Structuring Documents Efficiently

Robert Marshall, Steven Bird and Peter J. Stuckey*
Department of Computer Science and Software Engineering
University of Melbourne, Victoria 3010, Australia
{robertgm,sb,pjs}@csse.unimelb.edu.au
*NICTA Victoria Laboratory

Abstract

Documents are typically marked up to enable rendering and to facilitate reuse. However, retargetting a document often requires pervasive changes to the markup. Power et al. have proposed a new level of representation called *document structure* which captures just those aspects of graphical organisation that are significant for conveying meaning. These document structures can be generated automatically from *rhetorical structures*, abstract representations of the meaning of a text. The mapping is highly indeterminate, being governed by a large number of interacting constraints. We present a constraint programming approach to the problem, and report on early experiments with an implementation in Prolog.

1 Introduction

Documents are typically marked up to enable rendering and to facilitate reuse. Simple adjustments in layout and style can be implemented without touching the source document. However, retargetting a document often requires pervasive changes to the markup itself — e.g. changing a bulleted list to an inline list — a fact which suggests that the markup is not sufficiently abstract.

Recent research by Power et al. (2003) has identified a new level of representation called *document structure* which captures just those aspects of graphical organisation that are significant for conveying meaning. These representations can be generated automatically from *rhetorical structures*, abstract representations of the meaning of a text.

Different document structures corresponding to the same rhetorical structures represent different realizations of the text. We may want to consider different document structures for the same rhetorical structure for a number of reasons. One document structure may be easier to understand than another for the same rhetorical structure. For example a bulleted list usually provides a clearer separation of which items form part of the list than an inline comma separated list. Alternatively, some document structures may have a much more compact representation which may be essential for viewing on a PDA screen with limited size, a constraint that is irrelevant when viewing the same document on a large screen. In this paper we concentrate on finding document structures that minimize the number of "defects", a somewhat artificial measure of comprehensibility flaws from Power et al. (2003).

The mapping from rhetorical structure to document structure is highly indeterminate, being governed by a large number of interacting constraints. The existing implementation method is to generate all possible document structures corresponding to a given rhetorical structure, evaluate them against the constraints, and find the best solution (minimizing "defects"). However, this method does not scale since the search space is exponential in the size of the document.

We present a constraint programming approach to the problem, in which an objective function is stated in advance in order to greatly prune the search space. We report on early experiments with an implementation in SICStus Prolog.

This paper is organized as follows. First, in §2 we review the work of Power et al. (2003) on document structure, the mapping from rhetorical structure, and the scoring metrics. Next, in §3 we report on our constraint programming implementation, before reporting on the results of our experimental work in §4. We close by presenting our conclusions and identifying issues for future investigation.

2 Review of document structure

Natural language generation systems produce formatted text from abstract meaning representations. Power et al. (2003) have demonstrated that the graphical organization of this text – e.g. its headings, fonts, and linebreaks – can convey meaning. They propose to capture those aspects of graphical organisation which carry meaning using a new level

Proceedings of the Australasian Language Technology Workshop 2005, pages 120–126,
Sydney, Australia, December 2005.

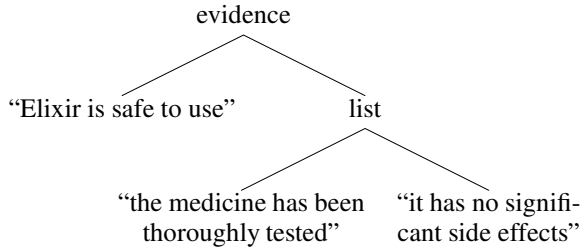
Elixir is safe to use since

- the medicine has been thoroughly tested,
- it has no significant side effects.

(a) Bulleted List Format

The medicine has been thoroughly tested; it has no significant side-effects. Therefore, Elixir is safe to use.

(b) Inline List Format



(c) Common Rhetorical Structure

Figure 1: Two formats for an excerpt from a patient information leaflet, and the underlying rhetorical structure, from (Power et al., 2003).

of representation called *document structure*. Power et al. (2003) define document structure as "the organization of a document into graphical constituents like sections, paragraphs, sentences, bulleted lists, and figures; it also covers some features within sentences, including quotation and emphasis." They argue that the same document structure can be rendered into formatted text in multiple ways, as illustrated in the patient information leaflet text in Figure 1(a) and 1(b).

