Nov. 1981

NOTE ON LEARNING IN MDS BASED ON PREDICATE SIGNATURES

by

Chitoor V. Srinivasan

DCS-TR-107

#211

Department of Computer Science
Hill Center, Busch Campus
Rutgers University, New Brunswick, N.J. 08903

In each state, s, PS(b,s) may be used to classify the constants that occur in the state as per rule,

$$(b \doteq c) \iff PS(b,s) = PS(c,s). \qquad (4)$$

The names of these classes, the cardinality of classes, and relationships between cardinalities (inequalities and equality) may all be now added to the language L to form its first extension. These classes may further be used to construct a graphical representation of the relations that occur in the states of D (as discussed in section 2.2), which may subsequently be used to identify constraints that are true in given states of D, or constraints that are true for all the states in D. The use of these constructs is illustrated in the two experiments discussed below.

This work is related to notions of "knowledge representation" in MDS (the Meta Description System): The characterization C[D] of a domain D is called the static theory of the domain [see Srinivasan,1981]. The learning method illustrated below is based on the schemes used in MDS to organize this static theory [Srinivasan,1980,1977].

2. The Experiments:

2.1. Recognition of Letters in the English Alphabet.

In the first experiment D is a finite set, the set of all letters in the English alphabet, in a fixed font. Each letter is formed by a configuration of points in a given grid of size 5x7. The snap-shot of a letter is a description of the relative positions of the various grid points using the relations from the vocabulary,

$$V1 = \{\text{north, south, east, west, northeast, northwest, southeast, southwest}\} \qquad (5)$$

The snap-shot for the letter A shown in Figure 1, is shown in figure 2. In Figure 2 each Cij is a constant, the grid point at row i and column j of the grid. Given snap-shots like these for the letters in the alphabet, in a fixed font, we ask our M to produce distinguishing descriptions of the letters, which it could use to quickly identify the letters.

[FIGURE 1 ABOUT HERE]

For the vocabulary V1 the total number of possible equivalence classes is 256. But, it seems, only about 50 or 60 will occur for any chosen alphabet. The letter A shown in Figure 1 has fifteen equivalence classes associated with it, EC001 through EC015. These are shown in Figure 2. The predicate signatures associated with the constants in Figure 1 are also shown in Figure 2.

Note that if a snap-shot has only one equivalence class associated with it then it should necessarily contain only a single constant, and its predicate signature is the null set, if none of the relations are symmetric or reflexive. For interior points, (those that are surrounded by other points in all directions) their predicate signature will be V1, and for corner and side points their predicate signatures will be subsets of V1.

1. Introduction

We report here on two short experiments on learning. The learning situation is the following: We have a domain consisting of a set of states. Each state consists of a finite set of constants (objects) in the domain. The states are viewed by a machine, M, in terms of the features (relations) which are true in the states. The features that the machine looks for are determined by the machine's vocabulary, V, which is assumed to be initially given to M:

$$V = \{r1, r2, \ldots, rn\} \qquad (1)$$

where each ri is a relation name (predicate letter) of arity ki (in a general case the vocabulary will have both predicate letters and function letters). In the discussion below we restrict ourselves to the special case where ki = 2 for all ri in V and there are no function letters. If s is a state in a domain D, then

$$S(s) = \{(r\ c\ d) \mid ((r\ c\ d) \text{ is true in } s)\} \qquad (2)$$

is called the snap-shot of the state s. A snap-shot may contain typically between a few tens to a few thousand relation instances. The set of all such snap-shots of the states in D will be the initial representation of D in our machine M, if D is finite. If D is infinite then our machine will never have direct access to all the states in D.

We give our machine the first order language L defined by the vocabulary V and the names of constants that are used in the snap-shots. Besides this we also give M an ability to extend L in a natural way using certain fixed (but general) schemes to abstract properties of snap-shots and classify the constants that occur in them. Let EL denote this extended first order language. Then a general learning problem is the following: Produce a characterizing description, C[D], of D in the language EL: i.e. for any state s consisting only of the constants in D, s belongs to D if any only if s satisfies the description C[D]. This is a hard problem. (Of course, one would want to know whether such a characterization is feasible in the context of the given fixed schemes of M. See [Srinivasan,1980] for a discussion of these issues). We believe that the methods discussed below provide us some significant tools that are generally useful to cut down the combinatorially explosive search that one usually encounters in problems of this kind. In simple cases, like the ones discussed below, the methods quickly yield the characterizations that are being sought. Predicate signatures play a central role in this.

We first introduce below in section 2 the experiments and the basic results obtained so far. Later in section 3 we briefly outline the underlying theory that make this kind of theory construction feasible. We begin with the definition of predicate signatures.

