Chapter XX

A NEURAL-NETWORK APPROACH TO IMPLEMENTING CONCEPTUAL GRAPHS

George G. Lendaris
Systems Science Ph.D. Program
Portland State University
Portland, OR 97207-0751

XX.1 INTRODUCTION

In this chapter, we explore the possibility of using neural networks to implement some important manipulations on conceptual graphs. A major benefit of implementing conceptual-graph manipulations on neural networks (NNs) would be a significant reduction in computation time (plus some other benefits that accrue from the distributed representation properties of NNs).

Conceptual graphs are a representation in symbolic form of the intellectual constructs called concepts and concept-relations. There has been no intrinsic difficulty with the symbolic nature of conceptual graphs regarding their implementation on our usual symbol-processing computers. Neural networks, however, are not symbol processors per se. Therefore, a necessary first step in the present endeavor is to develop a representation for conceptual graphs that functions as a non-symbolic representation when used as input to a neural network. In this section, we look at some aspects of how neural networks operate, to better understand the needs of a representation schema for input to such networks.

A number of books on neural networks have recently been published, many of them offering a good introduction to the subject. A readable general overview (no equations) is given in [Caudill & Butler, 1990]. A list of 10 other recent books is included at the end of this chapter. We will describe here those aspects of neural networks needed as background for the present problem context.

A neural network can be thought of as a device that has n input terminals and m output terminals (refer to Figure 1). The input and output signals may be either continuous or binary valued. The internal architecture of the device is generally parallel. At a given point in time, the input signals are set to some combination of values; the device processes all these inputs simultaneously; and at the end of
Figure 1. Neural Network: device with $n$ input terminals and $m$ output terminals, continuous or binary-valued signals.

the processing cycle (a fixed amount of time), presents appropriate
values on all the output terminals.

Moving inside the device, a neural network is typically made up of
a collection of simple (neuron-like) processing elements whose
interconnection parameters are adjusted during training (cf. Figure 2).
For any particular setting of the adjustments, the network performs one
of the mappings possible for that network, where the set of possible
mappings is determined by the type and number of elements and the
pattern of their interconnections [Lendaris & Stanley, 1963, 1965a,

Figure 2. Graphical representation of typical neuron-like processing
element in a connectionist network. Each element has a value on its output,
usually ranging from 0 to 1. The amount fed into an element is
determined by the adjustable parameter called a connection weight ($w_i$):
the weight can take on positive or negative values. The element adds up the
weighted inputs, and applies the transformation (on the right side of the
dashed line) to the sum to yield the element's output. The output of an
element feeds one or more other elements.

The term 'layers' refers to the organization of the elements in the
network. If each element receives input only from the environment, then
there is only 1 layer. If outputs from the units in the first layer feed into
another set of units, that set constitutes a 2nd layer. If outputs from the
units in the second layer feed into another set of units, the latter set of units
constitutes a 3rd layer, and the 2nd layer is said to be "hidden" (from the
environment).
1965b]. The set of mappings achievable by a NN as its internal parameters are adjusted is significantly smaller than the set of all possible ones (for a usefully large number of inputs, that is), and because of this, there is no guarantee that a given neural network device can be adjusted to perform a mapping that a user happens to desire. But, since any problem can be represented to a neural network in various ways, and since the desired mapping depends explicitly on the representation, it behooves the researcher to take care in selecting the representation, so that the mapping to be performed on the associated "signals" has a chance for successful implementation by the neural network. At the present time, there is no formal theoretical basis to guide such selections, so ad hoc methods still have to be used.

There are many tasks that human beings know how to do, but don't know how they do these tasks well enough to directly design a machine to do them (e.g., reading human handwriting, recognizing human faces, making good wine, etc.). We refer to the human performance ability in these cases as a skill. A person is able to pass a skill on to another person by entering into a teacher-pupil relationship with that person. A key aspect of what neural network researchers do is to develop methods that will allow a machine to emulate a "pupil" role, and learn certain tasks by interaction with its environment, with or without a teacher role (supervised or non-supervised learning). The algorithmic procedure for accomplishing this task is generally called a "training (or, learning) algorithm."

The supervised training process typically proceeds as follows: a) a network of neuron-like processors with a starting set of connections and connection weights is given an input pattern; b) the network computes an output; c) the output is inspected and compared with the desired output for that input; d) if not correct, then changes in the connection parameters within the network are made in such a way that it is less likely the same output would be calculated for the given input pattern; e) if correct, then usually nothing is done, but, in principle, a change in the weights could be made in such a way that it will be more likely the same output would be calculated for the given input pattern; f) steps a-e are repeated for all the input patterns in the 'training set'; g) the training algorithm determines how many times to repeat step f. A key difficulty during training is that of losing "good" information already accumulated when adjustments are made for a current training input.

Coupled with the above considerations is the important notion of generalization. When applied to human processing, this term normally means that after one is exposed to a certain set of inputs and associated correct outputs, a pattern is perceived that allows one to generate correct outputs for inputs not previously seen. In the context of training a neural network, (correct) generalization again means that
only a portion of the data from the problem environment is used during training, and the network gives correct answers to the rest of the available data. A crucial issue: how is correct generalization from a training set achieved? A key notion here is the underlying pattern in the data. Associated with any pattern are regularities, and with regularities, constraints. Much can be said about this topic, but the key idea is that REGULARITIES/CONSTRAINTS MUST BE PRESENT WITHIN AND AMONG THE TRAINING DATA FOR (GOOD) GENERALIZATION TO BE POSSIBLE. This condition is satisfied in normal real-world applications. The "rule(s)" developed by the NN to determine an output for a given input must in some sense embody the constraints implicit in the data.

The remainder of this chapter is organized as follows. In Section 2, a representation schema is developed for conceptual graphs that will allow them to be input to neural networks; Section 3 takes the reader on a walk through a candidate implementation of an important conceptual-graph manipulation called PROJECTION: Section 3a defines the PROJECTION operation, and develops two PROPERTIES that must hold for the PROJECTION; Section 3b describes a method to test for one of these properties via a neural network, and Section 3c cites some experimental results; Section 4 discusses an extension of the representation schema for multiple copies of relations and concept types; Section 5 sketches a possible extension of the methods given in Section 3 for PROJECTION to another important conceptual-graph manipulation called MAXIMAL JOIN; Section 6 suggests the possibility of using neural networks as memory for conceptual-graph knowledge bases; and finally, Section 7 provides a summary of the chapter.

XX.2 REPRESENTING CONCEPTUAL GRAPHS FOR PARALLEL PROCESSING

As indicated in the previous section, neural networks are not symbolic processors, so we are obliged to develop a non-symbolic means of representing conceptual graphs in order to use neural networks to implement operations on them. In the vocabulary of Figure 1, whatever representation schema we develop, its components will be the "signals" presented at the input terminals of the neural network device.

**Basics of Sowa's Conceptual Graph Formalism**

Conceptual graphs are defined in detail in this book in Sowa's chapter on notation. Of specific interest here is the fact that a conceptual graph consists of two different kinds of nodes (concept nodes and relation nodes) and directed arcs. An arc connects a concept node to a relation node, or a relation node to a concept node;
connections are not allowed between nodes of the same type. An example of a generic conceptual graph is given in Figure 3a.

Circles with relation labels, squares with concept labels, and lines with arrows on them are used to depict a conceptual graph in a graphical format—called the "display form," usually the form preferred by human observers. An alternative "linear form" with parentheses around relation labels, brackets around concept labels, and arrow indicators is used to represent a conceptual graph for easier input to computers via keyboard, and output via conventional printers. This latter form has been useful for the serial-type, symbolic processing typical in implementations to date. See Figure 3b for a linear-form representation of the conceptual graph shown via the display form in Figure 3a.

![Graphical Representation](image)

**Figure 3a.** Generic example of a conceptual graph (Display Form).

(RELATION 1)

- [CONCEPT 3] \( \rightarrow \) [RELATION 2] \( \rightarrow \) [CONCEPT 4]
- [CONCEPT 1] \( \rightarrow \) [CONCEPT 2]

**Figure 3b.** Linear Form representation for the conceptual graph shown via Display Form in Figure 3a.

**Components of Knowledge Systems using Sowa's Formalism**

The problem environment to be used in the remainder of this chapter will be knowledge systems that use conceptual graphs, and the tasks we will want the neural network to perform are manipulations related to conceptual graphs. Knowledge systems using Sowa's formalism typically comprise the following components:
1. First, a knowledge engineer gathers together data which are coalesced into (true) assertions, and these in turn are represented via conceptual graphs (CGs). [Note: The entire problem context (typically represented via a semantic network, which may or may not use the Sowa conceptual-graph formalism) is taken into account when defining and choosing the concept and relation labels to be used in the CGs.]

2. A catalog of all the concept types is created and maintained (let numC be the number of entries).

3. A hierarchy tree is created and maintained which encodes the sub- and super-type relationships among the concept types. This hierarchy is used for checking validity of restrictions and generalizations of CGs.

