High Precision Natural Language Interfaces to Databases: a Graph Theoretic Approach

Ana-Maria Popescu
University of Washington
Computer Science and Engineering
Box 352350
amp@cs.washington.edu
Seattle, WA 98195-2350

Oren Etzioni
University of Washington
Computer Science and Engineering
Box 352350
Seattle, WA 98195-2350
etzioni@cs.washington.edu

Henry Kautz
University of Washington
Computer Science and Engineering
Box 352350
Seattle, WA 98195-2350
kautz@cs.washington.edu

Abstract

The need for Natural Language Interfaces (NLIs) to databases has become increasingly acute as more nontechnical people access information through their web browsers, PDAs, and cell phones. Yet NLIs are only usable if they map natural language questions to SQL queries *correctly*. In this paper, we introduce the PRECISE NLI, which reduces the semantic interpretation challenge in NLIs to a graph matching problem. We show that, for a broad class of natural language questions, PRECISE is guaranteed to return a correct SQL query. We compare PRECISE with Microsoft's English Query product and with Mooney's latest learning NLI on three benchmark databases. We find that both PRECISE and the learning NLI make fewer errors than the Microsoft product by a factor of four or more in each database. PRECISE handles a narrower class of questions than the learning NLI, but does so with fewer errors and without making use of training examples.

Introduction

With a few exceptions, research on natural language interfaces to databases (NLIs) has tapered off since the mid 1980's (Androutsopoulos, Ritchie, & Thanisch 1995). Yet the need for NLIs has become increasingly acute as more and more nontechnical people access a wide range of databases through their web browsers, PDAs, and cell phones. These trends suggest an opportunity to leverage the recent advances in statistical parsing (Brill 1994; Charniak 2000) to produce a new generation of NLIs.

Our novel approach is based on two additional intuitions:

1. **Easy questions:** the questions that a "typical" person would want to pose to a movie database such as *moviefone* (*e.g.*, "where is 'the Lord of the Rings' showing?") or a restaurant database such as Zagats.com (*e.g.*, "what French restaurants are open on Sundays?") are relatively "easy" to map to the appropriate SQL queries.

2. **High precision:** To satisfy users, NLIs can only misinterpret their questions very rarely. Imagine a mouse that responds appropriately to a "click" 90% of the time, but periodically whisks the user to an apparently random location. We posit that users would have an even worse reaction if they were told a that a particular French restaurant was open on Sunday, but it turned out to be closed. If the NLI does not understand a user, it can indicate so and attempt to engage in a clarification dialog, but to actively *misunderstand* the user, form an inappropriate SQL query, and give the user an incorrect answer, would rapidly erode the user's trust and make the NLI unusable.

Our paper sketches a theoretical framework for formalizing these intuitions, and describes the fully implemented PRECISE NLI. At the heart of our problem is the challenge of *semantic interpretation:* mapping words and phrases in the question to the corresponding database values, attributes, and relations. Our main contributions are the following:

- We articulate a surprisingly simple set of restrictions on natural language questions, which facilitates reliable and efficient semantic interpretation.
- Our key insight is that semantic interpretation can be reduced to a graph matching problem, which can solved by computing maximum flow in the graph. Thus, we exhibit a sound, polynomial-time procedure for semantic interpretation.
- We introduce the fully implemented PRECISE NLI. PRECISE treats its parser as a modular "plug in" enabling us to leverage the state of the art in parsing technology. We report on experiments in which PRECISE plugs in to both the Charniak parser (Charniak 2000) and Brill's tagger (Brill 1994).
- We experimentally compare the performance of PRECISE with that of the latest release of Microsoft's English Query for SQL Server (Microsoft 2002), and with Mooney's learning NLI, on the three benchmark data sets introduced by (Tang & Mooney 2001). This is the first large scale experimental comparison of research prototypes with Microsoft's commercial NLI.

The remainder of this paper is organized as follows. First, we make our "easy questions" intuition more precise by specifying a natural subset of English that can be efficiently and accurately interpreted as nonrecursive Datalog clauses. Next, we sketch the formal results on the soundness and computational complexity of our approach. Third, we describe each component of the PRECISE NLI and give an example of PRECISE in action. Fourth, we present empirical results that show that PRECISE indeed has high coverage and accuracy over common English questions. We conclude with a discussion of related and future work.

