

Two-sided Function Filtering

M. Pujol-Gonzalez, J. Cerquides, P. Meseguer and J. A. Rodriguez-Aguilar

IIIA - CSIC, Universitat Autònoma de Barcelona
08193 Bellaterra, Catalonia, Spain.
{mpujol|cerquide|pedro|jar}@iiia.csic.es

Abstract. Function filtering enhances dynamic programming methods working on a tree decomposition of the constraint graph. It is based on bounds for tuples: if the lower bound of tuple t is equal to or higher than a suitable upper bound, t can be discarded, decrementing the size of the message to travel in the tree decomposition. We present a new form of lower bound that tightens the lower bound of the original function filtering, so this new version –called *two-sided function filtering*– is more powerful. We provide experimental evidence of its benefits.

1 Introduction

In constraint satisfaction, inference is widely used but in a very limited form. A simple example is arc consistency: by the inspection of constraints and domains, it is able to deduce that some values will never be in a solution so they can be removed. Arc consistency is incomplete inference since it cannot always produce a solution. Inference can also be complete. Some algorithms are adaptive consistency [5], cluster tree methods [7] and bucket elimination [3]. Their temporal and spatial complexities are exponential in some parameters of the constraint graph (see [4] for details). When compared with search methods (exponential complexity in time but linear complexity in space), they look unattractive, especially when search is enhanced with the powerful machinery of local consistency coupled with global constraints [14].

In the soft constraints realm, satisfaction is replaced by optimization. This causes that problems with soft constraints become more difficult to solve than their hard counterparts. The same solving ideas are recreated here. Search methods, based on a branch-and-bound schema, are combined with soft local consistencies to filter domains [10]. Complete inference methods are easily adapted to compute the optimum, at the cost of dragging large arity constraints. Their high spatial complexity is the main drawback to be used in practice. Nevertheless, this issue is not always unavoidable: when there are ways to control the spatial complexity, complete inference can provide excellent performance [9].

While search algorithms consider assignments of individual variables, dynamic programming methods handle whole cost functions which are combined and exchanged among nodes of a suitable decomposition of the problem instance. Function filtering [13] reduces the size of cost functions by filtering out those tuples that are found unfeasible to be extended into an optimal solution. Provided a lower bound on the cost

of the best extension of the tuple, and an upper bound on the cost of the optimal solution, a tuple is filtered out when its lower bound reaches the upper bound. Authors of [13] proposed a form of computing a lower bound of the cost of the best extension of a tuple t , that we call *one-sided lower bound*. It works as follows. When computing a cost function at node i for node j , the cost of the best extension of t is computed combining the cost of t in one particular cost function at node i with the cost of functions on t variables coming from node j . We extend this form of lower bound, producing the *two-sided lower bound* where all functions at node i (and not only one) are taken into account when computing the cost of the best extension of t . Since the new lower bound tightens the old one, it is direct to see that this new approach is more powerful than the original one. Combining this new lower bound with the function filtering idea, we obtain the *two-sided function filtering* approach, that is the paper contribution.

The structure of the paper is as follows. Section 2 contains some concepts used throughout the paper. Section 3 details the original one-sided filtering method. Section 4 focuses on improving lower bounds, presenting the two-sided function filtering approach. This approach is empirically evaluated in Section 5. Finally, Section 6 draws some conclusions from this work.

2 Preliminaries

In this paper we consider soft constraints that are represented as cost functions using the weighted model [12]. A weighted CSP (WCSP) is defined as $\langle X, D, C, S(k) \rangle$ where X and D are variables and domains as in CSP. C is a finite set of constraints as cost functions; $f_T \in C$ (T is the scope of f_T) assigns costs to value tuples $t \in \prod_{x_i \in T} D_i$, such that,

$$f(t) = \begin{cases} 0 & \text{if } t \text{ is allowed} \\ 1 \dots k - 1 & \text{if } t \text{ is partially allowed} \\ k & \text{if } t \text{ is totally forbidden} \end{cases}$$

$S(k) = \langle [0, 1, \dots, k], \oplus, \geq \rangle$ is a valuation structure such that $a \oplus b = \min\{k, a + b\}$, $\top = k$, $\perp = 0$ [8]. We assume that the reader is familiar with assignments or value tuples t_S with scope S , complete tuples ($S = X$), projections over $S' \subset S$, $t_S[S']$, and concatenation of two tuples $t_S \cdot t'_S$, defined only if common variables coincide in their corresponding values. We assume that $f_T(t_S)$ (with $T \subset S$) always means $f_T(t_S[T])$. A complete tuple t_X is *consistent* if $\bigoplus_{f_T \in C} f_T(t_X) < k$, else t_X is *inconsistent*. A *solution* is a complete consistent assignment with minimum cost. Finding a solution is NP-hard. With $k = 1$ WCSP reduces to CSP.

