TECHNIQUES FOR THE AUTOMATIC SELECTION OF DATA STRUCTURES

James Low and Paul Rovner
Computer Science Department
The University of Rochester

TR4

TECHNIQUES FOR THE AUTOMATIC SELECTION OF DATA STRUCTURES

James Low and Paul Rovner
Computer Science Department
University of Rochester
Rochester, New York 14627

We are all aware of the development of increasingly sophisticated, elaborate, and expensive computer programs, particularly in the fields of artificial intelligence, data base management, and intelligent systems. The need for techniques to deal with such complexity has renewed interest in programming language research. Recent work on structured programming, intelligent compilers, automatic program generation and verification, and high-level optimization has resulted.

A pattern of approach similar to that of earlier research on programming languages is emerging. The work divides naturally into two parts: the search for good linguistic tools for expressing algorithms and data, and the development of practical methods for translating these to working computer programs. Our emphasis in this paper is in the latter.

For programs that are inherently complex and expensive, the intelligent choice of implementation-level representations for high-level data is a central problem. This paper contains a discussion of several powerful techniques for automating such intelligent choices for a given program. Flow analysis, execution monitoring, and interactive sessions about the characteristics of the problem are considered. Built-in knowledge for a collection of data structures is assumed. For each data structure, this includes rules about its applicability and formulas for its memory space and execution time requirements as functions of the properties of the data and the operations of the program.

The context for the discussion is an experimental system [Low74, Rovner76] which chooses data structures and algorithms for sets, lists, and associations in the ALGOL-60 based programming language SAIL [VanLehn73]. (A brief description of SAIL is contained in Appendix I.) Similar abstract data structures are to be found in other experimental programming languages, including QA4 [Derksen72], MICRO-PLANNER [Sussman70, Baumgart72], SETL [Schwartz75], MADCAP [Morris73], VERS2 [Earley73] and CONNIVER [Sussman72, McDermott72].

This paper is organized in two parts. The first comprises an introduction to our methods and an overview of the selection system. The second contains a discussion of our recent work on automatic selection of associative data structures. An appendix contains detailed examples of this work.

System Overview

The selection system is based on the premise that there are many different ways of representing the sets, lists, and triples of a user's program. A fixed library of such representations is built into the system. A compiler (with occasional help from the user) selects representations which will tend to minimize some cost function (such as amount of cpu time) for the entire program. It does this by predicting the costs of using the various applicable representations for the user's abstract data structures. Then it chooses those representations which seem to minimize the toal cost. First it selects representations by acting as if the choices for the various abstract structures were independent. Later it considers how these representations interact. For example, one cost function is the space time product. If the program has two abstract data structures, the compiler has to consider the cost component of the space occupied by each data structure multiplied by the time of execution of operations on the other. The compiler may change its choice if these interactions so dictate.

The amount of storage needed for the representation will influence the total cost of a representation. The storage cost for a representation is usually quite easy to estimate. Consider the representation of a set by a binary tree. We estimate the number of storage cells needed by a set of N elements by simply adding the amount of storage needed by a header node to N times the number of storage cells needed for each node of the binary tree. A header might consist of two cells, a count of the number of elements in the set and a pointer to the root node of the tree. Each node of the tree contains three memory cells: one for the element of the set, and one each for the right and left links. The storage occupied by a set of four elements would be 14 memory cells: two for the header and three each for the nodes containing the elements of the set.

Note that this notion of storage cost is an approximation. A more precise system would have to include such factors as the storage allocation algorithm (in a buddy system we might be able to only allocate blocks which are powers of two and thus each node in the tree above would require four cells instead of three), whether some form of virtual memory such as paging or segmentation was being used, and whether there are more than one kind of local storage such as a fast semiconductor memory and a slower extended core storage.

The time cost of a representation is somewhat harder to estimate. The time cost of a representation is the sum of the costs of the individual primitive operations used in a given program. The time cost of a primitive operation may be estimated by looking at the routine which implements it. We count the number of machine instructions (weighted by their individual execution times) which will be executed. This will be a function of a number of relevant parameters of the abstract structure. Consider the following routine for determining if an item is an element of a set which is represented by an unsorted linked list terminated by a NIL pointer.