Theoretical Framework

Our formal definition of an easy question is based on the observation that many natural questions specify a set of attribute/value pairs and free-standing values, where the attribute is implicit. Attributes or relations are not paired with an implicit or explicit value only if they are associated with a "wh-word" (such as "what"). For example, in the question, "What French restaurants are located in downtown?", the word "French" refers to a possible value `French` of an *implicit* database attribute `cuisine`, the words "located" and "downtown" refer to the attribute `location` and its associated value `downtown` respectively, and the word "restaurant" refers to the relation `restaurant` and is associated with the wh-word "what".

The fact that attributes may be implicit allows a form of "ellipsis" in questions, and recovering this missing information is a central task in semantic interpretation. We also allow several words to refer to a database element, for example, the phrase "price of breakfast" refers to the database attribute `breakfast.price`. A word may even be repeated in a question when it is part of a reference to different elements. For example, in "What are French restaurants that serve French fries", the two occurrences of the word "French" in the question refer to the distinct database elements (`French` and `French fries`). We assume, however, that every database element in the interpretation of an easy question must be *distinct*: we do not allow questions such as "What are the states bordering the states bordering Montana?"

Definitions

We begin by defining the terminology we will use to characterize the easy subset of English.

Wh-words and syntactic markers: A *wh-word* is "who", "what", "where", *etc.*. A *syntactic marker* is one of a small number of closed-class words that indicates attachment (such as "of") or makes no semantic contribution to the interpretation of easy questions (such as "the").

Database element: A database is made up of three types of *elements*: relations, attributes, and values. Each element is distinct and unique: an attribute element is a particular column in a particular relation, and each value element is associated with a particular attribute. We also assume that for each attribute a special (unknown) value "wh" exists (used in forming queries, as described below). An attribute/value pair is *compatible* if the value element is indeed associated with the attribute element.

Token: A *token* is a set of word stems (perhaps a singleton set) that can name a database element. Many different tokens might name the same element, and conversely, a token might name several different elements. If a word's stem appears in a token we say the word *participates* in the token. If a token can name a database element we say it *matches* the element. The number of possible matches for a token is its *ambiguity*. As described below, a *complete tokenization* of a question is a set of tokens that exactly covers the question except for its syntactic markers.

Lexicon: A *lexicon* specifies the tokens in which each word participates and the elements that each token matches. Wh-words participate in *wh-tokens* that match corresponding wh-elements.

Characterization of Easy Questions

A question q is *easy* relative to a given lexicon if the following holds:

1. q has at least one complete tokenization T_q, and in every such tokenization it is the case that:
2. T_q contains at least one wh-token;
3. No two tokens in T_q can participate in the same database element;
4. T_q can be matched in a 1-to-1 fashion to a set of database elements E_q;
5. E_q can be extended to set E'_q by adding attribute elements such that E'_q can be grouped into compatible attribute/value, attribute/wh-value, and relation/wh-value pairs, with no element appearing more than once.

In essence, easy questions are natural language questions whose correct interpretation is a nonrecursive datalog clause. An easy question may be ambiguous, in which case we seek to identify the set of corresponding datalog clauses.

Formal Results

We now state two central results regarding the correctness and time complexity of the algorithms embodied by PRECISE (the algorithms are described in the next section). Correctness is based on the following definition:

Datalog interpretation: Suppose E'_q is a set of elements derived from a question q as described in the conditions (1)–(5) above. Then E'_q can be mapped to a nonrecursive Datalog clause ϕ_q. Relation elements become predicates in ϕ_q, attribute/value pairs fill corresponding arguments to the predicates, and attribute/wh-value pairs (if any) define a new head predicate for the clause which selects out those attributes. We call such a ϕ_q a *Datalog interpretation* of q.

The transformation from sets of attribute/value pairs to a query in the Datalog-equivalent subset of SQL is described in more detail below in the section on the Query Generator. This notion of a interpretation allows us to state that PRECISE is sound:

Theorem 1. *Given a question q with complete tokenization T_q, then:*
(i) if q is not an easy question, PRECISE *will reject the question and tell us so;*
(ii) Otherwise, PRECISE *will output the set of Datalog interpretations of the question. (Or equivalently:* PRECISE *will output a set of SQL queries that are in the subset of SQL equivalent to nonrecursive Datalog clauses.)*

The second central theorem concerns the complexity of the matcher's algorithm for disambiguating tokens and recovering implicit attributes:

Theorem 2. *Let q be an easy question. On input T_q, the* PRECISE *matcher runs in time polynomial in the length of the sentence and the maximum ambiguity of any token in T_s.*
Proof sketch: The graph construction algorithm sketched informally below results in a graph of whose size is bounded by the product of these quantities, and max-flow solutions can be found in time $O(v^3)$ by the Edmond-Karp algorithm (Cormen, Leiserson, & Rivest 1984).

