Appeared in: International Workshop on Preferences and Soft Constraints, Toronto, Canada, 2004

On-demand Bound Computation for Finding Leading Solutions to Soft Constraints

Martin Sachenbacher and Brian C. Williams

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
{sachenba, williams}@mit.edu

Abstract. An important class of algorithms for constraint optimization searches for solutions guided by a heuristic evaluation function (bound) that can be computed using tree decomposition and dynamic programming. Recently, it has been shown that if only a few best solutions are needed, the cost of pre-processing can be avoided by interleaving dynamic programming with search. In this paper, we extend this hybrid method to the case of A* search for semiring-CSPs with a total order. A* is a specialization of branch-and-bound that finds best solutions in an optimal number of steps, but requires more space due to a larger search tree. To address this, we show how the hybridization can be understood as demand-driven heuristics computation. This allows to use techniques from heuristic search to significantly limit the number of search nodes expanded. The resulting approach uses lazy, best-first variants of constraint projection and combination operators to compute only those bounds specifically required to generate a next best solution. Experiments on randomly generated Max-CSPs indicate performance improvements over classical dynamic programming methods for best-first search.

1 Introduction

Algorithms for constraint optimization are key to many problems in Artificial Intelligence, such as monitoring, diagnosis, planning, autonomous control, or reconfiguration. It is typical for such applications that only a few *leading* solutions are required, that is, a best solution and possibly a limited number of next best solutions. For instance, in fault diagnosis, it might be sufficient to compute the most likely diagnoses that cover most of the probability density space [15, 18]. In planning, it might be sufficient to compute a least-cost plan and some backup plans in case the best plan cannot be executed.

Algorithms such as A* or branch-and-bound find best solutions to constraint optimization problems using backtracking guided by a heuristic evaluation function (bound). Kask and Dechter [13, 10] have shown how such a heuristic function can be derived using a decomposition of the constraint network into a tree and an instance of dynamic programming. However, when only a few leading solutions are generated, performing dynamic programming prior to the search

becomes inefficient, because only a few of the bounds will typically be needed to compute the best solutions. Recently, it has been shown that the excessive cost of pre-processing can be avoided by interleaving dynamic programming with backtracking search [2, 12, 17]. This hybrid approach improves backtracking by recording information about solutions to subproblems (goods), leveraging the fact that their size is bounded by structural properties of the network. Terrioux and Jégou [17] present an instance of this method that uses tree decomposition and branch-and-bound to compute optimal solutions to binary valued CSPs [16].

In this paper, we extend this hybrid method to A* search, which is a specialization of branch-and-bound. Given a heuristics, A* is guaranteed to find the best solution in an optimal number of steps [6]. However, A* needs to maintain a larger search tree and therefore requires more space than branch-and-bound search. To address this difficulty, we show that the approach in [17] can also be understood as demand-driven dynamic programming (heuristics computation), leveraging the fact that the solutions are enumerated successively. This view then allows us to incorporate techniques from heuristic search to significantly reduce the size of the A* search tree: First, we use a novel expansion scheme [18] that limits the number of search nodes expanded by exploiting a preferential independence property. Second, we employ a tighter heuristics than in [12], based on a dual problem formulation that treats constraints as variables and tuples as domains. Third, we describe a scheme that allows distributed computation of the search tree. The approach is presented in the context of semiring-based CSPs [3] with a total order on the preferences (equivalent to valued CSPs [4]). It can be understood as devising lazy, best-first variants of the constraint combination and projection operators. The worst-case time complexity of the approach is similar to the algorithm in [12], but due to the A* search, it has a higher space complexity. However, the complexity is not worse than that of classical dynamic programming methods for best-first search [13], and it can derive the best solutions much faster. This is illustrated with preliminary experiments on randomly generated Max-CSPs.

2 Semiring-based Constraint Optimization Problems

Definition 1 (Semiring [3]). *A* c-semiring *is a tuple* $(A, +, \times, \mathbf{0}, \mathbf{1})$ *such that*

1. A *is a set and* $\mathbf{0}, \mathbf{1} \in A$*;*
2. $+$ *is a commutative, associative and idempotent (i.e.,* $a \in A$ *implies* $a + a = a$*) operation with unit element* $\mathbf{0}$ *and absorbing element* $\mathbf{1}$ *(i.e.,* $a + \mathbf{0} = a$ *and* $a + \mathbf{1} = \mathbf{1}$*);*
3. $\times$ *is a commutative, associative operation with unit element* $\mathbf{1}$ *and absorbing element* $\mathbf{0}$ *(i.e.,* $a \times \mathbf{1} = a$ *and* $a \times \mathbf{0} = \mathbf{0}$*);*
4. $\times$ *distributes over* $+$ *(i.e.,* $a \times (b + c) = (a \times b) + (a \times c)$*).*

For instance, $S_p = ([0, 1], \max, \cdot, 0, 1)$ forms a probabilistic c-semiring. The idempotency of the $+$ operation induces a partial order $\leq_S$ over A as follows: $a \leq_S b$ iff $a + b = b$ (for S_p, $\leq_S \equiv \leq$, and $+ \equiv \max$). In this paper, we assume that $\leq_S$ is a total order.

