# INESS-SEARCH: A SEARCH SYSTEM FOR LFG (AND OTHER) TREEBANKS

Paul Meurer
Uni Computing, Bergen

Proceedings of the LFG12 Conference

Miriam Butt and Tracy Holloway King (Editors)

2012

CSLI Publications

http://csli-publications.stanford.edu/

**Abstract**

This paper describes INESS-Search, a new search tool for constituency, dependency and LFG treebanks. The tool is derived from TIGERSearch and has been extended to encompass full first-order predicate logic over node variables. In addition, several operators have been implemented that are specific for querying c- and f-structures. The original TIGERSearch syntax has been extended and considerably simplified, thus making a graphical query input device less necessary. The search index is dynamically updated when the treebank is modified. The INESS-Search tool is usable via a Web interface as an integrated part of INESS, the Norwegian Infrastructure for the Exploration of Syntax and Semantics.

# 1 Introduction

In the last two decades, many tools for querying traditional dependency and constituency treebanks have been developed. They all differ in expressiveness, query language and formalism, ease of use, and applicability to specific kinds of treebanks. But no tool has been developed previously that can handle LFG treebanks, since LFG treebanks of a size that calls for a dedicated and powerful search tool have only recently been emerging.

The structural representation of syntactical analyses in Lexical Functional Grammar is quite different from and more complex than the tree-like structures that we encounter in traditional treebanks. Whereas c-structures in fact are proper (ordered) trees, f-structures can be described as unordered directed graphs, possibly with cycles. In addition, c- and f-structures are interconnected by virtue of the projection relation holding between c-structure nodes and sub-f-structures, and thus must be seen in combination (again, formally describable as a directed graph). In contrast, the structures that are prominent in traditional treebanks are the following:

- proper ordered trees, with or without labeled edges (e.g. the Penn Treebank)
- proper unordered trees as used in dependency treebanks derived from Constraint Grammar (e.g. the Sami treebanks in INESS)
- trees equipped with some additional structure peculiar to a specific framework or treebank (e.g. secondary edges and crossing edges in the Tiger treebank)
- unordered trees with or without secondary edges in some dependency treebanks (e.g. the PROIEL treebank (Haug, 2008) in INESS)

None of those tree varieties are equivalent to general directed graphs.

As a consequence, existing treebank search tools, which are designed to operate on traditional treebanks, are unable to cope with fully general directed graphs. Some of those tools are designed or implemented in a way that makes them in principle unsuited for general directed graphs, as there is no way to extend them beyond proper trees. Examples are LPath$^{+}$ (Lai & Bird, 2005) and MonaSearch (Maryns, 2009).

Other tools are in principle extensible to directed graphs, like TIGERSearch, Emdros (Petersen, 2005) and fsq (Kepser, 2003).

Among those tools, TIGERSearch (Koenig, Lezius, 2003) was identified as a suitable basis for the implementation of INESS-Search for the following reasons:

- TIGERSearch is equipped with an elegant and concise query language that can easily be extended to meet the needs of a more general search tool.
- The implementation of dominance relations in TIGERSearch via Gorn addressing naturally extends to directed graphs and lends itself to an elegant implementation of circularity detection.
- The Java implementation of TIGERSearch is reasonably fast, so one could expect that a reimplementation would have acceptable query execution speed.

INESS-Search contains extensions necessary to query fully general directed graphs like LFG f-structures, but also implements the full functionality of TIGERSearch and thus can be used to query constituency and dependency treebanks.

Whereas the expressive power of the query language of TIGERSearch can be characterized as roughly equivalent to the existential fragment of first-order predicate logic over node variables, the query language of INESS-Search is equivalent to full first-order predicate logic.

The INESS-Search tool is useable via a Web interface as an integrated part of INESS, the Norwegian Infrastructure for the Exploration of Syntax and Semantics (Rosén et al., 2012).

## 2 Abbreviated syntax and specialized operators

In order to make the syntax of the query language concise and easy to use, the original TIGERSearch syntax has been extended with convenient abbreviations and path-like concatenation of operators. Variables in operator expressions can be omitted when they are not needed for coreferencing in other relations. Examples for full and abbreviated syntax are given below.