Rhetorical structures represent the meaning of a text independently of its realization as a document (see Figure 1(c)). Power et al. (2003) use logic programming techniques to convert rhetorical structures into document structures which realise a given rhetorical structure. This generates a large number of candidates, and these are evaluated for conformance to a variety of heuristics such as "satellite precedes nucleus."

2.1 Rhetorical structure

Rhetorical structure is intended to represent the meaning of a text independently of its realisation as a document (Mann, 1999). Two documents with different formatting, using different words—or even written in different languages—could have the same rhetorical structure.

New World Guide to Wines	New World Guide to Wines

Figure 2: Example of how layout affects meaning (Power et al., 2003)

Rhetorical structure is expressed by rhetorical relations, which describe the relationship between facts, or between other rhetorical relations. A rhetorical relation consists of a type and some parameters, e.g. `justify(A, B)` has type `justify` and parameters `A` and `B`, and has the interpretation that we believe `A` to be true based on evidence `B`. Parameters may be other rhetorical relations, or they may be elementary propositions which are not dependent on any other information. Rhetorical relations are divided into two categories. Nucleus-satellite relations generally take two parameters: the nucleus is the central piece of information, and the satellite supports it (e.g. `justify`). Multinuclear relations can take many parameters, each of which is of equal importance to the others (e.g. `list`).

2.2 Document structure

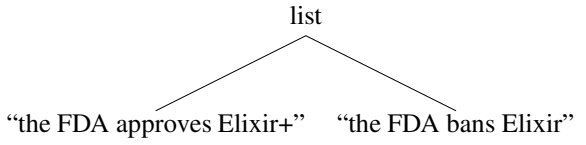
The theory of document structure was originally proposed by Power et al. (2003). The central insight is that the layout of a document affects its meaning. A simple illustration of this point appears in Figure 2 (Scott, pers. comm.). Power et al. (2003) contend that all texts have layout, even if it is very basic.

Document structure is related to markup languages such as HTML and LaTeX, which allow us to describe the structure of a document independently of its presentation. However, such markup languages are only suggestive of document structure, for they inconveniently blur the distinction between descriptive and presentational markup—c.f. (Coombs et al., 1987).

The formal theory of document structure is based around document units. *Document units* are elements such as phrases, sentences, paragraphs and chapters. Each unit can be made up of one or more sub-units, giving rise to a tree structure.

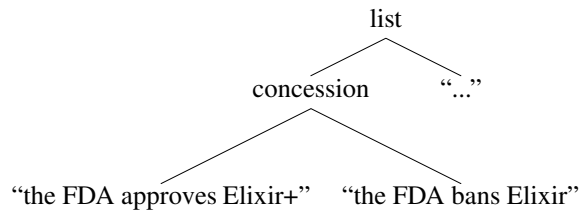
A document unit is represented by the following four variables: level, indentation, position, and connective.

Level: This represents the level of importance of the unit in the realised document. It is an integer from zero to five, corresponding to a realisation as



- The FDA approves Elixir+, and
- the FDA bans Elixir

(a) Bulleted List Format



- The FDA approves Elixir+, but the FDA bans Elixir.
- ...

(b) Inline List Format

Figure 3: Two different rhetorical structures that lead to level zero and indentation one for the elementary units.

a phrase (comma-terminated), clause (semicolon-terminated), sentence, paragraph, section, or chapter, respectively. In Figure 1, the children of the `list` relation are level zero in the first realisation, and level one in the second realisation.

Indentation: A document unit may be indented from its parent, allowing such items as bulleted lists. Power et al. (2003) represent indentation as an integer, indicating how many times a unit has been indented relative to the root node. In Figure 1, the list is indented in the first realisation, but not the second. However, we contend that the most important feature is not the total amount of indentation in an element, but rather whether it is indented relative to its parent.

If an element is indented relative to its parent, it must be realised on its own line and with its own bullet point, whereas if its parent is at the same level, it can be part of a larger structure, which is all indented. Consider the two rhetorical structures and realisations shown in Figures 3(a) and 3(b).