4. A catalog of all the relation types is created and maintained (let numR be the number of entries).

4a. A relation definition dictionary is created and maintained. For each type in the relation catalog, there is listed a set of rules (constraints) which dictate the number of arcs leaving and entering the node, and the maximal type of the concept to which each arc may be attached (maximal here means the highest concept type in the hierarchy to which it is allowed to connect the given arc).

5. A catalog of names for individuals (also called markers) appearing in the knowledge base is created and maintained. Each marker in this list carries an indication of its associated concept sub-type; the latter is used for making conformity checks.

6. A restriction operates on a concept to yield either a concept which is a sub-type of the original [e.g., person to man], or, a concept which has been specialized to an individual in the context [e.g., person to person:Gregory]--also called an instantiation.

7. A generalization operates on a concept to yield a concept which is a super-type of the original [e.g., man to person].

8. The conformity operation checks to determine whether the concept type associated with the name of the individual assigned during a restriction is appropriate to the concept to which it was assigned--e.g., man:Gregory vs. woman:Gregory (the reader may look ahead to Figure 10).

Hints from nature
As mentioned earlier, the task before us is to devise a non-symbolic way to represent conceptual graphs. A natural question is, if we don't use symbols, then what do we use? We need only look to biological systems for an answer: values. That is, biological systems obtain data from their environment via their sensors. The sensors provide the brain with a variety of signals, and these signals have values and time-of-occurrence relationships. The signals coming from each of the
sensors no doubt have their own definite meaning, and this meaning does not (normally) change with time. We use these observations to guide us in developing a representation for conceptual graphs. It is important to keep in mind that the "sensors" and "signals" can be defined in an abstract and/or arbitrary way, but a key requirement is that the resulting encoding must contain the information about the problem environment that is needed to perform the desired operations. As examples, the signals may be voltage values from an array of light sensors upon which photographic images are projected; the signals may be numbers representing a set of dynamic characteristics—position, velocity, acceleration, etc.—of a problem environment where physical objects are in motion; the signals may correspond to names of individuals in an extended family and the various relationships that can exist between pairs of the individuals (e.g., brother, husband, wife, sister, etc.) What can we use as the equivalent to sensors and signals to represent conceptual graphs?

The answer suggested here is to use a matrix form (and certain associated vector form) representation of conceptual graphs.

Full Matrix Representation

It is well known in graph theory [Roberts, 1976] that a unique connection matrix (CM) representation exists for certain types of graphs (including those of interest here). The connection matrix contains as many rows (and columns) as there are nodes in the graph. A distinct row and column of the matrix is assigned to each node in the graph. In conceptual graphs, the arcs have arrows on them (i.e., are directed); direction is normally from Row node to Column node in connection matrixes, but this choice is arbitrary. Using this convention, an entry of 1 is placed in the (Row i, Column j) slot of the matrix if there exists a directed arc connecting node i to node j in the graph. No entry (or equivalently, a 0 entry) is made where there is no corresponding connection.

Any conceptual graph (CG) can be represented via a connection matrix (CM) of the type described above. We let the CM play the role of "sensor", with each slot providing a "signal" whose value is 1 or 0. The CM may be constructed by assigning to each node in the CG a different row and column in the matrix, and by entering a 1 in those slots where a (directed) connection exists from the row to column nodes. See Figure 4a. However, if done in this way, each CG with different concept and relation nodes would have a connection matrix of different size, and with different row (and column) labels. But, the meaning of each signal coming into the neural network is supposed to stay the same over time—i.e., a requirement that the structure of the matrix be the same for all CGs, with only the matrix entries changing. One possibility is to create a template matrix with (numR + numC) rows
and columns, and assign to each row/column one of the concepts in the concept-type catalog or one of the relations in the relation catalog. A connection matrix for a given conceptual graph would then be created simply by entering 1's in the appropriate slots of the template matrix. In this way, each concept type and each relation corresponds to a particular position in the row sequence and a particular position in the column sequence of the CM. This position encoding of concept types and relations is the key attribute of the proposed encoding schema for our intended application.

**Reduced Matrix Representation**

Since there are no connections among nodes of the same type in a conceptual graph, if the concept nodes are assigned to rows (and columns) adjacent to one another, and the relation nodes assigned to the remaining rows (and columns), then there are no entries in the upper left and lower right quadrants of the matrix (see Figure 4b). This observation suggests another form. Choose a (reduced size) matrix having only as many rows as there are relation types in the knowledge base catalog (numR), and only as many columns as there are concept types (numC). [This row/column assignment is chosen to take advantage of the obvious mnemonic: the R for relation is assigned to rows, and the C for concepts to columns.]

Let the entries in the matrix be 3-valued, so that, for example, if a connection goes from the relation to a concept (row to column), use a +1; if the connection is in the other direction, use a -1; if there is no connection, use a 0. Let us call this matrix a Relation-Concept connection matrix, or, R-C connection matrix. See Figure 4c (left side).

It turns out that direction information can be relaxed in some cases and still allow for significant CG operations to be performed. For these cases, the requirement for ternary valued entries in the (reduced size) R-C connection matrix can be relaxed to 2-values: 1 for connection, 0 for none. For ease of discussion, binary-valued entries will be used for remainder of this paper. See Figure 4c (right side).
Figures 4b&c. Connection Matrix forms for conceptual graph shown in Figure 3a, with organized row and column assignments.

The rules listed in the relation definition dictionary (item 4a in the list given earlier in this section) entail corresponding constraints in what appears in a connection matrix. For each relation row, there is a specified number of 1's that should appear in the row (the number of arcs coming to and going from the node), and there is only a certain subset of concept columns in which each 1 may be placed (the specified maximal concept type or any of its subtypes). [For the ternary case, +1's correspond to arcs leaving the relation node, and -1's to the arcs entering the relation node. Constraints apply accordingly.] The reader may look ahead to Figure 5 for examples.

As described so far, the connection matrix (full and reduced) only allows representation of CGs containing at most one copy of any specified relation or concept type. There are cases where more than one copy of a relation or concept type is needed in a CG; the connection matrix template method can be expanded to accommodate these, and will be described in Section 4, but this simpler version is sufficient for development of the basic concepts.

Vector Representation

For database query type operations, a further reduction in representation is possible. The reduction here is based on the fact that for any given conceptual graph, the R-C connection matrix will be very sparse; only a small fraction of all possible rows will have non-zero
entries. Therefore, the suggestion is to store only those rows of the R-C connection matrix that have non-zero entries. We call these RC-vectors (their length is numC). With this method, another vector is required so the position information for each of the RC-vectors may be retrieved; this vector is called an R-vector. Its length is numR, and is constructed for the CG by placing a 1 in the slots corresponding to the relation nodes in the given CG. The RC-vectors are stored in an order corresponding to the sequence of 1's in the R-vector. Further, a C-vector is constructed; its length is numC, and 1's are placed in slots corresponding to all the concept type nodes in the given CG (the reader may look ahead to Figure 11 in Section 4).

**CG Canonical Operations via Matrix Representations**

We have thus succeeded in designing a representation schema for conceptual graphs that is non symbolic vis-a-vis its role as input device to neural networks. A key question remains: Is the information content of this schema sufficient to allow the manipulations required for conceptual graphs in knowledge systems? One way to answer this question is to demonstrate that the four CG canonical operations for conceptual graphs can be carried out using the representations.

In the following, each of the four canonical operations is described, and then shown to be implementable with the CM representation schema.

**JOIN:** If two conceptual graphs both contain a concept node of the same type, then the graphs may be JOINed at that (common) concept node. The resulting CG contains one copy of that concept node, and, appended to this node are all the (directed) arcs [and associated graph components] that were previously appended to the two (similar) nodes in the CGs being joined. See Figure 5.

![Diagram](attachment:image.png)

**Figure 5. Example of JOIN via R-C Connection Matrix representations.**
It is easy to see in principle, that the connection matrix (CM) of the JOINed graph is obtainable simply by doing a logical-or operation on the connection matrices for the two graphs to be joined. That is, create the CM of the JOIN by entering a 1 in a slot if there is a 1 in the corresponding slot in the CM of either of the two graphs being joined. See Figure 5 for an example. A slight complication arises if the CMs of the two graphs being joined have a 1 in the same slot—more about this in Section 4 and in the discussion about SIMPLIFY below.

We have the opportunity to observe here how easy it would be, in principle, to perform the entire JOIN operation in the time it takes to inspect one pair of corresponding slots in the two CMs—no matter how many nodes the given CG contains. This could be accomplished by having in hardware the equivalent of three CM template matrixes, and separate hardware included to do the logical-or operation for each slot in the templates—i.e., to look at each of the corresponding slots in two of the filled in matrixes, and enter the result in the corresponding slot of the third matrix. All of these operations would be done in parallel, so the whole matrix is done in the time it takes to do an operation on one slot.

SIMPLIFY: When two conceptual graphs are JOINed, some relations in the resulting graph may become redundant. Remember that all relation nodes appended to both of the original concept nodes that were JOINed are appended to the one copy of the concept node in the JOINed graph. It is possible that the same type of relation node had been appended to both concept nodes in their respective graphs before the JOIN. If so, the JOINed graph will have two copies of this relation node type (we are talking about the definition of JOIN here, not the present implementation). Every relation node has a number of arcs specified by its definition, and each arc has a specified maximal concept type associated with it. If the corresponding arcs for both of the duplicated relation nodes go to the same concept types, respectively, then one of these relation nodes is considered redundant. The SIMPLIFY operation consists of deleting one of each pair of redundant relation nodes.

