Adaptive and Opportunistic Exploitation of Tree-decompositions for Weighted CSPs

Philippe Jégou Hélène Kanso Cyril Terrioux
Aix-Marseille Univ, Université de Toulon, CNRS, ENSAM, LSIS, Marseille, France
{philippe.jegou, hanan.kanso, cyril.terrioux}@lsis.org

***Abstract*—When solving weighted constraint satisfaction problems, methods based on tree-decompositions constitute an interesting approach depending on the nature of the considered instances. The exploited decompositions often aim to reduce the maximal size of the clusters, which is known as the width of the decomposition. Indeed, the interest of this parameter is related to its importance with respect to the theoretical complexity of these methods. However, its practical interest for the solving of instances remains limited if we consider its multiple drawbacks, notably due to the restrictions imposed on the freedom of the variable ordering heuristic.**

So, we first propose to exploit new decompositions for solving the constraint optimization problem. These decompositions aim to take into account criteria allowing to increase the solving efficiency. Secondly, we propose to use these decompositions in a more dynamic manner in the sense that the solving of a subproblem would be based on the decomposition, totally or locally, only when it seems to be useful. The performed experiments show the practical interest of these new decompositions and the benefit of their dynamic exploitation.

***Index Terms*—Weighted CSP; solving; decomposition**

I. Preliminaries

The Weighted Constraint Satisfaction Problem (WCSP) offers a general framework that permits to model and solve efficiently many real problems like, for example, the frequency allocation problem [1] or some problems related to bioinformatics [2]. In this paper, our main goal is to improve the efficiency of the structural solving methods, namely those based on tree-decompositions. First, we recall the definition of a WCSP instance:

Definition 1: A *WCSP instance* is defined as a triplet (X, D, W) with:

- $X = \{x_1, \ldots, x_n\}$ is a set of n *variables*,
- $D = \{D_{x_1}, \ldots, D_{x_n}\}$ is a set of finite *domains* of *values* where each domain D_{x_i} is related to the variable x_i,
- W is a set of e *cost functions*. A cost function that involves the set $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\}$ of variables is a function that associates to each tuple of $D_{x_{i_1}} \times D_{x_{i_2}} \times \ldots \times D_{x_{i_k}}$ a given integer between 0 and k with k the maximal cost corresponding to a completely forbidden tuple (expressing hard constraints).

For the sake of simplicity, our presentation is restricted to binary instances, that is the instances for which each cost function involves at most two variables. However, our work can be easily extended to non-binary instances. In particular, we use both binary and non-binary instances in the experimentations. In the following, w_{ij} denotes the cost function involving the variables x_i and x_j, w_i denotes the unary cost function corresponding to x_i and $w_\emptyset$ denotes the zero arity constraint which represents a constant cost paid by any assignment. The cost of a complete assignment $(v_1, \ldots, v_n) \in D_{x_1} \times \ldots \times D_{x_n}$ is defined by $w_\emptyset + \sum_{x_i \in X} w_i(v_i) + \sum_{w_{ij} \in W} w_{ij}(v_i, v_j)$. Given an instance, the weighted constraint satisfaction problem is to find a complete assignment with minimum cost. This problem is well-known to be NP-hard. Its solving is usually performed thanks to *branch and bound* algorithms. These algorithms traditionally exploit two bounds, denoted by clb and cub, which represent respectively a lower bound and an upper bound of the optimum. While the lower bound clb is less than the upper bound cub, they extend progressively the current assignment. When the assignment is fully extended, they update the current upper bound cub with the cost of the current complete assignment. This upper bound cub can be initialized by the value k or using an incomplete method. The efficiency of such approaches depends on the quality of the lower bound which is used at each node of the search tree. At this level, many efforts have been made recently like in particular the definition of various local consistency properties [3]–[6]. Most of the accomplished works around the WCSP problem rely on *branch and bound* algorithms based on a depth-first search. Recently, a hybrid approach, called HBFS, that combines a best-first search (BFS) and a depth-first search (DFS) was proposed [7]. HBFS is able to provide an anytime global lower bound on the optimum like BFS and an anytime upper bound like DFS. In practice, it seems that HBFS usually outperforms other existing methods.

Another alternative for solving WCSPs is to exploit the structure of the instances via the notion of tree-decompositions [8]. This point is discussed in Section II. Then, Section III deals with the computation of relevant tree-decompositions, while Section IV explains how to exploit dynamically a tree-decomposition for solving WCSP instances. Finally, in Section V, we assess the practical interest of our propositions before concluding in Section VI.

II. Solving WCSPs using tree-decompositions

The structure of a WCSP instance can be represented by a graph, called the *constraint graph*, where each vertex corresponds to a variable of the instance and an edge links two vertices if the two corresponding variables are involved in a binary cost function of W. To exploit this structure, some

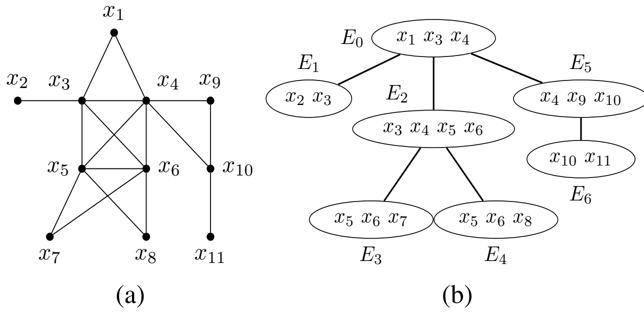


Fig. 1. A graph (a) and one possible optimal tree-decomposition (b).

methods [7], [9]–[11] use the notion of *tree-decomposition* [8] in order to identify independent subproblems.

Definition 2: A *tree-decomposition* of a graph $G = (X, C)$ is a pair (E, T) with $T = (I, F)$ a tree (I is the set of nodes and F the set of edges of T) and $E = \{E_i : i \in I\}$ a family of subsets of X, such that each subset (called cluster) E_i is a node of T and satisfies: (i) $\cup_{i \in I} E_i = X$, (ii) for each edge $\{x, y\} \in C$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$, and (iii) for all $i, j, \ell \in I$, if ℓ is on a path from i to j in T, then $E_i \cap E_j \subseteq E_\ell$. The width of a tree-decomposition (E, T) is equal to $max_{i \in I} |E_i| - 1$. The *tree-width* w of G is the minimal width over all the tree-decompositions of G.