In both cases, the elementary units have level zero and indentation one, meaning that they are realised as phrases which are terminated with a comma, and are indented once from the base. However, in the first case they are rendered on separate lines, while in the second they are on the same line.

The reason for this is that in the first case, they are list elements, while in the second, they are both part of a concession. It is clear that when list elements are indented, they should each be realised on a separate line, but this is not actually spelled out in the document structure. While this can be inferred from the relative indentations of the elementary units to their parents, it adds unwanted complexity to the implementation, as well as violating the modularity of the document units.

Accordingly, we define `IndentationHere` as a binary variable, indicating whether a given document unit is indented relative to its parent. Note that the indentation at any given node is just the sum of the `IndentationHere` variables for it and each of its ancestors.

Position: The position variable indicates the order in which the nucleus and satellite occur. The two realisations in Figure 1 show alternate positionings of the nucleus and satellite of the `evidence` relation. Because all of the children of a multinuclear relation are of the same importance, we have the freedom to reorder them in any way. But for the purposes of this paper these reorderings will not change the evaluation of the realised document, hence they are always rendered in the same order that they occur in the rhetorical structure.

Connective: A discourse connective is always used to denote the rhetorical relationship between different document units. The only exception is for the leaf nodes of the document structure, which represent basic propositions. Figure 1 shows two different connectives for the `evidence` relation.

2.3 Discourse connectives

Discourse connectives indicate the rhetorical relationships between consecutive spans of text in a discourse.

Power et al. (1999) model discourse connectives with four attributes: relation, locus, phrase and syntactic type. These specify: the rhetorical relation which the connective represents; whether it should be attached to the nucleus or satellite; the text which realises it; and the syntactic restrictions on where the connective can be used. They side-step the question of whether a discourse connective should be realized overtly, insisting that there be a one-to-one relationship between rhetorical relationships in the rhetorical structure and discourse connectives in the generated text.

They define three syntactic types: parenthetical, coordinating or subordinating. Coordinating

The FDA bans Elixir; however, the FDA approves Elixir+.

(a) A parenthetical connective

Although the FDA bans Elixir, the FDA approves Elixir+.

(b) A coordinating connective

The FDA bans Elixir, but the FDA approves Elixir+.

(c) A subordinating connective

Figure 4: Examples of the different types of discourse connectives

connectives force the children to be of level zero. Subordinating and parenthetical connectives require that the satellite to appear before the nucleus, with parenthetical connectives additionally forcing the children to be of level greater than zero.

For example, the rhetorical relation *concession* can be realised by the discourse connectives *however*, *although* and *but*. These connectives are of parenthetical, coordinating and subordinating types, respectively. We give three different realisations of the same rhetorical structure in Figure 4. Note that the order of the two elements is reversed for the coordinating connective, and the different levels (sentences, phrases or clauses) used in each example.

2.4 Generating a document structure

Power et al. (2003) examine the task of determining a document structure from a given rhetorical structure, which they call "document structuring" (Power, 2000; Power et al., 2003), and implement it in a system called ICONOCLAST (Integrating Constraints on Layout and Style). The input consists of a collection of simple propositions organized into a rhetorical structure tree. The *document structurer* arranges these into a coherent collection of paragraphs, text-sentences and the like. This is then converted into an actual document by a *syntactic realiser*.

Each node on the rhetorical structure tree corresponds to a node on the document structure tree. The four variables associated with the node are constrained in various ways. First, the level of a child unit must be less than or equal to that of its parent, and equal to that of its siblings. The only exception is when the child unit is indented, in which case its level is independent of its parent, although it must still be equal to that of its siblings. Second, the indentation of a child unit must be either equal to or one greater than that of its parent. The positions of siblings must obviously be distinct. Third, the connective must realise the rhetorical relation. Finally, the type of the connective places additional constraints on the level and position of the children of the current unit, which must be satisfied.

Power et al. (2003) implement this process using logic programming. The different choices in mapping the rhetorical structure to the document structure are represented by constraints in the logic program, with the exception of discourse connectives. These set choicepoints, as in a standard logic program.

