Graphical Query for Linguistic Treebanks

Steven Bird
Department of Computer Science
University of Melbourne
Victoria 3010, Australia
sb@csse.unimelb.edu.au

Haejoong Lee
Linguistic Data Consortium
University of Pennsylvania
Philadelphia PA 19104-2653, USA
haejoong@ldc.upenn.edu

Abstract

Databases of hierarchically annotated text occupy a central place in linguistic research and language technology development. We describe a new approach to tree query which we call "Query by Annotation". Users express a query by annotating a tree, and the annotation is compiled into an expression in a path language. The result trees are overlaid with the original query, permitting the user to see why they match. Since queries and results are annotated trees, users can easily refine and resubmit their queries. The approach to Query by Annotation is motivated and exemplified using databases of linguistic trees, or treebanks.

1 Introduction

Large repositories of text and speech data are routinely collected, curated, annotated, and analyzed as part of the task of developing and evaluating language technologies. These repositories contain millions of words of text, along with various annotations at the levels of phonetics, prosody, orthography, syntax, dialog, and gesture. The annotations are often hierarchical in nature, and are anchored to extents of text or speech. The hierarchical annotations can be stored as ordered trees.

Empirical investigations of hierarchically annotated linguistic data typically involve the identification or extraction of substructures, according to their position within the overall structure and their internal organisation. Consider the following kinds of access to syntactic trees: find instances of the dative construction (a verb phrase containing a verb followed by two noun phrases); extract all simplex noun phrases (noun phrases that do not contain other noun phrases) collect prepositional phrase attachment data (verb, preposition, head noun, attachment node). In each of these cases, and many others we could list, we may want to find instances of theoretical interest, or create a derived corpus, or extract features for training an automatic classifier. And in each case, we would like to be shielded from the physical storage of the corpus as a directory tree of formatted text files. All these needs are served by linguistic query languages.

Over a dozen linguistic query languages have been proposed, each with its own specialised interpreter for evaluating queries against a corpus (Rohde, 2001; König and Lezius, 2001; Kepser, 2003; Resnik and Elkiss, 2003; Mírovský, 2006; Lai and Bird, 2004). For concreteness, the work presented here will be based on the LPath language. This language has full first-order expressiveness (Lai, 2005), and can be translated into SQL for efficient evaluation (Bird et al., 2006). Although we have selected LPath, our approach is independent of the underlying tree query language and tool infrastructure. Also for concreteness, our examples will be drawn from English syntax and the Penn Treebank (Marcus et al., 1993). However, our approach is independent of the linguistic domain and data source, and can be applied to any hierarchically-annotated time-series data.

This paper presents a new approach to querying linguistic trees, namely *Query by Annotation* (QBA). In this approach, a query is expressed as an annotation of a given tree. Such a query denotes a set of trees which are similar to the given tree in precise ways. It is related to an existing approach to database query known as Query by Example (Zloof, 1977). It differs from XQBE, XQuery by Example (Braga et al., 2005), in that it only covers the selection component, and it is tailored for the specific domain of linguistic tree query. It differs from graphical interfaces to XPath that permit users to type an XPath query and see node-sets highlighted on an instance document, in

that it supports direct annotation of a query on a tree, displayed in the customary form of a parse tree.

QBA provides users with several benefits relative to direct use of a path language. First, QBA provides a high-level interface to a path query language, avoiding the need for users to learn a query syntax. Second, QBA queries are not created *ex nihilo*, but by annotating an object from the domain. Users find it easy to express queries of the form: "find me more trees that are like this one in the specified way." Third, result trees can be automatically overlaid with the original query, which means that queries and results are of the same type, namely annotated trees. A user sees why the query matched, and can edit the annotation to refine the query.

The main contributions of this work are as follows: a new approach to graphical, semi-structured query is presented, in which examples are annotated with a query graph, and queries are translated into SQL, and results are annotated with the original query; an application to linguistic databases is described, motivating and exemplifying the approach; and an implementation is reported, involving translation steps from an annotated tree to a path language, thence to SQL for evaluation.

