EVALUATING THE EFFICIENCY OF STORAGE STRUCTURES

by

Frank Wm. Tompa

Research Report CS-75-16

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

May 1975

EVALUATING THE EFFICIENCY OF STORAGE STRUCTURES

Frank Wm. Tompa
Department of Computer Science
University of Waterloo
Waterloo, Ontario

Keywords: Data structures, Efficiency of storage structures, Data structure design, Models of program execution, Storage mapping functions, Space/time considerations.

The design of data structures should be made systematically, rather than being haphazardly based on past experience. Data structures should be designed through step-wise refinement, using a framework which incorporates five distinct stages: data reality, data abstraction, information structure, storage structure, and machine encoding. The description of the proposal presented here includes its relation both to other research in data structures and to other approaches to data structures design. An application taken from the literature is then described in terms of the design framework.

One aspect of the design which can be approached algorithmically is the evaluation of storage structures (the fourth level of the design framework). If a library of potential implementations for each permissible data type is maintained, parametric formulas for expected run time and storage space can be derived for each member. The resulting formulas for individual data types are then used to design a storage structure by superimposing library members chosen to minimize the cost for a given information structure. The representation of the application described in the first section is thus redesigned using the algorithm described.

1. A scenario for data structure design

1.1 Levels of data refinement

Most data structures designers finally realize that data should be specified at two levels: the abstract, user-oriented information structure and the concrete, machine-oriented storage structure. Judging from the experience of algorithm designers, data structures should instead be viewed at many levels.

The design methodology suggested in this paper is based on five views of data:

- data reality: the data as it actually exists (including all aspects which are typically ignored except in philosophical treatises)
- data abstraction: a model of the real world which incorporates only those properties thought to be relevant to the application(s) at hand
- information structure: a refinement of the abstraction, often expressed in mathematical terms, in which only some data relationships are made explicit, the others derivable indirectly
- storage structure: a model of the information structure, often expressed in terms of diagrams which represent cells, linked and contiguous lists, and levels of storage media
- machine encoding: a final computer representation, including specifications for encoding primitive data objects (e.g., the representations for numbers and characters, the use of absolute as opposed to relative addresses, and the use of data packing).

The design of a data structure should proceed through successive levels, binding only those aspects which are necessary to specify each level. Within a level, the process of stepwise refinement should be adapted from algorithm design methodology to provide as smooth a transition as possible to the more detailed data level [Dijkstra 72, Wirth 73, Honig 74].

These five views of data correspond exactly to the levels of development in the corresponding program which describes the data operations to be performed. The first step defines the application's goal in terms of real objects. Next, the approach to a solution which achieves that goal is specified in terms of the data abstraction. An algorithm incorporating the approach is designed to operate on the information structure, after which a program implementing the algorithm is written to manipulate data in terms of its storage structure. Finally, the program is translated into object code which operates on data bound to a particular machine encoding.

The following paragraphs examine some aspects of research at each of the levels of data and at the interfaces between successive levels. This outline is not a complete review of current activities in data structures, but rather it is intended to provide additional insight into the interrelationships among the levels.

Top-level management must be able to consider all aspects of the data reality. For example, managers in a credit card company do care about the availability of plastic for the cards, the appeal of the design on the card's face, and other aspects which are probably "irrelevant" in the view of computer personnel dealing with the data. Therefore, such information must be included as part of the first view of data.

Translating an application from the reality into the abstraction is the interface at which data managers decide which aspects of data are relevant to the model (possibly by enumerating the types of queries and updates that may be applied to the data). This interface has also been studied by researchers in artificial intelligence who try to approach a problem by generating subgoals and assertions that lead to a solution.

Next, the data abstraction is the level at which management must communicate with computer personnel. In order to describe data formally, yet in terms understandable to non-mathematicians, many researchers have decided that the data abstraction is most conveniently expressed using a relational model of data [Codd 70]. Simultaneously, other research at this level has been conducted in terms of algebraic and graph-theoretic models [Furtado 75]. One important characteristic of any model at this level is that all relevant data relationships are equally accessible; thus, there are no preferred access paths to individual data elements.

Given an abstract specification of a data structure, many relationships can be expressed in terms of other, more basic ones. For example, finding a person's cousins can be accomplished by finding the children of that person's uncles and aunts, by finding the nieces and nephews of that person's parents, or by finding the children of that person's parents' siblings. Thus a data structures designer (typically the data base administrator) must determine which of the relationships are to be considered basic and therefore stored explicitly as part of the information structure. The rest will remain accessible only via one or more "access paths" through the stored relationships [Astrahan 74].

Recently, research at the information structures level has been concentrated on the study of data types [Hoare 72]. Each explicit relationship must be encoded by means of a data type such as a Cartesian product, sequence, set, or array, for which certain operations are defined. From these, complete information structures can be built and associated algorithms can be expressed in terms of the given primitive operations. Another area of study within information structures is the determination of optimal ordering within a particular data type (e.g., search trees) [Knuth 71].

For each data type there are many storage structures which can be used to model the data and operations. Thus, at the next interface, the choice of representation includes decisions between contiguity and links (i.e. pointers), between one-way and multiply-linked lists, and between direct addressing and scatter storage techniques. Some research has been conducted into choosing an optimal representation from only one class of storage structures [Randall 72], whereas section 2 describes a method for selecting the best representation available from a library of structures encompassing many classes.

Individual storage structures have been studied for many years. Before the mid-sixties numerous novel linking methods were developed in new programming languages [McCarthy 60, Weizenbaum 63, Roberts 65]. More recently, quantitative studies have been conducted to determine the performance characteristics for several storage methods [Lum 71, Nievergelt 72].

Binding a storage structure to a machine is the last interface in the design of data representation. Research at this interface includes the

choice of encoding for indivisible units of data such as integers (twos complement vs sign magnitude) and characters (EBCDIC vs ASCII). Research at this level includes the study of data compression techniques [Lesk 70] and the study of portable software [Waite 70].

Finally, the machine encoding is the level at which all implementations must eventually reside. Research at this level is typically hardware-oriented, concentrating on new memories and associated addressing techniques [Parhami 73].

It is important to realize that the design of a data structure can iterate through these levels. It may be obvious that the design of one level may give insight into deficiencies in the design of the previous level, thus requiring some backtracking to redesign. Less obvious is that "one man's indivisible unit of data is another's data reality". For example, a character string, which may be a primitive object to one user, may instead be interpreted to be a sequence of characters, thus requiring the design of a storage structure for the sequence and a data encoding for the individual characters.

1.2 Other descriptions of data levels

The design methodology proposed here is not new, but is instead a fresh look at methodologies suggested for several years. It is worthwhile to examine some previous proposals which are related to this framework.

The separation of the storage structure from the machine encoding was first incorporated into the COBOL language. By providing distinct data and program divisions, COBOL programmers were forced to separate the details of data encoding from the programs that manipulated the data,

thus achieving a degree of representation independence not previously available in other languages.†

In a paper which is now classic, Mealy described data at three levels [Mealy 67]. He reminded programmers that their view of data was only a theoretical model of real world objects, and therefore necessarily did not include some properties thought to be irrelevant. Furthermore, Mealy pointed out that the data as stored was, in turn, only a machine representation for the theoretical model. He claimed that programmers must be provided with facilities for accessing data from either of the latter viewpoints to allow representation independence while still providing control over implementation efficiency.

