DOCUMENT OFFICE 36-412
RESEARCH LABORATORY OF ELECTRONICS
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

#2

AN ANALYSIS OF OPTIMAL RETRIEVAL SYSTEMS WITH UPDATES

RICHARD A. FLOWER

TECHNICAL REPORT 488
June 12, 1975

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
RESEARCH LABORATORY OF ELECTRONICS
CAMBRIDGE, MASSACHUSETTS 02139

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

RESEARCH LABORATORY OF ELECTRONICS

Technical Report 488 June 12, 1975

AN ANALYSIS OF OPTIMAL RETRIEVAL SYSTEMS WITH UPDATES

Richard A. Flower

Submitted to the Department of Electrical Engineering at the Massachusetts Institute of Technology, August 21, 1974, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

(Manuscript received November 26, 1974)

ABSTRACT

The performance of computer-implemented systems for data storage, retrieval, and update is investigated. A data structure is modeled by a set $D = \{d_1, d_2, \ldots d_{|D|}\}$ of data bases. A set of questions $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ about any $d \in D$ may be answered. A memory that is bit-addressable by an algorithm or an automaton models a computer. A retrieval system is composed of a particular mapping of data bases onto memory representations and a particular algorithm or automaton. By accessing bits of memory the algorithm can answer any $\lambda \in \Lambda$ about the d represented in memory and can update memory to represent a new $d^* \in D$. Lower bounds are derived for the performance measures of storage efficiency, retrieval efficiency, and update efficiency. The minima are simultaneously attainable by a retrieval system for some data structures of interest. Trading relations between the measures exist for other data structures.

TABLE OF CONTENTS

I. INTRODUCTION 1

1.1 A Data Problem 1

1.2 Data Model 1

1.3 Retrieval Systems for Data 3

1.4 Computer-Implemented Retrieval Systems 3

II. STATIC RETRIEVAL SYSTEMS 5

2.1 Retrieval Problems 5

2.2 Computer Model 8

2.3 Computer-Implemented Retrieval Systems 10

2.4 System Performance 15

2.5 Memory Performance 17

2.6 Retrieval Performance 26

III. DYNAMIC RETRIEVAL SYSTEMS WITH UPDATES 33

3.1 Updates 33

3.2 Performance Measures for Total Updates 36

3.3 Total Update Performance 40

3.4 Partial Updates 53

IV. TRADING RELATIONS 58

V. FURTHER PROBLEMS 64

Acknowledgment 65

References 66

I. INTRODUCTION

1.1 A DATA PROBLEM

Consider the problem of keeping a course record of the most recent test scores for three students whose test scores range from 0 to 7. There is a need for a course record system that provides a method for representing the observed data (scores), retrieving part or all of the data (scores), and modifying or updating the data (scores) when necessary. A tabular listing of individual scores is the most common course record system.

Design of a data system for problems involving large amounts of data, retrieval of relevant parts of the data, and occasional update of the data requires careful consideration of the following questions: (i) What are the data? (ii) What services should the data system provide? The system designer and potential system users would want to know something of the system performance and expenses, or more precisely, (iii) How is system performance measured? (iv) How well can a system perform? (v) What is the "best" possible system?

This section gives an informal discussion of these questions. A definition of data is presented in section 2.1; a definition of a computer-implemented data system is presented in sections 2.2 and 2.3. System performance is considered in section 2.4 and results for some measures of system performance are presented in sections 2.5 and 2.6 and in Section III. The "best" system is considered in Section IV.

1.2 DATA MODEL

The model of data used in this report is based on two concepts that were developed by Elias[1] and Welch.[2] First, a data problem includes a finite collection or ensemble of data bases that might be observed. At any given time only a single data base from the collection of possible data bases is actually observed. For the course record example, the collection of possible data bases is the finite collection of all $8^3 = 512$ possible course records of three test scores. At any given time only a single course record from this collection is actually observed.

The concept of a collection of possible data bases is implicit in many practical data systems such as hash coding systems for tables, and, in fact, hash coding systems considered by Minsky and Papert[3] were helpful in formulating this model of data. In standard hash coding systems for tables it is implicitly assumed that each table entry is from a fixed range of values and that the number of table entries is no larger than some integer N (see Minsky and Papert,[3] Knuth,[4] or Morris[5]). These two assumptions imply that we can expect to observe a single table from the finite collection of tables of N or fewer entries from a fixed range of values. As another example of a collection of data bases, there are many possible shelf lists for a library. Any library has a single shelf list that is a member of the (very large) collection of possible shelf lists.

The second concept in the definition of data is that questions from a finite set of

retrieval questions determined by potential system users may be asked of the observed data base, no matter which data base from the collection of possible data bases is actually observed. For the class record example, the questions "What is the first student's score?" "What is the second student's score?" and "What is the third student's score?" may be asked of the observed class record, no matter which record is observed.

The concept of a single set of retrieval questions is implicit in many data systems. For example, standard hash coding systems for tables are designed to answer questions of the form "Is the value x a table entry?" (one question for each value x in the fixed entry value range) about the observed table no matter which table is observed. As another example, library retrieval schemes are designed to answer questions of the form "Are there books by the author ______ in this library, and if so where are they?" (one question for each author), "Are there books with the title ______ in this library and if so where are they?" (one question for each title), and "Are there books on the subject ______ in this library, and if so where are they?" (one question for each general subject).

Data are modeled by a collection of possible data bases and a set of retrieval questions. Any data base is completely characterized by the answers it gives to the questions. For the class record example, any class record is completely characterized by the individual test scores (i.e., the answers it gives to the individual score questions).

In standard hash coding systems for tables, any table is completely characterized by its entry values. In determining the value presence question set, the community of potential hash coding system users in effect determined that the presence or absence of values is the only characterization of tables necessary for their purposes. Ordering of table entries is irrelevant, and distinguishing between a table's first entry and last entry is not possible in the hashed representation of a table. As another example, consider libraries. For the purposes of a community of potential library users, a library shelf list is completely characterized by the locations of books with a certain author, title, or subject (i.e., the answers to the library retrieval questions).

In summary, data are modeled as follows:

(i) There is a finite collection of possible data bases, only a single one of which is observed at any given time.

(ii) There is a finite collection of retrieval questions that may be asked of the observed data base, no matter which data base is actually observed.

(iii) A data base is completely characterized by the answers that it gives to the retrieval questions.

Many practical data problems (see Knuth[4] for examples) are accurately modeled by a finite collection of data bases and a finite set of retrieval questions. But, as with any model, it is important to consider which practical data problems are not adequately modeled. The definition of data which we have given does not adequately model problems with an infinite collection of possible data bases or problems with hierarchical question sets (i.e., sets where some questions are appropriate only for certain observed data bases). A discussion of the first case is more appropriately left for Section V, since

it usually occurs in conjunction with certain types of updates on the observed data base. Hierarchical questions can be modeled by questions of the form "If an appropriate data base was observed, then what ...?" or "Assuming that an appropriate data base was observed, what ...?" An inappropriate data base will give an "error" answer to questions of the first form and a wrong answer to questions of the second form without indicating any error. Neither form seems entirely satisfactory, since the first form implies the existence of a large subset of questions that are inappropriate for an observed data base and the second form requires memorizing information by a user so that he can ask appropriate questions. Codd[6] and Earley[7] are among those who have developed other models of data.

1.3 RETRIEVAL SYSTEMS FOR DATA

Given a collection of possible data bases and a set of questions, it is useful to have a retrieval system enabling a user to obtain the answer to any question about the observed data base conveniently and, when necessary, to update the observed data base conveniently. For the class record example, a tabular list of the three test scores allows an interested person to retrieve the observed score of an individual student conveniently and an authorized person to change observed entry values (scores) in the tabular list conveniently.

A retrieval system must provide a method for representing any observed data base and a method for answering any retrieval question about the observed data base. For a given question, the method for answering the question must be independent of the observed data base. To allow the method to depend upon the observed data base would presuppose some knowledge of the observed data base by the user in order to determine which method is appropriate. Knowledgeable users would have no need of a retrieval system. For problems involving frequent modification or update of the observed data base, a retrieval system should provide a method for updating the observed data base.

1.4 COMPUTER-IMPLEMENTED RETRIEVAL SYSTEMS

Frequently, retrieval systems are implemented on a computer. Some of the costs of computer-implemented retrieval systems, with primary emphasis on updating costs, are investigated here. Rather than studying system costs for a particular computer, a model of a computer reflecting many important properties of most computers is used (sec. 2.2). A person with a retrieval problem should not be forced to alter his view of data in order to use a retrieval system. Such a computer-implemented retrieval system should provide (i) a method for representing any observed data base in computer memory, and (ii) for each retrieval question in the question set a single algorithmic method for answering the question about the observed data base.

Note that the requirement that the algorithmic method for answering a question be independent of the observed data base implies a strict separation of "program" and "data" for computer-implemented retrieval systems. The "program" (algorithmic

method) for answering a question must remain constant while presumably the computer memory state (representing the observed "data") differs for different observed data bases.

For problems involving frequent updates of the observed data base, computer-implemented retrieval systems should also provide (iii) a method for updating the observed data base represented in computer memory. Problems including no updates and systems providing no updates (called static retrieval problems and static retrieval systems, respectively) are presented in Section II, and updates are included in the data model in Section III.

II. STATIC RETRIEVAL SYSTEMS

2.1 RETRIEVAL PROBLEMS

The following model of a retrieval problem was developed by Elias[1] and by Welch.[2] A retrieval problem is a collection or ensemble of data bases that might be observed and a collection of retrieval questions that may be asked of any data base. Only a single data base from the collection of possible data bases is observed at a given time. The collection of retrieval questions is determined before observing a data base. No matter which data base is observed later, only questions from the predetermined collection of retrieval questions may be asked.

The collection or ensemble of data bases (called a data structure) is a finite set $D = \{d_1, d_2, \ldots, d_{|D|}\}$ of data bases d. Unfortunately, there appears to be no uniform terminology for retrieval problems. Because of possible conflict with other definitions of the word "data structure" appearing elsewhere, it must be emphasized that the data structure D is not a single complex entity constructed from basic elements d. A data structure is simply the ensemble or collection of data bases that might be observed.

The retrieval question set is a finite set $\Lambda = \{\lambda_j \mid j \in J\}$ of retrieval questions λ_j indexed by the members j of an index set J. Usually it will be most convenient to let the index set be the finite subset of integers $J = \{1, 2, \ldots, |\Lambda|\}$, where $|\cdot|$ denotes the cardinality of a set. The answer to retrieval question λ_j about data base d is denoted $\lambda_j d$. An answer may be a number, an English phrase, a list of books, and so forth. The set of all possible answers to the question λ_j is denoted by $\lambda_j D$.

The collection of data bases D and the collection of retrieval questions Λ constitute a static retrieval problem, denoted (D, Λ).

The following example of a retrieval problem is useful for illustrating the results that will be presented.

Example 1

The set of possible data bases is the collection of possible three-entry tables where the table entries are integers in the range 0-7 inclusive. The set of retrieval questions is λ_1: "What is the value of the first entry?" λ_2: "What is the value of the second entry?" λ_3: "What is the value of the third entry?" The retrieval problem is (D,Λ), where $D = \{0, 1, \ldots, 7\}^3$ and $\Lambda = \{\lambda_j \mid j \in J = \{1, 2, 3\}\}$.

The cardinality of the set D is 512, since there are $8^3 = 512$ different tables of three entries. Each three-entry table is a data base d in the collection of possible data bases D. For the data base (1, 3, 5) (i.e., the three-entry table whose first entry is 1, second entry is 3, and third entry is 5) $\lambda_1 d = 1$, $\lambda_2 d = 3$, and $\lambda_3 d = 5$. Over the collection of possible data bases, the set of possible answers to the question λ_1 is $\lambda_1 D = \{0, 1, 2, 3, 4, 5, 6, 7\}$.

A question λ partitions the collection of data bases into equivalence classes, where the equivalence class $[d]_\lambda$ contains all data bases that give the same answer to the

question λ as data base d gives.

$$[d]_\lambda = \{d' \in D \mid \lambda d = \lambda d'\}$$

In example 1 the question λ_1 divides the collection of data bases into 8 equivalence classes of 64 tables each. The equivalence class $[(1,3,5)]_{\lambda_1}$ contains all 64 data bases that give the same answer to the question λ_1 as the data base (1, 3, 5) (i.e., the class contains all 64 different tables with first-entry value 1). For a question λ let D_λ be the set of equivalence classes over the question λ.

$$D_\lambda = \{[d]_\lambda \mid d \in D\}$$

In example 1, D_{λ_1} is a set whose members are the 8 different equivalence classes over question λ_1.

In a similar manner the entire set of questions Λ divides the collection of data bases into equivalence classes. Two data bases d and d' are in the same equivalence class over all questions if and only if for each question λ, the answer to λ about the data base d is the same as the answer to λ about the data base d'.

$$[d]_\Lambda = \{d' \in D \mid \forall \lambda \in \Lambda, \ \lambda d = \lambda d'\}$$

$$D_\Lambda = \{[d]_\Lambda \mid d \in D\}$$

In example 1, the entire set of questions divides the collection of data bases into 512 equivalence classes, where each class contains a single table. No two tables give the same answer to each question. For any two tables there is at least one question that will be answered differently.

Two data bases from a retrieval problem (D, Λ) which give the same answer for each retrieval question (i. e., are in the same equivalence class $[d]_\Lambda$) are said to be pseudo-identical. If two data bases are indistinguishable over the question set Λ, then in determining the question set the users will have implicitly stated that there is no important difference between the two data bases. From the user's point of view they might as well be treated as a single data base with a new name, say $[d]_\Lambda$. If two data bases are indistinguishable over the question set Λ, then the two data bases may be represented in a retrieval system in exactly the same manner. From the point of view of a retrieval system designer, the system may be designed as if the two data bases were a single data base with a new name, say $[d]_\Lambda$.

A retrieval problem (D, Λ) is said to be complete if and only if no two data bases in D are pseudoidentical over Λ. If a retrieval problem (D, Λ) is not complete, then it can be redefined on the collection of data bases $D_\Lambda = \{[d]_\Lambda \mid d \in D\}$, and the problem (D_Λ, Λ) is complete.

Two questions λ and λ' from a retrieval problem (D, Λ) are said to be strongly related if and only if there exists an invertible function f: $\lambda D \xrightarrow[\text{onto}]{1-1} \lambda' D$ such that for any

$d \in D$ $\lambda' d = f(\lambda d)$. Two questions are strongly related if they are essentially rewordings of one another. If there is a pair of strongly related questions in a retrieval problem, one of the questions may be dropped from the question set with little loss in convenience to the user. If there is a pair of strongly related questions λ and λ' in a retrieval problem, a system designer can construct a method for answering the question λ' simply by adding a translator (computing f) to the method for answering the question λ.

A question λ from a retrieval problem (D, Λ) is said to be a constant question if and only if $\lambda d = \lambda d'$ for all data bases d and d' in D. If a retrieval problem includes a constant question λ, a system designer need not construct a method for answering the question λ, since the user already knows the answer to the question λ no matter which data base is observed.

A retrieval problem is said to be reduced if and only if it includes no strongly related questions and no constant questions. If a retrieval problem (D, Λ) is not reduced, it may be redefined over the question set Λ' formed by dropping a question from pairs of strongly related questions and dropping all constant questions. The problem (D, Λ') is reduced, but not necessarily complete.

Example 2

An example of a retrieval problem that is neither complete nor reduced is the following. The collection of data bases D is the set of tables of three entries drawn from the set of integers 0 through 7 inclusive. The retrieval question set Λ is {λ_1: "What is the smallest entry?" λ_2: "What is the second smallest entry?" λ_3: "What is the third smallest entry?" λ_4: "What is the largest entry?" λ_5: "What is the second largest entry?" λ_6: "What is the third largest entry?"}

The questions λ_1: "What is the smallest entry?" and λ_6: "What is the third largest entry?" are strongly related. Any data base in D gives exactly the same answer for λ_1 and λ_6, since each data base has exactly three entries. The identity map f satisfies $\lambda_6 d = f(\lambda_1 d)$ for any d in D. Similarly, the pairs of questions λ_2, λ_5 and λ_3, λ_4 are strongly related. The retrieval problem may be reduced by dropping λ_6, λ_5, λ_4 from the question set. The retrieval problem $(D, \Lambda' = \{\lambda_1, \lambda_2, \lambda_3\})$ is reduced but not complete.

The pair of data bases $d = (1, 3, 5)$ and $d' = (3, 5, 1)$ in the original problem (D, Λ) are pseudoidentical. Both d and d' give exactly the same answers for every question in the question set ($\lambda_1 d = \lambda_1 d' = 1$, $\lambda_2 d = \lambda_2 d' = 3$, $\lambda_3 = \lambda_3 d' = 5$, $\lambda_4 d = \lambda_4 d' = 5$, $\lambda_5 d = \lambda_5 d' = 3$, $\lambda_6 d = \lambda_6 d' = 1$). In choosing the question set, the community of users has implied that the ordering of entries within a table is not of importance, since reordering the entries of a table does not change the answer to any question. The retrieval problem may be made complete by considering the collection of equivalence classes over the entire question set to be the collection of data bases. For example, the class $[(1, 3, 5)]_\Lambda = \{(1, 3, 5), (1, 5, 3), (3, 1, 5), (3, 5, 1), (5, 1, 3), (5, 3, 1)\}$ would be a single data base in the collection of data bases D_Λ. The retrieval problem (D_Λ, Λ) is complete but not reduced.

The retrieval problem $(D_\Lambda, \Lambda' = \{\lambda_1, \lambda_2, \lambda_3\})$ is both reduced and complete.

This report presents results for reduced, complete retrieval problems. Any retrieval problem may be reduced and made complete by the methods discussed here. A user interested in a nonreduced or noncomplete retrieval problem may use a retrieval system solving the corresponding reduced complete retrieval problem with little loss in convenience. Alternatively, a retrieval system solving a nonreduced and noncomplete problem may be constructed with a simple modification to a system solving the corresponding reduced complete problem.

Several related formal definitions are possible for mathematical problems identical to the information-retrieval problem. One could consider a finite family of functions $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{|\Lambda|}\}$ defined on a common finite domain $D = \{d_1, d_2, \ldots, d_{|D|}\}$. The value of the ith function at domain point d is $\lambda_i d$. The common domain corresponds to the collection of data bases, and the family of functions corresponds to the set of retrieval questions. The construction of a computer system capable of representing any single domain point and of computing the value of each of the functions at the stored domain point could then be investigated. Elias has investigated[8] properties of static computer systems solving this problem for several particular families of functions.

Alternatively, we could consider a finite family of labeled partitions $\{D_\lambda | \lambda \in \Lambda\}$ of a finite set $D = \{d_1, d_2, \ldots, d_{|D|}\}$. The label of the class in the ith partition containing the set element d is $\lambda_i d$. The partitioned set corresponds to the collection of data bases. Each partition in the family of partitions corresponds to a retrieval question in the retrieval question set. The label of the class in the ith partition containing the set element d corresponds to the answer to the ith retrieval question about the data base d. We could then investigate the construction of a computer system capable of representing any single set element and of calculating the label in each of the partitions of the class containing the set element. This formulation of the problem was suggested by Professor D. W. Loveland, Chairman, Computer Science Department, Duke University.

2.2 COMPUTER MODEL

Once the information about a sample has been gathered, it would be useful to store the information in a computer system. The purpose of such a retrieval system is to make the information about the observed data base easily and conveniently available to a community of interested users. Rather than investigating the properties of retrieval systems for a specific computer, it is more appropriate to consider a model of a computer which reflects the capabilities and performance of a wide range of computers. A computer is modeled by a cell-addressable (random access) memory $\mathcal{M}$ and a set of algorithms or machines $\mathcal{A}$.

The memory $\mathcal{M}$ reflects the random-access storage capability of most computers. The memory $\mathcal{M}$ comprises a finite number of cells, each of which can be set to any value from a finite alphabet B. Let $\mathcal{L}(\mathcal{M})$ denote the number of cells in memory $\mathcal{M}$. The cells of memory $\mathcal{M}$ are indexed with the integers $1, 2, \ldots, \mathcal{L}(\mathcal{M})$ for accessing

purposes.

Let $\mathcal{S}(\mathcal{M}, t)$ be the state of memory $\mathcal{M}$ at time t. $\mathcal{S}(\mathcal{M})$ will be used when the time is unambiguous or where the memory state is not time-dependent. This state could be described by listing the values in the memory cells $\mathcal{M}(1), \mathcal{M}(2), \ldots, \mathcal{M}(\mathcal{L}(\mathcal{M}))$. For notational consistency this state will be described by a function from the integers $N_{\mathcal{L}(\mathcal{M})} = \{1, 2, \ldots, \mathcal{L}(\mathcal{M})\}$ to the alphabet B. Such a function is denoted by a set of ordered pairs, where the first member of a pair is an integer cell address and the second member is the value of the memory cell with that address.

$$\mathcal{S}(\mathcal{M}) = \{(1, \mathcal{M}(1))\ (2, \mathcal{M}(2))\ \ldots\ (\mathcal{L}(\mathcal{M}), \mathcal{M}(\mathcal{L}(\mathcal{M})))\}.$$

Although, strictly speaking, the state of memory is a function, it is convenient to refer to such a function as a sequence. Let B^L be the set of all functions defined on the domain $N_L = \{1, 2, \ldots, L\}$ and taking values in B (i. e., sequences of values from the alphabet B of length L). Let B^* be the set of all functions on some finite domain $N_j = \{1, 2, \ldots, j\}$ for some j and taking values in B (i. e., all finite sequences of values from the alphabet B regardless of length). Occasionally it will be more convenient to describe a memory state informally by a string of $\mathcal{L}(\mathcal{M})$ values from the alphabet B. The correspondence between the string and the function from $N_{\mathcal{L}(\mathcal{M})}$ to B is the obvious one.

The algorithms or machines correspond to the central processing unit of a computer. To reflect the memory accessing capabilities of most computer CPU's, the algorithms are given the capability of reading from any memory cell and writing into any memory cell. More precisely, an algorithm is connected to a memory $\mathcal{M}$ by an address line and a read-write line. There is an infinite set of memories $\mathcal{M}^* = \{\mathcal{M}_1, \mathcal{M}_2, \ldots\}$ to which the algorithms might be connected, each of different length $\mathcal{L}(\mathcal{M}_1) = 1$, $\mathcal{L}(\mathcal{M}_2) = 2, \ldots$. An algorithm $\mathfrak{a}$ is connected by the system designer to a single sufficiently large memory $\mathcal{M} \in \mathcal{M}^*$ (i. e., a memory including every cell the algorithm $\mathfrak{a}$ might ever access), so that the algorithm neither "knows" nor "cares" which (sufficiently large) memory $\mathcal{M}$ is actually connected.

If an algorithm specifies an integer cell address on the address line and the value "read" on the read-write line, the value of the memory cell with the specified address is returned from the memory $\mathcal{M}$ to the algorithm on the read-write line. The value of the specified memory cell is unchanged by the read operation. If the algorithm specifies an integer address on the address line and a value from the alphabet B on the read-write line, the memory cell with the specified address is set to the specified value.

