1

EFFICIENT EVALUATION OF VISUAL QUERIES USING DEDUCTIVE DATABASES

Dimitra Vista* and Peter T. Wood**

** Department of Computer Science*
University of Toronto
Toronto, Canada M5S 1A4

*** Department of Computer Science*
University of Cape Town
Rondebosch, South Africa 7700

ABSTRACT

We report on the efficiency of queries formulated with the Hy$^+$/GraphLog data visualization system, a substantial application which utilizes the CORAL deductive database system. Hy$^+$ itself has been used in a number of application areas, including software engineering, network management, and the debugging of distributed programs. We focus on the translation of GraphLog queries to CORAL programs as well as the performance of the resultant programs on large data sets. One source of inefficiency in programs involving recursion and query constants is that CORAL is not able to detect that the programs are factorable. We provide an alternative translation which always propagates constants to base relations, with a consequent improvement in performance.

1 INTRODUCTION

Hy$^+$ is a data visualization system under development at the University of Toronto. To visually represent and manipulate large structured data, Hy$^+$ introduces a common graph-like representation for databases, queries, and their results referred to as a hygraph (a hybrid between hypergraphs [2] and higraphs [12]). Queries in Hy$^+$ are formulated in GraphLog, the visual query language that the system supports and uses. The system has been used successfully in applications such as software design [8], network management [3], and debugging of distributed programs [4].

Query processing in Hy$^+$ is performed by translating GraphLog queries (and data, if necessary) into logic programs suitable for execution by one of three backends: CORAL, LDL, or Prolog. In this paper, we concentrate on the efficiency of queries executed on the CORAL backend [17]. Details of the translation from GraphLog without aggregation to Prolog can be found in [11], while that for LDL is described in [9]. In addition, we do not address the features of negation and aggregation which GraphLog also provides [9].

The efficiency of evaluating GraphLog queries is affected significantly by the choice and sophistication of the translation from GraphLog to CORAL. For example, with a careless ordering of subgoals by the present GraphLog translation, a query on a database representing relationships among functions and classes in the NIH public domain C++ class library (9124 tuples) can take over 3.5 minutes to execute, even using the `@factoring` annotation provided by CORAL. With the correct ordering of subgoals, this can be reduced to around 31 seconds. In this paper, we provide an alternative translation for GraphLog which reduces the execution time, in this case, to under 1.5 seconds.

Although our emphasis is on translating GraphLog queries, the techniques we develop can be applied independently as a rewriting technique for a restricted class of Datalog programs. In addition, we comment on other issues such as subgoal ordering and the efficiency of various annotations with which one can control the evaluation mechanism chosen by CORAL.

The outline of the rest of the paper is as follows. In the next section, we give a brief overview of the Hy$^+$ system. This is followed, in Section 3, by a description of the GraphLog query language, along with the current technique for translating GraphLog to CORAL. Section 4 is devoted to describing the alternative translation scheme, while Section 5 provides a comparison of the efficiency of the two techniques, using a number of sample queries and databases. Conclusions and directions for further work are discussed in Section 6.

2 AN OVERVIEW OF HY$^+$

In this section, we provide a brief description of the Hy$^+$ system; a more complete description can be found in [7]. Hy$^+$ is both a data visualization system and a visual query system. Data, in the form of tuples or Prolog facts, can be visualized as a labelled, directed graph or, more generally, as a hygraph which permits visual clustering, nesting and hiding of data. The graph layout can be specified by the user choosing from a comprehensive set of layout algorithms implemented in the Hy$^+$ system. In addition, the user can modify the layout interactively through a browser window.

