Domain Independent Disambiguation of Vague Query Specifications

Yezdi Lashkari

Technical Report #1181

July 1993

Domain Independent Disambiguation of Vague Query Specifications

YEZDI LASHKARI

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI.
July 1993.

Thesis Advisor ____

Yannis E. Ioannidis
Associate Professor of Computer Sciences

Certified By ____

Jude W. Shavlik
Assistant Professor of Computer Sciences

Abstract

To successfully query a database system, a user needs to know not only the data model of the underlying Database Management System (DBMS) and the associated query language, but also the exact arrangement of the data in that particular database. Recent years have witnessed a growing number of DBMS users who are not versed in the above skills, yet still need to interact regularly and productively with a DBMS. Further, even for a person having the requisite skills, the need to explicitly conform to the database structure while querying can be a time consuming and laborious task, especially as the number of ad-hoc queries to DBMSs increase.

When we talk to each other we have no trouble disambiguating what another person means, even though our statements are almost never meticulously specified down to the very last detail. We 'fill in the gaps' using our common-sense knowledge about the world. This thesis presents a formalism to allow users of Object Oriented DBMSs to specify queries to a database system in a manner closer to the way we pose questions to each other, which constitutes a vague query specification for the underlying DBMS. Our system takes as input this vague specification, and generates complete queries consistent with this vague specification that it believes are what the user most likely meant by the vague specification. The formalism we present is completely domain independent - it requires no knowledge about the specific entities in a particular database. The system works by exploiting the structure of the relationships between data entities in the database. A set of two preliminary experiments with human subjects indicates that such a purely structure-based approach to disambiguation of vague queries is quite powerful.

181

Contents

List of Figures

Chapter 1

INTRODUCTION

Consider the following fragment of conversation:

> *"Do you know Valerie ?"*
>
> *"No. Who is she ?"*
>
> *"She's a new teaching assistant in the department."*
>
> *"What are Valerie's courses ?"*

If we are asked the last question we have absolutely no difficulty in interpreting it as meaning either one of the following or both:

- "What are the courses *taught* by Valerie ?"
- "What are the courses *taken* by Valerie ?"

Of course, if we think hard enough, in a typical university setting there are a myriad of other options of courses that are associated with Valerie, some of them mildly plausible in the context of the original question, some frankly ludicrous. For example, the options

- "the courses taught by fellow teaching assistants in Valerie's department"
- "the courses taken by the students in Valerie's department"
- "the courses taught by Valerie's advisor"
- "the courses taught by all the professors in Valerie's department"

are all valid associations between Valerie and a set of courses. Technically, the question

"What are Valerie's courses ?"

is underspecified, and hence has a number of possible interpretations. As humans however, we know exactly what the question means. In everyday life we fill in the *gaps* in the specification of the question using our common-sense knowledge about teaching assistants, universities and student life with absolutely no difficulty. Is it possible for a computer program to fill in such gaps in an underspecified statement/query so that the complete statement/query reflects the user's intuition ?

Before we address this question further we present three trends in the usage of database management systems (DBMSs) that we feel have a direct bearing on the importance of the question above.

A noticeable trend in recent years has been a sharp increase in the number of *naive users* with access to DBMSs. A user of a DBMS system is said to be *naive* if the user is not familiar with at least one of the following:

- The query language of the DBMS,
- The data model of the DBMS,
- The schema of a database in the DBMS.

Most such users are not computer scientists or database experts, nor do they wish to become one. They need however, to manage vast quantities of data and require quick access to answers by formulating queries (often ad-hoc) to the DBMS in order to make decisions. Furthermore, most such users would not even have designed the schema for the data. Such users are then left with two equally unattractive options. Either learn the schema so that they can formulate queries that the DBMS will accept without choking (not a light task for most DBMSs which store large quantities of data about many different types of objects and whose schema may constantly evolve), or hire an intermediary (a computer expert) to interact with the DBMS and provide the necessary answers to their questions. For most business executives for example, the latter is currently the option of choice.

Another trend we expect to observe is a significant increase in the number of ad-hoc queries to DBMSs. Many current systems are used for business purposes (e.g. managing inventories), and have a fixed set of queries that are pre-compiled, pre-optimized and stored. The expansion of the application domains of DBMSs and the increase in the number of naive users will result in many more ad-hoc queries.

A final trend that we expect to see in the coming decades is the prevalence of the Object Oriented (OO) model for DBMSs [ABD+ 89]. The technology still suffers from the lack of a single formal model unlike its predecessor, the relational model. However the OO model provides a more *natural* way of organizing and thinking about data, as well as the advantages of imposing a natural structure on the data, inheritance of properties, and the ability to specialize objects via subclassing. Various OO DBMSs (OODBMSs) are currently available as products, and we expect to see more as the technology becomes more widespread.

The challenge is to give naive users with access to a DBMS the ability to formulate questions in the way they think naturally and common-sensically. The problem of course is that everything has to be specified exactly to a computer and a DBMS. With the number of ad-hoc queries increasing, this implies that the user *must know* exactly how the data is organized (must know the DBMS schema), *since all DBMSs require the user to explicitly conform to the schema structure while querying.* This is no trivial task for most real world databases even if a natural language or menu driven interface could be provided to formulate the exact query. Work on Graphical User Interfaces (GUIs) [ILH 92] is a step in this direction. A GUI provides a pictorial representation of the organization of the data in the DBMS to the user. However, there are drawbacks to this solution too. Most current DBMS schemas are too large to fit into one screen of most workstations. Printing out the schema is not too useful if it is constantly changing. Further, even with the entire schema in front, specifying a query to the DBMS completely may be a tedious chore. It would be much more convenient if the user were simply to click on a few entities on the schema diagram and have the system figure out a way to connect the entities selected into a query that is meaningful to the user.

This thesis introduces a mechanism that allows a user to specify incomplete (and therefore ambiguous) queries, without explicitly having to know how the data is structured inside the DBMS.

For users familiar with the database schema this mechanism provides a convenient shorthand with which to specify queries, which, due to the structure of the schema, may be long and difficult to formulate. In response the user is presented with a completely specified query (or possibly a set of queries) that is consistent with the original query and is its most likely completion based on the system's knowledge.

A straightforward approach to achieve the above is to associate several rules with a given database that capture some common-sense notions of the world represented by the data in the database. Such a solution however, is schema dependent and is hence quite useless for any other schema but the one for which the rules were designed. Furthermore, with a constantly evolving schema, the rule set may have to constantly evolve to keep track of the new data, and experience with production systems has shown that this is not a trivial task in a lot of cases.

In this thesis, we introduce a novel, lightweight approach to the problem above. Based on the third trend above, we assume the underlying DBMS is OO. Our procedure exploits the structure of the database schema *to choose likely completions of underspecified queries.* It has *absolutely no knowledge* about the entities in the underlying database - all it knows are the semantics of the various types of links supported by the data model of the DBMS. This implies that the system is very easily portable from one database to another.

At first glance, such an approach seems very restricted in terms of power. However, we present results from a preliminary set of experiments that indicate that a such a purely syntactic approach yields good performance at little cost.

This thesis is organized as follows. In Chapter 2 we introduce the OO data model we assume for the rest of this thesis, and introduce the notion of a path expression. In Chapter 3 we cast the problem as the familiar problem of computing an optimal path over a labeled directed graph and introduce the notions of path collapse and path selection functions. Chapter 4 presents the particular path collapse and path selection functions implemented by us as also certain properties satisfied by these functions that are useful for optimizing an implementation. In Chapter 5 we present the algorithm to generate plausible completions to vague specifications with optimizations for efficient traversal of a database schema. In Chapter 6 we present the results from a set of preliminary experiments conducted on two different schemas with human subjects. Chapter 7 discusses various possible options for increasing accuracy of the procedure based on the experimental results. In Chapter 8 we present related work. We conclude in Chapter 9 with some directions for future work.

Chapter 2

THE DATA MODEL AND ASSOCIATED QUERIES

In this chapter, we introduce the data model of the DBMS that the rest of the thesis assumes and present criteria for "correct" schemas in this model. We then introduce path expressions and show how our problem essentially reduces to finding appropriate path expressions in the schema graph. We also present the overall picture of where exactly we expect such a system to fit in the context of a full-fledged DBMS.

2.1 The Data Model

Our OODB model is based loosely on the MOOSE OO data model [WiIo 93]. However, it is general enough to easily map onto most existing OODB models. In the DBMS, real world entities are modeled by objects. Objects are grouped together by uniquely named *classes*, which capture the objects' common properties. Binary *relationships* describe the connections between objects in the schema classes.

This work is only concerned with the database schema, not the actual data stored in the database itself. We represent the schema as a graph: each class is a node in the graph, and each relationship is an edge, possibly directed, in the graph between two class nodes. Each relationship has a label *in each direction*, which if unspecified, is equal to the name of the target class of the relationship in that direction. Figure 2.1, shows a sample OODB schema representing information about students, professors, and universities.

2.1.1 Object Classes

Each class has a *kind* which describes the class's basic structure. We allow three kinds of classes in our OODB model: *primitive*, *tuple*, and *collection*.

The primitive classes are provided by the system. Currently our model assumes the existence of four types of primitive classes - *Integer*, *Real*, *Character String*, and *Boolean*; in Figure 2.1 they are represented by circles and abbreviated to the letters I, R, C, and B respectively.

Objects in tuple classes consist of a prespecified number of other objects called *parts*, usually from several different classes. Each part is identified by a relationship, uniquely labeled for that class. A special case of tuple classes are *atomic* classes, whose objects have no parts.

Objects in a collection class consist of an arbitrary number of other objects, all from a single class, called the *members* class. We allow two general types of collection classes: *sets* and *arrays*.

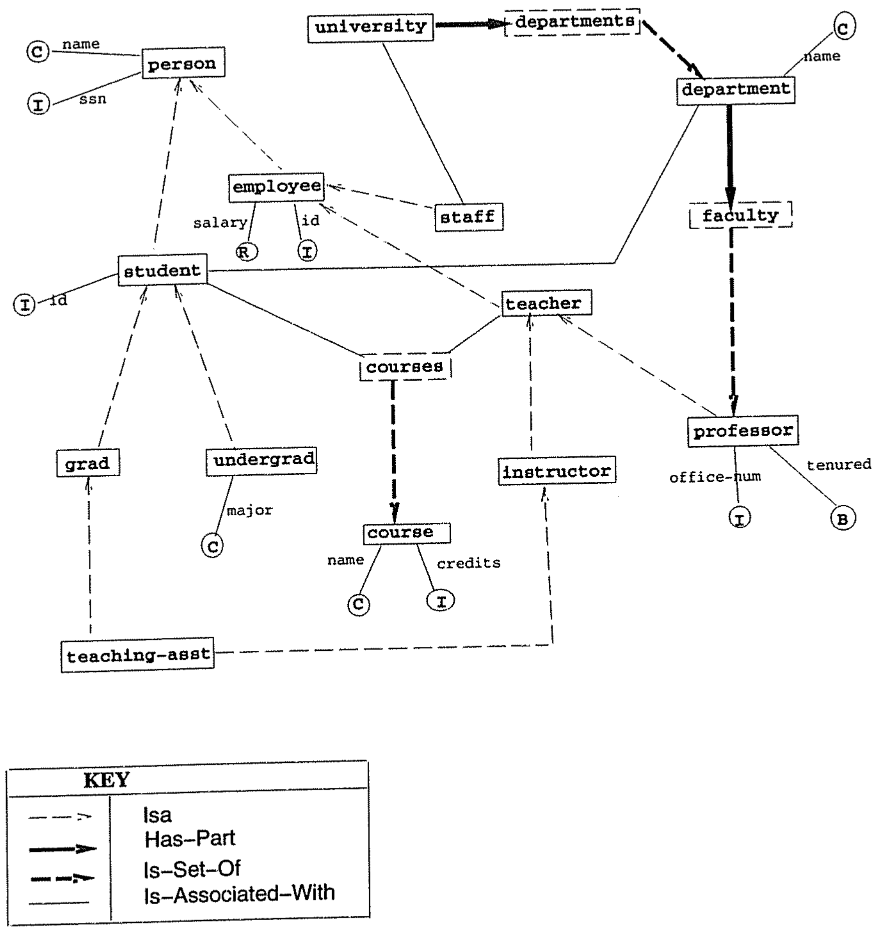

Figure 2.1: A Sample Schema in the OODB Model

An array is a set indexed by the set of consecutive integers {1, ..., n}. Sets may be multisets, strict sets, or indexed sets. The elements of an indexed set are indexed by (the elements of) another arbitrary collection object. This collection object is called the *keyset* for the indexed set, because its elements provide indexing keys into the indexed set. [1] For the purposes of this thesis we treat all collection objects in the schema uniformly.

Tuple objects are represented by solid rectangles and collection objects by broken rectangles in Figure 2.1.

2.1.2 Object Relationships

Relationships in our data model may be either directed or undirected. A directed relationship has two distinct interpretations when traversed in opposite directions. Such relationships are said to be *non-isotropic*. For all non-isotropic relationships, the data model designates one of the interpretations as the *dominant interpretation*. Informally the dominant interpretation corresponds to the more intuitive view of the relationship. In the schema graph diagram the dominant interpretation is the interpretation of the relationship in the direction of the arrowhead.

We assume two major types of relationships between classes in the schema.

[1]An indexed set indexed by the consecutive integers is an array.

1. **Inheritance Relationships**

   An inheritance relationship is a directed relationship between two classes, called the *superclass* and the *subclass*. An inheritance relationship implements *inclusion* and *specialization* semantics : *inclusion inheritance* means that all objects in the subclass are also instances of the superclass. Thus, in Figure 2.1 all instances of the class grad are also instances of the superclass student and its superclass person. *Specialization inheritance* means that the subclass inherits all the relationships of the superclass. The subclass may refine (redefine) these relationships and possibly define its own relationships in addition to those inherited from the superclass. For example, the class student inherits the relationships *name* and *ssn* from its superclass person. In addition student defines three new relationships for all objects of class student: *department*, *courses* and *id*, which are inherited by all objects of class grad and undergrad along with the properties *name* and *ssn*.

   Our model allows *multiple inheritance* ; i.e., a class may be a direct subclass of more than one superclass ; such a class inherits properties from all its superclasses. Inheritance relationships are also called *Isa* relationships in the direction from subclass to superclass, and *May-Be* in the direction from superclass to subclass. Hence, in Figure 2.1, student *Isa* person and person *May-Be* student. We use the symbols @> and <@ to denote the *Isa* and *May-Be* relationships respectively.

   Our model requires every object in the data base to be associated with at least one class in the schema graph. Thus, if subclasses a and b of superclass c share some common objects, then these objects must be members of an explicit class d in the schema that is a subclass of both a and b.

   The dominant interpretation for an inheritance relationship is *Isa*.

2. **Connection Relationships**

   A connection relationship between two classes implies a logical or physical relationship between their object instances. For example, connection relationships can be used to describe a composite object: the relationships will connect the composite object class (e.g. department) to the classes of its parts (e.g. faculty, student).

   Our data model allows four distinct kinds of connection relationships between classes in the schema to reflect the different ways that objects may be related and capture additional semantics implied by these relations. In general, each kind of relationship may be interpreted in two different ways from the perspective of each of the two classes it connects. In each interpretation, one of the two related classes plays the role of the *source* class and the other one plays the role of the *target* class, thus imposing a direction in the interpreted relationship.

   (a) **Has-Part (Is-Part-Of)**

   This is a structural relationship between two classes. Such relationships (also called *part* relationships) are used to describe relationships between classes that are physical or logical parts of a whole and vice versa. Such a relationship is directed from the class that contains the part to the class that is a part of it (the dominant interpretation of a *part* relationship is *Has-Part*). If the relationship is traversed in the reverse direction to the arrowhead it is read as *Is-Part-Of*. We use the symbols $> and <$ to denote the *Has-Part* and *Is-Part-Of* relationships respectively.

   (b) **Is-Set-Of (Is-Member-Of)**

   This is a structural relationship between a collection class and a members class. An *Is-Set-Of* relationship (also called a *set* relationship) connects objects of a collection class to the objects of the member class of that set. The dominant interpretation of a *set* relationship is *Is-Set-Of.* The relationship in the reverse direction is read as *Is-Member-Of.* We use the symbols => and <= to denote the *Is-Set-Of* and *Is-Member-Of* relationships respectively.

(c) **Is-Indexed-By (Indexes)**
This relationship connects an indexed set class to the collection class of its keyset. The arrowhead for this directed relationship (also called an *index* relationship) is in the direction of the *Is-Indexed-By* relationship (that is from the indexed set to its keyset). We use the symbols **-i->** and **<-i-** to denote *Is-Indexed-By* and *Indexes* relationships respectively. The dominant interpretation of a *index* relationship is *Is-Indexed-By.*

(d) **Is-Associated-With**
This is the only *undirected* relationship between two classes in the schema. This relationship connects two classes that are mutually associated with each other, and the relationship is equally dominant in both directions. Hence associations have the exact same interpretation in both directions (they are *isotropic*). Such a relationship is read as *Is-Associated-With* in both directions. For the case when one of the classes is a primitive class, the relationship is read as *Has-Attribute* from the non-primitive class to the primitive class and is meaningless in the reverse direction. For example, in Figure 2.1 person *Has-Attribute ssn*, but to say that the class of *Integers* is associated with the class of person is quite meaningless. We use the symbol . to denote the *Is-Associated-With* relationship.

Each of these relationships can have other properties such as a cardinality ratio in each direction, mutability constraints, existence of NULL values, etc. Our model does not preclude these orthogonal properties nor does it require them.

Another point to note is that quite often the decision on whether to use an *association* or a *part* relationship between two classes is subjective and depends on the designer of the database schema. For example, the *Is-Associated-With* relationship between student and department in Figure 2.1 could equally well have been modeled as a *Is-Part-Of* relationship between the two classes. Association relationships are essentially "weaker" than the corresponding structural relationships in terms of the relationship they encapsulate. There is no one "correct" modeling of a given world ; it is dependent, to a certain extent, on the schema designer.

2.1.3 Constraints on the Schema

Our OODB model imposes some constraints on the schema due to the semantics of the classes and the relationships connecting them. Specifically, the following cases of class-relationship combinations are considered incorrect for the schema:

- The structure of objects in a tuple class is defined by some arbitrary number of *Has-Part* relationships, which should be the only dominant non-isotropic connection relationships of the class.
- The structure of collection objects is defined by a single *Is-Set-Of* relationship, except for objects in an indexed set class, whose structure is defined by a single *Is-Set-Of* and a single *Is-Indexed-By* relationship. No other dominant non-isotropic connection relationships should exist for the class.

- This is a global constraint on the entire schema. Natural structural considerations imply that the directed subgraph constructed by merging all classes connected by *Isa* relationships, and restricted to the dominant interpretation of all the other non-isotropic relationships *must be acyclic.* This simply means that objects of a class cannot have objects of the same class as parts.
- The directed subgraph constructed by retaining only the dominant inheritance relationship should be acyclic. This means that a class cannot be a specialization of itself.

2.2 Queries and Path Expressions

We assume that queries in our model are specified in a SQL-like declarative syntax. The exact syntax is irrelevant for our purpose. What concerns us is the notion of a *path expression* in a query.

2.2.1 Path Expressions

Path Expressions are the primary mechanism of specifying relationships of data that are part of a query. *A path expression corresponds to a path in the schema graph.* The path expression starts at a class, called the *path expression root* (which cannot be a primitive class), and continues traversing relationships. For each relationship traversed, the path expression contains a connector symbol corresponding to the type of relationship and the relationship label. Inheritance relationships use the connector symbol @ , part relationships the connector symbol $, set relationships the connector symbol = , index relationships the connector symbol **-i-** , and association relationships the connector symbol **.** . Path expressions can traverse inheritance and connection relationships in both directions. For directed relationships, the direction of the relationship may be (optionally) specified by using the symbols > and < along with the relationship connector symbol, for the forward relationship traversal and reverse relationship traversal respectively.

Some sample path expressions for the schema of Figure 2.1 are:

- *student . courses . teacher*
- *student @ person . ssn*
- *student @> person . ssn*
- *department . student @ person . name*

Any path expression that contains a fragment of the form:

$$a \ @\!> b <\!@ \ c$$

where, b is a single relationship name, and a and c are arbitrary path expressions, is invalid. The class b, that the relationship b points to, is termed an *invalid Isa crossover point* for the path expression. This restriction stems from our requirement that every object in the database be associated with an explicit class in the schema. If such a class exists in the schema, d say, then the path expression fragment above can be replaced by the fragment:

$$a <\!@ \ d \ @\!> c$$

which designates the same set of objects that are members of the classes a and c. If the class d does not exist, then the initial fragment above, corresponds to a set of non-existent objects, and hence would produce a NULL answer.

