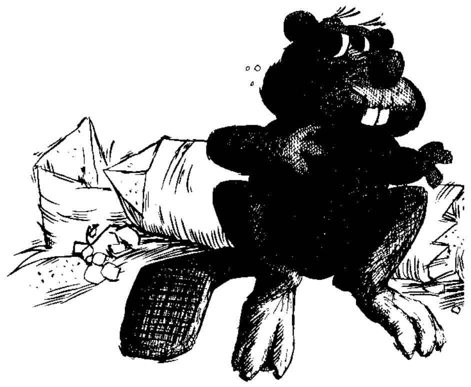

An Aid for the Selection of Efficient Storage Structures

Frank Wm. Tompa
Raúl J. Ramirez

CS-80-46
(replaces CS-79-40)

October, 1980

AN AID FOR THE SELECTION OF EFFICIENT STORAGE STRUCTURES

Frank Wm. Tompa
Raúl J. Ramirez †

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

ABSTRACT

The representations used to implement data structures play a large part in determining the execution cost for most applications. Because suitable representations may be chosen from a very large class, it is important to search systematically for the efficient ones.

In this paper, algorithms based on dynamic programming are presented. It is assumed that an application's behaviour is specified by means of evaluation maps which reflect the expected run time and storage space required by each component of the application's data structure. Those maps must be searched to find representations for each component which, when composed into a single storage structure, minimize the cost for the application according to a given cost formula. The algorithms incorporate bounds on the maximum allowable run time and storage space and solve the selection problem in pseudo-polynomial time and space.

Key phrases: Data structures design, space/time efficiency, storage structures, library of implementations, evaluation maps, dynamic programming.

CR *Categories:* 3.73, 4.33, 4.34, 5.25, 5.42

October 20, 1980

† Current address: Grupo Industrial Saltillo, Saltillo, Mexico.

AN AID FOR THE SELECTION OF EFFICIENT STORAGE STRUCTURES

Frank Wm. Tompa
Raul J. Ramirez †

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1
Canada

1. INTRODUCTION

Data structure design involves several levels of data abstraction [Tompa77]. At one of the levels, a data structure's schema is defined in terms of a composition of data type occurrences (e.g. occurrences of sets, dictionaries, priority queues, trees, sequences or tuples) and in terms of the valid operations over each data type. Indeed, programming languages such as Alphard [Shaw77] CLU [Liskov77], SETL [Dewar79] and Mesa [Geschke77] provide ideal frameworks for expressing such abstractions. This level of data abstraction has been termed the *abstract structure* level.

Through data type encapsulation, the only possible interaction with a data type is by invoking the given set of operations. As a result, "representation independence" is achievable; that is, the manipulation of a data type occurrence need not (in fact, cannot) rely on any particular implementation for the type. This independence gives an implementer the freedom to select or to alter the representation for the type so as to improve some desired measure of performance (e.g. efficiency, reliability, maintainability, or portability) without affecting the application's uses of the type. The choice of representations is the bridge from the abstract to the *storage structure* level.

This paper is addressed at the problem of designing efficient storage structures. Because the number of possible data representations is extremely large and the space of possibilities is apparently not well-structured, choosing on optimal storage structure appears to be unachievable. Even the design of reasonably efficient storage structures is sufficiently complex to benefit from the use of automated tools. It is such a tool that is described in this paper.

† Current address: Grupo Industrial Saltillo, Saltillo, Mexico.

2. A SIMPLIFIED DESIGN PROBLEM

In order to make the problem of storage structure design amenable to automated aids, several assumptions will be made. First, it is assumed that the class of applications to use the data structure are known in advance and well-understood. For example, the relative frequencies of queries and updates, the specific types of queries and their probabilities of occurrence, and the cardinalities of all structures and query responses are used as input to the design. Such information can be gathered from several sources, including interviews with users, simulation studies, monitoring of existing software, and designers' hypotheses; the less reliable the input, however, the less faith can be placed on the automated storage structure selection.

A second assumption is that the storage structure will be built as a composition of representations for basic data types, those representations being selected from a finite library. The first part is merely a formulation of well-structuring and modularity. The use of a library of representations is a restriction on storage stucture choice, but it is realistic for applications that are written in high-level code in terms of a fixed set of data types and automatically compiled in one of several possible ways. Even for hand-coded algorithms, the use of formal or informal libraries of code [Gonnet80] is likely to become more and more desirable in order to save programming and maintenance effort and expense.

Given an application and a library of data types and their representations, the problem of data structure design falls into two phases: the design of the abstract structure (i.e., determining which types to use explicitly for which relationships) and the design of the storage structure (i.e., determining which representations to use, once the types are fixed). Several researchers have focussed their attention on the first question ([Hubbard75], [March78],[Santoro80]). However, in this paper, it will be assumed that such a design has been completed and that the abstract structure is therefore fixed. Thus, the usage requirements imposed by the applications will be assumed to have been translated into parametric values for each of the data type occurrences.

As an example, consider a Huffman code generator (Program 1) and assume it is complete application. The data types for which representations are to be chosen are *set* (to store in the priority queue the set of characters having a combined probability), *priority queue* (for the variable QUEUE controlling the algorithm), and *dictionary* (for the variable DICT to maintain the codes); the other types (e.g. bitstring, char) will be assumed to have only one choice of representation. The algorithm can be analyzed to determine the number of times each operator is involved for each data type (Table 1); for example, if 128 symbols are to be encoded and $C = S \log_2 S = 896$, then there will be no set creations, 127 (disjoint) set unions, 896 set "get-next-members", etc.

