Multi-Level Hierarchical Retrieval

Robert Levinson*
and
Gerard Ellis †

Abstract

As large databases of conceptual graphs are developed for complex domains, efficient retrieval techniques must be developed to manage the complexity of graph-matching while maintaining reasonable space requirements. This paper describes a novel method "the multi-level hierarchical retrieval method" that exploits redundancy to improve both space and execution time efficiency. The method involves search in multiple partially ordered (by "more-general-than") hierarchies such that search in a simpler hierarchy reduces the search time in the hierarchy of next complexity. The specific hierarchies used are: the traditional partial order over conceptual graphs; a partial order over node descriptors; a partial order over "descriptor units"; and finally, the simplest partial order is the traditional type hierarchy.

1 Introduction

Although it is true for many natural language applications that due to a wide variation of concept and relation types that matching conceptual graphs [14] is relatively easy (and hence retrieval is efficient), no such claims can be made for general conceptual graphs applied to other domains such as chemistry [16], chess[9,11], VLSI CAD designs, structural designs, etc. In these domains, since many concepts and relations may be repeated within the same graph, the cost of graph-matching becomes the dominant cost. As these databases grow, effort must be taken to maintain space and execution time efficiency. As is often the case, optimizations can be achieved by exploiting redundancy. In the case of the system described here redundancy is exploited in multiple ways. Redundant features of graphs need only be stored once and redundant portions of the graph-matching tests may also be shared. Further efficiency is achieved by taking advantage of the structure of the partial order by "more-general-than" to do even further pruning.

While retrieval in the traditional partial order graph hierarchy has been shown empirically to require comparisons on the order of the log of the number of graphs in the database [6,7], the multi-level hierarchical technique described in this paper should perform even better by reducing the effort to compute the required comparisons. Due to the additional complexity of the database structure these savings may be practically realizable for only large databases for the types of applications described above. For simpler applications the uni-hierarchical method should be sufficient.

In addition to providing potential gains in efficiency the multi-level hierarchy serves as a conceptual tool also: it gives a spectrum on which the graph hierarchy and type hierarchies can be seen as the two extremes. The multi-level system maintains multiple hierarchies of increasing complexity:

*Department of Computer and Information Sciences, University of California, Santa Cruz, CA 95064 U.S.A.
†Computer Science Department, The University of Queensland, Brisbane, QLD 4072, Australia.

1. The simplest partial order is the traditional type hierarchy.

2. A partial order of "descriptor units". Descriptor units are the atomic information used in the node descriptors, they reflect the actual path distance between concept and relations in the conceptual graph and imply the adjacency relationships.

3. A partial order of node descriptors. (where a node descriptor for some concept or relation is a description of the neighborhood of concepts and vertices radiating from that concept or relation for a given radius).

4. The traditional partial order of conceptual graphs.

Search in the simpler hierarchy provides a rapid mechanism of indexing and searching the hierarchy of next complexity. The details of these hierarchies, the new method of graph description (as sets of node descriptors), and the retrieval method that allows one to move from one hierarchy to the next are described in the remainder of the paper. Note that potentially exponential subgraph-isomorphism tests are being replaced with $O(n^3)$ graph comparisons where n is the number of nodes in the graphs. *This paper is meant to be read after reading [11]. The first paper provides the background in graph matching and retrieval which has led to the multi-hierarchy system. The current paper provides extensions and further implementation details - in particular the addition of the descriptor unit hierarchy. Other supporting material can be found in [3] which discusses compilation of the conceptual graph hierarchy and gives improvements to the uni-hierarchy method.*

The most typical application of these techniques is bibliographic retrieval in which articles or abstracts of articles are stored as individual conceptual graphs. Along with the graphs is provided a type hierarchy. A query for such a database is a conceptual graph and answers to the query take the form of subgraphs (generalizations) or supergraphs (specializations) of the query that occur in the database. The database example that we will be using throughout the paper is shown in Fig. 1.

In the traditional graph hierarchy, graphs are partially ordered by subsumption [1,2,3,5,7,11,13]. Fig. 1 gives the generalization hierarchy for the conceptual graphs Sowa uses to demonstrate the canonical formation rules for conceptual graphs [14]. The ordering in the hierarchy assumes that PERSON is a subtype of GIRL and all other concept types are pairwise incomparable. The relations AGNT, MANR, and OBJ are pairwise incomparable.

In the next section we will describe how graphs are represented as sets of node descriptors and the resulting node descriptor hierarchy. Following sections will describe the relationships between the hierarchies, and give a search method that exploits the indexing information provided by the multiple hierarchies. Before going further it should be noted that the actual objects are not stored in the hierarchies directly but in order to take advantage of redundancy they are represented by the differences between neighbouring objects. In this manner computation can be shared through the mappings between neighbouring objects. This compilation method is discussed in section 3.

2 Node Descriptor Hierarchy

Every conceptual graph is represented as a set of node descriptors.

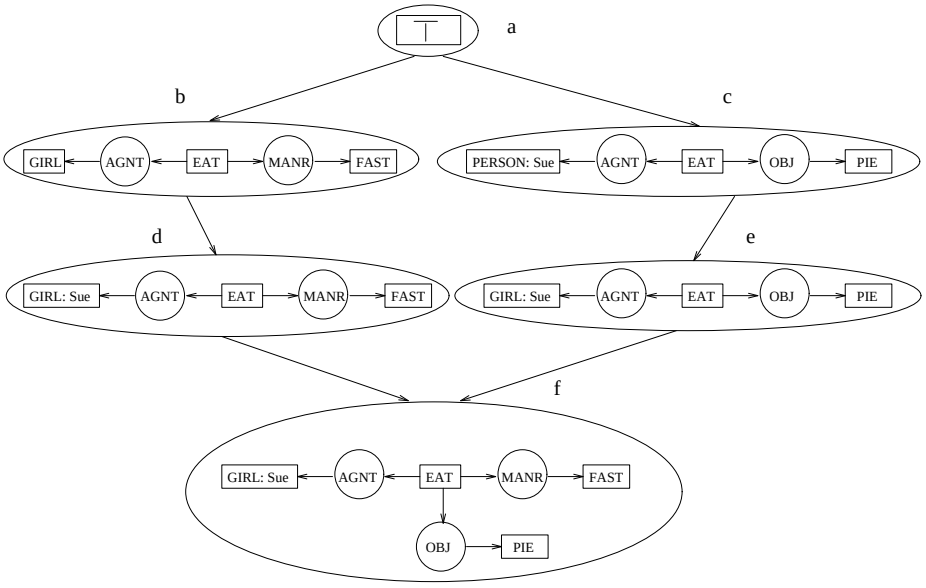

Figure 1: A generalization hierarchy

2.1 Node Descriptors

A node descriptor describes the neighbourhood of a concept or relation in a conceptual graph. The neighbourhood of radius r of a concept or relation c in a conceptual graph u is every concept and relation in u, that has a path from c of length less than or equal to n. Other properties are also used, such as cycle length, distances, and arc information. There are three types of descriptor units: one descriptor unit for the node itself; one descriptor unit for each adjacent node; and one descriptor unit for every other node within the given radius.

