A General Scheme for Multiple Lower Bound Computation in Constraint Optimization*

Rina Dechter[1], Kalev Kask[1], and Javier Larrosa[2]

[1] University of California at Irvine (UCI)
{dechter, kkask}@ics.uci.edu
[2] Universitat Politecnica de Catalunya (UPC)
larrosa@lsi.upc.es

Abstract. Computing lower bounds to the best-cost extension of a tuple is an ubiquous task in constraint optimization. A particular case of special interest is the computation of lower bounds to all singleton tuples, since it permits domain pruning in Branch and Bound algorithms. In this paper we introduce MCTE(z), a general algorithm which allows the computation of lower bounds to arbitrary sets of tasks. Its time and accuracy grows as a function of z allowing a controlled tradeoff between lower bound accuracy and time and space to fit available resources. Subsequently, a specialization of MCTE(z) called MBTE(z) is tailored to computing lower bounds to singleton tuples. Preliminary experiments on Max-CSP show that using MBTE(z) to guide dynamic variable and value orderings in branch and bound yields a dramatic reduction in the search space and, for some classes of problems, this reduction is highly cost-effective producing significant time savings and is competitive against specialized algorithms for Max-CSP.

1 Introduction

One of the main successes in constraint satisfaction is the development of *local consistency* properties and their corresponding *consistency enforcing algorithms* [19, 11]. They allow to *infer* and make explicit constraints that are implicit in the problem. Most useful in practice are consistency enforcing algorithms that *filter out* values that cannot participate in a solution. Filtering algorithms can be embedded into a search-based solver, propagating the effect of the current assignment towards future variables by pruning infeasible values under the current assignment [20, 3, 6].

Several attempts have been made in recent years to extend the notion of local consistency to *constraint optimization* problems [4, 5, 21]. The main difficulty being that inferred soft constraints cannot be carelessly added to the problem, due to the non-idempotency of the operator used to aggregate costs. A whole line of research mitigates this problem by extending only directional local consistency

* This work was supported in part by NSF grant IIS-0086529, by MURI ONR award N00014-00-1-0617 and Spanish Cicyt project TAP1999-1086-C03-03

to soft constraints and focuses on its most practical use: detecting *lower bounds* for the best extension of tuples[23, 9, 17, 21, 13, 14]. When there is an upper bound on the maximum cost of a solution, tuples having a lower bound higher than this bound cannot participate in an optimal solution and can be viewed as infeasible (i.e., a *nogood*). As in the CSP context, lower bounds for values (singleton tuples) are of special interest, because they can be used to filter out infeasible values.

This paper, introduces MCTE(z), a general tree decomposition method for *multiple* lower bound computation, and MBTE(z), its specialization to tree that compute singleton tuples. Our scheme is built on top of *cluster-tree elimination* (CTE), a tree-based decomposition schema which unifies several approaches for automated reasoning tasks. Algorithm MCTE(z) approximates CTE using a partitioning idea similar to *mini-buckets* [9]. The parameter z controls its complexity (which is exponential in z) as well as its accuracy, and can therefore be tuned to best fit the available resources.

After describing CTE and introducing MCTE (sections 3 and 4), we describe MBTE(z) in Section 5. As we show in the empirical section, MBTE(z) facilitates a parameterized dynamic look-ahead method for variable and value ordering heuristics in branch and bound. The parameter controls its pruning power and overhead, and can therefore adjust branch and bound to different levels of problem hardness: while low accuracy suffices for easy problems, higher accuracy may be more cost-effective when problems grow harder and larger.

Lower bounds for singleton tuples can be obtained by n runs of the mini-bucket elimination MBE(z) [9] which we will call nMBE(z). We contrast MBTE(z) against this alternative nMBE(z). We argue that for the same level of accuracy (same parameter z), MBTE(z) is considerably more efficient (up to linear speed-up). Time efficiency is of the essence when the ultimate goal is to use these algorithms at every node of a branch and bound search. Indeed, our preliminary experiments on Max-CSP (Section 7) support theory-based expectations regarding MBTE(z)'s accuracy as a function of z as well as its speed-up relative to nMBE(z). Most significantly, however, we demonstrate the potential of embedding MBTE(z) in Branch and Bound, showing a dramatic pruning power in search space relative to competitive Branch and Bound algorithms, which, for some problem classes is highly cost-effective. For space considerations, some of the experiments and proofs can be found in the full paper in [15] appearing in http://www.ics.uci.edu/˜ dechter/publications/.

2 Preliminaries

Definition 1 (sum of functions, variable elimination). *Let f and g be two functions defined over $var(f)$ and $var(g)$, respectively. Then,*

1. *The* sum *of f and g, denoted $f + g$, is a new function defined over $var(f) \cup var(g)$ which returns for each tuple the sum of values given by f and g, $(f + g)(t) = f(t) + g(t)$*

2. *The* elimination *of x_i from f by minimization, denoted* $\min_{x_i} f$, *is a new function defined over $var(f) - \{x_i\}$ which returns for each tuple the minimum cost extension to f, $(\min_{x_i} f)(t) = \min_{a \in D_i}\{f(t,a)\}$ where D_i denotes the domain of variable x_i and $f(t,a)$ denotes the value of f on the tuple t extended with value a assigned to x_i. We use $(\min_S f)(t)$, to denote the elimination of a set of variables $S \subseteq var(f)$.*

