UDS: A Universal Data Structure

Robert Levinson

UCSC-CRL-94-15
June 10, 1994

Board of Studies in Computer and Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064
(408)459-2087
FAX: (408)459-4829
E-mail: levinson@cse.ucsc.edu

ABSTRACT

This paper gives a data structure (UDS) for supporting database retrieval, inference and machine learning that attempts to unify and extend previous work in relational databases, semantic networks, conceptual graphs, RETE, neural networks and case-based reasoning. Foundational to this view is that all data can be viewed as a primitive set of objects and mathematical relations (as sets of tuples) over those objects. The data is stored in three partially-ordered hierarchies: a node hierarchy, a relation hierarchy, and a conceptual graphs hierarchy. All three hierarchies can be stored as "levels" in the conceptual graphs hierarchy. These multiple hierarchies support multiple views of the data with advantages over any of the individual methods. In particular, conceptual graphs are stored in a relation-based compact form that facilitates matching. UDS is currently being implemented in the Peirce conceptual graphs workbench and is being used as a domain-independent monitor for state-space search domains at a level that is faster than previous implementations designed specifically for those domains.In addition it provides a useful environment for pattern-based machine learning.

Keywords: RETE, pattern matching, conceptual structures, relational database, machine learning, conceptual graphs, state space search, rule matching.

I am for the delicate dance from parts to wholes and back again. We should not be captured at either end. The dance should go forever.
— P. Suppes [21]

1 Introduction

Three popular and competing database paradigms are **relational databases**, in which data is stored and accessed as a set of tables, **conceptual graphs**, in which data is stored as a set of graphs, or **semantic networks** in which data is stored in one large homogeneous network(graph). In this paper we give a data structure that can be seen to replace the others, with all the advantages of each and fewer of the disadvantages.

The basic idea will be to store the data as they traditionally are stored in a semantic network as a set of nodes with edges or hyperedges representing n-ary relations. [1]. Each node corresponds to a primitive domain object or value. To store the semantic network one requires a *node table* that gives for each node all the relation instances (edges) it is in and its position in each relation. A relation table stores for each relation instance, pointers to the nodes involved in that instance. One might then think of representing a relational database simply by giving the relation table. What is missing however are the relational schema: the type headings on individual relations in a relational database. Such schema can be created by making the node table and the relation table each into a partially-ordered hierarchy by more-general-than. Once this is done it is seen that the node table corresponds to the traditional type hierarchy used in conceptual graph theory and the more-general-than relation over the relational hierarchy forms the traditional tables of relational databases: each relational table sits directly under its schema definition.

The next key idea is that a conceptual graph (CG) can be viewed as a sequence of joins on a set of relations [2]. Thus, a conceptual graphs database can be stored simply as a partial order over ordered lists of relation instances from the relation hierarchy. Thus, the resulting data structure has three hierarchies:

- **The node hierarchy**
- **The relation hierarchy**
- **The conceptual graphs hierarchy**

Each of these hierarchies can be stored as one: the node hierarchy and relation hierarchy would sit at the top of the standard conceptual graph hierarchy. However, the distinction between the three levels needs to be retained to allow for the multiple access mechanisms to be supported.

Figure 1 below gives a conceptual graphs database of seven graphs and a query graph from which we will be taking our examples. Figure 2 shows the node hierarchy, relation hierarchy and graph hierarchy corresponding to the UDS representation of Figure 1. To be noted is the compact representation of the graphs in the graph hierarchy. Such compaction of conceptual structures is a hallmark of this new storage and retrieval method and is explained further below.

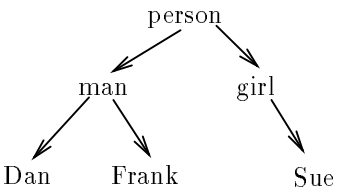
Although not made explicit in the diagram, there are bidirectional pointers from objects in the conceptual graphs hierarchy to the objects they contain in the relation hierarchy and bidirectional pointers from objects in the relation hierarchy to objects in the the node hierarchy. For example, there are pointers from Sue in the node hierarchy to the first argument of relations 4,7, and 12 in the relation hierarchy (and hence the label Sue need not actually be stored with the relations). Similarly relation 7 points to G4 in the graph hierarchy. Also not made explicit in the diagram is that siblings in the node hierarchy be sorted alphanumerically.

---

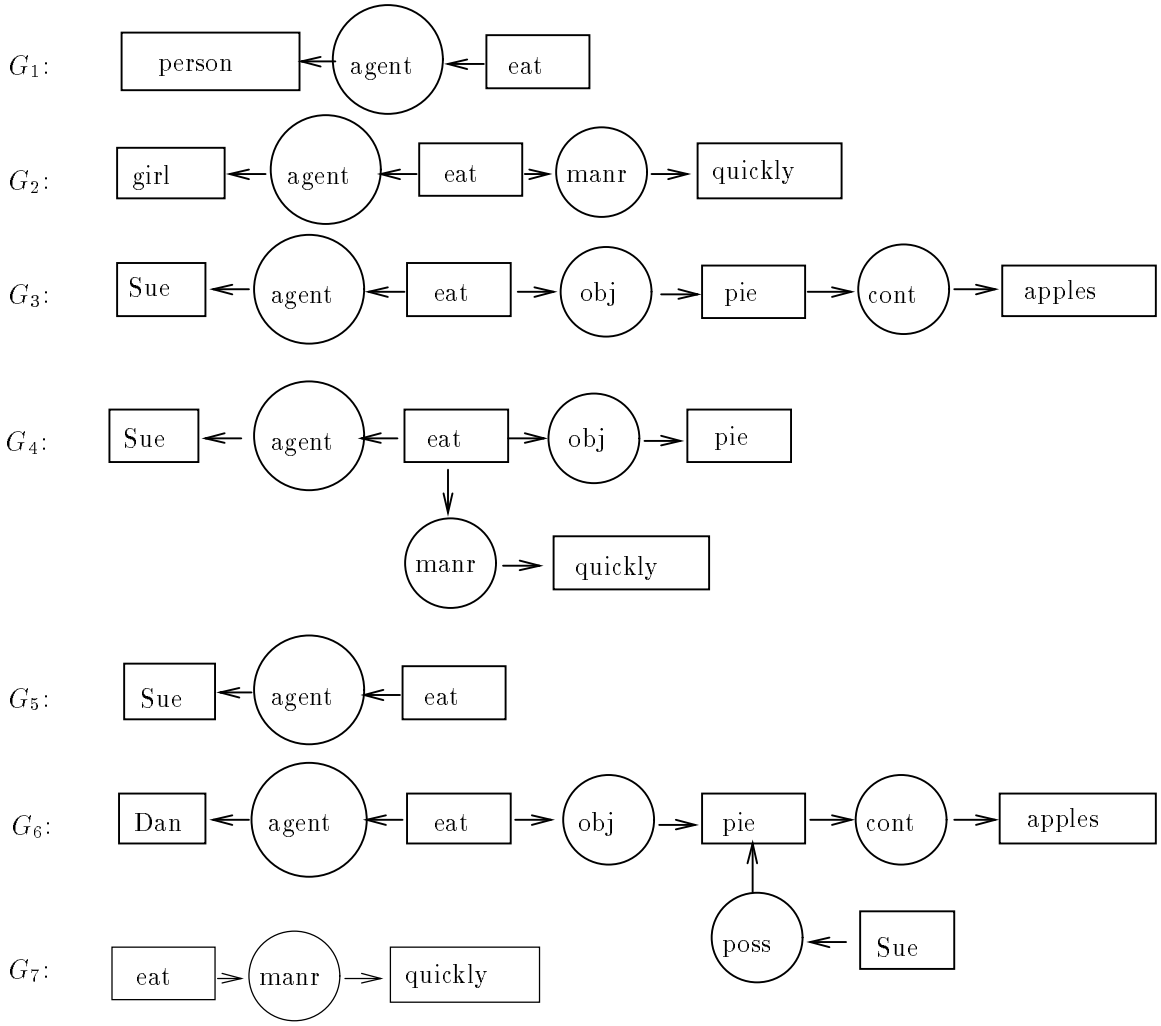
[1] A hypergraph is a set of nodes and a set of hyperedges which are subsets of 0 or more nodes. Thus a semantic network can be viewed as a directed-labelled hypergraph and partitioned semantic networks as nested-labelled directed hypergraphs[2]