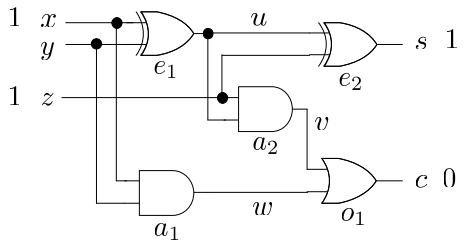

Fig. 1. The full adder example consists of two AND gates, one OR gate, and two XOR gates. Variables x, z, s, and c are observed as indicated.

Definition 2 (Semiring-based Constraint Optimization Problem). *A* constraint optimization problem *(COP) over a c-semiring is a triple* (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. The constraints* $f_j \in F$ *are functions defined over* $var(f_j)$ *assigning to each tuple a value in* A.

For example, diagnosis of the full adder circuit shown in Fig. 1 can be framed as a COP over S_p with variables $X = \{u, v, w, x, y, z, c, s, a_1, a_2, e_1, e_2, o_1\}$. Variables u to s are Boolean variables with domain $\{0, 1\}$. Variables a_1 to o_1 describe the mode of a component, "Good" or "Broken," and have domain {G,B}. If a component is good (G) then it correctly performs its Boolean function. If a component is broken (B) then no assumption is made about its behavior. We assume AND gates have a 1% probability of failure, whereas OR gates and XOR gates have a 5% probability of failure. We assume that input variables x and z are observed to be 1, whereas output variables s and c are observed to be 1 and 0, respectively (therefore, we omit them in the following). Table 1 shows the resulting constraints for the example, where each tuple is assigned the probability of its corresponding mode.

A semiring-CSP is solved by applying a series of combination and projection operations to its constraints:

Definition 3 (Combination and Projection). *Let* f *and* g *be two constraints defined over* $var(f)$ *and* $var(g)$*, respectively. Let* $t \downarrow_Y$ *denote the projection of a tuple on a subset* Y *of its variables. Then,*

1. *The* combination *of* f *and* g*, denoted* $f \otimes g$*, is a new constraint over* $\mathrm{var}(f) \cup \mathrm{var}(g)$ *where each tuple* t *has value* $f(t \downarrow_{\mathrm{var}(f)}) \times g(t \downarrow_{\mathrm{var}(g)})$*;*
2. *The* projection *of* f *on a set of variables* Y*, denoted* $f \Downarrow_Y$*, is a new constraint over* $Y \cap \mathrm{var}(f)$ *where each tuple* t *has value* $f(t_1) + f(t_2) + \ldots + f(t_k)$*, where* $t_1, t_2, \ldots, t_k$ *are all the tuples of* f *for which* $t_i \downarrow_Y = t$.

Given a COP (X, D, F) over a c-semiring, the constraint optimization task is to compute a function f over variables of interest $Z \subseteq X$ such that $f(t)$ is the best value attainable by extending t to X, that is, $f(t) = (\bigotimes_{j=1}^{m} f_j) \Downarrow_Z$.

3 Tree Decomposition and Bound-Guided Search

An important class of algorithms for constraint optimization finds the best solutions by searching through the space of possible assignments in best first order, guided by a heuristic evaluation function [13, 10, 18]. In the A* framework [6], the evaluation function is composed of the value of the partial assignment that has been made so far, g, and a heuristic h that provides an optimistic estimate (bound) on the optimal value that can be achieved for the complete assignment. In the case of semiring-CSPs, this is an upper bound:

Definition 4 (Upper Bound). *For two functions f_1, f_2 with $var(f_1) = var(f_1)$, f_2 is an* upper bound *of f_1, written $f_1 \leq_S f_2$, if $f_1(t) \leq_S f_2(t)$ for all t.*

Kask and Dechter [13] show how the bounding function h can be derived from a decomposition of the constraint network into an acyclic instance called a bucket tree:

Definition 5 (Induced Graph [9]). *For a COP (X, D, F), let H be a hypergraph which associates a node with each variable $x_i \in X$, and a hyperedge with each constraint $f_j \in F$. Let $x_1 \prec x_2 \prec \ldots \prec x_n$ an ordering on the variables X. Then the* induced graph *G^* of H is obtained by processing the variables from last to first, and interconnecting all the lower neighbors of each variable x_i.*

Definition 6 (Bucket Tree Decomposition [14]). *Given an induced graph G^*, a* bucket tree *is a triple (T, χ, λ). $T = (V, E)$ is a rooted tree that associates a vertex v_i with each variable x_i, such that the parent of v_i is v_j if x_j is the closest lower neighbor of x_i in G^*. χ and λ are labeling functions that associate with each node v_i, two sets $\chi(v_i) \subseteq X$ and $\lambda(v_i) \subseteq F$, such that*

1. *$\chi(v_i)$ contains x_i and every lower neighbor of x_i in G^*;*
2. *$\lambda(v_i)$ contains every $f_j \in F$ such that x_i is the highest variable in $var(f_j)$.*

In this paper, we derive a bounding function from a tree decomposition, which is a generalization of a bucket tree decomposition:

Definition 7 (Tree Decomposition [11, 14]). *A* tree decomposition *for a problem (X, D, F) is a triple (T, χ, λ), where $T = (V, E)$ is a rooted tree, and the labeling functions $\chi(v_i) \subseteq X$, and $\lambda(v_i) \subseteq F$ are defined such that*

Table 1. Constraints for the example (tuples with value **0** are not shown).

f_{a1}: $a1$ w y				f_{a2}: $a2$ u v				f_{e1}: $e1$ u y				f_{e2}: $e2$ u			f_{o1}: $o1$ v w			
G	0	0	.99	G	0	0	.99	G	1	0	.95	G	0	.95	G	0	0	.95
G	1	1	.99	G	1	1	.99	G	0	1	.95	B	0	.05	B	0	0	.05
B	0	0	.01	B	0	0	.01	B	0	0	.05	B	1	.05	B	0	1	.05
B	0	1	.01	B	0	1	.01	B	0	1	.05				B	1	0	.05
B	1	0	.01	B	1	0	.01	B	1	0	.05				B	1	1	.05
B	1	1	.01	B	1	1	.01	B	1	1	.05							

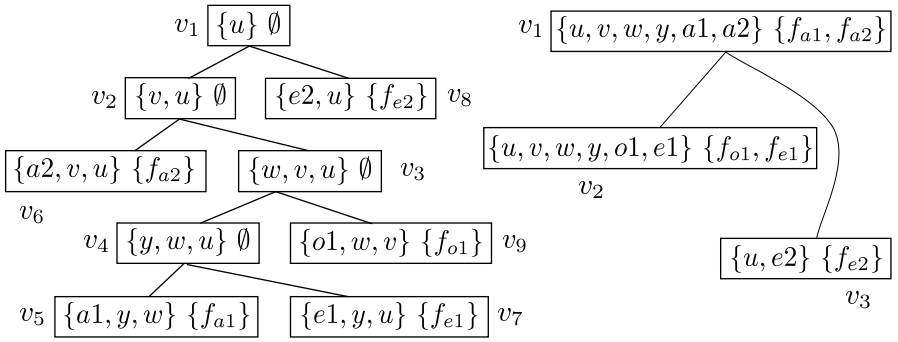

Fig. 2. Bucket tree (*left*) and tree decomposition (*right*) for the example in Fig. 1. The trees show the labels χ and λ for each node.

1. *For each $f_j \in F$, there exists exactly one v_i such that $f_j \in \lambda(v_i)$. For this v_i, $var(f_j) \subseteq \chi(v_i)$ (covering condition);*
2. *For each $x_i \in X$, the set $\{v_j \in V \mid x_i \in \chi(v_j)\}$ of vertices labeled with x_i induces a connected subtree of T (connectedness condition).*

The left-hand side of Fig. 2 shows a bucket tree for the full adder example, given the variable ordering $u, v, w, y, a_1, a_2, e_1, e_2, o_1$. The right-hand side of Fig. 2 shows a tree decomposition for the example. The tree T (bucket tree or tree decomposition) describes an equivalent, acyclic instance of the COP. To compute the evaluation function h, this acyclic instance can be evaluated by dynamic programming, implemented by a message-passing algorithm (called cluster-tree elimination in [10]) that processes the nodes of the tree bottom-up (that is, in post-order). At each node v_i, the constraint $\bigotimes_{f_k \in \lambda(v_i)} f_k$ is computed and combined with all constraints sent by the children of v_i (if any) to obtain a constraint f_{v_i}. If v_i has a parent node v_j, then v_i sends the constraint $h_{v_i} := f_{v_i} \Downarrow_{\chi(v_j)}$ to v_j.