(a) LOAD the "item" being sought.
(b) LOAD the "ptr" to the first node of the linked list.
(c) COMPARE "ptr" to NIL and
(d) if equal RESULTIS false
(e) COMPARE "item" to the data part of the node pointed to by "ptr"
(f) if equal RESULTIS true
(g) LOAD the "ptr" from the link field of the current node
(h) JUMP to (c)

The execution times of the instructions are: LOADs take two time units (microseconds); COMPAREs take three time units; RESULTIS takes (for a LOAD and a JUMP) three time units; and JUMPs take one time unit. We analyze the routine to see how many times each of its statements will be executed. Parts (a) and (b) will be executed once independent of the size of the set. If the item is in the set then part (c) will be executed from 1 to N times. It will be executed N+1 times if the item is not in the set. Part (d) will be executed no times if the item is in the set and once if the item is not in the set. Part (e) will be executed from 1 to N times if the item is in the set and N times if the item is not in the set. Part (f) will be executed once if the item is in the set and no times if the item is not in the set. Parts (g) and (h) will be executed from zero to N-1 times if the item is in the set and N times if the item is not in the set. We assume that if an item is in the set, it is equally likely to be in any node of the list. We thus estimate the average frequency of statements (c) and (e) when the item is an element of the set as (N+1)/2. The frequency of (g) and (h) is similarly estimated as (N-1)/2. Thus, there are only two important parameters which affect the execution time of this routine: N the size of the set and P the probability that the desired item is actually in the set.

The total time cost of the routine is derived by summing the weighted instruction times:

$$C(N,P) = 9N+10-P*(9N+3)/2$$

We repeated this type of analysis for each of the primitive operations implemented for representation in our library. See Table I for an example set of time cost functions.

TABLE I

PUT SET - insert item in set
Π = proportion of time item already in the set
λ = average size of set
M = maximum size of set

REPRESENTATION	Set Empty	Set Non-Empty
Linked list	100	$64 - 42*\Pi + 6*\lambda$
AVL tree	56	$180-166\Pi+16.8*LOG_2(\lambda)$
Bit - Array	$146 + 3*\lceil M/32 \rceil$	48
Hash table	521	$82-40\Pi + 3\lambda/16$
Bit-string with unsorted linked list	$265 + 3*\lceil M/32 \rceil$	$104-53\Pi$
Attribute bit	27	
Sorted variable length array	140	$96-80\Pi+5.05*\lambda + 20.5*LOG_2(\lambda) -.3\Pi*\lambda$

Our system has a library of representations for sets, sequences and triples and a library of corresponding space and time cost functions. Before the compiler attempts to choose representations for a given program it must obtain information about the use of the abstract data structures within the program. This information is obtained in three ways: static analysis, monitoring, and user interrogation.

The static analysis phase constructs a flow graph of the program and does a symbolic evaluation of the graph. This gives a model of the potential contents of all the set and list variables and the store of triples in terms of both pre-declared items and dynamically created items. By flow analysis, we can usually prove that variables will never contain certain items and that the associative store will never contain certain triples of items. Static analysis is also used to obtain a partition of variables into classes which should be considered as units when choices of representation are made. There is usually no inherent reason why any two sets of a program need to be represented in the same way. However, translation of representation is usually quite expensive and in general should be avoided. Our static analysis determines which variables are used in the same expressions (e.g. union and intersection) or are involved in the same assignment statements. Such variables will be represented in the same way. The partitioning also finds which operations are performed on members of each representation class. This information is used to eliminate certain representations as candidates for a class because they don't provide all the required primitives.

Monitoring sample executions of the program with user supplied data sets and direct interrogation of the user provide the frequency of each primitive operation as well as the parameters to the time and space functions. In the current implementation, monitoring is used to determine the frequency of the various statements of the program. The compiler asks the user the values of such things as the average size of a set at a given point in a program, amount of overlap between sets involved in a union or intersection and so forth. Thus the system can make reasonable guesses of the values of the cost functions.

The final selection is done by choosing those representations which minimize the total expected cost of a program (space-time product) based on the information drived above. Interactions between choices are considered in the analysis of the tradeoffs of space and time for representations of individual abstract data structures.

Selection of Associative Data Structures

The selection system provides a general framework in which to study data structure design. A large part of our recent effort has been directed towards the extension of the system to the selection of associative data structures. In particular, we are looking at several alternate data structures for the store of triples in SAIL (see Appendix II). These structures consist of variations and combinations of four basic types: records, hash tables, property lists, and inverted files.

There are many different associative data structures. We have chosen a representative collection of structures to analyze (see Appendix II). We purposely chose enough structures to require the development of methods for managing combinatoric explosion in the selection process. A real selection system would be impossible without such methods. In addition, the collection of structures is rich enough to allow the system to consider the issue of sharing vs. redundancy as it selects a data structure.