2.1. The library of Implementations

The creation of a repertory of implementations from which the selections are made has been addressed by Tompa [Tompa74] and by Low [Low76]. In database systems, such a given set of possible implementations is commonplace (see, for example, [CODASYL71]). A *library of implementations* contains a set

```
1    type    input-element = record  symb: char;
                                     probability: 0..100  end;
2            queue-element = record  prob: 0..100;
                                     members: set of char  end;
3             dict-element = record  symbol: char;
                                     code:  bit-string  end;
4    var  INPUT:  sequence of input-element;
5        QUEUE:  queue of  queue-element priority by prob;
6         DICT:  dictionary of dict-element key symbol;
7        M1,M2:  queue-element;
     begin
8        DICT := dictionary$create;
9        QUEUE := queue$create;
10       for each E in INPUT do
             begin
11           dictionary$insert(DICT, E.symb, " ");
12           queue$insert(QUEUE, E.probability, set$insert(set$create,E.symb))
             end;
13       M1 := queue$min(QUEUE);
14       while not queue$empty(QUEUE) do
             begin
15           for each CHAR in M1.members do
16               dictionary$update(DICT, CHAR, "0" || code);
17           M2 := queue$peek(QUEUE);
18           for each CHAR in M2.members do
19               dictionary$update(DICT, CHAR, "1" || code);
20           M2.prob := M1.prob + M2.prob;
21           M2.members := M1.members ∪ M2.members;
22           queue$replace(QUEUE, M2);
23           M1 := queue$min(QUEUE)
             end;
24       for each D in DICT do
25           write(D.symbol, D.code)
     end
```

Program 1: Pascal-type code for Huffman code generation

Let S be the number of distinct symbols for which codes are to be found
C be the total bit length over all codes generated.

	Operation Counts	Line Number(s)
sets		
create:	S	12
insert:	S	12
union:	S-1	21
get-next:	C	15,18
QUEUE		
create:	1	9
insert:	S	12
min:	S	13,23
empty:	S	14
peek:	S-1	17
replace:	S-1	22
DICT		
create:	1	8
insert:	S	11
update:	C	16,19
traverse:	1	24

Table 1: Operation counts for Huffman code generation

of possible representations for each member in a standard set of data types available at the abstract structure level. Each member is a cluster of code that implements the valid operations for a particular representation of the type. For example, a set of valid operations for the dictionary data type might be to create an empty dictionary, to locate an element of the dictionary, to read or to write its contents, and to destroy the dictionary. These operations can be implemented for different representations, e.g., a dictionary can be represented as a contiguous store, linearly addressed store, unary chain, bit map, binary tree structure, etc. [Gotlieb74]. Some of the implementations in the library may be better suited than others for a particular application. For example, if, relative to other operations, a large number of insertions are to be performed, the linearly addressed store is a good choice; however if the dictionary is sparse (with respect to all possible keys) and storage space is at a premium, a unary chain may be better.

As mentioned before, the reliance on a library of implementations restricts the solution space to that implicitly described by the library. It is felt that the restriction need not be severe if the library is allowed to be large. This, in turn, requires that its alternatives can be quickly appraised.

In order to evaluate the appropriateness of the various representations objectively, it is necessary to characterize each member of the library according to some measure. The measure selected for most studies is that of efficiency in terms of the expected run time for the required operations and the expected number of storage cells consumed by the data.

Program 2 depicts a typical member of the library. Together with the code are two parametric formulas representing the time required to execute the code and the space required to maintain the structures. (The example is indicative of the form of the library and is not intended to reflect the code's actual behaviour on a specific machine.)

There exist several techniques for parameterizing the expected run time and storage space of a program: counting techniques [Cohen74, Tompa74], complexity analyses of the algorithms involved [Aho74, Knuth73], as well as the monitoring of the program execution [Wichman72, Low78]. For this paper, it will be assumed that an appropriate library has been constructed by one or more of these techniques. The remainder of this paper will concentrate on the selection of efficient representations based on such a library. It should be noted, however, that performance improvements may be possible when certain combinations of structures are used; incorporating such improvements into the methodology are beyond the scope of this paper (see, for example [Rowe76]).

2.2. The evaluation map

Given an application's usage behaviour in terms of a collection of data type occurrences and a library of implementations for data types, the impact of a storage structure choice on the efficiency of the application can be determined. Because the number of occurrences of data types in an application is typically very large, it is important to aggregate them into *substructures*, homogeneous collections of data type occurrences defined at the abstract structure level. For example, although in principle each row of a matrix could be represented by a different implementation, it is convenient to treat them all homogeneously, that is, Unordered contiguous representation for a set:

```
type set = record last: 0..MAX;
                  elements: array [1..MAX] of ANY end;
function CREATE : set;
        begin
        CREATE.last := 0
        end;
procedure INSERT (S: set; D: ANY)
        begin
        S.last := S.last + 1;
        S.elements[S.last] := D
        end;
function UNION (S1,S2: set) : set ;
        /* find disjoint union */
        var T: set ;
            J: 0..MAX;
        begin
        for J:=1 to S1.last do
                T.elements[J] := S1.elements[J];
        for J := 1 to S2.last do
                T.elements[S1.last+J] := S2.elements[J];
        T.last := S1.last + S2.last;
        UNION := T
        end;
function GET-NEXT (S1: set; current: 0..MAX) : 0..MAX;
        begin
        if current ≥ S.last
                then GET-NEXT := 0
                else GET-NEXT := current+1
        end
```

a) code

$$(6)*CREATE + (10)*INSERT + (18+15*SIZE_1+18*SIZE_2)*UNION$$
$$+ (14-3*LAST)*GET\text{-}NEXT$$

b) *time* in terms of parameter for frequency of operations, set sizes, and whether or not seeking legitimate next

$$1 + MAX*ELEMENT-SIZE$$

c) *space* in terms of maximum size of set and size of components

Program 2: Example of a library entry

as one substructure [Tompa76]. Furthermore, in order to avoid excessive conversions between representations for closely interacting structures (e.g. operands for a common operation, two sides of an assignment statement, or arguments to a common subroutine), it is often convenient to coalesce such data type occurrences into substructures as well [Low78, Dewar79].