- *Terminal nodes*

| | |
|---|---|
| full: | [word="Sophie"] |
| abbreviated: | "Sophie" |

- *Node labels*

| | |
|---|---|
| full: | #c:[cat="NP"] |
| abbreviated: | #c:NP |

- *Operator concatenation*

| | |
|---|---|
| full: | [cat="IP"] $>$ #x:[cat="NP"] & #x $>$ [cat="N"] |
| abbreviated: | IP $>$ NP $>$ N |

- *Omission of variables in relations*

| | |
|---|---|
| full: | #f $>$OBJth #g |
| abbreviated: | $>$OBJth |

In addition, several operators have been implemented that are specific for querying complex tree node and f-structure constellations:

- A *rule operator*, which has the shape of a derivation rule and makes it possible to specify relations between mothers and daughters

  #c $\rightarrow$ AP .$*$ PP

- A *c-command operator*

  #n $>$c$>$ #c

Some operators are specific to LFG c- and f-structures:

- A *path operator* specifying regular expressions over f-structure attributes:

  $g$ is either the value of OBJth of $f$ or contained in the ADJUNCT set of $f$

  #f $>$( OBJth $|$ ADJUNCT \$ ) #g

  $g$ is both OBJ and TOPIC of $f$

  #f $>$( OBJ $\&$ TOPIC ) #g

- A *projection operator*: tree node $c$ projects to the f-structure $f$

  #c $>>$ #f

- A *projective equivalence operator*: nodes $c_1$ and $c_2$ are in the same projective domain

  #c1 $>><<$ #c2

- An *extended-head operator*: $n$ is the extended head of $c$ according to the definition given in Bresnan (2001)

  #n $>$h$>$ #c

Many of these operators could in principle be expressed and implemented using more primitive relations like dominance and labeled dominance. Defining them as dedicated operators however has two advantages: queries can be expressed more concisely, and the operators can be hard-coded, resulting in dramatically improved performance.

The syntax of INESS-Search is sufficiently compact and intuitive to make elaborate graphical query devices unnecessary, especially in the case of relatively simple searches. Moreover, in the case of more complex searches involving advanced operators and quantification, a GUI would face expressiveness challenges. Instead, we will in further work explore the possibilities offered by predefined examples and cached previous queries.

## 3 Querying parallel treebanks

INESS-Search is being extended with a parallel search mode (Dyvik, Meurer, Rosén & De Smedt, 2009). This mode is still in an experimental stage. The main idea is that for aligned sentence pairs, certain nodes (tree nodes or c-structure nodes and sub-f-structures) will be aligned. To make alignment searchable, an alignment relation has been introduced as shown in (1).

(1) #s $>>>$ #t

This relation holds if $s$ is instantiated by a node in the source c- or f-structure, $t$ is instantiated by a node in the target c- or f-structure, and those nodes are aligned. Thus, query (2) will match all aligned pairs of analyses in a Norwegian–English parallel treebank where a source c-structure lexical node “jente” is aligned with a target c-structure lexical node “girl”.

(2) #s:“jente” $>>>$ #t:“girl”

An alignment relation can of course be part of a more complex query expression, as (3) illustrates. This query will match a source c-structure node dominating a lexical node “jente”, aligned with a target c-structure node dominating a lexical node “girl”.

(3) #s $>$ “jente” & #t $>$ “girl” & #s $>>>$ #t

Our approach is influenced by Volk, Lundborg & Mettler (2007), who were the first to devise a syntax for querying node alignment based on TIGERSearch, which they implemented in the Stockholm Tree Aligner tool.

## 4 Expressivity

The expressive power of the original TIGERSearch query language is equivalent to the existential fragment of first-order predicate logic over node and value variables.

In TIGERSearch, all variables are implicitly existentially quantified and universal quantification is not available. Unfortunately, with existential quantification alone, many seemingly basic queries cannot be expressed, as we will see below.

Therefore, the query language of INESS-Search has been equipped with unrestricted universal quantification over node variables and a couple of new predicates and operators including the equality operator. Its expressivity is equivalent to full first-order predicate logic over node variables (with the less important addition of value variables, which are always existentially quantified).

The introduction of universal quantification increases the complexity of the query language; new notational devices have to be introduced, and they have to be provided with an interpretation in terms of predicate calculus. Since in TIGER-Search all variables are existentially quantified, quantification does not have to be specified explicitly, that is, no quantifier expressions (i.e., $\exists x \exists y$ : ...) are needed. When both existential ($\exists x$ : ...) and universal quantification ($\forall y$ : ...) are possible, quantification has to be specified explicitly. This, however, can clutter a query expression considerably. Therefore, notational conventions are introduced that make the use of explicit quantifiers unnecessary in most cases.

First, the variable marker $\#$ is interpreted as an existential quantifier marker; each variable occurring with a $\#$ (and being in a positive context; see below) introduces an existential quantifier in prenex form (i.e., standing to the left and scoping over all terms of the expression). Also implicit variables, variables that are tacitly introduced via an abbreviated syntax construction, are existentially quantified. Thus, a query expression like (4) is translated into the logical form (5). Since both quantifiers are of equal type, the quantifier order is insignificant.

(4) #x $>$ #y

(5) $\exists x \exists y : x > y$

In order to express universal quantification, a new variable marker $\%$ is introduced.[1] A variable marked with $\%$ is universally quantified and introduces a universal quantifier in prenex form. The expression (6) translates to the logical form (7).

