AN EFFICIENT GRAPH ALGORITHM FOR DOMINANCE CONSTRAINTS*

ERNST ALTHAUS†, DENYS DUCHIER‡, ALEXANDER KOLLER§, KURT MEHLHORN†, JOACHIM NIEHREN‡, AND SVEN THIEL†

Abstract. Dominance constraints are logical descriptions of trees that are widely used in computational linguistics. Their general satisfiability problem is known to be NP-complete. Here we identify *normal* dominance constraints and present an efficient graph algorithm for testing their satisfiablity in deterministic polynomial time. Previously, no polynomial time algorithm was known.

1. Introduction. The dominance relation of a tree is the ancestor relation between its nodes. Dominance constraints are logical descriptions of trees talking about the dominance relation.

Dominance based tree descriptions were first used in automata theory in the sixties [TW67], rediscovered in computational linguistics in the early eighties [MHF83], and investigated from a logical point of view in the early nineties [BRVS95]. Since then, they have found numerous applications in computational linguistics: they have been used for grammar formalisms [VS92, RVSW95, DT99, Per00], in natural language semantics [Mus95, ENRX98], and for discourse analysis [GW98].

The two most important computational tasks for dominance constraints are *satisfiability testing* – does the constraint describe a tree? – and *enumerating solutions*, i.e. the described trees. But as shown recently [KNT01], testing satisfiability is an NP-complete problem. Earlier attempts at processing dominance constraints [Cor94, VSWR95, DN00] all suffer from this fact. This has shed doubt on their practical usefulness.

In this article, we identify *normal* dominance constraints, a natural fragment of dominance constraints whose restrictions should be unproblematic for many applications. We present an efficient graph algorithm that decides satisfiability of normal dominance constraints in deterministic polynomial time. Previously, no polynomial time algorithm was known.

We derive the graph algorithm for testing satisfiablity in four steps. First, we introduce dominance graphs and define their configuration problem (that was inverstigated in [ADK$^+$01]). Second, we show that the configurability of dominance graphs is linear time equivalent to the satisfiability of normal dominance constraints (as noticed in [KMN00]). Third, we characterize the configurability of dominance graphs by the absence of particular cycles that we test in the final step by reduction to a matching problem.

We also discuss how to use the efficient satisfiablity test to enumerate solutions. We apply a choice rule exhaustively while checking for satisfiability after each step. Both procedures have been implemented in C++ by using the LEDA library [MN99] and applied to scope ambiguities in natural language semantics in the framework of [ENRX98, EKN01a, EKN01b].

*This article joins the results from two conference publications [ADK$^+$01] and [KMN00]. The authors are partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT)

†Max-Planck-Institute for Computer Science, Saarbrücken, Germany

‡Programming Systems Lab, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany

§Department of Computational Linguistics, Universität des Saarlandes, Saarbrücken, Germany

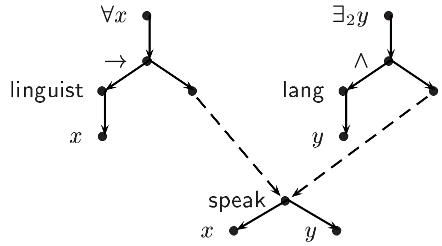


FIG. 2.1. *A simple dominance constraint.*

To complement our results, we finally investigate a close variant of the configuration problem of dominance graphs where closed leaves are permitted in addition. This variant is more general but also relevant for applications in computational linguistics [Bos96, CFS97]. We show that configurability is already NP-complete for the more general dominance graphs. Nevertheless, the presented algorithms can still help to solve this alternative problem more efficiently.

2. Motivation. As one of the many applications of dominance constraints in computational linguistics, we will give a brief introduction to *scope underspecification* [EKN01a, AC92, Rey93, Bos96].

This application is concerned with coping with ambiguous sentences such as the following:

(2.1) Every linguist speaks two languages.

Sentence (2.1) is ambiguous because it has two different possible meanings, indicated by the continuations

(2.2) ... namely, English and Chinese.

(2.3) ... but not necessarily the same ones.

In the first reading, each linguist must speak the same two languages. In the second, no two linguists necessarily speak a common language, but each speaks at least two.

Ambiguity is a real problem to language processing because the number of readings of a sentence grows quickly with the number of "quantifiers" such as "every linguist" and "two languages", and interacts with other sources of ambiguity besides. The sentence (2.4) has already 56 readings, and larger examples are easy to construct.[5]

(2.4) John says that some representative of every department in a company saw a sample of each product.

The key observation to scope underspecification is that the differences between the readings are very systematic; all contain the same "semantic material" (e.g. representations of the constituents "every linguist", "two languages", and "speak"), which is only combined in different ways. The constraints on these combinations can be specified using dominance constraints.

An example is Fig. 2.1. This *constraint graph* is a description of the two readings of (2.1); it can be seen as a graphical representation of a dominance constraint. Similarly, the 56 readings of (2.4) can be represented by the graph in Fig. 2.2. In the paper, we will use constraint graphs to link the (logic) work on dominance constraints to graph algorithms.

[5]The following sentence from [Hob83], which is interesting both in form and in content, has around 200 readings: "Many people feel that most sentences exhibit too few quantifier scope ambiguities for much effort to be devoted to this problem, but a casual inspection of several sentences from any text should convince almost everyone otherwise."

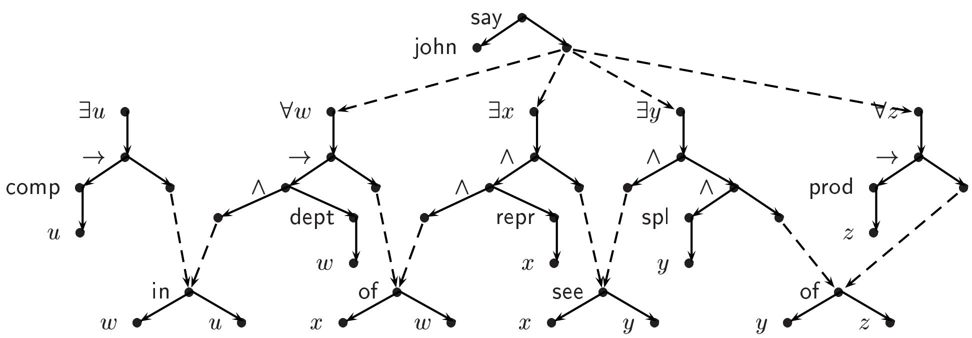


FIG. 2.2. *A dominance constraint describing the meaning of (2.4).*

Pictures as in Fig. 2.2 are being drawn in most modern approaches to scope underspecification. However, they are not always interpreted as dominance constraints [Rey93, Bos96]. The subtle difference in meaning has the surprising effect of making these other approaches NP-complete even when the graphs fall into the class where dominance constraints have polynomial satisfiability. We will show this in Section 10.

3. Satisfiability of Dominance Constraints. In this section, we define the the language of dominance constraints and recall known results on satisifability. The variant of dominance constraints we employ describes constructor trees – ground terms over a signature of function symbols – rather than feature trees as considered in [Smo92, BRVS95, ST94, MNP00].

3.1. Trees and Constructor Trees. We assume a signature Σ with function symbols ranged over by $f, g, \ldots$, each of which is equipped with an arity $\mathsf{ar}(f) \geq 0$. Constants are function symbols of arity 0 ranged over by a, b. We assume that Σ contains at least one constant and one symbol of arity at least 2.

A constructor tree can be defined either as a term or equivalently on the basis of directed graphs. The ground term $f(g(a,a))$, for instance, corresponds to the directed graph in Figure 3.1. Throughout this article, we will employ the graph based definition.

An *(unlabeled) tree* is a forest with exactely one root. A *forest* is a finite directed graph (V, E) where V is a finite set of *nodes* ranged over by u, v, w, and $E \subseteq V \times V$ a set of *edges* such that the indegree of each node is at most 1 and there is no cycle. Each forest has at least one *root*, i.e. a node with indegree 0. We call the nodes with outdegree 0 the *leaves* of the forest.

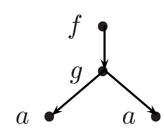


FIG. 3.1. $f(g(a,a))$

A *(finite) constructor tree* τ is a triple (V, E, L) consisting of a tree (V, E) and two labeling functions L, a *node labeling* $L : V \to \Sigma$ and an *edge labeling* $L : E \to \mathbb{N}$. The edge labels in a constructor tree determine the order of the children of a node: for each node $u \in V$ and each natural number $1 \leq k \leq \mathsf{ar}(L(u))$, there is exactly one edge $(u, v) \in E$ with $L((u, v)) = k$.

We draw constructor trees as in Figure 3.1, by annotating nodes with their labels and ordering the edges such that their labels increase from left to right.

3.2. Constraint Language. The language of dominance constraints is a logical language that is interpreted over the class of tree structures modelling the relations of a constructor tree.

Let $\tau = (V, E, L)$ be a constructor tree with nodes $u, v, v_1, \ldots v_n \in V$. The dominance relation $u \lhd^* v$ holds in τ iff there is a path from u to v; the labeling relation $u{:}f(v_1, \ldots, v_n)$ holds in τ iff u is labeled by the n-ary symbol f and has the children $v_1, \ldots, v_n$ in this order; that is, $L(u) = f$, $\mathsf{ar}(f) = n$, $\{(u, v_1), \ldots, (u, v_n)\} \subseteq E$, and $L((u, v_i)) = i$ for all $1 \leq i \leq n$.