The functions h_{v_i} computed by the dynamic programming algorithm can be exploited to guide the search for solutions to the to the constraint optimization task that consists of COP and the variables of interest are $Z = X$. For the case of a bucket tree, Kask and Dechter [13] describe an algorithm that assigns the variables top-down, that is, according to the variable order $x_1 \prec \ldots \prec x_n$ of the bucket tree. Consider a point in the search where the current assignment is $x_1 = x_1^0, \ldots, x_i = x_i^0$. Let function $g^{(i)}$ be defined as the combination of all constraint functions in the λ-label of nodes $v_1, \ldots, v_i$ in the bucket tree:

$$g^{(i)} = \bigotimes_{j=1}^{i} \Big(\bigotimes_{f_k \in \lambda(v_j)} f_k \Big).$$

Let function $h^{(i)}$ be defined as the combination of all functions sent by the nodes $c_1, \ldots, c_l$ that are children of $v_1, \ldots, v_i$:

$$h^{(i)} = \bigotimes_{j=1}^{l} h_{c_j}.$$

Then $g^{(i)} \times h^{(i)}(x_1^0, \ldots, x_i^0)$ is the best value achievable when completing this assignment. For example, consider the bucket tree on the left-hand side of Fig. 2 for the case where u to $a1$ have been assigned a value, that is, nodes v_1 to v_5 have been traversed. Then $g^{(5)} = f_{a1}$, and $h^{(5)} = h_{v_6} \otimes h_{v_7} \otimes h_{v_8} \otimes h_{v_9}$.

We generalize this idea from bucket trees to tree decompositions as follows. Let $p = v_1, \ldots, v_n$ be a pre-order of the tree nodes. The node order defines a partial variable order where the variables of a node precede the variables of the next node. Formally, let $\chi_{\mathrm{p}}(v_i) \subseteq \chi(v_i)$ be groups of variables defined by

$$\chi_p(v_1) = \chi(v_{\mathrm{root}}),\ \chi_p(v_{i+1}) = \chi(v_{i+1}) \setminus (\chi_p(v_1) \cup \ldots \cup \chi_p(v_i)).$$

Let $V_p(x_i)$ denote the node such that $x_i \in \chi_{\mathrm{p}}(v_k)$. Then the partial variable order defines $x_i \prec x_j$ if $V_p(x_i) \prec V_p(x_j)$.

A *compatible order* [12, 17] completes the partial variable order to a total order by ordering also the variables within the groups $\chi_{\mathrm{p}}(v_i)$. For example, for the tree on the right-hand side of Fig. 2, the node pre-order v_1, v_2, v_3 induces the three groups $\chi_{\mathrm{p}}(v_1) = \{u, v, w, y, a_1, a_2\}$, $\chi_{\mathrm{p}}(v_2) = \{e_1, o_1\}$, $\chi_{\mathrm{p}}(v_3) = \{e_2\}$. The partial order $\chi_{\mathrm{p}}(v_1) \prec \chi_{\mathrm{p}}(v_2) \prec \chi_{\mathrm{p}}(v_3)$ can be completed to the compatible order $u, v, w, y, a_1, a_2, e_1, o_1, e_2$.

The principle for deriving bounding functions for search carries over from bucket trees to tree decompositions, if compatible variable orders are used. Consider again an assignment $x_1 = x_1^0, \ldots, x_i = x_i^0$. Let function $g^{(i)}$ be generalized to be the combination of all constraints in the λ-label of nodes $v_1, \ldots, V_p(x_i)$ that are fully instantiated:

$$g^{(i)} = \bigotimes_{j=1}^{V_p(x_i)} \Big(\bigotimes_{f_k \in \lambda(v_j), \mathrm{var}(f_k) \subseteq \{x_1, \ldots, x_i\}} f_k \Big). \tag{1}$$

Let function $h^{(i)}$ be defined as the combination of all functions in the λ-label of $V_p(x_i)$ that are not fully instantiated, and all functions sent by the nodes $c_1, \ldots, c_l$ that are children of $v_1, \ldots, V_p(x_i)$, projected on $x_1, \ldots, x_i$:

$$h^{(i)} = \Big(\bigotimes_{j=1}^{l} h_{c_j} \bigotimes_{f_k \in \lambda(V_p(x_i)), \mathrm{var}(f_k) \not\subseteq \{x_1, \ldots, x_i\}} f_k \Big) \Downarrow_{\{x_1, \ldots, x_i\}}. \tag{2}$$

For example, consider the tree on the right-hand side of Fig. 2 and the case where the variables $\{u, v, w, y, a1\}$ have been assigned a value. Then $g^{(1)} \otimes h^{(1)}$ with $g^{(1)} = f_{a1}$ and $h^{(1)} = f_{a2} \Downarrow_{u,v} \otimes h_{v_2} \otimes h_{v_3}$ is a bounding function for the value that can be achieved when completing this assignment.

