EFFICIENT ALGORITHMS FOR SELECTING EFFICIENT DATA STORAGE STRUCTURES*

by

Raul Javier Ramirez

Research Report CS-80-18

March 1980

*A thesis
presented to the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1980

© (Raul Javier Ramirez Inurrigarro) 1980

Efficient algorithms for selecting efficient data storage structures.

by

Raul Javier Ramirez Inurrigarro

Abstract

The representations used to implement an application's data structures play an important role in determining its execution cost. Since suitable representations may be selected from a very large class, it is necessary to search for the efficient ones systematically. In this thesis, algorithms based on dynamic programming for selecting efficient composite storage structures are presented.

The general problem is to design a storage structure for an application whose behaviour is characterized by a set of (abstract) data type occurrences for which the relative operation frequencies are known, time-dependent functions. Given a library of possible implementations for each occurrence, this information can be represented by a collection of evaluation and conversion cost matrices as follows. An evaluation matrix reflects the expected run time and storage space required by each representation for every data type occurrence at one phase of the application's lifetime. A conversion cost matrix represents the costs of converting from the potential representations for a data type in one phase to any in the next phase. The goal is to select a representation for each data type occurrence at each phase, such that when considered as a whole, the total cost of the application (including conversions) according to a given cost formula is minimized.

Several special cases of this general problem are examined. When only one evaluation matrix is necessary to characterize the complete behaviour of the application, pseudo-polynomial algorithms that select the most efficient implementations for each data type occurrence are presented and discussed. When an initial set of implementations has been adopted and that selection is no longer the most efficient because the relative frequency of operations has changed, it is desirable to find the most efficient set of assignments for this new phase taking into account the initial set and the associated conversion costs. The algorithm presented to solve this problem is also pseudo-polynomial in its running time. If an application consists of only one data type occurrence, which must adapt to all the phases of the application lifetime, the selection of a storage structure is identified with finding the optimal set of reorganization points for a data structure that deteriorates with time, and the algorithms presented for the solution of these problems run in strictly polynomial time. Finally some solutions to the general problem are described; their running times are exponential in the number of data type occurrences or in the number of phases.

All of the algorithms presented incorporate bounds on the total amount of time and/or space available for each phase, and solve the problems for arbitrary, monotonic cost formulas.

Keywords and Phrases: data structures, storage structures, space/time efficiency, file organizations, reorganization points, information retrieval, library of implementations, evaluation matrices, dynamic programming, recursive optimization, shortest route.

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

A MIS PADRES

Acknowlegments

First, I wish to express my sincere gratitude to my advisor Professor Frank Tompa; his patience, encouragement, guidance and diligence made this work possible. For his continuous reassurance I am sincerely thankful.

I thank my parents, for providing me with a solid foundation in life.

I would like to thank J.I. Munro for his help and valuable suggestions. In particular the results in Sections II.4.2 and III.3 were developed jointly with him.

The proof of strong-NP-completeness in Section IV.3 was provided by M. Tompa.

Many persons provided me with insights, comments, suggestions and friendship. Sincere thanks to Frank Tompa, Ian Munro, Dave Morgan, Gaston Gonnet, Wes Graham, Mike Best, Nicola Santoro and John Morris.

The financial support from the University of Waterloo and from the Consejo Nacional de Ciencia y Tecnologia de Mexico are gratefully acknowledged.

Table of Contents.

I. Introducion.
I.1 A framework for data structure design.
I.2 The thesis problem.
I.2.1 Problem description.
I.2.2 Closely related research.
I.2.3 Summary of new results.
I.3 Thesis outline.

II. Selection of efficient composite storage structures.
II.1 Formal problem definition.
II.2 NP-completeness of storage structure selection.
II.3 Solution for separable cost functions.
II.3.1 Reducing the number of alternatives for the state variables.
II.3.2 Reducing the space complexity.
II.4 Solution for arbitrary cost functions.
II.5 Re-selection of a storage structure.
II.6 Examples.
II.7 Some remarks.

III. Selection of reorganization points for storage stuctures.
III.1 An efficient solution.
III.2 A non-linear application.
III.3 Lower bounds.
III.4 Partial reorganization.
III.5 The algorithms.
III.6 Some remarks.

IV. Linking selections of composite storage structures.
- IV.1 Mathematical formulation for the selection of a sequence of composite storage structures problem.
- IV.2 Linking selections involving one substructure at each phase.
- IV.3 Linking selections involving N substructures at each phase.
- IV.4 Transformation of a P-phase selection problem into a 1-phase problem.
- IV.5 An example.
- IV.6 Some remarks.

V. Conclusions.
- V.1 Thesis contributions.
- V.2 Applicability of results.
- V.3 Further research.

References.

List of Tables

II.1 Evaluation matrix for Gaussian elimination example.
II.2 Cost of different implementations for the row data type, single precision.
II.3 Cost of different implementations for the row data type, double precision.
III.1 Optimal reorganization points for non-linear application.
III.2 Reorganization points by approximating the cost by linear functions in k.
IV.1 Evaluation matrix for the initial phase.
IV.2 Evaluation matrix for the second phase.
IV.3 Conversion costs for attribute 1.
IV.4 Conversion costs for attribute 2.
IV.5 Conversion costs for attribute 3.

List of Illustrations

II.1	Implementations for a one-dimensional array data type.
III.1	Linear deterioration cost.
III.2	Time vs deterioration.
III.3	Partial reroganization.
IV.1	Linking selections involving one substructure at each phase.
IV.2	Linking selections involving some substructures at each phase.

CHAPTER I

I. Introduction.

I.1 A framework for data structure design.

Current research in data structure design has led to the specification of data at several levels of abstraction. In this manner the design process proceeds through each level independently, thus considering only those aspects important to the particular level under investigation. Since this partitioning into levels is hierarchical, changes in a level of abstraction affect the levels under it only, thus leaving the other levels unaltered. Some other advantages, both conceptual and practical, can be found in [Tompa77].

The levels of data refinement as introduced by Tompa are as follows:

i) data reality: this level represents the data for the application under consideration as it actually exists in the real world.

ii) conceptual model: a refinement of the previous level in which only those relationships considered to be of importance are included. This level defines the universe of discourse for all the uses of the data structure, and it can be defined, for example, in terms of n-ary relations or tables in which all relationships are equally accessible.

iii) abstract structure: a further refinement in which only certain relationships are made explicit. The others may be derived indirectly via algorithms that traverse this structure. It is common to specify this level in terms of a composition of abstract data type occurrences.

iv) storage structure: at this level a realization of the abstract structure is introduced in terms of cells, lists and storage media. This level is often diagramed using boxes to represent contiguous nodes and arrows to represent pointers.

v) primitive encoding: a final computer representation of a given storage structure that specifies the encoding of atomic objects.

Typically when designing a data structure, the designer is constrained to a specified conceptual model defined by an enterprise administrator, and to a given primitive structure imposed by the hardware characteristics, programming languages or operating systems under which the application is to be executed. The designer's freedom lies then on the selection of an appropriate abstract structure for the given conceptual model, and on the selection of a storage structure realization.

As an example, consider a chemical company which desires to automate the records of all its suppliers. The managers of such a company must consider all aspects of data reality, e.g., political stability of supplying countries, financial status of suppliers, and so forth, as well as those of particular interest to the data management application, e.g., identification of supplier, which substances are supplied by each supplier and at which cost.

The application managers communicate their needs at this level of abstraction to the enterprise administrator. It is the responsibility of this person (or persons) to identify and integrate those relationships deemed to be of importance. In this conceptual model all relationships are equally accessible, i.e., no idea of computing efficiency is yet introduced.

On the other hand, the remainder of the design is very strongly influenced by efficiency considerations and is therefore the responsibility of the data structure designer or the database administrator [Tsichritzis78].

The task at the next level is to abstract which relationships can be expressed in terms of other more basic ones. Thus the designer decides which relationships will be stored explicitly, as well as which objects are decomposable into more primitive ones (which implies the need to reconstitute the objects when desired as a whole). For example, one could choose to represent a matrix as a primitive object or as a composition of rows or columns. In fact, this level, called the abstract structure, is commonly expressed in terms of a composition of data type occurrences and in terms of the valid operations over the data types.

In the example of the chemical company, the data structure designer, knowing the type of queries likely to occur, might decide to represent the important attributes in the application via inverted lists. Each such list is an occurrence of a data type. Other queries involving attributes not inverted will be computed by means of algorithms that traverse the complete structure.

Thus, once the abstract structure is chosen, the corresponding algorithms are also determined.

Given that the abstract structure is defined in terms of a composition of data type occurrences, "representation independence" is achievable. This means that the use of an a data type needs not (in fact cannot) rely on any particular implementation for the type. This independence allows the freedom at the next level of refinement to select or to change the implementation of the type so as to improve some measure of performance without recoding any application's uses of the type. Indeed, programming languages such as Alphard [Shaw77], CLU [Liskov77], Mesa [Geschke77], and SETL [Dewar79] among others with the concepts of _form_, _cluster_, _class_ or _map_ provide an ideal mechanism for expressing the abstract structure of an application.

Under the framework just described it is possible to identify the next level of refinement, called the storage structure. At this level, the data structure designer, or a compiler for a high-level language, can select from among a set of implementations for each data type the ones that best suit the application. Furthermore, if desired, there is the flexibility to change implementations at will.

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 <u>library of implementations</u> contains a set of possible representations for each of 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 an array data type might be to create an empty array, to locate an element of the array, to read or to write its contents, and to destroy the array. These operations can be implemented for different representations, e.g., the array can be represented as a contiguous store, linearly addressed store, unary chain, bit map, binary tree structures 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 array is sparse and storage space is at a premium, a unary chain may be better.

The reliance on a library of implementations restricts the solution space to that implicitly described by the library. Another approach would be to allow the solution space to contain all possible implementations (an infinite space). However, given that in the latter approach the number of alternatives is so large and the characterization of all alternatives is an open problem in itself, this research is constrained to the restricted space mentioned first. It is felt that the restriction need not be severe if the library is allowed to be large (which, in turn, requires that its alternatives can be quickly appraised).

The time needed to select appropriate representations depends very strongly on the number of degrees of freedom, that is, the number of independent occurrences for which a choice is to be made. If the number of occurrences of data types is very large (as is often the case) it is important to aggregate them into substructures. These substructures are 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 usually convenient to treat them as a single substructure and insist they, have a common representation.

For this thesis, the choice of storage structure will be made on the basis of a cost formula which is used to evaluate the effectiveness of each selection.. The cost formulas most commonly used weigh the total space consumed by the total time this space was in use. However other monotonic non-decreasing cost formulas could be used. Throughout this thesis, cost will be assumed to be a function of space and time; thus each member of the library of implementations is characterized by parametric formulas that reflect the expected run time for the set of operations and the expected number of storage cells used by the data [Low76, Tompa76].

The library of implementations needs to be created and its members characterized by parametric formulae once only. Afterwards, to design the storage structure for a particular application, it is necessary to generate an evaluation matrix that can be used in an optimization procedure for selecting the most efficient (according to the cost formula) combination of implementations for each of the substructures. This evaluation matrix is produced by substituting parametric values that reflect the application usage of the data types in each substructure. In particular, the (i,j) element of this matrix represents the expected run time and the expected storage space consumed by the i-th substructure when implemented by the j-th possible implementation for this substructure in the library. The optimization procedure mentioned above consists of selecting an implementation for each of the substructures such that the total cost as specified by the cost formula is minimized.

In previous research, 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 formulae can be assigned values that reflect the application's characteristics as well as the computing environment. The evaluation matrix produced by these substitutions was used in a branch-and-bound optimization procedure to select the representations that minimize the application's cost according to a given cost formula [Tompa74, Gotlieb74, Tompa76].

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. Using this evaluation matrix, a hill-climbing optimization procedure was used to minimize the expected space-time integral (sum of space required during each unit of time) of the program execution [Low76, Low78].

The work of De, Haseman and Kriebel [De78] is also related. Their work presents a method for building an optimal network database from relational descriptions (see [Computing Surveys76] for a brief overview of database models) using a dynamic programming algorithm as the optimization procedure.

Berelian, Mitoma and Irani [Berelian73, Mitoma75] present an alternative design methodology to automate and optimize the production of network schema structures. Once again they use dynamic programming to solve the problem.

In all of these studies, the selection is a static selection, that is, no change in the relative frequency of operations over the data types is considered. Given that in certain applications it is possible to identify several _phases_, each phase being sufficiently important to warrant its own selection, it may be cost-effective to convert representations between phases. Thus it is necessary to consider the conversion costs between the implementations possible at each phase so that the overall cost (again according to a cost formula) can be minimized.

As mentioned above, Tompa determines _optimal_ solutions by means of the branch-and-bound algorithm. However, as the number of variables (substructures and implementations) increase, the amount of work as well as the amount of space required to find the selection may grow exponentially in the number of variables. With respect to the work of Low, the disadvantages of a hill-climbing approach are well known: a local optimum, or indeed a saddle-point, could be generated, thus global optimality cannot be guaranteed. Furthermore, the algorithm may run in time exponential in the number of variables.

The work of Irani, as well as that of De, is constrained to simple cost formulas, i.e., those in which the only parameter in the cost formula is the time consumed by the implementations as long as the selection does not exceed a bound on the total space used. This type of cost formula is called separable, since the total value is formed by individual contributions from each of the substructures.

I.2 The thesis problem.

In this section, an overview of the thesis problems and related results will be given. Precise terminology, formal problem statements, and further comments on related research will be included in later chapters as appropriate.

I.2.1 Problem description.

The goal of efficient storage structure design (for the simplest application) consists in finding an assignment for each substructure such that the final selection has the least expected cost among all possible selections. If the number of possible implementations for each of N substructures were M, an exhaustive search would require the explicit consideration of M**N alternatives. It will be shown that dynamic programming can be used as an optimization procedure to find the optimal assignment much more efficiently.

The input to the proposed dynamic programming algorithm consists of an evaluation matrix constructed by any of the methods mentioned above (which implies that the abstract structure, application's usage, and the machine environment are fixed), and a cost formula used to evaluate the efficiency of the selections. The algorithm produces an assignment, drawn from the library of implementations, for each of the substructures defined in the abstract structure. The assignment produced is the one with the minimum cost according to the given cost formula.

First, cost functions for which the total application cost increases monotonically with the sum of the costs for each substructure are considered, and the proposed algorithm incorporates bounds on the total amount of space and/or time allowed to be consumed by the selections. An algorithm for more general cost functions (e.g., the product of space and time) is then analyzed, and the algorithm again incorporates bounds on the total amount of resources available.

Several extensions to this basic selection problem are addressed. When the relative frequency of operations performed on the data types varies with time, it is possible to identify phases of activity for an application [Winslow75]. As a result, interesting selection problems can be solved. If an initial set of implementations has been adopted, but that set is no longer the most efficient, it may be worthwhile to convert to a more suitable storage structure. It is therefore desirable to find the most efficient selection of implementations for this new phase, taking into account the initial set and the conversion costs between the old and new selections. In this case the input of the algorithm is the new evaluation matrix (reflecting the application's usage of each substructure for the new phase), a conversion cost matrix and a cost formula. The output of the algorithm is the least costly assignment to adopt for each of the substructures in this new phase.

A generalization of this problem is one to optimize over several phases simultaneously. The input consists of a sequence of evaluation matrices, one for each phase of the application, a sequence of conversion cost matrices reflecting potential conversion costs between successive phases, and a cost formula that takes into account each phase and the conversion costs between phases. A solution consists of the assignments for each substructure at each of the phases such that the total cost is minimized according to the given cost formula. This problem is different from the previous one; there an initial assignment of implementations was already adopted, and here all the phases' assignments are simultaneously chosen.

