NASA Contractor Report 4539

A Path-Oriented Matrix-Based Knowledge Representation System

Stefan Feyock and Stamos T. Karamouzis
The College of William and Mary
Williamsburg, Virginia

Prepared for
Langley Research Center
under Cooperative Agreement NCC1-159

(NASA-CR-4539) A PATH-ORIENTED MATRIX-BASED KNOWLEDGE REPRESENTATION SYSTEM (College of William and Mary) 26 p

N94-13065

Unclas

H1/61 0185512

National Aeronautics and
Space Administration
Office of Management
Scientific and Technical
Information Program
1993

https://ntrs.nasa.gov/search.jsp?R=19940008592 2020-06-16T21:45:39+00:00Z

A Path-oriented Matrix-based Knowledge Representation System

Stefan Feyock and Stamos T. Karamouzis

The College of William and Mary

Abstract

Experience has shown that designing a good representation is often the key to turning hard problems into simple ones. Most AI search/representation techniques are oriented toward an infinite domain of objects and arbitrary relations among them. In reality much of what needs to be represented in AI can be expressed using a finite domain and unary or binary predicates. Well-known vector- and matrix-based representations can efficiently represent finite domains and unary/binary predicates, and allow effective extraction of path information by generalized transitive closure/path matrix computations. In order to avoid space limitations a set of abstract sparse matrix data types was developed along with a set of operations on them. This representation forms the basis of an intelligent information system for representing and manipulating relational data.

Keywords: Information System, Intelligent, Representation, Matrices, Paths

• Introduction

A *representation* is a set of syntactic and semantic conventions that make it possible to describe things. Experience has shown that designing a good representation is often the key to turning hard problems into simple ones. According to [Winston 1984] good representations:

- Make important things explicit
- Expose natural constraints, facilitating some class of computations
- Are complete and concise
- Facilitate computation. We can store and retrieve information rapidly.
- Suppress detail.
- Are computable by an existing procedure.

All representations must provide some way to denote objects and to describe the relations that hold among them. Consequently, many representations are built around some form of *semantic net*, since semantic nets denote objects and describe relations among them.

Most AI search/representation techniques are oriented toward a potentially infinite domain of objects and arbitrary relations among them. Experience has shown that in reality much of what needs to be represented in AI can be expressed using a finite domain and unary or binary predicates. Unary predicates can describe object attributes and binary predicates describe relations among two objects.

Well-known vector- and matrix-based representations are appropriate for finite domains and unary/binary predicates, since they satisfy the above-mentioned properties of a "good" representation, and allow the extraction of path information by generalized transitive closure/path matrix computations. In this scheme vectors are used for unary relations and matrices for binary relations. Unfortunately as the number of objects increases the size of matrices rapidly surpasses the amount of available memory in most machines.

Overcoming memory limitations raises the need for abstract data types to represent sparse matrices. These are well suited for most applications, since semantic nets usually represent a limited number of connections among objects, even when working in large domains.

• Matrices and Semantic Nets

A directed graph (digraph) is a 2-tuple <N, E>, where N is a finite set of nodes, and E a finite set of edges. An edge is a member <a, b> of N×N. A labeled digraph is a 3-tuple <N, E, L>, where N is as before, L is a finite set of labels, and E is a finite set of labeled edges, with labels in L. A labeled edge (with label in L) <a, l, b> is a member of N×L×N.

It is easy to see that digraphs are a graphic representation of binary predicates over finite domains. If P(x, y) is a predicate over domain D×D, then digraph G = <N, E> represents P if P(a, b) iff <a, b> ∈ E.

Whereas an unlabeled digraph can represent a single predicate, labeled digraphs whose label set is a set of predicate names can represent multiple binary predicates over the same domain D×D simply by letting edge <a, p, b> denote the fact that predicate p(a, b) is true; the absence of such an edge denotes that p(a, b) is false. Extending the notation, we allow edges to be labeled with *sets* of predicate names; an edge <a, $\{p_1, ..., p_n\}$, b> is an abbreviation for the set of edges <a, p_1, b>, ..., <a, p_n, b>. Labeled digraphs thus correspond to the familiar semantic net construct of AI.

Given the problem of representing a unary predicate P(x) over a finite domain D of fixed size n, an obvious and familiar solution is to use boolean vectors, a.k.a. bit strips: for any d_i in D, $P(d_i)$ is true (false) iff the i'th component of the vector representing P is a 1 (0). Boolean operations such as AND, OR, and NOT on predicates over D are then representable by the corresponding operations over bit strips, which are efficient on most computers. Similarly, binary predicates Q(x, y) over D×D can be efficiently represented by N×N matrices whose ij element is 1 if $Q(d_i, d_j)$ is true, else 0.

Boolean matrices can in principle represent labeled digraphs: a separate matrix is assigned to each label, and represents the subgraph of nodes connected by edges bearing that label. In practice this representation can become unwieldy. The number of different labels may be large, resulting in proliferation of adjacency matrices. Moreover, queries such as "is there any path (regardless of labels) from node a to node b?" require that the matrices for all labels be ORed together. An answer to the follow-up query "what are these paths?" is even more difficult to generate from this representation. Such considerations motivate the adoption of symbolic matrices as representation for labeled digraphs. Element ij of a symbolic matrix is P iff the arrow from d_i to d_j in the semantic net has label P, else NIL.

• Implementation

LIMAP (LIsp-based MAtrix Processor) is a set of Common LISP [Steele 1984] procedures that define and manipulate a vector/matrix-based knowledge representation. The user may represent relations among objects using boolean or symbolic matrices/vectors. These matrices/vectors are abstract data types that can represent data (boolean values or symbols) stored in arrays. The semantic interpretation of this data is left to the user.

