Extending the TIGER Query Language with Universal Quantification

Torsten Marek, Joakim Lundborg and Martin Volk

Abstract. The query language in TIGERSearch is limited due to its lack of universal quantification. This restriction makes it impossible to make simple queries like "Find sentences that do not include a certain word". We propose an easy way to formulate such queries. We have implemented this extension to the query language in a tool that allows querying parallel treebanks, while including their alignment constraints. Our implementation of universal quantification relies on the view of node sets rather than single node unification. Our query tool is freely available.

1   Introduction

In the last ten years, many languages and tools for querying syntactically annotated corpora have been developed. Most of these tools and query languages have been designed for a specific corpus and a specific annotation format. TGrep2[1] was originally designed for the Penn Treebank and its tree format. TIGERSearch (König and Lezius 2003; Lezius 2002) was designed for the NEGRA and TIGER corpora (Brants et al. 2002) and uses the TIGER-XML format.

TIGERSearch is a tool that allows the user to search a treebank by formulating queries in feature-value pairs. For example, one may search for the word 'can' with the part-of-speech 'noun' by querying

(1.1)   [word="can" & pos="NN"]

In addition to constraints over words, the TIGER query language allows the user to also use constraints over dominance relations (search for a node *n1* that dominates a node *n2* in the tree), precedence relations and node predicates (such as arity, discontinuous, and root).

In general, the design of the input format influences to a large extent the design of the query language, since it defines what can be queried. For instance, TIGER-XML supports crossing branches, leading to non-terminal nodes whose terminals

---

1. `http://tedlab.mit.edu/~dr/TGrep2/`

are not a proper substring of the sentence (discontinuous nodes). The TIGER query language thus has special functions for dealing with discontinuous nodes. In contrast, the Penn Treebank formalism does not support crossing branches directly, and thus TGrep2 has no means to handle this notion.[2]

While certain limitations of query languages are thus due to the original design and could only be approximated, other interesting queries may simply be missing from the query language. Lai and Bird (2004) list seven example queries, named Q1-Q7, which each formalism should support, regardless of the annotation formalism[3].

In this paper, we will deal with queries that require universal quantification, i.e. selecting a tree by stating constraints over possibly many nodes rather than individual nodes. The sample queries contain two examples where universal quantification is needed (Lai and Bird 2004: p. 2):

Q2. Find sentences that do not include the word 'saw'.

Q5. Find the first common ancestor of sequences of a noun phrase followed by a verb phrase.

With the TIGER query language and its implementation TIGERSearch, these queries can only be approximated. The result set generated for the approximated queries will likely contain errors.

In section 2, we will analyze the limitations of TIGER and TIGERSearch and look at the solution proposed in Lezius (2002). In section 3, we will develop an extension to the TIGER query formalism that is able to deal with Q2 and similar queries, and we will analyze the different requirements of Q2 and Q5 in section 5.

In section 4, we will talk about the implementation of the extensions from the previous section in our own implementation of the TIGER query language, which is part of the Stockholm TreeAligner[4] (Volk et al. 2007).

In the remainder of the paper, we will speak of syntax *graphs* rather than trees. These graphs are acyclic, directed and do not contain structure sharing (i.e. each node has exactly one direct ancestor). However, due to the existence of crossing branches, TIGER trees cannot be stored as nested lists or XML DOM trees directly, which is the usual format of trees.

Node descriptions are boolean expressions of feature constraints of the form "[feature=value]". They are used for finding nodes (assignments) in the corpus which are then used for the constraint resolution in TIGER queries.

---

2. Crossing branches should not be confused with secondary edges. While crossing branches lead to crossing edges in the primary level of a graph, secondary edges constitute a secondary level.

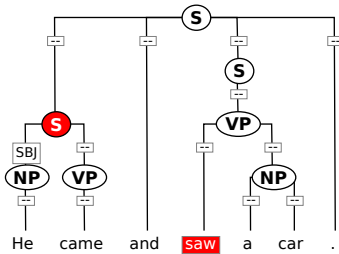3. with one exception: Q6 assumes multiple layers of annotation.

