Using the Syntactic Reference Corpus of Medieval French in TIGERSearch (SRCMF)

T. M. Rainsford, September 2015

Introduction

This tutorial provides a step-by-step guide to working with the Syntactic Reference Corpus of Medieval French (SRCMF) using the TIGERSearch query engine preferred by the project. It is intended for those who are familiar with syntactic analysis and the concept of treebank corpora.

The document is divided into four main chapters:

1. **Getting started**. Covers installation of the software (1.1), the basic structure of the syntactic annotation in the SRCMF (1.2) and the basics of the TIGERSearch query language (1.3)
2. **Writing successful queries**. Tips for writing effective queries to identify structures in the the SRCMF.
3. **Exporting the results as a KNIC concordance**. How to export the results of your searches in a tabular format.

This is a user guide intended to help new users start using the corpus, and is not a comprehensive reference manual. Those looking for a reference manual are referred to the following resources:

- The TIGERSearch manual (König, Lezius, and Voormann 2003), online at http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/TIGERSearch/manual.html
- The SRCMF annotation manual (in French), online at http://www.srcmf.org/fiches/index.html
- The Cattex annotation manual (part-of-speech tagset, in French) (Guillot, Prévost, and Lavrentiev 2013), online at http://bfm.ens-lyon.fr/spip.php?article323

1. Getting started

1.1 Installation

1.1.1 Installing TIGERSearch

You will need to install the TIGERSearch corpus query engine on your computer.

- ZIP archives containing both Windows and Mac versions are available on www.srcmf.org in the “Tools” section of the website (recommended)
- Alternatively, you can download the program from the TIGERSearch homepage (www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/tigersearch.html). This version does *not* include a Java Runtime Environment, which you will have to install separately.

1.1.2 Installing the SRCMF

The SRCMF corpus is available in the “Access” section of www.srcmf.org. You have the choice of downloading the texts as a single corpus, or individually. Each text is available in three formats:

- a TIGERSearch binary (tsbin) (recommended);
- a TIGER-XML file (tsxml): this must be processed using the TIGERRegistry program before it can be used with TIGERSearch;
- a RDF file (rdf): not compatible with TIGERSearch.

Note that the full corpus and some of the individual texts are subject to licensing restrictions: full details are provided on the website. For the purpose of this tutorial, we will be working with the *Yvain* text, which is available without any licensing restrictions.

To download and install *Yvain*:

➔ click on the “tsbin” link next to *Yvain de Chrétien de Troyes* in the list of texts, and save the zip archive to your computer.

➔ extract the folder in the archive to the folder “CorporaDir” in your TIGERSearch directory.

That’s it! The corpus will be available to use when you start TIGERSearch.

1.2 The TIGERSearch interface

This section covers basic use of the TIGERSearch interface. Users experienced with TIGERSearch may wish to skip to section 1.3.

1.2.1 Opening a corpus

To begin, launch TIGERSearch:

➔ Windows: double-click “tigersearch.exe” in the “bin” folder of your TIGERSearch installation.

➔ Mac: launch “runTS.command” in the “lib” folder of your TIGERSearch installation.

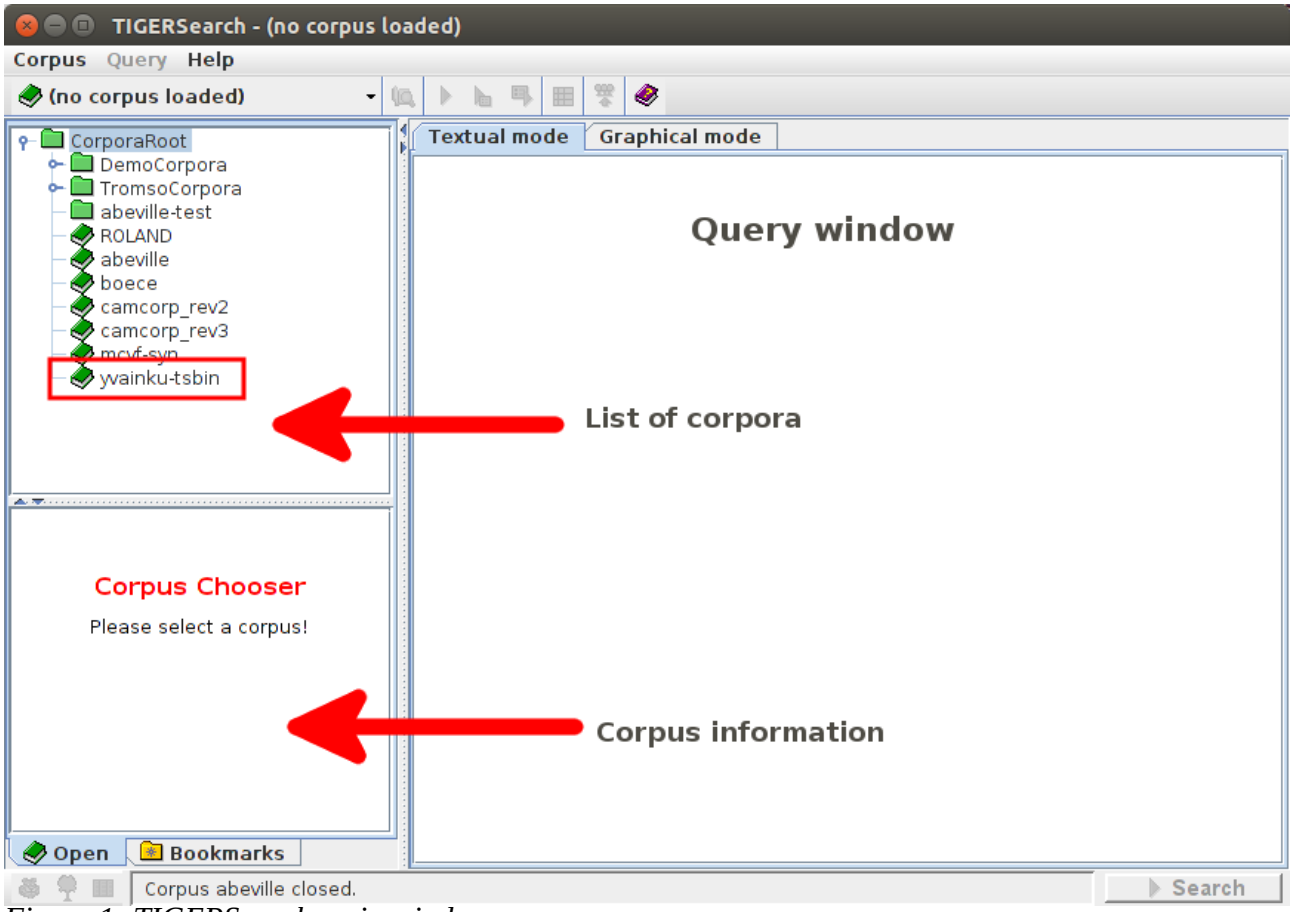

Figure 1: TIGERSearch main window

Once launched, you will see the TIGERSearch main window (see figure 1), which contains three main panels:

- a list of installed corpora (top left). The *Yvain* text (yvainku-tsbin) will be on this list;
- information about the currently open corpus (bottom left);
- a query window (right).

Double-click “yvainku-tsbin” in the corpus list to open the *Yvain* text.

1.2.2 Browsing the corpus: TIGERGraph Viewer

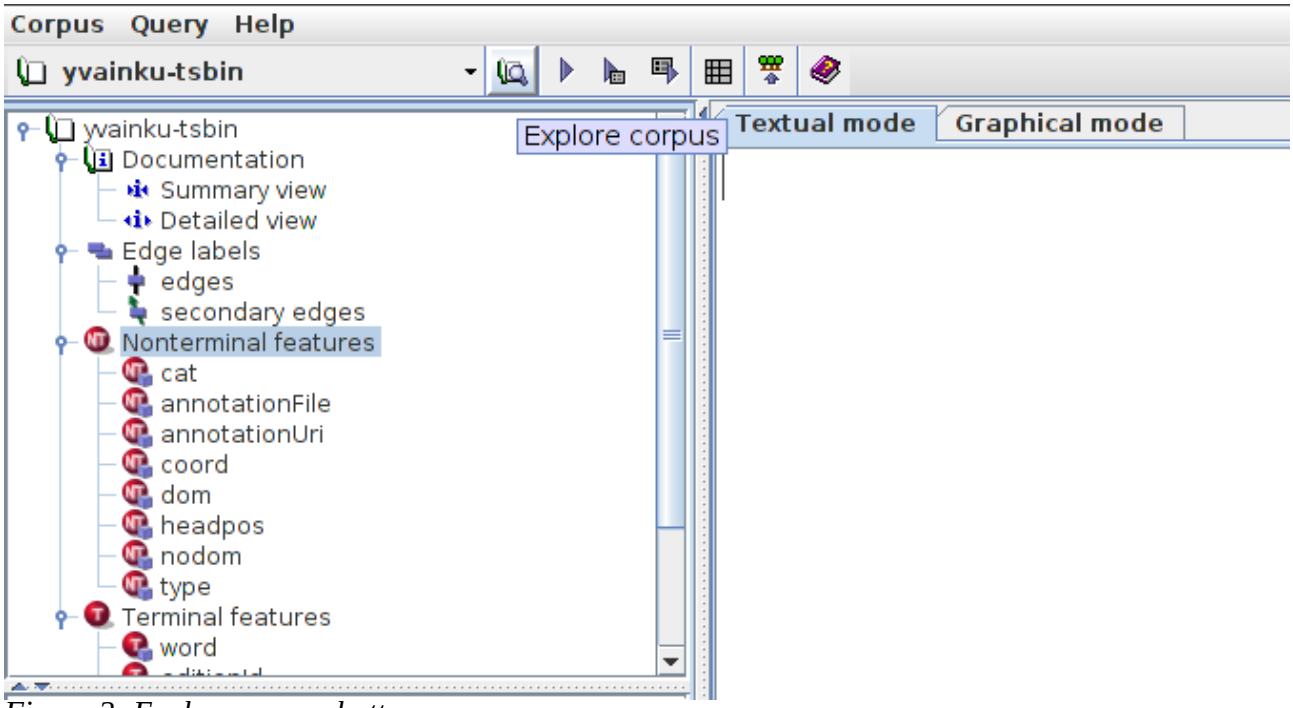

Figure 2: Explore corpus button

When opening a corpus for the first time, it's often helpful simply to browse the first few annotated sentence to get an idea of how it has been annotated. Once the *Yvain* corpus is open, click on the "Explore Corpus" button in the toolbar (see figure 2). This will open the Graph Viewer.