As is the case for a traditional database system, LIMAP's capabilities are invoked via a language interface that consists of two parts. One is the data definition language (DDL) for specifying both the data the system is to contain as well as "metadata;" i.e. information about the structure and constraints that govern the data contained in the system. The other is the data manipulation language (DML), the subset of the language concerned with the specification of queries and updates on the data. We will categorize the LIMAP functions accordingly. As in Common LISP, LIMAP's functions and arguments are not case sensitive.

DDL Operations

Figure 1 shows LIMAP's data definition procedures, and their associated syntax.

DEFREL <relname> <specs> <type> <rep>
DELREL <relname>

<relname> ::= <symbolic atom>
<type> ::= Bmatrix | Smatrix | Bvector | Svector
<specs> ::= (<length> <length>) | (<length>)
<length> ::= <symbolic atom> | <integer>
<rep> ::= vector-rep | array-rep | sparse-rep

Figure 1

DEFREL

The function *defrel* defines a relationship with name *relname* of type *type* and having particular specifications. The actual data of the relation is stored in a system-generated variable and is represented according to the *rep* field. Valid representations are array, vector, and sparse representations. The representation is transparent to the user since s/he views all relationships according to their type attribute. Valid *type* attributes are boolean matrices/vectors (Bmatrix/Bvector) and symbolic matrices/vectors (Smatrix/Svector). The *specs* field specifies the dimensions of the matrices/vectors. Matrices are two-dimensional and vectors one-dimensional. When assigning the dimensions of a relation the system expects a list with one or two numbers or symbols. For a matrix definition the first number specifies the number of rows and the second the number of columns. If a symbol is specified the system expects that the symbol is the name of a set of values and substitutes the size of the set for the symbol. Following the definition a change in the size of the set does not affect the dimensionality of the matrix/vector. Change of dimensionality is achieved via the RESIZE function, as is explained later. The following example defines a matrix named "example_mtrx" to be of boolean type, have 4x4 elements represented as a list, a vector named "is_sensor" to be of type boolean and have size of nine elements, and a matrix named "engine" to be a symbolic matrix of size 9x9 and be represented as an array.

```
(setvar '*comps* '(fan compressor combustor turb1 turb2 N1 N2 EGT EPR))
(defrel 'example_mtrx '(4 4) 'Bmatrix 'sparse-rep)
(defrel 'is_sensor '(*comps*) 'Bvector 'vector-rep)
(defrel 'engine '(9 *comps*) 'Smatrix 'array-rep)
```

Matrices/vectors of boolean type are matrices/vectors where each of the elements is either a "0" or "1",

representing false or true respectively. The elements of matrices/vectors which are declared as *symbolic* can contain arbitrary s-expressions such as numbers, symbols, or lists. When a matrix is declared as having *array-rep* representation the matrix is associated with a Common LISP two-dimensional array, and every operation on the matrix (such as a retrieval, multiplication etc) is performed on the two-dimensional array. A vector with a *vector-rep* representation is represented as a one-dimensional array. Matrices/vectors having a *sparse-rep* representation are represented as LISP lists. For example, if example_mtrx has only two elements, say a value 1 in row 10, column 30 and in row 33 column 90, then it is represented by the list ((10 30 1) (33 90 1)).

When a relation is defined using *defrel* it is placed in a system-maintained definition table which maintains information about all the defined relations. Additionally, a system-generated variable is bound to the data structure that will actually hold the data. This data structure is a list initialized to nil if the representation is *sparse-rep*. When the representation is *array-rep* the data structure is an array initialized with zeros or nil, depending on the type field. Matrices/vectors of boolean type are initialized to all zeros, and symbolic matrices/vectors to all nils. Figure 2 shows the contents of the definition table after the above definitions. The *path* and *flag* fields are explained in a later section during the description of the *paths* operation.

NAME	SPECS	TYPE	REP	PATH	FLAG
example_mtrx	(4 4)	Bmatrix	sparse-rep	P0	T
is_sensor	(*comps*)	Bvector	vector-rep	None	None
engine	(9 *comps*)	Smatrix	array-rep	P1	T

Figure 2

DELREL

The operation *Delrel* deletes a relation by extracting it from the definition table and disassociating all data variables with the relation.

DML Operations

Using the data manipulation language (DML) procedures the user may query a relation's type, specifications, representation, or the actual data stored. S/he may store/retrieve values to particular locations, multiply two matrices, copy one matrix to another, invert, transpose, resize, clear, or take the transitive closure of a matrix. Figure 3 tabulates the DML operations.

Predicate	Arg1	Arg2	Arg3	Arg4	Arg5	Description
DEFTABLE	[:ALL]	[T]				Display definition table
DISPLAY	relname					Display a relation
TYPE	relname					Relation's type
DIMS	relname					Relation's dimensions
REP	relname					Relation's representation
DATA	relname					Relation's data
DATA-NAME	relname					Relation's data variable
STORE	relname	value	[row]	column		Store value
RETRIEVE	relname	[row]	column			Retrieve contents
TCLOSE	relname					Transitive closure
PATHS	relname	row	column	[:NAME]	[T]	All paths
MULT	relname1	relname2	relname3			Multiply
TRANSPOSE	relname1	relname2				Transpose
CLEAR	relname					Initializes a relation
COMPLEMENT	relname1	relname2				Relation's complement
RESIZE	relname	specs				Changes the dimensions
COPYREL	relname1	relname2				Copy rel1 into rel2
COLUMN	column	relname	vectname			Copy a column into a vecto
ROW	row	relname	vectname			Copy a row into a vector
RELAND	relname1	relname2	relname3			Logical AND
RELOR	relname1	relname2	relname3			Logical OR

Figure 3

STORE and RETRIEVE

The function STORE allows the user to store a specific value to a particular location in some matrix/vector. The user must specify the matrix/vector name, the value to be stored and the coordinates of the location in row-major order. The function returns the stored value. An error is returned when there is a type mismatch between the value to be stored and the type of the matrix/vector. RETRIEVE retrieves

the contents of a particular location in a matrix/vector. In the event that that the specified matrix/vector in not defined, both functions display an appropriate message and return nil.