4 Best-First Optimization with On-Demand Bounds

When only a few best solutions are required, using dynamic programming to precompute all functions h_{v_i} is wasteful, because typically a large percentage of the bounds are not needed during heuristic search. The cost of pre-processing can be avoided by interleaving dynamic programming and search with each other. Recently, Terrioux and Jégou [12] presented such a hybrid algorithm for the case of branch-and-bound. BTD$_{val}$ [12] augments branch-and-bound search by recording information that is obtained from a tree decomposition. It uses a compatible order to first assign the variables and solve the constraints local to a tree node v_i, and then recursively computes the best extensions of this assignment for the subtrees rooted in v_i. The assignments shared between v_i and its subtrees are stored as goods, avoiding the re-computation of subproblems already solved.

Note that since the goods correspond to tuples of the functions h_{v_i}, this process can also be understood as a partial (or demand-driven) computation of the functions h_{v_i}. In the following, we develop a hybrid of dynamic programming and search for the case of A* search. Given a heuristic, A* search is faster than branch-and-bound because it finds the best solutions in an optimal number of expansion steps [6]. However, A* needs to maintain a larger search tree, which can make it practically infeasible. Viewing hybridization as demand-driven heuristics computation allows to use techniques from heuristic search to significantly reduce the size of the A* search tree. In this section, we present two such techniques. They consist of exploiting a heuristic to limit the number of search nodes expanded, and exploiting preferential independence to limit the number of successor nodes generated during an expansion step. We also present a third extension that does not reduce the size of the search tree, but allows for processing it in a distributed way. The resulting variant of A* search generates the bounding functions h partially and only to an extent that it is actually needed in order to generate a next best solution. We call this approach *best-first search with on-demand bound computation* (BFOB).

The first step to reduce the size of the search tree is to exploit the heuristic h. Note that within a node v_i, BTD$_{val}$ computes only the value of g, that is, it can be viewed as using a heuristic h that is simply equal to the identity. We can find a heuristic that is tighter, but not more difficult to compute, by switching to a dual problem representation that treats the constraints as variables and their tuples as domains. In the dual formulation, Equation 2 simplifies since each constraint appears only once in the tree decomposition, and each constraint is either completely instantiated or not instantiated at all. For example, consider node v_1 of the tree decomposition in Fig. 2, and assume a compatible order that first assigns a tuple to function f_{a1} and then a tuple to function f_{a2}. Using only the identity ($h = \mathbf{1}$) as a heuristic, A* would generate the search tree shown in Fig. 3(a). We improve on this by using a heuristic function $h^{(i)}_{\text{succ}}$ that consists of the next unassigned function f_{i+1} in the order (in the example, $h^{(i)}_{\text{succ}} = f_{a2}$). Note that $h^{(i)} \leq_S h^{(i)}_{\text{succ}} \leq_S \mathbf{1}$. Using the heuristic $h^{(i)}_{\text{succ}}$, the search tree in the example is reduced to the one shown in Fig. 3(b).