4. `http://dev.ling.su.se/treealigner`
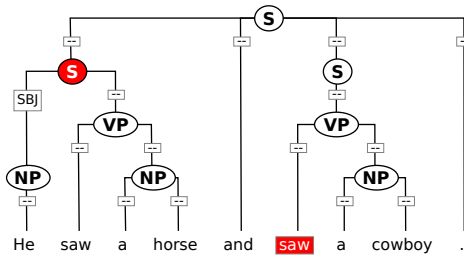
Figure 1. Wrong highlights.
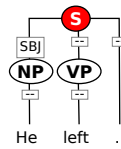

Figure 2. A false positive.　　　　　　　*Figure 3.* A false negative.

2　　Limitations of the TIGER Query Language

Figures 1 to 3 show three example graphs for Q2 ("Find sentences that do not include the word 'saw'"). If the query were evaluated correctly, the result set would only contain the first coordinated sentence from figure 1 ("He came") and the main sentence of figure 3 ("He left").

In the TIGER query language however, every node variable is implicitly existentially quantified. For example, the query

(1.2)　　　　　　　　　#s:[cat="S"] !>* #w:[word="saw"]

states that some node of category S does not dominate some word with the surface string 'saw'. This query is interpreted by TIGERSearch so that it returns all combinations of two nodes #s, #w[5] in all graphs, such that #s does not dominate #w.

---

5. the hash symbol '#' is used to mark variables

From the graphs that were actually meant to be matched by Q2, it will only return those that have a terminal 'saw' outside of any S node. Thus, the result set contains the graph from figure 1 (with extra highlighting, which is distracting, but not fatal), but also, for instance, the graph from figure 2, which should definitely not be matched by the query. But all graphs that do not contain 'saw', like the one in figure 3, will not show up in the result set.

Another attempt to formulate Q2 in TIGERSearch is the query

(1.3)   #s:[cat="S"] >* #w:[word!="saw]"

which states that some node of category S dominates some terminal that does not have the surface string 'saw'. Looking at the example graphs, it immediately becomes clear that there are many combinations of nodes that satisfy this query.

In general, there is no correct way to formulate the desired query in TIGER-Search. This limitation is acknowledged by the developers:

> The use of the universal quantifier causes computational overhead since universal quantification usually means that a possibly large number of copies of logical expressions have to be produced. For the sake of computational simplicity and tractability, the universal quantifier is (currently) not part of the TIGER language. (TIGERSearch Help, section 10.3)

Section 5.7 of Lezius (2002) contains a proposal for an extension of the TIGER query formalism that combines universal quantification and the implication operator. Using this syntax, Q2 can now be formulated as

(1.4) $\forall$#w ([cat="S"] >* #w) $\Rightarrow$ #w:[word!="saw"]

which means that every node dominated by an S node must not be the surface string 'saw'.

Expressing the queries with the implication operator would be natural for the unification-based evaluation of queries in TIGERSearch. As already mentioned, an actual implementation comes at great computational cost. For each $\forall$ clause in the query, all nodes in the graph have to be iterated to find out if they satisfy $l \Rightarrow r$. In some cases, using the logical equivalent $!r \lor l$ can be used to speed up the queries. For instance, Q2 can simply be evaluated by

1. retrieving all S nodes in all graphs

2. retrieving all nodes where [word="saw"].

3. for each S node, check if it dominates none of the nodes from step 2.

Apart from runtime complexity considerations, the syntax would be extended with a construct that is conceptually hard to grasp and that makes the grammar of the query language much more complex. We therefore decided to explore a different path.

3 Design of the Universal Quantification Extension

The Lezius solution presented in section 2 builds on the query calculus that is at the core of TIGERSearch's query evaluation engine. This calculus is based on unifying the partial graph description given by the query with any of the graph definitions in the corpus. If the unification succeeds, the graph matches the query and is returned in the result set.

In contrast, the query engine in the Stockholm TreeAligner is based on node sets, and combinations of nodes from the different sets to satisfy the constraints given in a query[6].

In the previous analysis of Q2, we showed that it is possible to rephrase the query using logical equivalents. Therefore, the query "get all S nodes that do not contain the word saw" can be rephrased into "get all graphs where all instances of 'saw', if any, are not dominated by a specific S node".