Figure 1 represents a graph and a possible tree-decomposition of width 3 (since the largest cluster has 4 vertices).

BTD-like algorithms [7], [10], [11] exploit a tree-decomposition rooted in a given cluster (we denote E_r the root cluster of the decomposition) which permits to decompose the original problem into many subproblems (one subproblem per cluster). Given a WCSP instance (X, D, W), the subproblem related to the cluster E_i corresponds to the descent of E_i in the considered tree-decomposition (E, T). More precisely, its variable set $Desc(E_i)$ is the union of the clusters E_j belonging to the subtree of T rooted in E_i (E_i included) while its cost functions are ones whose scope is included in $Desc(E_i)$ but not in the separator of E_i with its parent cluster $E_{p(i)}$ (i.e. the intersection $E_i \cap E_{p(i)}$, under the assumption that $E_{p(r)} = \emptyset$). As these subproblems rooted in E_i are also related to the current assignment $\mathcal{A}_i$ on the separator $E_i \cap E_{p(i)}$, we denote them $P_i|\mathcal{A}_i$. For the root cluster, the subproblem $P_r|\emptyset$ rooted in E_r corresponds to the initial problem.

These algorithms solve the initial problem by assigning the variables in such a way that the variables of a cluster E_i are always assigned before ones of its child clusters. By so doing, they can benefit from the independence between the subproblems induced by the used tree-decomposition. Notably, they exploit a lower bound and an upper bound per subproblem $P_i|\mathcal{A}_i$ whose recording often leads to a more powerful pruning and many redundancy saving. These algorithms may rely on DFS (BTD-DFS [7], [11]) or HBFS (BTD-HBFS [7]). To this end, from a theoretical viewpoint, the advantage of using such methods can be justified by the time complexity which is generally in $O(n.e.d^{w^+ + 1})$ (with w^+ the width of the exploited tree-decomposition and d the size of the largest domain) whereas classical methods have a complexity in $O(n.e.d^n)$. These methods have a space complexity in $O(n.s.d^s)$ where s is the size of the largest intersection between two clusters in the considered tree-decomposition. In the same time, from a practical viewpoint, BTD-like algorithms maintaining local consistencies make it possible to obtain very efficient methods outperforming classical methods, notably when the instances have nice structural properties [7], [11]. Of course, this efficiency depends on the used decompositions. Minimizing w^+ can be interesting from a theoretical viewpoint. However, in practice, two elements are often considered as crucial:

- controlling the value of s since this value is directly related to the amount of memory used during the solving. Using a tree-decomposition with a large value of s may lead the method to run out of memory.
- exploiting a relevant variable ordering. It is well known that the best variable ordering heuristics (e.g. [12], [13]) require a certain freedom in the choice of the next variable, what may be in contradiction with the restrictions imposed on the variable ordering by the use of a tree-decomposition.

We address these two points in Sections III and IV.

III. Computing tree-decompositions

Computing an optimal tree-decomposition (i.e. a tree-decomposition having a minimum width) is well known to be a NP-hard problem [14]. Hence, computing tree-decompositions is usually achieved by heuristic methods. In this context, the heuristic Min-Fill [15] is considered as the reference method by the CP community. Thanks to its use, tree-decompositions have been already exploited successfully for solving WCSP instances [7]. Nevertheless, they have not shown their full potential because of the quality of the computed decompositions. In fact, Min-Fill aims to minimize the width of the computed tree-decompositions. However, nothing guarantees that the obtained tree-decomposition is suitable for an efficient solving of WCSP instances. Notably, Min-Fill tends to produce decompositions whose clusters have few proper variables (i.e. variables belonging to a cluster but not to its parent) and whose largest separators have frequently a size close to w^+. As evoked above, such a large value of s may be problematic due to the amount of memory required for storing the local bounds for each subproblem $P_i|\mathcal{A}_i$. Hence, in [7], Allouche et al. address this problem by considering a variant of Min-Fill denoted Min-Fill^{r_4}. Min-Fill^{r_4} computes a tree-decomposition as Min-Fill does and then each cluster sharing more than 4 variables with its parent is merged with it. By so doing, BTD-like algorithm may exploit larger clusters (which may increase the freedom of the variable heuristic) while consuming a reasonable amount of memory.

Here, we address this question in a different way. More precisely, we consider a general framework, called *H-TD-WT* (for *Heuristic Tree-Decomposition Without Triangulation*), which has been recently introduced in [16]. Unlike Min-Fill, this framework aims to produce decompositions without

computing a triangulation of the constraint graph. However, it exploits some topological features (via the connected components) and the construction of each cluster is guided by a heuristic depending on the criteria we want to fulfill. Its time complexity is reasonable, generally in $O(n(n+e))$ whereas, in practice, it is often faster than Min-Fill (or Min-Fill^{r_4}). Different parameterizations are conceivable in order to produce tree-decompositions suitable for an efficient solving. For example, these last years, three possible parameterizations of *H-TD-WT* have been proposed in the frame of the decision problem CSP:

- H_2 [17] which guarantees that the clusters of the decomposition are connected,
- H_3 [16] which identifies independent parts of the graph and separates them as soon as possible by doing a breadth-first search.
- H_5 [18] which aims to compute tree-decompositions having separators of bounded size, with S as an upper bound on the size of the separators.

These heuristics have shown their practical interest for solving CSP instances [16], [18], [19], but have never been considered for the treatment of WCSP instances. Beyond, once the decomposition computed, it can be exploited statically like in [7], [10], [11] or dynamically as described in the next section.

IV. Dynamic exploitation of the decomposition

BTD-like methods have shown their practical interest when solving WCSP instances having nice topological features [7], [11]. Nonetheless, they may turn to be inefficient if the instance has no particular topological feature. One possible reason is related to the variable orderings they consider. These orderings have a freedom restricted by the use of the tree-decomposition since all the variables of a given cluster must be assigned before ones of their child clusters. In particular, these restrictions may prevent from taking benefits from *adaptive* techniques [12], [13] which, nowadays, contribute significantly to the practical efficiency of classical Branch and Bound methods. In this section, our goal is to exploit cleverly the tree-decomposition by avoiding using it systematically. More precisely, in the same spirit as adaptive techniques, whose choices rely both on the current state of the search and its previous ones, we add to BTD-like algorithms the ability to choose to exploit globally, partially or not the tree-decomposition depending on whether its exploitation seems to be beneficial or not. By so doing, we increase the freedom of the variable ordering while adapting the solving depending on the context and the nature of the instance.