Definition 2 (lower bound function). *Let f and g be two functions defined over the same scope (same set of arguments). We say that g is a lower bound of f, denoted $g \leq f$, iff $g(t) \leq f(t)$, for all t.*

Definition 3 (constraint optimization problem (COP), constraint graph). *A* constraint optimization problem *(COP) is a triplet $P =< X, D, F >$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of finite domains and $F = \{f_1, \ldots, f_m\}$ is a set of constraints. Constraints can be either* soft *(i.e., cost functions) or* hard *(i.e., sets of allowed tuples). Without loss of generality we assume that hard constraints are represented as (bi-valued) cost functions. Allowed and forbidden tuples have cost 0 and ∞, respectively. The* constraint graph *of a problem P has the variables as its nodes, and two nodes are connected if they appear in a scope of a function in F.*

Definition 4 (optimization tasks, global and singleton). *Given a COP instance P, a set of optimization tasks is defined by $Z = \{Z_i\}_{i=1}^k$, $Z_i \subseteq X$ where for each Z_i the task is to compute a function g_i over Z_i, such that $g_i(t)$ is the best cost attainable by extending t to X. Formally, $g_i(t) = \min_{X-Z_i}(\sum_{j=1}^m f_j)$. A* global optimization *is the task of finding the best global cost, namely $Z = \{\emptyset\}$.* Singleton optimization *is the task of finding the best-cost extension to every singleton tuple (x_i, a), namely $Z = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$.*

Bucket elimination (BE) [7] is an algorithm for global optimization. Roughly, the algorithm starts by partitioning the set of constraints into n buckets, one per variable. Then variables are eliminated one by one. For each variable x_i, a new constraint h_i is computed using the functions in its bucket, summarizing the effect of x_i on the rest of the problem. h_i is then placed in the bucket of the last variable in its scope. After processing the last variable, only an empty-scope constraint (i.e., a constant function) containing the cost of the best solution remains in the problem. The bucket-elimination algorithm is time and space exponential in a graph parameter called induced-width (to be defined later).

Mini-bucket elimination (MBE) [9] is an approximation of BE that mitigates its high time and space complexity. When processing variable x_i, its bucket is partitioned into mini-buckets. Each mini-bucket is processed independently, producing bounded arity functions which are cheaper to compute and store. This paper extends the idea of mini-bucket elimination from variable-elimination algorithms to tree-decomposition schemes.

3 Cluster-Tree Elimination (CTE)

In this Section we present *cluster-tree elimination* (CTE), a general decomposition method for automated reasoning tasks. The algorithm is not new, it is a unifying description of variants of such algorithms appearing in the past 2 decades both in the constraints community and the probabilistic reasoning community [18, 8, 22, 12]. We describe the scheme in some detail since it will allow presenting our approximation in the most general setting. We also provide refined complexity analysis (see [15] for additional details). CTE is based on the concept of *tree-decomposition*. We use notation borrowed from [12].

Definition 5 (tree-decomposition, separator, eliminator). *Given a COP instance P, a* tree-decomposition *is a triplet $< T, \chi, \psi >$, where $T = (V, E)$ is a tree, and χ and ψ are labeling functions which associate with each vertex $v \in V$ two sets, $\chi(v) \subseteq X$ and $\psi(v) \subseteq F$ that satisfy the following conditions:*

1. *For each function $f_i \in F$, there is* exactly *one vertex $v \in V$ such that $f_i \in \psi(v)$. Vertex v satisfies that $var(f_i) \subseteq \chi(v)$.*
2. *For each variable $x_i \in X$, the set $\{v \in V | x_i \in \chi(v)\}$ induces a connected subtree of T. This is called the* running intersection *property.*

Let (u, v) be an edge of a tree-decomposition, the separator *of u and v is defined as $sep(u, v) = \chi(u) \cap \chi(v)$; the* eliminator *of u and v is defined as $elim(u, v) = \chi(u) - sep(u, v)$.*

Definition 6 (tree-width, hyper-width, maximum separator size). *The tree-width of a tree-decomposition is $tw = \max_{v \in V} |\chi(v)| - 1$, its hyper-width is $hw = \max_{v \in V} |\psi(v)|$, and its maximum separator size is $s = \max_{(u,v) \in E} |sep(u, v)|$*

Definition 7 (valid tree-decomposition). *We say that the tree-decomposition $< T, \chi, \psi >$ is* valid *for a set of optimization tasks $Z = \{Z_i\}_{i=1}^k$ if for each Z_i there exists a vertex defined as $\{v \in V | \ \chi(v) = Z_i\}$. Such vertices are called* solution-vertices[1].

Example 1. Consider a constraint optimization problem P with six variables $\{x_1, \ldots, x_6\}$ and six constraints $\{f_1, \ldots, f_6\}$ with scopes: $var(f_1) = \{x_5, x_6\}$, $var(f_2) = \{x_1, x_6\}$, $var(f_3) = \{x_2, x_5\}$, $var(f_4) = \{x_1, x_4\}$, $var(f_5) = \{x_2, x_3\}$ and $var(f_6) = \{x_1, x_2\}$, respectively. Figure 2 depicts a tree-decomposition valid for $Z = \{\{x_1, x_5, x_6\}, \{x_1, x_2, x_5\}\}$ (v_1 and v_2 are solution-vertices for the first and second tasks, respectively).

