Tree Decomposition with Function Filtering

Martí Sánchez[1], Javier Larrosa[2], and Pedro Meseguer[1]

[1] Institut d'Investigació en Intel.ligència Artificial
Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{marti|pedro}@iiia.csic.es

[2] Dep. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Jordi Girona, 08028 Barcelona, Spain
larrosa@lsi.upc.es

Abstract. Besides search, complete inference methods can also be used to solve soft constraint problems. Their main drawback is the high spatial complexity. To improve its practical usage, we present an approach to decrease memory consumtion in tree decomposition methods, a class of complete inference algorithms. This approach, called function filtering, allows to detect and remove some tuples that appear to be consistent (with a cost below the upper bound) but that will become inconsistent (with a cost exceeding the upper bound) when extended to other variables. Using this idea, we have developed new algorithms CTEf, MCTEf and IMCTEf, standing for cluster, mini-cluster and iterative mini-cluster tree elimination with function filtering. We demonstrate empirically the benefits of our approach.

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 [6] and bucket elimination [7]. Their temporal and spatial complexities are exponential in some parameters of the constraint graph (see [8] for details). When compared with search methods (exponential complexity in time but lineal complexity in space), they look unattractive, especially when search is enhanced with the powerful machinery of local consistency coupled with global constraints.

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 [11]. 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 [10].

The simplest form to control spatial complexity is the use of an upper bound of the optimal cost. This allows one to remove tuples whose cost exceed the upper bound, because their will never contribute to the optimum. Good upper bounds can be found by problem inspection, sampling or local search. In addition to upper bound usage, the basic operation of complete inference, constraint combination, should be handled with extreme care due to its multiplicative nature. Strategies that anticipate if some tuples will not produce a solution should be used, to limit as much as possible the combinatorial explosion. This paper is a step in that direction.

In this paper, we present a new strategy to decrease the memory consumption of tree decomposition methods, a class of complete inference algorithms, when applied to weighted CSP. Tree decomposition methods work on a decomposition of the problem with a tree structure. They solve subproblems or "parts", sending the result of one part to the rest of the problem. The tree structure permits an orderly exchange of information. The key idea we have pursued is as follows. When combining constraints in one part of the problem, if it happens that some of the resulting tuples would become unacceptable after sending them to another part, would it not be better to detect those tuples before sending, and eliminate them once and for all? In some cases we are able to detect that some tuples, apparently acceptable when solving a subproblem (that is, with a cost below the upper bound), will become unacceptable (with a cost exceeding the upper bound) when used in another subproblem, so we can remove them and decrease the memory usage. Obviously, it is possible to find problems where our method causes no benefits, but in this case it causes no harm as well. [1]

This technique is called *function filtering*, and it has been applied to hard constraints [12]. In the soft case, approximation techniques that limit the arity of the subproblem to solve can be successfully combined with function filtering, so successive iterations can increment the size of the subproblem to solve without increasing memory usage.

The paper structure is as follows. In Section 2 we summarize the notions used in the rest of the paper. In Section 3, we present the idea of function filtering. In Section 4 we apply function filtering to tree decomposition methods, producing the CTEf, MCTEf and IMCTEf algorithms, that stand for cluster, mini-cluster and iterative mini-cluster tree elimination with function filtering. Experimental results appear in Section 5, showing the obtained benefits on a set of benchmarks. Finally, Section 6 contains some conclusions.

[1] A similarity with arc-consistency (AC) can be stated here. You can find problems on which AC causes no change, but this does not invalidate AC as a extremely useful notion in constraint reasoning.

2 Preliminaries

2.1 Weighted CSP

Valuation structures are algebraic entities to specify costs in valued CSP. We use a particular structure $S(k)$ [9], for weighted CSP. Formally,

Definition 1. *A* valuation structure *is a triple* $S = \langle E, \oplus, \succeq \rangle$, *where* E *is the set of costs totally ordered by* $\succeq$, *and* $\oplus$ *is the binary internal operation to combine costs. The maximum and minimum costs are denoted as* $\top$ *and* $\bot$. $\oplus$ *is commutative, associative, monotone,* $\bot$ *is the neutral element and* $\top$ *is the annihilator.*

Definition 2. The valuation structure $S(k)$ *is a triple* $\langle [0, 1, ..., k], \oplus, \geq \rangle$ *where* $\top = k$, $\bot = 0$ *and*

- $k \in [1, \ldots, \infty]$,
- $a \oplus b = min\{k, a + b\}$, *and*
- $\geq$ *is the standard order among naturals.*