An analysis of these structures to yield rules for their applicability and formulas for their space and time costs is complete. A detailed example of these formulas is contained in Appendix II.

As in the case of sets and lists, selection of a data structure for triples is based on the applicability of the structure to the operations of the program, and on estimates of program execution time and storage requirements. These estimates are derived from a static analysis of the program, from monitoring example executions, and from questions asked of the programmer. Appendix II contains examples of such questions.

The Associative Model

As the system analyzes a program, it builds a model of the store of triples and the operations that are performed on it.

The model of the store of triples reflects the use of these operations by the program. The set of "make" operations is divided into classes. Two make operations are put into the same class if there is an "erase" or "search" operation that could match triples created by both. For each class, the associated erase and search operations are listed. This partitioning reduces the problem of analyzing the entire store of triples for data structure selection to several smaller problems. No operation effects more than one class.

Within a class, the search and erase operations are categorized by their "form". This is determined by the arguments to the operation: whether they are given items, "bound variables", or "any". There are 27 (3x3x3) forms.

Proposing Candidate Data Structures

The system uses the structure inherent in a set of associative operations to identify candidate data structures for the class of triples. For example, assume that the form of all search and erase operations is such that the Attribute and Object positions are always specified and there are a fixed number of items (known at compile time) that could appear in the Attribute position. Each such item could be used to select a fixed field of the given Object in which to find a Value. Moreover, if there will be only one Value for a given Attribute-Object pair, then there is no need to allow for a set of Values. This data structure is termed a "field selection record". A similar collection of such rules is associated with each data structure that the system knows about.

One of the problems arising from our decision to deal with a large library of data structures is combinatoric explosion in the selection process. Much of our effort was spent in devising methods to manage this. In addition to the early use of applicability rules, the system computes and stores useful information about each operation to avoid re-computing it as each data structure candidate is considered. Also, sets of candidates which differ in minor ways are treated as units whenever

possible. For instance, in the previous example, if there is more than one choice for which component of the triple form to consider the Attribute, the candidate structure is not split into separate candidates for each choice unless a preliminary cost analysis indicates that different costs are associated with the different choices. If one choice is worse than some other in both space and time, it is eliminated.

The proposal of candidate structures proceeds in three phases. First, each search and each erase operation is analyzed to determine the set of applicable associative retrieval techniques. These include the selection of a field of an object record, hash table lookup, property list search, and searching various types of lists of triples. Next, each data structure is considered in a process of "matching": if the associative retrieval techniques provided by the structure realize the needs of the set of operations, and the applicability rules of the structure are satisfied, then the structure is proposed as a candidate for the set of operations. It should be noted that each such candidate might represent several instances of a more general structure. For each candidate, the process of matching identifies the associative retrieval technique to be used for each operation, and the set of choices for realizing each operation.

The third phase is the proposal of candidate structures that exhibit redundancy. The set of associative operations might be more efficiently realized as two (or more) subsets, each dealing with a data structure that is well-suited to its operations. The savings in time may outweigh the cost of storing more than one copy of the set of triples. One problem in this phase is combinatoric explosion of the number of candidates. Heuristics are required for selecting only plausible partitions of the set of associative operations.

The present system is working through phase two of candidate structure proposal. A detailed example of its output is contained in Appendix II. The immediate next steps in the research are to experiment with methods for phase three, and push through more examples. The task of finishing the code which implements the cost formulas and does final selection is straightforward, and will proceed in parallel.

Summary

Our system demonstrates the feasibility of the automatic selection of high-level data structures such as the sets, lists, and triples of SAIL. In combination, the techniques of analysis and information-gathering have been successfully applied [Low74] to programs that use sets and lists. Our preliminary analysis of the problem of selecting representations for triples shows that there is much structure in a store of triples and in the associative operations of a program. Our current research is an attempt to understand how this structure should be used with the other techniques to select associative data structures.

APPENDIX I

Brief Description of SAIL

SAIL is an ALGOL-60 based artificial intelligence language. The abstract data structures of SAIL [Feldman69, VanLehn73] include ITEMS, SETS of items, LISTS (sequences) of items, and TRIPLES (associations) of items.

An ITEM is essentially a reference to a variable allocated from a heap. Items are normally used to represent abstract objects and the names of binary relations.

A SET is an unordered collection of distinct items. A set variable is declared in the same way as any arithmetic variable. Set expressions include the union of sets, intersection of sets and explicit sets (e.g. {a, b, c} where "a", "b", and "c" are items).

A LIST is an ordered collection of items. An item may appear in a list more than once. List operations include concatenation, sublist extraction, and subscripting.

