April 25, 2007

FG 2007:
The 12th conference on Formal Grammar
Dublin, Ireland
August 4-5, 2007

Organizing Committee: Laura Kallmeyer
Paola Monachesi Gerald Penn
Giorgio Satta

CENTER FOR THE STUDY
OF LANGUAGE
AND INFORMATION

April 25, 2007

1

Solving Unrestricted Dominance Graphs

ALEXANDER KOLLER AND STEFAN THATER

Abstract

We present the first ever algorithm for solving unrestricted dominance graphs. The algorithm extends existing polynomial solvers for restricted classes of dominance graphs, which are not sufficient to model newer theories of scope ambiguity. Using the new solver, these theories now have access to an efficient solver for the first time. The solver runs in cubic time; for those graph classes that could be solved in the past, the runtime is reduced to the same quadratic time that the most efficient previous solvers achieved.

1.1 Introduction

Scope underspecification is the standard technique to deal with scope ambiguities in computational linguistics. The basic idea behind it is to derive for a sentence that exhibits a scope ambiguity a compact underspecified representation (USR) that describes the set of readings, rather than enumerating the individual readings directly. Dominance graphs (Althaus et al., 2003, Egg et al., 2001) offer a particular attractive framework for modelling scope underspecification: they have a clean and simple formal definition, can encode USRs in other formalisms, and efficient solvers are available (Bodirsky et al., 2004).

However, the existing efficient solvers are restricted to solving only *weakly normal* dominance graphs. This restriction is consistent with the mainstream theories of scope used in underspecification which go back to (Cooper, 1983, Hobbs and Shieber, 1987). Put simply, these approaches assume that all permutations of quantifiers that lead to well-formed object language formulas are actually readings. For instance, (1) below is assumed to have five readings.

(1) Two policemen spy on someone from every city

However, a number of researchers (e.g., Larson, 1985, Sauerland, 2004,

FG-2007.
Organizing Committee:, Laura Kallmeyer, Paola Monachesi, Gerald Penn, Giorgio Satta.

Joshi et al., 2004) have argued that sentences such as (1) have *four* rather than five readings because the subject NP quantifier cannot be placed between the two quantifiers from the object. Kallmeyer and Romero (to appear) provide an underspecification analysis based on this assumption. Their analysis can be straightforwardly modelled with dominance graphs, but the resulting graphs are not weakly normal, so existing algorithms cannot be applied. On the other hand, naive algorithms for enumerating all scopings from such an USR are too slow for practical use.

In this paper, we present an algorithm for solving *unrestricted* dominance graphs that permit an analysis of (1) as proposed by Kallmeyer and Romero. The algorithm generalises an earlier algorithm for weakly normal graphs (Bodirsky et al., 2004). The worst-case runtime for deciding solvability is cubic in the size of the graph. For the special case of weakly normal graphs, it runs in quadratic time, just like the earlier solver. We have implemented the algorithm and will make it available in an open-source implementation.

In Section 1.2, we will define dominance graphs and adapt the notion of *solved forms* to the non-weakly-normal setting. In Section 1.3, we will present the new solver for unrestricted dominance graphs. In Section 1.4, we will adapt the notion of *hypernormally connected* dominance graphs to the non-normal setting, so as to ensure that the solver's concept of solved forms actually corresponds to the intuition. Finally, Section 1.5 concludes and points to future work.

1.2 Dominance graphs

A *labelled dominance graph* (Althaus et al., 2003) consists of a collection of *tree fragments* which are connected by *dominance edges* (see Fig. 1).

Definition 1 A *dominance graph* is a directed graph $G = (V, E \uplus D)$ with two kinds of edges—*tree edges* E and *dominance edges* D—such that the graph (V, E) defines a collection of node disjoint trees. We call the trees in (V, E) the *fragments* of G. A node v is called a *root* if v does not have incoming tree edges; it is called a *leaf* if it does not have outgoing tree edges.

A *labelled dominance graph* over a ranked signature Σ is a triple $(V, E \uplus D, L)$ such that $(V, E \uplus D)$ is a dominance graph and $L : V \rightsquigarrow \Sigma$ is a partial *labelling function*. If $L(v)$ is defined, then the number of tree edges out of v must be equal to the arity of $L(v)$. L must be defined for all non-leaves, and every fragment must contain at least one labelled node. Unlabelled leaves are also called *holes*.