This is essentially expressed in query 1.2. But as we already showed, the usage of the existential quantification will not lead to the expected results. However, if one of the two operands is not understood to be a single node from the graph, but a *set* of nodes, the result will be correct. Therefore, we introduce a new data type into the query language, the node set which we indicate with the % symbol. Bare node descriptions are still bound by an implicit existential quantifier as before. A node set is only bound to a variable that starts with a percentage symbol:

(1.5)   #s:[cat="S"] !>* %w:[word="saw"]

If one operand in a constraint is a node set instead of a node, the semantics of the constraint are changed. Here, only those assignments to #s are returned where the constraint '!>*' holds for each node in the node set %w, which contains all terminals with the surface string 'saw'. When applied to the graphs from the small sample corpus, this query now does not yield any false positives like the graph in figure 2.

The semantics of node predicates as defined in the TIGER query language do not change, they still operate at the node level. In the query

(1.6)   %np:[cat="NP"] & tokenarity(%np, 2)

---

6. cf. section 4 for a brief outline of the evaluation strategy.

the node set %np will contain all NPs whose token arity is 2 (which means that each NP dominates exactly two tokens). See section 5 for a further discussion.

3.1   Node Sets

If each variable is bound by an existential quantifier, evaluation of a query can terminate as soon as one node description does not yield any results. Graphs that do not contain matching nodes for any of the descriptions will also be disregarded, which is why the graph from figure 3 will still not be matched by the query. To produce correct results, the semantics of node descriptions bound to node sets have to be changed. In contrast to existentially quantified nodes, which may not be undefined, a set can be the empty set $\emptyset$. If this is the case, a constraint is trivially true.

With this change in place, TIGER is in Cantor's paradise, and no one shall expel it from there. The basic semantics of set types are defined and new set predicates can be introduced to make set queries more powerful. As an example, consider the query "Return all NPs that do not contain any PP, but only if the graph contains PPs". Given that empty node sets are now allowed, the query has to be written as

(1.7)   [cat="NP"] !>* %pp:[cat="PP"] & [cat="PP"].

The last term ensures that at least one PP exists in the graph. As a side effect, the result set will contain one entry for each combination of NP and PP in a matching graph, which is slightly more than what the query was supposed to yield.

To express constraints on set cardinality, a syntax for set algebra operations could be added to TIGER. As an example, to make sure that the set %pp contains elements, one could think of something like

(1.8)   [cat="NP"] !>* %pp:[cat="PP"] & {size(%pp) > 0}

Any expression enclosed in curly brackets is evaluated as an operation on sets. This addition would add a lot of power to the query language, but would make it much harder to use. Also, it requires a nontrivial amount of implementation effort and makes the query grammar more cumbersome. In our opinion, node set operations go beyond the scope of what can conveniently be handled inside of a single expression. If these expressions were added, it would be desirable to store node sets, reuse them in later queries or combine node sets from different queries.

Instead of adding full support for set operations, we introduce two new predicates that operate exclusively on node sets: *empty* and *nonempty*. The semantics of the

predicates can be inferred from the names, and the previous query can be written in a straightforward manner:

(1.9)   [cat="NP"] !>* %pp:[cat="PP"] & nonempty(%pp)

It is now also possible to search for graphs that do not contain a specific kind of node by using *empty*. The query

(1.10)   %w:[pos="det"] & empty(%w)

returns all graphs that do not contain any determiner.

4   Implementation of the Query Language within the Stockholm TreeAligner

We have re-implemented the TIGER query language in a tool for the creation and exploration of parallel treebanks, the Stockholm TreeAligner. The TreeAligner allows the user to load two treebanks, typically with parallel (i.e. translated) sentences. For example, we have used the TreeAligner to work on a German treebank and its parallel English treebank. We have aligned the two treebanks first on the sentence level to get corresponding tree pairs and then on the word level and phrase level. Figure 4 shows a tree pair from our parallel treebank.

We currently distinguish between exact translation correspondence represented by green lines and approximate ('fuzzy') translation correspondence represented by red lines. Although it is useful in principle to make this distinction, applying it consistently in practice has proven to be difficult.