Following along these lines, d'Imperio explained the advantages of separating the design (and the description) of stored data into the two stages suggested by Mealy [d'Imperio 69]. She then reviewed several common structures in terms of those two levels which she called the data structure and the storage structure. This analysis aided her in comparing the facilities and efficiency of the data structures provided by a chosen set of programming languages.

A two-level approach to data has led several language designers to provide a syntax and semantics which will aid programmers in separating the design of the theoretical model from the design of the machine representation. Balzer developed a system which provided a uniform syntax for manipulating several data types [Balzer 67]. For example, the construct PERSON(AGE) could be used to access a component of a PL/I structure, access

---

† The format statement in FORTRAN does provide some independence, but at the input/output interface only.

an element in some array, or call a function which takes one parameter and returns some value. The particular machine representation to be used could be chosen after the program was completely written, and if inadequate it could be redesigned without any programming changes. Separating the two design decisions by this technique has since become known as uniform referents [Ross 70].

An extension and formalization of these ideas has recently received much attention under the name of abstract data types. One major approach has been conducted by Liskov and Zilles who have designed a language, CLU, which incorporates a construct named a cluster, through which the properties of a user-defined data type and valid operations on that type are defined separately from its representation [Liskov 74]. The translator for CLU enforces this separation by allowing access to a data type's representation inside the cluster which encodes it and disallowing such access elsewhere. The data encoding can thus be altered without affecting the rest of the program. In addition, the clusters permit proofs of correctness to be established in two stages: the uses of the abstract data type can first be shown to model reality, and the representation can then be shown to be a faithful implementation of that data type.

While some researchers explored the separation of the theoretical model from the machine representation, others realized that it is worthwhile to view data structures as consisting of more than two levels. McCuskey described a model containing four data views: the real world (problem level), the theoretical model (problem interpretation), the information structure (logical organization), and the machine representation

(physical organization) [McCuskey 70]. McCuskey thus separated Mealy's theoretical model level into two stages to distinguish between the specification of the relevant relations (to be made at the level of the problem interpretation by using a set-theoretic framework) and the specification of the explicit relations (to be made at the level of the logical organization). Earley furthered the ideas of McCuskey's model by developing unified language constructs to encompass the three levels dealing with stored data [Earley 73].

Another data model developed recently is the Data Independent Accessing Model (DIAM) [Senko 73]. DIAM has four levels of data specification: the entity set model for describing the data abstraction, the string model for describing the access structure, the encoding model for mapping access structures into an address space, and the physical device model for binding the address space to a machine. Thus, Mealy's machine representation has also been separated into two stages: the linking method (sometimes called the "storage mapping function") is to be specified at the former and the actual data layout at the latter.†

Figure 1 compares the several related proposals. From this summary, it should be apparent that the five views of data suggested here are not without a history, and that further motivation for each level can be found elsewhere .

1.3 An example of data description by levels

To understand the role of each view of data structures, it is worthwhile to examine an application taken from actual practice. This section therefore contains the description of a system for natural language

---

† McCuskey also recognized this division, which he named the relative vs absolute organization and incorporated into his model as sublevels of the physical organization.

Figure 1 Comparison of data level models

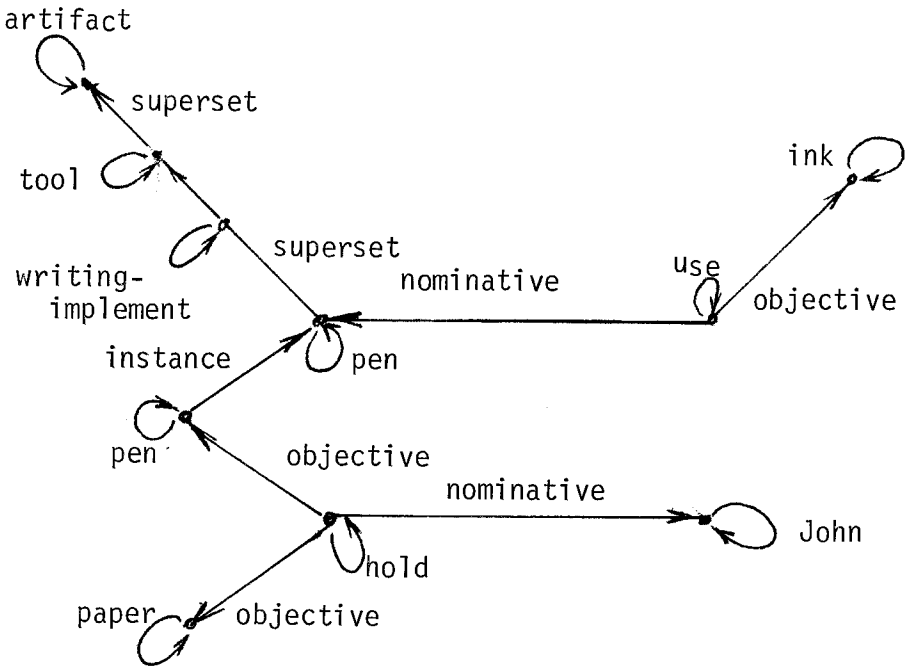
	data reality	data abstraction	information structure	storage structure	machine encoding
COBOL	-	-	-	program division	data division
Mealy	real world	theoretical model		machine representation	
d'Imperio	-	data structure		storage structure	
McCuskey	problem	problem interpretation	logical organization	physical organization	
Earley	-	relation	access path	machine	
Senko,et al	-	entity set	string	encoding	physical device

understanding which has recently been developed [Cohen 74b]. Naturally, it would be tedious, and likely confusing, to include all details of the design process or even of the final data design; many aspects are therefore omitted or simplified.

For this application, the data reality was natural language - in particular, written English. The data domain therefore included at least grammar (part of speech, tense, mood, voice), meaning (denotation, connotation), etymology (derivation, form), style, punctuation, and typesetting (type font, line width, spacing). Although it may not be apparent that all these could affect understanding, writers, publishers, and reading teachers might choose a different subset of these data properties as influencing a reader's comprehension.

Since the system was intended to be used to answer questions input from some computer terminal, the relevant relations were derived from some aspects of English syntax and semantics only. In particular, the syntax could be represented by a model in which a sentence consists of a verb together with a set of noun phrases assigned to roles determined by the verb's "case frame". The semantics were modeled by a "structural dictionary" which used a hierarchy of concepts (e.g., "pen - writing implement - tool - artifact"), some definitional information encoded using case grammar (e.g., "pens use ink"), and links to specific instances occurring in some text (e.g., "John holds the pen and the paper"). Cohen chose to represent the data abstraction by a labeled, directed graph in which the edges served either to identify a node or to link a verb to its noun phrases, a concept to its superset, or an instance to its dictionary entry (Figure 2).