[2] Informally: a CG is a very small relational database!

Type Hierarchy:



Database Graphs



Query Graph

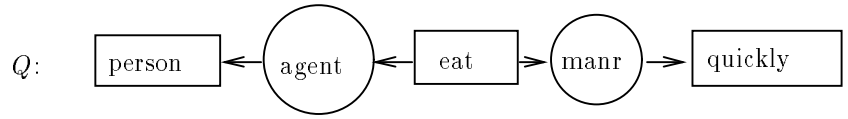
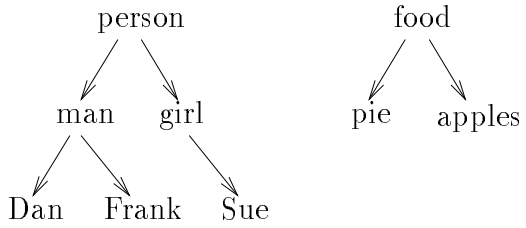
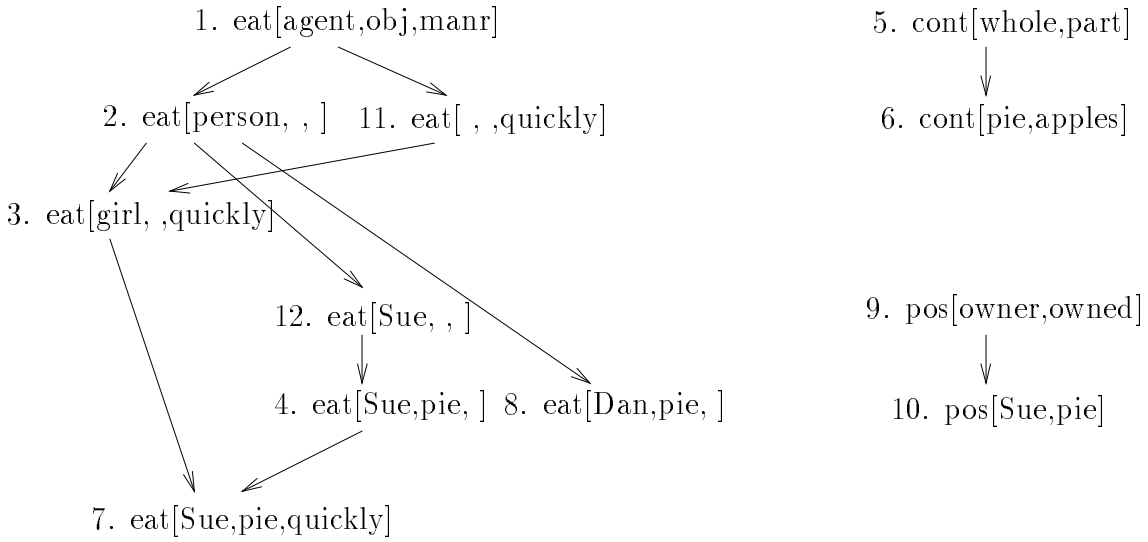


Figure 1.1: Type hierarchy, database and query graph.

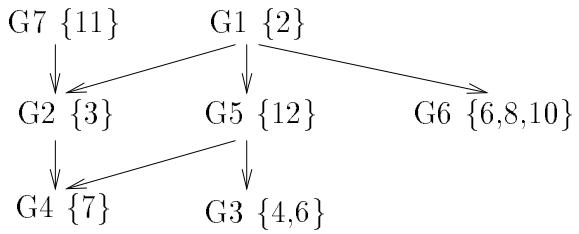
This example is adapted from [20] pp.92-93 and reprinted from [12] "manr" stands for "manner", "cont" stands for "contains" and "poss" stands for "possesses". G_1 and G_7 are generalizations of the query Q, G_2 and G_4 are specializations of Q and G_3, G_5, and G_6 are incomparable.



Node Hierarchy



Relation Hierarchy



Graph Hierarchy

Figure 1.2: The hierarchies corresponding to the graphs of Figure 1

Also associated with each hierarchy is an additional data structure that allows subsumption testing in the hierarchy to be done in constant time. The three standard data structures for this purpose are sparse arrays, hash tables and bitcodes[4]. Which scheme is selected depends on storage costs.

This use of multiple hierarchies is reminiscent of the data structure used in multi-level hierarchical retrieval [13]. The major difference is that the node descriptor hierarchy is being replaced by a relation hierarchy that better facilitates CG matching by incorporating the power of a relational database. UDS also incorporates ideas from the original universal graph method [11] proposed by us in 1984 and the universal relational model [19, 22]. Here we add some new twists that take advantage of a better understanding of the relationship between conceptual graphs, relational databases and marker-passing semantic networks[9, 6]. The new design exploits the following principles that we feel are critical in designing cost-effective information retrieval systems:

- **Every primitive data object, label or symbol should be stored only once with pointers used to denote the actual uses of the object.**
- **Every compound object should be stored with the minimum information required to represent the combinations of its parts.** Thus, for example, transitive relations should be stored by only giving the arcs represented in the transitive kernel, and the designation that the relation is transitive.
- **Given no loss of accuracy, objects should be processed at the highest level of abstraction possible.** For example, below we see that conceptual graphs can be processed at the "relational" level rather than at the node and edge level.

