Question answering system for a dedicated database

Max Martinsson
Department of Computer Science
Lund University, Sweden
max@serenity.nu

Anton Spanne
Department of Computer Science
Lund University, Sweden
anton@serenity.nu

Abstract

This report describes the implementation of a natural language question answering system for a specific problem domain, but we also consider the more general case. The main question for the project is whether it is more effective to create such a system than manually creating the database queries. Our results indicate that there is considerable overhead when constructing predefined rules for a new problem domain, which makes us draw the conclusion that effective methods of creating rules has to be developed before any gain from using this system emerges. We therefore propose three methods for constructing general constraints that span several domains.

1 Introduction

More and more information is being stored in digital form and is often organised in some form of database system, where it is supposed to be easily retrieved. The people who are using the data are, however, not themselves database experts. This means that they depend on someone else to extract the information they need from the system. An ideal solution would be if the information could be extracted by any user, without technical knowledge. The purpose of this project is therefore to construct a system which enables the extraction of data from a database using natural language.

1.1 Problem domain

The system will be built and evaluated against a specific domain, though the intention is to be able to construct a more general system later on. The domain that we use in this project will contain data concerning *root intrusions in water and sewage pipes*. The intrusion of tree roots is a big problem in urban areas. By analysing statistics about trees and known intrusions, researchers are trying to find guidelines to reduce the damage done by roots. A simple example could be a minimum distance from pipes to newly planted trees. The database for this domain contains information about more than 100,000 trees, 5,000 pipes and 25,000 intrusions. Information could only be extracted by writing complex SQL queries. This had to be done by an external expert which caused unnecessary delay and costs.

2 Lexical semantics

The ontology in our system reflects the structure of the database. We divide the world into three classes; entities, attributes and constraints. Attribute is the most primitive class and they are used to actually define the entities. That is, an entity *is* the attributes it contains. Furthermore, entities may themselves be attributes; for example, *the intrusions's pipe*. Constraints limit an attribute, filtering the entities in the data set. It may either limit the attribute to an absolute value (ex. `length < 5 meters`), or it may relate to another entity's attribute (ex. `position(a) = position(b)`). The ontology representing this attribute structure is illustrated in figure 1.

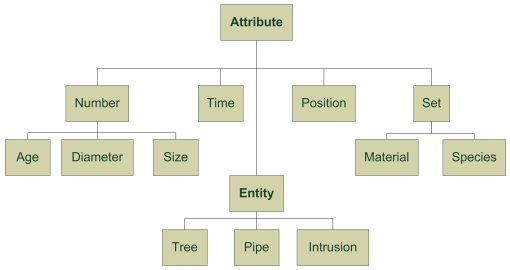

Figure 1: Attribute ontology

A simplified list of the entities and their attributes we used in the project is shown here:

```
Tree: position, size, age, species
Pipe: start position, end position,
    diameter, material, age
Intrusion: pipe, position,
    time of discovery
```

These attributes in combination with the ontology form the semantic network that the parser will use to rule out combinations of constraints and entities. The attributes are mainly meronyms, but also contain some hyponyms such as species of trees. This is mainly to make the structure more similar to that of the database.

A fourth class could be added, that would represent the overall action of the query – what to do with the data (ex. show, count, graph etc.). But this is beyond the scope of this project; in all examples, *show* will be used as the action.

Queries to the system will be transformed into a logical structure, containing action and a main entity. The entity will in turn have constraints, that could be recursive. This structure is then used to build SQL queries that retrieves the data from the database.

3 Implementation

Our implementation performs the transformation between natural language and the logical structure described above. It uses a part-of-speech tagger and a dependency parser to build a hierarchical model of the query. This model contains the dependency relations aswell as all the information from the POS tagger (word, lemma, part-of-speech). A Java program is then used to analyze this model using pre-defined rules to find the entities and their constraints.

3.1 Part-of-speech tagger

The part-of-speech tagger we used is the Granska tagger (Carlberger and Kann, 1999). It is a stochastic tagger based on a Markov model. It uses the Stockholm-Umeå-corpus (SUC) which uses the part-of-speech tagset with the same name. Other than tagging the words in the query with a part-of-speech, it also produces lemma information for the words that are being tagged.

3.2 Dependency parser

We used the MALT dependency parser (Nivre, Hall and Nilsson, 2006) to build the dependency tree, using the result from the Granska tagger. Since Granska uses the SUC tagset, and the pre-trained MALT model for Swedish uses the Talbanken tagset, we had to train a new model using the SUC tagset instead of Talbanken. This corpus was only a subset of the original Talbanken corpus, which might have been the cause of some problems with different query expression forms.

3.3 Java program

An example of the dependency tree for the question "Show trees that are at least 5 meters from any other tree" (in swedish: "Visa träd som befinner sig minst 5 meter från något annat träd") is shown here:

```
(ROOT) visa (vb)
    (OBJ) träd (nn)
        (ATT) befinner (vb)
            (SUB) som (hp)
            (OBJ) sig (pn)
            (OBJ) meter (nn)
                (DET) 5 (rg)
                    (ADV) minst (ab)
            (ADV) från (pp)
                (PR) träd (nn)
                    (DET) något (dt)
                    (DET) annat (jj)
    (ADV) . (P)
```

The dependency tree tends to vary in structure depending on different ways of expressing the same query. For example, in the expression "that are within", the "that are" can be left out. The dependency parser we used had a problem with this type of expressions where the verb is left out. As a result, we decided that the Java program should use *partial parsing* when applying the rules. This means that the rules are matched against keywords and dependency structures of the query. While this could

leed to that information from the query might get discarded, the goal is to use as much information from the query as possible.

