Aid to spatial navigation within a UIMA annotation index

Nicolas Hernandez

Université de Nantes

Abstract. In order to support the interoperability within UIMA workflows, we address the problem of accessing one annotation from another when the type system does not specify an explicit link between the two kinds of objects but when a semantic relation between them can be inferred from a spatial relation which connects them. We discuss the limitations of the framework and briefly present the interface we have developed to support such navigation.

Keywords: Apache UIMA, Type System interoperability, Annotation Index, Spatial navigation

1 Introduction

One of the main ideas in using document analysis frameworks such as *Apache Unstructured Information Management Architecture*[1] (UIMA) [3] is to move away from handling directly the raw *subject of analysis*. The idea is to enrich the raw data with descriptions which can be used as basis for the processing of subsequent components. In the UIMA framework, the descriptions are *typed feature structures*. The component developer defines a *type system* which informs about the features of a type (set of (attribute, typed value) pairs) as well as how the types are arranged together (through inheritance and aggregation relations). *Annotations* are feature structures attached to specific regions of documents.

In this paper, we address the problem of accessing one annotation from another when the type system does not specify an explicit link between the two kinds of objects but when a semantic relation between them can be inferred from a *spatial relation* which connects them. The situation is a case of interoperability issue which can be encountered when developing a component (e.g. a term extractor) that uses analysis results produced by two components developed by different developers (e.g. part-of-speech and lemma information being both hold by distinct annotations at the same spans).

In practice, most of the existing type systems define annotation types which inherit from the built-in `uima.tcas.Annotation` type [4, 5, 7]. This type contains begin and end features which are used to attach the description to a specific region of the text being analysed. Thanks to these features, it is possible to cross-reference the annotations which extend this type.

[1] `http://uima.apache.org`

In this paper, we argue that the Apache UIMA Application Programming Interface (API) is not enough intuitive for a Natural Language Processing (NLP) developer. We argue that it has some restrictions which prevent from a complete and free navigation among the annotations. We also argue that the API is forcing the way of developing algorithms. In section 2, we define the kind of spatial navigation we would like to perform within an annotation index. In section 3, we describe the Apache UIMA solutions to index and explore the annotations within the indexes. In section 4, we discuss the API and show its limitation to access annotations by spatial relations. Finally, in section 5, we briefly present the structures and the interface we have developed to support a spatial navigation.

2 The spatial navigation problem

By spatial relations we mean that we assume that the annotations in a text can be located in a two-dimensional space: One axis to represent the position in the text linearity and an orthogonal axis to represent the covering degrees between the annotations. Indeed annotations can cover, be covered by, precede, or follow (contiguously or not) other annotations. The spatiality may inform about the semantic relations. Two annotations at the same span may mean that they are different aspects of the same object. They can *have complementary features* or one of them can *be the property of* the other. One annotation covering some others may mean that the former *is made of* the others, and in the opposite, that the others *are part of* the former. The semantic interpretation of the spatial relations may depend on the considered linguistic paradigm.

To give examples of situations we are dealing with, let's assume the following type system: `Document`, `Source` (information about the document such as its URI), `Sentence`, `Chunk`, `Word` (having a feature whose value informs about the lemma), `POS` (having a feature whose value informs about the part-of-speech) and `NamedEntity`. Let's also assume that all these types do not hold explicit references to each other and that there is no inheritance relation between them. In that context, examples of access we would like to carry out are to get :The words of a given sentence (can be interpreted as a *made of* relation); The sentence of a given word (*is part of* relation); The words which are a verb (*is a property of* relation); The named entities followed by a word which is a verb and has the lemma *visit* (*followed by* relation, . . .). Indeed, we would like to be able to navigate within an annotation index from an annotation to its covering/covered annotations or to the spatially following/preceding annotation of a given type having such or such properties. We defined this problem as a navigation problem within an annotation index.

3 Accessing the annotations in the UIMA framework

The problem of accessing the annotations depends on the means offered by the framework[2] to build annotation indexes and to navigate within them.

[2] See the Reference Guide `http://uima.apache.org/d/uimaj-2.4.0/references.html` and the Javadoc `http://uima.apache.org/d/uimaj-2.4.0/apidocs`.