Another related problem addressed in this thesis is the one in which a storage structure deteriorates with insertions and deletions, given that the processing efficiency can be regained by reorganizing the structure at some cost. The question is, "When should this reorganization occur?" This problem is also generalized to incorporate the possibility of partial reorganizations, i.e., the potential to reorganize a storage structure to distinct levels at distinct costs. The question is in this case, "When and to what level should the structure be reorganized?"

I.2.2 Closely related research.

As already mentioned the work of Tompa [Tompa74, Tompa76] and of Low [Low76, Low78] are related to this research. The former used a branch-and-bound algorithm to find the optimal assignment of implementations according to a cost formula, the latter a hill-climbing procedure attempting the same goal.

The optimization procedure of Mitoma and Irani determines an optimal schema from the alternatives allowed by the network data base model [CODASYL71]. Their approach is based on a dynamic programming algorithm to find the shortest route through a graph, similar to the method in Section II.3 [Mitoma75]. The same approach and goal is considered by Berelian and Irani, however this work takes into account issues such as security as well as paged environments [Berelian77]. In both of these studies the cost fomulas used to characterize different selections are restricted to ones that are monotonic in the sum of the components' costs, and no attempt is made to allow more general cost formulas.

De, Haseman and Kriebel studied the same problem as Mitoma et al., however they start with a set of third normal form relations and the functional dependencies among the attributes, and produce optimal network data bases. They use a dynamic programming algorithm similar to the one described in Section II.3 and allow slightly more complex but still not arbitrary, cost functions. They also give some ideas on how to improve the time complexity of the algorithm similar to some of those in Section II.3.1.

The work of the SETL group is also related [Schwartz75, Dewar79, Schwartz79]. Their system for choosing representations for sets is based on heuristics, global program optimization and flow analysis, and is capable of selecting acceptable representations for a given storage structure. The emphasis of this work is on the grouping of program variables (data type occurrences) into equivalence classes, so as to minimize the number of some required operation, e.g., to reduce the number of hashing operations.

Cardenas [Cardenas73, Cardenas75, Cardenas79] describes a model and system for the selection of file organizations for data bases. This system generates estimates for the expected storage cost and for the average access time of queries for several file organizations. The estimates are based on the complexity of the query, characteristics of the data base, and characteristics of the given computing environment.

However, in this study consideration is given to just one substructure (file) at a time for which several representations (file organizations) exist.

Another systematic method for the selection of storage structures in secondary storage is described in the work of Severance [Severance72, Severance75], who identifies important parameters used in evaluating different structures. As an example, he observes that linked structures can be characterized by their position on a unit square of the cartesian plane in which the x-axis represents "indirectness" and the y-axis the proportion of linking. The system determines which of many representations is most appropriate for a specific application, searching by ad hoc and potentially exponential means.

Several particular instances have previously been recognized for the problem of determining reorganization points for storage structures that deteriorate with insertions and deletions. Shneiderman presented a closed form solution for linearly increasing deterioration and reorganization costs under the assumption that reorganization occurred at equidistant time intervals [Shneiderman73]. However, when these costs are not constant, the optimal intervals will not be equidistant. Tuel dropped the equidistant assumption and gave a closed form solution for arbitrary linear costs and an approximate policy independent of the structure lifetime [Tuel78]. Unfortunately this result applies to linearly growing files with linearly growing reorganization costs only. Yao, et al. have presented a heuristic that is near optimal for constant reorganization costs and claimed to be "superior" for increasing reorganization costs [Yao76]. To our knowledge, when these assumptions do not apply, no closed form solution is known and no previous work on this area has been reported in the computer science literature.

Lohman and Muckstadt identify the problem of finding optimal reorganization points with the problems of optimal checkpointing and batch updating, and they present a common approach using known results of inventory theory [Lohman77]. In the operations research literature a similar problem has been reported. The "equipment replacement" problem considers the decision to replace or overhaul a machine which deteriorates with age. Dynamic programming algorithms similar to the ones in Chapter III have been used for its solution [Dreyfus77].

To the best of our knowledge, no work has been reported on linking selections of composite storage structures, on the reselection of implementations, or on using arbitrary cost functions in a dynamic programming context.

I.2.3 Summary of new results.

The contribution of this research is to extend previously known algorithms for solving related storage structures selection problems. The algorithms presented incorporate bounds on the available space and time resources. Although other researchers have presented algorithms for solving similar problems, the algorithms previously suggested have exponential running time, produce suboptimal solutions or deal with restrictive cost formulas only. None of these problems remain.

For the re-selection of composite storage structures, an algorithm having pseudo-polynomial running time which also incorporates bounds on the total amount of available resources is presented. The problem is further generalized to a total optimization by considering all the phases of the application simultaneously, and two different approaches are presented for its solution.

The main contribution to the reorganization problem is to bring to the computer scientists' attention the fact that algorithms similar to the ones used to solve the equipment replacement problem can be used for solving this problem as well. In addition, the algorithm is shown to be optimal.

Another contribution of this thesis is contained in two theorems concerning theoretical intractability. In particular, even the simplest selection problem is shown to be NP-complete, and the most generalized problem is shown to be strong-NP-complete.

I.3 Thesis outline.

This thesis is composed of five chapters. After this introduction, Chapter II contains a formulation and presentation of a pseudo-polynomial algorithm for the solution of the selection of efficient storage structures. Initially the general problem is posed as a zero-one integer programming problem in order that it is well-defined, and it is shown to be NP-complete. This is followed by a dynamic programming solution for the case in which the cost function is monotonically increasing as the sum of the individual contributions from each substructure. Techniques that reduce the running time and the storage space required by the algorithm are described. Next a dynamic programming algorithm for arbitrary cost functions is presented. A similar algorithm is then described for the improvement of a composite storage structure, and examples illustrating the solution for both problems are given. The chapter concludes with some remarks on the algorithms presented.

In Chapter III the problem of selecting reorganization points for storage structures that deteriorate with time is addressed. The problem description is followed by a dynamic programming algorithm for its solution. The algorithm is then applied to an specific example, and it is proved that the algorithm itself is optimal. Next the solution is generalized for problems in which it is possible to have partial reorganization. Here an interesting application of the well known "divide and conquer" technique is described, and some remarks are made on the application of these results.

Chapter IV is devoted to the study of the selection of an efficient sequence of storage structures for an application in which several phases are encountered during its lifetime, each phase having different requirements. Once again dynamic programming is used in the selection algorithm. The problem is again initially described mathematically, followed by a solution procedure for the case in which the application has only one substructure that needs to adapt to all the application's phases. This solution is generalized to an arbitrary number of substructures at each phase and it is shown that this generalized storage structure selection problem is strong-NP-complete. Next a different approach is introduced: the algorithm is based on a scheme that transforms a P-phase selection problem into a 1-phase problem. An example illustrating the solution is presented, and the chapter concludes with some remarks regarding the algorithms presented.

Finally Chapter V contains the thesis conclusions, as well as some directions for further research.

CHAPTER II

II. Selection of efficient composite storage structures.

The problem investigated in this chapter is the selection of an _efficient_ _composite_ _storage_ _structure_ for the abstract structure of a given application. In particular, the representation is to be selected as the composition of appropriate representations for each data type occurrence in the abstract structure. As mentioned in the introduction, each representation must be chosen from a finite set of implementations, called the _library_ _of_ _implementations_, such that the composite storage structure obtained is the most efficient one in terms of a given cost formula. It will be assumed that an _evaluation_ _matrix_ that reflects the application's characteristics has been formed.

II.1. Formal problem definition.

The search of the evaluation matrix can be posed as a zero-one integer programming problem with the following definitions:

N	the number of substructures for which an assignment is sought,
$M(i)$	the number of implementations in the library for substructure i,
X	a zero-one matrix in which $x(i,j)$ represents whether or not implementation j is to be selected for substructure i,
$s(i, j)$	the estimated storage space consumed by implementation j when used for substructure i,
$t(i, j)$	the estimated run time of implementation j when used for substructure i,
S, T	the maximum amount of storage space and running time, respectively, available to be used by the combined selected implementations,
$COST(X,S,T)$	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 by the bounds S and T.

The problem is then to find

$$Z = \mathrm{MIN}\ \{\ \mathrm{COST}(X,T,S)\ \} \qquad \text{(II.1)}$$

for all X such that:

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

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} x(i,j) * t(i,j) \leq T \qquad \text{(II.3)}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} x(i,j) * s(i,j) \leq S \qquad \text{(II.4)}$$

$$x(i,\ j) = 0,\ 1\ \forall i=1..N,\ j=1..M(i) \qquad \text{(II.5)}$$

Equation (II.1) formulates the goal: to minimize a cost that is a function of the run time and the storage space consumed by the final selection. Constraints (II.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 (II.5)). Finally constraints (II.3) and (II.4) allow only assignments that do not exceed the given bounds in space and time. Section II.2 shows that the selection of composite storage structures problem, defined above, is NP-complete.

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.

One such class of algorithms is encompassed by dynamic programming, 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, as stated in II.1 to II.5, 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 that stage. The effect of such an assignment is to transform one state into a state associated with the next stage.

Thus a sequence of states results in assignments to each of the substructures. Given a particular state, the

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

optimal policy for the remaining stages is independent of the policies adopted in previous stages. Hence, an 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.

II.2. NP-completeness of storage structure selection.

NP is the class of all languages which can be recognized by a non-deterministic Turing machine in time bounded by some polynomial in the length of the input. A language or recognition problem L0 is said to be NP-complete if it is in NP and if, given a deterministic polynomial time algorithm to recognize L0, one can effectively find a deterministic polynomial time algorithm to recognize every language L in NP [Karp75]. It is said that L is <u>reducible</u>

to L0. The notion of NP-completeness is typically extended to Boolean questions by treating the problem statement followed by a candidate solution as the input to the language. The notion is further extended to optimization problems by solving a sequence of Boolean questions under a binary search scheme.

Given that the class is very wide and includes problems that have been studied for a long time, it has been conjectured that no polynomial time algorithms exist, although nobody has yet been able to prove or disprove this conjecture.

Theorem:

The selection of efficient composite storage structure belongs to the NP-complete class of problems.

Proof:

In order to show that a given problem is NP-complete one typically takes the following tack:

i) Demonstrate that the problem is in NP by showing that a correct solution or derivation can be checked in time polynomial in the original input.

ii) Reduce a known NP-complete problem to it in (deterministic) polynomial time.

It is easy to show that the selection of efficient composite storage structures problem is in NP; given a selection of storage structures, a non-deterministic Turing machine will simply check whether or not that solution meets the specified constraints II.1 through II.5. Clearly this validation of the solution can be performed in polynomial time. Thus, the selection of storage structures is in NP.

In order to satisfy requirement ii), the zero-one knapsack problem will be used. This problem, which is known to be NP-complete [Ibarra75], can be states as follows; given a set of objects characterized by their value and weight, an objective K and a constraint W, is it possible to select a subset of them such that:

$$\sum_{i=1}^{N} v(i) * x(i) \geq K$$

$$\sum_{i=1}^{N} w(i) * x(i) \leq W$$

$$x(i) = 0,\ 1 \quad \forall i=1..N$$

where:

v(i) represents the cost of object i

w(i) represents the weight of object i

W represents the size of the knapsack

x(i) represents whether or not object i will be included.

A special case of the selection of composite storage structures problem is to minimize time subject to a constraint in space. If there are only two implementations for each substructure, this can be stated as follows:

$$\sum_{i=1}^{N} \sum_{j=1}^{2} t(i,j) * x(i,j) \leq K$$

$$\sum_{i=1}^{N} \sum_{j=1}^{2} s(i,j) * x(i,j) \leq S$$

$$\sum_{i=1}^{N} x(i,j) = 1 \quad \forall j=1..2$$

$$x(i,\ j) = 0,\ 1\ \forall i=1..N,\ j=1,2$$

Clearly this minimization problem can be transformed into a maximization by setting t'(i,j) = K - t(i,j), where K is the maximum t(i,j). The reduction of the zero-one knapsack problem to this selection problem will be as follows:

Given a zero-one knapsack problem as above, set $t'(i,1) = v(i)$, $t'(i,2) = 0$ $\forall i=1..N$

$s(i,1) = w(i)$, $s(i,2) = 0$ $\forall i=1..N$

Next solve the following decision problem:

$$\sum_{i=1}^{N} \sum_{j=1}^{2} t'(i,j) * x(i,j) \geq K$$

$$\sum_{j=1}^{2} x(i,j) = 1 \quad \forall i=1..N$$

$$\sum_{i=1}^{N} \sum_{j=1}^{2} s(i,j) * x(i,j) \leq W$$

$$x(i, j) = 0, 1 \; \forall i=1..N, j=1,2$$

The $t'(i,2)$ and $s(i,2)$ represent slack variables to satisfy the requirement that exactly one implementation must be chosen for each substructure. If $x(i,2)$ is one then the corresponding object i will not be selected; however if $x(i,1)$ is one then the object will be selected.

The transformation shown is obviously polynomial, and thus a polynomial time algorithm for the storage structure selection problem will induce a polynomial time algorithm for the zero-one knapsack problem. Therefore the storage structure selection problem is NP-complete.

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 <u>and</u> the magnitude of the maximum number of a given problem instance that bounds the time complexity of the algorithm. In the following sections pseudo-polynomial time algorithms will be developed to solve related storage structures selection problems.

The input to the 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:

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} [\log(s(i,j))+\log(t(i,j))]+[\log(S)+\log(T)]+L =$$

$O(M*N*\log(S*T)+L)$, where L is the length of the description of the cost formula. 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 (i.e., the naive approach and those suggested

* Logarithms are taken to base 2.

by Tompa and Low) require O(M**N) evaluations in the worst case, whereas the one in Section II.3 will require O(N*M*S*T) and this is therefore pseudo-polynomial [Garey79].

II.3 Solution for separable cost functions.

A cost function is said to be separable, if it is possible to compose the total cost by individual contributions from each one of the components (substructures) [Bellman57].

A degenerate case is the one in which the cost function is separable and no restrictions are imposed on the total amount of space or time used by the final selection. In this case the best choice for each substructure is overall optimal. When this is not the case special techniques attempting to avoid the combinatorial explosion may be used.

When the cost function is separable and bounds on the total amount of time and/or space are imposed, the minimum cost of the composite storage structure is achievable by minimizing the cost for each substructure, such that the final assignment does not exceed these bounds.

Let $$\mathrm{COST}(X,S,T) = \sum_{i=1}^{N} \sum_{j=1}^{M(i)} \mathrm{cost}(i,j) * x(i,j) \qquad \text{(II.6)}$$

where cost(i,j) represents the contribution of implementation j for the substructure i towards the total cost, thus function II.6 is separable.

Let f(i,s,t) be the minimum cost obtainable from the composition of substructures i to N, given that s units of storage space and t units of time remain available to be allocated. It is easy to show that:

$$f(i,s,t) = \underset{\substack{s(i,j)\leq s,\ t(i,j)\leq t \\ j=1..M(i)}}{\mathrm{M\ I\ N}} \{\mathrm{cost}(i,j)+f(i+1,s-s(i,j),t-t(i,j))\} \qquad \text{(II.7)}$$

By computing f(i,s,t) for s=0..S, t=0..T and using the boundary condition

$$f(N,s,t) = \underset{\substack{s(N,j)\leq s,\ t(N,j)\leq t \\ j=1..M(i)}}{\mathrm{M\ I\ N}} \{\mathrm{cost}(N,j)\} \text{ for } s=0..S, t=0..T \qquad \text{(II.8)}$$

the answer is given by f(1,S,T).

Since at each stage of the procedure f(i,s,t) is selected from at most M(i) alternatives and computed for s=0..S, t=0..T, the time complexity of the procedure just described is O(N*M*S*T) operations, where M is the maximum value of M(i) for i=1..N.