2.2.2 The Vague Connector Symbol ~

A path expression corresponds to a specification of a path in the schema graph between a class node and a relationship having the label equal to that of the last relationship in the path expression. In general, there are multiple such paths in a given schema graph. However each of these paths refers to a potentially distinct set of objects in the given database.

For example, the paths:

1. *teaching-asst* @> *grad* @> *student* @> *person* . *name*
2. *teaching-asst* @> *grad* @> *student* . *courses* => *course* . *name*
3. *teaching-asst* @> *instructor* @> *teacher* . *courses* => *course* . *name*

are all valid paths in the schema of Figure 2.1 from the class *teaching-asst* to a connection relationship labeled *name*. However, the first path refers to the name of the teaching assistant as a person, the second to the names of the courses that the teaching assistant is taking as a student, and the third to the name of the courses that the teaching assistant is teaching as a teacher.

It is clear from the above, that the user formulating a path expression *must know the details of the schema graph.* For the reasons mentioned in Chapter 1, this is often undesirable.

Ideally, we would like to develop a mechanism that will allow a user to specify something like

"the names of the teaching-assistants"

and have the system figure out the path expressions

teaching-asst @> *grad* @> *student* @> *person* . *name*, and/or

teaching-asst @> *instructor* @> *teacher* @> *employee* @> *person* . *name*

as the path expressions the user meant to construct. This is achieved by introducing the ~ connector symbol for path expressions. When it appears in a path expression, the ~ symbol acts as a wildcard specification. *A path expression having one or more instances of the ~ symbol in it is termed a vague path expression.* This is similar to the * specification in regular expressions in UNIX with one important difference. While a specification of the form $a{*}b$ for a UNIX regular expression returns *all* strings beginning with a and ending in b, a path expression of the form a ~ b returns the *best* path expression(s) starting at the class a and ending in a relationship labeled b, under some definition of 'best' discussed in Chapter 4.

For example, the expression

"the names of the teaching-assistants"

would be stated as

teaching-asst ~ *name*

and the system should retrieve the path expressions

teaching-asst @> *grad* @> *student* @> *person* . *name*, and/or

teaching-asst @> *instructor* @> *teacher* @> *employee* @> *person* . *name*

as the best corresponding path expressions for the given vague path expression. It may be noted that the path expressions *teaching-asst* @> *grad* @> *student* . *courses* => *course* . *name*

teaching-asst @> *instructor* @> *teacher* . *courses* => *course* . *name*

teaching-asst @> *grad* @> *student* . *department* . *name*

are also consistent with the vague path expression above. However, they are obviously not as intuitive as the path expressions denoting the name of the teaching assistant.

A vague path expression can have more than one instance of the ˜ connector in it. Further, when the type of relationship connector for a particular relationship is known, it can appear in a vague path expression. This corresponds to providing additional information to guide the search for the appropriate path expression(s). For example,

student ˜ *courses* ˜ *name*

student . *course* ˜ *professor* ˜ *name*

student . *department* ˜ *teaching-asst*

are all valid vague path expressions.

2.3 Semantics of Vague Path Expressions

This section presents the notion of a path expression consistent with a given vague path expression, the treatment of collection classes and cycles while generating path expressions, and introduces the notion of the most plausible path expressions consistent with a given vague path expression.

2.3.1 Generating Completions of Vague Path Expressions

Consider a vague path expression

$$\eta = s\ \phi_1 l_1\ \phi_2 l_2\ ...\phi_k l_k$$

where,

1. s is the name of the root of the path expression(s) corresponding to this vague path expression.
2. $\forall_{i=1}^{k}\ \phi_i$ is a valid relationship connector symbol, and
   $\exists_{j=1}^{k}\ \phi_j = \tilde{}$
3. $\forall_{i=1}^{k}\ l_i$ is the label of a relationship in the schema graph.

Let Ψ denote the set of all valid path expressions that are consistent with η in the schema. A valid path expression ψ is said to be *consistent* with the vague path expression η iff it has root class s and the sequence of labels and connector symbols created by removing all instances of the connector symbol ˜ from η results in a subsequence of ψ, where each connector-label pair in η is considered a single unit for connector symbols $\neq$ ˜ .

For example, given the vague path expression

$$\eta = student\ .\ department\ \tilde{}\ name$$

the path expressions

- *student* . *department* . *name*

- *student* . *department* $> *faculty* => *professor* @> *teacher* @> *employee* @> *person* . *name*
- *student* . *department* $> *faculty* => *professor* @> *teacher* . *courses* => *course* . *name*
- *student* . *department* . *student* @> *person* . *name*

are all path expressions consistent with η in the schema of Figure 2.1, while the path expressions

- *student* . *courses* . *teacher* <@ *professor* <= *faculty* <$ *department* . *name*
- *student* @> *person* . *name*

are not.

2.3.2 Cyclic Path Expressions and Invisible Classes

Our data model allows all relationships to have a cardinality ratio. A relationship may have a 1:1, 1:N. M:1, or M:N cardinality ratio. This implies that strict collection classes as defined by our data model, are redundant as a mechanism for representing collections unless other (derived) properties of the collection (e.g. the highest/lowest member, the number of members etc) also need to be maintained.

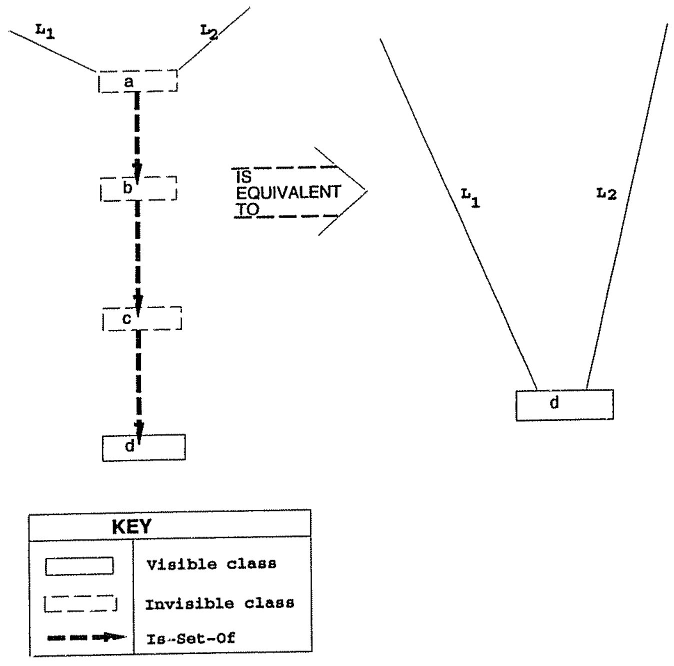

Figure 2.2: Dealing With Invisible Classes

In general, most users do not consider cyclic path expressions consistent with a given vague path expression to be very cognitively plausible, unless the cycle is explicitly specified in the vague path expression. Human beings do not think circularly for the most part, and when they do, such a circularity is explicitly stated. For this reason we do not allow cycles when generating path expressions between any two classes in the schema graph. However, this restriction on path expressions gives rise to a problem, illustrated by the example below.

Consider, the vague path expression

student ~ professor

corresponding to the statement, "the professors of this student". A logical completion of the above vague path expression, is the statement : "the professors who teach a course this student takes". This corresponds to the path expression

student . courses => course <= courses . teacher <@ professor

in the schema graph of Figure 2.1. The path expression above has a cycle through the collection class *courses*. The only non-cyclic path expression in the schema graph dealing with students, professors, and courses is

student . courses . teacher <@ professor

which designates "the professors who teach *exactly the same set of courses* taken by the student", a completion that is not very intuitive, and is unlikely to yield any answers from the database.

We as humans, do not think in terms of collections of objects as distinct collection and member entities. We tend to think of collections simply in terms of a given number of member entities. For example, we say

"John owns a Mercedes, a BMW, and a Porsche"

as opposed to

"John owns a set of cars and the set consists of a Mercedes, a BMW, and a Porsche"

The grouping of similar member entities into a collection is implicit.

However, our data model requires a schema graph to explicitly differentiate between collection and member classes, which is not very intuitive. To capture human intuition vis-a-vis collections of objects, we term all collection classes encountered while generating a path expression to be *invisible* except in the following cases :

1. The vague path expression explicitly denotes the collection class by a term of the form $\phi_i l_i$, where l_i is the name of a relationship to the collection class, and the connector symbol ϕ_i matches the kind of this relationship.

2. The vague path expression explicitly denotes a *derived property* of the collection class.

Our algorithm deals with invisible collection classes by acting as though they don't exist. When an invisible class is encountered, the algorithm attempts to descend the *Is-Set-Of* relationship from this class as far as possible (a class may be a collection of collection classes), and then ascend back up to the original collection class node. Since the class is invisible, this is not considered a cycle. If the class is visible, then this is a cycle, and is disallowed. For the example above, the set class *courses* is invisible, and hence the path expression generator algorithm can pass through it again 'without' cycling.

The invisible class mechanism can be thought of as essentially replacing the connection relationships to the invisible collection classes, by the identical connection relationships (but having a :N cardinality ratio) directly to its member class. This situation is shown pictorially in Figure 2.2.

2.3.3 Choosing the Most Plausible Path Expressions

Given the set Ψ of path expressions consistent with a given vague path expression η, our problem is to determine the subset $\Psi_{best} \subseteq \Psi$ that contains the 'best' (most cognitively plausible) path expressions consistent with the given vague path expression. Intuitively, each path expression in Ψ has a certain 'property' based on its structure. The set of path expressions in Ψ_{best} contains only those path expressions that have the 'best' properties from all those in Ψ. For our purposes, the 'best' path expressions from the set Ψ consist of the most intuitive path expressions in Ψ. How we rank path expressions on the basis of relevance to intuition and plausibility will be presented in Chapter 4.

2.4 Where It All Fits

We conclude this chapter with an overview of where a module such as ours may fit in the context of an overall DBMS.

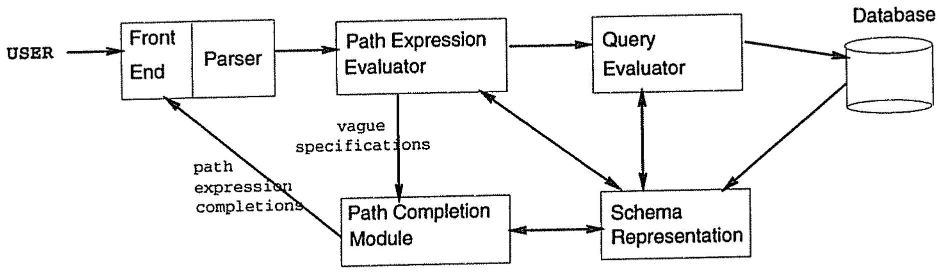

Figure 2.3: Relationship amongst modules in the DBMS

The structure of the system is shown in Figure 2.3. We envisage our path completion module to be callable from a path expression evaluator module. A user enters a query / vague path expression to a front end, which is then parsed by a parser module. Note that the front end could implement a Natural Language like interface to capture vague path expressions and use a synonym lexicon to relate the user input to the actual class/relationship names in the schema. The path expressions / vague path expressions are then sent to a path expression evaluator, which calls our module when a vague path expression is encountered. The path completion module returns a set of likely path completions (making use of a common schema graph representation) back to the front end which presents them to the user in an appropriate form (English like or path expression syntax). The user selects the path expression(s) desired which are then sent off to the path expression evaluator. If none of the path expressions returned are ones the user desires, then the user has to focus the query a bit further by providing additional information in terms of intermediate points to the vague path expression. The entire specification-path completion generation cycle is designed to take as little time as possible. Hence, we hope to provide close to real time disambiguation and query formulation of vague path expressions.

Chapter 3

COMPLETIONS AS OPTIMAL PATH COMPUTATIONS

In this chapter we cast the problem of generating path expressions consistent with a given vague path expression as an optimal path computation over a labeled directed graph. We begin with an introduction to the optimal path computation problem, model our problem as an optimal path computation, and then introduce the path collapse function CON.

3.1 Optimal Path Computation

There has been much work done on the problem of computing several properties that are specified over the set of paths in a labeled directed graph [Carr 79, Ros+ 86, ADJ 90, IRW 93]. Such properties are called *aggregate* properties and the computation of such a property is termed a *path computation.*

Most path computation algorithms use some variant of the path algebra formalism below, which is taken from [Carr 79, IRW 93]. There is a label L_{ij} associated with each arc (i, j) in the graph. A *path* p_{ij} from node i to node j is an ordered set of arcs $\{(source_k, destination_k)\}$, k = 1, ... , n, such that i = source$_1$, destination$_1$ = source$_2$, ... , destination$_n$ = j. A path is sometimes specified by the sequence of nodes on it. A label is associated with a path p_{ij}. Intuitively, this path label is computed as a function, called CON (for *concatenate*), of the sequence of labels of the arcs in p_{ij}. A label can be associated with a *path set* P as well. This *path set label* is computed as a function, called AGG (for *aggregate*), of the path labels of the paths in P.

Informally, the CON function computes the desired property of a path while the AGG function selects the paths having the optimal or 'best' such property. Formally, AGG and CON are defined as binary functions over path labels. Path Computation algorithms require both AGG and CON to possess identities, designated Φ and Θ respectively. Furthermore, several core properties must hold for the algorithms to be applicable. They are :

1. CON(L_1, CON(L_2, L_3)) = CON (CON(L_1, L_2), L_3)
2. AGG(L_1, AGG(L_2, L_3)) = AGG (AGG(L_1, L_2), L_3)
3. AGG(L_1, L_2) = AGG(L_2, L_1)
4. AGG(Φ, L_1) = L_1 ; CON(Θ, L_1) = CON(L_1, Θ) = L_1

The problem of optimal path computation is intimately linked to the computation of the *transitive closure* of a graph, and, in the general case, is exponential in time for cyclic graphs. To permit efficient path computation and ignore cyclic paths path computation algorithms require the following two properties in addition to the four enumerated above :

5. AGG (CON(L_1, L_2), CON(L_1, L_3)) = CON (L_1, AGG(L_2, L_3))
6. AGG (L_1, Θ) = Θ

If a graph satisfies the properties above, then efficient optimal path computation algorithms exist [IRW 93].

3.2 The Problem as Optimal Path Computation

Clearly we can cast the problem of generating plausible completions of vague path expressions as an optimal path computation over a labeled directed graph as follows.

- The graph in question is the schema graph with each relationship between two classes replaced by two directed arcs between the class nodes ; one for the forward traversal and one for the reverse traversal. The *labels* on the arcs in the path computation formalism are the *kinds* of the relationships when the relationship is traversed in that particular direction in the schema graph. To avoid any confusion between our use of the term *label* for a relationship as the *name* identifying that relationship and that used in the path computation formalism presented above, we henceforth use the term *connector* or *kind* to refer to the *label* of an arc in the graph in the path computation formalism. For example, in Figure 2.1 the *part* relationship between the classes department and faculty would be replaced by the following arcs between the class nodes for department and faculty:

  1. An arc with connector *Has-Part* from department to faculty with label *faculty.*
  2. An arc with connector *Is-Part-Of* from faculty to department with label *department.*

- A path in the graph above corresponds to a path expression in the schema graph.

- The function CON above corresponds to a *path collapse function* for our problem. *The path collapse function attempts to preserve the kind of the relationship between the two end points of a path fragment based on the kinds and order of the links in that path fragment.* The path collapse function defines several additional kinds of relationships in addition to those provided by the OO data model in Chapter 2. Such relationship kinds are termed *secondary*, as opposed to *primary* relationship kinds provided by the data model. For example, in Figure 2.1, consider the path fragment

  student *Is-Associated-With* courses *Is-Associated-With* teacher

  The path collapse function replaces the pair of primary relationships (*Is-Associated-With*, *Is-Associated-With*), between the nodes student and teacher, with the secondary relationship *Is-Indirectly-Associated-With* to indicate that the class of students is indirectly associated with the class of teachers through another class (in this example, the class courses).

  As a further example, consider the path fragment

  student *Is-Associated-With* courses *Is-Associated-With* teacher *May-Be* professor

As before, the path collapse function replaces the initial fragment to give the equivalent path fragment

student *Is-Indirectly-Associated-With* teacher *May-Be* professor

and finally replaces the relationship pair (*Is-Indirectly-Associated-With*, *May-Be*) between student and professor, with the secondary relationship *May-Be-Indirectly-Associated-With*, since none of the teachers the student *Is-Indirectly-Associated-With* may be professors (they may all be instructors).

- The function AGG above corresponds to a selection function over sets of path expressions consistent with a given vague path expression. As its primary discriminator AGG uses an ordering amongst the relationship kinds generated by the CON function. This ordering is based on surveys of psychological and cognitive validity of certain orderings amongst object relationships [CHW 88, ChAr 90, ChHe 89, ChHe 88, Bate 1979, WCH 87]. AGG also uses various other discriminators amongst path expressions which are presented in Chapter 4.

3.3 The Path Collapse Function

Each path expression has a single relationship kind (primary or secondary) associated with it, obtained by applying the Path Collapse Function CON to successive pairs of relationship kinds in it. The result represents the kind of relationship that holds between the path expression root class and the last relationship in the path expression. The single relationship kind obtained by applying CON repeatedly to the path expression is called the *collapsed relationship kind or collapsed kind* of that path expression.

The notion of a path collapse function is similar to the work in [CoLo 88] to generate plausible inference rules by the combination of pairs of relationships.

We define the *adjacent relationship collapse function* Ω. Ω operates on *ordered pairs of adjacent relationship kinds* (primary or secondary) and returns a relationship kind (primary or secondary). Ω is used to replace an ordered pair of adjacent relationship kinds by a single relationship kind that is semantically equivalent to the relationship designated by the original relationship kind ordered pair. For example,

Ω *(Is-Associated-With, Is-Associated-With) = Is-Indirectly-Associated-With*

Ω *(Is-Indirectly-Associated-With, May-Be) = May-Be-Indirectly-Associated-With*

Depending on the types of secondary relationships defined, Ω can vary greatly even for the same set of primary relationships. The particular Ω used in our experiments will be presented in Chapter 4.

The Path Collapse Function CON operates on an ordered sequence of relationship kinds (primary or secondary). This sequence is the sequence of primary relationships in a path expression, or the sequence of secondary and primary relationships at an intermediate stage in the collapse of a path expression. For the sequence of relationships

$$\Sigma = \gamma_1, \ \ldots, \ \gamma_n$$

CON is defined as follows.

1. CON $(\Sigma) = \Omega(\gamma_1, \gamma_2)$ when n = 2.

2. CON (Σ) = CON(CON (...(CON $(\gamma_1, \gamma_2), \gamma_3)..., \gamma_{n-1}), \gamma_n)$ otherwise.

For the path expression

$$\psi = s\ \phi_1 l_1\ ...\ \phi_k l_k$$

where

- s is the root class of the path expression ψ.
- $\forall_{i=1}^{k}\ \phi_i$ is a primary relationship.
- $\forall_{i=1}^{k}\ l_i$ is a relationship label of a relationship in the schema graph having kind ϕ_i.

the collapsed relationship C_ψ of ψ is given by

$$C_\psi = \text{CON}\ (\phi_1,\ ...,\ \phi_k)$$

We will use the CON function interchangeably with the path expression and the ordered sequence of connectors in the path expression, wherever there is no ambiguity. For the path expression ψ above, the expressions

$$C_\psi,\ \text{CON}\ (\psi),\ \text{ and CON}\ (\phi_1,\ ...,\ \phi_k)$$

denote exactly the same quantity - viz. the collapsed relation obtained by the left to right application of CON to successive pairs of relationship connectors in the path expression.

For example, consider the path expression

$\psi' =$ *teaching-asst* @> *grad* @> *student* @> *person* . *name*

The sequence of relationships in ψ' above, is

$\Sigma =$ *Isa*, *Isa*, *Isa*, *Is-Associated-With*

the following sequence of steps yields the collapsed relation $C_{\psi'}$ of ψ' [1]

1. $C_{\psi'}$ = CON (CON (CON (*Isa*, *Isa*), *Isa*), *Is-Associated-With*)
2. $C_{\psi'}$ = CON (CON (*Isa*, *Isa*), *Is-Associated-With*)
3. $C_{\psi'}$ = CON (*Isa*, *Is-Associated-With*)
4. $C_{\psi'}$ = *Is-Associated-With*

Each path expression ψ in the set Ψ of path expressions consistent with a given vague path expression η, has an associated collapsed relation C_ψ.

[1] Ω (*Isa*, *Isa*) = *Isa*, and Ω (*Isa*, *Is-Associated-With*) = *Is-Associated-With*.