In practice, the decomposition initially computed will never be exploited for a subproblem unless the solving without a decomposition seems to be inefficient. We consider, for example, the decomposition shown in Figure 2(a) represented by the set of clusters $E=\{E_i,E_j,E_k,E_l,E_m,E_n\}$. Let E_i be the current cluster and $\mathcal{A}$ the current assignment on $E_j \cap E_i$. We are interested by the subproblem $P_j|\mathcal{A}$ rooted in E_j and induced by the assignment $\mathcal{A}$ of the separator $E_j \cap E_i$ with

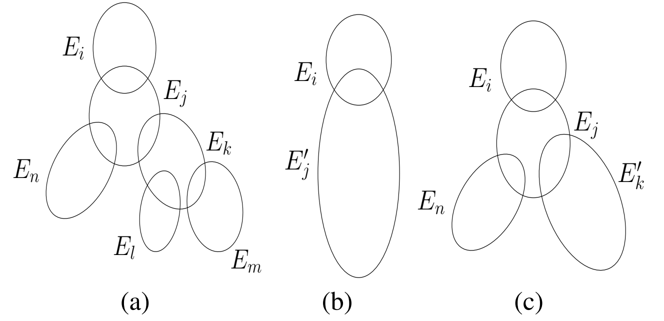


Fig. 2. The set of the clusters of the decomposition: initial (a) when E_j is merged with its descent (cluster E'_j) (b) when E_j is exploited (c).

E_i the parent cluster of E_j. The solving of the subproblem $P_j|\mathcal{A}$ does not necessarily use the decomposition, that is to say the set of the clusters $\{E_j,E_k,E_l,E_m,E_n\}$. In fact, in the beginning, the solving is based on the cluster E'_j resulting from the merging of the cluster E_j with its descendants E_k, E_l, E_m and E_n. Merging a cluster with its descendants consists in putting together all the variables of the descent of E_j (see Figure 2(b)). Hence, E'_j contains all the variables of the descent of E_j. Concerning the variable ordering heuristic, it benefits from a total freedom with respect to the choice of the next variable among the variables of the descent of E_j. At this level, there are two possible cases:

- The subproblem $P_j|\mathcal{A}$ can be easily solved when using E'_j. In this case, the exploitation of the decomposition does not seem useful.
- The solving of $P_j|\mathcal{A}$ is on the contrary inefficient. In this case, the exploitation of the decomposition seems wise. The cluster E_j is then re-used and the same reasoning is repeated for the cluster E_k which will be first considered as merged with its descent resulting in the cluster E'_k as shown in Figure 2(c).

Note that the reasoning is repeated for every new assignment of $E_i \cap E_j$. This choice is motivated by the fact that the assignment $\mathcal{A}$ of the separator $E_i \cap E_j$ induces a different subproblem from the one induced by an assignment $\mathcal{A}'$. Note also that, at the level of the root cluster E_r, the algorithm behaves like a classical Branch and Bound algorithm having total freedom with regard to the choice of the next variable among all the variables of the problem. Finally, the initial decomposition is fully exploited (all its clusters are exploited) when at all the levels, our algorithm decides to exploit the cluster itself rather than its descent.

For sake of simplicity, we describe now the new algorithm by exploiting DFS for solving clusters, resulting in the algorithm BTD-DFS+DYN (see Algorithm 1). Of course, we can also derive BTD-HBFS+DYN (which will be used for the experiments) in the same manner as Allouche et al. produce BTD-HBFS from BTD-DFS [7].

Like BTD-DFS, BTD-DFS+DYN is based on a tree-decomposition (E,T) which is computed before the solving and rooted in a cluster E_r. It takes as parameters the current assignment $\mathcal{A}$, the current cluster E_i, the set V of unassigned

Algorithm 1: BTD-DFS+DYN $(\mathcal{A}, E_i, V, V_{desc}, clb, cub)$

Input: The current assignment $\mathcal{A}$, the current cluster E_i, the lower bound clb

Input/output: The set V of unassigned variables of E_i, the set V_{desc} of unassigned variables of $Desc(E_i)$, the current upper bound cub

```
1  if Merge(E_i) then V' ← V_desc
2  else V' ← V
3  if V' ≠ ∅ then
4      x ← pop(V') /* Choose a variable of V' */
5      Update V and V_desc
6      a ← pop(D_x) /* Choose a value */
7      Maintain local consistency on subproblem P_i|A ∪ {(x = a)}
8      clb' ← max(clb, lb(P_i|A ∪ {(x = a)}))
9      if clb' < cub then cub ← BTD-DFS+DYN(A ∪ {(x = a)}, E_i,
           V, V_desc, clb', cub)
10     if max(clb, lb(P_i|A)) < cub then
11         Maintain local consistency on subproblem P_i|A ∪ {(x ≠ a)}
12         clb' ← max(clb, lb(P_i|A ∪ {(x ≠ a)}))
13         if clb' < cub then  cub ← BTD-DFS+DYN
              (A ∪ {(x ≠ a)}, E_i, V, V_desc, clb', cub)
14 else if ¬Merge(E_i) then
15     /* Solve all child clusters with unknown optimum */
16     S ← Children(E_i)
17     while S ≠ ∅ and lb(P_i|A) < cub do
18         E_j ← pop(S) /* Choose a child cluster */
19         if LB_{P_j|A} < UB_{P_j|A} then
20             cub' ← min (UB_{P_j|A}, cub − [lb(P_i|A) −lb(P_j|A)])
21             cub'' ← BTD-DFS+DYN (A, E_j, E_j\(E_i ∩ E_j),
                  Desc(E_j)\(E_i ∩ E_j), lb(P_j|A), cub')
22             Update LB_{P_f|A} and UB_{P_f|A} using cub''
23     cub ← min(cub, w^i_∅ + Σ_{E_j ∈ Children(E_i)} UB_{P_j|A})
24 else  cub ← min(cub, Σ_{E_j ⊆ Desc(E_i)} w^j_∅)
25 return cub
```

variables of E_i, the set V_{desc} of unassigned variables of the descent $Desc(E_i)$ of E_i, the current lower bound clb and the current upper bound cub. It aims to compute the optimum of the problem rooted in E_i and induced by the assignment $\mathcal{A}$. There are two possible cases:

- If the returned value of cub is strictly less than the input value of cub, then cub is the optimum of the subproblem.
- Otherwise, cub defines a lower bound on the optimum.