Predicate Signatures

Let us assume that for every r in the vocabulary, V, its converse r' is also in V: $(r\ x\ y) \Leftrightarrow (r'\ y\ x)$. Then the predicate signature of a constant, b, in a state, s, is,

$$PS(b,s) = \{r \mid (r \text{ is-in } V)\ \&\ ((\text{EXISTS } y)((r\ b\ y) \text{ is-true-in } s))\}. \qquad (3)$$

[GET FIGURE 2 ABOUT HERE]

One may order the set of all possible equivalence classes in some fixed order. For the ordering shown in Figure 2 the cardinality description of A is: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 2]. For fixed font characters, it so happens that the cardinality descriptions uniquely disambiguate the letters in D. At the time this happened this was an unexpected result. We now suspect that this would happen for any fixed font. A part of the reason for this is discussed in section 3. It is difficult to normalize these description with respect to the total number of points in the grid and make these descriptions independent of the size or density of points in the letters. This can be done in certain cases, but not always. However, the descriptions are translation invariant, but highly sensitive to rotation (in case of vocabulary V1, rotation by units of 45 degrees).

Another interesting characterization is one where each state is characterized simply by the set of equivalence classes that it contains. This disambiguates all the letters in the alphabet, excepting U and J. By imposing an ordering on the equivalence classes in each state, say according to increasing order of their cardinalities, one can now succeed in separating U and J. This characterization can now be scaled up or down to different grid sizes and point densities. This is thus more robust than the cardinality descriptions mentioned above. Ordering of the equivalence classes puts a rough measure on the relative density of points in different regions of a letter. A circle for example will have equal (or nearly equal) cardinalities for all its equivalence classes, and the letter A will have a greater density in the northern region, etc. There is considerable room here for further experimentation. We are now continuing with this work.

When fonts are mixed then there will be more than one version of a letter in each class. If the number of fonts is small (say 3 or 4) one could use disjunctions of normalized cardinality descriptions, or ordered equivalence classes, to characterize the classes. This is because the characterization of a letter in one font is quite unlikely to be the same as the characterization of another letter in another font. But this approach is not quite satisfactory. We would like to get more natural descriptions of the letters that are in an intuitive sense more intrinsic to the letters themselves. For example we would like to see the letter A as a closed loop with two tails hanging down from either side. Characterizations like these will be robust with respect to font changes. These can be done by representing the snap-shots as graphs called dependency graphs (discussed below) and analyzing the graphs. These experiments have not yet been completed. The use of dependency graphs occurs naturally in the context of the second experiment, presented below.

2.2. Recognition of Implicit Relational Dependencies.

Our snap-shots here are databases of family relationships. The problem we consider here is the one that was first proposed by Lindsay [Lindsay,1963] and solved by exhaustive search by [Brown,1973]. The objective is to identify relational equalities of the following kind that are implicit in the facts provided by the database:

$$r1.r2...rk = t1.t2...tn, \qquad (6)$$

where r1, r2, ..., rk and t1, t2, ..., tn are relation names in the vocabulary

for the domain (in our case, family relationships). The equality is interpreted as,

$$((x)\,(y)\,(x\ r1.r2...rk\ y) \Longleftrightarrow (x\ t1.t2...tn\ y)), \tag{7}$$

where (x r.t y) is true iff there exists a z such that (x r z) and (z r y) are both true. This is the usual relation composition operation. Brown identified equalities of this kind by constructing pairs

$$\begin{array}{l}\{(x\ y) \mid (x\ r1.r2....rk\ y)\}\ \text{and}\\ \{(u\ v) \mid (u\ t1.t2...tn\ v)\}\end{array} \tag{8}$$

and comparing the resultant sets. These are expensive operations. We shall present here a method for identifying equalities of this kind that is based on the use of predicate signatures and graphical representations of snap-shots. The method does not use constructions of sets of the kind shown in (8) and implicit relational equalities of the kind shown in (6) are identified with very little computational overhead.

Connection Graphs and Dependency Graphs

For a constant b in a snap-shot, s, its connection tree, C-tree(b,s), has b as its root node (b is at level 0), and its nodes at level (i+1) are recursively defined as

$$\text{LEVEL } (i+1) \text{ nodes} = \{c \mid ((\text{EXISTS } d)\,(\text{level } d\ i)\ \&\ ((r\ d\ c)\ \text{is-true-in } s))\}, \tag{9}$$

where the level of a node is determined by the first appearance of the node in the tree. If a node, d, whose level is i appears later at another level j, j>i, then the latter appearance of d is blocked, i.e. no further growth of the tree is spawned from this second appearance of d and the level of d is continued to be reckoned as being i. Every node at level (i+1) is then connected to its parent node at level i by an arc with its corresponding relation name, r, as its label: i.e. if the arc with label r connects node c to node d then the relation instance "(r c d) will be true in the snap-shot. Of course, it is possible that there are several arcs connecting a node in the tree with its parent node. See example in Figure 3. Since our snap-shots are always finite the C-tree(b,s) will also be finite. The level acquired by a node (constant) is recorded with the node in the tree.