1.2.2.1 Changing the graph viewer display options

We can change the node features which are visible in the graph viewer by editing the Display options.

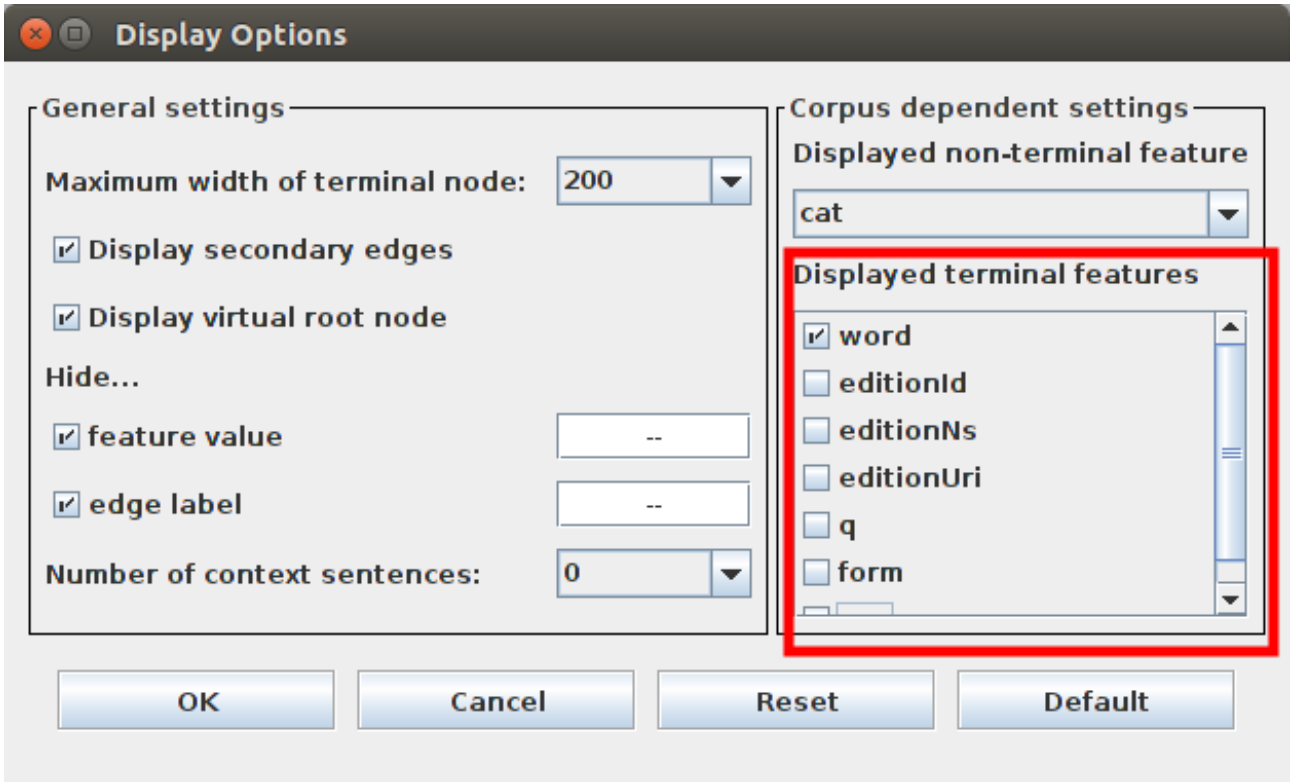

Figure 3: Display options

➔ Click “Options > Display Options” to open the Display Options dialogue box (see fig. 3).

➔ Select “word” and “pos” only in the list “Displayed Non-terminal features”.

You may also wish to increase the number of preceding/following sentences displayed below the graph: this can be adjusted with the “Number of context sentences” option.

2.1.2.2 A sample sentence

The first sentence in *Yvain* is rather long, so we’ll start with the second.

➔ Click the “Next >” button in the navigation panel at the bottom of the Graph Viewer to advance to the second sentence.

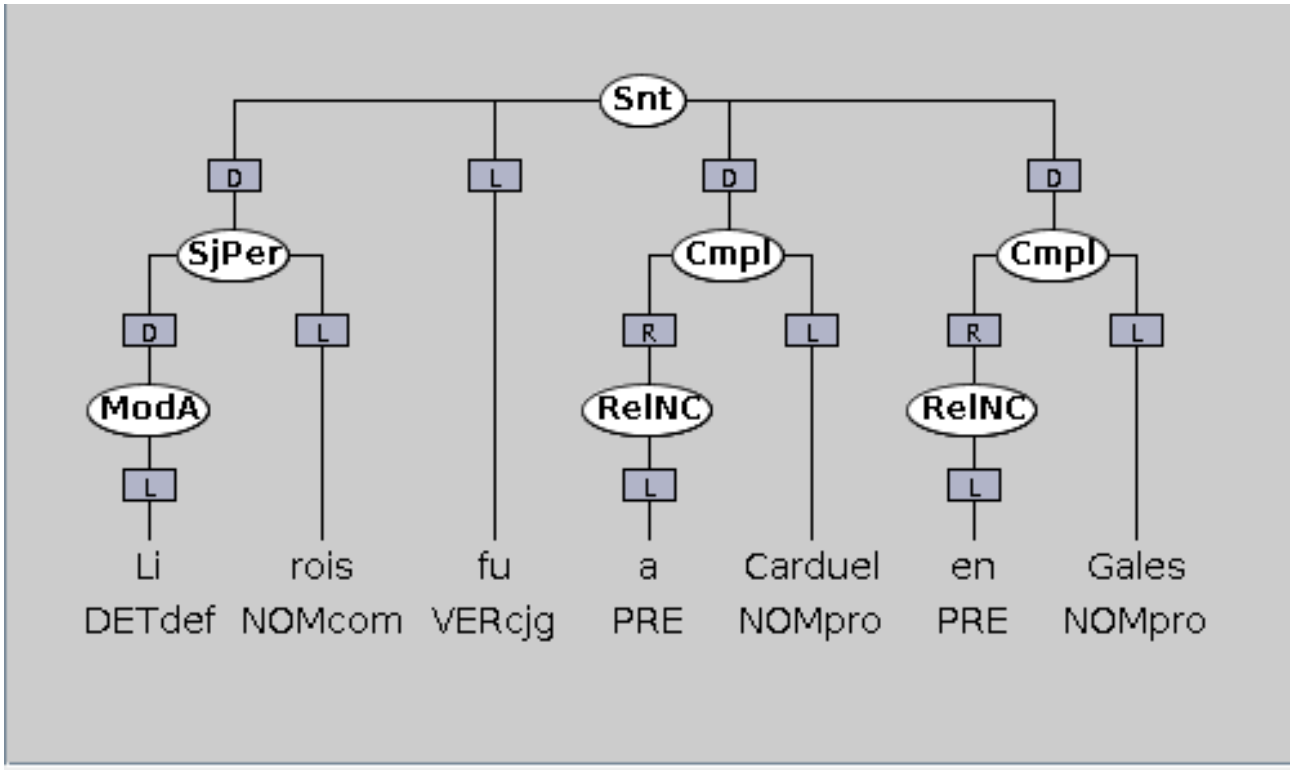

Figure 4: Yvain, second sentence

You should see the sentence tree in figure 4, for the following Old French sentence:

(1) Li rois fu a Carduel en Gales
“The king was at Carduel in Wales.”

The words appear along the bottom of the graph, with part-of-speech tags underneath (e.g. *rois* is tagged as **NOMcom** (common noun)). Syntactic functions are given in the white ovals within the graph itself (e.g. the group *Li rois* is tagged as **SjPer** (personal subject)).

1.2.2.3 Nodes and edges

To use the correct terminology, we can describe each TIGERSearch graph in terms of NODES and EDGES. There are two types of nodes in this sample graph:

- TERMINAL NODES: these always appear along the bottom line of the graph, and contain the words in the text. Terminal nodes, as the name suggests, have no “child” nodes.
- NON-TERMINAL NODES: these are represented by the white ovals. Non-terminal nodes *must*

have at least one child node.

Each node has a number of FEATURES. For instance, terminal nodes have the features “word” (word form) and “pos” (part-of-speech), which we are currently viewing, and a number of others which are not currently being shown in the graph. However, in the SRCMF, terminal and non-terminal nodes do not have the same features: for example, syntactic function tags such as **SjPer** (personal subject) or **Snt** (sentence) are stored as the “cat” feature of non-terminal nodes.

➔ Hover your mouse pointer over any node in the graph to see a full list of features.

The nodes are connected by EDGES. These are represented by a line connecting the two nodes. Edges in the SRCMF are labelled: this appears as a single letter (**L, D**) in a grey box on the line.