We will write $R(F)$ for the root of the fragment F, and we will typically just say "graph" instead of "labelled dominance graph."

Solved forms. Dominance graphs can be seen as a compact descriptions of sets of trees trees. These trees can be represented by the (minimal) *solved*

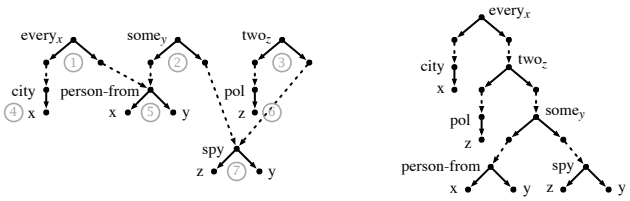


FIGURE 1 A dominance graph (left) and a minimal solved form (right)

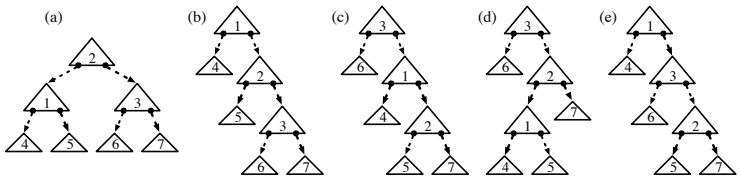


FIGURE 2 The five minimal solved forms of the graph in Fig. 1

forms of the graph.

Definition 2 A dominance graph S is a *solved form* if it is a forest. A solved form S is a *σ-solved form of* a graph G iff σ is a partial function $\sigma : V_G \rightsquigarrow V_S$ such that $V_S = \hat{\sigma}(V_G)$ and for all nodes $u, v \in V_G$:

1. if $(u, v) \in E_G$, then $(\hat{\sigma}(u), \hat{\sigma}(v)) \in E_S$;
2. if L_G is defined on v, then L_S is defined on $\hat{\sigma}(v)$ and $L_S(\hat{\sigma}(v)) = L_G(v)$;
3. if $(u, v) \in D_G$, then there is a (directed) path using tree and dominance edges from $\hat{\sigma}(u)$ to $\hat{\sigma}(v)$ in S;
4. if $u \neq v$ and L_G is defined both on u and v, then $\hat{\sigma}(u) \neq \hat{\sigma}(v)$,

where $\hat{\sigma}$ is the total function from V_G to V_S that maps a node u to $\sigma(u)$ if this is defined and to u otherwise. We call σ a *substitution*, and write $G\sigma$ for the graph obtained from G by replacing each node v by $\hat{\sigma}(v)$. If we don't care about σ, we will also call S simply a *solved form of* G.

A graph is *solvable* iff it has a solved form.

Note that a solved form S of a graph G can also be a solved form of another (less specific) solved form S' of G. The "is a solved form of" relation is thus a partial order on the (finite) set of all solved forms of the same graph G. We call the least specific elements of the order the *minimal solved forms* of G.

This definition of solved forms generalizes earlier definitions in the literature in that it allows different nodes of G to be mapped to the same node of S by σ. Solved forms according to earlier definitions are exactly the $\emptyset$-solved forms here.

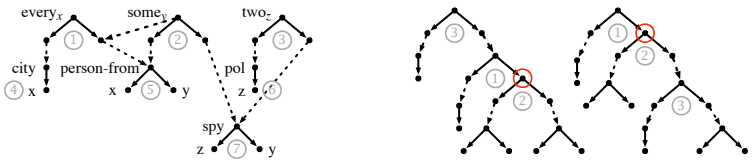


FIGURE 3 Kallmeyer and Romero's analysis of (1) with two of its four solved forms.

	dominance edge types			solvability
	h to r	r to r	r to h	
Thiel (2004)	yes	no	no	$O(n+m)$
Bodirsky et al. (2004)	yes	yes	no	$O(n(n+m))$
present paper	yes	yes	yes	$O(n^2(n+m))$

FIGURE 4 Complexity of dominance graph problems.