The competing methods may each operate over UDS while deriving benefits that were not there in the original methods:

- **Semantic network marker-passing algorithms can make use of the semantic network formed from the node hierarchy and the relation hierarchy, and the partitioning provided by the CG hierarchy.** Advantages over traditional semantic network algorithms are that the relation hierarchy allows multi-argument relations to be treated as individual chunks, and the partial order over the nodes and relations allows generalization queries to be handled more quickly and efficiently.
- **Relational database retrieval operations operate directly on the relation hierarchy by making use of the pointers to and from the node hierarchy to enhance the join process during query processing** (it is easy to determine which objects join with each object). The type hierarchy formed by the node hierarchy is a facility most relational databases don't have. Further, the sorting (see above) in the type hierarchy can be used to return answers to queries in a desired sorted order.
- **Conceptual graph operations can operate as before on the graphs represented in the conceptual graphs hierarchy supported by the type information in the node hierarchy.** For example, the Method III retrieval scheme [11, 12] and associated enhancements [13, 4] can operate directly on this hierarchy. Graph matching will be much faster than before due to the compact nature of the graphs (see below).

In addition to being able to process the data using the multiple viewpoints, other advantages to conceptual graph processing are accrued:

- **Storage of the conceptual graphs has been greatly reduced since nodes and relations are actually stored only once.** Redundancy has been removed. Similar ideas involving storage based on canonical rules are given in [3].
- **Conceptual graph matching can make use of the chunks provided by processing the graphs at the relational level rather than at the node level.**
- **Method V retrieval can also proceed naturally.** Method V is a previously unpublished method that proceeds exactly as in Method III (where a node is not queried unless its predecessors are known to match the query) except that information about how the predecessors bind to the query is propagated up the hierarchy. These bindings are reflected as relation-to-relation bindings in UDS as opposed to node-to-node bindings in a standard CG system, thus leading to substantial complexity reduction.(For further details, see Section 5 below.)

- **The fact that the semantic network represents a join over all the relations is a great advantage** : Suppose we have one CG: "Joe eats popcorn" and another CG: "popcorn is white". If we never take the join of these two CGs we might have difficulty discovering that "Joe eats something that is white." The semantic network representation facilitates the discovery of relationships that may not have been previously recognized.

2 Retrieval

In UDS, queries can be answered using either marker passing, relational processing or conceptual graph retrieval. Which method is best depends on the type of query. For instance in the example of Figures 1 and 2 one could use CG retrieval to find immediate generalizations of the object as G1 and G7 with an immediate specialization as G2 and a further specialization as G4. G4 gives the conclusion that "Sue ate quickly".

Ignoring the CG hierarchy and using just the relation hierarchy, one would check for the position of the relation eat[person,,quickly] finding generalizations of 2 and 11 and a specialization of 3 that leads to the specialization of 7 that "Sue ate quickly."

Marker passing would start with the node hierarchy at nodes eat, person and quickly, finding all relations involving only these nodes as 2 and 11. Through spreading activation to more specific instances of "person" it would be discovered that both "a girl" and Sue eat quickly.

In this example, in which the query was a single relation, relation-based retrieval worked best. For queries involving multiple relations and highly structured objects it is likely that CG retrieval would work best, having formed the chunks ahead of time. **In particular, conceptual graphs can be used to store complex relational database queries for faster processing in the future, much in the same way as** *views***[5] do currently.**

Semantic network marker-passing will work best for "free form" queries such as " In which ways are Joe and Sue related?".

3 Insertion and Deletion

The costs of insertion and deletion from the structure have decreased on the average due to the indexing properties of the multiple hierarchies and improved matching costs. If inserting a node, one adds it to the node hierarchy exactly as one would do with the type hierarchy in conceptual graphs. Inserting a relation or a new relational scheme definition into the relation hierarchy likewise proceeds exactly as in relational databases while benefiting from the hierarchical organization. Insertion of a conceptual graph also proceeds exactly as before except that the relations in the conceptual graph must also be inserted into the relation hierarchy. This small additional cost is outweighed by the benefits gained due to more efficient matching in the conceptual graphs hierarchy.

4 Compaction of Conceptual Graphs

The standard diagrammatic representation of conceptual graphs [20] makes these graphs appear to have more complexity than they actually do. **If one were to implement a conceptual graph based on the diagrammatic representation the costs associated with storage and matching would be much higher than they need to be.**

For example, a 3-ary relation in a diagrammatic conceptual graph requires 3 concept nodes, 1 relation node and 3 argument arcs. Taking the view [2] that conceptual graphs are nested directed hypergraphs, the 3-ary relation becomes simply 3 nodes and 1 labeled hyperedge. This is certainly an improvement.

But there is yet farther to go, if one takes the relational view of UDS. A 3-ary relation is a single node in the relation table! A conceptual graph becomes a set of relation nodes where the relations are those stored in the relational table. In those conceptual graphs in which an individual object appears only once it is enough to assume that all relations are joined to other relations in which one

or more nodes are shared commonly. For those conceptual graphs in which objects may be repeated and take different slots in the relations, explicit joins must be defined between the relation nodes in which it is stated exactly which arguments get joined to which arguments. **So, the final view is that a CG is a set of nodes representing individual relations and a set of labeled edges representing joins between these relations.**

Figure 3 below illustrates a compaction of a conceptual graph representing a molecule involving 2 oxygen atoms, a carbon atom, 2 nitrogen atoms, a double bond and 3 single bonds. Thus the standard chemical diagram contains 5 nodes (for atoms), and 4 edges (for bonds). From the diagram we see that the standard CG diagrammatic form requires 9 nodes and 8 edges. The hypergraph representation of the CG uses 5 nodes and 4 edges (as in the chemical graph). But the relation-based CG representation has managed to store everything as 4 nodes and 3 edges where the nodes point to relations in the relation hierarchy. The labels on the edges denote which arguments in one relation join to which arguments in the next relation. Note further that for the graphs in Figures 1 and 2 no such join information was required and the graphs were reduced to sets (of relations).

This compaction has favorable properties for supporting matching:

- **There are fewer nodes and edges.**
- **Since nodes denote more specific objects than before, their likelihood of matching is significantly lessened.** This leads to fewer false matches that require backtracking in subsumption tests.
- **Since relation nodes are stored only once, it is easy to find node correspondences between two CGs** since if a node is shared they will be pointing to the same node in the relation hierarchy, (or if one node is more-general-than the other there will be a path). That is, the first stage of most subgraph-isomorphism tests [1] in which one finds which nodes could bind (as "candidates") to other nodes can be performed by simply reading off correspondences from the hierarchy, or comparing bit encodings [4].

The same abstraction mechanism that goes from nodes to relations to graphs can be taken one step further to facilitate the storage and retrieval of nested-context conceptual graphs. A graph will point to nested-graphs in which it serves as a context. The encoding of negated contexts and lines of identity is analogous to the encoding of join-structure when going from relations to graphs. The abstraction mechanism may be continued on to multiple layers of nesting.