DEFINITION 3.1 (Tree Structure). *The* tree structure *of a constructor tree τ with node set V is a first-order structure with domain V which provides the* dominance relation $\lhd^*$ *of τ and the* labeling relation *of τ for each function symbol $f \in \Sigma$.*

Let Vars be an infinite set of (node) variables ranged over by X, Y, Z. A *dominance constraint* φ is a conjunction of *dominance*, *inequality*, and *labeling* literals of the following form where $\mathsf{ar}(f) = n$:

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

We freely identify a constraint with the set of its literals. Let $\mathsf{Var}(\varphi)$ be the set of variables of φ. A pair of a tree structure τ with node set V and a *variable assignment* $\alpha : \mathsf{Var}(\varphi) \to V$ *satisfies* φ iff it satisfies all its literal in the obvious way. We say that (τ, α) is a *solution* of φ in this case; φ is *satisfiable* if it has a solution.

For instance, the following constraint that happens to be unsatisfiable:

$$X{:}f(X_1, X_2) \wedge X_1 \lhd^* Y \wedge X_2 \lhd^* Y$$

. It requires that node values of X_1 and X_2 are sisters that are both ancestors of the node value of Y. This is clearly impossible in a tree, since trees cannot branch upwards.

3.3. Constraint Graphs. We usually draw a dominance constraint as a *constraint graph*. For instance, the unsatisfiable constraint from above is drawn in Figure 3.2. It illustrates clearly that the constraint requires upward branching and thus cannot be satisfied by any tree.

The nodes of a constraint graph are the variables of the corresponding constraint. Labeling constraints relate to solid edges called *tree edges*. Dominance constraints are drawn as dashed lines called *dominance edges*. As for trees, we annotate labels to nodes of the graph and order tree edges from left to right. Note that we ignore inequalities in constraint graphs.

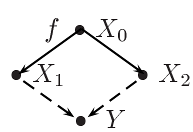


FIG. 3.2. *The unsatisfiable constraint* $X{:}f(X_1, X_2) \wedge X_1 \lhd^* Y \wedge X_2 \lhd^* Y$

We sometimes annotate variable names to the graph nodes. This is not always necessary since all occurences of the same variable are always represented by a single node in a constraint graph. We may thus freely omit variable names. In the motivating example (Figure 2.2), for instance, we have omitted all variable names.

Constraint graphs motivate the following notions to talk about constraints. We call a variable X *labeled* in a constraint φ if there exists a literal $X{:}f(\ldots)$ in φ. A *(solid) fragment* of constraint φ is a maximal set of variables in φ that are pairwise connected by labeling literals. A variable X is called a *root of a fragment* in φ if it does not occur in child position of a labeling literal in φ, i.e. if $Z{:}f(\ldots X \ldots)$ does not belong to φ. A *hole of a fragment* is a variable in φ that is unlabeled in φ. A *leaf of a fragment* is either a hole or a variable labeled by a constant, i.e. a variable X with $X{:}a$ in φ.

3.4. Satisfiability. We are interested in two natural problems concerning dominance constraints that are both motivated by our application: first of all we would like to test satisfiablity, and second, we would like to enumerate all solutions of a satisfiable dominance constraint.

The complexity of the satisfiability problem of dominance constraints was investigated in [KNT01] and shed doubts on their usefulness.

THEOREM 3.2. *Satisfiablitiy of dominance constraints is NP-complete.*

Deciding satisfiability in non-deterministic polynomial time is quite simple: In a first step one guesses whether $X \lhd^* Y$ or $\neg X \lhd^* Y$ for each two variables X, Y in a given constraint. In a second step, one tests the consistency of these relationships.

The NP-hardness proof relies on the fact that solid fragments of a constraint graph may *overlap* in a solution. This means that distinct labeled variables may be assigned to the same node of a tree.

For illustration, consider the constraint $X{:}f(X_1, X_2) \wedge Y{:}f(Y_1, Y_2) \wedge Y \lhd^* X \wedge X \lhd^* Y_1$ whose graph is shown in Figure 3.3. Every solution must map X to the same node as either Y or Y_1. We say that X overlaps with Y or Y_1 in a solution of this constraint.

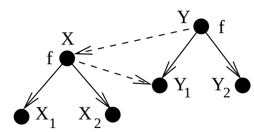


FIG. 3.3. *Overlap*

We call an overlap *proper* that concernce two labeled variables. In the applications in computational linguistics, we typically do not want proper overlap (but may accept overlaps of roots with holes). The subclass of dominance constraints that excludes proper overlap (and fixes some minor inconveniences) is the class of *normal* dominance constraints.

3.5. Normal Dominance Constraints. We next distiguish a fragment of dominance constraints which we will show to have a polynomial time satisfiability problem.

DEFINITION 3.3 (Normal dominance constraint). *A dominance constraint φ is called* normal *iff for all variables $X, Y \in \mathsf{Var}(\varphi)$:*

1. *There is no proper overlap in solutions of φ: $X \neq Y$ in φ if X and Y are distinct variables that are labeled in φ.*
2. *Solid fragments are tree shaped or cyclic: every variable in φ appears at most once as a parent and at most once as a child in a labeling literal of φ.*
3. *Dominance edges go from holes to roots: if $X \lhd^* Y$ in φ then X is unlabeled in φ whereas Y is labeled but does not occur in child position in φ.*
4. *There are no empty fragments: every hole of φ occurs in some child position.*

Conditions 1 and 4 say that only roots and holes have the permission to overlap in a solution of a normal constraint. Distinct holes cannot overlap since they must have distinct parents, which are labeled variables that cannot overlap. For a similar reason, it is impossible that a hole overlaps with a labeled node that has a parent.

Condition 2 requires acylic fragments to be tree shaped. This excludes many constraints, as for instance $X{:}f(Y, Y)$, $X{:}f(Y_1, Y_2) \wedge X{:}f(Z_1, Z_2)$ or $Y{:}f(X) \wedge Z{:}f(X)$. The last two examples are particularly difficult to treat when subsumed by a larger constraint: they entail equations ($Y_1{=}Z_1$, $Y_2{=}Z_2$, respectively $Y{=}Z$) whose global consequences are difficult to predict. W.l.o.g we can always restrict ourselves to normal constraint with acyclic fragments. Other constraints are unsatisfiable anyway.

Condition 3 forbids to express equality through two side dominance: $X \lhd^* Y \wedge Y \lhd^* X$ is not normal since a variable cannot be at the same time a root and a hole. Condition 3 is also vialoted by the dominance edge $X \lhd^* Y_1$ in the constraint from overlap example in Fig. 3.3. It goes from a root to a hole, instead vice versa.

In the following theorem we state the main result of this article, which will follow from the results presented in the succeeding sections.

THEOREM 3.4. *Satisfiablity of normal dominance constraints can be decided in deterministic polynomial time.*

3.6. Solved Forms. We stated above that we would like to have an algorithm that enumerates all solutions of a given normal dominance constraint. Interpreting this proposition literally makes not much sense as the reader already might have noticed. For instance, we can solve the constraint $X{:}a$ by all trees that have a node labeled by a. Indeed, every satisfiable constraint has an infinite number of solutions, so that we probably do not want to enumerate all of them.

What we want to do is to enumerate all solved forms of a normal dominance constraint instead of all solutions. The idea behind a solved form is that it should be similar to a solution but not describe its irrelevant parts. For instance, $X{:}a$ is a perfect solved form since all its solutions can be easily read off from this constraint.

We will now define an appropriate notion of solved forms. In particular, it should hold that a normal dominance constraint has a solution if and only if it has a solved form. Given a constraint φ we define a relation R_φ on the variables of φ that we call the *reachability relation* of φ. This relation is the transitive closure of the following relation:

$$\{(X,Y) \mid X{:}f(\ldots,Y,\ldots) \in \varphi \text{ or } X \triangleleft^* Y \in \varphi\}$$

We say that Y can be reached form X if $(X,Y) \in R_\varphi$. In this case, it clearly holds that φ entails the dominance $X \triangleleft^* Y$.

DEFINITION 3.5 (Solved Form). *A normal dominance constraint φ is in* solved form *if it satisfies the following two properties for all variables X, Y, Z in $\mathsf{Var}(\varphi)$:*

1. *Dominance edges do not branch upwards: if X and Y are distinct then not both $X \triangleleft^* Z$ in φ and $Y \triangleleft^* Z$ in φ.*
2. *The graph of φ is acyclic: $(X,X) \notin R_\varphi$.*

In other words, a normal dominance constraint φ is in solved form if and only if its constraint graph is a forest.

A *solved form of a normal constraint* φ is a normal constraint φ' that is in solved form, contains the same labeling literals as φ, and has a stronger reachability relation, which means $R_\varphi \subseteq R_{\varphi'}$.

LEMMA 3.6. *Every normal dominance constraint in solved form has a solution.*

Proof. We have to construct tree solutions for solved forms φ. The idea is that the constraint graph of a solved form is already a forest. It is sufficient to transform this forest into a tree without dominance edges. This is quite simple given the transformation illustrated in Figure 3.4. (Note that we assumed Σ to contain a function symbol f of arity at least two and a constant a.)

In the first step, we turn the forest into a tree by adding a top most fragment. Let $Y_1, \ldots, Y_m$ be the minimal elements in the reachability order R_φ which exist since R_φ is acyclic. We can then define a new solved form φ_1 with a top most fragment, by adding a new fragment to φ with a single hole with dominance whishes towards all roots $Y_1, \ldots, Y_m$.