1.2.2.4 Secondary edges

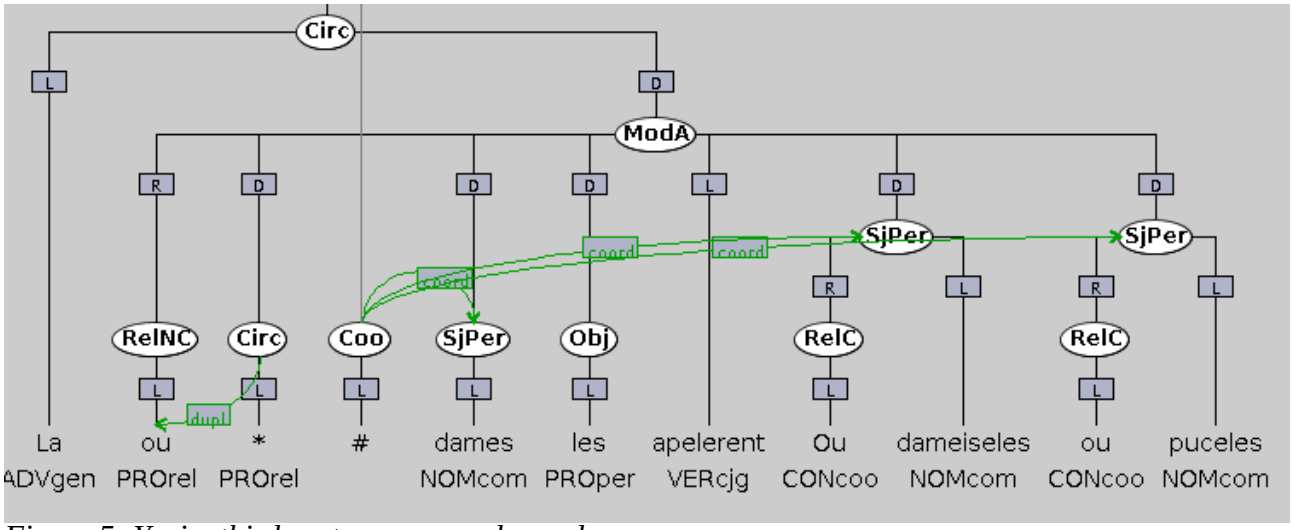

Figure 5: Yvain, third sentence, secondary edges

Let’s briefly advance to the third sentence in the *Yvain* text by clicking the “Next >” button. You’ll notice that this graph contains some green lines in the second half of the sentence (see fig. 5), which reads as follows:

(2) La ou dames les apelerent Ou dameiseles ou puceles
“Where ladies, damsels or maidens called them.”

The green lines are known as SECONDARY EDGES, and link nodes together in a non-hierarchical manner. There are two main uses of secondary edges in the SRCMF: in coordinate structures, and to mark words with a double syntactic function. In this case, you’ll see that the three conjuncts which make up the subject of the clause are linked to a node labelled **Coo** by secondary edges. We’ll return to this in more detail in section 1.3.2.2.

1.2.2 Viewing corpus information

➔ Close the Graph Viewer window to return to the TIGERSearch main window.

Now the *Yvain* text is open, the left-hand panels of the main window provide useful information about the annotation used in the corpus. In the top-left panel, you’ll see a list of the elements that make up a TIGERSearch graph: edges, non-terminal nodes and terminal nodes. A complete list of features is provided for each type of node.

➔ Click on the “cat” feature listed under “Nonterminal features”.

Clicking on some features (such as the “cat” feature) will bring up a *complete list of all possible values* of that feature in the lower left panel, including a gloss (in French). In the SRCMF, this information is provided for all features which form a closed class.

➔ Click on “Detailed view” listed under “Documentation” in the top-left panel.

The detailed view of the corpus information gives a easily-readable list of all information provided in the corpus header, including feature lists, number of tokens, and number of graphs in the corpus.

1.2.3 A quick corpus query

1.2.3.1 Launching a query

Corpus queries are written in the main (right-hand) panel of the TIGERSearch main window. Writing queries for the SRCMF will be covered in more detail in section 3.

➔ Copy the following text in the query window.

```
[word="rois"]
```

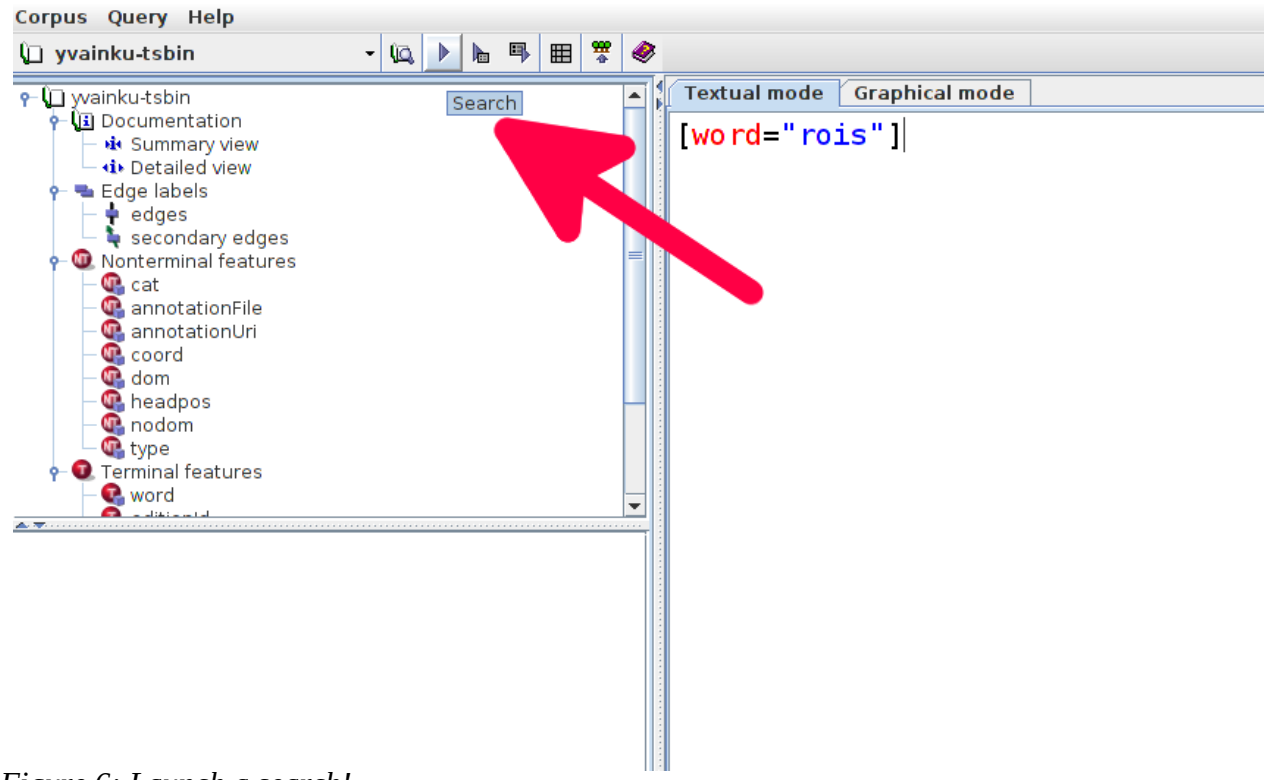

Figure 6: Launch a search!

You’ll see colour-coding appear in the search window: the feature name “word” is red and the value is blue.

➔ Click the “> Search” button in the bottom-right of the screen, or the “play” arrow in the toolbar (see fig. 6)

1.2.3.2 Results: graphs and subgraphs

The Graph Viewer will now pop up, but instead of showing the whole corpus, it will only show those graphs which match the query: in this case, all those containing the word *rois* “king(s)”. The matching node is highlighted in red.

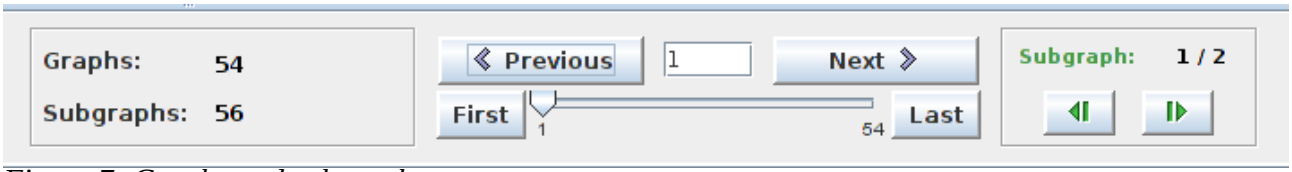

Figure 7: Graphs and subgraphs

The navigation panel below the graph contains some useful quantitative information about the query results (see figure 7), in particular the number of matching GRAPHS and SUBGRAPHS. It is very important to understand what these numbers mean!

- "Graphs: 54". There are 54 graphs in the corpus which contain a match for this query (i.e. 54 sentences contain the word *rois*).
- "Subgraphs: 56". There are 56 structures which match this query in the corpus (i.e. there are 56 tokens of the word *rois*).

The difference between the two figures tells us that one or two sentences contain *more than a single token* of the word *rois*. The first matching sentence is one such case. Note the green "Subgraph" box at the right of the navigation panel:

- "Subgraph: 1 / 2". There are two structures in this graph matching the query; the first is highlighted.

By clicking on the green forward arrow, the red highlight in the graph shifts to the second of the two matching subgraphs.

You can use the "< Previous" and "Next >" buttons in the navigation panel to view the other 53 matching graphs one by one.

1.2.3 Further reading

We have now covered the basics of the TIGERSearch interface. More details can be found in the TIGERSearch manual, available on the project homepage:

http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/tigersearch.html

Some users may be interested in exploring TIGERSearch's graphical interface for writing queries: full details may be found in the TIGERSearch manual.

1.3 The SRCMF in TIGERSearch

The SRCMF grammar model will not be presented in full in this tutorial. Readers are instead referred both to Prévost and Stein [REF] and to the SRCMF annotation guide (http://www.srcmf.org/fiches/index.html). We assume moreover that readers are familiar with the basics of dependency grammar.

The purpose of this section is twofold: firstly, to highlight some of the less canonical aspects of the SRCMF grammar model, which may be unfamiliar even to readers who have worked with dependency corpora before, and secondly, to document how the SRCMF grammar model is represented in TIGERSearch.

1.3.1 Dependency in the SRCMF treebank

1.3.1.1 A dependency treebank

The SRCMF is a dependency treebank:

- the main clause finite verb is the head of the sentence (exception: “non-sentences”);
- with the exception of the head of the sentence, each individual word “depends” on one other word in the sentence.