The TRIPLE is used for general mappings between items. Triples consist of three items, termed (in order) "Attribute", "Object" and "Value". The first item, the Attribute, is often used as the name of a binary relation between an Object and a set of Values. The store of triples and the operations on it are fully symmetrical, however. The language imposes no constraints on the interpretation of the position of an item in a triple. The syntax for triple forms is (<item>.<item>=item). There are three kinds of operations on triples in SAIL:

1. MAKE, which adds a given triple to the store of triples.

2. ERASE, which removes specified triples from the store of triples. Each of the three arguments to the erase operator is either an item, or the key-word "ANY". The arguments to erase are treated as a pattern, which is matched against the store of triples. Where "ANY" is used, any item will match. Triples which match are removed from the store of triples.

3. SEARCH, which locates specified triples in the store of triples. As for erase, the three arguments to search are treated as a pattern which is matched against the store of triples. In addition to items and the key-word "ANY", an argument to search can be a "bound variable". If the arguments are all items or "ANY", then search behaves like a Boolean-valued function, returning true if the pattern matches, false otherwise. If there are any arguments which are "bound variables", then search behaves like a "generator", enumerating the triples that are found to match the pattern. As each triple is found, the "bound variables" in the pattern will be assigned the corresponding items of the matching triple.

Selection of Associative Data Structures: Examples

This appendix contains several glimpses of detail from the experimental selection system. It has five parts:

A. A description of the data structures which are considered by the system.

B. Time and space formulas for one of them.

C. Examples of questions which the system asks the programmer.

D. Partial results of the system's analysis of an example SAIL program.

E. The example SAIL program listing.

A. The Associative Data Structures Known to the System

The candidate associative data structures consist of variations and combinations of four basic types: records, property lists, inverted files, and hash tables. From the many possibilities, we have chosen a representative collection of structures to analyze. These are listed and described below. In the discussion, the three positions of the triple form are identified with particular roles in the various data structures. This is done for expository reasons. The system recognizes symmetries and permutes positions of the triple form when appropriate.

1. Field Selection Records (FSR)

A field selection record is a block of contiguous storage cells. The address of the block is obtainable from one of the three items of the triple (the Object). Another of the items (the Attribute) determines a storage cell within the record and a field of this cell. The third item of the triple (the Value) is stored in the specified field. FSR's are convenient for associative searches in which two of the three items are given, and the Attribute is from a fixed set of items. FSR's are usually wasteful of memory space (compared to hash tables or inverted files) unless there is at least one Value for most Attributes of most Objects most of the time.

There are three types of FSR's:

a. FSRBIT: One bit is sufficient to represent the value.

b. FSR1VAL: The binary relation is single-valued.

c. FSRSET: There can be more than one value.

2. Property List

This data structure associates a set of (Attribute-Value) pairs with each Object. The address of a list of elements is obtained from the Object's record. Each element contains an Attribute and an ordered list of Values. The list of elements is ordered by Attribute.

Property lists are superior to hash tables where space is at a premium and the number of triples varies widely with time. They are superior to FSR's when space is at a premium and most Attributes do not have values most of the time.

3. Inverted Files

An inverted file is a list of triples. Each triple is represented as a block of storage cells which contains the three items of the triple. The list threads all blocks that have a particular item in a given position. There are two types of inverted files:

a. Simple: A one-way list of triples for one position, perhaps ordered by the items in another position.

b. Complex: multiply-threaded triples. Each triple block is an element in several lists. Each list can be ordered, and either one-way or two-way.

Inverted files are convenient for associative searches in which only one item is given. They are also useful for stores of triples that vary widely in size and in place of FSR's for cases where the set of Attributes is not fixed before program execution. They are wasteful of processing time for associative searches in which more than one item is given.

4. Hash Tables

A hash table is a block of contiguous storage cells. A function which maps triples to cell addresses within this block is used both to insert new triples and for associative retrieval. Each cell contains a pointer to the list of triples which map to the cell. Associative retrieval via hash tables requires a combination of address computation and searching. Care is required to match the design of the function to the properties of the set of triples, to avoid large discrepancies in the length of lists. Hash tables are convenient for associative searches in which more than one item is given, or the set of Attributes is not fixed and computation time is at a premium. Hash tables are not convenient when space is at a premium, and the size of the store of triples varies widely as the program runs.

There are two classes of hash tables:

a. All three items are hash operands.

   There are four types of data structure here:

   1) SIMPLE: each conflict list element (CLE) contains the three items of the triple.