4.1 Subsumption preservation in compacted form

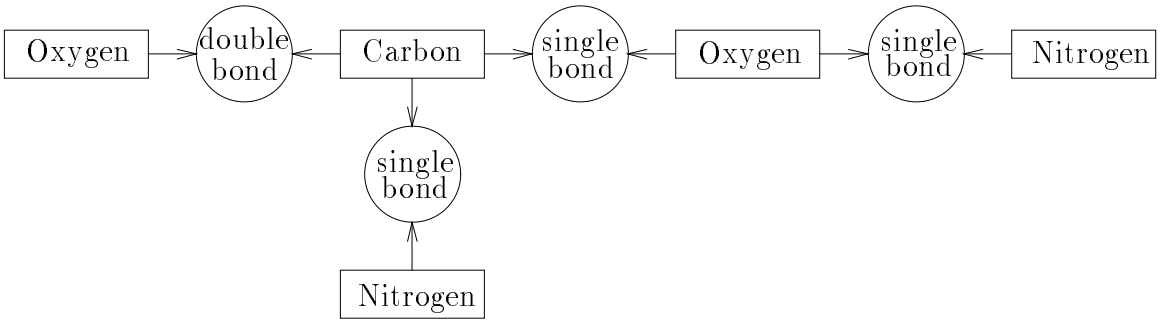
For the relation-based compacted form to be useful it is desirable that the more-general-than relation between two conceptual graphs G1 and G2 be checkable by subgraph-isomorphism on the compacted form. This is the case, since the structure of CGs is such that if G1 is more-general-than G2 each relation in G1 should have a corresponding more-specific relation in G2. This corresponds directly to comparing two nodes in the relation hierarchy.

The edges in the compacted form represents the join of two adjacent relations. Whether edges in G1 can match edges in G2 can be determined by simply checking that the fields on which the relations are joined correspond (given that the endpoints of the edges are known to correspond).

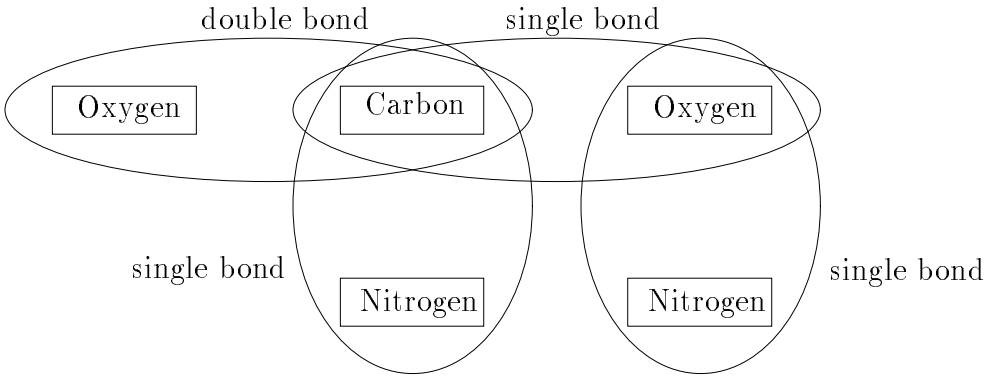
5 Method V retrieval algorithm

Method V retrieval is a previously unpublished algorithm that is designed to have the simplicity of Method III retrieval [11] coupled with the benefits of Method IV. We discuss it here since it can be used to even further enhance the efficiency of retrieval over the conceptual graph hierarchy.

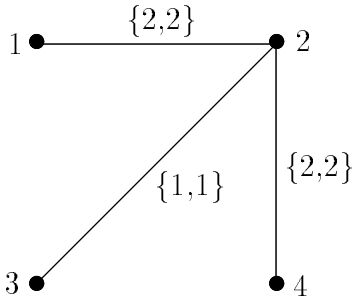
Method IV [12, 13] introduced the notion of "node descriptors" which are more specific descriptions of nodes in graphs that support matching due to their specificity much in the way that relation nodes do so in the above discussion. The idea of Method IV is to do many graph comparisons in parallel by taking advantage of their shared node descriptors. Method V goes one step further by



CG Diagramatic Representation



CG Hypergraph Representation



Relation Table

1 double bond [Oxygen,Carbon]
2 single bond [Oxygen,Carbon]
3 single bond [Oxygen,Nitrogen]
4 single bond [Nitrogen,Carbon]

Relation-Based CG Representation

Figure 4.1: Compacting Conceptual Structures

noting that **a graph is itself its best descriptor of its nodes** and that if A is a subgraph of B, the nodes in A (and hence their descriptors) have bindings to nodes (and descriptors) in B.

In the conceptual graph hierarchy we store bindings from the nodes of each graph to the nodes of their successors. (These are additional pointers than those discussed above). As a graph may bind in more than one place, there actually may be multiple sets of bindings that would have to be stored and sometimes, such as with star-graphs, an exponential number of bindings may be possible. To prevent this combinatorial explosion and for simplicity sake, for each node in a graph we store all the nodes it could possibly bind to in each of its successor graphs. Compaction as above reduces the number of such possibilities. This information can now be used to dramatically improve the performance of Method III, by reducing the size of the candidate binding lists. For example, suppose graph G has four immediate predecessor graphs H1, H2 , H3, H4. If we query graph G in Phase I of Method III we know that H1, H2 , H3 and H4 have all matched Q. This gives us a very good indication of how G may fit into (bind to) Q. For example, if node 1 of H1 mapped into node 5 of Q, and we know that this same node maps only into nodes 7 and 8 of G, we know that at least one of these nodes must bind to node 5 of Q - if G is to be a subgraph of Q. Further, constraints of the bindings of nodes 7 and 8 of G can be retrieved from the bindings of H1, H2, H3, H4, thus substantially reducing the candidate binding lists of G, before backtracking is taken up to resolve the remaining bindings. Note that in this framework it is necessary to find ALL bindings of each graph to the query graph(at least for insertion) and that finding them is facilitated by the method itself.

6 State-space search as relation-based hypergraph transformation

The state space search paradigm in which problems are broken down into initial conditions, terminal conditions (goals) and operators continues to be one of the most pervasive problem-solving models used in AI research [3] The fundamental importance of the paradigm is seen when one recognizes that each of automatic programming (in which the machine translates from one very high-level language into one less "high"), automatic theorem proving (in which one tries to prove formulas from axioms and inference rules), and pattern recognition (in which one parses from one representation into another) are special cases. We now show how all state space search problems as they are normally formalized [10] can be viewed as relation-based transformations over hypergraphs (which are a special case of conceptual graphs). In the next section we show how this relation-based view can allow these problems to be monitored incrementally and efficiently.

To reach a definition of generic search problem we consider notions of domain objects, static relations, dynamic relations, variables and bindings. The framework is inspired by Peirce's existential graphs[18], conceptual graphs, and our own work in experience-based planning [14].