[FIGURE 3 ABOUT HERE]

The connection graph, C-graph(b,s) is now obtained from C-tree(b,s) by superimposing all the blocked nodes back on their respective earlier occurrences. The root of the C-tree(b,s), namely, b, is called the anchor of the C-graph(b,s). See example in Figure 4.

[FIGURE 4 ABOUT HERE]

The dependency graph, D-graph([b,B],s), (B is the class to which b belongs) is now obtained from the C-graph(b,s) by replacing all the node labels (namely the constants) by their respective equivalence classes (the ones that are defined by the predicate signatures). The anchor node of the C-graph(b,s) will

become the anchor of the D-graph([b,B],s) with the node label changed to [b,B]. See example in Figure 5.

[FIGURE 5 ABOUT HERE]

We are here using only the positive relations to construct the dependency graphs. This need not be generally true. In general, the snap-shot may also contain negations of relation instances. In such a case relation names and their negations will both appear as possible labels for arcs in a dependency graph. We assume throughout that for each relation name r in the vocabulary, its converse is also in the vocabulary.

These D-graphs, one for each constant in a snap-shot, may now be themselves classified into equivalence classes: Two D-graphs are equivalent to each other if they are isomorphic (it may be noted that D-graphs are anchored graphs whose arcs and nodes are both labelled. By ordering the relations that occur in a D-graph in a linear order and by choosing the appropriate list representation for the D-graphs, the isomorphism test for D-graphs can be reduced to an equality test of their corresponding list representations).

Note the following properties of the D-Graphs:

a). If an implicit relation of the form shown in (6) is true, then for every pair of constants b and c, that satisfy (7) the following loops will occur in D-graph([b,B],s) and D-graph([c,C],s), respectively:

LOOP IN D-GRAPH([b,B],s):
([b,B] r1.r2...rk.t'1.t'2...t'n [b,B]),

where each t'i is the converse relation name of ti, and

LOOP IN D-GRAPH([c,C],s):
([c,C] t'1.t'2...t'n.r1.r2...rk [c,C]).

Notice that only the loops that begin and end at anchors need be considered. The following is also true:

b). If every D-graph that contains the path r1.r2...rk starting from its anchor also contains the path t1.t2...tn starting from the anchor and the end nodes of r1.r2...rk and t1.t2...tn coincide, then the identity (6) is true.

The search for loops of this kind in D-graphs can be organized very efficiently using the level labels associated with the nodes in a D-graph. At every level greater than 0, to search for loops one has to only look for arcs that go from the current node, say c, at level i, to another node, say d, whose level is i or less. Every time such an arc is found the potential for a loop exists. To confirm the loop all one has to test is that there is a path from c to the anchor node that does not intersect any of the nodes that were traversed along the path from the root to the node c. See illustration in Figure 6.

[FIGURE 6 ABOUT HERE].

Given a D-graph, using the above algorithm, one can quickly enumerate all the loops in the D-graph that include the anchor node of the graph. It may be

noticed that one need test only the distinct D-graphs (those that are not isomorphic). The construction of D-graphs themselves can be computationally expensive for a large database. But once constructed, as discussed in the next section, they can be used to identify (in principle) all the constraints that hold true in the states of a domain.

A significant aspect of search using the D-graphs is that it is not required that each snap-shot be complete (i.e. contain every time all the relations that are true for each constant). All that is needed is that the set of snap-shots that have been examined up to a certain stage contain among them cases that are relevant to the identification of possible relational identities. During search a system could postulate candidate identities and seek for counterexamples to refute them, and if not refutable then can propose them as identities in the domain.

The experimental system called KBLS (Knowledge Based Learning System) that was used to conduct the above described experiments was implemented by Donna Nagel, and Richard Keller as a class project. Bernard Nudel also contributed to certain parts of the implementation. We have not yet conducted experiments with incomplete snap-shots. We intend to do these after reimplementing the current KBLS to get better code and better algorithms.

Some of the properties of predicate signatures and dependency graphs that make them useful in ones search for constraints in a domain, are presented in the next section.