(6) #x $>$ $\%$y

(7) $\exists x \forall y : x > y$

When existential and universal variables cooccur in one query expression as in (6), quantifier order is no longer arbitrary. If the quantifier order is not specified explicitly, a default scoping rule determines that all universal variables are in the scope of all existential variables.

If the default scoping order is not the intended one, scoping can be specified explicitly by stating the intended quantifier order in parentheses at the beginning of the query expression:

(8) (%y #x): #x $>$ $\%$y

[1] See Marek, Lundborg & Volk (2008), who first introduced the use of $\%$ as a notational device for universal quantification, but gave it a different interpretation.

Query (8) translates to the logical form (9).

(9) $\forall y \exists x : x > y$

It is also important to note how constraints on variables are interpreted in the case of universal quantification. A constraint like #x:[cat='NP'] (stating that $x$ should be an NP node) can either be realized as a predicate clause in the logical form: $\exists x$ : cat($x$, 'NP'), or it could be interpreted as a *restricted quantifier*[2]: $\exists x$.cat($x$, 'NP'). In the case of existential quantification, the two interpretations are equivalent.

However if we consider an example like (10) that involves universal quantification, the two interpretations given in (11) and (12) are no longer equivalent.

(10) *Find all sentences where each NP directly dominates an N*

(%x #y): %x:NP $>$ #y:N

(11) $\forall x \exists y$: cat($x$, 'NP') $\wedge$ cat($y$, 'N') $\wedge$ $x > y$

(12) $\forall x$.cat($x$, 'NP') $\exists y$.cat($y$, 'N'): $x > y$

In interpretation (11), variable $x$ ranges unrestrictedly over all nodes, and the predicate cat($x$, 'NP') requires that every node be an NP node, which is clearly not the intended interpretation of (10). In interpretation (12) however, $x$ ranges over the restricted domain of NP nodes, and only for each of those, a dominated N node has to exist.

Thus, the restricted quantifier interpretation of constraints is the intended one, and the one that is implemented. To make this interpretation more explicit, the constraints can also be placed together with the quantifiers, as in (13).

(13) (%x:NP #y:N): %x $>$ #y

Further complications arise when we introduce negation. Consider example (14), where the node variable $z$ is only mentioned in a negative context.

(14) *A* PP *node dominating an* N *node with no intervening* PP *node*

#x:PP $>*$ #y:N & !(#x $>*$ #z:PP $>*$ #y)

The intended meaning of the query, phrased in prose, is: "Find nodes $x$ (PP) and $y$ (N) such that there is no node $z$ (PP) lying between $x$ and $y$." Thus, $z$ is interpreted as existentially quantified in the scope of the negation. (Note that $x$ and $y$ are already existentially quantified outside the scope of the negation.) This leads to the logical form (15).

(15) $\exists x$.cat($x$, 'PP') $\exists y$.cat($y$, 'N'): $x >* y \wedge \neg(\exists z$.cat($z$, 'PP'): $x >* z >* y)$

[2] A restricted quantifier expresses a restriction on the domain over which the variable in question ranges.

This logical form can be transformed into prenex form (16), which is the canonical form underlying the implementation of the query expressions.

(16) $\exists x$.cat($x$, 'PP') $\exists y$.cat($y$, 'N') $\forall z$.cat($z$, 'PP'): $x >\!* \; y \wedge \neg(x >\!* \; z >\!* \; y)$

Observe that by moving it out of the scope of the negation, the existential quantifier is transformed into a universal quantifier. In the same way, a negated universal quantifier resurfaces as an existential quantifier in prenex form.

We should keep in mind that the TIGERSearch query language does allow constraint variables and value variables, in addition to node variables. In query (17), $c$ is a value variable that is used to express that $x$ and $y$ should have equal *cat* values. The corresponding logical form is given in (18).

(17) #x:[cat=#c] $>*$ #y:[cat=#c]

(18) $\exists x \exists y \exists c$: $x >\!* \; y \wedge$ cat($x$, $c$) $\wedge$ cat($y$, $c$)

INESS-Search allows constraint and value variables to occur only with existentially quantified node variables that are not in the scope of a universal quantifier since it is otherwise difficult to give a sensible interpretation.

The rules that determine the interpretation of quantification and constraints in the extended query language of INESS-Search can be summarized as follows:

- **Prenex form**: all quantifiers precede the body of the logical form
- **Existentially quantified** are: $\#$-variables and implicit variables in a positive context; $\%$-variables in a negated context
- **Universally quantified** are: $\%$-variables in a positive context; implicit variables and $\#$-variables in a negated context that are not mentioned in a positive context
- **Default scoping**: universal variables are in the scope of all existential variables by default
- **Explicit scoping**: quantifier scoping can be explicitly specified in prenex form
- **Constraints on variables** are interpreted as restricted quantifiers

One could ask what the practical value of the increased expressiveness of INESS-Search might be. In their survey of treebank query systems, Lai & Bird (2004) list typical queries that a query system should be able to express. Among those queries that are relevant in our setting (Q1–Q5), TIGERSearch is not able to handle Q2 and Q5:

(19) 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.

