Two Practical Rhetorical Structure Theory Parsers

Mihai Surdeanu, Thomas Hicks, and Marco A. Valenzuela-Escárcega
University of Arizona, Tucson, AZ, USA
{msurdeanu, hickst, marcov}@email.arizona.edu

Abstract

We describe the design, development, and API for two discourse parsers for Rhetorical Structure Theory. The two parsers use the same underlying framework, but one uses features that rely on dependency syntax, produced by a fast shift-reduce parser, whereas the other uses a richer feature space, including both constituent- and dependency-syntax and coreference information, produced by the Stanford CoreNLP toolkit. Both parsers obtain state-of-the-art performance, and use a very simple API consisting of, minimally, two lines of Scala code. We accompany this code with a visualization library that runs the two parsers in parallel, and displays the two generated discourse trees side by side, which provides an intuitive way of comparing the two parsers.

1 Introduction

This paper describes the design and development of two practical parsers for Rhetorical Structure Theory (RST) discourse (Mann and Thompson, 1988). This work contributes to the already vast body of research on RST parsing (see, inter alia, Soricut and Marcu, 2003; Feng and Hirst, 2012; Joty et al., 2013, Joty et al., 2014) with the following:

1. We propose two parsers that use constituent-based and dependency-based syntax, respectively. The underlying framework, other than the syntax-based features, is identical between the parsers, which permits a rigorous analysis of the impact of constituent and dependency syntax to RST parsing. We describe the parsers in Section 2 and empirically compare the impact of the two syntactic representations in Section 3. Our analysis indicates that both parsers achieve state-of-the-art performance. The parser based on dependency syntax performs marginally worse (by 0.1 F_1 points) but runs approximately 2.5 times faster than the parser based on constituent syntax. On average, the faster parser processes one document from the RST corpus in 2.3 seconds.

2. Both parsers have been released as open-source Scala code with a very simple API; consisting of, minimally, two lines of code. We discuss this API in Section 4.

3. We also introduce a visualization tool that runs the two parsers in parallel, and displays the two generated discourse structures side by side. This allows users to directly compare the runtimes and outputs of the two parsers. This visualization tool will be the centerpiece of the proposed demo session. We summarize this tool in Section 5.

2 The Two Parsers

The proposed parsing approach follows the architecture introduced by Hernault et al. (2010), and Feng and Hirst (2012). The parser first segments the text into elementary discourse units (EDUs) using an i.i.d. classifier that identifies which tokens end an EDU. Then the parser iteratively constructs the discourse tree (consisting of binary relations between discourse units) using a greedy bottom-up approach that interleaves two classifiers: the first de-

tects which two adjacent discourse units are most likely to be connected given the current sequence of units; and the second labels the corresponding relation. The resulting discourse unit produced by the new relation replaces its two children. The process repeats until there is a single discourse unit spanning the text.[1]

We chose this algorithm rather than other recent proposed approaches (Joty et al., 2013; Joty and Moschitti, 2014) because: (a) it promotes a simple, modular architecture; (b) it is fast, and (c) as we show later, it performs well. For classification, we experimented with Support Vector Machines (SVM), Perceptron, and Logistic Regression (LR). The results reported here use Perceptron for EDU segmentation and relation detection, and LR for relation labeling, thus offering a good balance between performance and quick training.

With respect to features, our approach builds on previous work (Hernault et al., 2010; Feng and Hirst, 2012; Joty et al., 2013) and extends it in two ways. First, we implement all syntactic features using both constituent and dependency syntax. For example, a crucial feature used by the relation detection/labeling classifiers is the *dominance relations* of Soricut and Marcu (2003), which capture syntactic dominance between discourse units located in the same sentence. While originally these dominance relations were implemented using constituent syntax, we provide an equivalent implementation that relies on dependency syntax. There are two advantages to this approach: (a) we can now implement a full RST discourse parser using a (much faster) dependency parser; (b) when using a parser that produces both constituent and dependency syntax, such as Stanford's CoreNLP[2], our experiments show that using both these feature sets increases the performance of the model.

Our second contribution is adding features based on coreference links. We currently use coreference information in two of the latter classifiers (relation detection and labeling) by counting the number of coreference links crossing between the two discourse units under consideration. The intuition behind this feature is that the more coreferential relations exist between two discourse units, the more likely they are to be directly connected.