We define the *combination* of two functions f_T and g_S as a new function $f_T \bowtie g_S$ with scope $T \cup S$ and $\forall t \in \prod_{x_i \in T} D_i, \forall t' \in \prod_{x_j \in S} D_j$ such that $t \cdot t'$ is defined, $f_T \bowtie g_S(t \cdot t') = f_T(t) \oplus g_S(t')$. Let $F = \{f_{T_1}, \dots, f_{T_m}\}$ be a set of functions, *the combination of F* , $\bowtie F$, is the function resulting from the joint combination of every function in F , namely,

$$\bowtie F = f_{T_1} \bowtie \dots \bowtie f_{T_m}$$

Let $V \subseteq X$ be a subset of the variables of the problem and t_V a tuple that assigns values to each of the variables in V . An extension of t_V to X is a tuple that keeps the

assignments of t_V and assigns new values to the variables in $X \setminus V$. If the cost of each possible extension of t_V is larger than or equal to LB , we say that LB is a lower bound of the cost of the best extension of tuple t_V . Likewise, a function f_T is a *lower bound* of function f_S , noted $f_T \leq f_S$, iff $T \subseteq S$, and $\forall t_S f_T(t_S[T]) \leq f_S(t_S)$. A function g_V is a lower bound of a set of functions F if it is a lower bound of its combination $\bowtie F$.

The *min-marginal* $f_S[T]$ of a cost function f_S over $T \subset S$ is a new cost function on T variables which assigns to each tuple t_T the minimum cost among all the extensions of t_T to S . Formally,

$$\forall t_T \quad f_S[T](t_T) = \min_{t_S \text{ extension of } t_T} f_S(t_S[T]).$$

The tightest lower bound is provided by the min-marginal. Similarly, the *min-marginal of F over V* is the min-marginal of the combination of all the functions in F , that is $(\bowtie F)[V]$.

Given a set of functions F , the time to compute the min-marginal of F over V is bounded by $\mathcal{O}(d^{|T|})$, where $T = \bigcup_{i=1}^m T_i$, and d is the size of the common domain of the variables in T . In some scenarios, this can be overdemanding. For that reason we introduce a less costly way of computing a lower bound of a set of functions. Specifically, we define $\bigvee F$, *the combination of F under V* as the result of combining the min-marginals of each of its functions over V . That is,

$$\bigvee F = f_{T_1}[T_1 \cap V] \bowtie \dots \bowtie f_{T_m}[T_m \cap V].$$

$\bigvee F$ is a lower bound of F and can be assessed in $\mathcal{O}(d^k)$ time, where parameter $k = \max(\max_{i=1}^m |T_i|, |V|)$ ¹, which can be way smaller than $\mathcal{O}(d^{|T|})$. However, this lower bound can be inferior to $(\bowtie F)[V]$.

To apply dynamic programming methods, we need a suitable decomposition of the problem instance. A *tree decomposition* (also called *joint tree* or *junction tree*) of a WCSP $\langle X, D, C, S(k) \rangle$ is a triplet $\langle T, \chi, \psi \rangle$, where $T = \langle N, E \rangle$ is a tree (N is a set of nodes and E is a set of edges), χ and ψ are labeling functions which associate with each node $i \in N$ two sets, $\chi(i) \subseteq X$ and $\psi(i) \subseteq C$ such that: (1) for each function $f_S \in C$, there is exactly one node $i \in N$ such that $f_S \in \psi(i)$ and $S \subseteq \chi(i)$; (2) for each variable $x \in X$, the set $\{i \in N \mid x \in \chi(i)\}$ induces a connected subtree of T . The *tree-width* of a tree decomposition is $tw = \max_{i \in N} |\chi(i)|$. If $(i, j) \in E$, the *separator* is $sep(i, j) = \chi(i) \cap \chi(j)$. In the following, nodes of the tree decomposition are called *clusters*. The neighbors of cluster i , $neigh(i)$, is the set of clusters linked to i in the tree decomposition. Figure 1 shows cluster i and j linked by an edge in a tree decomposition. Observe that removing the edge connecting i and j splits the tree decomposition into two different connected components, which we call subproblems (see Figure 1). Formally, we say that the i -subproblem involves every cost function in the component containing i after the edge is removed. Subproblems i and j are coupled by a set of variables they share and must agree upon, namely their separator $sep(i, j)$.