The initial call is BTD-DFS+DYN($\emptyset$, E_r, V_r, $V_{r_{desc}}$, $lb(P_r|\emptyset)$, k) with $V_{r_{desc}}$ the set of variables of the descent of E_r, $lb(P_r|\emptyset)$ the initial lower bound obtained by exploiting a local consistency on the initial problem and k the maximal cost. Note that throughout the algorithm, $w^i_\emptyset$ denotes the cluster-localized lower bound for the cluster E_i. Like BTD-DFS, BTD-DFS+DYN assumes that the input problem is consistent according to the exploited local consistency and returns its optimum. Lines 3-13 of BTD-DFS+DYN aim to assign the variables of V' as BTD-DFS does for the variables of V. A pair (variable, value), (x, a), is selected according to the variable and value ordering heuristics. Given that binary branching is exploited, this choice results in either assigning the variable x to a (left branch, positive decision, line 7), or removing the value a from the domain of x (right branch, negative decision, line 11). For each branch, local consistency is maintained on the subproblem rooted in E_i and induced by the current assignment so that a lower bound lb is computed. If the maximum clb' of the lower bound lb, deduced via the application of the local consistency, and the current lower bound clb, is strictly less than cub, the search continues; otherwise, the corresponding subtree is pruned. Lines 15-23 of BTD-DFS+DYN solve the subproblems rooted in each child cluster E_j of E_i, similarly to BTD-DFS. The subproblem $P_j|\mathcal{A}$ is explored unless its optimum is already known. Anyway, BTD-DFS and BTD-DFS+DYN record, for each subproblem $P_j|\mathcal{A}$, two values denoted by $LB_{P_j|\mathcal{A}}$ and $UB_{P_j|\mathcal{A}}$. They represent respectively the best known lower and upper bounds for $P_j|\mathcal{A}$. If $LB_{P_j|\mathcal{A}} = UB_{P_j|\mathcal{A}}$, the optimum of P_j is already computed. The recursive call corresponding to the child cluster E_j (line 21) exploits an initial non-trivial upper bound computed at line 20, as explained in [11]. It is followed by an update of both $LB_{P_j|\mathcal{A}}$ and $UB_{P_j|\mathcal{A}}$ with the value of cub'' (line 22). The exploitation of all the child clusters ends finally by updating cub.

Now that the similarities between BTD-DFS and BTD-DFS+DYN are recalled, we describe the modifications made to BTD-DFS. The function $Merge$ represents the heuristic responsible for choosing whether the cluster E_i is exploited or its descent E'_i. The call to $Merge$ with the cluster E_i as an input returns true if and only if E'_i is exploited. Otherwise, the cluster E_i is exploited and the freedom of the next variable choice is limited to the variables of V. One of the two parameters V or V_{desc} is effectively used depending on whether E_i or E'_i is exploited. The choice between V and V_{desc} is done on lines 1-2 and is memorized in V'. Both are suitably updated on line 5. If $V' = \emptyset$ and E_i is not merged with its descent, BTD-DFS+DYN behaves like BTD-DFS (lines 3-13). If $V' = \emptyset$ and E_i is merged with its descent, BTD-DFS+DYN has no more variables to be assigned given that all the variables of the descent of E_i have been already assigned. In this case, BTD-DFS+DYN updates cub (line 24) at the basis of the local lower bound $w^j_\emptyset$ of each cluster E_j belonging to $Desc(E_i)$. If $V' \neq \emptyset$ (lines 3-13), BTD-DFS+DYN behaves the same way whether E_i or its descent E'_i is exploited, by trying to assign the variables of V'.

The algorithm BTD-DFS+DYN can be parameterized by the heuristic $Merge$ which decides when to stop exploiting the cluster E'_i and to exploit instead the current cluster E_i (we propose one heuristic in the next section). Certainly, the choice of a relevant heuristic is essential to improve the solving.

The dynamic exploitation of the decomposition changes both the time and the space complexity. In fact, the behavior of BTD-DFS+DYN can constantly evolve. It first acts like almost a classical DFS and then, if it decides at each level to exploit the original cluster instead of its descent, it finally acts like a standard BTD-DFS.

Theorem 1: The time complexity of BTD-DFS+DYN ranges from $O(exp(w^+ + 1))$ to $O(exp(n))$ with w^+ the width of the tree-decomposition computed before the solving. Its space complexity is in $O(n.s'.d^{s'})$ where s' is the size of the largest

exploited separator of the decomposition ($s' \leq s$).

V. Experiments

In this section, we assess the practical interest of the framework *H-TD-WT* for solving WCSP instances and we evaluate the relevancy of exploiting dynamic decompositions. But first, we describe the used experimental protocol.

A. Experimental protocol

We consider the solving algorithms HBFS and BTD-HBFS(+DYN) with tree-decompositions produced by Min-Fill and Min-Fill^{r_4}. Note that, until now, HBFS and BTD-HBFS with Min-Fill^{r_4} are considered as the reference algorithms for solving WCSP instances respectively without and with the exploitation of the structure [7], while Min-Fill is often used as the reference method for computing tree-decompositions by the CP community. Concerning H-TD-WT, we exploit the heuristic H_2, which guarantees the connectivity of the clusters, H_3, which computes decompositions having many child clusters and H_5 which controls the size of the separators. The heuristic H_5 is available in two variants, denoted by H_5^{25} and $H_5^{5\%}$. For H_5^{25}, the size of the separators is bounded by 25. Regarding $H_5^{5\%}$, the limit of the maximum size of separators depends on the instance and is fixed to 5% of the number of variables of the instance, normalized to a minimum upper bound of 4 ($S = 4$) and a maximal upper bound of 50 ($S = 50$).