An algorithm has an output tape for communicating with the user. Strictly speaking, the output of an algorithm should be a string from a finite output alphabet. Most computer output is in the form of a finite string of values from the roman alphabet, integer characters 0 through 9, and a few special characters such as "blank", *, etc. On the other hand, the answer to a question about a data base can be an English phrase, a list of books, a number, etc. After an algorithm halts, the user is left with the task of

decoding the finite string on the output tape (i. e., interpreting the string as an answer to his question). The decoding problem is not of direct importance to the problems that we are now considering. The output alphabet of an algorithm will be defined shortly in a manner designed to minimize the decoding problem.

An algorithm is formally a halting automaton defined by a 6-tuple $\{s_j, S_j, \nu_j, \theta_j, \Omega_j, \sigma_j\}$, where s_j is the starting state; S_j is the state set including a halting state s_h and a continuing state set $S_c = S_j - s_h$; ν_j is an address function $\nu_j : S_c \rightarrow N^+$; θ_j is an operation function $\theta_j : S_c \rightarrow B \cup \{\text{'read'}\}$; Ω_j is an output function with domain $S_c \times B \cup \{\text{'write'}\}$; and σ_j is a state transition function $\sigma_j : S_c \times B \cup \{\text{'write'}\} \rightarrow S_j$.

The algorithm is placed in starting state $s(t) = s_j$ at time $t = 1$. If at time t the algorithm is in state $s(t) = s$ and memory is in state $\delta(\mathscr{M}, t)$, the algorithm places the value $\nu_j(s)$ on the address line and places the value $\theta_j(s)$ on the read-write line. If $\theta_j(s) = \text{'read'}$, the algorithm reads the value $b = \mathscr{M}(\nu_j(s))$ from the $\nu_j(s)^{th}$ memory cell, writes the value $\Omega_j(s, b)$ on the output tape, and enters state $\sigma_j(s, b)$, leaving memory in state $\delta(\mathscr{M}, t+1) = \delta(\mathscr{M}, t)$. If $\theta_j(s) \in B$, the algorithm writes the value $b = \theta_j(s)$ into the $\nu_j(s)^{th}$ memory cell, writes the value $\Omega_j(s, \text{'write'})$ on the output tape, and enters state $\sigma_j(s, \text{'write'}) = s(t+1)$, leaving the memory in state $\delta(\mathscr{M}, t+1) = \delta(\mathscr{M}, t) - \{(\nu_j(s), \mathscr{M}(\nu_j(s), t))\} \cup \{(\nu_j(s), b)\}$.

A composite algorithm or machine $\boldsymbol{a}_J$ is the union of the algorithms indexed by the members of the index set J. Formally $\boldsymbol{a}_J = \{s_J, S_J, \nu_J, \theta_J, \Omega_J, \sigma_J\}$ where

$$\cdot_J = \bigcup_{j \in J} \cdot_j$$

Although $\boldsymbol{a}_J$ is a single composite algorithm with several starting states, it will be useful to refer to $\boldsymbol{a}_J$ informally as a set of individual algorithms $\boldsymbol{a}_j$.

2.3 COMPUTER-IMPLEMENTED RETRIEVAL SYSTEMS

A retrieval system for a retrieval problem provides a method for retaining the information about any single observed data base and for obtaining the answer to any retrieval question about the observed data base. A retrieval system implemented on the computer model must provide (a) a method for representing any single observed data base in computer memory and (b) a method for answering any retrieval question about the observed data base represented in computer memory. A user of a retrieval system might also want (c) a method for modifying or updating the observed data base represented in computer memory.

(a): <u>A method for representing any single observed data base in computer memory.</u>

An observed data base itself cannot be put into memory (memory is composed of indexed cells containing values from the finite alphabet B). A string of values from the alphabet B representing the observed data base is stored in memory. A computer-implemented

retrieval system provides a representation map ρ from D the collection of possible data bases to sequences in B^* which can be placed in memory to represent a data base.

Requiring that a memory representation specify values for each and every memory cell is unnecessarily restrictive. Most computer systems assign certain sections of memory to a user. Other users are allowed to write in cells in the unassigned sections of memory. For example, a linked list scheme assigns certain sections of memory to a user. A user's sections of memory are linearly chained together by including in each section a special link field giving the address of the next section of memory assigned to the user. Unassigned memory cells between the user's memory sections may be filled by other users or by the system programmer. Representations that specify values for only some of the cells of memory are necessary to model practical memory allocation schemes such as the linked list allocation scheme.

A representation for a data base is a specification of values from the alphabet B for some of the memory cells. A representation could be described by the sequence of values from the alphabet B specified for the indexed sequence of memory cells. A special place-holder symbol, -, which is not in the alphabet B, may be used in the sequence to indicate that the corresponding memory cell has no value specified by the representation. For example, the representation that specifies the value 0 for the first memory cell and the value 0 for the third memory cell can be described by the sequence 0-0 for a memory of length 3, or 0-0- for a memory of length 4, or 0-0-- for a memory of length 5, etc. To make different descriptions of a representation for different memory lengths unnecessary, all trailing place-holder symbols may be dropped from the description. The representation specifying value 0 for the first memory cell and value 0 for the third memory cell can be described by the sequence 0-0, regardless of the length of memory.

Formally, a representation is a function from a set of integer cell addresses to values from the alphabet B. The domain of the function is the set of addresses of memory cells that have values specified by the representation. Because the domain of the function is a finite subset of the positive integers, a representation is a partial function to the alphabet B. Let $B^\dagger$ denote the set of all partial functions from a finite subset of the positive integers to the alphabet B.

A representation is denoted by a set of ordered pairs where the first member of a pair is a cell address in the domain of the representation and the second member of a pair is the value specified by the representation for the memory cell with the corresponding address. For example, the representation that specifies value 0 for the first memory cell and value 0 for the third memory cell is denoted $\{(1, 0), (3, 0)\}$. Informal descriptions of a representation such as 0-0 will be used where it seems more convenient.

A representation m is placed in memory $\mathcal{M}$ by setting the memory cells with addresses i in the domain of the representation m to the value m(i) specified by the representation (partial function), so that $\mathcal{M}(i) = m(i)$ for all i in Domain(m). Note that after

a representation m has been placed in memory $\mathcal{M}$, $m \subseteq \mathcal{S}(\mathcal{M})$, where $\subseteq$ denotes inclusion in the set sense. For example, if the representation $\{(1,0), (3,0)\}$ specifying value 0 for cell 1 and value 0 for cell 3 is placed in memory $\mathcal{M}_3$ of length 3, and if another user fills in unspecified cell 2 with value 1, then $\mathcal{S}(\mathcal{M}_3) = \{(1,0), (2,1), (3,0)\}$ and $m = \{(1,0), (3,0)\} \subseteq \{(1,0), (2,1), (3,0)\} = \mathcal{S}(\mathcal{M}_3)$.

Requiring that the representation map ρ be a function is unnecessarily restrictive. Many computer systems allow several different representations for a single data base. For example, hashing schemes for tables allow several different representations for the same table. The hashed representation of a table depends upon the order in which the table entries were made. Representation maps allowing several memory representations for the same data base are necessary to model practical retrieval schemes such as the hashing scheme for tables. A memory representation map ρ may map a single data base in the collection of data bases D to a finite number of different representations (or more precisely partial functions) in the range $B^\dagger$.

Define $L(\rho)$, the length of a representation map $\rho: D \to B^\dagger$ to be $\max_{d \in D} \max_{m \in \rho(d)} \max$ Domain (m). The length of a representation map ρ is the largest address that has a value specified by some representation in ρ.

(b): <u>A method for answering any retrieval question about the observed data base represented in computer memory.</u>

A user interested in obtaining the answer to a retrieval question must use an algorithm to access the computer memory storing a representation of the observed data base. A retrieval system must provide a composite algorithm $\mathcal{a}_J$ (or set of algorithms $\mathcal{a}_J = \{\mathcal{a}_j \mid j \in J\}$) capable of printing the answer to any retrieval question $\lambda_j \in \Lambda = \{\lambda_j \mid j \in J\}$ about the observed data base represented in computer memory. There must be one starting state s_j (algorithm $\mathcal{a}_j$) for each retrieval question λ_j. To make the decoding as easy as possible, it will be assumed that the output alphabet of algorithm $\mathcal{a}_j$ is $\lambda_j D$, the answer space for retrieval question λ_j. When started, the algorithm $\mathcal{a}_j$ must print the answer $\lambda_j d$ to retrieval question λ_j about the observed data base d and halt without having changed the data base represented in memory, no matter which $d \in D$ is observed, no matter which representation $m \in \rho(d)$ for d is placed in memory $\mathcal{M}$, no matter which memory $\mathcal{M} \in \mathcal{M}^*$ is connected to the algorithm provided it is sufficiently long, and no matter how other users choose to fill in the cells of $\mathcal{M}$ that have no values specified by the representation m.

More precisely, an algorithm $\mathcal{a}_j$ is defined to <u>answer</u> λ_j <u>about</u> D <u>using</u> ρ <u>with</u> $\mathcal{M}_L$ iff $L \geq L(\rho)$ (i.e., memory is sufficiently large) and $\forall\, d \in D$, $\forall\, m \in \rho(d)$, $\forall\, \mathcal{S}(\mathcal{M}_L, t{=}1)$ such that $m \subseteq \mathcal{S}(\mathcal{M}, t{=}1)$, when connected to memory $\mathcal{M}_L$ in state $\mathcal{S}(\mathcal{M}_L, t{=}1)$ and started, the algorithm $\mathcal{a}_j$ accesses a sequence of integer addresses in $[1, L]$, prints the answer $\lambda_j d$, and halts in finite time t_f with memory in a state $\mathcal{S}(\mathcal{M}_L, t_f)$ such that $m' \subseteq \mathcal{S}(\mathcal{M}_L, t_f)$ for some representation $m' \in \rho(d)$.

Where unambiguous, it will be convenient to drop reference to the index set J and

refer to $\mathcal{a}_\Lambda$ as a set of algorithms $\mathcal{a}_\Lambda$ to answer questions $\lambda \in \Lambda$ rather than referring to $\mathcal{a}_J$ as a single composite algorithm containing subalgorithms $\mathcal{a}_j$ (starting states s_j) to answer question λ_j for $j \in J$.

(c): A method for updating the observed data base represented in computer memory.

The major topic of concern in this report is the characteristics of retrieval problems including updates and of systems providing updates. Section III gives a reasonably self-contained presentation of this topic. It is suggested, however, that the rest of this and the following sections on retrieval system performance be read for background before proceeding to Section III. Retrieval problems that need no update and retrieval systems that provide no update will be known as static retrieval problems and static retrieval systems.

A static retrieval system $(\rho, \mathcal{a}_\Lambda)$ including a set of algorithms $\mathcal{a}_\Lambda$ connected to memory $\mathcal{M}_{L(\rho)}$ is defined to solve static retrieval problem (D, Λ) iff $\mathcal{a}_\Lambda$ answers Λ about D using ρ with $\mathcal{M}_{L(\rho)}$. Note that a retrieval system could be defined more generally to include any memory $\mathcal{M}_L$ where $L \geq L(\rho)$ so that other users could be assigned high-order as well as low-order addresses with unspecified values and retrieval algorithms could access addresses greater than $L(\rho)$. Although the results presented here are true for this more general case, inclusion of this case in formal proofs does not seem worthwhile because of additional notational complexity. Let $R(D, \Lambda)$ be the set of all retrieval systems which solve static retrieval problem (D, Λ).

In section 2.1 a retrieval problem (example 1) was presented where the collection of data bases was the collection of tables of three entries from the integer range 0 through 7 inclusive and the collection of retrieval questions asked for the entry values of the observed table. There are many possible retrieval systems for this example retrieval problem.

Example 1a (continued). Enumerative Field Representation Map

For memories with binary-valued cells each table can be represented by the concatenation of the 3-bit standard binary representation for each table entry. The representation of table (1, 3, 5) is 001 011 101, the standard binary representation 001 for the integer value 1, followed by the standard binary representation 011 for the integer value 3, followed by the standard binary representation 101 for the integer value 5. Formally, $\rho((1,3,5)) = \{m\}$, where $m = \{(1,0)\ (2,0)\ (3,1)\ (4,0)\ (5,1)\ (6,1)\ (7,1)\ (8,0)\ (9,1)\}$.

The algorithm $\mathcal{a}_1$ to answer retrieval question λ_1: "What is the value of the first entry?" reads and remembers the values in the first three memory cells, prints the integer whose standard binary representation is the three values read, and halts. The formal definition of the algorithm along with a diagram is given in Fig. 1. Similarly, an algorithm to answer the second retrieval question λ_2: "What is the value of the

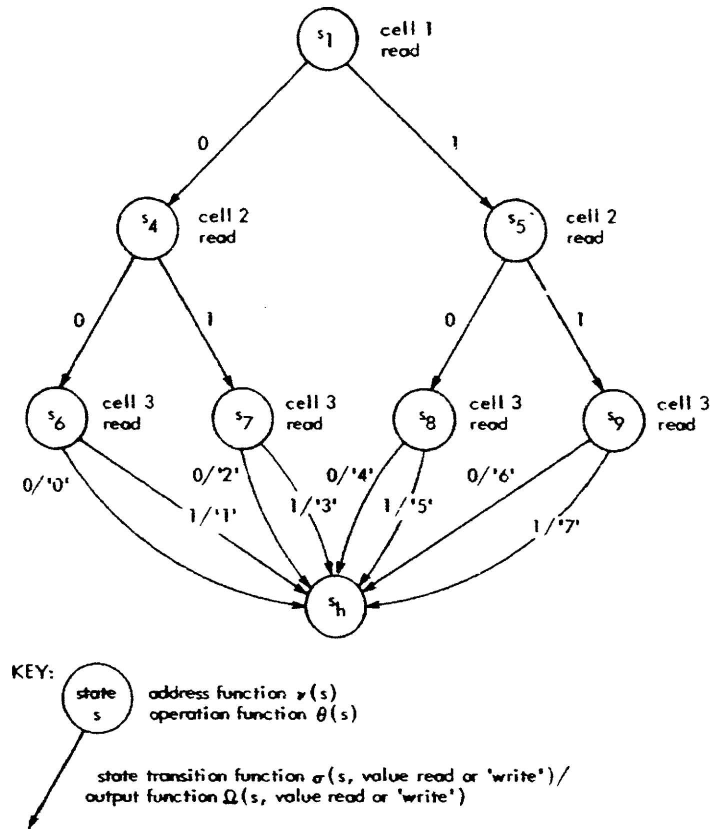


Fig. 1. Retrieval algorithm a_1 of example 1a.

second entry?" reads the 4th, 5th, and 6th memory cells and an algorithm to answer the third retrieval question reads the 7th, 8th, and 9th memory cells.

Example 1b (continued). Positional Field Representation Map

A second possible retrieval system for solving example 1 uses a different representation map and a different set of retrieval algorithms. The representation of a table is the union of three partial functions on subdomains (or fields) of 8 cell addresses each. Each partial function on a field is a positional encoding of an entry value. The positional encoding of an entry value i specifies value 1 for the $i+1^{th}$ cell address in the field and specifies value 0 for the other seven cell addresses in the field. The representation of the table (1, 3, 5) specifies value 1 for the 2nd cell address of the first field $\{1, 2, \ldots, 8\}$, value 1 for the 4th cell address of the second field $\{9, 10, \ldots, 16\}$, value 1 for the 6th cell address of the third field $\{17, 18, \ldots, 24\}$, and value 0 for the other cell address in the three fields.

$$\rho((1, 3, 5)) = \{\{01000000 \quad 00010000 \quad 00000100\}\}$$

The algorithm to answer λ_1: "What is the value of the first entry?" reads the value of the last cell in field 1. If this cell has value 1, the algorithm prints answer "7" and halts. Otherwise the algorithm reads the value of the next to last cell in field 1. If this cell has value 1, the algorithm prints the answer "6" and halts. Otherwise the algorithm continues to read sequentially across the first field right to left until it encounters the single cell in this field which is set to value 1. Upon encountering the cell, the algorithm prints the integer which is 8 minus the number of cells read and halts. Algorithms to answer the second and third questions behave in a similar manner on the second and third fields.

2.4 SYSTEM PERFORMANCE

Implementation of an information retrieval system requires the cooperative effort of several people. The system programmer constructs a representation map ρ from the collection of data bases D to representations for the data bases that can be placed in memory. The system programmer also constructs a set of algorithms $\mathcal{A}_\Lambda$ to answer the set of user questions Λ. Because the system programmer does not know which $d \in D$ will actually be observed, which representation $m \in \rho(d)$ for the observed d will be placed in memory, how other users will fill in the free cells, or which questions the user will ask, each algorithm must be designed to print the proper answer to the question no matter which data base $d \in D$ is actually observed, no matter which representation $m \in \rho(d)$ for d is placed in memory, and no matter how other users choose to fill in memory cells that have no value specified by the representation m. After collecting the information about the observed data base d, the data gatherer consults the representation map ρ to determine possible representations for d and attempts to place a representation $m \in \rho(d)$ for the observed data base in memory $\mathcal{M}$. With the retrieval system $(\rho, \mathcal{A}_\Lambda)$ constructed by the system programmer and the representation $m \in \rho(d)$ placed in memory by the data gatherer, the user is able to answer any desired question $\lambda \in \Lambda$ about the data base observed by specifying an algorithm $\mathcal{A}_\lambda$ or a starting state s_λ of the machine $\mathcal{A}_\Lambda$. After the algorithm halts, the user looks at the output tape and determines the answer λd to his question. The user may also want the ability to update or modify the data base under consideration.

During the construction of a retrieval system, each person assures himself that his portion of the system works correctly. Beyond that each person may ask himself how well he can do, or more basically what does he mean by doing well and how is it measured. These are complex questions to ask of a system. They are further complicated because each person involved in the design or operation of a system has a different viewpoint. Each person may answer the questions differently.

The system programmer is concerned with the complexity of the algorithm $\mathcal{A}_\Lambda$ and the amount of memory that must be available for storing a representation of the observed data base. The data gatherer is concerned with the complexity of his encoding task ρ and the amount of memory that must be available for storing a representation of the data

base he has observed. The user is concerned with the ease with which he can indicate the information he desires, the time necessary to retrieve the desired information, the cost of the service, and the flexibility of the service (i. e., the ease with which he can modify or update a data base).

The memory $\mathcal{M}$ is of concern either directly or indirectly to each person involved in the design or operation of an information retrieval system. Performance measures and bounds on the measures for memory are presented in section 2.5.

The time necessary to retrieve the desired information is of direct concern to the user and of indirect concern to the system programmer. Performance measures and bounds on the measures for information retrieval time are presented in section 2. 6.

The ease with which the observed data base can be modified or updated may be of some importance to the user. The questions "What is meant by an update?" "How is an update capability incorporated into an information retrieval system?" "What is meant by doing well with respect to an update?" and "How well is it possible to do?" are the primary concerns of this report. The meaning of the term update, appropriate measures for the update performance of a system, and bounds on the measures are presented in Section III. The overall performance of a system is considered in Section IV.

Before presenting these results, however, other areas of concern to the designer or operator of an information retrieval system merit further discussion.

The complexity of computing a representation for the data base observed by using the map ρ is of concern to the data gatherer; his task is a one-time initialization task. Once a representation is placed in memory, the system may be used indefinitely for static information retrieval without requiring further services from the data gatherer. Thus the complexity of computing the mapping is not of major importance to the long-term performance of an information retrieval system.

The ease with which the desired retrieval operation can be indicated is of direct concern to the user and of indirect concern to the system programmer. From the system programmer's point of view, this indication is the data to the program $\mathfrak{a}_\Lambda$ written by the system programmer to run on a general-purpose computer. From the user's point of view this indication is a program written by the user to run on a special-purpose machine $\mathfrak{a}_\Lambda$, which in turn has been simulated on a general-purpose computer by the system programmer. Which viewpoint is adopted is a matter of personal preference. Be it data or be it program, the indication of the information desired could not be much more simple than the indication of one algorithm out of a set of $|\Lambda|$ algorithms. This aspect of performance will not be investigated further.

The complexity of the algorithms $\mathfrak{a}_\Lambda$ is of direct concern to the system programmer and indirect concern to the user. There are several aspects to the complexity of the algorithms $\mathfrak{a}_\Lambda$.

One aspect is the amount of state information that must be maintained by an algorithm to remember its current state. For any particular algorithm $\mathfrak{a}_\Lambda$ the amount of this information is easily quantified by the size of the state set S, or more commonly

by the log of the size of the state set. This measure roughly corresponds to the number and size of the registers in the CPU of a computer.

Another aspect of the complexity of the algorithms $\boldsymbol{a}_\Lambda$ is the complexity of the state transition functions σ. Certainly the complexity of a function is a topic of considerable interest in its own right. While measures for the complexity of particular functions have been investigated, at present there seems to be no adequate measure for the complexity of a general function. For particular algorithms an intuitive opinion of the complexity of the state transition function will be given. Because of the lack of an appropriate measure, however, the complexity of the algorithms will not be a topic of major consideration.

In practice, the output of an algorithm is a string from a finite output alphabet which a user must decode (i. e., interpret as an answer to his question). The output alphabet has been redefined here to minimize this decoding task. The reader interested in the complexity of the decoding task should consult Elias.[9]

2.5 MEMORY PERFORMANCE

One measure for the memory performance of a retrieval system $(\rho, \boldsymbol{a}_\Lambda)$ solving a retrieval problem (D, Λ) is the number of memory cells dedicated to the storage of a representation m in memory. Let $|m|$ denote the number of memory cells for which the representation m specifies a value. Formally, $|m|$ is the cardinality of the set m of ordered pairs. For the representation $m = \{(1,0)\ (3,0)\}$ specifying value 0 for cell 1 and value 0 for cell 3, $|m| = |\{(1,0)\ (3,0)\}| = 2$. This measure is similar to a measure used by Minsky and Papert[3] and by Elias.[8]

The data gatherer places a representation in memory by setting the memory cells whose addresses are in the domain of the representation to the specified values from the alphabet B. After a representation is placed in memory, the precise state of memory may still not be known. For example, after the representation 0-0 is placed in a binary memory of length three by writing 0 in cell 1 and 0 in cell 3, the state of memory may be 000 or it may be 010.

A memory state is consistent with a representation if the memory cells with addresses in the domain of the representation have the values specified by the representation. For a binary memory of length 3 the memory states 000 and 010 are consistent with the representation 0-0. Formally, a state of memory $\mathcal{S}(\mathcal{M})$ is consistent with a representation m if and only if $m \subseteq \mathcal{S}(\mathcal{M})$. For a binary memory of length 3 memory states $\{(1,0)\ (2,0)\ (3,0)\}$ and $\{(1,0)\ (2,1)\ (3,0)\}$ are consistent with the representation $m = \{(1,0)\ (3,0)\}$ since $\{(1,0)\ (3,0)\} \subseteq \{(1,0)\ (2,0)\ (3,0)\}$ and $\{(1,0)\ (3,0)\} \subseteq \{(1,0)\ (2,1)\ (3,0)\}$. Note that if the set of memory addresses does not contain the domain of a representation (i. e., memory is too short), then no memory state is consistent with the representation.