The following is a description of the components of a search problem, with running examples taken from Tic-Tac-Toe [4] and Towers-of-Hanoi on three disks: [5]

- **Each domain will have a finite set of domain objects.** In Tic-Tac-Toe the objects will be squares {S1,S2,S3,S4,S5,S6,S7,S8,S9} and pieces {X,O,B (for blank)}.
  In Towers-of-Hanoi, the objects are Pegs{P1,P2,P3} and Disks {D1,D2,D3}.
- **Unary, binary and higher relations may be defined on these objects.** An n-ary relation is a set of n-tuples of domain objects. In the finite search domains, rather than defining types explicitly we shall simply note that they are implicitly defined as the set of objects that occur in any single field (attribute) of a relation. At the implementation level, relations that

[3] The objective of a state-space search problem is to find a sequence of operators (transformations) that will convert a state that satisfies the initial conditions into one that satisfies the terminal conditions, under various optimality criteria and resource constraints.

[4] Tic-Tac-Toe is played on a 3x3 board with two players X and O alternating turns selecting cells with X going first. The first player completing a row, column or diagonal wins.

[5] Towers-of-Hanoi is a single agent game involving three disks "small", "medium", "large" placed on top of each other in order on the first peg of three. The object is to move the disks one at a time such that at no time is a disk on top of a disk smaller than it on the same peg, and such that the disks finish all on the third peg.

are symmetric or transitive may be abbreviated by specifying a kernel set of tuples and then giving the mathematical properties from which the remaining tuples can be inferred. Other abbreviations and computation of relations are possible such as finding adjacent squares on the chessboard through calculation. We break relations into two classes: *static relations* are those whose definition (tuples) remain constant for a given game, and *dynamic relations* are those whose content can change from state to state. A frequent use of static relations is in defining board topology. Most domains have a dynamic relation corresponding to ON to say which pieces are on which squares.

In Tic-Tac-Toe, we define a "THREE-IN-A-ROW" static relation corresponding to winning sets of squares: { (S1, S2, S3), (S4, S5, S6),...} and the ON dynamic relation initialized as {(B,S1),(B,S2) etc. }. In Towers-of-Hanoi there is a static relation SMALLER-THAN initialized to:
{(D1,D2),(D2,D3),(D1,D3)}. Towers-of-Hanoi also has the dynamic relation ON, initialized to:
{(D3,P1),(D2,P1),(D1,P1)}.

- **Operators define transformations over states by changing the contents of dynamic relations. They are specified by giving sets of preconditions, additions and deletions.** Note that each of these sets (and their combination) can be viewed as a conceptual graph or hypergraph.

  For Tic-Tac-Toe we have an operator MOVE(piece,square): Pre: ON(B,S1), Add: ON(X,S1), Del: ON(B,S1). (we assume the board is always oriented with X to move.) For Towers-of-Hanoi we have the following MOVE operator:

```
MOVE(d1:disk,p1:peg,p2:peg) =
pre: ON(d1,p1) AND ~(p1=p2)  AND ~((ON(d2,p2) OR ON(d2,p1))
     AND SMALLER_THAN(d2,d1)).
add: ON(d1, p2)
del: ON(d1, p1)
```

- **States are hypergraphs over domain objects.** As the static relations are always true it is only necessary to give the dynamic relations when representing a state.
- **An operator is then applicable in a given state iff there is a 1-1 mapping in variables of the preconditions of the operator to domain objects such that all relations specified in the operator are true (or false, if negated) of those domain objects. The result of applying the operator is to remove from the current state those tuples corresponding to the deletions and add those tuples associated with the additions to the contents of the dynamic relations.**

  Since static relations are constant throughout the problem-solving process those bindings of objects to variables that could ever possibly (constrained by the static relations) satisfy an operator definition, can in principle be computed ahead of time leaving only the dynamic conditions to be checked.
- **Terminal conditions are defined in exactly the same way as preconditions, addition and deletion conditions of operators. It is automatically assumed that a player having no legal moves is terminal.** For tic-tac-toe, we have THREE-IN-A-ROW(s1,s2,s3) AND ON(X,s1) AND ON(X,s2) AND ON(X,s3) as terminal. In Towers-of-Hanoi, we have ON(D1, P3) AND ON(D2, P3) AND ON(D3, P3) as terminal conditions.
- **Reward conditions are conditions (often the same as terminal) that are coupled with a reward to each player based on the outcome of the game.**. For Towers-of-Hanoi and Tic-Tac-Toe, reward conditions are the same as terminals and assign a *win* to the player who has just moved.
- **Finally, FLIP is a static binary relation over domain objects used to define symmetries so that a game can be encoded from one player's perspective only.** For tic-tac-toe FLIP is:
  {(X,O),(O,X),(B,B),(S1,S1),(S2,S2)...}. Tower-of-Hanoi, being a single-agent game, does not require a FLIP operator.

- **In summary, an abstract game or search problem is defined as a finite set of domain objects, and finite sets of static relations, dynamic relations, operators, terminal conditions and reward conditions. Finally, for convenience in encoding, we define a symmetry condition (known as FLIP.)**

The conclusion is that a large spectrum of single-agent and multi-agent search problems can be viewed as games of (hyper)graph-to-graph transformation directly analogous to organic chemical synthesis: states are graphs, and operators are graph-to-graph productions, terminal and reward conditions may also be expressed as graphs. This conclusion is not surprising, given that conceptual graphs and other semantic network schemes have been shown to carry the same expressive power as first-order logic. **The conclusion is significant, however, in that it suggests the potential for graph-theoretic analysis of the rules of a domain and ensuing experience for uncovering powerful heuristics and decision-making strategies.** [16] It also suggests that state-space search can be monitored in a uniform manner, we take up this topic in the next section.

7 Monitoring state-space search incrementally using UDS

To claim universality for UDS we must show that it is an effective monitor and executor of the specifications of any given state space search domain. As it turns out, due to the relation-based perspective of UDS the following ideas from the RETE algorithm [7, 17] can be exploited with little adjustment to UDS as defined above:

- **The firing of an individual operator does not affect the current state radically.**
- **If an operator did not match in the previous cycle it most likely will not match in the current cycle.**
- **On each cycle we should only try to re-match operators that could have been affected by the previous operator application.**
- **Different operators may share a large amount of the same structure. Thus, separate conditions of operators should only be matched once per cycle.**
- **Variable bindings from cycle to cycle remain relatively consistent.**

UDS (our implementation is in C++) monitors search problems as follows:

The relations in the relational hierarchy are used to represent dynamic relations. Specific relations (tuples) that are true are stored beneath (as specifications) the schema declarations for the relations. A schema declaration and its tuples is equivalent to a table in a traditional relational database and we shall also use the term "table" in the following discussion. The preconditions of operators are stored using the graph hierarchy. Repeated parts of operators are only represented once in the relation hierarchy. Figure 7.1 depicts the initial UDS network for monitoring the Towers-of-Hanoi. There may be a difference between UDS and standard RETE implementations in that UDS exploits the relation-based representation of CGs to extend the types of patterns that can be matched and enhances the speed of doing so. However, **the important thing is that UDS naturally supports RETE in addition to a variety of other data manipulation methods appropriate to relational databases, CGs and semantic networks.**

Those dynamic relations that do not depend on any other relations in their definition are known as *primitive dynamic relations*. The post-conditions of operators work directly on the primitive dynamic relations (through pointers) by adding or deleting tuples from their contents. Static relations are compiled away at network generation time since by definition the set of tuples that satisfy them remains constant. Conceptual graphs representing the preconditions of operators are only re-matched if the content of one of their composing relations changes. Only that part of the conceptual graph affected by the change need be re-matched.

