A CG query engine based on relational power context families

Bernd Groh and Peter Eklund

Knowledge, Visualization and Ordering Laboratory
Griffith University
PMB 50, Gold Coast Mail Centre, QLD 9726, Australia
{b.groh, p.eklund}@gu.edu.au

Abstract. This paper presents a CG query engine which is based on the idea of power context families stored in a relational database. It describes the methodology used and aims to demonstrate how this methodology can replace the need of a CG subsumption engine with a simple query engine. It shows how ontologies are used as conceptual scales in order to achieve this goal. It further integrates the WebKB-2 ontology server, addresses the techniques adapted and presents an implementation of the engine in form of a C API.

1 Introduction

This paper reports on the integration of the WebKB-2 [6–11] ontology server[1] and relational power context families [3, 14]. Although a large part of the work has been done, the project is still in progress and the paper can only be considered work-in-progress. The functionality of the WebKB-2 ontology server has been implemented as a C API with a relational database backend. The ontology is stored, just like the knowledge base, as a relational power context family and is used automatically by the API functions. Once a graph is added to the knowledge base, the function not only stores the original graph, but all graphs which are more general. The ontology therefore takes the place of a conceptual scale in FCA terms. If the graph `[CAT: Tom]→(ON)→[MAT]` is added to the knowledge base and the ontology contains the information `CAT < ANIMAL` (meaning CAT is a subtype of ANIMAL, ANIMAL a supertype of CAT) then the graph `[ANIMAL: Tom]→(ON)→[MAT]` is automatically stored in the knowledge base, as well as any more general graph. This method has two disadvantages. First, the size of the knowledge base increases by a factor d, where d is the average depth of the ontology. As an example, d is around 7 for WordNet 1.6[2] hierarchy of nouns. Second, each update of the ontology requires an update of the knowledge base. This method has advantages, firstly, the query `[ANIMAL: ?]→(ON)→[MAT]` can be retrieved directly, as it is an explicit fact in the knowledge base. The subtypes of ANIMAL do not need to be investigated. `[ANIMAL: Tom]→(ON)→[MAT]` is a

[1] http://www.webkb.org/webkbShared.html
[2] http://www.cogsci.princeton.edu/~wn/

direct result of the query. The original graph and all more general graphs up to the query graph (in fact up to the top element) can be retrieved in one additional direct query. As a result, a subsumption engine, using this methodology, reduces to a simple query engine. Therefore, we anticipate an increase in speed. Furthermore, in this way it is possible to use a standard relational database to implement a conceptual graph subsumption engine. This has been done and is presented in Section 2 of the paper.

To test our hypothesis, two ontologies have been initialized, the WordNet 1.6 hierarchy for nouns and the WebKB-2 ontology. In addition, a parser is provided that allows the creation of a conceptual scale in the form of a relational power context family from the parsable[3] output of the WebKB-2 ontology server. A C API has been implemented that allows us to add and remove terms and relations between terms to and from the ontology, with automated update of the knowledge base, or alternatively the option of a later update through an explicit function call. The API also contains functions to add a conceptual graph to the knowledge base in linear form, automatically considering the ontology. This is shown in the following sections.

The final stage of this work is a performance comparison with WebKB-2 and an evaluation of the advantages and disadvantages of both systems. Also, a complexity analysis of the algorithms will be considered. This will be done in future work.

2 Methodology

As mentioned above, the methodology applied allows the reduction of a subsumption engine to a simple query engine. The subsumption is embedded in the knowledge base. An ontology can be used as a conceptual scale and is stored as a power context family in a relational database. Whenever new knowledge is added to the knowledge base the conceptual scale is considered. If, for example, the graph `[CAT: Tom]`→`(ON)`→`[MAT]` is added to the knowledge base the conceptual scale is used as follows – using the WebKB-2 ontology.

For the type **CAT** the formal context of the conceptual scale is shown in Fig. 1, the diagram of the concept lattice is shown in Fig. 3.

For the type **MAT** the formal context of the conceptual scale is shown in Fig. 2, the diagram of the concept lattice is shown in Fig. 4.

The relation is directly added to the context $\mathbb{K}_2$ as shown in Table 1. We do not use the unique names assumption for alphabetic strings of short length, such as *Tom*. In large knowledge bases it is unlikely that every *Tom* refers to the same individual. Therefore, further occurences of individuals, described by an alphabetic string will be hash-indexed to generated symbol names, for example, *Tom#1*, *Tom#2* and so on. This is explained in more detail in the next section.