Using the above framework, we implemented two discourse parsers. The first uses CoreNLP for syntactic parsing and coreference resolution. This parser uses both constituent- and dependency-based features generated using the parser of Manning and Klein (2003). The second discourse parser uses either Malt[3] or the recent neural-network-based parser of Chen and Manning (2014) for dependency parsing. The second discourse parser does not use constituent- nor coreference-based features. For all syntactic parsers, we used the "basic" Stanford dependency representation (de Marneffe et al., 2006). Empirically, we found that this representation yields better discourse parsing performance than any of the "collapsed" representations.

3 Analysis

We analyze the performance of the two discourse parsers in Table 1. For conciseness, we identify the parser that uses both constituent- and dependency-based syntax and coreference resolution (all produced using CoreNLP) as C, and the parser that uses only dependency-based features as D. The latter one is subclassed as D_{malt}, if the syntactic analysis is performed with the Malt parser, or $D_{stanford}$, if syntactic parsing is performed with the parser of Chen and Manning (2014). Because we are interested in end-to-end performance, we report solely end-to-end performance on the RST test corpus (Carlson et al., 2003). This analysis yields several observations:

- The overall performance of the proposed parsers compares favorably with the state of the art. Both the C and D parsers outperform the parser of Hernault et al. (2010), and perform comparably to the parser of Joty et al. (2013). The recent work of Joty et al. (2014), which uses a considerably more complex architecture based on reranking, outperforms our parsers by 1.8 F_1 points.

- In general, the C parser performs better than D on all metrics. This is to be expected

[1] Interleaving the two classifiers in this iterative procedure guarantees that the classifiers have access to features extracted from the discourse subtrees constructed in previous iterations.

[2] `http://nlp.stanford.edu/software/corenlp.shtml`

[3] `http://www.maltparser.org`

	Manual EDUs	Predicted EDUs		
	F_1	P	R	F_1
D_{malt}	54.3	48.3	47.5	47.9
$D_{stanford}$	55.2	49.1	48.5	48.8
C	55.5	**49.2**	**48.5**	**48.9**
C – dep	55.5	47.9	47.6	47.7
C – const	53.7	47.7	47.0	47.3
C – coref	55.2	49.0	48.3	48.7
C – const – coref	53.9	47.9	47.2	47.5
Hernault 2010	54.8	47.7	46.9	47.3
Joty 2013	55.8	–	–	–
Joty 2014	**57.3**	–	–	–

Table 1: Performance of the two discourse parsers: one relying on constituent-based syntactic parsing (C), and another using a dependency parser (D). We report end-to-end results on the 18 relations with nuclearity information used by (Hernault et al., 2010; Feng and Hirst, 2012), using both manual segmentation of text into EDUs (left table block), and EDUs predicted by the parser (right block). We used the Precision/Recall/F_1 metrics introduced by Marcu (2000). The ablation test removes various feature groups: features extracted from the dependency representation (dep), features from constituent syntax (const), and coreference features (coref). We compare against previous work that reported end-to-end performance of their corresponding approaches (Hernault et al., 2010; Joty et al., 2013; Joty and Moschitti, 2014).

considering that C uses both constituent- and dependency-based features, and coreference information. However, the improvement is small (e.g., 0.2 F_1 points when gold EDUs are used) and the D parser is faster: it processes the entire test dataset in 88 seconds (at an average of 2.3 seconds/document) vs. 224 seconds for C.[4] For comparison, the (Feng and Hirst, 2012) discourse parser processes the same dataset in 605 seconds.

- The comparison of the two configurations of the dependency-based parser ("D_{malt}" vs. "$D_{stanford}$") indicates that the parser of Chen and Manning (2014) yields better RST parsing performance than the Malt parser, e.g., by 0.9 F_1 points when predicted EDUs are used.

- The ablation test in rows 4–5 of the table indicate that the two syntactic representations complement each other well: removing dependency-based features (the "C – dep" row) drops the F_1 score for predicted EDUs by 1.2 points (because of the worse EDU segmentation); removing constituent-based features ("C – const") drops performance by 1.6 F_1 points.

- Feature wise, the "C – const – coref" system is equivalent to D, but with dependency parsing performed by converting the constituent trees produced by the Stanford parser to dependencies, rather than direct dependency parsing. It is interesting to note that the performance of this system is lower than both configurations of the D parser, suggesting that direct dependency parsing with a dedicated model is beneficial.

