Massively Parallel Knowledge Representation

James Geller*
Institute for Integrated Systems
CIS Department
New Jersey Institute of Technology
Newark, NJ 07102
email: geller@hertz.njit.edu

Abstract

We present a massively parallel representation of transitive relations, emphasizing the subclass relation, which extends our previous linear tree representation of class hierarchies. This representation makes use of a grid of processors and distributes information about one class over the processors of one column of the grid. As such, it can deal with a subset of directed acyclic graphs. We prove that a "node insertion" can be performed efficiently in our representation.

1 Introduction

Class hierarchies form the backbone of all frame systems, most semantic network theories, and all object-oriented languages, databases and systems. Their computational investigation goes back to Quillian [1] and forms a main topic of research in AI, Knowledge Representation [2, 3], Databases [4], and Object-Oriented systems [5]. Surprisingly, AI researchers had not given much thought to efficient implementations of large class hierarchies until Schubert [7] introduced a coding that permitted the fast verification or denial of a subclass relation between two given classes.

The fast, "reflexive" [8] responses of human subjects to questions involving IS-A hierarchies led some researchers such as Schubert and Shastri [9, 10] to exploit class hierarchies using special purpose reasoners. AI researchers have, in general, not extended this approach to other transitive relations. Humans can respond to questions such as, "What is older, the pyramids or the US constitution"? with equal "reflexive" speed. The same applies to relations such as taller, faster, heavier, and so on. In the (relational) database literature, transitive closure operations for other relations have drawn considerable attention [11, 12].

To achieve fast response times for special purpose IS-A reasoners, a few AI researchers have made use of massive parallelism. Shastri [10] uses a connectionist massively parallel method, while Evett, Hendler

*This work was partially supported by the National Science Foundation, Grant #IRI-9204655. This work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by and operates under contract to DARPA and the Air Force Systems Command, Rome Air Development Center (RADC), Griffiss Air Force Base, NY, under contract# F306002-88-C-0031.

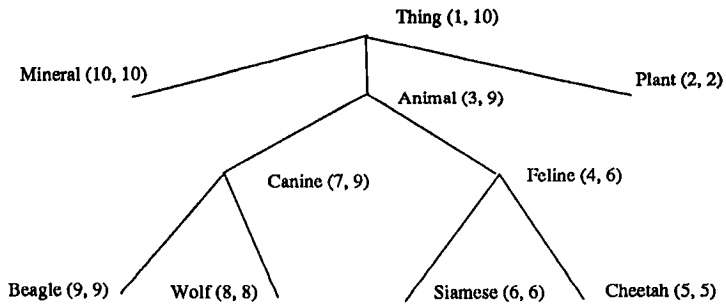

Figure 1: A class tree with (preorder, maximum) pairs

and Spector [13] and Geller [14] have used symbolic massively parallel techniques. We group such symbolic approaches under the label "Massively Parallel Knowledge Representation."

Geller and Du [14] expanded on Schubert's special-purpose reasoner and provided a fast, massively parallel update technique for his class tree encoding. The basic idea of this encoding is to assign a (preorder number, maximum number) pair to every node in a class tree. The maximum number for every node is the maximum preorder number of the subtree rooted at that node. The existence of a subclass relation between two nodes is then equivalent to the existence of an interval inclusion relation between their number pairs. Figure 1 shows a class tree with such a (right-to-left) preorder numbering. Arcs are assumed to point from a super-class to a subclass.

To expand on this, we eliminated the storage of the tree structure and organized nodes in a linear representation that maintains all important information of the tree. This *linear tree representation* is then implemented by assigning one node to one processor of a massively parallel computer. The major weakness of both implementations are their limitation to trees. The elegance of Schubert's ideas lead Agrawal, Borgida, and Jagadish [12] to examine the question of how to extend them to directed acyclic graphs.

While Agrawal *et al.* did not find a way to reduce the subclass verification to a single interval comparison, they introduced techniques to minimize the total number of pairs that occur in a class graph. They proved that their minima are theoretically optimal. Most of their results were derived for static systems where a class graph can be encoded at "compile time." For AI systems that require dynamic behavior, such as class reasoners, they suggested the use of a numbering scheme with *gaps*. Unfortunately, there are problems with this suggestion, one of these being that a sequence of classes can be constructed which would fill up even a large gap. In that case a complete sweep over all classes becomes necessary. In addition, their system requires book keeping concerning the numbers that are currently in use in the hierarchy.