This paper is organised as follows. In Section 2 we review key background topics including linguistic annotation, corpus curation, and query by example. In Section 3 we present our approach to query by annotation, and in Section 4 we show how annotated queries are translated into the LPath language, then in Section 5 we explain how queries are overlaid on result trees. Section 6 describes a prototype implementation, and Section 7 reports our conclusions.

2 Background

2.1 Linguistic Annotation

Linguistic databases consist of time-series data together with structured annotations. The time-series data represents an external linguistic artefact, and takes the form of a text or recording. The relationship between the primary data and its annotations is shown schematically in Figure 1.

A common data model for linguistic annotations is a labelled, ordered tree. (The nodes of the tree are ordered, by virtue of the linear ordering of the time-series data.) A natural candidate for representing trees is XML, as shown in Figure 2(a). These structures can be stored efficiently in relational form, using a span-based representation as shown in Figure 2, following (Bird and Liberman, 2001; Bird et al., 2006).

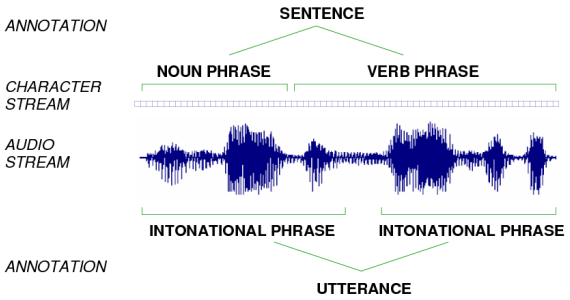


Figure 1: Linguistic Annotation: Structured Coding of Extents of Time-Series Data (e.g. Character Data, Audio Data)

2.2 Curating Treebanks

Treebanks are typically created over an extended period. Initial processing is done using a statistical parser, followed by substantial manual editing. During the course of this activity, highly-specific annotation conventions are developed; these conventions are further elaborated as new constructions are encountered. Thus, treebank creation involves significant *curation* work.

A typical part of the curation workflow is for a linguistically trained annotator to discover an incorrect parse (generated by a statistical parser), and then try to identify all other occurrences of the faulty structure and to rectify them as necessary. In the earliest days of treebank development, such curation was done by editing labelled bracketings in a text file, aided with editing macros and Perl scripts. Not surprisingly, this approach did not scale well. A common response has been to develop a tree query language.

2.3 Querying Linguistic Trees

Many tree query languages have been developed for parsed corpora, e.g. tgrep, TIGERSearch, fsq, LSE, Netgraph (Rohde, 2001; König and Lezius, 2001; Kepser, 2003; Resnik and Elkiss, 2003; Mírovský, 2006). For a survey, please see (Lai and Bird, 2004). Current graphical tree query interfaces permit users to draw partial trees from

```
<S>
  <NP lex="I"/>
  <VP>
    <V lex="saw"/>
    <NP>
      <NP>
        <Det lex="the"/>
        <Adj lex="old"/>
        <N lex="man"/>
      </NP>
      <PP>
        <Prep lex="with"/>
        <NP>
          <Det lex="a"/>
          <N lex="dog"/>
        </NP>
      </PP>
    </NP>
  </VP>
  <N lex="today"/>
</S>
```

(a) XML Representation

left	right	depth	id	pid	name	value
1	10	1	2	1	S	
1	2	2	3	2	NP	
1	2	2	3	2	@lex	I
2	9	2	4	2	VP	
2	3	3	5	4	V	
2	3	3	5	4	@lex	saw
3	9	3	6	4	NP	
3	6	4	7	6	NP	
3	4	5	8	7	Det	
3	4	5	8	7	@lex	the
		...				

(b) Relational Representation *T*

Figure 2: Linguistic Tree Representations

scratch, and the fragments are matched against trees in the treebank (Resnik and Elkiss, 2003; Mírovský, 2006). These existing approaches have shortcomings in the areas of interactiveness and expressiveness.

First, tree queries are seldom single-shot, but must be successively refined. As with web queries – where the original query is displayed in editable form along with the result – formulating suitable linguistic tree queries would be aided by a user interface that supports interactive query refinement.