For a JOIN operation to result in a redundant Relation (call it R), certain constraints exist on the relative positions of the 1s in the connection matrices of the CGs prior to the JOIN. The corresponding 1s would have to be in concept locations which are either 1) the same type, or 2) one is a subtype of the other. In case 1), one need only check whether the concept in either (or both) of the CGs being JOINed has been individualized (assigned a marker). If so, those concept nodes will have to be treated separately (e.g., via a CONFORMITY check). If neither has been specialized, then the logical-or operation is allowed to proceed, and the SIMPLIFY would be automatically performed. In case 2), it is possible (after the JOIN) to do a
RESTRICTION (see Section 3b) which would render the two concept nodes the same, and hence, a SIMPLIFY would then be in order.

COPY: In principle, there is no difficulty in making a copy of the connection matrix. It is possible that in some implementation it would be desirable to use the same hardware as for the JOIN to do the COPY, rather than to provide a means for making a direct transfer to the location where the copy is to be stored. This could be accomplished by moving the connection matrix of the CG to be COPIED into the (hardware) matrix template described in the JOIN discussion, and a logical union performed with a matrix template containing all 0 entries. The (hardware) result matrix would contain a COPY of the original CM.

RESTRICT: This operation (defined in item 6 of the list at the beginning of this section) requires access to the concept-type hierarchy (item 3). The process is different in nature from the above three, and will be discussed later (in Section 3b).

Sowa [1984] demonstrates that most operations on conceptual graphs can be accomplished using the above four canonical operations. A particular operation, PROJECTION, is discussed in the next section.

Comments
First comment: the above matrix-form representation was developed to serve as input to neural networks. Once accomplished, this same representation could be used to explore more conventional parallel implementations. The presentation strategy for the remainder of this chapter is to develop design ideas for such implementations. The reason for this is that in the process of developing a conventional means of implementing the desired conceptual graph manipulations, we also develop better insights for the intended neural network implementation(s).

Second comment: it would be possible to create and store a full connection matrix representation for each of the conceptual graphs in the given knowledge base. It is clear, however, that explicit use of such a representation would be wasteful of space in a computer, since only a small fraction of the possible nodes are ever used in an individual CG. There are methods for efficiently encoding and manipulating such sparse matrixes in serial machines, but these considerations will not be pursued here. For parallel implementations, each slot in the matrix would require a slot in a register, or a separate input to a neural network. Therefore, it behooves us to develop a more frugal way of representation. The reduced matrix and vector representations described earlier offer such a mechanism. These will require fewer input terminals for the neural networks, and of course, less memory for storing the information.
XX.3a PROJECTION OPERATION FOR DATA BASE QUERY

In Sowa's chapter on notation in this book, he discusses the PROJECTION operation for conceptual graphs, and suggests that this operation could be used to answer database queries. We here pick up on that suggestion, and use it as a candidate application upon which to begin the analysis required to check out some implementation details. We will walk through the process that would be required to accomplish PROJECTION using the connection matrix representation developed in the previous section---with the specific goal of parallel implementation of the operations. In the analysis given in this Section, it will become clear that the operations on conceptual graphs we consider can be implemented using parallel architectures of a standard variety. Our position for the remainder of the chapter, however, is that these operations could alternatively be implemented using neural networks, and that this is a good objective to strive for.

For the purposes of this Section's analysis, we assume the problem context to be a database which contains a large number of "fact" conceptual graphs (we here call these FactCGs). A query is to be made via a properly formulated query conceptual graph (QueryCG). The query process brings up each FactCG and determines if it contains a candidate answer for the query, i.e., a specialization of the QueryCG. The PROJECTION operation maps general conceptual graphs into more specialized ones, therefore, Sowa's suggestion is equivalent to testing if the QueryCG PROJECTS into the FactCG.

We borrow an example given in Sowa's chapter: "The sentence Yojo is chasing a grey mouse is more specialized than the sentence Some animal is chasing an animal." Conceptual graphs for these two sentences are given in Figure 6. The PROJECTION operation is said to map the upper (more general) graph of Figure 6 into the lower (more specialized) one. For each concept or relation in the upper graph, there corresponds one of the concepts or relations in the lower graph. In particular,

\[
\begin{align*}
&\text{[ANIMAL]} \longrightarrow \text{[CAT:Yojo]} \\
&\text{[ANIMAL]} \longrightarrow \text{[MOUSE]} \\
&\text{[CHASE]} \longrightarrow \text{[CHASE]} \\
&(\text{AGNT}) \longrightarrow \text{(AGNT)} \\
&(\text{PTNT}) \longrightarrow \text{(PTNT)}
\end{align*}
\]

We note that the lower (more specialized) graph contains the concept [GREY] and the relation (COLR) is not contained in the upper graph. It is typical for PROJECTION to map a general graph into a proper subgraph of a specialized one.

We state a modified version of the definition of PROJECTION given in [Sowa, 1984, p. 99], using the letters Q (for QueryCG) and F (for FactCG) in place of his v and u. The reader may find it
convenient to refer to Figure 6, and relate Q (Query) to the upper CG, and F (Fact) to the lower CG.

DEFINITION:
For any conceptual graphs Q and F where F is a specialization of Q (F ≤ Q), there exists a mapping pi: Q → F, where piQ is a subgraph of F called a projection of Q in F. The "projection operator" pi has the following properties:
For each relation R in Q, piR is a relation of the same type in piQ.

If relation R in Q has n arcs, then piR in piQ also has n arcs.

If an arc of R is linked to a concept C in Q, then the corresponding arc of piR must be linked to piC in piQ.

For each concept C in Q, piC is a concept in piQ where type(piC) is a SUBtype of type(C)
(i.e., piC is either identical to C, or, it is a restriction of C)

If C is individual, then referent(piC) = referent(C).
If C is generic, then piC may be either generic or individual.

We can deduce from this definition that for a given FactCG to be an answer to (i.e., a specialization of) the QueryCG, the following properties must be true:
PROPERTY 1: For each relation node in QueryCG, at least one relation node of the same type must exist in the FactCG.

Comment 1. For each relation node in QueryCG, there must be an identical relation node in the FactCG; however, since the projection of QueryCG is allowed to be a SUBgraph of the FactCG, the FactCG could have more than one relation node of the type being
checked—hence the "at least one" quantifier in the Property 1 statement.

Comment 2. The projection mapping is not necessarily 1:1 [Sowa, 1984, p. 99]. This means, for example, that two relation nodes of the same type in QueryCG could map onto one relation node (of the same type) in the FactCG. Therefore, even if there is more than one relation node of a given type in QueryCG, it is possible for there to be fewer relation nodes of that type in the FactCG. [For this possibility, however, each relation type involved must have all of its corresponding arcs present (refer to item 4a in the earlier list), and each of these arcs must be connected to a concept node of an appropriate sub-type.]

PROPERTY 2: a) For each concept type in QueryCG, at least one concept node which is a subtype (i.e., is identical to or is a specialization) of that type must exist in the FactCG; b) each concept node in the FactCG must be linked to the same relation type(s) as is its corresponding concept node in QueryCG.

Comments 1 & 2 regarding relation nodes apply also to concept nodes, vis-à-vis part a) of PROPERTY 2.

Comment 3. Regarding part b) of PROPERTY 2, it is possible that a concept node in one graph is linked to a relation node of one type, while in the other graph a concept node of the same type is linked to a relation node of another type. Therefore, even if the requirement that each concept type in QueryCG be found in the FactCG is met, a further test may be needed to check if in the FactCG the concept node of each specified type is linked to a corresponding relation node of the appropriate type, and perhaps, to check if the link has the correct direction.

We recall that the arcs used in conceptual graphs have arrows on them to indicate a direction, and further, we showed in the previous section how the connection matrix method represents this direction. Though this direction information is needed for manipulations of conceptual graphs in some contexts (e.g., in language processing applications), it appears we can accomplish the PROPERTY 1 and PROPERTY 2 tests without the direction information. Accordingly, in the method developed below, we temporarily relax the directionality requirements associated with the links attached to each relation node in order to effect a first level search; direction tests would be applied as a further sift, if and as needed. Preliminary analysis of this approach indicates that there is no loss related to the PROPERTY 1 test. For the PROPERTY 2 test, no correct FactCG would be missed, but it will be possible for a FactCG that does not satisfy the query to be selected (e.g., where Comment 3 applies). The latter FactCGs would have to be eliminated via a 2nd level operation.

This observation about direction information is an important one.
Under the assumption that direction information is not needed, the reduced matrix representation introduced in the previous section may be used, with only +1 connection entries. Recall, this R-C connection matrix is significantly smaller than the full matrix representation.