Definition 3. *A* Weighted CSP *(WCSP) is a tuple* $\langle X, D, C, S(k) \rangle$ *where,*

- $X = \{x_1, ..., x_n\}$ *is a set of* n *variables;*
- $D = \{D_1, ..., D_n\}$ *is a set of finite domains, each variable* $x_i \in X$ *taking its values in* D_i*;*
- C *is a finite set of constraints as cost functions. Each function* $f \in C$ *relates a number of variables* $var(f) = \{x_{i_1}, \ldots x_{i_r}\}$ *called its scope, and assigns costs to tuples* $t \in \prod_{x_i \in var(f)} D_i$ *such that,*

$$f(t) = \begin{cases} 0 & \textit{if } t \textit{ is allowed} \\ \textit{[1 \ldots\ k-1]} & \textit{if } t \textit{ is partially allowed} \\ k & \textit{if } t \textit{ is totally forbidden} \end{cases}$$

- $S(k)$ *is a valuation structure.*

An *assignment* or tuple t_S on a sequence of variables $S = (x_1, x_2, \ldots, x_k)$, is a sequence of values $(a_1, a_2, \ldots, a_k)$ such that a_1 is the value for x_1, a_2 is the value for x_2 and so on. An assignment t_S is complete if $S = X$. Given $S' \subset S$, $t_S[S']$ is the tuple obtained removing from t_S the values of variables in $S - S'$. If S is implicitly assumed or irrelevant, we write directly t. For clarity, we assume that $f(t_S)$ (with $var(f) \subset S$) always means $f(t_S[var(f)])$, so we select from tuple t_S the values of variables in f and ignore the others. The concatenation of two tuples t_S and t'_T, noted $t.t'_{S \cup T}$, is a new tuple on $S \cup T$ formed by the union of its values, and it is only defined if common variables coincide in their corresponding values. A complete assignment t_S is *consistent* if $\bigoplus_{f \in C} f(t) < k$. Else t_S is *inconsistent*. A *solution* of a WCSP is a complete consistent assignment with minimum cost. The problem of finding a solution is NP-hard. It is easy to check that WCSP with $k = 1$ reduces to classical CSP.

We define two operations on functions:

- **Projecting out**. Given a function f, projecting out variable $x \in var(f)$, denoted $f_{\Downarrow x}$, is a new function with scope $var(f) - x$. Every tuple removing x component is present in the projection and its cost is the minimum among all permitted x extensions: $f_{\Downarrow x}(t) = \min_{a \in D_x}(f(a.t))$. Projecting out the variable of a unary function produces a constant. Any constant can be considered an empty scope function.
- **Sum**. Given two functions f and g, its sum $f + g$ is a new function with scope $var(f) \cup var(g)$ and $\forall t \in \prod_{x_i \in var(f)} D_i$, $\forall t' \in \prod_{x_j \in var(g)} D_j$ such that $t.t'$ is defined, $(f + g)(t.t') = f(t) \oplus g(t')$.

Definition 4. *Function g is a* lower bound *of function f, denoted $g \leq f$, if $var(g) \subseteq var(f)$ and for all possible tuples t of f, $g(t) \leq f(t)$. Abusing notation, a set of functions G is a* lower bound *of function f iff $(\sum_{g \in G} g) \leq f$*

Property 1. For any function f, $(f_{\Downarrow x})$ is a lower bound of f.

Property 2. $\sum_{f \in F}(f_{\Downarrow x}) \leq (\sum_{f \in F} f)_{\Downarrow x}$ holds.

2.2 Tree decomposition

A tree decomposition of a WCSP is a clustering of the functions in C such that clusters are linked if they share variables and form an acyclic tree network. Formally,

Definition 5. *A* tree decomposition *for a WCSP $\langle X, D, C, S(k) \rangle$ is a triplet $\langle T, \chi, \psi \rangle$, where $T = \langle V, E \rangle$ is a tree. χ and ψ are labelling functions which associate with each vertex $v \in V$ two sets, $\chi(v) \subseteq X$ and $\psi(v) \subseteq C$ that satisfy the following conditions:*

1. *For each function $f \in C$, there is exactly one vertex $v \in V$ such that $f \in \psi(v)$. In addition, $var(f) \subseteq \chi(v)$.*
2. *For each variable $x \in X$, the set $\{v \in V | x \in \chi(v)\}$ induces a connected subtree of T.*

Tree decompositions for CSP often relax condition (1) by requiring that any function $f \in C$ must appear in *at least* one vertex $v \in V$ of the decomposition (see [8]). For WCSP, if function f appears in two vertices, tree decomposition methods could add twice its contribution. For this reason, the *exactly* condition is required.