Figure 2 Data abstraction

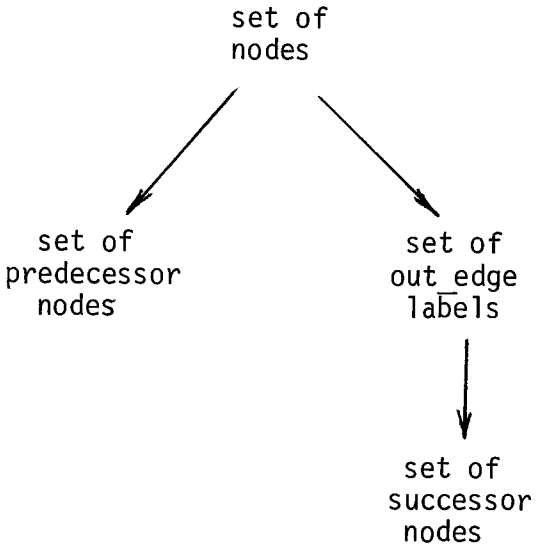


At the level of the information structure, only some relationships from the graph can be made explicit. It was decided that connectivitiy, reachability, and planarity, for example, were not used sufficiently often to justify their explicit inclusion. Furthermore, because graph traversal was expected to be performed in the direction of the edges more often than in reverse, edges originating at a node would be represented explicitly (those sharing a common label being grouped together), but those terminating at that node would only be accessible implicitly through a set of predecessor nodes (i.e., nodes from which there originates at least one edge terminating at the particular node in question). The following declarations could thus be used for the information structure (Figure 3):

```
type graph = set of node;
type node  = (set of node PREDECESSORS;
              set of edge_class OUT_EDGES);
type edge_class = (string LABEL; set of node SUCCESSORS)
```

This choice of information structure determined the algorithms which could be used to manipulate the data base. For example, finding the set of nodes adjacent to node N via an edge labeled E implied scanning the set OUT_EDGES(N) for an edge_class which has LABEL E and returning the corresponding SUCCESSORS. On the other hand, finding the set of nodes to which node N is adjacent via an edge labeled E implied traversing the set of PREDECESSORS(N) and returning all those nodes for which the set OUT_EDGES has an edge_class with LABEL E and with SUCCESSORS including N. The following Algol-like procedures encode these algorithms at the information structure level:

Figure 3 Information structure



```
(set of node) procedure SUCCESSOR (node N; string E);

        begin
        set of node S;
        S := empty;
        for I ∈ OUT EDGES(N) do
            if LABEL(I) = E
                then S := SUCCESSOR(I);
        S
        end;

(set of node) procedure PREDECESSOR (node N; string E);

        begin
        set of node S;
        S := empty;
        for P ∈ PREDECESSOR(N) do
            for I ∈ OUT EDGES(P) do
                if LABEL(I) = E
                    then for J ∈ SUCCESSORS(I) do
                        if J = N
                            then S := S ∪ {P};
        S
        end
```

At the next level, the storage structure must encode each of the data types comprising the information structure. Because sets were expected to be fairly small and insertions and deletions were expected to occur quite frequently, singly-linked lists were chosen to represent sets in storage. In addition, each set was to be headed by an owner element to which the last element of the list was linked.

Finally, the machine encoding binds the data structure to a machine. Since the application was intended to be core-resident on an IBM System/370, pointers were encoded as full-word, absolute addresses and strings were encoded in EBCDIC.

Describing the data structure from all five views allows some improvements to be incorporated into an application in such a way that only

those levels affected need to be altered. For example, moving Cohen's system to another machine or allowing for data segments to be relocated should require changes to the machine encoding only. Decreasing set search time (e.g., by ordering the edge_classes lexicographically by LABEL) should affect the storage structure only. Improving the algorithms (e.g., simplifying reversed edge traversal) will have its major impact on the information structure. Finally, including additional graphs in the data base (e.g., to represent a scenario predicted by the system from its present understanding) or revising the case grammar representation would involve changes to the data abstraction. The step-wise design of data produces a modularity which leads to clarity in description and convenience in data reorganization.

2. Evaluation storage structure alternatives

Although the design of data structures involves a great deal of insight and artistry by the designers, there are stages which can be made algorithmic. In particular, the transition from the third level, the information structure, to the fourth, the storage structure, can be made methodically enough to be automated. To incorporate the algorithm it will therefore be assumed that an application's data structure, being designed according to the scenario given here, has been refined to the level of the information structure (i.e., data relationships have been made explicit by choosing appropriate data types to superimpose and by writing corresponding algorithms to manipulate the data). The next step is therefore to design a storage structure which will be an efficient model of the information.

Because the number of possible storage structures is very large, choosing an efficient one has too often been based on the designer's

intuition and past experience. To reduce the number of choices, the efficiency of possible representations for each of the individual data types involved in the information structure can be evaluated first, and then, using these evaluations, the costs of storage structures produced by the superposition of the representations can be compared. Although the number of choices for the representation of a single data type is much smaller than the number which can represent a composite information structure, even for a single data type many representations have been described in the literature, and many more can still be described. There seems to be no method for effectively organizing the set of all possibilities, and to search the space of all representations would be extremely inefficient, if not impossible.

2.1 The library of storage structures

The approach adopted here is to consider only representations available from a given "library". To contain representations used in current applications, this library would likely include contiguous tables, hash tables, linear linked lists, and a variety of tree structures [Gotlieb 74]. In practice, for an application written in a high level language, the library would consist of those representations which can be generated by the language's translator. To choose a globally optimal storage structure is not the goal here, but rather to choose one which has the most efficient implementation as generated by the translator.

For any data type, the library should contain one or more clusters of code which implement the operations for that type, as could be done in the CLU framework reported in section 1.2. Furthermore, since the choice of representations will depend on the efficiency of the implementation,

it would be convenient for each data type to have an associated cost table. For each potential representation for a data type, this table should contain an entry indicating formulas for expected run time and storage space in terms of parameters which will reflect the role of the data type in the information structure. For example, the parametric table entries for a "sequence" (i.e., an arbitrarily large, ordered collection of similar elements) would contain formulas for representations by contiguous storage and by a singly-linked list. These formulas, expressed in terms of the expected number of elements in the sequence; the expected size of each element; the expected target position for a probe (e.g., whether all probes are at the start or, instead, uniformly distributed over the sequence); and the expected number of insertions, deletions, and sequences traversals, may then be evaluated in light of an application's parametric values as determined from the algorithms used on the information structure.

Recently, two systems have been described for measuring expected run time for a program. In the first [Cohen 74a], a program written in a language similar to Algol 60 is translated into a LISP structure in which each source-level construct is replaced by an expression that parametrically represents the time to execute that program. In the other (Appendix), the syntax of the Counter language was designed such that every symbol in a program has an associated run time, and the expected time for a complete program can be determined easily during parsing. By using either system to encode the basic operations defined for a data type, the run time for each representation in the library can be determined with respect to parameters which represent how many times each function is executed, as well as how

many times loops are executed, whether various conditions will be true or false, and the timing characteristics of the hardware. The resulting formulas form the run time entries in the parametric table.