Chapter 4

THE PATH COLLAPSE AND SELECTION FUNCTIONS

This chapter presents the particular path collapse function and path selection function used in our implementation. We begin with an introduction to secondary relationships and our path collapse function CON. We then present the various discriminators used by the path selection function AGG. We conclude by presenting various properties of AGG and CON that are important for the purposes of a schema traversal algorithm.

4.1 Collapsing Relationships in Path Expressions

This section presents the particular path collapse function used in our implementation. We begin by introducing the *secondary* relationship kinds defined by this function and then present the collapse function Ω. [1]

4.1.1 Secondary Relationships

The collapse functions define the following additional relationships that may hold between the classes at the two end points of a path expression. Just like primary relationships, secondary relationships may be directed and have a dominant interpretation.

1. **Contains (Is-Contained-By)**
   The *Contains* and *Is-Contained-By* relationship kinds are used to capture the relationship between two classes that are related by some sequence of set and part relationships *having their dominant interpretations in the same direction.* These relationships are also known as *meronymic* relationships from the Greek words *meros*, which means part, and *onoma*, which means name, since they are essentially structural relationships between objects. A meronymic relationship between two classes indicates that objects of one class are contained structurally in objects of the other class. For example, in Figure 2.1 we combine the two path expressions

   (i) departments *Is-Set-Of* department, and
   (ii) department *Has-Part* faculty

   to give

[1] Recall that CON is defined in terms of Ω.

departments *Contains* faculty

The dominant interpretation of a meronymic relationship is *Contains.* We use the symbols, **MER>** and **<MER** to designate the *Contains* and *Is-Contained-By* relationships respectively.

2. **Sharing**
This relationship between two classes a and b, indicates that members of these classes may share some objects of a third (common) class between them, where the common class is related to a and b by either a part, set, or meronymic relationship. The *sharing* relationship is undirected and hence has no dominant interpretation. However it has two *flavours*, depending on how the sharing occurs:

   (a) If the classes a and b share objects of a class that they either have as a part or as a collection member, then the *sharing* relationship is called a *Shares-SubParts-With* relationship, denoted by $\cdot_{FB}$. For example, we combine the two path expressions

      (i) engine *Has-Part* screw, and

      (ii) screw *Is-Part-Of* chassis

   to give

      engine *Shares-SubParts-With* chassis

   (b) If the classes a and b are members or parts of a common class, then the *sharing* relationship is called a *Shares-SuperParts-With* relationship, denoted by $\cdot_{BF}$. For example, we combine the two path expressions

      (i) helicopter-rotor-motor *Is-Part-Of* helicopter-rotor-assembly, and

      (ii) helicopter-rotor-assembly *Has-Part* helicopter-rotor-shaft

   to give

      helicopter-rotor-motor *Shares-SuperParts-With* helicopter-rotor-shaft

3. **Is-Indirectly-Associated-With**
This relationship is an isotropic undirected relationship similar to the *Is-Associated-With* primary relationship. This relationship denotes the fact that an object is indirectly associated with an object of another class through objects of another (or more than one) class. The *Is-Indirectly-Associated-With* relationship is denoted by the symbol $\cdot\cdot$.

   For example, we combine the two path expressions

      (i) department *Is-Associated-With* student, and

      (ii) student *Is-Associated-With* courses

   to give

      department *Is-Indirectly-Associated-With* courses

   which essentially captures the fact that a particular department is indirectly associated with the sets of courses taken by each of it's students.

4. **May-Be Versions of Relationships**
Each of the primary and secondary relationships has what is termed a *May-Be version* of the relationship. A *May-Be version* of a particular relationship γ between two classes a and b,

indicates that *some*, (as opposed to all), objects of class **a** maybe related to some objects of class **b** by the relationship γ.

For example, in Figure 2.1 we combine the two path expressions

(i) courses *Is-Associated-With* teacher, and

(ii) teacher *May-Be* professor

to give

courses *May-Be-Associated-With* professor

which captures the notion that some sets of courses maybe taught by a professor, but not all courses need be taught by a professor.

The *May-Be version* of a relationship γ is denoted by placing a $\star$ immediately after the symbol for the relationship γ. Hence, the *May-Be version* of the *Has-Part* relationship ($>), is denoted by $>⋆ . Note, however, that the *May-Be version* of the *Isa* relationship (@>), is simply *May-Be* (which is denoted by <@).

4.1.2 The Collapse Function

With the secondary relationships above, and the primary relationships defined in Chapter 2, we present the *adjacent relationship collapse function* Ω. Ω operates on an ordered pair of adjacent relationship kinds (primary or secondary), and returns a result relationship kind that designates the kind of relationship obtained by combining the two input relationships in order.

Input	@>	<@	$>	<$	=>	<=	MER>	<MER	·	·FB	·BF	··
@>	@>	<@	$>	<$	=>	<=	MER>	<MER	·	·FB	·BF	··
<@	<@	<@	$>⋆	<$⋆	=>⋆	<=⋆	MER>⋆	<MER⋆	·⋆	·FB⋆	·BF⋆	··⋆
$>	$>	$>⋆	$>	·FB	MER>	·FB	MER>	·FB	··	·FB	··	··
<$	<$	<$⋆	·BF	<$	INV	<MER	·BF	<MER	··	··	·BF	··
=>	=>	=>⋆	MER>	·FB	=>	·FB	MER>	·FB	·	·FB	··	··
<=	<=	<=⋆	INV	<MER	INV	<=	INV	<MER	··	INV	·BF	··
MER>	MER>	MER>⋆	MER>	·FB	MER>	·FB	MER>	·FB	··	·FB	··	··
<MER	<MER	<MER⋆	·BF	<MER	INV	<MER	·BF	<MER	··	··	·BF	··
·	·	·⋆	··	··	·	··	··	··	··	··	··	··
·FB	·FB	·FB⋆	··	·FB	INV	·FB	··	·FB	··	··	··	··
·BF	·BF	·BF⋆	·BF	··	·BF	··	·BF	··	··	··	··	··
··	··	··⋆	··	··	··	··	··	··	··	··	··	··

Figure 4.1: The Adjacent Relationship Collapse Function Ω

We present Ω as a table in Figure 4.1. The entries are read as follows:

- The value of Ω(r, c) is the entry in the table at row r and column c.
- If a particular entry in the table is marked as **INV** , it indicates that such a sequence of adjacent relationship kinds is *invalid*, as it is precluded by the structural constraints of our data model.
- The collapse function treats the *index* relationships *as though they do not exist.* Hence for an index relationship ϕ_i and any relationship γ

$$\Omega(\gamma, \phi_i) = \Omega(\phi_i, \gamma) = \gamma$$

- For the sake of conciseness, we have not shown the entire table. The remaining part of the table consists of the entries where either one (or both) of r or c is a *May-Be version* of a relationship kind. For any such case, the value of the resulting relationship is a *May-Be version* of the value of $\Omega(r', c')$, where r' and c' are the *non May-Be versions* of r and c. Put differently, once any of the input relationships to Ω is a *May-Be version*, the result relationship will always be a *May-Be version.*

4.2 Path Expression Discriminators

The AGG selection function is used to select the most cognitively plausible path expressions from a set of path expressions consistent with a given vague path expression. The AGG selection function uses various possible discriminators for a set of path expressions in decreasing order of importance. We present them one by one, below, in the same order.

4.2.1 The Ordering of Collapsed Relationship Kinds

Each path expression has a final collapsed relationship kind obtained by repeatedly applying Ω to the ordered sequence of relationships in the path expression. Based on cognitive science and psychology studies of semantic relations [Herr 87, MiFe 91, CoQu 69, ChHe 84, WCH 87, ChHe 88, ChHe 89, ChAr 90, CHW 88], we constructed an ordering of importance of collapsed relationship kinds of path expressions. The set of possible collapsed relationship kinds forms a partial order. The partial order used in our experiments is shown pictorially in Figure 4.2.

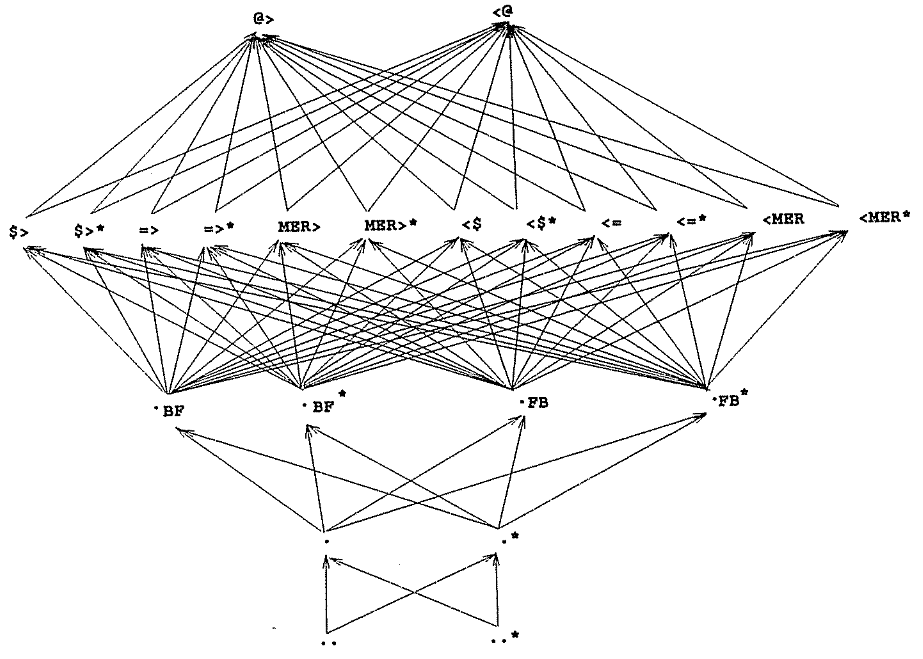

Figure 4.2: A Partial Order For Collapsed Relationship Kinds

There are a few noteworthy points about the ordering of relationship kinds in Figure 4.2.

- Two relationships γ_1 and γ_2 are said to be *incomparable*, iff there is no path from γ_1 to γ_2, and vice versa. Every relationship is incomparable to itself.
- The arrows in the diagram, denote a *worse than* relationship. A relationship γ_1 is said to be *worse than* a relationship γ_2, (denoted $\gamma_1 \prec \gamma_2$), if there exists a path from γ_1 to γ_2 in the diagram. If a path exists between two relationships, they are said to be *comparable.* The $\prec$ operator is transitive.
- We consider all the structural relationships equally important and hence all structural relationships are incomparable to each other. Further, the direction of the relation for a structural relationship, (*Has-Part* or *Is-Part-Of*), isn't important when distinguishing amongst them.
- A *May-Be version* of a relationship and the relationship itself, are indistinguishable for the purposes of this particular ordering.

It may be noted that there are many plausible, alternate orderings of relationship kinds. We found that the ordering in Figure 4.2 yielded the most accurate results most of the time from all of the orderings we tried.

The *worse than* operator $\prec$ for collapsed kinds is the primary discriminator for AGG. Since $\prec$ defines a partial order on relationship kinds, two relationship kinds may be incomparable to each other. Relationship kinds γ_1 and γ_2, are said to be incomparable on the basis of the $\prec$ ordering, iff *neither of the following holds*

1. $\gamma_1 \prec \gamma_2$, or
2. $\gamma_2 \prec \gamma_1$

We retain only those path expressions from the set Ψ of path expressions consistent with a vague path expression whose collapsed kinds are *not worse than* the collapsed kind of another path expression in Ψ. The paths in this subset are indistinguishable from each other on the basis of importance of collapsed kinds.

4.2.2 Isa Sub-Path Preemption Criterion

This is a structural discriminator based on the fact that a subclass of a given class may redefine a property of the class. The inheritance property of object oriented systems states that a property not defined by a particular class is inherited from the *nearest* superclass in which it is defined.

For two path expressions

$$\psi_1 = s \;@\!> l_1 \;@\!> l_2 \ldots @\!> l_j \; \phi_1 L \quad \text{and}$$

$$\psi_2 = s \;@\!> l_1 \;@\!> l_2 \ldots @\!> l_j \ldots @\!> l_k \; \phi_2 L$$

having *incomparable collapsed relationships*, C_{ψ_∞} and $C_{\psi_\in}$, the path expression ψ_1 is deemed preferable to the path expression ψ_2. The root class s inherits the property L from class l_j, not its superclass l_k. This situation is shown pictorially in Figure 4.3. When such a case arises, path expression ψ_1 is said to *preempt* path expression ψ_2, and path expressions ψ_1 and ψ_2 are said to satisfy the conditions of the Isa Sub-Path Preemption Criterion. Note that path expressions ψ_1 and ψ_2 must share a common sub-path of *Isa* relationships

$$s \;@\!> l_1 \;@\!> l_2 \ldots @\!> l_j$$

for one path to preempt the other.

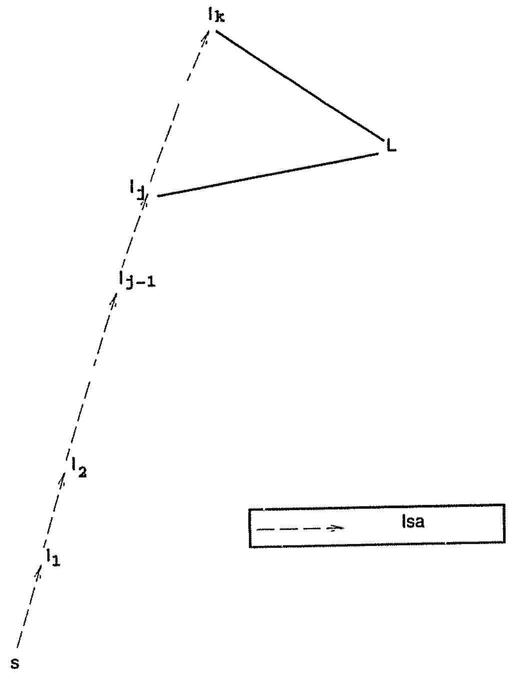

Figure 4.3: The Isa Sub-Path Preemption Criterion

4.2.3 Effective Lengths of Path Expressions

There is a further discriminating factor used to choose from amongst path expressions having *incomparable collapsed relations.* Cognitive Science studies support the assertion that *concepts with greater semantic distance between them are considered less plausible by humans than corresponding concepts with a lesser semantic distance* [CoQu 69]. We attempt to capture the notion of semantic distance by the heuristic of *effective length* of a particular path expression. The effective length of a path expression is a measure of how far apart (semantically) the concepts of the root class and the final relationship are for that particular path expression. The effective length notion should also try and capture quantities such as number of relationship reversals encountered when traversing a path expression.

Since effective length is a heuristic we could define any arbitrary function to compute the effective length of a particular path expression. At first glance, actual length may seem to suffice. However, this is not a very satisfactory measure of the semantic distance. Consider the path expression which is simply a long chain of contiguous *Isa* relationships. It is obvious that such a chain is equivalent to a single *Isa* relationship. Similar arguments can be made for chains of contiguous part relationships of the same relationship kind.

The system should also provide a user settable parameter N_EFF_LENGTHS, that specifies how many best effective length values to retain in the final answer presented to the user. The better the definition of the effective length function captures the notion of semantic distance, the smaller the value of N_EFF_LENGTHS. We will present the effective length function used in our experiments in Chapter 5.

4.2.4 Domain Specific Discriminators

A final type of discriminator that can be used is domain knowledge about the type of path expressions sought, where available. For example, a user may want to specify that he/she is really *not interested* in a certain part of the database. If this sort of information can be stated in some declarative form to the system, then the system need not even consider answers that deal with the particular part of the database the user is not interested in.

Another powerful discriminator is a subsumption check. If a database can provide a system with an answer size estimator for path expressions, the system could toss path expressions that may be subsumed by more focused path expressions.

The domain specific discriminators, where available, are useful in terms of keeping the search more focused and pruning extraneous answers. However, they are not an essential requirement for the completion procedure.

The function AGG takes a particular path expression ψ and a path expression set Ψ as input and returns a set of path expressions that are indistinguishable on the basis of the discriminators presented above.

4.3 Properties of AGG and CON

We present below, several important properties that our collapse function CON, and our selection function AGG satisfy. These properties are exploited by the algorithm while traversing the schema graph.

(I) **CON is Associative**
The function Ω, (and hence CON), is associative. That is, for any three valid adjacent relationships γ_1, γ_2, and γ_3,

$$\text{CON}\ (\gamma_1, \text{CON}\ (\gamma_2, \gamma_3)) = \text{CON}\ (\ \text{CON}\ (\gamma_1, \gamma_2), \gamma_3)$$

The associativity of the path collapse function implies that collapsing the relationships in a path expression *in any arbitrary order*, will yield the same collapsed relationship for that path expression.

(II) **AGG is Order Independent**
For any two arbitrary path expressions ψ_1 and ψ_2, and an arbitrary path expression set Ψ,

$$\text{AGG}\ (\psi_1, \text{AGG}\ (\psi_2, \Psi)) = \text{AGG}\ (\psi_2, \text{AGG}\ (\psi_1, \Psi))$$

This implies that we can compare a new path expression against all existing ones so far in any arbitrary order and get back the same set of path expressions.

(III) **CON is Monotonic**
For any relationship kinds γ_1, and γ_2,

$$\text{CON}(\gamma_1, \gamma_2)\ \textit{is always worse than or incomparable to}\ \gamma_1$$

This property implies that if the collapsed kind of an initial fragment of a path expression ψ_1 is worse than the collapsed kind of another path expression ψ_2, then $C_{\psi_\infty} \prec C_{\psi_\in}$ always holds, *irrespective of the remaining relationships in* ψ_1. The monotonicity property helps us perform a branch and bound type traversal of the schema graph.

Chapter 5

THE COMPLETION ALGORITHM

This chapter presents our algorithm to generate completions of vague path expressions. We begin by presenting the algorithm declaratively as a four stage domain independent procedure. We then present reasons why none of the standard path computation algorithms in the literature are applicable. We present two distinct ways to deal with intermediate points in vague path expressions, and justify our choice of one of them. The next section presents our algorithm along with optimizations for efficient traversal of the schema graph.

5.1 A Procedure To Generate Plausible Completions

We present below, a domain independent procedure to generate plausible path expressions matching a given vague path expression. The procedure is divided into four main stages followed by a post processing stage.

Given a vague path expression

$$\eta = s\ \phi_1 l_1\ \phi_2 l_2\ ...\phi_k l_k$$

where,

1. s is the name of the root of the path expression(s) corresponding to this vague path expression.
2. $\forall_{i=1}^{k}\ \phi_i$ is a valid relationship connector symbol, and $\exists_{j=1}^{k}\ \phi_j = \tilde{}$
3. $\forall_{i=1}^{k}\ l_i$ is the label of a relationship in the schema graph.

the procedure generates the set Ψ_{best} of the most cognitively plausible path expressions that are consistent with η through the following stages.

(I) **Generate Consistent Path Expressions**
Given η, we generate the set Ψ of all valid path expressions consistent with η in the schema. This set is usually quite large, and hence has to be pruned of most of its members.

(II) **Prune Ψ using the $\prec$ operator**
Do a pairwise comparison of the path expressions in Ψ using the $\prec$ operator to compare the collapsed relations of the path expressions on the basis of the partial ordering of collapsed relationship kinds. The set of path expressions remaining in Ψ after this pairwise comparison is considerably smaller than the original set.

```
foreach $\psi_i, \psi_j \in \Psi$ do
    $\mathcal{C}_{\psi_\rangle} = \mathrm{CON}\ (\psi_i)$
    $\mathcal{C}_{\psi_|} = \mathrm{CON}\ (\psi_j)$
    if $\mathcal{C}_{\psi_\rangle} \prec \mathcal{C}_{\psi_|}$ then
        remove $\psi_i$ from $\Psi$
    else if $\mathcal{C}_{\psi_|} \prec \mathcal{C}_{\psi_\rangle}$ then
        remove $\psi_j$ from $\Psi$
    fi
od
```

(III) **Apply Additional Discriminators**
For each pair of path expressions ψ_i, ψ_j that satisfy the conditions of the Isa Sub-Path Preemption Criterion, remove from Ψ the path expression that is preempted.

(IV) **Apply Heuristic Discriminators**
From the paths remaining in Ψ, retain only those paths with the best N_EFF_LENGTHS effective lengths.

(V) **Post Processing with Domain Specific Discriminators**
Apply any available domain specific discriminators to discard paths from those left in Ψ.

The set of path expressions remaining in Ψ after stages (I) through (V), is the set Ψ_{best} that is returned to the user as the most plausible path expressions.