Second, tree queries generally involve a variety of transitive relations (e.g. precedence) and negated expressions which cannot be expressed by drawing tree fragments.

2.4 Query by Example

Query by Example (QBE) was an early approach to user-friendly database query that shielded the user from the SQL query language (Zloof, 1977). Users search for data by partly completing a form, and this is then interpreted as an SQL query and submitted to the database engine. In short, the user initiates a search simply by providing an example of what they are seeking.

QBE is a natural way to explore and curate treebanks, given the typical workflow of progressing from an instance to a set of "similar" instances. Of course, the notion of tree similarity changes from one case to the next, and may depend on a combination of factors including structure, categories, and terminals. Thus we should not use a whole tree as the basis for identifying similar trees. Instead, we propose to *annotate* a tree in order to specify which properties must hold for any "similar" tree.

2.5 LPath

LPath is a language for querying linguistic trees which extends XPath (Clark and DeRose, 1999) with new primitive horizontal tree navigation axes, subtree scoping and edge alignment, summarised in Table 1. This language has full first-order expressiveness (Lai, 2005), and can be compiled into SQL for efficient evaluation (Bird et al., 2006). Here is the translation of the query `//A//B` into SQL:

```
select T1.* from T T0, T T1
  where T0.type='syn' and T0.name='A'
  and T1.type='syn' and T1.name='B'
  and T0.sid=T1.sid and T0.tid=T1.tid
  and T0.l<=T1.l and T0.r>=T1.r and T0.d<T1.d
```

Many useful queries turn out to be simple to express in this path language, owing to the fact that linguists identify tree nodes and relationships between nodes by reference to structurally local information.

3 Query by Annotation

Query by Annotation is an approach to tree query in which a user annotates an existing tree – the "base tree" with a query. The base tree may be any tree found in the treebank by browsing or

Table 1: LPath Navigation Axes

Type	LPath Axis	Abbreviation	Closure	Core XPath Support
Vertical	child	/		√
	descendant	/descendant::	$/^+$	√
	parent	\		√
	ancestor	/ancestor::	$\backslash^+$	√
Horizontal	immediate-following	$-\rangle$		×
	following	$--\rangle$	$-\rangle^+$	√
	immediate-preceding	$\langle-$		×
	preceding	$\langle--$	$\langle-^+$	√
Sibling	immediate-following-sibling	$=\rangle$		×
	following-sibling	$==\rangle$	$=\rangle^+$	√
	immediate-preceding-sibling	$\langle=$		×
	preceding-sibling	$\langle==$	$\langle=^+$	√
Other	self	.		√
	attribute	@		√

by an earlier search. The query is an annotation of the base tree in which a subset of the nodes are selected. Lines are drawn between pairs of selected nodes to indicate structural relationships such as "descendent" and "following" that should hold true in any results from a new query. Node labels and attributes are modified as necessary. The result of a query by annotation is a collection of trees, each annotated with the original query. This section describes the graphical elements of the query interface and shows how they correspond to LPath components.

3.1 Axes

The most basic component of a query is a relation between a pair of tree nodes. The inventory of atomic queries is shown in Figure 3, along with translations into LPath.

Observe that the expected relation can be inferred from the base tree. In the context of a graphical interface this saves effort because the user can simply connect nodes without needing to specify the relations. Users can override this default interpretation by clicking on the line to cycle through its possible interpretations, as shown in Figure 4.

3.2 Filters, nodes and attributes

We have observed that users often pose a query which generates far too many results. Evidently the user is not aware of the variety of data contained in a treebank. There are four main ways a result set can be narrowed. First, the user can make node relations more specific (e.g. change "descendent" to "child"). Second, the user can specify node attributes (e.g. mark an NP node as temporal by adding the TMP attribute). Third, the user can edit the existing query, adding new edges to more narrowly describe the desired result set. Finally, the user can add new negated edges to remove trees – such as the one currently being viewed – from the result set.