Definition 6. *The* tree-width *of a tree decomposition is the maximum number of variables in a vertex minus one $tw = max_{v \in V}|\chi(v)| - 1$. Let (u, v) be an edge of a tree-decomposition, the* separator *of u and v is $sep(u, v) = \chi(u) \cap \chi(v)$, formed by the common variables between two vertices of the decomposition. We will call s the maximum separator size $s = max_{(u,v) \in E}|sep(u, v)|$. The* eliminator *of u and v is defined as $elim(u, v) = \chi(u) - sep(u, v)$, and represent the variables in u that are not present in v.*

In [3] tree decomposition is defined only for binary graphs and hyper-tree decomposition for hyper-graphs. Following [8] we use the concept of tree decomposition of a CSP referring to an hyper-tree decomposition of the hyper-graph formed by the functions of the CSP. We also extend this definition for WCSP imposing that every constraint must appear exactly once in all clusters.

Example 1. The crossword puzzle of Figure 1 is a WCSP $\langle\{x_0,\ldots,x_9\},\{a,\ldots,z\},\{f_1,\ldots,f_4\},S(k)\rangle$, with a variable per cell, functions correspond to vertical and horizontal slots and accepts words of numbers from "zero" to "ten", that can also be reversed. The cost of each word is its number or its number plus one, if reversed. Any other word costs k.

For example: $f_1(x_1,x_2,x_3,x_4)=\{$(zero,0), (orez,1), (four,4), (rouf,5), (five,5), (evif,6), (nine,9), (enin,10)$\}$ and $f_2(x_7,x_8,x_9)=\{$(one,1), (eno,2), (two,2), (owt,3), (six,6), (xis,7), (ten,10), (net,11)$\}$. An optimal solution tuple is $\{(x_0,z),(x_1,z),(x_2,e),(x_3,r),(x_4,o),(x_5,r),(x_6,n),(x_7,o),(x_8,n),(x_9,e)\}$ with cost 2.

f_1	x_1	x_2	x_3	x_4
0	z	e	r	o
1	o	r	e	z
4	f	o	u	r
5	r	u	o	f
5	f	i	v	e
6	e	v	i	f
9	n	i	n	e
10	e	n	i	n

f_2	x_7	x_8	x_9
1	o	n	e
2	e	n	o
2	t	w	o
3	o	w	t
6	s	i	x
7	x	i	s
10	t	e	n
11	n	e	t

f_3	x_0	x_2	x_5	x_7
0	z	e	r	o
1	o	r	e	z
4	f	o	u	r
5	r	u	o	f
5	f	i	v	e
6	e	v	i	f
9	n	i	n	e
10	e	n	i	n

f_4	x_4	x_6	x_9
1	o	n	e
2	e	n	o
2	t	w	o
3	o	w	t
6	s	i	x
7	x	i	s
10	t	e	n
11	n	e	t

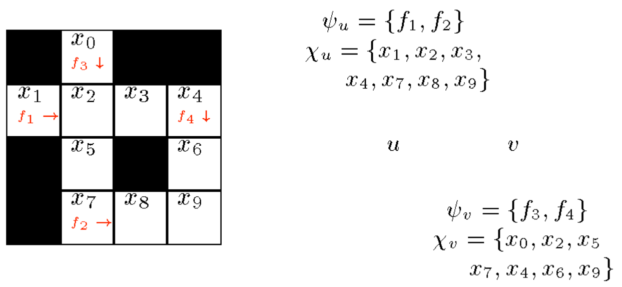


Fig. 1. Upper: crossword functions. Lower left: the crossword puzzle. Lower right: a possible tree decomposition.

```
procedure CTE($\langle X, D, C, k\rangle$, $\langle\langle V, E\rangle, \chi, \psi\rangle$)
1 for each $(u,v) \in E$ s.t. all $m_{(i,u)}, i \neq v$ have arrived do
2     $B \leftarrow \psi(u) \cup \{m_{(i,u)} \mid (i,u) \in E, i \neq v\}$;
3     $m_{(u,v)} \leftarrow (\sum_{f \in B} f) \Downarrow_{elim(u,v)}$;
4     send $m_{(u,v)}$;
```

Fig. 2. The CTE algorithm. $\langle X, D, C, k\rangle$ is a WCSP instance and $\langle\langle V, E\rangle, \chi, \psi\rangle$ is its tree decomposition.