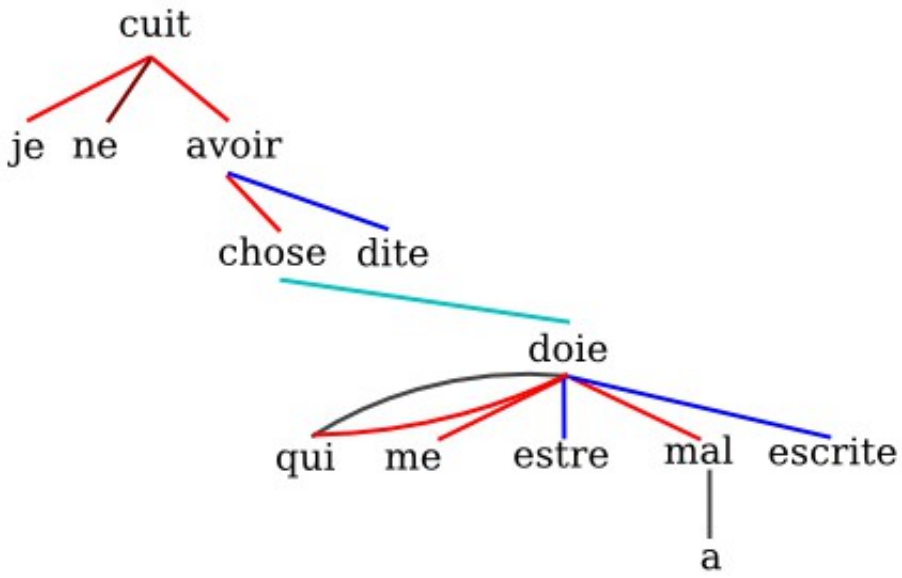

Figure 8: SRCMF dependency tree

Figure 8 shows the following sentence represented using a dependency graph which closely reflects the grammar model of the SRCMF:

(3) Je ne cuit avoir chose dite qui me doie estre a mal escrite
“I do not think I have said anything that should be held against me.”

1.3.1.2 Head–dependent directionality

Since in a number of cases there is no clear consensus among linguists as to the “correct” directionality of head–dependent relations, it is worth summarizing some of the key points of the linguistic analysis adopted in the SRCMF:

- All clause-level structures (arguments, adjuncts, and non-arguments such as parentheticals) depend on the finite verb.
- Subordinating conjunctions depend on the finite verb in the subordinate clause, they are not heads (*cf. qui me doie...* above).
- Prepositions depend on the non-finite verbal or non-verbal head of the following structure, they are not heads (*cf. a mal* above).
- In compound verb tenses (e.g. the perfect tense, the passive), the finite auxiliary verb is analysed as the head of the clause on which all other arguments and adjuncts depend (*cf. avoir dite* above). The past participle is a childless dependant of the finite verb.
- In constructions involving a modal verb (*pouvoir, devoir, savoir*), the finite modal verb is

analysed as the head of the clause on which all other arguments and adjuncts depend. The infinitive is a childless dependant of the finite verb. (*cf.* the flat structure of *doie estre [...] escrite* above).

- Determiners are treated like adjectives: they depend on the nominal element which they determine.

1.3.1.3 Departure from conventional dependency structure

Two structural features of the SRCMF call for particular comment, as they do not occur in traditional dependency corpora.[1]

1. A single node may have *more than one* parent, or indeed have *more than one* dependency relation with the *same* parent. In the example above, the word *qui* "who" has two separate dependencies on the head of the clause *doie* "should", since it is both the subject of the clause and a subordinating conjunction.
2. Coordination is not represented within the dependency structure. Coordinated nodes are instead "grouped" together, and this non-hierarchical grouping complements hierarchical dependency relations.

Due to the "more than one parent" approach, the SRCMF is not obliged to split word-forms containing an enclitic determiner (e.g. *del, al, des, as*) or an enclitic pronoun (e.g. *nel, nes, jel*), which necessarily have a double function. The Old French word division conventionally adopted in print edition is therefore observed throughout the corpus.

1.3.2 Representing dependency in TIGERSearch: node pairs, structures and heads.

The distinction made between terminal and non-terminal nodes in TIGERSearch (see §1.2.2.3) reflects the fact that it was also designed for constituency-based annotation, in which words (terminal nodes) are grouped into constituents (non-terminal nodes). While such a distinction is irrelevant in the SRCMF, it has had to be followed in order to make the corpus compatible with TIGERSearch.

Consequently, each word in the SRCMF dependency structure is represented by a NODE PAIR in the TIGERSearch graph:

- a terminal node, with features denoting the word-form "word" and the part-of-speech "pos";
- a non-terminal node, with a feature "cat" denoting the syntactic function of the word relative to its parent.

The two nodes are linked by an edge labelled **L** for "lexical form".

The dependency relations forming the hierarchical syntactic structure are marked between the non-terminal nodes of the node pair, and are labelled **D**. Prepositions and conjunctions are analysed as *relators*, and the relation between these and their governing node is labelled **R**.

As a general rule, the non-terminal node of a node pair contains *structural* information (e.g dependencies and syntactic function), while the terminal node contains only *word-level* features. For this reason, we will henceforth adopt the term STRUCTURE NODE to denote the non-terminal node of a node pair and the term WORD NODE to denote the terminal node.

1 Indeed, these features are incompatible with the standard CoNLL format used by dependency parsers.

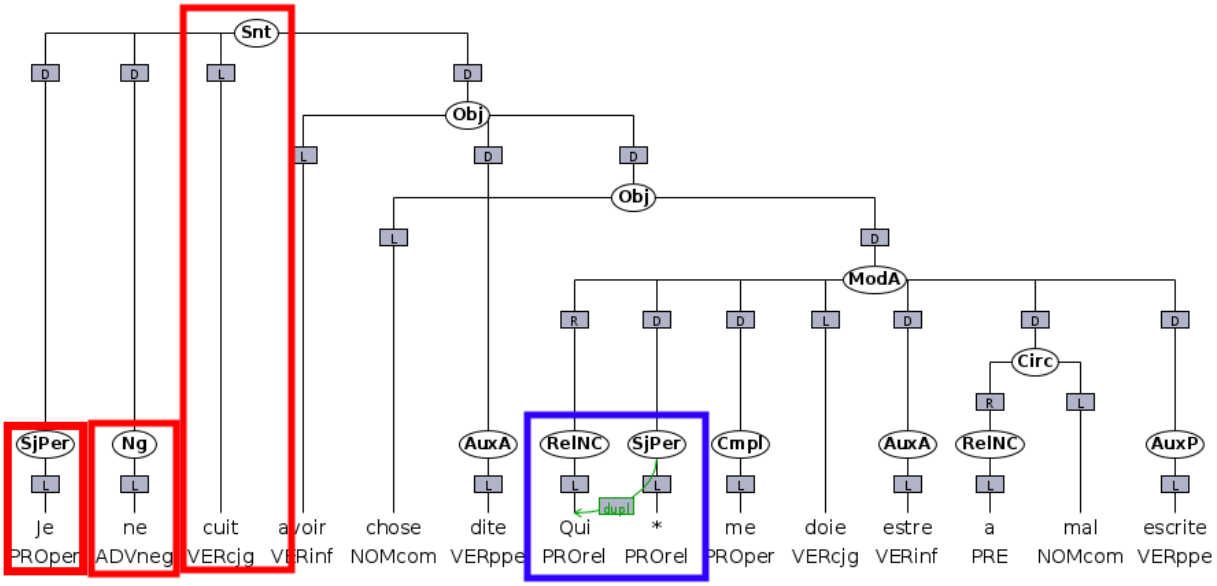

Figure 9: SRCMF corpus in TIGERSearch

Figure 9 shows the TIGERSearch graph for sentence (3). The node pairs representing the first three words are highlighted in red.

1.3.2.1 Representing nodes with two functions

The word *qui* in (3) has two separate functions. The representation of this structure in the TIGERSearch graph is highlighted in blue:

- The word *qui* is split into two node pairs.
- The first node pair links the word node *qui* with the first of its two functions, ordered alphabetically.
- The second node pair links the second of the two functions with a dummy word "*".
- A secondary edge labelled **dupl** links the structure node of the second node pair to its "true" word node, the terminal node *qui*.

1.3.2.2 Representing groups in coordination

Let us return briefly to example (2) and figure 5. Here, the subject is made up of three separate conjuncts: *dames*, *ou dameiseles* and *ou puceles*. In the TIGERSearch representation:

- Each conjunct has a separate dependency **SjPer** on the finite verb *apelerent*.
- The three conjuncts are linked to a non-terminal node labelled **Coo** by three secondary edges labelled **coord**.
- The node **Coo** is entirely outside the dependency hierarchy (it has no parent node other than the automatically generated VROOT element), and is paired with a dummy word node "#".

1.3.2.3 Complex coordination

Some cases of coordination involve more than a single conjunct. Consider example (4):

(4) Que je fui plus petiz de lui et ses chevax miaudres del mien.
"For I was smaller than he and his horse better than mine."

The SRCMF analysis of gapping constructions such as these is as follows:

- The arguments of the gapped clause (i.e. *ses chevax* “his horse” and *miaudres del mien* “better than mine”) depend on the expressed verb of the first clause.
- The arguments of the first clause (i.e. *je* “I” and *plus petiz de lui* “smaller than he”) and the arguments of the second clause constitute two COMPLEX CONJUNCTS.

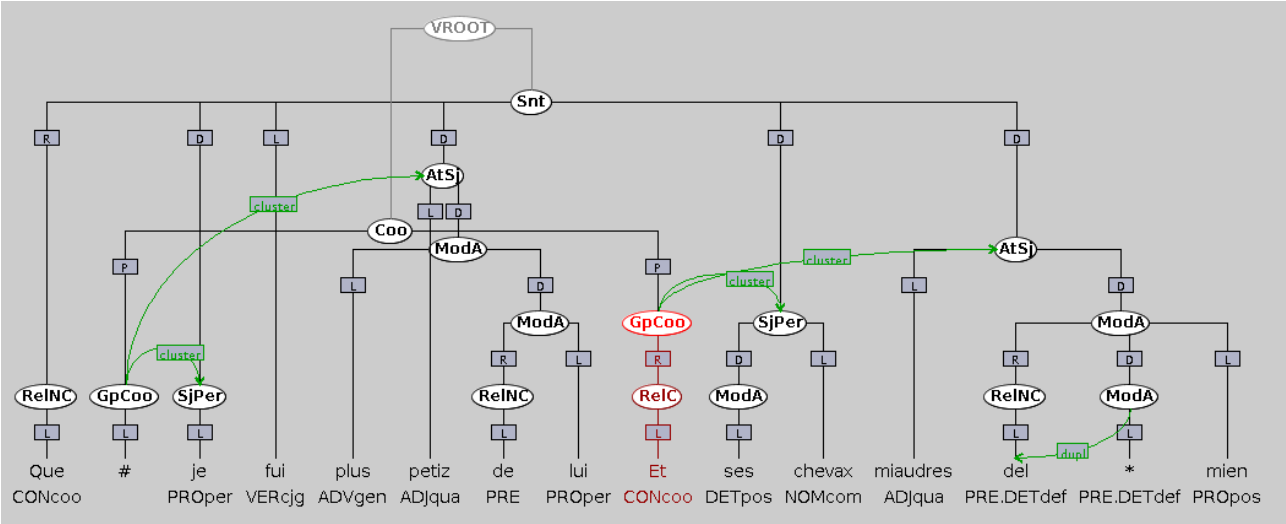

Figure 10: Complex coordination

The representation of this complex coordination structure in TIGERSearch is given in figure 10.

