The evolution of dominance constraint solvers

Alexander Koller and **Stefan Thater**
Dept. of Computational Linguistics
Saarland University, Saarbrücken, Germany
{koller,stth}@coli.uni-sb.de

Abstract

We describe the evolution of solvers for dominance constraints, a formalism used in underspecified semantics, and present a new graph-based solver using charts. An evaluation on real-world data shows that each solver (including the new one) is significantly faster than its predecessors. We believe that our strategy of successively tailoring a powerful formalism to the actual inputs is more generally applicable.

1 Introduction

In many areas of computational linguistics, there is a tension between a need for powerful formalisms and the desire for efficient processing. Expressive formalisms are useful because they allow us to specify linguistic facts at the right level of abstraction, and in a way that supports the creation and maintenance of large language resources. On the other hand, by choosing a more powerful formalism, we typically run the risk that our processing tasks (say, parsing or inference) can no longer be performed efficiently.

One way to address this tension is to switch to simpler formalisms. This makes processing more efficient, but sacrifices the benefits of expressive formalisms in terms of modelling. Another common strategy is to simply use the powerful formalisms anyway. This sometimes works pretty well in practice, but a system built in this way cannot give any runtime guarantees, and may become slow for certain inputs unpredictably.

In this paper, we advocate a third option: Use a general, powerful formalism, analyse what makes it complex and what inputs actually occur in practice, and then find a restricted fragment of the formalism that supports all practical inputs and can be processed efficiently. We demonstrate this approach by describing the evolution of solvers for *dominance constraints* (Egg et al., 2001), a certain formalism used for the underspecified description of scope ambiguities in computational semantics. General dominance constraints have an NP-complete satisfiability problem, but *normal* dominance constraints, which subsume all constraints that are used in practice, have linear-time satisfiability and can be solved extremely efficiently.

We describe a sequence of four solvers, ranging from a purely logic-based saturation algorithm (Koller et al., 1998) over a solver based on constraint programming (Duchier and Niehren, 2000) to efficient solvers based on graph algorithms (Bodirsky et al., 2004). The first three solvers have been described in the literature before, but we also present a new variant of the graph solver that uses caching to obtain a considerable speedup. Finally we present a new evaluation that compares all four solvers with each other and with a different underspecification solver from the LKB grammar development system (Copestake and Flickinger, 2000).

The paper is structured as follows. We will first sketch the problem that our algorithms solve (Section 2). Then we present the solvers (Section 3) and conclude with the evaluation (Section 4).

2 The Problem

The problem we use to illustrate the progress towards efficient solvers is that of enumerating all

readings of an *underspecified description*. Underspecification is a technique for dealing with the combinatorial problems associated with quantifier scope ambiguities, certain semantic ambiguities that occur in sentences such as the following:

(1) Every student reads a book.

This sentence has two different readings. Reading (2) expresses that each student reads a possibly different book, while reading (3) claims that there is a single book which is read by every student.

(2) $\forall x.\mathsf{student}(x) \rightarrow (\exists y.\mathsf{book}(y) \wedge \mathsf{read}(x,y))$

(3) $\exists y.\mathsf{book}(y) \wedge (\forall x.\mathsf{student}(x) \rightarrow \mathsf{read}(x,y))$

The number of readings can grow exponentially in the number of quantifiers and other scope-bearing operators occuring in the sentence. A particularly extreme example is the following sentence from the Rondane Treebank, which the English Resource Grammar (Copestake and Flickinger, 2000) claims to have about 2.4 trillion readings.

(4) Myrdal is the mountain terminus of the Flåm rail line (or Flåmsbana) which makes its way down the lovely Flåm Valley (Flåmsdalen) to its sea-level terminus at Flåm.
(Rondane 650)

Of course, this huge number of readings results not only from genuine meaning differences, but from the (quite reasonable) decision of the ERG developers to uniformly treat all noun phrases, including proper names and definites, as quantifiers. But a system that builds upon such a grammar still has to deal with these readings in some way.

The key idea of underspecification is now to not enumerate all these semantic readings from a syntactic analysis during or after parsing, but to derive from the syntactic analysis a single, compact *underspecified description*. The individual readings can be *enumerated* from the description if they are needed, and this enumeration process should be efficient; but it is also possible to eliminate readings that are infelicitous given knowledge about the world or the context on the level of underspecified descriptions.

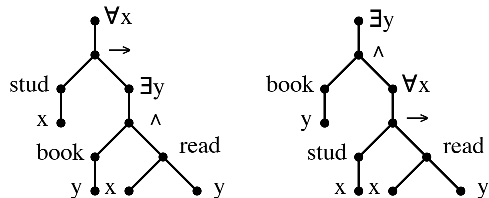

Figure 1: Trees for the readings (2) and (3).

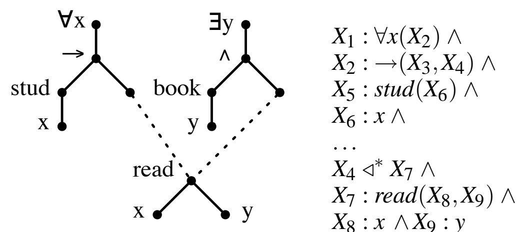

Figure 2: A dominance constraint (right) and its graphical representation (left); the solutions of the constraint are the two trees in Fig. 1.