Stages (II) through (IV) (and stage (V) where possible), comprise the AGG selection function for path expressions. Stages (I) through (IV) are completely devoid of any knowledge about the specific database. They operate on path expressions solely on the basis of the structure of the path expressions. Stage (V) is the only place where any existing domain specific knowledge is used. This implies that the entire procedure (Stages (I) through (IV)) should work as well on any database domain.

5.2 Inapplicability of Existing Algorithms

All the path computation algorithms in the literature are based on the following crucial property of of the AGG selection function :

AGG imposes a total order on path expressions

For our particular problem, AGG *never* forms a total order owing to the fact that many collapsed kinds are simply *incomparable* (in terms of being 'more intuitively plausible'), to other relationship kinds. In the general case, AGG is a *partial order*. This implies that, in general, AGG *may not even have a distinct identity* Φ, as required by condition (4) in Section 3.1.

In addition, all the algorithms [IRW 93] assume that AGG deals with a pair of single path expressions at a time. Due to the fact that AGG is a partial order, our AGG function needs to deal with a *set of path expressions.* This implies that conditions (5) and (6) in Section 3.1, *will not be satisfied in the general case.*

Furthermore, all the path computation algorithms presented in the literature assume either one of the two following cases :

1. Computation of the optimal path over all possible paths in a graph.
2. Computation of the optimal path over the set of all paths starting at a given node in the graph.

Our problem requires, the optimal path from the set of paths starting at the given vague path expression root, *and having arcs labeled* $l_1, l_2, ..., l_k$ *in the same relative order as those in the vague path expression.* This is a more general version of the path computation problem.

More importantly, the problems with AGG imply that none of the existing path computation algorithms are directly applicable to our problem. We present our algorithm to do efficient path computation below.

5.3 Dealing With Intermediate Point Specifications

Consider a vague path expression of the form

$$\eta = s \; ... \; \phi_n l_n$$

1. If n = 1, then η is termed an *end point specification*, as any path expression consistent with η simply needs to have s as the root class, and l_n as the last relationship (end point) of the path expression. For example,

   undergrad ˜ faculty

   is an end point specification for the schema of Figure 2.1.
2. If n $\neq$ 1, then η is termed an *intermediate point specification*, and the relationships $\phi_1 l_1, \phi_2 l_2, ..., \phi_{n-1}, l_{n-1}$ are termed the *intermediate points* of the vague path expression. For example,

   undergrad ˜ course ˜ faculty

   is an intermediate point specification for the schema of Figure 2.1.

For a path expression ψ to be consistent with an intermediate point specification, it must have root class s and the sequence of labels and connector symbols obtained by removing all instances of the ˜ connector symbol from η must be a subsequence of those in ψ, where a connector-label pair in η is treated as a single element of the sequence for all connector symbols $\neq$ ˜ .

In general, our algorithm for generating path expressions consistent with an end point specification does not allow any cycles in the path expression (except cycles through *invisible* collection classes), for the reasons presented in Section 2.3.2. A cyclic path expression however may often denote something quite meaningful. For example, the path expression

undergrad @> student . courses => course <= courses . student <@ grad

is a perfectly reasonable path expression for the schema of Figure 2.1. It designates all graduate students who are taking a course taken by at least one undergraduate. We take the approach that, if the user wants a path expression to cycle through a class (or classes), then he/she should use an intermediate point specification and may explicitly specify the required cycle.

There are two ways one can deal with intermediate point specifications.

5.3.1 Restricted Regular Expressions

An intermediate point specification could be treated as a restricted version of a regular expression matching a path in a graph, similar to the work of [MeWo 89]. The ˜ connector symbol in the specification acts as a wildcard specifier for that part of the expression. The path expression so generated is restricted in the sense that it is not allowed to have any cycles (except cycles through *invisible* collection classes). For example, the intermediate point specification

undergrad ˜ *course* ˜ *grad*

would yield the path expression

- *undergrad* @> *student* . *courses* => *course* <= *courses* . *teacher* <@ *instructor* <@ *teaching-asst* @> *grad*

from the schema graph of Figure 2.1 with this approach.

Since this approach does not allow cycles it is restricted in the types of path expressions it can return. Ideally we'd like an approach to intermediate point specifications that is tractable, allows cyclic path expressions, but does not overwhelm the user with a large number of meaningless cyclic path expressions.

5.3.2 Ordered Sequence of End Point Specifications

This approach treats an intermediate point specification as a series of end point specifications. The problem is thus subdivided into a number of smaller problems. Each of the end point specifications yields sets of 'best' path expressions. The path expressions in these sets are concatenated (in the same order as the intermediate points in the intermediate point specification), to give the final set of 'best' path expressions. Here too, we do not allow cycles when generating path expressions consistent with the end point specification subproblems.

Such an approach is based on our contention that when we connect up a number of entities our minds treat the likely connections between each pair of entities as a separate problem, rather than one long problem. This approach obviously generates cyclic path expressions, and is in fact faster than the restricted regular expression approach above. Further, this approach to intermediate points produces surprisingly few meaningless cyclic path expressions and generates path expressions that the restricted regular expression approach cannot. For example, the intermediate point specification

undergrad ˜ *course* ˜ *grad*

yields both path expressions

1. *undergrad* @> *student* . *courses* => *course* <= *courses* . *teacher* <@ *instructor* <@ *teaching-asst* @> *grad*
2. *undergrad* @> *student* . *courses* => *course* <= *courses* . *student* <@ *grad*

from the schema graph of Figure 2.1.

On the basis, of the above observations, we chose this approach to deal with intermediate point specifications. All the results reported in Chapter 6 are based on this approach, which *performed consistently better than the restricted regular expression approach on all the experiments.*

5.4 The Traversal Algorithm

We present below, the algorithm for traversing a schema graph to generate the 'best' path expressions consistent with a given vague path expression. As typical schema graphs in most real life databases have hundreds of classes and relationships, we would like to optimize our traversal of the schema to the extent possible. Further, the theory developed in this thesis is meant to help the user formulate complete queries quickly. If the system takes more than a few minutes to come up with meaningful suggestions, then its usefulness is severly limited. Since none of the path computation algorithms are applicable to our particular problem, we developed our own schema graph traversal algorithm. We first present our optimizations, followed by the pseudo code to calculate the effective length of a path expression. In subsequent sections we present the pseudo code for a end point specification traversal algorithm, and the top level procedure that handles intermediate point specifications.

5.4.1 Caution Sets

For two comparable collapsed relationship kinds γ_1 and γ_2, the $\prec$ operator can be thought of as defining a function BETTER that returns the better of the two relationship kinds. That is

$$\begin{aligned} \text{BETTER } (\gamma_1, \gamma_2) &= \gamma_1 \text{ if } \gamma_2 \prec \gamma_1 \\ &= \gamma_2 \text{ otherwise.} \end{aligned}$$

If the BETTER and CON functions were to obey the distributivity property

$$BETTER(CON(\gamma_1, \gamma_3), CON(\gamma_2, \gamma_3)) = CON(BETTER(\gamma_1, \gamma_2), \gamma_3) \tag{5.1}$$

for any three arbitrary relationship kinds, γ_1, γ_2, and γ_3, then a traversal algorithm could retain the best collapsed kind from a root class to a given class node in the schema, and when revisiting this class node, continue from this node only if the collapsed kind of the new path fragment to this class node is not worse than that already stored. This corresponds to pushing the selection (on the basis of the $\prec$ ordering) before the collapsed kind computation of the entire path expression. The Left Hand Side (LHS) of Equation 5.1 above, denotes the computation of both paths having a common node, and a sub-path from this node with the same collapsed kind, followed by a selection of the best one on the basis of their collapsed kinds. The Right Hand Side (RHS) corresponds to choosing the path with the best collapsed kind for the initial sub-path to the common node, and only generating this path. The LHS is what we have to do to generate all paths and take the best one. The RHS however, offers the opportunity of a significant speedup in the time complexity of the traversal algorithm, if the property in Equation 5.1 holds.

The property above is never violated, insofar as there is no case when two distinct relationship kinds appear on the LHS and RHS of Equation 5.1. However, due to the fact that certain relationship kinds are incomparable to each other, two types of interesting cases arise:

1. When the LHS has two incomparable relationship kinds, while the RHS has just one of them. This implies that pushing the selection ahead will tend to lose a possible path expression that may be as good as [1] the one found. We call such cases *two-one cases.* An example of a two-one case is the following:

   $$\gamma_1 = \$> \ , \gamma_2 = \ \cdot \ \ , \gamma_3 = \ \cdot$$

[1]The path expression lost may be 'better' than the one retained in terms of discriminators like effective length.

LHS = { .. , .. } (two distinct paths)
RHS = { .. } (one path lost)

2. When the RHS has two relationship kinds, while the LHS has just one. Such cases arise when the collapsed kinds of the initial sub-paths to the common node are incomparable, but the collapsed kinds of the entire paths are comparable, since the RHS of Equation 5.1 has only one call to the selection function BETTER. For example, if

   $\gamma_1 = \ \cdot_{FB}$, $\gamma_2 = \ \cdot_{BF}$, $\gamma_3 =$ \$>
   LHS = { $\cdot_{BF}$ }
   RHS = { .. , $\cdot_{BF}$ }

   We call such cases *one-two* cases. One-two cases are not really a problem. They are easily removed by placing an extra call to BETTER on both sides of Equation 5.1.

Two-one cases are a problem however. If performing a selection first implies that we may lose a valid path then it is obviously not correct. We deal with two-one cases by introducing the notion of a *caution set* for each relationship kind. The caution set of a relationship kind γ_1 is the set of all relationship kinds γ_2, which are worse than γ_1, but which, when combined with a third relationship kind γ_3, via CON, yield a relationship that is equivalent or incomparable to the relationship CON(γ_1, γ_3). Now the traversal algorithm can still perform a traversal equivalent to the RHS of Equation 5.1 - however, we have to generate both paths if the previous relationship kind at the common node is in the caution set of the collapsed kind of the new sub-path to this node.

5.4.2 Relationship Snoop

For a vague path expression

$$\eta = s\ \phi_1 l_1\ \phi_2 l_2\ \ldots\ \phi_n l_n$$

a (ϕ_i, l_i) pair is termed a *search target pair*, and the (ϕ_i, l_i) pair the algorithm is currently trying to match is termed the *current search target pair.* If $\phi_i = \ \tilde{}$, then the traversal algorithm looks for a relationship labeled l_i in the path.

As part of the initialization phase, the relationships at each class node are sorted and ordered according to importance. If the traversal algorithm is looking for a particular relationship labeled l_i from a given class node, it may have to search along paths from all the relationships ordered before l_i at that node. For large schemas this can translate into an unacceptably large amount of computation time exploring useless paths, even though the relationship needed is *right there.* We implement a *snoop* optimization to avoid this. When the traversal algorithm checks the first relationship at a given class node, it snoops through all the relationships at that class node to check whether one of them matches the current search target pair. If the snoop detects a match, then paths out of that relationship are explored first.

5.4.3 Effective Length Calculation

The block of code in Figure 5.1 calculates the effective length of a path expression p.

We treat the set relationships (*Is-Set-Of* and *Is-Member-Of*) as invisible, similar to the treatment of collection classes. For multiple part or inheritance relationships in the same direction, the effective length remains unchanged. Associations and index links contribute to the effective length. Every time a directed relationship reverses direction effective length is incremented. Hence our effective length function captures reversals in direction as well as changes in relation type.

```
lastlink := INVALID ; efflength := 0 ;
foreach relationship r in p do
    case (r) of
        Isa : if (lastlink = May-Be ) then efflength := efflength + 1 fi
        May-Be : if (lastlink = Isa ) then efflength := efflength + 1 fi
        Is-Set-Of , Is-Member-Of :
            if ((lastlink = May-Be ) or (lastlink = Isa)) then
                    lastlink := r fi
        Has-Part : if (lastlink <> Has-Part ) then efflength := efflength + 1 fi
        Is-Part-Of : if (lastlink <> Is-Part-Of ) then efflength := efflength + 1 fi
        Is-Associated-With , Is-Indexed-By , Indexes : efflength := efflength + 1
    endcase
    if ((r <> Is-Set-Of ) and (r <> Is-Member-Of)) then lastlink := r fi
od
```

Figure 5.1: Calculating Effective Length of a Path Expression

5.4.4 Traversing the Schema Graph

The algorithm for traversing the schema graph for end point specifications uses the following global variables and constants.

1. BestRelsSoFar : A set of the best collapsed relationship kinds of the path expressions found so far. Initialized to {}.
2. BestEffLenSoFar : The best effective length from the path expressions found so far. Initialized to **maxint**.
3. PathFound : A boolean variable that is initialized to **false**, and set to **true** as soon as the first path expression is found.
4. Visited [nClassesThisSchema] : A boolean array that is used to avoid cycles. Each element is initialized to **false**.
5. RelSets [nClassesThisSchema] : An array of sets of relationship kinds of the best collapsed kind from the root class to a class node. Each element is initialized to {}.
6. BestEffLength [nClassesThisSchema] : An array of integers of the best effective length of a path from the root class to a class node. Each element is initialized to **maxint**.
7. EFFLENSLOP : A constant value that indicates by how much a given path expression's effective length can vary from the best one found so far, and still be considered good enough.

In addition, as part of the initialization phase, the code constructs the following three sets of relationship kinds for each relationship kind r.

1. WorseSet [r] : Set of all relationship kinds r' s.t. $r' \prec r$.
2. BetterSet [r] : Set of all relationship kinds r' s.t. $r \prec r'$.
3. CautionSet[r] : Set of all relationship kinds r' s.t. $r' \prec r$, and $\exists\, r''$ s.t. CON (r, r'') and CON (r', r'') are incomparable.

The traversal algorithm assumes the existence of the following functions

- get_next_link (node)
  Returns the next link (relationship) to search out from for class **node**. Implements the relationship snoop, and returns NULL if no more relationships match the current search target pair.
- points_to (link)
  Returns the class node in the schema graph that relationship link points to.
- concat_path(oldpath, link)
  Concatenates relationship link onto the path expression oldpath returning a new path expression.
- worsethan (rel, relset)
  Returns **true** if $\exists r \in$ relationship kinds set relset, such that rel $\prec r$, else returns **false**.
- update_paths_found (newpath)
  Adds newpath to set of existing paths found, updates the values of the BestRelsSoFar set and BestEffLenSoFar, and sets PathFound to **true**. This function performs checks for path expression pairs satisfying the *Isa Sub-Path Preemption Criterion*, as well as comparisons using additional and domain specific discriminators, of newpath to all existing paths.

The traverse function in Figure 5.2 has the following three input parameters

1. thisnode : The current class node of the recursion.
2. psf : The path expression so far from the root class to thisnode.
3. frominvis : A boolean value indicating whether this particular call originated from an invisible class and reached thisnode via an *Is-Set-Of* link.

The traverse algorithm is presented in the form of a Depth First Search function in Figure 5.2. Our traversal differs from pure depth first search in a number of important ways

- Monotonicity allows us to terminate a search down a particular path if the collapsed relation of the sub-path from the root class is worse than the collapsed relation of a previously found path (line no. 8).
- If the effective length of the current path exceeds the sum of BestEffLenSoFar + EFFLENSLOP, then search is terminated along this path, since effective length is a monotonically increasing function of actual length (line no. 9).
- Recursion along a path with collapsed relation c to a class node n is only continued, if

  There does not exist a relationship r in RelSets [n], such that

  $r \in$ BetterSet[c],

$r \notin$ CautionSet$[c]$, **and**
effective length from root class to n $\leq$ BestEffLength[n] + EFFLENSLOP.

(line nos. 15-17).

- Since we have to deal with invisible classes as efficiently as possible, each recursive call passes back a *traverseret* structure containing two fields

    1. matchedtarget : A boolean field indicating that the current target was matched while descending an *Is-Set-Of* relationship.
    2. augmentedpath : The augmented path got by descending the *Is-Set-Of* relationship chain to the first visible class, and then ascending back to the invisible collection class.

    The *traverseret* return values are only of interest while returning from a series of calls down *Is-Set-Of* relationships (line nos. 20-23). In all other cases, the value of *traverseret* is NULL.

Note that although we separated the effective length heuristic discriminator from the collapsed relation discriminator in our declarative presentation of the procedure, we use the fact that effective length is a monotonically increasing function during the traversal of the schema graph itself to speed up the traversal. This optimization generates considerably fewer path expressions that have to be discarded on the basis of effective length considerations, as well as significantly improves the time of traversal on large schemas.

5.4.5 Matching Intermediate Point Specifications

As explained in Section 5.3.2, we treat an intermediate point specification as an ordered sequence of end point specifications. The classes that the last relationships in the path expressions consistent upto the previous search target pair point to, become the root classes of the path expressions for the next search target pair. We repeat this process for each search target pair, essentially treating it as a different problem. After each stage, the sets of path expressions obtained for each search target pair are concatenated with the path expressions set obtained for the previous search target pair. This may result in some of the newly formed path expressions being discarded owing to their final collapsed kinds, (since CON is associative, the final collapsed kind of each newly formed path expression is obtained by simply applying CON to the collapsed kinds of the two path expressions it is created from), or owing to their final effective length.

Before beginning the end point traversal for each search target pair, the following data structures are reinitialized to the following values :

- BestRelsSoFar := {} ;
- BestEffLenSoFar := **maxint** ;
- PathFound := **false** ;
- Visited []. Set to **false** for each class node.
- RelSets []. Set to {} for each class node.
- BestEffLength []. Set to **maxint** for each class node.

The traversal algorithm in Figure 5.2 is then called for each successive search target pair. The final set of path expressions returned after the call to traverse with the last search target pair is the set of 'best' path expressions for this intermediate point specification.

```
function traverse (thisnode, psf, frominvis)
    Visited [ thisnode ] := true ;
    invis := ((thisnode.type = Collection) and (psf.lastlink $\neq$ Is-Member-Of)) ;
    while (link returned by get_next_link(thisnode) $\neq$ NULL) do
        nextnode := points_to(link); newpath := concatpath(psf, link);
        colrel := CON (psf.collapsedrel, link.relation) ;
        if ((Visited[nextnode] = true) or (thisnode is at invalid Isa Crossover) or
            ((PathFound = true) and (worsethan(colrel, BestRelsSoFar))) or
            ($\exists$ a shorter path (effective length) of a comparable relation)) then
                goto next iteration of loop fi
        if (newpath is a complete path) then
                update_paths_found(newpath) fi
        if (nextnode is a leaf node) then
                goto next iteration of loop fi
        if (($\nexists$ r s.t. (r $\in$ RelSet[nextnode]) and (r $\in$ BetterSet[colrel])) and
            (r $\notin$ CautionSet[colrel]) and
            (newpath.efflen $\leq$ (BestEffLength[nextnode] + EFFLENSLOP)) then
            update RelSet[nextnode] and BestEffLength[nextnode] ;
            tretretd := traverse (nextnode,newpath, (invis and (link =Is-Set-Of)));
            if ((invis = true) and (link = Is-Set-Of)) then
                if (frominvis = true) then tret := tretretd fi
                if (tretretd.matchedtarget = true) then return (tret) fi
                psf := concatpath(tretretd.augmentedpath, link.reverse) fi fi od
    Visited [ thisnode ] := false ;
    if ((frominvis = true) and (tret = NULL)) then
            tret.matchedtarget := invis ; tret.augmentedpath := psf fi
    return (tret) ;
```

Figure 5.2: The Traversal Algorithm for End Point Specifications

Chapter 6

THE EXPERIMENTS

In this chapter we present the results of some preliminary experiments on human subjects. We conducted two sets of experiments with two unrelated schemas of vastly differing sizes. The two experiments were intended to test the efficacy of our system on two different tasks. We begin by presenting the parameters used to measure the effectiveness of the system, the motivation behind each set of experiments, the experimental set-up and methodology, and the results obtained.

6.1 Measures of Effectiveness

Two important parameters of information retrieval effectiveness are *recall* and *precision* [Salt 89].

Recall is defined as the proportion of relevant answers retrieved. In our case an answer corresponds to a path expression consistent with a given vague path expression. If the path expression corresponds to what the user had in mind when he/she posed the vague path expression, it is a relevant answer. Sometimes an answer cannot be expressed as a single path expression in a schema, but rather as a set of path expressions which together make up the answer a user has in mind. In such cases all the path expressions in this set are considered relevant. For any information retrieval system to be successful it should have close to perfect (100 %) recall.

Precision is defined as the proportion of retrieved articles that are relevant. It is obvious that one way of achieving a high recall rate is to retrieve all possible answers. Precision is a parameter that provides a measure of the percentage of retrieved answers that are irrelevant.