¹ Computing each $f_{T_i}[T_i \cap V]$ takes $\mathcal{O}(d^{|T_i|})$ time, while computing the whole expression takes $\mathcal{O}(d^{|V|})$ time.

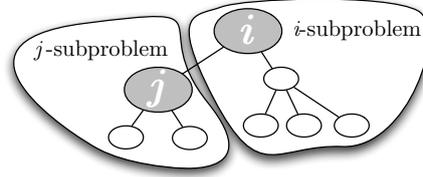


Fig. 1: Subproblems in a tree decomposition.

Cluster-Tree Elimination (CTE) is an algorithm that optimally solves WCSP by sending messages along tree decomposition edges [11, 6, 7]. CTE can be seen as the fully serial version of the Generalized Distributive Law (GDL) algorithm for the all-nodes (or all-clusters) problem [1]. Edge $(i, j) \in E$ has associated two CTE messages: $\hat{g}^{(i \rightarrow j)}$, from i to j , and $\hat{g}^{(j \rightarrow i)}$, from j to i . $\hat{g}^{(i \rightarrow j)}$ is the min-marginal computed combining all functions in $B(i, j)$ (set of functions formed by $\psi(i)$ with all incoming CTE messages except $\hat{g}^{(j \rightarrow i)}$) over $sep(i, j)$. CTE complexity is time $O(d^{tw})$ and space $O(d^s)$, where d is the largest domain size and s is the maximum separator size.

Mini-Cluster-Tree Elimination (MCTE(r)) [7]) approximates CTE (and therefore it computes solutions that are not necessarily optimal). If the number of variables in a cluster is high, it may be impossible to compute $\hat{g}^{(i \rightarrow j)}$ due to memory limitations. MCTE(r) computes a lower bound by limiting by r the arity of the functions sent in the messages. A MCTE(r) message, $G^{i \rightarrow j}$, is a set of functions that approximate the corresponding CTE message $\hat{g}^{(i \rightarrow j)}$. It is computed as $\hat{g}^{(i \rightarrow j)}$ but instead of combining all functions of set $B(i, j)$, it computes a partition $P = \{B_1, B_2, \dots, B_p\}$ of $B(i, j)$ such that the combination of the functions in every B_k does not exceed arity r . The MCTE(r) algorithm is time and space complexity $O(d^r)$.

3 One-sided Function Filtering

Function filtering [13] is a technique that reduces the size of cost functions by filtering out those tuples that are found unfeasible to be extended into an optimal solution, because their cost reach or surpass a suitable upper bound. To perform function filtering, each cluster i intending to send a cost function f_U to cluster j needs: (1) a lower bound $lb_U(t_U)$ on the cost of the best extension of each tuple t_U ; and (2) an upper bound UB on the value of the optimal solution. Provided that, we say that the cluster i filters f_U with lb_U and UB when it filters out those tuples t_U such that $lb_U(t_U) \geq UB$ (i.e. the ones that cannot be extended into an optimal solution), and sends the remaining ones.