7.1 What happens after an operator is selected.

1. A selected operator corresponds to a tuple in one of UDS's operator tables, e.g. Move(D1,P1,P3) in Towers-of-Hanoi.

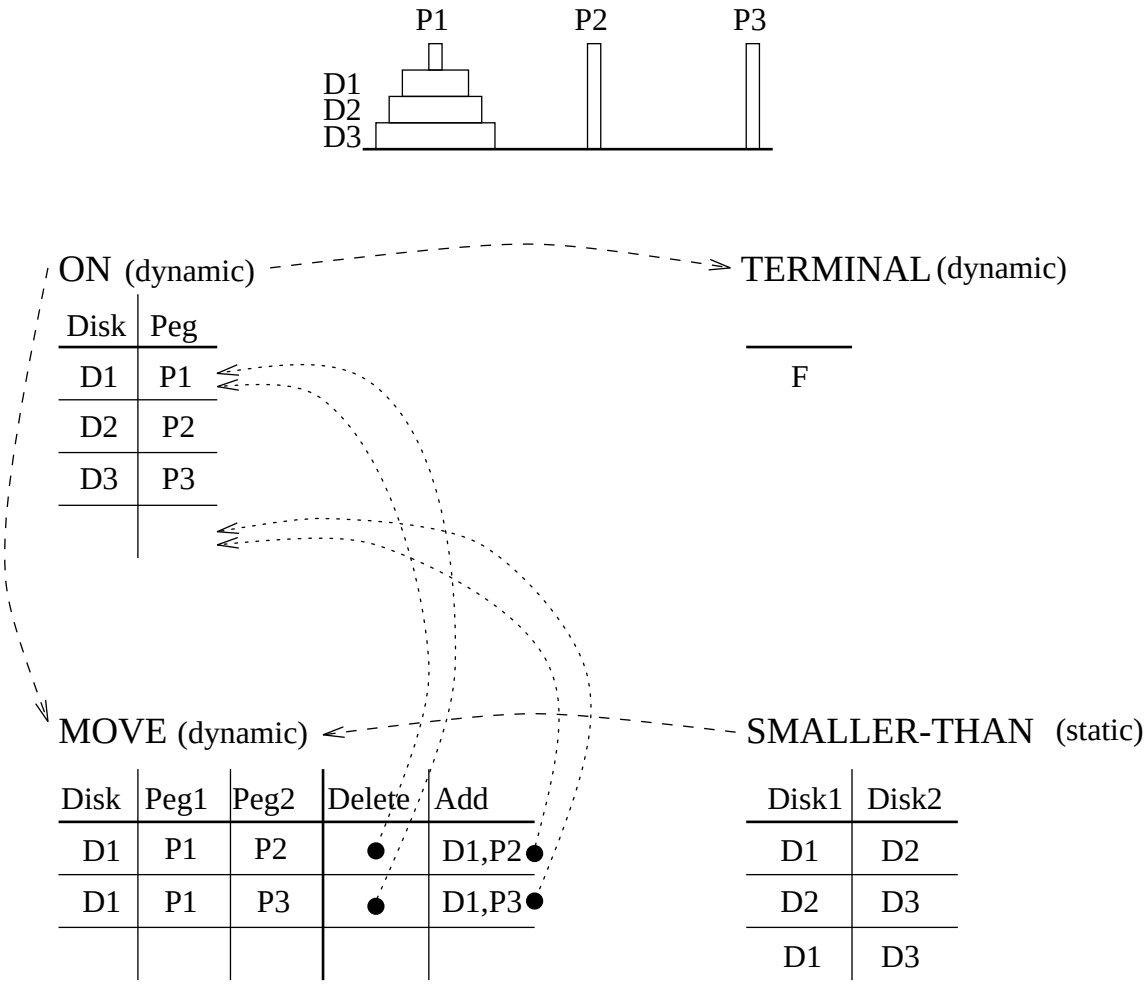


Figure 7.1: Initial state of UDS network for monitoring Towers-of-Hanoi

2. This tuple is bound to the given variable-arguments of that operator schema and lists of add and delete tuples are built based on the add and delete conditions of that schema. In our example of firing Move(D1,P1,P3): ON(D1,P1) will be deleted and ON(D1,P3) will be added.
3. These lists of tuples are then added and deleted immediately from the appropriate primitive dynamic relation tables in the net. Since the truth value of the net has been altered because of the immediate adding and deleting of the operator-generated tuples, these new changes are propagated up the net.
4. Each of the tuples in the add/delete lists created, are iterated thru for each table in the net that they may directly affect (these are their successors in the hierarchy) by changing the truth values of tuples in that table. If one of the added or deleted tuples match a pre-condition of a table, then a "re-join" procedure is called to update the tuples currently being stored in the table. Those tables not affected by the new add/delete tuples are ignored. Thus, in our example, the firing of Move(D1,P1,D3) removes (D1,P1,P2) and (D1,P1,D3) from the Move table and adds (D1,P2,P1) and (D2,P1,P3) to the Move table.

7.2 How Join works.

A relation table defined as the conjunction of two or more subtables can have its tuples calculated by taking a join of its subtables as in a relational database. In UDS we calculate the contents of a

table incrementally when one or more of its subtables have their contents changed (by deleting or adding tuples). Obviously, **Join is a critical operation and must be as efficient as possible. A join of an individual tuple is similar to a traditional subgraph-isomorphism test [1], but by working at the table (or schema) level a large number of subgraph bindings are found simultaneously, – illustrating another advantage of the UDS organization. Further advantages are accrued due to the incrementality.**

For each table T, Join is called once for each pre-condition table P that has changed.

Join works in two phases: If the current precondition table P has variables that have already been bound previously (on earlier preconditions in T's precondition list), then in Phase II, the system checks to see if the current tuples in T are valid given P's tuples and already bound variables, and removes any that are not. Otherwise (one or more variables are unbound) a *Merge* operation is done in Phase II based on iteration rather than Look-up of pre-condition Tuples.