3.1 Defining feature structures indexes

Adding a feature structure to a subject of analysis corresponds to the act of indexing the feature structure. By default, an unnamed built-in bag index exists which holds all feature structures which are indexed. The framework defines also a built-in annotation index, called `AnnotationIndex`, which automatically indexes all feature structures of type (and subtypes of) `uima.tcas.Annotation`. As reported in the documentation, "the index sorts annotations in the order in which they appear in the document. Annotations are sorted first by increasing begin position. Ties are then broken by decreasing end position (so that longer annotations come first). Annotations that match in both their begin and end features are sorted using a *type priority*". If no type priority is defined in the component descriptor[3], the order of the annotations sharing the same span in the text is undefined in the index. The UIMA API provides `getAnnotationIndex` methods to get all the annotations of that index (subtypes of `uima.tcas.Annotation`) or the annotations of a given subtype. The UIMA framework allows also to define indexes and possibly to sort the feature structures within them.

3.2 Parsing the annotation index

The UIMA API offers several methods to parse the `AnnotationIndex`. Given an annotation index, the *iterator* method returns an object of the same name which allows to move to the first (respectively the last) annotation of the index, the next (respectively the previous) annotation (depending on its position in the index) or to a given annotation in the index. It is also possible to get an *unambiguous* iterator to navigate among contiguous annotations in the text. In practice, this iterator consists of getting successively the first annotation in the index whose begin value is higher than the end of the current one. We will call this mechanism the *first-contiguous-in-the-index* principle.

The *subiterator* method returns an iterator whose annotations fall within the span of another annotation. It is possible to specify whether the returned annotations should be *strictly* covered (i.e. both begin and end offsets covered) or if it concerns only its begin offset. Subiterator can also be unambiguous. Annotations at the same span may be not returned depending on the order in the index as well as the type priority definition.

The *constrained iterator* allows to iterate over feature structures which satisfy given constraints. The constraints are objects that can test the type of a feature structure, or the type and the value of its features.

The *tree* method returns an *AnnotationTree* structure which contains nodes representing the results of doing recursively a strict, unambiguous subiterator over the span of a given annotation. The API offers methods to navigate within the tree from the root node. From any other nodes, it is possible to get the children nodes, the next or the previous sibling node, and the parent node.

[3] In a UIMA workflow, a component is interfaced by a text descriptor that indicates how to use the component.

4 Limitations of the UIMA framework

Table 1a shows the `AnnotationIndex` containing the analysis results of the data string "*Verne visited the seaport of Nantes.\n*". Annotations were initially added to the index in that order: First the `Document`, then the `Source`, the `Sentence`, the `Words`, the `POS`, the `Chunks` and the `NamedEntities`.

Offsets	**Annotations**	**Covered text**
(0,37)	`Document`	*Verne visited the seaport of Nantes.\n*
(0,36)	`Sentence1`	*Verne visited the seaport of Nantes.*
(0,5)	`Word1`	*Verne*
(0,5)	`NamedEntity1`	*Verne*
(0,5)	`POS1`	*Verne*
(0,5)	`Chunk1`	*Verne*
(0,0)	`Source`	
(6,13)	`Word2`	*visited*
(6,13)	`POS2`	*visited*
(6,13)	`Chunk2`	*visited*
(14,35)	`Chunk3`	*the seaport of Nantes*
(14,25)	`Chunk4`	*the seaport*
(14,17)	`Word3`	*the*
(14,17)	`POS3`	*the*
(18,25)	`Word4`	*seaport*
(18,25)	`POS4`	*seaport*
(26,35)	`Chunk5`	*of Nantes*
(26,28)	`Word5`	*of*
(26,28)	`POS5`	*of*
(29,35)	`Word6`	*Nantes*
(29,35)	`NamedEntity2`	*Nantes*
(29,35)	`POS6`	*Nantes*
(35,36)	`Word7`	.
(35,36)	`POS7`	.