The problem of building an evaluation map from a library and usage statistics is not trivial, often requiring additional analysis and insight on the part of the designer. For example, to evaluate the expected accumulated run time for the substructure consisting of all sets of characters in Huffman code generation, it must be realized that the C get-next operations will correspond to $2*(S-1)$ set traversals (thus $LAST=2*(S-1)/C$) and that the total of all set sizes involved in unions will be C (thus $SIZE_1 = SIZE_2 = C/2/UNION$). The use of unordered contiguous representations for the sets will therefore require $time=19S+44C+12$ and $space=S+1$. Since it cannot be expected that such analysis can be automated in the near future, human intervention is again required. An example of an evaluation map for Huffman code generation with $S=128$ and $C=896$ is shown in Table 2.

In previous research, Gotlieb and Tompa first coded the application program by means of the valid operations defined over the library's data types, and next counted the relative frequency of each operation, so that the parametric formulas can be assigned values that reflect the application's characteristics as well as the computing environment [Gotlieb74]. In related work, Low characterized each implementation by statistical information provided by the user or collected by monitoring the execution of the program when using default representations [Low78].

3. STORAGE STRUCTURE SELECTION FROM EVALUATION MAPS

The goal of efficient storage structure design is to find data representations that together result in the least cost according to a given cost formula (e.g., $space*time$, $space+time^2$, or $time*\log(time)+time*space^2/5$). In the context of the simplified design problem introduced in Section 2, this corresponds to choosing for each substructure, that implementation from the library that minimizes the total cost (over all such choices).

If an application involves N substructures and the number of possible implementations for substructure i is M_i, an exhaustive search of the evaluation map would require the computation of $\prod_{i=1}^{N} M_i$ alternatives, which is usually prohibitively high. Elsewhere it has been shown that one cannot circumvent this by merely choosing the least-cost implementation for each substructure, unless the cost formula is *separable*, that is, unless the total cost is proportional to the sum of the costs of the substructures [Tompa76]. For example, the formulas $space*time$, are $time+time*space^2$ and not separable, whereas $space+time$ and $time*\log(time)+space^2$ are. The remainder of this paper will address the problem of processing an evaluation map.

Substructure	Implementation	Time	Space
sets	bit map	528972	2048
	contiguous store	22213	16512
QUEUE	unary chain	161534	640
	contiguous store	219495	132
	heap	45951	132
DICT	binary ring	650321	2060
	unary chain	965844	1544
	binary tree	242064	2060
	avl tree	168452	2060
	contiguous store	286934	1028
	hash table	116214	1280

Table 2: Evaluation map for Huffman code generation

3.1. Formal background

Previous research dealt with the selection of unchanging storage structures (e.g., [Tompa76, Berelian77,Low78, Rowe78]). It was implicitly assumed that the relative frequency of operations over the data types remained constant or that the average frequency of operations over the lifetime was sufficient to characterize the application. Thus once a selection of implementations for the substructures was made, say at the beginning of the application life, it remained for the complete lifetime. However, there exist applications in which the frequency of operations changes as time passes, making some other implementations more attractive than the ones chosen at the start. For these cases it is said that the application passes through phases, each phase having different requirements.†

In general, converting from one implementation that is optimal for one phase to the implementation that is optimal for the next phase might not be overall optimal. It might be possible to make a selection that is not optimal for the first phase, and another that is also not optimal for the second phase, but when composed, cost less than the phase-optimal selections (see example below). Similarly, if a third phase has different requirements it might be more efficient to convert directly from the structure most suited for the first phase to the one most suited for the third, at the time that the application is only beginning the second phase.

Consider the simplified application represented in Table 3. The best selection for phase 1 alone is implementation 1, and the best selection for phase 2 alone is implementation 2. The combined cost of both selections, including the conversion cost of 100, is 120 cost units; that is, applying the cost formule to the expected run time and storage space results in a value of 120.

When the two phases are considered simultaneously, an algorithm solving this problem should select implementation 3 for both phases, with a combined cost of 30 units. An algorithm solving this type of problems must be supplied with the information regarding each phase before any selection can be made. Such an algorithm will in general, produce a *sequence of storage structures* that together minimize cost.

The selection of a sequence of storage structures can be described as an integer programming problems as follows:

given:

P	the number of phases for which a selection is sought, i.e., the application lifetime,
N	the number of substructures for which an assignment is sought,
M_i	the number of implementations in the library for substructure i,

† There exist studies that deal with the detection of phase changes for an application (see, for example, [Winslow75]).

		Cost of Phase			Conversion Cost		
		1	2		1	2	3
	1	10	20	1	0	100	25
Implementations	2	20	10	2	100	0	25
	3	25	25	3	25	25	0

Table 3: Simple application with phases

X^p	a (ragged) zero-one matrix in which $x^p_{i,j}$ indicates whether or not implementation j is to be selected for substructure i in phase p,
$s^p_{i,j}$	the estimated storage space consumed by implementation j when used for substructure i in phase p,
$t^p_{i,j}$	the estimated run time of implementation j when used for substructure i in phase p,
$c^p_{i,j,j'}$	the cost of converting substructure i from implementation j in phase p to implementation j' in phase $p+1$,
S^p, T^p	the maximum amount of storage space and running time respectively available to be used by the selected implementations in phase p,
$\$(X^p, S^p, T^p)$	a monotonic cost function in terms of the total amount of time consumed by the final selection X^p when constrained to the bounds S^p and T^p in phase p.