In the second step we repeatedly transform dominance edges into tree edges. We stop, once no dominance edge is left. The idea is illustrated in Figure 3.4. Recall that we assumed that our signature contains a constant a and a function symbol f of arity $n \geq 2$. We will also use a function dist on constraints, which is defined for all constraints φ' by $\mathsf{dist}(\varphi') = \varphi' \wedge \bigwedge\{X \neq Y \mid X, Y \in \mathsf{Var}(\varphi') \text{ distinct}\}$. Suppose that

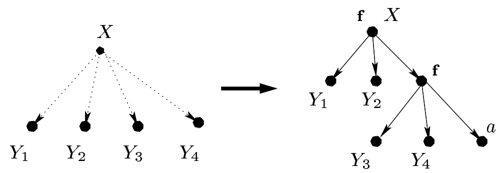


FIG. 3.4. *Transforming dominance edges into tree edges. Here, 4 dominance edges are transformed while using a function symbol of arity 3.*

there still exists a hole X in φ_1 from where dominance edges start. Let $Y_1, \ldots, Y_m$ be the roots of φ such that $X \triangleleft^* Y_i$ belongs to φ_1 for all $1 \leq i \leq m$. We fix fresh variables $Z_3, \ldots, Z_n$ and define:

$$\varphi_2 = \mathsf{dist}(\varphi_1 \wedge X{:}f(Y_1, Z_2, \ldots, Z_n) \wedge \bigwedge\nolimits_{i=2}^{m} Z_n \triangleleft^* Y_i \wedge \bigwedge\nolimits_{j=2}^{n-1} Z_j{:}a)$$

If we remove $X \triangleleft^* Y_1$ from φ_2 then we obtain again a solved form which still entails φ but contains one dominance literal less.

In the third step we can easily satisfy the resulting constraint by the constructor tree that corresponds to itself. ☐

The solutions we built in the proof of Lemma 3.6 are always overlap free. This means that a solved form cannot entail any equality $X{=}Y$ for distinct variables X and Y. The same property holds for normal constraints. This is a consequence of the following lemma which state that every satisfiable normal constraint has a solved form.

LEMMA 3.7. *Every solution of φ also satisfies some solved form φ' of φ.*

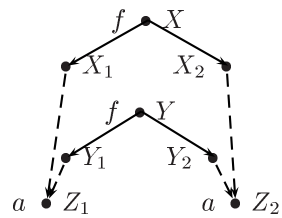
Proof. Given a solution (α, τ) of φ we define a constraint φ' by:

$$\varphi' = \varphi \wedge \bigwedge \{X \triangleleft^* Y \ \mid\ X \text{ hole in } \varphi,\ Y \text{ root in } \varphi,\ \alpha(X) \triangleleft^* \alpha(Y)\}$$

This constraint is clearly normal, satisfies $R_\varphi \subseteq R_{\varphi'}$, and is solved by (τ, α) so that $R_{\varphi'}$ is acylic. We call a dominance literal *redundant* in a constraint if it can be removed while preserving the reachability relation of the constraint. Let φ'' be a constraint that is obtained from φ' by exhaustively removing redundant dominance literals. It remains to show that φ'' is a solved form (it is then a solved form of φ).

It is clear that $R_{\varphi''}$ is acyclic. We show that dominance edges in φ'' do not branch upwards. Suppose that this were wrong, i.e. that $X_1 \triangleleft^* Y$ and $X_2 \triangleleft^* Y$ belong to φ'' where $X_1 \neq X_2$. Dominance edges go from holes to roots, so that X_1 and X_2 are distinct holes and Y is a root. The assumption that (τ, α) solves φ and thus φ'' yields that either $\alpha(X_1) \triangleleft^* \alpha(X_2)$ or vice versa. W.l.o.g, we assume the first case. Since distinct holes cannot overlap, we have $\alpha(X_1) \neq \alpha(X_2)$. Let Y_2 be the root of the fragment of X_2. Since holes cannot overlap with inner nodes of fragments, it follows that $\alpha(X_1) \triangleleft^* \alpha(Y_2)$. By definition of φ', we know that $X_1 \triangleleft^* Y_2$ belongs to φ'. Thus, $(X_1, Y_2) \in R_{\varphi''}$ and by transitivity $(X_1, X_2) \in R_{\varphi''}$. Suppose we obtain φ''' by removing the literal $X_1 \triangleleft^* Y$ from φ''. Then (X_1, X_2) is still in $R_{\varphi'''}$, otherwise $R_{\varphi''}$ contains (Y, X_2) and hence (X_2, X_2), which cannot be. Thus $(X_1, Y) \in R_{\varphi'''}$, which means that $X_1 \triangleleft^* Y$ is redundant in φ'', a contradiction. ☐

The validity of Lemma 3.7 heavilly depends on the absence of proper overlaps (Condition 1 of Def. 3.3). This is illustrated by the following example:



$$X\!:\!f(X_1,X_2)\wedge Y\!:\!f(Y_1,Y_2)\wedge\bigwedge_{i=1}^{2}(X_i\triangleleft^* Z_i\wedge Y_i\triangleleft^* Z_i\wedge Z_i\!:\!a)$$

This constraint satisfies all normality conditions except for the overlap restriction. It also has a solution but no normal form. The reason for this problem is that X and Y overlap properly in all solutions of this constraint.

The combination of Lemmas 3.6 and 3.7 yields the following proposition, which justifies computing with solved forms instead of solutions:

PROPOSITION 3.8. *A normal dominance constraints has a solved form if and only if it is satisfiable.*

4. Configurability of Dominance Graphs. For the satisfiability problem it turns out that we do not have to consider any labels. If we delete all labels from the constraint graph but keep the information whether an edge is a tree or a dominance edge, we get a dominance graph (given two minor assumptions):

DEFINITION 4.1 (Dominance Graph). *A* dominance graph *is a directed graph $G = (V, E \mathbin{\dot\cup} D)$ satisfying the following two conditions:*

1. *The graph $G = (V, E)$ defines a collection T of node disjoint trees of height at least 1.*
2. *Each edge in D goes from a leaf of some tree in the collection to the root of some tree in the collection.*

We will use analogous notions for dominance and constraint graphs: The edges in T are called *tree edges*, and the edges in D are called *dominance edges*. A *leaf* is a node with no outgoing tree edge and a *root* is a node with no incoming tree edge. A dominance edge $d = (v, w)$ is *redundant* if there is a path from v to w in $G \setminus d$. The *reachability relation* R_G of a dominance graph $G = (V, E \mathbin{\dot\cup} D)$ is the set of all pairs (u, v) such that there is a (directed) path from u to v in G, i.e. R_G is the transitive closure of the binary relation induced by the edge set of G.

We need a new notation that replaces the notion of a solution in the logical sense. Now the idea is that we want to assemble the trees in T by plugging roots into leaves.

DEFINITION 4.2. *We say that a dominance graph G is a* configuration *iff it is a forest and the edges in D' form a matching, i.e. every node of G is incident to at most one edge of D'*[6].

We call a dominance graph $G' = (V', E' \mathbin{\dot\cup} D')$ a *configuration of* G iff $V = V'$, $E = E'$, G' is a configuration, and $R_G \subseteq R_{G'}$, i.e. G' realizes all dominance wishes in G. A dominance graph is *configurable* if it has a configuration.

4.1. Solved forms. In the sequel we will prove the equivalence of the configurability problem for dominance graphs and the satisfiability problem for normal dominance constraints (Lemma 4.5). In order to do so, we extend the notion of solved forms to dominance graphs.

DEFINITION 4.3. *A dominance graph G' is* in solved form *iff it is a forest.*

[6]And hence D' defines a partial function from roots to holes which specifies for every matched root where to plug it.

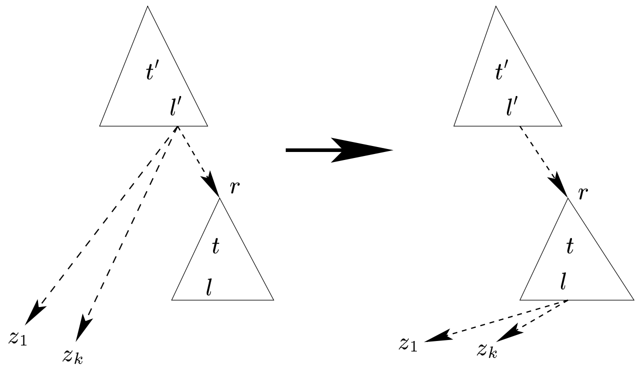


FIG. 4.1. *Application of Rule 1: All dominance wishes of l' except for (l', r) are shifted down to the leaf l.*

By definition, every configuration is a solved form. In the opposite to a configuration, a solved form does not require its dominance edges to form a matching.

We call a dominance graph $G' = (V', E' \dot\cup D')$ a *solved form of G* iff $V = V'$, $E = E'$, G' is a solved form and $R_G \subseteq R_{G'}$. A dominance graph is *solvable* if it has a solved form. The following lemma shows that configurability and solvability are equivalent for a dominance graphs.

LEMMA 4.4. *Every dominance graph in solved form is configurable.*

Proof. For the proof, we define a *problem leaf* to be a leaf with more than one outgoing dominance edge; our aim will be to eliminate problem leaves from solved forms.

The proof is by induction on weights (d, a) of graphs G, where d is the negative minimum depth of a problem leaf of G (or $-\infty$ if there is none), and a is the total number of dominance edges emanating from problem leaves of minimum depth (potentially 0). We consider the lexicographic order on these weights.

Solved forms without problem leaves (i.e. with weight $(-\infty, 0)$) are configurations, so the lemma is trivially true in this case. So let G be a solved form that does have problem leaves. Let G have weight (d, a), and assume that we know that all solved forms of lower weight do have configurations. Then we can apply the following rule to a problem leaf l' of minimum depth:

SIMPLIFICATION RULE 1. *Let $e = (l', r)$ be a dominance edge from the leaf l' of a tree t' to the root r of a tree t. Let l be an arbitrary leaf of t. Change any dominance edge (l', z) with $z \neq r$ into (l, z), see Figure 4.1.*

The result G' is still in solved form, and its weight is strictly lower than that of G; so by the induction hypothesis, G' has a configuration G_c. But G_c also realizes all dominance wishes of G. This is obvious for (l', r) and for all wishes which do not start in l'. For a wish (l', z) with $z \neq r$ we note that this wish is realized because there is a path from l' to l in G' and G_c realizes the wish (l, z). So G has a configuration as well. ∎

4.2. Dominance Graphs of Normal Constraints. We now map normal dominance constraints to dominance graphs while ignoring lablings.