As Figs. 3 and 4 show, there are several interpretations of the terms **CAT** and **MAT**.

[3] http://www.webkb.org/doc/F_languages.html

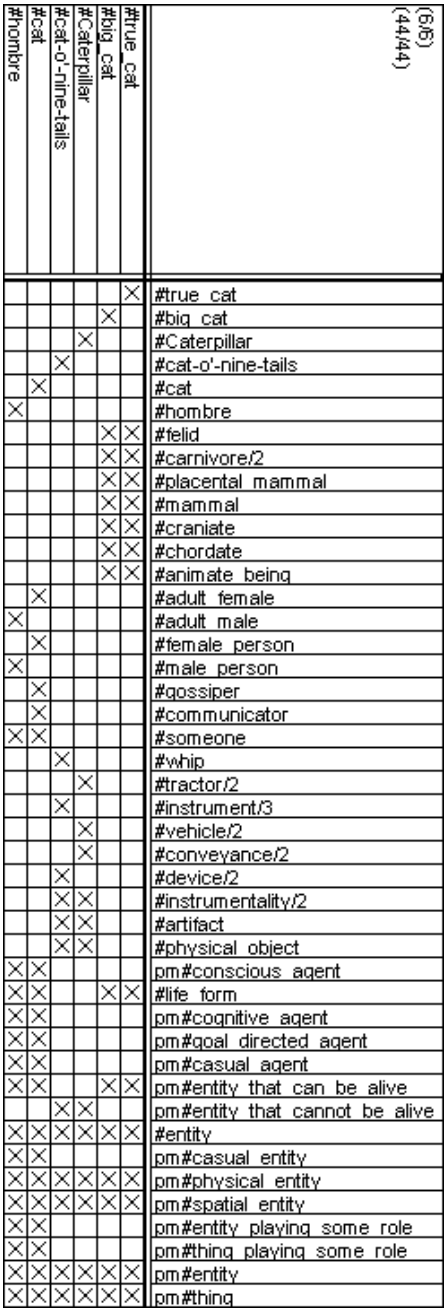

Fig. 1. Formal context of the conceptual scale for CAT.

Fig. 2. Formal context of the conceptual scale for MAT.

The options for **CAT** are:
1. #true_cat: *feline mammal usually having thick soft fur and being unable to roar; domestic cats; wildcats*; synonyms: cat

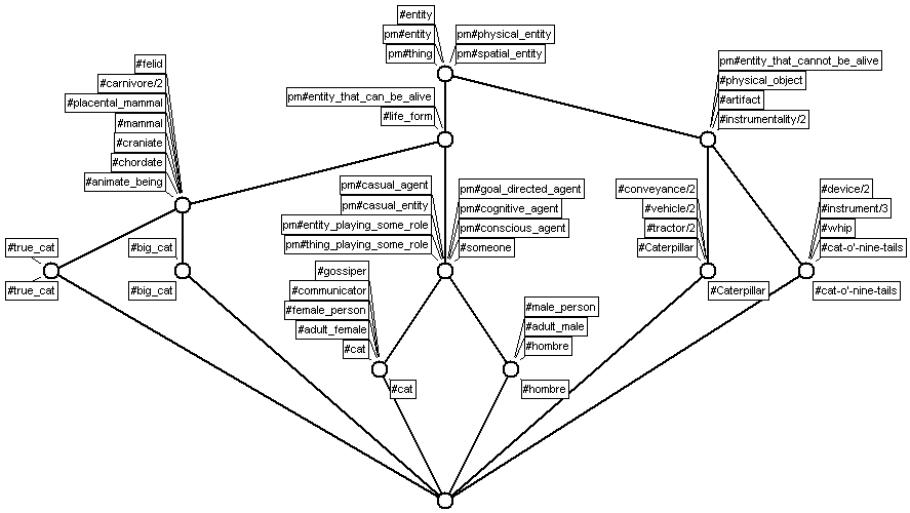

Fig. 3. Diagram of the concept lattice of the conceptual scale for CAT.

