SELECTION OF EFFICIENT STORAGE STRUCTURES *

Frank Wm.Tompa
Raul J. Ramirez

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

CS-79-40
November 1979

* The research reported in this paper was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant A9292 and by the University of Waterloo.

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 matrices which reflect the expected run time and storage space required by each component of the application's data structure. Those matrices 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.

Keywords and Phrases: data structure design, space/time efficiency, storage structures, library of implementations, evaluation matrices, dynamic programming.

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

I. Introduction.

Data structure design involves several levels of data abstraction [Tompa77]. At one of the levels, the data structure schema is defined in terms of a composition of data types occurrences (e.g. occurrences of sets, 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 use of an abstract data type need not (in fact, cannot) rely on any particular implementation for the type. This independence gives an implementer the freedom to select or to change the representation of 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 problem investigated in this paper is the selection of an efficient composite storage structure for the abstract

structure for a given application. In particular, the representation is to be selected by composing appropriate representations for each data type occurrence in the abstract structure. Each representation must be chosen from a finite set of implementations, henceforth called the library of implementations, such that the composite storage structure obtained is the most efficient one in terms of a given cost formula.

Closely related work has concentrated on two areas: the creation of a library of implemantations and the use of an evaluation matrix to choose efficient representations based on that library.

I.1 The library of implementations.

The library of implementations contains a set of possible representations for each of the abstract data types available at the abstract structure level. Thus each member is a cluster of code that implements the defined operations for a particular representation of the type.

Some of the implementations in the library may be better suited than others for a given operation. In order to evaluate the appropriateness of the different representa-

tions 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 set of operations and the expected number of storage cells consumed by the data.

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 appropiate 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.

I.2 The evaluation matrix.

Given a library of implementations whose members have been characterized by parametric formulas for the expected run time and storage space, the next step is to evaluate how the data types will be used in the particular application. 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, as one substructure [Low78, Tompa76, Dewar79].

Each substructure is evaluated by considering the parameters in each of the formulas for the type's implementations and substituting parametric values that represent the application usage of this data type on the substructure. These substitutions produce an evaluation matrix in which the (i, j) element represents the expected run time and expected storage space consumed by the ith substructure during the application's lifetime when implemented by the jth possible implementation for this substructure in the library.

The search of an evaluation matrix to find the most efficient structure for a given cost formula has been investigated by several researchers. If the number of possible implementations for each of the N substructures is bounded by M, an exhaustive search would require the evaluation of O(M**N) alternatives.

I.3. Storage structure selection as a zero-one integer programming problem.

The goal of storage structure design is to find an assignation for each substructure such that the final selection is the most efficient among all possible selections. Such a search of the evaluation matrix can be posed as an integer programming problem in the following manner:

Let N be the number of substructures for which assignations are sought,

M(i) be the number of implementations in the library for substructure i,

X be a matrix of zero-one variables in which x(i,j) represents whether or not implementation j is to be selected for substructure i,

s(i, j) be the estimated storage space consumed by implementation j when used for substructure i,

t(i, j) be the estimated run time of implementation j when used for substructure i,

S, T be the maximum amount of storage space and running time, respectively, available to be used by the combined selected implementations,

and COST(X,S,T) be a monotonic cost function in terms of the total amount of space and total amount of time consumed by the final selection X when constrained to the bounds S and T.

then:

$$Z = \min\{ COST(X,S,T) \} \qquad (I.1)$$

for all X such that:

$$\sum_{j=1}^{M(i)} x(i,j) = 1 \quad \text{for } i=1..N \qquad (I.2)$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} x(i,j) * s(i,j) \leq S \qquad (I.3)$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} x(i,j) * t(i,j) \leq T \qquad (I.4)$$

$$x(i, j) = 0, 1 \text{ for all } i=1..N, j=1..M(i) \qquad (I.5)$$

Equation (I.1) formulates the goal: to minimize a cost that is a function of the storage space and run time consumed by the final selection. Constraints (I.2) force the selection of just one implementation for each of the N substructures, since the x(i, j) can only assume zero or one values (constraints (I.5)). Finally constraints (I.3) and (I.4) allow only assignations that do not exceed the given bounds.