Let φ be a normal dominance constraint. We define a graph $G(\varphi) = (\mathsf{Var}(\varphi), E \dot\cup$

$D)$, where the set of *tree edges* E and *dominance edges* D are defined as follows:

$$\begin{array}{rcl} E & = & \{(X, X_i) \mid X{:}f(X_1, \ldots, X_n) \text{ in } \varphi,\ 1 \leq i \leq n\} \\ D & = & \{(X, Y) \mid X \triangleleft^* Y \text{ in } \varphi\} \end{array}$$

LEMMA 4.5. *For a normal dominance constraint φ the following holds if none of its roots is labeled by a constant and if all its fragments are acyclic.*

1. *The graph $G(\varphi)$ is a dominance graph.*
2. *The relations R_φ and $R_{G(\varphi)}$ are equal.*
3. *φ is in solved form iff $G(\varphi)$ is.*
4. *If φ' is a solved form of φ then $G(\varphi')$ is a solved form of $G(\varphi)$ and vice versa.*

Proof. We will proof the statements one by one:

1. Conditions 1 and 2 of Definition 3.3 ensure that all acyclic fragments of φ are trees. Since we assumed all fragments to be acyclic, it follows that $(\mathsf{Var}(\varphi), E)$ is a collection of node-disjoint trees. The roots of these trees must also be roots of fragments in φ since there are no empty fragments by Condition 4. This shows that the heigth of all trees in $G(\varphi)$ is at least one. And finally, from Condition 3 we can conclude that dominance edges can only go from leaves to roots.
2. We have equality since both relations are defined as the transitive closure of the same set.
3. Only the roots of the normal constraint φ may have more than one incoming edge in $G(\varphi)$ all of which are dominance edges. But every root in a solved form has by definition at most one incoming dominance edge. Since solved forms are acyclic by definition, it follows that their graphs are always forests and thus in solved form. The converse implication is obvious.
4. This an immediate consequence of the previous statements and the definitions of solved form for normal dominance constraints and dominance graphs.

□

We call a dominance graph G *arity consistent* if for all nodes v of G there exists a function symbol $f \in \Sigma$ such that the number of tree edges emanating from v is equal to the arity of f.

LEMMA 4.6. *For every arity consistent dominance graph G there exists a normal dominance constraint φ such that $G(\varphi) = G$.*

This lemma is obvious. The following theorem summarizes our results of the previous two lemmas:

THEOREM 4.7. *The following four problems are linear time equivalent:*

1. *Satisfiability of normal dominance constraints.*
2. *Existence of solved forms for normal dominance constraints.*
3. *Existence of solved forms for dominance graphs.*
4. *Configurability of dominance graphs.*

Proof. We have to check the application conditions of Lemmas 4.5 and 4.6.

1. We can always assume fragments to be acyclic. The existence of cycles can be checked easily in linear time, and normal constraints with cycles are unsatisfiable anyway.
2. We can assume that normal dominance constraints do not contain roots that are labeled by constants. Otherwise, we can replace the fragment $X{:}a$ by $X{:}f(X_1, \ldots, X_n) \wedge \bigwedge_{i=1}^{n} X_i{:}a$ for some fresh variables $X_1, \ldots, X_n$ and sufficiently many inequalities. This operation clearly preserves normality, but also satisfiablity as the inner structure of fragments is anyway irrelevant for the satisfiabity of normal constraints.

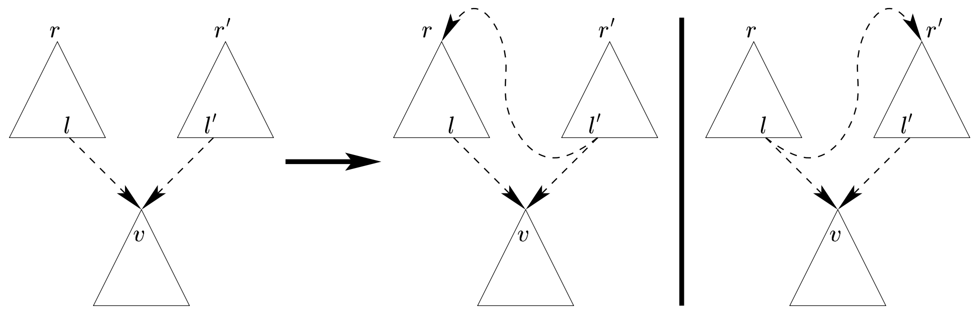


FIG. 5.1. *Two graphs H, H' are generated by applying the choice rule to the graph G on the left hand side.*

3. We can make all dominance graphs arity consistent with a similar transformation as the one shown in Figure 3.4.

∎

5. Enumeration of Solved Forms. In this section, we show how to enumerate the solved forms of a dominance graph G. The algorithm we present may take exponential time to produce even a single configuration because it blindly enumerates all cases. In Section 7, we will present a polynomial time algorithm for determining configurability. By plugging this algorithm into the enumeration algorithm, we can enumerate configurations in polynomial time per configuration (Section 8).

The enumeration algorithm applies the following simplification rules:

SIMPLIFICATION RULE 2 (Redundancy Elimination). *All redundant dominance edges, i.e. edges that are implied by transitivity, can be removed. In particular, parallel edges can be combined into one.*

SIMPLIFICATION RULE 3 (Choice). *Let v be a root with at least two incoming dominance wishes (l, v) and (l', v) and let r and r' be the roots of the trees containing leaves l and l', respectively. Generate two new graphs H and H' by adding either (l', r) or (l, r') to D, see Figure 5.1.*

The enumeration of the solved forms can be carried out by a recursive algorithm:

1. Make the graph reduced, i.e. apply Rule 2.
2. If the graph contains a (directed) cycle, terminate this recursion since the graph has no configuration.
3. If the graph is in solved form, report it and terminate this recursion.
4. Otherwise, apply the choice rule and apply the algorithm to the two newly generated graphs.

Every solved form derived by the algorithm is clearly a solved form of the original graph. On the other hand, the algorithm enumerates *all* of its solved forms. This is because an application of Rule 2 to a dominance graph does not change the set of solved forms. An application of Rule 3 partitions the set: The two new graphs have disjoint sets of solved forms, but the union of these sets is the same as the old set of solved forms. This can be seen as follows. In a solved form G_s of G the nodes l and l' are both ancestors of v and therefore either l' is ancestor of l and hence of r or vice versa. This implies that G_s is either a solved form of H or of H'.

To prove termination, we derive an upper bound for the maximum recursion depth. We reconsider the reachability relation R_G of a graph G. If G is acyclic, the cardinality of R_G is at most $\binom{n}{2} \le n^2$. Thus, whenever the size of the relation becomes greater than $\binom{n}{2}$, the recursion terminates immediately. But if we apply the choice rule to a reduced, acyclic dominance graph, the size of the relation increases strictly, i.e. $|R_G| < \min(|R_H|, |R_{H'}|)$. This is because $R_H \supseteq R_G$, and $(l', r) \in R_H$ but (l', r) cannot be in R_G, otherwise (l', v) would have been redundant. A similar argument holds for H'.

6. Characterization of the Existance of Solved Forms. We give a graph theoretic characterization of solvability; as this is equivalent to configurability by Lemma 4.4, the result carries over to configurability.

The *undirected dominance graph* $G_u = (V, E_u \dot\cup D_u)$ corresponding to the dominance graph $G = (V, E \dot\cup D)$ is the undirected graph obtained by making all edges of G undirected. More precisely, we set $E_u = \{(\{u, v\}, \mathit{tree}) \; ; \; (u, v) \in E\}$ and define $D_u = \{(\{u, v\}, \mathit{dom}) \; ; \; (u, v) \in D\}$. This explicit distinction is important. Consider the dominance graph consisting of the two nodes r and l, the tree edge (r, l) and the dominance edge (l, r). If there were no explicit distinction, then both edges would correspond to the same undirected edge. However, when we talk about undirected edges in the sequel, we will only state the first component because it will be clear whether we refer to a tree or a dominance edge.

Now, we want to define the notion of a *cycle* in an undirected graph, which may differ from the reader's usual notion. A cycle C in an undirected graph is a sequence $[v_0, e_0, v_1, e_1, \ldots, e_{n-1}, v_n]$ of nodes $v_0, \ldots, v_n$ and edges $e_0, \ldots, e_{n-1}$ with $n \geq 2$ such that $v_0 = v_n$ and for $i = 0, \ldots, n-1$ the edge e_i is incident to v_i and v_{i+1}, and $e_i \neq e_{(i+1) \bmod n}$. This means we require a cycle to consist of at least 2 edges, we do not exclude that a cycle uses a node or an edge more than once, however any two consecutive edges must be distinct. We call C *edge-simple* if the edges in the sequence are pairwise different. C is said to be *simple* if all the visited nodes $v_0, \ldots, v_{n-1}$ (and hence also the edges) are pairwise different. Usually, we are not interested in the sequence of nodes and we identify C with the sequence $e_0 \circ e_1 \circ \ldots \circ e_{n-1}$ of its edges.

6.1. Hypernormal Dominance Graphs.
Let us first investigate a simpler subproblem of the solvability problem. A dominance graph $G = (V, E \dot\cup D)$ is *hypernormal* if for every leaf l in (V, E) there is at most one dominance wish $(l, .)$ in D.

PROPOSITION 6.1. *Let $G = (V, E \dot\cup D)$ be a hypernormal dominance graph. If G_u contains a cycle then G is unsolvable.*

Proof. The proof is by induction on the minimal number k of dominance edges in a simple cycle C of G_u. Clearly, the case $k = 0$ cannot occur. If $k = 1$ then there exists a dominance edge from a leaf l to the root of the fragment of l, and hence G is not solvable. For $k > 1$, assume that we know the result to be true for $k - 1$. C either does not contain any nodes at which its edges change directions; then it is also a cycle in G and hence, G is clearly unsolvable. Or C does change directions, then it must contain two dominance edges (l, r) and (l', r) into the same root. Both results of applying the choice rule produce graphs with a simple cycle containing $k-1$ dominance edges, so both are unsolvable. But then, G must be unsolvable as well. ☐

The converse of the above proposition is also true. If G is not solvable, then G_u contains a cycle. This statement will be a corollary of Theorem 6.2, which we will prove below.