An ideal system has recall and precision values equal to 100 %. In reality the two parameters are often approximately inversely proportional to each other reflecting the trade-off inherent in the two. We define a quantity we term the *Recall-Precision Product* that measures the product of the two values. This quantity provides a rough guide as to the effectiveness of a particular information retrieval system.

The main input that our system accepts is the number of best effective length values used to choose the answers to a query. Informally, the number of best effective length values used corresponds to the 'pickiness' of the system. The greater this number the more answers retrieved, but precision is obviously affected. All our results plot either recall or precision, or the recall-precision product as a function of the number of best effective lengths.

6.2 Motivation

As mentioned in Chapter 1, there are two primary motivations behind the work in this thesis.

- The first is to give to *naive* users a tool with which to correctly query a database that contains the information they need, but to which they do not know how to get. Our goal is to not have the user even know what the schema looks like - the user should simply formulate a query in the form of a vague path expression, and the system should come up with a correct completion. An English language description of the database should suffice for the user to be able to query it. Our first set of experiments is based on the assumption that the user simply knows about the entities in a database and has a loose idea about how they are related.

- A second use we envisage for this work is for experienced database users and schema designers. As mentioned before, typical database schemas contain hundreds of classes and relationships amongst them. Specifying a query on such large schema graphs is sometimes a daunting prospect even for the schema designer. A GUI is not much help here as the schema is typically too large and richly connected to fit into one window. Since the DBMS requires the user to explicitly conform to the structure of the schema while querying, specifying queries on such schemas is laborious, even for database experts. Our second experiment was conducted on a real life schema in the MOOSE OODBMS [Wilo 93], to determine to what degree a system like ours can aid schema designers and database administrators in querying the database.

We present below details about both sets of experiments and results obtained.

6.3 Experiment 1 - Formulating Queries For Naive Users

6.3.1 Experimental Methodology

This set of experiments was actually split into four different parts. The motivation behind all four parts was to test how accurately our system allows naive users having limited knowledge about the relationships between entities in a database, to pose queries in the form of vague path expressions, and receive completed queries that are consistent with their intuition. We chose our set of human subjects carefully, so as not to bias our experiments toward one particular group of people.

We used nine human subjects for our experiments (eight graduate students and one staff member). Two of the subjects are senior grads working in the field of artificial intelligence, two are doing graduate work in databases, two are doing graduate work in operating systems, one in business, and one in textile science. The staff member chosen was the designer of the MOOSE schema used in the second experiment. Our motivation behind choosing such a group was to have people both familiar and unfamiliar with computers, as well as to have some people familiar with databases and database schemas and some not.

The schema we chose for our first set of experiments was the schema of Figure 2.1. The world it models is familiar and not too detailed, so it could be described in a paragraph or two. On the other hand, even a relatively simple schema such as that of Figure 2.1 captures almost all our intuition about a restricted university world.

The subjects were not shown the schema. All they were given was a description of a particular world reproduced in Figure 6.1. They were then asked to do the following tasks based on this description. [1]

1. We presented each subject with twelve identical, vague, English statements about this world and asked them to tell us what they thought the statements should mean based on their intuition and common-sense knowledge about the world. The twelve statements were arbitrarily

[1] We did not encourage subjects to draw diagrams to represent the world described. They were asked to fill in any gaps in the description either by asking questions or using their common-sense.

```
The world model we are using for our initial set of experiments
consists of universities and persons and related entities.

-   Each university has a set of departments.
-   Each department has a name and a set of professors who comprise its faculty.
-   All professors have offices which have office numbers. A professor also
has information associated with him/her indicating whether he/she is tenured.
-   All persons in the world have a name and a social security number.
-   A person in the world may be either a student or an employee.
-   An employee has a salary and an id number. Further, an employee may be
either a teacher or staff member (but not both). Each staff member is
associated with a particular university.
-   Each teacher in the world teaches a set of courses. A teacher may be either
an instructor or a professor. Unlike professors, instructors do *not*
have offices.
-   A student is associated with a particular department of a particular
university.  Each student has an id number. A student takes a set of courses.
-   Each course has a name (of the course) and a number of credits.
-   A student may be either an undergrad or a grad student (but not both).
-   An undergrad student has a major.
-   A grad student may be a teaching assistant. A teaching assistant is
a grad student *as well as* an instructor.
```

Figure 6.1: The World Model Description Used in Experiment 1

chosen by us. [2] For each of the statements we ran our system with the corresponding vague path expression and compared its answers against the ones given by the subjects.

2. For each of the statements in part 1 above, we presented the subjects with a series of alternatives (each statement had a minimum of three alternatives and some statements had up to nine alternatives), which essentially corresponded to valid path expressions consistent with the vague path expressions of the statements. We asked the subject to rank the alternatives as below

   (a) If the alternative was the intuitive completion (or part of the intuitive completion), for the vague statement, then subjects were asked to rank it as an (A).

   (b) If the alternative did not instantly spring to mind as an intuitive completion, but was not too far-fetched not to be an intuitive completion, subjects were asked to rank it as a (B).

   (c) If the alternative was too far-fetched to be an intuitive completion for the given vague statement, subjects were asked to rank it as a (C).

   The twelve statements used in parts 1 and 2 are presented in Appendix A.

3. For part 3, we asked each of the subjects to come up with ten vague statements of their own for the world described to them, and tell us what they meant in terms of the completions they expected to see from the system.

[2] In fact two of the statements were rejected as meaningless by a majority of the subjects, and hence the results shown only reflect the subjects' answers on the remaining ten.

4. Finally, for each of the vague statements in part 3, we also provided a series of alternatives and asked the subjects to rank them as (A), (B), or (C) above.

The motivation behind part 2 was to average out any idiosyncrasies of individual subjects for particular statements and contrast how the system performed with respect to part 1 (since it is the same set of vague path expressions we are testing on). Part 3 gave us a chance to test the system on a large set of arbitrary vague statements. Part 4 was used to test whether or not the system could come up with some answers the subject didn't think of initially, but which they would consider intuitive completions for the vague path expressions of part 3.

6.3.2 Results

This section presents the results of our experiments on the student-university world. We considered a path expression to be relevant in each of the parts of experiment 1 as follows :

- For parts 1 and 3 above, the path expression (or expressions) corresponding to each subject's intuitive completion of a vague path expression was considered relevant.
- For part 2, the path expressions corresponding to only those alternatives that received a majority of (A) rankings were considered relevant. Alternatives receiving a majority of (B) and (C) rankings were considered irrelevant.
- For part 4, the path expressions corresponding to the alternatives ranked (A) were considered relevant.

Figures 6.2 and 6.3 show the values of the average recall and precision fractions as a function of the number of best effective length values for part 1 of the experiment. The graphs show that recall does increase initially with the number of best effective lengths, as more relevant answers are retrieved, and then levels off. Precision on the other hand initially decreases as the number of answers retrieved increases, since the percentage of relevant answers in the new answers retrieved keeps decreasing. Recall did not hit a 100 % figure, because there were some answers given by users that were not considered as 'good' in terms of their collapsed kinds as some of the other answers retrieved.

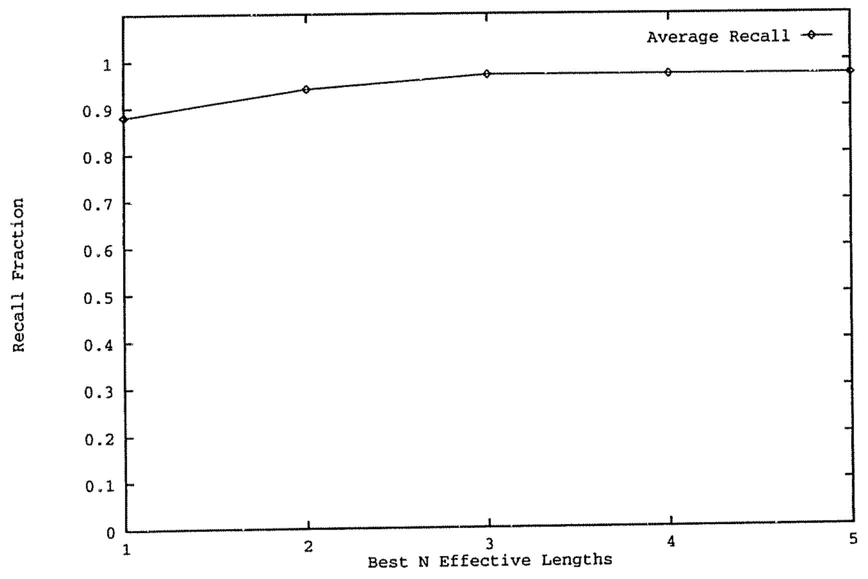

Figure 6.2: Average Recall Fraction (Experiment 1- Part 1)

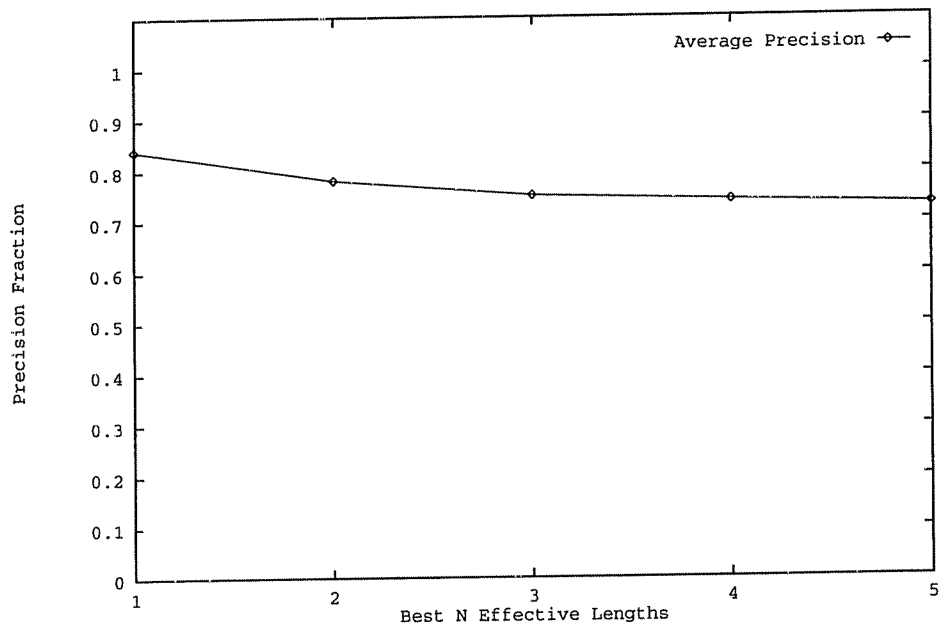

Figure 6.3: Average Precision Fraction (Experiment 1- Part 1)

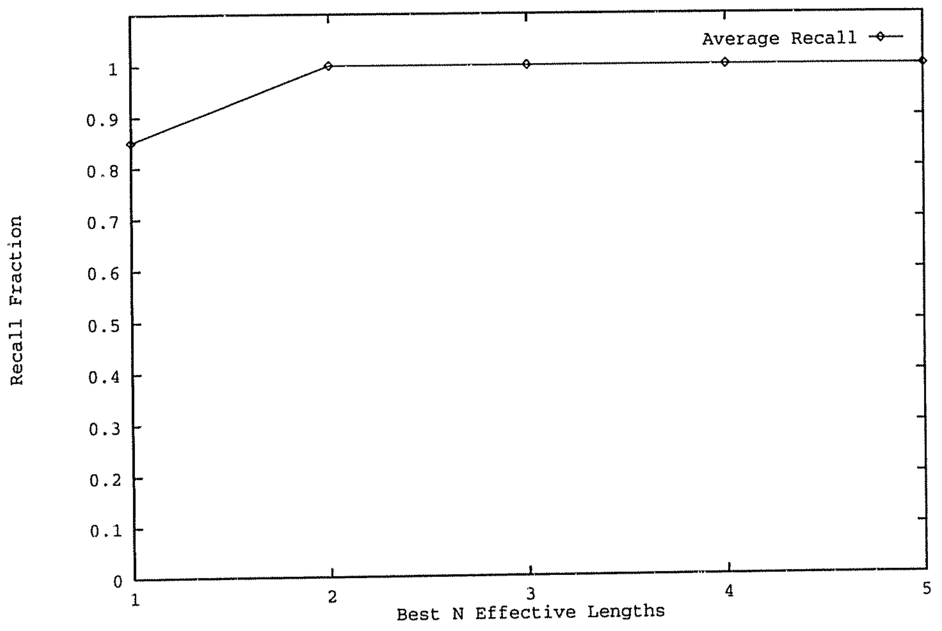

Figure 6.4: Average Recall Fraction (Experiment 1- Part 2)

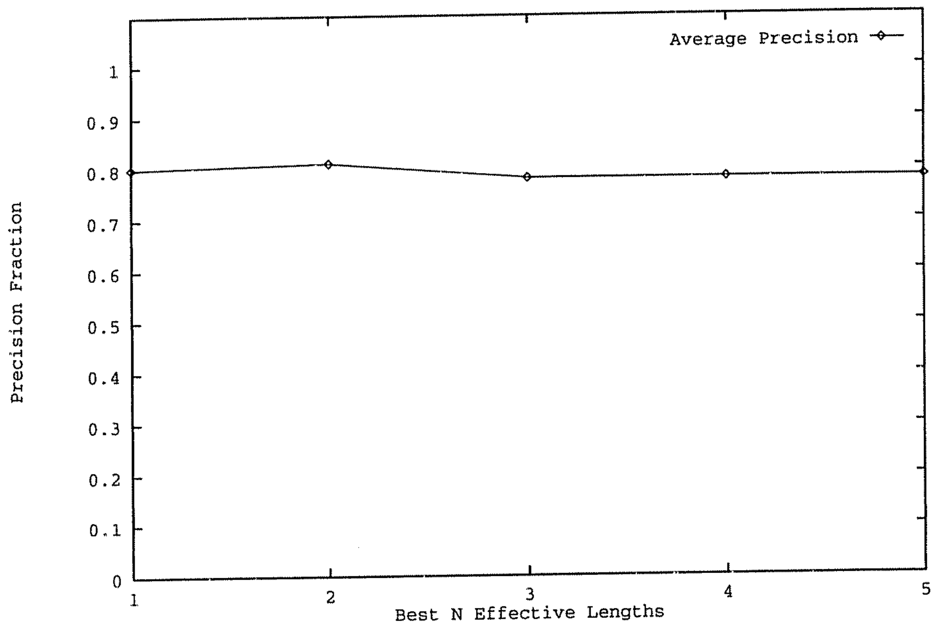

Figure 6.5: Average Precision Fraction (Experiment 1- Part 2)

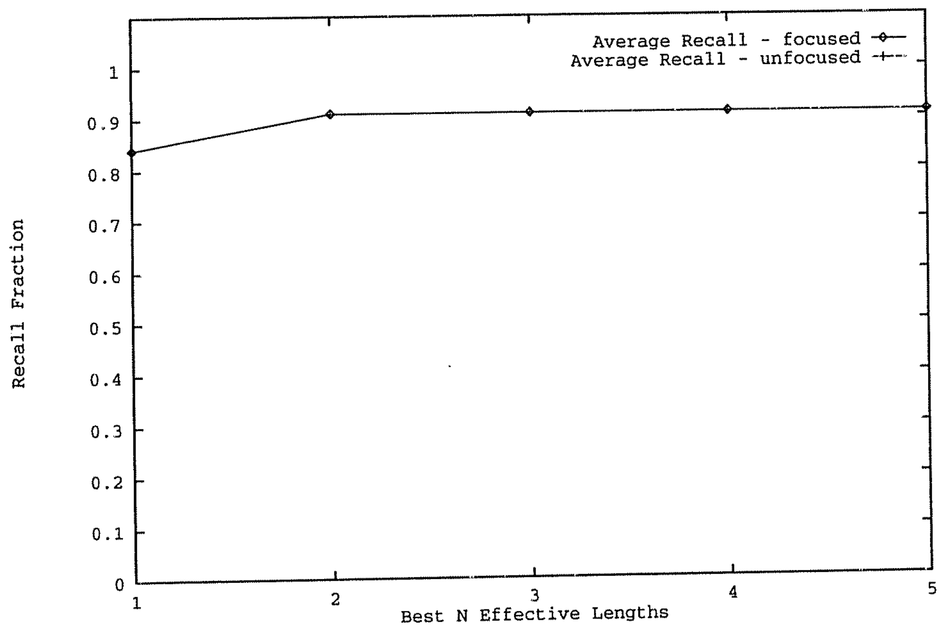

Figure 6.6: Average Recall Fraction (Experiment 1- Part 3)

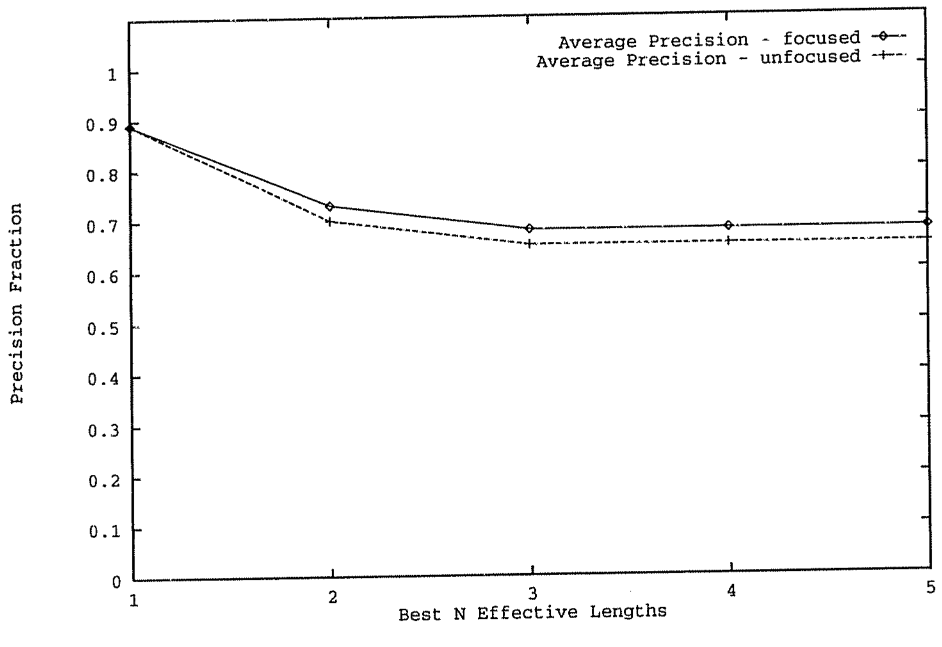

Figure 6.7: Average Precision Fraction (Experiment 1- Part 3)

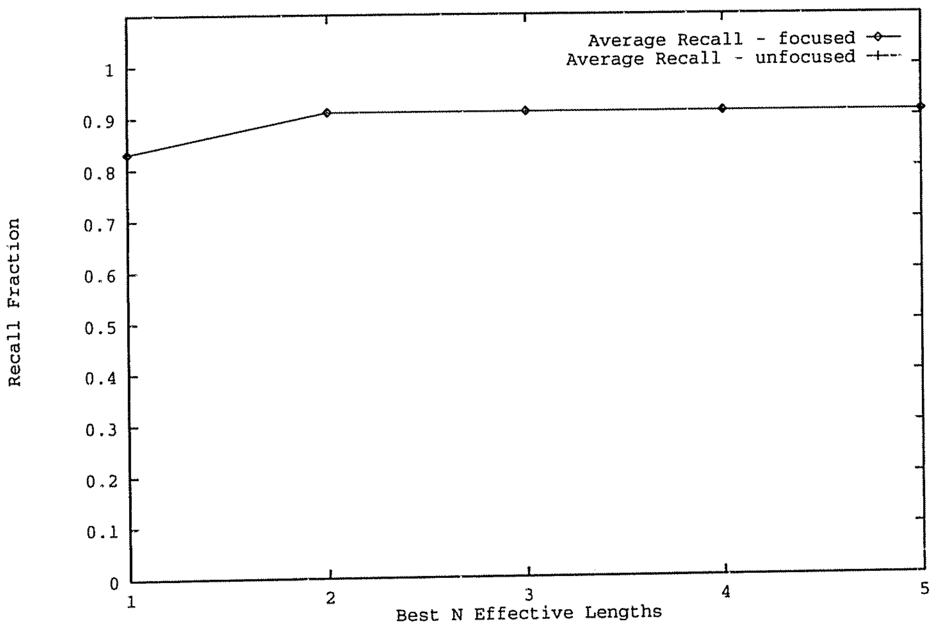

Figure 6.8: Average Recall Fraction (Experiment 1- Part 4)