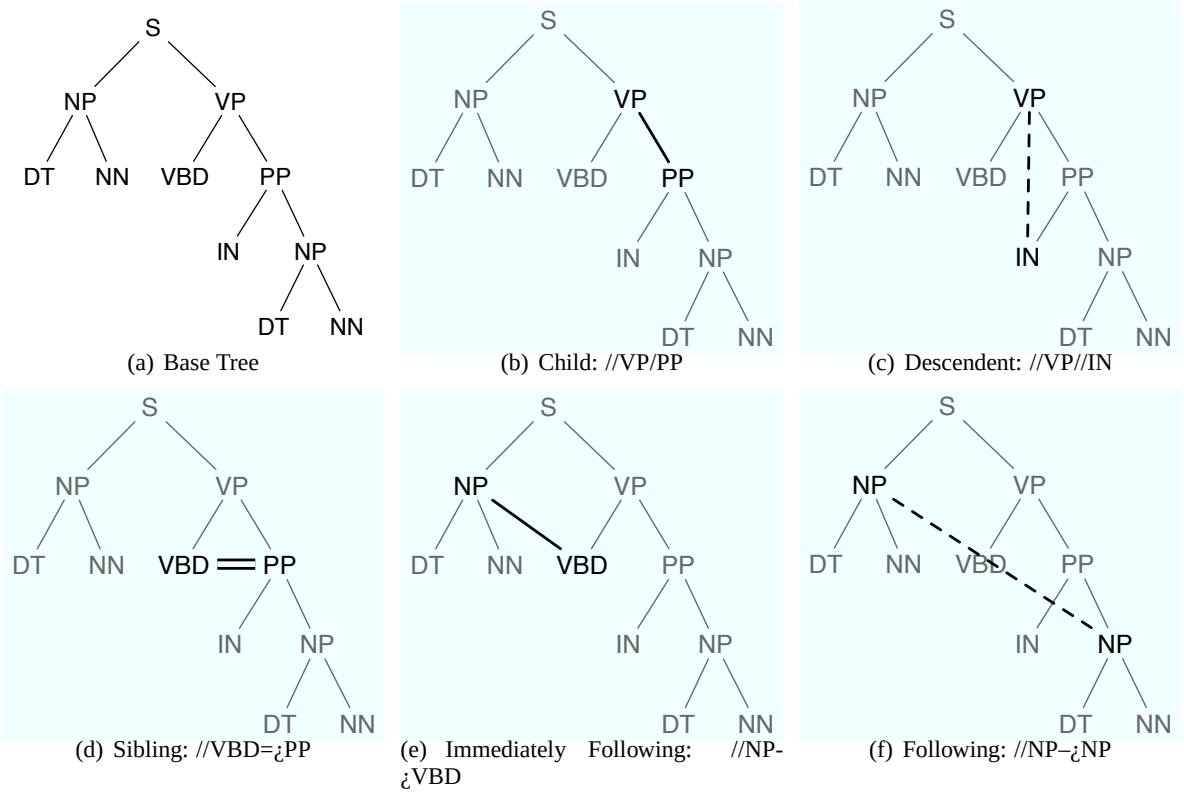
3.3 LPath alignments and scopes

The LPath language has additional features that make it more expressive than XPath. Two of these are alignment and scope, linguistically important properties that need to be represented in the graphical query.

LPath permits queries that stipulate the left- or right-alignment of a subtree within the scope of some ancestor node (e.g. to find a prepositional phrase that is final within an ancestor verb phrase). The GUI makes this expressiveness available by permitting users to right-click on a node and toggle the alignment information.

LPath also permits queries to specify that subexpressions remain within the scope of a particular node. All downward navigations – the "child" and "descendent" axes – introduce a new scope. Subsequent horizontal navigations remain inside the scope of the dominating node iff the corresonding node in the base tree also falls under that node.

For example, in the tree in Figure 3(a), a query starting at the left NP, which goes down to the child DT then across to the following NN has the following scope, by default: //NP{/DT=$\rangle$NN}. A query



(a) Base Tree (b) Child: //VP/PP (c) Descendent: //VP//IN

(d) Sibling: //VBD=¿PP (e) Immediately Following: //NP-¿VBD (f) Following: //NP–¿NP

Figure 3: Translation of QBA Primitives to LPath Axes

which starts from the same NP, then goes down to the child DT as before, then across to the following VBD actually leaves the scope of the NP, and the scope is as follows: //NP{/DT}--⟩VBD.