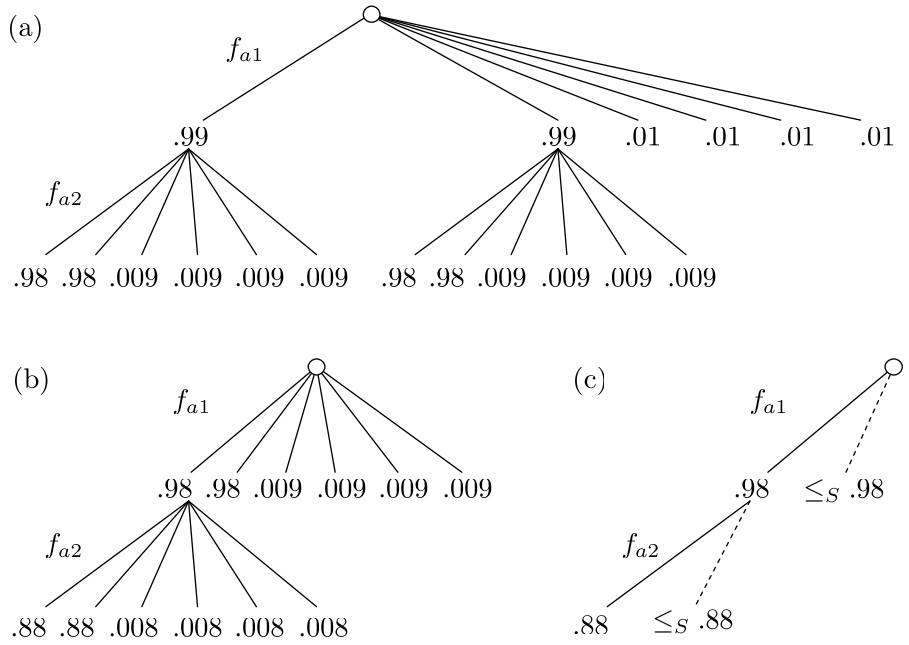

Fig. 3. A* search trees for the node v_1 in Fig. 2, shown with empty heuristic $h = \mathbf{1}$ (a), with heuristic $h^{(i)}_{\text{succ}}$ that takes into account the value of the next assignment (b), and with expansion limited to the best child by exploiting preferential independence (c). The branches of the trees correspond to tuples of the functions f_{a1} and f_{a2} in Table 1.

The second technique limits the number of successor nodes generated when a search node is expanded, by exploiting preferential independence [5, 18]. We first show that an instance of preferential independence holds for c-semirings:

Proposition 1. *If $h_0 \leq_S h_1$ for $h_0, h_1 \in A$, then for $g_0 \in A$, $g_0 \times h_0 \leq_S g_0 \times h_1$.*

Proof. Because $\times$ distributes over $+$, $(g_0 \times h_1) + (g_0 \times h_0) = g_0 \times (h_1 + h_0)$. Because $h_0 \leq_S h_1$, $h_1 + h_0 = h_1$. Thus, $(g_0 \times h_1) + (g_0 \times h_0) = g_0 \times h_1$.

Proposition 1 implies that if the value of a successor node is better than or equal to the values of all its siblings, then all siblings cannot immediately lead to solutions that have a better value. Consequently, for A* it is sufficient to generate this successor node only and to delay the generation of the siblings, rather than generating all possible successors at once (for details, see [18]). This can significantly limit the number of nodes generated at each expansion step. For instance, consider again assigning a tuple to constraint f_{a1} in node v_1. Because the tuples where a_1=G have a value that is better than or equal to the value of all other tuples of f_{a1}, it is sufficient to consider only one of those best tuples and keep a reference to the next best sibling assignment. This is illustrated in Fig. 3(c).

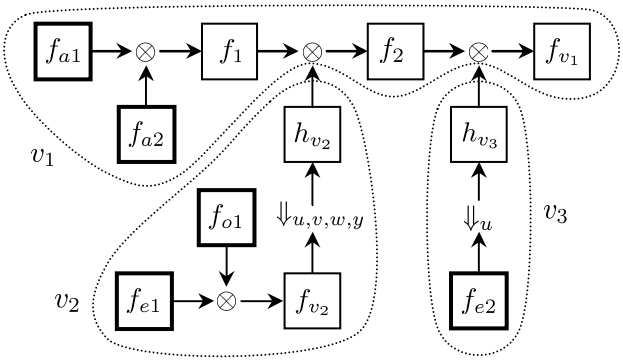

Fig. 4. Computational scheme for the tree decomposition in Fig. 2. The circled fragments correspond to the nodes v_1, v_2 and v_3 of the tree.

The third extension avoids maintaining an explicit search tree for each tree node. This is accomplished by replicating each level in the search tree as a constraint (the constraint consists of the current assignments at this level of the search tree). This does not reduce the size of the search tree (in fact, the replication incurs a higher constant for the space complexity), but it allows for computation to proceed independently not only for each subtree, but also within each tree node. For instance, consider again node v_1 of the tree decomposition in Fig. 2. In order to find a tuple of f_{v1}, the constraints f_{a1}, f_{a2}, h_{v2} and h_{v3} have to be assigned a tuple. Instead of maintaining a search tree with four levels (corresponding to the four constraints), we break it down using an intermediate function f_1 that is the result of combining f_{a1} and f_{a2}, and an intermediate function f_2 that is the result of combining f_1 and h_{v2}. The resulting scheme is illustrated in Fig. 4. In Fig. 4, the bold boxes correspond to given constraints, whereas the other boxes correspond to constraints that need to be computed. The variable order (sequence of functions in the dual representation) is implicitly encoded in this scheme: each constraint operator in the network operates as a consumer of the constraints of its children, and produces a constraint for its parent. We assume two functions producer() and consumer() that return the producing (preceding) and consuming (succeeding) operator of a constraint, respectively. Function producer() returns nil for given constraints at the leaves of the scheme, and function consumer() returns nil for the constraint $f_{v_{root}}$ at the root of the tree.