In this paper, we will investigate an alternative gap-free technique for the dynamic update of coded acyclic class graphs. This technique builds on our previous massively parallel implementation of class trees. Nothing in what follows is specific to class graphs, and these techniques may be applied to any transitive relation.

Figure 2 explains the basic approach of [12] (with a preorder numbering, used instead of their postorder numbering). A spanning tree is constructed such that at every node with multiple parents, we select the link to the parent with the maximum number of predecessors (i.e., nodes reachable in an exhaustive upward search). Then (preorder, maximum) pairs are assigned to the nodes of the spanning tree. After that, all

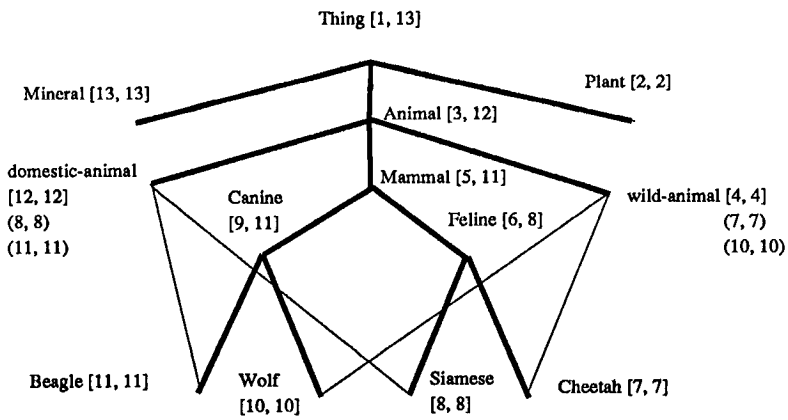

Figure 2: A class graph with tree pairs and non-tree pairs

edges are used, from the bottom (in inverse topological order), to propagate number pairs upward. A number pair (π, μ) is propagated upward until it reaches a node where a number pair (π_1, μ_1) exists and encloses it (i.e., $\pi_1 \leq \pi$ and $\mu \leq \mu_1$.)

A pair that is the result of numbering the spanning tree is called a *tree pair.* All pairs that are the result of propagation are called *non-tree pairs.* We are using square brackets for tree pairs and parentheses for propagated pairs. Obviously, it is never necessary to propagate a tree pair along a tree edge. The verification of a subclass relation between a node A with m pairs and a node B with n pairs might now require the comparison of all m pairs with all n pairs, which is not a constant time operation.

Agrawal *et al.* state (p. 254) "that a graph might need several spanning trees to cover all its nodes." They do not go into the details of how this might be done, but the following approach is natural. All pairs of nodes for which no lattice-join exists are made children of one unique top node.[1] Then a unique spanning tree can be constructed that covers all of the nodes. After assigning number pairs to all nodes, the top node may be erased again, creating a spanning forest. At times, we will assume that certain graphs must form an upper semi-lattice, which also results in a unique spanning tree.

2 A Massively Parallel Encoding

We now describe our incremental massively parallel encoding of such graphs. The possibility of multiple number pairs at every node requires that we find a new representation that maintains as many of the advantages of the linear tree representation as possible. The architecture of the underlying massively parallel machine model suggests at least two different solutions: (a) An array of number pairs at every node; (b) A distributed representation where every number pair is assigned to one processor such that all pairs of one class form one column in a grid of processors. Figure 3 shows how to apply (b) to Figure 2.

To choose between (a) and (b), we performed an analysis of different operations that have to be supported. An experiment, described below, indicates that for IS-A verification, representation (a) is slightly better.

[1] In many AI systems, this is the THING node.

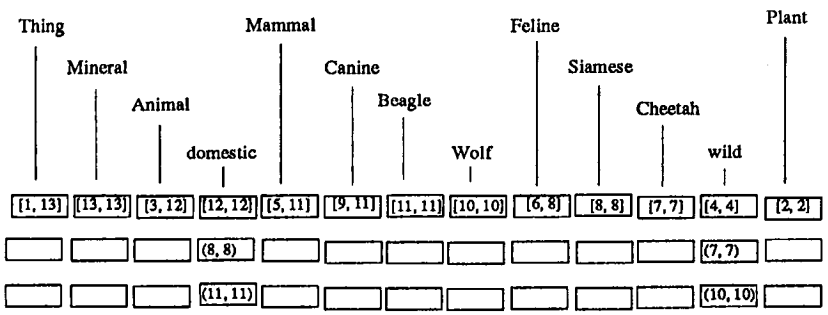

Figure 3: Distributed class graph representation

Nevertheless, we use (b) in this paper because it is better for update operations.