Modelling scope underspecification. An example of a labelled dominance graph is shown on the left of Fig. 1. Tree edges are drawn as solid arrows, and dominance edges as dotted ones. The graph can be seen as an USR for the sentence (1): The tree fragments specify the "semantic material" the individual readings are made up of, and the dominance edges restrict the way the tree fragments can be arranged. In total, the graph has five minimal σ-solved forms (all with $\sigma = \emptyset$), which correspond to the five readings of (1). The solved forms are shown in Fig. 2, where triangles stand for tree fragments; solved form (e) is also shown on the right of Fig. 1.

Kallmeyer and Romero (to appear) model their scope theory, which assumes only four readings for (1), by adding a dominance edge from the root of fragment 2 to the the right-hand hole of fragment 1 to the graph (their (37); see Fig. 3). This edge has the effect that whenever fragment 1 dominates fragment 2 in a solved form, the endpoints of the edge must be mapped to the same node by σ. So this graph has four solved forms: the two shown in Fig. 3, plus Fig. 2 (a) and (d).

Solved forms can still contain dominance edges, but in both examples we can obtain a concrete semantic representation (or *configuration* of the graph) from each solved form by identifying the endpoints of each dominance edge. We will explore this process further in Section 1.4.

Classes of dominance graphs. For efficiency reasons, previous solvers for dominance graphs have only been defined on restricted classes of dominance graphs (see Fig. 4). Althaus et al. (2003) introduced *normal* dominance graphs, in which all dominance edges must go from holes to roots. Solvability of normal dominance graphs is a linear-time problem (Thiel, 2004). Later, Bodirsky et al. (2004) presented a quadratic algorithm for solving dominance

graphs that are *weakly normal* (i.e., all dominance edges go into roots) and *compact* (i.e., all tree fragments have height at most one).

The dominance graph in Fig. 3 is not weakly normal because it contains a dominance edge from a root to a hole. In this paper, we present a cubic algorithm for solving arbitrary dominance graphs.

1.3 Solving dominance graphs

We will now present our solver for unrestricted dominance graphs in two steps. First, we will extend Bodirsky et al.'s (2004) solver to non-compact dominance graphs; this is necessary because even initially compact graphs can become non-compact during a run of our algorithm. Then we will deal with edges from roots to holes. Finally, we prove correctness and analyze the runtime of our algorithm.

To simplify the presentation, we assume that the input graph G has been preprocessed by (a) replacing each dominance edge (u, v) such that either v is neither a root nor a hole, or that v is a hole and u is not a root, by the dominance edge (u, r) where r is the root of v's fragment; (b) deleting all dominance edges (u, v) such that u dominates v and they are in the same fragment; and (c) rejecting G as unsolvable if it contains a dominance edge (u, v) such that u and v are in the same fragment, and u doesn't dominate v. The resulting graph has the same solved forms as G, and it only contains dominance edges into roots and dominance edges from roots to holes, and no dominance edges within a fragment. We also assume that G is weakly connected, but this restriction can be lifted by running the algorithm on each connected component and taking the unions of the solved forms for each component.

1.3.1 Solving non-compact dominance graphs

The main procedure of the solver is shown in Fig. 5. It is a function that accepts a connected subgraph G of a dominance graph as its input, and returns the set of its minimal solved forms. It is a recursive function that computes the solved forms by building them top-down. Assume for now that the substitution σ in the algorithm is always $\emptyset$, that $G\sigma = G$, and that $F^\sigma = F$; we will change this below.

This function first computes the *free fragments* of G. A free fragment is a fragment that can be at the root of a solved form of G. The function COMPUTE-FREE-FRAGMENTS returns all fragments whose root has no incoming dominance edges and in which any two sister nodes are in different biconnected components of G. It can be shown (along the lines of Bodirsky et al. (2004)) that this is a necessary and sufficient condition for freeness.