- The two complex conjuncts are coordinated.
- Each argument in both clauses has a dependency relation to the finite verb *fui*.
- The complex conjuncts are represented by two nodes labelled **GpCoo**, one linked to *je* and *plus petiz de lui*, and the other to *ses chevax* and *miaudres del mien*, by secondary edges labelled **cluster**.
- The **GpCoo** nodes are children of a **Coo** node which, as in the previous example, is outside the dependency structure.
- The coordinating conjunction *et* is a child of the second **GpCoo** node.

1.3.3 Further reading

This section has covered the basic structures of the SRCMF as found in TIGERSearch.

For more information on the SRCMF dependency grammar model, see Prévost and Stein [REF]. For a linguistic justification of the coordination analysis, see Mazziotta [REF].

For more information on the syntactic tagset and the analysis of specific structures, please see the SRCMF annotation guide (http://www.srcmf.org/fiches/index.html, in French).

1.4 The TIGERSearch query language

This section provides a brief introduction to the syntax of the TIGERSearch query language for the SRCMF.

1.4.1 Basics

A TIGERSearch query defines a particular subgraph that we wish to search for. The subgraph may contain a single or multiple nodes, and makes use of a range of wildcards.

The two most important aspects of a TIGERSearch query are therefore:

1. Node definitions
2. Relations between nodes

1.4.2 Node definitions

Node definitions consist of two parts:

- A node identifier
- A feature constraint

Either part may be omitted in a query.

1.4.2.1 Feature constraints

Feature constraints consist of one or more feature-value pairs enclosed in square brackets, for example:

```
[word="rois"]

[cat="SjPer"]

[word=/roi.*/ & pos="NOMcom"]
```

As we saw earlier, the query [word="rois"] will find all instances of the word **rois**. Put technically, it matches all nodes whose “word” feature is **rois**. This matches the word node of a node pair, as the “word” feature is on the terminal node. The query [cat="SjPer"] matches all nodes whose “cat” feature is **SjPer**: i.e. find all subjects. This matches the structure node of the node pair, since this has the “cat” feature.

Multiple properties can be combined in a node definition. The query [word=/roi.*/ & pos="NOMcom"] matches all nodes whose “word” feature matches the regular expression[2] "roi.*" and whose “pos” feature is "NOMcom": i.e. all nouns beginning "roi".

To express feature values, observe that exact values (strings) are given between double quote marks (" "), while regular expressions are given between forward slashes (/ /).

When combining two feature constraints, two operators are available:

- **&** (AND): both feature-value pairs match, e.g. [word=/roi.*/ & pos="NOMcom"]
- | (OR): at least one of the feature-value pairs match, e.g. [cat="SjPer" | cat="SjImp"] will find tokens of both personal and impersonal subjects.

It is also possible to negate a feature-value pair using the operator **!=** “does not match”. For example, to match instances of the word *les* other than as a common grammatical form (definite

2 This tutorial will not cover regular expression syntax, as it is neither specific to TIGERSearch nor to the SRCMF corpus. This does not imply that they are not useful! A good online tutorial may be found at http://www.regular-expressions.info, or see the TIGERSearch manual, sec. 3.5.

determiner and personal pronoun):

```
[word="les" & pos!="DETdef" & pos!="PROper"]
```

1.4.2.2 Node identifiers

A node identifier is a user-defined name attributed to a particular node in the query. It is introduced by the hash symbol.

```
#n

#n:[word="rois"]
```

The node identifier alone (i.e. #n) has no feature constraints. If entered as a query, it will match every node in every graph in the corpus! To combine a node identifier with a feature constraint, the colon (**:**) is used as a separator. The query #n: [word="rois"] returns identical results to [word="rois"], with the difference that each matched node is also assigned the label **n**.

1.4.3 Node relations

There exist two core node relations in a TIGERSearch graph:

- precedence, expressed by the dot operator (**.**)
- dominance, expressed by the "greater than" operator (>).

1.4.3.1 Precedence relations

Precedence relations are used to query the order of terminal nodes (i.e. words)[3] in the query. For example, if we wish to find all tokens of *li rois* "the king", we need to use the following query:

```
[word="li"] . [word="rois"]
```

Here, two nodes are defined using two separate feature constraints. The relation between these two nodes is defined as "**.**": direct precedence. This will match all tokens of *li* directly preceding *rois*.

There exist a number of variants of the precedence operator, of which the most useful is "**.***": precedes at any distance. For example, if we wish to find tokens of negated verbs, we could try to use the following query:

```
[pos="ADVneg"] .* [pos="VERcjg"]
```

The query identifies all cases of negative *ne* (pos tag **ADVneg**) preceding a finite verb (pos tag **VERcjg**). "Precedence at any distance" ensure that cases in which clitic pronouns intervene are counted along with cases in which *ne* is directly before the verb. However, this query also returns a lot of noise, since there is at present nothing to restrict the words to the same clause.

1.4.3.2 Dominance relations: querying edges

A dominance relation is used to query relations between nodes represented by EDGES in the TIGERSearch graph. A simple application of this operator in the SRCMF corpus is to associate a structure node with its word node (*cf.* §2.2.2). Suppose we wish to find every instance of the word *rois* as a subject. The syntactic function is marked on the structure node, but the lexical information

3 TIGERSearch can also calculate precedence between non-terminal nodes based on the relative position of the leftmost terminal node dominated. We do not recommend using this implicit "shortcut" in the SRCMF corpus, since it is not sensitive to the SRCMF "node pairs".

is marked on the word node:

```
[cat="SjPer"] >L [word="rois"]
```

Our feature constraints identify two nodes: a structure node with subject function, and a word node *rois*. The dominance relation ">L" specifies that an edge labelled **L** exists linking both nodes, with the structure node dominating the word node: i.e. we require these two nodes to form a node pair. (Try querying "[cat="SjPer"] & [word="rois"]" to see why we need to specify the relationship between the two nodes!)

Secondary edges can also be queried, using the operator ">~". For example, to find all tokens of words with a double function (see §2.2.2.1), we may search for secondary edges labelled **dupl**:

```
[] >~dupl []
```

Here, we use an empty feature constraint ([]) to define the two nodes, since we simply wish to find *all* instances of the secondary edge labelled **dupl** without any restrictions on the nodes that it links together.

There exist a number of variants of the dominance operator, but few are essential for beginners. See the TIGERSearch manual (ch. 3).

1.4.3.3 Negated operators: a word of caution

A glance at the TIGERSearch manual reveals that it is possible to negate both dominance and precedence operators, e.g.:

- **!.***: "does not precede"
- **!>**: "does not dominate"

However, these are confusing, since they *only negate the relationship, not the existence of the nodes*!

For example, the following query does *not* find every null subject main clause (try it!):

```
[cat="Snt"] !> [cat="SjPer" | cat="SjImp"]
```

Instead, it will return graphs containing a subject somewhere other than in the main clause. There may well be a subject in the main clause too.

These operators are occasionally useful in very specific circumstances, but should be avoided by beginners!

1.4.3.4 Putting it all together

Most queries are made up of a number of node definitions and node relations joined together with the & (AND) or the | (OR) operator.

Let us return for a moment to our (unsatisfactory) query to find negated verbs:

```
[pos="ADVneg"] .* [pos="VERcjg"]
```

We need to add a requirement which states that these two nodes must be within the same clause. In graphical terms, we need to state that the structure node for the finite verb dominates the structure node for the negation.

```
#negword:[pos="ADVneg"] .* #verbword:[pos="VERcjg"]
& #verbstructure >L #verbword
& #negstructure >L #negword
& #verbstructure >D #negstructure
```

Three node relations are added to the query: two which identify the structure nodes “verbstructure” and “negstructure” from their word nodes, and one which defines the relationship between these two structure nodes. The four node relations are conjoined by the ampersand (**&**).

Note the importance of node identifiers, which enable us to refer to the same node multiple times within a query.[4]

2.3.4 Further reading

This section has covered the basic syntax of TIGERSearch queries. For further information, refer to the TIGERSearch manual (ch.3), and in particular the quick reference (ch. 3, §12).

4 Note also that this is not the simplest way to search for negated verbs!

2. Writing successful queries

This chapter is intended for users with a basic knowledge of:

- the structure of the SRCMF corpus in TIGERSearch (see §1.3)
- TIGERSearch query syntax (see §1.4)

It aims to explain and to provide models for common types of query.

2.1 General structures

2.1.1 "How do I find all words *x* with syntactic function *y*?"

E.g. pronominal objects, the word *Yvain* as subject

These are very straightforward queries. Use feature constraints to identify the node. In some cases, both a structure node and word node may need to be defined.

- Pronominal objects

```
[cat="Obj" & headpos=/.*PROper/]
```

- *Yvain* heads the subject:[5]

```
[cat=/Sj.*/] >L [word=/Yvain[sz]?/]
```

2.1.2 "How do I find *x*s that dominate a *y* (and a *z*)?"

E.g. verbs with subject and object, nouns with a relative clause, infinitives introduced by a preposition.

These are straightforward queries. Use feature constraints to define each node, and dominance relations to specify the relations between them.