The L-closure of a representation m is the set of memory states of length L that are consistent with a representation m. The L-closure of a representation is the set

of possible memory states after the representation has been placed in the memory of length L. Formally, the L-closure of a partial function m (denoted $\overline{m}_L$) is defined as follows:

$$\overline{m}_L = \{\mu \in B^L \mid m \subseteq \mu\}.$$

The 3-closure of the representation m = 0-0 is $\overline{m}_3 = \{000, 010\}$. The *-closure of a representation m (denoted $\overline{m}_*$) is the $L(\rho)$-closure of m.

For any map ρ from D, the collection of data bases to partial functions in $B^{\dagger}$, let $\overline{\rho_L(d)} = \bigcup_{m \in \rho(d)} \overline{m}_L$ and let $\overline{\rho_*(d)} = \bigcup_{m \in \rho(d)} \overline{m}_*$. For example, if $\rho(d) = \{1, -11\}$, then $\overline{m}_3 = \{100, 101, 110, 111\}$, $\overline{m}'_3 = \{011, 111\}$, and $\overline{\rho_3(d)} = \overline{m}_3 \cup \overline{m}'_3 = \{011, 100, 101, 110, 111\}$. The set $\overline{\rho_L(d)}$ is the set of all possible memory states of length L after some representation for d has been placed in memory $\mathscr{M}_L$. The set $\overline{\rho_*(d)}$ is the set of possible memory states of length $L(\rho)$ after some representation for d has been placed in memory $\mathscr{M}_{L(\rho)}$.

Similarly, let $\overline{\rho_L(D)} = \bigcup_{d \in D} \overline{\rho_L(d)}$ and $\overline{\rho_*(D)} = \bigcup_{d \in D} \overline{\rho_*(d)}$. The set $\overline{\rho_L(D)}$ is the set of possible memory states of length L after some representation for some data base has been placed in memory $\mathscr{M}_L$ and the set $\overline{\rho_*(D)}$ is the set of possible memory states of length $L(\rho)$ after some representation for some data base has been placed in memory $\mathscr{M}_{L(\rho)}$. Note that $\overline{\rho_L(D)}$ may not be B^L the set of all sequences of length L.

A map ρ from D, a collection of data bases to $B^{\dagger}$, the set of partial functions is a representation map for D iff for any complete reduced retrieval problem (D, Λ) and for some set of retrieval algorithms $\mathscr{a}_\Lambda$, $(\rho, \mathscr{a}_\Lambda)$ solves (D, Λ). In other words, a map ρ is a representation map for a data structure D if and only if for any retrieval problem on D, some system using ρ solves the problem. While the definition of a representation map is certainly pragmatic, the definition is difficult to apply to any given map $\rho: D \rightarrow B^{\dagger}$. A simple test for any map ρ is presented in the following theorem which has been proved by Elias.

Theorem 1 (Elias[1])

A map $\rho: D \rightarrow B^{\dagger}$ is a representation map for D iff for any two distinct data bases d and d' in D $\overline{\rho_*(d)} \cap \overline{\rho_*(d')} = \emptyset$.

Theorem 1 states that a map ρ is a representation map for D if and only if the *-closures of representations for distinct data bases are distinct. It is relatively easy to perform a test for distinct closures on a map.

For example, consider the map $\rho: D = \{d_1, d_2\} \rightarrow \{0, 1\}^{\dagger}$, $\rho(d_1) = \{0\text{-}0\}$ and $\rho(d_2) = \{1, -11\}$. Any sequence in $\overline{\rho_*(d_1)}$ must have a 0 in the first position and a 0 in the third position. Any sequence in $\overline{\rho_*(d_2)}$ must have either a 1 in the first position or a 1 in the third position. Under these conditions no sequence could be in both $\overline{\rho_*(d_1)}$ and $\overline{\rho_*(d_2)}$. Thus $\overline{\rho_*(d_1)} \cap \overline{\rho_*(d_2)} = \emptyset$ and ρ is a representation map for D. For any complete

reduced retrieval problem on D there is a retrieval system using ρ which solves the problem.

<u>Example</u> 3

Consider the retrieval problem (D, Λ) where

$$D = \{d_1, d_2\}$$

$$\Lambda = \{\lambda: \text{"What is the index of the observed data base?"}\}.$$

The following retrieval system using the representation map ρ solves the problem.

System (ρ, a_λ) where

$$\rho(d_1) = \{0\text{-}0\}$$

$$\rho(d_2) = \{1, -11\}$$

a_λ:
```
read cell 3
if value = 1 then print '2' and halt
read cell 1
if value = 1 then print '2' and halt
print '1' and halt.
```

If the system designer had tried to include a short memory representation for the data base d_2, say -0, the resulting map $\rho'(d_1) = \{1, -11\}$, $\rho'(d_2) = \{-0, 0\text{-}0\}$ would not have disjoint closures, since $101 \in \overline{\rho'_3(d_1)} = \{011, 100, 101, 110, 111\}$ $101 \in \overline{\rho'_3(d_2)} = \{000, 001, 010, 100, 101\}$.

It would not be possible to construct a retrieval system using ρ' to solve the example retrieval problem 3. No matter how the algorithm a_λ is designed, when connected to a memory of length $L(\rho) = 3$ and confronted with the memory state 101 the algorithm must print some answer and halt, since $101 \in \overline{\rho_*(D)}$. Suppose the algorithm prints '1' and halts. The answer will be wrong if the data gatherer observed d_2, placed representation -0 for d_2 in memory by writing value 0 in cell 2, and other users wrote value 1 in cell 1 and value 1 in cell 3, leaving memory in state 101. Suppose the algorithm prints '2' and halts. The answer will be wrong if the data gatherer observed d_1, placed representation 1 for d_1 in memory by writing value 1 in cell 1, and other users wrote value 0 in cell 2 and value 1 in cell 3, leaving memory in state 101. It is impossible to construct a retrieval algorithm a_λ such that (ρ', a_λ) solves (D, Λ). Thus ρ' is not a representation map for D.

<u>Proof of Theorem</u> 1: Suppose a map $\rho: D \to B^\dagger$ has disjoint L-closures for $L = L(\rho)$. For any retrieval problem (D, Λ) on D, it is possible to construct a retrieval system (ρ, a_Λ) solving (D, Λ) as follows. An algorithm a_λ to answer question λ reads and remembers the sequence of values $\mu \in B^L$ in the first L cells of memory, does a table

look-up to determine for which $d\,\mu \in \overline{\rho_L(d)}$ (this is true for only one d, since the L-closures of ρ are disjoint), prints the answer λd, and halts.

If a map $\rho: D \rightarrow B^\dagger$ does not have disjoint *-closures, then there is a sequence μ such that $\mu \in \overline{\rho_*(d)}$ and $\mu \in \overline{\rho_*(d')}$ for some d and d' in D. There is some representation $m \in \rho(d)$ for d and some other representation $m' \in \rho(d')$ for d' such that $\mu \in \overline{m}_*$ and $\mu \in \overline{m}'_*$. Suppose the system designer attempted to construct a retrieval system $(\rho, \boldsymbol{a}_\Lambda)$ using ρ and solving the retrieval problem (D, Λ) defined on D. When confronted with the memory state μ, $\boldsymbol{a}_\lambda$, the algorithm to answer question $\lambda \in \Lambda$ must print an answer and halt, since $\mu \in \overline{\rho_*(D)}$. The memory may be in state μ because the data gatherer observed data base d, placed representation $m \in \rho(d)$ for d in memory, and other users filled in the unspecified cells in a manner leaving memory in state μ. Or the memory may be in state μ because the data gatherer observed data base d', placed representation $m' \in \rho(d')$ for d' in memory, and other users filled in the unspecified cells in a manner leaving memory in state μ. In one of these cases an algorithm $\boldsymbol{a}_\lambda$ to answer a question $\lambda \in \Lambda$ will print the wrong answer, since for some λ $\lambda d \neq \lambda d'$ (otherwise the problem (D, Λ) would not be reduced). The system $(\rho, \boldsymbol{a}_\Lambda)$ using ρ cannot solve the problem (D,Λ) and ρ is not a representation map of D.

Given any map $\rho: D \rightarrow B^\dagger$, the disjoint closure property of Theorem 1 provides a straightforward method for determining whether ρ is a representation map for D. But the disjoint closure property does not directly provide information about the performance measure $|m|$. For a representation map ρ for D, define

$$m(d) = m_1 \in \rho(d) \mid \forall\ m_2 \in \rho(d) |m_1| \leq |m_2|$$

$$m'(d) = m_3 \in \rho(d) \mid \forall\ m_4 \in \rho(d) |m_3| \geq |m_4|.$$

The representation m(d) is one of the shortest representations for d and the representation m'(d) is one of the longest representations for d. For the example representation map 3 $\rho(d_1) = \{0\text{-}0\}$ $\rho(d_2) = \{1, -11\}$, the shortest and longest representations are $m(d_1) =$ 0-0, $m'(d_1) =$ 0-0, $m(d_2) = 1$, and $m'(d_2) = -11$. The following theorem proved by Elias deals directly with the performance measure $|m|$.

Theorem 2 (Elias[1])

Let D be a data structure and let the measure $|m(d)|$ be as defined above. If $\rho: D \rightarrow B^\dagger$ is a representation map for D, then:

$$\sum_{d \in D} |B|^{-|m(d)|} \leq 1.$$

Theorem 2 is an inequality on a summation of terms of the memory alphabet size raised to a negative exponent. Corresponding to each data base d in D there is one

term in the summation with negative exponent $|m(d)|$.

For the example representation map 3 discussed above, the summation of Theorem 2 becomes

$$\sum_{d \in D} |B|^{-|m(d)|} = 2^{-|m(d_1)|} + 2^{-|m(d_2)|} = 2^{-|0-0|} + 2^{-|1|} = 2^{-2} + 2^{-1} = \frac{3}{4}$$

and Theorem 2 is obeyed with strict inequality.

For the enumerative field representation map 1a for tables, each representation specifies values for exactly 9 memory cells and there are 512 data bases in D. The summation of Theorem 2 becomes

$$\sum_{d \in D} |B|^{-|m(d)|} = 512 \cdot 2^{-9} = 1$$

and Theorem 2 is obeyed with equality.

For the positional field representation map 1b for tables, each representation specifies values for exactly 24 memory cells. The summation of Theorem 2 becomes $512 \cdot 2^{-24} = 2^{-15}$ and Theorem 2 is obeyed with strict inequality.

Theorem 2 is a statement about distributions of the performance measure $|m(d)|$ for any representation map of D. Not all of the data bases in D can have short representations and if some of the data bases have relatively short representations, then others must have relatively long representations. The term in the summation corresponding to a data base d with a short representation m(d) will have a small negative exponent $-|m(d)|$ and hence a relatively large value. For the inequality to be satisfied, the other terms in the summation corresponding to other data bases d' must have relatively small values and hence relatively large negative exponents $-|m(d')|$. And when the exponent $|m(d')|$ is relatively large, m(d'), the shortest representation for data base d', is relatively long.

Proof of Theorem 2: For a representation map $\rho: D \to B^{\dagger}$ let $L = L(\rho)$. The L-closure of m(d) contains $|B|^{L-|m(d)|}$ sequences, since there are $L - |m(d)|$ cells with unspecified values and $|B|$ ways to fill in each unspecified cell.

$$\sum_{d \in D} |\overline{m_L(d)}| = \sum_{d \in D} |B|^{L-|m(d)|} \leq |B|^L.$$

The inequality follows from the observations:

(i) The L-closures of ρ over $d \in D$ are disjoint

(ii) The L-closure of m(d) is contained in the L-closure of $\rho(d)$, since $m(d) \in \rho(d)$

(iii) There are $|B|^L$ different sequences of values from the alphabet B of length L.

Dividing this equation by $|B|^L$ yields Theorem 2.

While Theorem 2 deals directly with the performance measure $|m|$, it is relatively difficult to fathom the implications of the inequality for system performance parameters of direct interest to the system user or system programmer.

The user with a static retrieval problem deals with the single data base d observed by the data gatherer for a long period of time. For the use of a retrieval system he must pay for use of $|m|$ memory cells if the data gatherer placed the representation $m \in \rho(d)$ for the observed data base in memory. A helpful data gatherer interested in minimizing memory costs would place m(d), the shortest representation for observed data base d, in memory. For other reasons, the data gatherer might place m'(d), the longest representation for the observed data base d, in memory. In either case a user of a system would be interested in the greatest cost he might find himself paying for memory, since he deals with a single data base for long periods of time. More precisely, he would be interested in the following parameters of his system:

$$\max_{d \in D} |m(d)| \quad \text{and} \quad \max_{d \in D} |m'(d)|.$$

The system programmer might find it difficult to allocate the unspecified memory cells efficiently between other memory cells dedicated to a single user. In this case he would want to assign a contiguous section of memory to a retrieval system. The span of a representation map ρ (denoted span(ρ)) is the smallest set of contiguous memory cells capable of holding any representation in the representation map. More precisely, the span of a representation map ρ for D is the set of memory cells with addresses between

$$\min_{m \in \rho(D)} \min \text{Domain}(m) \quad \text{and} \quad \max_{m \in \rho(D)} \max \text{Domain}(m) = L(\rho)$$

inclusive. The span of example representation map 3 is the set of cells with addresses 1, 2, 3, since

$$\min_{m \in \rho(D)} \min \text{Domain}(m) = \min \text{Domain}(0\text{-}0) = 1$$

and

$$\max_{m \in \rho(D)} \max \text{Domain}(m) = \max \text{Domain}(0\text{-}0) = 3.$$

Any representation m in $\rho(D)$ may be placed in the first three memory cells. The following theorem deals with system parameters of direct interest to the system designer and system user.

Theorem 3 (Elias[1])

Let D be a data structure and let the measures $|m(d)|$ be as defined above.

(i) If $\rho: D \to B^{\dagger}$ is a representation map for D, then

$$|\text{span}(\rho)| \geq \max_{d \in D} |m(d)| \geq \lceil \log_{|B|} |D| \rceil.$$

(ii) There is a representation map $\rho: D \to B^{\dagger}$ for D such that

$$|\text{span}(\rho)| = \max_{d \in D} |m'(d)| = \lceil \log_{|B|} |D| \rceil.$$

Part (i) states that in any representation map $\rho: D \to B^\dagger$ for D the largest representation specifies values for at least $\lceil \log_{|B|} |D| \rceil$ memory cells. Even if the data gatherer always conserves memory resources by placing in memory the shortest representation m(d) for observed data base d, observing some data base will require placing in memory a representation specifying values for at least $\lceil \log_{|B|} |D| \rceil$ cells. Similarly, the span of any representation map $\rho: D \to B^\dagger$ for D contains at least $\lceil \log_{|B|} |D| \rceil$ consecutive memory cells.

For the example data structure 3, $D = \{d_1, d_2\}$, the lower bound on the performance measures of a representation map to a binary alphabet is $\lceil \log_{|B|} |D| \rceil = \lceil \log_2 |\{d_1, d_2\}| \rceil = 1$. The performance measures of any representation map to a binary alphabet are no less than 1. For the example representation map 3 the performance measures are

$$\max_{d \in D} |m(d)| = \max \{|m(d_1)|, |m(d_2)|\} = \max \{|0\text{-}0|, |1|\} = 2$$

$$\max_{d \in D} |m'(d)| = \max \{|m'(d_1)|, |m'(d_2)|\} = \max \{|0\text{-}0|, |\text{-}1\,1|\} = 2$$

and

$$|\text{span}(\rho)| = |\{\text{cell 1, cell 2, cell 3}\}| = 3.$$

For the example data structure 1 containing all tables of three entries drawn from the integer range [0, 7], the lower bound on the performance measures of any representation map to a binary alphabet is $\log_2 |512| = 9$. The performance measures of the enumerative field representation map 1a are

$$|\text{span}(\rho)| = \max_{d \in D} |m'(d)| = \max_{d \in D} |m(d)| = 9.$$

The performance measures of the positional field representation map 1b are

$$|\text{span}(\rho)| = \max_{d \in D} |m'(d)| = \max_{d \in D} |m(d)| = 24.$$

Part (ii) states that for any data structure D, there is a representation map which attains the lower bound of part (i) in all memory performance measures.

For the example data structure 3, the lower bound is 1, and the performance measures of the map $\rho(d_1) = \{0\}$, $\rho(d_2) = \{1\}$ are all 1. For the table example 1, the lower bound is 9 and the performance measures of the enumerative field representation map are all 9. For any data structure D, one representation map which attains the bound of part (i) is an enumerative encoding of D into sequences of length $L = \lceil \log_{|B|} |D| \rceil$. The L-closures of an enumerative encoding are disjoint so by Theorem 1 the encoding is a representation map for D. This enumerative encoding may be difficult for the data gatherer to compute.[10]

Proof of Theorem 3: Let $m_{max} = \max_{d \in D} |m(d)|$.

$$|D|\,|B|^{-m_{max}} \leq \sum_{d \in D} |B|^{-|m(d)|} \leq 1.$$

The first inequality follows from $|B|^{-m_{max}} \leq |B|^{-|m(d)|} \; \forall \; d \in D$ and the second inequality is Theorem 2. Rearranging the end terms of the inequality, taking logarithms to the base $|B|$, and using the fact that m_{max} is an integer, yields part (i).

As we have mentioned, an enumerative encoding attains the bounds of part (i).

The worst-case memory performance measures of Theorem 3 are of interest to a potential user with a static retrieval problem, since he might find himself paying the costs of worst-case performance for a long time. Average performance measures are more appropriate when the system programmer supplies several copies of a retrieval system to several different users whose individual data gatherers independently observe possibly different data bases from D or when in a single system the existing data base is changed from time to time (as it might be if the user can update the existing data base).

Let P be a probability distribution on D. The distribution P may be an ensemble distribution (e. g., several copies of a system with independent data gatherers) or a time distribution (e. g., independent updates of the existing data base on a single system over a period of time). Define the measures avg(m) and avg(m') as follows:

$$avg(m) = \sum_{d \in D} P(d)\,|m(d)|$$

$$avg(m') = \sum_{d \in D} P(d)\,|m'(d)|.$$

The quantity avg(m) is the (time or ensemble) average number of memory cells dedicated to a user if the shortest representation m(d) for existing data base d is always placed in memory. The quantity avg(m') is the (time or ensemble) average number of memory cells dedicated to a user if the longest representation m'(d) for existing data base d is always placed in memory. If neither the shortest nor the longest representation for existing data base d is always placed in memory, then the average number of memory cells dedicated to a user is somewhere between the quantities avg(m) and avg(m'). The following theorem deals with average memory performance measures.

Theorem 4 (Elias[9])

Let D be a data structure, let P be a probability distribution on D, and let the average performance measure avg(m) be as defined above.

(i) If $\rho: D \rightarrow B^{\dagger}$ is a representation map for D, then

$$\text{avg}(m') \geq \text{avg}(m) \geq H(D)$$

where

$$H(D) = -\sum_{d \in D} P(d) \log_{|B|} P(d).$$

(ii) There is a representation map $\rho: D \rightarrow B^{\dagger}$ for D such that

$$\text{avg}(m) = \text{avg}(m') < H(D) + 1.$$

Part (i) states that the average memory performance measures of any representation map for D are no less than the quantity H(D). For the example data structure 3 and the probability distribution $P(d_1) = P(d_2) = 1/2$, the average performance measures of any representation map $\rho: D \rightarrow \{0, 1\}^{\dagger}$ are no less than $H(D) = -\sum_{d \in D} P(d) \log_2 P(d) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1$. The performance measures for example representation map 3 are

$$\text{avg}(m) = P(d_1)|m(d_1)| + P(d_2)|m(d_2)| = \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 1 = 1\frac{1}{2}$$

and

$$\text{avg}(m') = P(d_1)|m'(d_1)| + P(d_2)|m'(d_2)| = \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 2 = 2.$$

For the table example 1 and the probability distribution $P((7, 7, 7)) = \frac{1}{2}$, $P((6,7,7)) = P((7, 6, 7)) = P((7, 7, 6)) = \frac{1}{8}$ and for any other table d, $P(d) = \frac{1}{8 \cdot 508}$, the average performance measures of any representation map $\rho: D \rightarrow \{0, 1\}^{\dagger}$ are no less $H(D) = -\frac{1}{2} \log_2 \frac{1}{2} - 3 \cdot \frac{1}{8} \cdot \log_2 \frac{1}{8} - 508 \cdot \frac{1}{8 \cdot 508} \cdot \log_2 \frac{1}{8 \cdot 508} = 3.1236$. The average performance measures for the enumerative field representation map 1a are both 9. The average performance measures for the positional field representation map 1b are both 24.

Part (ii) states that for any structure D and any probability distribution P, it is possible to construct a representation map with average performance measures less than $H(D) + 1$ (i.e., close to the lower bound of part (i)).

For the example representation map 3, $\text{avg}(m') = 2 = H(D) + 1$. Part (ii) states that it is possible to construct a representation map for D with average performance measures less than 2. The representation map $\rho(d_1) = 0$, $\rho(d_2) = 1$ has average performance measures $\text{avg}(m) = \text{avg}(m') = 1 < H(D) + 1$. Construction of an efficient representation map involves assigning short representations to the more probable data bases.

For the given probability distribution the average performance measures of the enumerative field representation map and the positional field representation map for tables are much greater than $H(D) + 1 \cong 4$. The following representation map performs much better by assigning short representations to the more likely tables:

$\rho((7,7,7)) = \{1\}$

$\rho((6,7,7)) = \{001\}$

$\rho((7,6,7)) = \{010\}$

$\rho((7,7,6)) = \{011\}$.

The representation of any other table d is the sequence 000 followed by the concise field representation of length 9 for table d. The average performance :.easures for this representation map are $avg(m) = avg(m') = \frac{1}{2} \cdot 1 + 3 \cdot \frac{1}{8} \cdot 3 + 508 \cdot \frac{1}{8 \cdot 508} \cdot 12 = 3\frac{1}{8}$, which is certainly within 1 of H(D).

Proof of Theorem 4:

(i)
$$H(D) - avg(m) = -\sum_{d \in D} P(d) \log_{|B|} P(d) - \sum_{d \in D} P(d)|m(d)|$$

$$= \sum_{d \in D} P(d) \log_{|B|} \frac{1}{P(d)|B|^{|m(d)|}}$$

$$\leq (\log_{|B|} e) \sum_{d \in D} P(d) \frac{1}{P(d)|B|^{|m(d)|}} - 1,$$

since $\log_{|B|} x < (\log_{|B|} e)(x-1)$

$$\leq (\log_{|B|} e)((\sum_{d \in D} |B|^{-|m(d)|}) - 1)$$

$$\leq 0$$

where the final inequality follows from Theorem 2.

This proof is similar to proofs of the source coding theorem of information theory (Gallager[11]).

(ii) Huffman encoding is a procedure for constructing short representations for highly probable data bases and longer representations for less probable data bases (Huffman[12] or Gallager[11]). The Huffman encoding of a collection of data bases has disjoint closures and has average memory performance less than H(D) + 1. To construct a Huffman representation map for D, the system programmer must know first the exact probability distribution P on D. Other preconstructed universal representation sets perform almost as well as Huffman representation maps, provided the shorter preconstructed representations are assigned to the more probable data bases.[13]

2.6 RETRIEVAL PERFORMANCE

The information retrieval performance of a retrieval system is of interest to the

user and the system programmer. The user is concerned with the time it takes to obtain the answer to his information retrieval question. The retrieval time is not only a matter of user convenience but also a matter of efficient utilization of computer resources. If retrieval times are relatively long, then the system must be maintained on the computer for a relatively long period of time to allow a user to accomplish a given task. The length of time that the system must be maintained is reflected in the cost of the service to the user.