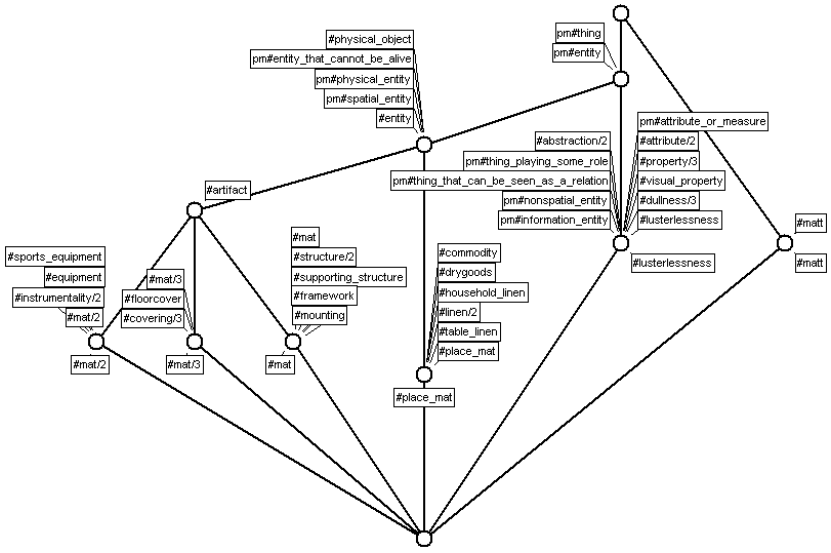

Fig. 4. Diagram of the concept lattice of the conceptual scale for MAT.

2. #big_cat: *any of several large cats typically able to roar and living in the wild*; synonyms: cat
3. #Caterpillar: *(trademark) a tractor that is driven by caterpillar tracks*; synonyms: cat

Relation	Instance1	Instance2
ON	Tom#127	#3487146

Table 1. Rows added to $\mathbb{K}_2$.

4. #cat-o'-nine-tails: *a whip with nine cords; "British sailors feared the cat"*; synonyms: cat
5. #cat: *a spiteful woman gossip; "what a cat she is!"*
6. #hombre: *an informal term for a youth or man; "a nice guy"; "the guy's only doing it for some doll"*; synonyms: guy, cat

The options for **MAT** are:
1. #mat/2: *a piece of thick padding up on the floor for gymnastic sports*
2. #mat/3: *a thick flat pad used as a floor covering*
3. #mat: *a border or background for a picture*; synonyms: matting
4. #place_mat: *table linen for an individual place setting*; synonyms: mat
5. #lusterlessness: *the property of having little or no contrast*; synonyms: flatness, mat, matt, matte
6. #matt: *not reflecting light; not glossy; "flat wall paint"; "a photograph with a matte finish"*; synonyms: flat, mat, matte, matted

There are several approaches to restricting choice against these options. One could define rules over relations, this is how it is done in WebKB-2. It is known, for example, that the relation **ON** requires spatial entities as instances. Therefore, the options *#lusterlessness* and *#matt*, given in Fig. 4, cannot apply in conjunction with the **ON** relation. The next section addresses this issue. At the moment the first meaning is used, with a warning message to the user that there have been multiple alternatives and that the first has been selected. The next section addresses alternative options.

To add the types, the conceptual scales in Fig. 3 and Fig. 4 are used, not to select the specific meaning of the term, but to add the specific term to the knowledge base. Not only with the attribute at the specific node, but with the intent. The rows added to $\mathbb{K}$ are shown in Table 2.

The disadvantage of this method is that the knowledge base increases in size by a factor d, the average depth of the ontology. For the WebKB-2 ontology this factor is greater than 7. Another disadvantage is that whenever the ontology is updated, the knowledge base has to be updated as well. The advantage here is that the subsumption engine over the knowledge base reduces to a simple query engine. A query `[ANIMAL: ?]→(ON)→[MAT]`, which is equal to `[#animate_being: ?]→(ON)→[#mat/2]` would directly retrieve `[ANIMAL: Tom#127]→(ON)→[MAT]`.
It wouldn't be necessary to investigate all subtypes of ANIMAL. As most ontologies are a tree-like partially ordered structure – a term has many more subtypes than supertypes – it is reasonable to believe that storing all supertypes within the knowledge base is generally more efficient than having to investigate all sub-