There exist several methods for solving zero-one

integer programming problems (e.g. cutting plane techniques [Salkin75] and enumerative techniques [Salkin75, Wagner75]). However, specialized algorithms that exploit the special structure of the given problem and that are more efficient from the computational point of view are often employed.

In order to reduce the expected number of alternatives examined, Low used a hill-climbing technique attempting to minimize the expected space-time integral of the program execution [Low78]. This technique cannot ensure a global optimum, and its running time is potentially exponential in the number of implementations and substructures.

Tompa proposed a branch-and-bound technique to find the assignment that minimizes the specified cost formula [Tompa76]. Branch-and-bound will always produce a global optimum, but it, too, may consume an amount of time that is exponential in the number of susbstuctures.

In this paper, it will be shown that dynamic programming can be used to find the optimal assignment in pseudo-polynomial time, i.e. time polynomial in the length of the input and some specified integer [Garey79].

II. 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].

This technique is applicable since the original problem I.1 to I.5 can be partitioned into stages, each stage representing a substructure for which an assignation 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 assignation for that stage. The effect of such an assignation is to transform one state into a state associated with next stage.

Thus a sequence of states results in assignations 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. Thus, an algorithm solving this problem finds first the optimal policy for each state with no stages remaining, composes it next with the

* The order of the substructures does not affect the final selection; however, it may affect the efficiency of the algorithm.

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.

II.1 The selection algorithm.

To solve the selection problem stated in I.1 to I.5, let

$$F(i,s,t) = \begin{cases} 1 & \text{if there exists a set of implementations (assignations) one for each of the substructures i to N, such that SUM(k=i..N) s(k) = s and SUM(k=i..N) t(k) = t, that is, if a set of implementations fit } \underline{\text{exactly}} \text{ in the resources available.} \\ 0 & \text{otherwise} \end{cases}$$

The following recursive relationship can be derived:

$$F(1,s,t) = \begin{cases} 1 & \text{if } \exists j \text{ such that } F(i+1,s-s(i,j),t-t(i,j)) = 1 \\ 0 & \text{otherwise} \end{cases}$$

The boundary condition is given by:

$$F(N,s,t) = \begin{cases} 1 & \text{if } \exists j \text{ such that} \\ & \quad s = s(N,j) \text{ and } t = t(N,j) \\ 0 & \text{otherwise} \end{cases}$$

If COST(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(1,s,t) = 1. In other words F(1,*,*) 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 constraints I.3 and I.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.

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.

II.2 Reducing the number of alternatives for the state variables.

In the previous section the state variable s ranged from 0 to S, the maximum amount of space available, at each stage in the recursion; however, it is possible to apply the dynamic programming recursion to a subset of the values of s only.

Define:

$$smin(i) = \underset{j}{MIN}\ s(i,j) \qquad smax(i) = \underset{j}{MAX}\ s(i,j)$$

$$sminN(i) = \sum_{k=i}^{N} smin(k) \qquad smaxN(i) = \sum_{k=i}^{N} smax(k)$$

$$smin1(i) = \sum_{k=1}^{i-1} smin(k)$$

Theorem:

Given a problem having a cost function in terms of a state variable s, the range of s at stage i of the recursion is bounded by:

$$sminN(i) \leq s(i) \leq MIN\ (S-smin1(i),\ smaxN(i)).$$

Proof:

Upper bound: if $s(i) \geq smaxN(i)$ then every implementation fits into the resource constraints, thus it is

possible to select the best element for each implementation and the optimal solution for F(i,s(i),t(i)) is identical to that for F(i,smaxN(i),t(i)). Substructures 1 to i-1 need at least smin1(i) resource units since otherwise no assignation is possible, violating restriction I.2, thus S-smin1(i) resource units at most will be left for substructures i to N. Therefore s(i) $\leq$ MIN(S-smin1(i), smaxN(i)).

Lower bound: At stage i it is necessary to have at least sminN(i) units of resource in order that some selection will be possible for each substructure between i and N. Thus s(i) $\geq$ sminN(i).