Next, the algorithm iterates over all free fragments F of G and computes the *split* induced by F^σ. (Ignore the call to MAKE-SUPER-FRAGMENT for

```
UNRESTRICTED-SOLVER(G)
 1  if G is already in solved form
 2    then return {G}
 3  S ← ∅
 4  free ← COMPUTE-FREE-FRAGMENTS(G)
 5  for each fragment F in free
 6  do σ ← MAKE-SUPER-FRAGMENT(R(F), G, ∅)
 7     if this call did not fail
 8       then (G_1 ↦ u_1, ..., G_n ↦ u_n) ← SPLIT(Gσ, F^σ)
 9            for i from 1 to n
10            do S_i ← UNRESTRICTED-SOLVER(G_i)
11               if S_i = ∅
12                 then return ∅
13            S ← S ∪ {combine(F^σ, S_1, ..., S_n) | S_1 ∈ S_1, ..., S_n ∈ S_n}
14  return S
```

FIGURE 5 A solver for unrestricted dominance graphs.

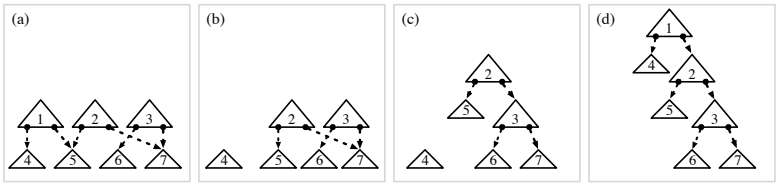


FIGURE 6 Computation of solved form Fig. 2b

now.) A split consists of the weakly connected components (wccs) $G_1, \ldots, G_n$ of the graph $G\sigma - F^\sigma$, and records for each G_i the lowest node u_i in F^σ out of which a dominance edge goes into G_i.

Finally, the algorithm calls itself recursively on each G_i and records the set of solved forms in $\mathcal{S}_i$. If all the G_i were found to be solvable by the recursive calls, the algorithm constructs the set of solved forms for G by combining the solved forms of the subgraphs. For each $(S_1, \ldots, S_n) \in \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$, the function combine builds a new graph $S = (V_S, E_S \uplus D_S)$ such that V_S and E_S are the unions of the node sets and tree edge sets, respectively, of F^σ and all the S_i, and

$$D_S := D_{S_1} \cup \ldots \cup D_{S_n} \cup \{(u_i, r_i) \mid r_i \text{ is root node of } S_i, 1 \leq i \leq n\}$$

In other words, combine merges the free fragment with the solved forms of the subgraphs by adding dominance edges to the root of each sub-solved form S_i from the dominating node u_i recorded in the split.

Fig. 6 shows an example computation. The input graph is shown in (a). The algorithm chooses 1 as free fragment and removes it from the graph, which splits the graph in two wccs in (b). Then it recursively computes a solved form for each wcc; we only show one particular solved form for each wcc in (c). Finally, fragment 1 and the two solved forms are combined into a complete

```
MAKE-SUPER-FRAGMENT(u, G, A)
 1  visited ← {u} ∪ visited
 2  σ ← ∅
 3  for h a hole of the fragment with root u
 4  do P ← DOM-PARENTS(h, G, A)
 5     if |P| > 1
 6       then fail
 7     if |P| = 1
 8       then r ← the unique element of P
 9            if r ∈ visited
10              then fail
11            σ1 ← {h ↦ r} ∪ MAKE-SUPER-FRAGMENT(r, G, {u} ∪ A)
12            σ ← σ ∪ σ1
13  return σ
```

FIGURE 7 Computing super-fragments.

solved form in (d).

1.3.2 Solving unrestricted graphs

The algorithm up to this point will solve non-compact weakly normal dominance graphs. We now extend it to deal with *cross edges* i.e., dominance edges from roots to holes as in Fig. 3.

The key challenge about cross edges is that a dominance edge (u, v) of this kind expresses that either $\hat{\sigma}(v)$ dominates the root of $\hat{\sigma}(u)$'s fragment, or $\hat{\sigma}(u) = \hat{\sigma}(v)$. In other words, it gives us a way to require equality of nodes, which is not possible in weakly normal graphs. For instance, if 1 should be at the root of a σ-solved form of the example graph in Fig. 3, then the right-hand hole of 1 and the root of 2 must be mapped to the *same* node by σ. More generally, whenever a fragment F has a hole h with an incoming dominance edge (r, h), σ must identify h and r, as well as the endpoints of any cross edges into holes below r, etc.