6.2. Dominance Graphs.
The Proposition 6.1 does not carry over literally to the general case: Figure 6.1 is a counterexample. In order to state our theorem for the general case, we call a subgraph

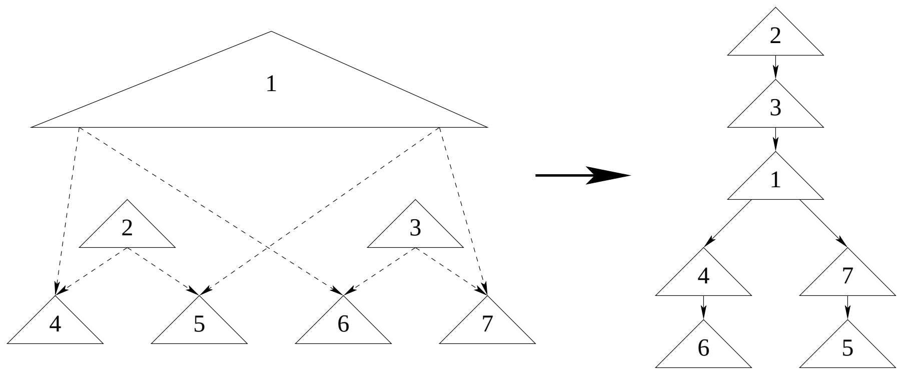


FIG. 6.1. *A solvable dominance graph and one of its solved forms. The graph contains an undirected cycle, but no hypernormal cycle.*

H_u of G_u *hypernormal* if the corresponding directed subgraph H of G is hypernormal. In particular, a *hypernormal cycle* in G_u is a cycle that contains for every leaf l at most one incident dominance edge.

THEOREM 6.2. *Let $G = (V, E \dot\cup D)$ be a dominance graph.*

(a) G is solvable iff G_u does not contain a hypernormal cycle.

(b) G is solvable iff every hypernormal subgraph of G is.

Note that this implies that a graph G is configurable iff G_u has no hypernormal cycle, by Lemma 4.4.

Proof. Part (b) follows immediately from part (a). If some hypernormal subgraph of G is unsolvable, G is unsolvable. If every hypernormal subgraph of G is solvable, G_u contains no hypernormal cycle, and hence G is solvable by part (a). We turn to part (a).

Assume first that G_u contains a hypernormal cycle C. Let D' be the dominance edges of G corresponding to edges in C. Then $G' = (V, E \dot\cup D')$ is a hypernormal dominance graph such that G'_u contains C. By Proposition 6.1, G' is unsolvable and hence G is unsolvable.

It remains to prove the converse: If G is unsolvable, G_u contains a simple hypernormal cycle. Assume that the statement is false. W.l.o.g. we may restrict our attention to reduced dominance graphs. If the choice rule is not applicable to a graph, this graph is either in solved form or has a directed cycle, so the theorem is true for these cases. Hence, there must be a "minimal" counterexample G to the statement, in the following sense:

- G is reduced and unsolvable
- G_u does not contain a hypernormal cycle
- The choice rule can be applied to G (cf. Figure 5.1). Both graphs H and H' which are generated by it are unsolvable, and both H_u and H'_u do contain hypernormal cycles.

We will derive a contradiction by showing that this implies that G_u contains a hypernormal cycle.

Suppose that v is the root and that (l, v) and (l', v) are the wishes considered in the above application of the choice rule. Let r be the root of the tree with the leaf l and r' be the root of the tree with the leaf l'.

We have a hypernormal cycle $C_1 = \{l, r'\} \circ P_1$ in H_u and a hypernormal cycle $C_2 = \{l', r\} \circ P_2$ in H'_u. Since C_1 is hypernormal, P_1 does not use any dominance edge

incident to l. If P_1 does not use some dominance edge $\{l', w\}$ incident to l', we are done since G_u contains the hypernormal cycle $\{l,v\}\{v,l'\}\circ(l' \to r')\circ P_1$, where $l' \to r'$ is the tree-path from l' to r'. So assume we can decompose $P_1 = Q_1 \circ \{l', w\} \circ R_1$, then R_1 does not use any dominance edge incident to l or l'. A similar argument gives us a decomposition $P_2 = Q_2 \circ \{l, u\} \circ R_2$ such that R_2 avoids all dominance edges incident to l and l'.
Thus we have the cycle $C = \{l', w\} \circ R_1 \circ \{l, u\} \circ R_2$ in G_u. This cycle is not necessarily hypernormal but the paths $\{l', w\} \circ R_1 \circ \{l, u\}$ and $\{l, u\} \circ R_2 \circ \{l', w\}$ are. The following lemma shows that this suffices to prove that G_u contains a hypernormal cycle:

LEMMA 6.3. *If G_u contains a cycle $C = e_S \circ S \circ e_T \circ T$ where e_S, e_T are edges and S, T are paths such that $e_S \circ S \circ e_T$ and $e_T \circ T \circ e_S$ are hypernormal, then G_u contains a hypernormal cycle.*

Before we prove the lemma, we want to give some intuition of its statement: If we can "glue" two hypernormal paths ($e_S \circ S$ and $e_T \circ T$) together such that there are "hypernormal bends"[7] at the "glue" nodes, then G_u contains a hypernormal cycle.

Proof. We may assume that C is the smallest cycle (i.e. with the least number of edges) in G_u fulfilling the condition of the lemma. We consider two cases:

- C is simple:
  Since every node on C is an inner node of at least one of the two hypernormal paths, we have that C does not contain a pair of consecutive dominance edges incident to the same leaf. And as every leaf is visited only once by C, we have that C is hypernormal.
- C is not simple:
  By the choice of C the paths $P_1 = e_S \circ S$ and $P_2 = e_T \circ T$ are simple. We may assume that none of the two is a cycle, otherwise we are done. Since C is not simple, P_1 and P_2 must have a common inner node[8]. Let v be the first inner node on P_1 which is also an inner node of P_2. We can decompose the paths at v: $P_1 = e_S \circ S' \circ f \circ S''$ and $P_2 = e_T \circ T' \circ g \circ T''$. Here f and g are edges and S', S'', T', T'' are (possibly empty) paths such that $e_S \circ S'$ and $e_T \circ T'$ end at v. Altogether, we have the following decomposition:

  $$C = e_S \circ S' \circ f \circ S'' \circ e_T \circ T' \circ g \circ T''$$

  We distinguish two cases:
  1. f is not a dominance edge:
     Then $C' = e_T \circ T' \circ f \circ S''$ is a cycle and $e_T \circ T' \circ f$ and $f \circ S'' \circ e_T$ are hypernormal paths.
  2. f is a dominance edge:
     Then the path $e_S \circ S'$ ends with a tree edge or v is not a leaf. Assume that $Q = e_S \circ S' \circ g$ is not hypernormal, then the edge $g = \{v, u\}$ must be a dominance edge and $e_S \circ S'$ must contain an other dominance edge h incident to the leaf u. By the choice of v, the node u cannot be a common inner node of P_1 and P_2. Since both paths are simple and have the same endpoints, we can conclude that u is the first node of P_1 and the last node of P_2, and hence $h = e_S$. But then the path $e_T \circ T \circ e_S$ contains both g and h, a contradiction. So, Q is hypernormal and clearly $g \circ T'' \circ e_S$ is hypernormal and $C' = e_S \circ S' \circ g \circ T''$ is a cycle.

[7] We will formally define this term in Section 7.
[8] By an inner node of P_i, we mean a node different from the first and the last node of P_i.

In both cases C' is smaller than C, and C' fulfills the condition of the lemma, a contradiction.

$\square$

7. Testing for Hypernormal Cycles. Now we show how to test for the presence of hypernormal cycles in a dominance graph in polynomial time. This immediately gives us a polynomial algorithm for testing solvability (and hence, configurability) of dominance graphs.

7.1. Reduction to Matching Theory. First, we want to recall some basic definitions from matching theory. A *matching* M in a graph H is a set of edges of H such that every node of H is incident to at most one edge in M. We call the edges in M *matching edges* and the other edges *non-matching edges*. An *alternating path* with respect to M is a (simple) path which alternately uses a matching edge and a non-matching edge. An *alternating cycle* is a cycle (of even length) which is an alternating path.

For the test we construct the following auxiliary graph A. For every edge $e = \{v, w\} \in G_u$ we have two nodes $e_v = (\{v, w\}, v)$ and $e_w = (\{v, w\}, w)$ in A. Before we define the edges of A, we want to introduce some more definitions. Let v denote a node of G_u, we call a pair of distinct edges $e = \{u, v\}$ and $f = \{v, w\}$ incident to v a *bend at* v and denote it by $\langle e, v, f\rangle$. The bend is called a *hypernormal bend* if either v is not a leaf or v is a leaf and either e or f is the tree edge incident to v. Now we are ready to define the edge set of A. We have two types of edges:

(a) For every edge $e = \{v, w\}$ we have the edge $a(e) = \{e_v, e_w\}$.

(b) For every hypernormal bend $\langle e, v, f\rangle$ we have the edge $b(\langle e, v, f\rangle) = \{e_v, f_v\}$.