2.3 Cluster and mini-cluster tree elimination

Cluster-Tree Elimination (CTE) is a generic algorithm that can be used for CSP solving and unifies other inference algorithms such as Bucket Elimination [1, 7]. CTE also solves a constraint optimization problem by sending messages along every edge of a tree decomposition of the problem. Concepts of this Section are more extensively described in [8].

Given a tree decomposition $\langle\langle V, E\rangle, \chi, \psi\rangle$, every edge $(u,v) \in E$ has associated two *CTE messages* that we denote $m_{(u,v)}$, from u to v, and $m_{(v,u)}$, from v to u. Message $m_{(u,v)}$ is a function computed summing all functions in $\psi(v)$ with all incoming CTE messages except from $m_{(v,u)}$ and then projecting out the variables in u not mentioned by v, that is variables in $elim(u,v)$. $m_{(u,v)}$ has scope $sep(u,v)$.

In Figure 2 we present the CTE algorithm. Line 1 is a loop that looks for edges such that all their incoming messages but one have arrived. Lines 2 gathers the set of functions to be summed. Line 3 performs the sum and projection.

Let $T(u,v)$ (resp. $T(v,u)$) denote the subtree of T containing the connected component containing vertex u (resp. v) after the removal of edge (u,v).

Property 3. $m_{(u,v)}(t)$ is equal to the minimum cost of extending tuple t to the subproblem induced by $T(u,v)$. This is guaranteed by the correctness of the algorithm.

The complexity of CTE is time $O(d^{tw+1})$ and space $O(d^s)$ where tw is the tree-width, d is the largest domain size and s is the maximum separator size.

In Figure 3 we can see an execution of CTE on example 1. Original functions have size 8. Once the messages have been sent we can compute the solution in any of the two nodes. For example, in v the minimum cost of $f_2 + f_3 + m_{(u,v)}$ is the optimal solution.

Mini-Cluster-Tree Elimination (MCTE(r)) is an approximation schema for the CTE algorithm. When the number of variables in a cluster is too high, it

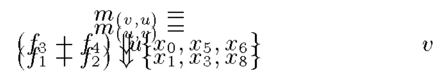

$m_{(v,u)} \equiv$
$m_{(u,v)} \equiv$
$(f_3 + f_4) \Downarrow \{x_0, x_5, x_6\}$
$(f_1 + f_2) \Downarrow \{x_1, x_3, x_8\}$
u
v

Fig. 3. The 2 CTE messages of edge (u,v) of example 1.

is not possible to compute a single message that captures the joint effect of all functions of the cluster plus all incoming messages due to memory limitations. In this case, MCTE(r) computes a lower bound of the problem by limiting by a constant r the arity of the functions sent in the messages. This is because we can not afford to compute one single function that will be of high arity and then project it.

A *MCTE(r) message*, noted $M_{(u,v)}$, is a set of functions that approximate the corresponding CTE message $m_{(u,v)}$ (namely $M_{(u,v)} \leq m_{(u,v)}$). It is computed as $m_{(u,v)}$ but instead of summing all functions of set B (line 2 of CTE algorithm in Figure 2), it computes a partition $P = \{B_1, B_2, \ldots, B_p\}$ of B such that the sum of the functions in every B_i does not exceed arity r. We compute $M_{(u,v)}$ from P by summing all functions in every partition and project out from each resulting function the variables not mentioned by node v. The MCTE(r) algorithm is obtained replacing line 3 of the CTE algorithm by the following lines,

```
3.1    $P \leftarrow partitioning(B, r)$;
3.2    $M_{(u,v)} \leftarrow \{(\sum_{f \in B_i} f) \Downarrow elim(u,v) | B_i \in P\}$;
```

MCTE(r) time and space complexity is $O(d^r)$.

3 Function Filtering

A *nogood* is a tuple t that cannot be extended into a complete consistent assignment. Nogoods are useless for solution generation, so they can be eliminated as soon as are detected.

Typically, a cost function f is stored in an array T_f, where tuples t are the indexes and $T_f[t]$ stores $f(t)$, the cost of t. The space used by T_f is $\Theta(\prod_{i \in var(f)} |D_i|)$. Costs can be retrieved in constant time. An alternative is to store function f as a set S_f containing all pairs $(t, f(t))$ with cost less than k, $S_f = \{(t, f(t)| \; t \in \prod_{x_i \in var(f)} D_i, f(t) < k\}$. We define the *size* of a function f, denoted $|f|$, as the number of tuples with cost less than k. The space used by S_f is $\Theta(|f|)$, which can be smaller than the space of T_f if f contains many inconsistent tuples. If S_f is implemented as a hash table, $f(t)$ can be retrieved in constant time.