Like [7], we exploit the implementations of HBFS, BTD-HBFS, Min-Fill and Min-Fill^{r_4} provided in the open-source WCSP solver *Toulbar2*[1]. In order to guarantee a fair comparison, we have also implemented BTD-HBFS+DYN in *Toulbar2*. The H_i decompositions are computed by our own library and transmitted to *Toulbar2* by means of a file.

The dynamic exploitation of the decomposition is based on a heuristic ($\mathcal{F}$) that is responsible for deciding to exploit the cluster E_i or its descent E'_i. The heuristic $\mathcal{F}$ uses the feedback given by BTD-HBFS. In fact, each call to BTD-HBFS on a subproblem $P_i|\mathcal{A}$ takes as input both the known lower and upper bounds, clb and cub. If within the allowed number of backtracks, BTD-HBFS does not succeed in improving any bounds, $\mathcal{F}$ considers that the solving of $P_i|\mathcal{A}$ is not progressing and increments a counter related to it. We start henceforth exploiting E_i instead of E'_i, when the counter corresponding to $P_i|\mathcal{A}$ attains a given limit. In the reported results, we have exploited the value 5 for this limit. This value seems to lead to the best trade-off between BTD-HBFS and HBFS among the extensive experiments we performed.

Regarding the configuration of *Toulbar2*, a preprocessing step is performed by enforcing VAC (for *Virtual Arc-Consistency* [20]) and applying the MSD (for *Min Sum Diffusion*) algorithm with 1,000 iterations. Then, EDAC (for Existential Directional Arc-Consistency [21] is enforced, at each node, during the solving. The variable ordering includes both weighted-degree [12] and last conflict [22]. Regarding the algorithms based on BTD, the chosen root cluster is the largest cluster (for Min-Fill) or the cluster which maximizes the ratio of the number of constraints of the cluster to its size (for the other decompositions). For the solving algorithms based on HBFS, the other parameters (e.g. α, β or N) are set with the same values as in [7]. The experiments were performed on blade servers running Linux Ubuntu 14.04 each with two Intel Xeon processors E5-2609 v2 2.4 GHz and 32 GB of memory. We use the benchmark instances available at http://genoweb.toulouse.inra.fr/~degivry/evalgm. It includes more than 3,000 instances containing stochastic graphical models from the *UAI evaluation* 2008 and 2010, instances from the MiniZinc Challenge 2012 and 2013 and instances from the *Probabilistic Inference Challenge* 2011. Compared to [7], we have discarded instances solved during the preprocessing, resulting in a benchmark of 2,444 instances. For each instance, the solving is performed with a timeout of 20 minutes (including when applicable, the computation of the decomposition) and 16 GB of memory.

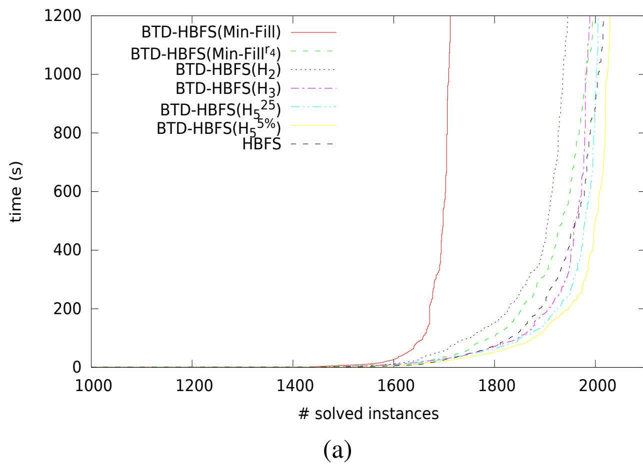
Before giving in details the experimental results, it seems necessary to give additional comments on the comparisons between the respective runtimes of the different approaches. Indeed, one method could be considered as more efficient if its runtime is better. However, we must also take into account the number of solved instances. For example, when we report in Figures 3 (a) and (b) that BTD-HBFS using Min-Fill solves 1,712 instances in 26,291 s, while BTD-HBFS+DYN using H_5^{25} solves 2,039 instances in 52,304 s, we can be interested by the bench solved by at least one of both methods which involves 2,049 instances. We can see that BTD-HBFS using Min-Fill have solved only 1,712 instances among the 2,049 instances while consuming 430,691 seconds. Here, we add the cost of the 337 unsolved instances in 20 minutes, that is 404,400 seconds which must be added to the 26,291 s used to solve the 1,712 instances. Concerning BTD-HBFS+DYN using H_5^{25}, after adding the cost of the 10 instances unsolved among the 2,049 instances, we obtain a total of 64,304 s. By this way, we see that finally, BTD-HBFS+DYN (using H_5^{25}) runs 6 times faster than BTD-HBFS (using Min-Fill) while it solves 327 additional instances.

B. H-TD-WT vs. Min-Fill / Min-Fill^{r_4}

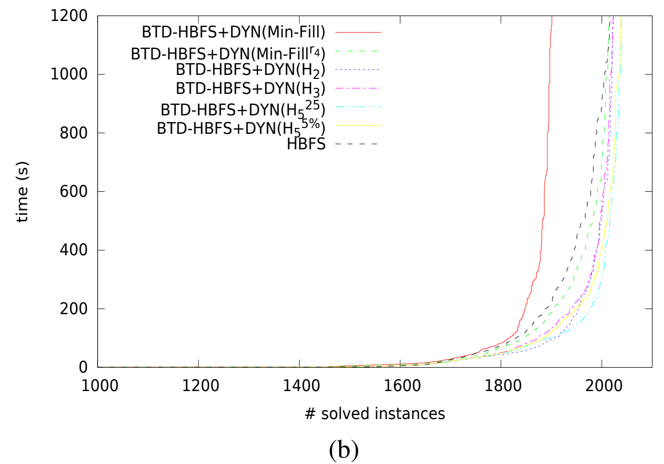
In this part, we compare the behavior of BTD-HBFS depending on the different exploited decompositions. Regarding the computation of decompositions, we notice that the computation of H_i decompositions (with $i \in \{2, 3, 5\}$) is much faster than Min-Fill. More precisely, any H_i is able to decompose 2,431 instances in less than 1,200 s whereas Min-Fill requires 19,044 s to decompose 2,415 instances.