   2) POINTERS TO TRIPLE BLOCKS: each CLE contains a pointer to a triple block. This allows sharing with inverted file structures.

   3) SINGLE LINK: each CLE contains a pointer to a triple block and is threaded in a one-way inverted file. This allows more intimate sharing.

4) MULTIPLE TWO-WAY LINKS: each CLE contains a pointer to a triple block and is threaded in one or more two-way inverted files.

b. Two items are hash operands.

There are twelve types of data structure here:

1) SIMPLE: each CLE contains the two hash operands (Attribute and Object) and a pointer to a list of Values.

2) POINTERS TO TRIPLE BLOCKS: each CLE contains a pointer to a list of pointers to triple blocks. This replaces the pointer to a list of Values. As for 1. above, each CLE represents a particular Attribute-Object pair, and each entry of the list represents one triple having that pair.

3) SINGLE LINK: each CLE contains the two hash operands and is threaded in a one-way inverted file. The key for the inverted file must be one of the hash operands. Each CLE contains a pointer to a list of Values.

4) SINGLE LINK WITH POINTERS TO TRIPLE BLOCKS: combination of 2 and 3 above.

5) MULTIPLE TWO-WAY LINKS: each CLE contains the two hash operands and is threaded in one or more two-way inverted files. The key(s) for the inverted file(s) must be from the two hash operands. Each CLE contains a pointer to a list of Values.

6) MULTIPLE LINKS WITH POINTERS TO TRIPLE BLOCKS: combination of 2 and 5 above.

The remaining six types of data structure (7 through 12) are variations of 1 to 6 above. Each entry on either the list of Values or the list of pointers to triple blocks is threaded in a one-way inverted file. The key for this inverted file is the Value.

B. Example of Time and Space Formulas: Field Selection Records

1. SPACE (in 36-bit memory cells) = $(NA*NO)*(\frac{1}{2}+X)$

where

NA = the number of Attributes
NO = the number of Objects
X = the average number of Values for a given Attribute and Object.

2. TIME

MAKE: $C_0+C_1+Q*\frac{X*(C_0+C_4)}{2} + M*C_2$

ERASE

(3 items given) $C_0+C_1+R*\frac{X*(C_0+C_4)}{2} + E*C_3$

(Value = ANY) $C_0+C_1+DELCOST(X)$

FIND

(3 items given) $C_0+ C_1+Z*\frac{X*(C_0+C_4)}{2}$

(Value = ANY) C_0+C_1

(Value a variable) $C_0+C_1+V*GENCOST(X)$

where

C_0 = time to compare two pointers

C_1 = time to select a given field of a given Object record, and pick up its contents

C_2 = time to insert an element at a given position in a list

C_3 = time to remove a given element of a list

C_4 = time to pick up the pointer to the next element of a list and jump

M = the fraction of MAKE operations which create new triples

Q = the fraction of MAKE operations that find a similar triple (A·O=ANY)

R = the fraction of ERASE operations that find a similar triple (A·O=ANY)

E = the fraction of ERASE operations that find a triple to erase

Z = the fraction of fully specified FIND operations that find a similar triple

V = the fraction of FIND operations for (A·O=variable) that find answers

DELCOST(X) = the time to reclaim a list of length X

GENCOST(X) = the time to generate elements from a list of length X

C. Example Questions Asked by the System About the Store of Triples

1. For a given Attribute and Object, what is the average size of the set of Values?

2. For a given Object, how many Attributes (on average) will have at least one Value?

3. What fraction of (A·O=V) questions would find answers if they were (A·O=ANY) questions?

4. How many triples (on average) will have a given value?

5. What fraction of (A·O=ANY) questions have answers (on average)?

6. What fraction of MAKE operation executions would create a triple that already exists?

7. What fraction of MAKE operation executions would find a triple which has the same Attribute and Object?

8. What fraction of ERASE operation executions find something to erase?

D. Example Analysis

This section contains partial results of the system's analysis of an example SAIL program. These consist of a collection of candidate representations for subsequent cost analysis and final selection. After a brief introduction to the example program, the candidate representations are described. Each candidate is listed with a reference to the description of its prototype in section A of this appendix.

The attached test program constructs the minimal spanning tree for a given graph and prints out information about its "cost". The algorithm deals with disjoint sets of nodes, and selects edges with the smallest cost which connect nodes

from different sets. Each time an edge is found, the two sets are merged. The result is a set of edges which form the minimum cost tree which spans the graph. A set of nodes is represented by a (single valued) binary relation of the following form:

(SETOF ·· NODE = SET)

The primitive associative operations of the program are:

1. MAKE (three given items);
2. ERASE (given item · ANY= given item);
3. SEARCH (given item · given item = variable);
4. SEARCH (given item · variable = given item).

