a_i, b_i) means that

$$A[x] = {}^{-} \quad \text{for } 0 \le x < a_1$$

$$A[x] = b_i \quad \text{for } a_i \le x < a_{i+1} \quad \text{for } i = 1..k-1$$

$$A[x] = b_k \quad \text{for } a_k \le x$$

(See Figure 3.6 for an example.) In the worst case, the list will be M elements long, and we have gained nothing. However, it takes a while for the length of the lists to build up, and until it does (if it ever does), we enjoy enormous savings of space and time (see below).

We now discuss how to form the convolution C of the vectors A and B using this list representation. First, rewrite the definition of C as

$$C[k] = \min_{0 \leq i,j \leq k} (A[i] + B[j])$$

which is clearly equivalent to the original definition because A and B are monotonically non-increasing. (This is easily seen; allowing more storage cannot increase the cost.)

Definition: i is a _critical point_ of a vector A iff $A[i] < A[i-1]$ or $i = 0$.

We now show that to determine C[k] it suffices to take the minimum of $A[i] + B[j]$ where i is a critical point of A and j is a critical point of B. If i and j are not both critical points, let i' and j' be the first critical points less than or equal to i and j, respectively. Since $A[i] + B[j] = A[i'] + B[j']$ it does not hurt to discard the pair (i,j) when we take the minimum because the pair (i',j') _will_ be considered. Therefore to find C[k] we minimize over $A[i] + B[j]$ where $i + j \leq k$ and i and j are critical points of A and B respectively.

To find the entire C vector, we generate A[i] + B[j] for all i and j which are critical points of A and B respectively. C[k] is the minimum of those with $i + j \leq k$. If we sort the values A[i] + B[j] on i+j, C could be found by sequencing through this list and keeping track of the minimum so far. However, we do not want C, only its critical points. These can be found as follows (this is illustrated in Figure 3.6):

Finding the Critical Points of $C = A * B$

(Let A and B be represented by the list of pairs $\{(a_i, x_i)\}$ and $\{(b_i, y_i)\}$ respectively.)

Step 1: Generate all pairs $(a_i + b_j, x_i + y_j)$ such that $a_i + b_j \leq M$.

Step 2: Sort the pairs

Step 3: Sequence through the pairs, eliminating (c_i, z_i) if there is a pair (c_j, z_j) with $c_j \leq c_i$ and $z_j \leq z_i$.

Two possibilities suggest themselves for the sorting: some O(n log n) sort, or a bucket sort. We choose to use the former because it does not require an array of size M which would be excessive in space and time (the time would be especially excessive for a very short list). Thus we can form the convolution of two lists of length a and b in time O(ab log ab).

Section 3.4: Analysis for the List Representation

This method of forming the convolution does not improve the worst case performance since we may have a=b=M, but in practice it results in substantial savings. Before discussing some cases where this method is practical, we need a lemma to bound the growth of these lists.

Lemma 3.1: Given a tree with n nodes with sizes $S_1,\ldots,S_n$, let k be the number of distinct values that can be produced by summing the elements of a subset of the multiset $\{S_1,\ldots,S_n\}$. The length of the list for the root is bounded by

$$2^n,\ k,\ \sum_{i=1}^{n} S_i,\ \text{and } M.$$

Proof: Clearly the list for a leaf has length 2. Since the convolution of two lists with lengths a and b has length at most a*b, the 2^n bound follows by induction. The second bound follows from the fact that a critical point corresponds to an actual marking of the tree (which will have size which is equal to the sum of the elements of $\{S_1,\ldots,S_n\}$). The third is a corollary of the second, and the fourth is obvious. []

The following theorem uses Lemma 3.1 to analyze the complexity of some tractable cases:

Theorem 3.2: Let c be a constant then

(a) On a complete binary tree whose node sizes are all multiples of M/c the algorithm requires storage $o(cn \log n)$ and time $o(c^2 \log c * n \log n)$

(b) On a complete binary tree whose node sizes are all less than or equal to c the algorithm requires storage $o(cn^2)$ and time $o(n^2 c^2 [\log c + \log n])$

Proof: In case (a) all lists have size at most c. In case (b), the list for a k node subtree has length at most ck. The time bounds follow from the fact that the cost equals

$$\sum_{v} [\text{depth}(v) + 1] * [\max(\text{degree}(v) - 1, 0) * O(M^2) + O(M)]$$

The space bounds follow on the assumption that we release all the blocks in a list once it is used. (Note that once a list is convoluted with another, it is no longer needed.) [ ]

Section 3.5: Extension to Inverted Trees

The algorithm can be extended to handle inverted trees. An inverted tree is a tree with the direction of each arc reversed. Suppose we are given an instance of Problem 1 with G as an inverted tree and c, s, and p as the cost, size and probability functions. Let COST(V',G) denote the cost of a marking V' for this problem, and reach(u,V',G) the function reach(u,V') for the graph G. We have by definition

$$\mathrm{COST}(V', G) = \underset{u \text{ in } V}{E} \; p(u) \; * \underset{v \text{ in } \mathrm{reach}(u, V', G)}{E} c(v)$$

We define a second instance of Problem 1 whose DAG is G' which is G with the direction of each edge reversed and cost size and probability functions c', s' and p'. Let COST(V',G') be the cost of marking V' in this second problem and tail(u,V',G') be the function tail(u,V') for G'. By Lemma 1.1 we have

$$\mathrm{COST}(V', G') = \underset{u \text{ in } V}{E} \; c'(u) \; * \underset{v \text{ in } \mathrm{tail}(u, V', G')}{E} p'(v)$$

and since reach(u,V',G) = tail(u,V',G')

$$\mathrm{COST}(V', G') = \underset{u \text{ in } V}{E} \; c'(u) \; * \underset{v \text{ in } \mathrm{reach}(u, V', G)}{E} p'(v)$$

If we choose c'(v) = p(v), p'(v) = c(v) and s'(v) = s(v) we will have COST(V',G) = COST(V',G'). That is, to find the cost for an inverted tree, we merely reverse the direction of each edge and interchange the c and p functions. Using the algorithm to solve this problem gives us the cost for the original inverted tree.