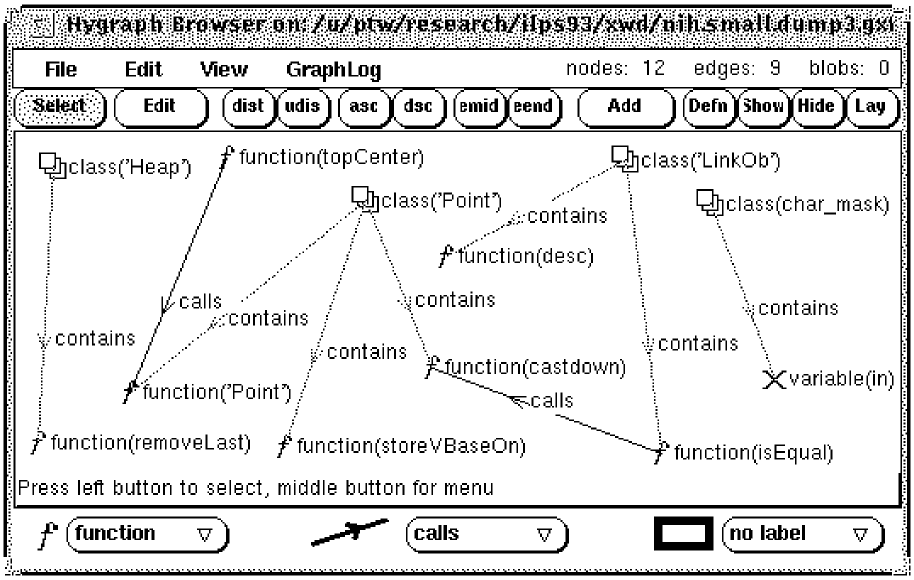

Figure 1 Visualizing a small portion of the NIH class library.

As an example, Figure 1 shows a small portion of the structure of the NIH public domain C++ class library, visualized as a labelled, directed graph. The objects of interest, represented by nodes, are classes, functions and variables. Relationships include the subclass and friend relationships between classes, the contains relationship between a class and its member functions, a reference relationship between functions and variables, and a calls relationship between functions. Icons for the various nodes can either be selected from the existing list or can be designed by the user. In addition, different colours can be associated with different edge labels in order to make the visualization more informative.

An attempt to visualize the entire NIH class library (9124 tuples) would result in an unintelligible mass of edges. This classic problem in data visualization is addressed by two classes of queries provided by the GraphLog visual query language. The first is that of *define* queries, which permit the definition of new relationships among the data. The second class is that of *filter* queries, which allow the selective filtering of relevant information.

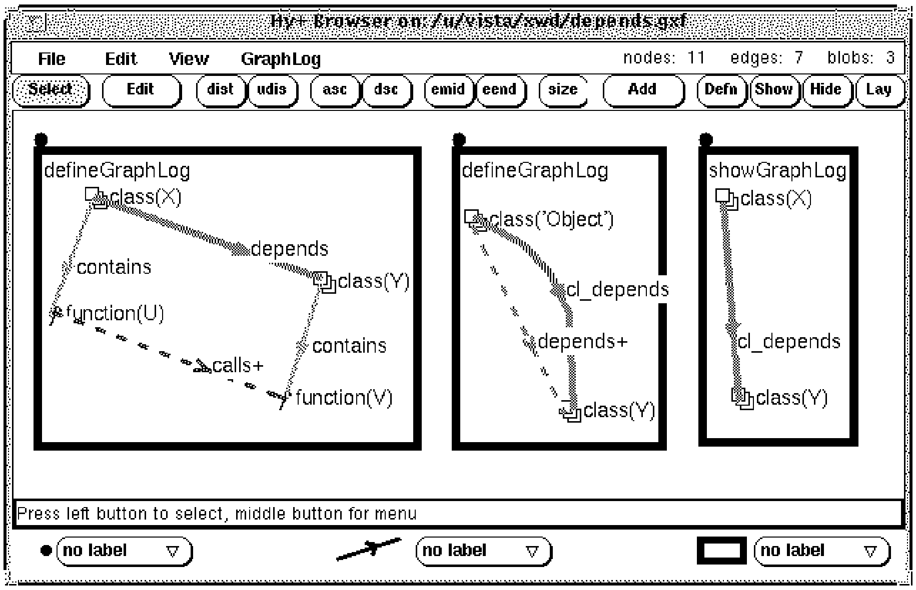

Figure 2 A GraphLog query on the database of Figure 1.

A GraphLog query on the graph of Figure 1 is shown in Figure 2. The query comprises three subqueries, as reflected in the three boxes (or *blobs*) of the figure. The lefthand query defines a relationship *depends* (indicated by the boldface, or *distinguished*, edge) between classes, such that class X depends on class Y if X contains a member function which directly or indirectly calls a member function of Y. The edge labelled $calls^+$ is interpreted as asking for a sequence of edges labelled *calls* in the graph being queried. The middle query then asks for the transitive closure, *cl_depends*, of the newly defined *depends* relationship, restricted to those classes which depend directly or indirectly on the class "Object." The righthand *filter* query (in the *showGraphLog* box in Figure 2) specifies that the user wants to see only these transitive edges.

The GraphLog query of Figure 2 is translated to Datalog and evaluated by CORAL. The answer is a set of *cl_depends* tuples which is presented to the user as a directed labelled graph. At first glance it may appear that *cl_depends* contains no more tuples than *depends* itself. However, assume that class c_1 has member function f_1 which calls function f_2 of class c_2, and that class c_2 has member function f_3 which calls function f_4 of class c_3. Then tuple (c_1, c_3) will be in *cl_depends*, while it may not appear in *depends* itself.