Example: (STORE 'engine 'N 0 5) stores the symbol "N" in location (0 5) of the symbolic matrix "engine", and returns "N".

TYPE and REP

The functions TYPE and REP notify the user about the type and representation scheme of a particular matrix. The possible return values of *Type* are Bmatrix, Bvector, Smatrix and Svector. REP returns either *array-rep, vector-rep* or *sparse-rep* depending on the specified matrix/vector.

Example: (TYPE 'engine) and (REP 'engine) return *Smatrix* and *array-rep* respectively.

DIMS, DATA and DATA-NAME

The function DIMS returns a list of the dimensions of a specified matrix/vector. This list contains one number, the number of elements, for a vector and two numbers, the number of rows and the number of columns, when the specified structure is a matrix. Functions DATA and DATA-NAME, respectively, return the data structure and the symbolic name of the data structure that holds the actual data of the specified abstracted matrix/vector. All of the above functions terminate gracefully with an appropriate message when the specified matrix/vector is not defined.

Example: (DIMS 'engine) returns *(9 9)* and (DIMS 'engin) returns nil and displays the message

"*** From LIMAP, *engin* is not defined".

(DATA-NAME 'engine) returns "engine", while (DATA 'engine) returns

```
#2A((nil  N  nil nil  N  N  nil nil nil)
    (nil nil  N  N  nil nil  N  nil nil)
    (nil nil nil  N  nil nil nil nil nil)
    (nil  N  nil nil  N  nil nil nil nil)
    ( N  nil nil nil nil nil nil  N  N )
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
```

MULT

MULT is a function that allows the user to multiply two matrices, a matrix and a vector or two vectors of the same type. The resulting matrix/vector is placed in a user-specified matrix/vector, which constitutes the third argument in the function. If the specified matrices/vectors are not defined, have incompatible types, or incompatible dimensions the function terminates gracefully by displaying appropriate error messages. MULT operates on the following principle

For boolean matrices/vectors such as b1 an m×n, and b2 an n×r (MULT 'b1 'b2 'b3):

$$b3[i,j] = \bigvee_{h=1}^{n} f(b1[i,h],b2[h,i])$$

where f(x,y) = 1 if both x,y are 1, else 0

For symbolic matrices/vectors such as s1 an m×n, and s2 an n×r (MULT 's1 's2 's3):

$$s3[i,j] = \bigvee_{h=1}^{n} f(s1[i,h],s2[h,i])$$

where f(x,y) = t if both x,y are non-nil, else nil

Examples: The following shows the contents of *example_mtrx* before and after the operation (MULT 'example_mtrx 'example_mtrx 'example_mtrx).

Before:	**After:**
1 1 1 1	1 1 1 1
1 . . .	1 1 1 1
1 . . .	1 1 1 1
. . . .	. . . .

Assuming that S1, S2, and S3 are 4 × 4 symbolic matrices, the following shows the contents of S1, S2, and S3 after the operation (MULT 'S1 'S2 'S3).

S1:	**S2:**	**S3:**
a nil nil nil	b b b b	t t t t
a nil nil nil	nil nil nil nil	t t t t
a nil nil nil	nil nil nil nil	t t t t
a nil nil nil	nil nil nil nil	t t t t

TRANSPOSE and COMPLEMENT

The functions TRANSPOSE and COMPLEMENT respectively transpose and complement a specified matrix/vector. TRANSPOSE works only on matrices, and COMPLEMENT inverts zeroes to ones and vice-versa on boolean matrices/vectors. The resulting complemented matrix/vector replaces the specified matrix/vector, but the result of the transposition is placed in a new matrix specified by the user. Successful termination of the above functions returns true.

Examples: The following shows the contents of *example_mtrx* before and after the operation (TRANSPOSE 'example_mtrx 'example_mtrx).

Before:	**After:**
1 . . .	1 1 1 .
1 . . .	. . . .
1 . . .	. . . .
. . . .	. . . .

The contents of *example_mtrx* before and after the operation (COMPLEMENT 'example_mtrx 'example_mtrx) is as follows:

Before:	**After:**
1 . . .	. 1 1 1
1 . . .	. 1 1 1
1 . . .	. 1 1 1
. . . .	1 1 1 1

CLEAR and COPYREL

CLEAR initializes the contents of a specified abstracted matrix/vector. Matrices/vectors of a boolean type are initialized to all zeroes and matrices/vectors of symbolic type to all nils. COPYREL copies one matrix/vector to another. Both arguments must be of the same type and have the same representation. The first argument is the source and the second the destination.