Section 3.6: Execution Results

The algorithm was implemented in PASCAL on a DEC-10 and was timed on two non-trivial test cases. The graph shown in Figure 3.7 required 1.446 seconds and required 214 blocks of 5 words for storing the lists (i.e. about 1K words). This is the maximum number of blocks in use at any time; the program is very careful to return blocks which are no longer in use to the pool of free storage. For the graph shown in Figure 3.8, 3.109 seconds and 404 blocks (i.e. about 2K words) were required.

For purposes of comparison, a program using a backtrack branch-and-bound type search was described by Davis and Roussopoulos in [ROU 79]. It was implemented on a CDC 6600 (a faster machine than the DEC) and required 38.5 seconds and 666.4 seconds to process Figures 3.7 and 3.8 respectively. Our algorithm is 30 to 200 times faster and requires very modest amounts of space. (The comparison is somewhat unfair because their algorithm handles the general case where the DAG is not nnecessarily a tree.) For these examples (with "random" sizes and costs) the algorithm is very practical in both its time and space requirements.

Another nice property of the algorithm is that to calculate the cost of a tree for some M, we automatically calculate the cost for all smaller storage allocations, thus making the nature of the time/space tradeoff very clear. This information can guide the user in deciding how much space to allocate for the storing of indices.

Section 4.

In this section, we show Problem 2 is NP-complete, even if the cost and size of every node is one and all nodes are equally probable. To show NP-completeness, we will reduce Boolean Satisfiability (proven NP-complete in [COO 71]) to the following simplification of Problem 2.

Problem 4.

Given

1. A finite directed acyclic graph with vertex set W
2. Size of marking set $M > 0$
3. Cost $F > 0$

is there a subset W' of W such that

1. $|W'| = M$
2. $F \geq$ number of pairs of vertices u, v in W (with $u \neq v$) for which there is a path from u to v which does not pass through any vertex in W' ?

(Note that the cost is defined in a slightly different manner from Problem 2, this will simplify matters later.) To see the correspondence between the two problems, we show how to map each instance of Problem 4 into an instance of Problem 2: Take W, add one node x, and add an edge from every other vertex to x. This forms the graph for Problem 2. Let $c(x) = \tilde{}$, $s(x) = 1$, $p(x) = 1$. For all other nodes let $c(w) = s(w) = p(w) = 1$ and let the size bound be $M + 1$. Finally, let the target cost be $F + |W| - M$ (since Problem 2 counts one unit cost for each of the $|W| - M$ unmarked nodes and Problem 4 does not.)

Since this new problem can be reduced to Problem 2, we must show that Boolean Satisfiability can be reduced to Problem 4.

<u>Boolean Satisfiability</u>: Given a boolean expression B in conjunctive normal form, with m clauses and n distinct literals, is there a set of truth values which satisfy B?

At this point, we define three quantities and some terminology that will become important in the following step of the proof.

<u>Definition</u>: The number of pairs of distinct vertices u, v in W that has a path that does not go through W' will be referred to as the <u>COST of marking W'</u>, which we will write as <u>COST(W')</u>. We will also use <u>COST(W',X,Y)</u>, for W', X, Y all subsets of W, to denote the number of pairs of distinct vertices u, v, with u in X, and v in Y such that there is a path from u to v not containing any vertex in W'.

We now define a reduction from Boolean Satisfiability to Problem 4.

Set k to be some number such that

$$4kmn^2 + kmn + 4kmn^2(n-1) + mn(n-1)/2 \leq k^2 - 1$$

Clearly, k can be chosen so that it is bounded by a polynomial in m and n. Define the cost, F by

$$F = 2k^2 - 1 - kmn - 4kmn^2(n-1) + 7/2 * n(n-1)k^2 .$$

Finally, set M, the number of nodes we can mark, to m * n.

Given a boolean expression B, we construct a graph that has a marking with "low" cost provided that B is satisfiable. We will then add nodes and arcs to insure that a marking of the graph that is of low cost corresponds to a set of variables that satisfies B.

Our initial graph consists of a 2n x m matrix, with one column for each clause and one row for each variable and one row for each complement. Arcs only connect vertices in the same column. There is an arc from the row corresponding to variable a in column i to the row corresponding to variable b in column i iff clause i of the boolean expression B is of the form (...+ a + b +...). We assume no clause has the same literal occurring twice or has both a literal and its complement. Thus, each clause has at most n variables.

Next, we add a set of k nodes (called the A-nodes) to the top of the graph. For each clause, there is an arc from each of the A-nodes to the vertex corresponding to the first variable of the clause in the column that corresponds to the clause. Finally, we add at the bottom of the graph another set of k nodes, called the B-nodes. For each clause, there is an arc from the vertex in the row corresponding to the last variable of the clause, and in the column corresponding to the clause, to each of the B-nodes. (See Figure 4.1 for an example.)

Thus, if B is satisfiable, we can mark all the vertices in the rows corresponding to variables which are true. There will be no path from any A-node to any B-node, and so the cost of the marking will be "low". If there is a path from an A-node to a B-node, then there is a path from all A-nodes to all B-nodes and the cost is at least k^2, which will be "high".

Definition: A marking which blocks all paths from the A-nodes to the B-nodes is called blocking.

The above is not sufficient; we might be able to achieve a low cost by marking only part of a row or marking nodes in one row and in a row which corresponds to the complement of the literal corresponding to the first row. We must guarantee a marking of low cost corresponds to a choice of true/false values for the literals.

Definition: A marking is called proper iff

1. A row is entirely marked or entirely unmarked, and
2. For each variable x, either the row corresponding to x or not x is marked, but not both.