Although not utilized in the above example, filter queries can include more than one distinguished edge, thereby allowing the user to view collections of subsets of the data. In addition, we have not shown the full visual power of hygraphs in the above example. For these, and other features of Hy$^+$, the reader is referred to [7].

3 THE GRAPHLOG VISUAL QUERY LANGUAGE

GraphLog is a graphical formalism for visual manipulations of database visualizations [6]. It is suitable for applications of database technology where the data has a graph-like structure which can be exploited to provide visual presentations of the data [5, 8].

3.1 Syntax and Semantics

GraphLog visualizations are based on the notion of hygraphs. Hygraphs are directed labelled graphs which, in addition to labelled nodes and edges, also contain labelled *blobs*. A *blob* in a hygraph generalizes the concept of an edge: it is a relation from a node, called the *container* node, to a set of other nodes, called the *contained* nodes. It can be used to group similar objects together. Visually, it is represented as a rectangular box associated with the container node. All contained nodes are displayed inside the container's box.

In GraphLog, a *term* is either a constant, a variable, an anonymous variable (as in Prolog), or a function f applied to a number of terms. An *edge* or *blob label* is an expression generated by the following grammar, where $\bar{T}$ is a sequence of terms and p is a predicate:

$$E \rightarrow E|E;\ E \cdot E;\ -E;\ (E);\ E+;\ E*;\ p(\bar{T});\ \neg p(\bar{T})$$

This is essentially a grammar for regular expressions, with "|" representing alternation, "·" concatenation, "−" inversion (a traversal from head to tail, rather than tail to head), "¬" negation (the absence of a path), "+" transitive closure, and "∗" reflexive transitive closure.

Database instances are hygraphs whose nodes are labelled with ground terms and whose edges and blobs are labelled with ground predicates. Database instances of the object-oriented or relational model can easily be visualized as hygraphs. For example, an edge (blob) labelled $p(\bar{X})$ from a node labelled T_1 to a node (containing a node) labelled T_2, corresponds to tuple $(T_1, T_2, \bar{X})$ of relation p in the relational model. In the object-oriented model, such an edge could correspond to method p of object T_1 which results in T_2 when applied to T_1 with arguments $\bar{X}$.

Queries are hygraphs whose nodes are labelled by terms and each edge (blob) is labelled by an edge (blob) label. As mentioned in the previous section, there are two types of queries: *define* and *filter*. In both types, the query hygraph represents a pattern; the query evaluator searches the database hygraph for all occurrences of that pattern. The difference between the two types of queries stems from their interpretation of *distinguished edges*, explained below.

A *define* query must have one *distinguished* edge or blob, labelled by a positive literal. The meaning of a *define query* is to define the predicate in this distinguished literal in terms of the rest of the pattern. The semantics of *define* queries is given by a translation to stratified Datalog. Each *define* hygraph G translates to a rule with the label of the distinguished edge or blob in the head, and as many literals in the body as there are non-distinguished edges and blobs in G. If an edge is labelled with a regular expression, additional rules to define its predicate are necessary. Although the language is powerful enough to express linear recursion, through the use of closure literals, other forms of recursion are not expressible.

A *filter* query can have several distinguished edges and blobs; they represent those objects that the user wishes to see. The query evaluator searches the database for all occurrences of the pattern, and retains the objects that match the distinguished edges and blobs from each occurrence. Edges and blobs defined by *define* queries can of course be used in *filter* queries.

3.2 The Translation

As explained above, the meaning of a *define query* in GraphLog is to define the predicate in the distinguished literal of the query. Suppose that, between nodes labelled X and Y, there is an edge labelled $p(\bar{Z})$, where X, Y are terms and $\bar{Z}$ is a sequence of terms. We will denote this by $p(X, Y, \bar{Z})$. In fact, the meaning of such an edge is given by the Datalog fact $p(X, Y, \bar{Z})$.

The translation for a *define* GraphLog query which has nondistinguished edges $p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$ contains the following rule which defines the predicate p of the distinguished edge $p(\bar{X})$:

$$p(\bar{X}) :- \ l_1(\bar{F}_1), l_2(\bar{F}_2), \ldots, l_n(\bar{F}_n)^1.$$

where $l_i(\bar{F}_i)$ is $p_i(\bar{X}_i)$, if the edge is positive, $\neg p_i(\bar{X}_i)$, if the edge is negated, or it is defined recursively on the structure of the label according to the following algorithm:

- *Inversion*: Label $-e(\bar{Z})$ corresponds to the literal $e(Y, X, \bar{Z})$.
- *Concatenation*: Label $e_1(\bar{Z}_1) \cdot e_2(\bar{Z}_2)$ corresponds to the rule

  $$conc(X, Y, \bar{F}_1, \bar{F}_2) \quad :- \quad e_1(X, T, \bar{Z}_1), e_2(T, Y, \bar{Z}_2).$$

  where $\bar{F}_i$ contains the named free variables of $\bar{Z}_i$ (without bound terms and anonymous variables), and T is a variable appearing nowhere else in the rule.
- *Alternation*: Label $e_1(\bar{Z}_1)|e_2(\bar{Z}_2)$ corresponds to the rules

  $$\begin{aligned} alter(X, Y, \bar{F}_1, \bar{F}_2) \quad &:- \quad e_1(X, Y, \bar{Z}_1). \\ alter(X, Y, \bar{F}_1, \bar{F}_2) \quad &:- \quad e_2(X, Y, \bar{Z}_2). \end{aligned}$$

  where $\bar{F}_i$ contains the named free variables of $\bar{Z}_i$. Note that if $\bar{Z}_1$ and $\bar{Z}_2$ contain different variables, the translation generates non-range-restricted rules.
- *Closure*: Label $e(\bar{Z})+$ corresponds to the rules

  $$\begin{aligned} tc\_e(X, Y, \bar{F}) \quad &:- \quad e(X, Y, \bar{Z}). \\ tc\_e(X, Y, \bar{F}) \quad &:- \quad e(X, T, \bar{Z}), tc\_e(T, Y, \bar{F}). \end{aligned}$$

[1] The idea of using $\bar{F}_i$ instead of $\bar{X}_i$ is that constants and anonymous variables need not appear in the call of a predicate that is associated with an expression (defined with additional rules).

where $\bar{F}$ contains the named free variables of $\bar{Z}$ (which are restricted to have the same value along the path), and T is a variable appearing nowhere else in the rule.

- *Kleene closure*: Label $e(\bar{Z})*$ corresponds to the rules

$$\begin{array}{lcl} kleene\_e(X,Y,\bar{F}) & :- & X = Y. \\ kleene\_e(X,Y,\bar{F}) & :- & e(X,T,\bar{Z}), kleene\_e(T,Y,\bar{F}). \end{array}$$

  where $\bar{F}$ contains the named free variables of $\bar{Z}$, and T is a variable appearing nowhere else in the rule. If $\bar{F}$ is non-empty, non-range restricted rules are generated.

Since, for the purposes of this paper, we assume only trivial filter queries, we will not describe their translation here. As an example, the query of Figure 2 is translated to the following Datalog program (we have omitted the full module definition for CORAL):

```
cl_depends(class("Object"), class(Y)) :-
                     tc_depends(class("Object"), class(Y)).
tc_depends(X, Y) :- depends(X, Y).
tc_depends(X, Y) :- depends(X, Z), tc_depends(Z, Y).
depends(class(X), class(Y)) :-
                     tc_calls(function(U), function(V)),
                     contains(class(Y), function(V)),
                     contains(class(X), function(U)).
tc_calls(X, Y) :- calls(X, Y).
tc_calls(X, Y) :- calls(X, Z), tc_calls(Z, Y).
```

Note, in the fourth rule, how the (non-deterministic) ordering of subgoals is poor since the complete transitive closure of `calls` must be computed. If, instead of formulating the query as in Figure 2, `cl_depends` was expressed using a single edge labelled with the regular expression $(contains \cdot calls^+ \cdot -contains)^+$, the following equivalent (if applied to the same database) Datalog program would be produced.

```
cl_depends(class("Object"), class(Y)) :-
                    tc_conc1(class("Object"), class(Y)).
tc_conc1(X, Y) :- conc1(X, Y).
```

```
tc_conc1(X, Y) :- conc1(X, Z), tc_conc1(Z, Y).
conc1(X, Y)    :- contains(X, T), conc2(T, Y).
conc2(T0, Y)   :- tc_calls(T0, T1), contains(Y, T1).
tc_calls(X, Y) :- calls(X, Y).
tc_calls(X, Y) :- calls(X, Z), tc_calls(Z, Y).
```

As evident in Figure 2, GraphLog users are allowed to label nodes with (single) constants. Thus, any translation of such a query to Datalog will result in a program containing a single constant in the query goal, whereupon the rewriting techniques such as magic sets [1], factoring [15] and context rewriting [13] become applicable. We assume that readers are familiar with each of these.

CORAL provides an annotation for controlling evaluation called **`@factoring`**. This applies the context rewriting transformation of [13]. We will use the term *factoring* in the same way as in [15] to mean the reduction in the number of arguments in recursive predicates. Context rewriting does not do this, so to avoid confusion we will refer to the CORAL annotation as *context rewriting*.