Dominance constraints. The particular underspecification formalism whose enumeration problem we consider in this paper is the formalism of *dominance constraints* (Egg et al., 2001). The basic idea behind using dominance constraints in underspecification is that the semantic representations (2) and (3) can be considered as *trees* (see Fig. 1). Then a set of semantic representations can be characterised as the set of models of a formula in the following language:

$$\varphi ::= X{:}f(X_1,\ldots,X_n) \mid X \triangleleft^* Y \mid X \neq Y \mid \varphi \wedge \varphi$$

The *labelling atom* $X{:}f(X_1,\ldots,X_n)$ expresses that the node in the tree which is denoted by the variable X has the label f, and its children are denoted by the variables X_1 to X_n. *Dominance atoms* $X \triangleleft^* Y$ say that there is a path (of length 0 or more) from the node denoted by X to the node denoted by Y; and *inequality atoms* $X \neq Y$ require that X and Y denote different nodes.

Dominance constraints φ can be drawn informally as graphs, as shown in Fig. 2. Each node of the graph stands for a variable; node labels and solid edges stand for labelling atoms; and the dotted edges represent dominance atoms. The constraint represented by the drawing in Fig. 2 is *satisfied* by both trees shown in Fig. 1. Thus we can

use it as an underspecified description representing these two readings.

The two obvious processing problems connected to dominance constraints are *satisfiability* (is there a model that satisfies the constraint?) and *enumeration* (compute all models of a constraint). Because every satisfiable dominance constraint technically has an infinite number of models, the algorithms below solve the enumeration problem by computing *solved forms* of the constraint, which are finite characterisations of infinite model sets.

3 The Solvers

We present four different solvers for dominance constraints. As we go along, we analyse what makes dominance constraint solving hard, and what characterises the constraints that occur in practice.

3.1 A saturation algorithm

The first dominance constraint solver (Koller et al., 1998; Duchier and Niehren, 2000) is an algorithm that operates directly on the constraint as a logical formula. It is a *saturation algorithm*, which successively enriches the constraint using *saturation rules*. The algorithm terminates if it either derives a contradiction (marked by the special atom **false**), or if no rule can contribute any new atoms. In the first case, it claims that the constraint is unsatisfiable; in the second case, it reports the end result of the computation as a *solved form* and claims that it is satisfiable.

The saturation rules in the solver try to match their preconditions to the constraint, and if they do match, add their conclusions to the constraint. For example, the following rules express that dominance is a transitive relation, and that trees have no cycles:

$$\begin{array}{lcl} X \triangleleft^* Y \wedge Y \triangleleft^* Z & \rightarrow & X \triangleleft^* Z \\ X{:}f(\ldots,Y,\ldots) \wedge Y \triangleleft^* X & \rightarrow & \textbf{false} \end{array}$$

Some rules have disjunctive right-hand sides; if they are applicable, they perform a case distinction and add one of the disjuncts. One example is the *Choice Rule*, which looks as follows:

$$X \triangleleft^* Z \wedge Y \triangleleft^* Z \rightarrow X \triangleleft^* Y \vee Y \triangleleft^* X$$

This rule checks for the presence of two variables X and Y that are known to both dominate the same variable Z. Because models must be trees, this means that X and Y must dominate each other in some order; but we can't know yet whether it is X or Y that dominates the other one. Hence the solver tries both choices. This makes it possible to derive multiple solved forms (one for each reading of the sentence), such as the two different trees in Fig. 1.

It can be shown that a dominance constraint is satisfiable iff it is not possible to derive **false** from it using the rules in the algorithm. In addition, every model of the original constraint satisfies exactly one solved form. So the saturation algorithm can indeed be used to solve dominance constraints. However, even checking satisfiability takes nondeterministic polynomial time. Because all choices in the distribution rule applications have to be checked, a deterministic program will take exponential time to check satisfiability in the worst case.

Indeed, satisfiability of dominance constraints is an NP-complete problem (Koller et al., 1998), and hence it is likely that any solver for dominance constraints will take exponential worst-case runtime. At first sight, it seems that we have fallen into the expressivity trap: We have a formalism that allows us to model scope underspecification very cleanly, but actually computing with this formalism is expensive.

3.2 Reduction to Set Constraints

In reaction to this NP-completeness result, Duchier and Niehren (2000) applied techniques from *constraint programming* to the problem in order to get a more efficient solver. Constraint programming (Apt, 2003) is a standard approach to solving NP-complete combinatorial problems. In this paradigm, a problem is modelled as a formula in a logical constraint language. The program searches for values for the variables in the formula that satisfy the formula. In order to reduce the size of the search space, it performs cheap deterministic inferences that exclude some values of the variables *(propagation)*, and only after propagation can supply no further information it performs a non-deterministic case distinction *(distribution)*.

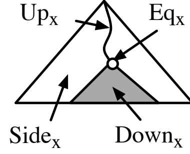

Figure 3: The four node sets

Duchier and Niehren solved dominance constraints by encoding them as *finite set constraints*. Finite set constraints (Müller and Müller, 1997) are formulas that talk about relations between (terms that denote) finite sets of integers, such as inclusion $X \subseteq Y$ or equality $X = Y$. Efficient solvers for set constraints are available, e.g. as part of the Mozart/Oz programming system (Oz Development Team, 2004).