3. Equivalence Classes and Dependency Graphs.

Predicate signatures and equivalence classes of dependency graphs represent two of the extremes of classifications of constants in a domain for a given vocabulary and extended language EL. The most detailed partitioning of the constants in a domain is done by the types associated with the constants in a given first order language. The TYPE of a constant, c, is the set of all formulas, P(x), in EL with exactly one free variable, x, for which P(c) is true (see [Shoenfield,1967]):

$$TYPE(c) = \{P(x) \mid P(c)\}. \qquad (10)$$

Thus, if TYPE(c) = TYPE(d) in a domain for the given language EL then the constants c and d are indistinguishable from each other in EL, i.e. every property that is expressible in EL and which is satisfied by c is also satisfied by d. The only way c and d can be distinguished is by their names. Thus the type of a constant, c, may be interpreted as the set of constants that are indistinguishable from c in the language EL. The dependency graphs are related to the TYPE of a constant by the following:

The set of all equivalence classes of dependency graphs with [c,C] as their anchor (let us denote this by GRAPH-EQCLASS[c,C]) characterizes the type of c in the following sense, if each extension in EL is definable in L:

$$(TYPE(c) = TYPE(d)) \iff (GRAPH\text{-}EQCLASS[c,C] = GRAPH\text{-}EQCLASS[d,D]).$$

Thus the D-graphs carry with them the most detailed available information that one can possibly get from the language EL about the states in D. However, whereas TYPE(c) can be a very large set, GRAPH-EQCLASS[c,C] need not necessarily be large. Thus, if one knew how to analyze the dependency graphs in GRAPH-EQCLASS[c,C] one can extract from them all the constraints associated with c. Therefore, dependency graphs, if they are of manageable size, are useful entities to have around if one is interested in the constraint identification problem.

The dependency graphs however do not necessarily reduce the search problem involved in identifying the constraints that are relevant to the static theory C[D] mentioned in section 1. However, for specific classes of constraints they provide a good organization to conduct the search. For relational identities like (6), and for identifying the relational properties like symmetry, transitivity and reflexivity, the graphs are decidedly useful. They are also useful, to identify constraints of the form:

$$(x\ r\ y) \Rightarrow P(x,y),$$

(where P(x,y) is an expression in EL with two free variables) provided one has some understanding for the structure of P(x,y), like for example that P(x,y) is a conjunction of literals, or that it is a disjunction of conjunctions with each conjunction containing at most 3 literals, etc. We are now studying the problem of identifying constraints of this kind.

The predicate signatures provide a classification at the other extreme in the following sense: Clearly, if there exists an s in D for which PS(c,s) is not equal to PS(d,s) then TYPE(c) is not equal to TYPE(d). Thus two constants that belong to different equivalence classes defined by the predicate signatures cannot belong to the same type. In this sense the equivalence class partitioning of the constants in a domain is a safe classification to make when very little is known about the domain: We are guaranteed that if things are identical (i.e. their types are equal) then predicate signature partitioning will never put them apart.

Classifications based on predicate signatures have also the following additional properties that make them very useful in pattern recognition problems of the kind we considered in the first experiment above. Suppose c1, c2, ..., cn are the constants that occur in a state s, and S(s) is the snap-shot of s. Let S(s,c1) be the transitive closure of the set of relation instances in S(s) in which the constant c1 occurs. If for every pair of constants in s, S(s,c1) is not equal to S(s,c2) then the vocabulary that is used to produce the snap-shot is said to be a <u>prime vocabulary</u>. This says that the features of the state, s, that were extracted by the vocabulary are sufficiently detailed to distinguish every object that occurs in the state from every other object that also occurs in the state. For two dimensional scenes the vocabulary {north, sourth, east, west} is a prime vocabulary. Thus the vocabulary V1 in (5) is also a prime vocabulary. Thus even a single misplaced grid point in a letter will produce a different equivalence class partitioning. This is a part of the reason why one is able to use disjunctions of cardinality descriptions to classify letters in mixed font alphabets. The other part is, of course, the characters are indeed essentially different as two dimensional objects.

I do not yet completely understand the full significance and possible uses for predicate signatures. In frame based systems like MDS, KRL [Bobrow,1977], FRL [Goldstein,1977] and others, the predicate signatures in fact define the basic units of frames. Thus they are useful to organize domain knowledge. In the above experiments we have shown that they are also useful to organize raw data in ones search for the domain knowledge. Much further experimentation remains to be done.

4. Acknowledgements.

I am thankful to David Sandford for carefully reading through an earlier version of this paper and suggesting revisions and corrections. My discussions with David on the concept of predicate signatures (this name was suggested by David), their use in learning and organization of domain knowledge were very helpful in formulating the above ideas. David also suggested the alphabet recognition problem as a good candidate domain to test what a system can learn using the predicate signatures.

5. References

[Bobrow,1977] Bobrow, D. G., Kaplan, R. M., Kay, M., Norman, D. A., Thompson, H., and Winograd, T., "GUS, A Frame-Driven Dialog System", AI Jl., 8,2, pp 155-173.