Clearly, the edges of type (a) form a perfect matching M in A. The following lemma shows how hypernormal cycles in G_u are related to the auxiliary graph A:

LEMMA 7.1. *The graph G_u contains a hypernormal cycle iff the graph A contains an alternating cycle with respect to M.*

Proof. Suppose first that G_u contains a hypernormal cycle C. We may assume that C is simple. Every pair of consecutive edges on C is a hypernormal bend. Suppose $C = e_0 \circ e_1 \circ \ldots \circ e_{n-1}$, where $e_i = \{v_i, v_{(i+1) \bmod n}\}$ for $i = 0, \ldots, n-1$. Then $C' = a(e_0) \circ b(\langle e_0, v_1, e_1\rangle) \circ a(e_1) \circ \ldots \circ a(e_{n-1}) \circ b(\langle e_{n_1}, v_0, e_0\rangle)$ is an alternating cycle in A.

Suppose next that A contains an alternating cycle C'. Look at an edge of type (b) and its neighboring edges of type (a): $a(e) \circ b(\langle f, v, g\rangle) \circ a(h)$. From the construction of A we can conclude that either $e = f \wedge h = g$ or $e = g \wedge h = f$, and hence $\langle e, v, h\rangle$ is a hypernormal bend at node v. So if we delete all the edges of type (b) from C' we get a sequence $a(e_0) \circ \ldots \circ a(e_{n-1})$ of type (a) edges. Then $C = e_0 \circ \ldots \circ e_{n-1}$ is an edge-simple cycle and any pair of consecutive edges is a hypernormal bend. Now fix any leaf l visited by C. Every hypernormal bend at l contains the tree edge incident to l, and since C is edge-simple, we can conclude that C visits l only once and hence contains only one dominance edge incident to l. This proves that C is a hypernormal cycle in G_u. $\square$

Now, we want to bound the size of the auxiliary graph A. We assume that all non-leaves in the dominance graph G have outdegree at most two[9]. Observe that we have one edge of type (a) for every edge of G. For the edges of type (b) we

[9]We can replace each non-leaf with outdegree more than two and its children by a small binary tree. This construction increases the number of nodes and the number of edges only by a constant factor.

count the number of hypernormal bends at a node v. For a leaf we get outdeg_v, and for a non-leaf we have $\binom{\deg_v}{2}$. Thus the auxiliary graph A has $n' = 2m$ nodes and $m' = O(m + \sum_{v \in V}(\text{indeg}_v + 2)^2)$ edges. The graph G can be made reduced in time $O(nm)$, see [GK79, Sim88]. Then we have no parallel edges, and hence a root r with indegree greater than n must have two dominance edges from different leaves of the same tree to r, which is trivial to recognize in time $O(nm)$. So we can assume that the indegree of any root is at most n. Let us say that we have $r \le n$ roots and let d_i be the indegree of the i-th root. We have $\sum_{i=1}^{r} d_i \le m$ and $d_i \le n$. What is the maximum value of $S = \sum_i (2 + d_i)^2$? We have $S = O(n + m) + \sum_i d_i^2$. The sum $\sum_i d_i^2$ is maximized if we make the d_is as unequal as possible. So we attain the maximum if we set m/n of the d_is equal to n and all others equal to zero. Thus $\sum_{v \in V}(2+\text{indeg}_v)^2 = O(n+m)+O(m/n \cdot n^2) = O(nm)$. In our application $m = O(n)$ and the indegrees are bounded (the outdegrees are not) and hence the auxiliary graph has $n' = O(n)$ nodes and $m' = O(n)$ edges.

7.2. A first Algorithm. In this section we describe a polynomial time algorithm for testing whether a graph G_u contains a hypernormal cycle. By Lemma 7.1 and standard matching theory (see [MN99, Section 7.6.2]) this is equivalent to testing whether A contains a perfect matching different from M. A perfect matching in A has size $m = n'/2$. So the straight-forward way to do the test is to give the edges of type (a) weight m and the edges of type (b) weight $m+1$; then G_u contains a hypernormal cycle iff there is a matching in A with weight greater than m^2. A maximal weighted matching in a graph with n' nodes and m' edges can be found in time $O(n'm' \log n')$, see [GMG86].

7.3. An improved Algorithm. In this section we will describe an algorithm which directly searches for an alternating cycle in A. This algorithm has only a slightly better asymptotic running time than the algorithm above, but in practice it has performed considerably faster.

The algorithm (see Algorithm 7.1) makes at most $n'/2$ iterations. In iteration i it explores the current graph A_i in a depth first search like manner; it either detects an alternating cycle or it removes at least two nodes and some edges from the graph which cannot lie on alternating cycle. The algorithm also uses the idea of blossom-shrinking by Edmonds ([Edm65a, Edm65b]).

We describe now the ith iteration of the main loop on the graph A_i. For every node the algorithm maintains a label and a depth first search completion number. A node can be labeled as EVEN, ODD or UNLABELED. At the beginning all nodes are UNLABELED and their completion number is set to -1 to indicate that they have not been completed yet. As long as there are unlabeled nodes the algorithm picks an unlabeled node s and performs a top-level call to the (recursive) DFS procedure.

The task of this procedure is to explore and label all (currently unlabeled) nodes which lie on an alternating path starting in s. If a node v gets labeled EVEN(ODD), then there is an alternating path from s to v of even(odd) length. The algorithm stores the nodes on the path P from s to the current node on a stack called *Path*. The path consists of all nodes in the graph which have been assigned a label but no completion number.

The main portion of the work is done by the procedure DFS_explore which explores all non-matching edges $\{v, w\}$ out of an ODD labeled node v. If w is unlabeled then the current path is simply enlarged by w and its matching mate via a recursive call to DFS.

Algorithm 7.1 Test for an alternating cycle in the graph A

```
repeat // Main loop
  for all nodes v do
    Mark v as UNLABELED and assign completion number -1 to v
    S(v) ← {v} // and [v] = v
  time ← 0 // next completion number
  while there is an unlabeled node s do
    DFS(s)
  for all nodes v do
    if v = [v] then
      Remove v and all incident edges
until there is no node left in the graph
Report "No alternating cycle"

Procedure: DFS(u) // u is unlabeled
  Label u as EVEN and its matching mate v as ODD
  Push u and then v on the stack Path
  DFS_explore(v)
  if v is on top of Path then
    Pop v and u from Path
    Assign time as completion number to v and time + 1 to u, increment time by 2

Procedure: DFS_explore(v) // [v] is labeled ODD
  for all non-matching edges {v, w} with [v] ≠ [w] do
    if [w] is not labeled then
      DFS(w)
    else if [w] is labeled ODD then // shrink blossom
      b ← [w] // [b] is the base
      while [top of Path] ≠ b do
        Pop v' and u' from Path
        Unite S(b), S(v'), S(u') and make b the representant of the union
        Append u' to the list O of new odd nodes
      for all u' in O do
        DFS_explore(u')
    else // [w] is labeled EVEN
      if [w] has no completion number then
        Report "Alternating cycle found" and terminate
```

Suppose now that w is also labeled ODD, then we have discovered a so-called blossom, see Figure 7.1. As we shall see later, the node w cannot be completed, and hence it lies on P. Let $P = P' \circ P''$ be the decomposition of P such that P' ends in w. The blossom consists of the edges in the cycle $P'' \circ \{v, w\}$. The path P' is called the *stem* and the node w is the *base* of the blossom. The path $P' \circ \{w, v\}$ is an alternating path from s to v of even length, and in fact it is easy to see that for all nodes $u \neq w$ in the blossom there is an alternating path of odd length from s to u (and also one of even length).

Whenever we discover a blossom, we shrink it, this means we collapse all its nodes into the base. Conceptually, this removes all edges from the graph which connect two

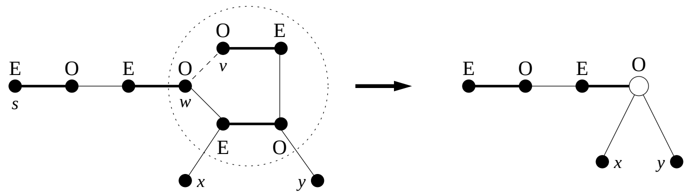


FIG. 7.1. *Discovery of a blossom (indicated by the dotted circle) and the shrinking of it. Matching edges are drawn in bold. The node w is the base of the blossom and the dashed edge $\{v, w\}$ closes the blossom.*

nodes in the blossom and replaces all edges $\{u, z\}$ by the edge $\{b, z\}$, where u is a node in the blossom and z is a node not in the blossom. We denote the current graph by A_i^{cur}. In order to do the shrinking algorithmically, we maintain a partition of the nodes into disjoint sets. For a node v, we denote by $S(v)$ the set to which v belongs. In order to identify the sets, we store for each set a canonical representant $[v]$. At the beginning, every node forms a singleton set. When we shrink a blossom, we unite the sets of all nodes in the blossom and make the base represent the resulting set. Then the nodes of A_i^{cur} correspond one-to-one to the sets in the partition and its edge set consits of the edges $\{[v], [w]\}$ such that $\{v, w\}$ is an edge of A_i with $[v] \neq [w]$. The shrink operation is justified by the following proposition:

PROPOSITION 7.2. *Let the graph G' be obtained from a graph G by shrinking a blossom of G. If G' contains an alternating cycle (of even length), then so does G.*

Proof. Standard matching theory (see for example [MN99, Section 7.7.2]). ◻
This guarantees that if the algorithm finds an alternating cycle in A_i^{cur}, we are sure A_i also contains one. Unfortunately, the converse of the above proposition is not true, i.e. shrinking may lose cycles. We will come back to this later, but let us look now at the case where DFS_explore finds an edge $\{v, w\}$ to an EVEN labeled node w. If w is already completed, we do nothing. Otherwise w lies on P and we can decompose $P = P' \circ P''$ at w such that the path P'' is an alternating path from w to v of odd length starting with a matching edge. Therefore $C = P'' \circ \{v, w\}$ is an alternating cycle in A_i^{cur} and we can terminate by Proposition 7.2.