One measure of retrieval performance is the number of accesses made to memory by a retrieval algorithm. The number of memory accesses made by an algorithm before halting is one direct indication of the retrieval time. This memory access measure is similar to a measure used by Minsky and Papert[3] and by Elias.[8]

Retrieval algorithms that do not alter the memory state (i. e., do not write in memory) during their operation will be investigated first. Later it will be shown that retrieval algorithms that write in memory during their operation do not perform as well as read-only algorithms in the measures considered.

The number of accesses to memory will depend not only on the retrieval algorithm that is used but also on the state of memory when the algorithm is started. Consider example retrieval problem and retrieval system 3.

Before printing the answer '2' and halting, this algorithm will read a value 1 if the memory state is 011, 101, or 111 and will read the sequence of values 01 if the memory state is 100, or 110. Before printing the answer '1' and halting, this algorithm will read the sequence of values 00 if the memory state is 000 or 010.

Let $a_\lambda(\mu)$ be the sequence of values read by retrieval algorithm $\boldsymbol{a}_\lambda$ before halting when the memory state is $\mu \in \overline{\rho_*(D)}$. In example 3, $a_\lambda(000) = a_\lambda(010) = 00$, $a_\lambda(011) = a_\lambda(101) = a_\lambda(111) = 1$, and $a_\lambda(100) = a_\lambda(110) = 01$.

For a data base $d \in D$ let $A_\lambda(d)$ be one of the shortest sequences of values read by $\boldsymbol{a}_\lambda$ when data base d is represented in memory and let $A'_\lambda(d)$ be one of the longest.

$$A_\lambda(d) = \Psi \in \underline{\bigcup_{\mu \in \rho_*(d)}} \{a_\lambda(\mu)\} \text{ such that } \forall\, \Psi' \in \underline{\bigcup_{\mu \in \rho_*(d)}} \{a_\lambda(\mu)\},\ |\Psi| \leq |\Psi'|$$

$$A'_\lambda(d) = \Psi' \in \underline{\bigcup_{\mu \in \rho_*(d)}} \{a_\lambda(\mu)\} \text{ such that } \forall\, \Psi \in \underline{\bigcup_{\mu \in \rho_*(d)}} \{a_\lambda(\mu)\},\ |\Psi| \leq |\Psi'|.$$

Similarly, let $\alpha_\lambda(r)$ be one of the shortest sequences of values read by $\boldsymbol{a}_\lambda$ before printing answer r and let α'_λ be one of the longest.

$$\alpha_\lambda(r) = \Psi \in \bigcup_{d \in D | \lambda d = r} \{A_\lambda(d)\} \text{ such that } \forall\, \Psi' \in \bigcup_{d \in D | \lambda d = r} \{A_\lambda(d)\},\ |\Psi| \leq |\Psi'|$$

$$\alpha'_\lambda(r) = \Psi' \in \bigcup_{d \in D | \lambda d = r} \{A_\lambda(d)\} \text{ such that } \forall\, \Psi \in \bigcup_{d \in D | \lambda d = r} \{A_\lambda(d)\},\ |\Psi| \leq |\Psi'|.$$

In example 3, $A_\lambda(d_1) = A'_\lambda(d_1) = 00$, $A_\lambda(d_2) = 1$, and $A'_\lambda(d_2) = 01$. In example 3 only d_1

gives answer 1 and only d_2 gives answer 2, so $a_\lambda(1) = a'_\lambda(1) = 00$, $a_\lambda(2) = 1$, and $a'_\lambda(2) = 01$.

If $a_\lambda(r) = a_\lambda(r^*)$, then $r = r^*$, since a deterministic algorithm can only print a single answer after reading a single sequence of values. Also, for $r \neq r^*$, the sequence $a_\lambda(r)$ is not a prefix of the sequence $a_\lambda(r^*)$, since after reading the sequence of values $a_\lambda(r)$ the algorithm $\boldsymbol{a}_\lambda$ will print answer r and halt without reading more values. Using these properties, Elias has proved the following theorem.

Theorem 5 (Elias[1])

Let (D, Λ) be a retrieval problem, and let the measures $|a_\lambda(r)|$ be as defined above. If $|B|$-ary system $(\rho, \boldsymbol{a}_\Lambda)$ solves (D, Λ), then

$$\forall \lambda \in \Lambda, \quad \sum_{r \in \lambda D} |B|^{-|a_\lambda(r)|} \leq 1.$$

Theorem 5 is an inequality on a summation of terms composed of the memory alphabet size raised to a negative exponent. Corresponding to each answer r in λD, the answer space of question λ, there is one term in the summation with negative exponent $|a_\lambda(r)|$.

For the example retrieval system 3, the summation of Theorem 5 becomes

$$\sum_{r \in \{1,2\}} 2^{-|a_\lambda(r)|} = 2^{-|00|} + 2^{-|1|} = \frac{3}{4}$$

and Theorem 5 is obeyed with strict inequality.

For the enumerative retrieval system 1a for tables and for retrieval question λ_1: "What is the value of the first entry?" $a_\lambda(0) = a'_\lambda(0) = 000$, $a_\lambda(1) = a'_\lambda(1) = 001$, $a_\lambda(2) = a'_\lambda(2) = 010, \ldots,$ $a_\lambda(7) = a'_\lambda(7) = 111$. The summation of Theorem 5 becomes

$$\sum_{r \in [0,7]} 2^{-|a(r)|} = 8 \cdot 2^{-3} = 1$$

and Theorem 5 is obeyed with equality.

For the positional field retrieval system 1b for tables and for retrieval question λ_1, $a_\lambda(7) = a'_\lambda(7) = 1$, $a_\lambda(6) = a'_\lambda(6) = 01$, $a_\lambda(5) = a'_\lambda(5) = 001, \ldots, a_\lambda(0) = a'_\lambda(0) = 00000001$. The summation of Theorem 5 becomes

$$\sum_{r \in [0,7]} 2^{-|a_\lambda(r)|} = 2^{-1} + 2^{-2} + 2^{-3} + \ldots + 2^{-8} = 1 - 2^{-8} = 255/256$$

and Theorem 5 is obeyed with strict inequality.

Theorem 5 is a statement about distributions on the measure $|a_\lambda(r)|$ for any retrieval

system solving (D, Λ). Not all of the answers in λD can have short retrieval times, and if some of the answers in λD have short retrieval times then others must have relatively long retrieval times. The term in the summation corresponding to an answer r with a short retrieval time $|\alpha(r)|$ will have a small negative exponent $-|\alpha(r)|$ and hence a relatively large value. For the inequality to be satisfied, the other terms in the summation corresponding to other answers $r^* \in \lambda D$ must have relatively small values and hence relatively large negative exponents $-|\alpha_\lambda(r^*)|$. When the exponent $|\alpha_\lambda(r^*)|$ is relatively large, $\alpha_\lambda(r^*)$, the shortest possible sequence of reads for answer r^*, is relatively long.

Theorem 5 is proved by noticing that

(i) $\alpha_\lambda(r) = \alpha_\lambda(r^*) \rightarrow r = r^*$

(ii) $\bigcup_{r \in \lambda D} \{\alpha_\lambda(r)\}$ is a prefix set

and then using a well-known property of prefix sets (see Gallager[11]).

As with Theorem 2 for memory, Theorem 5 deals directly with the measure $|\alpha_\lambda(r)|$, but it is difficult to fathom the implications of the inequality for system performance parameters of direct interest to system user or system programmer.

The user with a static retrieval problem deals with the single data base d observed by the data gatherer for a long period of time. To retrieve the answer to question λ, he must wait for $|a_\lambda(\mu)|$ accesses to memory if the data gatherer placed representation $m \in \rho(d)$ for the observed data base d in memory and other users filled in unspecified cells so that $\delta(\mathcal{M}) = \mu \in \overline{\rho_*(d)}$. If he is lucky, $|a_\lambda(\mu)| = |A_\lambda(d)|$ and the retrieval time for question λ about data base d will be as short as possible. If he is not so lucky, $|a_\lambda(\mu)| = |A'_\lambda(d)|$ and the retrieval time will be longer. In either case the user of a system would be interested in the longest time that he might have to wait for the answer to his question λ. The following theorem proved by Elias deals with the worst-case parameters of direct interest to a user.

<u>Theorem</u> 6 (Elias[1])

Let (D, Λ) be a retrieval problem and let the measures $|A_\lambda(d)|$ and $|\alpha_\lambda(r)|$ be as defined above.

(i) If $|B|$-ary system $(\rho, \boldsymbol{\alpha}_\Lambda)$ solves (D, Λ), then

$$\forall \lambda \in \Lambda, \quad \max_{d \in D} |A_\lambda(d)| \geq \max_{r \in \lambda D} |\alpha_\lambda(r)| \geq \lceil \log_{|B|} |\lambda D| \rceil.$$

(ii) There is a $|B|$-ary system $(\rho, \boldsymbol{\alpha}_\Lambda)$ solving (D, Λ) such that

$$\forall \lambda \in \Lambda, \quad \max_{r \in \lambda D} |\alpha'_\lambda(r)| = \max_{d \in D} |A'_\lambda(d)| = \lceil \log_{|B|} |\lambda D| \rceil.$$

Part (i) states that in any retrieval system solving (D, Λ) a user must wait for at

least $\lceil \log_{|B|} |\lambda D| \rceil$ accesses to memory before obtaining the answer to question λ about some data base.

For the example problem 3, the lower bound on the worst-case retrieval performance measures for a binary memory alphabet is $\lceil \log_{|B|} |\lambda D| \rceil = \lceil \log_2 |\{1, 2\}| \rceil = 1$. For the example retrieval system 3, before obtaining the answer to question λ about data base d_1, the user must wait for two memory accesses ($\alpha_\lambda(1) = \alpha'_\lambda(1) = 00$).

For the table example problem 1 and the retrieval question λ_1, the lower bound on the worst-case retrieval performance measure for a binary memory alphabet is $\lceil \log_2 |\{0, 1, \ldots, 7\}| \rceil = 3$. With the enumerative retrieval system 1a, the user never needs to wait for more than 3 memory accesses before obtaining the answer to his retrieval question. With the positional field retrieval system 1b, the user may have to wait for 8 memory accesses before obtaining the answer to his retrieval question ($\alpha_\lambda(0) = 00000001$).

Part (ii) states that for any retrieval problem, there is a retrieval system that attains the lower bound of part (i) in all retrieval performance measures.

For the example retrieval problem 3, the lower bound for binary-valued memory cells is 1. The retrieval system $\rho(d_1) = \{0\}$, $\rho(d_2) = \{1\}$, α_λ: read cell 1; if value = 0 print '1' and halt; print '2' and halt; has retrieval performance measures of 1. For the table example 1, the lower bound for binary valued memory cells is 3. The enumerative retrieval system 1a has retrieval performance measures of 3.

For any retrieval problem one system that attains the bounds uses a representation map which is a concatenation of enumerative encodings of the answers to the questions $\lambda_j \in \Lambda$ into fields of $\lceil \log_{|B|} |\lambda_j D| \rceil$ memory cells. The retrieval algorithm α_j to answer question λ_j reads and remembers the values in the jth field, determines the proper answer from the enumerative encoding code book for answer space $\lambda_j D$, prints the answer, and halts having made $\lceil \log_{|B|} |\lambda_j D| \rceil$ memory accesses. The proof of part (i) is similar to the proof of Theorem 3 part (i).

The worst-case performance measures of Theorem 6 are of interest to a user with a static retrieval problem, since he might find that his memory state causes the longest retrieval times. Average performance measures are more appropriate when in a single system the existing data base is changed from time to time (as it might be if the user can update the existing data base).

Let P be a probability distribution on D, where P(d) is the fraction of time a user expects to be dealing with the data base d. The probability distribution P on D induces a probability distribution P_λ on λD defined by

$$P_\lambda(r) = \sum_{d \in D | \lambda d = r} P(d).$$

The probability $P_\lambda(r)$ is the fraction of time a user expects the answer $r \in \lambda D$ from the retrieval algorithm for question λ.

For example retrieval problem 3, $\{d \mid \lambda d=1\} = \{d_1\}$ and $\{d \mid \lambda d=2\} = \{d_2\}$. The probability distribution $P(d_1) = \frac{1}{2}$, $P(d_2) = \frac{1}{2}$ on D induces probability distribution $P_\lambda(1) = P(d_1) = \frac{1}{2}$, $P_\lambda(2) = P(d_2) = \frac{1}{2}$ on the answer space $\{1, 2\}$ for question λ.

For the table retrieval problem 1, consider the probability distribution P on D defined by $P((7,7,7)) = \frac{1}{2}$, $P((6,7,7)) = P((7,6,7)) = P((7,7,6)) = \frac{1}{8}$, and for any other table d, $P(d) = 1/8 \cdot 508$. For question $\lambda = \lambda_1$, $P_\lambda(0) = 8/508$, since 64 different tables d, each with probability $P(d) = 1/8 \cdot 508$, give the answer 0 to question $\lambda = \lambda_1$. A similar analysis shows that $P_\lambda(1) = P_\lambda(2) = \ldots = P_\lambda(5) = 8/508$, $P_\lambda(6) = 1/8 + 63/8 \cdot 508$, and $P_\lambda(7) = 3/4 + 61/8 \cdot 508$.

Define average retrieval performance measures $\text{avg}(A_\lambda)$ and $\text{avg}(\alpha_\lambda)$ as follows.

$$\text{avg}(A_\lambda) = \sum_{d \in D} P(d) \left| A_\lambda(d) \right|$$

$$\text{avg}(\alpha_\lambda) = \sum_{r \in \lambda D} P_\lambda(r) \left| \alpha_\lambda(r) \right|.$$

Define $\text{avg}(A'_\lambda)$ and $\text{avg}(\alpha'_\lambda)$ in a similar manner on the measures $|A'_\lambda|$ and $|\alpha'_\lambda|$. The quantity $\text{avg}(A_\lambda)$ is the average time a user must wait for the answer to question λ when for each data base d he waits the minimum time $|A_\lambda(d)|$ for <u>his</u> system. Note that $\text{avg}(A_\lambda) \geq \text{avg}(\alpha_\lambda)$, since

$$\left| \alpha_\lambda(r) \right| = \min_{d \in D \mid \lambda d = r} \left| A_\lambda(d) \right|.$$

For example retrieval system 3, the average measures are $\text{avg}(A_\lambda) = \text{avg}(\alpha_\lambda) = \frac{1}{2} |\alpha_\lambda(1)| + \frac{1}{2} |\alpha_\lambda(2)| = 1\frac{1}{2}$ and $\text{avg}(A'_\lambda) = \text{avg}(\alpha'_\lambda) = \frac{1}{2} |\alpha'_\lambda(1)| + \frac{1}{2} |\alpha'_\lambda(2)| = 2$. For the enumerative retrieval system 1a for tables, the average retrieval measures are all 3. For the positional field retrieval system 1b, the measures are all $P_\lambda(7) \cdot |\alpha_\lambda(7)| + P_\lambda(6)|\alpha_\lambda(6)| + \ldots + P_\lambda(0)|\alpha_\lambda(0)| \cong 1.4$.

The following theorem proved by Elias deals with these average retrieval performance measures.

<u>Theorem</u> 7 (Elias[1])

Let (D, Λ) be a retrieval problem, let P be a probability distribution on D, and let the average retrieval performance measures be as defined above.

(i) If $|B|$-ary system (ρ, α_Λ) solves (D, Λ), then

$$\forall \lambda \in \Lambda \quad \text{avg}(A_\lambda) \geq \text{avg}(\alpha_\lambda) \geq H(\lambda D)$$

where $H(\lambda D) = -\sum_{r \in \lambda D} P_\lambda(r) \log_{|B|} P_\lambda(r)$.

(ii) There is a $|B|$-ary retrieval system solving (D, Λ) such that

$$\forall \lambda \in \Lambda \quad \text{avg}(A'_\lambda) \leq \text{avg}(\alpha'_\lambda) < H(\lambda D) + 1.$$

Part (i) states that the average retrieval performance measures for any $|B|$-ary retrieval system solving (D, Λ) are no less than the quantity $H(\lambda D)$.

For example retrieval problem 3 and binary retrieval systems the lower bound of part (i) is $H(\lambda D) = -P_{\lambda_1}(1) \log_2 P_{\lambda_1}(1) - P_{\lambda_1}(2) \log_2 P_{\lambda_1}(2) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1$. The measures for example retrieval system 3 were $\text{avg}(A'_\lambda) = \text{avg}(a'_\lambda) = 2$ and $\text{avg}(A_\lambda) = \text{avg}(a_\lambda) = 1\frac{1}{2}$.

For the table retrieval problem 1 and binary systems, the lower bound of part (i) is $H(\lambda D) \cong 1.2$. For the enumerative retrieval system 1a the average retrieval measures were 3. For the positional field retrieval system 1b the measures were approximately 1.4.

Part (ii) states that for any retrieval problem there is a retrieval system with average retrieval performance measures less than $H(\lambda D) + 1$ (i. e., close to the lower bound of part (i)).

The performance measures for example retrieval system 3, $\text{avg}(a) = 1\frac{1}{2}$, $\text{avg}(a') = 2$, were not both within 1 of the bound $H(D) = 1$. Part (ii) states that there is a retrieval system solving the problem with better average performance measures. The system $\rho(d_1) = 0$, $\rho(d_2) = 1$, a_λ: read cell 1; if value = 0 then print '1' and halt; print '2' and halt; has average measures of 1 which is less than $H(\lambda D) + 1 = 2$.

The performance measures for the enumerative field retrieval system 1a were 3 and were not within 1 of the bound $H(\lambda D) \cong 1.2$. The performance measures for the positional field retrieval system 1b were approximately 1.4 and were less than $H(\lambda D) + 1 \cong 2.2$.

For any retrieval problem one system that attains the bound is the following. The representation map is the concatenation of Huffman encodings of the answers to questions $\lambda_j \in \Lambda$ into fixed length fields of memory cells sufficiently long to hold the longest Huffman encoded answer to question λ_j. The retrieval algorithm a_j to answer question λ_j reads and remembers the values of the Huffman encoding in the jth field, determines the answer from the Huffman encoding code book for answer space $\lambda_j D$, prints the answer and halts. On the average this algorithm will access less than $H(\lambda D) + 1$ memory cells. The proof of part (i) is similar to the proof of Theorem 4 part (i).

Retrieval algorithms that write as well as read in memory should certainly not change the existing data base represented in memory as a result of their operation. When started with memory in a state $\mu \in \overline{\rho_*(d)}$ representing data base d, the retrieval algorithm a_λ must print the answer λd and halt with memory in a state $\mu' \in \overline{\rho_*(d)}$ also representing the data base d.

The sequence of values read by an algorithm that both writes and reads before halting must still satisfy the properties investigated for read-only algorithms. In particular, Theorems 5, 6, and 7 remain true for the sequence of read accesses to memory. Algorithms that make additional memory accesses for writing cannot perform as well as some read-only algorithms.

III. DYNAMIC RETRIEVAL SYSTEMS WITH UPDATES

3.1 UPDATES

Many practical situations involving data (see Knuth[4] for some examples) are adequately modeled by a finite collection of possible data bases, only a single one of which is actually observed, and a finite collection of retrieval questions (i.e., a static retrieval problem). In these situations a static retrieval system provides a convenient method for retrieving information about the single observed data base. Other situations involving data are not well modeled by a static retrieval problem. In many situations the data base is changed from time to time. For example, a potential retrieval system user might want to record and retrieve the most recent test scores of three students. He might need the capability of recording a new set of test scores for all students, changing a single score if a student's test is misgraded, or adding new students and dropping old students in the course record.

Many of these data problems only differ from static retrieval problems in that the existing data are changed from time to time. These situations are adequately modeled by a finite collection of data bases, of which only a single one is observed _at any given time_, a finite collection of retrieval questions, and a set of updates. In some updates, called _total updates_, the new data base is independent of the previous data base. This is the case, for example, when new test scores for three students are entered in the record and the outdated previous test scores are removed. In other updates, called _partial updates_, the new data base depends upon (i.e., is some modification of) the previous data base. This is the case when a single student's regraded test score is entered in the record and the incorrect score is removed.

A convenient notation indicating the updates required by a user is a set of update maps $U_K = \{U_k:D \to D \mid k \in K\}$ indexed by the members of an index set K. The meaning of the _update_ $U_k:D \to D$ is that if the existing data base prior to update U_k is d, then the new data base after update U_k is $U_k(d)$. In a _total_ update $U_k:D \to d$ the new data base d following the update is completely specified, while in a _partial_ update $U_k:D \to D$ such that $|U_k(D)| > 1$ the new data base d following the update depends upon the existing data base prior to the update. When the notation is unambiguous total updates $U_d:D \to d$ will be indexed with the specified data base.

A _dynamic retrieval problem_ [denoted (D,Λ,U_K)] includes a collection of possible data bases D, of which only a single one is observed at any given time, a collection of retrieval questions Λ, and a set of updates U_K. A user with a dynamic retrieval problem (D,Λ,U_K) could use a static retrieval system (ρ, α_Λ) solving (D,Λ). To make update U_k of the existing data base, the user passes the information (i.e., the index $k \in K$) to the data gatherer, who shuts down the retrieval system, calculates the new data base d from the information given to him, and when necessary (i.e., for partial updates) from the existing data base represented in memory, determines possible

representations for the new data base from the representation map, places one of the representations in memory, and finally brings the retrieval system back up. When updates are rare, this method might be workable, although somewhat awkward. When updates are frequent, this method is unworkable. A system in which frequent calls to the data gatherer and subsequent system shutdowns are unnecessary would be more satisfactory.

A computer-implemented dynamic retrieval system solving a dynamic retrieval problem provides (a) a method for representing any single observed data base in computer memory, (b) a method for answering any retrieval question about the observed data base represented in computer memory, and (c) a method for modifying or updating the existing data base represented in computer memory. The first two properties were discussed in section 2.3.

(c): <u>A method for updating the existing data base represented in computer memory.</u>

A computer-implemented system should not force the user to change his view of data substantially. In particular, a computer-implemented dynamic retrieval system should allow a user to make update U_k of the existing data base by indicating the index $k \in K$, and hence must provide for each update $U_k \in U_K$ an update algorithm $\mathscr{U}_k \in \mathscr{U}_K = \{\mathscr{U}_k | k \in K\}$ making update U_k on the existing data base. When started, the algorithm $\mathscr{U}_k$ must halt in finite time with memory in a state representing the new data base $U_k(d)$ no matter which data base $d \in D$ was previously observed, no matter which representation $m \in \rho(d)$ for data base d was placed in memory $\mathscr{M}_L$, no matter which memory $\mathscr{M}_L \in \mathscr{M}^*$ is connected to the algorithm (provided it is sufficiently long), and no matter how other users choose to fill in cells with no value specified by the representation m. The formal definition of an update algorithm $\mathscr{U}_k = \{s_k, S_k, \nu_k, \theta_k, \sigma_k\}$ is similar to the formal definition of a retrieval algorithm $\mathscr{a}_j$ except for the absence of an output function Ω. An update algorithm $\mathscr{U}_k$ is defined to make <u>update</u> U_k <u>on</u> D <u>using</u> ρ <u>with</u> $\mathscr{M}_L$ iff $L \geq L(\rho)$ and $\forall\ d \in D$, $\forall\ \mu \in \overline{\rho_L(d)}$ when connected to memory $\mathscr{M}_L$ in state $\mathcal{S}(\mathscr{M}_L, t{=}1) = \mu$ and started, the algorithm $\mathscr{U}_k$ <u>accesses</u> addresses in $[1, L]$ and halts in finite time t_f with memory in state $\mathcal{S}(\mathscr{M}_L, t_f) \in \overline{\rho_L(U_k(d))}$.