The TreeAligner is an editor which allows the user to create and modify alignments. It can also be used to browse and search parallel treebanks and this is where the query language comes in. We have re-implemented and extended the TIGER query language within the TreeAligner to search both treebanks in parallel. We have also added alignment constraints in order to combine the queries. Let us consider the following example query:

	treebank1	#node1:[cat="NP"] > [cat="PP"]
(1.11)	treebank2	#node2:[cat="NP"] > [cat="PP"]
	alignment	#node1 * #node2

Here, #node1 is a variable that identifies a node of category NP in a graph in treebank1. Likewise, #node2 identifies a node of category NP in *treebank2*. These
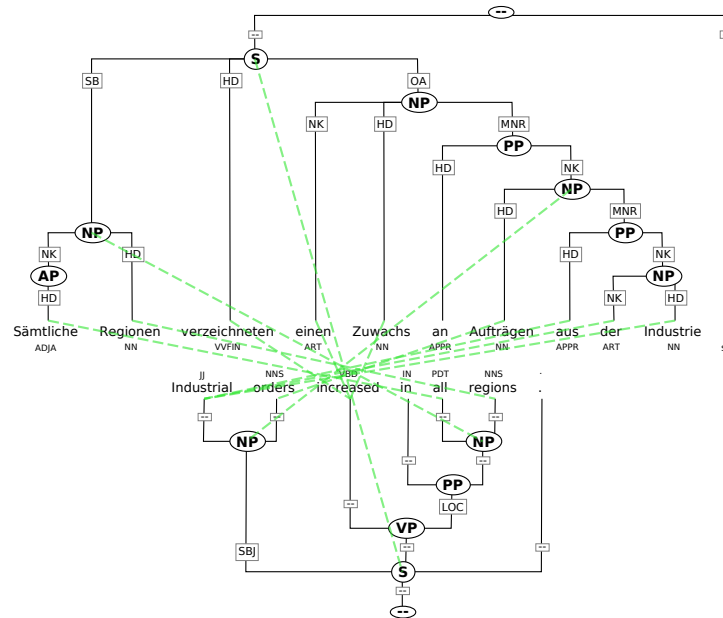
Figure 4. An example of two aligned sentences from the SMULTRON corpus

variables correspond exactly to the syntax in the TIGERSearch query language. We then use these variables in the alignment query. The general alignment relation is indicated by the '*' operator. For a detailed description of the alignment query syntax, see Volk et al. (2007). This query searches through both *treebank1* and *treebank2* for noun phrases that dominate a prepositional phrase and returns all matches where these noun phrases are aligned with each other.

The implementation of the TIGER query language in the Stockholm TreeAligner is based on sets of nodes and constrained Cartesian products over these sets. Because of that, it was possible to implement the extensions described in section 3 with little effort. In contrast to existentially quantified nodes, node sets are subject to some restrictions. In a constraint, at most one operand may be a node set. Constraints that have two node sets as operands will lead to a runtime error[7]. In the result display, in contrast to existentially quantified nodes, nodes from sets are not highlighted. While

---

7. This behavior may change in the future. However, to this date we do not have linguistically interesting queries that require a constraint with two node sets.

there are no technical reasons for this, having node sets stand out from the rest is not helpful, simply because too many nodes would be highlighted. This would lead to a confusing result display.

5   Beyond Node Descriptions

Using the implication operator, Q5 (finding the first common ancestor of a sequence NP VP) can be expressed in the following manner (where NT stands for any non-terminal):

(1.12)

$$
\begin{aligned}
&\text{\#a:[NT] >* \#np:[cat="NP"] \&} \\
&\text{\#a >* \#vp:[cat="VP"] \&} \\
&\text{\#np .* \#vp \&} \\
&\text{\#np !>* \#vp \& \#vp !>* \#np \&} \\
&\forall\text{\#b}\,(\text{\#a >* \#b \& \#b >* \#np}) \Rightarrow (\text{\#b !>* \#vp})
\end{aligned}
$$

The query looks for a node #a that dominates a sequence of nodes #np #vp, but only if no node on the path from #a to #np (the condition of the $\forall$ clause) also dominates #vp. The fourth line of the query is needed because one of the nodes #np, #np could dominate the other one, as well as precede it, given that its left corner precedes the left corner of the other one. With these restrictions in place, it is made sure that #a really is the first common ancestor.