Algorithm CTE (Figure 1) computes the solution to a set of tasks by processing a valid tree-decomposition. It works by computing *messages* that are sent along edges in the tree. Message $m_{(u,v)}$ is a function computed at vertex u and sent to vertex v. For each edge, two messages are computed. One in each

[1] Normally, solution-vertices are only implicitly required. In our formulation we require them explicitly in order to simplify the algorithmic presentation.

Procedure CTE
Input: A COP instance P, a set of tasks $Z = \{Z_i\}_{i=1}^{k}$ and a valid tree-decomposition $< T, \chi, \psi >$.
Output: An augmented tree such that each solution-vertex for Z_i contains the solution to task Z_i.

Repeat

1. Select and edge (u, v) such that $m_{(u,v)}$ has not been computed and u has received messages from all adjacent vertices other than v
2. $m_{(u,v)} \leftarrow \min_{elim(u,v)} \sum_{g \in cluster(u), g \neq m_{(v,u)}} g$ (where $cluster(u) = \psi(u) \cup \{m_{(w,u)} | (w, u) \in T\}$)

Until all messages have been computed

Fig. 1. Algorithm cluster-tree elimination (CTE)

direction. Message $m_{(u,v)}$ can be computed as soon as all incoming messages to u other than $m_{(v,u)}$ have been received. Initially, only messages at leaves qualify. The set of functions associated with a vertex u augmented with the set of incoming messages is called a *cluster*, $cluster(u) = \psi(u) \cup_{(w,u) \in T} m_{(w,u)}$. A message $m_{(u,v)}$ is computed as the sum of all functions in $cluster(u)$ excluding $m_{(v,u)}$ and the subsequent elimination of variables in the eliminator of u and v. Formally, $m_{(u,v)} = \min_{elim(u,v)} (\sum_{g \in cluster(u), g \neq m_{(v,u)}} g)$. The algorithm terminates when all messages are computed. A solution to task Z_i is contained in any of its solution-vertices, as the sum of all functions in the cluster, $\sum_{g \in cluster(u)} g$.

Example 2. Figure 2 also shows the execution trace of CTE along the tree-decomposition, as the messages sent along the tree edges. Once messages are computed, solutions are contained in the solution-vertices. For instance, the solution to task $\{x_1, x_2, x_5\}$ is contained in $cluster(v_2)$ as $m_{(v_1,v_2)} + m_{(v_3,v_2)} + f_3 + f_6$. Similarly, the solution to task $\{x_1, x_5, x_6\}$ is contained in $cluster(v_1)$ as $f_1 + f_2 + m_{(v_2,v_1)}$.

Theorem 1 (correctness [18, 8, 22]). *Algorithm CTE is correct. Namely, for each solution-vertex v of Z_i, $\sum_{g \in cluster(v)} g = \min_{X - Z_i} (\sum_{j=1}^{n} f_j)$*

We can show that,

Theorem 2 (complexity). *The complexity of CTE is time $O(r \cdot (hw + dg) \cdot d^{tw+1})$ and space $O(r \cdot d^s)$, where r is the number of vertices in the tree-decomposition, hw is the hyper-width, dg is the maximum degree (i.e., number of adjacent vertices) in the graph, tw is the tree-width, d is the largest domain size in the problem and s is the maximum separator size.*

Since CTE is time and space exponential in tw and s, respectively, low width tree-decompositions are desirable (note that $tw+1 \geq s$). Finding the minimum width decomposition-tree is known to be NP-complete [1], but various approximation algorithms are available [2].

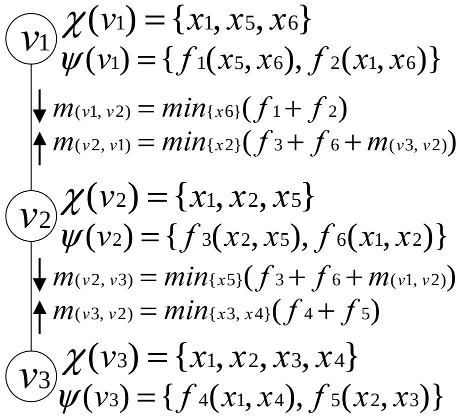

Fig. 2. Execution-trace of CTE along a tree-decomposition.

4 Mini-Cluster-Tree Elimination (MCTE)

The time and especially the space complexity of CTE renders the method infeasible for high-width tree-decompositions. One way to decrease the algorithm's complexity is to bound the size of the messages' arity to a predefined size z. This idea, called *mini-buckets*, was first introduced in the bucket elimination context [9]. Here we extend it from approximating bucket elimination to the more general setting of approximating CTE.

Let G be a set of functions having variable x_i in their scope. Suppose we want to compute a target function as the sum of functions in G and subsequently eliminate variable x_i (i.e., $\min_{x_i}(\sum_{g \in G} g)$). If exact computation is too costly, we can partition G into sets of functions $\mathcal{P}(G) = \{\mathcal{P}_j\}_{j=1}^{k}$ called mini-buckets, each one having a combined scope of size bounded by z. Such a partition is called a *z-partition*. If more than one partition is possible, any one is suitable. Subsequently, a bounded arity function h^j is computed at each mini-bucket $\mathcal{P}_j$ as the sum of all its included functions followed by the elimination of x_i (i.e., $h^j = \min_{x_i}(\sum_{g \in \mathcal{P}_j} g)$). The result is a set of functions $\{h^j\}_{j=1}^{k}$ which provides a lower bound to the target function. Namely, $\sum_j h^j \leq \min_{x_i} \sum_{g \in G} g$.