As a consequence of this theorem the required amount of computation for the solution of the problem can be reduced in practice. When more than one resource is consumed by the implementations (e.g. space and time), it is possible to find similar inequalities for each of the state variables, i.e. a multidimensional bound will apply.

III. 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 insertion and updates as compared to the number of queries, and once reaching steady-state, requires fewer insertions and updates and relatively more queries.

It is clear that in general, the best set of implementations for one phase of the application is not necessarily the best for the next phase. It may therefore be worthwhile to change the implementation of some (or all) of the substructures between phases.* These changes have an associated cost to convert the data from one representation to another, and these costs might outweigh the savings gained by the new assignation. Therefore special care should be taken when such situations arise.

The problem studied in this section is the one in which

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

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 assignations 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 :

$$Z = \text{MIN} \{ \text{COST}(X,S,T) + \sum_{i=1}^{N} \sum_{j=1}^{M(i)} c(i,j)*x(i,j) \}$$

where c(i,j) is the conversion cost from the initial assignation for substructure i to implementation j. The restrictions for this problem are identical to the problem in Section I.3; for the sake of brevity, they are not repeated here.

The solution of this problem is achieved by defining

G(i,s,t) to be the minimum cost for converting the implementation of substructures i to N from

the initial assignation of implementations.

It is now possible to derive the following recursive relationship for the solution of the problem:

$$G(i,s,t)=\begin{cases} \underset{j}{MIN}\{c(i,j)+G(i+1,s-s(i,j),t-t(i,j))\} & \forall j \text{ such that} \\ \qquad G(i+1,s-s(i,j),t-t(i,j)) \text{ is finite} \\ \text{infinite otherwise} \end{cases}$$

The boundary condition is given by:

$$G(N,s,t)=\begin{cases} \underset{j}{MIN}\{c(i,j)\} & \forall j \text{ such that} \\ \qquad s(N,j)=s \text{ and } t(N,j)=t \\ \text{infinite otherwise} \end{cases}$$

and the solution will be obtained by taking:

$$Z = \underset{s,t}{MIN}\{G(1,s,t) + f(s, t)\}$$

The function G(1,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(1,*,*) will contain the minimum cost of converting the implementations of substructures 1 to N from the initial assignation. 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. Furthermore, very similar bounds to the ones in Section II.2 can be easily devised for this problem. Thus, the number of operations is O(N*M*S*T) and O(N*S*T) storage cells will be required.

IV. An Example.

The example presented in this section will help to illustrate the application of the previous ideas. The example selected is a hypotetical one in order that the main ideas can be explained without overwhelming the discussion with unnecessary details; however the values used in the example closely resemble those of actual applications.

Consider an application composed of four substructures (e.g. array, set, tree, list, table) and a library of implementarions containing five different implementations for each of the substructures (e.g. contiguous store, unary chain, binary tree, bit map). The following evaluation matrices may apply:

Implementations	Substructures 1	2	3	4
1	2	2	10	10
2	6	3	9	4
3	4	5	8	3
4	3	4	4	2
5	1	1	1	1

SPACE

	Substructures 1	2	3	4
1	7	11	1	1
2	2	10	2	7
3	3	8	3	8
4	5	9	7	9
5	10	12	17	17

TIME

In each of the above matrices, entry (i, j) represents the amount of space or amount of time consumed by substructure i when implementation j is employed. For example, if substructure 3 were represented by implementation 2, 5 units of

space would be needed for that substructure and 8 units of time would be spent on that substructure's operations.