RESIZE

The function RESIZE changes the dimensions of a specified matrix/vector. The first argument is the specified matrix/vector and the second a list containing the new dimensions. After a RESIZE operation that increases the size of the matrix/vector the matrix/vector retains its elements and the newly created locations are initialized with the default values. The newly created locations are appended at the ends of vectors, and the right and bottom margins of matrices. A RESIZE operation that decreases dimension

sizes drops higher indexed elements. Thus following a RESIZE operation where the new dimensions are smaller than the previous if the users tries to access locations that don't exist the operation returns nil and prints an out of range error message. **Example:** If the contents of vector *is_sensor* (defined previously as a boolean vector of size 9) is:

Location: 0 1 2 3 4 5 6 7 8
Contents: 0 0 0 0 0 1 1 1 1

then after the operation (RESIZE 'is_sensor '(11)) the contents of the same vector will be:

Location: 0 1 2 3 4 5 6 7 8 9 10
Contents: 0 0 0 0 0 1 1 1 1 0 0

RELAND and RELOR

The logical functions RELAND and RELOR perform the logical *AND* and *OR* among two vectors/matrices. These vectors/matrices must be of a boolean type and have the same size. The result of either function becomes the contents of the third argument.

Example: The following shows the result of (RELAND 'example_mtrx 'other_mtrx 'result_mtrx), where the contens of *example_mtrx* and *other_mtrx* is:

example_mtrx:	**other_mtrx:**	**result_mtrx:**
1 . . .	. . . .	. . . .
1 . . .	1 . . .	1 . . .
1 . . .	1 . . .	1 . . .
. . . .	1 . . .	. . . .

When *example_mtrx* and *other_mtrx* have the same contents as above, then following the operation (RELOR 'example_mtrx 'other_mtrx 'result_mtrx) the contents of *result_mtrx* is:

example_mtrx:	**other_mtrx:**	**result_mtrx:**
1 . . .	. . . .	1 . . .
1 . . .	1 . . .	1 . . .
1 . . .	1 . . .	1 . . .
. . . .	1 . . .	1 . . .

DISPLAY, SHOW-ARRAY, and DEFTABLE

LIMAP provides the user with functions that allow him/her to view the contents of matrices/vectors. DISPLAY produces a formatted display of a matrix or a vector. In case the abstracted data types are of symbolic type an "S" is displayed at the location that a symbol exists. In order to see the actual symbols SHOW-ARRAY should be used. The function DEFTABLE displays the contents of the definition table, which contains all the defined matrices/vectors and their associated attributes.

Example: The outputs of (DISPLAY 'enginea) and (SHOW-ARRAY 'enginea) are as follows:

(DISPLAY 'enginea)

```
0:  .  S  .  .  S  S  .  .  .
1:  .  .  S  S  .  .  S  .  .
2:  .  .  .  S  .  .  .  .  .
3:  .  S  .  .  S  .  .  .  .
4:  S  .  .  .  .  .  .  S  S
5:  .  .  .  .  .  .  .  .  .
6:  .  .  .  .  .  .  .  .  .
7:  .  .  .  .  .  .  .  .  .
8:  .  .  .  .  .  .  .  .  .
```

(SHOW-ARRAY 'enginea)

```
nil  1  nil  nil  1  1  nil  nil  nil
nil  nil  1  1  nil  nil  1  nil  nil
nil  nil  nil  1  nil  nil  nil  nil  nil
nil  1  nil  nil  1  nil  nil  nil  nil
 1  nil  nil  nil  nil  nil  nil  1  1
nil  nil  nil  nil  nil  nil  nil  nil  nil
nil  nil  nil  nil  nil  nil  nil  nil  nil
nil  nil  nil  nil  nil  nil  nil  nil  nil
nil  nil  nil  nil  nil  nil  nil  nil  nil
```

Path Operations: TCLOSE and PATHS

The TCLOSE and PATHS operations form the core of LIMAP's path manipulation capability. The function TCLOSE calculates the transitive closure of a specific matrix. The transitive closure of a matrix M is a matrix M* that contains an entry in location <a, b> iff the directed graph represented by M contains a path (of length 0 or greater) from a to b. In LIMAP M* inherits M's type and representation attributes. Warshall's Algorithm is an efficient method for computing M*, given a matrix M. Intuitively, the algorithm scans the matrix top to bottom, left to right. If an entry is encountered, say in row i, column j, then row i is replaced by row i OR row j, and the scan continues from position ij. Figure 4 shows the code that performs the transitive closure for boolean and symbolic matrices using Warshall's Algorithm.

```
(DEFUN BTclose (rel)                          ; Function to compute transitive
  (LET ((max (FIRST (dims rel)))              ;  closure of a boolean matrix
    (DO ((k 0 (+ k 1))) ((= k max) nil)       ; Scan top to bottom
      (DO ((i 0 (+ i 1))) ((= i max) nil)     ; Scan left to right
        (COND ((= (retrieve  rel i k) 1)      ; If there is an entry
            (DO ((j 0 (+ j 1))) ((= j max) nil)
              (store rel (LOGIOR (retrieve rel i j)   ; Swap i and j
                                 (retrieve rel k j)) i j))
            )                                 ; Close DO
          )                                   ; Close COND
        )                                     ; Close DO
      )                                       ; Close DO
    )                                         ; Close LET
)                                             ; Close BTclose


(DEFUN STclose (rel)                          ; Function to compute transitive
  (LET ((max (FIRST (dims rel)))              ;  closure of a symbolic matrix
    (DO ((k 0 (+ k 1))) ((= k max) nil)       ; Scan top to bottom
      (DO ((i 0 (+ i 1))) ((= i max) nil)     ; Scan left to right
        (COND ((NOT (NULL (retrieve  rel i k)))   ; If there is an entry
            (DO ((j 0 (+ j 1))) ((= j max) nil)
              (COND ((NOT (NULL (OR
                                 (retrieve rel i j)   ; If there is a symbol in (i, j)
                                 (retrieve rel k j))))   ; OR in (k, j)
                                 (store rel t i j)))     ; Then flag that (i,j) are connected
            )                                 ; Close DO
          )                                   ; Close COND
        )                                     ; Close DO
      )                                       ; Close DO
    )                                         ; Close LET
)                                             ; Close STclose
```