In many other similar systems, the user input is done by speech and in such systems it is often a good strategy to have a template where the blanks are filled in iteratively during multiple queries or in dialog with the user. Often information is gathered from the querys using keywords. This means that important information could be easily missed, especially due to the speech recognition process. In these cases, it is important that the system gives understandable feedback about which "blanks" are unknown. (Mast, Kummert, Ehrlich, Fink, Kuhn, Niemann and Sagerer, 1994)

Since our system doesn't share the speech recognition problem, we can rely on that the information given to the parsers is semantically correct and well behaved. This means that it should be possible to extract most of the vital information from the queries, even without using templates and keywords. This is an important feature since we want the user to have very free hands with the queries. Asking the user to clarify what he want, would require the system to know what the user is asking for and limiting questions to a predefined set of query-templates.

The rules are written in XML format and they specify which combinations of dependencies, part-of-speech and lemma that constitutes a certain constraint. Since there are many ways of expressing the same constraint, several rules may specify the same type of constraint. An example of a rule specifying the distance between entities is:

```
<constraint class="Distance">
     <word dep="(CC|ATT)">
          <word dep="(PRD|OBJ)"
                id="unit">
               <word dep="DET"
                     pos="rg"
                     lemma="\d+"
                     id="length" />
          </word>
          <word dep="ADV"
                id="root">
               <word dep="PR" />
          </word>
    </word>
</constraint>
```

This rule will match the previously shown dependency tree. It is applied on the branch under the main entity `(OBJ) träd (vb)`. Hence `<word dep="(CC|ATT)">` will match `(ATT) befinner (vb)` and so forth. The partial parsing causes the two words "som" and "sig" that does not match anything to be discarded.

The Java program performs the following tasks when it parses the dependency tree:

1. Find action by parsing the ROOT (which should be a verb)

2. Find the main entity by looking for the noun closest to the root

3. Match rules to parts of the dependency tree, originating at the same depth as the entity. If a match is found, the corresponding constraint is created and is then responsible for parsing subtrees within itself.

   (a) If a constraint requires an attribute from the entity, that the entity does not have, then the constraint discards itself.

   (b) If a constraint doesn't find all the data it needs, it discards itself.

   (c) If a constraint is relative to another entity, the process described in (3) is repeated for that entity.

After these steps, the Java program has created the logical representation of the natural language query. The textual representation of the result is:

```
Query: {
    Action: ShowAction
    Subject: TreeSubject. Constraints: {
        Distance: {
            5 meter. Subject: TreeSubject
        }
    }
}
```

This shows the important building blocks of the logical structure. The structure is called a query, which consists of an action and a subject. The action describes what to to with data and how to collect it. The subject references to the main entity, and also the constraints that filters which entities form the database that should be used.

4 Results

The purpose of the project was to examine if the process of extracting data could be made more efficient using natural language queries. The original problem was that the user had to ask a database expert to extract the data, and in our solution the natural language interface would do this for user. But in order to be able to interpret the wide range of queries that can be expressed by a user, some expert has to write a large number of constraint rules. These rules has to be extracted from existing natural language queries, which may have to be unique to the problem at hand. This means that there is a large overhead in addressing a new problem domain. A successful and effective implementation of our system would have to consist of general rules, applicable to several domains, thus reducing this overhead. It is hard to say how difficult it would be to create such general rules. In the next section, we describe some ideas concerning such an approach.

5 Future work

General constraints could be based upon general attribute types instead of being constructed based on the attributes of the entities in the problem domain. For example, such a general constraint might concern the size of entities; largest, smallest etc. This constraint would then be applicable to all entities with the *size* attribute. With a large enough collection of such general constraints, the overhead of new problem domains could probably be reduced significantly since many attributes reoccur in several domains.

The use of attributes as a identifier for both entities and constraints makes it easier to resolve coreferences. This could be used both within a single query, and between sentences in a dialog system. Such a system could be used to construct a more complex logical structure, adding constraints iteratively. It could also be used to define new constraints and entities, that the system does not know about yet. The new entities and constraints would then be represented with the same type of logical structure as the queries. This would enable the domain experts to take over much of the work from the system expert.

It would probably be a good idea to include all hyponyms in the ontology, rather than intruducing them as attributes. An oak should for example be considered a tree, and not a species of a tree. This would make the parsing for entities more straight forward.

Finally, machine-learning methods could be used to create rules using an annotated set of queries. This could be done in at least two ways. Either the dependency tree or the linear sequence of words could be annotated. It is hard to say which of these would be more efficient without trying it.

References

Carlberger, J. and Kann, V. (1999) Implementing an Efficient Part-Of-Speech Tagger. *Software Practice and Experience*, 29(9):815–832

Mast, M., Kummert, F., Ehrlich, U., Fink, GA, Kuhn, T., Niemann, H., Sagerer, G. (1994) A speech understanding and dialog system with a homogeneouslinguistic knowledge base *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(2):179–194

Nivre, J., Hall, J. and Nilsson, J. (2006) MaltParser: A Data-Driven Parser-Generator for Dependency Parsing. In *Proceedings of the fifth international conference on Language Resources and Evaluation (LREC2006)*, May 24-26, 2006, Genoa, Italy, pp. 2216-2219

Nugues, P. (2006) *An Introduction to Language Processing with Perl and Prolog*. Springer-Verlag, Berlin, DE.