These queries can easily be expressed in INESS-Search as:

(20) Q2: !(#x:"saw" = #x)

Q5: #c $>*$ #n:NP !$>*$ #v &
#c $>*$ #v:VP !$>*$ #n &
!(#c $>*$ #x $>*$ #n & #x $>*$ #v)

The formulation of Q2 might seem slightly odd at first glance, but its meaning becomes clearer when we look at the corresponding logical form (21), where the constraint is transformed into a restricted quantifier.

(21) Q2: $\forall x$.word($x$,"saw"): $\neg(x = x)$

A tree matches the query Q2 if every node whose word attribute has the value "saw" is not equal to itself. Since $x = x$ is tautologically true for every node instantiation of $x$, this means that the restricted domain defined by word($x$,"saw") must be empty, that is, the tree must not contain any such node.

One might consider introducing a more intuitive abbreviated syntax for Q2, e.g., !"saw".

Full first-order predicate logic is not the most powerful logical system conceivable. Most importantly, transitive closure of binary relations cannot be expressed in first-order predicate logic. Since the transitive closure of some basic relations, notably direct dominance and direct precedence, are of crucial importance in a linguistic querying system, they are normally implemented as basic operators (dominance and precedence).

Other useful complex relations like the c-command relation and the extended-head relation that could hardly be defined efficiently using more basic relations have been implemented in INESS-Search as hard-coded relations.

It is however not possible to define transitive closures of arbitrary ad-hoc relations. Maryns (2009) mentions as an example the transitive closure of the dominance relation PP $>$ NP, which could be used to find arbitrarily long chains of embedded PPs dominating NPs. This query cannot be expressed in first-order predicate logic, but it can be expressed in MonaSearch, which is based on an implementation of Monadic second-order logic. It is not clear to me whether such queries are of great practical importance. MonaSearch, however, cannot be extended to general directed graphs; the tree automata that MonaSearch query expressions are compiled into can only handle proper trees.

INESS-Search is not the only attempt to extend TIGERSearch with universal quantification. In their paper entitled "Extending the TIGER query language with universal quantification", Marek, Lundborg & Volk (2008) point out the lack of expressive power in TIGERSearch and try to outline a design of a universal quantification extension to TIGERSearch. They introduce the notion of a “node set”; variables instantiated by node sets are marked with a %. Marek et al. do not explicitly equate node set variables with universally quantified variables, although their definition makes it clear that the concepts are the same. Unfortunately, by not seeing this equivalence, they also do not see how %-variables interact with negation and implication, and instead try to extend their “node set” notion in a rather complicated way by introducing “subqueries” in order to cope with queries of type Q5.

Marek et al. seem to have partially implemented the “node set” extension in their adaptation of TIGERSearch, whereas “subqueries” are only proposed as an extension. While they state that their approach is easy to implement, they also mention that it is very slow, and they cite the arguments of the developers of TIGERSearch for not having implemented universal quantification:

> 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)

This, however, is a misconception; as I show in the outline of the implementation, the computational complexity introduced by a universally quantified variable is not significantly higher than the complexity originating from existential variables.

# 5 Implementation

INESS-Search is written in Common Lisp. The implementation is heavily inspired by the TIGERSearch implementation, and parts of the query parser are a reimplementation of the code of the Stockholm Tree Aligner (Marek, Lundborg & Volk, 2008).

## 5.1 Static and dynamic indices

In INESS-Search, the various search indices are static and are stored in files on disk. Using the Unix system call *mmap*, those index files are mapped onto virtual memory addresses. Since *mmap* implements demand paging, only those parts (pages) of the index files that are actually needed are loaded into main memory in a lazy manner. This obviates the need for loading the files entirely into main memory, as is done in TIGERSearch.

The treebank index consists of inverted indices for the various features that are represented in the treebank (including *word*, *cat*, *parent-edges* and *child-edges*), and a graph file encoding the graphs of the entire treebank. Whereas the graph file can only be traversed sequentially, the inverted indices allow a quick lookup of all graphs containing a node with a given feature value, and of all nodes with a given feature value. In addition, since the lexicon part of the inverted index is organized as a *suffix array* (Manber & Myers, 1991), sentences and nodes whose feature values satisfy a given regular expression can be looked up equally quickly.[3] This ability to look up all and only those graphs and nodes that satisfy given constraints is crucial in the implementation of an efficient query evaluation strategy.

An alternative to storing the treebank index in static files which is persued in some query tools (e.g., ANNIS2[4]) is to use a relational database. The advantages of a relational database approach are immediate: index lookup and joins are built-in functionality and do not have to be implemented in the tool, and, most importantly, relational databases are dynamic; it is easy to add trees to the treebank index, or to delete trees from it. This flexibility, however, comes at a price. When querying a relational database, there is some overhead connected to keeping track of transactions and concurrency, and to client-server communication. This means in practice that querying a database is potentially much slower than reading from an *mmap*-ed file with a dedicated index structure.[5] On the other hand, as most treebanks that have been constructed so far are quite static in nature, there is little need to change them dynamically.