In the graphical interface, the depth of scope nesting is indicated using a superscript integer. The user can toggle its value to expand or shrink the scope, thereby constraining or relaxing the query (respectively).

4 Query Translation

4.1 Approaches to translation

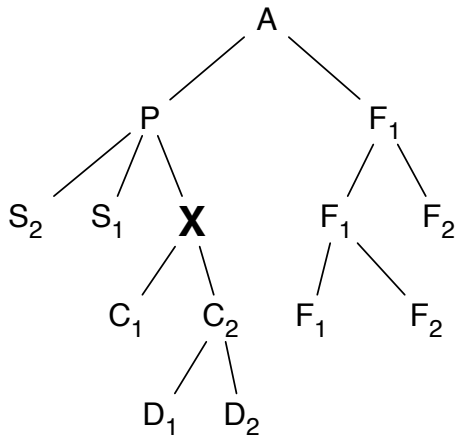
Several approaches to query translation have been investigated. Perhaps the most obvious is direct graph matching, in which a query is matched against a tree using powerful graph matching techniques (Messmer and Bunke, 1998). However, this approach is inadequate, for two simple reasons. First, the query graph is not a subgraph of the result tree. The transitive axes such as "descendent" are not explicit in the treebank. Second, queries involve negation and it is meaningless to match negated edges against actual edges in a treebank. In these respects, queries are not partial trees, but partial *descriptions* of trees.

Another approach is to use first-order logic over trees or annotation graphs (Bird and Liberman, 2001). In the case of trees, direct translation is not possible as the "immediate following" axis cannot be expressed (it would involve an arbitrary number of joins, inexpressible in a first order language). In the case of annotation graphs the dominance axes cannot be expressed (these also involve an arbitrary number of joins to navigate from an edge to another edge included within its span).

Instead, we convert the graphical queries to LPath, and to use the existing LPath query interpreter for onward translation from LPath to SQL (Bird et al., 2006). This approach imposes the restriction that the query graph must be connected, and we have not found this restriction to pose any problems in practice. (The restriction is conveniently implemented in the user interface, as it implements the notion of an active node, and new edges can only be added by linking back to this active node.)

4.2 Linear queries

So far we have seen how atomic queries are translated. The interpretation of individual edges has a well-defined default, and various alternatives that



target	default axis	other axes
A	\\	\
P	\	\\
S_1	<=	<==, <-, <--
S_2	<==	<=, <-, <--
F_1	->	-->, =>, ==>
F_2	-->	->, =>, ==>
C	/	//
D	//	/

Figure 4: Default and Alternative Interpretations of QBA Primitives Relative to Node X

are selected via the user interface (Figure 4). Consequently, we will abstract away from the identity of each axis and focus on the structure of complex queries.

The next step in increasing query complexity is a query involving two relations r_1 and r_2 and a single shared node (Figure 5(a)). The translation can start at either end of this path, and simply generate a query of the form: `Ar1Br2C`. This method generalises to linear queries of arbitrary length. It is immaterial which end we start from, as the result of a query is always a whole tree, not a set of nodes (as is the case for XPath queries).

4.3 Branching queries

In general, the structure of a query graph is a *free tree*, a tree with no specified root node and no sibling order. The smallest possible branching query involves three relations r_1, r_2 and r_3 with a single shared node, in a Y structure. The interpreter breaks this structure into a linear component and a branch (Figure 5(b)). The linear component can be translated as before. The branch is also linear and can be translated in the same way. The final step is to connect the pieces together. This is done using a *filter expression*, an expression contained inside brackets and anchored at a particular node: `Ar1B[r3D]r2C`. In this expression, `B` is located at the centre, and is related to `A`, `D`, and `C`.

Negated branches are also handled in this way. Thus if a negated edge links `B` and `D` we would have: `Ar1B[notr3D]r2C` (Figure 5(c)). Multiple branches emanating from a single node can be expressed using conjunction within the filter expression. We use nested filter expressions to translate branches upon branches (Figure 5(d)). Now, given that the linear components of queries can have arbitrary length, and branch points can occur at any node, we can translate query graphs of arbitrary complexity.