Type	Instance
#true_cat	Tom#127
#felid	Tom#127
#carnivore/2	Tom#127
#placental_mammal	Tom#127
#mammal	Tom#127
#craniate	Tom#127
#chordate	Tom#127
#animate_being	Tom#127
#life_form	Tom#127
pm#entitiy_that_can_be_alive	Tom#127
#entity	Tom#127
pm#physical_entity	Tom#127
pm#spatial_entity	Tom#127
pm#entity	Tom#127
pm#thing	Tom#127
#mat/2	#3487146
#sports_equipment	#3487146
#equipment	#3487146
#instrumentality/2	#3487146
#artifact	#3487146
#physical_object	#3487146
pm#entity_that_cannot_be_alive	#3487146
#entity	#3487146
pm#physical_entity	#3487146
pm#spatial_entity	#3487146
pm#entity	#3487146
pm#thing	#3487146

Table 2. Rows added to $\mathbb{K}$.

types within the query. This has the effect of an increase in size of the knowledge base compared to a decrease in query response time and performance scalability. Anecdotally, storing the type hierarchy in this way, is the way it has been done in FCA for many years and has proven to be a good solution. Any relational database with an ODBC driver can be used to implement a CG subsumption engine, no specifically designed database or knowledge base is required.

3 Ontology and Knowledge Base

It is possible to create your own ontology, all you have to do is call a specific C API function to add and remove terms or add and remove relations between terms. The functions allow us to either say whether registered knowledge bases should be updated immediately – by the function – or not. In this case another function has to be called later on to bring registered knowledge bases up to date.

Two ontologies have been initialized. The WordNet ontology, which contains the entire WordNet 1.6 hierarchy for nouns, clean from cycles and self-referents, but using WordNet terminology. The second is the WebKB-2 ontology. This includes the WordNet hierarchy for nouns, but adapted to a certain degree using a somewhat different terminology[4]. Nevertheless it is more suitable for working with CG's. Section 5 talks in more detail about the WebKB-2 ontology, as it is used for a comparison of both systems.

A parser is also provided, explained in the next section, that automatically creates an ontology in form of a relational power context family from the parsable output format to obtain from the WebKB-2 ontology server.

Both, the ontology and the knowledge base are stored in relational power context families as Tables 1 and 2 show. For the ontology a row is displayed in Table 3.

Relation	**Instance1**	**Instance2**
$<$	#true_cat	#felid

Table 3. Row of the ontology with a subtype relation.

To have both in the same format also has the advantage that both can be investigated in the same way by other tools, either by Toscana [13] or by possible extensions of Toscana [4, 5], or by other tools, like CEM [1], understanding that specific database format. The only difference between the ontology and the knowledge base is how they are used by the C API functions.

If a term is added to the ontology, this is done in exactly the same way as adding a term to the knowledge base. The difference is that the function to add a term to the ontology has the option of updating the knowledge base, while the function to add a term to the knowledge base uses information from the ontology.

In our example `[CAT: Tom]`→`(ON)`→`[MAT]`, the following would happen. First the function splits the graph into tokens and uniquely identifies individual referents. We do not use the unique names assumption, as it is unlikely in large databases that names, using alphabetic strings of short length, like *Tom*, would uniquely refer to the same individual, as mentioned before. Therefore a function is provided to explicitly express that one term is identical to another and identical names are hash-indexed. If the *Tom* with the highest index is *Tom#126*, the individual *Tom* would be converted into *Tom#127*. Later on all *Tom#127* could be converted into *Tom* or *Tom#33*. This already shows the advantage an interactive version would have. All Tom's could be listed and displayed to the user and s/he could select the specific *Tom* with a mouse click, or remain undecided.

[4] The precise content is described by Martin and Eklund [11] (ICCS 2001) and also methodologically by Martin [8] (ICCS 2000).

Individual referents which are not given, will be created in the same manner, i.e., as was the case with **MAT**. If the highest index is *#3487145* the next index is set to *#3487146*. The ontology does not contain a hierarchy for relations, only for types. Therefore the relation can be added directly to the knowledge base, as Table 1 shows.