On a CM-5 Connection Machine[2] with 32 physical processors and a virtual processor set of 8 rows and 128 columns, two tests corresponding to the options (a) and (b) above, were performed. In the first test, an array of 8 rows and 2 columns was assigned to all processors. Only the first row of processors was used. The arrays of two of the first row processors were transferred to the front end computer and compared there. In the second test, the number pairs of all processors of one column were compared with the number pairs of all processors of another column. Number pairs were stored in two parallel variables, one for preorder numbers and one for maximum numbers. The results of these experiments are shown in Table 1. Representation (a), the array representation, is clearly faster.

Array		Distributed	
real time	CM-5 time	real time	CM-5 time
0.033566	0.000889	0.157592	0.007113
0.033537	0.000889	0.201446	0.007020
0.063595	0.000907	0.162658	0.006532
0.041867	0.000903	0.168910	0.006511

Table 1

We are currently not concerned about the wastefulness of this representation.[3] First of all, as was pointed out before, we are interested in maintaining other binary transitive relations, not just IS-A. Clearly, these different relations will not interact during relation verification, and we can assign all relations to the same set of processors. Therefore, one processor may have one number pair per relation and class.

Secondly, we feel strongly that AI should be concerned about solving real problems, and not about parsimony, which we will worry about when all other problems are solved. Minsky has supported this observation in another context [15]. If nature has found it necessary to use ten billion (simple) processing elements for solving the problem of intelligence, then maybe this is indeed the required magnitude of processing elements, real or simulated.

[2] Connection Machine and CM-5 are trademarks of Thinking Machines Corporation.

[3] Reviewers have previously considered even the idea of one class per processor as completely unacceptable.

3 Combining Tangled Hierarchies of Transitive Relations

As with class trees, we also need operations that combine two acyclic graphs of transitive relations. Such a combination operation establishes one or more links from a graph H to a graph G. For class hierarchies, a link from the node P in H to C in G asserts that C IS-A P. Because in the graphical representation, G is "moved into" H, we will call H the *target graph* and G the *inserted graph.* The result of the operation will be called the *combined graph I.* The spanning trees of these graphs will be called *target tree*, *inserted tree* and *combined tree*, respectively. P and C may be thought of as *parent* and *child* nodes.

A combination operation may connect one node in H to one node in G (one-to-one), one node in H to many nodes in G (one-to-many), many in H to one in G (many-to-one) or many in H to many in G (many-to-many). In this paper, we will take the view that a combination that involves "many" either in G or in H (or in both) should be decomposed into a one-to-one graph combination operation followed by a series of intra-graph link insertions. We will discuss only the one-to-one graph combination for the case that G is an upper semi-lattice, and C is the root of G. This will be called a *node insertion.* The case of an intra-graph *link insertion* will be treated in a forthcoming paper [16].

Without loss of generality, C is inserted at the leftmost position under P, where "leftmost" is defined with respect to the spanning tree of H. A linear graph representation is generated by a left-to-right preorder traversal of the spanning tree.

The steps that have to be taken to perform a node insertion operation rely on (1) the techniques developed for class trees [14] and (2) the use of a "pointer" from every non-tree pair to the tree pair from which it is derived. Because a class is represented by a column in the grid of processors, and because all tree pairs are stored in row 0, all we need for this pointer is the column number of the class that is the source of propagation. The propagation of all tree pairs to non-tree pairs is then reduced to two parallel retrieval/set operations performed on all nodes for which preorder and maximum numbers are currently defined.

Node Insertion Algorithm:

- All tree pairs in the target graph are updated exactly as if G and H were trees.
- All tree pairs in the inserted graph are updated exactly as if G and H were trees.
- The propagation pointers are updated to reflect the new location of classes due to the previous insertion step. The details of this step are shown below as a parallel algorithm.
- The linear representation of G is inserted immediately after the parent node P into the linear representation of the target graph H.
- All tree pairs are copied, using the updated propagation pointers, to those classes where they function as non-tree pairs.