To insure that only proper markings have a low cost, we add, for each pair of rows (except for the case when the rows correspond to a variable and its complement) k nodes to the left of the graph. There is an arc from each of these k nodes to each of the vertices in both rows. In addition, we add a set of k nodes, called the D-nodes, to the right of the graph, and add an arc from each node of the matrix to each of the D-nodes (see Figure 4.2). The set of all the nodes on the left is called the C-nodes. There are $2kn(n-1)$ of them, since for each of the $n(n-1)/2$ pairs of distinct variables, there are 4 sets of k nodes.

A proper marking cannot block all the sets of C-nodes from reaching the D-nodes. However, it does block $n(n-1)/2$ and we will define the resulting cost of $3k^2 n(n-1)/2$ to be "low". An improper marking will incur an additional cost of k^2, which we will define to be "high".

Finally, we will add an arc from each of the A-nodes to each of the D-nodes, and an arc from each of the C-nodes to each of the B-nodes. This way, we are sure every A-node can reach every D-node, and every C-node can reach every B-node independently of how the matrix is marked. Schematically, the graph now looks as shown in Figure 4.3.

Now, we are ready to prove the theorem:

The graph constructed above has a marking W', $|W'| = m * n$, with $COST(W') \leq F$ iff the boolean expression B is satisfiable.

Clearly, we have already established the following lemma:

<u>Lemma 4.1</u>: There is a proper blocking marking W' for the graph iff B is satisfiable.

If we can prove that a marking W' has cost $\leq F$ iff it is proper and blocking, then we have established the theorem. This is done in the following four Lemmas.

<u>Lemma 4.2</u>: $COST(W', C, D) \geq 3/2 * k^2 n(n-1)$

<u>Proof</u>: The Lemma is clearly true if W' is proper, and further this is the minimum value of COST(W',C,D). If we don't have a proper marking, then at most (n(n-1)/2 - 1) pairs of non-complementary rows are blocked. However, we can block at most 2n(n-1)kmn arcs by marking mn D-nodes. Thus, for any non-blocking marking,

$$COST(W', C, D) \geq (2n(n-1) - n(n-1)/2 + 1)k^2 - 2n(n-1)kmn$$

$$= (2n(n-1) - n(n-1)/2)k^2 + k^2 - 2kmn^2(n-1).$$

Since, $k^2 > 2kmn^2(n-1)$, the cost of a blocking marking is minimum. []

<u>Lemma 4.3</u>: If W' is proper and blocking, $COST(W') \leq F$.

<u>Proof</u>: We will calculate an upper bound on the cost of a marking W', by examining the components of the cost for each sections of the graph: A-nodes, B-nodes, C-nodes, D-nodes, and the nodes of the matrix which we will call Mx-nodes. Then for some of these component costs, we will take the upper bound to be the number of pairs of connected nodes assuming no marking at all. Note COST(W',A,A) = 0, COST(W',B,B) = 0, etc. Thus,

$$\begin{aligned}\mathrm{COST}(W') = {} & \mathrm{COST}(W',A,B) + \mathrm{COST}(W',A,Mx) + \mathrm{COST}(W',A,D) + \\ & \mathrm{COST}(W',C,Mx) + \mathrm{COST}(W',C,D) + \mathrm{COST}(W',C,B) + \\ & \mathrm{COST}(W',Mx,Mx) + \mathrm{COST}(W',Mx,B) + \mathrm{COST}(W',Mx,D)\end{aligned}$$

Looking at these costs in detail, we have:

$\mathrm{COST}(W',A,Mx) \le kmn$. In a given column, an A-node can only connect to n nodes, since there are n literals per clause.

$\mathrm{COST}(W',A,D) \le k^2$. The k A-nodes and the k D-nodes are all cross-connected.

$\mathrm{COST}(W',C,Mx) \le 4kmn(n-1)$. Each of the $2kn(n-1)$ C-nodes connects to 2 rows, or 2m nodes.

$\mathrm{COST}(W',C,B) \le 2k^2 * n(n-1)$. Each of the $2kn(n-1)$ C-nodes connects to each of the k B-nodes.

$\mathrm{COST}(W',Mx,Mx) \le mn(n-1)/2$. In each of the m columns, the nodes of the clause are connected together and there are at most n variables in the clause.

$\mathrm{COST}(W',Mx,B) \le k^2$. Same as $\mathrm{COST}(W',A,Mx)$.

$\mathrm{COST}(W',Mx,D) \le 2kmn$. Each of the nodes of the matrix is connected to each of the D-nodes.

Thus,

$$\begin{aligned}\mathrm{COST}(W') \le {} & \mathrm{COST}(W',A,B) + kmn + k^2 + 4kmn(n-1) + \mathrm{COST}(W',C,D) \\ & 2k^2 * mn(n-1) + mn(n-1)/2 + kmn + 2kmn \\ = {} & k^2 + 4kmn^2 + 2k^2 * n(n-1) + mn(n-1)/2 \\ & + \mathrm{COST}(W',A,B) + \mathrm{COST}(W',C,D)\end{aligned}$$

If W' is proper and blocking, then

$\mathrm{COST}(W',A,B) = 0$

$\mathrm{COST}(W',C,D) = (2n(n-1) - n(n-1)/2)k^2 = 3/2 * k^2 * n(n-1)$.

The latter quantity comes from the fact that we block $n(n-1)/2$ pairs of non-complementary rows.

The total cost for a proper blocking marking is

$$\begin{aligned}\mathrm{COST}(W') &= k^2 + 4kmn^2 + 7/2 * k^2 * n(n-1) + mn(n-1)/2 \\ &\le 2k^2 - 1 - kmn - 4kmn\,(n-1) + 7/2 * k^2 * n(n-1) = F. \;[]\end{aligned}$$

Lemma 4.4: If W' is not blocking then COST(W') > F.