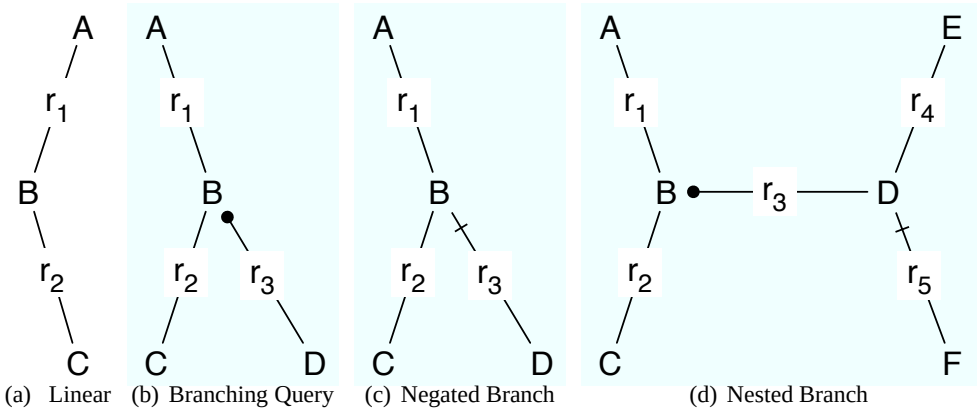
5 Result Overlay

In order to help the user identify where the query matches with the result tree, the graphical query that the user has drawn is overlayed over the new tree. This also helps the user to refine the query, which otherwise has to be redrawn from scratch.

5.1 Database query and result rendering

The graphical query drawn by the user is translated into an LPath query. This in turn is translated into an SQL query by an LPath-to-SQL translator (Bird et al., 2006) (source code available from `http://nltk.org/nltk_contrib/lpath/`). The obtained SQL query is sent to a remote database server and a result table is sent back. The result table contains a set of rows. Each row is a node in a tree that matches with the query, and it also corresponds to the last node of the original LPath query.

Unique tree ids are retrieved from the result table. For each of the ids, as requested by the user, the entire tree is retrieved (i.e. a set of rows, each for a node from the tree in question). This table is transformed to a tree and rendered on the display. Node ids in the original table are kept during query translation to permit overlay of the original query in the graphical display.



(a) Linear Query (b) Branching Query (c) Negated Branch (d) Nested Branch

Figure 5: Structure of Complex Queries

Algorithm 1 Compute and Display Overlay

```
1: procedure ComputeOverlay(G,A,S)  ▷ G is the graphical tree object; A is the nodes of query expression in DFS order; L is the node ids for one tree returned by query engine; Returns mappings of query nodes to tree nodes
2:     LA ←  ▷ mapping from L to A
3:     c ← 0
4:     for all i in L do  ▷ for all node ids
5:         LA[i] ← A[c]
6:         c+ = 1
7:     end for
8:     LG ←  ▷ mapping from L to G
9:     for all i in L do  ▷ for all node ids
10:        LG[i] ← G.search(i)
11:    end for
12:    return LA ⋈ LG
13: end procedure
14: procedure DisplayOverlay(Q,M)  ▷ Q is the query graph; M is the mapping from ComputeOverlay; Updates display
15:    for all a(i, j,t) in Q do  ▷ each axis in query
16:        draw (M[i],M[j],t)
17:    end for
18: end procedure
```

5.2 Overlay

The LPath query returns a set of nodes in the database that match the last element in the query. An extension is made to the LPath-to-SQL translator so that all nodes in the query are selected rather than just the last node in the query:

```
select T0.*, T1.* from T T0, T T1
  where T0.type='syn' and T0.name='A'
  and T1.type='syn' and T1.name='B'
  and T0.sid=T1.sid and T0.tid=T1.tid
  and T0.l<=T1.l and T0.r>=T1.r and T0.d<T1.d
```

Each row of the result table returned by this modified SQL query is thus a super-tuple that is a concatenation of sub-tuples, one per node. And in the super-tuple, sub-tuples appear in depth-first-search order relative to the tree representation of the LPath query. Then, for each super-tuple, a list of node ids is extracted, and a mapping from this list to the LPath query tree is computed. Also, using this list of ids, nodes involved in the overlay are identified from the rendered result tree. Finally, axes between nodes are recovered and displayed. These steps are shown in Algorithm 1

6 Prototype

We have implemented a prototype of QBA tool using Python and PyQt. The main components of the tool are described below. Figure 6 provides a screenshot of the tool.