find:

$$Z = \min \sum_{p=1}^{P} \left\{ \$(X^p, S^p, T^p) + K * \left[\sum_{i=1}^{N} \sum_{j=1}^{M_i} \sum_{j'=1}^{M_i} c^p_{i,j,j'} * x^p_{i,j} * x^{p+1}_{i,j'} \right] \right\} \quad (1)$$

such that:

$$\sum_{j=1}^{M_i} x^p_{i,j} = 1 \qquad \text{for all } i = 1..N,\ p = 1...P \qquad (2)$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M_i} s^p_{i,j} * x^p_{i,j} \leqslant S_p \qquad \text{for all } p = 1...P \qquad (3)$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M_i} t^p_{i,j} * x^p_{i,j} \leqslant T_p \qquad \text{for all } p = 1...P \qquad (4)$$

$$x^p_{i,j} = 0, 1 \qquad \text{for all } i = 1..N,\ j = 1..M_i,\ p = 1....P \qquad (5)$$

$$K \in \mathbf{R}^+ \qquad (6)$$

The last expression of Equation (1) accounts for the conversion cost between phases, since the conversion cost $c^p_{i,j,j'}$ applies only when both zero-one variables are one. Equations (2) and (5) guarantee the selection of only one implementation per substructure per phase. Inequations (3) and (4) restrict the solution to fit within bounded space and time at each phase, and Expression (6) allows conversion costs to be weighted more or less heavily as desired.

THEOREM: The selection of a sequence of storage structures problem belongs to the strong-*NP*-complete class of problems.

PROOF (M. Tompa): In order to prove this theorem, it will be shown that it is possible to transform the well-known *NP*-complete problem concerning the satisfiability of Boolean expressions (SAT) [Karp75] to the selection of a sequence of storage structures problem:

The satisfiability problem can be stated as follows:

> Given a set U of Boolean variables and a collection C of clauses in conjunctive normal form, is there a satisfying truth assignment for C?

Given an instance of the satisfiability problem, transform it into a selection problem in which for all $i = 1 ... |U|$ and $k = 1 ... |C|$

$$s_{i,1}^{k} = \begin{cases} 0 & \text{if variable } U_i \text{ is in clause } C_k \\ 1 & \text{if } U_i \text{ is not in } C_k \end{cases}$$

$$s_{i,2}^{k} = \begin{cases} 0 & \text{if the negation of } U_i\ (\overline{U_i}) \text{ is in } C_k \\ 1 & \text{if } \overline{U_i} \text{ is not in } C_k \end{cases}$$

$$t_{i,j}^{k} = 0$$

$$c_{i,j,j'}^{k} = \begin{cases} 0 & \text{if } j = j' \\ 1 & \text{otherwise} \end{cases}$$

$$T^k = 0, \quad S^k = |U| - 1$$

For $N = |U|$ and $P = |C|$, solve the following selection problem:

$$Z = \min \sum_{k=1}^{P} \left\{ \sum_{i=1}^{N} \sum_{j=1}^{2} t_{i,j}^{k} * x_{i,j}^{k} + \sum_{i=1}^{N} \sum_{j=1}^{2} \sum_{j'=1}^{2} c_{i,j,j'}^{k} * x_{i,j}^{k} * x_{i,j'}^{k+1} \right\}$$

such that

$$\sum_{i=1}^{N} \sum_{j=1}^{2} s_{i,j}^{k} * x_{k(i,j)} \leqslant N-1 \qquad \text{for all } k = 1...P$$

The interpretation of this restriction when applied to the original satisfiability problem is to allow at most N-1 variables in each clause to be false, i.e., at least one variable in each clause to be true; consequently, the whole expression must be satisfied.

If the selection problem just described is solvable and if its solution has a cost of zero, then the original problem is satisfiable by setting

$$U_i = \begin{cases} \text{true} & \text{if } x_{i,1}^{k} = 1 \\ \text{false} & \text{if } x_{i,2}^{k} = 1 \end{cases}$$

If there is no solution or if the cost of the solution is greater than zero, then the original problem is not satisfiable: since all the $t_{i,j}^{k}$ are zero, at least one conversion cost was employed, which in turn means that the truth assignment of a variable changes between clauses (which is not possible). Since the input for this selection problem consists only of zeroes and ones (N is a count of the number of zeroes and ones) the selection of a sequence of composite storage structures is strongly *NP*-complete [Garey79]. □

It is common belief that NP-completeness means intractability; however it has been recently pointed out [Garey79], that for certain NP-complete problems, called number problems, there can exist pseudo-polynomial time algorithms for their solution. Pseudo-polynomial time means that the time complexity of the algorithm can be bounded by some polynomial in the input length *and* the magnitude of the maximum number of a given problem instance that bounds the time complexity of the algorithm.

For example, the input to a one-phase selection problem consists of the evaluation matrix, the maximum amount of space and time, and the cost formula. Thus the length of the input is:

$$\sum_{i=1}^{N} \sum_{j=1}^{M_i} \left\lceil \log(s_{i,j}) + \log(t_{i,j}) \right\rceil + \left\lceil \log(S) + \log(T) \right\rceil + L = O(M*N*\log(S*T) + L)$$

where L is the length of the description of the cost formula and M is a bound for all M_i. Assuming that each evaluation of the cost formula requires only polynomial space and time in L, a selection algorithm will be polynomial if it requires only $O(M*N*\log(S*T))$ cost evaluations. All solution algorithms used to date (including exhaustive search, bounch-and-bound [Tompa76], and hill-climbing [Low78]) require $O(M^N)$ evaluations in the worst case.

Elsewhere it has been shown that the one-phase storage structure selection problem is also NP-complete [Ramirez80]. Thus, unless $P = NP$, there is no deterministic polynomial time algorithm to solve even the simplest problem. However, because the one-phase problem is a so-called number prolem, a pseudo-polynomial time algorithm can be found. As demonstrated in the next section, dynamic programming yields a one-phase selection algorithm that runs in time $O(M*N*\min(S,T))$.