Proof: If the marking, W', is not blocking, then

$$\mathrm{COST}(W',A,B) \geq k*(k - mn),$$

(since there is a path from the A-nodes to the B-nodes, but we can block mnk^2 paths by marking mn B-nodes.) Hence

$$\begin{aligned}
\mathrm{COST}(W') &\geq \mathrm{COST}(W',A,B) + \mathrm{COST}(W',C,D) + \mathrm{COST}(W',A,D) + \mathrm{COST}(W',C,B) \\
&\geq k^2 - kmn + 3n(n-1)k^2/2 + (k - mn)k + 2n(n-1)k*(k-mn) \\
&= k^2 - 2kmn + 3/2*k^2*n(n-1) + k^2 + 2k^2*n(n-1) - 2kmn^2*(n-1) \\
&= 2k^2 - 2kmn + 7/2*k^2*n(n-1) - 2kmn^2*(n-1) \\
&\geq 2k^2 - kmn + 7/2*k^2*n(n-1) - 4kmn^2(n-1) > F.
\end{aligned}$$

The lower bounds on COST(W',A,D) and COST(W',C,B) are achieved by using the mn nodes to mark nodes in the smaller of the two sets. [ ]

Lemma 4.5: If W' is not proper then COST(W') > F.

Proof: If W' partially blocks one row, or W' blocks two rows corresponding to a variable and its complement, then at most n(n-1)/2 + 1 pairs of non-complementary rows can be blocked, so

$$\mathrm{COST}(W',C,D) \geq 3/2*k^2*n(n-1) + k^2 - 2kmn^2(n-1).$$

Thus,

$$\begin{aligned}
\mathrm{COST}(W') &\geq \mathrm{COST}(W',C,D) + \mathrm{COST}(W',A,D) + \mathrm{COST}(W',C,B) \\
&= 3/2*k^2*n(n-1) + k^2 - 2kmn^2(n-1) + k(k-mn) + 2n(n-1)k*(k-mn) \\
&> F. \qquad [\ ]
\end{aligned}$$

By all of the preceding, we have:

Theorem 4.1: Problem 4 is NP-complete.

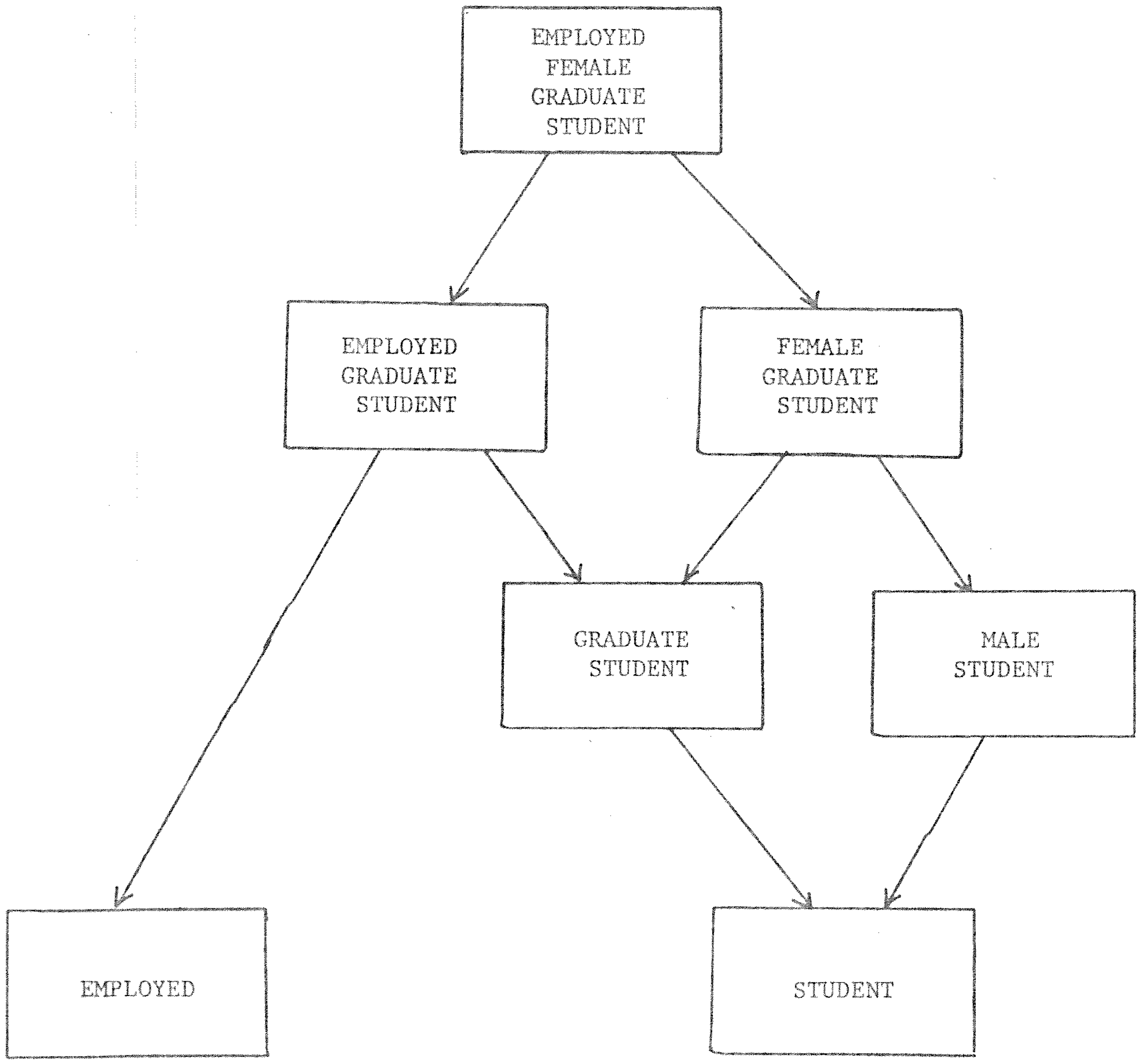

Figure 1.1

A DAG describing the relationship between different views of a database.