Observe that the space complexity of this procedure as described by recursion II.7 has the Markovian property: the value of f(i,s,t) depends only on the current value of the state variables s, t and on the values of f(i+1,*,*). Thus all the dependent history of the process can be contained in just one column of values f(i+1,*,*) of size at most S*T. Therefore only 2*S*T storage cells are required to compute the value of the optimal value function. However, since not the value of the best assignment F(1,S,T) is sought, but rather the set of implementations that achieved this value, it is necessary to store the alternatives selected. Since at each stage there are at most S*T states and since there are N stages, O(N*S*T) storage cells are required, where each cell is assumed to be capable of containing values between 0 and M (i.e., O(N*S*T*log(M)) space is required).

II.3.1 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:

$$\sup(i) = \underset{j}{\mathrm{MAX}}\ s(i,j) \qquad \inf(i) = \underset{j}{\mathrm{MIN}}\ s(i,j)$$

$$u_{\lhd}(i) = \sum_{q=1}^{i-1} \sup(q) \qquad l_{\lhd}(i) = \sum_{q=1}^{i-1} \inf(q)$$

$$u_{\rhd}(i) = \sum_{q=i}^{N} \sup(q) \qquad l_{\rhd}(i) = \sum_{q=i}^{N} \inf(q)$$

Theorem:

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

$$\mathrm{Max}(l_{\rhd}(i),\ S-u_{\lhd}(i)) \leq s(i) \leq \mathrm{Min}(S-l_{\lhd}(i),\ u_{\rhd}(i))$$

Proof:

Upper bound: if $s(i) \geq u_{\rhd}(i)$ 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 $f(i,s(i))$ is identical to that for $f(i,u_{\rhd}(i))$. Substructures 1 to i-1 need at least $l_{\lhd}(i)$ resource units since otherwise no assignment is possible, violating restriction III.2, thus $S-l_{\lhd}(i)$ resource units at most will be left for substructures i to N. Therefore $s(i) \leq \mathrm{Min}(S-l_{\lhd}(i),\ u_{\rhd}(i))$.

Lower bound: substructures 1 to i-1 can use at most $u_{\lhd}(i)$ resource units, thus leaving at least $S-u_{\lhd}(i)$ units for substructures i to N. At stage i it is necessary to have at least $l_{\rightarrow}(i)$ units of resource in order that some selection will be possible for each substructure between i and N. Thus $s(i) \geq \text{Max}(l_{\rightarrow}(i), S-u_{\leftarrow}(i))$.

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, time, etc), it is possible to find similar inequalities for each of the state variables i.e., a multidimensional bound will apply.

II.3.2 Reducing the space complexity.

In Section II.3 it was observed that the straight forward implementation of recursion II.7 requires O(N*S*T) storage cells, since not only the value of the best assignment was required, but also the set of assignments (i.e., implementations) that achieved that value. In order to identify these implementations it was suggested to store the alternative selected for each state at each stage.

A modification of the basic scheme permits a solution that uses only O(S*T) storage space without a significant increase of the time bound.

Theorem (R. Ramirez, J.I. Munro):

The space required to solve the selection of composite storage strutures problem for separable cost functions can be reduced to O(S*T) storage cells by using an algorithm having time complexity of O(N*M*S*T*log(N)).

Proof:

The modified algorithm is based on the well-known technique called divide-and-conquer. The application of this technique relies on the fact that in order to find the value of the optimal assignment, it is not necessary to store the assignments that achieved this value.

First solve the original problem as if only the value of the best assignment were sought rather than the assignment itself. However, when solving for substructure (stage) N/2 (i.e halfway through) label each assignment (state) by the amount of resources required, that is, from 0 to S*T (there are at most S*T states). From stage N/2-1 down to stage 1 carry for each state at each stage the state that this assignment went through when it was at stage N/2. When the value of f(1,S,T) is computed, the identification of the mid-assignment that the optimal set of assignments took at stage N/2 will be known; call it assignment y'.

Once this assignment y' is known, solve the following two subproblems using the same strategy in a recursive manner:

i) find the optimal set of implementations for a problem with substructures 1 to N/2 forcing the selection of implementation y' at stage N/2.

ii) find the optimal set of implementations for a problem with substructures N/2+1 to N.

The first of the subproblems above is identical to the original problem, except that it forces the selection of implementation y' for substructure N/2 and that it is half the size. The second problem is identical to the original one, but only half its size, and therefore the same algorithm could be applied recursively.

Since at any time in the solution of the problem only one column of decisions of at most size S*T is conserved, it immediately follows that only O(S*T) storage cells are required by the given algorithm. It remains to show that the above strategy does not significantly increase the time complexity of the procedure.

Let $\Phi(N)$ represent the computation time to find the optimal assignment, and assume $\Phi(1) = M$, the number of evaluations to determine the best implementation for specified maximal s and t. Since N*M*S*T evaluations are required to find the <u>value</u> of the optimal assignment, the application of the above recursive scheme produces the following equation:

$$\Phi(N) = N*M*S*T + 2*\Phi(N/2)$$

The solution of this equation is given by:

$$\Phi(N) = N*M*S*T * \log(N) + M*N$$

This shows that the proposed method increases the time complexity by only a log(N) factor.

II.4 Solution for arbitrary cost functions.

Section II.3 was devoted to the study of the optimal selection of storage structures when the selection criterion was limited to one parameter consumed by the implementations. In this section, the criterion upon which a selection is to be made is extended to include costs that involve arbitrary functions of both time and space as used by the library's implementations. The method will also allow for constraining the solution to selections that do not exceed given bounds on space and/or time. For example consider the following problem:

$$\begin{aligned} COST(X,S,T) = \{C1((& \sum_{i=1}^{N} \sum_{j=1}^{M(i)} s(i,j)*x(i,j)) \\ * (& \sum_{i=1}^{N} \sum_{j=1}^{M(i)} t(i,j)*x(i,j))) \qquad \text{(II.9)} \\ + C2(& \sum_{i=1}^{N} \sum_{j=1}^{M(i)} s(i,j)*x(i,j)) ** 2\} \end{aligned}$$

The selection criterion II.9 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 heavily penalizing the large usage of scarce resources.

To solve this problem let:

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

The following recurrence relation can be derived:

$$F(i,s,t) = \begin{cases} 1 & \text{if } \exists j \text{ such that} \\ & \quad F(i+1,s-s(i,j),t-t(i,j)) = 1 \\ 0 & \text{otherwise} \end{cases} \qquad \text{(II.10)}$$

The boundary condition is given by:

$$F(N,s,t) = \begin{cases} 1 & \text{if } \exists j \text{ such that} \\ & \quad s = s(N,j) \text{ and } t = t(N,j) \\ 0 & \text{otherwise} \end{cases} \qquad \text{(II.11)}$$

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 best one according to cost criterion II.1 could be easily selected. Therefore the evaluation of the cost function (e.g., II.9) is only required when choosing from F(1,*,*); this drastically reduces the number of function calculations.

For applications in which the constraints II.3 and II.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 recurrence II.10 involves the computation of at most O(N*M*S*T) operations, since again 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.

The results of Section II.3.1 on how to reduce the computation time by reducing the number of states at each stage could be applied to this recursion as well. In this case, very similar bounds can be derived. One of these bounds restricts the number of possible states for the s state variable, and the other restricts the number of states for the t state variable. However, since for this criterion the algorithm finds the set of implementations that fit exactly into the available resources rather than the best fit as in Section II.3, the bounds must be adjusted slightly. In fact $l_{\rhd}(i) \leq s(i) \leq \mathrm{Min}(S-l_{\lhd}(i),\ u_{\rhd}(i))$. The lower bound is changed to allow the search for the implementations that achieve an exact fit (i.e., the lower bound is for this case only $l_{\rhd}(i)$). This again makes it possible to reduce the amount of computation and, consequently, to solve problems of large size. Unfortunately, since the cost criterion in this case is not separable, the technique of Section II.3.2 cannot be applied for this problem.

II.5. Re-selection of a storage structure.

A related problem frequently encountered in practice is that in which the relative frequency of operations performed on the data types varies with time. Typically a small number of distinct phases can be identified. For example, this behaviour may be exhibited by a database that is first created and consequently requires a relatively high number of insertion and updates as compared to the number of queries. Once the database has reached steady-state, the number of insertions and updates would probably decrease and the number of queries increase.

The best assignment 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 for converting the data from one representation to another, and these costs might outweigh the savings gained by the new assignment. Therefore special care should be taken when such situations arise.

The problem studied in this section is the one in which an initial set of implementations has been adopted, but it is suspected that this set may no longer be the most efficient one because the relative frequency of operations has changed. It is desired to find the most efficient assignment of implementations 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. A more general version of this problem is examined in Chapter IV.

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

The re-selection problem can be formulated as follows (using the same notation as in the preceding section):

$$Z=\mathrm{MIN}\{\ \mathrm{COST}(X,S,T) + (\sum_{i=1}^{N} \sum_{j=1}^{M(i)} c(i,j)*x(i,j))\} \qquad (II.12)$$

where c(i,j) is the conversion cost from the initial implementation for substructure i to implementation j. The restrictions for this problem are identical to the problem in Section II.1.

The solution of this problem is achieved by defining

G(i,s,t)	to be the minimum cost of converting the implementation of substructures i to N from the initial assignment of implementations.

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

$$G(i,s,t) = \begin{cases} \underset{j}{\mathrm{M\ I\ N}}\ \{c(i,j)+G(i+1,s-s(i,j),t-t(i,j))\} & \forall j \text{ such that} \\ \qquad G(i+1,s-s(i,j),t-t(i,j)) \text{ is finite} & \\ \text{infinite otherwise} & \end{cases} \qquad (II.13)$$

The boundary condition is given by:

$$G(N,s,t) = \begin{cases} \underset{j}{MIN}\{c(i,j)\} \ \forall j \text{ such that} \\ \qquad s(N,j)=s \text{ and } t(N,j)=t \\ \text{infinite otherwise} \end{cases} \qquad \text{(II.14)}$$

and the solution will be obtained by taking:

$$Z = \underset{s,t}{MIN}\{G(1,s,t) + f(s,t)\} \qquad \text{(II.15)}$$

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 assignments. The second term in the above minimization formula accounts for the cost of the implementations in this new phase.

The run time of this algorithm is of the same order as that in the previous sections, although more operations might actually be performed. Thus, the number of operations is $O(N*M*S*T)$ and $O(N*S*T)$ storage cells will be required. Bounds differing only slightly from those of Section II.3.1 can again easily be derived.

II.6. Examples.

The examples presented in this section will help to illustrate the application of the previous ideas. The first is a simplistic example which serves to express the main ideas without overwhelming detail. The values used in this example do, however, closely resemble those of actual applications.

Consider an application composed of four substructures (e.g., chosen from array, set, tree, list, table) and a library containing five different implementations for each of the substructures. Suppose the following evaluation matrices apply:

		Substructures			
		1	2	3	4
Implementations	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 2 were represented by implementation 3, 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:

$$f(s, t) = (s * t) + (.05 * s ** 2) \qquad \text{(II.16)}$$

where S and T represent the sum of the spaces and the sum of the times of the selected implementations. 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 II.4.

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 II.10 of Section II.4. This formula establishes that F(i,s,t) will have a non-zero value for a particular (s, t) combination if an only if there exists an 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. In other words, F(i,s,t) will have a non-zero value if and only if there exists a set of implementations one for each of the substructures i to N such that the sum of the spaces is s and the sum of the times is t.

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 substructure 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 18 entries, representing the 5*5=25 possible representations.

The same recurrence relation is next applied to determine 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 the amount of computation.

- 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. In fact only the new F(k,s,t) vectors 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 for 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 solutions to the problem 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 second phase (i.e., the remaining lifetime of this application) are as follows:

SPACE

Implementations	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

TIME

Implementations	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

It is possible to find the optimal set of implementations for this phase independently of the original one by merely repeating the above procedure using these new matrices. 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 the following:

CONVERSION

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

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 optimal implementation for each of the substructures in the initial phase, no conversion cost is incurred if the same set of implementations is selected for this phase.

As discussed in Section II.5, the method of solving this type of problem is similar to that used when there is only one phase. The difference is that when only one phase is involved, if two or more assignments 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 made. If the resources consumed by the assignments are identical, the algorithm will pick the one with the smaller conversion cost. In fact, a more costly set of assignments may be chosen if the conversion costs are sufficiently low.

Initially the algorithm uses the boundary condition II.14 of Section II.5, thus computing G(N,s,t), for all implementations j for which s(N,j)=s and t(N,j)=t thus 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 II.13. 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 smallest 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 15+G(4,4,7) is more than 10+G(4,5,6), the latter 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 smallest 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 (i.e., if there are several selections that consume the same s and t, the entry will contain the smallest conversion cost).

Thus to find the best selection, the cost formula should be applied to every entry in the G(1,*,*) vector. In this example the best choice for this second phase is to convert to implementations 3 and 2 for substructures 1 and 3, respectively and to leave substructures 2 and 4 implemented as was (with a total cost of 501.05 units). This selection would not be optimal if this phase were 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%.

As a practical example of the results, consider Gaussian elimination, for which the algorithms are well understood (see, for example, [Conte72]).

The principal data type in Gaussian elimination algorithms is the matrix. One suitable definition of a matrix is:

```
type matrix = array 1..N of row;
 type row = array 1..N of real;
```

Thus a matrix can be thought as linear array of linear arrays of real numbers. A one-dimensional array may easily be one of the basic building blocks for which there exist several implementations (see Figure II.1). The implementations used for this example are described in most data structures textbooks:

i) Linearly addressed store: a block of contiguous memory in which all entries are represented (including null entries) and in which entry i is stored in the ith position of the block of storage.

ii) Contiguous store: an implementation in which only the non-null entries are stored, each entry consisting of an ordered pair of index and value; all such pairs are stored contiguously and in increasing order of their index.