The PRECISE System

This section begins with an example of PRECISE in action. Subsequently, we describe each of PRECISE's modules in more detail (see Figure 2 for an overview).

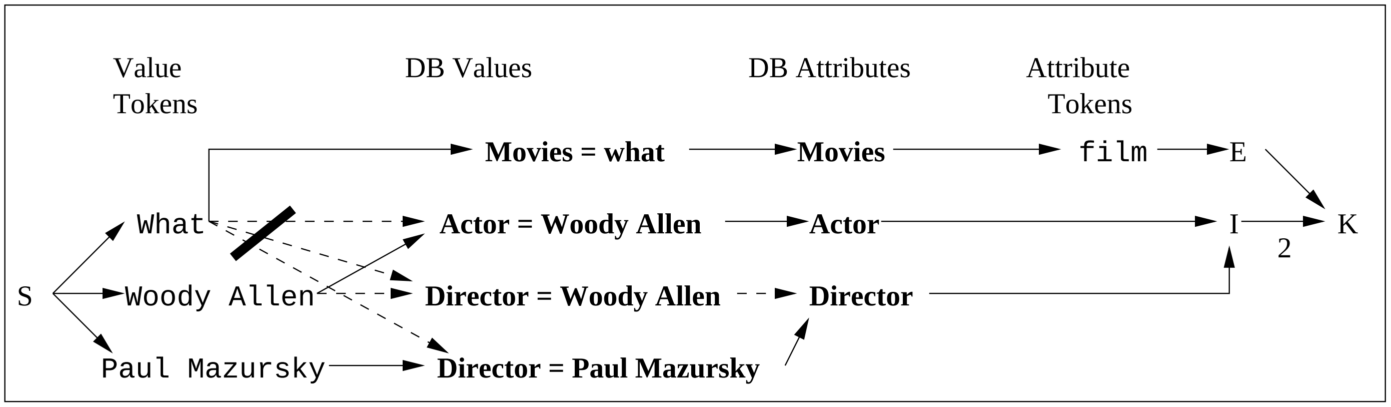

Figure 1: The graph created by PRECISE for the question *“What are the Paul Mazursky films with Woody Allen?”*

.

PRECISE in Action

To illustrate PRECISE’s behavior consider how it maps the example question “What are the Paul Mazursky films with Woody Allen?” to an SQL query (more precisely, to an SQL query in the Datalog subset of SQL). This question is chosen to illustrate the sort of ambiguity that PRECISE is able to automatically resolve. For brevity and clarity, we consider a single relation database, although PRECISE fully handles multiple relation databases as shown by our experiments on the Jobs, Restaurants and Geography databases.

The tokenizer produces a single complete tokenization of this question: `(What, Paul Mazursky, films, Woody Allen)`. Note that the tokenizer strips syntactic markers such as “the” and “with”, uses the lexicon to group names such as “Woody Allen” into multi-word tokens, and stems the plural “films” to create the token `film`.

Next, the matcher constructs the graph shown in Figure 1. To understand the meaning of nodes in the graph, it is helpful to read it column by column from left to right. The leftmost node is a source node; the Value Tokens column consists of the tokens that are identified as value tokens by the tokenizer (after checking the lexicon); the DB Values column consists of the database values (and their associated attributes) that each value token could potentially map to. For example, the token `Woody Allen` is ambiguous as it could be a value of the **Actor** attribute or the **Director** attribute. Edges are added from each value token to each possible corresponding database value. The correspondence between tokens and database elements is computed efficiently by looking up the tokens in the lexicon. Attachment information from the parser is used to eliminate some of these edges. For instance, the parser determines that “what” refers to “films” and thus to the database attribute **Movies**. As a result, three of the four outgoing edges from the token `What` are eliminated.

In the next column, the matcher connects each database value to its database attribute; then, one or more DB attributes are linked to each Attribute Token. If the attribute name does not explicitly appear in the question, then the DB attribute is linked to the node **I**, which stands for implicit attributes. In our example, the DB attribute **Movies** links to the attribute token `film`.[1] All attribute tokens link to the node **E**, which stands for explicit attributes. Finally, both **E** and **I** link to the sink node **K**.