2.1.1 Self Descriptor Units (abbreviated "S-dus")

A self descriptor unit describes the concept or relation that is the focus of the node description. It takes the form (S, v, d) where

- S identifies the descriptor unit as being a self descriptor unit.
- v = the information contained in the concept or relation, a concept type and referent or a relation type respectively. We shall refer to the v field of all descriptor unit kinds as the label field for that kind.
- d = the length of the shortest non-trivial cycle the node is on, 0 if none. This will be an even number, since conceptual graphs are bipartite.

The partial ordering over self descriptor units is defined: (S, v_1, d_1) $\leq$ (S, v_2, d_2) if and only if

- $v_1 \leq v_2$ and
- $(d_2 = 0 \vee (d_2 \neq 0 \wedge d_2 \geq_Z d_1))$ where $\geq_Z$ refers to the ordering over positive integers.

For instance the concept [PERSON: Sue] in graph c in Fig. 1 has a self descriptor of (S, PERSON: Sue, 0). The concept [GIRL: Sue] in graph e has the self descriptor (S, GIRL: Sue, 0). The ordering between these is (S, GIRL: Sue, 0) $\leq$ (S, PERSON: Sue, 0).

2.1.2 Adjacent Descriptor Units (A-dus)

An adjacent descriptor unit describes some concept or relation adjacent to the focus concept or relation. The descriptor is of the form (A, v, a). The main difference from a self descriptor unit is that the field a contains the label of the arc that attaches the node v to the focus. For conceptual graphs the label of the arc is the number of the link. The arc number identifies the argument position of the n-ary relation. An adjacent descriptor for the relation (AGNT) when the concept [PERSON: Sue] is the focus in graph c in Fig. 1 would look like (S, AGNT, 2), since the 2nd arc of the relation (AGNT) is attached to the focus.

The partial order over adjacent descriptor units is defined:

$$(\mathrm{A}, v_1, a_1) \leq (\mathrm{A}, v_2, a_2) \Leftrightarrow (v_1 \leq v_2 \wedge a_1 = a_2).$$

2.1.3 "Other" Descriptor Units (O-dus)

An other descriptor unit is used to describe concepts or relations which are not adjacent to the focus concept or relation, but are within the radius prescribed. In Section 2.1.5 we prescribe a radius of two for concepts and a radius of one for relations. The descriptor is of the form (O, v, d) where d is the shortest distance v is from the focus. (If we take the prescribed radii then this field is not needed.) The descriptor for the concept [EAT: *] in graph c given the focus is [PERSON: Sue] is (O, EAT: *, 2). The notation [EAT] is a short notation for the generic EAT concept [EAT: *] .

The partial order over Other descriptor units is defined:
$(\mathrm{O}, v_1, d_1) \leq (\mathrm{O}, v_2, d_2) \Leftrightarrow (v_1 \leq v_2 \wedge d_1 \geq_N d_2)$, where $\geq_N$ is the ordering over natural numbers.

2.1.4 Comparing A-dus to O-dus

Sometimes it is necessary to see whether an O-du is a generalization of an A-du, since A-dus are really O-dus but have distance 1 (and contain arc information). The partial order is as defined as:

$$(\mathrm{O}, v_1, d_1) \leq (\mathrm{A}, v_2, a_2) \Leftrightarrow (v_1 \leq v_2).$$

2.1.5 Radius of Neighbourhood

The radius of the neighbourhood of each node is kept constant for some domain. Here we argue that the radius of concept nodes and relation nodes should differ (by one). The radius of concepts should be even and relations odd. If the radius is odd for a concept it will include relations but not necessarily all the concepts it is attached to. The neighbourhood in this case scribes a subgraph of the original conceptual graph. This subgraph is not necessarily a conceptual graph. Similar arguments hold for an even radius for relations.

A radius of one for a relation gives neighbourhoods which represent subgraphs containing the arguments of each relation. A radius of two for a concept gives neighbourhoods which represent subgraphs that define what concepts the concept can be associated with. They are like selectional constraints (the minimal graphs that certain combinations of concepts and relations can occur in).

2.2 Node Descriptor Construction

Here we give a process by which the set of node descriptors of a given graph can be constructed. The object is to bring the node descriptors to exactly the correct level of specificity to "virtually" (i.e. not guaranteed in the worst case) insure that two nodes with identical descriptions are indeed isomorphic in the given graph. The process starts with all nodes in the same equivalence class and then iteratively increases the descriptions of non-unique nodes until further "refinement" can no longer be made. Descriptions can be "increased" by considering the equivalence class information from the previous iteration in the comparisons for the next iteration.

Every node in every database graph (and every query graph) is to be represented as a node descriptor as above. The following algorithm gives the necessary details: (Two nodes are in the same equivalence class if they have the same node descriptor. Thus, as descriptors become more specific, equivalence classes may become smaller and more numerous.)

BEGIN(* Generate Node Descriptors *)

1. Represent each node as a set of dus as described above. Start with all nodes in equivalence classes based on their S-dus. Two nodes are in the same equivalence class iff their S-dus are identical.

2. REPEAT

   2.1 Record current node descriptors and equivalence classes. (classes need only be recomputed for nodes that are not in singleton classes already). Two label fields will be considered to match (in O-dus and A-dus) iff they are in the same equivalence class in the previous iteration.

   2.2

   UNTIL equivalence classes of nodes have not changed from the previous iteration.

3. Return node descriptors from the previous iteration.

END

Except for very unusual graphs, the resulting node descriptors from this algorithm are such that two nodes with the same descriptor are truly symmetric in the given graph. For many graphs only one iteration is required. The algorithm can be viewed as forming equivalence classes of nodes based on transitive closures of their neighbourhoods.

The node descriptors in Fig. 2 come from the graphs in the generalization hierarchy in Fig. 1. Because there are no isomorphic nodes within an individual graph only one iteration of the above algorithm was required to generate the node descriptors.

In calculating node descriptors as above, a node descriptor equivalence predicate must be available. Here is the algorithm; (for detecting exact matches, ignoring any type hierarchy and assuming S-dus are known to match):

```
BEGIN (* Node Descriptor Equivalence Test for two node descriptors nd1
and nd2 *)
```

```
1.  FOR each du x in node descriptor nd1
    1.1 Generate a candidate binding set (a set of dus in nd2 that
        x can bind to given the above partial order definitions).
    1.2 IF the set generated in 1.1 is empty, RETURN false (no binding for
        x, so nd1 and nd2 are not equivalent).
    1.3 IF the set generated is a singleton, x has a unique match so
dequeue it from further consideration and increment count of
unique matches.
2.  IF number of unique matches = number of dus, RETURN true
    ELSE call a bipartite matching algorithm to try to find a 1-1
    mapping between dus in nd1 and their candidates in nd2, RETURN result.
END
```

2.3 Comparing Node Descriptors

The comparison tests for determining if one node descriptor subsumes another in the partial ordering of node descriptors is similar to the equivalence test above which is the exact match case of the subsumption test. The candidate binding list are formed over the partial order over dus.