The LFG treebanks stored in the INESS system are in fact an exception in that respect. Since it is possible to disambiguate the parses of a given sentence in the treebank, an INESS LFG treebank is quite dynamic while it is being constructed. In order to keep the treebank index synchronized with the evolving treebank and make it seem dynamic, the index has been divided into two layers. The main index layer is a static index reflecting the treebank state at the time when the index was generated. In addition, there is an incremental layer which indexes only those sentences that have been added or edited since the main index layer was compiled. It also keeps track of deleted sentences. Since the incremental index is quite small, it can be compiled very fast, and thus can be regenerated every time the treebank changes. To keep the incremental index small, the main index is regenerated offline when the incremental index exceeds a certain size.

## 5.2 Query evaluation strategy

Every INESS-Search query is equivalent to a logical form Q such that all quantifiers are in prenex form, all node constraints are expressed as quantifier restrictions, and the body of the form is a boolean combination of binary relations and predicates. We can assume that the body is normalized, in the sense that it is equal

[3]See Meurer (2012) for a detailed account on the indexing techniques used here.

[4]See http://www.sfb632.uni-potsdam.de/d1/annis/.

[5]Experience from the ANNIS2 project (Rosenfeld, 2010) suggests that this can be compensated for by using a sophisticated indexing strategy, which, however, results in long indexing times and a large on-disk index.

to a disjunction of unions of relations, predicates and negated terms, where each negated term is the negation of a union of relations and predicates.

A query is parsed into an internal representation that is close to the logical form, but where auxiliary node, constraint and value variables are introduced that make it possible to represent the query in a flat form.

A *match* of a query Q($x_1, ..., x_n$) with variables $x_1, ..., x_n$ is a graph Γ together with an instantiation of all the existential variables up to the first universal variable with nodes $X_1, ..., X_k$ from Γ such that Q($X_1, ..., X_k$, $x_{k+1}, ..., x_n)$ evaluates to true.

Let us look at the example query (22), which corresponds to the logical form (23) and has the internal representation (24). The slashes /... / denote a regular expression; plus and minus signs mark whether a variable or value occurs in an existential context.

(22) $($#x:IP %s:S$*$ #y:PROP$)$: #x $>*$ %s $>*$ #y

(23) $\exists x$.cat($x$, 'IP') $\forall z$.cat($z$, /S.$*$/) $\exists y$.cat($y$, 'PROP'): $x >* s$ & $s >* y$

(24)
```
node-order: #x, %s, #y
node-var:   #x, node: [#fc_1] (+)
node-var:   %s, node: [#fc_2] (-)
node-var:   #y, node: [#fc_3] (+)
fc-var:     #fc_1, constraint: cat=#fv_1/+
fc-var:     #fc_2, constraint: cat=#fv_2/+
fc-var:     #fc_3, constraint: cat=#fv_3/+
fv-var:     #fv_1, value: 'IP' (+)
fv-var:     #fv_2, value: /S.*/ (+)
fv-var:     #fv_3, value: 'PROP' (+)
relations:  %s >* #y, #x >* %s
```

A simple-minded algorithm for evaluating a query on a set of graphs (a treebank) would be to go through the graphs one by one, and check for every possible instantiation of the variables (by doing a depth-first traversal of the search space) whether the body of the logical form evaluates to true. This algorithm is actually correct, although not necessarily very efficient, when all variables are existentially quantified.

Some improvements are immediate: We only have to consider graphs that for every quantifier contain nodes that match the node constraints (i.e., that are lying in the domain of the restricted quantifier), and each node variable again only needs to be instantiated with those nodes that match the respective restrictions. As has been shown, finding those candidate graphs and nodes can be done very efficiently by a reverse-index lookup.

The set of candidate nodes can be restricted further by using relation and predicate signatures. For a given relation or predicate, certain types of nodes can be excluded a priori from the set of node candidates. For instance, in the dominance relation $x >* y$, $x$ can only be instantiated by non-terminal nodes, and in the projection relation $c >> f$, $c$ must be a c-structure node and $f$ an f-structure node. The restrictions on the node types of a relation or a predicate is called the *signature* of the relation or predicate. Since the type of a node is coded in the inverted index, the signature information can effectively be used in reverse-index lookup.

When there are universally quantified node variables involved, a correct algorithm is substantially more complex, since it is not sufficient to evaluate the body of the logical form for each instantiation of the variables in isolation. The outline given below is quite close to the actual implementation, although it does not spell out details of the technically rather intricate treatment of dependent disjunctions, negation of unions of relations, and of variable binding and backtracking for value variables.