The graph is interpreted as a flow network where the capacity on each edge is 1, unless otherwise indicated. The capacity on edge from **E** to **K** is the number of attribute tokens (1 in our example). The capacity on the edge from **I** to **K** is the number of Value Tokens minus the number of Attribute Tokens. That difference is 2 in our example. Setting the capacity to be this difference forces the maxflow algorithm to send one unit of flow from some value token to each explicit DB attribute. The matcher runs the maxflow algorithm on the graph subject to these capacity constraints and searching for an integer solution. The maximum flow through the network in this example is 3. In fact, the maximum flow in any graph constructed by the PRECISE matcher is equal to the number of value token because each such token has to participate in the match produced by the algorithm (see the definition in the theoretical section).

The solid arrows in the Figure indicate the path chosen by the maxflow algorithm. Note how the ambiguity regarding whether Woody Allen is an actor or a director in the film is automatically resolved by maximizing the flow. The algorithm “decides” that Woody Allen is the actor because this choice allows flow along two edges with capacity 1 into node **I**. Because the edge **(I,K)** has capacity 2, this choice maximizes the flow through the graph.

There are no equivalent queries, so the Query Generator finally returns:

```
SELECT DISTINCT Movies
FROM MovieDatabase
WHERE (Director = 'Paul Mazursky')
      AND (Actor = 'Woody Allen')
```

The PRECISE Architecture

Lexicon PRECISE’s lexicon is a set of ordered pairs of the form: `(token, database element)`. The set of database elements is extracted automatically from the database.[2] Natural language tokens are identified by the to-

[1]The lexicon identifies **Movies** as a synonym for “film”.

[2]In SQL Server, we rely on the GETMETADATA, GETCOLUMNLABEL, GETSTRING commands.

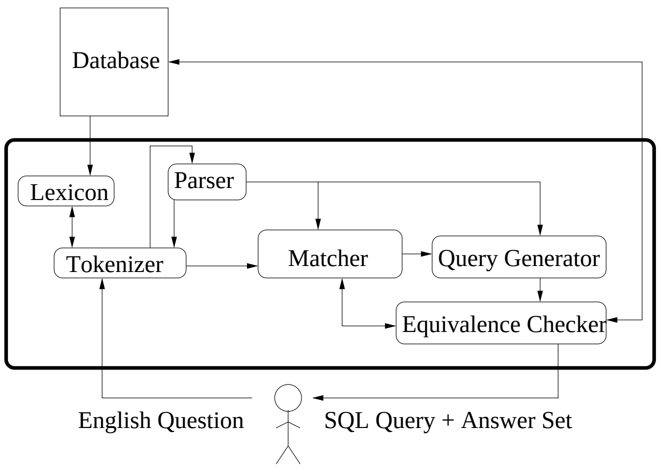

Figure 2: PRECISE Architecture

kenizer (see below). We rely on WordNet to identify synonyms for database elements. For example, since "food" is a synonym of the database attribute `cuisine`, we expand the lexicon to include both as names for the database attribute `cuisine`. The average number of pairs in the lexicons in our experiments is 5314 and the average number of synonyms per database element is 3.

Tokenizer The tokenizer maps a natural language question to the set of all possible complete tokenizations of the question. The tokenizer proceeds by stemming each word in the question, and the looking up in the lexicon the set of database elements it could potentially participate in. For each potential element, the tokenizer checks whether the other words in the element are also present in the question. For example, the word "price" matches several database attributes (`breakfast.price, lunch.price, dinner.price`). However, when the question contains the phrase "price of breakfast" the only relevant attribute is `breakfast.price` and the only token generated is `price breakfast`. Finally, the tokenizer also assigns to each token the *types* of database elements it could potentially match to (*e.g.*, value, attribute, etc.). A token may match multiple elements each with a distinct type.

Once potential tokens are identified, computing the set of complete tokenizations is equivalent to the NP-hard problem of exact set covering. In practice, however, the average number of complete tokenizations is close to one and tokenization takes less than two seconds.

Matcher The matcher embodies the key innovation in PRECISE. We reduce the problem of appropriately mapping ambiguous natural language tokens to database elements to a graph matching problem. More precisely, according to (Cormen, Leiserson, & Rivest 1984), our reduction is to a maximum-bipartite-matching problem with the side constraints that all Value Token and Attribute Token nodes must be matched, and a specified subset of the DB Value and DB Attribute nodes be matched (Anonymous 2001). As stated earlier, the matcher runs in polynomial time in the length of the natural language question and in the maximum ambiguity of question tokens.