For example, consider a table, T, to be updated which contains the following tuples: (s1, WP, S5), (s1, BN, S45), and (s1, WK, S4). Variables s2 and s3 have already been bound for all three tuples (constants are in uppercase) , while s1 remains unbound. Now suppose Join is iterating upon the following pre-condition argument P: ON(WP, s3). Then Phase I immediately determines that all variables in pre-condition ON(WP, s3) ({s3} in this case) are already bound, and so PHASE II will be a Look-up phase. In this phase, tuples in T will be iterated over, and by combining the bound variables with ON(WP, s3), the following subtuples will have there existence checked in ON: (WP, S5), (WP, S45) and (WP, S40). If the subtuple queried is not present in ON, the corresponding tuple of T will be eliminated.

Merge.

If a Merge is required in Phase II, then Join iterates through the tuples in T, binding each tuple to all possible variables in turn, then iterating through P's tuples, and building a new tuple based on comparing the current T tuple to the pre-condition tuple, seeing if currently bound variables match, and adding new objects to the unbound variables.

For example, again start with table T above. Suppose that join is iterating upon pre-condition argument: CAPTURE(s1, WP). We see that "s1", the only variable in the pre-condition, is unbound in T. Merge is then called to iterate over the CAPTURE tuples, stopping at any tuples where the second argument is "WP" (the constant) and will then bind "s1" to the first argument. Thus an "s1" bind with (for example) CAPTURE(S33, WP), would create new Tuples in T: (S33, WP, S5), (S33, BN, S45) and (S33, WK, S4). Note that "s1" can be bound more than once, and in this particular example, three new tuples would then be created each time a bind occurred.

7.3 Implementation and Extensions

Join, as defined above, is used to combine subtables that are connected through conjunction (AND). Tables that are combined through OR use an implicit UNION operation (as in relational databases). Since UNION simply involves including all tuples in each subtable in a new table , UNION is implemented by simply storing a list of subtables (rather than creating a new table). NOT is also simulated (as opposed to creating a new table of tuples) by returning exactly the opposite answers about tuple existence as its subtable.

Relation tables in our implementation of UDS are stored as hashtables over their tuples that also allow iteration (through a linked-list of buckets) over all tuples. Note that the pointers from the node hierarchy into the relation hierarchy can be used as indices to further facilitate the Look-Up and Merge operations of join. For example, the WP node will point to all tuples involving WP.

7.4 Performance results.

With this scheme, we have been able to monitor a variety of domains including Tic-Tac-Toe, Tower-of-Hanoi, 8-puzzle, and Hexpawn at a level of efficiency that is faster (in some cases up to

10 times) than previous programs of ours that had been written specifically for these domains but were not incremental.

In Figure 7.4 below we give initial timings for UDS in playing (at various search depths) and monitoring a variety of search domains that have been compiled into its tuple network from rule definitions as in Section 6. As UDS is domain-independent and not hard-coded for any of these domains it clearly should not be as fast as any domain-specific system. Still, through incrementality, its results, as shown, are reasonable and would likely beat (for domains with high branching factors) a hard-coded program that does not exploit incrementality. As this is an early implementation with several optimizations not yet completed, we expect even stronger results in the near future. The results reported are using a C++ implementation on a SUN SparcII with 64 megabytes of main memory.

8 Support for pattern matching

A common problem-solving operation used in case-based reasoning systems is to find all generalizations of previously seen situations that match a given state. This information is used to reason by analogy with the past, perhaps by retrieving the previous situations themselves.

Other systems such as Morph[8, 15] construct an evaluation function that is a combination of the weights of the most-specific patterns that match a state.

Both types of reasoning require matching a large database of patterns against the current state. Methods III-V were designed exactly with this operation in mind. However, until the integration of the ideas from RETE these algorithms were not necessarily as fast as they could be since they did not support incrementality (in which there is little change from one query to the next). **Thus the Morph chess system, using UDS, after a pawn moves need only rematch those patterns affected by the pawn move rather than starting from scratch by reconsidering its database of (up to 50000) patterns.** Such savings due to incrementality should provide great benefits in most inference settings.

9 Support for machine learning

Not only does UDS support retrieval, problem-space monitoring and pattern-matching, it also provides a structure that is highly convenient for developing machine learning algorithms for learning evaluation functions [16].

The multi-level hierarchical view of UDS shows directly how nodes are combined into relations and in turn how relations are combined into conceptual graphs. This hierarchical decomposition into wholes-and-parts is also fundamental in machine learning. **Much of the work in neural networks, statistics and pattern recognition deals directly with "how" the values of parts combine to create the value of the whole.** Such methods include gradient-descent, linear-regression and nearest neighbor algorithms [23]. UDS gives one the capability of employing these methods to produce a combining rule for each decomposition as one traverses the hierarchy.

For example, in chess one might like to determine the value of having a piece between the king and an attacking piece. In our declarative representation of the rules of chess, we have defined the BETWEEN-KING-ATTACKER relation as follows:

```
BETWEEN-KING-ATTACKER(SQUARE:s1,SQUARE:s2,SQUARE:s3;BLACKPIECE:p1,
WHITEPIECE:p2) = ATTACK(s1,s2) & ON(WK, s2) & BETWEEN(s1, s2, s3)
& MOVEABLE(s2, s3) & MOVEABLE(s1, s3) & ON(p1, s1) & ON(p2, s3).
```

This relation represents a conceptual graph composed of more primitive relations. Informally, the meaning is *a white piece p2 on s3 is between a white king on s2 and an attacking piece p1 on s1, if s3 is between s1 and s2 and it is possible to move p1 from s1 to s2 and then to s3.* The table for this relation will sit below (in the CG hierarchy) the tables for the relations it is composed of. The APSII domain-independent game-playing system [16] uses UDS to evaluate a BETWEEN-KING-ATTACKER tuple, say (10,11,12,br,wp) as follows:

MONITORING AND EXECUTION TIME FOR VARIOUS DOMAINS

SINGLE AGENT GAMES (IN MOVES TAKEN):

TOWERS-OF-HANOI

	100 Turns		500 Turns (With Game Restarts)		
	total	ave	total	ave	(min:secs).
R vs R	< :01	-	:02	0.004	
1 ply	:03	0.03	:15	0.03	
2 ply	:19	0.19	1:42	0.20	
3 ply	1:50	1.10	10:18	1.24	

8 PUZZLE

	100 Turns		500 Turns (Randomly, only 1 Game)		
	total	ave	total	ave	(min:secs).
R vs R	< :01	0.01	:02	0.004	
1 ply	:04	0.04	:22	0.04	
2 ply	:42	0.42	3:25	0.41	
3 ply	4:58	2.98	26:45	3.21	

DOUBLE AGENT (IN GAMES):

HEX PAWN

	100 Games		500 Games		
	total	ave	total	ave	(min:secs).
R vs R	:02	0.02	:10	0.02	
1 ply	:06	0.06	:30	0.06	
2 ply	:22	0.22	1:50	0.22	
3 ply	1:20	0.80	6:45	0.81	

TIC-TAC-TOE

	100 Games		500 Games		
	total	ave	total	ave	(min:secs).
R vs R	:03	0.03	:15	0.03	
1 ply	:09	0.09	:45	0.09	
2 ply	:55	0.55	4:50	0.58	
3 ply	6:55	4.15	-		

R vs. R means "random-player vs. random-player".
n-Ply means one agent is searching this deep, the other agent is playing randomly.