Reduction to set constraints. The basic idea underlying the reduction is that a tree can be represented by specifying for each node v of this tree which nodes are dominated by v, which ones dominate v, which ones are equal to v (i.e. just v itself), and which ones are "disjoint" from v (Fig. 3). These four node sets are a partition of the nodes in the tree.

Now the solver introduces for each variable X in a dominance constraint φ four variables Eq_X, Up_X, $Down_X$, $Side_X$ for the sets of node variables that denote nodes in the respective region of the tree, relative to X. The atoms in φ are translated into constraints on these variables. For instance, a dominance atom $X \triangleleft^* Y$ is translated into

$$Up_X \subseteq Up_Y \wedge Down_Y \subseteq Down_X \wedge Side_X \subseteq Side_Y$$

This constraint encodes that all variables whose denotation dominates the denotation of X (Up_X) must also dominate the denotation of Y (Up_Y), and the analogous statements for the dominated and disjoint variables.

In addition, the constraint program contains various redundant constraints that improve propagation. Now the search for solutions consists in finding satisfying assignments to the set variables. The result is a search tree as shown in Fig. 4: The blue circles represent case distinctions, whereas each green diamond represents a solution of the set constraint (and therefore, a solved form of

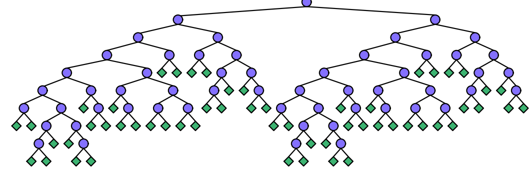

Figure 4: Search tree for constraint 42 from the Rondane Treebank.

the dominance constraint). Interestingly, all leaves of the search tree in Fig. 4 are solution nodes; the search never runs into inconsistent constraints. This seems to happen systematically when solving any constraints that come from underspecification.

3.3 A graph-based solver

This behaviour of the set-constraint solver is extremely surprising: The key characteristic of an NP-complete problem is that the search tree must necessarily contain failed nodes on some inputs. The fact that the solver never runs into failure is a strong indication that there is a fragment of dominance constraints that contains all constraints that are used in practice, and that the solver automatically exploits this fragment. This begs the question: What is this fragment, and can we develop even faster solvers that are specialised to it?

One such fragment is the fragment of *normal* dominance constraints (Althaus et al., 2003). The most important restriction that a normal dominance constraint φ must satisfy is that it is *overlap-free*: Whenever φ contains two labelling atoms $X{:}f(\ldots)$ and $Y{:}g(\ldots)$ (where f and g may be equal), it must also contain an inequality atom $X \neq Y$. As a consequence, no two labelled variables in a normal constraint may be mapped to the same node. This is acceptable or even desirable in underspecification: We are not interested in solutions of the constraint in Fig. 2 in which the quantifier representations overlap. On the other hand, the NP-completeness proof in (Koller et al., 1998) is no longer applicable to overlap-free constraints. Hence normal dominance constraints are a fragment that is sufficient from a modelling perspective, and possibly admits polynomial-time solvers.

Indeed, it can be shown that the satisfiability problem of normal dominance constraints can be

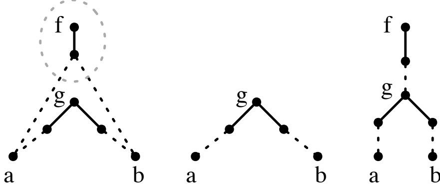

Figure 5: An example computation of the graph solver.

decided in linear time (Thiel, 2004), and the linear algorithm can be used to enumerate N solved forms of a constraint of size n in time $O(n^2N)$. We now present the simpler $O(n^2N)$ enumeration algorithm by Bodirsky et al. (2004).[1] Note that N may still be exponential in n.

Dominance Graphs. The crucial insight underlying the fast solvers for normal dominance constraints is that such constraints can be seen as *dominance graphs*, and can be processed using graph algorithms. Dominance graphs are directed graphs with two kinds of edges: *tree edges* and *dominance edges*. The graph without the dominance edges must be a forest; the trees of this forest are called the *fragments* of the graph. In addition, the dominance edges must go from *holes* (i.e., unlabelled leaves) of fragments to *roots* of other fragments. For instance, we can view the graph in Fig. 2, which we introduced as an informal notation for a dominance constraint, directly as a dominance graph with three fragments and two (dotted) dominance edges.

A dominance graph G which is a forest is called *in solved form*. We say that G' is a *solved form of* a graph G iff G' is in solved form, G and G' contain the same tree edges, and the reachability relation of G' extends that of G. Using this definition, it is possible to define a mapping between normal dominance constraints and dominance graphs such that the solved forms of the graph can serve as solved forms of the constraint – i.e., we can reduce constraint solving to graph solving.

By way of example, consider Fig. 5. The dominance graph on the left is not in solved form, because it contains nodes with more than one incoming dominance edge. By contrast, the other two dominance graphs are in solved form. Because the graph on the right has the same tree edges as the one on the left and extends its reachability relation, it is also a solved form of the left-hand graph.

[1] The original paper defines the algorithm for *weakly* normal dominance constraints, a slight generalisation.