Parser We rely on the parser to compute attachment relationships between words in the question. PRECISE uses attachment information in two ways. First, the tokenizer uses attachment to create multi-word tokens. For example, the phrase "jobs requiring five years or more of programming experience" can yield the token `require experience`. Second, the matcher reduces ambiguity by deleting edges from its graph that are inconsistent with attachment relationships detected by the parser (as shown in Figure 1).

Query Generator The query generator takes the database elements selected by the matcher and weaves them into a well-formed SQL query. In the case of single-relation queries, this process is straightforward. The SELECT portion of the query contains the database elements paired with wh-words; the WHERE portion contains a conjunction of attributes and their values, and the FROM portion contains the relevant relation name for the attributes in WHERE.

In the case of multi-relation queries, the generator adds *join conditions* to the WHERE clause, which reflect a join path that contains all the relations implicitly invoked by attributes in the query. The participating relation names are also listed in the FROM clause. If the join path is unique, the generator terminates. Otherwise, the generator generates a query for each join path and submits the queries to the equivalence checker (below). It is possible for multiple join paths to yield SQL queries that are not equivalent, leading PRECISE to flag a question as ambiguous. We omit many technical details here for brevity, but see (Anonymous 2001) for formalization and detailed examples.

Equivalence Checker The equivalence checker tests whether there are multiple distinct solutions to the maxflow problem and whether these solutions translate into distinct SQL queries. This is done by a breadth first search of the tree of subgraphs created by eliminating edges connecting tokens to elements in the matcher's graph (see (Anonymous 2001) for details). PRECISE checks for query equivalence using the algorithm in (Chekuri & Rajamaran 1998), which is polynomial time for acyclic conjunctive queries. If PRECISE finds two distinct SQL queries, it does not output an answer, since it cannot be certain which query is the right one. This conservative design decision improves precision but reduces recall. In future versions, PRECISE will ask the user to help disambiguate between the multiple possible queries.

This completes our overview of PRECISE. The implemented system goes beyond our theoretical framework by handling negated values (*e.g.*, "downtown restaurants that are not vegetarian") and several SQL operators (*e.g.*, SUM, MAXIMUM, COUNT). PRECISE relies on attachment information from the parser to identify the arguments for each operator; the query generator inserts the SQL operators and their arguments where appropriate. Finally, while modules like the tokenizer and the equivalence checker are intractable in the worst case, in practice PRECISE is quite fast taking an average of close to 5 seconds per query (before any attempt to optimize the code).

Experimental Results

To quantify the performance of NLIs, we use the precision and recall metrics adapted from the information retrieval literature by (Tang & Mooney 2001). We report on recall and precision error rates to highlight the differences between the NLIs studied. The *recall failure rate* of a NLI is the fraction of questions for which it fails to produce an SQL query;

the *precision error rate* of a NLI is the fraction of the SQL queries produced that are incorrect.

We ran our experiments on three benchmark databases in the domains of restaurants, jobs, and geography. Each database was tested on a set of several hundred English questions whose corresponding SQL query is known.[3]

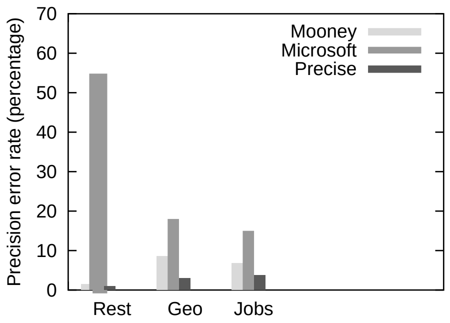

Figure 3: Precision error rate: PRECISE makes fewer errors than Mooney's latest learning NLI and Microsoft's English Query.

Our experiments aim to answer the following questions: 1. **Can PRECISE achieve the low precision error rate we believe is essential for a successful NLI?** Figure 3 shows that PRECISE achieves close to zero errors on the questions it does answer. The few errors made are largely due to incorrect attachment information computed by the parser. PRECISE shows better precision than the learning NLI and outperforms Microsoft's product by a factor of four or more on each of the databases tested.