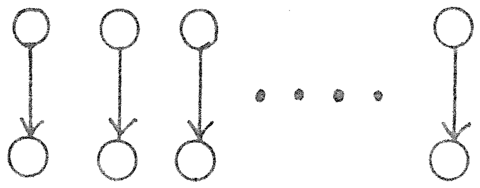

FIGURE 2.1

An especially simple class of graphs
for which Problem 2 is NP-complete

```
For all x in V, traversing G in postorder, do begin
    Let the sons of x be $y_1,\ldots,y_K$

    For m := 0 to M do
        if m < s(x) then minclosed[m] : =
                else minclosed[m] := $\displaystyle\min_{m_1+\ldots+m_k=m-s(x)} \sum_{i=1}^{k} \text{mincost}[y_i, 0, m_i]$;

    anc := x;
    tailprob := p(x);

    For tlen := 0 to depth(x) do begin
        For tlen := 0 to M do begin
            minopen[m] := c(x)*tailprob + $\displaystyle\min_{m_1+\ldots+m_k=m} \sum_{i=1}^{k} \text{mincost}[y_i, \text{tlen}+1, m_i]$;
            mincost[x,tlen,m] := min(minclosed[m], minopen[m])
        end

        anc <> root then begin
            anc := father(anc);
            tailprob := tailprob + p(anc)
        end
    end
```

Figure 3.1

A high-level version of the algorithm.

(We assume all leaves have been deleted and their total size subtracted from M.)

```
For all x in V, traversing G in postorder, do begin
    Let the sons of x be $y_1,\ldots,y_k$

    For m := 0 to M do
        if m < s(x) then minclosed[m] :=
                    else minclosed[m] := 0;

    For i := 1 to k do
        minclosed := convolute(minclosed,mincost[$y_i$,0]);


    anc := x;
    tailprob := p(x);

    For tlen := 0 to depth(x) do begin
        for m := 0 to M do
            minopen[m] := c(x) * tailprob;

        for i := 1 to k do
            convolute(minopen,mincost[$y_i$,tlen+1]);

        for m := 0 to M do
            mincost[x,tlen,m] := min(minclosed[m],minopen[m]);

        if anc <> root then begin
            anc := father(anc);
            tailprob := tailprob + p(anc)
        end
    end
```

Figure 3.2

A PASCAL implementation of the algorithm.

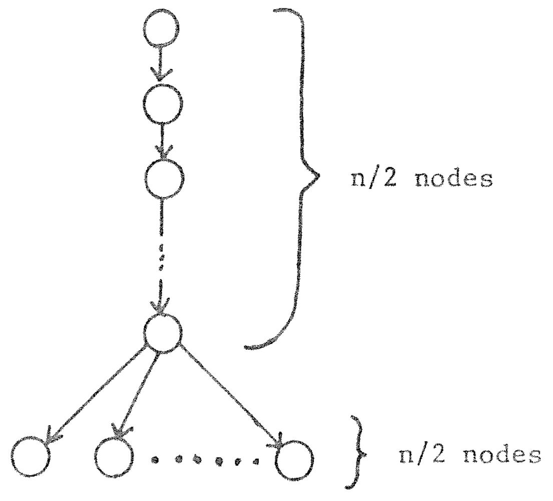

Figure 3.3

A worst-case tree.

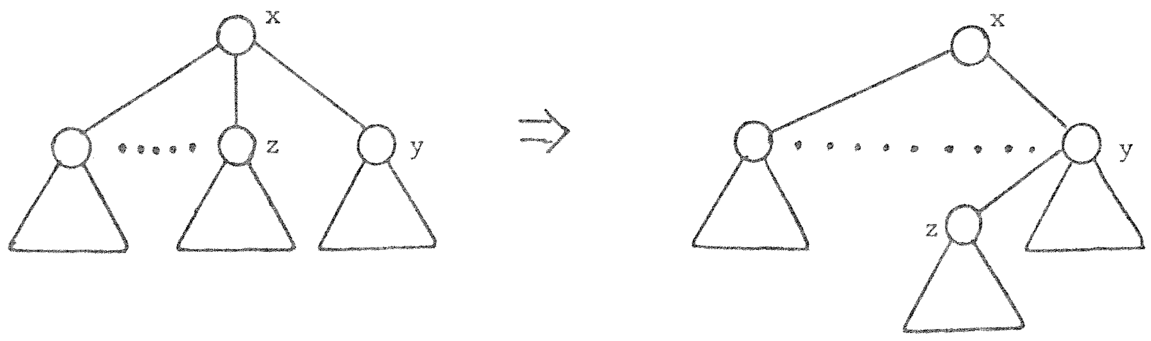

Figure 3.4

A transformation that increases f.

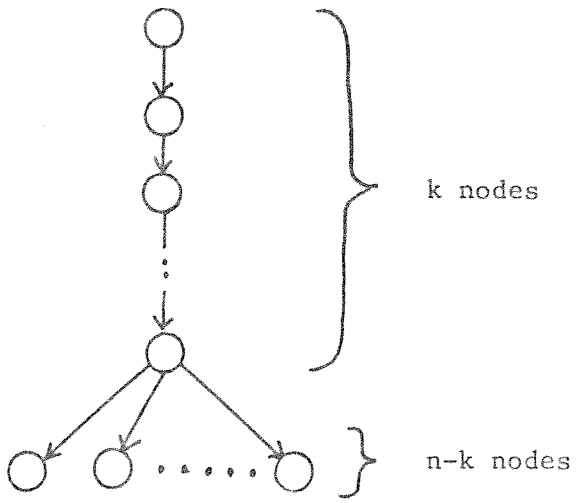

Figure 3.5

The form of a tree whose cost cannot be increased by the transformation.