GRAPH-SOLVER(G')

1 **if** G' is already in solved form
2   **then return** G'
3 *free* $\leftarrow$ FREE-FRAGMENTS(G')
4 **if** *free* $= \emptyset$
5   **then fail**
6 **choose** $F \in$ *free*
7 $G_1, \ldots, G_k \leftarrow$ WCCS($G' - F$)
8 **for** each $G_i \in G_1, \ldots, G_k$
9 **do** $S_i \leftarrow$ GRAPH-SOLVER(G_i)
10 $S \leftarrow$ Attach $S_1, \ldots, S_k$ under F
11 **return** S

Figure 6: The graph solver.

The algorithm. The graph-based enumeration algorithm is a recursive procedure that successively splits a dominance graph into smaller parts, solves them recursively, and combines them into complete solved forms. In each step, the algorithm identifies the *free fragments* of the dominance (sub-)graph. A fragment is free if it has no incoming dominance edges, and all of its holes are in different biconnected components of the undirected version of the dominance graph. It can be shown (Bodirsky et al., 2004) that if a graph G has any solved form and F is a free fragment of G, then G has a solved form in which F is at the root.

The exact algorithm is shown in Fig. 6. It computes the free fragments of a sub-dominance graph G' in line 3. Then it chooses one of the free fragments, removes it from the graph, and calls itself recursively on the weakly connected components $G_1, \ldots, G_k$ of the resulting graph. Each recursive call will compute a solved form S_i of the connected component G_i. Now for each G_i there is exactly one hole h_i of F that is connected to some node in G_i by a dominance edge. We can obtain a solved form for G' by combining F and all the S_i with dominance edges from h_i to the root of S_i for each i.

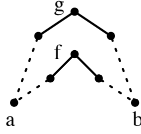

Figure 7: An unsolvable dominance graph.

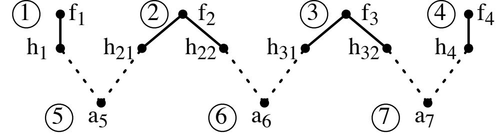

Figure 8: The chain of length 4.

Subgraph	Splits
$\{1,2,3,4,5,6,7\}$:	$\langle 1, h_1 \mapsto \{2,3,4,5,6,7\}\rangle$
	$\langle 2, h_{21} \mapsto \{1,5\}, h_{22} \mapsto \{3,4,6,7\}\rangle$
	$\langle 3, h_{31} \mapsto \{1,2,5,6\}, h_{32} \mapsto \{4,7\}\rangle$
	$\langle 4, h_4 \mapsto \{1,2,3,5,6,7\}\rangle$
$\{2,3,4,5,6,7\}$:	$\langle 2, h_{21} \mapsto \{5\}, h_{22} \mapsto \{3,4,6,7\}\rangle$
	$\langle 3, h_{31} \mapsto \{2,5,6\}, h_{32} \mapsto \{4,7\}\rangle$
	$\langle 4, h_4 \mapsto \{\mathbf{2,3,5,6,7}\}\rangle$
$\{1,2,3,5,6,7\}$:	$\langle 1, h_1 \mapsto \{\mathbf{2,3,5,6,7}\}\rangle$
	$\langle 2, h_{21} \mapsto \{1,5\}, h_{22} \mapsto \{3,6,7\}\rangle$
	$\langle 3, h_{31} \mapsto \{1,2,5,6\}, h_{32} \mapsto \{7\}\rangle$
$\{\mathbf{2,3,5,6,7}\}$:	$\langle 2, h_{21} \mapsto \{5\}, h_{22} \mapsto \{3,6,7\}\rangle$
	$\langle 3, h_{31} \mapsto \{2,5,6\}, h_{32} \mapsto \{7\}\rangle$
...	...

Figure 9: A part of the chart computed for the constraint in Fig. 8.

The algorithm is written as a nondeterministic procedure which makes a nondeterministic choice in line 6, and can fail in line 5. We can turn it into a deterministic algorithm by considering the nondeterministic choices as case distinctions in a search tree, as in Fig. 4. However, if the input graph G is solvable, we know that every single leaf of the search tree must correspond to a (different) solved form, because for every free fragment that can be chosen in line 6, there is a solved form that has this fragment as its root. Conversely, if G is unsolvable, every single branch of the search tree will run into failure, because it would claim the existence of a solved form otherwise. So the algorithm decides solvability in polynomial time.

An example computation of GRAPH-SOLVER is shown in Fig. 5. The input graph is shown on the left. It contains exactly one free fragment F; this is the fragment whose root is labelled with f. (The single-node fragments both have incoming dominance edges, and the two holes of the fragment with label g are in the same biconnected component.) So the algorithm removes F from the graph, resulting in the graph in the middle. This graph is in solved form (it is a tree), so we are finished. Finally the algorithm builds a solved form for the whole graph by plugging the solved form in the middle into the single hole of F; the result is shown on the right. By contrast, the graph in Fig. 7 has no solved forms. The solver will recognise this immediately, because none of the fragments is free (they either have incoming dominance edges, or their holes are biconnected).