In the following, we will assume that functions are stored as sets of pairs. Then, computing $f \Downarrow_x$ has time complexity $O(|f|)$. Regarding the sum of two functions $f + g$, there are two basic ways to compute it: (i) iterate over all the combinations of $(t, f(t)) \in S_f$ and $(t', g(t')) \in S_g$ and, if they match, compute $(t \cdot t', f(t) \oplus g(t'))$, which has complexity $O(|f||g|)$, and (ii) compute every tuple t over $var(f) \cup var(g)$ and retrieve from S_f and S_g the $f(t)$ and $g(t)$ values, which has complexity $O(exp(|var(f) \cup var(g)|))$. Since one can choose the best option beforehand, the cost is $O(\min\{|f||g|, exp(|var(f) \cup var(g)|)\})$. Observe that the efficiency of computing the previous operations depends on the size of the functions.

We now introduce the *function filtering* operation, which allows us to reduce the size of a function f before operating with it. The idea is to anticipate the detection of nogoods of f in order to remove them from S_f as soon as possible.

Definition 7. *The* function filtering *operation applied to a function f from a set of functions H, noted $\overline{f}^H$, is the process of performing a consistency test to every tuple t of f after adding the contribution of every function in H. Every tuple that reaches the upper bound k is removed from S_f.*

$$\overline{f}^H(t) = \begin{cases} f(t) & \textit{if } \big(\bigoplus_{h \in H} h(t)\big) \oplus f(t) < k \\ k & \textit{otherwise} \end{cases}$$

Suppose that we know that f will be eventually summed with g. If there is a tuple $t, (t, f(t)) \in S_f$ such that $t \cdot t'$ will become a nogood after the sum for any $t', (t', g(t')) \in S_g$, we can safely remove $(t, f(t))$ from S_f right away. The following Property formalizes the previous observation.

Property 4. Let f (resp. g) be a function and F (resp. G) a lower bound. When summing f and g, if previously we filter each function with the lower bound of the other function, the result is preserved. Namely,

$$\overline{f}^G + \overline{g}^F = f + g$$

Besides, the sum is done with functions of smaller size. Thus, it is presumably done more efficiently.

Example 2. Consider node u of tree decomposition of example 1 with $\psi_u = \{f_1, f_2\}$. Potentially $|f_1| = 26^4$ but as we record consistent tuples only ($k = \infty$) we have $|f_1| = |f_2| = 8$. They do not share any variable so $|f_1 + f_2| = 64$. If we set $k = 5$, this causes that some tuples of f_1 and f_2 become nogoods and they can be eliminated. Now, $|f_1| = 3$, $|f_2| = 4$. To make $|f_1 + f_2| = 8$, we need $3 * 4 = 12$ operations. To use Property 4, we take as G the set formed by the function $f_2 \Downarrow \{x_7, x_8, x_9\}$, that is, $G = \{f_2 \Downarrow \{x_7, x_8, x_9\}\} = \{1\}$ (G is a lower bound of f_2 by Property 1). $|\overline{f_1}^G| = 2$. Filtering with G allows us to add 1 to every tuple of f_1, which causes that tuple $(four, 4)$ becomes a nogood (it reaches $k = 5$) and can be eliminated. Therefore, we only need $2 * 4 = 8$ operations to compute the sum.

The following property shows that filtering functions can be safely brought inside summations, anticipating the detection of nogoods and reducing the size of functions.

Property 5. Let f and g be two functions, and H a set of functions, $f \notin H, g \notin H$. We have that,

$$\overline{f + g}^H = \overline{\overline{f}^H + \overline{g}^H}^H$$

Example 3. Property 5 is better understood taking H as a lower bound of a function h that has to be added with f and g. In the example 1 solved by CTE, functions f_1 and f_2 of node u have to be added with $m_{(v,u)}$. Functions in node v projecting out variables in $elim(v, u)$ form a set that is a lower bound of $m_{(v,u)}$

(see Properties 1 and 2). So we take $H = \{f_3 \Downarrow \{x_0, x_5\}, f_4 \Downarrow \{x_6\}\}$, which are as follows,

$f_3 \Downarrow_{\{x_0,x_5\}}$	x_2	x_7
0	e	o
1	r	z
4	o	r
5	u	f
5	i	e
6	v	f
10	n	n