We assume that all the conceptual graphs created by the knowledge engineers (FactCGs) are stored, perhaps in linear form, and that they are accessible to the data-base query program (the problem context for this presentation). The tack taken here is to encode each of the FactCGs with the vector representation (smaller still from the R-C connection matrix) introduced in the previous section. The operations for the PROJECTION test are to be performed using the vector representations. A certain amount of discrimination power in the tests is lost in these reductions. Although this could result in FactCGs being selected which do not satisfy the query, more importantly, all correct FactCGs should be selected. Thus, the operations based on the vector representation will select a (presumably) small subset of the FactCGs as candidate answers to the query, and a refinement kind of check will then be done on these candidates, perhaps using the complete conceptual graph (linear form?), as is currently done in other implementations. The payoff for allocating the extra memory required for storing the vector (partial) representations is increased speed of processing.

We begin by creating two vector templates: an R-vector template and a C-vector template. Each position of the R-vector template corresponds to one of the relations in the knowledge base catalog, and each position of the C-vector template corresponds to one of the concept types in the knowledge base catalog. Thus, the dimension of each of these vector templates will be NumR and NumC, respectively.

An R-vector is constructed for each FactCG as follows: a 1 is entered into the R-vector template at the positions corresponding to each relation type appearing in the FactCG. Similarly, a C-vector is constructed for each FactCG: a 1 is entered into the C-vector template at the positions corresponding to each concept type appearing in the FactCG. In addition, an R-vector and a C-vector are constructed for the QueryCG.

In a sense, the operations to be proposed with these vectors are akin to a "key word search," where the key words used in the search are the relations and concept types included in the QueryCG. The search begins with a PROPERTY-1 test to find those FactCGs that have the same relation types as does the QueryCG (this test simultaneously uses all the relations in QueryCG as "key words"). Next, the PROPERTY-2 test is used on the FactCGs which passed the PROPERTY-1 test, and finds those that have the same concept types, or sub-types thereof, as does the QueryCG (this test simultaneously uses all the concept types in QueryCG as "key words"). Following this, a CONFORMITY check is
performed (discussed in the next Section) as appropriate on the FactCGs that pass the PROPERTY-2 test. Finally, fetch the full representation of the FactCGs that pass the CONFORMITY check, and do more refined checks and manipulations as needed for the given query, and as appropriate to the specific implementation. [In practice, an implementation could be designed to do the concept "key word" check on a FactCG as soon as it passes the relation "key word" test. Hardware and time considerations would be the deciding factors. For applications where the first fact matching with the query is acceptable output, this approach would be preferable.]

The test for PROPERTY 1 is particularly simple:

Have three hardware registers whose length is at least numR. Create an R-vector for the QueryCG (call it Query-R-Vector), and store it in one of the registers. Then, sequentially bring into one of the other registers the R-vector for each of the FactCGs (call them Fact-R-vectors). For each one, perform the following test: Fact-R-vector $\geq$ Query-R-Vector, on a bit by bit basis. Enter a 1 in the third register for each bit that fails the test. If at least one bit is set to 1 in the third register (this can be checked in hardware via a large OR gate), the FactCG fails the test, so go on to the next FactCG. See Figure 7. If no bits in the third register are set to 1, then the current FactCG may be tested for PROPERTY 2. Depending on the implementation, the FactCGs that pass the PROPERTY 1 test may be stockpiled in an appropriate place in memory for the next test (PROPERTY 2), or, the next test could be performed immediately.

The reader may imagine that the upper graph of Figure 6 is represented via 1's entered in the slots corresponding to (AGNT) and (PTNT) in the 2nd register from the left in Figure 7, and the lower graph of Figure 6 is represented via 1's entered in the slots corresponding to (AGNT), (PTNT) and (COLR) in the leftmost register. The $\geq$ test will be passed for each slot, since for each 1 in the 2nd register there is a 1 in the corresponding slot of the 1st register. Thus the answer register will contain all zeros, and the FactCG will be shown to possess PROPERTY 1.

The above is all that is needed to check for PROPERTY 1. The R-vector indicates which of the relations are used at least once in the corresponding conceptual graph. Thus, the simple $\geq$ test suggested above accomplishes the "at least one" criterion of PROPERTY 1. Because of Comments 1 & 2, it is not needed to check the specific number of times the relation appears in the compared conceptual graphs.

The test for PROPERTY 2 is a bit more complicated than the one shown above for PROPERTY 1. This is occasioned by the fact that a
Figure 7. Parallel implementation schema for PROPERTY-1 test.

SUBtype of the concepts is allowed (this is where the restriction comes in) in contrast to the equality of relations required for PROPERTY 1. To check for the existence of a subtype relationship between a pair of concept types, the concept-type hierarchy will have to be accessed. A possible implementation for this test using standard parallel registers (similar to the configuration shown in Figure 7) is described in the next paragraph. This method would be straightforward, but entails a huge memory requirement. Circumventing the memory requirement, as well as additional speed enhancement, are direct motivations for pursuing the neural network possibility instead. This will be the topic in Section 3b, wherein we take advantage of the C-vector representation schema we have developed, and use these as inputs to neural networks.

For completeness, we mention a possibility using conventional parallel registers to effect a PROPERTY-2 test. This method consists of creating a sub-hierarchy vector whose template vector is the same as that for the C-vector (i.e., is length numC). One of these vectors would be created for each concept. A 1 would be entered in the slot for the
given concept, and, additional 1's would be placed in the slots for all the concepts which are sub-types of the given concept—i.e., all those concept types in the sub-hierarchy whose root is the given concept type. Then, to determine whether concept A is a sub-type of concept B, simply look in the sub-hierarchy vector for concept B at the slot corresponding to concept A and see if there is a 1 there. If so, then concept A is a sub-type of concept B; if not, then the sub-type relation is not satisfied. This test could be performed with a hardware configuration similar to that shown in Figure 7. For this test, the registers would have to have numC slots, and the test hardware would only need to perform the AND function, bit by bit. Nevertheless, this method would require that there be stored numC sub-hierarchy vectors, each of which is numC long, and this entails a storage requirement of numC^2 bits. If numC is on the order of 10,000, the numbers get quite large.

We have demonstrated so far that it is possible to implement the PROPERTY-1 and the PROPERTY-2 tests for the PROJECTION operation using standard parallel architectures. The suggestions so far are implementable with current VLSI technology. However, as indicated earlier, our objective in this chapter is to explore the possibility for implementing these operations via a neural network. We went through the above analysis to give a concrete demonstration to ourselves that 1) the desired operations on conceptual graphs are in fact doable using the chosen representation schema, and 2) the operations are implementable in a straightforward way, so our confidence in setting out to train neural networks to do the same operations is enhanced. One might ask why use a neural network at all, since the implementation is already so straightforward? Researchers in neural networks are demonstrating that a number of attributes accrue to connectionist networks that make them appealing. Fault tolerance is a key one of these attributes, and this accrues from the distributed representation nature of connectionist networks. Because the information is stored in a distributed way throughout the electronic (or other) implementation, even if a few percent of the internal devices fail, the impact on the overall operation can be negligible. In local representation devices, failure of internal components means total absence of the pieces of data that were stored therein. Also, for accesses to the concept hierarchy (re. the PROPERTY-2 test), there will be significant speed and memory size advantages on the side of the neural networks vs. parallel hardware added to serial machines.
XX.3b NEURAL NETWORK for TEST of PROJECTION PROPERTY 2

In the previous Section, we explored using conventional parallel registers to implement the PROPERTY-1 and PROPERTY-2 tests for the PROJECTION operation. In this Section, we proceed to explore implementing the PROPERTY-2 test using connectionist (neural) networks. The best situation would be if a neural network could perform the entire PROPERTY-2 test in one step. A procedure is described for using a neural network with this capability assumed. In addition, however, two other procedures are given, each using a neural network of successively less capability. For even the lowest assumed capability, it is argued that use of neural networks to perform the suggested operations on conceptual graphs can still effect a significant reduction of processing time (due to the potentially large number of accesses to the type/subtype checking operation to be discussed later, and, whereas the time for this operation in serial machines will depend upon the depth of the concept tree, in neural networks the operation will always be done in one "chunk" of processing time).

For the remainder of our discussion, we may assume a kind of neural network "black box." Referring to Figure 8, the connectionist network box is to have 2 groups of numC (the number of concept types in the knowledge base catalog) input terminals. Each of the terminals in the first group will be assigned to one of the concept types, and a similar assignment given to the terminals in the second group. The assignment of concept types to these terminals is made to correspond in a useful way to the assignment of concept types to the slots in the C-vector template. Having two groups of input terminals allows

![Diagram](image_url)

Figure 8. Neural network is trained to accept two C-vectors (or, RC-vectors), and to compare the two via the concept-type hierarchy.)
presenting concepts from two conceptual graphs as ordered pairs, and asking about their relative level on the hierarchy. The output of the network may be as simple as four wires; one each for the answers: lower, same, higher, not comparable (i.e., on a different branch of the tree).

What specific neural network configuration should be inside the box is a research issue of its own. Further, what training algorithm should be used is also a topic for research. The larger the number of concepts in the knowledge base catalog, the more input nodes will be required. The number of input nodes has direct impact on the amount of time it takes to train a network, and an important research topic will be to develop methods of modularizing the present problem so the train time scales reasonably.