- Let Q be a query with node variables $x_1$, ..., $x_n$, constraints, predicates and relations.
- Begin by calculating candidate graphs using reverse index lookup for existential constraints up to the first universal variable (in (24): sentences having an IP).
- For each candidate graph $\Gamma$, calculate candidate node sets for each variable that match the constraints (in (24): all IP, S$*$, PROP nodes for $x$, $s$, $y$), or a dummy node for a universal variable if it is not instantiable.

The matches of Q for a given graph $\Gamma$ can be calculated by recursion over the candidate node sets. We first need some definitions:

- A *partial matching tuple* $(X_1, ..., X_i)$ of nodes in $\Gamma$ for some $i \leq n$ is an instantiation of $x_1, ..., x_i$ such that all constraints and relations involving $x_1, ..., x_i$ are satisfied.
- If $x_{i+1}$ is *existential*, then $(\Gamma, X_1, ..., X_i)$ is a *partial match* if $(X_1, ..., X_i)$ is a partial matching tuple and *there is* an instantiation $X_{i+1}$ of $x_{i+1}$ such that $(X_1, ..., X_{i+1})$ is a partial matching tuple.
- If $x_{i+1}$ is *universal*, then $(\Gamma, X_1, ..., X_i)$ is a *partial match* if $(X_1, ..., X_i)$ is a partial matching tuple and *for all* instantiations $X_{i+1}$ of $x_{i+1}$ matching the constraints on $x_{i+1}$, the tuple $(X_1, ..., X_{i+1})$ is a partial matching tuple.
- $(\Gamma, X_1, ..., X_n)$ is a *partial match* if $(X_1, ..., X_n)$ is a partial matching tuple.

Then, $(\Gamma, X_1, ..., X_k)$ is a *match* of Q if $(\Gamma, X_1, ..., X_k)$ is a partial match and $k$ is maximal such that all $x_1, ..., x_k$ are existentially quantified. This includes the case $k = 0$.

It is left as an exercise to the reader to verify that the outlined algorithm is correct.

When the first variable in a query Q is existential and a match does not consist of a graph alone ($k > 0$), there might exist more than one match of Q for the same graph $\Gamma$. The given algorithm will enumerate all such matches. In the search interface however, a lazy evaluation strategy is used: For every graph, only the first match is calculated, which can speed up the calculation of the set of matching graphs considerably. Only when the user inspects a particular graph, the remaining matches for that graph are calculated.

An informal evaluation of INESS-Search against some treebank search systems (i.e., TIGERSearch, MonaSearch and Emdros) based on the TIGER treebank indicates that our system is as fast or significantly faster on most types of queries.

### 5.3 Gorn adressing of directed graphs

In TIGERSearch, node dominance and precedence are coded using Gorn addresses (Gorn, 1967). Each node has a Gorn address, which is an encoding of the path starting from the tree root and leading to the node. In concrete terms, a Gorn address is a sequence of integers, each one telling which child node to chose when traversing the path through the tree.

Using Gorn addressing, dominance and precedence relations are straightforward to check: node X dominates node Y if g(X) (the Gorn address of X) is a proper prefix of g(Y), and X precedes Y if g(X) is alphabetically smaller than g(Y).

This addressing scheme extends easily to directed acyclic graphs. As opposed to trees, there may be more than one path from the root to a given node in a graph. So we simply associate to each graph node the set of Gorn addresses that describe the possible paths from the root to the node. (Note that this addressing scheme assumes that the children of every node are ordered.) With these extended Gorn addresses in place, a graph node X dominates node Y if there is an address in g(X) that is a proper prefix of some address in g(Y).

Determining the Gorn addresses of the nodes in a graph is done by traversing the graph in a depth-first traversal; each step corresponds to one path to the node in focus and contributes to the extended Gorn address of that node.

When we try to extend this algorithm to arbitrary directed graphs, the problem arises that circularity would give rise to infinitely many Gorn addresses, each being a prefix of infinitely many others, since a path can wind arbitrarily often around a cycle. For all practical purposes however, given any two nodes in a cycle, we only need to be able to detect that they dominate each other along that cycle.

A query like (25) that explicitly specifies a double cycle in an f-structure would in fact fail to match that f-structure (e.g., 27), but such queries are quite unintuitive and artificial.

(25) #x >(ADJUNCT $ SUBJ ADJUNCT $ SUBJ) #x

Here is an outline of the algorithm that assigns Gorn addresses in directed graphs with cycles.

- Do a depth-first traversal of the cyclic structure;
- Assign Gorn addresses to nodes as you proceed;
- Stop and backtrack when you detect that **two** assigned Gorn addresses would be prefixes of the new Gorn address. (It is **not** sufficient to stop when one assigned Gorn address is already a prefix of some other assigned Gorn address.)

Consider example (26) and its f-structure in (27). Figure 1 illustrates the Gorn addressing for such a circular f-structure. The boxed numbers are node IDs, and the number sequences below are the calculated Gorn addresses.