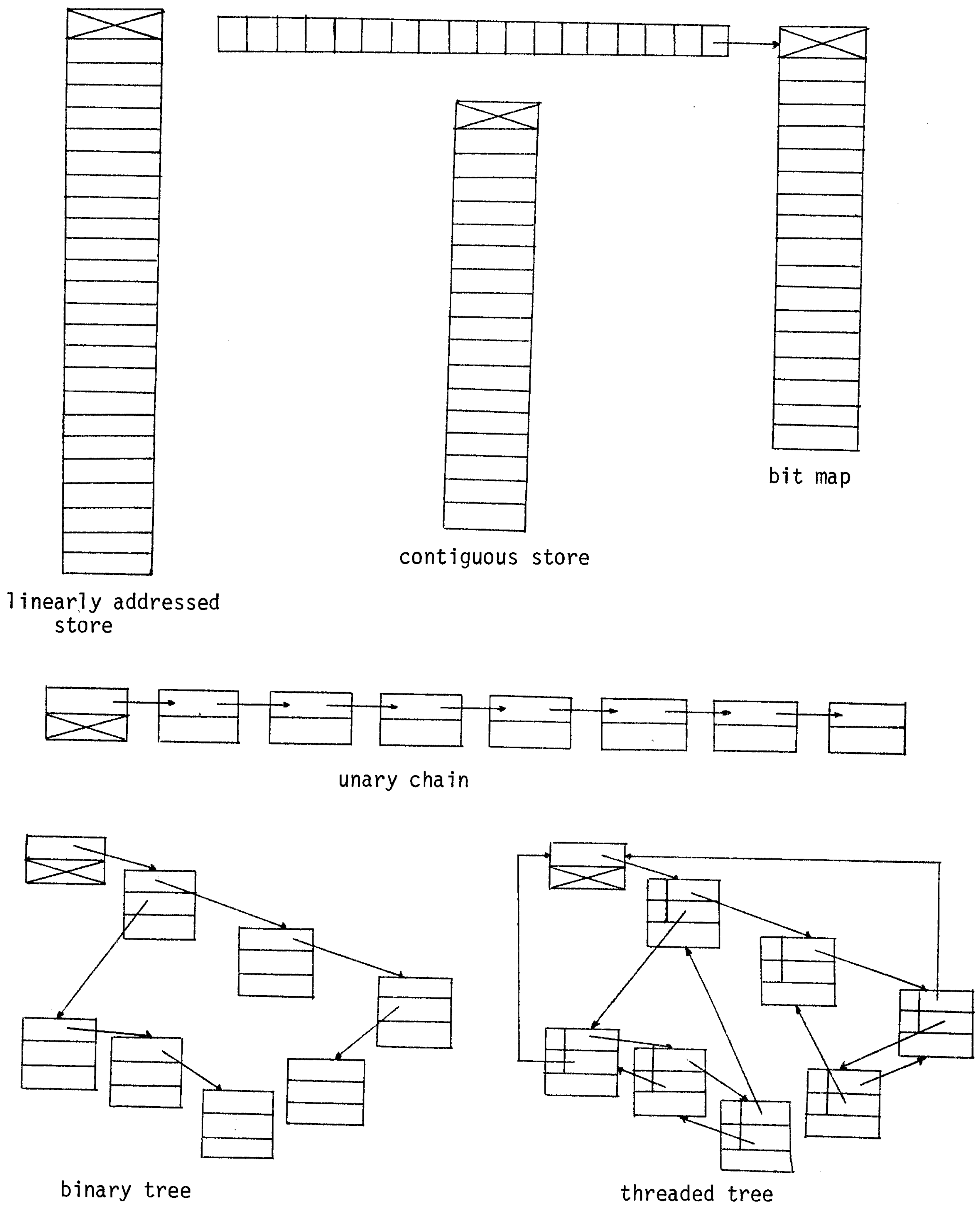

Figure II.1 Implementations for a one dimensional array.

iii) Unary chain: a sequence of non-null entries, each entry being a node that contains an index, a value, and a pointer to the next node in the sequence. The nodes in the sequence are ordered by their index, however they are not necessarily contiguous.

iv) Bit map: an implementation in which a boolean vector acts as an indexing scheme as follows: all non-null entries in the array have true stored in the corresponding place in the boolean vector, and following the boolean vector a contiguous block of storage contains the non-null values in increasing order of their indices.

v) Binary tree: an implementation in which each element is represented by a node in a tree structure. Each node contains an index, a value and two pointers to binary subtrees. The nodes are arranged lexicographically according to the index.

vi) Threaded tree: an implementation identical to a binary tree except that the nodes with less than two non-empty subtrees contain pointers back to an appropriate ancestor forming a thread through all the nodes.

Since in this given choice for abstract structure, a matrix is composed of two substructures, the array of rows and the arrays of reals, and since six possible implementations have been defined for arrays, it follows that there are thirty six different implementations for the matrix data type. (The number of different representations grows as the square of the number of implementations for the basic type.) For example the selection that consists of linearly addressed stores for both substructures is commonly used for small, dense matrices, and the selection that consists of a linearly addressed store for the array of rows and a bit map for each row is commonly suggested for large, sparse matrices.

A possible set of valid operations are (see [Tompa74]):

generate: create an array containing default values for all elements, such that the structure is accessible from a given cell.

probe:	search a given array for the element having a given index and return a pointer to the corresponding element or null if not found.
locate:	find the position of an element having a given index within a given structure in order to add or delete it, and return a pointer to the appropriate location.
insert:	assign a non-null value to the element in a given array at a given index position.
remove:	assign the null value to the element in a given index position.
next:	find the element having non-null value whose index is the lexicographic successor of a given element's index, and return a pointer to it or null if no successor.
head:	find the start of a given array.
access:	obtain a particular field within an element of an array, either to read it or to change it.

For example, in order to access the value of the (i, j) element of the matrix defined above, the following operations are required:

probe for the ith entry of the matrix returning a row

probe for the j-th entry of the row returning an element

access the value field of the desired element

Gaussian elimination consists of two steps [Conte72]:

i) factorization: the original matrix A is factored as PLU, where P is the permutation matrix associated with the pivoting strategy, L is a lower triangular matrix containing the multipliers used during the elimination process, and U is an upper triangular matrix "equivalent" to the A matrix.

ii) substitution: U is used to compute the unknowns.

Before the techniques of the previous section can be applied, it is necessary to code the Gaussian elimination algorithm in terms of the primitive array operations and to compute the relative frequency of each operation parametrically.

The substitution for the parameters in the library of implementations by values that reflect the application usage of the data types produces an evaluation matrix. For a concrete example, assume that Gaussian elimination is to be performed on a matrix of order 500 by 500, also assume that the original matrix has 20% non-zero elements and that the elimination process will produce a matrix with 40% non-zero elements. The evaluation matrix produced is shown in Table II.1, where both single and double precision costs are displayed.

	TIME		SPACE		
	MATRIX	ROW	MATRIX	ROW	
				single	double
lin. add.	.13	5.35	.5	250	500
cont. st.	.65	5.98	1.	250	375
unary ch.	15.74	9.22	1.5	225	300
bit map	24.27	118.30	.52	133	258
bin. tree	1.11	10.66	1.	300	375
thr. tree	1.14	5.37	1.	300	375

Table II.1 Evaluation matrix.

(time is in minutes, space in kilo-words)

It is clear that in order for a matrix to be invertible all of its rows must be present, i.e., the matrix is non-singular. As can be seen from Table II.1, linearly addressed store is the best implementation for the array of rows structure, since it is the one from this library that uses the least time and space. The assignment of an implementation for the row (array of reals) structure is unfortunately not as straightforward, since it is heavily dependent on the cost formula being used. Tables II.2 and II.3 show the best assignment for the row data type for different related cost formulas. The cost formula used for these tables is similar to that of Section II.4 with the parameter C2 set to 1 and C1 varying as shown in the tables. For the purposes of this example, again no restrictions on the total amount of space or time have been imposed.

	C1=2		C1=5		C1=50	
impl.	cost	ratio to optimal	cost	ratio to optimal	cost	ratio to optimal
lin. add.	65496	1.32	69614	1.13	131387	1.
cont. st.	65811	1.33	70328	1.15	139278	1.06
unary ch.	55067	1.11	61392	1.	156272	1.19
bit map	49443	1.	96874	1.58	808343	6.15
bin. tree	96785	1.96	106512	1.73	252420	1.92
thr. tree	93606	1.89	98564	1.61	172938	1.32

Table II.2. Cost of different implementations for the row data type.

(single precision real numbers)

	C1=.5		C1=2		C1=100	
impl.	cost	ratio to optimal	cost	ratio to optimal	cost	ratio to optimal
lin. add.	251872	3.07	255968	2.67	524774	1.51
cont. st.	142898	1.74	145589	1.52	370431	1.07
unary ch.	91705	1.12	95920	1.	371268	1.07
bit map	82129	1.	128051	1.33	3128238	9.
bin. tree	143026	1.74	149104	1.55	546165	1.57
thr. tree	142033	1.73	145131	1.51	347525	1.

Table II.3. Cost of different implementations for the row data type.
(double precision real numbers)

Table II.2 presents the best assignment when the row data type is composed of single precision real numbers occupying each one storage cell. When the extensive use of storage space is heavily penalized (C1=2) the time consumed by the operations is of little concern and thus a compact scheme is to be preferred, in this case the bit map implementation. Alternatively, when the amount of time consumed by the operations is also of interest (C1=5) the unary chain implementation is selected; and when the most important factor is the time consumed, the linearly addressed store (dense representation) is the one preferred.

Table II.3 shows the best assignment for the case in which the elements of the row are double precision numbers. Here the displayed values for C1 are different from those in

Table II.2 to stress the fact that the critical points for preferring particular implementations occur at different values of C1.

II.7. Some remarks.

In this chapter algorithms for solving related storage structure selection problems were presented. The core of the algorithms is 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 types 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. It also shows that hill-climbing or branch-and-bound methods may not always be appropriate. As the problem size grows, the advantages of the algorithms presented become 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(where N**3 * log N), (where N denotes the number of substructures in the application (see, the minimal cost-to-time ratio cycle problem [Lawler76]).

CHAPTER III

III. Selection of reorganization points for storage structures.

In this chapter the problem of finding optimal reorganization points for a storage structure that deteriorates with time is addressed. A similar problem has been reported in the operation research literature under the heading of "equipment replacement" and a dynamic programming algorithm similar to the one presented here has been used for its solution (see [Dreyfus77, Chapter 2]). Consider the following typical behaviour: a database is created and its contents are organized in a manner convenient for efficient processing of queries and updates. As updates are made, however, performance of the storage structure degrades to a point at which reorganization is required or, at least, justified. For example, if the information is stored in a large sorted table, and a few new items are to be added, it may be convenient to enter the new data in an unordered auxiliary list temporarily, since insertion into the primary table would force the movement of many elements in the system. Direct insertion into the primary table would not only be expensive, but is very likely to be completely prohibitive for an online database system. Using an auxiliary list to collect a reasonable number of updates which can be merged into the primary table at some convenient time is a very attractive approach. An important question is, of course, "When should this merging occur?"

Although this example will be investigated further in subsequent sections, the main thrust of this chapter is to consider the more general problem of deciding when an arbitrary storage structure should be reorganized.

Consider the case, then, in which the demands to be made on a storage structure are reasonably predictable, at least for some fixed period of time. This predictability does not preclude situations in which the volume of data stored and the number of queries and updates vary widely in time. Such time-varying but predictable situations exist in practice, for example, in keeping track of the inventory of a holiday supplies shop or maintaining student records for an academic institution. As a consequence of the predictability, the cost of reorganizing a structure at any given point, as well as the cost of using it in either form can be determined. The problem is, then, to determine the _reorganization points_ so as to minimize the total cost of the operation. This total cost includes the _operating cost_ of using the storage structure (and thus implicitly the deterioration cost incurred by not restructuring) as well as the reorganization cost. Figure III.1 illustrates some of these and related concepts.

If the costs are linear then previous results may be applied to give a closed form solution [Shneiderman73, Tuel78]. The example of a sorted table with an auxiliary unordered list does not fit this linear criterion. Indeed, many "updating systems" are sub-linear in their deterioration cost. Unfortunately, when the reorganization or the deterioration costs are non-linear, no closed form solution is known for most cases.

In this chapter an algorithmic solution to the problem for arbitrary reorganization and deterioration costs is presented. The algorithm is similar to the one reported in Dreyfus and Law [Dreyfus77]. The basic concern is a storage structure whose performance degrades with the number of updates and for which there is associated a reorganization cost. The problem is to determine when to reorganize it to minimize the expected overall cost.

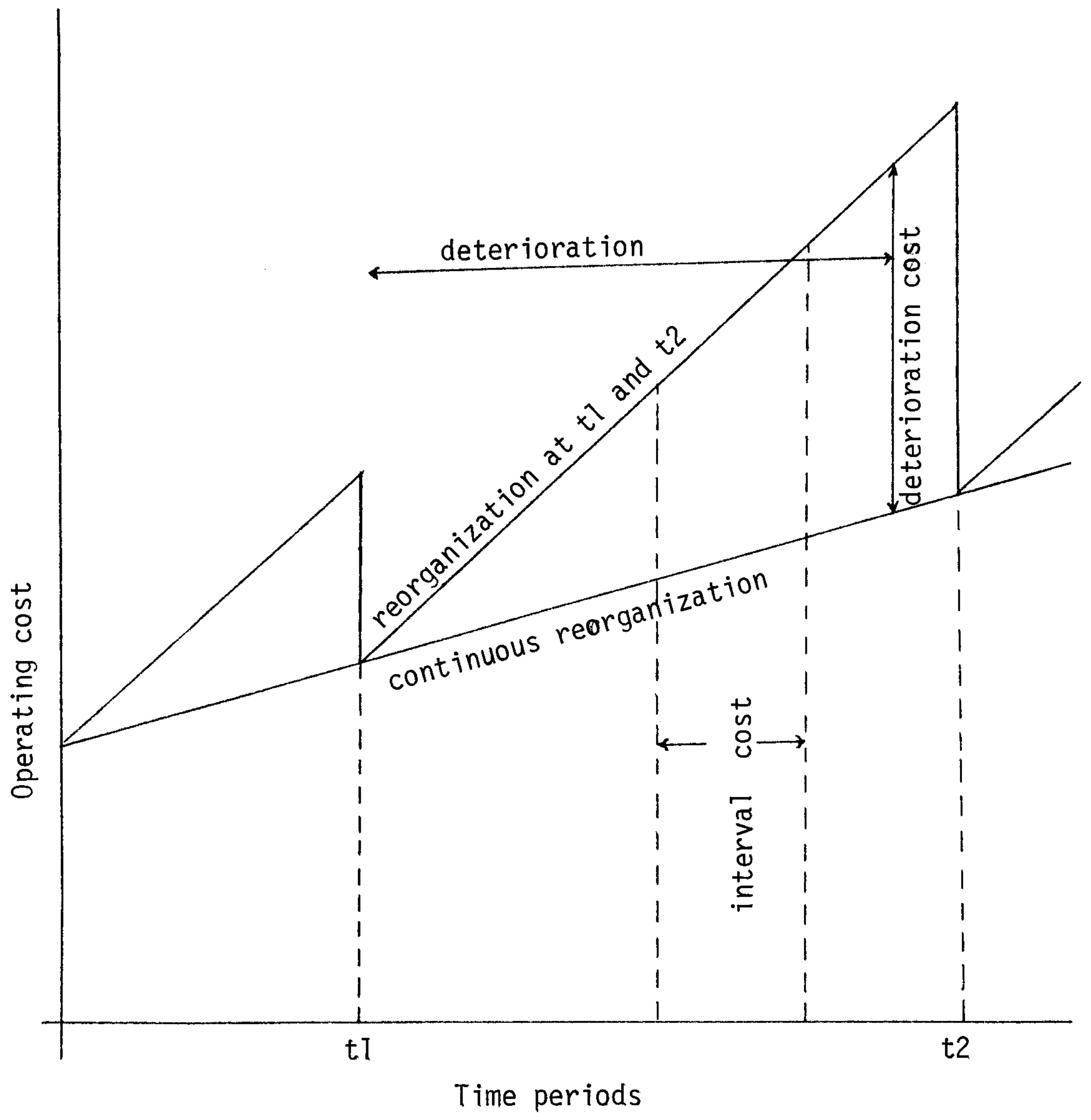

Figure III.1 Linear deterioration cost.

It is assumed that the remaining lifetime of the storage structure (i.e., the amount of time until the storage structure is no longer used, measured in hours, days, months, etc.). is quantized into P time periods (not necessarily of equal length). Furthermore if the storage structure is to be reorganized, it is assumed the process will take place at the beginning of a time period.

Enumeration of all possible sets of reorganization points requires the computation of 2**P costs, since at each period the decision whether or not to reorganize can be made independently. Such a computation is, of course, infeasible for P greater than 20 or 30.

The problem of determining optimal reorganization points for a storage structure can be identified with the problem of finding the shortest route in an acyclic network (i.e., finding the route of minimal cost from source to sink). Consider a grid of vertices at the non-negative integral points in a portion of the plane (Figure III.2). The x and y coordinates of each vertex denote respectively the time and the storage structure deterioration or the number of periods since the last reorganization.