(a)

LocatedAnnotations
`Document`
`Sentence`
`Word1 NamedEntity1 Chunk1 POS1`
`Source`
`Word2 Chunk2 POS2`
`Chunk3`
`Chunk4`
`Word3 POS3`
`Word4 POS4`
`Chunk5`
`Word5 POS5`
`Word6 NamedEntity2 POS6`
`Word7 POS7`

(b)

Table 1: An `AnnotationIndex` (a) and its corresponding `LocatedAnnotationIndex` (b). Both tables are aligned for comparison. `Annotations` and `LocatedAnnotations` are sorted in increasing order from the top of the tables. Annotations are identified by their type and an index number.

4.1 Index limitations

The definition of an index is usually done in the component descriptor. The defined index can only contain one specific type (and subtypes) of feature structures. So, to get an index made of two distinct types, the trick would be to declare them as subtypes of the same common type in the type system, and get the index of this super type. This can lead to make a less consistent type system from a linguistic point of view, but this is still coherent with the UIMA approach of doing whatever you need in your component.

The framework allows also so to sort the feature structures of a defined index. There are some restrictions. The sorting key, which should be a feature of the indexed type, can only be a string or a numerical value. Only the natural way of sorting such elements is available. There is no way to declare its own comparator to set the order between two elements. To sort on a different kind of key, the developer has to come down to the available systems. In addition, the type system may need to be modified to add a feature to play the role of the sorting key, which can also make the type system less consistent.

4.2 Navigation limitations within an annotation index

Iterator With an ambiguous iterator, the result of a move to the previous/next annotation in the index may not correspond to the annotation which precedes/follows spatially in the text. It can also be a covering or a covered one. In Table 1a, the preceding of `Word3` is the covering `Chunk4`. Unambiguous iterators force the methods to return only spatially contiguous annotations. In practice, the method does not always return the expected result. When called on the full annotation index, it starts from the first annotation in the index. In Table 1a, it only returns the `Document` annotation and no more next annotation. When calling a unambiguous iterator on a typed annotation index, the effect of the *first-contiguous-in-the-index* principle will be remarkable if some annotations occur at the same span. In that situation, the developer has no access to all the annotations which effectively follow/precede spatially the current annotation. In Table 1a, an unambiguous iteration over the `Chunk` type returns `Chunk1`, `Chunk2` and `Chunk3`. `Chunk4` and `Chunk5` are not reachable. To iterate unambiguously over annotations of distinct types (e.g. Named Entities and POS to get the Named Entities followed by a verb), the developer has to create a super-type over them and call the iterator method on this super-type. The super-type may not have linguistic consistency and the iterator will still suffer from the limitation we have previously mentioned. Another drawback of the unambiguous iterator can be noticed when iterating an index in reverse order. If two overlapping annotations precede the current one, the one returned will be the one whose begin offset is the smallest and not the one with the highest end value, lower than the begin value of the current one. The iterator follows the *first-contiguous-in-the-index* principle in the normal order. Finally, the API does not allow to iterate over the index and in the text spatiality in the same time. It is not possible to switch from an ambiguous iterator to an unambiguous one (and vice-versa).

Subiterator is the kind of method to get the covered annotations of another one, like the words of a given sentence. Its major drawback is that, without a type priority definition, there is no assurance that annotations occurring at the same text span will fit an expected conceptual order. In Table 1a, an ambiguous `subiterator` over each chunk annotation for getting the words returns the `Source` annotation for `Chunk1`, nothing for `Chunk2`, and the expected words (and more to filter) for the all remaining Chunks. Concerning the unambiguous subiterator, the *first-contiguous-in-the-index* principle causes to hide some annotations. In Table 1a, when applying an unambiguous `subiterator` over each chunk, then `Chunk1` and `Chunk2` return the same bad result as previously. `Chunk3` only returns `Chunk4` and `Chunk5` annotations while `Chunk4` and `Chunk5` return the right word annotations. To subiterate unambiguously over a set of specific types, a super-type, which encompasses both the covered and the covering types, has to be defined in the type system. But the problem of the unambiguous iteration remains.