The system finds that most of the associative retrieval techniques are applicable to each of the "erase" and "search" operations, but discovers only six candidate representations to propose for the operations taken together:

1. Threaded Triple Blocks (A3b)

Each triple is represented as a block of storage cells having two threads. Each thread is associated with a position in the triple, and is part of a list of triples that have the indicated item in the indicated position. The first thread is in either the Attribute position or the Object position, and is used by operation 3. The second thread is in either the Attribute or the Value position, and is shared by operations 2 and 4. The first thread is two-way to expedite the removal of triples which are to be erased. Both threads are ordered. In this example, the first thread represents either a list of all triples, ordered by nodes, or the one triple which identifies the set containing a given node. The second thread represents either a list of all triples, ordered by sets, or a list of triples which identify nodes belonging to a given set.

2. Object-Threaded Hash Table Entries (A4b7)

In this case, operations 2 and 4 share a hash table to find a list of Objects, given the Attribute and Value. In the example, this is a list of nodes which are in a given set. Each Object is threaded in a list of hash-table entries for the Object. Operation 3 uses this list to find the set for a given node.

3. Value-Threaded Hash Table Entries (A4b7)

This is a variation on candidate 2. Operation 3 uses the hash table to find a list of Values, given the Attribute and Object. In the example, this list would have one entry: the set which contains a given node. Operations 2 and 4 share the thread through Values. The thread represents a list of nodes for a given set.

4. Attribute-Threaded Hash Table Entries (A4b3)

This is similar to candidate 3, except that operations 2 and 4 share a thread through Attributes. In the example, this thread represents a list of all triples. For non-single valued relations, each element of the thread would represent the collection of triples that have a particular Attribute and a particular Object.

5. Hash Table and Threaded Triple Blocks(A4b2)

In this case, operations 2 and 4 share a hash table to find a list of Objects, given the Attribute and Value. As for candidate 2, this is a list of nodes which are in a given set. The difference is that hash table entries are pointers to triple blocks, which are threaded either in the Attribute position or the Object position. Operation 3 uses this thread. The thread is two-way, to expedite erases by operation 2.

6. Attribute Threaded Hash Table and Threaded Triple Blocks (A4b4)

This candidate provides a hash table for operation 3, an Attribute thread for operation 2, and a two-way triple block thread for operation 4. We would expect the preliminary cost analysis to reject it in favor of other candidates.

As the system analyzes its model, it asks questions of the user when it needs to do so. For this example, it asked two questions:

1. For a given Attribute and Object, is there only one Value?

2. For a given Attribute and Value, is there only one Object?

If operation 3 were not present in the example, the system would have proposed a candidate representation when provides a list of nodes for each set. Each Value would be represented by a record having a fixed field for the SETOF Attribute. The field would contain a pointer to a list of NODE items. Operations 2 and 4 would use the given Attribute to select this field of the given Value. If only operation 3 were present, a similar candidate representation would associate a given node with its set. One of the next steps in the research will be to consider multiple (redundant) representations as candidates. In our example, the system will then be able to consider the combination of the above two representations as a single candidate. One difficulty here is dealing with the erase operation. A general technique is to convert erase operations to loops with two operations: a search with variables in place of "ANY"s, and an erase with all items specified.