$f_4 \Downarrow_{x_6}$	x_4	x_9
1	o	e
2	e	o
2	t	o
3	o	t
6	s	x
7	x	s
10	t	n
11	n	t

To compute $\overline{f_1 + f_2}^H$, we first compute $f_1 + f_2$. Since they have no variables in common $|f_1+f_2| = 64$. Applying filtering with H, we realize that values z, r, f for x_4 are not permitted by $f_4 \Downarrow_{x_6}$, so all tuples including them are eliminated (32 tuples). We also realize that values t, s, x are not permitted by $f_3 \Downarrow_{\{x_0,x_5\}}$, so all remaining tuples including them are eliminated (16 tuples). Now $|\overline{f_1 + f_2}^H| = 16$. As Property 5 says, we could realize this fact by filtering functions f_1 and f_2 in advance, and then filtering their sum. Filtering f_1 removes tuples with forbidden values for x_4, $|\overline{f_1}^H| = 4$, and filtering f_2 removes tuples with forbidden values for x_7, $|\overline{f_2}^H| = 4$. Then, $|\overline{f_1}^H + \overline{f_2}^H| = 16$, and additional filtering causes no removals. So the application of Property 5 allows us to save $64 - 16 = 48$ tuples, a 75% of the initial memory.

Previous discussion implicitly assumes $k = \infty$. Lower values of k causes further savings. For instance, let us assume $k = 5$. Then, $|f_1+f_2| = 8$. Filtering with H causes to remove all tuples with z for x_4 and t for x_7 (5 tuples). In addition, the cost of two tuples reaches k so they are eliminated. Now, $|\overline{f_1 + f_2}^H| = 1$. Applying Property 5, we first filter f_1 and f_2 with H, which leaves a single tuple in each function $|\overline{f_1}^H| = |\overline{f_2}^H| = 1$, and additional filtering causes no removals. So the application of Property 5 allows us to save $8 - 1 = 7$ tuples, a 87% of the initial memory.

4 CTE and MCTE with Function Filtering

Now we integrate the idea of filtering into the CTE schema. First, we define a *filtering* tree-decomposition which adds a new labelling ϕ to a tree-decomposition that will be used for filtering purposes.

Definition 8. *A filtering tree-decomposition of a WCSP is a tuple $\langle T, \chi, \psi, \phi\rangle$ where:*

- *$\langle T, \chi, \psi\rangle$ is a tree-decomposition as in definition5.*
- *ϕ is a labelling. $\phi(u, v)$ is a set of functions associated to edge $(u, v) \in E$ with scope included in sep(u, v). $\phi(u, v)$ must be a lower bound of the corresponding $m_{(u,v)}$ CTE message (namely, $\phi(u, v) \leq m_{(u,v)}$).*

The new algorithms CTEf and MCTEf(r) use a filtering tree decomposition. They are essentially equivalent to CTE and MCTE(r) except in that they use $\phi(u,v)$ for filtering functions before computing the message $m_{(u,v)}$ or $M_{(u,v)}$. The pseudo-code of CTEf is obtained by replacing line 3 of the algorithm by line,

```
3    $m_{(u,v)} \leftarrow \overline{\sum_{f \in B} f}^{\phi(u,v)} \Downarrow_{elim(u,v)}$;
```

Besides, the computation of the new line 3, will make discretional use of Property 5. Similarly for MCTEf(r) we replace line 3 by two lines,

```
3.1    $P \leftarrow partitioning(B, r)$;
3.2    $M_{(u,v)} \leftarrow \{\overline{(\sum_{f \in B_i} f)}^{\phi(u,v)} \Downarrow_{elim(u,v)} | B_i \in P\}$;
```

The effectiveness of the new algorithms will depend on the ability of finding good lower bounds $\phi(u,v)$ for the messages $m_{(u,v)}$ (resp. $M_{(u,v)}$). If we use dummy lower bounds (i.e, $\phi(u,v) = \emptyset$, for all $(u,v) \in E$), CTEf (resp. MCTEf(r)) is clearly equivalent to CTE (resp. MCTE(r)). It is important to note that the algorithms will be correct as long as $\phi(u,v)$ is a true lower bound which can be computed with either a domain-specific or general technique (see [3] [2] [4] for a collection of general lower bound techniques). An option is to include in $\phi(u,v)$ all the original functions used to compute $m_{(v,u)}$ properly projected,

$$\phi(u,v) = \{f \Downarrow_S \mid f \in \psi(w), w \in T(u,v), S = var(f) - \chi(u)\}$$

Our CTEf and MCTEf implementations use this lower bound.