The present translation from GraphLog to Datalog results in rules in which the only form of recursion is transitive closure (possibly with additional arguments carried in the head and recursive subgoal). The variable patterns are such that, given most binding patterns for the recursive predicate, the subgoals of the recursive rule could be reordered so that the rule is either right-linear or left-linear. As such, context rewriting is almost always applicable (if both head and tail nodes are bound to constants the resulting program might be neither left- nor right-linear). This is related to the problem of overbound queries studied in [14]. With a translation to CORAL, one could circumvent this problem by never exporting a binding pattern for a module which included both the first two arguments of a recursive predicate being bound.

In fact, the present translation always produces right-recursive rules which, depending on the binding pattern, may or may not be right-linear. Nevertheless, it would be a simple matter to direct the translation based on the bindings in the query. A more fundamental problem is that context rewriting may be an order of magnitude less efficient than a true factoring transformation.

Rather than a translation based on the structure of the original regular expression in the GraphLog query, we propose an alternative translation based on a nondeterministic finite automaton constructed from the regular expression, as originally proposed in [19]. From now on, we will refer to the former technique as the RE-*translation* and the latter as the NFA-*translation*.

The NFA-translation can result in a Datalog program containing mutually recursive rules. As a result, neither context rewriting nor the original factoring transformation in [15] are applicable. In fact, the resulting programs are also not necessarily weakly right-linear as defined by Ross in [18][2]. Nevertheless, as we demonstrate in the next section, our translation always produces a program in which every IDB predicate is factored.

4 TRANSLATING GRAPHLOG TO FACTORED DATALOG

We assume initially that we are given a GraphLog query comprising one define query and one filter query with a single distinguished edge. Furthermore, the define query has only a single nondistinguished edge, in addition to its distinguished edge. The nondistinguished edge can be labelled with an arbitrary regular expression. The filter query simply refers to the distinguished edge of the define query.

Let R denote the regular expression in the query. The first step in the translation is to construct a nondeterministic finite automaton (NFA) M which accepts the language $L(R)$. Now we essentially construct a regular grammar G from M in the standard way, followed by a regular chain program based on G. Because of space limitations, we only describe the translation when one of the nodes in the GraphLog query is labelled with a constant. If it is the head node, we reverse the automaton M and perform the inversion of each term labelling a transition in M.

1. Generate the fact `s(c)`, where s is the initial state of the automaton, and `c` is the node constant in the GraphLog query.
2. For each transition t from p to q labelled with $e(\bar{Z})$, where $\bar{Z}$ may be a sequence of terms, generate a Datalog rule as follows:

   $$\texttt{q(Y) :- p(X), e(X,Y,}\bar{\texttt{Z}}\texttt{).}$$

   If, instead, t is labelled with $-e(\bar{Z})$, generate:

   $$\texttt{q(Y) :- e(Y,X,}\bar{\texttt{Z}}\texttt{), p(X).}$$

[2]This is true because the recursive subgoal may have more arguments bound than the head.

Call the resulting program P.

3. From P generate a new program Q by performing a bottom-up propagation of the named variables as follows.
   (a) Add each fact `s(c)` to Q.
   (b) For each rule in P containing an `s` subgoal, say,

   $$\mathtt{t(Y)\ :\!-\ s(X),\ e(X,Y,\bar{Z}).}$$

   add the rule

   $$\mathtt{t_{\bar{U}}(Y,\bar{U})\ :\!-\ s(X),\ e(X,Y,}\bar{Z}\mathtt{).}$$

   to Q, where $\bar{\mathtt{U}}$ comprises all the named variables in $\bar{\mathtt{Z}}$.
   (c) Repeat the following process until no (syntactically) new rule is added to Q. If there is a rule with head $\mathtt{t(Y,\bar{U})}$ in Q and a rule of the form

   $$\mathtt{p(Y)\ :\!-\ t(X),\ e(X,Y,\bar{Z}).}$$

   in P, then add the rule

   $$\mathtt{p_{\bar{V}}(Y,\bar{V})\ :\!-\ t_{\bar{U}}(X,\ \bar{U}),\ e(X,Y,\bar{Z}).}$$

   to Q, where $\bar{\mathtt{V}}$ is a sequence of named variables which includes all of those in $\bar{\mathtt{Z}}$ and $\bar{\mathtt{U}}$ such that the ordering of variables is consistent throughout the set of rules.
4. Finally, add rules for the distinguished edge. These rules have as their head the distinguished predicate and each body contains a single predicate $\mathtt{p}_{\bar{\mathtt{V}}}$, where p is a final state of the automaton.