Figure 4

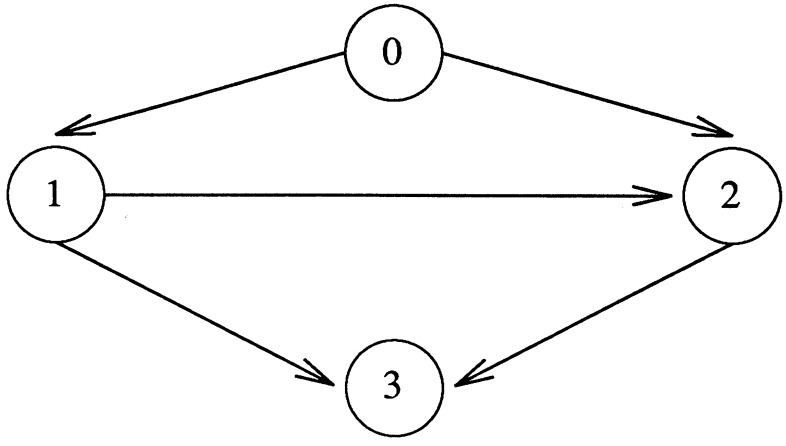


Figure 5

Example: Let us assume that we have a network of four nodes labeled as 0, 1, 2, and 3. Assume that there are direct connections from 0 to 1, 0 to 2, 1 to 2, 2 to 3, and 1 to 3 as indicated in Figure 5. The following displays how the network may be represented in a boolean matrix along with the contents of the same matrix after the transitive closure has been computed.

example_mtrx:

```
   0 1 2 3
   -------
0: . 1 1 .
1: . . 1 1
2: . . . 1
3: . . . .
```

example_mtrx,
after (TCLOSE 'example_mtrx):

```
   0 1 2 3
   -------
0: . 1 1 1
1: . . 1 1
2: . . . 1
3: . . . .
```

A value of 1 in locations (0, 1), (0, 2), (1, 2), (1, 3), and (2, 3) means that there is a direct connection between the corresponding nodes. Following the operation of transitive closure a value of 1 in location (*i, j*) means that there is a connection from node *i* to node *j*. This connection may be direct or indirect. An example of indirect connection is the connection between node 0 and 3. Such a connection is achieved via node 1 or node 2 (Figure 5).

Assuming that LIMAP's matrices (boolean or symbolic) represent directed graphs, the function PATHS returns all the paths between two specific nodes in a network. Besides the user-defined attributes that characterize each matrix in LIMAP, every matrix is associated with an internal system matrix called the *path matrix*, and a *flag* field. When PATHS is invoked for the first time on a particular matrix it does the following. First, using an extension of Warshall's Algorithm, all possible paths among every node in the matrix are calculated. A path is a list of node numbers. The resulting lists of paths become the entries of the associated *path matrix* and the *flag* is set to *nil*. At the end only the paths among the two specific nodes specified by the user are returned. A subsequent request for paths need not recalculate all the paths, but merely retrieve the appropriate entry from the *path matrix*. In case that there is a change in the contents of the user defined matrix (i.e. a change in the graph) the flag field is set to "t" and a subsequent user request for paths triggers a recalculation of the *path matrix*.

In order to operate on a symbolic matrix and produce a *path matrix* whose ij entry contains the set of all paths from node i to node j, Warshall's algorithm was extended as Figure 6 indicates.

Let M be an NxN square matrix

```
for k=1 to N  ; Scan from top down
  for i=1 to N  ; Scan from left to right
    if ( i ≠ k AND M[i,k] ≠ nil ) then
      for j=1 to N
        M[i,j] := UNION ( M[i,j], LINK( M[i,k], M[k,j] ))
```

Figure 6

In the algorithm of Figure 6, UNION is the normal union operation on sets and LINK operates on lists of paths. If p, q are paths where $p = (v_1, \ldots, v_k)$ and $q = (v_k, \ldots, v_r)$ then LINK (p,q) returns $(v_1, \ldots, v_k, \ldots, v_r)$.

More precisely: let E(k) denote the set of all paths going through nodes numbered ≤ k only (not including the endpoints, which can be arbitrary). Then the original adjacency matrix represents E(0). More precisely, the original matrix has the list ((i j)) in element ij if there is an arrow from i to j, otherwise nil. (This discussion assumes vertices numbered from 1, although Common Lisp dimensions are actually indexed from 0; in that case, the original matrix would represent E(-1).) Warshall's Algorithm scans the matrix from top to bottom, left to right. The scan of the k'th column computes E(k), as follows. When a non-nil element is encountered in an off-diagonal position in column k, say at ik, that element will be a list of E(k-1)-paths from node i to node k. Consider arbitrary element ij of row i; it contains all E(k-1)-paths (if any) from i to j. Now that we also know the E(k-1)-paths from i to k, we can reach j from i either by the E(k-1) paths in ij, or by going from node i to node k, and then from node k to node j by any path (if any) in element kj. Such paths will, of course, be E(k) paths. Thus we add to the paths already at ij the link of all paths from i to k and all paths from k to j.