The complexity of comparing node descriptors reduces to the complexity of bipartite matching in the worst case, $O(n^3)$, but in practice is usually no worse than $O(n)$. Comparing this with the potentially exponential graph-isomorphism tests gives a sense of the possible savings.

3 Constructing and Compiling the Hierarchies

Subsumption of node descriptors implies a hierarchy over node descriptors. Fig 3 gives a node descriptor hierarchy for the graphs of Fig. 1.

Node descriptors in the node descriptor hierarchy point directly to the graphs that they describe in the generalization hierarchy, except that transitive links (based on only inter-hierarchy links) are ignored.

3.1 Compilation of Hierarchies

To reduce both space and processing requirements it is possible to further compile the information in the hierarchies by replacing the objects in the hierarchies with differences between adjacent objects in the hierarchies. Node descriptors represent subgraphs of the original graphs, thus they can be treated as conceptual graphs. We believe that we can use the same formation rules for their construction. The differences can then be replaced with instances of the formation rules. Hence we can take advantage of a method [3] of sharing computation amongst conceptual graphs in the generalization hierarchy. Fig. 5 shows the result of compiling the node descriptor hierarchy based on this method. Node descriptors 1 through 11 are in the basis so have no derivation rule other than copy, and hence they contain full node descriptors as in Fig 2.

Node Descriptor Number	Node Descriptor
a:	
1	{ (S, $\top$, 0) }
b:	
2	{(S, GIRL: *, 0), (A, AGNT, 2), (O, EAT: *)}
3	{(S, AGNT, 0), (A, GIRL: *, 2), (A, EAT: *, 1)}
4	{(S, EAT: *, 0), (A, AGNT, 1), (A, MANR, 1), (O, GIRL: *), (O, FAST: *)}
5	{(S, MANR, 0), (A, EAT: *, 1), (A, FAST: *, 2)}
6	{(S, FAST: *, 0), (A, MANR, 2), (O, EAT: *)}
c:	
7	{(S, PERSON: Sue, 0), (A, AGNT, 2), (O, EAT: *)}
8	{(S, AGNT, 0), (A, PERSON: Sue, 2), (A, EAT: *, 1)}
9	{(S, EAT: *, 0), (A, AGNT, 1), (A, OBJ, 1), (O, PERSON: Sue), (O, PIE: *)}
10	{(S, OBJ, 0), (A, EAT: *, 1), (A, PIE: *, 2)}
11	{(S, PIE: *, 0), (A, OBJ, 2), (O, EAT: *, 2)}
d:	
12-16	substitute GIRL: Sue for GIRL: * (restrict-individual) in 2-6 respectively
e:	
17-21	substitute GIRL: Sue for PERSON: Sue (restrict-type) in 7-11 respectively
f:	
22	{(S, GIRL: Sue, 0), (A, AGNT, 2), (O, EAT: *)}
23	{(S, AGNT, 0), (A, GIRL: Sue, 2), (A, EAT: *, 1)}
24	{(S, EAT: *, 0), (A, AGNT, 1), (A, MANR, 1), (A, OBJ, 1), (O, GIRL: Sue), (O, FAST: *), (O, PIE: *)}
25	{(S, MANR, 0), (A, EAT: *, 1), (A, FAST: *, 2) }
26	{(S, FAST: *, 0), (A, MANR, 2), (O, EAT: *) }
27	{(S, OBJ, 0), (A, EAT: *, 1), (A, PIE: *, 1) }
28	{(S, PIE: *, 0), (A, OBJ, 2), (O, EAT: *) }

Figure 2: Node Descriptors for radius (1, 2) (relation, concept) of the graphs in Fig. 1

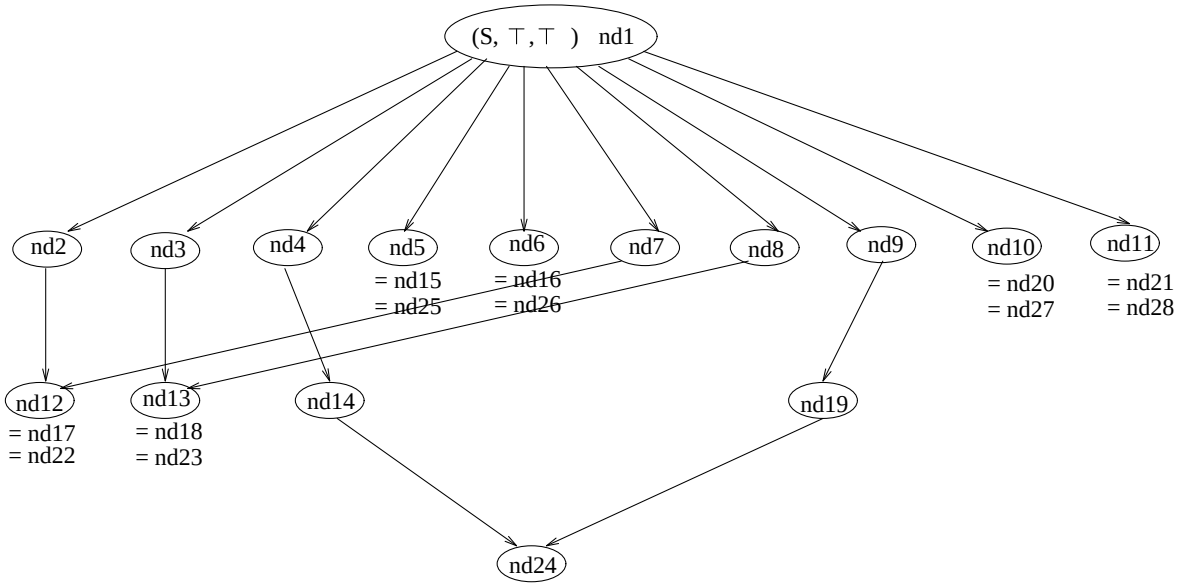

Figure 3: A Node Descriptor hierarchy (see Fig. 1)