Our approach can then be understood as best-first, incremental variants of the constraint operators. For example, a best tuple of the function f_{v_1} in Fig. 4 is computed as follows: Consider the best tuple of the function f_{e2}, which is $\langle e_2 = \mathrm{G}, u = 0\rangle$ with value .95 (first tuple of f_{e2} in Table 1). The projection of this tuple on u, which is $\langle u = 0\rangle$ with value .95, is necessarily a best tuple of h_{v_3}. Similarly, a best tuple of f_{a1} can be combined with a best tuple of f_{a2}, for instance the first tuples of f_{a1} and f_{a2} in Table 1. The resulting tuple

```
function BFOB(s, i)
  if (i ≤ length(s)) then
    return s[i]
  end if
  op ← producer(s)
  if (op ≠ nil) then
    case op
      proj: ⟨t, v⟩ ← nextBestProj(op)
      comb: ⟨t, v⟩ ← nextBestComb(op)
    end case
    if (⟨t, v⟩ ≠ nil) then
      append(s,⟨t, v⟩)
      return ⟨t, v⟩
    end if
  end if
  return nil
```

Fig. 5. Function BFOB for best-first search with on-demand bound computation.

$\langle u = 0, v = 0, w = 0, y = 0, a_1 = \mathrm{G}, a_2 = \mathrm{G}\rangle$ is necessarily a best tuple of constraint f_1. This tuple needs to be combined with a tuple of h_{v_2}. A best tuple for h_{v_2} is generated by combining the best tuple of f_{o1} with a best tuple of f_{e1} and projecting the result onto u, v, w, and y, yielding $\langle u = 1, v = 0, w = 0, y = 0\rangle$ with value .90. Since this tuple does not combine with the tuple found for f_1 so far, generation of a next best tuple is triggered for both h_{v_2} and f_1. The next best tuple of h_{v_2} is $\langle u = 0, v = 0, w = 0, y = 1\rangle$ with value .90. This tuple also does not combine with any of the tuples for f_1 generated so far. The process continues until a third tuple for h_{v_2} is generated; for example, by combining the third tuple of f_{e1} in Table 1 with the best tuple of f_{o1}. The resulting tuple $\langle u = 0, v = 0, w = 0, y = 0\rangle$ for h_{v_2} combines with the first tuple that has been generated for f_1 and the tuple in h_{v_3} to a best tuple for f_{v_1}, $\langle u = 0, v = 0, w = 0, y = 0, a_1 = \mathrm{G}, a_2 = \mathrm{G}\rangle$ with value 0.044. Notice that in order to compute this best tuple, large parts of the constraints f_{a1}, f_{a2}, f_{e1}, f_{e2}, and f_{o1} never needed to be visited.

Fig. 5 shows the pseudocode of BFOB. BFOB(s, i) returns the i-th best tuple of a constraint s, or generates it, if necessary, by calling the constraint operator that produces the constraint. Each constraint is represented as a list of pairs $\langle t, v\rangle$, where t is a tuple and $v \in A$. The tuples are listed in decreasing order according to their value v. Function length() returns the length of the list. Function $s[i]$ returns the i-th tuple-value pair of a constraint s, $i \leq$ length(s). Function append() appends a tuple t with value v to the constraint. BFOB() is based on the two functions nextBestProj() and nextBestComb() shown in Fig. 6, that implement best-first variants of the constraint operators $\Downarrow$ and $\otimes$, respectively.

```
function nextBestProj(op)
  while (index(op) ≠ 0) do
    ⟨t, v⟩ ← BFOB(input(op),index(op))
    if (⟨t, v⟩ ≠ nil) then
      t1 ← t ⇓var(output(op))
      index(op) ← index(op) + 1
      // check if result exists
      for each ⟨t2, v2⟩ in output(op) do
        if (t1 = t2) then goto while
        end if
      end for
      // output next best result
      return ⟨t1, v⟩
    else
      index(op) ← 0
    end if
  end while
  return nil


function nextBestComb(op)
  while (queue(op) ≠ ∅) do
    ⟨i, j, v⟩ ← pop(queue(op))
    ⟨t1, v1⟩ ← BFOB(input1(op),i)
    if (⟨t1, v1⟩ ≠ nil) then
      ⟨t2, v2⟩ ← BFOB(input2(op),j)
      if (⟨t2, v2⟩ ≠ nil) then
        t ← t1 ⊗ t2
        if (var(input1(op)) ⊉ var(input2(op))) then
          // create next best sibling w.r.t. input1
          ⟨t1′, v1′⟩ ← BFOB(input1(op),i+1)
          if (⟨t1′, v1′⟩ ≠ nil) then
            push(queue(op),⟨i + 1, j, v1′ × v2⟩)
          end if
        end if
        if (i = 1) then
          // create next best sibling w.r.t. input2
          ⟨t2′, v2′⟩ ← BFOB(input2(op),j+1)
          if (⟨t2′, v2′⟩ ≠ nil) then
            push(queue(op),⟨i, j + 1, v1 × v2′⟩)
          end if
        end if
        // output next best result
        if (t ≠ nil) then
          return ⟨t, v1 × v2⟩
        end if
      end if
    end if
  end while
  return nil
```