Effectively, once we choose a free fragment F, we must compute a larger, possibly non-compact *super-fragment* which must be at the root of any solved form that has F at the root. This computation can be done by the function MAKE-SUPER-FRAGMENT shown in Fig. 7. The algorithm assumes that the fragment F with root u should be used as the root fragment for a solved form of G. It computes a substitution σ that maps holes to roots, in such a way that the fragment with root u in $G\sigma$ is the super-fragment of F in $G\sigma$; it fails if it is not such possible to construct such a super-fragment (or it is not tree-shaped). We denote the super-fragment of F under the substitution σ by F^{σ}.

The MAKE-SUPER-FRAGMENT algorithm proceeds as follows. It maintains a global variable visited that contains all fragment roots that have been visited during the computation, and maintains the invariant that it is never called on an u that was already visited. In addition, it accepts as third argu-

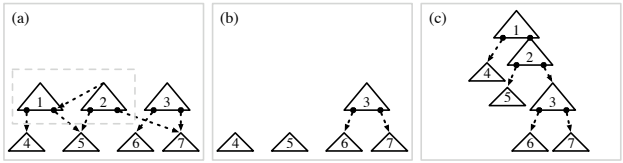


FIGURE 8 Computation of the right solved form of Fig. 3

ment A the set of all fragment roots that dominate u in the super-fragment (u's *ancestors*). The algorithm iterates over all holes of the fragment with root u. For each hole h, it computes the set P of all nodes from which there is a (cross) dominance edge into h, such that P only contains nodes in G, and P contains no ancestors of u. If there is more than one such parent, then the algorithm fails, as this would require us to identify the hole with two different roots. However, it is acceptable to have incoming dominance edges from ancestors, as these are implicitly realized by the superfragment anyway. If the hole has a single incoming dominance edge (from r), the algorithm checks whether r (which must be a root) has been visited before. If yes, it fails: This means that r has already been placed in the super-fragment in a position that is disjoint from u, so the dominance edge can't be satisfied. Otherwise the algorithm calls itself recursively on r (adding u to r's ancestors), and adds the substitution computed by the recursive call, together with a substitution of r for h, to the current substitution.

Let's consider the example graph in Fig. 3 again. If we call MAKE-SUPER-FRAGMENT on the fragment 1 in this graph, it will compute the partial function $\sigma = \{h_{12} \mapsto r_2\}$, where h_{12} is the second hole of 1 and r_2 the root of 2. This corresponds to constructing the super-fragment at the root of the second solved form in Fig. 3. On the other hand, if we call it for the fragment 2, it will return the empty substitution, $\emptyset$, leading to the solved form in Fig. 2a. This is because by the time the algorithm reaches the cross edge, its source has been stored as an ancestor, and so it is trivially satisfied at this point.

Now we can go back to the solver in Fig. 5 in more detail, without the assumption that σ must be $\emptyset$. We have already seen that the solver iterates over all fragments F that are free in G. But now G is allowed to contain cross edges, so we call MAKE-SUPER-FRAGMENT to compute the substitution σ that induces the super-fragment F^σ of F. The algorithm calls itself recursively on the connected components of $G\sigma - F^\sigma$, and combines their solved forms with F^σ in the way described above.

Fig. 8 shows an example computation. The input graph is shown in (a). The algorithm chooses 1 as free fragment. It then computes the super-fragment for 1 (fragments 1 and 2), and removes it from the graph. This splits the graph into three wccs in (b). All three wccs are in solved form, and the recursive call to

UNRESTRICTED-SOLVER returns immediately. Finally, the super-fragment and the wccs in (b) are combined into a complete solved form in (c).

1.3.3 Correctness of the algorithm

We claim that the algorithm computes exactly the minimal solved forms of a dominance graph. The key lemma in this proof is as follows.

Lemma 1 *Let F be a fragment in a solvable graph G, and let $\sigma =$ MAKE-SUPER-FRAGMENT$(R(F), G, \emptyset)$ not be failure. Then the following statements are equivalent:*

1. *$R(F)$ has no incoming dominance edges in G, and any two nodes u and v that are sisters in F are in different biconnected components of G.*
2. *$G\sigma$ has a solved form at whose root is F^σ.*

Proof. $\underline{1 \Rightarrow 2.}$ Assume that (1) is true. Then it can be shown by induction over the number of cross edges traversed by MAKE-SUPER-FRAGMENT that F^σ has the same properties in $G\sigma$. By adapting Corollary 8.1 of Bodirsky et al. (2004), (2) follows.