3.4 A graph solver with charts

The graph solver is a great step forward towards efficient constraint solving, and towards an understanding of why (normal) dominance constraints can be solved efficiently. But it wastes time when it is called multiple times for the same subgraph, because it will solve it anew each time. In solving, for instance, the graph shown in Fig. 8, it will solve the subgraph consisting of the fragments $\{2,3,5,6,7\}$ twice, because it can pick the fragments 1 and 4 in either order.

We will now present a previously unpublished optimisation for the solver that uses caching to alleviate this problem. The data structure we use for caching (we call it "chart" below because of its obvious parallels to charts in parsing) assigns each subgraph of the original graph a set of *splits*. Splits encode the splittings of the graph into weakly connected components that take place when a free fragment is removed. Formally, a split for the subgraph G' consists of a reference to a fragment F that is free in G' and a partial function that maps some nodes of F to subgraphs of G'. A split is determined uniquely by G' and F.

Consider, by way of example, Fig. 9, which displays a part of the chart that we want to compute for the constraint in Fig. 8. In the entire graph G (represented by the set $\{1,\ldots,7\}$ of fragments), the fragments 1, 2, 3, and 4 are free. As a consequence, the chart contains a split for each of these four fragments. If we remove fragment 1 from G, we end up with a weakly connected graph G_1 containing the fragments $\{2,\ldots,7\}$. There is a dom-

```
GRAPH-SOLVER-CHART(G′)
 1  if there is an entry for G′ in the chart
 2     then return true
 3  free ← FREE-FRAGMENTS(G′)
 4  if free = ∅
 5     then return false
 6  if G′ contains only one fragment
 7     then return true
 8
 9  for each F ∈ free
10  do split ← SPLIT(G′, F)
11      for each S ∈ WCCS(G′ − F)
12      do if GRAPH-SOLVER-CHART(S) = false
13            then return false
14      add (G′, split) to the chart
15  return true
```

Figure 10: The graph solver with charts

inance edge from the hole h_1 into G_1, so once we have a solved form of G_1, we will have to plug it into h_1 to get a solved form of G; therefore G_1 is assigned to h_1 in the split. On the other hand, if we remove fragment 2 from G, G is split into two weakly connected components $\{1,5\}$ and $\{3,4,6,7\}$, whose solved forms must be plugged into h_{21} and h_{22} respectively.

We can compute a chart like this using the algorithm shown in Fig. 10. This recursive algorithm gets some subgraph G' of the original graph G as its first argument. It returns **true** if G' is solvable, and **false** if it isn't. If an entry for its argument G' was already computed and recorded in the chart, the procedure returns immediately. Otherwise, it computes the free fragments of G'. If there are no free fragments, G was unsolvable, and thus the algorithm returns **false**; on the other hand, if G' only contains one fragment, it is solved and we can immediately return **true**.

If none of these special cases apply, the algorithm iterates over all free fragments F of G' and computes the (unique) split that places F at the root of the solved forms. If all weakly connected components represented in the split are solvable, it records the split as valid for G', and returns **true**.

If the algorithm returns with value **true**, the chart will be filled with splits for all subgraphs of G that the GRAPH-SOLVER algorithm would have visited. It is also guaranteed that every split in the chart is used in a solved form of the graph. Extracting the actual solved forms from the chart is straightforward, and can be done essentially like for parse charts of context-free grammar.

Runtime analysis. The chart computed by the chart solver for a dominance graph with n nodes and m edges can grow to at most $O(n \cdot \mathrm{wcsg}(G))$ entries, where $\mathrm{wcsg}(G)$ is the number of weakly connected subgraphs of G: All subgraphs for which GRAPH-SOLVER-CHART is called are weakly connected, and for each such subgraph there can be at most n different splits. Because a recursive call returns immediately if its argument is already present in the chart, this means that at most $O(n \cdot \mathrm{wcsg}(G))$ calls spend more than the expected constant time that it takes to look up G' in the chart. Each of these calls needs time $O(m+n)$, the cost of computing the free fragments.

As a consequence, the total time that GRAPH-SOLVER-CHART takes to fill the chart is $O(n(n+m)\mathrm{wcsg}(G))$. Applied to a dominance *constraint* with k atoms, the runtime is $O(k^2 \mathrm{wcsg}(G))$. On the other hand, if G has N solved forms, it takes time $O(N)$ to extract these solved forms from the chart. This is a significant improvement over the $O(n(n+m)N)$ time that GRAPH-SOLVER takes to enumerate all solved forms. A particularly dramatic case is that of *chains* – graphs with a zig-zag shape of n upper and $n-1$ lower fragments such as in Fig. 8, which occur frequently as part of underspecified descriptions. A chain has only $O(n^2)$ weakly connected subgraphs and $O(n)$ edges, so the chart can be filled in time $O(n^4)$, despite the fact that the chain has $\frac{1}{n+1}\binom{2n}{n}$ solved forms (this is the n-th Catalan number, which grows faster than $n!$). The worst case for the chart size is shown in Fig. 11. If such a graph has n upper fragments, it has $O(2^n)$ weakly connected subgraphs, so the chart-filling phase takes time $O(n^2 2^n)$. But this is still dominated by the $N = n!$ solved forms that this graph has.