(26) *Jagede hunder bjeffer.* “Chased dogs bark.”

(27)
$$\left[\begin{array}{ll} \text{PRED} & \text{‘bjeffe<[8:hund]>’} \\ \text{TOPIC} & {}_{8}\left[\begin{array}{ll}\text{PRED} & \text{‘hund’}\\ \text{ADJUNCT} & \left\{\left[\begin{array}{ll}\text{PRED} & \text{‘jage<NULL, [8:hund]>’}\\ \text{SUBJ} & \left[\mathit{8}\right] \\ \text{VFORM} & \text{pastpart}\end{array}\right]\right\}\end{array}\right] \\ \text{SUBJ} & \left[\mathit{8}\right]\end{array}\right]$$

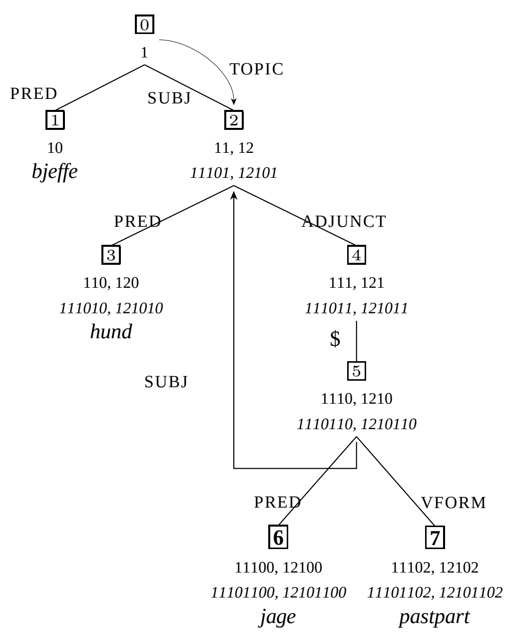


Figure 1: Gorn addressing of a circular f-structure

# 6 Interface and visualization

The INESS-Search tool is an integrated part of INESS, the Norwegian Infrastructure for the Exploration of Syntax and Semantics[6] and can be used to query all treebanks hosted in that infrastructure via a Web interface. In the display of the search results, matching tree/c-structure and sub-f-structures are hightlighted, and the user can choose to see one sub-match at a time, or all possible matches at once. Figure 2 illustrates the display of a match to the query (28) in the German Tiger LFG treebank.

(28) V >>( TNS-ASP TENSE ) "pres"

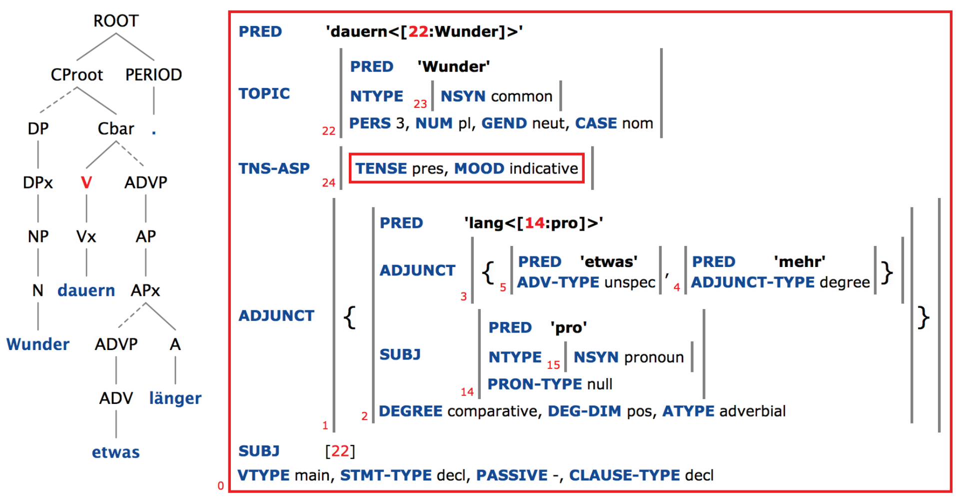


Figure 2: Visualization of a query match

# 7 Future plans

INESS-Search is still work in progress. Even though the basic functionality as described in this article is implemented and stable, there are many conceivable extensions that would make the tool even more useful. Below is a list of those features that will be implemented in the course of the ongoing INESS project.

*Query refinement.* Instead of writing a complex search expression, it is often easier to start with a simple expression and refine it by searching in the set of graphs matching the first expression. Since queries operate on single graphs in isolation, query refinement is well-defined and easy to implement. This stands in contrast to query refinement in a traditional corpus, where the scope of a query expression can span over arbitrarily many corpus positions.

[6]See http://iness.uib.no.