Assume that the cost formula is given by:

$$f(s, t) = (s * t) + (.05 * s ** 2)$$

This selection criterion is commonly used as a charging formula by computer centers, since it reflects the total amount of resources used weighted by the total amount of time used and penalizing the large usage of scarse resources. Assume furthermore that no restrictions on the total amount of space or time have been imposed. The solution to this problem will use the method presented in Section I.3, where

$$\begin{aligned} COST(X,S,T) = \; & \Big(\Big(\sum_{i=1}^{N} \sum_{j=1}^{M(i)} s(i,j)*x(i,j) \Big) \\ & * \Big(\sum_{i=1}^{N} \sum_{j=1}^{M(i)} t(i,j)*x(i,j) \Big) \Big) \\ & + .05 \Big(\sum_{i=1}^{N} \sum_{j=1}^{M(i)} s(i,j)*x(i,j) \Big) ** 2 \end{aligned}$$

An iterative implementation of the proposed method proceeds from substructure N to substructure 1. Thus, the first step is to use the boundary condition to compute F(N,*,*). The value will be one for all possible space and time combinations for which there exists an implementation

that fits exactly into that combination of resources. For this example F(4,s,t) will have non-zero values for (s,t) $\in$ {(10,1), (4,7), (3,8), (2,9), (1,17)}, and zero values for all other pairs.

The next step is to compute F(3,*,*) using the recursive relationship of Section I.3. This formula establishes that F(i,s,t) will have a non-zero value for a particular (s, t) combination if and only if there exists and implementation j for substructure i that uses s(i,j) units of space and t(i,j) units of time and F(i+1, s-s(i,j), t-t(i,j)) is non-zero.

At this point it is important to notice that, given s and t units of resources there may exist several implementations that satisfy the above criterion, i.e. they collapse into one entry of F(i,*,*). As an example when considering the third substructure the implementations with (s,t) $\in$ {(10,1), (9,2), (8,3)} collapse for F(3,12,10) since F(4,2,9), F(4,3,8) and F(4,4,7) respectively have non-zero values. That is, solving for substructures 3 and 4 with 12 units of space and 10 units of time available has three (equivalent) solutions. In fact F(3,s,t) will have non-zero values for only 18 entries, representing the 5*5=25 possible representations.

The same recursive relationship is next applied to compute F(2,*,*) for which there are 37 non-zero entries and to compute F(1,*,*) with 157 non-zero entries in this example. Once F(1,*,*) has been calculated the cost formula is applied to each entry of this vector, and the minimum value is the cost of the optimal selection. For this example the optimal selection is implementation 5 for each of the substructures; this selection uses 4 units of space and 56 units of time and consequently has a cost of 224.8 cost units.

Some of the advantages of this approach are:

- the method produces all available space/time possibilities at no extra computation, thus once F(1,*,*) has been computed, several cost functions (charging rates) can be applied without the need to solve the complete problem again,

- bounds on the total amount of space and/or time can be specified without increasing the amount of computation (in fact, such bounds actually reduce it),

- at any time during the execution of the procedure the optimal solution for a subproblem is at hand,

- if new substructures are added to the problem there is

no need to solve the complete problem again; rather only the new F(k,s,t) values need be computed using F(1,*,*) as the boundary condition.

When the hill-climbing algorithm proposed by Low is applied to this example, it produces a solution which is 86% worse than the optimal. Branch-and-bound performs rather well for this example, using only 165 cost function evaluations; however if there were bounds on the total amount of space or time used by a solution the technique will not be able to prune as much as for this example. Furthermore, if all the implemantation costs are desired, an exponential amount of computation is required using branch-and-bound, and it cannot be known a priori whether or not the technique will be exponential in time or in space.

To continue the example, assume that the relative frequency of operations has changed such that it is suspected that the original selection of implementations may no longer be the best. Assume furthermore that the new evaluation matrices for the total lifetime of this phase of the application are as follows:

Implementation	Substructures 1	2	3	4
1	6	6	3	1
2	7	4	10	5
3	10	5	9	4
4	9	8	8	3
5	8	7	4	2

SPACE

	Substructures 1	2	3	4
1	9	6	8	10
2	7	10	1	6
3	1	8	2	7
4	3	3	3	8
5	5	4	7	9

TIME

It is possible to find the optimal set of implementations for this phase independently of the original one. Unfortunately, if both phases are taken into consideration this pair of selections may not be overall optimal because of the cost of converting from one implementation to another between phases. Assume that the conversion costs from the initial implementation are as follows:

Implementations	Substructures 1	2	3	4
1	12	35	14	49
2	10	34	15	52
3	9	30	10	50
4	7	31	8	48
5	0	0	0	0

CONVERSION

Entry (i, j) in this matrix represents the conversion cost from the implementation for substructure i in the initial phase to implementation j in this phase. Notice that since implementation 5 was the chosen implementation for each of the substructures in the initial phase, no conversion cost

is incurred if these implementations are selected for this phase.