- The "C – coref" ablation experiment indicates that coreference information has a small contribution to the overall performance (0.3 F_1 points when gold EDUs are used). Nevertheless, we find this result exciting, considering that this is a first attempt at using coreference information for discourse parsing.

4 Usage

With respect to usage, we adhere to the simplicity principles promoted by Stanford's CoreNLP, which introduced a simple, concrete Java API rather than relying on heavier frameworks, such as UIMA (Ferrucci and Lally, 2004). This guarantees that a user is "up and running in ten minutes or less", by "doing one thing well" and "avoiding over-design" (Manning et al., 2014). Following this idea, our API contains two `Processor` objects, one for each discourse parser, and a single method call, `annotate()`, which implements the complete analysis of a document (represented as a `String`), from tokenization to discourse parsing.[5] Figure 1 shows sample API usage. The `annotate()` method produces a `Document` object, which stores all NLP annotations: tokens, part-of-speech tags, constituent trees, dependency graphs, coreference relations, and discourse trees.

[4]These times were measured on a laptop with an i7 Intel CPU and 16GB of RAM. The times include end-to-end execution, including model loading and complete preprocessing of text, from tokenization to syntactic parsing and coreference resolution.

[5]Additional methods are provided for pre-existing tokenization and/or sentence segmentation.

```
import edu.arizona.sista.processors.corenlp._
import edu.arizona.sista.processors.fastnlp._
//
// CoreNLPProcessor:
// - syntax/coref with CoreNLP;
// - constituent-based RST parser.
// FastNLPProcessor:
// - syntax with Malt or CoreNLP.
// - dependency-based RST parser.
//
val processor = new CoreNLPProcessor()
val document = processor.annotate(
  "Tandy Corp. said it won't join U.S.
   Memories, the group that seeks to battle
   the Japanese in the market for computer
   memory chips.")
println(document.discourseTree.get)
```

Figure 1: Minimal (but complete) code for using the discourse parser. Use `CoreNLPProcessor` for the constituent-based RST parser, and `FastNLPProcessor` for the dependency-based discourse parser. Other than the different constructors, the APIs are identical.

The `DiscourseTree` class is summarized in Figure 2.

The code for the two parsers is available on GitHub, and is also packaged as two JAR files in the Maven Central Repository (one JAR file for code, and another for the pre-trained models), which guarantees that others can install and use it with minimal effort. For code and more information, please see the project's GitHub page: https://github.com/sistanlp/processors.

5 Visualization of Discourse Trees

We accompany the above Scala library with a web-based visualization tool that runs the two parsers in parallel and visualizes the two outputs for the same text side by side. This allows the users to: (a) directly compare the runtimes of the two systems in realtime for arbitrary texts; (b) analyze the qualitative difference in the outputs of two parsers; and (c) debug incorrect outputs (e.g., is the constituent tree correct?). Figure 3 shows a screenshot of this visualization tool.

The visualization tool is implemented as a client-server Grails[6] web application which runs the parsers (on the server) and collects and displays the results (on the client side). The application's client-side code displays both the discourse trees and syntactic information using Dagre-d3[7], a D3-based[8] renderer for the Dagre graph layout engine.

```
class DiscourseTree (
  /** Label of this tree, if non-terminal */
  var relationLabel:String,
  /** Direction of the relation,
    * if non-terminal. It can be:
    * LeftToRight, RightToLeft,
    * or None. */
  var relationDir:RelationDirection.Value,
  /** Children of this non-terminal node */
  var children:Array[DiscourseTree],
  /** Raw text attached to this node */
  val rawText:String,
  /** Position of the first token in the
    * text covered by this discourse tree */
  var firstToken: TokenOffset,
  /** Position of the last token in the
    * text covered by this discourse tree;
    * this is inclusive! */
  var lastToken: TokenOffset
)
```

Figure 2: Relevant fields in the `DiscourseTree` class, which stores the RST tree produced by the parsers for a given document. The token offsets point to tokens stored in the `Document` class returned by the `annotate()` method above.

6 Conclusions

This work described the design, development and the resulting open-source software for a parsing framework for Rhetorical Structure Theory. Within this framework, we offer two parsers, one built on top of constituent-based syntax, and the other that uses dependency-based syntax. Both parsers obtain state-of-the-art performance, are fast, and are easy to use through a simple API.