Unlike the evaluation of run time, the measurement of storage space is not directly dependent on the processes involved in the application. In fact, the amount of storage space can be specified as a function of

- the number of occurring elements,
- the maximum number of elements likely to occur,
- the number of possible elements,
- the number of cells per element, and
- the size of each cell.†

Therefore, parametric formulas which represent the storage space can also be entered into the table.

2.2 Using an evaluation matrix to choose a storage structure

To choose a storage structure for a given information structure, the appropriate parametric table must be evaluated for every "substructure" (i.e., for every instance of a data type appearing as an explicit relation). These calculations will each produce a column of an evaluation matrix, such that the element pair in row i and column j represents the run time and storage space contributed by the j-th substructure when represented by the i-th cluster in the library.††

---

† The possibility of overlapping clusters to share data complicates the analysis of storage space requirements. The user must supply additional semantic information to allow the system to consider shared data. The problem can be circumvented by assuming that the parametric values have been adjusted by the user such that each item of data is reflected once only.

†† The orderings of the substructures and the library's members do not affect the evaluation's outcome, but they may affect the efficiency of the evaluation itself.

The calculation of expected run time is straightforward. From the algorithms at the information structure level, the values of most of a cluster's parameters can be determined, the rest being assigned default values. These values are then substituted into the time formula for the cluster, which will yield the expected run time (in microseconds). This result is then assigned to the appropriate time entry of an evaluation matrix.

The space required by a substructure is the product of the number of cells used by the cluster representing that substructure, and the number of copies of that substructure in the application. Thus, for example, the space required by a cluster representing a personnel record must be multiplied by the number of personnel records in the file. If this product were entered into the matrix, however, the space used by a dependent substructure (one wholly contained within another) may be represented twice: first as the dependent substructure itself, and second as a set of fields within one of the member elements of the substructure on which it depends. To avoid this duplication, it is necessary to multiply only the number of cells for that part of each substructure which would be located outside all other substructures by the expected number of instances. The space entry will therefore contain that product (henceforth called the separate space) separately from the space which will lie within another substructure (the nested space).†

---

† Where programs and data are allocated storage from the same memory, the space required for manipulation routines can also be entered into the matrix separately.

Given such an evaluation matrix and a user-supplied cost formula as a function of space and/or time, a cluster may be assigned to each substructure such that the resulting composite storage structure has minimal cost over all such cluster assignments. Unfortunately, in general, the least cost for the whole application is not achieved when each substructure uses the least-cost cluster. For example, consider an application having the following evaluation matrix in which, for simplicity, only one space entry per matrix element is shown:

		substructures	
		1	2
clusters	1	(50 cells, 50 msec.)	(1 cell, 100 msec.)
	2	(100 cells, 1 msec.)	(40 cells, 60 msec.)

For the cost formula represented by space*time, a matrix of costs could be derived by evaluating each element individually:

		substructures	
		1	2
clusters	1	2500 cell-msec.	100 cell-msec.
	2	100 cell-msec.	2400 cell-msec.

Thus cluster 2 has the least cost for substructure 1, and cluster 1 has least cost for substructure 2. Using these assignments to implement the application would result in a cost of (100+1 cells) * (1+100 msec.) = 10201 cell-msec. This is clearly not minimal: using the first cluster for both

substructures would result in a lesser cost, namely (50+1 cells) * (50+100 msec.) = 7650 cell-msec.

Because choosing the cluster of least cost for each substructure will not guarantee an efficient storage structure, it would seem at first that all possible combinations of clusters must be examined. This is impractical for large applications, however, because the number of cluster assignments grows exponentially in the number of substructures. Therefore, a technique must be adopted for reducing the expected number which need to be evaluated explicitly.

Branch and bound, which has been used successfully in many areas of operations research [Lawler 66], can be applied to direct the search for an efficient storage structure. "The space of all feasible solutions is repeatedly partitioned into smaller and smaller subsets, and a lower bound ... is calculated for the cost of the solutions within each subset. After each partitioning, those subsets with a bound that exceeds the cost of a known feasible solution are excluded from all further partitioning."

Applied to this algorithm, the branching phase partitions sets of potential storage structures by considering one substructure at a time. Consider, for example, an application consisting of three substructures, each of which can be represented by any of four clusters (thus there are $4^3 = 64$ potential storage structures). The set of all potential storage structures can be partitioned along the first substructure into four subsets: those in which the first substructure is represented by cluster 1, by cluster 2, by cluster 3 and by cluster 4, respectively. It is convenient to represent the state of cluster assignments by an assignment vector in which each position corresponds to a substructure and each non-null entry corresponds

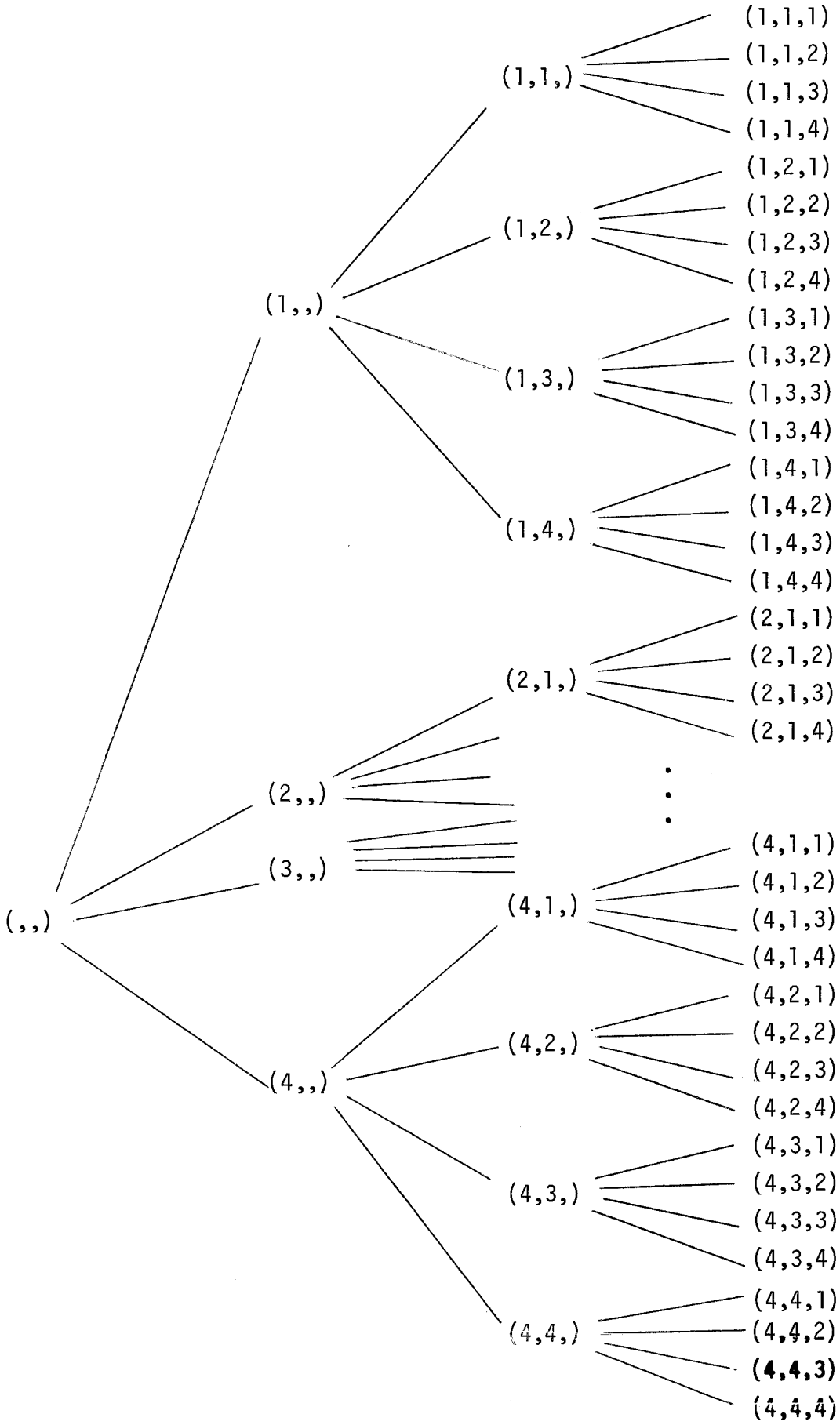
to the number of the cluster assigned. For example, the initial assignment vector would be (,,). After branching along the first substructure, four new assignment vectors would be generated (1,,), (2,,), (3,,), and (4,,). Choosing the second of these for subsequent branching would result in four new subsets of potential storage structures, represented by (2,1,), (2,2,), (2,3,), and (2,4,). Finally partitioning the first of these would produce four fully-defined storage structures (2,1,1), (2,1,2), (2,1,3), and (2,1,4). The complete branching into subsets can be illustrated in the form of a tree of assignment vectors (Figure 4).