Another option for CTEf is to include in $\phi(u,v)$ a message $M_{(v,u)}$ from a previously computed execution of MCTE(r). When we apply the previous idea to MCTEf, we obtain a recursive algorithm which naturally produces an elegant iterative approximating method that we call iterative MCTEf (IMCTEf). The idea is to execute MCTEf(r) using as lower bounds $\phi(u,v)$ the messages $M^{r-1}_{(v,u)}$ computed by MCTEf($r-1$) which, recursively, uses the messages $M^{r-2}_{(v,u)}$ computed by MCTEf($r-2$), an so on. Algorithm 4 develops this idea. Starting from dummy lower bounds (line 1), we execute MCTEf(r) for increasing values of r (line 4). The lower bounds computed by MCTEf(r) will be used to detect and filter nogoods during the execution of MCTEf($r+1$) (line 5). The algorithm follows this process until the exact solution is computed (namely, MCTEf does not break messages into smaller functions), or the available resources are exhausted.

5 Experimental Results

Experiments are focused in two aspects:

1. Showing that CTEf versus state of the art CTE uses less tuples to find the exact solution.
2. Inside an approximation schema we show that MCTEf(r), exhausts resources at a smaller r and finds worst LB than the iterative version IMCTEf where the previous messages of MCTEf(r)^{UB} execution are used as filters.

```
procedure IMCTE($\langle X, D, C, k\rangle$, $\langle\langle V, E\rangle, \chi, \psi\rangle$)
1 for each $(u,v) \in E$ do $\phi(u,v) := \{\emptyset\}$;
2 $r := 1$;
3 repeat
4   MCTEf($r$);
5   for each $(u,v) \in E$ do $\phi(u,v) := M_{(u,v)}$;
6   $r := r + 1$;
7 until exact solution or exhausted resources
```

Fig. 4. The IMCTE algorithm. $\langle X, D, C, k\rangle$ is a WCSP instance and $\langle\langle V, E\rangle, \chi, \psi\rangle$ is its tree decomposition.

We have tested CTE, CTEf, MCTEf(r) and IMCTEf on DIMACS dubois Max-Sat instances, Borchers Weigthed Max-Sat instances and SPOT instances. Tree decompositions of tested instances where computed using the ToolBar library that implements a min fill heuristic for this purpose. This library is available at http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP.

The efficiency of inference algorithms strongly relies on achieving a good tree decomposition of the problem, ideally one with small maximum separator size, the bottleneck of CTE based algorithms. State of the art CTE always assumes that the memory spent by the algorithm is always equal to the worst case d^s for every sent message. Here we want to prove that assuming that functions only store consistent tuples with the joint effect of applying filtering techniques to anticipate inconsistent tuples, the memory stored in the solving process is actually much less than the worst case space complexity assumed by usual CTE.

When d^s is small usual CTE is feasible. For example in dubois100 we have $2^3 = 8$ and we can hardly see the improvement of CTEf. In instances where both CTE and CTEf are feasible (see the first Borchers and first SPOT instances) the latter solves the problem with one order of magnitude less tuples. As the separator size increases CTE becomes at some point infeasible. This happens in wp2200 where we have $d^s = 2^{19}$ and however CTEf is still feasible spending 733k tuples. In instances wp2250 and wp2300 an interesting thing happens; neither CTE nor CTEf can solve them, but the iterative version IMCTEf can solve it. In figure 5 the execution of the algorithm is plotted for instance wp2250 and we can see that there is a critical arity where a maximum of tuples is generated. Iterations corresponding to last r's generate less tuples and so, they are quicker to compute.

When the separator size increases and instances cannot be optimally solved by any algorithm (CTE, CTEf, MCTEf, IMCTEf) the latter approximates the problem with a higher lower bound in all cases and reaches a higher arity in some of them.

When sending a particular message an important fact is how we sum all the available functions for that message. The direct way is to sum them two by two if the arity limit permits, applying filtering at each sum with all possible