Constraints objects aim at testing one given feature structure at a time. The framework does not allow to define dynamic constraints. This means that the values to test cannot be instantiated relatively to the feature structure in the index. A constraint cannot be set to select annotations whose begin feature value is higher than the end feature value of another one. Rather, we have to specify at the creation the exact value to be higher than. Constraints objects are complex to understand and to set. It requires, for example, seven lines of code for creating an iterator which will get the annotations with a lemma feature. Constraint iterators remain iterators with the same limitations.

The **Tree** method returns an object close to the kind of structure we would like to manipulate to navigate within. Unfortunately, it can only give the children of a covering annotation. So to get the parent of an annotation, a trick could be to build the tree of the whole document by taking the most covering annotation as the root, then to browse the tree until finding the desired annotations for finally getting its parent. But in any case, there is no way to get directly a node and the structure will still suffer from the remarks we made about unambiguous subiterators (consequently some annotations may not be present in the tree).

Missing Methods The existing methods partially answer the problem and some navigation methods are missing. There is no dedicated method: to *super-iterate* and to get the annotations covering a given annotation; to move to the first/next annotation of a given type (respectively the last/previous annotation of a given type); or to get partially-covering preceding or following annotations.

All these remarks lead the developers to use preferentially ambiguous iterators and subiterators, even if, this causes to write more code to search the index backward/forward and tests to filter the desired annotations.

5 Supporting the spatial navigation

To support a spatial navigation among the annotations we propose to index the annotations by their offsets in a structure called `LocatedAnnotationIndex`, and to merge the annotations occurring at the same spans in a structure called `LocatedAnnotation`. Table 1b illustrates the transformation of the `AnnotationIndex` depicted in Table 1a into a `LocatedAnnotationIndex`. Figure 1 shows the spatial links which interconnect the `LocatedAnnotation`.

The `LocatedAnnotationIndex` is a sorted structure which follows the same sorting order than the `AnnotationIndex`: From a given `LocatedAnnotation`, covering and preceding `LocatedAnnotations` are located backward in the index, and the covered and following `LocatedAnnotations` forward in the index. The structure allows to access directly to a `LocatedAnnotation` thanks to a pair of begin/end offsets. The first characteristic of a `LocatedAnnotation` is to list all the annotations occurring at the same offsets. This prevents from having to define a type priority for handling the limitation of the subiterator. The structure comes with several kinds of links to navigate both within the `LocatedAnnotationIndex` and spatially in the text. Indeed, the structure has links to visit its spatial vicinity (parent/children/following/preceding) `LocatedAnnotation`. The structure has also links to access the previous/next element in the index. The contiguous spatial vicinity of each `LocatedAnnotation` is computed when the `LocatedAnnotationIndex` is built. The API also offers some methods to dynamically search `LocatedAnnotation` containing annotations of a given type among the ancestor/descendant or self. Similarly, it is also possible to search the first/last (respectively following/preceding) `LocatedAnnotation` containing annotations of a given type.

In terms of memory consumption, the built `LocatedAnnotationIndex` takes approximatively as much memory as its `AnnotationIndex`; only the local vicinity of each `LocatedAnnotation` is kept in memory. The CPU time for building the index depends on the `AnnotationIndex` size. Some preliminary tests indicate that the time increases by a factor of three when doubling the size of the annotated text. It takes about 2 seconds for building the index of a 50-sentences text analysed with sentences, chunks and words.