Now suppose the algorithm terminates with a graph A_i^{fin} without reporting a cycle. We can make the following observations:

1. When an ODD labeled node v of A_i^{cur} gets assigned a completion number, then all adjacent nodes except for its matching mate u are already completed, i.e. have a non-negative completion number. And hence in A_i^{fin}, there is no edge between two ODD labeled nodes.
2. An EVEN node v in A_i^{fin} stands for itself, i.e. $S(v) = \{v\}$. And it always has an ODD matching mate with smaller completion number.

From this, we can conclude that A_i^{fin} does not contain an alternating cycle. Suppose otherwise, i.e. there is an alternating cycle $C' = \{u_0, v_0\} \circ \{v_0, u_1\} \circ \ldots \circ \{u_k, v_k\} \circ \{v_k, u_0\}$ such that the first edge is a matching edge and u_i is EVEN and v_i is ODD for $i = 0, \ldots, k$. If we denote the completion number of a node w by $c(w)$ we get by the observations from above: $c(u_0) > c(v_0) > c(u_1) > \ldots > c(u_k) > c(v_k) > c(u_0)$, a contradiction.

Now suppose, that all nodes have been labeled (and completed) and no alternating has been found, although A_i contains an alternating cycle C. We claim that for any pair v, w of nodes visited by C we have $[v] = [w]$. Or in other words, if we translate C to A_i^{fin} we get a single node which is the canonical representant of all the nodes on the cycle. Assume otherwise. We consider the edges of $C = e_0 \circ \ldots \circ e_{n-1}$ one after the other and construct a cycle C'. If for $e_i = \{u, v\}$ we have $[u] \neq [v]$, we add the edge $\{[u], [v]\}$ to C', otherwise we do nothing. This gives us a non-empty cycle in A_i^{fin}. C' visits at least two nodes, and since there are no edges between ODD nodes, we know that C' must vist an EVEN node v in A_i^{fin}. We also know that EVEN nodes stand for themselves, and hence C' contains the matching edge e incident to v. So we may assume that C' starts with e and first visits v. The second node visited by C' must be the ODD matching mate u of v. C' must leave u via a non-matching edge and enter an EVEN node v'. Repeating the arguments from above, we can conclude that C' alternates between matching edges and non-matching edges. So we have constructed an alternating in A_i^{fin}, a contradiction.

In order to finish the correctness proof of the algorithm, we just have to note that for a node v of A_i the condition $v = [v]$ holds after the construction of A_i^{fin} iff v stands for an EVEN node in A_i^{fin} or v is matched to such a node.

Let us look at the running time of the algorithm. In the ith iteration we process the graph A_i with n_i nodes and $m_i \geq n_i$ edges. In order to maintain the partition of the n_i nodes we use a union-find data structure [Tar75], thus we can perform all operations on the partition in time $O(m_i \cdot \alpha(n_i, m_i))$, where α is the so-called inverse Ackermann function[10]. For the total running time of the ith iteration we get the same asymptotic time bound. We have seen that in each iteration the algorithm removes at least two nodes from the graph, and hence there are at most $n'/2$ iterations. Therefore the total running time is $O(n'm' \cdot \alpha(n', m'))$. Together with Lemma 7.1 this yields the following theorem:

THEOREM 7.3. *The existence of a hypernormal cycle can be decided in time* $O(n'm' \cdot \alpha(n', m'))$, *where* $n' = O(m)$ *and* $m' = O(m) + \sum_{v \in V}(2 + \text{indeg}_v)^2$.

Let us finish this section with a remark about the performance of the algorithm in practice. In our examples, we never observed more than 5 iterations and the number of nodes and edges decreased drastically from iteration to iteration. When the graph contained an alternating cycle, the algorithm found it in most cases in the first iteration. So the observed total running time in practice was linear.

8. Efficient Enumeration. A first application of the polynomial-time configurability test from the previous section is to make the enumeration of solved forms more efficient. We modify the enumeration algorithm from Section 5 by testing for (undirected) hypernormal cycles in step 2 instead of directed arbitrary cycles. The recursion will terminate immediately once the graph becomes unconfigurable, and we know that the recursion depth is bounded by n^2. Thus:

COROLLARY 8.1. *A solved form of a solvable dominance graph can be constructed in time* $O(n^3 m^2 \alpha(n, nm))$. *If a dominance graph has* N *solved forms, they can be enumerated in time* $O(Nn^3 m^2 \alpha(n, nm))$.

Note that N can still be exponential in n. Also note that we can get configurations instead of solved forms in the same asymptotic time, by applying Lemma 4.4: Simplification Rule 1 can only be applied at most n^2 times either, by a similar argument about the reachability relation.

8.1. Incremental Redundancy Elimination. The enumeration algorithm in Section 5 has to maintain the transitive reduction of the dominance graph G. This

[10]This function grows extremely slow and can be considered a small constant for all practical applications (see [CLR90, Chapter 22]).

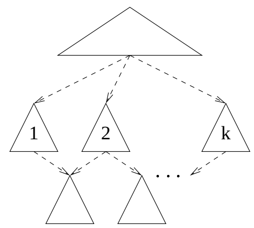


FIG. 9.1. *Embedded chain of length k.*

k	N	Time (new)	Time (old)
3	5	20	180
4	14	190	670
5	42	1210	5900
6	132	4130	12740
7	429	16630	46340
8	1430	255000	n/a

FIG. 9.2. *Runtimes on embedded chains of length k. N is the respective number of configurations. Times are in milliseconds CPU time.*

can be done in time $O(nm)$ (see [GK79, Sim88]). But for all recursive calls of the algorithm the reduction can be computed much faster, only the top-level needs to do the full-fledged reduction. This is because the instances on which recursive calls work are just reduced graphs where one irredundant edge has been added.

So we are faced with the following problem: We are given a reduced dominance graph G and an irredundant dominance edge $d = (s, t)$ which is not contained in G, and we are to compute all edges of G which become redundant by the insertion of d into G. An edge $e = (v, w)$ of G becomes redundant iff there is a path P from v to w in the graph $G \cup d$. Since G is reduced, P must use the edge d. And hence e is redundant iff in G there is a path from v to s and a path from t to w. Further, we observe that if $G \cup d$ is cyclic then any cycle must use the edge d. And hence $G \cup d$ is cyclic iff there is a node v such that in G there is a path from v to s and a path from t to v. Thus we can make $G \cup d$ reduced and test its acyclicity with the following algorithm:

- Start a depth first search in G at the node t and color all reachable nodes *red*.
- Start a depth first search in G_{rev} at the node s and color all reachable nodes *green*. (G_{rev} is obtained from G by reversing all the edges.)
- If there is a two-colored node v, report that $G \cup d$ is cyclic and stop.
- Delete all edges with *red* target node and *green* source node, and insert the edge d.

It is easy to see that the running time of the algorithm is $O(n + m)$. Note that this improvement does not lead to a better asymptotic running time of the enumeration algorithm, but it has shown considerable impact in practice.

9. Implementation and Evaluation. Going back to the application in computational linguistics described in the introduction, the algorithm for enumerating configurations that we have just sketched gives us a straightforward algorithm for enumerating models of a normal dominance constraint. We have implemented this algorithm, and this gives us a significant improvement in runtimes over earlier solvers

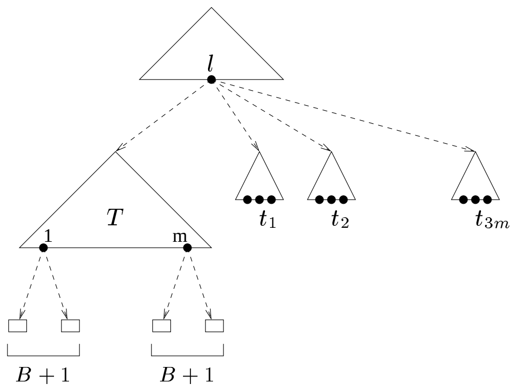


FIG. 10.1. *The dominance graph constructed in the reduction of 3-partition.*

for dominance constraints. By way of example, consider the dominance graph in Figure 9.1. This graph is an *embedded chain* of length k. Such graphs appear in the application; for instance, the graph for "John says that every linguist speaks two languages" is an embedded chain of length 2. Runtimes for enumerating all configurations of embedded chains of various lengths (on a 550 MHz Pentium III) are displayed in Figure 9.2. In the table, "new" refers to the algorithm sketched above; "old" refers to the dominance constraint solver described in [DG99].

10. Dominance Graphs with Closed Leaves. A slight extension of the configuration problem by *closed leaves* becomes NP-complete again. Beside of the principle insight, this result is also of interest to the application in computational liguistics.

In some approaches [Rey93, Bos96], pictures as in Fig. 2.2 are interpreted as dominance graphs with closed leaves instead of dominance constraints. The result of this section shows that the subtle difference in interpretation has a important consequences for complexity.

A dominance graph with closed leaves is given by a dominance graph $G = (V, E \dot\cup D)$ and a set L of leaves. The members of L are called *closed*, all other leaves are called *open*. Closed leaves cannot be the source of dominance wishes. A solved form of (G, L) with closed leaves L is a solved form $G' = (V, E \cup D')$ of G which has the additional property that there is no edge $(l, v) \in E'$ with $l \in L$, but there is an edge $(l, v) \in D'$ for every $l \notin L$. In other words, it is not allowed to attach a tree to a closed leaf, and every open leaf must be "plugged" with some other tree. We show that the configuration problem of dominance graphs with closed leaves is NP-complete by reducing the 3-partition problem to it.[11]

FACT 1 (3-partition). *Let A denote a multiset $\{a_1, \ldots, a_{3m}\}$ of integers and $B \in \mathbb{N}$ such that $B/4 < a_i < B/2$ for all i; and $\sum_{i=1}^{3m} a_i = mB$. The question is whether there is a partition $A_1 \uplus \ldots \uplus A_m$ of A such that for all i, $\sum_{a \in A_i} a = B$. The problem is NP-complete in the strong sense [GJ79, problem SP15, page 224].*

We describe the reduction now, which is shown in Figure 10.1. The tree T has m leaves. Each leaf wants to dominate $B+1$ closed subtrees (i.e., subtrees which have only closed leaves). T is required to be the child of some node l. This node l also wants to dominate the trees $t_1, \ldots, t_{3m}$. For all i, the tree t_i has $a_i + 1$ open leaves.