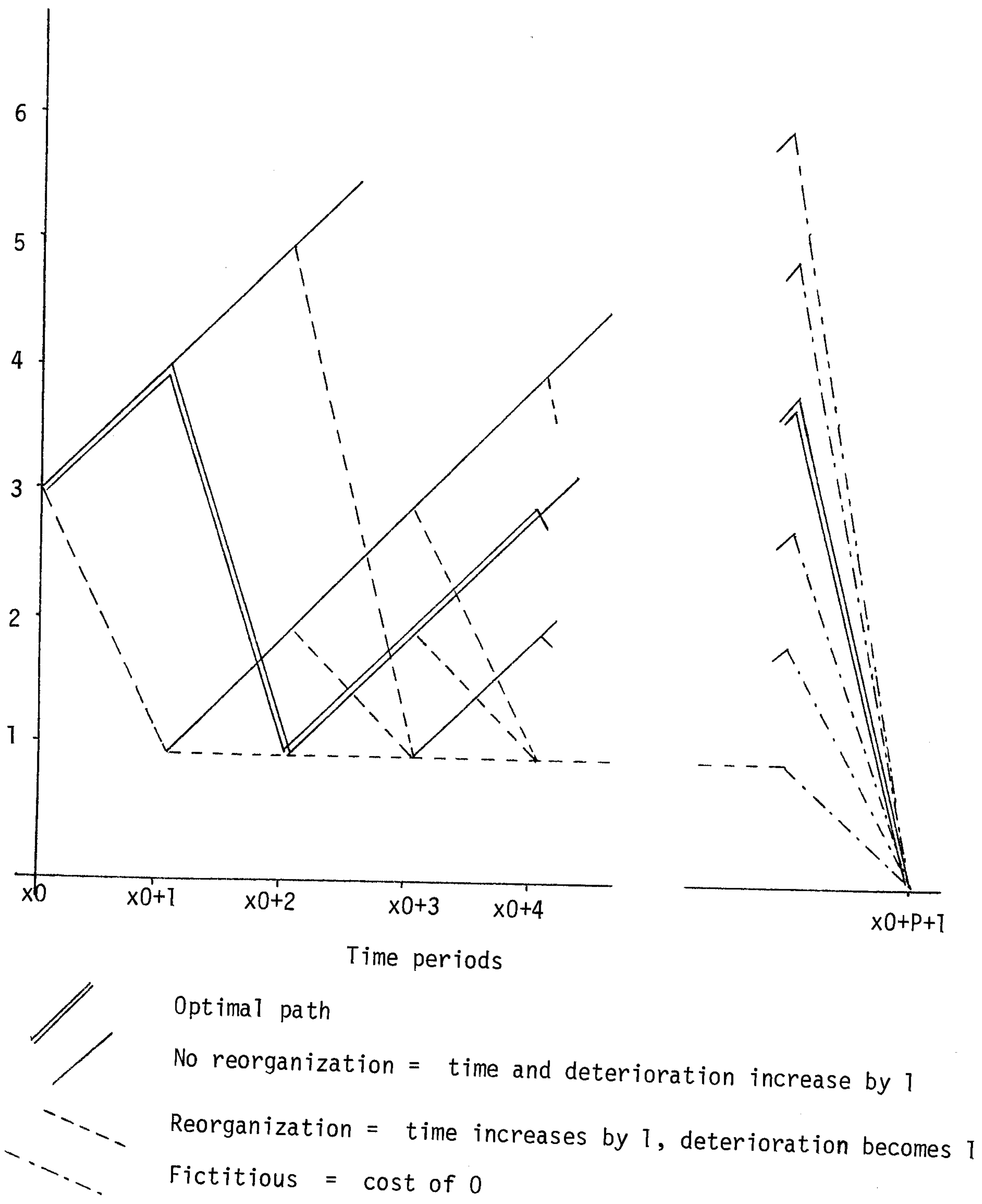
6
5
4
3
2
1
x0
x0+1
x0+2
x0+3
x0+4
x0+P+1
Time periods
Optimal path
No reorganization = time and deterioration increase by 1
Reorganization = time increases by 1, deterioration becomes 1
Fictitious = cost of 0

Figure III.2 Time vs. deterioration.

The source of the network is vertex (x0,y0), where x0 is the time at which the study begins (perhaps when the storage structure is formed), and y0 is the time since the last reorganization (y0 may well be 0). The decision not to reorganize the structure in condition (x,y) is denoted by an edge from (x,y) to vertex (x+1,y+1). The weight of this edge is the operational cost of running a system from time x to x+1 starting with deterioration y. The decision to reorganize corresponds to an edge from (x,y) to (x+1,1), with weight equal to the sum of the reorganization cost of a storage structure of deterioration y in period x and the operating cost of a newly reorganized storage structure in period x. If P is the storage structure lifetime, then every vertex with x-coordinate x0+P is connected by a zero-valued edge to the sink vertex (x0+P+1,0). As can be seen from the above description, the network is acyclic (i.e., no cycles are formed since time always moves to the right), and the solution is represented by the minimal cost route from vertex (x0,y0) to vertex (x0+P+1,0).

Section III.1 contains a dynamic programming formulation of the solution for this shortest route problem and a discussion of its optimality. In Section III.2 the algorithm is applied to a non-linear example. Section III.3 contains proofs of the optimality of the time and space required for this algorithm, and Section III.4 extends all the results to applications in which at the beginning of each period the storage structure can be partially reorganized to any of several levels at various costs. Section III.5 contains PASCAL code for the algorithms discussed.

III.1. An Efficient Solution.

The storage structure reorganization problem can be expressed in terms of dynamic programming as follows [Dreyfus77]:

Let d(x,y) denote the operating cost from period x to period x+1 of a storage structure with deterioration y at the beginning of the period,

r(x,y) denote the reorganization cost at the beginning of period x of a storage structure of deterioration y,

and F(x,y) denote the minimum cost to get to the state in which the storage structure has deterioration y at the beginning of period x, given that the process began period x0 with a storage structure whose deterioration was y0.

No assumptions (e.g., continuity, monotonicity, or even non-negativity) have been made for the above functions r(x,y) and d(x,y). In fact, the functions may be represented by arbitrary tables of discrete values (not necessarily equidistant in time) computed or estimated by monitoring, simulating, or analyzing the storage structure under consideration.

At the beginning of each period there is the option to reorganize the storage structure, and so, at the end of the period the deterioration of the storage structure could be 1 or it could be one more than at the beginning. From Figure III.2 it is apparent that the minimal cost expended from time x0 until some later time x is the minimal cost to reach period x-1 with deterioration y-1 plus either the reorganization cost plus the operating cost for the newly reorganized structure, or the operating cost in period x-1 of a storage structure of deterioration y-1.

This leads to the following recurrence relation for F:

$$F(x, 1) = \underset{y=2..x-1,\ y0+x-1}{\mathrm{MIN}} \quad F(x-1, y-1)+r(x-1, y-1)+d(x-1, 0)\}$$

and (III.1)

$$F(x, y) = F(x-1, y-1)+d(x-1, y-1)$$

for y=2..x-1, y0+x-1

The boundary condition is:

$$F(x0, y0) = 0 \qquad \text{(III.2)}$$

It is relatively straightforward to write a program to determine $F(x0+P+1, 0) = \min F(x0+P, y)$ over all choices of $y=1,2, .. P$ and $y0+P$: simply use the above recursion to determine the optimal cost for each time step and state of deterioration based on the optimal cost up to the previous time step. Note that the value of $F(x, y)$ need not be retained after $F(x+1, y+1)$ and $F(x+1, 1)$ have been computed. Hence at most P storage locations are required to retain these values. In fact, each arc in the network is inspected and used in some arithmetic computation once only, and thus if the values of $r(x, y)$ and $d(x, y)$ can be computed, only $\Theta(P)$ storage locations are needed to maintain the costs. Furthermore, since each arc is inspected only once, it immediately follows that $\Theta(P{*}{*}2)$ basic operations are performed. Note as well that the above recursion produces the shortest route from the source vertex $(x0, y0)$ to every other vertex in the network.

However, the real problem is not to discover the cost of the optimal reorganization scheme, but rather to determine the reorganization points that lead to that cost.

Taking a closer look at Figure III.2, it is realized that the only vertices that must store information regarding the optimal path are those with y-coordinate 1. The only way to reach the vertex (x, y) for $y \neq 1$ is through vertex (x-1, y-1), and so there is no need to store this information while determining the optimal path. With this observation in mind, and since there are only P vertices with y-coordinate 1, it follows that, if the values of r(x,y) and d(x,y) can be computed, only $\Theta(P)$ units of storage are required to determine the optimal reorganization scheme as well as its cost*. (Section III.5 contains an implementation of this implied algorithm written in PASCAL.)

Backward recursion, typically used in operations research circles, consists of solving a problem starting with the last stage and proceeding backwards toward stage 1 composing the solutions of each stage. Forward recursion computes the solution in the opposite direction, i.e., starting with the first stage and proceeding towards stage N.

* A function g(x) is $\Theta(f(x))$ if there exist constants c1, c2, and x0 such that c1*f(x) < g(x) < c2*f(x) for x>x0.

Equation III.1 uses forward recursion which is preferred for the solution of this problem: it usesonly Θ(P) storage cells whereas using backward recursion would require O(P**2) storage cells.

III.2. A Non-linear Application.

This section deals with the reorganization of a storage structure whose deterioration and reorganization costs are non-linear. In particular the application's deterioration cost is logarithmic (i.e., sub-linear), and the reorganization cost is super-linear.

Recall the example in the introduction to Section III in which one of the structures maintained by the application under consideration is an ordered table (i.e., a set of consecutive locations each containing one element, the elements to be kept in sorted order). Using binary search, the number of comparisons required to access a randomly designated element is essentially log(n), where n is the number of elements in the structure. Inserting a new element into the structure requires that a hole be created by shifting some elements down one position and then making the insertion into the newly vacated cell. Since this operation is expensive, it may be decided to keep the elements to be inserted in an unordered secondary list. If an element is not found in the primary table, the search continues with a sequential scan of the secondary list. Since, when looking for a randomly selected element, the primary table will always be searched and the secondary list will be searched in proportion to its relative size, the cost of accessing a particular element is $\Theta(\log(n-k) + k*k/n)$ where n is the total number of elements in both structures and k is the number of elements in the secondary list. In fact, for this example, the access cost used will be $\log(n-k+1)-1 + (1+k/2)*(k/n)$, the expected number of comparisons required.

If there are no deletions, the structure will grow. From time to time the elements in the secondary list may be merged with the ones in the ordered table (i.e., the structure will be reorganized) at a cost of $\Theta(k*\log(k) + n)$ operations: the $k*\log(k)$ term accounts for the average time required to sort k elements (e.g., using Quicksort), and $\Theta(n)$ operations are used in merging the two sorted lists.

In summary, assuming that there are a total of n elements, (n-k) in the ordered structure and k in the secondary structure, the following costs apply:

access cost: log(n-k+1)-1 + (1+k/2) * (k/n)

insertion cost: 1 (III.3)

reorganization cost: C1 * (k*log(k) + n)

where the parameter C1 is introduced solely to illustrate the effects of various related costs.

A structure that is continuously reorganized has an access cost of log(n+1) operations. Using these formulas and assuming that the primary table initially has 1000 elements and the secondary list is empty, that there are 5000 accesses and 100 insertions uniformly distributed in each interval, and that the lifetime (P) is 50 periods, the following results may be obtained from the algorithm presented in Section III.1:

C1	optimal reorganization points	optimal cost (in 10000's)
1	3,5,7,10,13,16,19,22,25 28,31,34,37,40,43,46	272
10	4,8,13,18,23,29,35,41	335
20	5,10,16,22,29,37	391
50	6,13,22,31	531
100	7,15,25	712
500	no reorganization	1225

Table III.1.

In previous work, optimal reorganization points were determined for linear costs only [Shneiderman73, Tuel78].

Thus one might be tempted to compute those points by approximating the costs by functions that are linear in k. A reasonable linear model of the behaviour of the system, derived by examining the optimal solution, is:

$$\begin{aligned}&\text{access cost: } \log(3500-K(Cl)+1)-1 + (1+k/2)*(K(Cl)/3500)\\ &\text{insertion cost: } 1 \qquad\qquad (III.4)\\ &\text{reorganization cost: } Cl * (k*\log(K(Cl)) + 3500)\end{aligned}$$

where 3500 is the average value of n and K(Cl) is the average number of probes for searching the secondary list when reorganization occurs under the optimal scheme.

The following results may be obtained under this linear model:

Cl	K(Cl)	reorganization points	actual cost (in 10000's)	ratio to optimal
1	147	1,2,3..49	281	1.033
10	278	2,4,6..48	383	1.143
20	357	3,5,7..47	495	1.266
50	550	3,6,9..45	704	1.326
100	625	4,8,12..44	1020	1.433
500	2500	5,10,14,18,22 26,30,34,38,42	3795	3.098

Table III.2.

The column labeled "actual cost" indicates the cost charged according to the formulae in III.3 and using the reorganization points suggested by this approximate solution. Comparing these results with the ones in Table III.1, it is seen that, even when knowledge of the optimal solution is used to derive the approximations, results based on assumptions of linear costs can give solutions which differ substantially from the optimal. Therefore the algorithm that permits the removal of all assumptions regarding the costs is a more desirable tool for storage structure administration.

III.3. Lower bounds.

In Section III.1 it was demonstrated that the optimal reorganization scheme can be determined in time quadratic in the number of potential reorganization points and space linear in this parameter. Of course, it is of interest to find whether or not a better algorithm exists.

It will be assumed throughout this section that r(x,y) and d(x,y) are computable rather than stored in tables of discrete values; otherwise it is obvious that Θ(T**2) space is required merely to store the algorithm's input.

That the space bound cannot be appreciably improved follows from the fact that the output (number of reorganization points used) may be of length Θ(T) *.

Intuitively, the quadratic time bound seems optimal as well, since there are Θ(T**2) potential situations for reorganization. The following theorem and its proof formalize this notion.

Theorem (R. Ramirez, J.I. Munro):

> Θ(T**2) operations and Θ(T) storage locations are necessary and sufficient to determine the optimal reorganization points even if the operation costs and the reorganization costs are known to be monotonically increasing as functions in the deterioration.

Proof:

> The space bounds and the sufficiency of the time bound follow from the algorithm presented and the observations above. To show that Θ(T**2) basic operations are necessary, it suffices to exhibit a case in which it is more or less irrelevant which reorganizations are done,

* A more intricate argument shows that even ignoring the space required to store the results, this bound still applies.

as long as a reorganization is performed at the particular time that allows the application to incur a "cheap" operation cost at a specific, but unknown node. That is, the shortest route must go through some unknown point (x, y) and any path through that point has the same cost. Since there are Θ(T**2) potential "cheap" edges, finding the crucial one requires that all edges be inspected. A scheme which is not strictly monotonic is outlined first. It is then modified slightly to achieve monotonicity.

Consider an application for which the operation and reorganization costs at each time step other than the last are 2. At the last time step the reorganization costs are also 2, but the simple operation costs are very large and all equal. Now alter an arbitrary operation cost of weight 2 to 1. Hence the optimal reorganization scheme must take advantage of this reduced cost; anything else which is done is irrelevant, proving the Θ(T**2) lower bound without the monotonicity assumption.

The weights can be arranged to be monotonically increasing functions of y by setting the operating costs to be d(x, y)=y, the reorganization costs for a storage structure of deterioration 1 to be r(x, 1)=2x and all other reorganization costs to be r(x, y)=2xy-(3/2)y**2-y/2. Reducing one arbitrary operating cost by 1/2 results in a system that establishes the Θ(T**2) lower bound.

Repeated application of this proof technique also leads to a bound on the time required to determine near optimal solutions.

Corollary : Θ(T**2/k) basic operations are necessary to determine a reorganization schedule which is within a factor of (1+1/k) of the optimal. This lower bound holds even if the operation and reorganization costs are monotonically increasing as functions of the the deterioration.