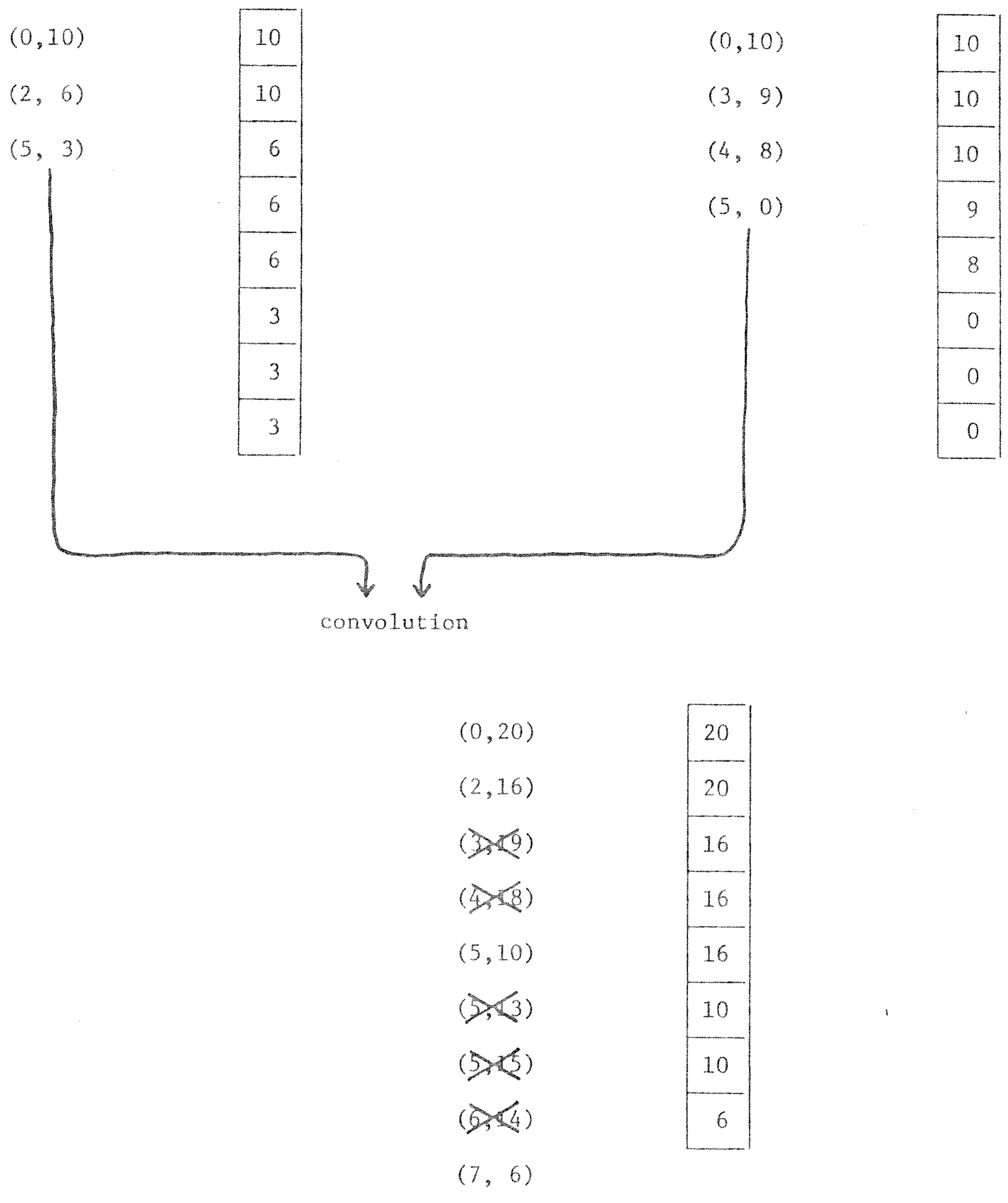

FIGURE 3.6

Forming a convolution using lists.

The convolution process is illustrated on two lists with M = 7. The pairs in the result which are crossed out are generated and then eliminated. For reference the vector corresponding to each list is shown.