Because the clusters are to be superimposed in a given machine, and not only in a mathematical model, the characteristics of the machine must be considered in order to avoid any contention (e.g., for limited memory space) [Gotlieb 74]. Thus, given an assignment vector, v, and a substructure, j, for which v(j) = null, a preliminary pruning phase must be invoked to eliminate those clusters which would lead to contention if assigned to the j-th substructure. That phase can further reduce the number of possible cluster assignments for v(j) by eliminating those for which the cost for performing a needed operation is prohibitive (e.g., an implementation requiring a complete scan of memory).

The branching of a set of cluster assignments, represented by vector v, into subsets is therefore accomplished as follows:

- Find the index of the first null entry, J. (The algorithm is such that if v(J) = null, then v(j) = null for all j > J.)
- Use the preliminary pruning phase to eliminate all clusters which cannot be chosen for the J-th substructure.
- For each of the k clusters $i_J^1,\ldots,i_J^k$ remaining in the library, create a new assignment vector v^I such that $v^I(j) = v(j)$ for $j \neq J$, and $v^I(J) = i_J^I$.

Figure 4 Branching of a set of storage structures into subsets



If the set of all potential storage structures were to be fully partitioned, as implied by Figure 4, and each node of the tree were to be evaluated, the cost for evaluation would not be decreased. It is the second stage of branch and bound which attempts to prune the tree by finding lower bounds on cost for all members of designated subsets. For example, if a lower bound on cost for the storage structures in the set represented by assignment vector (1,,) were found to be greater than the cost for the storage structure represented by (2,1,3), there would be no need to evaluate the left quarter of the tree in Figure 4. When the cost formula, F, can be assumed to be a non-decreasing function of run time and storage space, one lower bound on cost is the value of F applied to the fastest possible time and the smallest possible space (which are not necessarily simultaneously achievable).

To calculate the fastest possible time for a set of assignments v, a temporary assignment vector Tv is constructed, in which if v(j) ≠ null Tv(j) is set to v(j), and if v(j) = null, Tv(j) is set to the index of the cluster requiring least time after pruning. (If several clusters have the same minimal time, one of those with least space should be chosen for Tv(j).) It is important that, for each substructure, the library is pruned with respect to v, rather than to that part of Tv which has already been constructed, in order that the clusters are chosen independently for all null entries in v. The least time for v, t(v), is the sum of the run times for the components of Tv.

To calculate the least possible space for v, a second temporary vector Sv is constructed similarly, except that Sv(j) is set to the index of the cluster requiring least space for the member elements after pruning

for each of the null entries in v. The total space for a component of Sv must include the cells required for the nested space of any dependent substructures if and only if the index in v for that dependent substructure is not null; otherwise, the value for the smallest possible nested space should be used. Furthermore, for a substructure which is not contained within any others, the value for its own nested space must be added, regardless of the null entries in v. The least space for v, s(v), is then the sum of the space required for the components of Sv.

The lower bound on cost for v, denoted c(v), is therefore F(t(v),s(v)). If (1) Tv = Sv, (2) each null entry in v which corresponds to a dependent substructure has been assigned a cluster in which the nested space is minimal in size, and (3) all the clusters represented by Tv happen to be free from contention, then the storage structure is a feasible solution for which the cost therefore provides an upper bound for the application. It is this upper bound which is used to eliminate sets of storage structures having greater lower bounds on cost.

The complete algorithm for searching an evaluation matrix is thus illustrated in Figure 5. The algorithm, given an evaluation matrix and a cost formula, returns the assignment vector and cost for the most efficient storage structure.

2.3 An example of storage structure design

Consider again the natural language understanding system described in section 1.3. The information structure chosen for that application (Figure 3) includes the following processes:

Figure 5 Algorithm for searching an evaluation matrix

```
Determine Tv, Sv, and c(v^0) for v^0, the assignment vector having
   null entries only
If Tv is a feasible solution
   Then Set Choice to the pair <Tv,c(v^0)>
   Else Initialize Choice with the pair <null, infinite cost>
        Initialize S, a set of vectors to consider, with the
           ordered pair <v^0,c(v^0)> as its only member
        While S is not empty:
           Choose the assignment vector, V in S, with least c(V)
              and delete the corresponding ordered pair from S
           Apply the branching to V to obtain new vectors, V'
           For each of the V':
              Determine Tv, Sv, and c(V')
              If c(V') is less than the cost for Choice
                 Then If Tv is a feasible solution
                         Then Replace Choice by <Tv,c(V')>
                                 For each ordered pair <v,c(v)> in S
                                    If c(v) > c(V')
                                       Then Delete the pair from S
                                    end "For each" loop
                         Else Add <V',c(V')> to S
              end "For each" loop
           end "While" loop
If the cost of Choice is finite
   Then The assignment vector chosen has minimal cost
   Else There is no solution for the application
```

P1. Determine whether there is an edge beginning and ending at given nodes (hereafter called the source node and target node, respectively), such that the edge has a given label.

P2. Find a target node, given a source node and an edge label.