For the application here contemplated, once the catalog of concept types and the associated concept-type hierarchy for a knowledge base is developed, the neural network could be trained and used without subsequent modification. A practical issue, however, is what happens when one more concept is added to the catalog? Does the training algorithm allow for simple addition of one more piece of data with its relationships to the other data, or does the training process have to start over from scratch? Difficulties of this nature have been experienced by others [Rumelhart, McClelland & Williams, 1986], as well as in the experiments to be described below. The author's experiments to date, however, have suggested ways to recast the problem definition in a way that will (hopefully) allow training (sub) neural networks to learn portions of the concept-type hierarchy lattice, and put these together as modules in a composite neural network. The implications of this refined method to reducing training time are potentially significant.

From the point of view of the present development, the training problem, per se, is not the focus; rather, the focus is to argue the plausibility that a neural network can implement the desired operations on conceptual graphs. For this purpose, we assume that a neural network "black box" can be developed, and proceed to consider if and how it could be used to effect the PROPERTY 2 test for the PROJECTION operation in the data-base context.

We assume that information about which FactCGs passed the PROPERTY 1 test is available, and that only these FactCGs will be dealt with in this pass. From the overall process point of view, it would be easiest if we were able to present the C-vector of the QueryCG (call it Query-C-vector) to the first group of input nodes (cf. Figure 8), and the C-vector of the current FactCG (Fact-C-vector) to the second group of input nodes, and then have the network (trained to) answer the following question: for each concept type with a 1 input in the first group, does there correspond at least one subtype of it with a 1 input in the second group? If the network could be trained to
accomplish this procedure, then it would only need two outputs: one for yes, and one for no. This would solve the PROPERTY 2 test directly. But the likelihood of developing a connectionist network to solve as complex a problem as this is no doubt low at the present time. Even if the training-algorithm difficulties were to be solved for this kind of application, designing the set of inputs to be used for training the network so it is possible for the network to infer the task we want it to perform remains as an important problem. These are all subjects for research.

Since it is presently unknown whether a neural network could learn to do the entire PROPERTY-2 test directly, we explore below a more modest function for the network to learn. To accommodate this more modest function, however, a number of supporting sub-procedures need to be designed by us for the rest of the system to perform.

The more modest task we propose for the neural network is as follows (cf. Figure 8): present to the first group of input nodes one of the concept types which appears in the QueryCG (i.e., one of the active slots in the Query-C-vector), and to the second group of input nodes, present one of the concept types which appears in the current FactCG (i.e., one of the active slots in the Fact-C-vector). Then, for this pair of activated terminals, have the neural network answer the (simpler) question regarding the location of the second concept type on the hierarchy relative to the first (e.g., lower, same, higher, not comparable). The author has obtained positive results on this (sub)task. There are a number of degrees of complexity in testing for PROPERTY 2 between the simple test suggested in this paragraph, and the much more complex one of testing for PROPERTY 2 directly. An example of a test between the two extremes so far suggested is to present to the first group of input nodes one of the concept types which appears in the QueryCG (as in the simplest test), and to the second group of input nodes, present the entire C-vector of the current FactCG (as in the most complex test), and then have the neural network answer the question whether there is at least one instance of the query concept type, or one of its subtypes, in the FactCG. After the "simplest" test has been shown to be doable by neural networks, a progression of tests leading to the direct test could be researched. So far, though it now has been demonstrated that the simplest task can be performed by a neural network, a lot of collateral design experiments are required before proceeding with the progression of experiments suggested in the previous sentence.

Returning to the simplest test, one possibility for implementation is as follows:

Assume we have a neural network box which has two groups of input terminals equivalent to two C-vector templates, \( C_1 \) and \( C_2 \) [i.e., \( 2 \times (\text{numC}) \) inputs]. The rules for input are that one slot in each of
the C-vectors is to be active at any one time, and the output will have four terminals to represent the following (cf. Figure 8):

The active slot of C₂ is a restriction of the active slot of C₁ (<)

The active slot of C₂ is the same type as the active slot of C₁ (=)

The active slot of C₂ is a generalization of the active slot of C₁ (>)

The active slot of C₂ is not comparable to the active slot of C₁ (@)

Recall our definition of the R-C connection matrix, and refer again to Figure 11 (in Section 4): each of the rows corresponds to one of the relations in the knowledge base catalog, and each of the columns corresponds to one of the concept types. A 1 is entered in those slots where there exists a relation node and concept node in the network, and there is a connection between them. This will be a very sparse matrix, because each row will typically have only 2-3 entries, and based on reported experience, with a maximum of some 5 entries (even when the number of columns is very large). Also, the number of relations used in a given FactCG is typically a small subset of numR.

Recall also, that the R-vector is derived by "projecting" the rows of the R-C connection matrix into a column vector whose length is the same as that of the matrix (numR). A 1 is entered into a slot of the R-vector when the corresponding relation exists in the graph (and therefore there is at least one 1 entry in the corresponding row of the matrix). Thus, the "active" slots (those with a 1 in them) of the R-vector indicate which relations are included in the graph, and simultaneously, which rows in the R-C connection matrix are non-zero. There are a number of ways to efficiently encode sparse matrixes; however, to keep the graphical presentation of the present development a little more obvious, the active slots in the R-vector will be our indicator for the rows being stored, and we will keep the rows stored in their entirety. Thus, we will say that associated with each active slot in the R-vector for a conceptual graph, is an RC-vector which shows the connections of the associated relation node to the concept nodes of the graph. This RC-vector is simply the corresponding row of the R-C connection matrix. The length of the RC-vector is thus numC.

By storing the R-vector and associated RC-vectors for a conceptual graph, we store only those rows of the R-C connection matrix that have at least one 1 in them, plus the R-vector. The rows are stored as an ordered set, and the R-vector plays the role of key for the RC-vectors, and in addition, the important role of facilitating a fast test for PROPERTY 1 in a data base query process using the PROJECTION operation.

Recall also, that the C-vector is derived by "projecting" the columns of the R-C connection matrix into a row vector whose length is the same as the width of the matrix--i.e., numC. The difference between the C-vector and the RC-vectors is that the C-vector has active slots for all the concept types used in the associated graph, whereas the
RC-vectors each have active slots only for those concept types connected to the specified relation node of the graph. The C-vector is the union of all the RC-vectors for a graph, i.e., the union of all rows in the R-C connection matrix for the graph.

At the present stage of the author's research it is not clear whether the proposed procedure for simply comparing the C-vector of the QueryCG with the C-vectors of the FactCGs to test for PROPERTY 2 will be efficient in actual practice. It was pointed out earlier that it is possible for FactCGs that do not satisfy the query QueryCG to pass the PROPERTY-2 test. Determination of the quantity of such false responses can only be made empirically, as they will depend upon the characteristics of the particular knowledge base. For the purpose of exploring the various possibilities, assume that the method would not be acceptable, and consider the implementation consequences. First of all, recourse would have to be made to the RC-vectors, and this entails a requirement for considerably more memory. On the other hand, the subtype (restriction) checking needed for PROPERTY 2 will no doubt be faster for the RC-vectors (since fewer possibilities need to be cycled through) than it would be for the C-vector, so it could in fact turn out being worth the extra allocation of memory to gain the speed advantage. Let us assume for the rest of the development that the RC-vectors will be required, and determine the associated process.

Since the FactCGs we will be testing for PROPERTY 2 have already passed the test for PROPERTY 1, we are guaranteed that for every active slot in the Query-R-vector, the same slot in the Fact-R-vector will also be active--i.e., if a relation node of a given type exists in QueryCG, then one also exists in the FactCG. We fetch the pair of RC-vectors corresponding, respectively, to a selected active slot in the Query-R-vector and to the same slot in the current Fact-R-vector being tested for PROPERTY-2. The entries in these two RC-vectors indicate, respectively, the concept nodes connected to the specified relation node in the QueryCG and to those connected to the specified relation node in the FactCG. In practice, each of these RC-vectors will typically have 2-3 non-zero entries, and at most 5. For each of these entries in the RC-vector for the QueryCG, we are to determine if there is an entry in the FactCG RC-vector which is a subtype. The neural network box previously shown in Figure 8 will be used to make this test.

Since we assume there will normally be only 2-3 entries in each RC-vector, it may be possible for a neural network to perform a direct test on the RC-vectors for PROPERTY 2 (in contrast to our earlier speculation that this is unlikely for the C-vectors, which potentially have many more entries). If so, then a procedure such as the following would accomplish the PROPERTY 2 test for the candidate FactCG.

Sequentially apply to the neural network (NN) one of the RC-vectors
for the QueryCG and the corresponding RC-vector for the FactCG; if the NN indicates a lower- or equal- type relation, go to the next pair of RC-vectors; continue until all RC-vectors of the QueryCG are considered. If a higher or not-comparable response is given for any pair of RC-vectors, then because of part b) of PROPERTY-2, discard the present candidate FactCG and go on to the next one in the stockpile which passed part a) of the PROPERTY-2 test. Those FactCGs which pass this test are stockpiled in memory for the next part of the processing.