3.2. A selection algorithm based on dynamic programming

Dynamic programming is an optimization technique used to make a sequence of interrelated decisions which maximize (or minimize) some measure of value [Bellman57, Dreyfus77]. Although used to solve several important problems in other areas of computer science [Brown79], it has not been applied to the selection of storage structures.

This technique is applicable since the one-phase problem can be partitioned into stages, each stage representing a substructure for which an assignment is to be made.†

Each stage has a number of associated states corresponding to the value of the amount of storage space and time remaining to be allocated. These states are used to represent the various possible conditions in which the system might find itself when trying to make an assignment for the stage. The effect of such an

† The order of the substructures does not affect the outcome but may affect the algorithm's efficiency.

assignment is to transform one state into a state associated with next stage.

For the Huffman code generation algorithm, the three stages correspond to the implementation selections to be made for the sets, QUEUE, and DICT, respectively. The initial system state represents the condition that no selection has been made and S storage cells and T time units are available. If unordered contiguous stores were selected for the sets, the system would then be in a state indicating that choice, no selections for the other two substructures, and $S-16512$ storage cells and $T-22213$ time units remaining for further allocation.

Thus a sequence of states results in assignments to each of the substructures. Given a particular state, the optimal policy for the remaining stages is independent of the policies adopted in previous stages. A selection algorithm solving this problem finds first the optimal policy for each state with no stages remaining, composes it next with the policy for each state with one stage remaining, etc., until the final solution is computed. The principle of optimality is central to dynamic programming:

"an optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." [Bellman57]

Therefore recursive formulations result.

To solve the one-phase selection problem, let

$$F_i(s,t) = \begin{cases} 1 & \text{if there exists a sequence of implementations one for each of the substructures 1 to } i \text{, such that the space and time required by the } k^{th} \text{ implementation are } s_{k,j_k} \text{ and } t_{k,j_k} \text{ and } \sum_{k=1}^{i} s_{k,j_k} = s \text{ and } \sum_{k=1}^{i} t_{k,j_k} = t \text{, that is, if a sequence of implementations fits } \textit{exactly} \text{ in the resources available.} \\ 0 & \text{otherwise} \end{cases}$$

The following recursive relationship can be derived:

$$F_i(s,t) = \begin{cases} 1 & \text{if there exists } j \text{ such that } F_{i-1}(s-s_{i,j},\ t-t_{i,j}) = 1 \\ 0 & \text{otherwise} \end{cases}$$

The boundary condition is given by:

$$F_1(s,t) = \begin{cases} 1 & \text{if there exists } j \text{ such that } s = s_{1,j} \text{ and } t = t_{1,j} \\ 0 & \text{otherwise} \end{cases}$$

If $\$(X,S,T)$ is expressed as a function $f(space,time)$, the solution will be found by taking the min $f(s,t)$ such that $F_N(s,t) = 1$. In other words F_N will have non-zero entries for all feasible solutions; thus the one that minimizes the cost criterion is easily selected.

For applications in which the space and time constraints (Inequations (3) and (4)) are not present, let the maximum values S and T of these equations be the sum of the largest spaces and times respectively. This makes every combination of implementations feasible.

In practical situations, all measures of space and time will be integral multiples of some underlying units. Thus the straightforward implementation of the recursion involves the computation of at most $O(N*M*S*T)$ operations, since for each stage (substructure) there are at most $S*T$ possible states (combinations of space and time available) and each state requires at most M calculations. The space required to trace the solution is $O(N*S*T)$ storage cells, since at each stage it is necessary to store the outcome for each state.

There are several techniques for reducing the run time for the selection algorithm. The primary one results from the realization that because the application's cost formula is monotonically non-decreasing in space and in time (i.e. $f(s,t) \leqslant f(s+s',t)$ and $f(s,t) \leqslant f(s,t+t')$ for all non-negative s, t, s', and t'), $F_i(s,t)$ can be modified to be

$$F_i(s,t) = \begin{cases} 1 & \text{if there exists a sequence of implementations} \\ & \text{one for each of the substructures 1 to } i \text{, such} \\ & \text{that } \sum_{k=1}^{i} s_{k,j_k} = s \text{ and } \sum_{k=1}^{i} t_{k,j_k} = t \textit{ and} \\ & F_i(s',t') = 0 \text{ for all } s'<s \text{ and } t' \leqslant t \text{ or } s' \leqslant s \\ & \text{and } t'<t. \\ 0 & \text{otherwise} \end{cases} \qquad (7)$$

and recursive or iterative formulations can be similarly augmented. The effect of this change is to guarantee that for all s, $F_i(s,*) = 1$ for at most one value of t and similarly, for all t, $F_i(*,t) = 1$ for at most one value of s. Hence there are only $\min(S,T)$ possible states at each stage, which implies that the selection algorithm runs in time $O(N*M*\min(S,T))$ and space $O(N*\min(S,T))$. Similar modifications have been reported for other storage structure selection algorithms [Berelian77, Low78].

A second realization is to note that the selection algorithm runs more quickly when S and T are small. The implication of this observation is that the time required to select a storage structure can be reduced by considering larger units of space and time, for example, multiples of kilowords and milliseconds rather than bytes and microseconds. As these units are increased however, the discrimination away implementations' performances is blurred, thus implying the possible selection of suboptimal structures. Further techniques for reducing the algorithm's running time can be found elsewhere [Ramirez80].

To illustrate the selection algorithm, it will be applied to find the most efficient representations for the directories of a file for which several attributes (domains) have been inverted. In this example, appropriate implementations for each *directory* are sought, i.e., the implementation for the inverted list themselves is not addressed (for a discussion of how to represent inverted lists, see, for example [Cardenas79]).