The fact that when connected to $|B|$-ary memory $\mathscr{M}_L$ the update algorithms $\mathscr{U}_K$ are required to operate properly only for initial memory states $\mu \in \overline{\rho_L(D)}$ merits some consideration, since $\overline{\rho_L(D)}$, the set of possible memory states after some representation for some data base has been placed in memory, may not be the set B^L of all possible memory states. Partial updates of initial memory states $\mu \in B^L - \overline{\rho_L(D)}$ are meaningless, since there is no previously existing data base. Total updates U_d of initial memory states $\mu \in B^L - \overline{\rho_L(D)}$ might be more meaningful. If the total update algorithm $\mathscr{U}_d$ were required to leave memory in a state $\mu \in \overline{\rho_L(d)}$ representing data base d for any initial state $\mu \in B^L$, the data gatherer, after observing data base d, could start total update algorithm $\mathscr{U}_d$ instead of determining possible representations for d from the

representation map and directly writing into memory a representation for d. But the algorithms $\mathscr{U}_d$ designed to work properly for any initial memory state $\mu \in B^L$ would be more complex because they must operate on a larger set of initial memory states. Update algorithms will be operated primarily by a user updating the existing data base; in this case memory is in some state in $\overline{\rho_L(D)}$ representing the existing data base. Designing more complex algorithms in order to simplify the data gatherer's one-time initialization task does not seem worthwhile.

A <u>dynamic retrieval system</u> $(\rho, \mathscr{A}_\Lambda, \mathscr{U}_K)$ including a set of retrieval algorithms and a set of update algorithms connected to memory $\mathscr{M}_{L(\rho)}$ is defined to <u>solve</u> a dynamic retrieval problem (D, Λ, U_K) iff $\mathscr{A}_\Lambda$ answers Λ about D and $\mathscr{U}_K$ makes update U_K on D using ρ with $\mathscr{M}_{L(\rho)}$. Let $R(D, \Lambda, U_K)$ be the set of all dynamic retrieval systems solving (D, Λ, U_K). Note that the investigation of update performance can focus on ρ and $\mathscr{U}_K$ because whenever $\mathscr{U}_K$ makes update U_K on D using ρ with $\mathscr{M}_{L(\rho)}$ there exists some $\mathscr{A}_\Lambda$ completing the system $(\rho, \mathscr{A}_\Lambda, \mathscr{U}_K)$ by the definition of representation map.

As with static retrieval systems, dynamic retrieval systems could be defined more generally to include <u>any</u> memory $\mathscr{M}_L$, where $L \geq L(\rho)$, so that other users could be assigned high-order, as well as low-order, addresses with unspecified values, and update algorithms could access addresses greater than $L(\rho)$. The results to be presented are true for this more general case, but inclusion of this case in formal proofs does not seem worthwhile because of additional notational complexity.

The performance of a <u>complete</u> set of total update algorithms $\mathscr{U}_D = \{\mathscr{U}_d \mid d \in D\}$ including one total update algorithm $\mathscr{U}_d$ for each data base in the collection is investigated first, followed by application of the results to partial update algorithms. The following example problem will be instructive for illustrating results.

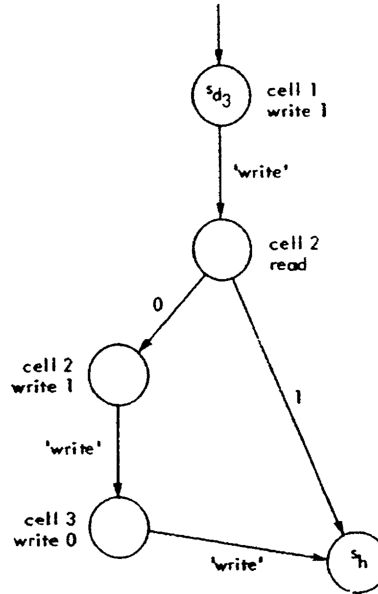
<u>Example 4</u>

Let $D = \{d_1, d_2, d_3\}$, $\Lambda = \{\lambda$: "What is the index of the observed data base?"$\}$ and $U_D = \{U_{d_1}, U_{d_2}, U_{d_3}\}$, where $U_{d_i}: D \rightarrow d_i$ are total updates.

Consider the binary representation map ρ for D defined by $\rho(d_1) = \{010\}$, $\rho(d_2) = \{001\}$, and $\rho(d_3) = \{110\}$. Connected to $\mathscr{M}_{L(\rho)} = \mathscr{M}_3$ the following update algorithm $\mathscr{U}_{d_3}$ halts with memory in state $110 \in \overline{\rho_3(d_3)}$ when started with any initial memory state $\mu \in \overline{\rho_3(D)}$.

$\mathscr{U}_{d_3}$: write value 1 in cell 1
read cell 2
if value 1 then halt
write value 1 in cell 2
write value 0 in cell 3
halt

The state transition diagram is given in Fig. 2. Note that if started with initial memory state $011 \notin \overline{\rho_3(D)}$, the algorithm will halt with memory state 111 which does not represent d_3.



KEY:

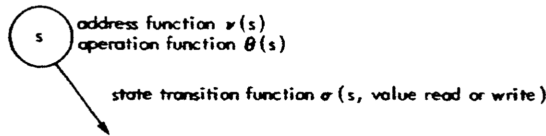


Fig. 2. Update algorithm $\mathscr{U}_{d_3}$ of example 4.

3.2 PERFORMANCE MEASURES FOR TOTAL UPDATES

Since the number of memory cells read by a retrieval algorithm before printing an answer and halting proved to be an appropriate measure for retrieval performance of a system, it is worthwhile to consider whether or not the number of memory cells read by a total update algorithm $\mathscr{U}_d$ before halting is an appropriate measure for update performance of a system including total update algorithms. Although an analysis of read access will prove that this is not an appropriate performance measure, it will indicate that write accesses might be a more appropriate performance measure. But analysis of write accesses will prove that write accesses is not an appropriate performance measure. The analysis will lead, however, to consideration of total accesses (reads and writes), and Theorems 10, 11, 12, and 13 will show that total accesses is indeed an appropriate measure of update performance.

First, we consider read accesses. For a dynamic retrieval system $(\rho, \mathscr{A}_\Lambda, \mathscr{U}_D)$, let $r_d(\mu)$ be the sequence of cell values read by the update algorithm $\mathscr{U}_d$ before halting when started with initial memory state $\mu \in \overline{\rho_*(D)}$. If the initial memory state is 010 when the update algorithm $\mathscr{U}_{d_3}$ of example system 4 is started, the algorithm will write into cell 1, read from cell 2 the value 1, and halt. The sequence r_{d_3} (010) is 1. Similarly,

$r_{d_3}(001) = 0$ and $r_{d_3}(110) = 1$. The number of memory cells read by an update algorithm $\mathcal{U}_d$ before halting when started with initial memory state $\mu \in \overline{\rho_*(D)}$ is $|r_d(\mu)|$, the length of the sequence $r_d(\mu)$.

The following theorem proves that the number of read accesses to memory by a total update algorithm is <u>not</u> an appropriate performance measure.

<u>Theorem</u> 8

Let (D, Λ, U_D) be a dynamic retrieval problem including a complete set of total updates, let ρ be a representation map for D, and let the measure $|r_d(\mu)|$ be as defined above. There is a system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_D)$ solving (D, Λ, U_D) such that

$$\forall\, d \in D \quad \forall\, \mu \in \overline{\rho_*(D)} \quad |r_d(\mu)| = 0.$$

The theorem states that for any dynamic retrieval problem and any representation map for the data structure, it is possible to construct a system in which the update algorithms make no read accesses to memory. For the example problem 4 and the representation map ρ, an algorithm $\mathcal{U}'_{d_3}$ to update memory to a state representing d_3 without reading memory cells is the following.

$\mathcal{U}'_{d_3}$: write value 1 in cell 1
write value 1 in cell 2
write value 0 in cell 3
halt

For this algorithm $r_{d_3}(010) = r_{d_3}(001) = r_{d_3}(110) = \emptyset$ and $|r_{d_3}(\mu)| = 0 \;\forall\, \mu \in \overline{\rho_*(D)}$. The algorithm simply writes the representation 110 for d_3 into memory and halts without reading memory cells.

<u>Proof of Theorem</u> 8: Let ρ be any representation map for D. Let $m = \{(n_1, m(n_1)), (n_2, m(n_2)), \ldots, (n_{|m|}, m(n_{|m|}))\} \in \rho(d)$ be a representation for d. Let the update algorithm $\mathcal{U}_d$ be $\{s_1, S, \nu, \theta, \sigma\}$, where

$$S = \{s_1, s_2, \ldots, s_{|m|}, s_h\}$$

$$\nu(s_i) = n_i$$

$$\theta(s_i) = m(n_i)$$

$$\sigma(s_i) = s_{i+1} \qquad i = 1, 2, \ldots, |m|-1$$

$$\sigma(s_{|m|}) = s_h.$$

When placed in starting state s_1, the algorithm $\mathcal{U}_d$ writes the representation m for d into memory without reading memory cells and halts. Update algorithms $\mathcal{U}_{d'}$ for the other update maps $U_{d'}: D \to d'$ are similar. Since ρ is a representation map for D, there

exists a set of retrieval algorithms completing the system $(\rho, \mathscr{A}_\Lambda, \mathscr{U}_D)$.

Since any total update can be made by an algorithm without reading memory cells, the number of read accesses to memory is not an appropriate performance measure. Nevertheless, total update algorithms that make no read accesses write an entire representation into memory. We consider next whether or not the number of memory cells overwritten by a total update algorithm before halting is a more appropriate performance measure.

For a dynamic retrieval system $(\rho, \mathscr{A}_\Lambda, \mathscr{U}_D)$ let $w_d(\mu)$ be the sequence of cell values overwritten by the update algorithm $\mathscr{U}_d$ before halting when started with initial memory state $\mu \in \overline{\rho_*(D)}$. The values in the sequence $w_d(\mu)$ are the initial values originally in the cell immediately before being overwritten with some value by the algorithm $\mathscr{U}_d$. In the example system 4, when started with initial memory state 010, the update algorithm $\mathscr{U}_{d_3}$ performs a write on cell 1. The original value of cell 1 before being overwritten is 0. The update algorithm then reads the value of cell 2 and halts. The sequence $w_{d_3}(010)$ is 0. Similarly, $w_{d_3}(001)$ is 001 and $w_{d_3}(110)$ is 1.

The number of memory cells overwritten by an update algorithm $\mathscr{U}_d$ before halting when started with initial memory state $\mu \in \overline{\rho_*(D)}$ is $|w_d(\mu)|$ the length of the sequence $w_d(\mu)$. The following theorem proves that the number of write accesses to memory by an update algorithm is not an appropriate performance measure.

Theorem 9

Let (D, Λ, U_D) be a dynamic retrieval problem including a complete set of total updates, and let the measure $|w_d(\mu)|$ be as defined above. There is a system $(\rho, \mathscr{A}_\Lambda, \mathscr{U}_D)$ solving (D, Λ, U_D) such that

$$\forall d \in D \quad \forall \mu \in \overline{\rho_*(D)} \quad |w_d(\mu)| \leq 2.$$

Theorem 9 states that for any dynamic retrieval problem it is possible to construct a system in which the updating algorithms make no more than two write accesses to memory. For the example problem 4, a system using ρ and $\mathscr{U}_{d_3}$ would not always overwrite two or fewer memory cells because when started with memory state 001 $\mathscr{U}_{d_3}$ overwrites three memory cells ($w_{d_3}(001) = 001$). By using the representation map $\rho'(d_1) = 100$, $\rho'(d_2) = 010$, and $\rho'(d_3) = 001$, however, the following update algorithm $\mathscr{U}'_{d_3}$ never overwrites more than two memory cells before halting for any initial memory state.

```
U'd3:  read cell 1
       if value = 1 then write value 0 in cell 1 and
                         go to A
       read cell 2
       if value = 1 then write value 0 in cell 2 and
                         go to A
   A:  write value 1 in cell 3
       halt
```

If the initial memory state is 100, the algorithm $\mathscr{U}'_{d_3}$ when started reads cell 1, overwrites the value 1 in cell 1, overwrites the value 0 in cell 3, and halts with memory in a state representing d_3. The sequence $w_{d_3}(100)$ is 10. Similarly, $w_{d_3}(010)$ is 10 and $w_{d_3}(001)$ is 1. The algorithm $\mathscr{U}'_{d_3}$ never makes more than two write accesses to memory before halting.

Proof of Theorem 9: Let ρ be a positional representation map for D, where each representation in $\rho(D)$ is a function on the domain $N_{|D|} = \{1, 2, \ldots, |D|\}$. The representation for the ith data base specifies value 1 for the memory cell with address i and value 0 for all other memory cells with addresses in $N_{|D|}$.

$$\rho(d_i) = \{\{(1,0), (2,0), \ldots, (i-1,0), (i,1), (i+1,0), \ldots, (|D|,0)\}\}$$

Let the update algorithm $\mathscr{U}_{d_i}$ be $\{s_2, S, \nu, \theta, \sigma\}$ where

$$\nu(s_{2j}) = \nu(s_{2j+1}) = j$$

$$\theta(s_{2j}) = \text{'read'}$$

$$\theta(s_{2j+1}) = 0$$

$$\sigma(s_{2j}, 0) = s_{2j+2}$$

$$\sigma(s_{2j}, 1) = s_{2j+1}$$

$$\sigma(s_{2j+1}, B) = s_o \qquad \text{for } j = 1, 2, \ldots, |D|$$

and

$$\nu(s_o) = i$$

$$\theta(s_o) = 1$$

$$\sigma(s_o) = s_h.$$

When placed in starting state 2, the algorithm $\mathscr{U}_{d_i}$ reads the memory cells sequentially with addresses in $N_{|D|}$ until it finds the cell with value 1, rewrites this cell to value 0, writes cell i to value 1, and halts, having overwritten two memory cells.

If the read accesses to memory by an update algorithm $\mathscr{U}_d$ are unmeasured, the algorithm $\mathscr{U}_d$ can determine the state of memory before performing any writes without increasing the measured complexity. Such an algorithm could determine which representation $m \in \rho(d)$ for d differed from the existing state of memory in the fewest number of cell values and only overwrite those cells. The measure $|w_d(\mu)|$ becomes a measure of the distance between $\mu \in \overline{\rho_*(D)}$ and the closest representation $m \in \rho(d)$.

In summary, neither $|r_d(\mu)|$ nor $|w_d(\mu)|$ alone are appropriate measures of total update performance because for any problem one system needs no read accesses and

another system needs at most two write accesses for a total update. However, update algorithms reading no memory cells write an entire representation into memory, while update algorithms writing at most two memory cells often read most of memory. Thus the total number of cells accessed by an update algorithm may be a more appropriate performance measure.

3.3 TOTAL UPDATE PERFORMANCE

The update performance measure considered in this section is the number of memory accesses (reads and writes) made by a total update algorithm before halting. For a dynamic retrieval system $(\rho, \mathcal{a}_\Lambda, \mathcal{U}_D)$ let $\ell_d(\mu)$ be the sequence of cell values read or overwritten by update algorithm $\mathcal{U}_d$ when started with initial memory state $\mu \in \overline{\rho_*(D)}$. Note that $|\ell_d(\mu)|$, the length of the sequence $\ell_d(\mu)$, is the sum of $|r_d(\mu)|$, the number of read accesses, and $|w_d(\mu)|$, the number of write accesses to memory.

Consider algorithm $\mathcal{U}_{d_3}$ of example system 4.

Example 4a. Single Fixed-Length Representations

$\rho(d_1) = 010$

$\rho(d_2) = 001$

$\rho(d_3) = 110$

$\mathcal{U}_{d_3}$: write value 1 in cell 1
read cell 2
if value 1 then halt
write value 1 in cell 2
write value 0 in cell 3
halt

When started with initial memory state 010, the algorithm $\mathcal{U}_{d_3}$ overwrites the 0 already in cell 1, reads value 1 from cell 2, and halts, so that $\ell_{d_3}(010) = 01$. Similarly, $\ell_{d_3}(001) = 0001$ and $\ell_{d_3}(110) = 11$.

A tree corresponding to algorithm $\mathcal{U}_{d_3}$ may be constructed as follows. Consider the algorithm's operation on some initial memory state $\mu \in \overline{\rho_*(D)}$, say 010. Corresponding to the first memory access construct a unique root node labeled with the address accessed (cell 1) and operation performed (write 1) (see Fig. 3). From the root node construct an output branch labeled with the value overwritten (value 0). Connected to this output branch construct a node corresponding to the next memory access labeled with the address accessed (cell 2) and operation performed (read). From this node construct an output branch labeled with the value read (value 1). Connected to this output branch construct a terminal node (since the algorithm halts) labeled with the initial memory state (010) and the final memory state (110). Continue this process for other initial memory states, adding nodes and output branches where necessary. From a node

corresponding to a read access, construct an output branch labeled with the value read. From a node corresponding to a write access, construct an output branch labeled with the overwritten value (i.e., the cell value before being overwritten).

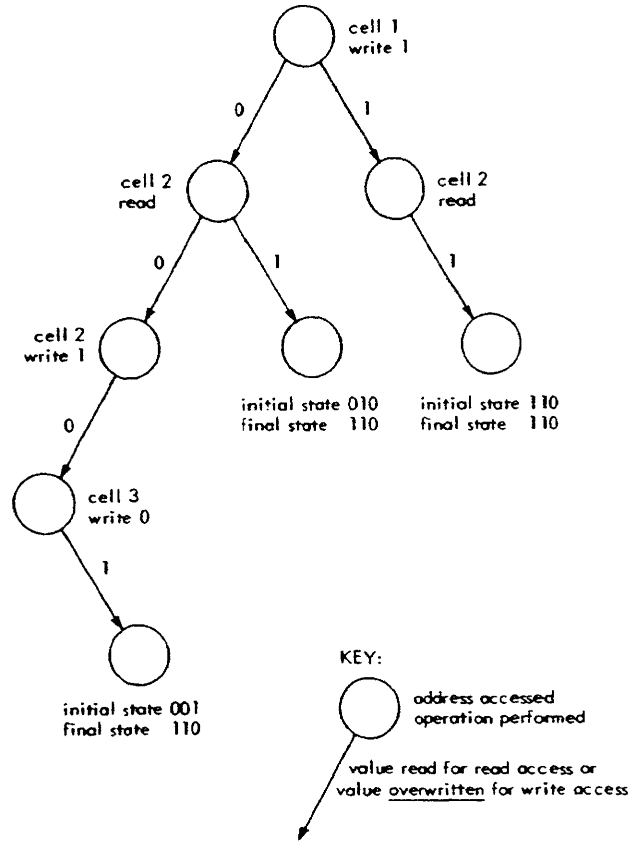


Fig. 3. Tree corresponding to update algorithm $\mathcal{U}_{d_3}$ of example 4.

We are concerned with finding optimum update strategies. Update algorithms $\mathcal{U}'_k$ that read a cell more than once, overwrite a cell more than once, read a cell already overwritten, or overwrite a cell with a value already read from the cell may all be improved to algorithms $\mathcal{U}_k$ that have some memory to save the appropriate values. The algorithms $\mathcal{U}_k$ will require fewer accesses and hence algorithms $\mathcal{U}'_k$ will not be considered.

Associated with any $|B|$-ary algorithm $\mathcal{U}$ is a tree composed of a set of nodes Q and a set of branches. Among the nodes Q is a set of terminal nodes Q_t and a set of interior nodes $Q_i = Q - Q_t$, including a distinguished root node q_o. Associated with each interior node q is an address label $\nu(q) \in N^+$ and an operation label $\theta(q) \in \{\text{'read'}\} \cup B$. Associated with each branch is a branch label from the alphabet B.

The action of algorithm $\mathcal{U}_d$ at an interior node q at time t is to read cell $\nu(q)$ if

$\theta(q)$ = 'read', or overwrite cell $\nu(q)$ with $\theta(q)$ if $\theta(q) \in B$ so that $\mathscr{M}(\nu(q), t+1) = \theta(q)$. The algorithm then leaves at time t+1 by the branch labeled $\mathscr{M}(\nu(q), t)$, the initial cell value prior to the access at time t at node q. Note that a cell value overwritten with $\theta(q)$ will not be known to the algorithm. Thus this tree is <u>not</u> the state transition diagram with appropriate labels (compare Figs. 2 and 3).

Associated with each node $q \in Q$ are two partial functions $\beta_q, \beta'_q \in B^\dagger$ constructed as follows. For the root node q_o, $\beta_{q_o} = \beta'_{q_o} = \emptyset$. If node q is connected to an output branch of node p with branch label b, $\beta_q = \beta_p \cup \{(\nu(p), b)\}$. If $\theta(p)$ = 'read', $\beta'_q = \beta'_p \cup \{(\nu(p), b)\}$ or if $\theta(p) \in B$, $\beta'_q = \beta'_p - \{(\nu(p), -)\} \cup \{(\nu(p), \theta(p))\}$, where $(\nu(p), -)$ is any pair in β'_p with domain value $\nu(p)$. Note that β_q and β'_q would not necessarily be partial functions if algorithms that read a cell already overwritten or overwrite a cell more than once were considered. The domain of the partial functions β_q and β'_q is the set of cell addresses already accessed before entering node q. The value $\beta_q(i)$ is the initial cell value in the initial memory state(s) causing the algorithm to enter node q, and $\beta'_q(i)$ is the cell value upon entering node q.

Let algorithm $\mathscr{U}_d$ make total update $U_d: D \to d$ on D using ρ with $|B|$-ary memory $\mathscr{M}_L$. Each node q of the tree corresponding to $\mathscr{U}_d$ defines two quotient sets M_q and M'_q, where

$$M_q = \{\mu \in \overline{\rho_L(D)} \mid \beta_q \subseteq \mu\}$$

$$M'_q = \{\mu \in B^L \mid \beta'_q \subseteq \mu \wedge (\mu - \beta'_q \cup \beta_q) \in \overline{\rho_L(D)}\}.$$

The set M_q is the set of initial memory states causing $\mathscr{U}_d$ to enter node q, and M'_q is the set of possible memory states upon entering node q.

Several properties of the partial functions β and quotient sets M are important.

Property (a): $\{M_q \mid q \in Q_t\}$ is a partition of $\overline{\rho_L(D)}$.