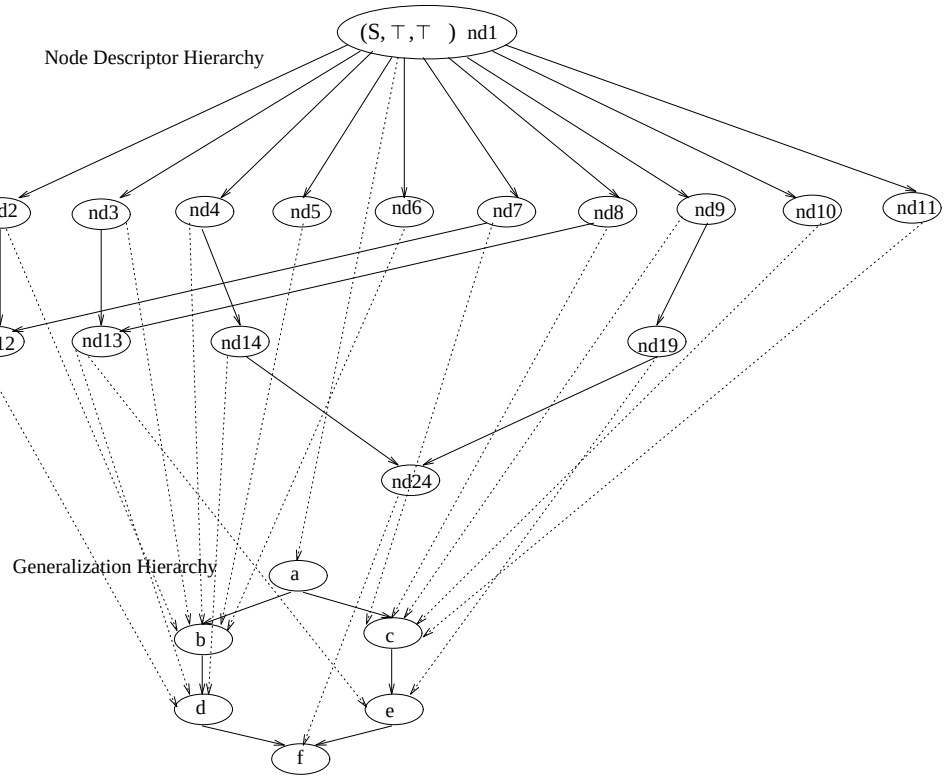

Figure 4: The relationship of the node descriptor hierarchy in Fig. 3 to the generalization hierarchy in Fig. 1

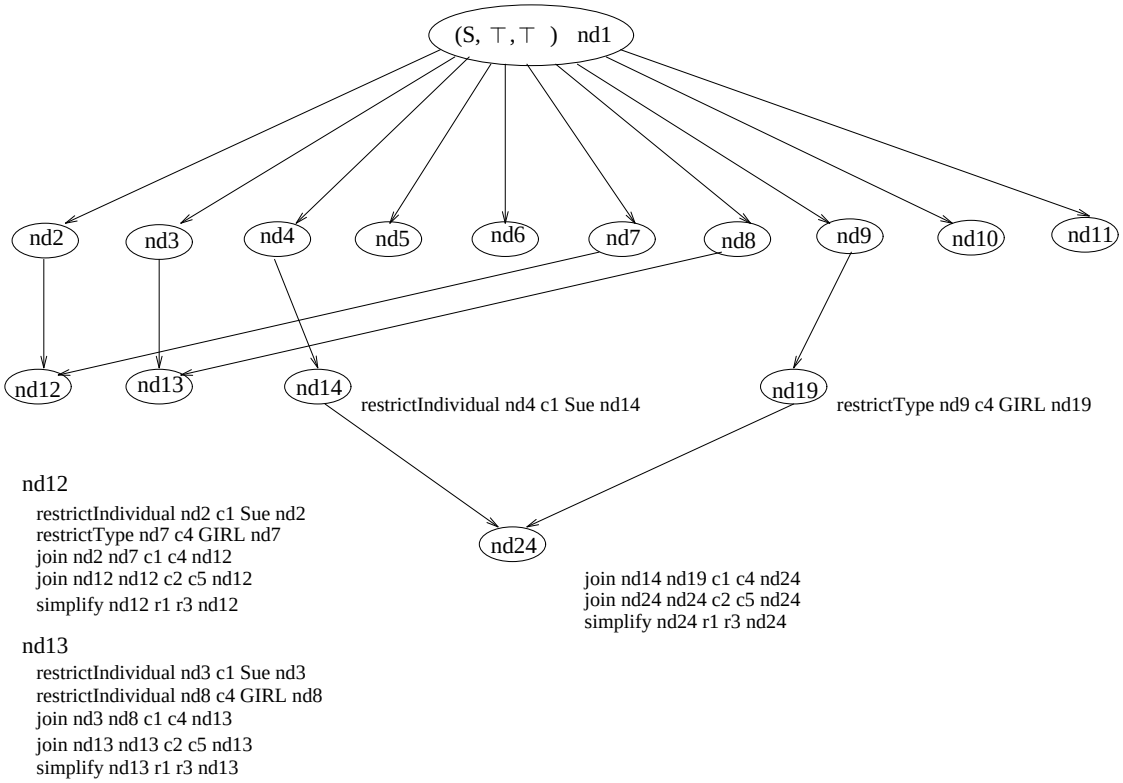

Figure 5: Compilation of the node descriptor hierarchy in Fig. 3

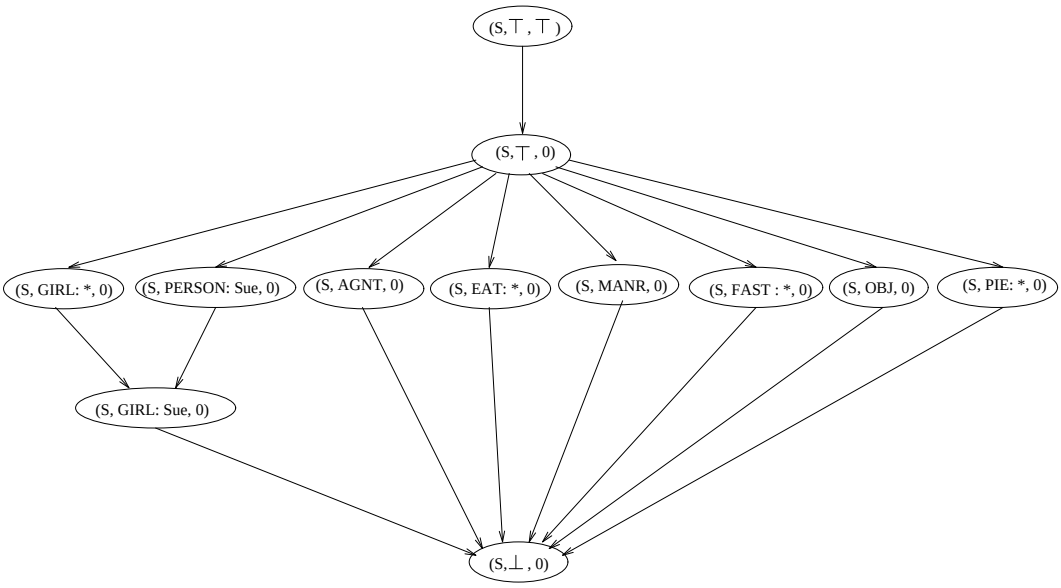

Figure 6: A Self Descriptor Unit hierarchy

3.2 Descriptor Unit Hierarchy

The descriptor units making up the node descriptors allow similar gains that were achieved from moving from the graph hierarchy to the node descriptors to be achieved by taking advantage of the fact that many node descriptors share dus. A "descriptor unit hierarchy" is generated to achieve this. Figs. 6, 7, 8, and 9 show the hierarchies which represent the partial orders over each kind of du.