- Finite verbs with subject and object:

```
#verb:[headpos="VERcjg"] >D [cat=/Sj.*/]
& #verb >D [cat="Obj"]
```

- Nouns governing a relative clause:

```
[headpos=/NOM.*/] >D [cat="ModA" & headpos="VERcjg"]
```

- Infinitival clauses introduced by a preposition (recall from §2.2.2 that the edge linking a governing node to a preposition is labelled **R**).

```
[headpos="VERinf"] >R [headpos=/PRE.*/]
```

In some cases, e.g. when interested in a particular lexical form, we may need to query a property of the word node too. For example, if we are searching for all cases in which the word *Yvain* occurs as the object of a finite verb:

5 Note that when *Yvain* is preceded by a title (e.g. *messire Yvain*), it is the title that heads the subject.

```
#object:[cat="Obj"] >L #objectword:[word=/Yvain[sz]?/]
& #verb:[headpos="VERcjg"] >D #object
```

2.1.3 "How do I find all *x*s that precede a *y*?"

E.g. verb-subject inversion, pre-nominal adjectives

These are relatively straightforward queries. Use the "precedes at any distance" operator (**.***) in combination with a definition of the dependencies.[6]

- Post-verbal subjects:

```
#verbword:[pos="VERcjg"] .* #subjectword
& #subject:[cat=/Sj.*/] >L #subjectword
& #verb >L #verbword
& #verb >D #subject
```

- Pre-nominal adjectives:

```
#adjword:[pos=/ADJ.*/] .* #nounword:[pos=/NOM.*/]
& #noun >L #nounword
& #adj:[cat="ModA"] >L #adjword
& #noun >D #adj
```

2.1.4 "How do I find all *x*s that don't dominate a *y*?": the *dom* feature

E.g. verbs without subjects, nouns without determiners, etc.

It is impossible in a TIGERSearch query to require the *non-existence* of a particular kind of node (see §1.4.3.3). However, the SRCMF includes a feature to make the most common requests of this type possible.

If *y* can be defined solely by its syntactic function (i.e. the "cat" feature), use the "dom" feature with a regular expression. The "dom" feature is present on each structure node and lists the functions of all its dependants and relators in alphabetical order, separated by underscores. For example, if a verb has a subject, object and two adjuncts, the "dom" feature will have the value **Circ_Circ_Obj_SjPer**.

- Clauses without a subject:

```
[headpos="VERcjg" & dom!=/.*Sj.*/]
```

- Unmodified nouns:

```
[headpos=/NOM.*/ & dom!=/.*ModA.*/]
```

- Single-part clausal negation (i.e. *ne* only, no reinforcing particle such as *pas* or *mie*):

```
[headpos="VERcjg" & dom!=/.*NgPrt.*/] >D [cat="Ng"]
```

If *y* cannot be defined solely by its syntactic function, the query is impossible.

For example, while finding unmodified nouns is straightforward (**ModA** is a syntactic function), finding nouns not modified by a *determiner* is impossible with a single query ("determiner" is not a

6 The "directly precedes" (.) operator is less useful, especially since the SRCMF contains "dummy" words.

syntactic function).

Users are advised to think laterally and to formulate these corpus searches in a slightly different way. For example, while we cannot find all nouns without a determiner, we *can* find all nominal structures which begin with an element that is *not* a determiner. Note: the operator "**>@l**" (left corner dominance) indicates the leftmost dominated terminal node.

```
[headpos=/NOM.*/] >@l [pos!=/(PRE\.)?DET.*/]
```

This query produces a lot of noise, since determiners are not always the first word in a nominal structure. In particular, prepositions and conjunctions will precede any determiner. We should modify our query to check that, if a nominal structure begins with a preposition or a conjunction, the *second* word is not a determiner:

```
[headpos=/NOM.*/] >@l [pos!=/(PRE\.)?DET.*/ & pos!="PRE" &
pos!=/CON.*/]
| ( #n:[headpos=/NOM.*/] >@l #w:[pos="PRE" | pos=/CON.*/]
& #w . #w2:[pos!=/(PRE\.)?DET.*/]
& #n >* #w2 )
```

This query produces far less noise, even though it is still not perfect: structures with a conjunction *and* a preposition before the determiner will slip through the net.

2.1.5 "How do I find all (non-)coordinated *x*s?": the *coord* feature

E.g. conjoined objects, gapping constructions, non-conjoined object

If you only wish to specify whether or not *x* forms part of a coordinate structure, use the "coord" feature. "coord" has the value **y** when the node is part of a coordinate structure.

- Find all objects in a coordinate structure:

```
[cat="Obj" & coord="y"]
```

- Find all objects not in a coordinate structure:

```
[cat="Obj" & coord!="y"]
```

If you wish to place more specific constraints on coordination, you must define the structure using node relations. See sections §1.3.2.2-§1.3.2.3 for an overview of how coordination is represented in the SRCMF.

- Find all "gapping" constructions (two complex conjuncts) involving a subject and an object. Note here the use of the "siblings with precedence" operator (**$.***): this ensures that (i) #conj1 and #conj2 are separate nodes and (ii) #conj1 always refers to the first conjunct, while #conj2 always refers to the second conjunct.

```
#coo:[cat="Coo"] >P #conj1:[cat="GpCoo"]
& #conj1 $.* #conj2:[cat="GpCoo"]
& #conj1 >~cluster [cat=/Sj.*/]
& #conj1 >~cluster [cat="Obj"]
& #conj2 >~cluster [cat=/Sj.*/]
& #conj2 >~cluster [cat="Obj"]
```

2.1.6 “How do I find structures with *only one x*?”

E.g. Verbs with only one complement, nouns with only one preceding adjective.

These queries are often impossible to formulate in TIGERSearch. However, there are a couple of techniques which may arrive at something close to the desired result.

Firstly, you may be able to use the “dom” property with a regular expression (see §2.1.4 above).

- all verbs with only one complement, object included:

```
[dom=/.*(Obj|Cmpl).*/ & dom!=/.*Cmpl_Cmpl.*/ & dom!
=/.*Cmpl.*Obj/]
```

Effectively, we identify all “dom” strings containing **Obj** or **Cmpl**, but then exclude those containing **Cmpl** twice or **Obj** and **Cmpl**. However, the query will also exclude verbs with two *coordinated* complements.

Secondly, you can write a query to find all structures with “at least one *x*” and then generate a pivot and block concordance (see below, §2.1.3). This can then be sorted using spreadsheet software to separate occurrences of one *x*, two *x*s, three *x*s and so on.

- nouns with one preceding adjective (postprocessing necessary using pivot and block concordance to identify which matches contain *only one*):

```
#noun:[headpos=/NOM.*/] >L #pivot
& #adj:[headpos=/ADJ.*/] >L #block1
& #noun >D #adj
& #block1 .* #pivot
```

2.2 Specific structures

This section provides simple queries to identify common structures which are *not* directly annotated in the corpus.

2.2.1 “How do I find all negated *x*s?”

In terms of syntactic functions (the “cat” property), negation is annotated using two separate labels:

- **Ng**, when it is dependant on a verbal head
- **ModA**, when it is dependant on a non-verbal head. However, this is rare in the SRCMF as negation is attached to the verb wherever possible.

In the part-of-speech tagging, negative *ne*, *nen*, *non* are tagged as **ADVneg**. Note that in cases of pronominal enclisis (*nel*, *nes*), the tag will be **ADVneg.PROper**.

Consequently, the following query will find all negated *x*s (*#x* should be replaced with a feature constraint):

```
#x >D [headpos=/ADVneg.*/]
```

2.2.2 “How do I identify subordinate clauses?”

The SRCMF has no specific tag for subordinate clauses; however it does for main clauses: **Snt**. The easiest way to identify subordinate clauses is therefore to identify all finite verbs that do not head a

sentence:

```
[cat!="Snt" & headpos="VERcjg"]
```

2.2.3 “How do I identify auxiliary verbs/compound verb forms?”

The past participle of a compound verb is tagged as **AuxA**. The two parts of a compound verb forms can be identified with the following query:

```
#auxiliary:[headpos=/VER.*/] >D #participle:[cat="AuxA" &
headpos="VERppe"]
```

Passives can be identified with a similar query, replacing **AuxA** with **AuxP**.

2.2.4 “How do I identify clitics?”

The best query to match clitics is as follows:[7]

```
(#verb:[headpos="VERcjg"] >D #clitic:[cat=/Ng|Obj|Cmpl|Rfc/
& headpos=/(PROper|ADVneg).*/]
| #clitic:[headpos=/.*PROadv/])
& tokenarity(#clitic,1)
```

Most clitics are captured by the first option in the query, which matches personal pronouns with **Obj**, **Cmpl** or **Rfc** function, or the negative particle. The second option (following the OR operator “|”) simply matches all words tagged as **PROadv** (*en* or *i* only). Finally, we used the tokenarity() function to ensure that the node has only one child: i.e. that the **Obj/Ng/Cmpl/Rfc** dependant consists only of a single word.

3. Exporting the results as a “Key node in context“ (KNIC) concordance

The SRCMF team has developed an alternative export format for TIGERSearch, the KNIC concordance (Rainsford and Heiden 2014). This presents the results of a syntactic query in a tabular format, with one particular structure, the “key node” occupying the central column in the table. It is based on the “Key Word in Context” concordances generated by a number of corpus query programs (e.g. TXM for the *Base de Français médiéval*, Philologic).

TIGERSearch also provides a number of other export options which we will not cover here. Further details may be found in the TIGERSearch manual, ch. 4, §9.

In this chapter, we will first present the knicmaker tool (§3.1) before describing the use of concordances in greater detail (§3.2).

3.1 Creating KNIC concordances

KNIC concordances present the results of a syntactic query in a tabular format, with one particular structure, the “key node” occupying the central column in the table. The concordance is generated by the knicmaker tool (see §3.1.2), which post-processes an XML file exported from TIGERSearch containing structures which match the user’s query.

7 It is not perfect: it will fail to match clitics with modifiers (i.e. where a pronominal clitic is modified by a dislocated argument or a discontinuous relative clause), and it will also match the rare cases of stressed personal pronouns used with object or complement function.