If more than one variable has to be eliminated, the process is repeated for each, according to a predefined ordering. Procedure $\mathsf{MiniBucketsApprox}(V, G, z)$ (Fig. 3) describes this process. Each iteration of the loop performs the elimination of one variable.[2]

Applying this idea to CTE yields a new algorithm called *mini-cluster-tree elimination* (MCTE(z)) The algorithm can be obtained by replacing line 2 in CTE by:

[2] Another option is to eliminate all variables *at once* from each mini-bucket (i.e., $h^j = \min_V(\sum_{g \in \mathcal{P}_j} g)$). While correct, it will provide less accurate lower bounds.

Procedure MiniBucketsApprox(V, G, z)
Input: a set of ordered variables V, a set of functions G, parameter z
Output: a set of functions $\{h^j\}_{j=1}^k$ that provide a lower bound as $\sum_{j=1}^k h^j \leq \min_V(\sum_{g \in G} g)$
for each $x_i \in V$ from last to first **do**
    $G' \leftarrow \{g \in G \mid x_i \in var(g)\}$
    compute $\mathcal{P}(G') = \{\mathcal{P}_j\}_{j=1}^k$ a z-partition of G'
    $h^j \leftarrow \min_{x_i}(\sum_{g \in \mathcal{P}_j} g)$, for $j = 1..k$
    $G \leftarrow (G - G') \cup \{h^j\}$
Return: G

Fig. 3. Procedure MiniBucketsApprox(V, G, z).

2. $M_{(u,v)} \leftarrow MiniBucketApprox(elim(u,v), cluster(u) - M_{(v,u)}, z)$ (where a message $M_{(u,v)}$ is a set of functions and $cluster(u) = \psi(u) \cup \{M_{(w,u)} | (w,u) \in T\})$

It works similar to CTE except that each time that a message has to be computed, the set of functions required for the computation are partitioned into *mini-buckets*, producing a set of *mini-messages* that are transmitted along the corresponding edge. Thus, in MCTE(z), a message is a *set* of bounded arity functions, $M_{(u,v)} = \{m^j_{(u,v)}\}$ (note that we use upper-case to distinguish MCTE(z) messages from CTE messages). A cluster is now the union of all messages and all functions in a node, $cluster(u) = \psi(u) \cup_{(w,u) \in T} M_{(w,u)}$ The message $M_{(u,v)}$ is computed by calling MiniBucketsApprox(V, G, z) with $V = elim(u,v)$ and $G = cluster(u) - M_{(v,u)}$. When all messages have been computed, a lower bound to task Z_i is contained in its solution-vertex v as the sum of all the functions in its cluster, $\sum_{g \in cluster(v)} g$.

Example 3. Figure 4 shows the execution-trace of MCTE(2) with our running example and the tree-decomposition of Fig. 2. For instance, the computation of $M_{(v_3,v_2)}$ requires a 2-partition of $(cluster(v_3) - M_{(v_2,v_3)}) = \{f_4(x_1,x_4), f_5(x_2,x_3)\}$. The only 2-partition here is $\mathcal{P}_1 = \{f_4\}$ and $\mathcal{P}_2 = \{f_5\}$, which yields a two-functions message $M_{(v_3,v_2)} = \{\min_{x_4}(f_4), \min_{x_3}(f_5)\}$.

Theorem 3 (correctness). *Given a valid tree-decomposition, MCTE(z) computes a lower bound for each task Z_i. Specifically, if u is a solution-vertex of task Z_i then,* $\sum_{g \in cluster(u)} g \leq \min_{X - Z_i}(\sum_{j=1}^n f_j)$

In order to analyze the complexity of MCTE(z) we define a new labeling ψ^*, which depends on the tree-decomposition structure.

Definition 8 (ψ^*, induced hyper-width (hw^*)). *Let $P =< X, D, F >$ be a COP instance and $< T, \chi, \psi >$ be a tree-decomposition. We define a labeling function ψ^* over nodes in the tree as, $\psi^*(v) = \{f \in F | \ var(f) \cap \chi(v) \neq \emptyset\}$. The* induced hyper-width *of a tree-decomposition is $hw^* = \max_{v \in V} |\psi^*(v)|$*

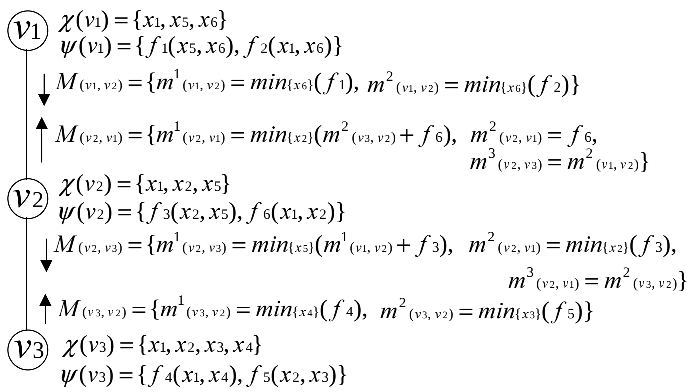

Fig. 4. An execution trace of MCTE(2).