4 Insertion and Retrieval

An integrated search method has been developed that is used at each level of the search: through the descriptor unit hierarchy, node descriptor hierarchy, and the conceptual graph hierarchy. This is a generalization of the uni-hierarchy method developed in [1,2,3,5,7,11,13]. We shall assume that the type-hierarchy (the highest level hierarchy) is "compiled" into a hash table such that given any two types, in constant time it can be determined if one is a generalization of the other.

Insertion and retrieval operations into a multi-hierarchy scheme are quite similar. They both require finding the position of the query graph in the conceptual graph (bottom level) hierarchy. Once done, generalizations and specializations of the query can be read out immediately. Further, because of the low granularity level provided by the multi-hierarchy scheme, close or partial matches are also immediately available.

First we give the method for insertion into a single hierarchy without regard to the other hierarchies. This may be used to search the conceptual graph hierarchy directly as in the uni-hierarchy method, or in the case of a multi-hierarchy scheme, to search the descriptor unit hierarchy:

The immediate generalization and immediate specialization sets are found in two phases. Phase II makes use of the immediate generalizations found in Phase I. Both phases attempt to use the

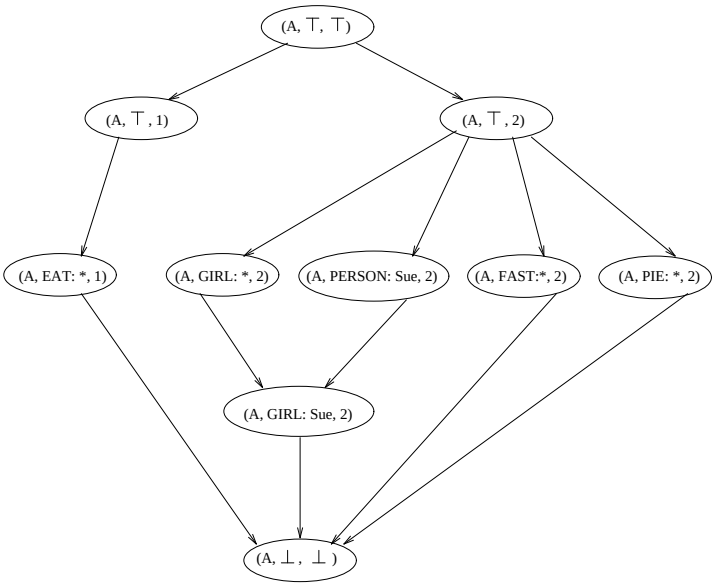

Figure 7: An Adjacent Concept Descriptor Unit hierarchy

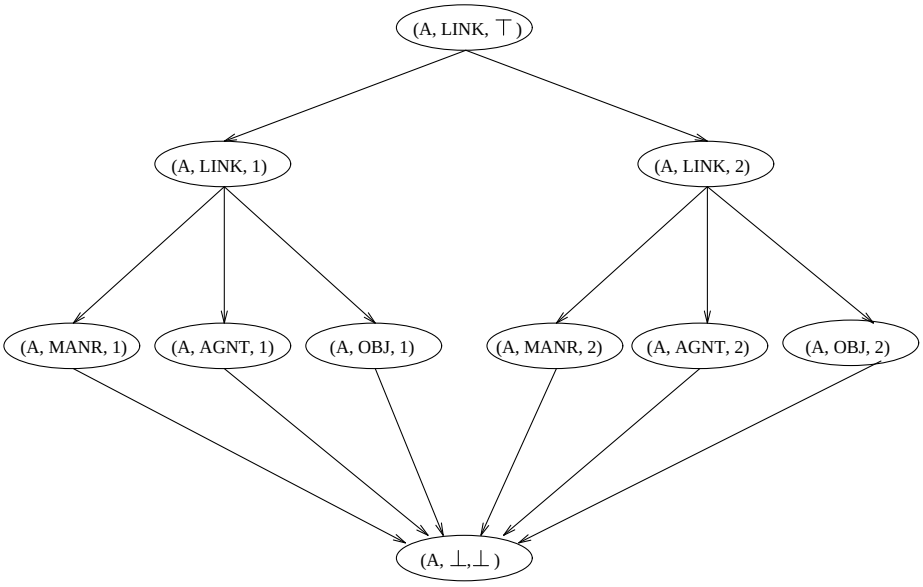

Figure 8: An Adjacent Relation Descriptor Unit hierarchy

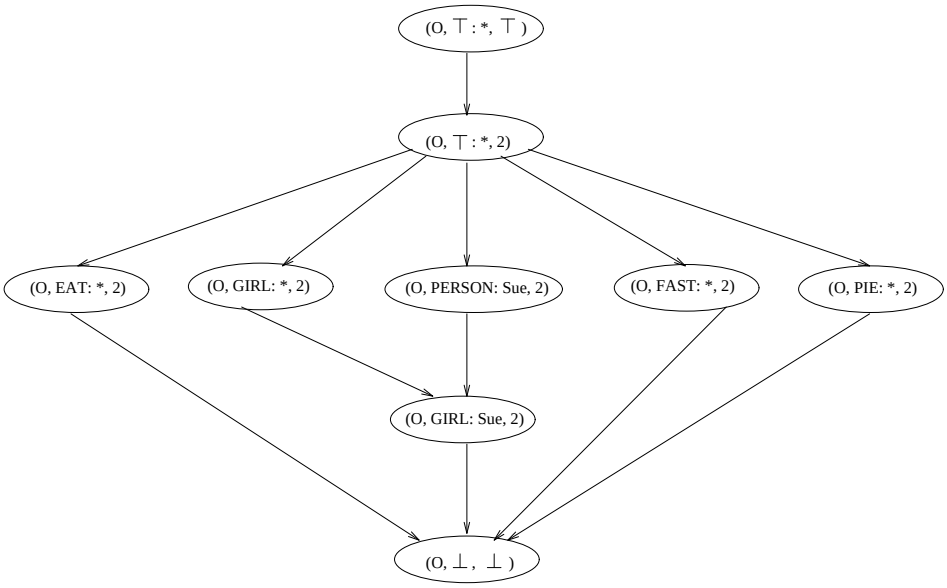

Figure 9: An Other Concept Descriptor Unit hierarchy

information in the hierarchy to minimize the number of comparison tests.

Phase I: (find IG(Q), the immediate generalizations of Q)

(1) Ordering the database objects to produce a topologically sorted list, i.e. a total ordering that embeds the original partial ordering by more-general-than.

(2) S := $\emptyset$.

(3) While there is a member X in the list
If X is a generalization of Q (comparison test) then
S := S $\cup$ {X} $-$ IG(X)
Remove X from the list.
Else
Remove X and all specializations of X from the list.

(4) Return S.

Since all database objects will be preceded by their generalizations in the topologically ordered list they only will make it to the front of the list if their generalizations have been found to be generalizations of Q. Thus, the proper screening is taking place. Now let us return to our example, Fig. 1. One ordered list (among several) for this database would be a,c,b,d,e,f.