In serial machines, the subtype checks required for the PROPERTY 2 test would likely have to be done pairwise--i.e., separately testing each (active) concept in the QueryCG with each (active) concept in the FactCG. The number of steps for this process will depend on numC, and where in the hierarchy the particular concepts being tested lie. For the purpose of exploring the least-capability situation, let us assume that we assign only this bare-bones task to the neural network. Even in this case, the neural network approach will no doubt still be faster than serial searching of the hierarchy--especially for large values of numC. This is because, if a neural network can perform this check (preliminary experiments indicate that one could), it should do so in a constant increment of time, no matter where the concepts lie in the hierarchy tree.

The (bare-bones) task of making the subtype test on only one pair of entries at at a time could be accomplished with a connectionist network as follows:

Start with the query Query-R-vector. Go to its first active slot and fetch the associated RC-vector. Fetch the corresponding RC-vector for the current FactCG (via the Fact-R-vector). The assumption here is that only one of the entries in each of the RC-vectors can be tested at a time. Going from left to right in the Query-RC-vector, find the first active slot. Pick the same corresponding slot in the candidate Fact-RC-Vector. (Remember, there are typically 2-3, with max 5, active slots in the RC-Vectors.) If both slots have a 1 in them, then there is a type match. If they do not, then, keeping the same slot in the Query-RC-vector, find another* active slot in the Fact-RC-vector. Present this pair of slot positions to the neural net.

* The strategy will be determined by how the concepts are ordered in the RC-vector. E.g., if we always have subtypes of a given concept type assigned to slots to the right of the slot for that given type, then the word "another" could be replaced by "next". Generally, this implies building a kind of meta-knowledge into the hardware rather than having to incorporate it into the software control (thus, the "work" has only to be done once).
(NN) and determine if they pass the test. If not, repeat the process. Continue until find a pair that passes the subtype check, or, until the end of the Fact-RC-vector is reached (if this happens, then this FactCG fails the test, so go on to next FactCG to be checked.) When the subtype check is passed for a given slot in the Query-RC-vector, then find its next active slot, and repeat the process. Continue through all the active slots in the Query-RC-vector (at most 5 of these). If successful for each of these, then go to the next active slot in the Query-RC-vector, fetch the associated RC-vector, and repeat the process above. Continue this process until cover all active slots in the query Query-R-vector. If successful for each of these, then the current FactCG is a candidate answer to the query. If fail test at any point in the process, then discard the current FactCG and go to the next one.

There are many details that will have to be tended to when developing an actual implementation of the above. The purpose here was to demonstrate how a neural network could be used to perform some of the conceptual graph processing in constant time, and hence speed up the process by a potentially significant factor. The more of the tasks the connectionist network could be trained to do, the greater the potential speed up. An important topic of research is to determine which level of tasks a neural network can be trained to do in practical contexts within a reasonable amount of time. With the rapid advancements occurring in the hardware arena, the limitations are not likely to come from this aspect. Rather, as indicated earlier, the development which will make the largest step toward practical application of the connectionist approach will be creation of training strategies that will converge in a reasonable amount of time—and just as important, will require short update times, for adding new data after the initial training is done.

One aspect that has not been given attention yet here is the case where a concept node has an individual marker assigned to it—i.e., the concept has been restricted to an individual. An example might be [person:Irene] or [man:Gregory]. This becomes important, for example, when in the process of testing PROPERTY 2 it is determined that a concept node of the FactCG is lower on the hierarchy tree than the corresponding concept node of the QueryCG. If this FactCG concept node has an individual marker assigned to it, a CONFORMITY test is required before performing the RESTRICTION.

As stated before, we assume a table is available which contains a list of all markers (names) used for individuals, and with each marker there is listed the concept subtype to which the marker normally applies.

Let us go through an example (cf. Figure 9). Suppose we are comparing concept node [girl] with the restricted concept node
Figure 9. Names in the marker list include the concept-type referent for each marker. These are used to accomplish the CONFORMITY check. A marker can only be assigned to its referent concept type or to a concept type above or below it in the hierarchy. With the drawing as a visual aid, it is easy to see that the name Irene should not be assigned to the concept type boy (the referent concept type for Irene is said to not conform to the concept type boy). A suggested process for determining presence or absence of conformity via a neural network is given in the text.

[person:Irene] during the PROPERTY-2 test, and want to know if the RESTRICTION to [girl:Irene] is legitimate. A restriction is 'legitimate' if the referent of the token CONFORMS to the concept being restricted; with respect to the hierarchy tree, this means that the token's referent concept type is on a common branch of the tree with the concept to be restricted. We assume that when the FactCG containing the [person:Irene] concept node was originally created for the knowledge base, a check was made that it was legitimate to restrict [person] via the Irene marker (whose referent is [female-person]). With this assumption, it is only necessary to check that [girl] CONFORMS to [female-person]. If we have a neural network such as the one in Figure 8, this test if very simple: checking [girl] and [female-person], the network yields the "<" output signal (i.e., "lower, in the same branch of the tree"). On the other hand, if we had been comparing [boy] with [person:Irene], and wanted to know if the RESTRICTION [boy:Irene] is legitimate, when we check [boy] with [female-person], the neural network yields the "not comparable" output signal--i.e., [boy] is on a different branch of the tree. In this case, CONFORMITY fails, and we cannot make the RESTRICTION [boy:Irene].

In the above example, if we were not willing to make the assumption that the FactCG restriction was properly checked, then the test here must include this test as well. The procedure would be as follows. We check [girl] with [person] via the neural network of Figure 8, and find that [person] > [girl]. Now take into account that
[person] is restricted to Irene; look up Irene in the marker list, and note that this marker has [female-person] as its referent. Check [female-person] with [girl] via the neural network to find that [female-person] > [girl]. Therefore, CONFORMITY holds, and [girl:Irene] is ok. On the other hand, suppose we are comparing [boy] with [person:Irene] and want to know if [boy:Irene] is legitimate. When we check [boy] with [person] via the neural network, we find that [person] > [boy]. As before, we now take into account that [person] is restricted to Irene; we look up Irene in the marker list, and again note that this marker has [female-person] as its referent. This time, when we check [female-person] with [boy] via the neural network, we find that [female-person] is not comparable to [boy]--i.e., [boy] is on a different branch of the tree. Therefore, CONFORMITY check fails, and we cannot make the RESTRICTION [boy:Irene].

This example demonstrates the potentially large number of accesses to the type/subtype checking operation that could be needed to perform the RESTRICTION & CONFORMITY check operations. As indicated before, the neural network makes the type/subtype operation possible in constant time, no matter where in the hierarchy the pair being checked falls. Given such a potential time savings for each type/subtype check, the cumulative reduction of throughput time could be truly significant.

XX.3c EARLY EXPERIMENTS TO TRAIN NEURAL NETWORKS ON CONCEPT SUB-TYPE TEST

In the previous Section, we assumed that a neural network "black box" could be developed to perform the concept sub-type test, and then proceeded to demonstrate that such a NN black box could be used to effect the PROPERTY-2 test for the PROJECTION operation in the data-base context. In this Section, we mention some early experiments which do indeed demonstrate that the concept sub-type test can be done by a neural network.

From a theoretical point of view, a concept-type hierarchy is a lattice. Accordingly, the experimental procedure was to define an abstract lattice, and use this as a proxy for a concept-type hierarchy. The lattice used is shown in Figure 10. This lattice has 16 nodes, and was designed to provide paths of different length and complexity. The graphical representation of the data structure shown in Figure 10 contains within it the answers to all possible questions "What relation is concept Ci to concept Cj?" (Notation: [Ci ?R Cj]). This is obvious to each of us human observers. The key experimental question is how do we get this information inside the neural network, and such that the NN can provide correct answers for every possible [Ci ?R Cj] question?
Figure 10. Sample lattice for concept-type hierarchy experiments.

An important sub-problem here turns out being how to design the set of data to be used in training the NN. To answer questions of the type [Ci R Cj], we need to have more information than that represented directly in the Connection Matrix. Consider Figure 10 again. By visual inspection, it is possible to state that C3 > C14, but we must also note that the path from node 3 to node 14 contains three links (path length = 3). It is not immediately obvious how (in general) to determine that C3 > C14 directly from the Connection Matrix. Fortunately, there exists a technique for creating a Reachability Matrix of a graph [Warfield, 1976], where the Reachability Matrix contains a 1 in the (i,j) position if a directed path of any length exists between node i and node j. The Reachability Matrix is calculated from the Connection Matrix via an algorithm that involves raising a modified version of the Connection Matrix to increasing powers until some stopping condition is met. In the experiments carried out to date, the Reachability Matrix has been found to be useful for an organized development of the training data sets.

For these experiments, a neural network of the "backpropagation" paradigm was used. With reference to Figure 8, the NN was provided with 2 groups of 16 inputs. Each group of 16 inputs is fully interconnected (feedforward only) to a separate group of 4 neurodes, these two groups of 4 constituting a layer of 8 neurodes. The outputs of these 8 neurodes are fully connected to a next layer of 8 neurodes, and finally, these 8 are fully connected to an output layer of 4 neurodes. This configuration has what is known as two "hidden" layers.