Example: Assume that *symbolic_mtrx* is a 4 × 4 symbolic matrix that represents the network of Figure 5. If the contents of *symbolic_mtrx* is as follows, then (PATHS 'example_mtrx 0 1) returns "(0 1)", and (PATHS 'symbolic_mtrx 0 3) returns "((0 1 3) (0 2 3) (0 1 2 3))". The contents of *symbolic_mtrx* and the internally maintained path matrix are:

symbolic_mtrx:

	0	1	2	3
0:	nil	1	1	nil
1:	nil	nil	1	1
2:	nil	nil	nil	1
3:	nil	nil	nil	nil

corresponding path matrix:

	0	1	2	3
0:	nil	((0 1))	((0 2))	((0 1 3) (0 2 3) (0 1 2 3))
1:	nil	nil	((1 2))	((1 3))
2:	nil	nil	nil	((2 3))
3:	nil	nil	nil	nil

Following our exemplar definitions, Figure 7 displays the contents of the definition table including the attributes of each path matrix (compare with figure 2). This is achieved by using the optional *all* flag of the DEFTABLE function, i.e by calling (DEFTABLE :all t). In the path field of the table is stored the name of the associated path matrix. The associated path matrices have no value in the flag field.

	NAME	TYPE	SPECS	REP	PATH	FLAG
0:	example_mtrx	(4 4)	Bmatrix	sparse-rep	P0	T
1:	P0	(4 4)	Bmatrix	sparse-rep	None	None
2:	is_sensor	(*comps*)	Bvector	vector-rep	None	None
3:	engine	(9 *comps*)	Smatrix	array-rep	P1	T
4:	P1	(9 *comps*)	Smatrix	array-rep	None	None

Figure 7

Control Structures

Queries of the form "is there a relation R such that nodes a and b are in relation R? "is there a path from x to y? a path fulfilling constraint C? where can I go from x? how can I get to x?" arise frequently both in AI and elsewhere. Such queries, which involve quantification over relations, correspond to statements in the *second-order predicate calculus*. This section describes the control structures that make LIMAP an efficient second-order predicate calculus programming system.

The distinction between procedural and non-procedural predicate calculus specifications blurs if the underlying domain is finite, since the FORALL and EXISTS quantifiers map in an obvious way to loops ranging over the domain elements. It has been our goal to give the LIMAP data manipulation language as non-procedural a character as possible. In particular, LIMAP notation is an adaptation of the (function-less) predicate calculus, with extensions to allow data retrieval in addition to data specification. For example, a "yes" answer to (EXISTS X) (FORALL Y) P(X,Y) is insufficient; the actual X-value must be retrieved. We have found that minimal modifications of the control macros described in [Charniak et al., 1987] were suitable for the task of expressing the required quantifications. Following is a summary of the general form of the control structure implemented by these macros:

(FOR (<*variable*$_1$> :IN <*set*$_1$>)
. . .
(<*variable*$_n$> :IN <*set*$_n$>)
[:WHEN <*when-expression*>]
<*FOR-keyword*> <*expression*$_1$> . . . <*expression*$_n$>)

The *expression*$_i$ following *FOR-keyword* are called the *body* of the FOR. The construct (<*variable*$_i$> :IN <*set*$_i$>) causes the variable to iterate over the elements of the set, which may be specified as a list, a vector, or a matrix row or column. When there are several sets the FOR interates over the elements of each set in the following way. Initially the first element of each set is assigned to the corresponding *variable*$_i$ and the body of the FOR is evaluated. Then the second element of each set is assigned to the corresponding *variable*$_i$ and the body is evaluated again. This is repeated until some set runs out of elements or the final value of the FOR is determined as governed by the *FOR-keyword.*

FOR-keywords

:ALWAYS	return true if all the values of body are true
:FILTER	produce a list of the non-nil values of body
:FIRST	produce the first non-nil value of body
:SAVE	produce a list of all values of body

While the description of these constructs is procedural in form, the effect when programming in this notation is that of writing FORALLs and EXISTs, with the proviso that any variable values that are found to "EXIST" are collected in accordance with the FOR-keyword and returned as value. The following section contains an example application of LIMAP.

• Application

The reasoning portions of intelligent information systems are usually not specific to any domain. Therefore, enough domain information must be provided to allow them to work. FAULTFINDER [Abbott et al., 1987], conceived and developed to detect and diagnose inflight failures in an aircraft, is such a system. One of its most important subsystems of FAULTFINDER is DRAPHYS, a model-based reasoner that determines which components' malfunction best account for the observed symptoms.

DRAPHYS representation using LIMAP

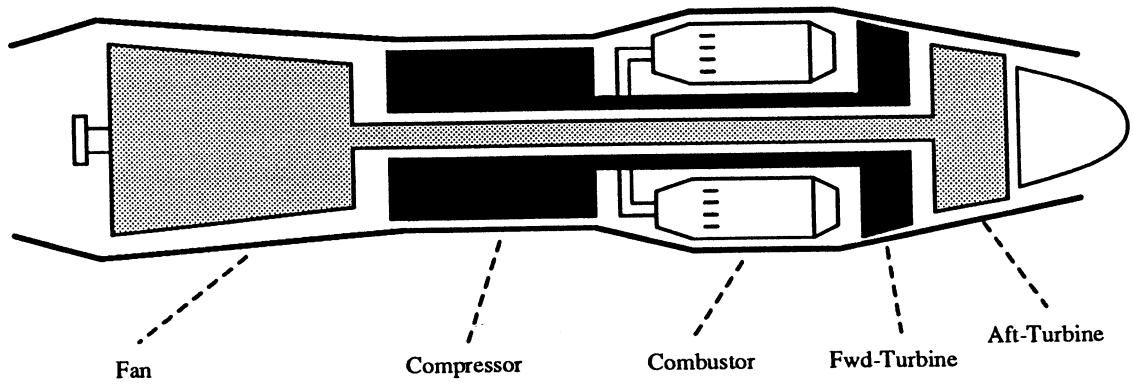
DRAPHYS uses a digraph model of an aircraft system, with nodes representing primitive components, and arrows connecting nodes representing functional and physical dependencies. Component B is said to be *functionally dependent* on component A if the proper functioning of B depends on the proper functioning of A. Component B is *physically dependent* on component A if damage to A can propagate through space to component B. For example, the control surfaces of an aircraft are functionally dependent on the hydraulic system, since they will cease operating if the latter fails. On the other hand, if a hydraulic line can be severed by a disintegrating turbine, the line is physically dependent on the turbine.