In particular, consider a file for which three important attributes have been inverted (e.g., for a chemical substance file, the weights, cost per ton and supplier of the substance). The characteristics of the attributes are assumed to be as follows:

attribute 1:	of 5000 possible distinct values, at most 3000 are expected,
attribute 2:	of 1000 possible distincts values, at most 1000 are expected,

attribute 3: of 500 possible distinct values, at most 100 are expected.

Assume that 90% of the application's activity involves the insertion of records into the file (and therefore occasional insertions into all three directories) and 10% involves searching the directories. Furthermore, assume that a library of implementations for a directory consists of a linearly addressed store, contiguous store, unary chain, binary tree, and threaded binary tree as formulated by Tompa [Tompa74]. The resulting evaluation map is given in Table 4.

Let us assume that the application must run in less than 10000 words, but may have unlimited time (i.e. $S=9999$ and $T=66170$). An iterative formulation of Equation (7) calculates values for F_1 first, using the initial boundary condition. Because the fourth and fifth implementations for attribute 1 have identical space and time, the second implementation uses the same space but more time, and $S=9999$, $F_1(*,*) = 1$ for two entries only (Table 5a). Notice that preserving 4 rather than 5 as the selection number in the third column is arbitrary, but 2 could not have been used. Next the values for F_2 are computed, and because of the monotonicity condition there are only six non-zero entries (Table 5b). The routing data in the fourth column indicates which of the non-zero entries from F_1 are needed to reconstruct a selected composite storage structure, as demonstrated below. Finally F_3 is calculated giving seven possible minimal cost selections (Table 5c).

Depending on the cost formula used by the application, any one of the non-zero entries in F_3 may be the minimal cost storage structure. For example, if $f(s,t)=.000001*s*t$, then the third entry ($s=7580$, $t=1700$) gives a cost of 12.886 which is minimal. Thus the *third* substructure should use implementation 4 and the others are indicated by routing 2. The second non-zero entry in F_2 shows that the *second* substructure in the minimal cost selection uses implementation 1 and routing 2, which in turn indicates that the *first* substructures uses implementation 4. Thus the optimal selection is to use binary trees for the directories for attributes 1 and 3 and to use a linearly addressed store (i.e., conventional array) for attribute 2. (As mentioned above, the preference for binary trees over threaded binary trees is arbitrary for those two attributes, since both space and time are equal.)

It is interesting to notice the effect of using a larger granularity for space and time. If, for example, space were measured in multiples of kilowords and time in multiples of seconds, all s and t entries x in Table 5 would be replaced by rounded values of $.001x$. Applying the algorithm with $S=10$, $T=66$, and $f(s,t)=s*t$ yields the data in Table 6. The number of non-zero entries in each F_i is typically smaller than previously, since the coarser granularity results in more selections having equal values for space and time. Using the same cost formula as before, but adjusted to the new units, the minimal selection is obtained when $s=7$ and $t=2$. The routings in the F_i indicate the same choices as before for the first two substructes, but the directory for attribute 3 is to be implemented by using a contiguous store. Thus, it is seen that adopting a coarser granularity, although typically requiring less computation, may produce a suboptimal solution.

		Implementation				
		I_1	I_2	I_3	I_4	I_5
Substructure	A_1	10000	5400	4000	5400	5400
	A_2	2000	1800	1400	1800	1800
	A_3	1000	180	1400	180	180

(a) $\left[s_{i,j}\right]$ in words

		Implementation				
		I_1	I_2	I_3	I_4	I_5
Substructure	A_1	160	59050	41340	1520	1520
	A_2	60	6840	6900	620	650
	A_3	20	170	220	120	120

(b) $\left[t_{i,j}\right]$ in milliseconds

Table 4: Space and time components of an evaluation map for one-phase selection

(I_1 = linearly addressed store, I_2 = contiguous store, I_3 = unary chain, I_4 = binary tree, I_5 = threaded binary tree)

	s	t	selection
1	4000	41340	3
2	5400	1520	4

(a) Non-zero entries for F_1

	s	t	selection	routing
1	6000	41400	1	1
2	7400	1580	1	2
3	5400	48240	3	1
4	6800	8420	3	2
5	5800	41960	4	1
6	7200	2140	4	2

(b) Non-zero entries for F_2

	s	t	selection	routing	cost
1	8400	1600	1	2	13.44
2	6180	41520	4	1	256.59
3	7580	1700	4	2	12.89
4	5580	48360	4	3	269.85
5	6980	8540	4	4	59.61
6	5980	42080	4	5	251.64
7	7380	2260	4	6	16.68

(c) Non-zero entries for F_3

Table 5: Application of the one-phase selection algorithm to the map in Table 4.

	s	t	selection
1	4	41	3
2	5	2	4

(a) Non-zero entries for F_1

	s	t	selection	routing
1	7	2	1	2
2	5	48	3	1
3	6	9	3	2

(b) Non-zero entries for F_2

	s	t	selection	routing	cost
1	7	2	2	1	14
2	5	48	2	2	240
3	6	9	2	3	54

(c) Non-zero entries for F_3

Table 6: Application of the algorithm with coarser granularity in space and time

3.3. Re-selection of a storage structure

An interesting related problem that is frequently encountered in practice is the one in which the relative frequency of operations performed on the abstract structure's data types changes from time to time. For example, this behaviour may be exhibited by a database that first requires a relatively high number of insertions and updates as compared to the number of queries, and once reaching steady-state, requires fewer insertions and updates and relatively more queries.

Although related to the multi-phase selection problem as outlined at the start of Section 3, storage structure re-selection is far more restrictive. Rather than preprocessing the application's requirements over the whole lifetime to find an overall optimal sequence of representations, only one phase change is considered at any time.