There are 16x16 = 256 possible [C1 R C2] questions of the type described earlier for the given lattice. A training set consisting of 1 each of these questions was created, but it was decided not to use the
[Ci \neq Cj] question, because this is trivial to answer prior to applying the NN to the task. The NN learned to answer all the remaining 240 questions perfectly. Thus, the in-principle question of whether a NN can be trained to answer the concept sub-type question is answered in the affirmative.

As a knowledge base is being developed, it could happen that the underlying lattice of concept types expands over time. It is important to determine if the NN can be made to learn this knowledge in a piecemeal fashion, i.e., to train the NN to incorporate the new concept types as they evolve in the lattice (i.e., an updating procedure). An experiment was fashioned that trained the top 4 nodes first, then nodes (5,6,9,10,14), and then the remaining nodes. As one might expect, one cannot simply train the net with only the new nodes at each stage; the net "forgets" what it learned in the first stage if it is not "reminded" once in a while what it learned before. Empirically, it was found that a reminder rate of about 1 in 4 served to maintain the old knowledge while the new was being trained. The negative result of this part of the experiment was that it took about 100,000 presentations to learn each stage of training--therefore, it took approximately 300,000 presentations to learn the lattice via staged training whereas, it only took about 100,000 to learn the whole lattice from scratch.

A more circumscribing set of experiments is getting underway to explore the above and related phenomena further. In particular, ideas are emerging for training sub-NNs to learn sub-lattices of the larger lattice, including special sub-lattices which represent the linkages between the other sub-lattices, and joining the sub-NNs together appropriately into a composite NN. These ideas are in their developmental stages, and nothing is ready to be reported yet.

**XX.4 HANDLING MULTIPLE COPIES OF RELATIONS AND CONCEPT TYPES**

An aspect that was mentioned earlier, but not yet addressed, is the case where a conceptual graph contains more than one copy of a relation or of a concept type. To accommodate these, the R-C connection matrix, the R-vector, the C-vector, and the RC-vectors need to be augmented.

The main requirement we must take into account in augmenting the representation schema, is that it remain standardized. For use in neural networks, it is important that the R-vector, C-vector, and RC-vectors (non-zero rows of the R-C connection matrix), have a fixed assignment of relation/concept types to the slot positions. If it weren't for this latter constraint, we could simply add extra rows for the additional relation nodes, and extra columns for the extra concept nodes. Each of the extra rows and columns would be labeled, and in
symbol processing machines, the labels can be used to keep track of everything. BUT, for the neural network application, the position is the label, and once the network is trained, the terminals each represent a specific role in the problem context.

Consider first the case where there are multiple copies of relations, but not of concept types. We define an R-duplication-vector template. This template has the same number of slots and the same relation assignments to the slots as does the R-vector template. The R-duplication-vector for a conceptual graph (CG) is made as follows: if a relation occurs more than once in the CG, the excess number is entered in the corresponding slot of the R-duplication-vector template. Refer to Figure 11. From an implementation point of view, use of 2-bit subregisters for the duplication vectors allows up to 4 occurrences of a given relation; with a 3-bit subregister, 8 occurrences of a relation are allowed (anecdotal reports based on research implementations indicate that this is probably an extreme upper limit for typical uses). In practice, there will likely be very few entries in the duplication vectors, so it will probably be more efficient to carry this data in a list rather than via a full vector representation. But for graphic purposes in the present development, and for potential use with neural networks, we will continue to use the vector representation.

For each extra copy of a relation in the conceptual graph, create a new RC-vector (this represents connections of the given duplicate relation node to concept nodes). In Figure 11, these new RC-vectors

![Figure 11. Relation-Concept Connection Matrix (R-C connection matrix). Rows in matrix are called RC-vectors.](image-url)
are shown stock-piled in the 3rd dimension so the number of rows in the R-C connection matrix is not changed. Whatever scheme is created in an implementation to store/fetch RC-vectors pertaining to the active slots in the R-vector can be used to store/fetch the RC-vectors created for the duplicate nodes, only using the R-duplication-vector as the key.

Thus, what we have is an additional set of RC-vectors (corresponding to the duplicated relations), and these can be handled in the same way as the other RC-vectors for operations discussed to this point. It is important to keep in mind that the dimensions of the R-vector template or RC-vector template did not change in this part of the augmentation.

Duplication of concept types in a conceptual graph presents a greater difficulty. It seems we are stuck with having to add columns to the connection matrix to allow for representing the duplicated concept types. The classical AI example of an arch [Sowa, 1984, p. 71] provides a case in point. One portion of the conceptual graph for an arch is [brick] -> (right) -> [brick]. Here we have the relation (right) connected to two distinct nodes of type [brick]. The RC-vector for the relation (right) must be able to show two connections, and to do this, there need to be two slots in the RC-vector template, both of which correspond to separate instances of the concept [brick]. Our augmentation will thus have to provide a means for adding a new slot to the RC-vector template (and therefore, a new column to the R-C connection matrix and, of course, a new entry for all the RC-vectors—this latter is important because other relations could connect, separately, to these two instances of [brick] as well.)

If we were doing this by hand, we would probably insert a second column for the concept [brick] next to the original one in the R-C connection matrix. Then, in doing the PROJECTION tests by hand, we would have visual access to the labels for the columns to help us keep track of the operations. However, since we need to have position invariance for the concept types, we explore another alternative.

Create a C-duplication-vector in the same way as the R-duplication-vector was described above. This entails placing a number in a slot of a C-vector-like template to indicate the extra number of copies of the associated concept type which appear in the graph. Do this with the original definition of the C-vector template. We will add extra slots to the RC-vector template (and therefore extra columns to the R-C connection matrix) according to information contained in the C-duplication-vector. The additions are, therefore, tailored to each conceptual graph. Even though there will be slots added to the RC-vector template, the C-vector template will stay fixed.

To accomplish position invariance, we propose the following schema. Moving from left to right in the C-duplication-vector, as we encounter a non-zero entry, read the number in the slot, and add that
many slots on the right end of the RC-vector template. These slots will thereafter represent (for the given conceptual graph) the concept type associated with the active slot in the C-duplication-vector. Repeat this addition process for each non-zero entry in the C-duplication-vector, as we traverse it from left to right. The concept type represented by each of the new slots in the RC-vector template has a position referent in the C-duplication-vector, and the position referent for different graphs is made unique by using a fixed procedure (left-to-right, in this case) in assigning the new slots according to the entries in the C-duplication-vector.

There are a number of intricacies that will have to be attended to for implementing the above, but it should be conceptually clear that the PROPERTY 2 tests can be performed with this representation schema. A simple modification in the next-to-last sentence in the procedure given earlier: viz., "If a higher or not-comparable response is given for any pair of RC-vectors, then bring in the connection row for each duplicated relation and perform the same test. If all of these give higher or not-comparable responses, then, because of part b) of PROPERTY 2, discard the present candidate FactCG and go on...." In the context of implementing this in a serial machine augmented with parallel registers as suggested earlier, the main consideration will be how many additional slots to make available in the registers that are to accommodate the augmented RC-vectors. There is no apparent theoretical basis for making this determination—rather, the number will have to be determined empirically. The knowledge base to be implemented will have to be studied to determine a typical maximum number of duplications of concept types that occur in the FactCGs—perhaps forfeiting certain extreme cases. This portion of the register will be used only when there are duplications occurring in the conceptual graph. Perhaps this portion of the register can be dubbed 'duplication reservoir.'

The next important question for us to consider is, can this means of representation be used with connectionist machines?

Let us provide the connectionist network with two additional sets of numC input terminals (see Figure 12) to accommodate the C-duplication-vector templates (for the QueryCG and the FactCG) plus enough extra terminals in the two sets of terminals associated with the RC-vector templates to accommodate the size of the duplication reservoir. Up to this point in our discussions, we have used only binary inputs for the connectionist networks. The C-duplication-vectors will be allowed to give input values appropriate for indicating the number of duplications. With this kind of hardware configuration, the network still learns what it did before with respect to the first numC slots of the RC-vector templates. The meaning of the remaining terminals in the (augmented) RC-vector groups is
Figure 12. Input terminals added to network shown in Figure 8 to accommodate multiple copies of concept-type node. So far, all inputs to the neural networks have been binary valued. The C-duplication vectors are here allowed to have values > 1, to indicate number of duplications.

defined by entries in the C-duplication-vectors. As before, we have two sets of all the terminals, so the network can be given the QueryCG and a FactCG upon which to perform the PROPERTY-2 test (row at a time from their respective R-C connection matrixes.) This means of accommodating multiple copies of concept types is expensive—in the sense that there is more than a doubling of input terminals. A more efficient encoding of the duplications is another item for research. Another possible approach to this issue, however, is implicit in research reported by Touretsky & Hinton [1985] wherein they show that connectionist networks are capable of dealing with linked lists. This is one way of looking at what we have here—and represents another item for research.