Moreover, they evaluate the defects of a structure (as described in the following section) after it has been completely generated. These factors force them to generate all possible document structures for a given rhetorical structure, using both a branch-and-bound and standard Prolog backtracking, before they can choose the best structure.

There are exponentially many (in the number of nodes in the rhetorical structure) candidate documents for any rhetorical structure. The number of candidates quickly blows up so that the approach is impractical for rhetorical structures with more than 10 nodes.

2.5 Scoring document structures

In order to choose between different document structures, Power et al. (2003) generate all valid document structures, then score each one by counting its undesirable features. The larger the score, the worse the structure. They give an example of a rhetorical structure with one relation between two facts, with seven renderings, and one with three relations and four facts, which has 58 renderings. They identify six kinds of undesirable features, which we describe below.

Nucleus before satellite: The nucleus appears before the satellite. This is undesirable, according to Power et al. (2003), because of psycholinguistic evidence which suggests that the more important information should be placed at the end of a sentence (and this is the common practise in English). The first realisation in Figure 1 shows a nucleus before satellite defect.

Left-branching structure: The left side of the document structure tree branches, while the right side does not. The second realisation in Figure 1 contains a left-branching structure, as the `list` child of the root node is realised before the `elementary` child.

Lost rhetorical grouping: The document structure can conflate distinct levels of the rhetorical structure. That is, a child and parent (and possibly higher-level ancestor) nodes in the rhetorical structure can be realised at the same level in the document structure. This makes it more difficult to infer the underlying structure from the text. For example, in Figure 4 the second and third examples contain this defect, while the first does not.

Single-sentence paragraph: A paragraph contains only one sentence.

Oversimple text-clauses: A sentence is composed of two text-clauses (clauses separated by a semi-colon), each of which expresses a single elementary proposition. The first sentence of the second realisation in Figure 1 contains is an oversimple text-clause.

Repeated discourse connective: A single rhetorical structure is represented twice in the document structure, by the same connective, and in such a way that one of the occurrences is on a descendent node of the other.

2.6 Summary

Rhetorical structures represent the meaning of a text independently of its realisation. Document structures include those realisation details which are relevant to its meaning. A document structure can be generated from a rhetorical structure using a constraint-based approach as described in (Power et al., 2003), but the current implementation is too inefficient to be used on large rhetorical structures.

3 Constraint Programming Implementation

Constraint programming (Marriott and Stuckey, 1998) allows us to specify relationships between variables, without having to actually calculate the values that the variables may take. In this work we use constraint logic programming over finite domains (Van Hentenryck, 1989), which augments a traditional logic programming language with the capacity to apply mathematical constraints over Boolean and integer variables over fixed ranges.

In order to express the complex constraints that arise in defining document structure and defects we make use of *reified constraints*. These allow us to attach a Boolean variable to the result of a constraint. For example $B \Leftrightarrow X > 3$ is a constraint that holds if $B = 1$ and X takes a value greater than 3, or $B = 0$ and X takes a value less than or equal to 3.

We implemented a document structurer in SICStus Prolog, using the same constraint model as Power et al. (2003). However, our program differs in that it simultaneously evaluates both the required constraints to create the document structure, and the constraints required to find the defects. It produces as output both a document structure and a count of its defects. By contrast, the document structurer of Power et al. (2003) produces a document structure which must then be evaluated. Moreover, our document structure contains positioning information for each word in the output.

Mapping a rhetorical structure to a document structure involves a large number of independent choices. As the rhetorical structure grows, the number of corresponding document structures grows exponentially. We would like to choose only those with less than a fixed number of defects, or perhaps the one having the fewest defects. Generating all possible document structures, to only choose one, is both unnecessary and impractical as the number of candidates grows so rapidly.

Our constraint programming model constrains the count of defects while we are generating document structures. This allows us to stop generating a structure as soon as it has more defects than the upper limit (for minimization this limit is defined by the number of defects in the best answer so far). Moreover, because a partially generated document structure may in fact lead to several document structures, each of which will have at least as many defects as the partially generated structure, we can prune entire branches from the search tree.