4 Evaluation

We conclude this paper with a comparative runtime evaluation of the presented dominance con-

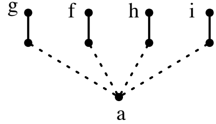

Figure 11: A worst-case graph for the chart solver.

		constraints	max. solved forms
Rondane		961	?
Nets		879	$2.4 \cdot 10^{12}$
Nets $< 10^6$ solved forms		852	997920
Solver		solvable	max. solved forms
Saturation	(§3.1)	757	10030
Set constraints	(§3.2)	841	557472
Graph	(§3.3)	850	768254
Chart	(§3.4)	852	997920
LKB		682	17760
All		682	7742

Figure 12: Sizes of the data sets.

straint solvers. To put the results into context, we also compare the runtimes with a solver for Minimal Recursion Semantics (MRS) (Copestake et al., 2004), a different formalism for scope underspecification.

Resources. As our test set we use constraints extracted from the Rondane treebank, which is distributed as part of the English Resource Grammar (Copestake and Flickinger, 2000). The treebank contains syntactic annotations for sentences from the tourism domain such as (4) above, together with corresponding semantic representations.

The semantics is represented using MRS descriptions, which we convert into normal dominance constraints using the translation specified by Niehren and Thater (2003). The translation is restricted to MRS constraints having certain structural properties (called *nets*). The treebank contains 961 MRS constrains, 879 of which are nets.

For the runtime evaluation, we restricted the test set to the 852 nets with less than one million solved forms. The distribution of these constraints over the different constraint sizes (i.e. number of fragments) is shown in Fig. 15. We solved them using implementations of the presented dominance constraint solvers, as well as with the MRS solver in the LKB system (Copestake and Flickinger, 2000).

Runtimes. As Fig. 12 shows, the chart solver is the only solver that could solve all constraints in the test set; all other solvers ran into memory limitations on some inputs.[2] The increased complexity of constraints that each solver can handle (given as the maximum number of solved forms of a solvable constraint) is a first indication that the repeated analysis and improvement of dominance constraint solvers described earlier was successful.

Fig. 13 displays the result of the runtime comparison, taking into account only those 682 constraints that all solvers could solve. For each constraint size (counted in number of fragments), the graph shows the mean quotient of the time to enumerate all solved forms by the number of solved forms, averaged over all constraints of this size. Note that the vertical axis is logarithmic, and that the runtimes of the LKB and the chart solver for constraints up to size 6 are too small for accurate measurement.

The figure shows that each new generation of dominance constraint solvers improves the performance by an order of magnitude. Another difference is in the slopes of the graphs. While the saturation solver takes increasingly more time per solved form as the constraint grows, the set constraint and graph solvers remain mostly constant for larger constraints, and the line for the chart solver even goes down. This demonstrates an improved management of the combinatorial explosion. It is also interesting that the line of the set-constraint solver is almost parallel to that of the graph solver, which means that the solver really does exploit a polynomial fragment on real-world data.

The LKB solver performs very well for smaller constraints (which make up about half of the data set): Except for the chart algorithm introduced in this paper, it outperforms all other solvers. For larger constraints, however, the LKB solver gets very slow. What isn't visible in this graph is that the LKB solver also exhibits a dramatically higher variation in runtimes for constraints of the same size, compared to the dominance solvers. We believe this is because the LKB solver has been optimised by hand to deal with certain classes of in-

[2] On a 1.2 GHz PC with 2 GB memory.

puts, but at its core is still an uncontrolled exponential algorithm.

We should note that the chart-based solver is implemented in C++, while the other dominance solvers are implemented in Oz, and the MRS solver is implemented in Common Lisp. This accounts for some constant factor in the runtime, but shouldn't affect the differences in slope and variability.

Effect of the chart. Because the chart solver is especially efficient if the chart remains small, we have compared how the number of solved forms and the chart size (i.e. number of splits) grow with the constraint size (Fig. 14). The graph shows that the chart size grows much more slowly than the number of solved forms, which supports our intuition that the runtime of the chart solver is asymptotically less than that of the graph solver by a significant margin. The chart for the most ambiguous sentence in the treebank (sentence (4) above) contains 74.960 splits. It can be computed in less than ten seconds. By comparison, enumerating all solved forms of the constraint would take about a year on a modern PC. Even determining the number of solved forms of this constraint is only possible based on the chart.

5 Conclusion

In this paper we described the evolution of solvers for dominance constraints, a logical formalism used for the underspecified processing of scope ambiguities. We also presented a new solver, which caches the intermediate results of a graph solver in a chart. An empirical evaluation shows that each solver is significantly faster than the previous one, and that the new chart-based solver is the fastest underspecification solver available today. It is available online at `http://utool.sourceforge.net`.

Each new solver was based on an analysis of the main sources of inefficiency in the previous solver, as well as an increasingly good understanding of the input data. The main breakthrough was the realisation that normal dominance constraints have polynomial satisfiability and can be solved using graph algorithms. We believe that this strategy of starting with a clean, powerful formalism and then successively searching for a fragment that contains all practically relevant inputs and excludes the pathologically hard cases is applicable to other problems in computational linguistics as well.

However, it is clear that the concept of "all practically relevant inputs" is a moving target. In this paper, we have equated it with "all inputs that can be generated by a specific large-scale grammar", but new grammars or different linguistic theories may generate underspecified descriptions that no longer fall into the efficient fragments. In our case, it is hard to imagine what dominance constraint used in scope underspecification wouldn't be normal, and we have strong intuitions that all useful constraints must be nets, but it is definitely an interesting question how our algorithms could be adapted to, say, the alternative scope theory advocated by Joshi et al. (2003).