2. **Do the assumptions underlying PRECISE result in a high recall failure rate relative to other NLIs?** PRECISE's recall failure rate is substantially lower than that of Microsoft's English Query, but substantially higher than that of the learning NLI (Figure 4). To satisfy the "high precision" requirement in the introduction, we have chosen an extreme point on the precision-recall tradeoff curve. This is due, in part, to our decision not to return an SQL query for ambiguous questions where PRECISE identifies two or more distinct candidate queries. If PRECISE is allowed to return its top three candidate queries for the user to choose from, then the recall failure rate drops sharply.

Although our recall failure rate may still seem high, it is important to keep in mind that PRECISE is a highly portable system that does not require training examples (each manually labeled with the appropriate SQL query), nor does PRECISE require extensive manual customization as is the case with Microsoft English Query.[4] The lexicon is generated automatically, and the other modules are completely generic. This is in sharp contrast to early NLIs, based on semantic grammars, which typically took months of programmer-time per database (Woods, Kaplan, & Webber 1972).

3. **How much benefit does PRECISE reap from using state of the art parsing technology such as Charniak's parser and Brill's tagger?** Figures 5 and 6 show that the parser increases substantially the set of questions that PRECISE can translate to SQL at the cost of a small increase in the number of errors in the geography database. The errors are due to the fact that attachment information extracted from the Charniak parser is not always correct. We were surprised to find that there is very little difference, for our purposes, between relying on Brill's tagger and relying on Charniak's parser — both were almost equally effective at extracting attachment information.

[3]The databases, English questions, and corresponding queries were generously supplied to us by Mooney and his group. They were used in (Tang & Mooney 2001).

[4]An undergraduate spent over 15 hours per database to customize the Microsoft product.

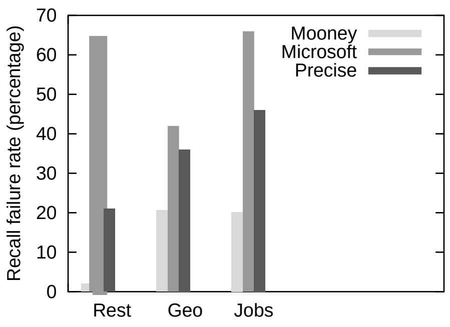

Figure 4: Recall failure rate: PRECISE handles fewer questions than Mooney's latest learning NLI, but more questions than Microsoft's English Query.

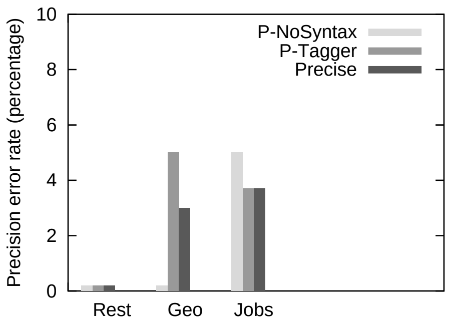

Figure 5: Precision error rate: PRECISE with Charniak's parser and PRECISE with Brill's tagger make slightly fewer errors than PRECISE with no syntactic information.

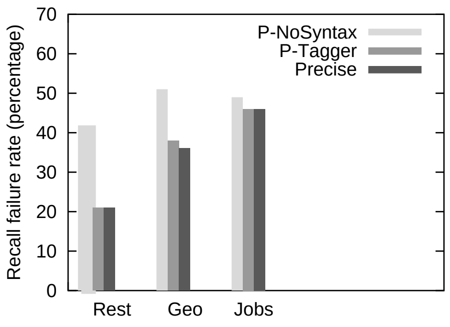

Figure 6: Recall failure rate: PRECISE with Charniak's parser and PRECISE with Brill's tagger can handle more questions than PRECISE with no syntactic information.

Discussion

We were surprised by the extent to which both research prototypes outperformed Microsoft's English Query (MEQ) in both precision and recall. MEQ suffers from an overly broad semantic model. For example, it can fail to realize that "Chinese" refers to cuisine in the context of the Restaurant database. In addition, both MEQ and the learning NLI ignore parts of the question that they fail to parse, yielding incorrect answers. In contrast, PRECISE does not return an SQL query in such cases.

When compared with Mooney's learning NLI, PRECISE has a slight edge in precision but substantially lower recall. We believe that we can improve PRECISE's recall by allowing it to engage the user in a disambiguation dialog, but this is a topic for future work. Our informal experiments show that, like many algorithms that generalize from training examples, the learning NLI is sensitive to the distribution of test cases; if test questions are very different from the questions that the NLI was trained on, the error rate can increase dramatically. Finally, PRECISE is particularly handy when training examples are not available or when manually labeling each example with the appropriate SQL query is prohibitively expensive.