In future work, we will aim at improving the performance of the parsers using joint parsing models. Nevertheless, it is important to note that RST parsers have already demonstrated their potential to improve natural language processing applications. For example, in our previous work we used features extracted from RST discourse relations to enhance a non-factoid question answering system (Jansen et al., 2014). In recent work, we showed how to use discourse relations to generate artificial training data for mono-lingual alignment models for question answering (Sharp et al., 2015).

[6] https://grails.org

[7] https://github.com/cpettitt/dagre-d3

[8] http://d3js.org

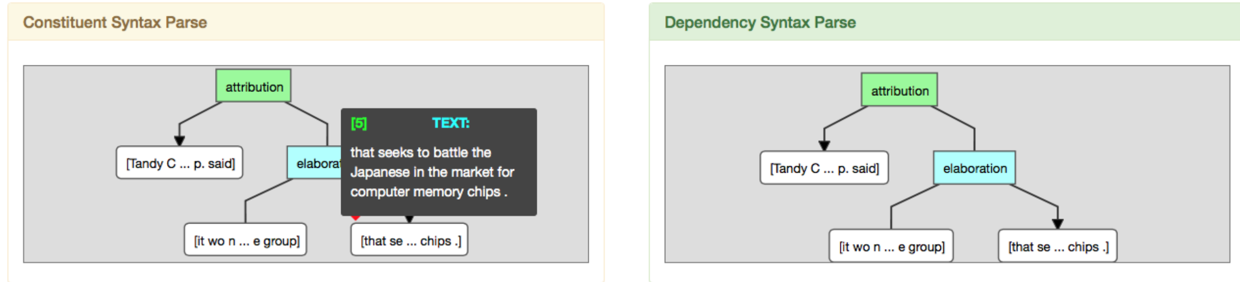

Figure 3: Screenshot of the discourse parser visualization tool for the input: *"Tandy Corp. said it won't join U.S. Memories, the group that seeks to battle the Japanese in the market for computer memory chips."* The left pane shows the output of the C parser; the right one shows the output of the D parser. Hovering with the cursor over a tree node shows its full content. Not shown here but included in the visualization: syntactic analyses used by the two parses and runtimes for each component (from tokenization to syntactic analysis).

Acknowledgments

This work was funded by the DARPA Big Mechanism program under ARO contract W911NF-14-1-0395.

References

L. Carlson, D. Marcu, and M. E. Okurowski. 2003. Building a Discourse-Tagged Corpus in the Framework of Rhetorical Structure Theory. In Jan van Kuppevelt and Ronnie Smith, editors, *Current Directions in Discourse and Dialogue*, pages 85–112. Kluwer Academic Publishers.

D. Chen and C. D. Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

M.-C. de Marneffe, B. MacCartney, and C. D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC)*.

V. W. Feng and G. Hirst. 2012. Text-level discourse parsing with rich linguistic features. In *Proceedings of the Association for Computational Linguistics*.

D. Ferrucci and A. Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10:327–348.

H. Hernault, H. Prendinger, D. duVerle, and M. Ishizuka. 2010. HILDA: A discourse parser using support vector machine classification. *Dialogue and Discourse*, 1(3):1–33.

P. Jansen, M. Surdeanu, and P. Clark. 2014. Discourse complements lexical semantics for non-factoid answer reranking. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*.

S. Joty and A. Moschitti. 2014. Discriminative reranking of discourse parses using tree kernels. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

S. Joty, G. Carenini, R. Ng, and Y. Mehdad. 2013. Combining intra- and multi-sentential rhetorical parsing for document-level discourse analysis. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*.

D. Klein and C. D. Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL)*.

W. C. Mann and S. A. Thompson. 1988. Rhetorical structure theory: Toward a functional theory of text organization. *Text*, 8(3):243–281.

C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*.

D. Marcu. 2000. *The Theory and Practice of Discourse Parsing and Summarization*. MIT Press.

R. Sharp, P. Jansen, M. Surdeanu, and P. Clark. 2015. Spinning straw into gold: Using free text to train monolingual alignment models for non-factoid question answering. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL HLT)*.

R. Soricut and D. Marcu. 2003. Sentence level discourse parsing using syntactic and lexical information. In *Proceedings of the Human Language Technology and North American Association for Computational Linguistics Conference*.