We accomplish this by expressing the rules for the defects as constraints. Each constraint is evaluated at every node on the document structure tree, indicating whether or not the defect occurs at that point. We simply sum them all to obtain the total count of defects found so far. The constraints are stored for each node, along with the other document structure parameters. The rules for the nucleus before satellite and left-branching defects are given below. In the following, PN and PS are the positions of the nucleus and satellite nodes, while EN and ES are Boolean variables indicating whether the nucleus and satellite are elementary.

$$PN < PS \;\Rightarrow\; NucleusBeforeSatellite$$

$$\begin{array}{l}(EN \wedge \neg ES \wedge PS < PN) \\ \vee (ES \wedge \neg EN \wedge PN < PS)\end{array} \Rightarrow LeftBranching$$

Because the total number of solutions to be checked is exponential in the size of the structure, the problem can easily become intractable for large

structures. Therefore, we set a maximum number of assignments which can be made, and simply fail to find any solutions after this point, and return the best solution which has been found thus far.

3.1 Improving the searching efficiency

Expressing the entire model in a constraint-based form allows us to search for a solution with the fewest defects more quickly than a non constraint-based implementation. However, the search space is still very large, so we have implemented several other techniques to improve the searching.

Labelling order: Because labelling one type of variable will affect the domain of others, the order in which variables are labelled can make a significant difference. For example, if we set the connective to a coordinating type, then the levels of any child nodes must be zero, while the converse does not necessarily hold. Therefore, at least in this case, it will be more efficient to label the connective before the level. The relationships between the variables are relatively complicated, so it is unclear what the best labelling order is without conducting some experiments. We test two methods for ordering variables. One approach is to label the document structure by traversing the tree, depth-first. We make one pass for each variable type. For example, we might first label all the `Indentation` variables, then all the level variables, and so on.

The other method we use is known as first fail labelling (Haralick and Elliott, 1980). Using this method, we label the variables in order of domain size, starting with the smallest. This is intended to reduce the amount of branching which occurs, and prune large regions of the search space as early as possible. Either of these methods can be used in conjunction with any of the following strategies.

Iterating through the goal variable: One technique which can improve efficiency is to iterate through the variable to be minimised, starting from zero, attempting to find a solution at each value.

While it may seem inefficient, this procedure can sometimes be faster than a simple search, because the goal variable is constrained to just one value for each call. Moreover, as soon as one of the calls succeeds, we are guaranteed to have the minimum value of the goal variable. Using a simple minimisation search, it is generally not immediately clear if a given solution is in fact the minimum, requiring further search.

Limited-discrepancy search: Limited-discrepancy searching (Harvey and Ginsberg, 1995) requires a heuristic, which guesses the best value for each labelling choice. Any choice which differs from the heuristic is called a discrepancy. Using this method, we search as before, but with the number of discrepancies limited to some upper bound. This has the effect of reducing the search space, potentially rejecting many possible solutions, but also allowing for a much faster search.

Our heuristic works as follows. If the level variable is not already set, we choose the second largest possible value for it. We are attempting to avoid two nodes having the same level, thereby incurring a lost rhetorical grouping defect. The larger the value, the more values will be available for descendant nodes to use, but the largest possible value will generally be the same as its parent.

We always choose to indent multinuclear relations, thereby allowing children to take any available level, and place the satellite first, avoiding a nucleus before satellite defect.

Subordinating connectives place the least restrictive constraints on the connected nodes, followed by parenthetical and then coordinating connectives. In particular, we do not wish to use a coordinating connective, as it forces the nucleus and satellite (and hence any child nodes they may have) to be of level zero, and will incur a lost rhetorical grouping defect on all further child nodes. Hence, we choose a subordinating connective if one is available; otherwise a parenthetical one, and finally a coordinating connective.

Unfortunately, it is unclear how many discrepancies to allow when using limited-discrepancy searching. We have tested our implementation using a maximum of both 3 and 10 discrepancies, but these choices are quite arbitrary.