[11]Exactly the same reduction works if we do not require open leaves to have outgoing dominance edges in solved forms; so this modified problem is NP-complete as well.

THEOREM 10.1. *The configurability problem for dominance graphs with closed leaves is NP-complete.*

Proof. Consider an instance (A, B) of the 3-partition problem and the dominance graph G constructed in the reduction. We show that the instance (A, B) has a solution iff G is configurable.

Assume first that the 3-partition problem has a solution. Observe that each of the sets A_i must have cardinality three. Let $A_i = \{a_{x_i}, a_{y_i}, a_{z_i}\}$ be one of the sets in the partition. Then $a_{x_i} + a_{y_i} + a_{z_i} = B$. We plug t_{x_i} as child into the i-th leaf of T, t_{y_i} into some leaf of t_{x_i} and t_{z_i} into some leaf of t_{y_i}. Then the tree T has $a_{x_i} + 1 + a_{y_i} + 1 + a_{z_i} + 1 - 2 = B + 1$ open leaves below its i-th leaf. These leaves are plugged with the $B + 1$ closed subtrees which the i-th leaf of T wants to dominate. Finally, we plug l with T and obtain a configuration of G.

Assume next that the dominance graph G has a configuration. Consider the subtree plugged to the i-th leaf of T. It contains a subset A_i of the trees $\{t_1, \ldots, t_{3m}\}$. We must have $\sum_{t_j \in A_i}(a_j + 1) \geq B + 1 + |A_i| - 1$, since $B + 1$ closed subtrees must be plugged into some open leaf and since every subtree in A_i also requires an open leaf. We next show that $|A_i| \geq 3$ for all i. It is clear that A_i cannot be empty (since $B > 0$). If A_i is a singleton, i.e. $A_i = \{t_x\}$, we have a contradiction since t_x has $a_x + 1 < B/2 + 1 \leq B + 1$ leaves. Now consider the case, where A_i consists of two elements t_x and t_y. By attaching t_x and t_y below the i-th leaf of T, we obtain $a_x + 1 + a_y + 1 - 1 < B/2 + 1 + B/2 = B + 1$ open leaves, which is also a contradiction. Since each set A_i has cardinality at least three, since we have m sets, and since there are $3m$ elements to distribute, we conclude that $|A_i| = 3$ for all i. Thus $\sum_{t_j \in A_i} a_j \geq B$ for every i. Finally, we observe that we have equality since $\sum_{a \in A} a = mB$. Thus we also have a solution for the 3-partition problem. ☐

Note that for *solvability* of dominance graphs with closed leaves, Theorem 6.2 still holds. That is, solvability is still a polynomial problem. The difference to the unrestricted problem is that Lemma 4.4 breaks down: All the graphs we construct in the encoding of 3-partition are in solved form, but they may well be unconfigurable.

The relevance of this result is again in its relation to computational linguistics. There are alternative approaches to scope [Bos96] which require that the holes and roots of the trees must be paired uniquely: The roots must be "plugged" into the roots, and every hole must be plugged. This corresponds to making the holes open leaves, and all others closed leaves. Hence, we can show that the satisfiability problems of these alternative approaches must be NP-complete as well.

11. Conclusion. We have have distinguished the large and natural fragment of normal dominance constraints and shown that it has a polynomial time satisfiability problem. We have shown that this satisfiablity problem is equivalent to the configuration problem of dominance graphs, we introduced. Configurability was then reduced to the existence of particular cycles, which could finally be checked by solving a weighted matching problem.

The efficient graph algorithm we presented eliminates any doubts about the computational practicability of dominance constraints which were raised by the NP-completeness result for the general language [KNT01] and expressed e.g. in [WM99]. First experiments confirm the efficiency of the new algorithm – it is superior to the NP algorithms especially on larger constraints.

Two main directions are to be persued in the future. On the one hand side, one might want to extend the presented graph algorithm to more expressive languages than normal dominance constraints. In particular, one might try to find a polynomial time fragment of parallelism constraints [EKN01b] that are also useful for computational linguistics [EKN01a]. One the other hand side, it might be worthwile in general to investigate graph algorithms for other problems in the areas constraint programming or computational linguistics.

REFERENCES

[AC92] H. Alshawi and R. Crouch. Monotonic semantic interpretation. In *Proceedings of the 30th ACL*, pages 32–39, Kyoto, 1992.

[ADK+01] E. Althaus, D. Duchier, A. Koller, K. Mehlhorn, J. Niehren, and S. Thiel. An efficient algorithm for the configuration problem of dominance graphs. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 815–824, Washington, DC, 2001.

[Bos96] Johan Bos. Predicate logic unplugged. In *Proceedings of the 10th Amsterdam Colloquium*, 1996.

[BRVS95] R. Backofen, J. Rogers, and K. Vijay-Shanker. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language, and Information*, 4:5–39, 1995.

[CFS97] A. Copestake, D. Flickinger, and I. Sag. Minimal Recursion Semantics. An Introduction. Manuscript, `ftp://csli-ftp.stanford.edu/linguistics/sag/mrs.ps.gz`, 1997.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill Book Company, 1990.

[Cor94] Thomas Cornell. On determining the consistency of partial descriptions of trees. In *Proc. of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 163–170, 1994.

[DG99] Denys Duchier and Claire Gardent. A constraint-based treatment of descriptions. In *Proceedings of IWCS-3*, Tilburg, 1999.

[DN00] Denys Duchier and Joachim Niehren. Dominance constraints with set operators. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, LNCS. Springer, July 2000.

[DT99] D. Duchier and S. Thater. Parsing with tree descriptions: a constraint-based approach. In *Proc. NLULP'99*, Las Cruces, New Mexico, 1999.

[Edm65a] J. Edmonds. Maximum matching and a polyhedron with 0,1 - vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[Edm65b] J. Edmonds. Paths, trees, and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.

[EKN01a] Markus Egg, Alexander Koller, and Joachim Niehren. The constraint language for lambda structures. *Journal of Logic, Language, and Information*, 2001. To appear.

[EKN01b] Katrin Erk, Alexander Koller, and Joachim Niehren. Processing underspecified semantic representations in the constraint language for lambda structures. *Journal of Language and Computation*, 2001. To appear.

[ENRX98] M. Egg, J. Niehren, P. Ruhrberg, and F. Xu. Constraints over Lambda-Structures in Semantic Underspecification. In *Proceedings COLING/ACL'98*, Montreal, 1998.

[GJ79] M.R. Garey and D.S Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman and Company, 1979.

[GK79] A. Goralcikova and V. Koubek. A reduct-and-closure algorithm for graphs. In *Proceedings of the 8th Symposium on Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 301–307. Springer, 1979.

[GMG86] Z. Galil, S. Micali, and H.N. Gabow. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM Journal on Computing*, 15:120–130, 1986.

[GW98] Claire Gardent and Bonnie Webber. Describing discourse semantics. In *Proceedings of the 4th TAG+ Workshop*, Philadelphia, 1998.

[Hob83] Jerry Hobbs. An improper treatment of quantification in ordinary english. In *Proceedings of the 21st ACL*, pages 57–63, Cambridge, Massachusetts, 1983.

[KMN00] Alexander Koller, Kurt Mehlhorn, and Joachim Niehren. A polynomial-time fragment of dominance constraints. In *Proceedings of the 38th Annual Meeting of the Association of Computational Linguistics*, pages 368–375, Hong Kong, 3–6October 2000.

[KNT01] Alexander Koller, Joachim Niehren, and Ralf Treinen. Dominance constraints: Algorithms and complexity. In M. Moortgat, editor, *Proceedings of the Third Conference on Logical Aspects of Computational Linguistics (Dec. 1998)*, volume 2014 of *Lecture Note in Artificial Intelligence*, Heidelberg, 2001. Springer-Verlag.

[MHF83] M. P. Marcus, D. Hindle, and M. M. Fleck. D-theory: Talking about talking about trees. In *Proceedings of the 21st ACL*, 1983.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, Cambridge, November 1999.

[MNP00] Martin Mller, Joachim Niehren, and Andreas Podelski. Ordering constraints over feature trees. *Constraints, an International Journal*, 5(1–2):7–42, January 2000. Special Issue on CP'97, Linz, Austria.

[Mus95] R.A. Muskens. Order-independence and underspecification. In J. Groenendijk, editor, *Ellipsis, Underspecification, Events and More in Dynamic Semantics*. DYANA Deliverable R.2.2.C, 1995.

[Per00] Guy Perrier. From intuitionistic proof nets to interaction grammars. In *Proceedings of the 5th TAG+ Workshop*, Paris, 2000.

[Rey93] Uwe Reyle. Dealing with ambiguities by underspecification: construction, representation, and deduction. *Journal of Semantics*, 10:123–179, 1993.

[RVSW95] O. Rambow, K. Vijay-Shanker, and D. Weir. D-Tree grammars. In *Proceedings of the 33rd ACL*, pages 151–158, 1995.

[Sim88] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1–3):325–346, 1988.

[Smo92] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[TW67] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, August 1967.

[VS92] K. Vijay-Shanker. Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics*, 18:481–518, 1992.

[VSWR95] K. Vijay-Shanker, David Weir, and Owen Rambow. Parsing D-Tree Grammars. In *Proceedings of the Intl. Workshop on Parsing Technologies*, 1995.

[WM99] A. Willis and S. Manandhar. Two accounts of scope availability and semantic underspecification. In *Proceedings of the 37th ACL*, 1999.