Figure 3(a) shows the cumulative number of solved instances w.r.t. the runtime for HBFS and BTD-HBFS with Min-Fill and the decompositions computed thanks to H-TD-WT. First, we note that Min-Fill solves the weakest number of instances within 20 minutes (only 1,712 instances). It shows that, despite the difficulty of the WCSP problem, the heuristics that aim only to minimize the size of clusters are not the

[1]Available at https://mulcyber.toulouse.inra.fr/projects/toulbar2/



(a)



(b)

Fig. 3. Cumulative number of solved instances w.r.t. the runtime in seconds for HBFS and BTD-HBFS (a) or BTD-HBFS+DYN (b) depending on the decompositions.

more efficient. In fact, in order to minimize the width of the produced tree-decomposition, Min-Fill tends to produce clusters which have few proper variables. So the variable ordering heuristic is almost useless and the exploited variable ordering is nearly static. The results also show that the others parameters have more impact on the solving than the width of the decomposition, like the connectivity of the clusters (H_2), the number of child clusters (H_3) or the size of the separators (Min-Fill^{r_4} and H_5). Limiting the size of the separators seems to increase significantly the efficiency of the solving. In fact, if BTD-HBFS with H_2 and H_3 solves respectively 1,945 and 1,989 instances, the decompositions having bounded separators size lead to solve more than 2,000 instances. However, comparing Min-Fill^{r_4} to H_5 shows that BTD-HBFS associated to H_5 permits to solve more instances especially with $H_5^{5\%}$, which allows to solve 2,029 instances against 2,000 instances solved by BTD-HBFS with Min-Fill^{r_4}. Furthermore, BTD-HBFS exploiting Min-Fill^{r_4} requires 98,861 s to solve these instances whereas BTD-HBFS exploiting $H_5^{5\%}$ requires only 58,792 s. Finally, note that these results are consistent with those obtained for the decision problem CSP in [16].

C. Dynamic decomposition assessment

We evaluate now BTD-HBFS+DYN. Both Figures 3 (a) and (b) show that, regardless of the exploited decomposition, BTD-HBFS+DYN solves more instances than BTD-HBFS. The increase of the number of solved instances can be remarkable like in the case of Min-Fill that permits to solve now 1,905 instances instead of 1,712 instances. The exploitation of H_2 and H_3 with BTD-HBFS+DYN allows to solve 2,023 instances against respectively 1,945 and 1 989 solved by BTD-HBFS. The exploitation of Min-Fill^{r_4} and H_5 with BTD-HBFS+DYN is also improved with regard to their utilization with BTD-HBFS. In particular, H_5^{25} allows to solve 2,039 instances, which constitutes the largest number of solved instances among all the possible combinations of a solving algorithm and a decomposition we consider. Beyond

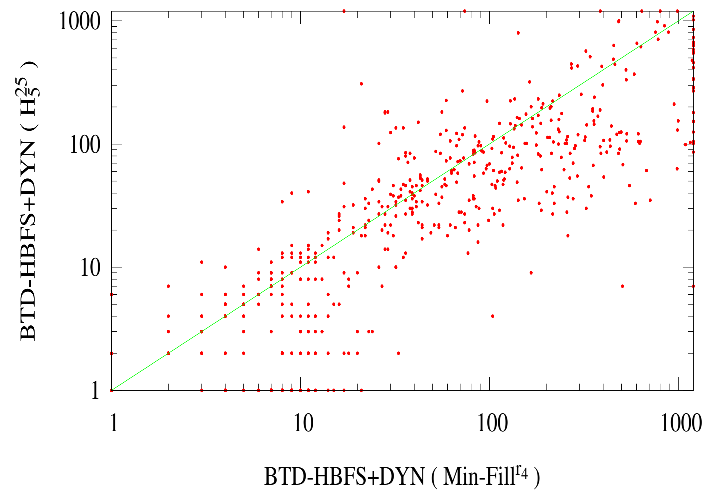


Fig. 4. Runtime comparison of BTD-HBFS+DYN with H_5^{25} and Min-Fill^{r_4}.

the increase in the number of solved instances, the cumulative runtime decreases. For instance, BTD-HBFS+DYN requires 74,159 s for solving 2,019 instances with Min-Fill^{r_4} (resp. 52,304 s for 2,039 instances with H_5^{25}), while BTD-HBFS solves 2,000 instances in 98,861 s (resp. 2,006 instances in 56,553 s).

We now focus on comparing H_5^{25} and Min-Fill^{r_4} with BTD-HBFS+DYN. Figures 3(b) and 4 show clearly that BTD-HBFS+DYN is globally more efficient with H_5^{25} than with Min-Fill^{r_4}. This trend still holds when we consider the benchmark of instances solved by both BTD-HBFS+DYN with Min-Fill^{r4} and BTD-HBFS+DYN with H_5^{25}, in order to fairly compare their respective runtime. Indeed, this benchmark includes 2,013 instances solved by BTD-HBFS+DYN with Min-Fill^{r4} in 71,249 s against only 41,372 s by BTD-HBFS+DYN with H_5^{25}.