Optimistic partitioning: Optimistic partitioning (Prestwich and Mudambi, 1995) requires no further information except the total range of the search space, which is from zero to the total number of nodes multiplied by 6 (the number of defect types). We split the search space in two and search for a solution in the first half of the space. If one is found, we partition again, using this solution as the new upper bound. If there is no solution in the first half, we search again, in the upper half of the search space. If there is a solution in the upper half, we partition again, using the midpoint and the solution in the second half of the search space as the lower and upper bounds, respectively.

In either case, we store the current solution as the best one thus far, and if at any time we fail to find a solution in the given range, then we know that the previous best solution is the overall minimum.

Restricting the number of assignments: A final, crude method of improving performance is to simply restrict the total number of assignments which can be made. Once we have reached this limit, we can simply return the best solution found up to that point. While it is preferable to find the best solution, the search space is exponential, and there may be cases in which this problem is intractable.

4 Results

Our testing data comes from Marcu's rhetorical structure corpus (Marcu, 2000). We have chosen a particular article from his corpus, and rendered the individual rhetorical structures which make up this article.

The rhetorical relations used in this corpus generally do not specify corresponding discourse connectives, so we used random placeholders. While this produces rather ugly output, it is sufficient for the purposes of testing the efficiency of the various labeling techniques.

As described in §3.1, we iterate through the document structure tree several times, once for each variable type. We have tested all 24 possible variable orderings. The numbers given in Table 1 represent the number of variable assignments required to find the optimal solution, divided by the number of variable assignments required by the most efficient variable ordering for the same structure.

The order in which variables are labelled has a large impact on the efficiency of the search, by a factor of about 4 on our testing data. The fastest method seems to be to label the $Level$ and $Position$ variables first, and then either of the other two. For the next table, we used the "Level,Position,IndentationHere,ConnectiveIndex" labelling strategy.

We have tested all of the search strategies discussed in the previous section, using both the tree-traversal and first-fail methods of choosing which variable to label. We forced all searches to terminate after 100,000 assignments and return the best solution found at that point, in order to prevent inordinately long execution times. Results are shown in Tables 2 and 3. We show the number of nodes in the rhetorical structure, **Size**, as well as the minimal number of defects possible (if known), **Min Defects**. The remaining columns give the number of assignments required to find the best solution, except the column **First** which gives the number of assignments to find the first solution with simple search. **Basic** is the simple minimization search, **Iterating** iterates the defect count upwards from 0, **OP** uses optimistic partitioning, **FF** is first fail search, and **LD(n)** is limited discrepancy search with a max discrepancy of n. If the best solution found is not optimal it is shown in parentheses after the number of assignments figure. A dash indicates no solution was found with 100,000 assignments. Note that we have not normalised these results, as it is sometimes unclear which is the best solution; some strategies may perform faster than others but return a non-optimal solution, and some are terminated early for the sake of tractability.

For the search strategy, limited-discrepancy search and iterating through the defect variable (using tree-traversal labelling) seem to require the least number of assignments in order to find a solution. However, the best search strategy varies considerably, depending on the structure. However, if we reach the maximum number of assignments without finding an optimal solution, iteration cannot provide a sub-optimal solution, because by definition, the first solution it finds is the optimal one. For this reason, limited-discrepancy may be a better choice when realising large structures.

Limited-discrepancy search often does not find a solution when used with first-fail labelling. We believe that this is because first-fail labelling will choose labelling variables from all over the document structure. Therefore, once a discrepancy has been incurred, the next variable to be labelled may be from an entirely different part of the document structure. This may happen several times, causing several discrepancies to occur, before the choices which have been made can affect new ones.

By contrast, when using tree-traversal labelling, once a discrepancy has been incurred, the next variables to be labelled will come from the same node, or its children, which will be constrained by the choice which has just been made.

Tree-traversal labelling seems to outperform first-fail labelling in most other cases too, but much less dramatically.

Basic search performs more poorly than the other methods in most cases, as we might expect. Optimistic partitioning represents an improvement on this, but not as much as limited-discrepancy or iterative searching.

Finding the first solution is generally quite fast, but the first solution is almost never optimal. It is also interesting to note that even merely finding the first solution is often slower than finding the optimal limited-discrepancy solution, indicating that our heuristic is improving performance considerably.

Overall, the iterating approach followed by a limited discrepancy search when this fails to find the