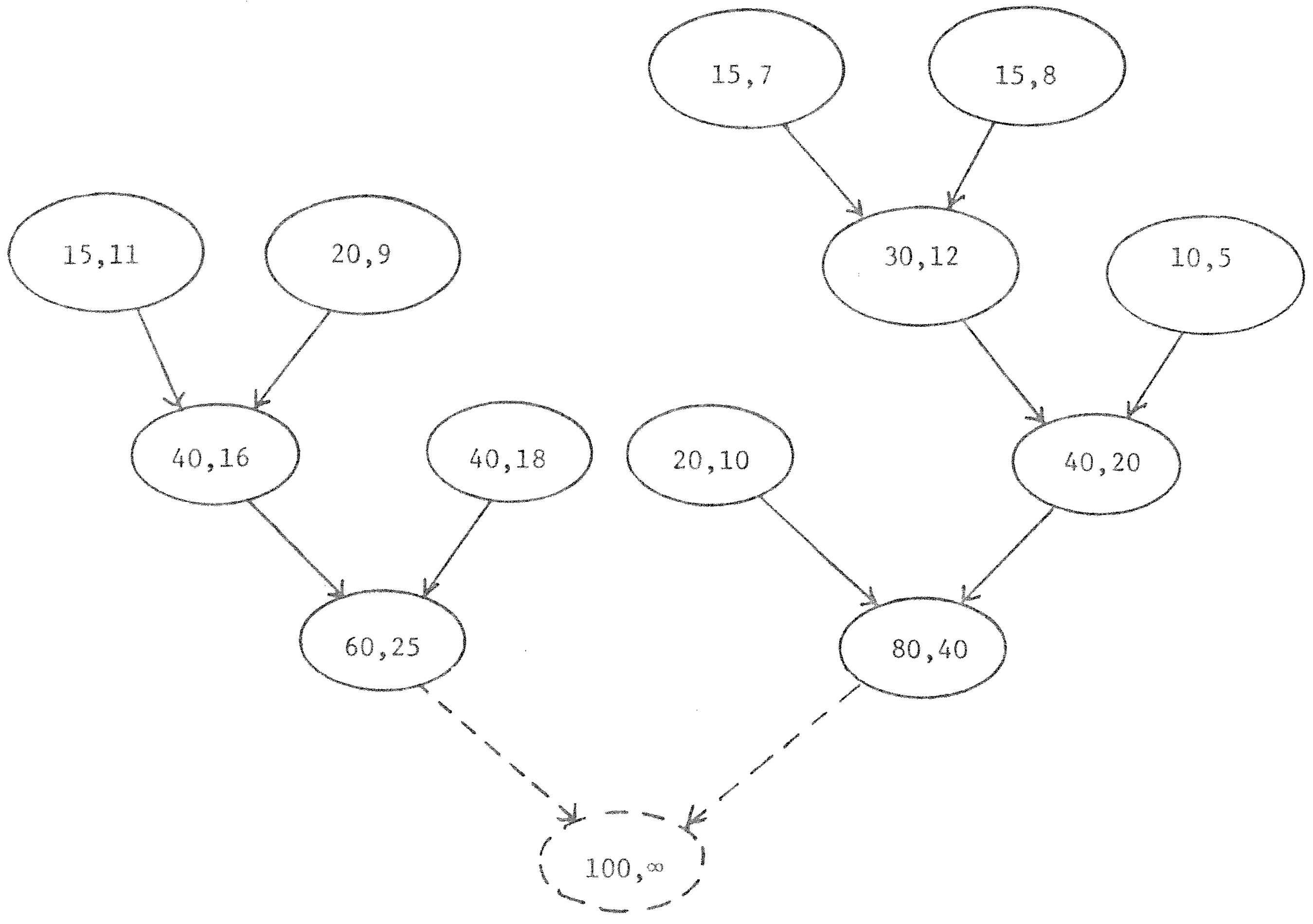

Figure 3.7

An input DAG to the algorithm.

The first number within each node gives that view's size. The second gives its cost. The dotted node is a leaf and has been deleted. All nodes have equal probability.

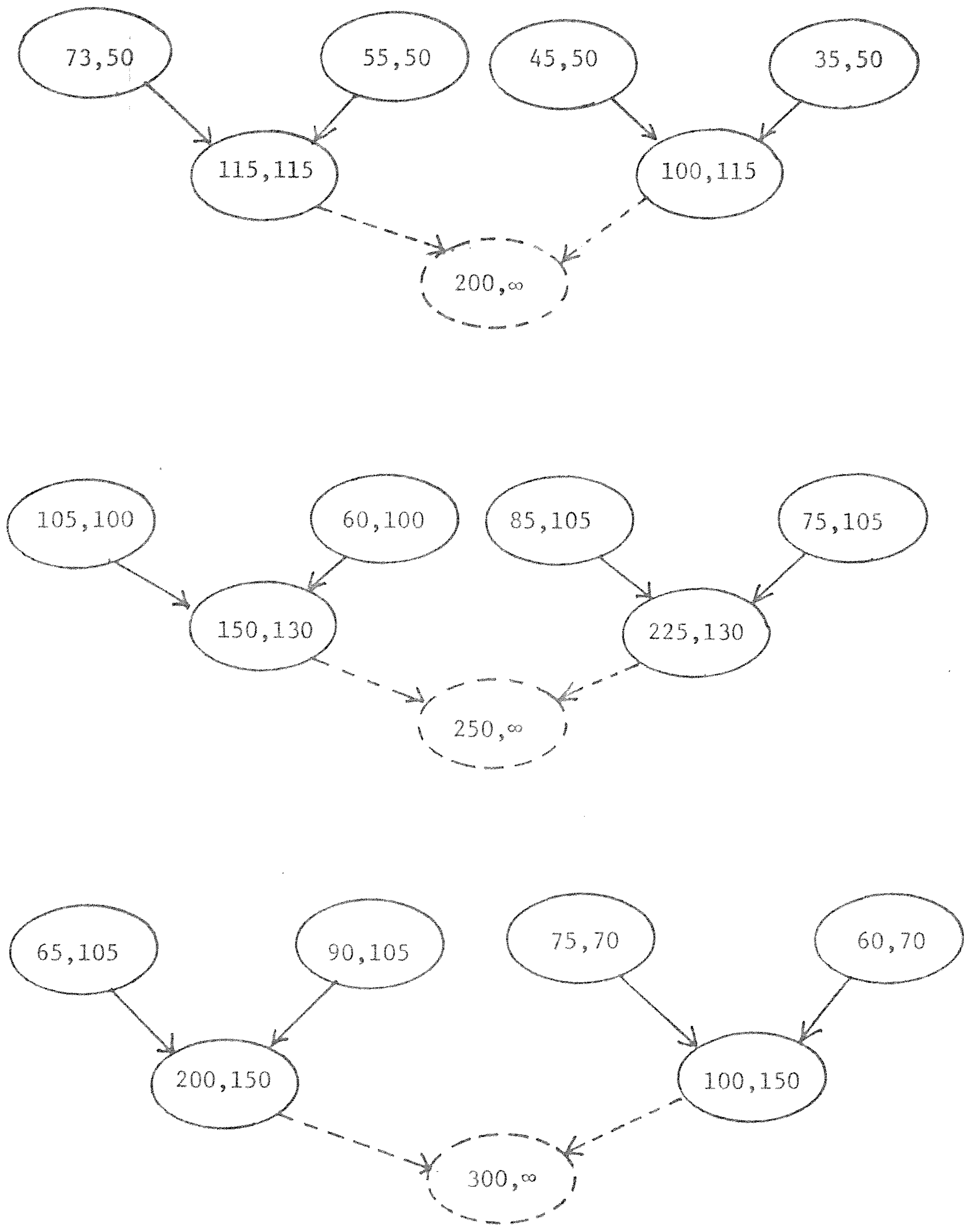
73,50
55,50
45,50
35,50
115,115
100,115
200,∞
105,100
60,100
85,105
75,105
150,130
225,130
250,∞
65,105
90,105
75,70
60,70
200,150
100,150
300,∞