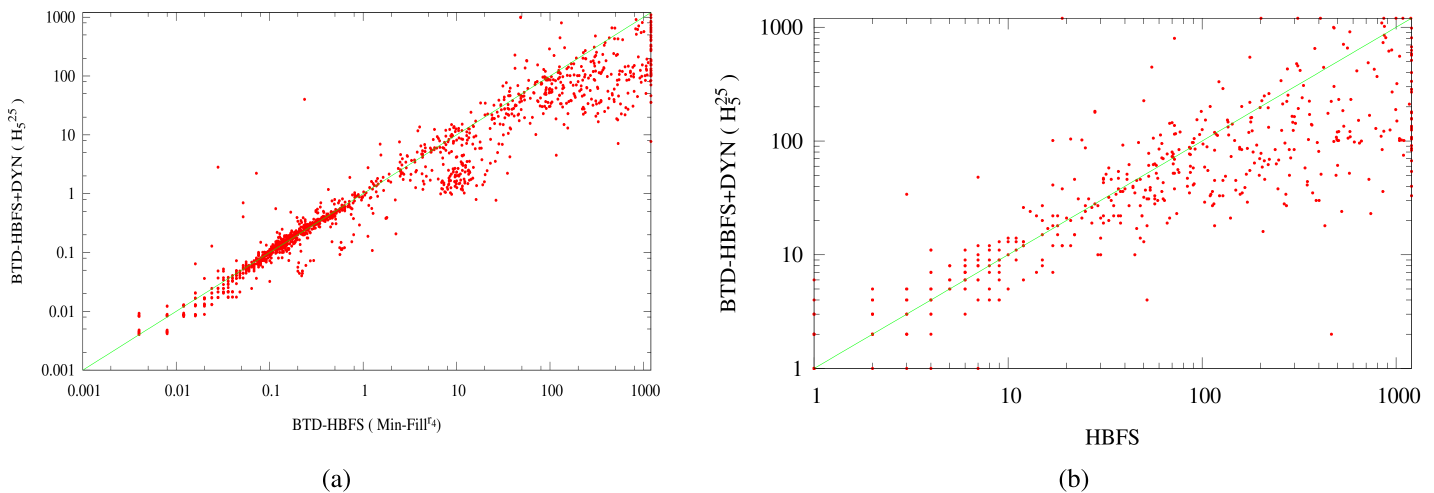


Fig. 5. (a) Runtime comparison between BTD-HBFS+DYN with H_5^{25} and BTD-HBFS with Min-Fill^{r_4} (a) and HBFS (b).

D. BTD-HBFS(+DYN) vs. the best known methods

At now, BTD-HBFS with Min-Fill^{r_4} and HBFS are considered as the best known methods for solving WCSPs respectively with and without exploiting the structure. So, it is quite natural to compare them to BTD-HBFS(+DYN) with H_5. Figures 3(a), 3(b) and 5(a) clearly show that BTD-HBFS and BTD-HBFS+DYN with $H_5^{5\%}$ or H_5^{25} outperform significantly BTD-HBFS with Min-Fill^{r_4}, w.r.t. both the number of solved instances and the cumulative runtime. For instance, BTD-HBFS+DYN with H_5^{25} solves 2,039 instances in 52,304 s while BTD-HBFS with Min-Fill^{r_4} requires 98,861 s to solve 2,000 instances.

We now compare BTD-HBFS(+DYN) with H_5 to HBFS. First we can note that BTD-HBFS with $H_5^{5\%}$ is more efficient than HBFS. Notably, it solves more instances than HBFS, namely 2,029 instances against 2,017 instances by HBFS and, in addition, it has a better cumulative runtime with 58,792 s against 84,555 s for HBFS. This can be essentially explained by the recordings achieved by BTD-HBFS via the separators, that is to say a lower bound and an upper bound on the optimum of the corresponding subproblem. Furthermore, BTD-HBFS distinguishes independent subproblems, which is not the case of HBFS.

Then a similar trend can be observed if we compare BTD-HBFS+DYN associated to H_5^{25} to HBFS (see Figure 5(b)). Indeed, BTD-HBFS+DYN with H_5^{25} turns to solve more instances and faster than HBFS. Moreover, if we consider the 2,008 instances solved by both algorithms, HBFS solves them in 79,020 s against only 43,914 s for BTD-HBFS+DYN with H_5^{25}.

We now focus our comparison to the hardest instances. More precisely, we discard easy instances, which are even trivial sometimes (that is to say instances being solved in less than 10 s by HBFS). Note that these instances are also easily solved by BTD-HBFS+DYN provided that at the level of the root cluster BTD-HBFS+DYN behaves as HBFS (when the descent of the root cluster is exploited). Among the 794 remaining instances, at least one leaf cluster of the decomposition is exploited for 279 instances. In other words, there exists at least one branch of the decomposition that is entirely exploited (in the sense of the set of the clusters of the original decomposition). For 423 instances, the decomposition is never exploited which means that BTD-HBFS+DYN behaves as HBFS. Regarding the remaining instances, the decomposition is exploited till a particular depth without reaching a leaf cluster of the decomposition. Hence, in practice, BTD-HBFS+DYN can simply behave like HBFS or progressively decide to exploit the original clusters of the decomposition (instead of their descent) until reaching the behavior of BTD-HBFS.

Finally, we compare the lower and the upper bounds reported by HBFS and BTD-HBFS+DYN when a timeout occurs. Among the 375 instances not solved neither by HBFS nor by BTD-HBFS+DYN, for 252 instances, the upper bound computed by BTD-HBFS+DYN is strictly less than the one computed by HBFS against only 52 instances for HBFS. Also, for 229 instances, BTD-HBFS+DYN reports a higher lower bound than HBFS against 146 instances for HBFS. Figure 6 compares the gap between the lower and the upper bounds for both algorithms. Clearly, BTD-HBFS+DYN offers a reduced gap compared to HBFS. This shows that even when the instance is not solved, BTD-HBFS+DYN is able to give approximations of better quality than HBFS.

E. Summary

All of these results show that exploiting a decomposition dynamically seems to enhance the efficiency of the solving by enabling BTD-HBFS+DYN to adapt the solving to the nature of the instance and by avoiding using a decomposition when solving a problem (or a subproblem) without a decomposition is more efficient. It results than BTD-HBFS+DYN with H_5^{25} is able to outperform the best known methods for solving WCSPs like BTD-HBFS with Min-Fill^{r_4} or HBFS w.r.t. the number of solved instances and the cumulative runtime.

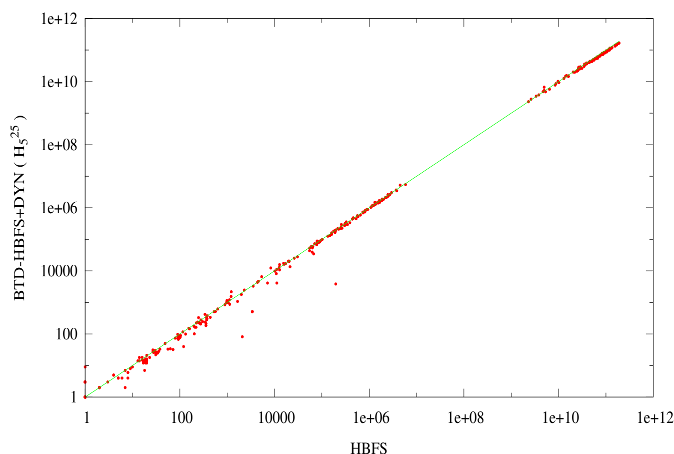


Fig. 6. Gap between the lower and the upper bounds for both BTD-HBFS+DYN with H_5^{25} and HBFS.