Observe that $\psi^*(u)$ is a superset of $\psi(u)$ which includes those cost functions not in $\psi(u)$ that *may travel* to cluster u via message-passing. It can be shown that the *induced hyper-width* bounds the maximum number of functions that can be in a cluster, and therefore the number of mini-buckets in a cluster. Namely $hw^* \geq \max_{v \in V} |cluster(v)|$. Note that $hw \leq hw^* \leq m$, where hw is the hyper-width and m is the number of input functions.

Theorem 4 (complexity). *Given a problem P and a tree-decomposition T having induced hyper width hw^*, MCTE(z) is time and space $O(r \times hw^* \times d^z)$, where r is the number of nodes in T, and d bounds the domain size.*

Clearly, increasing z is likely to provide better lower bounds at a higher cost. Therefore, MCTE(z) allows trading lower bound accuracy for time and space complexity. There is no guaranteed improvement however.

5 MBTE(z): Computing Bounds to Singleton Tuples

There are a variety of ways in which valid tree-decompositions can be obtained. We analyze a special decomposition called bucket-trees, which is particularly suitable for the multiple singleton optimality task (def. 4). The concept of bucket-tree is inspired from viewing bucket-elimination algorithms as message-passing along a tree [7]. A *bucket-tree* can be defined over the *induced graph* relative to a variable ordering.

Definition 9 (induced graph, induced width [7]). *An* ordered constraint graph *is a pair (G, o), where G is a constraint graph and $o = x_1, ..., x_n$ is an ordering of its nodes. Its* induced graph $G^*(o)$ *is obtained by processing the nodes recursively, from last to first: when node x_i is processed, all its lower neighbors are connected. The* induced width $w^*(o)$ *is the maximum number of lower neighbors over all vertices of the induced graph.*

Definition 10 (bucket-tree). *Given the induced graph $G^*(o)$ of a problem P along ordering o, a* bucket-tree *is a tree-decomposition $< T, \chi, \psi >$ is defined as follows. (i) There is a vertex v_i associated with each variable x_i. The parent of v_i is v_j iff x_j is the closest lower neighbor of x_i in $G^*(o)$. (ii) $\chi(v_i)$ contains x_i and every lower neighbor of x_i in $G^*(o)$. (iii) $\psi(v_i)$ contains every constraint having x_i as the highest indexed variable in its scope.*

Notice that in a bucket-tree, vertex v_1, the root, is a solution-vertex for the task $\{x_1\}$. The bucket-tree can be augmented with solution-vertices for each singleton-optimality task. A vertex u_i with $\chi(u_i) = \{x_i\}$ and $\psi(u_i) = \emptyset$ is added for $i = 2..n$. Vertex v_i is the parent of u_i. Subsequently, we define algorithm *Bucket-tree elimination* (BTE) to be CTE applied to the augmented bucket-tree.

Example 4. Figure 5 shows the execution-trace of BTE on our running example. Observe that messages from u-nodes to v-nodes do not need to be sent because they are null functions ($\psi(u_i) = \emptyset$).

Observe that BTE computes the exact singleton optimality problem. Observe also that BTE can be viewed as a two-phases algorithm. The first phase (where messages from leaves to root are transmitted) is equivalent to *bucket elimination* (BE) [7]: Cluster v_i is the bucket of x_i. Incoming messages are new functions derived from higher buckets and added to the bucket of x_i. Computing message $m_{(v_i,p(v_i))}$, where $p(v_i)$ is the parent of v_i, performs the elimination of variable v_i and produces a new function (the message) that is sent to a lower bucket (the parent of v_i).

Next, *Mini bucket tree elimination* (MBTE(z)) is defined by approximating BTE via mini-buckets or, equivalently, by executing MCTE(z) over the augmented bucket-tree. BTE and MBTE(z) process the same tree-decomposition but in MBTE(z) clusters are z-partitioned, producing mini-bucket-based messages. From Theorem 4 we can conclude,

Theorem 5 (complexity). *Given a variable ordering, the complexity of MBTE(z) is $O(n \cdot hw^* \cdot d^z)$, when n is the number of variables and hw^* is the bucket-tree induced hyper-width.*

MBTE vs nMBE: It is easy to see that *mini-bucket elimination* MBE(z) [9] is equivalent to the first message-passing phase of MBTE(z). In particular, running MBE(z) n times, an algorithm that we call nMBE(z), each time having a different variable initiating the ordering, is an alternative for the singleton optimality problem. MBTE and nMBE are closely related in terms of accuracy. Specifically, if MBE(z) is executed each time with the appropriate variable orderings, both approaches will produce exactly the same bounds, when using the same bucket-partitioning strategy. Clearly, however, MBTE(z) is always more efficient than multiple executions of MBE(z), since MBE(z) repeats message computation at different executions. The following Theorem summarizes these properties.