1. The tuple is looked up in a table of previously seen tuples (each relation table has a history table for learning purposes). If the tuple is found, the learned weight for the tuple is returned, else the system continues on to steps 2 and 3.

2. Subtables are called to get the weights of subtuples (based on the table's definition) of the original tuple. In our example, the weights of ATTACK(10,11), ON(WK,11), MOVEABLE(11,12), MOVEABLE(10,12), ON(BR,10), and ON(WP,12) are asked for. In general, such calls may themselves induce deeper recursive calls to subtables of subtables and so on, until tuples are found or we bottom out in the node tables that represent individual domain objects (and serve as arguments to the primitive dynamic relations). A bottom level table returns a default (neutral) value if a tuple is not found in its history table. Note that BETWEEN is not called on (10,11,12) since it is a static relation. [6]

3. Once the values of all subtuples are found, they are combined linearly using coefficients (the higher the coefficient the more accurate (with respect to feedback sent to the BETWEEN-KING-ATTACKER table) the subtable has been in the past) that have been learned for the table. The result of the linear combination is returned. In our example, suppose weights of 0.55,0.45,0.43,0.42,0.50 had been returned from the subtables. If the learned coefficients for the BETWEEN-KING-ATTACKER table are 0.3, 0.3, 0.2, 0.1, 0.1 (the coefficients are normalized to sum to 1) the value returned for the tuple (10,11,12,br,wp) will be 0.478.

10 Conclusion

Here we have shown how UDS can be used to combine the best features of semantic network marker passing retrieval, relational databases and conceptual graphs. A standout features of this new data structure is the degree to which it is able to simplify and compact the storage of a conceptual graphs database.

In addition to giving Method V as an improvement over Methods III-IV retrieval we have shown how UDS can be used to incrementally monitor state-space search domains (since they are based on transformations of conceptual graphs) and to support pattern-matching and machine learning.

Although there are details still to be worked out in how to best integrate the major database paradigms in UDS, we hope that the ideas presented here will lead to a further unification of currently disparate information retrieval and problem-solving methods.

Future directions are directed at the fact that in UDS as in its underlying components, query processing time is too heavily influenced by exactly *which* structures and joins are pre-stored in the database. We feel that a truly universal model should not be as susceptible to the original representation of the facts and should be able to reorganize itself to make better use of them. Along similar lines, the determination of key intermediate concepts remains a critical and fundamental issue in machine learning and one that we are studying.

11 Acknowledgements

This research was supported in part by NSF research grant IRI-9112862. Thanks to Don Roberts who provided diagrams and feedback during the construction of the paper. Thanks to Yuxia Zhang for her assistance with UDS and search-space definition. Finally, thanks to John Amenta who is largely responsible for the current UDS implementation, helped obtain the performance results, and supplied some algorithmic descriptions for the paper.

[6] The presence of a tuple in a static relation is invariant across positions and hence supplies no new information to base an evaluation on.

References

[1] J.M. Barnard. Problems of substructure search and their solution. In Wendy Warr, editor, *Chemical Structures the International Language of Chemistry,*. Springer-Verlag, 1988.

[2] H. Boley. Declarative operations on nets. *Computers and Mathematics with Applications*, 23(6-9):601–638, 1992. Part 2 of Special Issue on Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor. Also reprinted on pages 601–638 of the book, Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor, Pergammon Press, 1992.

[3] G. Ellis. Compiled hierarchical retrieval. In E. Way, editor, *Proceedings of Sixth Annual Workshop on Conceptual Structures)*, pages 187–208, SUNY-Binghamton, 1991. To Appear.

[4] G. Ellis. Efficient retrieval from hierarchies of objects using lattice operations. In G. Mineau and B. Moulin, editors, *Proceedings of First International Conference on Conceptual Structures (ICCS-93)*, Montreal, 1993. To Appear.

[5] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, California, 2 edition, 1994.

[6] S. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge.* MIT Press, Massachusetts, 1979.

[7] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object patern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[8] J. Gould and R. Levinson. Experience-based adaptive search. In R. Michalski and G. Tecuci, editors, *Machine Learning:A Multi-Strategy Approach*, volume 4, pages 579–604. Morgan Kauffman, 1994.

[9] J.A. Hendler. Massively-parallel marker-passing in semantic networks. In Fritz Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 277–292. Pergamon Press, 1992.

[10] R. E. Korf. Planning as search. *Artificial Intelligence*, 1987.

[11] R. Levinson. A self-organizing retrieval system for graphs. In *AAAI-84*, pages 203–206. Morgan Kaufman, 1984.

[12] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers and Mathematics with Applications*, 23(6-9):573–600, 1992. Part 2 of Special Issue on Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor. Also reprinted on pages 573–600 of the book, Semantic Networks in Artificial Intelligence, Fritz Lehmann, editor, Pergammon Press, 1992.

[13] R. Levinson and G. Ellis. Multilevel hierarchical retrieval. *Knowledge-Based Systems*, 5(3):233–244, September 1992. Special Issue on Conceptual Graphs.

[14] R. Levinson and K. Karplus. Graph-isomorphism and experience-based planning. In D. Subramaniam, editor, *Proceedings of Workshop on Knowledge Compilation and Speed-Up Learning*, Amherst, MA., June 1993.

[15] R. Levinson and R. Snyder. Adaptive pattern oriented chess. In *Proceedings of AAAI-91*, pages 601–605. Morgan-Kaufman, 1991.

[16] R.A. Levinson. Exploiting the physics of state-space search. In *Proceedings of AAAI Symposium on Games:Planning and Learning*, pages 157–165. AAAI Press, 1993.

[17] D. P. Miranker. Treat: A better match algorithm for ai production systems. In *Proceedings of AAAI-87*, pages 42–47, 1987.

[18] D.D. Roberts. The existential graphs. In *Semantic Networks in Artificial Intelligence*, pages 639–664. Roberts, 1992.

[19] E. Sciore. A complete axiomatization for join dependencies. *JACM*, 29(2):373–393, April 1982.

[20] J. F. Sowa. *Conceptual Structures.* Addison-Wesley, 1983.

[21] P. Suppes. Models of data. In *Logic, Methodology and Philosophy of Science*, pages 252–261. Stanford, East Lansing, 1962.

[22] J.D. Ullman. The u.r. strikes back. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 10–22, 1982.

[23] S. Watanabe. *Pattern Recognition:Human and Mechanical.* Wiley, New York, 1985.