Any initial memory state in $\overline{\rho_L(D)}$ will cause the algorithm to operate along a unique path from the root node to a terminal node. More precisely, (i) $\bigcup_{q \in Q_t} M_q = \overline{\rho_L(D)}$ because the algorithm $\mathscr{U}_d$ must halt for any initial memory state in $\overline{\rho_L(D)}$, and (ii) a memory state $\mu \in \overline{\rho_L(D)}$ is a member of a unique quotient set M_q for $q \in Q_t$. To prove (ii) assume $\mu \in M_p$ and $\mu \in M_q$ for $p, q \in Q_t$. The common domain, Domain(β_p) $\cap$ Domain(β_q), is nonempty, since each domain includes $\nu(q_0)$. If $\beta_p(i) = \beta_q(i)$ for all i in the common domain, p = q, since there is no node where β_p and β_q correspond to different output branches. Suppose for some $i \in$ Domain(β_p) $\cap$ Domain(β_q) that $\beta_p(i) \neq \beta_q(i)$. Since $\beta_p \subseteq \mu$, $(i, \beta_p(i)) \in \mu$ and since $\beta_q \subseteq \mu$, $(i, \beta_q(i)) \in \mu$. This cannot be true because μ is a function on N_L assigning one range value to domain value i.

Let $q(\mu)$ be the unique $q \in Q_t$ for which $\mu \in M_q$ and let ℓ_q be the path length from root node q_0 to node q.

Property (b): $|\ell_d(\mu)| = \ell_{q(\mu)}$.
In the tree associated with algorithm $\mathscr{U}_d$, the path length $\ell_{q(\mu)}$ is equal to $|\ell_d(\mu)|$ the number of memory accesses by algorithm $\mathscr{U}_d$ before halting when started with initial memory state $\mu \in \overline{\rho_L(D)}$.

Property (c): $M'_q \subseteq \overline{\rho_L(d)}$ for $q \in Q_t$.
Update algorithm $\mathscr{U}_d$ must halt with memory in a state representing data base d, and M'_q is the set of possible memory states when algorithm $\mathscr{U}_d$ enters terminal node q.

Property (d): $|M_q| = |M'_q|$.
Any initial memory state causing algorithm $\mathscr{U}_d$ to enter node q is associated with a unique memory state upon entering node q, since the algorithm is deterministic. More precisely, define $\xi_q: M_q \to B^L$ to be $\xi_q(\mu) = \mu - \beta_q \cup \beta'_q$, and define $\xi_q^{-1}: M'_q \to B^L$ to be $\xi_q^{-1}(\mu) = \mu - \beta'_q \cup \beta_q$. That $\xi_q(\mu)$ is a sequence in B^L for $\mu \in M_q$ may be proved as follows. The domain of $\mu - \beta_q$ is N_L - Domain(β_q), since μ is a function on N_L and $\beta_q \subseteq \mu$. The domain of $\mu - \beta_q \cup \beta'_q$ [i.e., the domain of $\xi_q(\mu)$] is N_L, since Domain(β_q) = Domain(β'_q) by the definitions of β_q and β'_q. That $\xi_q^{-1}(\mu)$ is a sequence in B^L for $\mu \in M'_q$ is proved in a similar manner. To prove $|M_q| = |M'_q|$ it is sufficient to prove:

(i) $\xi_q(M_q) \subseteq M'_q$

(ii) $\xi_q^{-1}(M'_q) \subseteq M_q$

(iii) $\xi_q^{-1}(\xi_q(\mu)) = \mu$

(i.e., ξ_q is a one-one onto function from M_q to M'_q). Part (i) may be proved as follows. For $\mu \in M_q$, $\mu \in \overline{\rho_L(D)}$ and $\beta_q \subseteq \mu$ from the definition of M_q. Note that $\beta'_q \cap (\mu - \beta_q) = \emptyset$ because Domain($\mu - \beta_q$) = N_L - Domain(β_q) = N_L - Domain(β'_q) as shown above. Observe that $\xi_q(\mu) - \beta'_q \cup \beta_q = (\mu - \beta_q \cup \beta'_q) - \beta'_q \cup \beta_q = \mu - \beta_q \cup \beta_q = \mu \in \overline{\rho_L(D)}$, where the first equality follows from the definition of $\xi_q(\mu)$, the second equality follows from $\beta'_q \cap \mu - \beta_q = \emptyset$ as noted, and the third equality follows from $\beta_q \subseteq \mu$. Also observe that $\beta'_q \subseteq \xi_q(\mu) = \mu - \beta_q \cup \beta'_q$. Then $\xi_q(\mu) \in M'_q$ from the observations above and the definition of M'_q. Parts (ii) and (iii) may be proved in a similar manner.

If initial memory state $\mu \in \overline{\rho_L(D)}$ causes the algorithm $\mathscr{U}_d$ to enter node q in its operation, then the memory state upon entering node q is $\mu - \beta_q \cup \beta'_q = \xi_q(\mu)$, since the domain of β_q and β'_q is the set of cell addresses accessed, $\beta_q(i)$ is the initial value of cell i, and $\beta'_q(i)$ is the cell value upon entering node q. Define $\xi_d(\mu)$ to be $\xi_{q(\mu)}(\mu)$; the final memory state (after halting) for update algorithm $\mathscr{U}_d$ started with memory state μ.

The following theorem presents results for $|\ell_d(\mu)|$ when the representation map is subject to the following restrictions:

(i) Only a single representation is allowed for each data base.

(ii) A representation must specify values for exactly the first $L = L(\rho)$ memory cells.

Theorem 10

Let (D, Λ, U_D) be a dynamic retrieval problem including a complete set of total updates, and let the measure $|\ell_d(\mu)|$ be as defined above. Consider the following conditions on representation maps ρ for D.

Condition (1) $\forall\, d \in D, \quad |\rho(d)| = 1$

Condition (2) $\forall\, m \in \rho(D), \quad \text{Domain}(m) = N_L = \{1, 2, \ldots, L\}.$

(i) If $|B|$-ary system $(\rho, \mathscr{a}_\Lambda, \mathscr{U}_D)$ solves (D, Λ, U_D) subject to conditions 1 and 2, then

$$\forall\, \mathscr{U}_d \in \mathscr{U}_D \qquad \sum_{\mu \in \overline{\rho_*(D)}} |B|^{-|\ell_d(\mu)|} \leq 1.$$

(ii) If $|B|$-ary system $(\rho, \mathscr{a}_\Lambda, \mathscr{U}_D)$ solves (D, Λ, U_D) subject to conditions (1) and (2), then

$$\forall\, \mathscr{U}_d \in \mathscr{U}_D \qquad \max_{\mu \in \overline{\rho_*(D)}} |\ell_d(\mu)| \geq \lceil \log_{|B|} |D| \rceil.$$

(iii) There is a $|B|$-ary system $(\rho, \mathscr{a}_\Lambda, \mathscr{U}_D)$ satisfying conditions (1) and (2) and solving (D, Λ, U_D) such that

$$\forall\, \mathscr{U}_d \in \mathscr{U}_D, \qquad \max_{\mu \in \overline{\rho_*(D)}} |\ell_d(\mu)| = \lceil \log_{|B|} |D| \rceil.$$

Part (i) is an inequality on a summation of terms composed of the memory alphabet size raised to a negative exponent. Corresponding to each memory state $\mu \in \overline{\rho_*(D)}$ there is one term in the summation with negative exponent $|\ell_d(\mu)|$.

The representation map ρ of example 4 satisfies the fixed-length single representation conditions. For algorithm $\mathscr{U}_{d_3}$ the summation becomes

$$\sum_{\mu \in \overline{\rho_3(D)}} 2^{-|\ell_d(\mu)|} = 2^{-|\ell_d(010)|} + 2^{-|\ell_d(001)|} + 2^{-|\ell_d(110)|}$$

$$= 2^{-2} + 2^{-4} + 2^{-2} = \frac{9}{16}$$

and part (i) is obeyed with strict inequality.

Part (i) is a statement about distributions on the performance measure $|\ell_d(\mu)|$ for any system solving (D, Λ, U_D) and obeying conditions (1) and (2). Not all initial memory states in $\overline{\rho_*(D)}$ can have short update times, and if some memory states have short update times then others must have relatively long update times. The term in the summation corresponding to memory state μ with a short update time $|\ell_d(\mu)|$ will have a

small negative exponent and hence a relatively large value. For the inequality to be satisfied, the other terms in the summation corresponding to other memory states μ' must have relatively small values and hence relatively large negative exponents and long update times $|\ell_d(\mu)|$.

Part (ii) states that for any $|B|$-ary update algorithm $\mathcal{U}_d$, some initial memory state will require at least $\lceil \log_{|B|} |D| \rceil$ memory accesses. The lower bound for example problem 4 using binary-valued memory is $\lceil \log_2 |3| \rceil = 2$. For the algorithm $\mathcal{U}_{d_3}$ initial memory state 001 requires 4 memory accesses.

Part (iii) states that for any dynamic retrieval problem there is a $|B|$-ary system solving the problem and obeying conditions (1) and (2) such that no initial memory state requires more than $\lceil \log_{|B|} |D| \rceil$ memory accesses by any complete update algorithm. For example problem 4 no initial memory state in the binary system partially described below requires more than 2 memory accesses by algorithm $\mathcal{U}'_{d_3}$.

$$\rho'(d_1) = \{01\}$$

$$\rho'(d_2) = \{00\}$$

$$\rho'(d_3) = \{11\}$$

$\mathcal{U}'_{d_3}$: write value 1 in cell 1
write value 1 in cell 2
halt

Proof of Theorem 10:

(i) Consider the tree associated with $\mathcal{U}_d$, where $L = L(\rho)$.

$$\sum_{\mu \in \overline{\rho_L(D)}} |B|^{-|\ell_d(\mu)|} = \sum_{\mu \in \overline{\rho_L(D)}} |B|^{-\ell_{q(\mu)}} = \sum_{q \in Q_t} |B|^{-\ell_q} \leqslant 1.$$

The first equality follows from $|\ell_d(\mu)| = \ell_{q(\mu)}$. The second equality follows from the facts that $\{M_q | q \in Q_t\}$ is a partition of $\overline{\rho_L(D)}$; for $q^* \in Q_t$ the term $|B|^{-\ell_{q^*}}$ appears in the summation over $\mu \in \overline{\rho_L(D)}$ once for each $\mu \in M_{q^*}$; and $|M_{q^*}| = 1$ because $|M_q| = |M'_q|$, $M'_q \subseteq \overline{\rho_L(d)}$, and $|\overline{\rho_L(d)}| = 1$ for the conditions on the representation map. The inequality is a well-known property of the path lengths of a finite tree (see Gallager[11]).

(ii) Let $\ell_{max} = \max_{\mu \in \overline{\rho_L(D)}} |\ell_d(\mu)|$.

$$|D|\,|B|^{-\ell_{max}} \leqslant \sum_{\mu \in \overline{\rho_L(D)}} |B|^{-|\ell_d(\mu)|} \leqslant 1.$$

The first inequality follows from the fact that each of the $|\overline{\rho_L(D)}|$ terms in the summation

is no less than $|B|^{-\ell_{max}}$ and $|\overline{\rho_L(D)}| = |D|$ for the conditions on the representation map. The second inequality is part (i) above. Taking logarithms to the base $|B|$ and using the fact that ℓ_{max} is integer-valued yields part (ii).

(iii) Let $L = \lceil \log_{|B|} |D| \rceil$. For an enumerative representation map $\rho : D \rightarrow B^L$, update algorithms $\mathscr{U}_d$ that write the representation for d into memory always make $\lceil \log_{|B|} |D| \rceil$ accesses before halting.

Theorem 10 covers a restricted class of systems, those systems in which the single representation (condition (1)) for each data base is a function on N_L (condition (2)). Proof of Theorem 10 requires that each data base be represented by a single memory state (i.e., $|\overline{\rho_*(d)}| = 1$). Representation maps that do not meet both conditions allow some data bases to be represented by several memory states.

Consider a representation map in which the single representation for a data base is a function on N_j for some j, but j is different for some pair of data bases (i.e., each representation specifies values for a contiguous set of cells at the beginning of memory, but different representations may be of different lengths). The binary representation map $\rho(d_1) = \{1\}$, $\rho(d_2) = \{01\}$, and $\rho(d_3) = \{001\}$ is an example. In any binary memory, including the span of the representation map, each data base with a short representation may be represented by several memory states, since cells with large addresses will have no value specified by the short representation. For the example above, in a binary memory of length 3, the data base d_1 with short representation 1 may be represented by memory states 100, 101, 110, and 111.

Similarly, where the representation map includes strict partial functions or several representations for the same data base, some data base may be represented by several memory states.

The conditions of Theorem 10 are not met by many practical systems (e.g., hashing schemes, linked-list schemes). Removal of the conditions on the representation map is necessary for consideration of such systems.

Theorem 11

Let (D, Λ, U_D) be a dynamic retrieval problem including a complete set of updates, and let the measure $|\ell_d(\mu)|$ be as defined above.

(i), (ii), and (iii) If $|B|$-ary system $(\rho, \mathscr{a}_\Lambda, \mathscr{U}_D)$ solves (D, Λ, U_D), then

$$\text{(i)} \quad \forall\, d \in D \quad \sum_{\mu \in \overline{\rho_*(D)}} |B|^{-|\ell_d(\mu)|} \leq |\overline{\rho_*(d)}|$$

$$\text{(ii)} \quad \forall\, d \in D \quad \max_{\mu \in \overline{\rho_*(D)}} |\ell_d(\mu)| \geq \left\lceil \log_{|B|} \frac{|\overline{\rho_*(D)}|}{|\overline{\rho_*(d)}|} \right\rceil$$

(iii) $\max_{d \in D} \max_{\mu \in \overline{\rho_*(D)}} |\ell_d(\mu)| \geq \lceil \log_{|B|} |D| \rceil$

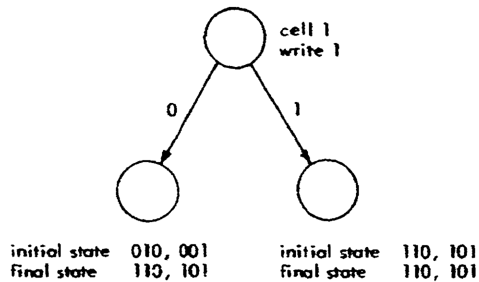
(iv) There is a $|B|$-ary system $(\rho, \mathcal{a}_\Lambda, \mathcal{U}_D)$ solving (D, Λ, U_D) such that

$$\max_{d \in D} \max_{\mu \in \overline{\rho_*(D)}} |\ell_d(\mu)| = \lceil \log_{|B|} |D| \rceil .$$

Part (i) is an inequality on a summation of terms, one term for each initial memory state $\mu \in \overline{\rho_*(D)}$. With the restrictions removed on ρ, however, the summation must be no greater than $|\overline{\rho_*(d)}|$ rather than 1 as in Theorem 10. When d is represented by many memory states of $\mathcal{M}_{L(\rho)}$, updates to d might require fewer accesses than when d is represented by a single memory state of $\mathcal{M}_{L(\rho)}$. Intuitively, fewer memory accesses might be required because any initial memory state should be relatively "close" to some memory state representing d. Mathematically, fewer memory accesses might be required, since the terms of the summation may have smaller negative exponents $|\ell_d(\mu)|$ and still satisfy the inequality ($\leq |\overline{\rho_*(d)}|$ rather than ≤ 1).

Example 4b. Multiple Representations for d_3

Consider a binary representation map which allows another representation 101 for d_3 of example 4, $\rho(d_1) = \{010\}$, $\rho(d_2) = \{001\}$, and $\rho(d_3) = \{110, 101\}$. An update algorithm $\mathcal{U}_{d_3}$ for this representation map is $\mathcal{U}_{d_3}$:write value 1 in cell 1 and halt. The tree



KEY:

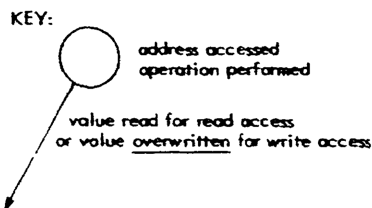


Fig. 4. Tree corresponding to update algorithm $\mathcal{U}_{d_3}$ of example 4b.

corresponding to this algorithm is shown in Fig. 4. For this algorithm $\ell_d(010) = \ell_d(001) = 0$ and $\ell_d(110) = \ell_d(101) = 1$. The inequality of part (i) becomes

$$\sum_{\mu \in \overline{\rho_*(D)}} 2^{-|\ell_d(\mu)|} = 4 \cdot 2^{-1} = 2 \leq |\overline{\rho_*(d)}| = 2.$$

Part (ii) states that for any system solving the problem, some initial memory state will require at least $\lceil \log_{|B|} (|\overline{\rho_*(D)}| / |\overline{\rho_*(d)}|) \rceil$ memory accesses by algorithm $\mathcal{U}_d$. For algorithm $\mathcal{U}_{d_3}$ of example 4b the bound becomes $\lceil \log_2 (|\overline{\rho_3(D)}| / \overline{\rho_3(d_3)}) \rceil = \lceil \log_2 (4/2) \rceil = 1$. Every initial memory state requires 1 access by $\mathcal{U}_{d_3}$, so the bound is satisfied with equality.

Part (iii) states that for any $|B|$-ary system solving the problem, some initial memory state will require at least $\lceil \log_{|B|} |D| \rceil$ accesses by some update algorithm $\mathcal{U}_d$. This is the same numerical lower bound as Theorem 10(ii), yet we concluded that allowing data base d to be represented by several memory states (i.e., $|\rho_*(d)| > 1$) can reduce the number of accesses needed for update U_d. This point merits further discussion.

Intuitively, allowing data base d to be represented by several memory states increases the number of possible initial memory states. For data bases d' other than d, update algorithms $\mathcal{U}_{d'}$ may require more accesses to deal with the larger initial memory state set. Mathematically, allowing data base d to be represented by several memory states increases the number of terms in the summation of part (i), since $|\overline{\rho_*(D)}|$ increases without increasing $|\overline{\rho_*(d')}|$, the upper bound on the sum for data bases d' other than d. Larger negative exponents $|\ell_{d'}(\mu)|$ might be necessary to satisfy the inequality.

For any binary system solving example problem 4, some initial memory state will require $\lceil \log_2 |D| \rceil = \lceil \log_2 |3| \rceil = 2$ memory accesses by some update algorithm. Algorithm $\mathcal{U}_{d_3}$ of example system 4b (multiple representations for d_3) requires only 1 memory access for any initial state. By part (iii), it is not possible to design update algorithms $\mathcal{U}_{d_1}$ and $\mathcal{U}_{d_2}$ which access memory only once for any initial memory state in $\overline{\rho_*(D)}$.

Part (iv) states that for any problem there is a $|B|$-ary system such that every update algorithm accesses memory $\lceil \log_{|B|} |D| \rceil$ or fewer times before halting. For example problem 4, a binary system with representation map $\rho(d_1) = 01$, $\rho(d_2) = 00$, $\rho(d_3) = 11$, and update algorithms $\mathcal{U}_d$ writing the representation for d into memory makes 2 ($= \lceil \log_2 3 \rceil$) memory accesses for any update on any initial memory state.

Proof of Theorem 11:

(i) Consider the tree associated with $\mathcal{U}_d$, where $L = L(\rho)$.

$$\sum_{\mu \in \overline{\rho_L(D)}} |B|^{-|\ell_d(\mu)|} = \sum_{\mu \in \overline{\rho_L(D)}} |B|^{-\ell_q(\mu)} = \sum_{q \in Q_t} |M_q| \, |B|^{-\ell_q}$$

$$\leq |\overline{\rho_L(d)}| \sum_{q \in Q_t} |B|^{-\ell_q} \leq |\overline{\rho_L(d)}|.$$

The first equality follows from $|\ell_d(\mu)| = \ell_{q(\mu)}$. The second equality follows from the facts that $\{M_q \mid q \in Q_t\}$ is a partition of $\overline{\rho_L(D)}$ and for $q^* \in Q_t$ the term $|B|^{-\ell_{q^*}}$ appears in the sum over $\mu \in \overline{\rho_L(D)}$ once for each $\mu \in M_{q^*}$. The first inequality follows from $|M_q| = |M'_q|$ and $M'_q \subseteq \overline{\rho_L(d)}$ for $q \in Q_t$. The final inequality follows from $\sum_{q \in Q_t} |B|^{-\ell_q} \leq 1$, a well-known property of the path lengths of a finite tree (see Gallager[11]).

(ii) Let $\ell_{max} = \max_{\mu \in \overline{\rho_*(d)}} |\ell_d(\mu)|$.

$$|\overline{\rho_L(D)}|\,|B|^{-\ell_{max}} \leq \sum_{\mu \in \overline{\rho_L(D)}} |B|^{-|\ell_d(\mu)|} \leq |\overline{\rho_L(d)}|.$$

The first equality follows from the fact that each of the $|\overline{\rho_L(D)}|$ terms in the sum is no less than $|B|^{-\ell_{max}}$ and the second inequality is part (i) above. Taking logarithms to the base $|B|$ and using the fact that ℓ_{max} is integer-valued yields part (ii).

(iii) For some $d \in D$ $|\overline{\rho_L(d)}| \leq \dfrac{|\overline{\rho_L(D)}|}{|D|}$, since $\overline{\rho_L(D)} = \bigcup_{d \in D} \overline{\rho_L(d)}$. (Some data base must be represented by fewer than the average number of memory states per data base.) This inequality along with (ii) yields the desired result.

(iv) The enumerative representation map ρ and updating algorithms $\mathscr{U}_D$ presented in the proof of Theorem 10(iii) satisfy the conditions of this theorem.

A user making frequent updates may be more concerned with average update performance than with worst-case performance. A few update algorithms making more than $\lceil \log_{|B|} |D| \rceil$ memory accesses may be acceptable (or preferable) if most update algorithms make fewer accesses for most initial memory states. Let a user of a system make a sequence $U_{d(1)} U_{d(2)} \cdots U_{d(N)}$ of N complete updates where successive updates are chosen independently from $U_D = \{U_d : D \to d \mid d \in D\}$ according to some probability distribution P on D. Let $avg(\ell_N(\mu))$ be the number of memory accesses per total update averaged over the $|D|^N$ possible sequences of N complete updates from initial memory state μ (i.e., $\mathcal{S}(\mathscr{M}) = \mu$ when algorithm $\mathscr{U}_{d(1)}$ is started). Define the measures $avg(\ell_N)$ and $avg(\ell'_N)$ to be the minimum and maximum, respectively, of $avg(\ell_N(\mu))$ over initial memory states $\mu \in \overline{\rho_*(D)}$. The quantity $avg(\ell_N)$ is the smallest average number of memory accesses per update that a user can expect for a sequence of N updates and $avg(\ell'_N)$ the largest.