In particular, the problem studied in this section is the one in which an initial set of implementations has been adopted, and it is suspected that that selection may no longer be the most efficient one because the relative frequency of operations has changed. It is desired to find the most efficient set of assignments for this new phase taking into account the initial set and the associated conversion costs. In other words, the problem is to determine whether or not it will be profitable to change the implementation of some (or all) of the substructures and to which new implementations they should be changed.

Mathematically this problem can be formulated as follows:

$$Z = \min \left\{ \$(X,S,T) + \sum_{i=1}^{N} \sum_{j=1}^{M_i} c_{i,j} * x_{i,j} \right\}$$

where $c_{i,j}$ is the conversion cost from the initial assignment for substructure i to implementation j. The restrictions for this problem are as for one-phase selection.

The solution of this problem is achieved by defining $G_i(s,t)$ to be the minimum cost for converting the implementation of substructures 1 to i from the initial assignment of implementations. It is now possible to derive the following recursive relationship for the solution of the problem:

$$G_i(s,t) = \begin{cases} \min\limits_j \left\{ c_{i,j} + G_{i-1}(s - s_{i,j}, t - t_{i,j}) \right\} & \\ \quad \text{if there exists a } j \text{ such that} & \\ \quad G_{i-1}(s - s_{i,j}, t - t_{i,j}) \text{ is finite and } G_i(s',t') = \infty & \\ \quad \text{for all } s' < s \text{ and } t' \leqslant t \text{ or } s' \leqslant s \text{ and } t' < t & \\ \infty \quad \text{otherwise} & \end{cases}$$

The boundary condition is given by:

$$G_1(s,t) = \begin{cases} \min\limits_j \left\{ c_{1,j} \right\} \quad \text{if there exists a } j \text{ such that } s_{1,j} = s & \\ \quad \text{and } t_{1,j} = t \text{ and } G_1(s',t') = \infty \text{ for all } s' < s \text{ and} & \\ \quad t' \leqslant t \text{ or } s' \leqslant s \text{ and } t' < t & \\ \infty \quad \text{otherwise} & \end{cases}$$

and the solution will be obtained by taking:

$$Z = \min_{s,t} \left\{ G_N(s,t) + f(s,t) \right\}$$

The function $G_N(s,t)$ will be finite if there is a feasible solution that uses exactly s space and t time. However, rather than being a Boolean function as was F, G_N will contain the minimum cost of converting the implementations of substructures 1 to N from the initial assignment. The second term in the above minimization formula accounts for the cost of the implementations in this new phase.

The computational complexity of this algorithm is of the same order as that in the previous section, although more operations might actually be performed. Thus, the number of operations is $O(N*M*\min(S,T))$ and $O(N*\min(S,T))$ storage cells will be required.

Storage structure reselection can be illustrated by extending the inverted list directories example of Section 3.1. Assume that after some time, a second phase of operation begins in which the number of insertions decreases to 10% of the activity and the number of queries increases to 90%. Table 7 contains the evaluation map for this second phase and a conversion cost table for the substructures. The application of the re-selection algorithm is similar to that of the one-phase selection algorithm. If there are the same bounds on space and time in the second phase, the minimal cost solution using the same cost formula as for the first phase, but with the addition of conversion costs, results in the conversion of the first and third substructures to contiguous stores and the second remaining unconverted (Table 8).

4. Conclusion

In this paper algorithms for solving two related storage structure selection problems were presented. The algorithms rely on several assumptions about the application environment: the application is expressed as a set of algorithms, the application's performance characteristics are known, the storage structure is restricted to be a composition of members chosen from a finite library, and the selections for each substructure† can be made independently (subject to meeting overall criteria, such as bounded total space). The core of each algorithm is based on the principle of optimality for dynamic programming. As a result it is possible to obtain pseudo-polynomial bounds for their running times.

An example involving few data type occurrences and few library implementations was presented in order to demonstrate that intuition and *a priori* selections might not be the best manner of solving such problems and that hill-climbing or branch-and-bound methods may not be appropriate. As the problem size grows, the advantages of the algorithms presented here are even more striking.

There exist some special cases for which it is possible to reduce the amount of computation required and/or the amount of storage space consumed. For example, when the cost formula is the ratio of two resources (e.g., the total number of input/output operations per time unit) it is possible to devise algorithms whose running time is strictly polynomial, in fact $O(N^3 \log N)$, N being the number of substructures in the application (see the minimal cost-to-time ratio cycle problem [Lawler76]). As a second example, it may be the case that there

† Recall that inter-dependencies among data type *occurrences* can be accomodated by a suitable aggregation into substructures.

		Implementation				
		I_1	I_2	I_3	I_4	I_5
Substructure	A_1	10000	6000	8500	11400	11400
	A_2	2000	2000	2900	3800	3800
	A_3	1000	290	290	380	380

(a) $\left[s_{i,j}\right]$ in words

		Implementation				
		I_1	I_2	I_3	I_4	I_5
Substructure	A_1	360	15160	199820	3530	3470
	A_2	300	2820	61670	2950	2770
	A_3	150	740	3330	1010	1000

(b) $\left[t_{i,j}\right]$ in milliseconds

		Implementation				
		I_1	I_2	I_3	I_4	I_5
Substructure	A_1	470000	282000	552500	0	5244000
	A_2	0	220000	49300	798000	912000
	A_3	20000	5800	8700	0	7600

(c) $\left[c_{i,j}\right]$ in millisecond-words assuming initial selection I_4 for A_1, I_1 for A_2, and I_4 for A_3

Table 7: Evaluation map components and conversion table for reselection

	s	t	c	selection
1	6000	15160	2820000	2

(a) Non-zero entries for G_1

	s	t	c	selection	routing
1	8000	15460	2820000	1	1

(b) Non-zero entries for G_2

	s	t	c	selection	routing	cost
1	9000	15610	2840000	1	1	143.33
2	8290	16200	2825800	2	1	137.12
3	8380	16470	2820000	4	1	140.84