Theorem 1: The node insertion algorithm correctly transforms the linear representations of the inserted and target graphs into the linear representation of the combined graph.

Proof: The proof follows from the following lemmas **1–6**. Due to space limitations, the proofs of the Lemmas are omitted.

Lemma 1: The spanning tree of the target graph is not changed by the node insertion operation.

Lemma 2: The spanning tree of the inserted graph is not changed by the node insertion operation.

Lemma 3: The rules that govern the update of number pairs for the insertion of a tree into another tree [14] can be applied unchanged to the tree pairs of the inserted and target graphs.

Remark: A complete proof of this lemma would require the presentation of the rules for inserting a tree into another tree [14]. This would be redundant and, in fact, all we need is the much weaker lemma **3b.**

Lemma 3b: Once the spanning tree has been established, the non-tree arcs of a graph have no influence on the tree pairs.

Lemma 4: The connection between the inserted graph and the target graph will be part of the spanning tree of the combined graph.

Definition: *A path of propagation* is any sequence of arcs such that the same pair appears at the end of the path as a tree pair and at the beginning of the path as a non-tree pair.

(Note: Due to the assumption that arcs in the graph point against the IS-A direction, the propagation also happens against the direction of the path of propagation.)

Lemma 5: After a node insertion operation, no path of propagation exists that passes through the link connecting the target graph with the inserted graph.

Pointer Update Operation: In the description of the pointer update algorithm, the lower index i describes a property of the inserted graph, and the lower index t describes a property of the target graph. The upper index r stands for the root of a spanning tree. Variables marked with !! are parallel variables, and operations marked with !! or involving parallel variables are parallel operations. The function address() returns the processor number of a specific node. *Parent* is the node under which the inserted graph is placed. The parallel variable self-address-x!! maintains for every processor its column number in the grid. Undefined pointer values are represented by a value of −1 and are not updated. (This is not shown).

```
IF    address(root_i) > address(root_t)
      THEN IF!! address(root_t) ≤ self-address-x!! ≤ address(root_t) + μ_t^r − 1 AND!!
                address(Parent) + 1 ≤ Pointer!!
           THEN!! Pointer!! := Pointer + μ_i^r;
           IF!! address(root_i) ≤ self-address-x!! ≤ address(root_i) + μ_i^r − 1
           THEN!! Pointer!! := Pointer!! + address(Parent) + 1 − address(root_i);
      ELSE IF!! address(root_t) ≤ self-address-x!! ≤ address(root_t) + μ_t^r −1 AND!!
                Pointer ≤ address(Parent)
           THEN!! Pointer!! := Pointer − μ_i^r;
           IF!! address(root_i) ≤ self-address-x!! ≤ address(root_i) + μ_i^r − 1
           THEN!! Pointer!! := Pointer!! + address(Parent) − address(r_i) − μ_i^r + 1;
```

Lemma 6: The described pointer update operation correctly maintains all paths of propagation in the inserted graph and the target graph.

The following Table 2 presents the results of four series of experiments with the node insertion algorithm on a CM-5 Connection Machine configured as a 128 × 8 grid. All times are in seconds. It is notable that run times for left moves in the second part of Table 2 appear constant, while run times for right moves are steadily growing. An analysis of the precise operations performed during those tests indicates that there are differences between the amounts of data moved. For every node insertion, we recorded the number of nodes moved, the distance they were moved, and the product of these two numbers. For right moves, the numbers of nodes grew from 80 to 912, while for left moves they grew only from 72 to 160. The products for right moves grew from 256 to 3584, while for left moves they grew only from 128 to 1536.

# of nodes in graph		right move		# of nodes in graph		left move	
target gr.	inserted gr.	real time	CM-5 time	target gr.	inserted gr.	real time	CM-5 time
8	8	0.669272	0.035810	8	8	0.668897	0.034845
16	16	0.203851	0.039741	16	16	0.188714	0.038770
32	32	0.207186	0.046521	32	32	0.313182	0.047159
64	64	0.264549	0.061211	64	64	0.256379	0.060117
8	16	0.198215	0.039704	16	8	0.171111	0.035295
8	32	0.219306	0.046881	32	8	0.170450	0.035410
8	48	0.242117	0.053720	48	8	0.175289	0.035563
8	64	0.263427	0.060482	64	8	0.170569	0.035393
8	80	0.281709	0.067063	80	8	0.174794	0.035438
8	96	0.317737	0.074421	96	8	0.176339	0.035926
8	112	0.329498	0.080531	112	8	0.174068	0.035866