Theorem 6. *Let P be a constraint optimization problem and o a variable ordering. Lets consider the execution of MBTE(z) over the bucket-tree relative to o.*

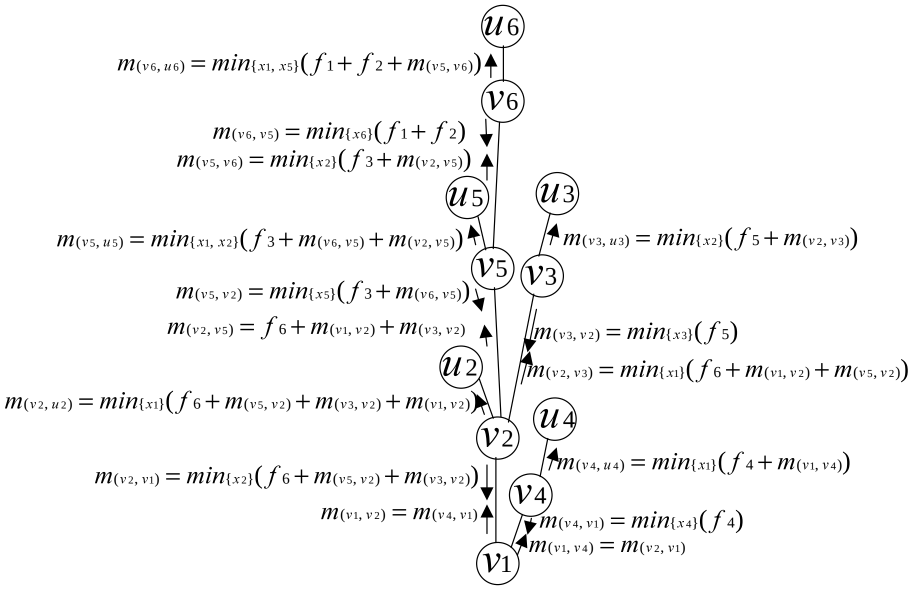

Fig. 5. An execution trace of BTE for the task of computing the best extension of all singleton tuples. If only top-down messages are considered, the algorithm is equivalent to BE

- *(Accuracy) For each variable x_i, there is an ordering o_i initiated by x_i such that executing MBE(z) with o_i produces the same lower bound as MBTE(z) for task $\{x_i\}$, provided that both algorithms use the same criterion to select z-partitions.*
- *(time comparison) Let nMBE(z) be n executions of MBE(z) using the n previously defined o_i orderings. Then, every message computed by MBTE(z) is also computed by nMBE(z), and there are some messages that are computed multiple times (up to n) by nMBE(z).*

Thus, MBTE(z) is never worse than nMBE(z). Since the complexity of running MBE(z) n times is $O(n \cdot m \cdot d^z)$ and MBTE(z) is $O(n \cdot hw^* \cdot d^z)$, significant gains are expected when hw^* is smaller relative to m.

6 Comparison of MBTE with Soft Arc-consistency

Soft arc-consistency (SAC) [21] is the most general of a sequence of bounds for singleton optimization. They are based in different forms of arc-consistency [17]. We consider the most general algorithm for SAC. Namely, the algorithm that after achieving soft arc-consistency, is allowed to iterate non-deterministically projecting and extending cost functions in order to increase, if possible, the available bounds ([21], Sec. 5).

In the following we briefly argue that there is no dominance relation between SAC and MBTE. Namely, there exist instances in which either approach computes better bounds than the other. In the full paper [15] we provide two examples illustrating this fact.

On the one hand, tree-decomposition based bounds such as MBTE need to transform the problem into an acyclic structure and each cost function has a single path to be propagated from one vertex to another. SAC works directly on the (possibly cyclic) constraint graph. Then the same function can be propagated simultaneously through different paths. As a result, information from a cost function may split and merge again. This fact allows SAC to outperform MBTE in some problem instances.

On the other hand, SAC algorithms can only project functions one by one, while MBTE can sum functions and project from the result. In a simplistic way, it is as if SAC is only allowed to compute bounds using $\sum_{f \in F} \min_V f$, while MBTE can perform $\min_V \sum_{f \in F} f$ as long as arities do not surpass value z. This fact allows MBTE to outperform SAC in some problem instances.

7 Empirical Results

We performed a preliminary experimental evaluation of the performance of MBTE(z) on solving the singleton optimality task. *i*) we have investigated the performance of MBTE(z) against its obvious brute-force alternative – nMBE(z), and showed that MBTE(z) achieves a significant speedup over nMBE(z). *ii*) we demonstrated that as expected, MBTE(z) accuracy grows as function of z, thus allowing a trade-off between accuracy and complexity. *iii*) we evaluated the effectiveness of MBTE(z) in improving Branch and Bound search. For space reasons we report only the search experiments. Details on experiments with speed-up and accuracy are available in the full paper [15].

All our experiments are done using the Max-CSP task as a sample domain. Max-CSP is an optimization version of Constraint Satisfaction and its task is to find an assignment that satisfies the most constraints. We use its formulation as a minimization problem where each constraint is a cost function that assigns a cost 1 to each nogood and cost 0 to each allowed tuple. We used the well known four parameter model, $< N, K, C, T >$, for random problem generation, where N is the number of variables, K is the domain size, C is the number of constraints, and T is the tightness of each constraint (see [16] for details).