VI. Conclusion

The tree-decompositions have been already exploited with success to solve WCSP instances. At the same time, their potential was not fully revealed especially due to the quality of the used decompositions. In fact, like Min-Fill, the computed decompositions usually aim to minimize the size of the clusters (in order to minimize the theoretical time complexity bound) to the detriment of the practical efficiency of the solving. So, in order to take into account other relevant parameters (i.e. having additional impact on the solving efficiency w.r.t. the width), we have first exploited the framework H-TD-WT, which has been previously proposed to solve CSP instances. This framework aims to compute decompositions much faster than by classical heuristics. It also permits to take into consideration relevant properties with respect to the solving efficiency such as the maximal size of separators with the H_5 heuristic. Utilizing H_5 with BTD-HBFS has clearly proven the interest of H-TD-WT. Indeed, H_5 does not only improve the performance of BTD-HBFS compared to BTD-HBFS exploiting Min-Fill, but has permitted to outperform significantly HBFS. On the other hand, we have proposed an extension of BTD-HBFS, called BTD-HBFS+DYN, which exploits the decomposition dynamically. The idea consists in using the decomposition only for a subproblem whose solving without decomposition seems to be inefficient. It notably avoids to restrict unnecessarily the freedom of the variable ordering heuristic for "easy" subproblems. In practice, we have shown that BTD-HBFS+DYN improves BTD-HBFS, regardless of the exploited decomposition, with regard to both the number of solved instances and the cumulative runtime. Consequently, thanks to this extension, structural methods are henceforth highlighted versus non-structural methods.

Several extensions of this work seem relevant. For example, the computation of the decomposition made before the solving can also be achieved dynamically. In fact, the decomposition is often partially used, not to mention the case where the decomposition is not exploited at all by BTD-HBFS+DYN. Moreover, it would also make it possible to calculate better and more adapted decompositions for the solving of WCSP instances than those calculated solely on the basis of structural criteria. Beyond that, other heuristics for the dynamic exploitation of decompositions may be assessed.

Acknowledgment

This work has been funded by the french Agence nationale de la Recherche, reference ANR-16-C40-0028.

References

[1] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners, "Radio Link Frequency Assignment," *Constraints*, vol. 4, pp. 79–89, 1999.
[2] M. Sanchez, S. de Givry, and T. Schiex., "Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques," *Constraints*, vol. 13, no. 1-2, pp. 130–154, 2008.
[3] M. Cooper and T. Schiex, "Arc consistency for soft constraints," *Artificial Intelligence*, vol. 154(1-2), pp. 199–227, 2004.
[4] S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki, "Existential arc consistency: closer to full arc consistency in weighted CSPs," in *IJCAI*, 2005.
[5] M. C. Cooper, S. de Givry, and T. Schiex, "Optimal Soft Arc Consistency," in *IJCAI*, 2007, pp. 68–73.
[6] M. Cooper, S. Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner, "Soft arc consistency revisited," *Artificial Intelligence*, vol. 174, pp. 449–478, 2010.
[7] D. Allouche, S. de Givry, G. Katsirelos, T. Schiex, and M. Zytnicki, "Anytime hybrid best-first search with tree decomposition for weighted CSP," in *CP*, 2015, pp. 12–29.
[8] N. Robertson and P. Seymour, "Graph minors II: Algorithmic aspects of treewidth," *Algorithms*, vol. 7, pp. 309–322, 1986.
[9] A. Koster, "Frequency Assignment - Models and Algorithms," Ph.D. dissertation, University of Maastricht, Novembre 1999.
[10] C. Terrioux and P. Jégou, "Bounded backtracking for the valued constraint satisfaction problems," in *CP*, 2003, pp. 709–723.
[11] S. de Givry, T. Schiex, and G. Verfaillie, "Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP," in *AAAI*, 2006, pp. 22–27.
[12] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *ECAI*, 2004, pp. 146–150.
[13] P. Refalo, "Impact-based search strategies for constraint programming," in *CP*, 2004, pp. 557–571.
[14] S. Arnborg, D. Corneil, and A. Proskuroswki, "Complexity of finding embeddings in a k-tree," *SIAM Journal of Disc. Math.*, vol. 8, pp. 277–284, 1987.
[15] D. J. Rose, "Triangulated Graphs and the Elimination Process," *Journal of Mathematical Analysis and Application*, vol. 32, pp. 597–609, 1970.
[16] P. Jégou, H. Kanso, and C. Terrioux, "An Algorithmic Framework for Decomposing Constraint Networks," in *ICTAI*, 2015, pp. 1–8.
[17] P. Jégou and C. Terrioux, "Combining Restarts, Nogoods and Decompositions for Solving CSPs," in *ECAI*, 2014, pp. 465–470.
[18] P. Jégou, H. Kanso, and C. Terrioux, "Towards a Dynamic Decomposition of CSPs with Separators of Bounded Size." in *CP*, 2016, pp. 298–315.
[19] P. Jégou and C. Terrioux, "Tree-decompositions with connected clusters for solving constraint networks," in *CP*, 2014, pp. 407–423.
[20] M. C. Cooper, S. de Givry, M. Sánchez, T. Schiex, and M. Zytnicki, "Virtual arc consistency for weighted csp," in *AAAI*, 2008, pp. 253–258.
[21] S. de Givry, F. Heras, M. Zytnicki, and J. Larrosa, "Existential arc consistency: Getting closer to full arc consistency in weighted csps," in *IJCAI*, 2005, pp. 84–89.
[22] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal, "Reasoning from last conflict (s) in constraint programming," *Artificial Intelligence*, vol. 173, no. 18, pp. 1592–1614, 2009.