While CTE exchanges exact cost functions among clusters, MCTE(r) exchanges approximate cost functions (one or several of arity up to r), first bottom-up and then top-down the tree decomposition. After this, each cluster i has: (1) a set of functions $\psi(i)$, containing its stake at the problem; and (2) for each neighbor j a set of functions $G^{j \rightarrow i}$ defined on the variables $sep(i, j)$. $G^{j \rightarrow i}$ stands for a summary of the j -subproblem, namely a lower bound on the cost of each tuple in that subproblem. Observe that cluster

i can assess the cost of an assignment by adding its own costs and the costs of its neighbors' subproblems. Likewise, the cluster can assess a lower bound for the costs of a tuple in the complete problem by combining its own cost functions with those received from its neighbors. Formally,

$$lb_{\chi(i)}(t_{\chi(i)}) = (\bowtie F)(t_{\chi(i)}) \quad (1)$$

where $F = \psi(i) \cup \bigcup_{j \in \text{neigh}(i)} G^{j \rightarrow i}$ (that is, the combination of all received functions in cluster i with the functions initially present in cluster i). However, the lower bound assessed in Equation (1) requires $\mathcal{O}(d^{|\chi(i)|})$ time, where $\chi(i)$ are the variables of cluster i and d the common domain size. This can be very costly to compute, both in terms of time and memory (in fact, this is the temporal complexity of the exact CTE algorithm).

As an alternative, [13] proposes the following. Let us assume that cluster i wants to send a set of functions to cluster j . Let f_U be one of these functions, $U \subseteq \text{sep}(i, j) \subsetneq \chi(i)$. A lower bound of the cost of the best extension of tuple t_U can be computed by adding a lower bound on the cost of the best extension of tuple t_U in the j -subproblem to the cost that f_U assigns to t_U . Formally,

$$lb_U(t_U) = (g_U^{j \rightarrow i} \bowtie f_U)(t_U), \quad (2)$$

where $g_U^{j \rightarrow i} = \bigcup G^{j \rightarrow i}$. Henceforth we shall refer to this lower bound as *one-sided lower bound*. In this case, the complexity of combining under U is $\mathcal{O}(d^k)$ time, where now parameter k is the maximum between $|U|$ and the arity of any function in $G^{j \rightarrow i}$, that could be cheaper than using Equation (1).

When $\text{MCTE}(r)$ runs with increasing r , the cost of the best solution found so far is an upper bound UB . At each iteration, cluster i willing to send a set of cost functions to cluster j can filter each of them separately. For each function, cluster i (a) uses Equation (2) to assess a lower bound from the last message received from j ; and (b) filters the function with this lower bound and UB . The resulting algorithm is known as IMCTEf [13].

4 Two-sided Function Filtering

Next, we aim at tightening the one-sided lower bound described above. Consider that cluster i has already received $G^{j \rightarrow i}$ from cluster j . After that, it intends to send a set of functions $G^{i \rightarrow j}$ (set that contains the function f_U mentioned in Equation (2)), summarizing the cost information in the i -subproblem, to cluster j . Since no cost function appears in both the i -subproblem and the j -subproblem, we can assess a lower bound for the complete problem by adding a lower bound of each of them. Notice that the one-sided lower bound in Equation (2) already assesses the summary of the costs of the j -subproblem from $G^{j \rightarrow i}$. Likewise, we can assess the summary of the costs of the i -subproblem from $G^{i \rightarrow j}$. Therefore, we can employ the cost summaries of both subproblems to obtain a tighter bound.

Formally, when sending cost function $f_U \in G^{i \rightarrow j}$, we compute the lower bound of tuple t_U as:

$$lb_U(t_U) = (g_U^{j \rightarrow i} \bowtie g_U^{i \rightarrow j})(t_U) \quad (3)$$

where

Given	xy	$g_{xy}^{j \rightarrow i}$,	xy	f_{xy}	,	xz	f_{xz}	, and $UB=10$
	aa	3		aa	5		aa	4	
	ab	4		ab	2		ab	3	
	ba	3		ba	8		ba	5	
	bb	3		bb	6		bb	2	

xy	$f_{xy} \bowtie f_{xz}[x] = g_{xy}^{i \rightarrow j}$		One-sided $f_{xy} \bowtie g_{xy}^{j \rightarrow i}$	Two-sided $g_{xy}^{i \rightarrow j} \bowtie g_{xy}^{j \rightarrow i}$
aa	5	3	8	5 + 3
ab	2	3	5	2 + 4
ba	8	2	10	8 + 3 \times
bb	6	2	8	6 + 3 \times

Fig. 2: One-sided vs. two-sided filtering. Ticked tuples (\times) are the ones being filtered out.

- $g_U^{i \rightarrow j} = \bigcup G^{i \rightarrow j}$ is a lower bound on the contribution of the i -subproblem.
- $g_U^{j \rightarrow i} = \bigcup G^{j \rightarrow i}$ is a lower bound on the contribution of the j -subproblem.