*Search in cross-sentential annotation.* Some linguistic phenomena, such as discourse structure and anaphora resolution, are not restricted to isolated sentences, since they may cross sentence boundaries. In INESS, the PROIEL treebank is an example of a treebank featuring such cross-sentential annotation. With a slight adaptation of the search algorithms and the index layout, INESS-Search will be able to handle cross-sentential search.

*Search in metadata.* Large treebanks often consist of several different analyzed documents, where each document comes with its own set of metadata such as title, author, publishing year, and so on. These metadata have to be searchable in combination with syntactic queries, thus enabling the user to restrict the scope of a query to a subset of the documents.

*Aggregation and export of query results.* For many purposes, it is not sufficient to be able to browse through the matches of a query. One should be able to aggregate the query results in tabular form in order to feed them into a statistics package or the like. The anchor points for aggregation would be the matching graphs and the matching nodes in each graph, and the table entries could be a user-selectable function of the graph and the nodes, such as for instance the node label or the value of any other node feature, or some more complicated expression that can be calculated on a match.

*HPSG support.* Starting with the Redwoods treebank in 2001, quite large treebanks have been compiled in the HPSG framework.[7] To our knowledge, there exists no dedicated query tool for searching in HPSG treebanks. We are planning to adapt the INESS infrastructure and the INESS-Search tool to accommodate HPSG treebanks.

## 8 Acknowledgements

INESS is a project cofunded by the Norwegian Research Council and the University of Bergen.

I would like to thank Victoria Rosén, Koenraad De Smedt and the reviewers for valuable comments and suggestions.

## References

Bresnan, Joan: Lexical Functional Syntax. Blackwell Publishers, 2001.

Dyvik, Helge, Paul Meurer, Victoria Rosén and Koenraad De Smedt: Linguistically Motivated Parallel Parsebanks. In: Passarotti et al. (eds.): Proceedings of the Eighth International Workshop on Treebanks and Linguistic Theories, Milano, 2009, pp. 71–82.

[7]See e.g. http://www.delph-in.net.

Gorn, Saul: Explicit Definitions and Linguistic Dominoes. Systems and Computer Science, Eds. J. Hart & S. Takasu. 77-115. University of Toronto Press, Toronto Canada, 1967.

Haug, Dag Trygve Truslew and Marius Jøhndal. Creating a Parallel Treebank of the Old Indo-European Bible Translations. In: Proceedings of the Sixth International Language Resources and Evaluation (LREC'08), Paris, 2008.

König, Esther, Lezius, Wolfgang, Voormann, Holger: Tigersearch 2.1 User's Manual. Technical report, IMS Stuttgart, 2003.

Kepser, Stephan: Finite Structure Query – A Tool for Querying Syntactically Annotated Corpora. In EACL 2003, Ann Copestake and Jan Hajič (eds.), pp. 179–186.

Lai, Catherine and Steven Bird: Querying and Updating Yreebanks: A Critical Survey and Requirements Analysis. In: Proceedings of the Australasian Language Technology Workshop, Sydney, Australia, 2004.

Lai, Catherine and Steven Bird: LPath$^{+}$: A First-Order Complete Language for Linguistic Tree Query. In: Proceedings of PACLIC 19, 2005.

Manber, Udi and Gene Myers: Suffix Arrays: A New Method for On-line String Searches. In: SIAM Journal on Computing, Vol. 22/5, 1993, pp. 935–948.

Marek, Torsten, Joakim Lundborg and Martin Volk: Extending the TIGER Query Language with Universal Quantification. In: KONVENS 2008: 9. Konferenz zur Verarbeitung natürlicher Sprache, Berlin, pp. 5–17.

Maryns, Hendrik and Stephan Kepser: MonaSearch – A Tool for Querying Linguistic Treebanks. In: Proceedings of the 7th International Workshop on Treebanks and Linguistic Theories (TLT 2009), Groningen, Netherlands, 2009.

Meurer, Paul: Corpuscle – A New Corpus Management Platform for Annotated Corpora. In: Gisle Andersen (ed.): Exploring Newspaper Language, Studies in Corpus Linguistics 49, Benjamin, 2012.

Petersen, Ulrik: Evaluating Corpus Query Systems on Functionality and Speed: TIGERSearch and Emdros. In: Angelova et al. (eds): International Conference on Recent Advances in Natural Language Processing 2005, Proceedings, Borovets, Bulgaria, 21-23 September 2005, pp. 387–391.

Rosén, Victoria, Koenraad De Smedt, Paul Meurer and Helge Dyvik: An Open Infrastructure for Advanced Treebanking. In: META-RESEARCH Workshop on Advanced Treebanking at LREC2012, İstanbul, 2012, pp. 22–29.

Rosenfeld, Victor: An Implementation of the Annis 2 Query Language, Master thesis, Humboldt University Berlin, 2010.

Volk, Martin, Joakim Lundborg and Maël Mettler: A Search Tool for Parallel Treebanks. In: Proceedings of the Linguistic Annotation Workshop, 2007, pp. 85–92.