Phase II. (find IS(Q), the immediate specializations of Q)

(5) S := $\emptyset$.

(6) Y := some element of IG(Q)

(7) I := intersection of the specialization sets of each element of IP(Q) except Y

We suggest implementation of step 7 as follows. [3] gives a more efficient method.

(7') For each z in IP(Q) except Y do
    For each specialization s of z do
        Increment count(s)
    For each item s do
        If count(s) = |IP(Q)| − 1 then I := I ∪ {s}

(8) For each specialization X of Y in topological order (as in step (1) above) do
    If X is in I and X is a specialization of Q (isomorphism test) then
        S := S ∪ {X}
        Eliminate specializations of X from the rest of the for loop.

(9) Return S.

If we actually wish to insert Q into the hierarchy, the IG and IS sets of other objects have to be updated. This is done in Phase III:

Phase III. (update immediate predecessor and successor sets of other items)

(10) For each x in IP(Q) do
    S(x) := IS(x) ∪ {Q} − IS(Q)

(11) For each x in IS(Q) do
    P(x) := IP(x) ∪ {Q} − IP(Q)

KL-ONE's classification algorithm [12] is somewhat different: in phase I an object is compared to the query as soon as one of its predecessors match Q (A depth-first approach as opposed to the breadth-first approach described here). Our experimental studies have shown that the predecessor information gained for free by this method (usually simple comparisons) do not pay for the additional predecessor tests (usually more complex) required by this method. Other variations may be feasible though, such as comparing an object as a predecessor as an IP when a certain proportion of its immediate predecessors have succeeded. Since Phase I is not the expensive phase the differences here are not that significant. Some systems that maintain a partial order have Phase II work exactly as Phase I but from the other end of the hierarchy. This is not as efficient as the method here since at the minimum all successors of Q (and some others) must be queried, whereas the Phase II here only does comparisons on the immediate successors (and some others). Two other things point to the deficiencies of this approach: the immediate predecessor information from Phase I is not taken into account and by starting at the other end of the hierarchy the system is required to do comparisons on the most complex objects!

We have explored alternative algorithms to these that do not query the partial order in a bottom-up or top-down fashion but instead use an information-theoretic heuristic that attempts to maximize the ratio of expected information gained to comparison cost and using a few levels of lookahead [13]. We've had only limited success with these algorithms: only improvements of about 15-20 percent despite a large amount of off-line pre-processing.

Now lets move to the multi-hierarchy scheme. Here the idea is that for object A to be more-general-than B at any hierarchical level it is necessary that each syntactic subunit of A must be

a generalization of some syntactic subunit of B. The comparison between subunits is found by searching the hierarchy of next higher level. Now, although the condition stated above is necessary but not necessarily sufficient we find that in the context of the multi-hierarchy scheme, "acting" as if sufficiency is the case brings tremendous efficiency gains with only a slight chance of error. If such error can not be tolerated the answers produced by the system can, of course, be double-checked using a standard comparison algorithm for objects of a given hierarchy.

Thus, the multi-hierarchical method proceeds as follows with the only actual comparison tests being performed in the descriptor unit and type hierarchies (where they are easiest):

1. Calculate node descriptors for the query graph (as described in Section 2.2)
2. With each graph in the conceptual graph hierarchy maintain two fields graph-g-count and graph-s-count both initialized to 0.
3. With each node descriptor in the node descriptor hierarchy maintain two fields nd-g-count and nd-s-count, both initialized to 0.
4. For each node descriptor q of the query graph:
   (a) Find its place in the node descriptor hierarchy as follows:
      i. For each du of the node descriptor:
         A. Find its place in the descriptor unit hierarchy, by searching that hierarchy directly (and consulting the type hierarchy)
         B. Increment by 1 the nd-s-count of node descriptors that have a du that is more specific than the given du. (by tracing inter-hierarchy links)
         C. Increment by 1 the nd-g-count of node descriptors that have a du that is more general than the given du. (by tracing inter-hierarchy links)
      ii. Node descriptor q is then more-general-than node descriptors with an nd-s-count equal to the cardinality of q.
      iii. Node descriptor q is then more-specific-than node descriptors with an nd-g-count equal to their cardinality.
   (b) Increment by 1 the graph-g-count of graphs that have a node descriptor that is more-specific-than the given node descriptor (by tracing inter-hierarchy links)
   (c) Increment by 1 the graph-s-count of graphs that have a node descriptor that is more-general-than the given node descriptor (by tracing inter-hierarchy links)
   (d) The query graph is more-general-than graphs with a graph-s-count equal to the number of nodes in the query graph.
   (e) The query graph is more-specific-than graphs with a graph-g-count equal to the number of nodes.
   (f) Partial matches may be found by consulting the graph-s-count and graph-g-count fields of graphs with their maximum values.

Since the relationship between hierarchies is uniform the multi-hierarchy algorithm can be specified and implemented in a recursive manner. Above we have given the expanded version

in order to remove some of the mystery. The algorithm can be further enhanced by using the improved intersection method discussed in [3]. This intersection algorithm obviates the need for maintaining counts. Here is the recursive version of the algorithm:

```
Place(x,H)

(* Finds the place of node x in hierarchy H by returning two lists
IG(x) and IS(x). The immediate generalizations IG and immediate
specializations of IG respectively.  Once these are found, all
generalizations and all specializations can be found by tracing
pointers. If it is necessary to insert into the hierarchy, Phase III of
the uni-hierarchical method given above may be used at this point.
This algorithm is a recursive algorithm that can be called with the
graph hierarchy, node descriptor hierarchy, or descriptor unit
hierarchy.  Normally, Place will be called with the graph hierarchy,
with other hierarchies being called recursively. *)

Begin
If hierarchy = du-hierarchy then
   call the uni-hierarchical method and return IG sets and IS sets.
   The comparison test in this method consults the compiled type
   hierarchy directly.
Else
   Generate all subunits of x (node descriptors if x is a graph, dus if
   x is a node descriptor.)

   For each subunit y call Place(y,next(H)) where next(H) is the next
   level hierarchy and temporarily insert the subunits in their place.

   Now all successors of x in H can be found by intersecting the
   successor sets of its subunits in H using the intersection algorithm
   (by following first inter-hierarchy links and then intra-hierarchy
   links). From these the IS set can be found.

   To find all predecessors of x in H, one runs the intersection
   algorithm on the predecessor sets of the subunits of x.  Now the
   predecessors of x are a superset of this intersection (which is
   often empty).  A node z is a predecessor if it has been reached by
   size(z) subunits where size(z) is the number of nodes in z. Such
   nodes can be calculated naturally in the normal processing of the
   intersection algorithm.  Once the predecessors and successors of Z
   have been found - it is straightforward to calculate and return the
   IG and IS sets.
End
```