Recall that $\xi_d(\mu)$ is defined to be the final (halting) memory state for update algorithm $\mathscr{U}_d$ operating on initial (starting) memory state μ. Let $\{\mu_i\}_{i=1}^N$ be the sequence of final (halting) memory states for the sequence of update algorithms $\{\mathscr{U}_{d(i)}\}_{i=1}^N$ invoked on initial memory state μ_0 for algorithm $\mathscr{U}_{d(1)}$ defined by $\mu_i = \xi_{d(i)}(\mu_{i-1})$ $i = 1, 2, \ldots, N$. Define $avg(\ell_N(\mu_0))$ as follows:

$$\mathrm{avg}(\ell_N(\mu_0)) = \sum_{d(1)\in D} \sum_{d(2)\in D} \cdots \sum_{d(N)\in D} \left[\prod_{i=1}^{N} P(d(i))\right]\left[\frac{1}{N}\sum_{i=1}^{N} |\ell_{d(i)}(\mu_{i-1})|\right].$$

Define $\mathrm{avg}(\ell_N)$ and $\mathrm{avg}(\ell'_N)$ to be

$$\mathrm{avg}(\ell_N) = \underline{\min}_{\mu\in\rho_*(D)} \mathrm{avg}(\ell_N(\mu))$$

$$\mathrm{avg}(\ell'_N) = \underline{\max}_{\mu\in\rho_*(D)} \mathrm{avg}(\ell_N(\mu)).$$

Let $\mathrm{avg}(\ell) = \lim_{j\to\infty} \inf \left\{\mathrm{avg}(\ell_N)\right\}_{N=j}^{\infty}$ and let

$$\mathrm{avg}(\ell') = \lim_{j\to\infty} \sup \left\{\mathrm{avg}(\ell'_N)\right\}_{N=j}^{\infty}.$$

Theorem 12

Let (D, Λ, U_D) be a dynamic retrieval problem including a complete set of total updates, let P be a probability distribution on D, and let the average performance measure $\mathrm{avg}(\ell)$ be as defined above.

(i) If $|B|$-ary system $(\rho, \mathcal{a}_\Lambda, \mathcal{U}_D)$ solves (D, Λ, U_D), then

$$\mathrm{avg}(\ell) \geq H(D)$$

where $H(D) = -\sum_{d\in D} P(d) \log_{|B|} P(d)$.

(ii) There exists a $|B|$-ary system $(\rho, \mathcal{a}_\Lambda, \mathcal{U}_D)$ solving (D, Λ, U_D) such that

$$\mathrm{avg}(\ell') < H(D) + 1.$$

Part (i) states that for any system solving the problem, the expected number of memory accesses per update is no less than H(D). For example problem 4 solved on a binary memory and the probability distribution $P(d_1) = \frac{1}{4}$, $P(d_2) = \frac{1}{4}$, $P(d_3) = \frac{1}{2}$, the lower bound of part (i) is $H(D) = -\frac{1}{2}\log_2 \frac{1}{2} - \frac{1}{4}\log_2 \frac{1}{4} - \frac{1}{4}\log_2 \frac{1}{4} = \frac{3}{2}$. Consider example system 4a including the update algorithms

$\mathcal{U}_{d_1}$: write value 0 in cell 1
write value 1 in cell 2
write value 0 in cell 3
halt

$\mathcal{U}_{d_2}$: write value 0 in cell 1
write value 0 in cell 2
write value 1 in cell 3
halt

$\mathcal{U}_{d_3}$ as previously defined.

With the system connected to binary memory $\mathscr{M}_3$ in initial state 010, the expected number of memory accesses for one update is

$$\text{avg}(\ell_1(010)) = P(d_1)|\ell_{d_1}(010)| + P(d_2)|\ell_{d_2}(010)| + P(d_3)|\ell_{d_3}(010)|$$

$$= \frac{1}{4}\cdot 3 + \frac{1}{4}\cdot 3 + \frac{1}{2}\cdot 2 = 2\frac{1}{2}.$$

Similarly, $\text{avg}(\ell_1(001)) = 3\frac{1}{2}$ and $\text{avg}(\ell_1(110)) = 2\frac{1}{2}$. The quantity $\text{avg}(\ell_1)$ is the minimum over the initial memory states of $\text{avg}(\ell_1(\mu))$, so that $\text{avg}(\ell_1) = 2\frac{1}{2}$. Similarly, $\text{avg}(\ell_1') = 3\frac{1}{2}$. For this system $\text{avg}(\ell) = \lim_{N\to\infty} \text{avg}(\ell_N) = 2\frac{3}{4}$ and part (i) is satisfied with a strict inequality.

Part (ii) states that for any problem, there is a $|B|$-ary system solving the problem such that the average number of accesses for an update is within 1 of the lower bound. For the partial example system 4b supplemented with the algorithms $\mathscr{U}_{d_1}$ and $\mathscr{U}_{d_2}$ (which perform correctly because $\rho(d_1)$ and $\rho(d_2)$ remain unchanged), the average number of accesses is easily calculated to be $\text{avg}(\ell) = P(d_1)\cdot 3 + P(d_2)\cdot 3 + P(d_1)\cdot 1 = 2\frac{1}{2}$, since each update algorithm makes the same number of accesses for any initial memory state. Since $\text{avg}(\ell) = 2\frac{1}{2}$ is still not less than $H(D)+1 = 2\frac{1}{2}$, some other binary system solving problem 4 requires still fewer accesses per update on the average.

Proof of Theorem 12:

(i) Let the sequence $\{d(i)\}_{i=1}^N$ be a message to be sent from a user to a receiver as follows (see Fig. 5). When the user makes update $U_{d(1)}$ from initial memory state μ_0, the encoder monitors the memory cells accessed by update algorithm $\mathscr{U}_{d(1)}$, remembering $\ell_{d(1)}(\mu_0)$, the chronological sequence of cell values read or overwritten. For

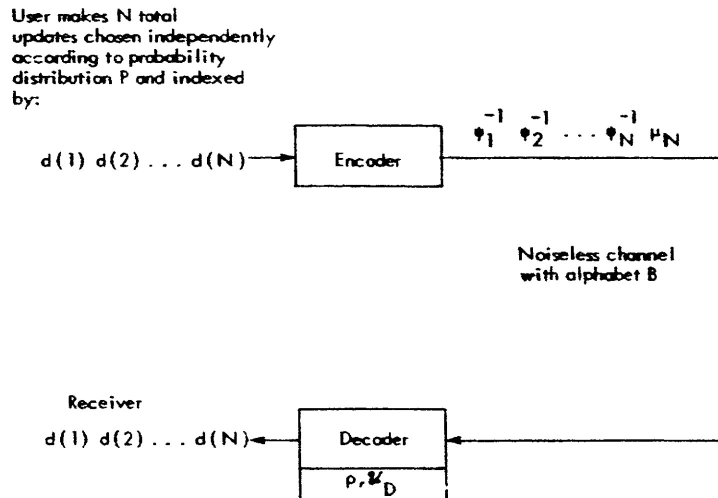


Fig. 5. Sending a message from ensemble D^N (for fixed N and μ_0) with entropy NH(D).

notational convenience define Ψ_i to be $\ell_{d(i)}(\mu_{i-1})$ and define Ψ_i^{-1} to be the reverse of sequence Ψ_i. When $\mathscr{U}_{d(1)}$ halts, the encoder sends the sequence Ψ_1^{-1} over the noiseless channel. After repeating this process for i = 2, 3, ..., N, the encoder sends the sequence of cell values $\mathscr{M}(1)\ \mathscr{M}(2)\ \dots\ \mathscr{M}(L(\rho))$ of the final memory state μ_N.

After storing the channel values on a stack (which may need as many as $N \cdot \max_{d\in D} \max_{\mu\in\overline{\rho_*(D)}} |\ell_d(\mu)| + L(\rho)$ units of stack storage) the receiver decodes the message with the aid of the representation map ρ, the memory size $L(\rho)$, and the set of trees associated with the set of algorithms $\{\mathscr{U}_d \mid d \in D\}$ as follows. From the top $L(\rho)$ stack values (popped and discarded) forming μ_N and the representation map ρ, d(N) is determined. A path through the tree associated with $\mathscr{U}_{d(N)}$ is followed by popping and discarding stack values until reaching a terminal node q. The memory state μ_{N-1}, determined from the formula $\mu_{N-1} = \xi^{-1}_{d(N)}(\mu_N) = \mu_N - \beta'_q \cup \beta_q$, with the representation map ρ yields d(N-1), and iterating this procedure yields d(N-2), d(N-3), ..., d(1).

Since the encoding is a uniquely decodable code, the average encoding length $N\ \text{avg}(\ell_N(\mu_0)) + L(\rho)$ must be no less than the entropy NH(D) of the message ensemble D^N (see Gallager[11]):

$$N\ \text{avg}(\ell_N(\mu_0)) + L(\rho) \geq NH(D).$$

Dividing by N and taking the lim inf of the sequence

$$\{ \min_{\mu_0\in\overline{\rho_*(D)}} \text{avg}(\ell_N(\mu_0))\}_{N=1}^{\infty}$$

yields the desired result.

(ii) Given P, a Huffman representation map $\rho: D \to B^*$ and update algorithms $\mathscr{U}_d$ that write the representation for d into memory satisfy

$$\forall N,\quad \forall \mu_0\ \text{avg}(\ell_N(\mu_0)) < H(D) + 1.$$

Where P is unknown, a preconstructed universal representation map (Elias[13]) and similar update algorithms do nearly as well, provided the more probable data bases are assigned the shorter representations.

Consider an update algorithm $\underset{\sim}{\mathscr{U}}$ that can read and then write at one address in a <u>single</u> memory access. An algorithm $\underset{\sim}{\mathscr{U}}$ is a 5-tuple $\{s_0, S, \nu, \omega, \sigma\}$ where s_0 is the starting state, S is the state set including a halting state s_h and a continuing state set $S_c = S - s_h$, $\nu: S \to N^+$ is an address function, $\omega: S \times B \to B$ is a write function, and $\sigma: S \times B \to S$ is a state transition function. The action of the algorithm $\underset{\sim}{\mathscr{U}}$ in state s at time t is to access the address $\nu(s)$, read the value $b = \mathscr{M}(\nu(s), t)$, write the value $\omega(s,b)$ so that $\mathscr{M}(\nu(s), t+1) = \omega(s, b)$, and enter state $\sigma(s, b)$. By combining reads and writes, such an update algorithm might make only half the memory accesses required of more primitive algorithms. The following theorem proves that the lower bounds of

Theorems 10, 11, and 12 hold also for algorithms $\underset{\sim}{\mathcal{U}}$.

Theorem 13

Let (D, Λ, U_D) be a dynamic retrieval problem solved by the $|B|$-ary system $(\rho, \mathcal{A}_\Lambda, \underset{\sim}{\mathcal{U}}_D)$, where $\underset{\sim}{\mathcal{U}}_D = \{\underset{\sim}{\mathcal{U}}_d \mid d \in D\}$ is a complete set of total update algorithms $\underset{\sim}{\mathcal{U}}_d$ that can read and then write at one address in a single memory access. Let $\underset{\sim}{\ell}_d(\mu)$ be the number of memory accesses by algorithm $\underset{\sim}{\mathcal{U}}_d$ for initial memory state μ and let $\text{avg}(\underset{\sim}{\ell})$ be defined in terms of $\underset{\sim}{\ell}_d(\mu)$. Theorems 10, 11, and 12 remain true when $\mathcal{U}_D$, $\mathcal{U}_d$, $\ell_d(\mu)$, and $\text{avg}(\ell)$ are replaced by $\underset{\sim}{\mathcal{U}}_D$, $\underset{\sim}{\mathcal{U}}_d$, $\underset{\sim}{\ell}_d(\mu)$, and $\text{avg}(\underset{\sim}{\ell})$, respectively.

Proof of Theorem 13: Associated with an update algorithm $\underset{\sim}{\mathcal{U}}_d$ is a tree including a set of nodes Q and a set of branches. Among the nodes Q is a set of terminal nodes Q_t and a set of interior nodes $Q_i = Q - Q_t$ including a distinguished root node q_0. Associated with each interior node q is an address label $\nu(q_0)$. Associated with each branch h of the tree is a read label $\pi(h) \in B$ and a write label $\omega(h) \in B$.

The action of algorithm $\underset{\sim}{\mathcal{U}}_d$ at an interior node q at time t is to read cell $\nu(q)$ and leave by the output branch h with read label $\pi(h) = \mathcal{M}(\nu(q), t)$ after writing value $\omega(h)$ in cell $\nu(q)$ so that $\mathcal{M}(\nu(q), t+1) = \omega(h)$.

Associated with each node $q \in Q$ are two partial functions β_q and β'_q constructed as follows. For the root node q_0, $\beta_{q_0} = \beta'_{q_0} = \emptyset$. If node q is connected to an output branch h of node ρ, then $\beta_q = \beta_\rho \cup \{\nu(\rho), \pi(h)\}$ and $\beta'_q = \beta'_\rho \cup \{\nu(\rho), \omega(h)\}$.

Let algorithm $\underset{\sim}{\mathcal{U}}_d$ be connected to a $|B|$-ary memory $\mathcal{M}_{L(\rho)}$. Quotient sets M_q and M'_q are defined as previously. Properties a, b, c, and d hold with $\underset{\sim}{\ell}_d(\mu)$ substituted for $\ell_d(\mu)$, and hence the proofs of Theorems 10, 11, and 12 hold with the appropriate substitutions.

3.4 PARTIAL UPDATES

Two points should be kept in mind when reading this section. First, several results for partial updates that reflect practical update problems can be derived from the results for total updates. Second, the theoretical model of partial updates is probably not the most satisfactory one.

This section closely parallels sections 3.2 and 3.3 in considering whether or not read accesses, write accesses, and total accesses are appropriate measures for the performance of a partial update algorithm. It will be proved that write accesses alone is not an appropriate performance measure. It will be shown, however, that read accesses is one measure of the complexity of a partial update. Then several results for total accesses will be derived.

First, consider read accesses. For a dynamic retrieval system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ let $r_k(\mu)$ be the sequence of cell values read by update algorithm $\mathcal{U}_k$ before halting when started with initial memory state $\mu \in \overline{\rho_*(D)}$. The following theorem proves that some partial

updates require read accesses to memory.

Theorem 14

(i) Let (D, Λ, U_K) be a dynamic retrieval problem including an update U_k such that $\forall\, d \in U_k(D),\ U_k(d) = d$. There is a system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ solving (D, Λ, U_K) such that

$$\forall\, \mu \in \overline{\rho_*(D)},\quad |r_k(\mu)| = 0.$$

(ii) There exists a dynamic retrieval problem (D, Λ, U) such that for any $(\rho, \mathcal{A}_\Lambda, \mathcal{U})$ solving the problem the following holds:

$$\forall\, \mu \in \overline{\rho_*(D)},\quad |r_k(\mu)| > 0.$$

Part (i) states that for any problem including a partial update U_k which leaves all data bases in its range unchanged, there is a system solving the problem which requires no read accesses for update U_k. Part (ii) states that for some problems, every system solving the problem requires read accesses. Note that some partial updates require read accesses, while any total update can be made without read accesses (Theorem 8). Thus read accesses is one indication of the complexity of a partial update.

Proof of Theorem 14:

(i) A partial update U_k satisfying $\forall\, d \in U_k(D),\ U_k(d) = d$ induces a partition on D with equivalence classes $[d]$ defined by $[d] = \{d' \in D \mid U_k(d') = d\}$. Let $L = \lceil \log_{|B|} (\max_{d \in D} |[d]|) \rceil$. Consider a representation map ρ in which the representation $\rho(d)$ for d is the concatenation of an enumerative representation map $\rho_{[d]}$ of the members of the class $[d]$ on cell addresses $[1, L]$ and an enumerative representation map of the partition of D. Furthermore, for the unique $d' \in [d]$ such that $U_k(d') = d'$ let $\rho_{[d']}$ specify value 0 for addresses in $[1, L]$. An algorithm $\mathcal{U}_k$ that writes value 0 at addresses in $[1, L]$ makes update U_k on D using ρ.

(ii) Consider the dynamic retrieval problem (D, Λ, U), where $D = \{d_1, d_2\}$ and U is defined by $U(d_1) = d_2$, $U(d_2) = d_1$. Suppose the problem is solved by a system $(\rho, \mathcal{A}_\Lambda, \mathcal{U})$ in which update algorithm $\mathcal{U}$ never makes read accesses, and suppose that when started with initial memory state $\mu \in \overline{\rho_*(d_1)}$, $\mathcal{U}$ halts with final memory state $\mu' \in \overline{\rho_*(d_2)}$. Then when started with initial memory state μ', algorithm $\mathcal{U}$ halts with final memory state μ', since no read accesses were made, and hence algorithm $\mathcal{U}$ does not make update U.

Now consider write accesses. For a dynamic retrieval system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ let $w_k(\mu)$ be the sequence of cell values overwritten by update algorithm $\mathcal{U}_k$ before halting when started with initial memory state $\mu \in \overline{\rho_*(D)}$. The following theorem shows that $|w_k(\mu)|$ is not an appropriate performance measure.

Corollary to Theorem 9: For any dynamic retrieval problem (D, Λ, U_K) there is a system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ solving the problem such that

$\forall\ k \in K, \quad \forall\ \mu \in \overline{\rho_*(D)} \quad |w_k(\mu)| \leq 2.$

Proof: One system solving the problem and never making more than 2 write accesses uses a positional representation map (described in the proof of Theorem 9) and update algorithms $\mathcal{U}_k$ which sequentially read memory until finding $\mathcal{M}(i) = 1$, set $\mathcal{M}(i)$ to 0, set $\mathcal{M}(U_k(d_i))$ to 1, and halt.

Now consider total accesses (reads and writes). Even though some updates may require read accesses, total accesses should be another appropriate measure of update performance. For a dynamic retrieval system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ let $\ell_k(\mu)$ be the sequence of cell values read or overwritten by algorithm $\mathcal{U}_k$ when started with initial memory state μ. The following theorem proves some results for the measure $|\ell_k(\mu)|$.

Corollary to Theorem 11: Let (D, Λ, U_K) be a dynamic retrieval problem and let the measure $|\ell_k(\mu)|$ be as defined above.

(i) and (ii) If $|B|$-ary system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ solves (D, Λ, U_K), then

$$\text{(i)} \quad \forall\ k \in K, \quad \sum_{\mu \in \overline{\rho_*(D)}} |B|^{-|\ell_k(\mu)|} \leq \Big| \bigcup_{d \in U_k(D)} \overline{\rho_*(d)} \Big|$$

$$\text{(ii)} \quad \forall\ k \in K, \quad \max_{\mu \in \overline{\rho_*(D)}} |\ell_k(\mu)| \geq \left\lceil \log_{|B|} \frac{|\overline{\rho_*(D)}|}{\Big| \bigcup_{d \in U_k(D)} \overline{\rho_*(d)} \Big|} \right\rceil .$$

(iii) There exists a dynamic retrieval problem (D, Λ, U) such that no system $(\rho, \mathcal{A}_\Lambda, \mathcal{U})$ solving the problem attains the lower bound of part (ii).

(iv) There exists a dynamic retrieval problem (D, Λ, U_K) including partial updates U_K and a system $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ solving the problem such that every update algorithm $\mathcal{U}_k \in \mathcal{U}_K$ attains the lower bound of part (ii).

Part (i) is a statement about distributions on the update measure for any algorithm making partial update U, and part (ii) is a statement about the maximum of the measure over $\mu \in \overline{\rho_*(D)}$. Not all problems are solvable, however, with systems attaining the lower bound to the maximum of the measure.

Proof:

(i) and (ii) For a partial update U in the problem (D, Λ, U_K) consider the data structure $\hat{D} = \{\hat{d}_1, \hat{d}_2\}$ defined by $\hat{d}_1 = D - U(D)$ and $\hat{d}_2 = U(D)$. Consider the total update $\hat{U}: \hat{D} \to \hat{d}_2$ on data structure $\hat{D}$. If an algorithm $\mathcal{U}$ makes update $\hat{U}$ on $\hat{D}$ using $\hat{\rho}$, then the results of Theorem 11 hold.

Let $(\rho, \mathcal{A}_\Lambda, \mathcal{U}_K)$ solve (D, Λ, U_K) and let the algorithm $\mathcal{U} \in \mathcal{U}_K$ make partial update U on D using ρ. Then algorithm $\mathcal{U}$ also makes total update $\hat{U}$ on $\hat{D}$ using $\hat{\rho}$ defined

by $\hat{\rho}(\hat{d}_1) = \bigcup_{d \in \hat{d}_1} \rho(d)$ and $\hat{\rho}(\hat{d}_2) = \bigcup_{d \in \hat{d}_2} \rho(d)$.

(iii) For the problem considered in the proof of Theorem 14, $U(D) = D$ and the bound of part (ii) becomes $\log_{|B|} 1 = 0$.

(iv) Example 5 in Section IV will satisfy all bounds simultaneously and one of the examples in the following discussion satisfies the bound with equality.

Two observations concerning partial updates are important. First, several results for partial updates may be derived from the results for a single total update algorithm. The results for partial updates are not as strong in the sense that the lower bounds are not always attainable. The model of partial updates is not entirely satisfactory. By allowing any map $D \to D$ to be a partial update, many partial updates are allowed that do not reflect reasonable modifications of data bases in practical situations. Further consideration of this problem is left for Section V.

Second, in many practical problems sets of partial updates are complete sets of total updates when considered from the appropriate viewpoint, and hence the stronger results of Theorem 11(iii) and Theorem 12 hold. These problems are generally characterized by data bases composed of several basic elements and updates that are changes of a single element value (see Knuth[4] for examples). Recall that a data structure is a collection of possible data bases, only a single one of which is observed.

An example problem includes a data structure of three entry tables with integer entry values in $[0,7]$ (see example 1) and partial updates $U_{nj} \in U = \{U_{nj} \mid n \in [1,3], j \in [0,7]\}$ that change the nth entry of the observed table to value j. Consider the subset of partial updates $U_2 = \{U_{2j} \mid j \in [0,7]\}$ that change the second entry value and the partition of D into equivalence classes $[d]_j$ defined by $[d]_j = \{d \in D \mid d(2) = j\}$. The class $[d]_j$ contains all tables with second entry value j.

Consider the data structure $\hat{D} = \{\hat{d}_j \mid j \in [0,7]\}$, where the data base $\hat{d}_j = [d]_j$ is the class of tables with second entry value j. Also consider the complete set of total updates $\hat{U} = \{\hat{U}_j : \hat{D} \to \hat{d}_j \mid j \in [0,7]\}$. Any algorithms $\hat{\mathscr{U}}$ making the complete set of total updates $\hat{U}$ on $\hat{D}$ using binary representation map $\hat{\rho}$ must satisfy

(i) For some algorithm $\hat{\mathscr{U}}_j$ and some initial memory state $\mu \in \overline{\hat{\rho}_*(\hat{D})}$

$$|\ell_j(\mu)| \geq \lceil \log_{|B|} |\hat{D}| \rceil = \lceil \log_2 8 \rceil = 3 \text{ Theorem 11(iii).}$$

Any set of algorithms $\mathscr{U}_2 = \{\mathscr{U}_{2j} \mid j \in [0,7]\}$ making partial updates U_2 on D using ρ also makes the complete set of total updates $\hat{U}$ on $\hat{D}$ using $\hat{\rho}$, where $\hat{\rho}$ is defined by $\hat{\rho}(\hat{d}_j) = \bigcup_{d \in [d]_j} \rho(d)$. Thus for some partial update algorithm $\mathscr{U}_{2j}$ and some initial memory state $\mu \in \overline{\rho_*(D)}$, $|\ell_{2j}(\mu)| \geq 3$.

In example static retrieval system 1b, partial update algorithms $\mathscr{U}_{2j}$ that sequentially read the second field until finding the cell with value 1, set this cell to value 0, set the $j+1^{th}$ cell of the second field to value 1 and halt may make as many as 8 memory

accesses. In example static retrieval system 1a, partial update algorithms that write the 3-bit binary representation for value j into the second field always make exactly 3 memory accesses.