The translation for GraphLog queries in which neither node is labelled with a constant is a simple modification of the above. The fact `s(c).` is replaced by `s(X,X).`, and each IDB predicate takes an additional, new first argument, say `W`. Because of space limitations, we cannot describe the translation for more general queries containing more than one nondistinguished edge, except to say that the set of nodes produced by each "edge query" can be passed to the next edge according to some traversal ordering of the query graph.

Note that, for the well-known ancestor query, the above translation effectively produces the left-linear transformation of [16] when the tail node is labelled with a constant, and produces the right-linear transformation when the head of the edge is bound to a constant.

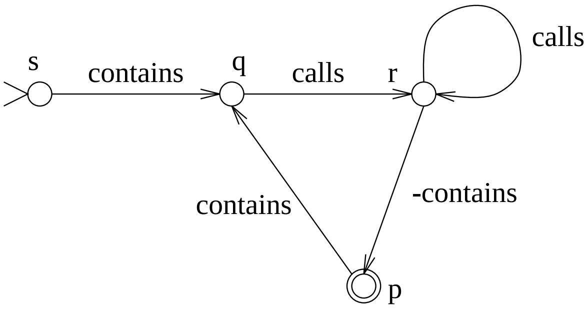

Figure 3 NFA for (contains·calls$^+$· -contains)$^+$

If the GraphLog query of Figure 2 is rewritten to contain only one nondistinguished edge labelled with $(contains \cdot calls^+ \cdot -contains)^+$, the resulting NFA is the one depicted in Figure 3. Being at the initial state of the automaton represents being at the node of the database graph labelled `Object`. Each transition in the NFA represents the traversal of the corresponding edge. Hence, performing the transition from state s to state q corresponds to following a `contains` edge from the node labelled `Object` to some node labelled `Y`. All such nodes are computable by the Datalog rule

```
q(Y) :- s(X), contains(X,Y).
```

If each transition of the NFA is thus translated into a Datalog rule, evaluation of the generated program determines the set of nodes reachable from `Object` by following a path of the form $(contains \cdot calls^+ \cdot -contains)^+$. The complete CORAL program that is produced by the above translation is shown below.

```
s(class("Object")).
q(Y) :- s(X), contains(X, Y).
r(Y) :- q(X), calls(X, Y).
r(Y) :- r(X), calls(X, Y).
p(Y) :- contains(Y, X), r(X).
q(Y) :- p(X), contains(X, Y).
cl_depends_bf(class("Object"), class(Y)) :- p(class(Y)).
```

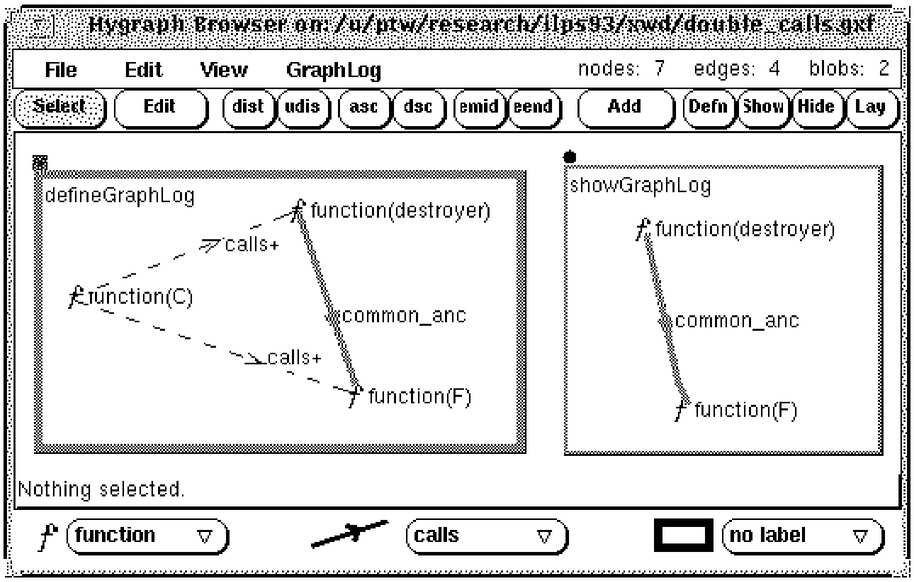

Figure 4 Another GraphLog query on the database of Figure 1.

Note how the IDB predicates have been factored to be unary rather than binary. A performance comparison between this translation and the original is given in the next section.

Let us now consider the query shown in Figure 4, which is a form of "common ancestor" query applied to function calls. We are interested in all functions which are called (directly or indirectly) by a function which calls (directly or indirectly) the function `destroyer`. Once again, we assume that the query is rewritten in terms of a single nondistinguished edge labelled with the regular expression $-calls^{+} \cdot calls^{+}$.