[Brown,1973] Brown, J. S., "Steps toward Automatic Theory Formation", IJCAI-3, pp 121-129, 1973.

[Goldstein,1977] Goldstein, I. P. and Roberts, R. B., "NUDGE, a Knowledge-Based Scheduling Program", IJCAI-5, MIT, Cambridge, Mass, pp. 257-263.

[Lindsay,1963] Lindsay, R. K., "Inferential Memory as the Basis of Machines which Understand Natural Language", Computers and Thought, Feigenbaum and Feldman, Eds., McGraw Hill, New York, pp.217-233, 1963.

[Shoenfield,1967] Shoenfield, J. R., Mathematical Logic, Addison-Wesley, Reading, Mass., 1967.

[Srinivasan,1981] Srinivasan, C. V., "Knowledge Representation and Problem Solving in MDS", Companion paper submitted to IJCAI 1981.

[Srinivasan,1980] Srinivasan, C. V., "Knowledge Based Learning Systems, Part II: Introduction to the Meta-Theory, and Part III: Logical Foundations", Department of Computer Science Technical report, DCS-TR-89, Rutgers University, Hill Center, Busch Campus, New Brunswick, N.J. 08903.

[Srinivasan,1977] Srinivasan,C. V., "The Meta Description System: A System to Generate Intelligent Information System, Part I: The Model Space," SOSAP-TR-20A, Department of Computer Science, Hill Center, Busch Campus, Rutgers University, New Brunswick, N.J. 08903.

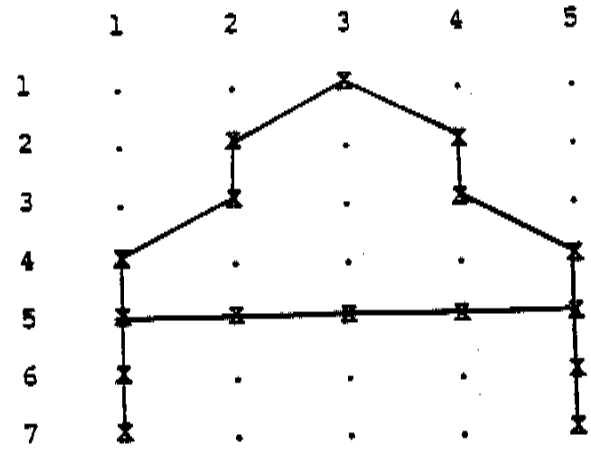

Figure 1: The letter A on the 5x7 grid.

SNAP-SHOT FOR A:

```
[(SOUTH C75  C65)
 (SOUTH C71  C61)
 (SOUTHEAST C65  C54)
 (SOUTH C65  C55)
 (NORTH C65  C75)
 (EAST C54  C53)
 (SOUTHWEST C54  C45)
 (WEST C54  C55)
 (NORTHWEST C54  C65)
 (EAST C53  C52)
 (WEST C53  C54)
 (SOUTH C61 C51)
 (SOUTHWEST C61 C52)
 (NORTH C61 C71)
 (SOUTH C55 C45)
 (EAST C55 C54)
 (NORTH C55 C65)
 (SOUTHEAST C52 C41)
 (EAST C52 C51)
 (WEST C52 C53)
 (NORTHEAST C52 C61)
 (SOUTH C51 C41)
 (WEST C51 C52)
 (NORTH C51 C61)
 (SOUTHWEST C41 C32)
 (NORTH C41 C51)
 (NORTHWEST C41 C52)
 (SOUTHEAST C45 C34)
 (NORTH C45 C55)
 (NORTHEAST C45 C54)
 (SOUTH C34 C24)
 (NORTHWEST C34 C45)
 (SOUTH C32 C22)
 (NORTHEAST C32 C41)
 (SOUTHWEST C22 C13)
 (NORTH C22 C32)
 (SOUTHEAST C24 C13)
 (NORTH C24 C34)
 (NORTHWEST C13 C24)
 (NORTHEAST C13 C22)]
```

CONSTANTS IN A:

```
[C13 C24 C22 C32 C34 C45 C41 C51
 C52 C55 C61 C53 C54 C65 C71 C75]
```

EQUIVALENCE CLASSES IN A:

```
[EC001 EC002 EC003 EC004 EC005
 EC006 EC007 EC008 EC009 EC010
 EC011 EC012 EC013 EC014 EC015 EC015]
```