7.1 BBBT: Branch and Bound with MBTE(z)

Since MBTE(z) computes lower bounds for each singleton-variable assignment, when incorporated within a Branch-and-Bound search, MBTE(z) can facilitate domain pruning and dynamic variable ordering. In this section we investigate the performance of such a new algorithm, called BBBT(z) (Branch-and-Bound with Bucket-Tree heuristics), and compare it against BBMB(z) [13].

$N = 50$, $K = 5$, $C = 150$. $w^* = 17.6$. 10 instances. time = 600sec.							
	BBMB					BBBT	PFC-MRDAC
T	z=2	z=3	z=4	z=5	z=6	z=2	
	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks
5	6 45 1.11M	7 54 1.51M	6 6.2 177K	9 75 2.29M	10 6.2 123K	10 1.9 55	10 0.01 436
7	4 134 5.86M	5 150 4.62M	7 213 5.3M	8 208 5.14M	9 97 2.1M	10 2.5 94	10 1.7 15K
9	-	-	1 325 7.4M	3 227 4.97M	3 229 4.85M	10 14.3 2.1K	10 27.3 242K

Table 1. BBBT(z) vs. BBMB(z).

$N = 100$, $K = 5$, $C = 300$. $w^* = 33.9$. 10 instances. time = 600sec.								
	BBMB						BBBT	PFC-MRDAC
T	z=2	z=3	z=4	z=5	z=6	z=7	z=2	
	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks
3	6 6 150K	6 6 150K	6 6 150K	6 5 115K	8 6.8 115K	8 15 8	10 7.73 60	10 0.03 750
5	2 36 980K	2 32 880K	2 24 650K	2 5.3 130K	3 38 870K	3 33 434K	10 14.3 114	10 0.06 1.5K
7	0	0	0	0	0	0	10 29 331	6 267 1.6M

Table 2. BBBT(z) vs. BBMB(z).

BBMB(z) [13] is a Branch-and-Bound search algorithm that uses Mini-Bucket Elimination (MBE(z)) as a pre-processing step. MBE(z) generates intermediate functions that are used to compute a heuristic value for each node in the search space. Since these intermediate functions are pre-computed, before search starts, BBMB(z) uses the same fixed variable ordering as MBE(z). Unlike BBBT(z), BBMB(z) does not prune domains of variables. In the past [14] we showed that BBMB(z) was effective and competitive with alternative state-of-the-art algorithms for Max-CSP.

BBBT(z) is a Branch-and-Bound search algorithm that uses MBTE(z) at each node in the search space. Unlike BBMB(z), BBBT(z) has no pre-processing step. At each node in the search space, MBTE(z) is used to compute lower bounds for each variable-value assignment of future variables. These lower bounds are used for domain pruning – whenever a lower bound of a variable-value assignment is not less than the global upper bound, the value is deleted. BBBT(z) backtracks whenever an empty domain of a future variable is created. BBBT(z) also uses dynamic variable ordering – when picking the next variable to instantiate, it selects a variable with the smallest domain size. Ties are broken by picking a variable with the largest sum of lower bounds associated with each value. In addition, for value selection, BBBT(z) selects a value with the smallest lower bound.

In Tables 1-3 we have the results of experiments with three sets of Max-CSP problems: N=50, K=5, C=150, $5 \leq \mathrm{T} \leq 9$, N=100, K=5, C=300, $3 \leq \mathrm{T} \leq 7$, and N=50, K=5, C=100, T=15. On each problem instance we ran BBMB(z) for different values of z, as well as BBBT(2). We also ran BBBT(z) for larger values of z, but BBBT(2) was most cost effective on these problems. For comparison, we also report the results with PFC-MRDAC [17] that is currently one of the best algorithms for Max-CSP.

$N = 50$, $K = 5$, $C = 100$. $w^* = 10.6$. 10 instances. time = 600sec.							
	BBMB			BBBT			PFC-MRDAC
T	z=4	z=6	z=8	z=2	z=5	z=8	
	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks	# solved time backtracks
15	8 184 13M	10 4.51 120K	10 12.4 24K	3 394 22K	9 190 565	8 91 30	9 108 1.0M

Table 3. BBBT(z) vs. BBMB(z).

In column 1 we have the tightness, in the last two columns we report BBBT(2) and PFC-MRDAC, and in the middle columns we have BBMB(z). For each set of problems, we report the number of problems solved (within the time bound of 600 seconds), the average CPU time and number of deadends for solved problems. For example, we see from Table 1 (N=50, K=5, C=150), that when tightness T is 5, BBMB(6) solved all 10 problems, taking 6.2 seconds and 123 thousand backtracking steps, on the average, whereas BBBT(2) also solved all 10 problems, taking 1.9 seconds and 55 backtracking steps, on the average.

We see from Tables 1 and 2 that on these two sets of problems, BBBT(2) is vastly superior to BBMB(z), especially as the tightness increases. Average CPU time of BBBT(2) is as much as an order of magnitude less than BBMB(z). Sporadic experiments with 200 and 300 variable instances showed that BBBT(2) continues to scale up very nicely on these problems. BBBT(2) is also faster than PFC-MRDAC on tight constraints.