The RE-translation yields the following:

```
common_anc(function(destroyer), function(F)) :-
                  tc_calls(function(C), function(F)),
                  tc_calls(function(C), function(destroyer)).
tc_calls(X,Y) :- calls(X,Y).
tc_calls(X,Y) :- calls(X,Z), tc_calls(Z,Y).
```

The subgoals in the first rule are once again poorly ordered. However, the common ancestor program is interesting because, even if the subgoals in the first rule are interchanged, it too cannot be factored using traditional techniques. The problem this time is that, no matter what ordering of subgoals is chosen, the `tc_calls` subgoal is called with two different binding patterns. This means that CORAL defaults to supplementary magic sets, with a possible order of magnitude slow down. In contrast, our factored translation is given below:

```
s(function(destroyer)).
anc(Y) :- calls(Y, X), s(X).
anc(Y) :- calls(Y, X), anc(X).
des(Y) :- anc(X), calls(X, Y).
des(Y) :- des(X), calls(X, Y).
common_anc_bf(function(destroyer), function(F)) :-
        des(function(F)).
```

Here we see that both left- and right-linear rules have been generated, once again computing only sets of nodes at each step. The performance of the two translations is compared in the next section.

5 PERFORMANCE COMPARISONS

In this section, we present some of the performance results obtained during our investigation. A number of different databases of facts were used in our tests; only two are reported on below. The first is a database of flight information for airlines. It comprises 112 facts for a single predicate, involving source and destination cities, as well as airline names. There are 31 distinct cities, and 6 different airlines represented.

The second database has already been mentioned, namely, the NIH C++ class library. It comprises 9124 facts, of which 2406 are `calls` facts, and 2755 are `contains` facts. Some typical results are shown in Figure 5 (all times are in seconds and are averaged over a number of runs on a lightly loaded Sun Sparc10).

The column labelled A is the time to execute a program produced by the RE-translation with context rewriting by CORAL. Because the RE-translation does not always choose the best ordering of subgoals, we have manually modified

			A	B	**A/B**	C	**B/C**
	query	no. of answers	RE trans.	Opt. RE trans.		NFA trans.	
1.	`alt_bfff`	121	3.90	0.81	**4.80**	0.32	**2.52**
2.	`cl_depends_bf`	32	214.94	31.31	**6.86**	1.42	**22.11**
3.	`common_anc_bf`	507	225.48	44.07	**5.12**	1.34	**32.89**

Figure 5 Query execution times for the two translations.

the RE-translation with the best subgoal order. The execution of the resulting program with context rewriting by CORAL is reflected in column B. Column C corresponds to the NFA-translation with no rewriting by CORAL.

We have already seen test cases 2 and 3; they correspond to Figures 2 and 4, respectively. Test case 1 asks for all cities reachable from Toronto using flights which alternate between a pair of airlines. Each test case was chosen to demonstrate a particular situation. Test case 1 cannot be context rewritten as produced by the RE-translation. However, if left-recursive rather than right-recursive rules are produced, context rewriting can be done. This is done by hand and is reflected in column B. Nevertheless, factoring still shows a slight improvement over context rewriting.

In test cases 2 and 3, the programs produced by the RE-translation result in complete transitive closures being computed. Execution speeds are improved by 5 to 6 times by choosing a subgoal ordering which avoids this. As a result, test case 2 is context rewritten by CORAL. However, one can show that context rewriting can be an order of magnitude less efficient than factoring on this program, a fact which seems to be borne out by the empirical results. Test case 3 as produced by the RE-translation cannot be context rewritten, no matter what subgoal ordering is chosen; hence, CORAL uses supplementary magic sets. In this case, factoring achieves a speedup of over 30.

A further indication of where context rewriting can break down is given when we want to take the *reflexive* transitive closure of the `depends` relationship, rather than simply the transitive closure. In the above example, the answer turns out to be the same because class `Object` depends transitively on itself. The program produced by the NFA-translation exhibits no apparent difference in execution speed. However, the program produced by the RE-translation, even with the best subgoal ordering, slows down from an execution speed of 31 seconds to one of 249 seconds.

The test results shown in Figure 5 are by no means meant to reflect a comprehensive benchmark for GraphLog queries; rather they give an indication of the kinds of speedup which can be achieved by a factored program compared to a context rewritten program. Further tests and results are reported in [10].