Next, types have to be uniquely identified. From the ontology, all types with the name **CAT**, or that are synonym to **CAT** will be retrieved. That is *#true_cat*, *#big_cat*, *#Caterpillar*, *#cat-o'-nine-tails*, *#cat* and *#hombre* in that order. Here another advantage of an interactive system becomes clear. The user could be given all the possible types with explanation to make a selection, or s/he could be presented with the diagram in Fig. 3 and then click the specific concept to make a selection. In an API we have to rely on more similar matters, like selecting the first term and return a warning to the user about our selection and the multiple alternatives. An API requires a more specific input, like `[#big_cat: Tom]→(ON)→[MAT]` , if *Tom* should happen to be a big cat. The same goes for **MAT**. All types with the name **MAT** or synonym to **MAT** will be retrieved from the ontology. That is *#mat/2*, *#mat/3*, *#mat*, *#place_mat*, *#lusterlessness* and *#matt* in that order. Also here, the first term will be selected and a warning will be returned to the user. Our initial graph `[CAT: Tom]→(ON)→[MAT]` is therefore equal to `[#true_cat: Tom]→(ON)→[#mat/2]` , even though most people are more likely to associate it with `[#true_cat: Tom]→(ON)→[#mat/3]`. This is something that has to be done interactively and is therefore more a software and user-interface development issue than part of our current research. An interactive tool could however use the diagrams in Figs. 3 and 4.

There is one other approach, not to select, but to further restrict possible types. A set of rules that defines signatures to relations has been applied in WebKB-2. We know, for example, that both referents of the relation **ON** have to be spatial entities, as mentioned before. This is defined in the WebKB-2 ontology as follows:

```
ON(spatial_entity,spatial_entity);
```

But we might also know a lot more about cats on mats. Of course those rules cannot be rules to cover the whole world, but to cover a certain context. Such sets of rules are currently being considered, but are more an aspect of future work.

Once types have been uniquely identified, all the supertypes of those types will be retrieved. *Tom#127* will be added as an instance of all supertypes of *#true_cat* and *#3487146* will be added as an instance of all supertypes of *#mat/2* as Table 2 shows.

4 Parser

The parser has been developed using Lemon[5]. It allows the parsing of the WebKB-2 ontology server parsable output format, as well as the linear CG

[5] http://www.hwaci.com/sw/lemon/

notation. It can easily be updated to parse other formats, like CGIF. This shows another advantage of the ontology and the knowledge base having the same format, you can use the same parser. You could also parse the ontology in linear CG format.

The WebKB-2 parsable output looks as follows (truncated to fit the page):

```
#true_cat__cat__true_cat (^feline mammal usually having thick soft fur an
 <  #felid,
 >  #domestic_cat  #wildcat,
 N #feline;

   #felid__feline__felid (^any of various lithe-bodied round-headed fissi
    >  #true_cat  #big_cat,
    p #paw,
    M #Felidae;

      #carnivore/2 (^terrestrial or aquatic flesh-eating mammal; terrestr
       >  #fissiped_mammal  #canid  #felid  #bear/2  #viverrine  #musteli
       M #Carnivora,
       N #carnivorous/a2;

         #placental_mammal__placental__placental_mammal__eutherian__euthe
          >  #livestock  #bull/3  #cow/2  #yearling/2  #buck  #doe  #inse
          M #Eutheria;

            #mammal (^any warm-blooded vertebrate having the skin more or
             >  #female_mammal  #prototherian  #metatherian  #placental_m
             p #coat/2 #hair/2,
             M #Mammalia;
```

Any given output can be parsed and can form an ontology in form of a relational power context family. Instead of having to define ontologies for different contexts, WebKB-2 can be used to produce specific ontologies and then the parser can be used to create the relational power context families from it.

The lemon-grammar for the WebKB-2 output looks as follows:

```
ontology ::= expressions.
expressions ::= expressions expr.
expressions ::= .
expr ::= term description date relationlist SEMI.
 { addSignature(); }
term ::= id signature.
id ::= ID(a).
 { setId(a->str_val()); }
signature ::= LEFT ID(a) COMMA ID(b) RIGHT.
 { setSignature(a->str_val(), b->str_val()); }
signature ::= .
```

```
description ::= DESCRIPTION(a).
 { setDescription(a->str_val()); }
description ::= .
date ::= DATE(a).
 { setDate(a->str_val()); }
date ::= .
relationlist ::= relation targets relationsublist.
relationlist ::= .
relationsublist ::= relationsublist COMMA relation targets.
relationsublist ::= .
relation ::= RELATION(a).
 { setRelation(a->kind()); }
targets ::= targets target.
targets ::= targets startexclusion targetset endexclusion.
targets ::= .
startexclusion ::= LLEFT.
 { setExclusion(); }
endexclusion ::= RRIGHT.
 { resetExclusion(); }
targetset ::= targetset target.
targetset ::= .
target ::= ID(a) creator.
 { addRelation(a->str_val()); }
creator ::= LEFT ID RIGHT.
creator ::= .
```