IV. TRADING RELATIONS

For any problem the measures of memory, retrieval, and update performance for any system solving the problem satisfy a Kraft-like inequality, thereby implying certain distributional properties for the measures. In particular, (nearly) attainable lower bounds for worst-case and average measures can be derived from the Kraft-like inequalities. A user considering one performance measure to be of paramount importance can minimize performance in the measure with the appropriate system. A system minimizing performance by one measure, however, may perform poorly by other measures. For tables, example system 1a using an enumerative representation map minimized worst-case memory and retrieval, while system 1b using a concatenated positional field representation map did not. On the other hand, system 1b performed well in average retrieval, while system 1a did not.

In practice, the user decides the personal relative importance of the performance measures, and then chooses a system giving good "overall" performance. A system is <u>optimal</u> with respect to a set of performance measures if any other system improving performance in one measure degrades performance in another. In making his choice, a user would find a set of trading relations between performance measures of optimal systems expressed in terms of problem parameters (e.g., $|D|$, $|\lambda D|$) most helpful. No trading relation holds for all problems (D, Λ, U_D) as the following example shows.

<u>Example 5</u>

Let $D = \{0, 1\}^N$, where each data base $d = d(1)\ d(2) \ldots d(N)$ is a table of N binary-valued entries, $\Lambda = \{\lambda_i$: "What is the value of the ith entry?" $|\ i \in I = [1, N]\}$, and $U = U_D \cup \{U_{ij} \mid i \in [1, N]\ j \in \{0, 1\}\}$, where U_D is a complete set of total updates and U_{ij} is a partial update changing the ith entry of the observed data base to value j. Let P be a probability distribution on D, where $P(d) = \frac{1}{|D|} = 2^{-N}$.

Lower bounds on the performance measures of any binary dynamic retrieval system solving (D, Λ, U) are the following.

$$|\text{span}\ (\rho)| \geq \max_{d \in D} |m(d)| \geq \lceil \log_2 |D| \rceil = N \qquad \text{Theorem 3(i)}$$

$$\text{avg}(m) \geq H(D) = -\sum_{d \in D} 2^{-N} \log_2 2^{-N} = N \qquad \text{Theorem 4(i)}$$

$$\forall\, \lambda \in \Lambda \quad \max_{r \in \lambda D} |\alpha_\lambda(r)| \geq \lceil \log_2 |\lambda D| \rceil = \log_2 |\{0, 1\}| = 1 \qquad \text{Theorem 6(i)}$$

$$\forall\, \lambda \in \Lambda \quad \text{avg}(\alpha_\lambda) \geq -\sum_{r \in \lambda D} P_\lambda(r) \log_2 P_\lambda(r) = -\sum_{r \in \{0, 1\}} \frac{1}{2} \log_2 \frac{1}{2} = 1 \qquad \text{Theorem 7(i)}$$

$$\max_{d \in D} \max_{\mu \in \overline{\rho_*(D)}} |\ell_d(\mu)| \geq \lceil \log_2 |D| \rceil = N \qquad \text{Theorem 11(iii)}$$

$avg(\ell) \geq H(D) = N$ Theorem 12(i)

$$\forall\ i \in \{1, 2, \ldots, N\} \quad \max_{j \in \{0,1\}} \ \max_{\mu \in \overline{\rho_*(D)}} |\ell_{ij}(\mu)| \geq \log_2 2 = 1$$

$$\forall\ i \in \{1, 2, \ldots, N\} \quad avg(\ell_i) \geq 1.$$

The last two bounds can be derived by noticing that U_{i0} and U_{i1} are a complete set of total updates on $\hat{D} = \{\hat{d}_0, \hat{d}_1\}$, where $\hat{d}_0 = \{d \in D \mid d(i) = 0\}$ and $\hat{d}_1 = \{d \in D \mid d(i) = 1\}$, but they can be derived more easily by noticing that any nontrivial update must access at least one memory cell.

Consider the binary dynamic retrieval system $(\rho_N, \mathcal{A}_\Lambda, \mathcal{U})$ solving (D, Λ, U) using the identity representation map ρ_N. The representation $\rho_N(d)$ for d specifies value d(i) for address $i \in [1, N]$, and hence the memory measures $|span(\rho)|$, $\max_{d \in D} |m(d)|$, and avg(m) are minimal. The retrieval algorithm $\mathcal{A}_i$ reads and prints the value at address i in 1 access. The total update algorithm $\mathcal{U}_d$ writes d into memory with N accesses, and the partial update algorithm $\mathcal{U}_{ij}$ writes value j at address i in one access. A system is defined to be <u>minimal</u> with respect to a set of performance measures if no measure can be improved by any system. The system $(\rho_N, \mathcal{A}_\Lambda, \mathcal{U})$ is minimal with respect to the measures discussed, since each measure equals the lower bound taken over all systems.

It is informative to investigate the characteristics of a system minimizing all performance measures. The system minimizing worst-case memory and total update measures in the proofs of Theorem 3(ii) and Theorem 11(iv) uses an enumerative representation map, while the system minimizing worst-case retrieval measures in the proof of Theorem 6(ii) uses a concatenated field representation map in which the answer to a question is enumeratively represented in an associated field. For problems (D, Λ, U_D), where the questions are independent in the sense that for any $\vec{r} = (r_1, r_2, \ldots, r_n) \in (\lambda_1 D, \lambda_2 D, \ldots, \lambda_n D)$ there exists a $d \in D$ such that $\Lambda d = \vec{r}$, and when the size of the answer space of each question is an integer power of $|B|$, a $|B|$-ary concatenated field representation map minimizing the worst-case retrieval measure is also a $|B|$-ary enumerative representation map minimizing worst-case memory and total update measures. For these problems and the probability distribution defined by $P(d) = \frac{1}{|D|}$, the lower bounds for average performance are equal to the lower bounds for worst-case performance, so that average performance measures are also minimized. Also, worst-case and average lower bounds are derivable for the set of partial updates each of which changes the answer to a single question and are attainable by algorithms overwriting the associated field.

That no trading relation between measures holds for all problems has still not been shown, since any problem might have some solution minimizing all measures. Unfortunately this is not the case either, as the following examples investigated by Elias[14] show.

Example 6. Exact Match Question Set

Let $D = \{0,1\}^N$ and let P be a probability distribution on D defined by $P = 1/|D| = 2^{-N}$. To questions λ_e: "Is data base e the observed data base?" in the exact match question set $\Lambda_E = \{\lambda_e \mid e \in \{0,1\}^N = D\}$ observed data base d gives answer '1' if d = e and '0' otherwise. Performance measures considered for binary systems solving the dynamic retrieval problem (D, Λ_E, U_D) are $|\text{span}(\rho)|$ for memory, $\text{avg}(\underset{\sim}{\ell}_d)$ for total update, and $\text{avg}(A_\lambda)$ for retrieval. Recall that $\text{avg}(A_\lambda)$ is $\sum_{d \in D} P(d)|A_\lambda(d)|$, the minimum number of accesses by $\mathcal{a}_\lambda$ averaged over the collection of data bases. The measure $|\mathcal{a}_\lambda(r)|$ satisfies $|\mathcal{a}_\lambda(r)| = \min_{d \in D | \lambda d = r} |A_\lambda(d)|$ and thus $\text{avg}(A_\lambda) \geq \text{avg}(\mathcal{a}_\lambda)$. The lower bound on $|\text{span}(\rho)|$ and $\text{avg}(\underset{\sim}{\ell}_d)$ is $\lceil \log_2 |D| \rceil = N$, while a lower bound to $\text{avg}(A_\lambda)$ is 1, since any retrieval algorithm must access at least one memory cell before printing an answer from the binary answer space $\{0,1\}$.

The system $R_N = (\rho_N, \mathcal{a}_E, \underset{\sim}{\mathcal{U}}_D)$ uses the identity representation map ρ_N of example 5 and attains the bound on update with total update algorithms $\mathcal{U}_d$ writing d(i) at address i, $i \in [1,N]$. The retrieval algorithm $\mathcal{a}_e$ sequentially reads addresses $i \in [1,N]$ until either finding a mismatch between e(i) and $\mathcal{M}(i) = d(i)$ in which case it prints '0' and halts, or finding no mismatch after reading address N in which case it prints '1' and halts. For half the data bases $d \in D$ retrieval algorithm $\mathcal{a}_e$ finds a mismatch in cell 1, and for half the data bases producing the match d(1) = e(1) retrieval algorithm $\mathcal{a}_e$ finds a mismatch in the second cell. The number of retrieval accesses averaged over all data bases is $\text{avg}(A_\lambda) = 1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{N-1}} = 2(1-2^{-N})$, which is approximately double the lower bound.

The lower bound on the retrieval measure is attained by the system $R_1 = (\rho_1, \mathcal{a}_E, \underset{\sim}{\mathcal{U}}_D)$ using a representation map $\rho_1 = \rho_0 \cup \rho'_N$ which concatenates a positional map ρ_0 with an enumerative map ρ'_N. The positional representation $\rho_0(d)$ for d specifies value 1 for the cell whose address -1 in binary form is d, and value 0 for the other cells with addresses in $[1, 2^N]$. The enumerative representation $\rho'_N(d)$ for d specifies value d(i) for cell 2^N+i, $i \in [1,N]$. The retrieval algorithm $\mathcal{a}_e$ reads and prints the value in the cell whose address -1 in binary form is e. The update algorithm $\underset{\sim}{\mathcal{U}}_d$ first reads and remembers $\rho'_N(d')$ for the existing data base d', while simultaneously writing $\rho'_N(d)$. Then if $d \neq d'$, it sets the cell whose address -1 in binary form is d' to value 0 and sets the cell whose address -1 in binary form is d to value 1. The memory, update, and retrieval measures are $2^N + N$, $N + 2(1 - \frac{1}{N})$, and 1.

For system R_N, memory and update measures are minimal, while the retrieval measure is not. For system R_1 the retrieval measure is minimal, the update measure is nearly minimal, but the memory measure is far from minimal. A family of systems $\{R(k) \mid 1 \leq k \leq N-1\}$ improves memory performance from 2^N to N while allowing average retrieval performance to slip from 1 to 2.

The representation map ρ_k in system $R(k) = (\rho_k, \mathcal{a}_E, \underset{\sim}{\mathcal{U}}_D)$ concatenates a positional

representation map for the first N-k bits of a data base with the identity representation map. The positional representation for d specifies value 1 for the cell whose address -1 in binary form is the first N-k bits of d and specifies value 0 for other cells with addresses in $[1, 2^{N-k}]$. The retrieval algorithm $\boldsymbol{a}_e$ reads the cell whose address -1 in binary form is the first N-k bits of e, then continues to check the last N-k bits of the identity representation map for a match only when one of the 2^k out of 2^N data bases matching e in the first N-k bits is observed. The update algorithm $\mathcal{U}_d$ first reads and remembers the identity representation for the observed data base d' while simultaneously writing d, then if $d \neq d'$ updates the positional representation map with 2 additional accesses. For system R(k) memory, update, and retrieval measures are $2^{N-k} + N$, $N + 2(1 - \frac{1}{N}) < N+2$, and $1 + 2^{-N+k+1}(1 - 2^{-k}) < 1 + 2^{-N+k+1}$. Increasing k trades improved memory performance for reduced retrieval performance. The performance measures of system $R(N - \frac{1}{2}\lceil \log N \rceil)$ are less than $N(1 + \frac{2}{\sqrt{N}})$ in memory, less than $N(1 + \frac{2}{N})$ in average update, and less than $1 + \frac{2}{\sqrt{N}}$ in average retrieval. For sufficiently large N, memory, update, and retrieval performance are less than a factor of ϵ above their bounds, and the system is nearly minimal. The family R(k) is believed to be, but has not been proved to be, optimal. In either case the trading relations for the problem (D, Λ_E, U_D) have not proved to be very strong, since $R(N - \frac{1}{2}\lceil \log N \rceil)$ nearly minimizes all measures simultaneously. The trading relations of the following problem are stronger.

Example 7. Total Question Set

Let $D = \{0,1\}^N$ and let the total question set $\Lambda_T = \{\lambda_t \mid t \in T\}$ include all binary-valued questions on D except strongly related or constant questions. Any binary-valued question λ on D partitions D into two classes, the class $[0]_\lambda = \{d \in D \mid \lambda d = 0\}$ of all data bases giving answer 0 and the class $[1]_\lambda = \{d \in D \mid \lambda d = 1\}$ of all data bases giving answer 1. There are $2^{|D|}$ different partitions of D, but some of them correspond to strongly related or constant questions. The two partitions $\{[0]_\lambda, [1]_\lambda\}$ and $\{[0]_{\lambda'}, [1]_{\lambda'}\}$, where $[0]_{\lambda'} = [1]_\lambda$, correspond to a pair of strongly related questions λ and λ', since the classes contain the same data bases and simply switch the answers given by the member of a class (i.e., $\forall\, d \in D\ \lambda d = 1 - \lambda' d$). Eliminating pairs of strongly related questions (arbitrarily retaining from each pair λ and λ' the question giving answer 0 for the distinguished data base d = 0) leaves $2^{|D|-1}$ questions, and eliminating the constant question λ, where $[0]_\lambda = D$, leaves $|\Lambda_T| = 2^{|D|-1} - 1 = 2^{2^N - 1} - 1$ questions in the reduced question set Λ_T. Each question λ_t is indexed by a sequence $t \in T$ of $2^N - 1$ binary values where the ith value in the sequence t is the answer given by the data base which is the N-bit binary form of the number i (i.e., t lists the values $\lambda_t(d = 00\ldots01)$, $\lambda_t(d = 0\ldots10), \ldots$). Lower bounds on the performance measures of binary systems solving (D, Λ_T, U_D) are N for memory and update and 1 for retrieval.

The system $R_1(\rho_1, a_T, \mathscr{U}_D)$ uses a representation map ρ_1 which stores the answer to λ_t about observed data base d in the cell whose address in binary form is t, and retrieval algorithms a_t which read and print the value of that cell. Update algorithms $\mathscr{U}_d$ which write the representation for d into memory access $2^{2^N-1} - 1$ cells, where each cell stores the answer to one question in Λ_T. While the algorithms $\mathscr{U}_D$ are not optimal, we next show that optimal algorithms $\mathscr{U}'_d$ can at best halve this very large number of accesses.

Consider two distinct data bases d and d', where $d' \neq 0$. For every question $\lambda \in \Lambda_T$ such that d and d' give the same answer to λ (i.e., $d' \in [d]_\lambda$, where $[d]_\lambda$ is defined to be $\{d^* \in D \mid \lambda d^* = \lambda d\}$), there is a unique corresponding question $\lambda' \in \Lambda_T$ for which d and d' give different answers. The question λ' is the question corresponding to the partition $\{[d]_\lambda - d', D-[d]_\lambda \cup \{d'\}\}$. Including the partition $\{D-d', \{d'\}\}$ (since the partition $\{D, \emptyset\}$ is associated with the excluded constant question) yields the fact that for $d \neq d'$, data bases d and d' will give different answers for 2^{2^N-2} of the questions in Λ_T. From these considerations any update algorithm (partial or total) which changes the existing data base and uses a representation map that associates one cell with each question must access at least 2^{2^N-2} memory cells. This yields a lower bound of $2^{2^N-2}(1-2^{-N})$ on avg $\underset{\sim}{\ell}$ for the represesentation map ρ_1 and probability distribution $P(d) = \frac{1}{|D|}$.

The system $R'_1 = (\rho'_1, a_T, \mathscr{U}_D)$ does nearly this well by using a representation map ρ'_1 which is the concatenation of ρ_1 with the identity representation map ρ_N, and using update algorithms $\mathscr{U}_d$ that read the existing data base d' from ρ_N while writing d into ρ_N and then for $d \neq d'$ make the 2^{2^N-2} necessary changes in ρ_1. For this system the memory, update, and retrieval measures are $2^{2^N-1} -1 + N$, $2^{2^N-2}(1 - 2^{-N}) + N$, and 1. Although retrieval performance is minimal, memory and update performance are far from minimal.

The lower bounds on memory and update are attained by the system $R_N = (\rho_N, a_\Lambda, \mathscr{U}_D)$ using the identity representation map ρ_N and update algorithms $\mathscr{U}_d$ that write d(i) at address i, $i \in [1, N]$. Retrieval algorithms a_t read addresses 1 through N to determine d before printing the answer $\lambda_t d$. The system is not optimal, since some questions (such as "What is d(1)?") can be answered with fewer than N accesses. Elias has proved[14] that most of the retrieval algorithms must access most of the identity representation for most of the data bases.

For retrieval and memory performance, Elias has proved[14] that $\max_{\lambda \in \Lambda_T} \text{avg}(A_\lambda) \geq 0.226N$ for systems with nearly minimal memory performance, and $\max_{\lambda \in \Lambda_T} \text{avg}(A_\lambda)$ is near 1 only for systems with memory performance near 2^{2^N-1}. This trading relation between retrieval and memory performance is stronger than the relation of example 6, since no system can have nearly minimal retrieval and memory performance.

For retrieval and update performance, it has been proved that systems with minimal retrieval performance must have update performance greater than $2^{2^N - 2}(1 - \frac{1}{N})$ which is far from the minimal performance N. A strong trading relation between retrieval and update performance is conjectured.

Some problems (such as example 5) are solvable with minimal systems, some (such as example 6) are solvable with nearly minimal systems, and some (such as example 7) lead to strong trading relations between the performance measures of optimal solutions. Thus trading relations do not depend upon general problem parameters such as $|D|$, but do depend upon properties of the particular problem and the performance measures considered.

V. FURTHER PROBLEMS

Further work seems appropriate in the areas of partial updates, infinite data structures, and performance measures.

1. Partial Updates

In many practical data problems, partial updates are incremental additions or deletions of information in the observed data base. For example, partial updates of a library shelf list are generally additions or deletions of newly purchased or lost books. In hashing systems for tables partial updates are an addition or deletion of a table entry.

The mathematical model of a partial update is not entirely satisfactory. While a map $D \to D$ certainly includes any partial update in a practical problem with a finite collection of data bases, it also includes many formal partial updates which do not reflect sensible partial updates in practical problems. For a library shelf list a map $D \to D$ may reflect an update to an entirely new shelf list bearing no sensible relation to the previous shelf list. In a hashing system for tables, a map $D \to D$ may reflect an update to an entirely new table bearing no sensible relation to the previous table. A better model of partial updates is needed which accurately reflects sensible updates in practical problems, but excludes formal partial updates not reflecting reasonable updates in practical problems.

One possibility is to model a partial update of a data base by a change in the answer given to a particular question. In problems with independent questions, this change corresponds to a change in an entry value. In problems with some correlation between questions, changing the answer given to a particular question will also necessarily change the answers to other questions. For example, in library shelf lists, changing the answer to the question "Are there any books by author Smith ... ?" may also necessarily change the answer to the question "Are there any books on the subject physics ... ?" As another example (Elias[8]), consider a finite collection of tables with the question set "How many entries have value less than x?" (one question for each value in the fixed-entry value range). Increasing the answer to the question "How many entries have value less than x?" necessarily increases the answers to the questions "How many entries have values less than x+1?" "How many entries have value less than x+2?" etc.

2. Infinite Data Structures

Many partial updates in practical problems are incremental additions of information to the observed data base. Unless some finite limit is put on the total number of incremental additions allowed, the collection of possible data bases must be infinite. Standard hashing systems for tables are restricted to tables of N or fewer entries, and hence allow only N entry additions if no deletions are made. Practical library retrieval systems cannot be arbitrarily restricted in this manner.

This point merits further consideration, especially since library shelf lists were

given as an example of a collection of data bases. There is only a finite number of books in print today, and hence only a (very large but) finite number of possible shelf lists today. Thus it is possible to make statements about measures of static library retrieval systems (see Welch[12]) and measures for partial update algorithms adding a book in print to a shelf list. Furthermore, practical library retrieval systems must provide for additions of books not yet in print, the common provision being the addition of cards to a card catalogue retrieval system. Note that card catalogue retrieval systems are designed to answer an infinite set of retrieval questions, one author (title, and subject) question for each string of roman letters. Most of these questions are rarely, if ever, asked. Note also that for representation maps of infinite data structures, $L(\rho) = \infty$ and no finite memory is always sufficiently long. However, retrieval algorithms $\boldsymbol{a}_\Lambda$ can answer Λ about the observed data base d using ρ whenever connected to a memory sufficiently large to contain a representation $m \in \rho(d)$ of the observed data base.

The Kraft-like inequality and average results for system memory and retrieval performance are true for infinite collections of possible data bases (Elias[1]).

It is conjectured that the results for system update performance can also be extended to include infinite collections of possible data bases, and further work in this area is anticipated. Satisfactory progress in the library shelf list and similar practical retrieval problems depends not only on this extension but also on a more satisfactory model of partial updates.

3. Performance Measures

The number of accesses needed is one measure of the complexity of an update. The amount of computation performed between accesses is another. For example, the enumerative representation map that minimizes worst-case access for total updates may require difficult computations by the update algorithm between accesses (Cover[10]). Better measures are needed for the difficulty of the computations performed by the update algorithms.

Acknowledgment

I would like to thank Professor Peter Elias for his patience, encouragement, and insight throughout this work. I also wish to thank my readers, Professor Robert G. Gallager and Dr. Myer M. Kessler for their helpful comments in putting this thesis into final form.

The work reported here was supported in part by the National Science Foundation Graduate Fellowship Program.

References

1. P. Elias, Class Notes Course 6.891, Fall 1973, M.I.T., Cambridge, Massachusetts.
2. T. A. Welch, "Bounds on Information Retrieval Efficiency in Static File Structures," Ph.D. Thesis, Department of Electrical Engineering, M.I.T., 1971; also MAC TR-88, 1971.
3. M. Minsky and S. Papert, Perceptrons (The M.I.T. Press, Cambridge, Mass., 1969), pp. 215-225.
4. D. E. Knuth, The Art of Computer Programming, Vol. 3 (Addison-Wesley, Reading, Mass., 1969).
5. R. Morris, "Scatter Storage Techniques," Communs. ACM 11, 38-44 (1968).
6. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communs. ACM 13, 377-387 (1970).
7. J. Earley, "Toward an Understanding of Data Structures," Communs. ACM 14, 617-627 (1971).
8. P. Elias, Efficient Storage and Retrieval by Content and Address of Static Files," J. ACM 21, 246-260 (1974).
9. P. Elias, "Minimum Times and Memories Needed to Compute the Values of a Function," J. Comput. Syst. Sci., Vol. 9, No. 2, pp. 96-212, October 1974.
10. T. M. Cover, "Enumerative Source Encoding," IEEE Trans. on Information Theory, Vol. IT-19, No. 1, pp. 73-77, January 1973.
11. R. G. Gallager, Information Theory and Reliable Communication (John Wiley and Sons, New York, 1968), see Chap. 3.
12. D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE 40, 1098-1101 (1952).
13. P. Elias, "Universal Codeword Sets and Representations of the Integers," IEEE Trans. on Information Theory, Vol. IT-21, No. 2, pp. 194-203, March 1975.
14. P. Elias and R. A. Flower, "The Complexity of Some Simple Retrieval Problems," accepted for publication by J. ACM.