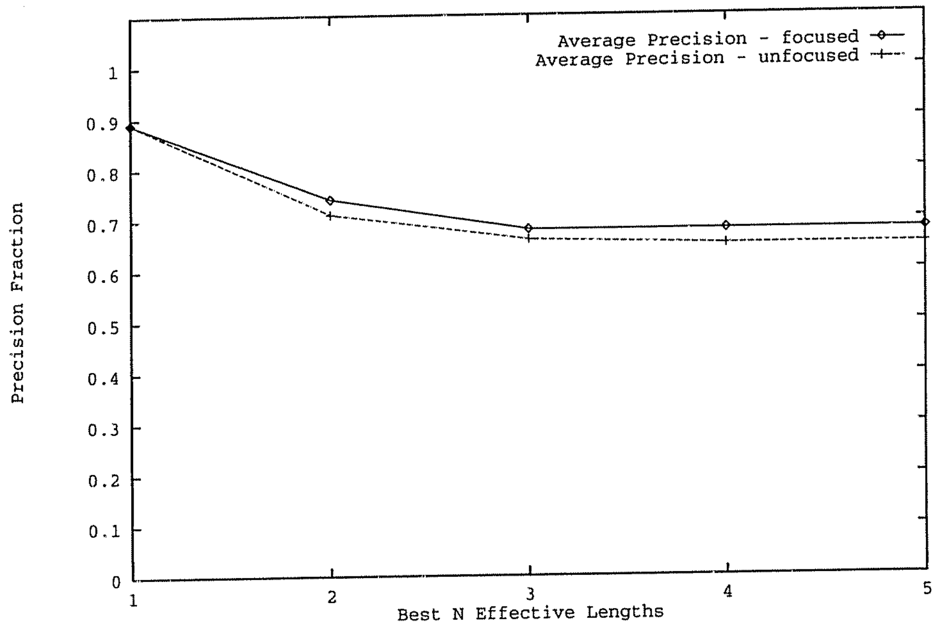

Figure 6.9: Average Precision Fraction (Experiment 1- Part 4)

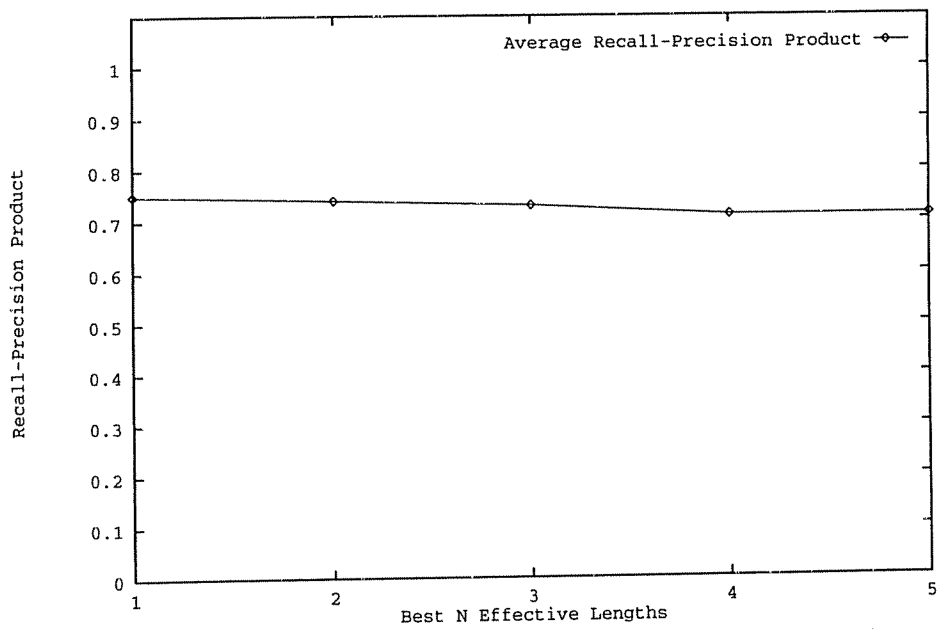

Figure 6.10: Average Recall-Precision Product (Experiment 1- Part 1)

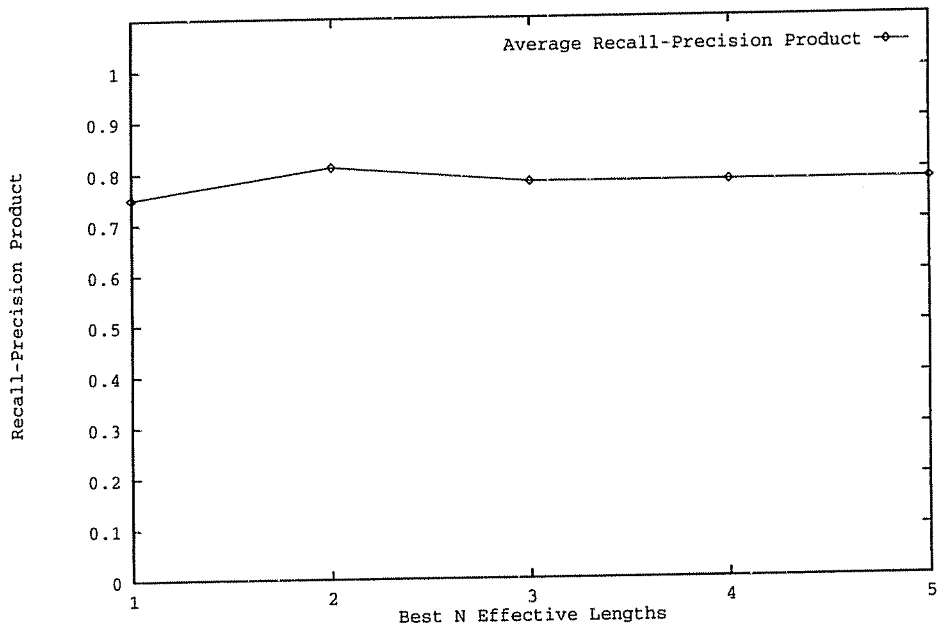

Figure 6.11: Average Recall-Precision Product (Experiment 1- Part 2)

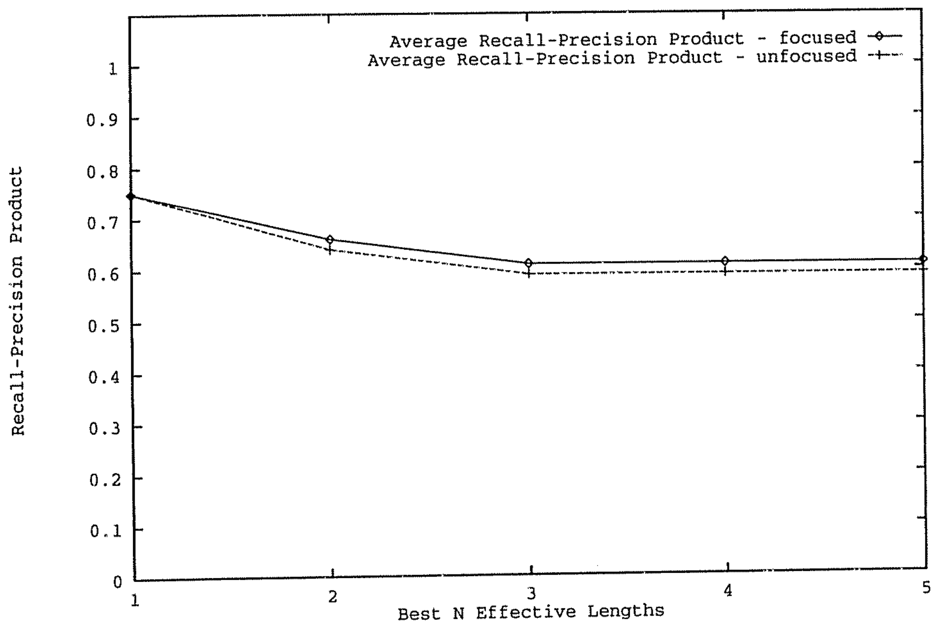

Figure 6.12: Average Recall-Precision Product (Experiment 1- Part 3)

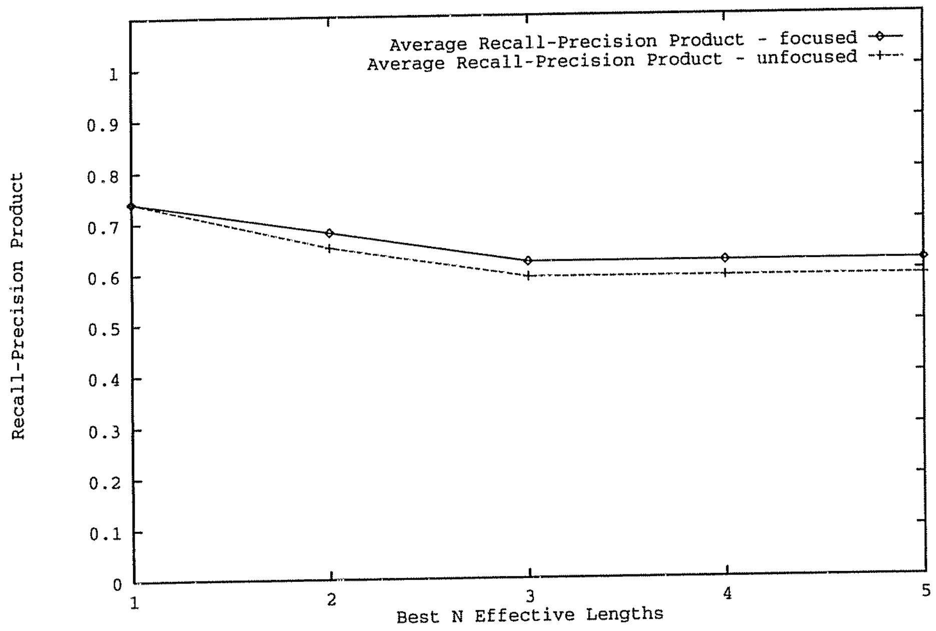

Figure 6.13: Average Recall-Precision Product (Experiment 1- Part 4)

Part 2 was meant to present the user with the alternatives retrieved by the system along with other likely completions, to see which of the system's answers the user considered likely. Part 2 was also meant as an averaging step for part 1. The results are shown in Figures 6.4 and 6.5. Here, recall does hit 100 % very quickly, as in general, the majority consensus for alternatives results in a smaller subset of answers being considered likely. The graph of the precision fraction in Figure 6.5 shows a slight increase in precision from 1 to 2. This apparent anomaly was due to the fact that the relevant answer for one of the queries did not have the best effective length according to the effective length heuristic of our implementation. We believe that this glitch in the behavior of precision is insignificant. In general, precision does decrease with the number of best effective lengths.

Parts 3 and 4 were intended to test the effectiveness of the system on a large set of random vague path expressions. Some users' vague path expressions contained additional information, such as the specific kind of relationship desired between a pair of entities. We tested such vague path expressions, in which a user provided additional information, in two different ways. We ran our system both with and without this additional information. The results shown are labeled *focused* and *unfocused* respectively in the graphs of Figures 6.6, 6.7, 6.8, and 6.9. For example, a statement such as

student names associated with this university

could be translated into either of the following vague path expressions

- *university* ~ *student* . *name* (focused specification)
- *university* ~ *student* ~ *name* (unfocused specification)

As can be seen from the graphs, focusing a vague path expression does improve the precision, but not too significantly. Recall is unaffected. The reason that the increase in precision is not greater is because we treat each search target pair in a given intermediate point specification as a separate problem, and hence even an unfocused specification retrieves only a few additional answers as compared to the focused one. [3]

In parts 3 and 4, the maximum recall obtained is slightly lower than in parts 1 and 2. It may also be noticed that precision was lower than in parts 1 and 2.

Figures 6.10, 6.11, 6.12, and 6.13, show the behaviour of the average Recall-Precision Product for the four parts of Experiment 1. As expected, this quantity stays approximately constant. The reason for the slight decrease in the value of the Recall-Precision Product between the number of effective lengths from 1 to 2 in parts 3 and 4 can be attributed to the corresponding sharp decrease in precision in this interval.

In general, the system performed well in terms of providing intuitive path completions to vague path expressions provided by users. As has been indicated by work in adaptive interfaces [ShMa 93], users are not alike. A system that is to work optimally must adapt to a user using some form of learning and feedback. Our current implementation and theory does not provide for this. Thus, although our path collapse function and selection functions are good, they are not perfect in quite a few cases. We address the possibility of integrating learning into this framework, in Chapter 7.

[3] Experiments we conducted using the restricted regular expression approach to intermediate point specifications, indicate that focusing a vague path expression for this approach often dramatically improves the precision.

6.4 Experiment 2 - Shorthand Mechanism for Formulating Queries

6.4.1 Experimental Methodology

This experiment was intended to test how effective a shorthand query formulation mechanism is provided by our system for non-naive database users. The schema we used for this experiment was the input part of the CUPID [NoCa 83] schema, which represents the structure of the various entities used in soil science experiments. The CUPID schema was designed by Larry Murdock of the Soil Sciences department using the MOOSE OODB data model. The schema has 219 classes and 364 relationships. It has all the kinds of relationships allowed by our model as well as derived relationships. Hence, it is a sufficiently large schema to make the formulation of ad-hoc queries a non-trivial task, even for the schema designer.

We asked Larry Murdock to come up with ten ad-hoc vague path expressions with the schema diagram in front of him. For each of these vague path expressions, he had to have the corresponding path expression(s) he wanted to see returned by the system ready. In some cases, the system came up with a path expression that he hadn't thought of, but which he felt was good enough to be considered an intuitive completion for his vague path expression. For this experiment a path expression was considered relevant if it was one Larry Murdock had in mind or was one retrieved by the system that he felt was an intuitive completion. We chose this definition because the sheer size and complexity of the CUPID schema graph resulted in a few intuitive completions being overlooked initially.

Further, having the schema designer with us enabled us to use domain specific knowledge as well. For example, one rule that had to be obeyed by all path expressions returned by the system was that they *should not* pass through two specific classes in the schema. [4] Such a declarative statement did help the precision fraction tremendously. [5]

6.4.2 Results

The results of the experiments are shown in Figures 6.14, 6.15, and 6.16. The system performed remarkably well in terms of being able to select the 'correct' path expression(s) from the set of path expressions matching a given vague path expression (to get an idea of the discernibility of the selection function, an average of over 500 path expressions are consistent with each vague path expression).

Figures 6.15 and 6.16 also indicate that even a small amount of domain knowldege can dramatically improve the precision of the system, especially for database schemas like the CUPID schema which have certain 'universal indexing' classes serving as indexes for most of the classes in the database. Without additional knowledge of the specific database domain such classes are indistinguishable from other indexing classes in the schema. It should however be noted that our effective length heuristic performed remarkably well even without the domain knowledge (perfect precision for the best effective length values). Even with domain knowledge the precision fell from a perfect value (100 %) for answers with the best effective length, to a slightly lower value as the effective length was increased.

The recall was not affected by the effective length increasing.

Due to the fact that the recall did not vary with effective length (indicating that our effective length heuristic was very effective), the recall-precision product graph follows the trend of the precision graph.

[4]These two reference classes were created as a sort of index for the plethora of different experiments carried out.

[5]Although our procedure does have a stage for domain specific discriminators, there is no corresponding stage implemented as yet. We performed the domain specific discrimination amongst path expressions returned, by hand.

The length of the average path expression returned as an answer from the CUPID schema was about 15 relationships long. Thus specifying a query is not a trivial task even for the schema designer. As the experiments show, our system does provide a convenient and powerful shorthand mechanism to specify complete answers.

6.4.3 Efficiency of Traversal

The CUPID schema graph is a realistic size schema for most databases. We ran a final experiment on this schema graph to measure the relative efficiency of traversal of our two approaches to intermediate point specifications. The traversal algorithm formulation presented in Chapter 5 is recursive irrespective of whether we use the restricted regular expression approach or the ordered sequence of end point specification approach. Each recursive call corresponds to an exploration of a class node in the schema graph. Hence we use the number of recursive calls as a measure of the efficiency of traversal of the schema graph for a given vague path expression. We tested the efficiency of the two approaches on the same set of ten vague path expressions used in Experiment 2. While the recall and precision figures turned out to be identical to those above, there was a marked difference in terms of efficiency of traversal of the schema graph.

Figure 6.17 plots the number of recursive calls taken on each of the ten vague path expressions by the two algorithms. While both algorithms followed the same general trend the restricted regular expression traversal was significantly more expensive in terms of number of class nodes visited. Furthermore, the restricted regular expression cannot generate any path expressions with cycles, and in general, does not perform as well as our current method of dealing with intermediate point specifications.

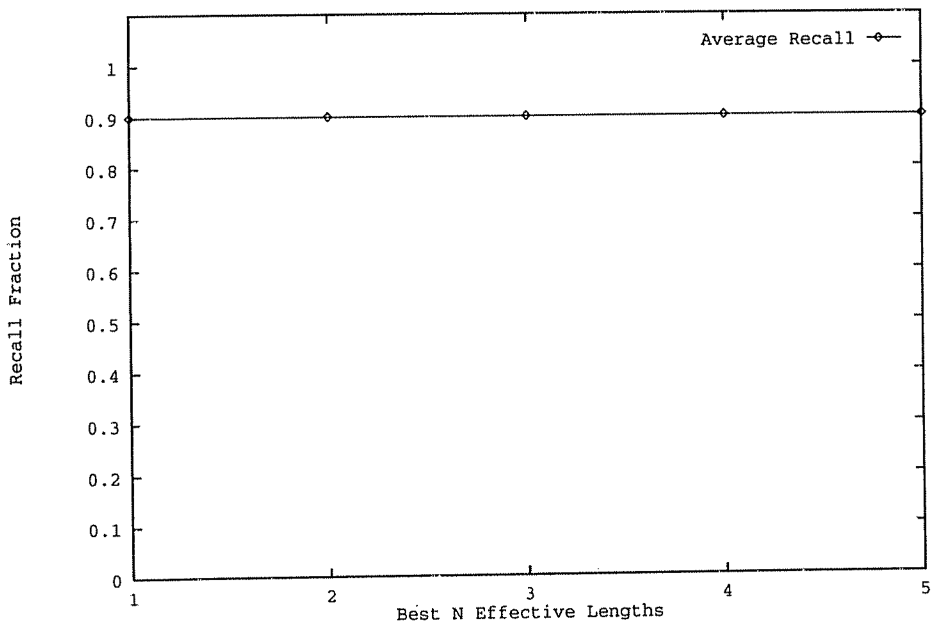

Figure 6.14: Average Recall Fraction (Experiment 2)

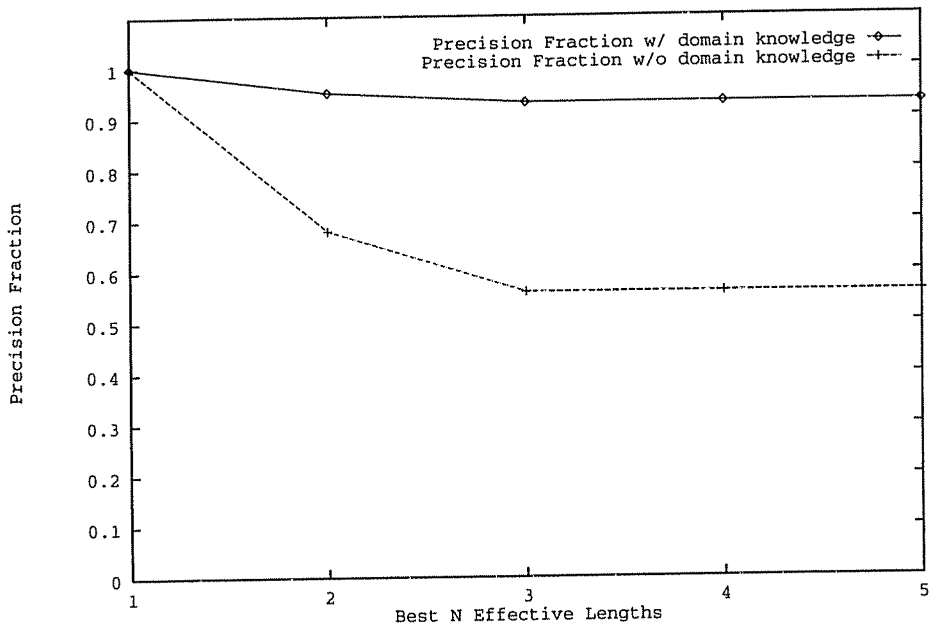

Figure 6.15: Average Precision Fraction (Experiment 2)