III.4. Partial Reorganization.

For some applications it is possible to have more than the simple choice of reorganizing or not at each stage. For example, there may be the option of several partial reorganization algorithms each transforming the storage structure to a different level of operation cost as well as having different reorganization cost. It is now necessary to know not only when to reorganize the storage structure but also which reorganization algorithm to use, in order to minimize the overall cost. This extension is also addressed by Dreyfus and Law [Dreyfus77, p. 29].

An extreme case is that in which it is possible to reorganize the storage structure of deterioration y to a level equivalent to that of any deterioration represented by an integer in the range 0 to y. Of course the reorganization and operating costs are again assumed to be arbitrary. A simple modification of the previously discussed algorithm to evaluate the O(P) reorganization alternatives at each step will take Θ(P**3) operations. In fact by using the same argument as in the previous case, it can be shown that Θ(P**3) operations are necessary for any algorithm solving the problem.

Unfortunately, if the algorithm is implemented as outlined in Section III.1, at least Θ(P**2) storage cells are necessary to determine the reorganization points and levels even if the reorganization and operating costs are computable. From Figure III.3, it is clear that virtually every node may be the target of one or more reorganizations in the previous time period.

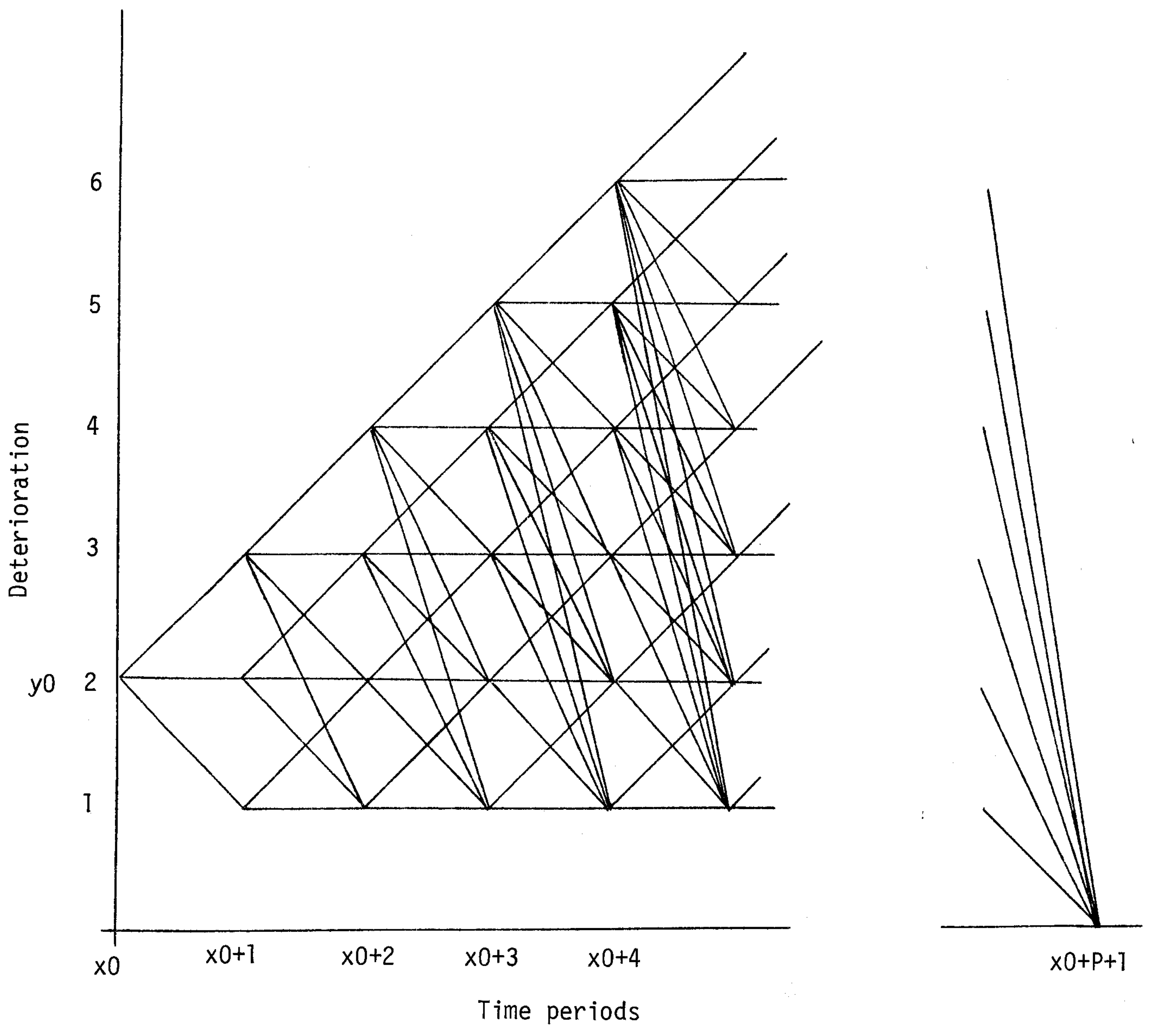

Figure III.3 Partial reorganization

Thus, unlike the situation described in Section III.1, for <u>each</u> of the O(P) values of F for a given time, O(P) reorganization points may have to be stored. When P is large this storage requirement is at best annoying and at worst prohibitive.

The following modification of the basic scheme, similar in nature to that of Section II.3.2, permits a solution using only $\Theta(P)$ space while (roughly) doubling the previous time bound. Again in discussing the possibility of linear space, it is assumed that the deterioration and reorganization costs are computable.

It has been noted in Section III.1 that storing the optimal path to the point (x,y) is not essential for computing the length of the shortest path, but rather it is used only to reconstruct the shortest path itself. Because of the optimality of the shortest route, if follows that if the shortest route from (x0, y0) to (x0+P+1, 0) passes through points (x1, y1) and (x2, y2), then the shortest route from (x1, y1) to (x2, y2) must coincide with the shortest path for the complete problem in the [(x1, y1), (x2, y2)] interval. Consequently the optimal reorganization points for the subproblem are the same as those for the original problem (in the [x1, x2] interval).

First consider solving the original problem as if only the cost of the shortest path were required and not the path itself. Now, at the midpoint (i.e., x0+P/2) label each node in the period by its y-coordinate, i.e., a value from {1, 2,..,y0+P/2}. From time period (x0+P/2)+1 to the end of the lifetime, record for each node the node through which the route passed at period x0+P/2. In fact the values can be recorded as they are carried forward during the computation of the cost of the shortest route. When time x0+P+1 is reached and the value of the shortest route computed, the mid-node, y', that this (optimal) route passed through will be known. Since at any time period there are only $\Theta(P)$ nodes, it follows that only $\Theta(P)$ space is required for forwarding mid-node identification.

Once the node (x0+P/2, y') is known, solve the following two subproblems (recursively):

(i) find the optimal reorganization points for a storage structure starting period x0 with deterioration y0 and running until time x0+P/2, with the constraint that it must go through y' at the last time step

and,

(ii) find the optimal reorganization points for a storage structure starting period x0+P/2 with deterioration y' and having a lifetime of P/2 periods.

The first subproblem is equivalent to finding the set of optimal partial reorganization points of a storage structure starting period x0 with deterioration y0 and terminating period x0+T/2 with deterioration y'. The second subproblem is identical to the original, except that it is half the size and starts with a storage structure of deterioration y' at period x0+P/2, and thus the original algorithm can be applied without alteration. (Section III.5 contains an implementation of this method in PASCAL.)

The recursive application of this "divide-and-conquer" technique will produce the set of optimal reorganization points for the original problem. It remains to show that the application of this technique will conserve the Θ(P**3) time bound. Let c*P**3 denote the number of basic operations required to find the <u>cost</u> of the optimal arrangement by the dynamic programming scheme first proposed, and let Φ(P) represent the computation time required in the worst case for the above scheme. Then, ignoring the minor cost of recursive calls and some trivial pointer operations,

$$\Phi\ (P) = c*P**3 + 2*\Phi(P/2) \quad \text{for } P\geq 2 \qquad \text{(III.5)}$$

and suppose

$$\Phi (1) = 1 \qquad \text{(III.6)}$$

As a result Φ(P) $\simeq$ 2*c*P**3, that is, the time to find the optimal reorganization points is approximately twice that required to find the minimum cost alone and so is still Θ(P**3). If the scheme maintained the 1/3 and 2/3 positions (rather than the midpoint) of the optimal path and carried them through on the first pass, the running time of the algorithm would be roughly 3/2 that of the basic scheme, but the space requirement, although still Θ(P) would be noticeably greater than for the method outlined. It is straightforward to develop this time-space trade-off for maintaining any fraction of the points.

III.5 The algorithms.

This section presents the PASCAL code for the algorithms discussed in the previous sections of Chapter III.

Algorithm for selecting optimal reorganization points.

```
function shortest (x0, y0, p : integer) : real;
var fl, ftop, fopt : real;
       x, y, yopt, who : integer;
       from : array [0..P] of integer;
       f : array [1..P] of real;
   begin
   ftop := 0; from[x0]:=TOP;  {boundary condition}
   for x := 2 to p do
      begin
      fl := r(x0+x-2,x+y0-2) + d(x0+x-2,0) + ftop;
      from[x0+x-1]:=TOP;
      ftop := d(x0+x-2,x+y0-2) + ftop;
      for y := x-2 downto 1 do
         begin
         f[y+1] := d(x0+x-2,y)+f[y];
         fl := min(fl, r(x0+x-2,y)+d(x0+x-2,0)+f[y], who);
         if who = 2 then from[x0+x-1]:=y;
         end;
      f[1] := fl;
      end;
      { find optimal value }
      fopt:=ftop; yopt:=TOP;
      for y:=1 to p-1 do
         begin
         fopt:=min(fopt, f[y], who);
         if who = 2 then yopt:=y;
         end;
      shortest:=fopt;
      {retrace optimal path}
      while yopt <> TOP do
         begin
         p:=p-yopp;
         writeln('reorganize at stage',p);
         yopt:=from[x0+p+1];
         end
   end;
```

Algorithm for selecting partial reorganization points.

```
var   f : array [0..PY0] of real;
   half : array [0..PY0] of integer;
function shortest(x0, y0, p, yprime : integer;
                  forced : boolean) : real;
   var ft, oft : real;
       x, y, z, halft, ohalft, yp, who : integer;
   begin
   for y:=0 to p div 2 + y0 do half[y]:=y;
   for y:=0 to y0 do f[y]:=INFINITE;
   f[y0]:=0; oft:=0; halft:=0;
   for x:=x0+1 to x0+p do
      begin
      for y:=1 to x-x0+y0 do
         begin
         if y = 1 then ft:=INFINITE
         else ft:=d(x-1,y-1)+f[y-1];
         f[y-1]:=oft;
         if x-x0 > p div 2 then
            begin
            halft:=half[y-1]; half[y-1]:=ohalft;
            end;
         for z:=y to x-x0+y0-1 do
            begin
            ft:=min(ft,r(x-1,z,y)+d(x-1,y-1)+f[z],who);
            if (x-x0 > p div 2) and (who = 2)
               then halft:=half[z];
            end;
         oft:=ft; ohalft:=halft;
         end;
      f[x-x0+y0]:=ft;
      if x-x0 > p div 2 then half[x-x0+y0]:=halft;
      end;
   if forced then
      begin
      shortest:=f[yprime]; yp:=half[yprime];
      end
   else begin
      ft:=f[1]; yp:=half[1]; yprime:=1;
      for y:=2 to p+y0-1 do
         begin
         ft:=min(ft,f[y],who);
         if who = 2 then begin yp:=half[y]; yprime:=y end
         end;
      shortest:=ft;
      end;
   if p = 2 then
      begin
      writeln('stage',x0,' from',y0,' to',yp);
      writeln('stage',x0+1,' from',yp,' to',yprime);
      end
      else if p = 3 then
           writeln('stage',x0,' from',y0,' to',yp);
   if p >= 4 { p div 2 >= 2}
      then ft:=shortest(x0,y0,p div 2, yp, true);
   if p >= 3 { p div 2 >= 2}
      then
       ft:=shortest(x0+p div 2, yp,p-p div 2,yprime,true);
end;
```

III.6. Some remarks.

It has been shown that Θ(P**2) basic operations and Θ (P) storage locations are necessary and sufficient to compute the reorganization points for arbitrary or for monotonic costs, where P is the storage structure lifetime. Furthermore, it has also been shown that Θ(P**3) basic operations and Θ(P) space are required to compute partial reorganization points. The space-saving divide-and-conquer technique presented in Section III.4 is applicable to any shortest route path problem in which the weights of edges are computable and thus do not have to be stored explicitly.

For some applications, reorganizing the storage structure may imply that all users must be locked out during the reorganization period. A possible minor extension to the algorithm is to compute reorganization points optimally given a limit on the maximum number of reorganizations and/or the total reorganization time for a given time interval P (and thus guaranteeing a minimal availability for the storage structure).

The major limitation of this algorithmic approach is its dependence on a discretized, finite storage structure lifetime. There exist some special cases for which the algorithm could be modified to handle unbounded lifetime, for example when the deterioration and reorganization costs are identical in every stage after some time P or when they are periodic after P.

It should be noted that when the reorganization and deterioration costs are linear, Tuel's closed form solution [Tuel78] is to be preferred to any algorithm since it requires virtually no computation. Similarly, if other closed forms can be found for particular cases, they should be preferred as well; unfortunately no work has been reported other than for the linear case. Therefore, the simplicity, universality, and practicality of this reorganization algorithm make it a worthwhile tool for storage structure or data structure designers.

CHAPTER IV

IV. Linking selections of composite storage structures.

Previous chapters dealt with the selection of unchanging storage structures. 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, as mentioned in Section II.5, 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 previous 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 application represented by application IV.1, for which every combination of implementations is assumed to be feasible. 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 is 120 cost units.

If the selection algorithm of Section II.5 is applied to this example (i.e., it is assumed that the best selection was adopted for phase 1) and it is now desired to find the best selection for phase 2, that algorithm will select implementation 1 for phase 2, with a combined cost of 40 units. This is clearly better because a local improvement is achieved.

When the two phases are considered simultaneously, (a total optimization) 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.

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

Data for Example IV.1.

IV.1. Mathematical formulation for the selection of a sequence of composite storage structures problem.

The selection of a sequence of storage structures problem can be described as a generalization of the selection problem discussed in previous chapters as follows:

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,

$X(p)$	a zero-one matrix in which $x(i,j,p)$ indicates whether or not implementation j is to be selected for substructure i in phase p,
$s(i,j,p)$	the estimated storage space consumed by implementation j when used for substructure i in phase p,
$t(i,j,p)$	the estimated run time of implementation j when used for substructure i in phase p,
$c(i,j,p,j')$	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,
$COST(X(p),S(p),T(p))$	a monotonic cost function in terms of the total amount of time consumed by the final selection $X(p)$ when constrained to the bound $S(p)$ and $T(p)$ in phase p.

The problem can now be represented as:

$$Z = \text{MIN} \sum_{p=1}^{P} \{ \text{COST}(X(p),S(p),T(p)) \ + \tag{IV.1}$$

$$K *(\sum_{i=1}^{N} \sum_{j=1}^{M(i)} \sum_{j'=1}^{M(i)} c(i,j,p,j')*x(i,j,p)*x(i,j',p+1)\}$$

such that:

$$\sum_{j=1}^{M(i)} x(i,j,p) = 1 \quad \forall i=1..N,\ p=1..P \tag{IV.2}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} s(i,j,p)*x(i,j,p) \leq S(p) \ \forall p=1..P \tag{IV.3}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} t(i,j,p)*x(i,j,p) \leq T(p) \ \forall p=1..P \tag{IV.4}$$

$$x(i,j,p) = 0,\ 1 \ \forall i=1..N,\ j=1..M(i),\ p=1..P \tag{IV.5}$$

$$K \in R+ \tag{IV.6}$$

The last expression of Equation IV.1 accounts for the conversion cost between phases, since the conversion cost c(i,j,p,j') applies only when both zero-one variables are one. The other equations are merely generalizations of the ones presented in previous chapters.