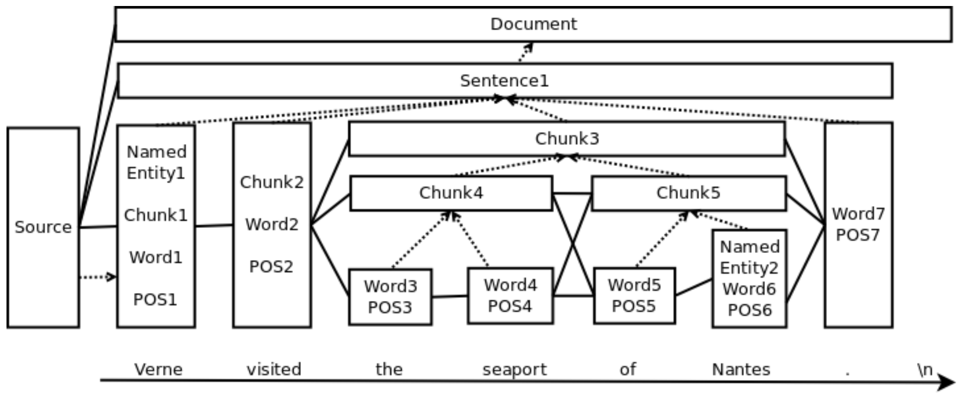

Fig. 1: Example of `LocatedAnnotationIndex`. The boxes represent the `LocatedAnnotation`. They are aligned on the text span they cover. The solid lines represent the spatial preceding/following relation while the dotted lines represent the parent/child relations. The parent is indicated by an arrow.

6 Related works

With the prospect of developing a pattern matching engine over annotations, [2] have addressed some design considerations for navigating annotation lattices. They have so exposed a language for specifying spatial constraints among annotations. An engine has been implemented within the UIMA framework. Due to this technical choice, the design of the language and its implementation may suffer from the drawbacks we have enumerated. Indeed there is no example of patterns which involve annotation types without inheritance relation. In addition, as pointed out in the perspectives of the authors, it is not clear how the engine will behave when handling multiple annotations over the same spans without the guarantee of a consistent type priority. The `LocatedAnnotation` structure is a solution to the need of defining type priorities. More generally, the methods of our API can play the role of the navigation devices required to the development of a pattern matching engine.

`uimaFIT`[4] is a well-known library which aims at simplifying the UIMA developments. One appealing navigation option it offers is similar to our API. Some methods are designated to move from one annotation to the closest (covering/covered/preceding/following) ones by specifying the type of the annotations to get. In practice, nevertheless, the implementation relies on the UIMA API and may have some of the restrictions. The `selectFollowing` method, for example, follows the *first-contiguous-in-the-index* mechanism. In Table 1a, it returns only the `Chunk` 1 to 3, and misses the 4th and 5th, when calling it successively to get the following chunk from the first chunk.

7 Conclusion and perspectives

Solving the interoperability issues in the UIMA framework is a serious problem [1, 6]. Our opinion is to give the means to developers to do what they want. We show that the UIMA API presents some limitations regarding the spatial navigation within annotations in a text. We also show that by adapting his problem definition to the framework requirements the developer may succeed to accomplish his task. But the adaptation has a cost in development time and requires skills in the framework. To overcome this problem, we have developed a library which transforms an `AnnotationIndex` into a navigable structure which can be used in a UIMA component. It is available in the *uima-common* project[5]. Our perspectives are twofold: Reducing the processing time and adding a mechanism for updating the `LocatedAnnotationIndex`.

References

1. Ananiadou, S., Thompson, P., Kano, Y., McNaught, J., Attwood, T.K., Day, P.J.R., Keane, J., Jackson, D., Pettifer, S.: Towards interoperability of european language resources. Ariadne 67 (2011)
2. Boguraev, B., Neff, M.S.: A framework for traversing dense annotation lattices. Language Resources and Evaluation 44(3), 183–203 (2010)
3. Ferrucci, D., Lally, A.: Uima: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering 10(3-4), 327–348 (2004)
4. Gurevych, I., Mühlhäuser, M., Müller, C., Steimle, J., Weimer, M., Zesch, T.: Darmstadt knowledge processing repository based on uima. In: First Workshop on UIMA at GSCL. Tübingen, Germany (2007)
5. Hahn, U., Buyko, E., Tomanek, K., Piao, S., McNaught, J., Tsuruoka, Y., Ananiadou, S.: An annotation type system for a data-driven nlp pipeline. In: The LAW at ACL 2007. pp. 33–40 (2007)
6. Hernandez, N.: Tackling interoperability issues within uima workflows. In: LREC. pp. 3618–3625 (2012)
7. Kano, Y., McCrohon, L., Ananiadou, S., Tsujii, J.: Integrated NLP evaluation system for pluggable evaluation metrics with extensive interoperable toolkit. In: SETQA-NLP. pp. 22–30 (2009)

[4] `http://uimafit.googlecode.com`

[5] `https://uima-common.google.com`