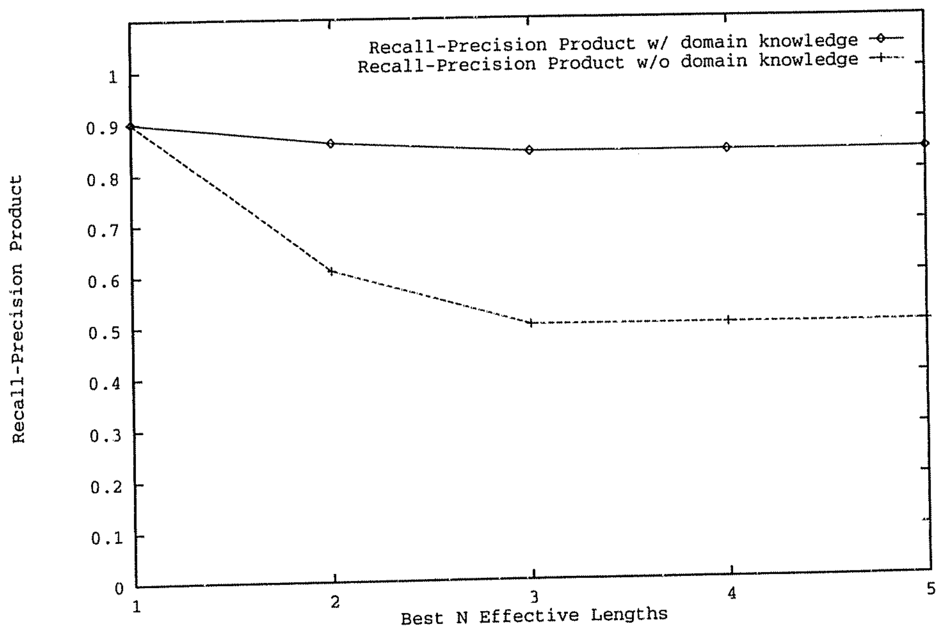

Figure 6.16: Average Recall-Precision Product (Experiment 2)

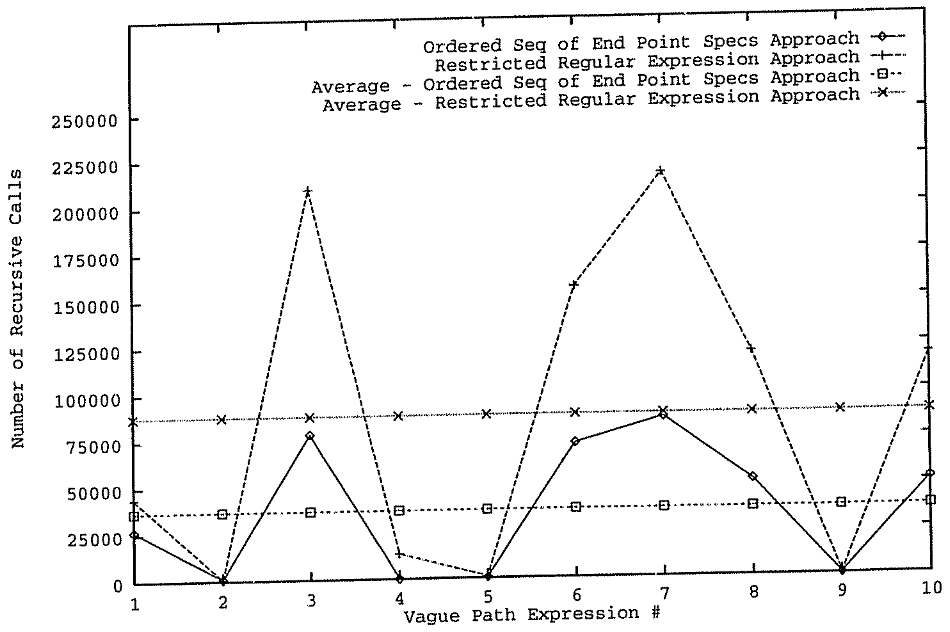

Figure 6.17: Number of Recursive Calls Per Query (Experiment 2)

6.5 Summary of Results

The experiments in the preceding sections of this chapter, while by no means exhaustive, indicate that our approach to disambiguating vaguely specified queries to a database is promising. The results of Experiment 1 indicate that a purely English level description of the data in the database suffices quite well for a naive user (in this case a user who does not know the database schema) to successfully query the database without having to know the explicit schema layout. While the schema for Experiment 1 is quite small, we believe that this approach will scale reasonably well.

The results of Experiment 2 indicate that our theory can also serve as a valuable query formulation shorthand mechanism even for database designers and administrators by relieving them of the tedium of formulating large queries. Once again, the results here are encouraging but more experiments with different 'real life' schemas are required.

Domain specific knowledge, where available, can improve the precision of the retrieved answers.

The approach we have chosen to dealing with intermediate point specifications not only generates better answers (meaningful cyclic path expressions), but is also more efficient in terms of the number of nodes explored in the schema graph. The efficiency of traversal of the schema graph is crucial as we expect to deal with large database schemas and require as close to real time response as possible from the completion code module.

Chapter 7

DISCUSSION

In this chapter we discuss certain techniques that could possibly be used to enhance the theory presented in earlier chapters, so as to achieve better performance of the system in terms of precision and recall.

7.1 Estimation of Answer Sizes

Each path expression retrieved corresponds to a set of objects of a particular class, or attributes of objects of a particular class, in the database. For example :

- *course* <= *courses* . *student* @> *person* . *name*, and
- *course* <= *courses* . *teacher* <@ *professor* <= *faculty* <$ *department* . *student* @> *person* . *name*

both denote the *name* attribute of objects belonging to the class **student** in the schema of Figure 2.1. However the first path expression represents *the set of names of students taking a particular course*, while the latter denotes *the set of names of students in the department of the faculty member teaching that particular course.* In most cases the number of objects in the database satisfying the former path expression is going to be considerably smaller than the number of objects satisfying the latter one. In such a case, we say that the former path expression *subsumes* the latter one, as the latter path expression has a much larger possible set of answers than the former one.

The motivation behind the above is based on various psychological studies that indicate that when confronted with two answers of widely differing sizes, humans tend to prefer the more *specific* or *focused* answer of the two. This also corresponds to the case of picking the most specific rule/explanation applicable to a particular problem situation in production systems and Explanation Based Learning system.

We attempt to capture the notion of specificity of answers as follows. We assume the database provides some sort of answer size estimator for path expressions. [1] A pair of path expressions ψ_1 and ψ_2 are said to be *comparable* for answer size estimation, if

- The final relationship in both path expressions is the same.
- If the final relationship is *not* a *Has-Attribute* relationship, then the most specific subclass of the class pointed to by the final relationship in both path expressions is the same.

[1] How this answer size estimator is to be implemented is beyond the scope of the current work. It may be through domain specific rules, periodic sampling, integrity constraints like functional/inclusion dependencies, or any other convenient technique.

- If the final relationship is a *Has-Attribute* relationship, then the most specific subclass of the class that has the final relationship as its attribute in both path expressions is the same. In the example above, in both path expressions, the most specific subclass present of the class person having attribute *name* is the subclass student.

Consider two comparable path expressions ψ_1 and ψ_2 with estimated answer sizes $\mathcal{A}_\infty$ and $\mathcal{A}_\in$ with $\mathcal{A}_\infty \neq \prime$. If

$$\mathcal{A}_\in / \mathcal{A}_\infty \geq \mathsf{MAX\_ASZ\_MULTIPLE} \geq 1$$

where MAX_ASZ_MULTIPLE is a database specific constant, then the path expression ψ_1, (with the smaller answer size estimate) *subsumes* the path expression ψ_2, (with the larger answer size estimate). Subsumed path expressions are discarded from the answer set returned.

If the estimator estimates that the answer size of a given path expression is likely to be *zero*, then the path expression is discarded from the answer set. Our objective is to present the user with as specific an answer as possible ; however it is quite useless presenting the user with a path expression that is likely to yield no answer from the database.

Answer size estimation is a powerful domain specific discriminator, if available. It can be applied as a post-processing step to pairs of comparable path expressions surviving the initial discriminatory phases. Answer size estimation will not normally affect the recall fraction of a system. However it tends to improve the precision fraction (and hence the Recall-Precision Product) of a system, by presenting users with fewer irrelevant answers.

We ran an answer size estimation post-processing step by hand for the results in parts 1 through 3 of experiment 1. The results are shown in Figures 7.1, 7.2, 7.3, 7.4, 7.5, and 7.6. Figure 7.5 indicates that such a step can improve precision dramatically in some cases. The reason the improvement is so insignificant in parts 1 and 2 is because these parts of experiment 1 only deal with ten vague path expressions, of which only one contained pairs of comparable path expressions. Part 3 on the other hand, deals with ninety vague path expressions, offering a much greater scope for path expression pairs to be comparable. In all the cases, recall was unaffected by answer size estimation.

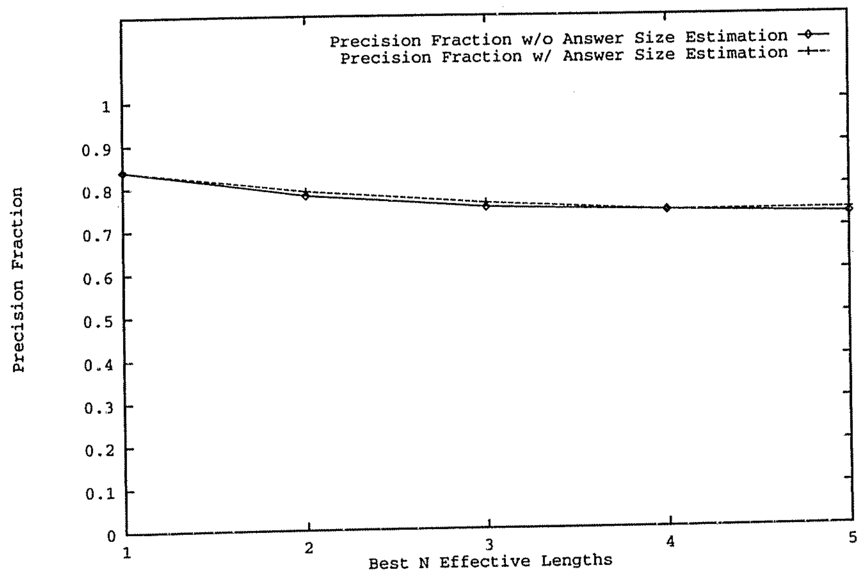

Figure 7.1: Improvement in Precision with Answer Size Estimation - Part 1

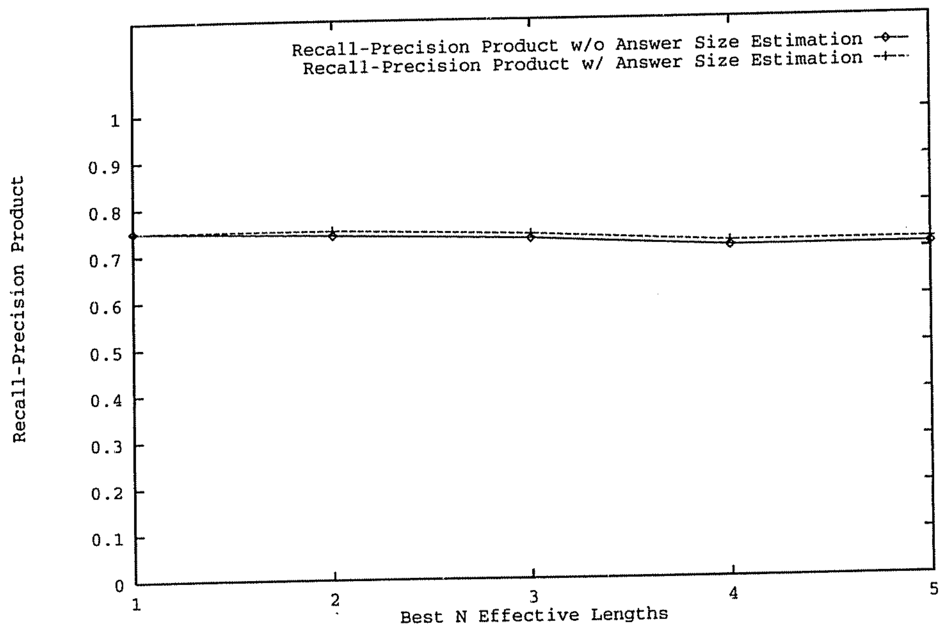

Figure 7.2: Improvement in Recall-Precision Product with Answer Size Estimation - Part 1

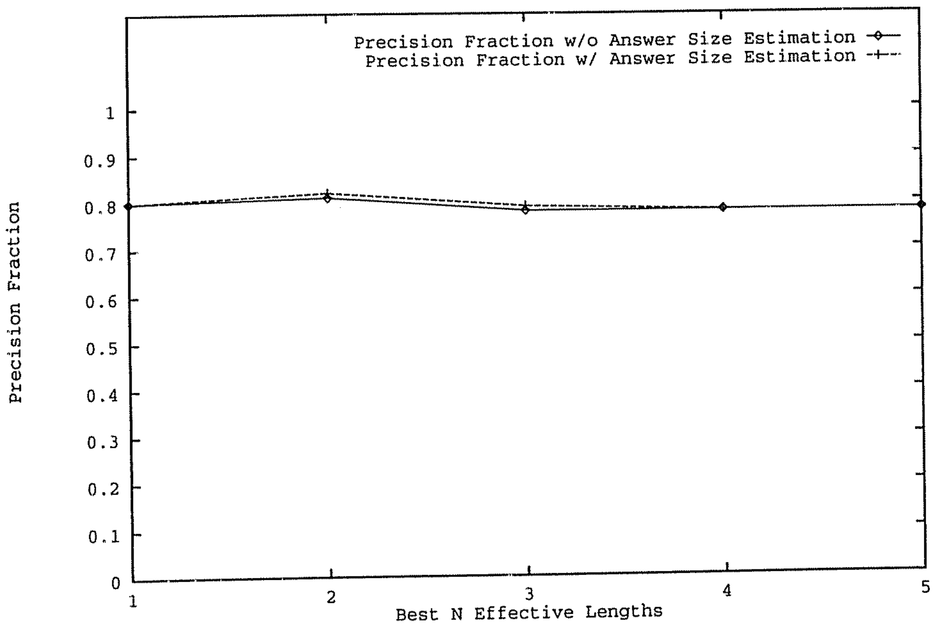

Figure 7.3: Improvement in Precision with Answer Size Estimation - Part 2

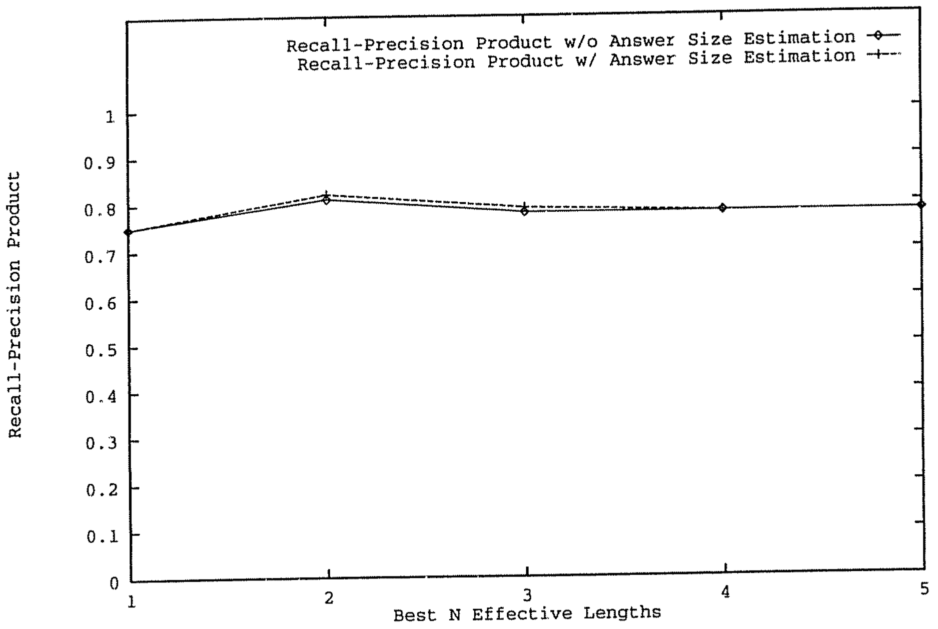

Figure 7.4: Improvement in Recall-Precision Product with Answer Size Estimation - Part 2

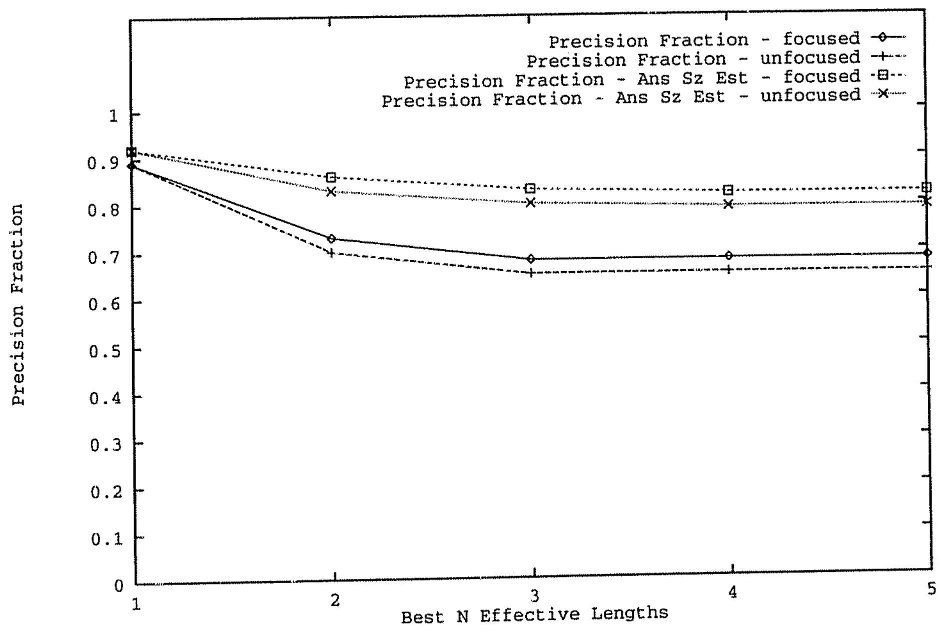

Figure 7.5: Improvement in Precision with Answer Size Estimation - Part 3

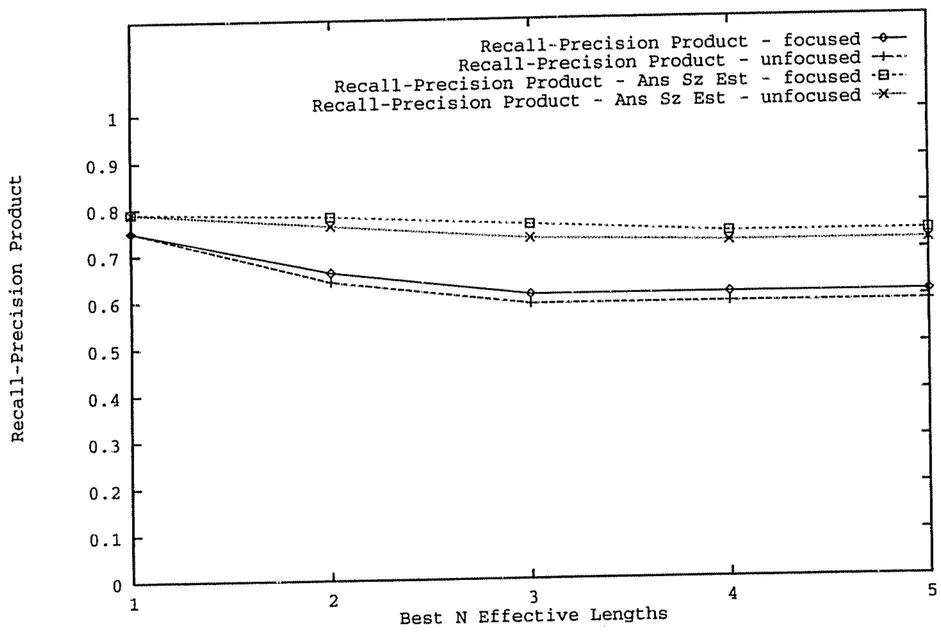

Figure 7.6: Improvement in Recall-Precision Product with Answer Size Estimation - Part 3

7.2 Alternate Collapse Functions and Orderings

Our implementation employed a particular collapse function (and associated secondary relationships), and a particular ordering of the relationships shown in Figure 4.2. We explored over ten different collapse functions and over twenty orderings of the associated relationships. We felt that this particular combination most accurately modeled the way humans think in terms of discernment on the basis of collapsed relationships. However we make absolutely no claim that this is the 'best' modeling.