IV.2. Linking selections involving one substructure at each phase.

Consider the problem in which there exists only one substructure in each phase of the application. Thus for each phase there are many possible implementations for this substructure, and the problem consists of selecting one implementation for each phase such that the total cost is minimized. The solution for this simplified problem is instructive in understanding one solution for the general one.

Consider the graph in Figure IV.1, each node (except for nodes S and E) represent one possible implementation for the substructure under consideration. Thus each phase of the application is composed of M(i) nodes (a column in the graph), each node (except for node E) is connected to all the nodes immediately to its right (next column) by means of directed edges. Each edge of the graph has a weight that corresponds to the cost of using the implementation at the target of the edge in the corresponding phase plus the conversion cost to that particular implementation.

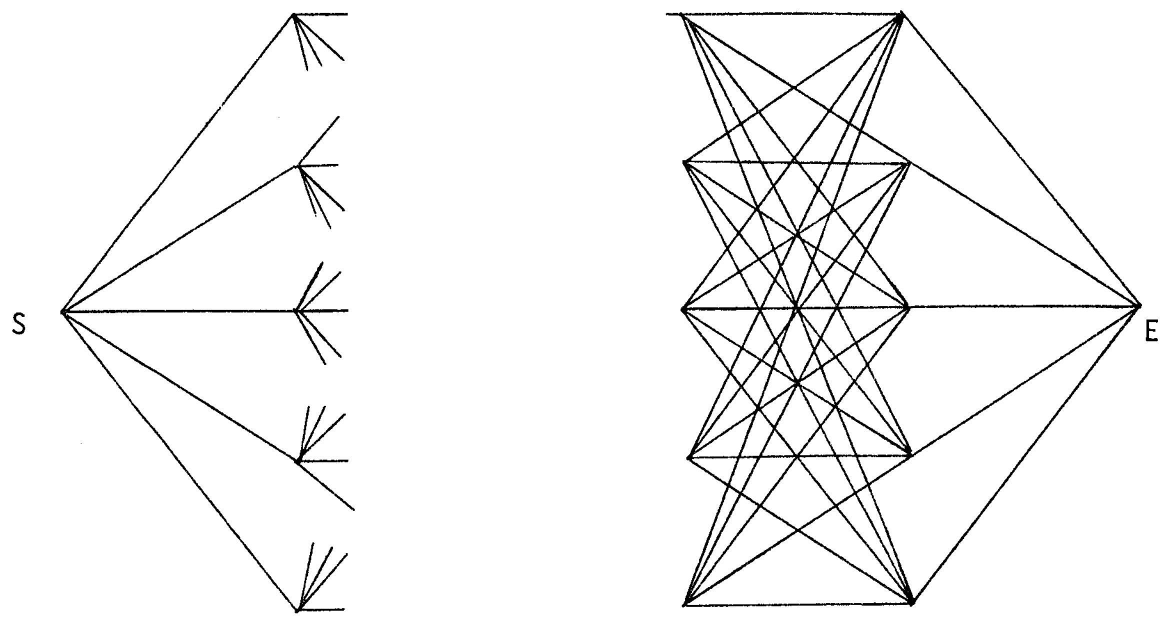

Figure IV.1 Linking selections involving one substructure at each phase.

Since nodes S and E are special, the edges departing from node S are weighted by the cost of using the implementation at the target of the edge only, and all the edges pointing to node E have a weight of zero.

Given that the total cost for this problem is additive, (Equation IV.1 when N=1) each phase contributes a piece of the total cost. Therefore, the problem has been reduced to a shortest route problem from node S to node E. This problem has been investigated in Chapter III under the heading of optimum reorganization points for data structures, and consequently the algorithms developed in that chapter are applicable here as well.

IV.3. Linking selections involving several substructures at each phase.

The problem of linking P composite storage structure selections can be represented graphically in 3-dimensions (Figure IV.2) such that the x-coordinate represents the substructures for which an assignment is sought, the y-coordinate represents the possible implementations for each one of the substructures, and the z-coordinate represents the phases in the lifetime of the application.

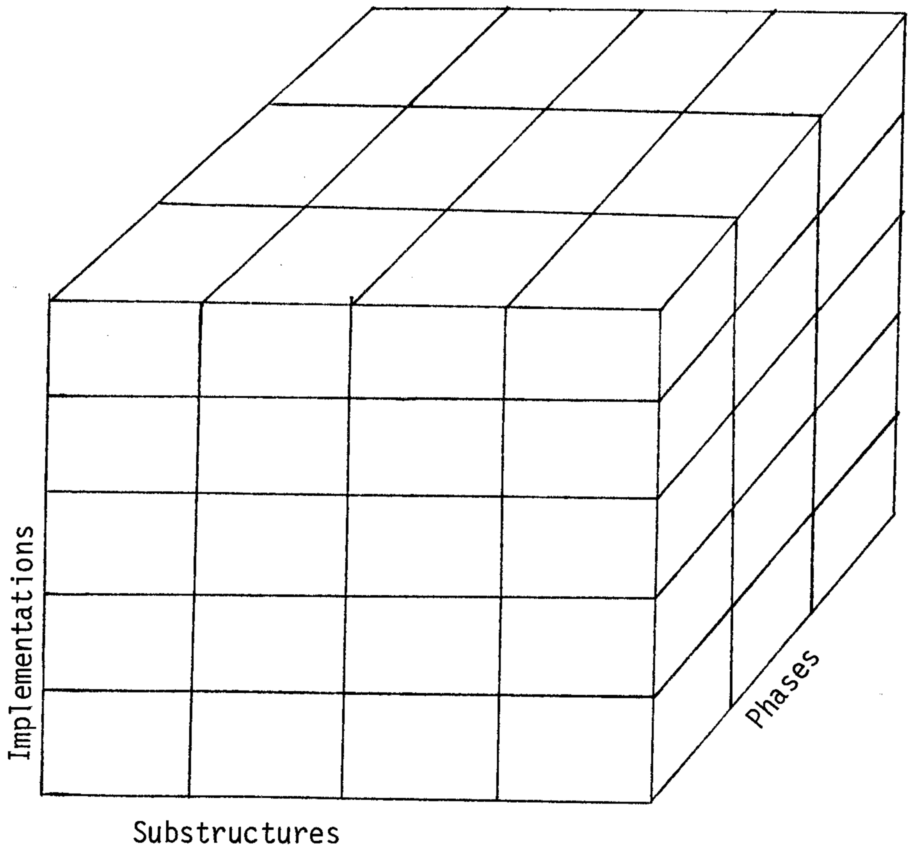

Figure IV.2 Linking selections involving N substructures at each phase.

Thus, each x-y plane characterizes one phase of the application, and is related to the plane behind it by means of conversion costs between the phases.

The problem consists now in selecting for each plane, a set of cells, one from each column, such that the cost of the selection at each plane plus the conversion cost between planes is as small as possible.

Observing the solution for the problem of the previous section, in which only one substructure was present at any one phase, one can devise a solution algorithm for this problem as well. In particular if one had all complete assignments for each phase, i.e., those that do not violate the given constraints, one could apply a shortest route algorithm in the following manner:

Each node in the graph of Figure IV.1 (except S and E) represents one feasible selection (rather than just one possible implementation for a substructure as in the previous section). A column in the graph consequently constitutes all feasible selections for one phase of the application. Each node is connected via directed edges to all the nodes in the next column (phase) of the graph. The label (weight) of each edge corresponds to the cost of using the selection of implementations at the target of the edge in the corresponding phase plus the sum of the conversion costs to that particular set of implementations from the node where this edge originates. Again, since nodes S and E are special, the edges departing from node S are labeled by the cost of using the set of implementations pointed by the edge only, and all the edges pointing to node E will have a label of zero. (Given that every edge in the graph uniquely determines the set of implementations at each phase, their labels can be pre-computed prior to the execution of any algorithm.)

Thus, the problem has again been reduced to a shortest route problem from node S to node E. However, the number of feasible solutions for each phase may be exponential i.e., O(M**N). Therefore, in this case the time complexity of the algorithm will be exponential in the number of library implementations. In fact the time complexity in the worst case is O((P-1)*(M**(2*N))).

The following theorem and its proof establishes that, unless P = NP, no polynomial or pseudo-polynomial algorithm exists for selecting a sequence of composite storage structures.

Theorem:

The selection of a sequence of composite 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 composite storage structures problem.

The satisfiability problem can be stated as follows:

Given: set U of Boolean variables, collection C of clauses in conjunctive normal form.

Question: 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|

$$s(i,1,k) = \begin{cases} 0 & \text{if variable } U(i) \text{ is in clause } C(k) \\ 1 & \text{if } U(i) \text{ is not in } C(k) \end{cases}$$

$$s(i,2,k) = \begin{cases} 0 & \text{if the negation of } U(i)\ (\tilde{}U(i)) \text{ is in } C(k) \\ 1 & \text{if } \tilde{}U(i) \text{ is not in } C(k) \end{cases}$$

$$t(i,j,k) = 0$$

$$c(i,j,k,j') = \begin{cases} 0 & \text{if } j = j' \\ 1 & \text{otherwise} \end{cases}$$

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

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

$$Z = \text{MIN} \sum_{k=1}^{P} \left\{ \sum_{i=1}^{N} \sum_{j=1}^{2} t(i,j,k)*x(i,j,k) + \sum_{i=1}^{N} \sum_{j=1}^{2} \sum_{j'=1}^{2} c(i,j,k,j')*x(i,j,k)*x(i,j',k+1) \right\}$$

such that:

$$\sum_{i=1}^{N} \sum_{j=1}^{2} s(i,j,k)*x(i,j,k) \leq N-1 \quad \forall k=1..P$$

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).

Given that the input for this selection problem consisted only of zeroes and ones (there is no need to enter the number N, just count the number of zeroes and ones) the selection of a sequence of composite storage structures is NP-complete in the strong sense [Garey79].

IV.4. Transformation of a P-phase selection problem into a 1-phase problem.

In this section the transformation of the problem of selecting a sequence of P composite storage structures problem into a 1-phase selection problem is investigated. The virtue of such a transformation is that the algorithms presented in Chapter II can be used with only slight changes.

To simplify the discussion, consider a 2-phase problem in which there are N substructures for which an assignment is sought. For each substructure i, there are M(i) library implementations. The problem is characterized by two evaluation matrices (one for each phase) and by N conversion cost matrices (one for each substructure).

The transformation is indeed very simple. For each implementation of substructure i in phase 1, construct a vector segment by pairing it with all the implementations for that substructure in phase 2. After all the implementations for the substructure in phase 1 have been considered, a vector of size M(i)*M(i) will be generated. Each entry of the vector will be characterized by a five tuple

<s(i,j,1),t(i,j,1),s(i,j',2),t(i,j',2),c(i,j,1,j')>

identifying the pairing of the implementations for that substructure. That is, implementation j in phase 1 using s(i,j,1) space, t(i,j,1) time and implementation j' in phase 2 using s(i,j',2) space, t(i,j',2) time, the conversion cost between the two given by c(i,j,1,j').

Once this transformation is performed on all substructures, an evaluation matrix of size N*(M**2) will have been formed. Utilizing this evaluation matrix, the following problem should be solved:

$$Z = \text{Min} \{ \text{COST}(X(1),S(1),T(1)) + \text{COST}(X(2),S(2),T(2)) + K * \sum_{i=1}^{N} \sum_{j=1}^{M(i)} \sum_{j'=1}^{M(i)} c(i,j,1,j')*x(i,j,1)*x(i,j',2) \} \quad \text{(IV.7)}$$

under the following constraints:

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} s(i,j,1)*x(i,j,1) \leq S(1) \quad \text{(IV.8)}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M(i)} t(i,j,1)*x(i,j,1) \leq T(1) \quad \text{(IV.9)}$$

$$\sum_{j=1}^{M(i)} x(i,j,1) = 1 \quad \forall i=1..N \quad \text{(IV.10)}$$

$$\sum_{i=1}^{N} \sum_{j'=1}^{M(i)} s(i,j',2)*x(i,j',2) \leq S(2) \quad \text{(IV.11)}$$

$$\sum_{i=1}^{N} \sum_{j'=1}^{M(i)} t(i,j',2)*x(i,j',2) \leq T(2) \quad \text{(IV.12)}$$

$$\sum_{j'=1}^{M(i)} x(i,j',2) = 1 \quad \forall i=1..N \quad \text{(IV.13)}$$

$$x(i,j,1),\ x(i,k,2) = 0,1 \quad \forall i=1..N,\ j=1..M(i),\ k=1..M(i) \quad \text{(IV.14)}$$

This selection problem is similar to the one of Section II.5, except that four state variables (rather than just two) are required for its solution.

The following recursive relationships can be derived:

$$F(i,s(1),t(1),s(2),t(2)) = \begin{cases} \underset{j,k}{\mathrm{MIN}} \{c(i,j,1,k)+F(i+1,s(1)-s(i,j,1),t(1)-t(i,j,1), \\ \qquad s(2)-s(i,k,2),t(2)-t(i,k,2))\} \\ \qquad \forall j,k \text{ such that } F(i+1,s(1)-s(i,j,1),t(1)-t(i,j,1), \\ \qquad s(2)-s(i,k,2),t(2)-t(i,k,2)) \text{ exist} \\ \text{infinite otherwise} \end{cases} \tag{IV.15}$$

the boundary condition is given by:

$$F(N,s(1),t(1),s(2),t(2)) = \begin{cases} \underset{j,k}{\mathrm{MIN}} \{c(N,j,1,k)\} \quad \forall j,k \text{ such that} \\ \qquad s(1)=s(N,j,1),t(1)=t(N,j,1), \\ \qquad s(2)=s(N,k,2),t(2)=t(N,k,2) \\ \text{infinite otherwise} \end{cases} \tag{IV.16}$$

the solution will be found by:

$$Z = \underset{\substack{s(1),t(1) \\ s(2),t(2)}}{\mathrm{MIN}} \{F(1,s(1),t(1),s(2),t(2))+f(s(1),t(1),s(2),t(2))\} \tag{IV.17}$$

Where f represents the value of COST(X,S,T) for a specific pair of assignments. Clearly F(1,s(1),t(1),s(2),t(2)) will be finite for all feasible solutions that use exactly s(1) space and t(1) time in phase 1, and exactly s(2) space and t(2) time in phase 2.

The implementation of the above recursive relationship will require O(N*(M*S*T)**2) operations, given that for each stage, there are at most S(1)*S(2)*T(1)*T(2) possible states (combinations of space and time at each of the two phases), and since the value of each state is selected from at most M(i)*M(i) alternatives. The space required is O(N*(S*T)**2) storage cells, because the outcome of each state must be stored.

Clearly similar bounds to those of Section II.3.1 can be applied to each of the state variables of this recursion, thus reducing the required time in practice. Furthermore, if the functions COST(X(p),S(p),T(p)) were separable, the approach of Section II.3.2 will reduce the space complexity to O((S*T)**2).

If the transformation just described is applied to a P-phase problem, i.e., by collapsing P implementations into a single vector entry, an algorithm whose worst time complexity will be of O(N*(M*S*T)**P) will be obtained. This perhaps is more desirable to the approach of Section IV.3, since it is expected that the number of phases will be smaller than the number of substructures. This solution, however, is still exponential, now in the number of phases.

IV.5 An example.