It is clear that this query cannot be handled by the node set definition syntax, because, unlike in Q2, the right-hand side of the implication tests for a non-local feature of the node.

For these more complex queries, we propose an extension that is functionally equivalent to the implication syntax, but also relies on the notion of node sets. If the right-hand side of a node set definition is enclosed in curly braces, the contents of the definition are interpreted as a *sub-query* rather than a node description, which are enclosed in square brackets. In this sub-query, all existential node variables from the outer query may be referenced. In the example query, the node set of all nodes on the path between #a and #np would be defined as

(1.13)   %b:{#a >* %:[NT] & % >* #np}.

The free-standing percent sign is used as the placeholder for those nodes that are going to be elements of the set[8]. With the node set %b defined this way, query 1.12

---

8. This syntax is still subject to change.

can be rephrased as

(1.14)
$$\#np:[cat="NP"] .* \#vp:[cat="VP"] \&$$
$$\#a:[NT] >* \#np \&$$
$$\#a >* \#vp \&$$
$$\#np !>* \#vp \& \#vp !>* \#np \&$$
$$\%b:\{\#a >* \%:[NT] \& \% >* \#np\} !>* \#vp$$

While this query will need more time for evaluation, the evaluation order is essentially unchanged:

1. Get all nodes that can match #np, #vp, #a and %

2. For a given assignment, execute the sub-query that defines %b

3. Evaluate all other constraints.

While this approach is easy to implement, it is also very slow. Greater speeds can be achieved by early pruning of the search space, for instance by employing additional indices or intelligent constraint evaluation reordering and caching.

With sub-queries, the semantics of node predicates should be changed as well. A node predicate that is invoked on a set of nodes will only return true if all nodes in the set satisfy the predicate. To build the set of all nodes that fulfill a certain condition, a sub-query should be used. For instance, the set of all NP nodes with exactly two children can be created with

(1.15)
$$\%x:\{\%:[cat="NP"] \& arity(\%, 2)\}$$

6   Related Research

The Stockholm TreeAligner is unique in its ability to display parallel syntax trees and their phrase alignment. It can be seen as an advancement over tools like Cairo (Smith and Jahr 2000) or I*Link (Merkel et al. 2003) which were developed for creating and visualizing word alignments but are unable to display trees. The TreeAligner is also related to Yawat and Kwipc (Germann 2007), two software tools for the creation and display of sub-sentential alignments in matrices.

Our implementation of the query language owes a lot to TIGERSearch, whose ground-breaking work and robust implementation we gratefully acknowledge. But our work is also related to Emdros (Petersen 2004), which has its own system for

parallel corpus searches. In Petersen (2005), the author of Emdros shows that his system can handle Q2 correctly and is faster than TIGERSearch. Other related query systems are NXT search/NiteQL (Evert and Voormann 2003; Heid et al. 2004) and LPath (Bird et al. 2005) none of which is specifically geared towards parallel tree-banks.

Extending the TIGER query language with set logic operations has been proposed and implemented in Saers (2006), however in a different manner. The results of several queries can be combined with set operations, which allows to express queries which need a universal quantifier if expressed in a single query.

LPath is an extension of XPath to support querying syntactic trees that are encoded in XML info sets. All syntactical relations (dominance, precedence) are directly encoded in the XML structure and thus inherit the restrictions that apply to XML trees. Secondary layers of annotation, even simple ones like the secondary edge feature in TIGER-XML, are not supported. LPath also provides an extension to the XPath syntax allowing `not` modifiers on predicates, making it possible to express Q2 in a straight-forward manner.

NiteQL has been designed for corpora with multiple layers of annotation on one and the same text, but also heavily relies on XML to encode syntactical structure, with the same restrictions. According to Lai and Bird (2004), NiteQL cannot handle queries that need universal quantification. The authors of NiteQL also say

> TigerSearch (sic) has a nice graphical interface and again supports structural operators missing in NQL. (Nite XML Toolkit Documentation, section 8.4)

but do not give any examples for the differences.