P3. Find a source node, given a target node and an edge label.

P4. Add a labeled edge between two (not necessarily distinct) given nodes.

P5. Delete a given edge between two given nodes.

P6. Add a node to the graph.

P7. Delete a node from the graph, and delete all edges to and from that node.

For each of these processes, an algorithm has been written in terms of the operations defined on the data types involved (i.e. sets).

The next step in the design process is to determine the values for the parameters appearing in the library's tables, which for the example will be assumed to contain formulas for fifteen clusters for representing sets. For representing a short English-language discourse it may be assumed that there are approximately 100 nodes, an average of 5 edge labels per node, 1.5 target nodes per edge label, and 4 predecessors per node. Furthermore, it will be assumed that there is exactly one loop on each node (specifying the "name" of that node), and 25 distinct labels appearing on edges between pairs of nodes anywhere in the graph. Finally, the relative frequencies of invoking P1 through P7 is estimated to be 25%, 20%, 20%, 10%, 10%, 5%, and 5%, respectively.†

† The expected values were estimated by P. Cohen and L. Melli from their experience with this application.

This presentation will by-pass choosing a cluster for the first substructure, the set of nodes. For reasons which are superfluous to the illustration, it will be assumed that that substructure will be represented by the second cluster in the library; thus v^0, the initial vector for the algorithm, is (2,,,).

Figure 6 illustrates the evaluation matrix for this application, in which the run times represent the expected time for any one invocation of a process. When the branch and bound method is invoked, Tv and Sv must first be calculated for v^0. From Figure 6 it can be seen that Tv = (2,2,6,2) and that Sv = (2,11,11,11). Since the application was designed to be used at the University of Toronto, that university's charging algorithm will be assumed; thus,

$$F(s,t) = (1.42*10^{-7})t + (.7*10^{-10})t*s + (.56*10^{-18})t*s^2$$

where t is in microseconds and s is in cells (which are assumed to be four bytes each). Therefore, assuming that data manipulation dominates, the lower bound on cost is $c(v^0) = \$.99*10^{-4}$ for each process invocation. Since Tv $\neq$ Sv, Tv is not a feasible solution, and the branch and bound algorithm proceeds to search the matrix.

After finding the costs of 73 assignment vectors (which represent 729 potential storage structures having finite cost), the algorithm terminates producing the vector (2,6,6,6) and a cost of $\$1.91*10^{-4}$ for each process invocation.† This choice represents a 14% improvement in run time over the

† The sixth cluster represents sets as described in section 1.3, except that member elements are ordered by value and the owner element contains a "dummy value" which is greater than any member's value.

Figure 6 Evaluation matrix for Cohen's system

Substructure 1 is to be implemented using the second cluster: thus the single matrix entry required is (33, (2,3)).

	Substructure 2			Substructure 3			Substructure 4		
		space			space			space	
cluster	time	separate	nested	time	separate	nested	time	separate	nested
1	186.6	15	3	134.3	22.5	3	81.6	12	3
2	156.7	15	3	127.6	22.5	3	73.5	12	3
3	190.9	20	4	213.3	30	4	104.1	16	4
4	157.7	20	4	131.3	30	4	74.3	16	4
5	∞	10	2	∞	15	2	∞	8	2
6	161.4	10	2	124.3	15	2	75.3	8	2
7	∞	15	3	∞	22.5	3	∞	12	3
8	323.2	20	4	264.4	30	4	148.8	16	4
9	∞	25	5	∞	37.5	5	∞	20	5
10	320.8	25	5	267.2	37.5	5	153.1	20	5
11	274.9	0	23	153.9	0	7	100.8	0	27
12	736.5	0	23	409.3	0	7	677.1	0	27
13	∞	0	∞	∞	0	∞	∞	0	∞
14	∞	5	14	∞	7.5	9	∞	4	29
15	∞	15	3	∞	22.5	3	∞	12	3

storage structure originally designed and adopted for the application. Unfortunately, the actual cost reduction is only 4% because of an increase in storage space.

After reconsidering the information structure in light of insights provided by using this algorithm, it was redesigned to consist of three substructures only: a set of labeled nodes, a set of out-edges for each node (i.e., edges for which the node is a source), and a set of in-edges for each node (i.e., edges for which the node is a target). This revision requires new algorithms to be written for the processes. Again an evaluation matrix can be constructed using the new parametric values, and the branch and bound algorithm can be invoked to find the best representation. The cluster assignment which results is (2,6,6), for which the cost is only $\$1.66*10^{-4}$ per process invocation. This represents a 16% cost reduction with respect to the application's original design.

3. Conclusions

This paper presents a procedure for choosing a storage structure which will be efficient for a given application. The procedure is divided into three phases:

- design of the information structure
- calculation of the corresponding evaluation matrix
- search for an efficient composition of simple clusters

The algorithmic approach to choosing storage structures is superior to intuitive methodologies in several respects. Foremost, the algorithm includes a comparison of all structures which can be realized by superimposing

components from a library. Therefore, a cluster which is traditionally used in one specific application, and which is therefore included in the library, will be considered for any substructure having the same data type. Although the resulting choice may not be counter-intuitive, it is often not what intuition would dictate.

A second aspect favouring the algorithm is its speed. Because of the parametric approach, the evaluation matrix entries for each substructure can be calculated quickly. While it is true that searching the matrix may require an amount of time that grows exponentially in the number of substructures, the branch and bound technique is a systematic search which attempts to eliminate many potential computations. Furthermore, there have been several heuristics suggested for increasing the efficiency in branch and bound searches, including methods for computing suboptimal solutions [Lawler 66]. Additional improvement in the algorithm's speed can be obtained by judiciously ordering the substructures (matrix columns), such that the substructures having the least number of choices and the "clearest" choice of clusters occur first. (In this way, the exponentiality of the search will be reduced.) Therefore, unlike traditional hand analyses, the method is fast enough to allow many alternatives to be considered. In fact, using a PL/C [Conway 73] implementation of the algorithm, each of several examples required less than three seconds execution time on an IBM 370/165 [Tompa 74].

Research into the evaluation of storage structures can be continued by examining data base applications such as those which use the Information Management System [IBM 71]. The description manual for that system raises the questions that have been considered here, but it fails to answer them

satisfactorily. For example, "Within any physical data base a mixture of... pointers may be employed... The reader is advised to review the advantages of each pointer technique."

The choices which confront the Information Management System user can be made systematically. After a data abstraction is designed for an application, the Information Management System requires that the data be separated into segments. This step would be part of the design of the information structure, and the relations which result correspond to sub-structures. Before applying the algorithm to choose a storage structure, the model used to generate parametric formulas must be extended to encompass a two-level store [Tompa 74]. Next, a library must be compiled to contain clusters which utilize the possible linking methods for a segment, including contiguous stores, singly and double linked lists, hashes, and any other implementation which may be supported by the Information Management System. (The basic set of operations implemented for each cluster would consist of all data management commands defined, including GET UNIQUE, GET NEXT, INSERT, and REPLACE.) Finally, the preliminary pruning phase must incorporate any restrictions imposed by the Information Management System, e.g., "hierarchical pointers cannot be used to connect segments in different data set groups." After these steps are taken, users of the Information Management System will be able to apply the algorithm presented here to design (or redesign) data bases.