Table 2

4 Conclusions

We have argued for a representation of transitive relationships on massively parallel computers that stands between that of Evett *et al.* [13] and the neural network approaches. Our approach, like Evett's, uses a symbolic representation based on a class hierarchy. In their representation, one frame (class) is assigned to exactly one processor, which is not the case for us. Our approach shares a distributed representation with many neural network models [17]. Information about one class is assigned to many processors. Our approach also uses numeric encodings, in common with neural network representations. However, the number pairs have a clear interpretation, which is not the case for the weights of neural networks.

We demonstrated that our previous results for parallel insertion of class trees into other class trees can be extended to the parallel insertion of directed acyclic graphs of transitive relations. This extension maintains the rules for tree pair updates and accommodates non-tree pair updates by a parallel propagation scheme using a column pointer. A major result was to show a parallel algorithm for maintaining this pointer correctly. Run times for the node insertion algorithm were presented.

ACKNOWLEDGEMENT

Yehoshua Perl and Mike Halper have made helpful comments about an earlier draft of this paper.

References

[1] M. Quillian, "Semantic memory," in *Semantic Information Processing* (M. Minsky, ed.), pp. 227–270, Cambridge, MA: The MIT Press, 1968.

[2] A. Borgida, R. Brachman, D. McGuinness, and L. A. Resnick, "Classic: A structural data model for objects," in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, (Portland, OR), pp. 58–67, 1989.

[3] R. Brachman, "On the epistemological status of semantic networks," in *Associative Networks* (N. Findler, ed.), pp. 3–50, New York, NY: Academic Press, 1979.

[4] W. Kim, N. Ballou, H.-T. Chou, and J. Garza, "Features of the ORION object-oriented database system," in *Object-Oriented Concepts, Databases, and Applications* (W. Kim and F. Lochovsky, eds.), Reading, MA: Addison Wesley, 1989.

[5] S. Keene, *Object-Oriented Programming in COMMON LISP.* Reading, MA: Addison-Wesley, 1989.

[6] L. Schubert, M. Papalaskaris, and J. Taugher, "Determining type, part, color, and time relationships," *Computer*, vol. 16, no. 10, 1983.

[7] L. Schubert, M. Papalaskaris, and J. Taugher, "Accelerating deductive inference: special methods for taxonomies, colors and times," in *The Knowledge Frontier* (N. Cercone and G. McCalla, eds.), pp. 187–220, New York, NY: Springer Verlag, 1987.

[8] L. Shastri and V. Ajjanagadde, "An optimally efficient limited inference system," in *Proceedings of IJCAI-90*, (Boston, MA), pp. 563–570, 1990.

[9] L. Shastri, *Semantic Networks: An Evidential Formalization and its Connectionist Realization.* San Mateo, CA: Morgan Kaufmann Publishers, 1988.

[10] L. Shastri, "Default reasoning in semantic networks: A formalization of recognition and inheritance," *Artificial Intelligence*, vol. 39, no. 3, pp. 283–356, 1989.

[11] R. Agrawal, S. Dar, and H. Jagadish, "Direct transitive closure algorithms: Design and performance evaluation," *ACM Transactions on Database Systems*, vol. 15, no. 3, pp. 428–458, 1990.

[12] R. Agrawal, A. Borgida, and H. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, (Portland, OR), pp. 253–262, 1989.

[13] M. Evett, J. Hendler, and L. Spector, "PARKA: Parallel knowledge representation on the Connection Machine," Tech. Rep. UMIACS-TR-90-22, Department of Computer Science, 1990.

[14] J. Geller and C. Du, "Parallel implementation of a class reasoner," *Journal for Experimental and Theoretical Artificial Intelligence*, vol. 3, pp. 109–127, 1991.

[15] M. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision* (P. Winston, ed.), pp. 211–277, McGraw-Hill, New York, 1975.

[16] J. Geller, "Combining massively parallel tangled hierarchies of relations," tech. rep., Manuscript in preparation, 1993.

[17] D. Touretzky, *Neural Information Processing Systems* 1. San Mateo, CA: Morgan Kaufmann Publishers, 1989.