PREDICATE SIGNATURES OF CONSTANTS IN A:

```
[((C13) (NORTHWEST NORTHEAST))
 ((C24) (SOUTHEAST NORTH))
 ((C22) (SOUTHWEST NORTH))
 ((C32) (SOUTH NORTHEAST))
 ((C34) (SOUTH NORTHWEST))
 ((C45) (SOUTHEAST NORTH NORTHEAST))
 ((C41) (SOUTHWEST NORTH NORTHWEST))
 ((C51) (SOUTH WEST NORTH))
 ((C52) (SOUTHEAST EAST WEST NORTHEAST))
 ((C55) (SOUTH EAST NORTH))
 ((C61) (SOUTH SOUTHWEST NORTH))
 ((C53) (EAST WEST))
 ((C54) (EAST SOUTHWEST WEST NORTHWEST))
 ((C65) (SOUTHEAST SOUTH NORTH))
 ((C71 C75) (SOUTH))]
```

THE EQUIVALENCE CLASSES IN A:

```
[(EC001 (NORTHWEST NORTHEAST) (C31))
 (EC002 (SOUTHEAST NORTH) (C24))
 (EC003 (SOUTHWEST NORTH) (C22))
 (EC004 (SOUTH NORTHEAST) (C32))
 (EC005 (SOUTH NORTHWEST) (C34))
 (EC006 (SOUTHEAST NORTH NORTHEAST) (C45))
 (EC007 (SOUTHWEST NORTH NORTHWEST) (C41))
 (EC008 (SOUTH WEST NORTH) (C51))
 (EC009 (SOUTHEAST EAST WEST NORTHEAST) (C5
 (EC010 (SOUTH EAST NORTH) (C55))
 (EC011 (SOUTH SOUTHWEST NORTH) (C61))
 (EC012 (EAST WEST) (C53))
 (EC013 (EAST SOUTHWEST WEST NORTHWEST) (C5
 (EC014 (SOUTHEAST SOUTH NORTH) (C65))
 (EC015 (SOUTH) (C71 C75))]
```

CARDINALITY DESCRIPTION OF A:

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 2]
```

Figure 2: The snap-shot for the letter A and its associated preciate signatures and equivalence classes.

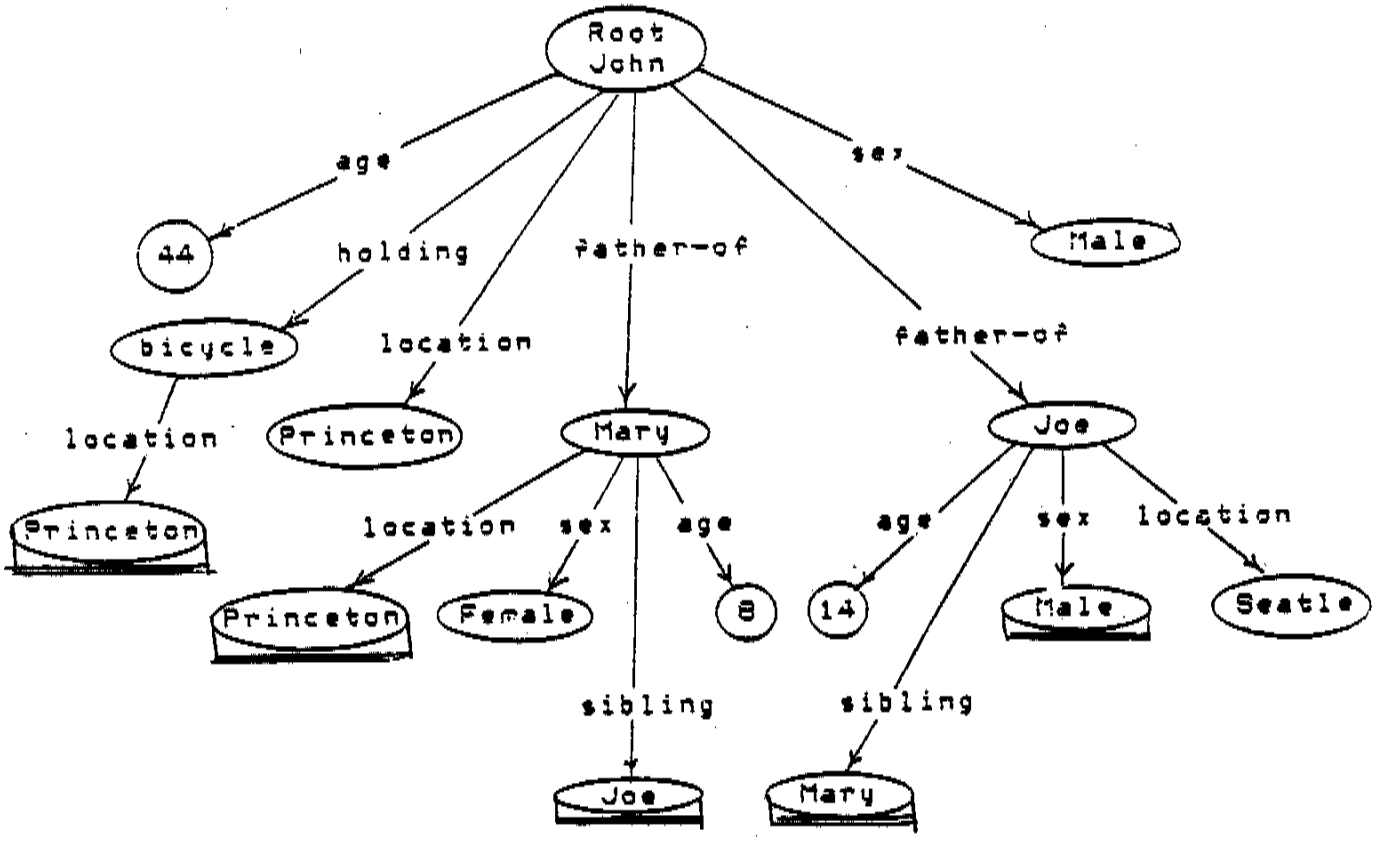

Figure 3: The Connection tree, C-Tree(John, s).
To maintain clarity not all the arcs between nodes are shown. Arcs