Related Work

While there has been extensive work on NLIs (Androutsopoulos, Ritchie, & Thanisch 1995), most of the earlier work is very different from our own. PRECISE is *transportable* to arbitrary databases in the sense of (Grosz *et al.* 1987), and in contrast with hand crafted semantic grammars, which are tailored to an individual database (*e.g.*, (Woods, Kaplan, & Webber 1972)). Below we concentrate on the most relevant NLI systems for brevity.

Our work was inspired by that of (Tang & Mooney 2001). Mooney argued convincingly for renewed interest in NLIs and showed how strong performance can be achieved by a combination of novel learning methods. We have built directly on Mooney's experimental framework, but have chosen to explore a different point in the NLI design space. Namely, we focus on high precision NLIs at the expense of recall; ours is not a learning approach, which obviates creating training examples and labeling each example with the appropriate SQL queries; finally, we insist on being able to plug in state-of-the-art parsers to leverage their ongoing improvements, which Mooney cannot easily accommodate. It is natural to consider synergies between the two approaches as they have complimentary strengths. For example, could PRECISE be a source of training examples for Mooney's learning systems?

After developing PRECISE independently, our literature search uncovered (Chu & Meng 1999). Chu and Meng's system stores information about a given database in a graph whose nodes represent database tables and whose edges represent different types of relationships between tables. Given a natural language question, the system uses a statistical approach in order to recognize the database elements (relations/attributes/values) referred to by the question. Next, the system relies on heuristics to supplement the information extracted from the question (*e.g.*, adding join conditions.) so that a valid SQL query can be formed. Like PRECISE, the system is transportable and graph based. However, the graph representation and the associated semantic interpretation algorithm are very different. Because Chu and Meng's system does not reduce semantic interpretation to a matching problem, is not able to offer any theoretical guarantees. Finally, the system was never evaluated empirically so it is difficult to assess its effectiveness in practice.

Conclusions and Future Work

We have described a novel approach to the problem of producing a high precision NLI to databases. The key to our approach is reducing the semantic interpretation problem to a graph matching problem, which we solve in polynomial time by the maxflow algorithm. PRECISE is only effective on easy questions as formalized earlier; in future work, we plan to explore increasingly broad classes of questions both experimentally and analytically. In addition, we plan to increase PRECISE's recall through a clarification dialog with the user. Finally, we are also investigating whether PRECISE can be extended to function as an NLI for command interfaces for the Unix shell, for robots, and for other intelligent agents.

References

Androutsopoulos, I.; Ritchie, G. D.; and Thanisch, P. 1995. Natural Language Interfaces to Databases - An Introduction. In *Natural Language Engineering, vol 1, part 1*, 29–81.

Anonymous. 2001. High Precision Natural Language Interfaces to Databases: The Full Story. Technical report.

Brill, E. 1994. Some Advances in Rule-Based Part of Speech Tagging. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*.

Charniak, E. 2000. A Maximum-Entropy-Inspired Parser. In *Proceedings of NAACL-2000*.

Chekuri, C., and Rajamaran, A. 1998. Conjunctive Query Containment Revisited. In *Proceedings of the Sixth International Conference on Database Theory*.

Chu, W., and Meng, F. 1999. Database Query Formation from Natural Language using Semantic Modeling and Statistical Keyword Meaning Disambiguation. Technical Report 990003, UCLA CS Dept.

Cormen, T.; Leiserson, C.; and Rivest, R. 1984. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts.

Grosz, B.; Appelt, D.; Martin, P.; and Pereira, F. 1987. TEAM: An Experiment in the Design of Transportable Natural Language Interfaces. In *Artificial Intelligence 32*, 173–243.

Microsoft, C. 2002. http://www.microsoft.com/sql/evaluation /features/english.asp. Technical report, Microsoft, C.

Tang, L., and Mooney, R. 2001. Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing. In *Proceedings of the 12th European Conference on Machine Learning (ECML-2001), Freiburg, Germany*, 466–477.

Woods, W.; Kaplan, R.; and Webber, B. 1972. The Lunar Sciences Natural Language Information System: Final Report, Report 3438. Technical report, Bolt Beranek and Newman Inc.