An immediate line of future research is to explore uses of the chart data structure that go beyond pure caching. The general aim of underspecification is not to simply enumerate all readings of a sentence, but to use the underspecified description as a platform on which readings that are theoretically possible, but infelicitous in the actual context, can be eliminated. The chart may prove to be an interesting platform for such operations, which combines advantages of the underspecified description (size) and the readings themselves (explicitness).

Acknowledgements. The work has been funded by the DFG in the Collaborative Research Centre 378 *Ressource-Adaptive Cognitive Processes*, project MI 2 (CHORUS).

We would like to thank Joachim Niehren and Denys Duchier for the extremely fruitful collaboration on dominance constraint solving, Ann Copestake and Dan Flickinger for helpful discussions about the ERG and the LKB solver, and our reviewers for their comments. The primary implementors of the various earlier constraint solvers were Katrin Erk and Sebastian Padó (§3.1), Denys Duchier (§3.2), and Sebastian Miele (§3.3).

References

Ernst Althaus, Denys Duchier, Alexander Koller, Kurt Mehlhorn, Joachim Niehren, and Sven Thiel. 2003. An efficient graph algorithm for dominance constraints. *Journal of Algorithms*, 48:194–219.

Krzysztof R. Apt. 2003. *Principles of Constraint Programming*. Cambridge University Press.

Manuel Bodirsky, Denys Duchier, Joachim Niehren, and Sebastian Miele. 2004. An efficient algorithm for weakly normal dominance constraints. In *ACM-SIAM Symposium on Discrete Algorithms*. The ACM Press.

Ann Copestake and Dan Flickinger. 2000. An open-source grammar development environment and broad-coverage english grammar using HPSG. In *Conference on Language Resources and Evaluation*. The LKB system is available at `http://www.delph-in.net/lkb/`.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan Sag. 2004. Minimal recursion semantics: An introduction. *Journal of Language and Computation*. To appear.

Denys Duchier and Joachim Niehren. 2000. Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic*, number 1861 in Lecture Notes in Computer Science, pages 326–341. Springer-Verlag, Berlin.

Markus Egg, Alexander Koller, and Joachim Niehren. 2001. The Constraint Language for Lambda Structures. *Logic, Language, and Information*, 10:457–485.

Aravind Joshi, Laura Kallmeyer, and Maribel Romero. 2003. Flexible composition in LTAG, quantifier scope and inverse linking. In Harry Bunt, Ielka van der Sluis, and Roser Morante, editors, *Proceedings of the Fifth International Workshop on Computational Semantics*, pages 179–194, Tilburg.

Alexander Koller, Joachim Niehren, and Ralf Treinen. 1998. Dominance constraints: Algorithms and complexity. In *Proceedings of LACL*, pages 106–125. Appeared in 2001 as volume 2014 of LNAI, Springer Verlag.

Tobias Müller and Martin Müller. 1997. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *13. Workshop Logische Programmierung*, pages 104–115, Technische Universität München.

Joachim Niehren and Stefan Thater. 2003. Bridging the gap between underspecification formalisms: Minimal recursion semantics as dominance constraints. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*.

Oz Development Team. 2004. The Mozart Programming System. Web pages. `http://www.mozart-oz.org`.

Sven Thiel. 2004. *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. Ph.D. thesis, Department of Computer Science, Saarland University.

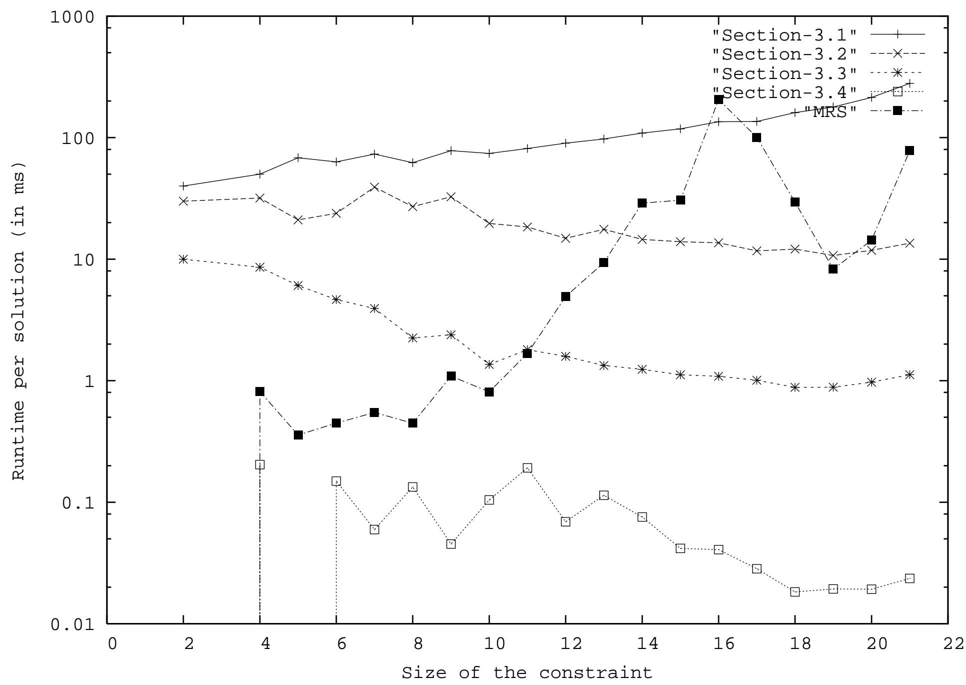

Figure 13: Average runtimes per solved form, for each constraint size (number of fragments).