This section illustrates the application of the algorithm presented in Section IV.4, namely, the transformation of selecting a sequence of P composite storage structures into a 1-phase selection problem. The example under consideration is 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:	out of 5000 possible distinct values, at most 3000 are expected,
attribute 2:	out of 1000 possible distincts values, at most 1000 are expected,

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

Assume as well that in the application making use of this file, one can distinguish two phases. In the initial one, namely when the file is created, there will be a relatively high number of insertions as compared to the number of queries. In the second phase, the number of insertions decreases and the number of updates increases. For this example, 90% of the directory insertions and 10% of the directory queries are assumed to be performed in the initial phase, and for the second phase, 10% of the insertions and 90% of the queries will be performed.

The library of implementations used in this example consist of:

linearly addressed store

contiguous store

unary chain

binary tree

threaded tree

Under the assumptions just described and using the formulas produced by Tompa [Tompa74], the following evaluation and conversion cost matrices may be obtained.

	TIME			SPACE		
	A1	A2	A3	A1	A2	A3
lin. add.	.16	.06	.02	10	2	1
cont. st.	59.05	6.84	.17	5.4	1.8	.18
unary ch.	41.34	6.90	.22	4	1.4	.14
bin. tree	1.52	.62	.12	5.4	1.8	.18
thr. tree	1.52	.65	.12	5.4	1.8	.18

Table IV.1 Evaluation matrix for the initial phase.

(time is in seconds, space is in kilo-words)

	TIME			SPACE		
	A1	A2	A3	A1	A2	A3
lin. add.	.36	.30	.15	10	2	1.
cont. st.	15.16	2.82	.74	6	2	.29
unary ch.	199.82	61.67	3.33	8.5	2.9	.29
bin. tree	3.53	2.95	1.01	11.4	3.8	.38
thr. tree	3.47	2.77	1.	11.4	3.8	.38

Table IV.2 Evaluation matrix for the second phase.

(time is in seconds, space is in kilo-words)

From To	lin. add.	cont. st.	unary ch.	bin. tree	thr. tree
lin. add	0	.34	.52	.63	.71
cont. st	.30	0	.49	.58	.67
unary ch.	.29	.29	0	.56	.65
bin. tree	.47	.47	.65	0	.46
thr. tree	.35	.35	.53	.12	0

Table IV.3 Conversion costs for attribute 1

(time is in seconds)

From To	lin. add.	cont. st.	unary ch.	bin.tree	thr. tree
lin. add.	0	.11	.17	.21	.24
cont. st.	.10	0	.16	.20	.22
unary ch.	.10	.19	0	.19	.22
bin. tree	.19	.19	.28	0	.15
thr. tree	.12	.12	.18	.04	0

Table IV.4 Conversion costs for attribute 2.

(time is in seconds)

From To	lin. add.	cont. st.	unary ch.	bin. tree	thr. tree
lin. add.	0	.01	.02	.02	.02
cont. st.	.01	0	.02	.02	.02
unary ch.	.01	.01	0	.02	.02
bin. tree	.02	.02	.03	0	.02
thr. tree	.01	.01	.02	.01	0

Table IV.5 Conversion costs for attribute 3.

(time is in seconds)

In order to apply the algorithms suggested in Section IV.4, it is necessary to create an evaluation matrix of size 3*25, by pairing each implementation for each substructure in the initial phase with all the implementations for that substructure in the final phase. As mentioned in that section, each matrix is a five tuple that identifies the pairing of implementations for that substructure.

The total number of possible selections in this example is (5**3)**2 = 15625, making manual calculation and analysis virtually impossible. The cost formula assumed for this example is given by:

$$
\begin{aligned}
&(\textstyle\sum \text{spaces} * \sum \text{times}) \text{ in phase 1} + \\
&\text{MAX}(\textstyle\sum \text{spaces in phase 1}, \sum \text{spaces in phase 2}) \\
&\qquad * (\textstyle\sum \text{conversion times}) + \\
&(\textstyle\sum \text{spaces} * \sum \text{times}) \text{ in phase 2}
\end{aligned}
$$

The first step of the algorithm consists of applying the boundary condition IV.16, i.e., compute the optimal value function for a problem of size one. This optimal value function is characterized by a vector of size 25 for this example, i.e., one entry for each possible sequence of representations for substructure 1.

A problem of size two can now be solved by applying the recursive relationship IV.15, producing a vector of size at most 625. The phenomenon of several implementations collapsing as described in Chapter II will also occur for this procedure. Finally, using the previous vector and the same recursive relationship, the complete problem is solved.

If no restrictions are imposed on the total amount of space or time at each phase, and the total conversion cost is only the sum of the individual costs (i.e., not charging for the space consumed by the conversion algorithms), the algorithm will select the linearly addressed store for all three substructures in both phases with a total cost of 13.65 kiloword-seconds. This is because the linearly addressed store is very efficient in time and the extensive use of space it requires is not heavily penalized by the cost formula.

However, if the total conversion cost is given by the product of the sum of the times and the sum of the spaces (as suggested in the above cost formula) and space and time bounds are imposed at each of the phases, interesting alternatives arise. For example, it may be required that the accumulated response from the directories be bounded by 60 seconds and that 6 kilowords and 10 kilowords are available for the initial and the final phase respectively. As a result the algorithm selects unary chains for phase 1 and contiguous store for phase 2 to be used for both attributes 1 and 3, whereas for attribute 2 threaded tree should be used in phase in phase 1 and this should be converted to linearly addressed store for phase 2. The total cost of this selection is 388.507 kiloword-seconds and the solution is within the required bounds.

IV.6 Some remarks.

In this chapter two approaches for selecting a sequence of composite storage structures were presented. The former is based on enumerating the possible selections for each one of the phases, and then solving a shortest route problem among the phases. The latter is based on transforming a P-phase selection problem into a 1-phase problem. Both approaches lead to algorithms whose running time and/or required space is exponential, in the first case in the number of substructures for which a selection is required and the second in the number of phases in the problem. Consequently, when both of these variables grow, the applicability of these algorithms is questionable.

However, for applications in which the conversion costs between phases is relatively small compared with the cost of the phases, each phase can be solved independently by using one of the algorithms presented in Chapter II, i.e., a set of independent composite storage structures selection problems. Conversely, when the cost of the phases is relatively small compared with the conversion costs, each substructure can be solved independently using one of the algorithms presented in Chapter III, i.e., a set of independent reorganization points problems. Therefore, if one of the above characteristics hold, one can in practice obtain approximate solutions to problems of large size.

The algorithms presented in this chapter assumed that no bounds on space and/or time are present when the conversion algorithms are executed between the phases. One possible extension will be to constrain the conversion algorithms to specified bounds between phases. Another extension will be to incorporate the possibility that at each phase there are a different number of substructures or implementations than in the previous.

CHAPTER V

V. Conclusions.

V.1 Thesis contributions.

The objective of this thesis has been to motivate and solve several related problems concerning the selection of storage structures.

Although, other researchers have presented solutions for a simple version of the problem, the methods previously proposed have had exponential worst case running time (in the number of substructures and implementations) and/or they have not guaranteed an optimal selection. Consequently, the thrust of this thesis was to extend previously known algorithms for solving this type of problem to incorporate bounds on the available resources as well as being pseudo-polynomial in their running time. Moreover, given that there exist natural bounds for these resources (e.g., the size of the primary of secondary storage and bounded response time), the algorithms in practice, run in polynomial time. In this sense, the efficient selection of composite storage structures can be considered as a well-solved combinatorial optimization problem.

The re-selection of storage structures problem is applicable to situations in which the future behaviour of an application is uncertain (or not know a priori), and information regarding the next phase can be known (or predicted) only when a phase is near completion. Under this framework, it is not possible to make an overall optimal selection; only a local optimization can be applied at the boundaries of each phase. This re-selection algorithm also incorporates bounds on the amount of resources available, as well as having a pseudo-polynomial running time. To our knowledge, no previous research has been reported for this problem. Given that there exist applications currently using sub-optimal assignments, this algorithm represents a desirable tool for re-assignment of implementations.

Another contribution of this research is the algorithmic approach to solve the optimal selection of reorganization points for storage structures that deteriorate with time, or for selecting implementations for a storage structure in an application consisting of several phases. There exist some closed form solutions, but only for applications restricted to linear deterioration and reorganization costs. The approach presented in this thesis removes this restrictions by allowing arbitrary cost functions and solving the problem in strictly polynomial time. Moreover, it is demonstrated that the algorithm itself is optimal. Furthermore, from a pragmatic viewpoint the problem can be also considered well-solved.

Finally, the selection of sequences of composite storage structures represents a total optimization problem, since all the information regarding each phase (i.e., full characterization of the lifetime) must be known before any selection can be made. Although the algorithms presented in this thesis are exponential (in the number of substructures, or in the number of phases), they constitute a first step towards designing efficient storage structures for applications with predictable phase-oriented lifetimes. Given that it is expected that the number of phases be bounded by a small integer, the described algorithms can again be used in a practical framework.

V.2. Applicability of results.

From a global point of view, the application of the algorithms presented in this thesis should provide a designer of data structures with a set of tools to be part of a general data structure design methodology.

The core of the difficulties in selecting composite storage structures for a given application results from the combinatorial explosion of alternatives. This makes the manual consideration and evaluation of every library implementation for each of the data substructures defined at the abstract structure level a very labor intensive and error prone task. Furthermore, even if a manual selection of implementations is counter-intuitive, a hand re-evaluation is often impractical. Therefore, an algorithmic procedure for selection composite storage structures is a more viable and reliable approach. When the application under consideration can be partitioned into distinct phases, each being sufficiently important to warrant its own selection, the algorithmic approach is even more desirable.

The algorithms presented in this thesis assumed the existence of techniques for partitioning of data type occurrences into homogeneous collections (substructures), and for generating of the required evaluation and conversion cost matrices. Consequently, in order for these algorithms to be successful, it is necessary to substantiate such assumptions.

It should be mentioned that neither problem is trivial. However, it is fortunate that for the former problem some work has already been reported and is currently under way, principally by the SETL group [Schwartz75, Dewar79, Schwartz79]. With respect to the latter problem, several alternatives had been proposed, see for example [Tompa74, Gotlieb74, Cohen74, Wichman72, Low76]. In fact, the whole field concerning the analysis of algorithms (see, for example [Aho74]) can be interpreted as a search for suitable values to be used in the parametric formulas. An integration of the algorithms contained in this thesis with these techniques will constitute a powerful and desirable generalized procedure for selecting composite storage structures.

V.3. Further research.

This section presents some ideas regarding possible research extensions to the problems and algorithms in this thesis.

Throughout this thesis the algorithms presented to solve a particular problem have been characterized by their worst case behaviour. An interesting research extension may attempt to characterize their average behaviour and to compare it with other known approaches, e.g., for the case of selection of composite storage structures, compare the dynamic programming approach suggested here to branch-and-bound as suggested by Tompa, and to hill-climbing by Low.

Given that the algorithms presented here heavily depend on the granularity of available resources, a possible extension to this work will be to tune the algorithms according to a practical framework, i.e., decide the dimensions of the state variables for practical situations. Since the algorithms made no assumptions with respect to the difference of two successive values for a state variable, it is possible to study the effects of dynamic granularity (intervals of different size). Such dynamic granularity could be used to model the different levels of a hierarchical store (e.g., caches, cores, bubble memories, drums etc.)

Another alternative is to study approximate solutions. Given that the selection of composite storage structures problem is related to the knapsack problem, and since there exist polynomial approximations to the knapsack, one could study approximate solutions for this problem as well. Lagrange multipliers is another alternative, commonly used to reduce the dimensionality of a given problem by combining some of the constraints with the objective function. Since one drawback of dynamic programming is the exponential growth of the required computation as the number of state variables increases, reducing the number of state variables is an attractive alternative. Unfortunately when this approach is employed, there is no guarantee that the optimal solution will be found.

Other possible research extensions are mentioned by Tompa and by Low. Among these are the creation of a suitable library of implementations for real situations and extending the approach to several levels of secondary memory. Developing a system for the fully automatic selection of storage structures is obviously the ultimate extension and research goal.

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 third international conference on very large databases, Tokyo, Japan, (October 1977) 545-555.

[Cardenas73] Cardenas A.F. Evaluation and selection of file organizations: A model and system. Communications of the ACM 16, 6 (June 1973) 540-548.

[Cardenas75] Cardenas A.F. Analysis and performance of inverted database structures. Communications of the ACM 18, 5 (May 1975) 253-263.

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

[CODASYL71] CODASYL Database task group, April 1971 report, ACM, New York, 1971.

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

[Comp. Surveys76] Special issue: Data-base management systems, ACM Computing Surveys 8, 1 (March 1976) 1-151.

[Conte72] Conte S.D. and deBoor C. Elementary Numerical Analysis. McGraw-Hill, New York, 1972.

[De78] De P., Haseman W.D., and Kriebel C.H. Toward an optimal design of a network database form relational descriptions, Operations Research 26, 5 (September-October 1978) 805-823.

[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.

[Gotlieb74] Gotlieb C.C. and Tompa F.W. Choosing a storage schema. Acta Informatica 3 (1974) 297-319.

[Ibarra75] Ibarra O.H. and Kim C.H. Fast approximation algorithms for the knapsack and sum of subsets problems. Journal of the ACM 22, 4 (October 1975) 463-468.

[Karp75] Karp R.M. On the complexity of combinatorial problems, Networks 5, 1975, 45-68.

[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.

[Lohman77] Lohman G.M. and Muckstadt J.A. Optimal policy for batch operations: checkpointing, reorganization and updating. *ACM Transactions on database systems 2*, 3 (September 1977) 209-222.

[Low76] Low J.R. *Automatic Coding: Choice of Data Structures*. Interdisciplinary Systems Research Report 16. Birkhauser Verlag, Basel 1976.

[Low78] Low J.R. Automatic data structure selection: an example and overview. *Communications of the ACM 21*, 5 (May 1978) 65-77.

[Mitoma75] Mitoma M.F. and Irani K.B. Automatic database schema design and optimization, *Proceedings of the ACM international conference on very large databases*, Framingham, MA., (September 1975) 286-321.

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

[Schwartz75] Schwartz J.T. Automatic data structure choice in a language of very high level. *Communications of the ACM 18*, 12 (December 1975) 722-727.

[Schwartz79] Schwartz J.T. and Sharir M. Experience with automatic data structure choice in the SETL system. Second program transformation workshop, Boston (September 1979).

[Severance72] Severance D.J. Some generalized modelling structures for use in design file organizations. Ph.D. thesis, University of Michigan, 1972.

[Severance75] Severance D.J. Identifier search mechanisms: A survey and generalized model. ACM Computing Surveys 6, 3 (September 1974) 175-194.

[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.

[Shneiderman73] Shneiderman B. Optimum database reorganization points. Communications of the ACM 16, 6 (June 1973) 362-365.

[Tompa74] Tompa F.W. Choosing a data storage schema. Ph.D. thesis, University of Toronto, 1974.

[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.**

[Tsichritzis78] Tsichritzis D. and Klug A. (eds) The ANSI/X3/SPARC DBMS framework report of the study group on database management systems. Information Systems 3, 1978, 173-191.

[Tuel78] Tuel W. Optimum reorganization points for linearly growing files. ACM Transactions on database systems 3, 1 (March 1978) 32-40.

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

[Yao76] Yao S.B, Das K.S. and Teorey T.J. A dynamic database reorganization algorithm. ACM Transactions on database systems 1, 2 (June 1976) 159-174.

EFFICIENT ALGORITHMS FOR SELECTING EFFICIENT DATA STORAGE STRUCTURES

by

Raul Javier Ramirez Inurrigarro

A thesis
presented to the University of Waterloo
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1980

© (Raul Javier Ramirez Inurrigarro) 1980