The advantage of LEMON is that you can call functions within the grammar. The functions used above are API functions which are connected to the ODBC API. On startup an SQL statement is initialized as follows:

```
INSERT INTO [K2] (Relation, Instance1, Instance2) VALUES (?, ?, ?);
```

The question-marks will be associated with variables. The above *set*-functions set those specific variables, while *add*-functions execute the statement. Whatever values are in the variables on execution, are added to the knowledge base. In this way the parser can be adapted to different input formats, only by defining the grammar of the input format and the call of the specific API functions, when they have to be applied. This is the same way the parser for the linear CG notation is written and this is the way it could be done for CGIF, Frame-CG or other formats, using the existing API.

Using this parser, in combination with WEBKB-2, is a good and easy way to create user-defined ontologies.

5 Comparison with WEBKB-2

The final stage of our work is a comparison to the WEBKB-2 ontology server. Therefore, the WEBKB-2 ontology server has to be presented in more detail.

WebKB-2 is initialized with the natural language ontology WordNet and has its own top-level ontology. As to see in Figs. 1, 2, 3 and 4 as well as Table 2, the ontology contains types that start with # and types that start with *pm#*. The id before the # refers to the user who created the term. The creator is *wn* by default. All types starting with # are types of the WordNet ontology, all types starting with *pm#* are types of the top-level ontology, created by **P**hilippe **M**artin. Every user gets a unique id, that is exactly how it is realized in our work. The ontology contains 94,500 nouns, 66,000 categories referred to by nouns, 21,000 adjectives and 7,900 categories referred to by adjectives. As in our work, the domain ontology is a large database and new categories can be added at any time. This being the case, the ontology cannot be stored directly within the schema of an object-oriented database. WebKB-2 uses a GPL licensed object-oriented main-memory database system called FastDB[6] with a C++ API. We use a standard relational database via a ODBC connection with a C API. We have to expect that WebKB-2 will outperform our approach, as the database is a main-memory database system and faster than a standard relational database, which is disk-based. There is, however, a disk-based database version called GigaBASE that can be used with the same API. We are currently investigating that option to have a more meaningful comparison. The comparison to the FastDB version will be done for completeness.

There are two main points to the comparison of the two approaches:
1. Graph Search
2. Subsumption

Subsumption is necessary for graph search via generalization and specialization. WebKB-2 allows the minimal common supertypes of two categories to be returned from the ontology server. The minimal common supertype of **#true_cat** and **#mat/2**, for example, is **#entity**. The minimal common supertype of **#true_cat** and **#cat** is **#life_form**, as Fig. 3 shows. In our approach, subsumption does not differ from directed graph search, as any given common supertype is stored as direct fact within the knowledge base, like the type itself.

To compare the two approaches, both systems have to be initialized with the same knowledge base. For the graph search comparison the international flight database of an Australian airline has been chosen, namely QANTAS. It contains all flights of that specific airline and all associated airlines as well as served airports and information about airports and flights. Other test-sets are currently being investigated, especially for the comparison of the graph search via generalization and specialization. A larger variety of types and placement within the ontology is more desirable here than for direct graph search.

Testing or benchmarking the two knowledge servers is considered at two levels. In the first we measure the speed and timings to compute subsumption. In the second, we measure the speed that graph specialisation is performed.

[6] http://www.ispras.ru/~knizhnik/fastdb.html

Two sets of 500 leaf types from WordNet 1.6 will be selected and stored. The first 500 categories will then be exhaustively compared to each of the 500 categories in the second set and the least common supertype computed for each. This will be timed for each of the knowledge servers.