Observe that there is no double counting of costs because no cost function appears in both the i -subproblem and the j -subproblem. Henceforth, we will refer to the lower bound in Equation (3) as *two-sided lower bound*. The name stems from the symmetrical use of both subproblems. Hereafter, two-sided filtering refers to filtering employing the two-sided lower bound.

Comparing both lower bounds, we observe that one-sided lower bound computes a different lower bound for each function f_U that cluster i wants to send to cluster j (Equation (2)), while two-sided lower bound computes the same lower bound for all functions to be sent from cluster i to cluster j , namely the lower bound given by Equation (3).

As example, consider that cluster i has received a set of functions $G^{j \rightarrow i}$, which combined under $\{x, y\}$ produces the function $g_{xy}^{j \rightarrow i}$ shown in Figure 2. Furthermore, cluster i knows that the cost of the optimal solution is smaller than or equal to 10 ($UB = 10$). Now, it wants to send functions $G^{i \rightarrow j} = \{f_{xy}, f_{xz}\}$ (in Figure 2) to cluster j . Consider that it starts by sending function f_{xy} . Cluster i can calculate the one-sided lower bound using Equation (2), filtering out tuple $(x=b, y=a)$ as shown in Figure 2. Alternatively, the cluster can compute the two-sided lower bound using Equation (3), by assessing the lower bound on the contribution of its own subproblem, namely $g_{xy}^{i \rightarrow j} = \bigcup_{xy} G^{i \rightarrow j} = f_{xy} \bowtie f_{xz}[x]$. Figure 2 shows that two-sided filtering performs better, keeping only the tuple $(x=a, y=b)$ as feasible.

5 Empirical evaluation

In this section we empirically compare the performance of IMCTef when using one-sided filtering and two-sided filtering. For each experiment, we track the amount of memory used by the algorithm (the maximum amount of memory required by the algorithm in its whole execution, from $r = 2$ until the problem is solved) along with the total amount of computation (as the number of constraint checks performed, which are directly related with the CPU time used). Moreover, we conducted signed rank tests [15]

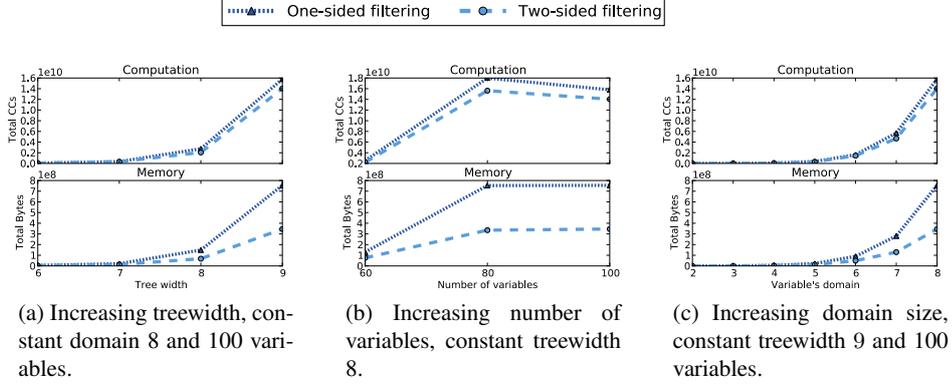


Fig. 3: Experimental results of one-sided filtering against two-sided filtering. Median of constraint checks and bytes exchanged are reported.

on all results to ensure that differences between methods are statistically significant ($\alpha = 0.01$).

One may be curious about the number of tuples that are filtered by our method. We report the amount of memory used and not the number of filtered tuples because not all filtered tuples cause the same savings in memory (savings depends on tuple's arity). Since we provide results aggregating increasing arity limits, we believe that the number of saved bytes is a more precise measure than the number of filtered tuples. It is worth noting that filtering a tuple in an early iteration implies that none of its extensions have to be considered in the future ones; therefore, filtering a tuple at a given iteration has a multiplicative effect in future iterations.