The theory we have put forward does not preclude alternate collapse functions or orderings of collapsed relationships. In fact alternatives may be used very easily, especially if they satisfy the following properties :

- If the collapse function is associative, then our method of dealing with intermediate point specifications can be employed as is. If not, then additional computation is required for each path expression and search target pair.
- The selection function *must* be order independent for any arbitrary path expressions.
- If monotonicity holds, then the schema graph traversal can be optimized.
- If the distributivity equation 5.1 holds, then the traversal can be optimized via the caution sets optimization.

Another point to note while constructing alternative collapse functions and orderings, is the distinction between sharing and association relationships. Our ordering of the collapsed relationships in Figure 4.2, places the sharing relationships *Shares-SubParts-With* and *Shares-SuperParts-With* (and their corresponding *May-Be versions*), above a direct association (*Is-Associated-With*) relationship (and its *May-Be versions*). Our rationale behind this decision was that the sharing relationships are *structural* inasmuch as they arise out of the combination of two oppositely directed structural relationships (part or set). The *Is-Associated-With* relationship, while denoting a *direct association* between two entities, is non-structural, and hence 'weaker' in terms of the type of relationship it encapsulates, than a structural one.

It can be argued that a sharing relationship is also a form of association since it is undirected (owing to the fact that the structural relationships from which it arises have opposing directions in terms of dominant interpretations). If such is the case, it may be preferable to order the sharing relationships along with the direct associations, or even as a weaker form of association (*Is-Indirectly-Associated-With* for example), to yield better discernment amongst collapsed relationships. We were not able to test this hypothesis convincingly on the two test schemas we had since the schema in Figure 2.1 cannot give rise to a sharing relationship, while the distinction between the two was not conclusive enough, either way, from the path expressions generated on the larger CUPID schema of experiment 2. In general, we do not expect such comparisons between path expressions to arise very often in schemas.

7.3 Introduction of Learning

It is obvious that however we attempt to model the human thought and selection process in terms of collapsed relationship kinds and orderings, there are going to be some users with which the particular collapse and selection functions chosen are not going to work too well. It would be ideal if the system can use some form of learning from the feedback it receives after presenting a set of 'best' answers to a user, to improve its performance on the next iteration, or simply to adapt to a user's particular quirks.

Work with learning interface agents [ShMa 93] has shown that it is possible to have a population of interface agents, all working on the same task a bit differently, to increase accuracy of a system like ours. These interface agents typically have some sort of genetic representation of their task encoded in them. The better an agent's performance vis-a-vis the task at hand, the more reward it obtains. At the end of a particular period, (termed a generation), the agents receiving the highest rewards reproduce amongst themselves to form a new generation of agents while the weaker ones do not survive. The idea behind the reproduction phase is to mix the genetic material of successful agents and hopefully produce even more successful agents. The genetic algorithm [Holl 92] which is the basis of the entire technique is a proven optimal parallel search technique that adapts quickly to a changing environment.

In our case, the space to be searched is the space of possible collapsed relationship kind orderings, given a particular collapse function. An agent is a selection function based on a particular ordering. The changing environment, is either different users of the system, or a user on whom a current modeling is not very successful. Some sort of genetic representation has to be devised for the ordering of relationships. One possible solution could be to code the partial order as a C macro, (or a LISP function), and use the associated technique of genetic programming [Koza 92].

The genetic search technique however, often takes a long time to converge to a solution. We need our system to start operating at near optimal efficiency from startup. This implies that the initial orderings of the agents have to be handcrafted, instead of randomized, which is not unreasonable. Further, reproduction need only take place if the overall efficiency falls below a certain threshold, or at times the system is not in use.

Chapter 8

RELATED WORK

This chapter briefly reviews other work that has a relation to ours. We have not been able to find any work that has a direct bearing on ours. All of the work we review however, has some similarity with certain aspects of our work. We begin this review by presenting some of the current work in DBMS User-Interfaces (UIs), then present the concept of spreading activation in a semantic net, and end with a brief review of the use of object-oriented hierarchies for common-sense reasoning.

8.1 Database Interfaces

There has been a large body of work on various types of user interfaces to DBMSs, each system designed to address a specific problem. There are numerous database browsers [MDT 92], [Motr 86a] present for DBMSs. Most such browsers allow a user to browse through the schema and the data in the database, and are used when the user does not know the database schema, or the data model or query language. Browsers are also used when the user has only a vague idea of what he/she wants to retrieve from the database. Browsers suffer from the same disadvantages as GUIs [ILH 92] mentioned in Chapter 1.

It often happens that the user may be familiar with a schema but may not know the exact values stored in the database itself. Interfaces such as VAGUE [Motr 88] allow a user to specify queries based on some predefined notions of similarity or closeness to other data values. For example, VAGUE allows users to formulate queries like

select all theatres close to LosAngeles that are showing films like Psycho

where the interface selects data values from the database for the terms

- *close to LosAngeles*, and
- *films like Psycho*

based on some data metric of closeness of attribute values for an attribute domain. Such interfaces are powerful, but are also dependent to a large extent on the database domain.

Certain other interfaces such as SEAVE [Motr 86b] take the approach that every user query is based on certain presuppositions about the data values in the database. If a query fails owing to the absence of the required data values in the database, then with a little deviation from the values specified by the user, it may be possible to formulate a query that succeeds, and that the user is interested in. For example, if the query

select all female employees with age < 30 *and salary* > 40000

fails, then the interface attempts to formulate similar queries, but with data values that are likely to succeed. For the example above the interface may attempt to formulate alternate queries such as

- *select all female employees with age < 30 and salary > 35000*
- *select all female employees with age < 36 and salary > 40000*

based on its knowledge of actual data values in the database.

Interfaces such as FLEX [Motr 90] attempt to combine the approaches above, so that experienced users can directly formulate queries, while naive users may use one or more of the mechanisms above to get at the data they are seeking.

8.2 Semantic Networks and Spreading Activation

Semantic Networks are general cases of Object Oriented Systems. There has been significant work done on semantic networks in the Artificial Intelligence community. Almost all of this work has concentrated on Natural Language Understanding through semantic networks.

A semantic network is a network of inter-related concepts (which act as the nodes in the network). Semantic network programs attempt to draw inferences between various concepts by doing a search out from the concept nodes, until the searches intersect at a given node or nodes. For example, the concepts *large*, *gray*, and *animal* will intersect at the concept node *elephant*. Different semantic network systems perform differing degrees of inference, and support differing degrees of natural-language like semantics. However, for the most part, they all perform a search for an intersection set of concept nodes, from a given start set of concept nodes.

There have been various algorithms published in the literature to make this search as efficient as possible. Most of the efficient algorithms for semantic networks use a technique termed *marker passing* to spread *activation* out from the original concept nodes [CoLo 75, Hend 87, Char 83, Char 86, Hirs 92]. Perhaps the most famous is the NETL [Fahl 79] algorithm that performs parallel local processing at each node (or set of nodes) to speed the search.

The work in semantic networks is similar to ours inasmuch as we too perform a search over a network of relationships and concepts (classes). However, we have only the start concept to begin with and various concepts (intermediate points) to string along the way. Performing a search on the schema using a marker passing algorithm similar to [Fahl 79, Char 83, CoLo 75, Hend 87, Char 86], is not very straightforward for intermediate point specifications especially for the restricted regular expression approach. Even for an end point specification there may be multiple classes in a schema pointed to by the last relationship in the vague path expression. For example there are three relationships labeled *name* in the schema of Figure 2.1 itself. A crucial reason why we do not use a spreading activation scheme is because most such schemes are inherently breadth first. Our algorithm is depth first. As soon as we find a path expression consistent with a given vague path expression, we can use the monotonicity property of Chapter 4 to bound the search.

8.3 Object Hierarchies and Common-Sense Reasoning

Another significant body of work in Artificial Intelligence is in the field of object hierarchies and common-sense reasoning. Various attempts have been published to formalize the mathematics of object hierarchies, especially those that contain defeasible links, and hence non-monotonic inferences [Tour 84, Brac 83, Tour 90].

Most of the work in this area has concentrated on non-monotonic inferences from object hierarchies. Some of the work has also been concerned with approximate reasoning [ToCo 89, Tabo 88, BoCo 75].

None of the work above is directly related to our task at hand, as our model precludes the existence of defeasible links in a database schema.

A final piece of research that bears some similarity to our notion of a path collapse function is the work in [CoLo 88] on constructing plausible inferences from pairs of adjacent relationships. However the research in [CoLo 88] uses a larger set of relationships and is more oriented to various actions in English than the relationships in our data model.

Chapter 9

CONCLUSIONS

With the increasing proliferation of DBMSs amongst non-computer experts there is a growing need for user friendly interfaces. With a growing number of queries to DBMSs being ad-hoc we argue that there is a compelling requirement for a query formulation mechanism that takes as input a given vaguely specified query to a database, and presents a user with a set of completely specified query for that particular database by somehow 'filling in the gaps' in the vague path expression. This mechanism should be database independent, as well as data model independent, to allow easy translation and portability to any DBMS and database. This precludes the usage of database specific rules for the task. The mechanism used also has to be fast (close to real time response), to be effective. If the user has to wait for more than a few minutes to receive likely completions for his/her vague path expression, then its utility is severly limited.

We have introduced a theory of generating plausible completions of vague path expressions. Our OO data model is general enough to easily map onto most existing data models. The completion generation procedure is deliberately split into various phases, with the database dependent phases used as a last resort or post-processing step. As Chapters 6 and 7 demonstrate, database specific knowledge, where available, can improve precision, (and occasionally recall), in most cases. However our theory *does not* require this knowledge as an essential component of the completion generation procedure.

A fundamental contribution of this thesis is the casting of the entire completion generation problem as the familiar problem of optimal path computation over a labeled directed graph. As none of the path computation algorithms in the literature are suited to the peculiarities of our problem, we designed the *caution sets optimization* for efficient traversal of the schema graph. If the path collapse function and the selection function satisfy the properties in Chapter 4, then the traversal algorithm presented guarantees an efficient traversal.

The preliminary experiments of Chapter 6 indicate that our knowledge lean approach of exploiting the structure of the relationships in a path in the schema graph is very promising. A lack of alternative large schemas in the MOOSE data model prevented further experiments, as also time constraints. However additional experiments on different 'real life' schemas like the CUPID schema need to be done to confirm the validity of these initial results.

The introduction of a feedback and learning step to the procedure of Chapter 5 will assure that the system will not suffer large differences in performance with different users. We feel that the learning agent approach combined with genetic evolution is the most promising, owing to its ability to quickly adapt to a changing environment as well as retain properties from previous successful agents. The evolution of new generations can take place when the system is not in use thus assuring a fast response time during a user session.

Another interesting direction for this work would be to provide users the ability to specify the properties of relationships used in a particular data model declaratively, (in terms of certain *deep structure* properties of these relationships such as underlying hierarchical or spatial interpretations), and have a computer algorithm to automatically generate an adjacent relationship collapse function and the associated secondary relationships. The postulating of an associated deep structure for every relationship kind and the usage of this deep structure to generate plausible inferences from adjacent relationship pairs, is akin to the research reported in [CoLo 88].

The interface for such a system can also be made more user friendly with a few easily implementable modules. Addition of a synonym lexicon for class and relationship names does not constrain the user to know the particular class names/relationship names in the schema. A module to translate the path expressions generated into a more Natural Language like form when presenting the options to a user, would also improve the system's ease of use for naive users. Satisfactory versions of both the above modules are relatively easy to implement, and we believe necessary, before a system based on this theory can be used successfully by naive users.

This thesis has presented an effective procedure for generating completions consistent with a given vague path expression. The procedure is domain independent and hence portable, while the implementation presented is fast, efficient, and modular, and can easily be integrated into the overall context of a DBMS. The experiments presented here indicate that this approach bears promise.

Appendix A

Vague Queries For Experiment 1 (parts 1 and 2)

The following twelve queries were the vague statements used in parts 1 and 2 of Experiment 1. The wording of all the statements was deliberately kept the same so as not to influence subjects. Subjects were urged not to be biased by verbal cues to the extent possible and to think of the entities in the statements in terms of their most intuitive associations. Questions 5 and 7 were considered meaningless by a majority of the subjects.

1. "The name or names of this teaching assistant"
2. "The name or names of this course"
3. "The course or courses of this teaching assistant"
4. "The professor or professors of this student"
5. "The credit or credits of this department"
6. "The name or names of this faculty"
7. "The office number or numbers of this student"
8. "The course or courses of this grad"
9. "The social security number or numbers of this course"
10. "The ID number or numbers of this university"
11. "The teacher or teachers of a course or courses of this student"
12. "The teacher or teachers of an undergrad or undergrads of this department"

Bibliography

[ADJ 90] R. Agrawal, S. Dar, and H. Jagadish, *Direct Transitive Closure Algorithms: Design and Performance Evaluation*, ACM TODS, Vol 15 (3), pp 427-458, September 1990.

[ABD+ 89] M. Atkinson, F. Banchilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, *The Object-Oriented Database System Manifesto*, in Proc of the International Conference on Deductive and Object-Oriented Databases, pp 223-240, 1989.

[Bate 1979] M. Bates, *Information Search Tactics*, J. of the American Society for Information Science, pp 205-213, July 1979.

[BoCo 75] D. Bobrow, and A. Collins (eds), *Representation and Understanding*, Academic Press, New York, NY, 1975.

[Brac 83] R. Brachman, *What IS-A is and isnt: An analysis of taxonomic links in semantic networks*, IEEE Computer Magazine, Vol 6 (10), pp 30-36, 1983.

[Carr 79] B. Carre, *Graphs and Networks*, Clarendon Press, Oxford, England, 1979.

[CHW 88] R. Chaffin, D. Herrmann, and M. Winston, *An Empirical Taxonomy of Part-Whole Relations : Effects of Part-Whole Relation Type on Relation Identification*, Language and Cognitive Processes, Vol 3 (10), pp 17-48, July 1988.

[ChAr 90] R. Chaffin, and A. Glass, *A Comparison of Hyponym and Synonym Decisions*, J. of Psycholinguistic Research, Vol 19 (4), pp 265-280, 1990.

[ChHe 89] R. Chaffin, and D. Herrmann, *Retrieval and Comparison Processes in Part-Whole Decisions*, J. of General Psychology, Vol 116 (4), pp 393-406, October 1989.

[ChHe 88] R. Chaffin, and D. Herrmann, *Effects of Relation Similarity on Part-Whole Decisions*, J. of General Psychology, Vol 115 (2), pp 131-139, April 1988.

[ChHe 84] R. Chaffin, and D. Herrmann, *The Similarity and Diversity of Semantic Relations*, Memory and Cognition, Vol 12, pp 134-141, 1984.

[Char 83] E. Charniak, *Passing Markers : A Theory of Contextual Influence in Language Comprehension*, Cognitive Science, Vol 7, pp 173-190, 1983.

[Char 86] E. Charniak, *A Neat Theory of Marker Passing*, in Proc 5th National Conference on Artificial Intelligence, pp 584-588, 1986.

[CoLo 88] P. Cohen, and C. Loiselle, *Beyond ISA : Structures for Plausible Inference in Semantic Networks*, in Proc 7th National Conference on Artificial Intelligence, St. Paul, Minnesota, pp 415-420, August 1988.

[CoQu 69] A. Collins, and M. Quillian, *Retrieval Time From Semantic Memory*, J. of Verbal Learning and Verbal Behaviour, Vol 8, pp 240-247, 1969.

[CoLo 75] A. Collins, and E. Loftus, *A Spreading Activation Theory of Semantic Processing*, Psychological Review, Vol 82, pp 407-428, 1975.

[Fahl 79] S. Fahlman, *NETL: A System for Representing and Using Real World Knowledge*, MIT Press, Cambridge, MA, 1979.

[GlCo 85] M. Gluck, and J. Corter, *Information, uncertainty, and the utility of categories*, in Proc of the 7th Annual Conference of the Cognitive Science Society, Lawrence Erlbaum Assoc, Irvine, CA, pp 283-287, 1985.

[Hend 87] J. Hendler, *Integrating Marker-Passing and Problem Solving : A Spreading Activation Approach to Improved Choice in Planning*, Norwood, N. J., 1987.

[Herr 87] D. Herrmann, *Representational Forms of Semantic Relations and the modeling of Relation Comprehension*, in "Knowledge Aided Information Processing", E. van der Meer, and J. Hoffmann, (eds), 1987.

[Hirs 92] G. Hirst, *Semantic Interpretation and the Resolution of Ambiguity*, Cambridge University Press, Cambridge, England, 1992.

[Holl 92] J. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge, MA, 1992.

[IRW 93] Y. Ioannidis, R. Ramakrishnan, and L. Winger, *Transitive Closure Algorithms Based on Graph Traversal*, ACM TODS, Vol 18 (3), September 1993 (to appear).

[ILH 92] Y. Ioannidis, M. Livny, and E. Haber, *Graphical User Interfaces for the Management of Scientific Experiments and Data*, ACM SIGMOD Record, March 1992.

[Koza 92] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[MeWo 89] A. Mendelzon, and P. Wood, *Finding Regular Simple Paths in Graph Databases*, in Proc 15th International Conference on Very Large Data Bases, Amsterdam, the Netherlands, pp 185-194, August 1989.

[MiFe 91] G. Miller, and C. Fellbaum, *Semantic Networks of English*, Cognition, Vol 41 (1-3), pp 197-229, December 1991.

[MIR 93a] R. Miller, Y. Ioannidis, and R. Ramakrishnan, *Understanding Schemas*, in Proc International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems, IEEE Computer Society Press, Vienna, Austria, pp 170-173, April 1993.

[MIR 93b] R. Miller, Y. Ioannidis, and R. Ramakrishnan, *The Use of Information Capacity in Schema Integration and Translation*, in Proc of the International Conference on Very Large Data Bases, Dublin, Ireland, August 1993.

[MIR 93c] R. Miller, Y. Ioannidis, and R. Ramakrishnan, *Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice*, (Submitted for Publication), 1993.

[MDT 92] A. Motro, A. D'Atri, and L. Tarantino, *ViewFinder: An Object Oriented Browser*, draft, 1992.

[Motr 86a] A. Motro, *BAROQUE: A Browser for Relational Databases*, ACM Transactions on Office Information Systems, Vol 4 (2), pp 164-181, 1986.

[Motr 86b] A. Motro, *SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems*, ACM Transactions on Office Information Systems, Vol 4 (4), pp 312-330, 1986.

[Motr 88] A. Motro, *VAGUE: A User Interface to Relational Databases*, ACM Transactions on Office Information Systems, Vol 6 (3), pp 187-214, 1988.

[Motr 90] A. Motro, *FLEX: A Tolerant and Cooperative User Interface to Databases*, IEEE Transactions on Knowledge and Data Engineering, Vol 2 (3), pp 231-246, 1990.

[NoCa 83] J. Norman, and G. Campbell, *Application of a Plant-Environment Model to Problems in Irrigation*, in "Advances in Irrigation II", D. Hillel (ed), Academic Press, New York, NY, pp 155-188, 1983.

[Ros+ 86] A. Rosenthal, et al, *Traversal Recursion: A Practical Approach to Supporting Recursive Applications*, in Proc ACM-SIGMOD Conference, Washington DC, pp 166-176, May 1986.

[Salt 89] G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison Wesley Publishing, 1989.

[ShMa 93] B. Sheth, and P. Maes, *Evolving Agents For Personalized Information Retrieval*, in Proc of the 9th IEEE Conference on Artificial Intelligence for Applications, Orlando Florida, March 1993.

[Tabo 88] P. Tabossi, *Accessing Lexical Ambiguity in Different Types of Sentential Contexts*, J. of Memory and Language, Vol 27 (3), pp 324-240, June 1988.

[ToCo 89] P. Torasso, and L. Console, *Approximate Reasoning and Prototypical Knowledge*, International Journal of Approximate Reasoning, Vol 3 (2), pp 157-177, March 1989.

[Tour 90] D. Touretzky, *Implicit Ordering of Defaults in Inheritance Systems*, in "Readings in Uncertain Reasoning", G. Shafer, and J. Pearl (eds), 1990.

[Tour 84] D. Touretzky, , *The Mathematics of Inheritance Systems*, Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, 1984.

[WiIo 93] J. Wiener, and Y. Ioannidis, *A Moose and a Fox Can Aid Scientists with Data Management Problems*, in Proc 4th DBPL Workshop, Manhattan, NY, August 1993 (to appear).

[WCH 87] M. Winston, R. Chaffin, and D. Herrmann, *A Taxonomy of Part-Whole Relations*, Cognitive Science, Vol 11, pp 417-444, 1987.