3.1.1 Specifying the key node: the #pivot identifier

The knicmaker identifies the "keynode" in the query from the node identifier "#pivot". In order for a TIGERSearch query to be compatible with the knicmaker, it *must* contain the node identifier "#pivot".

For example, the following query is correct in TIGERSearch it finds all subjects headed by the word "rois" butcannot be used to produce a concordance:

```
[cat="SjPer"] >L [word="rois"]
```

In order to make the query compatible with the knicmaker, we must use the node identifier #pivot to identify the key node. In this case, there are two possibilities:

- The whole of the subject is the key node:

```
#pivot:[cat="SjPer"] >L [word="rois"]
```

- Only the word "rois" is the key node:

```
[cat="SjPer"] >L #pivot:[word="rois"]
```

3.1.2 Exporting the results from TIGERSearch

Firstly, you need to design and execute a TIGERSearch query containing the #pivot node identifiers, as described above.

Once you have run the query:

➔ Select 'Query > Export Matches' from the toolbar above;

➔ In the export window which pops up:

- set "Export Format" to **XML**;
- set "Export to file" to the output file of your choice;
- set "Export includes" to **Whole corpus**.

➔ Click "Submit".

This will create a Tiger-XML file compatible with the knicmaker.

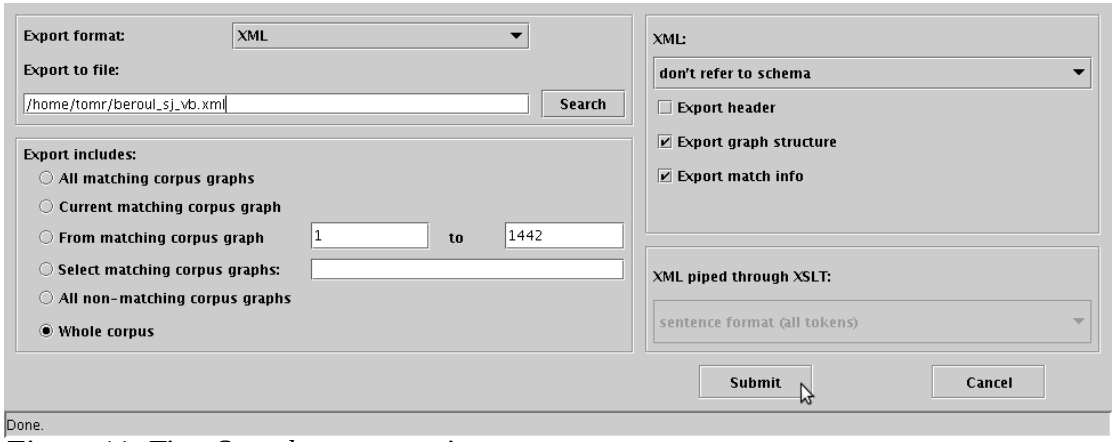

Figure 11: TigerSearch export settings

3.1.2 Installing the knicmaker

The knicmaker is tested on Windows and Linux systems[8], and requires a Java runtime environment to be installed on your computer.

- Download the knicmaker from: http://sourceforge.net/projects/knicconcordances/files/software/
- Unzip the ZIP archive to a convenient location on your computer.

3.1.3 Configuring the knicmaker

The knicmaker has no user interface:

- It must be configured using the text file knicmaker.properties *before* it is run!
- It outputs any error / success messages to the file knicmaker.log, not to the screen or terminal.

Before running the knicmaker, you should edit the file knicmaker.properties:

➔ **InFile=**: enter the full path and filename of the XML file you have just exported from TIGERSearch (e.g. /home/tomr/beroul_sj_vb.xml).

➔ **OutFile=**: enter the full path and filename of the output file you wish to generate.

➔ **ConcordanceType=**: select the type of concordance you would like to generate, using the keywords:

- **simple** for the basic concordance
- **word-pivot** for the single word pivot concordance
- **blocks** for the pivot and block concordance

➔ **TFeatures=** : enter a comma-separated list of all the word node features you would like to have in the final concordance (apart from "word", which is always shown!). Leave blank if

8 Mac users have reported problems with the jar. We suggest that Mac users either (i) run the compiled jar in a Linux virtual machine or (ii) download and run the source code directly using the Groovy interpreter (this latter option is reported to work).

no features required. E.g.:

- **TFeatures=pos,form —** show the “pos” and the “form” features.
- **TFeatures=** (i.e. blank) show no features.

➔ **NTFeatures=**: enter a comma-separated list of all the structure node features that should be shown in the concordance. Leave blank if no features required. E.g.:

- **NTFeatures=cat,headpos** — show the “cat” and “headpos” features
- **NTFeatures=cat** — show the “cat”feature

➔ **ContextSize=**: Size of the context to show in the concordance, in number of words. Ignores sentence boundaries.

Remember to save the changes to the knicmaker.properties file!

3.1.4 Launching the knicmaker

Once the knicmaker.properties file has been edited, launch the knicmaker:

➔ on Windows, by double-clicking knicmaker.exe

➔ on Linux:

- Open a terminal
- Change the working directory to where you installed the knicmaker using the “cd” command, e.g.

```
cd /home/tmr/knicmaker
```

- Launch knicmaker.jar using the following command:

```
java -jar knicmaker.jar
```

Check the knicmaker.log file for any errors during execution (e.g. mistyped file names).

3.1.5 Opening the output file

The knicmaker exports the concordance as a unicode (UTF-8) tab-separated text file. This simple format is easily loaded in LibreOffice/OpenOffice:

➔ open the file

➔ the spreadsheet software should open a configuration window such as the following (from LibreOffice Calc v. 3.4.4):

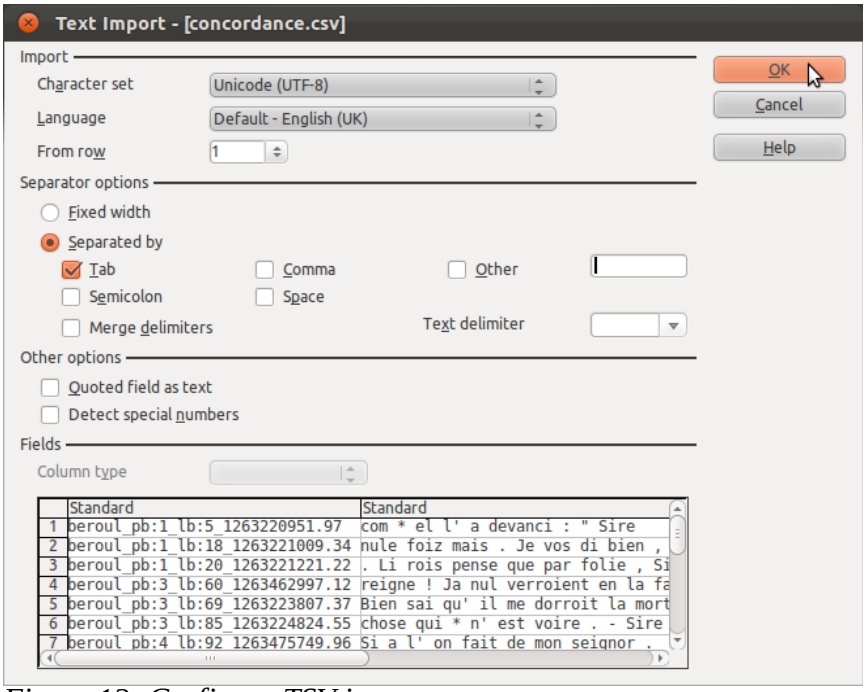

Figure 12: Configure TSV import

➔ The following settings are required for the import to function:

- Character set: Unicode (UTF-8);
- Separated by Tab (ONLY);
- Merge delimiters OFF;
- Text delimiter: NONE (empty box)

If your spreadsheet software will not load the file directly:

➔ open the file first in a text editor, making sure the encoding is set to unicode (UTF-8)

➔ copy-paste the table into the spreadsheet program.

If the concordance does not appear correctly, check the following likely problems:

➔ If accented characters do not appear correctly > check the character set is UTF-8;

➔ If some rows do not seem to have the correct number of columns > check that Text Delimiter is set to nothing (the default is usually double quote, which will cause an error where the text contains double quotes), merge delimiters is OFF, and TAB is the only separator selected.

If zeros appear rather than punctuation (unlikely), use the ‘Fields’ section of the import window to set every column type to ‘Text’ rather than ‘Standard’.

3.2 Design of KNIC concordances

Three variants of the KNIC concordance are currently implemented in the knicmaker:

- basic concordance

- pivot and block concordance
- single word pivot concordance

3.2.1 Basic concordance

The basic concordance has a minimum of six columns:

- sentence ID
- left context outside sentence
- left context within sentence
- pivot
- right context within sentence
- right context outside sentence

The pivot can be any node in the graph, either a word node or a structure node.

For example, we may wish to create a concordance of all the cases in which the word “Yvain” occurs within the main clause subject:

```
[cat="Snt"] >D #pivot:[cat="SjPer"]
& #pivot >* [word=/Yvain[sz]?/]
```

The pivot node denotes the subject of the clause. Below is a selection of the results from the concordance:

ID	LeftCxInsideSnt	pivot	RightCxInsideSnt
YvainKu_pb:79_lb:56Bea%20%-%20%1/YvainKu_01_1309961775.91	Et si i fu	messire Yvains	Et avoec ax Qualogrenanz Uns chevaliers mout avenanz Qui
YvainKu_pb:90_lb:2774Bea%20%-%20%1/YvainKu_10_1321888863.94		Yvains	respondre ne li puet
YvainKu_pb:91_lb:3266Bea%20%-%20%1/YvainKu_12_1322227968.63	Mes	messire Yvains [pas] [ne] [fuit] Qui * de lui siudre ne se faint	

Table 1: Basic concordance, subject contains “Yvain”

Note that the pivot may be one or more words.

The third example in table 1 contains square brackets ([]) in the pivot.

(x) Mes messire Yvains pas ne fuit Qui de lui siudre ne se faint.
"But Lord Yvain does not flee; he does not hesitate to follow him [lit. 'who does not hesitate...']"