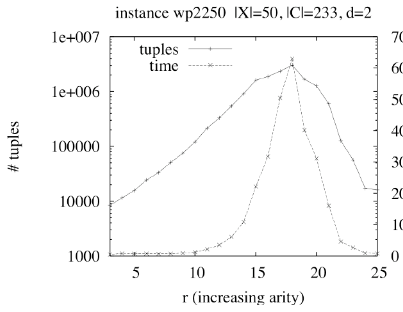
							MCTEf(r)		IMCTEf		
	$\|X\|$	$\|C\|$	d	sep	CTE	CTEf	r	LB	r	LB	UB
dubois100	75	200	2	3	3k	2k					1*
wp2100	50	95	2	9	6k	1k					16*
wp2150	50	138	2	15	302k	40k					34*
wp2200	50	186	2	19	-	733k					69*
wp2250	50	233	2	24	-	-	23	71	25	96	96*
wp2300	50	261	2	26	-	-	22	84	26	132	132*
wp2350	50	302	2	30	-	-	21	129	21	159	212
wp2400	50	340	2	30	-	-	20	70	20	137	212
wp2450	50	378	2	31	-	-	20	130	20	187	257
wp2500	50	418	2	34	-	-	20	168	20	251	318
spot54	67	271	4	11	754k	16k					37*
spot29	82	462	4	14	-	63k					8059*
spot503	143	635	4	8	-	34k					11113*
spot505	240	2242	4	22	-	-	12	8044	15	19217	21254
spot42	190	1394	4	26	-	-	13	116001	15	127050	155051

Table 1. Columns are: instance, number of variables, number of constraints, maximum domain size, maximum separator size, tuples consumed by CTE algorithm, tuples consumed by CTEf algorithm (- denotes exhausted memory), arity r reached by MCTE(r), LB computed by MCTE(r), arity r reached by IMCTEf, LB computed by IMCTE (before resources exhausted), optimal UB of the problem. When marked with (*) means that the instance is optimally by at least one of the algorithms.

available filters. We must be careful with summing first functions with low cost, because they can quickly exhaust memory since almost no tuple will reach the level to be detected as inconsistent. So at this point some heuristics have been tested to select the pairs of functions to be summed. The two giving the best results are the following ones: (i) minimize mean cost of function tuples and (ii) minimum arity of the generated function. When minimum arity coincides then we minimize cost.

6 Conclusions

We have presented the idea of function filtering for WCSP case, where constraints are cost functions, inside a complete inference schema. This idea has been nicely combined with tree decomposition algorithms, producing new algorithms which experimentally require far less memory than their original counterparts. This represent an important step forward the practical applicability of complete inference for WCSP solving.



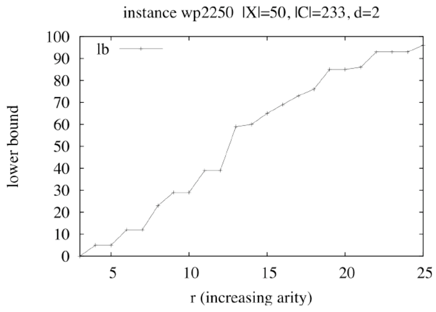


Fig. 5. IMCTEf execution in Borchers instance wp2250. On the left, y-axis is the total number of computed tuples and time respectively. On the right, y-axis is the lower bound achieved for each arity r.

So far, the use of upper and lower bounds for WCSP solving was limited to search methods, namely branch-and-bound search. This is the first time that bounds are used inside complete inference methods, to speed up their execution and to reduce their memory consumption. As results show, this combination has been quite beneficial. Combining other inference methods with bounds usage seems a promising line of research, which deserves further exploration in the future.

References

1. U. Bertele and F. Brioschi. *Nonserial Dynamic Programming.* Academic Press, 1972.
2. B. Cabon, S. Givry, and G. Verfaillie. Anytime lower bounds for constraint violation minimization problems. In *Proceedings of the 4th Conference on Principles and Practice of Constraint Programming*, volume 1520, pages 117–131, 1998.
3. S. de Givry, G. Verfaillie, and T. Schiex. Bounding the optimum of constraint optimization problems. In *Proceedings of the 3th Conference on Principles and Practice of Constraint Programming*, pages 405–419, Schloss Hagenberg, Austria.
4. R. Dechter, K. Kask, and J. Larrosa. A general scheme for multiple lower bound computation in constraint optimization. In *Proceedings of the 6th Conference on Principles and Practice of Constraint Programming*, pages 346–360, 2001.
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 constraints networks. *Artifical Intelligence*, 38, 1989.
7. Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artifical Intelligence*, 113:41–85, 1999.
8. Rina Dechter. *Constraint Processing.* Elsevier Science, 2003.
9. J. Larrosa. Node and arc consistency for weighted csp. In *Proc. AAAI*, 2002.
10. 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*, 23:421–440, 2005.

11. J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159, 2004.
12. M. Sanchez, P. Meseguer, and J. Larrosa. Improving the applicability of adaptive consistency. In *Proceedings of the 10th Conference on Principles and Practice of Constraint Programming*, Toronto, Canda, 2004.