7   Conclusion and Outlook

We have shown how the TIGERSearch query language has been integrated in a parallel treebank exploration tool, the Stockholm TreeAligner. This tool allows for parallel querying over two treebanks combined with their alignment constraints and is thus a useful tool for cross-language comparisons and translation studies (cf. Lundborg et al. (2007) for an introduction).

The TIGERSearch query language lacks universal quantification and is thus unable to handle certain queries. We have added this functionality to the query language and implemented it in the TreeAligner. In this paper, we have also sketched a possible syntax for queries that contain more complex universal quantification.

The extensions described in section 3 will be part of the next release of the Stockholm TreeAligner later this year, and are already implemented in the development version. This release will also contain a browser and query interface for monolingual

treebanks, similar to the one provided by TIGERSearch. While the functionality for monolingual queries is already (and has always been) provided by the underlying implementation, so far the focus of the TreeAligner has been on creating and querying parallel treebanks.

We will also follow up on the planned extensions sketched in section 5. However, the TIGER implementation in the Stockholm TreeAligner is still in a very early stage. Feature completeness and a faster query evaluation will become more important for the upcoming releases. We will also provide data on the efficiency of queries with universal quantification as well as a comparison of query evaluation speed in both the Stockholm TreeAligner and TIGERSearch.

8   Acknowledgements

Bibliography

Bird, Steven, Yi Chen, Susan Davidson, Haejoong Lee, and Yifeng Zheng (2005). Extending XPath to support linguistic queries. In *Proc. of Programming Language Technologies for XML (PLANX)*, 35–46, Long Beach, California.

Brants, Sabine, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith (2002). The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories*, Sozopol, URL `citeseer.ist.psu.edu/brants02tiger.html`.

Evert, Stefan and Holger Voormann (2003). NQL – A query language for multi-modal language data. Technical report, IMS, University of Stuttgart, Stuttgart.

Germann, Ulrich (2007). Two tools for creating and visualizing sub-sentential alignments of parallel text. In *Proc. of The Linguistic Annotation Workshop at ACL 2007*, 121–124, Prague.

Heid, Ulrich, Holger Voormann, Jan-Torsten Milde, Ulrike Gut, Katrin Erk, and Sebastian Padó (2004). Querying both time-aligned and hierarchical corpora with NXT search. In *Proc. of The Fourth Language Resources and Evaluation Conference*, Lisbon.

König, Esther and Wolfgang Lezius (2003). The TIGER language – A description language for syntax graphs, formal definition. Technical report, IMS, University of Stuttgart.

Lai, Catherine and Steven Bird (2004). Querying and updating treebanks: A critical survey and requirements analysis. In *Proceedings of the Australasian Language Technology Workshop*.

Lezius, Wolfgang (2002). *Ein Suchwerkzeug für syntaktisch annotierte Korpora*. Ph.D. thesis, IMS, University of Stuttgart.

Lundborg, Joakim, Torsten Marek, Maël Mettler, and Martin Volk (2007). Using the Stockholm TreeAligner. In *Proc. of The 6th Workshop on Treebanks and Linguistic Theories*, Bergen.

Merkel, Magnus, Michael Petterstedt, and Lars Ahrenberg (2003). Interactive word alignment for corpus linguistics. In *Proc. of Corpus Linguistics 2003*, Lancaster.

Petersen, Ulrik (2004). Emdros: A text database engine for analyzed or annotated text. In *COLING '04: Proceedings of the 20th international conference on Computational Linguistics*, 1190, Morristown, NJ, USA: Association for Computational Linguistics, doi:http://dx.doi.org/10.3115/1220355.1220527.

Petersen, Ulrik (2005). Evaluating corpus query systems on functionality and speed: TIGERSearch and Emdros. In *Proc. of The International Conference on Recent Advances in NLP*, Borovets.

Saers, Markus (2006). TiSS: Tiger set search. Termpaper in the GSLT course on "Treebanks: Formats, Tools and Usage".

Smith, Noah A. and Michael E. Jahr (2000). Cairo: An alignment visualization tool. In *Proc. of LREC-2000*, Athens.

Volk, Martin, Joakim Lundborg, and Maël Mettler (2007). A search tool for parallel treebanks. In *Proc. of The Linguistic Annotation Workshop (LAW) at ACL*, Prague.