This concludes a first pass at considering the usability of the proposed matrix-form representation and its associated vector-form representations to carry out a database query task via the PROJECTION operation. All the major functions that need to be done are seen to be doable in principle. Further, suggestions for implementing these in serial machines augmented with parallel registers for the key operations have been provided. And most pertinent to the purpose of this chapter, subtasks of the database query task have been defined which appear to be plausible candidates for implementation with neural networks.
XX.5 PERFORMING MAXIMAL JOIN OPERATION WITH PROPOSED PROCEDURES

In Sowa's notation chapter in this book, and in Section 3.5 of his 1984 book, Sowa stresses the importance of the operation called MAXIMAL JOIN. Its importance lies in its being analogous to similar operations in a variety of other AI systems, as well as being the basis for the applications presented in Sections 3.6, 4.1, 4.7, 5.6, and 6.5 in [Sowa, 1984].

The one diminishing attribute of MAXIMAL JOIN cited by Sowa in Section 2.4, is that "...maximal join requires a nondeterministic search for nuclei before the process can begin." This issue has not yet been researched for the representation schema proposed here, but it appears that the methods presented in the previous section for performing a database query via the PROJECTION operation would be applicable to the MAXIMAL JOIN operation with little modification. If this turns out to be true, then the difficulty mentioned by Sowa is largely overcome.

The definition for MAXIMAL JOIN given by Sowa is as follows: "The join rule allows two concepts to be joined at a time. By repeated joins, restricts, and simplifies, a simple join of two concepts may be extended to a much larger join of two subgraphs. If the join is extended as far as possible, it is called a maximal join."

In Section 3 (of this chapter), it was shown that the JOIN operation is accomplished via a logical-or operation on the cells of the connection matrices of the two graphs being operated on. It was not emphasized there, but via this operation, all concepts which appear in both CGs being joined are automatically merged in the one step. Thus, if we are given two conceptual graphs CG1 and CG2 to be (maximally) JOINed, then via the logical-or operation on the connection matrices of CG1 and CG2, the connection matrix for the resultant graph CG3 contains simple joins of all the concept nodes that appeared in both CG1 and CG2. This is a major step in the process of accomplishing a MAXIMAL JOIN. (Caution: a check for restriction to individuals will have to be made on each of the concept node pairs before they are merged. Also, details remain to be worked out regarding duplicate relations.) Regarding a potential SIMPLIFY operation in CG3, by definition, there must have existed common relations in CG1 and CG2. Existence of such common relation nodes can be checked easily via the R-vectors for the two CGs. The test would be similar to the PROPERTY 1 test for the PROJECTION operation, except that an AND of the two R-vectors would be performed instead of the ≤ test. Via such a (modified) test, the only candidate relations for a SIMPLIFY in CG3 would be those slots in the answer R-vector register containing a 1 (cf. Figure 7). A SIMPLIFY can be done only if the two RC-vectors
associated with each pair of common relations identified are identical.
To make them identical, it could require a process of RESTRICTION
and CONFORMITY checks, etc., but this process is similar to the
PROPERTY-2 test for the PROJECTION operation.

Clearly, a more detailed analysis of the above will be needed before
definite conclusions can be drawn. However, if the PROPERTY 1 and
PROPERTY 2 tests can be adapted to the MAXIMAL JOIN process,
then—as is true for the PROJECTION operation—the process could be
speeded up significantly over those methods indicated in Sowa’s earlier
statements. Such a speed up could be effected primarily via
application of connectionist type devices, as the PROPERTY-2 test is
the more significant time consumer. But even if the connectionist
technology is slow in developing, VLSI technology is here, and special
purpose parallel equipment could be appended to serial machines for
performing the PROPERTY-1 tests (and the PROPERTY-2 tests if the
associated large memory requirement is acceptable) for a still-
significant improvement in speed.

XX.6 USING NEURAL NETWORKS AS MEMORY FOR
CONCEPTUAL GRAPHS

In the preceding discussions, the R-, R-duplication-, RC-, C-, and C-
duplication vectors were created as a means for representing conceptual
graphs. All of these, except the C-vector, are to be stored in memory,
and to be available for the PROPERTY-1 and PROPERTY-2 tests as
indicated. It has been assumed that these vectors will be available for
parallel transfer into either the registers of Figure 7 or the neural
network device of Figure 8. The actual physical type of memory
system to be used has not yet been discussed.

Except for the type/subtype portion of the PROPERTY-2 test, all
of the procedures discussed so far could be implemented via a serial
machine with access to certain special-purpose hardware (parallel
registers, per Figure 7). The purpose of this section is to mention the
possibility of using connectionist networks as memory for conceptual
graphs.

Discussion of this topic could occupy a chapter of its own; indeed,
others have spent the better part of entire books dealing with this issue
[e.g., Kohonen, 1980; Kohonen, 1984; Hinton & Anderson, 1981]. We
simply mention that parallel distributed memory systems have a
potential for manifesting properties of a "higher level" than properties
normally associated with simple memorizing devices. Descriptors that
give hint of such properties are: associative, content addressable,
inductive generalization, implicit knowledge, graceful degradation,
fault tolerance, etc.
The methods discussed in the preceding sections actually implement some of these attributes. For example, as described, the query process via PROJECTION can legitimately be considered an associative retrieval or content addressing method, depending on how one chooses to view the various procedures introduced. But these have more to do with the properties of conceptual graphs, the operations defined on them, and the representation schema suggested herein, rather than on the neural network implementation.

There is ample evidence in the connectionist literature to indicate that vector patterns of the type defined above can be stored in connectionist networks. Further, there is evidence that associative and/or content addressing can be performed in such memories. Conceptual graph applications would serve a useful problem context for guiding research in using neural networks for memory. This is because the operations to be performed on conceptual graphs are becoming well defined for various types of applications (the application chapters in this book attest to this), so researchers will have definite processing requirements to shoot for in developing connectionist memories. It could turn out, for example, that some of the PROPERTY-1 and PROPERTY-2 tests could be done implicitly in such memories. If so, then, for example, a QueryCG could be specified for a query, represented appropriately, and presented to the input terminals of the connectionist memory. The output of this memory device could be, say, those FactCGs that have PROPERTY 1, and these are passed to the neural-network device that performs PROPERTY-2 tests. As the capabilities of these devices are improved, perhaps it is not too far fetched to expect that one day, the connectionist memory can be trained to do the entire process.

XX.7 CONCLUSION

Connectionism is a term that refers to massively parallel networks (of neural-type elements) in which information is stored mainly in parameters associated with connections of the elements rather than explicitly in the elements themselves. The connection parameters in such networks are determined during a process of "training" (or "learning") rather than by traditional design methods. There are a number of desirable properties that accrue to connectionist (neural) networks by virtue of their distributed nature (e.g., fault tolerance, associative properties, etc.), and these provide good reasons for pursuing research to bring such networks to reality. As researchers in the field are aware, significant issues remain to be resolved before neural networks can be used to implement large knowledge bases.

An objective of this chapter has been to demonstrate that
connectionist devices could meaningfully be used in implementing conceptual graphs, and indeed offer significant speed up of the two most important of the operations for conceptual graphs, namely, PROJECTION and MAXIMAL JOIN. These two operations extend into a wide range of AI and database type applications, so speeding them up by, potentially, several orders of magnitude can have important ramifications. In addition, it was suggested that a neural-network type memory system could prove important to conceptual-graph knowledge bases. To accomplish these glowing potentialities, however, substantive progress must be made in the realization of connectionist machines with large numbers of elements and interconnections, and specifically, in the development of training algorithms whose convergence time scales polynomially rather than exponentially. Whether this latter is a realistic hope or not, only time will tell--there are, however, tantalizing insights being offered that fuel the hope.

The state of the art in building knowledge bases with conceptual graphs is also in its early stages. Numbers currently associated with research knowledge bases are 70-100 relations, and 100-200 concept types. As we learn more about how to do it, more substantive contexts will be implemented, and for these, the numbers will change. Though the number of relations will likely stay on the order of 100-200, the number of concept types will more likely go over 1,000--perhaps to 10,000. Numbers of this magnitude will pose serious challenges for any kind of implementation, including neural networks.

It is known that successful development of neural network applications requires that careful attention will have to be given to analyzing and understanding the structure and constraints of the problem context as well as to those of the network. To improve the likelihood that the task to be performed is contained within the repertoire of functions achievable by the network (quite aside from the training issue), the researcher will have to undertake the specific activity of matching constraints implicit in the problem environment with those involved in the network structure.

For this reason, the author feels that research in conceptual graph applications and in connectionist machine development will both benefit by a strong interaction of the two. Requirements from each field can be used jointly as motivation for design of experiments and for the insightful thinking that will be required. The latter is a very necessary ingredient because, as progress is made in building solid state and/or optical hardware for implementing "simulated neurons," the biggest issue will continue to be: what do we do with them? We know a lot more now than we did in the 1960's, but there are still some quantum leaps for us to make.
XX.8 REFERENCES


---- 1989b, "Testing the Use of a Neural Network to Implement a Basic Data-base Function,"


RECENT BOOKS ON NEURAL NETWORKS

Caudill, M. & C. Butler (1990), NATURALLY INTELLIGENT SYSTEMS, MIT Press.
Hecht-Nielsen, R. (1990), NEUROCOMPUTING, Addison-Wesley.
Khanna, T. (1990), FOUNDATIONS OF NEURAL NETWORKS, Addison-Wesley.
Simpex, P.K. (1990), ARTIFICIAL NEURAL SYSTEMS, Pergamon.