E. Example SAIL Program Listing: Minimal Spanning Tree Construction Algorithm

```
BEGIN "SPNTRE"

  REQUIRE 100 NEW!ITEMS;
  LIST EDGES; COMMENT THE PRIORITY QUEUE OF EDGES OF THE GRAPH;
  LIST ITEM A,B,C,D,E,F,G,H; COMMENT NODES OF HOMEMADE GRAPH;
  STRING ITEM NAMEA, NAMEB, NAMEC, NAMED, NAMEE, NAMEF, NAMEG, NAMEH; COMMENT NAMES OF NOD
  LIST ITEM EDGEAB, EDGEAC, EDGEAH, EDGECE, EDGEBC, EDGEBH, EDGEEF, EDGEGH, EDGEEH, EDGEBE
               EDGEDF, EDGEBG, EDGEFG, EDGECD, EDGEEG;

  ITEM SETOF; COMMENT SET MEMBERSHIP RELATION:  (SETOF . ELT = SET);

  COMMENT EDGE COSTS;
  INTEGER ITEM EcAB,EcAC,EcAH,EcCE,EcBC,EcBH,EcEF,EcGH,EcEH,EcBE,EcDF,EcBG,EcFG,EcCD,EcEG;
  SET SETOFVERTICES; COMMENT THE SET OF THE NODES OF GRAPH;
  SET TREESET; COMMENT SET OF EDGES MAKING UP MINIMAL SPANNING TREE;
  LIST ITEMVAR EDGETEMP; COMMENT WILL REFER TO AN EDGE ITEM;
  LIST ITEMVAR V,W,VERTEX; COMMENT WILL REFER TO VERTEX ITEMS;
  INTEGER ITEMVAR Ec; COMMENT WILL REFER TO COST OF AN EDGE;
  INTEGER COSTS; COMMENT COST SO FAR OF SPANNING TREE;
  INTEGER NVERTEXSETS; COMMENT NUMBER OF DISJOINT SETS OF NODES;

BOOLEAN PROCEDURE DISJOINTUNION(ITEMVAR MEMBER1,MEMBER2);
BEGIN "DISJOINTUNION"
    ITEMVAR SETNAME1, SETNAME2, TEMP1;
    IF NOT (SETOF . MEMBER1 = BIND SETNAME1) AND (SETOF . MEMBER2 = BIND SETNAME2) THEN
               ERROR;
    IF SETNAME1 = SETNAME2 THEN RETURN(FALSE);
    FOREACH TEMP1 | (SETOF . TEMP1 = SETNAME1) DO
               MAKE (SETOF . TEMP1 = SETNAME2);
    ERASE (SETOF . ANY = SETNAME1);
    DELETE(SETNAME1);
    RETURN(TRUE);
END "DISJOINTUNION";
```

```
COMMENT START EXECUTION HERE;
    COSTS := 0;
    TREESET := PHI;

COMMENT HOMEMADE GRAPH;

    DATUM(A) := {{ NAMEA }};
    DATUM(NAMEA) := "A";

    DATUM(B) := {{ NAMEB }};
    DATUM(NAMEB) := "B";
    DATUM(C) := {{ NAMEC }};
    DATUM(NAMEC) := "C";

    DATUM(D) := {{ NAMED }};
    DATUM(NAMED) := "D";
    DATUM(E) := {{ NAMEE }};
    DATUM(NAMEE) := "E";
    DATUM(F) := {{ NAMEF }};
    DATUM(NAMEF) := "F";
    DATUM(G) := {{ NAMEG }};
    DATUM(NAMEG) := "G";
    DATUM(H) := {{ NAMEH }};
    DATUM(NAMEH) := "H";

    SETOFVERTICES := {A,B,C,D,E,F,G,H};
```

```
COMMENT EDGES IS AN ORDERED LIST OF EDGES, ORDERED BY COST;

    EDGES := {{ EDGEAB, EDGEAC, EDGEAH, EDGECE, EDGEBC, EDGEBH, EDGEEF, EDGEGH,
                EDGEEH, EDGEBE, EDGEDF, EDGEBG, EDGEFG, EDGECD, EDGEEG }};
    DATUM(EDGEAB) := {{ A, B, EcAB }};
    DATUM(EDGEAC) := {{ A, C, EcAC }};
    DATUM(EDGEAH) := {{ A, H, EcAH }};
    DATUM(EDGECE) := {{ C, E, EcCE }};
    DATUM(EDGEBC) := {{ B, C, EcBC }};
    DATUM(EDGEBH) := {{ B, H, EcBH }};
    DATUM(EDGEEF) := {{ E, F, EcEF }};
    DATUM(EDGEGH) := {{ G, H, EcGH }};
    DATUM(EDGEEH) := {{ E, H, EcEH }};
    DATUM(EDGEBE) := {{ B, E, EcBE }};
    DATUM(EDGEDF) := {{ D, F, EcDF }};
    DATUM(EDGEBG) := {{ B, G, EcBG }};
    DATUM(EDGEFG) := {{ F, G, EcFG }};
    DATUM(EDGECD) := {{ C, D, EcCD }};
    DATUM(EDGEEG) := {{ E, G, EcEG }};
    DATUM(EcAB) := 1;
    DATUM(EcAC) := 1;
    DATUM(EcAH) := 1;
    DATUM(EcCE) := 1;
    DATUM(EcBC) := 2;
    DATUM(EcBH) := 2;
    DATUM(EcEF) := 2;
    DATUM(EcGH) := 2;
    DATUM(EcEH) := 3;
    DATUM(EcBE) := 3;
    DATUM(EcDF) := 3;
    DATUM(EcBG) := 4;
    DATUM(EcFG) := 6;
    DATUM(EcCD) := 8;
    DATUM(EcEG) := 9;
```

```
COMMENT INITIALIZE SET OF DISJOINT SETS AND THE MAPPING BETWEEN A NODE AND THE
    DISJOINT SET IN WHICH IT APPEARS:

    FOREACH VERTEX | VERTEX IN SETOFVERTICES DO
                MAKE (SETOF · VERTEX = NEW);
    NVERTEXSETS:=LENGTH(SETOFVERTICES);

COMMENT NOW CONSTRUCT THE SPANNING TREE:

    WHILE NVERTEXSETS > 1 DO
        BEGIN
            EDGETEMP := LOP(EDGES);
            V := DATUM(EDGETEMP) [1];
            W := DATUM(EDGETEMP) [2];
            Ec := DATUM(EDGETEMP) [3];
           IF DISJOINTUNION(V,W) THEN
               BEGIN
                   COSTS := COSTS + DATUM(Ec);
                   PUT EDGETEMP IN TREESET;
                   NVERTEXSETS:=NVERTEXSETS-1;
               END;
        END;

COMMENT PRINT OUT THE SET OF EDGES OF THE MINIMAL SPANNING TREE;

    OUTSTR(CRLF& "EDGES AND COSTS OF EDGES");
    FOREACH EDGETEMP SUCH THAT EDGETEMP IN TREESET DO
        BEGIN
            STRING ITEMVAR NODENAME1, NODENAME2;
            V := DATUM(EDGETEMP) [1]; W := DATUM(EDGETEMP) [2]; Ec := DATUM(EDGETEMP) [3]
            NODENAME1 := DATUM(V) [1];  NODENAME2 := DATUM(W) [1];
            OUTSTR(CRLF& DATUM(NODENAME1)&DATUM(NODENAME2) & TAB & CVS(DATUM(Ec)));
        END;
    OUTSTR(CRLF& "TOTAL COST OF SPANNING TREE =" & CVS(COSTS));
END "SPNTRE"
```