$\underline{2 \Rightarrow 1.}$ Let $G\sigma$ have a solved form with root F^σ. Then by adapting Lemma 7.1 of Bodirsky et al. (2004), we can prove that the root of F^σ has no incoming dominance edges in $G\sigma$, and all sisters are in different BCCs. But this means that these conditions are also satisfied for F in G. □

In other words, a fragment of a subgraph G' of G is processed in lines 8–13 of the algorithm if and only if its super-fragment can be at the root of a solved form of G'. This insight can be completed to a proof of the following correctness statement, which we can't show here for lack of space.

Proposition 2 *For any dominance graph G,* UNRESTRICTED-SOLVER(G) *returns the set of minimal solved forms of G.*

This means that UNRESTRICTED-SOLVER is a solver for unrestricted dominance graphs.

1.3.4 Runtime analysis

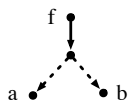
Now let's analyze the runtime of the new solver. Every recursive call of the algorithm spends time $O(m + n)$ on computing the set of free fragments by computing the biconnected components of G. Next, the algorithm iterates over $O(n)$ free fragments. For each of these, it computes the induced super-fragment in time $O(n)$ (MAKE-SUPER-FRAGMENT may have to visit every node in the graph once), and it computes the weakly connected components of $G\sigma - F^\sigma$ in time $O(m+n)$. Finally, it calls itself recursively and combines the sub-solved forms.

Our solver shares the property of the Bodirsky algorithm that if any choice of a free fragment for which MAKE-SUPER-FRAGMENT doesn't fail leads to

the discovery of a solved form, then every choice will. This means that like them, we only need to pursue a single choice in each iteration to decide solvability, i.e. we only need to make $O(n)$ recursive calls throughout the entire run of the algorithm. In total, this means that our algorithm decides solvability in time $O(n^2(m+n))$. This is slower than the $O(n(m+n))$ that the Bodirsky solver takes; the difference is due to the fact that a fragment that passes the freeness test in our line 4 may still make MAKE-SUPER-FRAGMENT fail, and detecting this failure may take linear time. However, their solver is restricted to weakly normal graphs. For such graphs, MAKE-SUPER-FRAGMENT will never fail, and so our algorithm will automatically decide solvability in time $O(n(m+n))$ as well.

In order to enumerate all minimal solved forms, our algorithm can be implemented as a chart solver (Koller and Thater, 2005) to make sure that no recursive call is made twice. In this version of the algorithm, the iteration over the free fragments is actually carried out, and the runtime for each iteration step is dominated by the linear-time computation of the connected components. This means that our solver can be used to compute a chart in asymptotically exactly the same runtime as the Koller & Thater solver.

1.4 Downward connected graphs



Our solver computes the minimal solved forms of a dominance graph. However, in the application to underspecification we are not really interested in the solved forms of a graph; we want its *configurations*, i.e. all arrangements of the fragments into a tree that contains no more dominance edges and that obeys the dominance requirements. Not all graphs in solved form have configurations: The solved form to the right contains a hole with two outgoing dominance edges, so we would have to "plug" a hole with two different roots to get a configuration.

However, if a solved form is *simple* – i.e., all their dominance edges go from holes to roots, and every hole has exactly one outgoing dominance edge –, it is straightforward to construct a configuration from the solved form by identifying the endpoints of each dominance edge. Conversely, every configuration can be constructed from some solved form. So if we have a subclass of dominance graphs for which we can prove beforehand that all minimal solved forms are simple, we can use the above solver to compute the configurations.

For normal dominance graphs, it has been shown that all solved forms of dominance graphs that are *leaf-labelled* and *hypernormally connected* are simple (Koller et al., 2003). These graph classes have somewhat technical definitions, but it has been shown (Flickinger et al., 2005) that they are extremely natural and seem to encompass all USRs that are used in practice.