corresponding to the converse relations are not shown.

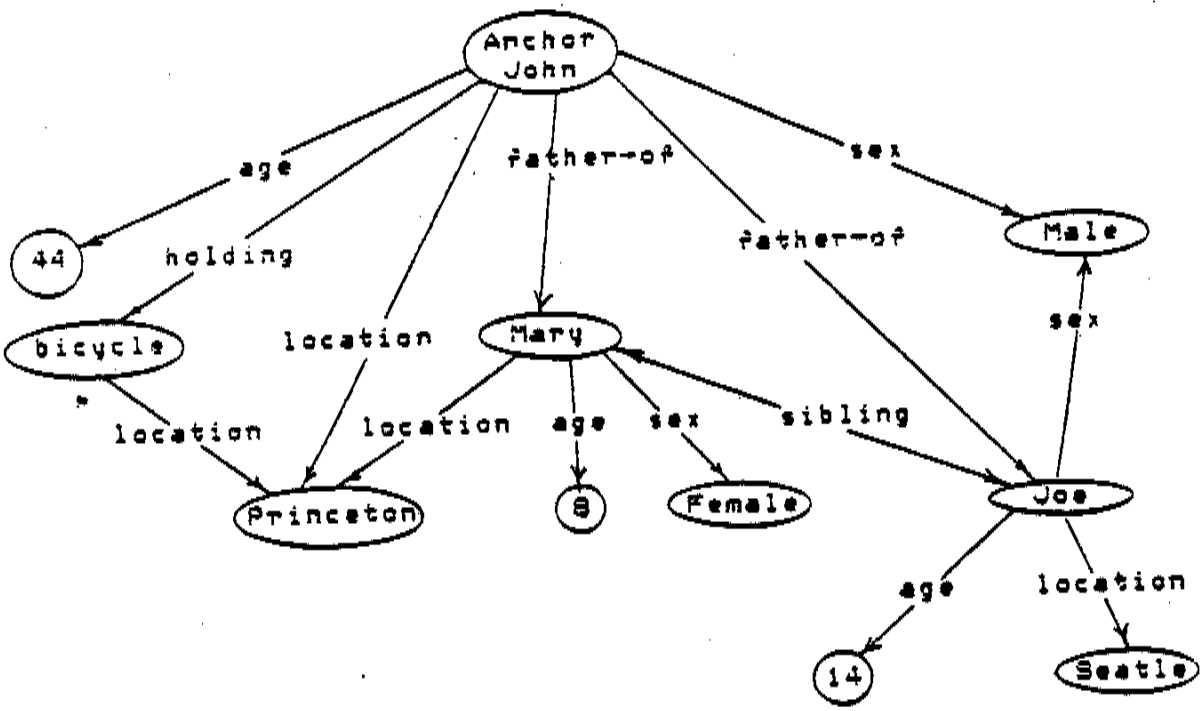

Figure 4: The connection Graph C-Graph(John,s)

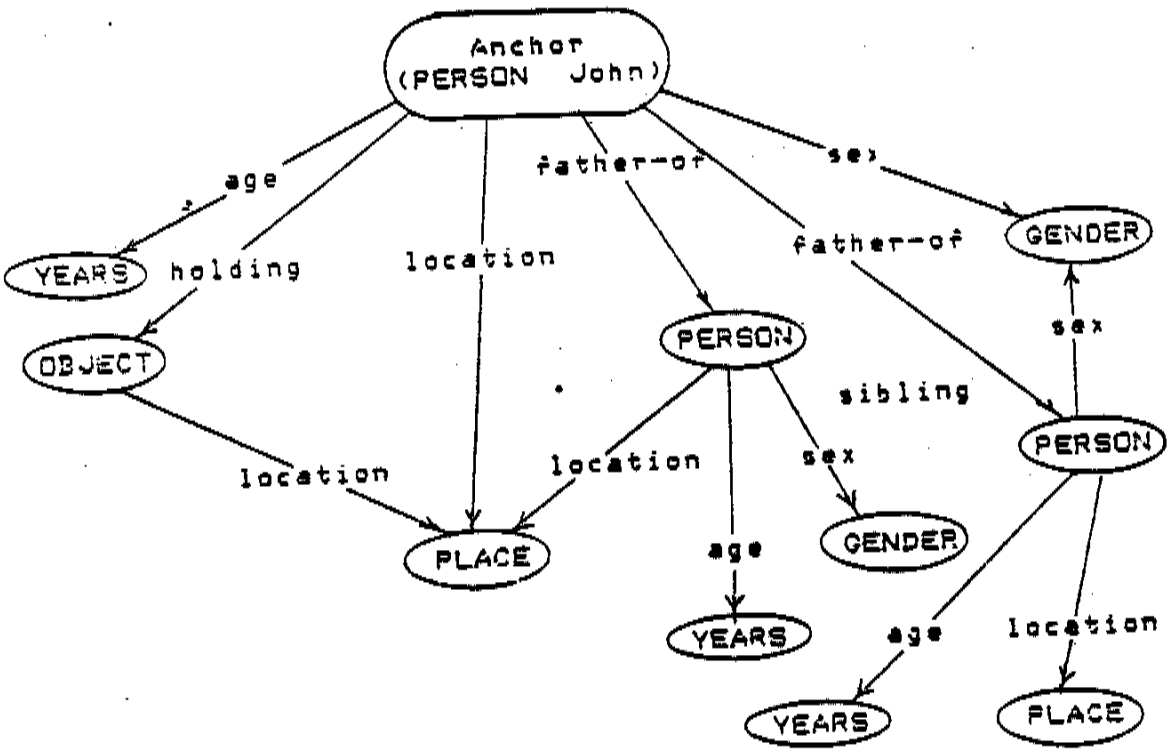

Figure 5: The Dependency Graph, D-Graph((John,PERSON),s).

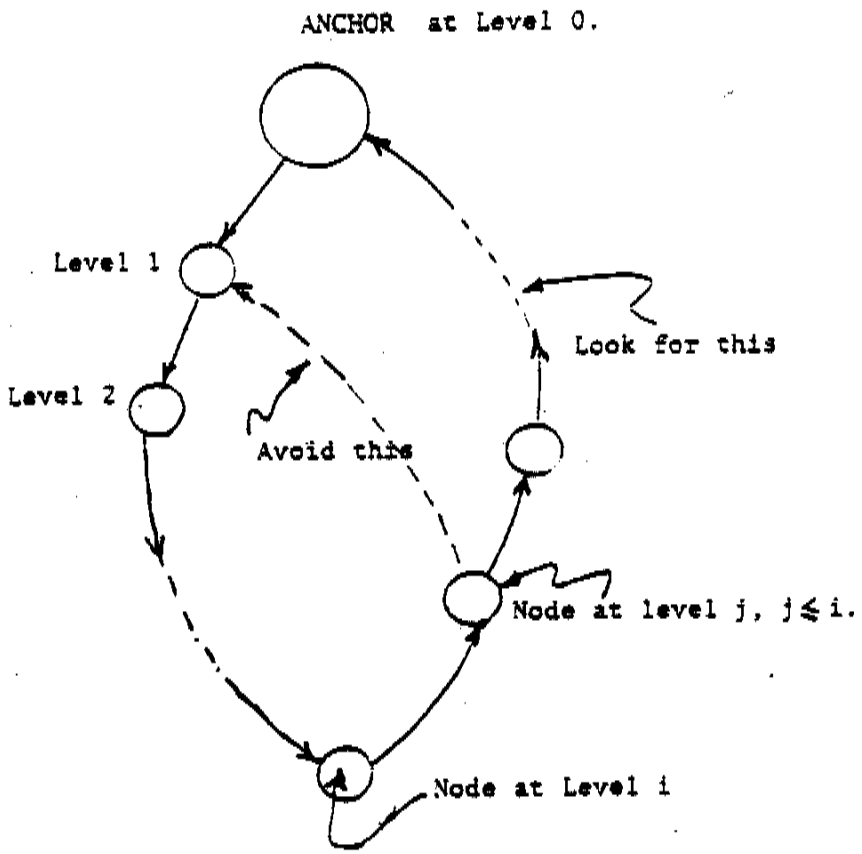

Figure 6: Search for Loops in Dependency Graphs.