ACKNOWLEDGEMENTS

Thanks are due to C.C. Gotlieb, R. Peebles, S. Osborn, and B. Jolliffe for their help in formulating these ideas, and to P. Cohen for his donation of the natural language understanding system. Acknowledgement is also given to the University of Toronto, the University of Waterloo, and to the National Research Council of Canada for their financial support.

APPENDIX

Counter, a tool for measuring run time

A program's run time is assumed to be a linear function of the number of executions of each of three primitive instruction types: memory accesses, arithmetic and logical instructions, and transfers of control. That is, the time can be expressed as

TIME = c1*M + c2*A + c3*T (*)

where M is the (average) time to access one memory cell; A, the average time to perform one arithmetic or logical instruction: T, the time to transfer control; and the coefficients represent the number of executions of each primitive type.

The Counter language was designed to allow the expected run time for a program to be calculated easily at compile time. The output from the parse phase of program translation includes an expected run time for the program in terms of the number of each primitive instruction type which will be executed.

The complete syntax for Counter is described elsewhere [Tompa 74]. Every symbol in a Counter program has an associated run time (i.e., there are no superfluous punctuation marks or dummy keywords).† Counter's tokens (the symbols which are separable by a lexical analyzer) are divisible into six categories, each of which is recognizable by its "type font".

- Tokens written in upper case letters, numbers, and underscores (e.g., VAR, PARSE_PTR, 25, 3_) are identifiers or constants, which each represent one memory access at execution time.
- Periods signify implicit memory accesses, and therefore represent one access each.

† Counter was not designed to be a user-oriented language, but rather the target language for translators of higher level languages.

- Tokens consisting of special symbols, such as +, =, and *, each require one arithmetic or logical operation. (For simplicity, they will be said to represent one arithmetic each.)
- Tokens written in underscored lower case letters (e.g., <u>if</u>, <u>out</u>) are instructions which each represent one transfer of control.
- Tokens written in lower case letters and surrounded by parentheses (e.g., (cycle), (procedures)) are control instructions which do not add to run time themselves, but rather indicate that the subsequent sections of code may be repeated, and thus that their run times will be multiplied by some factor.
- Tokens written in quoted lower case letters represent "macro instructions", each of which can be expanded in terms of the three primitive instruction types. In order that the run time calculated remains independent of any given operating system, the macro instructions are assigned parametric times as follows:

<u>macro</u>	<u>time</u>
"call"	call.time = call.m*M+call.a*A+call.t*T
"allocate"	alloc.time = alloc.m*M+alloc.a*A+alloc.t*T
"free"	free.time = free.m*M+free.a*A+free.t*T
I/O (i.e. "get" "put","title")	io.time = io.m*M+io.a*A+io.t*T

Thus, by scanning a line of code for type fonts alone, a parser for Counter can immediately calculate the run time for executing that line. For example, a comparison between "the contents of the second cell past the location referenced by A" and "the contents of the third cell past the location referenced by the seventh cell past the location referenced by 3" can be made using the following line of code:

```
if 2  A = 3  7  B
```

Execution of this line requires five memory accesses, one arithmetic (for =), and one transfer of control (if) to the "then" or "else" sections of code.

In order to determine the run time for a program, the times for individual lines in that program must be aggregated. The parser for Counter regards time as a polynomial in many unknowns, where each unknown represents a probability for the number of times a cycle or a subroutine is executed, the probability that an exit is taken from a cycle, and the probability that a condition in a if statement is met. The coefficients of the polynomials are seven-tuples whose components represent a number of memory accesses, arithmetics, transfers of control, calls, allocates, frees, and I/O instructions, respectively.

Each production in the Counter grammar has a corresponding formula which generates a polynomial, named Time, and a set ordered pairs which is used to maintain the run time for the paths of control flow involved in leaving cycles. If, for each production invoked during the recognition of a program, these two are calculated using Counter's formulas, the run time for the complete program will be represented by the Time generated for the root of the parse tree.†

As an example of the results from Counter, consider Figure 7. Beside each line of code is its run time in terms of the three primitives instruction types (represented by MA, AR, and TR) and the four macro-level

---

† The correctness of the formulas as applied a particular program could be proven by induction on the parse tree associated with the recognition of that program.

Figure 7 Counter applied to a routine with conditionals and loops

```
MA AR TR C A F I  #   CARD IMAGE                                              CONDITION
                     ** this subroutine sets PARMN000 depending on the      **
                     ** values of A and B, and then cycles through a linked **
                     ** list looking for a value less than PARMN002         **
 2  1  1 0 0 0 0  1     if A < B                                          @ ALESS
                     *     compare A to B, and if less:
 3  0  0 0 0 0 0  2       PARMN000 1_ PARMN001
                     *       assign to PARMN000 the value found in the first
                     *       cell of the block referenced by PARMN001
 2  0  0 0 0 0 0  3       PARMN001 NULL
                     *       set PARMN001 to null
 0  0  1 0 0 0 0  4     out
                     *     leave the then part
 3  0  0 0 0 0 0  5       PARMN000 2_ PARMN002
                     *       assign to PARMN000 the value found in the second
                     *       cell of the block referenced by PARMN002
 3  1  0 0 0 0 0  6       A A 1 +
                     *       increase A by 1
 0  0  0 0 0 0 0  7     (cycle)                                           @ N
                     *     loop until exit is executed
 3  1  1 0 0 0 0  8       if 2_ PARMN000 < PARMN002
                     *       if comparison yields true, then exit the cycle
 0  0  1 0 0 0 0  9         exit                                          @ 1
 3  0  0 0 0 0 0  10      PARMN000 1_ PARMN000
                     *       set PARMN000 to the value found in the first
                     *       cell of the block referenced by PARMN000
 0  0  1 0 0 0 0  11      end
                     *       repeat the cycle
 0  0  1 0 0 0 0  12    return
                     *     return from the subroutine
          produces:    (2, 1, 2)
                      + (5, 0, 1) (ALESS)
                      + (6, 1, 0) (1-ALESS)
                      + (6, 1, 2) (N)*
                      + (3, 1, 2) (1)
```

instructions (represented by C, A, F, and I). The time for a complete conditional statement is equal to the time for the if instruction plus the aggregated time for the "then" part when the condition is met, or the aggregated time for the "else" part (if it exists) when the condition is not met.† The fraction of times the condition is met is represented by a parameter in the condition column on the line of code containing the if. (When the "then" part ends with an exit, the parameter is instead listed on the exit line.) The expected time is thus reflected in the output from Counter by a triple representing the aggregate time for the "then" part times the parameter, plus the aggregate time for the "else" part times one minus the parameter. For example, lines 2 through 4 correspond to the "then" part of a conditional and therefore the run time is reflected in Counter's output by

+ (5, 0, 1) (ALESS)

In evaluating the time for a cycle, the aggregate time for the body must be multiplied by the number of times the cycle is executed. For each exit from the cycle, the time for that part of the body leading to the exit must be multiplied by the probabiility of taking that exit. Again, the parameters representing the expected number of cycles and the exit probabilities are shown in the condition column of the listing. The time which is output from Counter for the cycle in lines 7 through 11 of Figure 7 is

+ (6, 1, 2) (N)*

+ (3, 1, 2) (1)

---

† In Counter, the conditional statement has an explicit if, but the "then" and "else" parts are indicated by formatting.