Figure 8 shows the schematic of a turbo-fan jet engine and Figure 9 the functional dependency graph for this engine. Using LIMAP the physical and functional dependencies can be represented in symbolic matrices. Figure 10 depicts the adjacency matrix representing the jet engine functional dependency predicate Engine(x,y) of Figure 9 over the domain Comps={fan, compressor, combustor, fwd-turbine, aft-turbine, N1-sensor, N2-sensor, EGT-sensor, EPR-sensor}. A value of 1 in location i,j represents the fact that component i is connected to component j. LIMAP's boolean vectors can be used to represent unary predicates. The vector Is_Sensor = < 0, 0, 0, 0, 0, 1, 1, 1, 1 >, for example, represents the Is_Sensor predicate where a 1 in location i denotes that element i is a sensor.

The large number of components in a realistic aircraft system results in corresponding large storage requirements. These may be reduced to manageable proportions by judicious use of *LIMAP's* sparse vector/matrix capability. Additionally, with the above representation many useful second-order operations can be expressed concisely and efficiently:

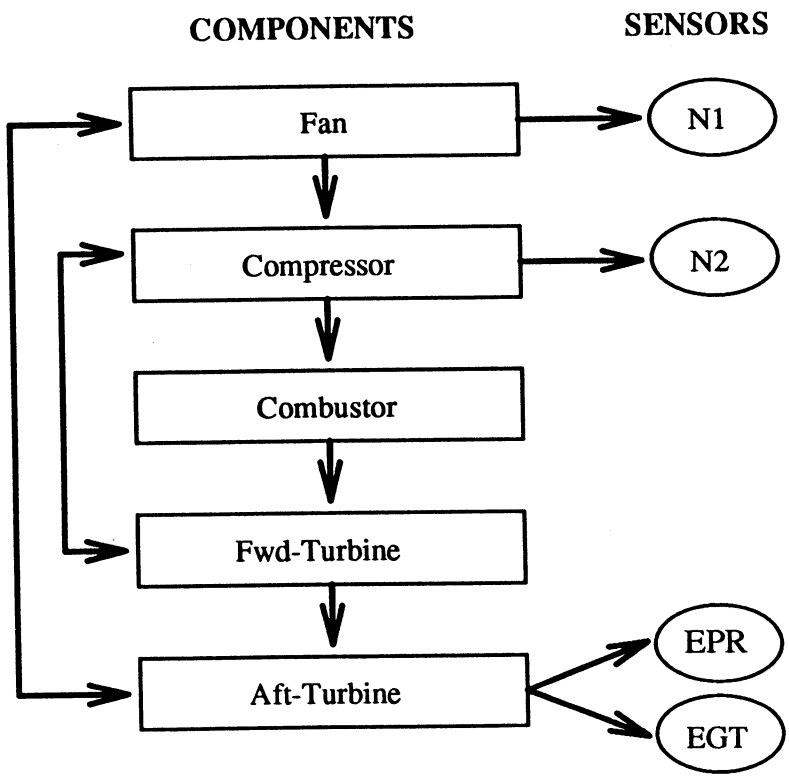
(EXISTS X) X(a,b) ?
(FORALL X) X(a,b) ?
(EXISTS P) P a path from node a to node b ?

Such queries correspond to finding parents, siblings, descendants, routes between nodes, etc. For



Dual-Compressor Turboprop

Figure 8



Functional Dependencies

Figure 9

example, we can obtain the set of instrumented components simply by performing the boolean matrix multiplication Engine × Is-Sensor. Similarly, the question of whether a path exists between, say, the fan and the EGT sensor is trivially answered by noting whether Engine*[1,8] contains a non-nil value. Additionally P1[1,8] gives all the possible routes between the fan and EGT, where P1 is the path matrix associated with the matrix Engine.

COMPONENT		(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Fan	(0)	.	1	.	.	1	1	.	.	.
Compressor	(1)	.	.	1	1	.	.	1	.	.
Combustor	(2)	.	.	.	1	.	.	.	.	.
Fwd-turbine	(3)	.	1	.	.	1	.	.	.	.
Aft-turbine	(4)	1	.	.	.	.	.	.	1	1
N1 Sensor	(5)	.	.	.	.	.	.	.	.	.
N2 Sensor	(6)	.	.	.	.	.	.	.	.	.
EGT Sensor	(7)	.	.	.	.	.	.	.	.	.
EPR Sensor	(8)	.	.	.	.	.	.	.	.	.