As discussed in Section III, the method to solve this type of problem is similar to the one in which there is only one phase to the application. The difference is that when only one phase is involved, if two or more assignations use the same amount of resources the algorithm arbitrarily selected one (since all have the same cost). However when conversion costs link two phases these decisions cannot be arbitrarily; rather if the resources consumed by the assignations are identical, the algorithm will pick the one with the smaller conversion cost. In fact, a more costly set of assignations may be chosen if the conversion costs are sufficiently low.

Initially the algorithm uses the boundary condition in Section III, thus computing G(N,s,t), for all implementations j for which s(N,j)=s and t(N,j)=t. For this example G(4,1,10)=49, G(4,5,6)=52, G(4,4,7)=50, G(4,3,8)=48, G(4,2,9)=0, and any other G(N,s,t) have infinite values.

The next step of the procedure is to compute G(3,*,*) using the recursive relationship presented in Section III. Here is where the procedure differs from the one involving only one phase: when two (or more) implementations collapse

into one value of G(3,s,t) the algorithm selects the one that has the minimal conversion cost. For example, when solving for G(3,14,8), i.e. solving for substructure 3 with 14 units of space and 8 units of time available to be allocated, implementation 2 for which (s,t)=(10,1) and implementation 3 for which (s,t)=(9,2) are candidates to represent substructure 3, since both G(4,4,7) and G(4,5,6) are finite. However, since G(4,4,7) is less than G(4,5,6), the former is selected when this combination of resources is available. For this example, G(3,*,*) will have finite values for 12 entries.

When solving for substructures 2 and 1, the same recursive relationship is used, i.e. every time several implementations collapse into the same G(i,s,t) the one with the least conversion cost is preferred. In fact, for this example there are 49 finite entries for G(2,*,*) and 101 for G(1,*,*).

Once G(1,*,*) has been computed, there will be finite entries for every possible selection of implementations. Moreover, the entry for a particular s and t combination contains the smallest cost of converting from the initial selection to a selection of implementations that use s and t units of space and time.

Thus in order to find the best selection, the cost formula should be applied to every entry in the G(1,*,*) vector. In this example the best selection is given by implementations 3, 5, 2 and 5 for substructures 1, 2, 3 and 4, respectively (with a total cost of 501.05 units), even though this selection is not optimal when this phase is considered by itself. Clearly the algorithmic advantages mentioned for the first example are applicable to this procedure as well, and the savings gained by converting implementations in this case is approximately 10%.

V. Conclusion.

In this paper algorithms for solving two related storage structures selection problems were presented. 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, for problems whose cost functions are separable (i.e. the

total cost of the final selection is composed of independent contributions from each substructure), the use of a divide-and-conquer technique will reduce the space required to solve the problem to O(S*T) without significantly increasing the running time [Ramirez80].

Finally when the relative frequency of operations performed over the data type occurrences changes over time, the optimal selections of implementations for consecutive phases of the application might not be optimal overall because of conversion costs. This dynamic selection problem is a further generalization of the one presented in Section III where it was assumed that there are only two phases to the application and the implementations for the first of them have already been selected. One approach to solving this problem is to use a suitable form of dynamic programming to compose solutions to the storage structure problems for each phase [Ramirez80]. Unfortunately at this time no pseudo-polynomial algorithm is known for this general problem.

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.

[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 Volume 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.

[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.

[Ramirez80] Ramirez R.J. Efficient algorithms for selecting efficient data storage structures. Ph.D. thesis, Department of Computer Science, University of Waterloo 1980 (in preparation).

[Salkin75] Salkin H.M. Integer Programming. Addison-Wesley, Reading, 1975.

[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, 1977.

[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 Bases, Framingham Mass., 1975, 353-363.