The resulting time for the complete routine is therefore printed as a sum of triples which represents the TIME formula

$$(6N+11-ALESS)*M + (N+3-ALESS)*A + (2N+4+ALESS)*T$$

where N represents the number of repetitions of the cycle and ALESS is one if A<B, and zero otherwise. This expression can be evaluated for any set of parameter values provided for the unknowns.

Although Counter has been designed to model a very simple computer, the techniques used to generate the parametric table are applicable to more complex models. As a first extension, rather than merely dividing the primitive instructions into three types, a finer division can be used to account for the variance in time for several arithmetic and logical instructions or branches (e.g., addition and subtraction may take two microseconds each, while multiplication and division require ten). New instructions can also be added to Counter's repertoire to access blocks of cells (such as IBM 370's "move character", "compare logical character", and "translate and test") and IBM 1130's "shift left and count accumulator"), to simplify control structures (e.g., a "case" statement, implicit function calls, and dynamic stacking of activation records), or to improve input/output.† The techniques involved in computing run time remain the same after any of these modifications.

---

† In fact, many of these syntactic extensions can be achieved simply through additional macros.

References

[Astrahan74] M.M. Astrahan and S.P. Ghosh, A search path selection algorithm for the Data Independent Accessing Model (DIAM). ACM SIGMOD Workshop on Data Description, Access, and Control (1974) 367-388.

[Balzer67] R.M. Balzer, Dataless programming. Proc. AFIPS 31 (FJCC 1967) 535-544.

[Codd70] E.F. Codd, A relational model for large shared data banks, Comm. ACM 13, 6 (June 1970) 377-387.

[Cohen74a] J. Cohen and C. Zuckerman, Two languages for estimating program efficiency, Comm. ACM 17, 6 (June 1974) 301-308.

[Cohen74b] P.R. Cohen, A prototype natural language understanding system. Department of Computer Science, University of Toronto, Tech. Report 64 (March 1974), 149 pp.

[Conway73] R. Conway and D. Gries. An Introduction to Programming. Winthrop Publishers, Inc. (1973),460 pp.

[Dijkstra72] E.W. Dijkstra, Notes on structured programming. in Structured Programming (Dahl, Dijkstra, and Hoare, ed) Academic Press, New York (1972), 1-82.

[d'Imperio69] M.E. d'Imperio, Data structures and their representation in storage. Annual Rev. in Auto. Prog. 5. Pergamon Press, Oxford (1969), 1-75.

[Earley71] J. Earley, Towards an understanding of data structures. Comm. ACM 14, 10 (October 1971), 617-627.

[Fry70] J.P. Fry, Introduction to storage structure definition. Proc. of the ACM SICFIDET Workshop on Data Description and Access. (November 1970, 2nd edition, April 1971), 315-337.

[Furtado75] A.L. Furtado, Characterizing sets of data structures by graph grammars. Proc. of the Computer Graphics, Pattern Recognition, and Data Structures Conference (1975).

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

[Hoare72] C.A.R. Hoare, Notes on data structuring. in Structured Programming (Dahl, Dijkstra, and Hoare, ed.) Academic Press, New York (1972), 83-174.

[Honig74] W.L. Honig, Bringing data base technology to the programmer. FDT 6, 3 (1974), 2-15.

[IBM71] IBM. Information Management System/360, version 2 system/application design. IBM Corp., White Plains (1971).

[Knuth71] D.E. Knuth, Optimum binary search trees, Acta Informatica 1 (1971), 14-25, 270.

[Lawler66] E.L. Lawler and D.E. Wood, Branch and bound methods: a survey. Op.Res. 14, 4 (1966), 699-719.

[Lesk70] M.E. Lesk, Compressed text storage. Bell Labs. Tech. Rep. 3 (Nov. 1970), 34.

[Liskov74] B. Liskov and S. Zilles, Programming with abstract data types. Proc. of a Symp. on Very High Level Languages. SIGPLAN Notices 9,4 (April 1974), 50-59.

[Lum71] V.Y. Lum, P.S.T. Yuen, and M. Dodd, Key-to-address transform techniques: a fundamental performance study of large existing formatted files. Comm. ACM 41, 4 (April 1971), 228-239.

[McCarthy60] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine - part 1. Comm. ACM 3, 4 (April 1960), 184-195.

[McCuskey70] W.A. McCuskey, On automatic design of data organization. Proc. AFIPS 37 (FJCC, 1970), 184-199.

[Mealy67] G. Mealy, Another look at data. Proc. AFIPS 31 (FJCC, 1967), 525-534.

[Nievergelt72] J. Nievergelt and E.M. Reingold, Binary search trees of bounded balance. Proc. of the 4th Annual ACM Symp. on the Theory of Computing (1972), 137-142.

[Parhami73] B. Parhami, Associative memories and processors: an overview and selected bibliography. Proc. IEEE 61, 6 (June 1973), 722-730.

[Randall72] L.S. Randall, A relational model of data for the determination of optimum computer storage structures. Systems Engineering Lab., Univ. of Mich., Tech. Rep. 54, RADC-TR-72-25 (Feb. 1972), 468.

[Roberts65] L.G. Roberts, Graphical communication and control languages. Second Congress on the Information Systems Sciences (1965), 211-217.

[Ross70] D.T. Ross, Uniform referents: an essential property for a software engineering language. Software Engineering 1 (Tou, ed.) Academic Press, New York (1970), 91-101.

[Senko73] M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Fehder, Data structures and accessing in data base systems. IBM Sys. J. 12, 1 (1973), 30-93.

[Shneiderman74] B. Shneiderman and P. Scheuermann, Structured data structures. Comm. ACM 17, 10 (October 1974), 566-574.

[Smith71] D.P. Smith, An approach to data description and conversion. Ph.D. thesis, Moore School, University of Pennsylvania (1971).

[Tarjan72] R.E. Tarjan, Depth-first search and linear graph algorithms. SIAM J. Computing 1, 2 (June 1972), 146-160.

[Tompa74] F.W. Tompa, Choosing a data storage schema. Ph.D. thesis, Department of Computer Science, University of Toronto (1974), 241 pp.

[Waite70] W.M. Waite, The mobile programming system: STAGE 2. Comm. ACM 13, 7 (July 1970), 415-421.

[Weizenbaum63] J. Weizenbaum, Symmetric list processor. Comm. ACM 6, 9 (Sept. 1963), 524-536.

[Wirth73] N. Wirth, Systematic Programming: An Introduction. Prentice-Hall, Englewood Cliffs (1973), 169 pp.