Definition 3 A *hypernormal path* (Althaus et al., 2003) in a normal graph G is a path in the undirected version of G that does not use two dominance edges that are incident to the same node. G is *hypernormally connected* iff all nodes in G are pairwise connected by simple hypernormal paths. G is *leaf-labelled* if all holes have outgoing dominance edges.

This result is not necessarily true for graphs that are not normal. However, we will now show that the analogous notion of *downward connected* dominance graphs is sufficient to guarantee the simplicity of all solved forms in the general case.

Definition 4 Let G be a dominance graph. The *normal backbone* of G is the result of removing all dominance edges that do not go from holes to roots in G. A dominance graph is called *downward connected* iff its normal backbone is hypernormally connected.

Proposition 3 *All solved forms of a leaf-labelled, downward connected dominance graph G are simple.*

Proof. Consider the normal backbone G' of G. G' is normal, hypernormally connected and leaf-labelled, hence all minimal solved forms $S'_1, \ldots, S'_k$ of G' are simple. G' is a subgraph of G, so every solved form S of G must be a σ-solved form of some S'_i ($1 \leq i \leq k$). It is easy to see that a σ-solved form of a simple solved form must be simple. □

Corollary 4 *The configurability problem of downward connected dominance graphs is in $O(n^2(m+n))$.*

1.5 Conclusion

In this paper, we have presented the first polynomial solver for unrestricted dominance graphs. The algorithm decides the solvability problem of dominance graphs in cubic time and solvability of weakly normal graphs in quadratic time. So far, no polynomial solver for unrestricted dominance graphs existed, and the practically most efficient solvers for weakly normal graphs ran in quadratic time as well. The main technical problem that we had to overcome was the ability of cross edges to express node equalities.

The existence of the new solver has the immediate effect that recent theories of scope ambiguity, which predict four rather than five readings for the three-quantifier sentence (1), now have access to an underspecification formalism with an efficient solver. This will facilitate the experimentation with such scope theories and approaches to semantics construction based on them.

We have implemented our solver and will make it available as an open-source project. In the future, the most obvious extension of our work would be to investigate whether our algorithm can be improved to worst-case quadratic

runtime. This would involve computing all free fragments that can be extended to super-fragments in linear time.

References

Althaus, E., D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. 2003. An efficient graph algorithm for dominance constraints. *J. Algorithms* 48:194–219.

Bodirsky, M., D. Duchier, J. Niehren, and S. Miele. 2004. An efficient algorithm for weakly normal dominance constraints. In *ACM-SIAM Symposium on Discrete Algorithms*. The ACM Press.

Cooper, R. 1983. *Quantification and Syntactic Theory*. Dordrecht: Reidel.

Egg, M., A. Koller, and J. Niehren. 2001. The Constraint Language for Lambda Structures. *Logic, Language, and Information* 10:457–485.

Flickinger, D., A. Koller, and S. Thater. 2005. A new well-formedness criterion for semantics debugging. In *Proceedings of the 12th HPSG Conference*. Lisbon.

Hobbs, J. and S. Shieber. 1987. An algorithm for generating quantifier scopings. *Computational Linguistics* 13(1-2):47–63.

Joshi, A. K., L. Kallmeyer, and M. Romero. 2004. Flexible Composition in LTAG: Quantifier Scope and Inverse Linking. In R. Muskens and H. Bunt, eds., *Computing Meaning, Volume 3*. Kluwer. To appear.

Kallmeyer, L. and M. Romero. to appear. Scope and situation binding in ltag using semantic unification. *Research on Language and Computation* Available at `http://www.sfb441.uni-tuebingen.de/~lk/papers/KallmRomROLC07.pdf`.

Koller, A., J. Niehren, and S. Thater. 2003. Bridging the gap between underspecification formalisms: Hole semantics as dominance constraints. In *Proceedings of the 10th EACL*. Budapest.

Koller, A. and S. Thater. 2005. The evolution of dominance constraint solvers. In *Proceedings of the ACL-05 Workshop on Software*.

Larson, R. K. 1985. Quantifying into NP. Unpublished manuscript, available at `http://semlab5.sbs.sunysb.edu/~rlarson/qnp.pdf`.

Sauerland, U. 2004. DP is not a scope island. *Linguistic Inquiry* 36:303–314.

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