Node Descriptor Number	Node Descriptor
u:	[PERSON]←(AGNT)←[EAT]→(OBJ)→[PIE]
q_1	{(S, PERSON: *, 0), (A, AGNT, 2), (O, EAT: *)}
q_2	{(S, AGNT, 0), (A, PERSON: *, 2), (A, EAT: *, 1)}
q_3	{(S, EAT: *, 0), (A, AGNT, 1), (A, OBJ, 1), (O, GIRL: *), (O, PIE: *)}
q_4	{(S, OBJ, 0), (A, EAT: *, 1), (A, PIE: *, 2)}
q_5	{(S, PIE: *, 0), (A, OBJ, 2), (O, EAT: *)}

Figure 10: Node descriptors for the query u

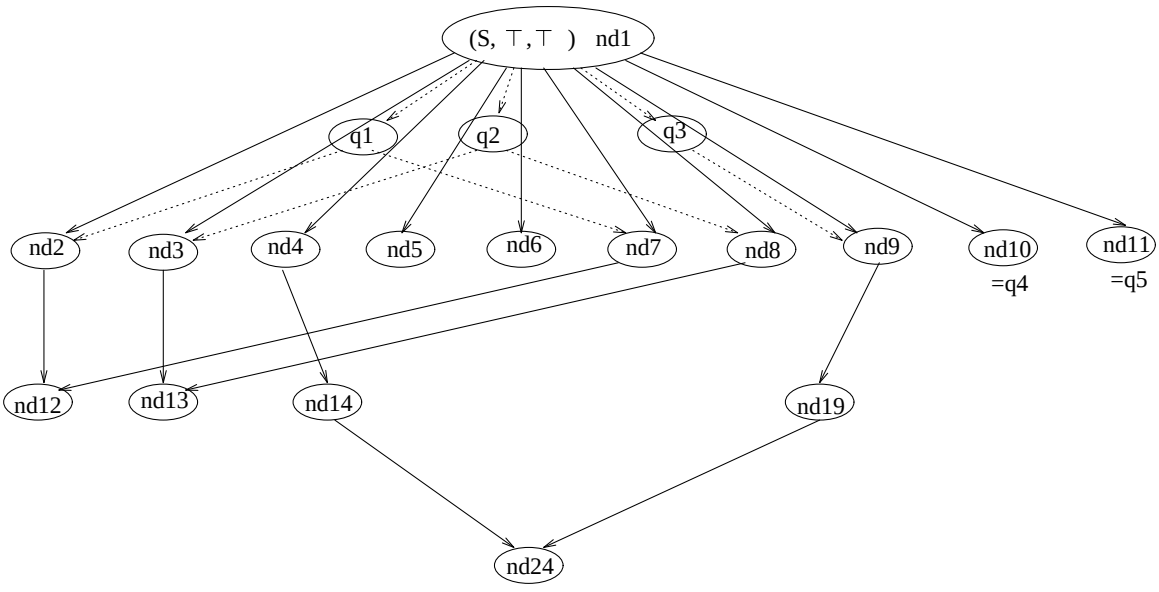

Figure 11: The location of the query node descriptors of Fig. 10

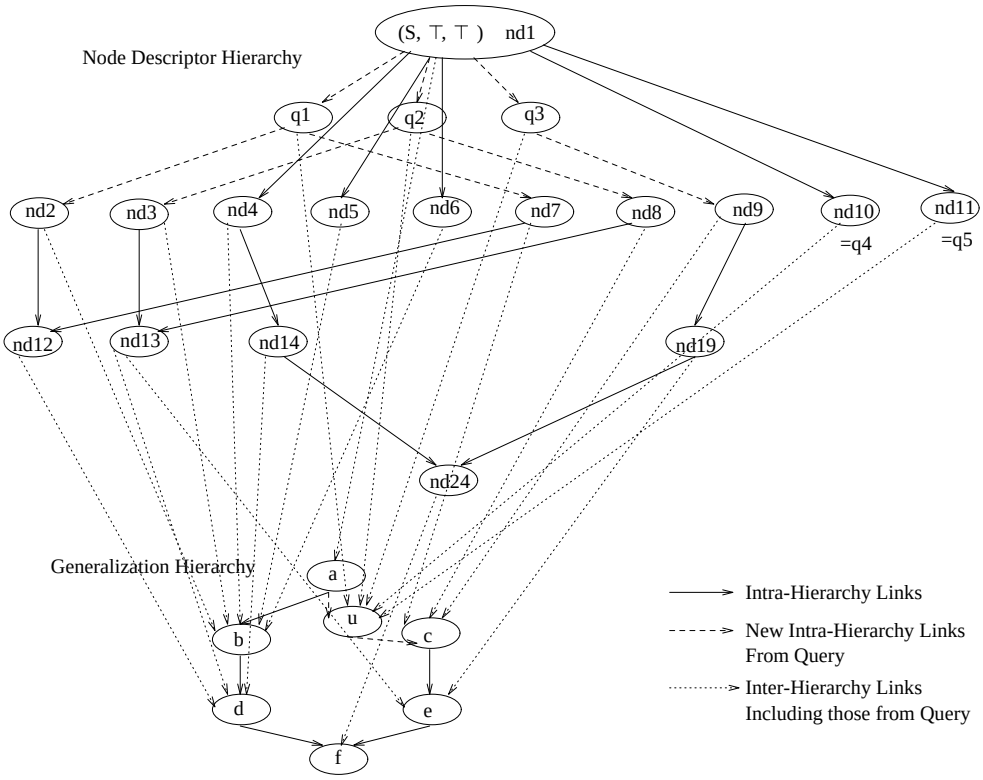

Figure 12: The result of adding the query graph in Fig. 12 to the node descriptor hierarchy and generalization hierarchy

Figs. 10 and 11 show the locations of the node descriptors for a sample query in the node descriptor hierarchy. Fig. 12 shows the resulting links in the node descriptor hierarchy and the generalization hierarchy after the query graph has been inserted. Fig. 13 shows all the hierarchies generated from the graphs of Fig. 1, and their interrelationships.

5 Ongoing Work

Adapting the multi-hierarchy method for abstraction remains to be done. The algorithms described here have been based on the assumption that generalization and subgraph (modulo typing and referents) are equivalent for conceptual graphs. When using abstraction in the form of concept type and relation type contraction and expansion this is not, the case generally.

We are in the process of fully implementing the multi-hierarchical node descriptor method in C++ and making relevant experimental comparisons with the uni-hierarchical system. We intend to incorporate these retrieval methods in a principled conceptual graph processor based on first-order conceptual graphs [14]. It is hoped that this tool will support large scale applications of conceptual graphs.