For graph specialisation, 500 flights routings will be synthesized from the QANTAS flights database, these routings will be constructed to represent routings that are known to exist. These will be formated as CGs and run against each of the knowledge servers. Timings are then collected. As much as we wish to measure the performance of WebKB-2 and our approach against known graphs, it is also necessary to measure the time each takes to determined that a graph does not exist in the KB. We do this by randoming changing one of the concepts in the positive routings test set to a concept known not to exist in the knowledge base. i.e. change the name of any occurance of a city like "Sydney" to a misspelling such as "Cydney". This substitution generates a negative test set of CGs known to be absent from the flights knowledge base. Timings are then collected for this negative set against each of the knowledge servers.

Further, it is yet to be decided on the system on which the comparison will run. WebKB-2 is currently running on Sun Solaris 2.7, while our approach is developed under Microsoft Windows NT. To port our system to Unix, we will have to install a ODBC environment on Unix that provides us with the required drivers and the ODBC API. A relational database is also required to run on that platform. The unixODBC[7] project provides the ODBC API and several database-drivers. The next step is to test on which Unix platform our system will run without requiring any changes. Sun Solaris 2.7 is our first choice.

Once the test-bed has been set up, the comparison of the two systems and an evaluation of how the systems perform in comparison to each other will be done. This work is still in progress.

References

1. R. Cole, G. Stumme: CEM: A Conceptual Email Manager. In: B. Ganter, G. Mineau (eds.): *Proceedings to the 8th International Conference on Conceptual Graphs*. LNAI **1867**. Springer, Heidelberg 2000.
2. B. Ganter, R. Wille: *Formal Concept Analysis: Mathematical Foundations*. Springer, Heidelberg 1999. (Translation of: *Formale Begriffsanalyse: Mathematische Grundlagen*. Springer, Heidelberg 1996.)
3. B. Groh, P. Eklund: Algorithms for creating relational power context families from conceptual graphs. In: W. Tepfenhart, W. Cyre (eds.): *Conceptual Structures: Standards and Practices*. LNAI **1640**. Springer, Heidelberg 1999, 389–400.
4. B. Groh, S. Strahringer, R. Wille: TOSCANA-systems based on thesauri. In: M.-L. Mugnier, M. Chein (eds.): *Conceptual Structures: Theory, Tools and Applications*. LNAI **1453**. Springer, Heidelberg 1998, 127–138.

[7] http://www.unixodbc.org

5. P. Eklund, B. Groh, G. Stumme, R. Wille: A Contextual-Logic extension of TOSCANA. In: B. Ganter, G. Mineau (eds.): *Proceedings to the 8th International Conference on Conceptual Graphs*. LNAI **1867**. Springer, Heidelberg 2000.
6. P. Martin, P. Eklund: Embedding Knowledge in Web Documents. In: WWW8, 8th International World Wide Web Conference, Toronto, Canada (1999).
7. P. Martin, P. Eklund: Embedding Knowledge in Web Documents: CGs versus XML-based Metadata Languages. In: ICCS'99, 7th International Conference on Conceptual Structures, Springer Verlag, LNAI 1640 (1999) 230–246.
8. P. Martin: Conventions and Notations for Knowledge Representation and Retrieval. In: ICCS'00, 8th International Conference on Conceptual Structures, Springer Verlag, LNAI 1867 (2000) 41–54.
9. P. Martin, P. Eklund: Knowledge Indexation and Retrieval and the Word Wide Web. IEEE Intelligent Systems, special issue Knowledge Management and Knowledge Distribution over the Internet", 18– 25, May/June 2000.
10. P. Martin, P. Eklund: Conventions for Knowledge Representation via RDF. In: WebNet2000, San Antonio, Texas, ACCE press, 378–383, November, 2000.
11. P. Martin, P. Eklund: Large-scale Cooperatively build Knowledge Bases. (Submitted to ICCS 2001)
12. J. F. Sowa: *Conceptual structures: Information processing in mind and machine*. Adison-Wesley, Reading 1984.
13. F. Vogt, R. Wille: TOSCANA – A graphical tool for analyzing and exploring data. In: R. Tamassia, I. G. Tollis (eds.): *GraphDrawing '94*. LNCS **894**. Springer, Heidelberg 1995, 226–233.
14. R. Wille: Conceptual Graphs and Formal Concept Analysis. In: D. Lukose, H. Delugach, M. Keeler, L. Searle, J. F. Sowa (eds.): *Conceptual Structures: Fulfilling Peirce's Dream*. LNAI **1257**. Springer, Heidelberg 1997, 290–303.