Figure 3.8

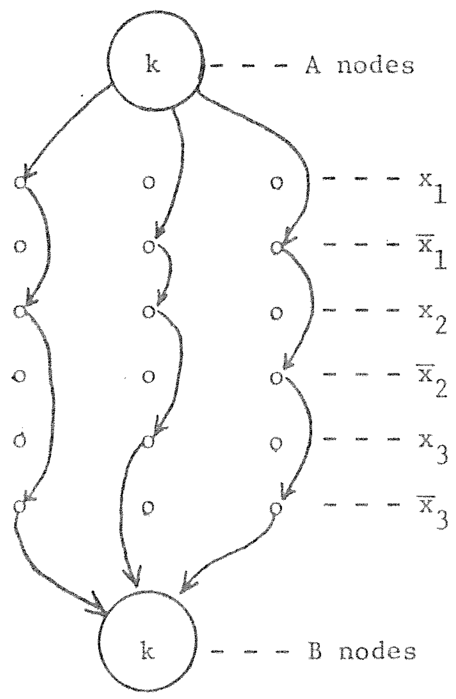

Figure 4.1

The connections between the A-node, B-nodes and matrix for the expression $(x_1 + x_2 + \overline{x}_3)(\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$. Each small circle corresponds to one vertex in the matrix. Each large circle corresponds to k vertices.

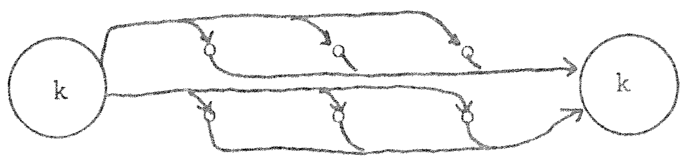

Figure 4.2

The connections between one group of the C-nodes and the D-nodes. Two rows of the matrix are shown. They do not correspond to a variable and its complement.

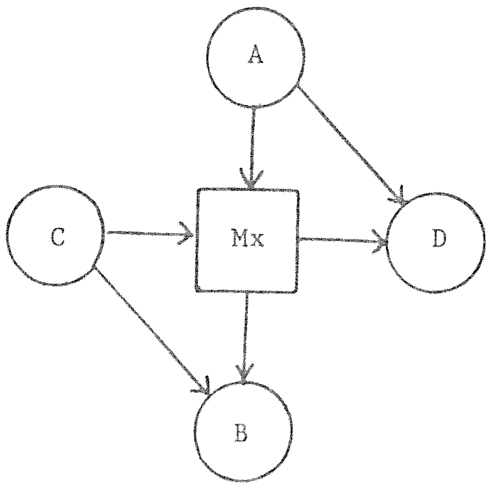

Figure 4.3

A schematic view of the entire graph.

REFERENCES

[AND 77] Anderson, H. D., and Berra, P. B., "Minimum Cost Selection of Secondary Indexes," ACM Transactions on Database Systems, vol. 2, No. 1, March 1977, pp. 68-90.

[AST 76] Astrahan, M. M., et. al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, vol. 1, No. 2, June 1976, pp. 97-137.

[CAR 75] Cardenas, A. F., "Analysis and Performance of Inverted Database Structures," Comm. of ACM, vol. 18, No. 5, May 1975, pp. 253-263.

[CHA 77] Chandy, K. M., "Models of Distributed Systems," Proc. of Third VLDB, Tokyo, Japan, 1977, pp. 105-120.

[CHA 78] Chang, S. K., and Cheng, W. H., "Database Skeleton and Its Application to Logical Database Systhesi," IEEE Transactions on Software Engineering, vol. SE-U, No. 1, January 1978, pp 18-30.

[COO 71] Cook, S. A., "The Complexity of Theorem Proving Procedures," <u>Proc. 3rd Annual ACM Conference on Theory of Computing</u>, 1971, 151-158.

[FAR 75] Farley, J. H. G. and Schuster, S. A., "Query Execution and Index Selction for Relational Data Bases," TR CSRG-553, University of Toronto, March 1975.

[KAR 72] Karp, R. M., "Reducibility Among Combinatorial Problems," in <u>Complexity of Computer Computations</u>, Eds. Miller, R. E., and Thatcher, J. W., Plenum Press 1972, pp. 85-104.

[KIN 74] King, W. F., "On the Selection of Indices for a File," IBM Research Report RJ1341, San Jose, CA, 1974.

[MAR 78] March, S. T., and Severance, D. G., "A Mathematical Modelling Approach to the Automatic Selection of Database Design," Proc. of SIGMOD 1978, Austin, TX.

[MYL 75] Mylopoulos, J., Schuster, and Tsichritizis, D., "A Multi-Level Relational System," Proceedings of NCC, 1975.

[NIL 71] Nilsson, N. J., <u>Problem-Solving Methods in Artificial Intelligence</u>, McGraw-Hill, 1971.

[ROU 79] Roussopoulos, N., and Davis, L., "View Indexing in Relational Data Bases," TR-101, University of Texas, Austin, Texas, 1979.

[SAH 75] Sahni, S., "Approximate Algorithms for the 0/1 Knapsack Problem," Journal of ACM, vol. 22, No. 1, January 1975, pp. 115-124.

[SCH 75] Schkolnick, M., "Secondary Index Optimization," Proc. ACM SIGMOD, 1975, pp. 186-192.

[SCH 78] Schkolnick, M., "Physical Database Design Techniques," NYU Symposium on Database Design, New York 1978.

[STO 76] Stonebraker, M. R., Wong, E., Kreps, P., and Held, G., "The Design and Implementatin of INGRESS," ACM Transactions on Database Systems, vol. 1, No. 3, September 1976, pp. 189-222.