These methods are being explored in the associative graph database that is being used in

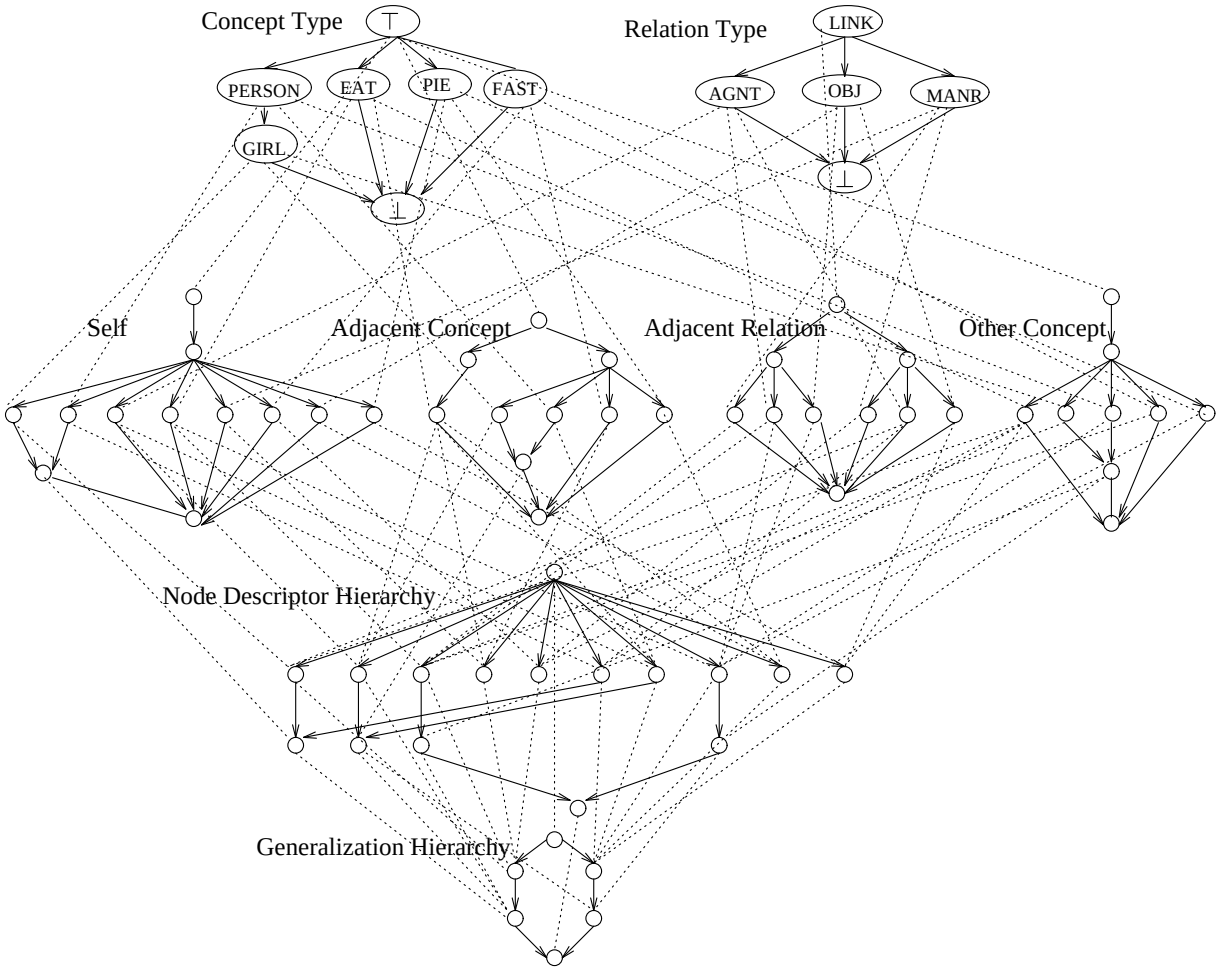

Figure 13: The relationship between the different hierarchies

Morph - the graph-based adaptive pattern-oriented chess system[9,11]. Chess patterns in this system are represented as graphs. Because the symmetry in these graphs is high compared to most conceptual graph domains avoiding direct comparisons and sharing computation is critical.

6 Acknowledgements

The research has benefited from the constructive suggestions of Fritz Lehmann. J. Don Roberts helped implement and clarify the node descriptor generation process. Finally we would like to thank John Sowa for sharing our enthusiasm for these methods.

References

[1] **Ellis, G**, 'Efficient retrieval from the generalization hierarchy' Technical Report No 114, Dept of Computer Science, University of Queensland, Australia (May 1989)

[2] **Ellis, G**, 'Deterministic all-solutions retrieval from the generalization hierarchy' *Proc 4th Annual Conceptual Structures Workshop* Nagle, J A and Nagle, T E (Eds.) Detroit (August 1989)

[3] **Ellis, G** 'Compiled hierarchical retrieval' Proceedings of the 6th Workshop on Conceptual Graphs, Eileen Way (Ed.) SUNY Binghamton, (July 1991). Also, to appear in Current Directions in Conceptual Structures Research, Nagle, T et.al (Eds.), Springer-Verlag, (1991)

[4] **Levinson, R A** A Self-Organizing Retrieval System for Graphs, PhD thesis, University of Texas, May 1985

[5] **Levinson, R A** 'A self-organizing retrieval system for graphs' Proc AAAI-84 (1984)

[6] **Levinson, R A and Helman, D and Oswalt, E** 'Intelligent signal analysis and recognition' Proc 1st Int'l Conference on Industrial and Engineering Applications of Artificial Intelligence, ACM (1988)

[7] **Levinson, R A** 'A self-organizing pattern retrieval system and its applications' Technical Report UCSC-CRL-89-21, University of California at Santa Cruz (1989). (To be published in International Journal of Intelligent Systems)

[8] **Levinson, R A** 'Pattern formation, associative recall and search: a proposal' Technical Report UCSC-CRL-89-22, University of California at Santa Cruz (1989)

[9] **Levinson, R A** 'A self-learning, pattern-oriented, chess program' Proceedings of Workshop on New Directions in Game-Tree Search, Marsland, T A (Ed), International Computer Chess Association (1989). Also in International Computer Chess Association Journal, Edmonton, Vol 12 No 4 (December 1989) pp207-215

[10] **Levinson, R A and Snyder, R** 'Adaptive pattern oriented chess' Proceedings of AAAI-91, Morgan-Kaufman, (1991) pp601-605

[11] **Levinson, R A** 'Pattern associativity and the retrieval of semantic networks' Computers and Mathematics with Applications, (Jan 1992) To appear in the Special Edition on Semantic Networks.

[12] **Lipkis, T** 'A KL-ONE classifier' *Proceedings of the 1981 KL-ONE Workshop* Schmolze, J G and Brachman, R J (Eds.), pp128-145, Cambridge, MA, (1982). The Proceedings have been published as BBN Report No 4842 and Fairchild Technical Report No 618

[13] **Riff, B** Searching a partially-ordered knowledge base of complex objects, Master's Thesis, University of California at Santa Cruz (1988)

[14] **Sowa, J F** Conceptual structures: information processing in mind and machine Addison Wesley, Reading, MA (1984)

[15] **Sowa, J F** 'Semantic networks' Shapiro, S C (Ed), Encyclopedia of Artificial Intelligence, Wiley, New York (1987) pp1011-1024

[16] **Wilcox, C S and Levinson, R A** 'A self-organized knowledge base for recall, design, and discovery in organic chemistry' Artificial Intelligence Applications in Chemistry, ACS Symposium Series, 306 (1986)