An enriched tree data structure is used to store trees and graphical queries drawn by users. The GUI component renders this structure as a tree, and allows users to annotate it with a graphical query. Once the graphical query is translated to LPath, it is further translated into an SQL query by an LPath-to-SQL translator. We use NLTK to compile the LPath grammar and to parse LPath queries (`nltk.org`, Bird (2006)). In order to support query overlay, the translator also provides a modified SQL translation described in Section 5.2.

The database component maintains a connection to a database server. When a request arrives with an LPath query, it uses the LPath-to-SQL translator to translate it into an SQL query, sends the translated query to the server, and returns the result to the client. A user can connect to either Oracle or PostgreSQL database. Depending on user's connection choice, the database component is configured to one of the database systems at runtime.

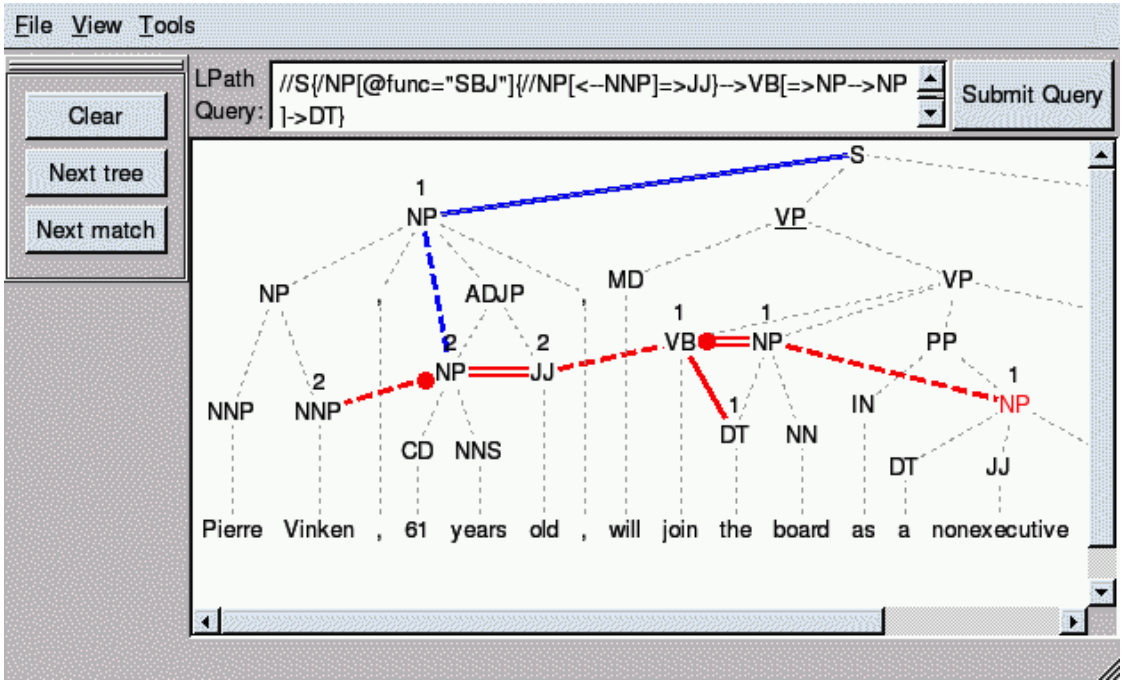


Figure 6: Screenshot of LPath QBA Tool

7 Conclusion

Treebanks have become centrally important in linguistic research and language technology. Treebanks are now being developed for dozens of languages,[1] and for a wider range of linguistic phenomena (e.g. discourse analysis, Miltsakaki et al. (2004)). Earlier work has showed how such tree data can be represented in a relational database and interrogated using a linguistically motivated path language called LPath. Here we have described a new, high-level approach to querying linguistically annotated data, which we call Query by Annotation. Unlike existing approaches to tree query, queries and results are of the same type: annotated trees. Users see why their query matches a result tree. Moreover, they can easily refine their query, constraining or expanding its scope as needed. The resulting approach to graphical query fits naturally into common workflows in linguistic data exploration and curation.