The experiments also demonstrate the pruning power of MBTE(z). The number of backtracking steps used by BBBT(2) is up to three orders of magnitude less than BBMB(z). For example, we see from Table 1 that when tightness T is 7, BBMB(6) solved 9 problems out of 10, taking 2.1 million backtracking steps in 97 seconds, whereas BBBT(2) solved all 10 problems, taking 94 backtracking steps in 2.5 seconds.

We observed a different behavior on problems having sparser constraint graphs and tight constraints. While still very effective in pruning the search space, BBBT was not as cost-effective as BBMB(z) (which invests in heuristic computation only once). Table 3 exhibits a typical performance (N=50,C=100, K=5, T=15). We observe that here BBBT's performance exhibit a U-shape, improving with z up to an optimal z value. However, BBBT's slope of improvement is much more moderate as compared with BBMB.

8 Conclusions and future work

Since constraint optimization is NP-hard, approximation algorithms are of clear practical interest. In the paper we extend the mini-bucket scheme proposed for variable elimination to tree-decomposition. We have introduced a new algorithm for lower bound computation, MCTE(z), applicable to arbitrary sets of tasks. The parameter z allows trading accuracy for complexity and can be adjusted to best fit the available resources. MBTE(z) is a special case of MCTE(z) for the computation of lower bounds to singleton optimization, based on a bucket-tree. This task is relevant in the context of branch and bound solvers. Both algorithms have been derived to approximate CTE, a tree-decomposition schema for reasoning tasks which unifies a number of approaches appearing in the past 2 decades in the constraint satisfaction and probabilistic reasoning context.

We have shown that bounds obtained with MBTE(z) have the same accuracy as if computed with n runs of plain mini-buckets. The quality of such accuracy has already been demonstrated in a number of domains [9]. We have also shown that MBTE(z) can be up to n times faster than the alternative of running plain mini-buckets n times. This speed-up is essential if the algorithm is to be used at every node within a branch and bound solver. Our preliminary experiments suggest that MBTE(z) is very promising. It generates good quality bounds at a reasonable cost. When incorporated within branch and bound, it reduces dramatically the search space explored which sometimes translates into great time savings. Note that our implementation is general and has not yet been optimized.

Our approach leaves plenty of room for future improvement, which are likely to make it more cost effective in practice. For instance, it can be modified to treat separately hard and soft constraints, since hard constraints can be more efficiently processed and propagated [10]. As a matter of fact, even if the original problem has no hard constraints, our approach can be used to infer them (i.e., detect infeasible tuples). Also, currently our partitioning to mini-buckets was always random. Investigating heuristics for partitioning may increase the accuracy of the algorithms.

References

[1] S.A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25:2–23, 1985.

[2] A. Becker and D. Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Uncertainty in AI (UAI'96)*, pages 81–89, 1996.

[3] C. Bessiere and J.-C. Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. *Lecture Notes in Computer Science*, 1118:61–75, 1996.

[4] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Frameworks, properties and comparison. *Constraints*, 4:199–240, 1999.

[5] S. Bistarelly, R. Gennari, and F. Rossi. Constraint propagation for soft constraints: Generalization and termination conditions. In *Proc. of the 6^{th} CP*, pages 83–97, Singapore, 2000.

[6] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proc. of the 16^{th} IJCAI*, pages 412–417, Stockholm, Sweden, 1999.

[7] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[8] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[9] R. Dechter and I. Rish. A scheme for approximating probabilistic inference. In *Proceedings of the 13th UAI-97*, pages 132–141, San Francisco, 1997. Morgan Kaufmann Publishers.

[10] R. Dechter and P. van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 20 February 1997.

[11] E. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29:24–32, March 1982.

[12] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol1)*, pages 394–399, S.F., July 31–August 6 1999. Morgan Kaufmann Publishers.

[13] K. Kask. new search heuristics for max-csp. In *Proc. of the 6^{th} CP*, pages 262–277, Singapore, 2000.

[14] K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artificial Intelligence*, 129(1-2):91–131, 2001.

[15] K. Kask, J. Larrosa, and R. Dechter. A general scheme for multiple lower bound computation in constraint optimization. Technical report, University of California at Irvine, 2001.

[16] J. Larrosa and P. Meseguer. Partial lazy forward checking for max-csp. In *Proc. of the 13^{th} ECAI*, pages 229–233, Brighton, United Kingdom, 1998.

[17] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible DAC for max-CSP. *Artificial Intelligence*, 107(1):149–163, 1999.

[18] S. L. Lauritzen and D. J. Spiegelhalter. Local computation with probabilities on graphical structures and their applications to expert systems. *Journal of the Royal Statistical Society, Series B*, 34:157–224, 1988.

[19] A. Mackworth. Consistency in networks of constraints. *Artificial Intelligence*, 8, 1977.

[20] B. Nudel. Tree search and arc consistency in constraint satisfaction algorithms. *Search in Artifical Intelligence*, 999:287–342, 1988.

[21] T. Schiex. Arc consistency for soft constraints. In *Proc. of the 6^{th} CP*, pages 411–424, Singapore, 2000.

[22] P.P. Shenoy. Binary join-trees for computing marginals in the shenoy-shafer architecture. *International Journal of Approximate Reasoning*, 2-3:239–263, 1997.

[23] G. Verfaillie, M. Lemaître, and T. Schiex. Russian doll search. In *Proc. of the 13^{th} AAAI*, pages 181–187, Portland, OR, 1996.