6 CONCLUSIONS

We have discussed our experiences with the query evaluation component of the Hy$^+$ data visualization system developed at the University of Toronto. We have focussed on translating GraphLog queries into CORAL programs, providing two alternative translation strategies. In so doing, we have also demonstrated enormous differences in efficiency between equivalent Datalog programs. In other words, although we have used CORAL for our experiments, our arguments hold independently of the choice of backend, since the translations to CORAL, LDL and Prolog are almost identical for the class of GraphLog programs that we have investigated.

The results of the performance comparisons presented in this paper might lead one to conclude that the NFA-translation always performs better than the RE-translation, or at least for queries containing nodes labelled with constants. However, this is not the case. Each translation can outperform the other, irrespective of the presence of constants. The reason is that each translation can impose a different join ordering when evaluating recursive programs, and the impact of this can be greater than that of factoring. This issue is explored further in [10].

The choice between producing left- or right-linear rules when no node is labelled with a constant should also be carefully considered. We ran tests on a database representing the genealogy of theoretical computer science. When visualized as a graph, the fan-in of each node is at most two, while the fan-out can be over 20. Worst-case analysis in [14] suggests that, for such a case, left-linear rules are superior to right-linear rules. Empirical results confirmed this: right-linear rules were on average about 40% slower than left-linear rules for the ancestor program on this database.

Our experiments highlight a couple of points. The first is that the order of subgoals in the rules affects the performance of the generated Datalog programs, considerably. Therefore, we believe that chosing the best subgoal ordering is an important requirement for deductive database applications. The second point is that, even in the case of best subgoal ordering, factoring can result in substantial efficiency gains. We believe that handling subgoal reordering and extending the class of programs to which the factoring transformation is applicable, would both be useful optimizations for a future implementation of CORAL to support.

The class of GraphLog programs that are factored using the NFA-translation includes all queries not involving aggregation, in which the single distinguished edge in the filter part of the query is labelled with a constant. Whether the NFA-translation has any advantage when extended to queries involving aggregation or those requiring complex filtering, which can be particularly inefficient, remains to be investigated. Also we have restricted our attention to recursive queries; however, nonrecursive queries involving large joins can also be inefficient using the present RE-translation. We hope to be able to improve on this too in the future.

Acknowledgements

This work was done while the second author was visiting the Computer Systems Research Institute at the University of Toronto, supported by that Institute and a sabbatical grant from the Foundation for Research Development, South Africa.

REFERENCES

[1] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. Sixth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 269–283, 1987.

[2] C. Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[3] M. P. Consens and M. Hasan. Supporting network management through declaratively specified data visualizations. In H. Hegering and Y. Yemini,

editors, *Proc. IEEE/IFIP Third Int. Symp. on Integrated Network Management, III*, pages 725–738. Elsevier North Holland, April 1993.

[4] M. P. Consens, M. Hasan, and A. O. Mendelzon. Debugging distributed programs by visualizing and querying event traces. In *Proc. 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 181–183, May 1993. Extended abstract.

[5] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proc. Second ACM Hypertext Conf.*, pages 269–292, 1989.

[6] M. P. Consens and A. O. Mendelzon. GraphLog: A visual formalism for real life recursion. In *Proc. Ninth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 404–416, 1990.

[7] M. P. Consens and A. O. Mendelzon. Hy^+: A hygraph-based query and visualization system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 511–516, 1993.

[8] M. P. Consens, A. O. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proc. 14th Int. Conf. on Software Engineering*, pages 138–156, 1992.

[9] M. P. Consens, A. O. Mendelzon, and D. Vista. Deductive database support for data visualization. In *Proc. 4th Int. Conf. on Extending Database Technology*, pages 45–58, 1994.

[10] M. P. Consens, A. O. Mendelzon, D. Vista, and P. T. Wood. Translating visual GraphLog queries to Datalog (or, when constant propagation considered harmful). In preparation.

[11] M. Fukar. Translating GraphLog into Prolog. Master's thesis, Department of Computer Science, University of Toronto, 1991.

[12] D. Harel. On Visual Formalisms. *Communication of ACM*, 31(5):514–530, 1988.

[13] D. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule transformations that maintain context information. In *Proc. 16th Int. Conf. on Very Large Data Bases*, pages 380–391, 1990.

[14] I. S. Mumick and H. Pirahesh. Overbound and right-linear queries. In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 127–141, 1991.

[15] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument reduction by factoring. In *Proc. 15th Int. Conf. on Very Large Data Bases*, pages 173–182, 1989.

[16] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and combined-linear rules. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 235–242, 1989.

[17] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 167–176, 1993.

[18] K. A. Ross. Modular acyclicity and tail recursion in logic programs. In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 92–101, 1991.

[19] P. T. Wood. Factoring augmented regular chain programs. In *Proc. 16th Int. Conf. on Very Large Data Bases*, pages 255–263, 1990.