(c) Non-zero entries for G_3

Table 8: Application of the re-selection algorithm to the map and conversion table in Table 7.

are no restrictions on space nor time and the cost function is separable (i.e., the total cost is monotonically non-decreasing in the cost of each component). For example, an installation may operate under a "fixed charge" policy such as $f(s,t) = c_1*g_1(s) + c_2*g_2(t)$ where g_1 and g_2 are monotonically non-decreasing functions, and it may allow an application virtually unbounded resources. In this case, the minimal cost solution is, in fact, achieved when each *component* is implemented using minimal cost; this requires only $O(N*M)$ time. Finally, if the cost function is separable but there are restrictions on space and/or time, a divide-and-conquer technique can be used to reduce the space complexity from $O(N*\min(S,T))$ to $O(\min(S,T))$ without significantly increasing the running time [Ramirez80].

Acknowledgements

We wish to acknowledge the helpful discussions with Gaston Gonnet, Ian Munro, and Martin Tompa, and the financial support of the University of Waterloo and the Natural Sciences and Engineering Research Council of Canada under grant A9292.

References

[Aho74] Aho A.V., Hopcroft J.E. and Ullman J.D. *The Design and Analysis of Computer Algorithms.* Addision-Wesley, Reading, 1974.

[Bellman57] Bellman R. *Dynamic Programming.* Princeton University Press, Princeton, 1957.

[Berelian77] Berelian E. and Irani K.B. "Evaluation and Optimization," *Proceedings of the International Conference on Very Large Data Bases 3* (1977), 545-555.

[Brown79] Brown K.Q. "Dynamic proprammming in computer science," Department of Computer Science, Carnegie Mellon University Technical Report CMU-CS-79-106 (1979).

[Cardenas79] Cardenas A.F. *Data Base Management Systems*, Allyn and Bacon, Boston, 1979.

[Cohen74] Cohen J. and Zuckerman C. "Two languages for estimating program efficiency.," *Communications of the ACM 17*, 6 (June 1974), 301-308.

[Dewar79] Dewar R.B.K., Grand A., Liu S-C., Schwartz J.T., and Shonberg E. "Programming by refinement, as exemplified by the SETL representation sublanguage, *ACM Transactions of Programming Languages and Systems 1*, 1 (July 1979), 27-49.

[Dreyfus77] Dreyfus S.E. and Law A.M. *The Art and Theory of Dynamic Programming*, Mathematics in Science and Engineering, Vol. 130, Academic Press, New York, 1977.

[Garey79] Garey M.R. and Johnson D.S. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* Freeman Co., San Francisco, 1979.

[Geschke77]	Geschke C.M., Morris J.H. and Satterwaite E.H. "Early experiences with Mesa," *Communications of the ACM 20,* 8 (August 1977), 540-553.
[Hubbard75]	Hubbard G. and Raver N. "Automating logical file design," *Proceedings of the International Conference on Very Large Data Bases* (1975), 227-253.
[Knuth73]	Knuth D.E. *Sorting and Searching. The Art of Computer Programming 3,* Addison-Wesley, Reading, 1973.
[Lawler76]	Lawler E.L. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, Toronto, 1976.
[Liskov77]	Liskov B., Snyder A., Atkinson R. and Schaffert C. "Abstraction mechanisms in CLU," *Communications of the ACM 20,* 8 (August 1977), 564-576.
[Low78]	Low J.R. "Automatic data structure selection: an example and overview," *Communications of the ACM 21,* 5 (May 1978), 65-77.
[March78]	March S.T. and Severance D.G. "A mathematical modelling approach to the automatic selection of database designs," *Proceedings of ACM SIGMOD,* 1978, 52-65.
[Ramirez80]	Ramirez R.J. "Efficient algorithms for selecting efficient data storage structures," Department of Computer Science, University of Waterloo, Technical Report CS-80-18 (1980).
[Rowe76]	Rowe L.A. "A formalization of modeling structures and the generation of efficient implementation structures," Ph.D. thesis, Department of Information and Computer Science, University of California-Irvine (1976).
[Rowe78]	Rowe L.A. and Tonge F.M. "Automating the selection of implementation structures," *IEEE Transactions on Software Engineering SE-4,* 6 (November 1978), 494-506.
[Salkin75]	Salkin H.M. *Integer Programming.* Addison-Wesley, Reading, 1975.
[Santoro80]	Santoro N. "Efficient abstract implementations for relational data structures," Department of Computer Science, University of Waterloo, Technical Report CS-80-21 (1980).
[Shaw77]	Shaw M., Wulf W.A., and London R.L. "Abstraction and verification in Alphard: Defining and specifying iteration and generators," *Communications of the ACM 20,* 8 (August 1977), 553-564.
[Tompa76]	Tompa F.W. "Choosing an efficient internal schema," *Systems for Large Data Bases,* Lockemann and Neuhold (Eds.) North-Holland, New York, 1976, 65-77.

[Tompa77]	Tompa F.W. "Data structure design," *Data Structures, Computer Graphics and Pattern Recognition.* Klinger, Kunii and Fu (eds.), Academic Press, New York, 1977, 3-30.
[Wagner75]	Wagner H.M. *Principles of Operation Research*, second edition. Prentice-Hall, Englewood Cliffs, 1975.
[Wichman72]	Wichman B. "Estimating the execution time of an ALGOL program," *SIGPLAN Notices 6*, 8 (August 1972), 24-44.
[Winslow75]	Winslow L.E. and Lee J.C. "Optimal choice of data restructuring points," *Proceedings of the International Conference on Very Large Data Base,* Framingham Mass., 1975, 353-363.