[1] For a list of languages for which treebanks are available, see `http://en.wikipedia.org/wiki/Treebank`

References

Steven Bird. 2006. NLTK: The Natural Language Toolkit. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, pages 69–72. Association for Computational Linguistics, Sydney, Australia. `http://www.aclweb.org/anthology/P/P06/P06-4018`.

Steven Bird, Yi Chen, Susan B. Davidson, Haejoong Lee, and Yifeng Zheng. 2006. Designing and evaluating an XPath dialect for linguistic queries. In *22nd International Conference on Data Engineering*, pages 52–61. `http://eprints.unimelb.edu.au/archive/00001455/`.

Steven Bird and Mark Liberman. 2001. A formal framework for linguistic annotation. *Speech Communication*, 33:23–60. `http://arxiv.org/abs/cs/0010033`.

Daniele Braga, Alessandro Campi, and Stefano Ceri. 2005. XQBE (XQuery By Example): A visual interface to the standard XML query language. *ACM Transactions on Database Systems*, 30(2):398–443.

James Clark and Steve DeRose. 1999. *XML Path language (XPath)*. W3C. `http://www.w3.org/TR/xpath`.

Stephan Kepser. 2003. Finite structure query: a tool for querying syntactically annotated cor-

pora. In *Proceedings of the Tenth Conference of the European Chapter of the Association for Computational Linguistics*, pages 179–186.

Esther König and Wolfgang Lezius. 2001. The TIGER language: a description language for syntax graphs. part 1: User's guidelines. Technical report, University of Stuttgart, Stuttgart, Germany. http://citeseer.ist.psu.edu/article/knig01tiger.html.

Catherine Lai. 2005. *A Formal Framework for Linguistic Tree Query*. Master's thesis, Department of Computer Science and Software Engineering, University of Melbourne.

Catherine Lai and Steven Bird. 2004. Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop*, pages 139–146. http://eprints.unimelb.edu.au/archive/00000774/.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–30. http://www.cis.upenn.edu/~treebank/home.html.

Bruno T. Messmer and Horst Bunke. 1998. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–504.

Eleni Miltsakaki, Rashmi Prasad, Aravind Joshi, and Bonnie Webber. 2004. The penn discourse treebank. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*. Paris: European Language Resources Association. http://www.seas.upenn.edu/~pdtb/papers/lrec04.pdf.

Jiří Mírovský. 2006. Netgraph: a tool for searching in prague dependency treebank 2.0. In *Proceedings of The Fifth International Conference on Treebanks and Linguistic Theories*, pages 211–222. http://quest.ms.mff.cuni.cz/netgraph/pub/2006_tlt.pdf.

P. Resnik and A. Elkiss. 2003. The linguist's search engine: Getting started guide. Technical Report LAMP-TR-108/CS-TR-4541/UMIACS-TR-2003-109, University of Maryland, College Park. http://lse.umiacs.umd.edu:8080/.

D. Rohde. 2001. Tgrep2 user manual. http://citeseer.ist.psu.edu/569487.html.

M. M. Zloof. 1977. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343.

Author/s:
BIRD, STEVEN; Lee, Haejoong

Title:
Graphical query for linguistic treebanks

Date:
2007

Citation:
Bird, S., & Haejoong, L. (2007). Graphical query for linguistic treebanks. In, Proceedings, PACLING 2007 - 10th Conference of the Pacific Association for Computational Linguistics, Melbourne.

Publication Status:
Published

Persistent Link:
http://hdl.handle.net/11343/34835

File Description:
Graphical Query for Linguistic Treebanks