REFERENCES

[BAUMGART72] B. Baumgart. Micro Planner Alternate Reference Manual. Stanford Artificial Intelligence Laboratory. Operating Note 67, Apr. 1972.

[EARLEY71a] J. Earley. Comments on SETL (Symmetric Use of Relations). SETL Newsletter 52. Courant Institute NYU. September 1971.

[EARLEY71b] J. Earley. Toward an Understanding of Data Structures. CACM, Vol. 14, 10, October 1971.

[EARLEY73] J. Earley. An Overview of the VERS2 Project. Electronic Research Laboratory, College of Engineering memorandum ERL-M416, December 1973, University of California at Berkeley.

[EARLEY74] J. Earley. High Level Iterators and a Method of Automatically Designing Data Structure Representation. Electronic Research Laboratory, College of Engineering memorandum ERL-M425, February 1974, University of California at Berkeley.

[FELDMAN69] J. Feldman and P. Rovner. An Algol-Based Associative Language. CACM, Vol. 12, 8, August 1969.

[LOW74] J. Low. Automatic Coding: Choice of Data Structures. Technical Report #1, Computer Science Dept., University of Rochester.

[MCDERMOTT72] D. McDermott and G. Sussman. The Conniver Reference Manual. AI Memo No. 259, M.I.T., May 1972.

[MORRIS73] J. Morris. A Comparison of MADCAP and SETL. University of California, Los Alamos Scientific Laboratory, 1973.

[ROVNER76] P. Rovner. Automatic Selection of Associative Data Structures. Ph.D. thesis, Dept. of Mathematics, Harvard University (in preparation).

[SCHWARTZ75a] J. Schwartz. Automatic Data Structure Choice in a Language of Very High Level. Second Symposium on Principles of Programming Languages. Palo Alto, California, January 1975.

[SCHWARTZ75b] J. Schwartz. Optimization of Very High Level Languages--I. Value Transmission and its Corollaries. In Computer Languages, Vol. 1, pp. 161-194, Pergamon Press, 1975.

[SUSSMAN70] G. Sussman, T. Winograd, and E. Charniak. MICRO-PLANNER Reference Manual. AI Memo 203. Project MAC, M.I.T., July 1970.

[SUSSMAN72] G. Sussman. Why Conniving is Better than Planning. AI Memo 255. M.I.T. Artificial Intelligence Laboratory, February 1972.

[VANLEHN73] K. VanLehn. SAIL User Manual. Stanford Computer Science Technical Report STAN-CS73-373, July 1973.