These are used in all concordances when the structure in the pivot column is *discontinuous*. The annotated subject in this sentence is *messire Yvains ... qui de lui siudre ne se faint*. In between the first part of the subject (*messire Yvains*) and the relative which modifies the subject (*qui de lui siudre ne se faint* "who does not hesitate to follow him") is the negated main verb of the sentence, *pas ne fuit* "does not flee". The words *pas ne fuit*, which separate the two parts of the subject, are included in the pivot column surrounded by square brackets.

This means that:

- the pivot column contains *all parts* of discontinuous pivots;
- reading the concordance from left to right will *always* give the original sentence.

Asterisks (*) and hashes (#) are "dummy" words in the SRCMF, and may be ignored (see §1.3.2).

Slashes (/) indicate divisions between sentences in the syntactic annotation, and are only present in the "context outside sentence" columns of the concordance.

3.2.2 Pivot and block concordance

The pivot and block concordance is designed to highlight the position of certain structures, called "blocks" (e.g. the subject) with respect to a pivot (e.g. the verb). The resulting tables are complex, with a large number of columns, and are intended as the basis for more detailed analysis.

3.2.2.1 Basic structure of the pivot and block concordance

The pivot and block concordance has the following basic structure:

- sentence ID
- left context outside sentence
- left context within sentence
- pre-pivot blocks
- pivot
- post-pivot blocks
- right context within sentence
- right context outside sentence

As with the basic concordance, TIGERSearch queries must define a #pivot node. Additionally, users may define any number of other nodes in the query as blocks, using a node identifier of the form **#block*n*** (i.e. #block1, #block2, #block3...).

For example, the following query will generate a pivot and block concordance to show the position of the subject (#block1) with respect to the verb in main clauses (#pivot):

```
#snt:[cat="Snt"] >D #block1:[cat=/Sj.*/]
& #snt >L #pivot
```

The key section of the resulting concordance will take the following form:

Left context	Block	Pivot	Block	Right context
	Li rois	fu		a Carduel en Gales
Aprés mangier parmi ces sales	Cil chevalier {s'}	atropelerent		La ou * # dames les apelerent Ou dameiseles ou puceles
Or		est	Amors	tornee a fable Por ce que cil qui * rien n' en santent Dïent qu' il aiment

Table 2: Simplified pivot and block concordance: block is subject, pivot is finite verb.

Where the subject is pre-verbal, it appears in the block column to the left of the pivot. Where it is post-verbal, it appears in the block column to the right of the pivot.

Pivot and block concordances may contain braces (**{ }**) in the block column (e.g. *cil chevalier {s'}*). These mark words which intervene between the block and the following column (in this case, the pivot). The reflexive pronoun *s'* is here not part of the subject, but intervenes between the subject and the verb.

As with the basic concordance, text word order is maintained if the concordance is read from right to left.

3.2.2.2 Why so many columns?

The pivot and block concordance shows *only one result per pivot.* Continuing to work with the same example, if a single verb-pivot has multiple subjects (which is quite possible in cases of coordination), each subject occupies a separate column:[9]

Pivot	Block	Block	Block	Block
Fu	{#} Didonez	et Sagremors	Et Kex	et messire Gauvains

Table 3: Coordinated subjects as blocks

However, due to the way the number of columns is calculated, some may be empty throughout the concordance. These may be deleted in the spreadsheet software, if you wish.

9 Those used to TIGERSearch will note that the concordance combines multiple matches within a single sentence where the #pivot variable is the same.

3.2.2.3 Discontinuous single structures *vs* two structures matching #block

Discontinuous structures are always in a single column:

Left context	Block	Pivot	Block	Right context
Et	cil come mautalentis [Vint] [plus] [tost] [c'] [uns] [alerions] Fiers par sanblant come lions	Vint		plus tost c' uns alerions Fiers par sanblant come lions

Table 4: Discontinuous block

This representation shows:

- a single discontinuous subject *Cil come mautalentis [...] Fiers par sanblant come lions* in the block column;
- within the block column, words that intervene between the two parts of the subject appear between square brackets (**[]**), even if these words appear later in other columns;
- the position of the block relative to the pivot is determined by its first word.

This case is distinguished from that in which there is more than one structure matching the definition of the #block node in the sentence. In this second case, more than one block column is filled. Compare the following result, which shows the gapping construction discussed in §2.2.2.3, with table 4:

Left context	Block	Pivot	Block	Right context
Que #	je	fui	{plus} {petiz} {de} {lui} {Et} ses chevax	miaudres del * mien

Table 5: Coordinated subjects: two separate blocks

This representation shows *two separate subjects*, one preverbal (*je*) and one postverbal (*ses chevax*). The words *plus petiz de lui* intervene between the verb and the postverbal subject, and are marked using braces.

3.2.3 Adding features to the concordance

It is also possible to create concordances which show node features other than lexical form. For example, we may wish to create a pivot and block concordance showing the position of the subject and non-pronominal object relative to the main clause verb. In this case, it is vital that the exported concordance labels each of the blocks with its syntactic function.

```
#snt:[cat="Snt"] >D #block1:[cat=/Sj.*/]
& #snt >D #block2:[cat="Obj" & headpos!=/.*PROper/]
& #snt >L #pivot
```

Block		Block		Pivot	Block	
		Li un	SjPer	recontoient	noveles	Obj
		que	Obj	voldroies	tu	SjPer
		divers chanz	Obj	chantoit	chascuns	SjPer
Li uns	SjPer	l'autre {a} {l'} {espee}	Obj	assaut		

Table 5: Simplified pivot and block concordance: blocks are subject and object, pivot is finite verb.

By inserting the "cat" feature of each block into the concordance, a very clear overview of the word order in each sentence is obtained. The resulting table can also be sorted by the columns containing the "cat" feature in order to group sentences with a similar word order together.

3.2.4 Single word pivot concordance

The final type of concordance can only take a word node as its pivot, and contains a minimum of 7 columns, based on the following structure:

- sentence ID
- left context outside sentence
- left context within sentence
- pivot (single word)
- structure of which pivot is head
- right context within sentence
- right context outside sentence

The single word pivot concordance is designed to give as much information as possible about a single word. It is designed to be used with multiple columns showing node features inserted.

For example, a single word pivot concordance could be created around the word *Yvain*:

```
#pivot:[word=/Yvain[sz]?/]
```

Below is a selection of the results from the concordance (some columns are omitted):

Left context in sentence	Pivot	Pivot "form"	Pivot-headed structure	Pivot-headed structure "cat"	Right context in sentence
Et si i fu messire	Yvains	vers_fin	Yvains	ModA	Et avoec ax Qualogrenanz Uns chevaliers

					mout avenanz Qui
[...] Et toz les autres fors	Yvain	vers_fin	fors Yvain Le mançongier le guileor Le desleal le tricheor	ModA	Le mançongier le guileor Le desleal le tricheor
Dame je ai	Yvain	vers	Yvain [trové] Le chevalier [#] mialz esprové Del * monde et le mialz antechié	Obj	trové Le chevalier # mialz esprové Del * monde

Table 6: Single word pivot concordance, pivot is "Yvain[sz]?"

The concordance is similar to the basic concordance, but with the addition of a "pivot-headed structure" column, which shows the pivot with its dependents. In the third example, for instance, the word *Yvain* heads the structure *Yvain [...] Le chevalier mialz esprové del monde et le mialz antechié* "Yvain, the most distinguished knight in the world and the most noble", which is the object of the sentence. Note that words appearing in the 'pivot-headed structure' column are also found in the two context columns. The original sentence may be read across the columns left context — pivot — right context.

Appendix: Node features and edge labels in the SRCMF corpus

Features of terminal nodes

- **word**: word form as found in the base edition, e.g. *rois*
- **pos**: part-of-speech tag, using the Cattex schema developed by the *Base de Français Médiéval*. See Guillot *et al.* (2013).
- **form**: shows whether the word is in a prose text or a verse text, and its position in the line. Possible values:
  - **prose**: prose text
  - **vers**: verse text, word within the line
  - **vers_debut**: verse text, word at the beginning of the line

    - **vers_fin**: verse text, word at the end of the line
- **q**: shows whether the word occurs in direct discourse or not. N.b.: property is not implemented for the *Yvain* text.
    - **y**: word occurs in direct discourse
    - **n**: word does not occur in direct discourse
- **editionId**: xml:id identifier of the word used in the source edition.
- **editionNs**: namespace of the source edition
- **editionUri**: URI for the word in the SRCMF corpus (editionNs + editionId)

Features of non-terminal nodes

- **cat**: syntactic function of the node. For the tagset, please refer ....?
- **dom**: list of all syntactic functions which depend on the node, ordered alphabetically and separated by "_". For example, a verb with a subject, an object and no other arguments will have the dom value **Obj_SjPer**.
- **headpos**: Part-of-speech tag of the word node (see “pos” for finite verbs)
- **type**: Simplified “headpos”: gives the part-of-speech of the head node as:
    - **VFin**: finite verb
    - **VInf**: verb infinitive
    - **VPar**: verb participle
    - **nV**: not a verb
- **coord**: value is "y" if the node is a conjunct (or part of a complex conjunct).
- **annotationUri**: URI of the structural annotation in the SRCMF corpus.
- **annotationFile**: used in corpus development
- **nodom**: redundant

Edge labels

- **L** “lexical”: links two members of a node pair.
- **D** “dependency”: marks a dependency relation between two structure nodes.
- **R** “relator”: links a governing node to prepositions and conjunctions.
- **P** “part”: links the coordination node **Coo** to nodes representing complex conjuncts (**GpCoo**)

Secondary edge labels

- **cluster**: links the node representing a complex conjunct (**GpCoo**) to the structures which form the conjunct.
- **coord**: links the coordination node **Coo** to its conjunctions (simple coordination).
- **dupl**: links a structure node to its “true” word node in cases of word with two functions.

References

König, Esther, Wolfgang Lezius, and Helger Voormann (2003) *TIGERSearch User’s Manual* (Stuttgart: IMS, University of Stuttgart), http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/TIGERSearch/manual.html

Rainsford, T. M. and Serge Heiden (2014) ‘Key Node in Context (KNIC) Concordances: Improving Usability of an Old French Treebank’, *SHS Web of Conferences* 8: 2707–2718 <DOI: 10.1051/shsconf/20140801250>.