Figure 10

DRAPHYS in LIMAP

The following procedure represents a portion of the reasoning process performed by DRAPHYS [Abbott, 1991] when a system malfunction occurs:

- Suspect every possible component
- Use the model to determine consequences of each failure
- Eliminate suspects inconsistent with model predictions

More precisely:

```
for each C in SET-OF-PRIMITIVE COMPONENTS do
    if "C has failed" is a valid hypothesis
    then add C to SET-OF-VALID-HYPOTHESES
end for
```

With luck, SET-OF-VALID-HYPOTHESES will contain only one element. If it contains more, DRAPHYS waits for more symptoms to develop and disambiguate the diagnosis.

We must specify how it is determined that "C has failed" is a valid hypothesis (possible diagnosis). Here is an informal description of how this is done:

> Primitive component C is a valid hypothesis iff
> there is a POSSIBLE PROPAGATION PATH from C to every symptomatic sensor

A path is a POSSIBLE PROPAGATION PATH iff every instrumented component on the path has at least one symptomatic sensor

Here is a predicate calculus formulation of this stipulation:

```
valid-hypothesis(comp) <=>
    is-primitive(comp) &
    (FORALL s) {is-sensor(s) & is-symptomatic(s) =>
    (EXISTS p) [(path(p, /*from*/ comp, /*to*/ s) &
    (FORALL n) [is-node(n, /*on path*/ p) => not is-ok(n) or is-unknown(n)]]}

is-ok(n) <=>
    (FORALL s) [is-sensor(s) & instruments(s,n) => not is-symptomatic(s)]

is-unknown(n) <=>
    not (EXISTS s) [is-sensor(s) & instruments(s,n)]
```

Instruments(s,n) is true iff s is a sensor measuring some attribute of n.

Based on this description of DRAPHYS, the following is how DRAPHYS' reasoning is implemented in LIMAP. Since the notation bears strong analogies to the predicate calculus specification, we present it without further explanation.

```
; The list *comps* contains all engine components, including sensors
;
(defun determine-hypotheses (*comps* symptomatic-sensors)
    (for (c :in *comps*)
        :when (is-valid-hypothesis c)
        :filter c)
)

(defun is-valid-hypothesis (c symptomatic-sensors)
    (for (s :in symptomatic-sensors)
        :always (exists-bad-path c s) )
)

(defun exists-bad-path (c s)
    (for (p :in (paths 'engine c s) )                    ; paths from c to s
        :first (for (c :in p)
                :always (not-known-ok c) )
))

(defun not-known-ok (c)
    (or (null (instrumentation c)) (symptomatic c))
)                                                        ; symptomatic is a boolean vector

(defun instrumentation (c)  ; returns list of sensors associated with c
(for (s :in *comps*)
    :when (and (is-sensor s) (retrieve 'engine c s))
    :save s)
)
```

• Conclusion

We have described a programming system oriented toward efficient information representation/manipulation over fixed finite domains, and quantification over paths and predicates. The initial motivation for the creation of such a system was the fact that the need for such operations arose frequently in the domain of diagnosis/prognosis generation problem domain. Since then it has become

apparent that the facilities provided are applicable to problems both within and outside of AI.

Our experience to date has shown that LIMAP is applicable to a wide range of problems. While LIMAP, if abused, is as capable of inefficient operation as any other misused programming system, we have found that for every problem yet attempted there has existed a LIMAP formulation that was concise, comprehensible, and for which LIMAP's facilities constituted an efficient problem representation.

• References

- Abbott, K. *Robust Fault Diagnosis of Physical Systems in Operation.* NASA Technical Memorandum 102767, NASA Langley, Hampton, VA., 1991

- Abbott, K., Schutte, P., Palmer, M., Ricks, W. *Faultfinder: A Diagnostic System with Graceful Degradation for Onboard Aircraft Applications.* 14th International Symposium on Aircraft Integrated Monitoring Systems, Friedrichshafen, 1987

- Aho, A., Hopcroft, J. and Ullman, J. *Data Structures and Algorithms.* Addison Wesley Publishing Company, Reading, Massachusetts, 1983

- Charniak, E., Riesbeck, C. K., McDermott, D., and Meehan, J. R. *Artificial Intelligence Programming,* 2nd ed., Lawrence Earlbaum Associates, Hillsdale, N.J., 1987

- Horowitz, E. and Sahni, S. *Fundamentals of Data Structures.* Computer Science Press, Maryland, 1986

- Steele, Guy. *Common LISP: The Language.* Digital Press, 1984

- Winston, P. *Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Contractor Report

4. TITLE AND SUBTITLE
A Path-Oriented Matrix-Based Knowledge Representation System

5. FUNDING NUMBERS
NCC1-159

505-64-13-22

6. AUTHOR(S)
Stefan Feyock and Stamos T. Karamouzis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The College of William and Mary
Computer Sciences Department
Williamsburg, VA 23185

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-4539

11. SUPPLEMENTARY NOTES
Langley Technical Monitor: Paul C. Schutte

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Experience has shown that designing a good representation is often the key to turning hard problems into simple ones. Most AI search/representation techniques are oriented toward an infinite domain of objects and arbitrary relations among them. In reality much of what needs to be represented in AI can be expressed using a finite domain and unary or binary predicates. Well-known vector- and matrix-based representations can efficiently represent finite domains and unary/binary predicates, and allow effective extraction of path information by generalized transitive closure/path matrix computations. In order to avoid space limitations a set of abstract sparse matrix data types was developed along with a set of operations on them. This representation forms the basis of an intelligent information system for representing and manipulating relational data.

14. SUBJECT TERMS
Artificial intelligence; Predicate Calculus; Information System; Matrices

15. NUMBER OF PAGES
28

16. PRICE CODE
A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		