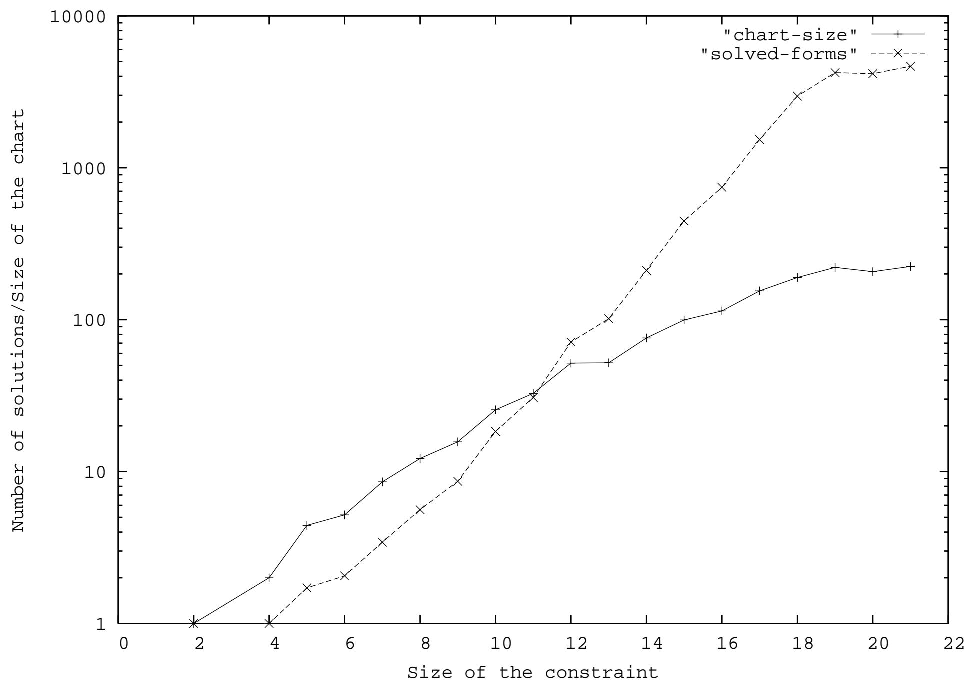

Figure 14: Average size of the chart compared to the average number of solved forms, for each constraint size. Notice that the measurements are based upon the same set of constraints as in Fig. 13, which contains very few constraints of size 20 or more.

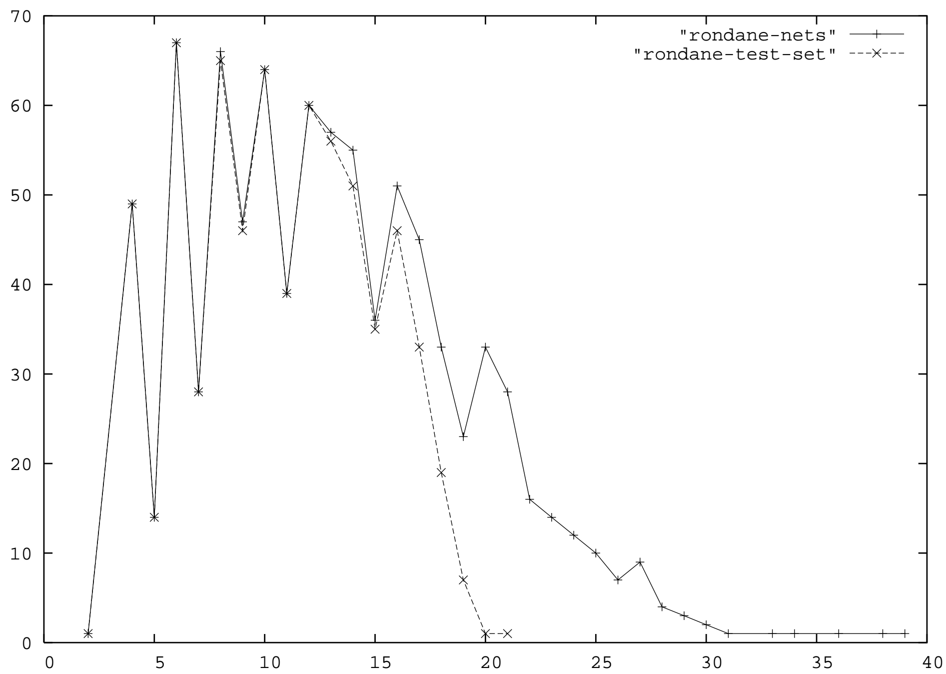

Figure 15: Distribution of the constraints in Rondane over the different constraint sizes. The solid line indicates the 852 nets with less than one million solved forms; the dashed line indicates the 682 constraints that all solvers could solve.