It is well-known that the treewidth is the most important indicator of problem hardness for CTE-based algorithms (the temporal complexity is exponential in such parameter). Hence, we segmented our experiments according to this parameter, and ensured that all algorithms use the very same tree decomposition when solving the same problem instance. We generated hard instances characterizing each scenario by three parameters: number of variables, variables' domain size, and treewidth. For each scenario, we generated 100 problems by: (1) randomly drawing problem structures following an Erdős-Rényi $G(n, p)$ model [2]; (2) selecting those structures having the treewidth requested for the scenario; and (3) randomly drawing costs from a $\mathcal{N}(0, 1)$ normal distribution (function costs are made positive by adding its minimum value to each function).

First, we ran an experiment to evaluate the savings as the treewidth increases. We generated scenarios with 100 variables of domain 8, and treewidths ranging from 6 to 9. Figure 3a shows that two-sided filtering reduces, with respect to one-sided filtering, the amount of memory required by a median of more than 25% for the easier problems (treewidth 6). It achieves even better results for the harder problems (more than 50% for the set with treewidth 9).

Next, we arranged an experiment to assess the impact of increasing numbers of variables. Hence, we generated 5 sets of 100 problems, with an increasing number of variables for each set. Since we only wanted to measure the impact of the varying number of variables, we generated many random problem structures and selected only those that yielded a fixed treewidth of 8. Figure 3b shows the median results achieved by both approaches on each set of problems. Notice that using two-sided instead of one-sided filtering reduces the total amount of bytes by more than 40% in most cases, while achieving a 54% reduction in the 100-variables problems set. Nevertheless, there is an interesting trend change in both computation and memory requirements past the 80-variables set. The cause of this change is that all problems have the very same treewidth of 8. Therefore, as the number of variables increases, the resulting problems are sparser.

Finally, we designed an experiment to measure the trend of both filtering styles as the variables' domain sizes increase. Thus, we generated scenarios with 100 variables, treewidth 9 and domain sizes ranging from 2 to 8. Once again, two-sided filtering achieves significant memory savings for all the experiment's problems. Further, as the domain increases, so do the savings with respect to one-sided filtering: starting with a narrow 7% reduction for domains of size 2, and reaching more than 50% reduction for the toughest scenario (domain size 8).

6 Conclusions

We have presented the two-sided function filtering approach in the context of WCSP, when soft constraints are represented as cost functions. This approach comes from the combination of the two-sided lower bound computation with the function filtering idea. Given a tree decomposition, two-sided lower bound considers the aggregation of costs coming from the two disjoint subproblems, such that its union constitutes the whole problem instance. Specifically, two-sided lower bound for tuple t at cluster i considers the costs of t in the subproblem i coming not only from function f_U to be send to cluster j , but also from other functions of cluster i . This cost is combined with the cost coming from subproblem j , to compute the cost of t 's best extension. This two-sided lower bound directly extends one-sided lower bound, and it is straightforward to see that it is more powerful. Experimentally, we have shown the benefits of this approach with respect to one-sided function filtering in both time and memory, using a number of experiments.

References

1. S. M. Aji and R. J. Eliece. The generalized distributive law. *IEEE Trans. Inf. Theory*, 46(2):325–343, 2000.
2. B. Bollobas. *Random Graphs*. Cambridge University Press, 2001.
3. R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.
4. R. Dechter. *Constraint Processing*. Elsevier Science, 2003.
5. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1987.
6. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38, 1989.
7. K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166:165–193, 2005.
8. J. Larrosa. Node and arc consistency for weighted csp. In *Proc. AAAI*, 2002.
9. J. Larrosa, E. Morancho, and D. Niso. On the practical applicability of bucket elimination: Still-life as a case study. *Journal of Artificial Intelligence Research*, to appear, 2005.
10. J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159, 2004.
11. S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal statistical Society, Series B*, 50(2):157–224, 1988.
12. P. Meseguer, F. Rossi, and T. Schiex. *Soft Constraints*, chapter 9 of Handbook of Constraint Programming, pages 281–328. Elsevier, 2006.
13. M. Sánchez, J. Larrosa, and P. Meseguer. Improving tree decomposition methods with function filtering. In *IJCAI*, pages 1537–1538, 2005.
14. W. J. van Hoes and I. Katriel. *Global Constraints*, chapter 6 of Handbook of Constraint Programming, pages 169–208. Elsevier, 2006.
15. F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1:80–83, 1945.