Fig. 6. Best-first variants of constraint projection and constraint combination.

Function nextBestProj() in in Fig. 6 consumes an input constraint input(). It takes a next best tuple from this constraint, computes its projection, and then checks whether the resulting tuple already exists on the output constraint output(). If the tuple does not already exist, it is a next best tuple of the output constraint. An index index() is used to keep track of which tuple from the input is processed next.

Function nextBestComb() in in Fig. 6 consumes two input constraints input1() and input2(). The tuples in the input constraints are combined in a best-first manner using A* search as described above. A search queue queue() is used to keep track of which tuples from input1() and input2() are combined next. Each entry in queue() is a triple $\langle i, j, v\rangle$, where i is the index of a tuple $\langle t_1, v_1\rangle$ in input1(), j is the index of a tuple $\langle t_2, v_2\rangle$ in input2(), and $v \in A$ is the heuristic value $h^{(i)}_{\text{succ}} = v_1 \times v_2$.

Function nextBestComb() pops an entry with a best value v from the queue and computes the respective combination of tuples from input1() and input2(). If the result is not empty (that is, the tuples match), then the combination is a next best tuple of the output constraint. A next best sibling of the entry is generated that points to the next entry on stream input1(). For the first tuple of input1(), in addition a next best sibling is generated that points to the next entry on stream input2(). An optimization is possible for the special case where the variables of the constraint of input1() are a superset of the variables of the constraint of input2(). In this case (it is known as semi-join), each tuple of input2() can combine with at most one tuple of input1(). Hence, no next best sibling needs to be generated that points to the next tuple of input1().

Initially, the tuples of the constraints are sorted according to their values. For each constraint projection operator, index() is initially set to 1. For each constraint combination operator, queue() is initially the singleton $\{\langle 1, 1, \mathbf{1}\rangle\}$. The tuples of the function f_{v_1} at the root of the scheme, and thus the solutions of the constraint optimization problem, can then be obtained in best-first order by calling BFOB(f_{v_1},1), BFOB(f_{v_1},2), etc.

Theorem 1 (Correctness). *The algorithm BFOB is sound, complete, and terminates.*

Apart from overhead due to additional data structures, on-demand function computation is not computationally more complex than classical dynamic programming methods for best-first search described in Sec. 3:

Theorem 2 (Complexity). *Let (T, χ, λ) be a tree decomposition, $T = (V, E)$. Let $w = \max_{v_i \in V}(\mid \chi(v_i) \mid) - 1$ be the width of the tree decomposition. Then the algorithm BFOB computes an optimal solution in time $O((\mid F \mid + \mid V \mid)\cdot\exp(w))$ and space $O((\mid V \mid)\cdot\exp(w))$.*

However, the average complexity of on-demand function computation can be much lower if only some best tuples of the resulting function are required.

Table 2. Results for random Max-CSPs, low density networks.

T	C	N	K	BFTC (% time)	BFOB (% time)
4 (25%)	20	15	4	100%	1.4%
8 (50%)	20	15	4	100%	3.2%

Table 3. Results for random Max-CSPs, medium density networks.

T	C	N	K	BFTC (% time)	BFOB (% time)
4 (25%)	15	10	4	100%	4.5%
8 (50%)	15	10	4	100%	14.3%

Table 4. Results for random Max-CSPs, medium to high density networks.

T	C	N	K	BFTC (% time)	BFOB (% time)
4 (25%)	20	10	4	100%	9.7%
8 (50%)	20	10	4	100%	38.8%

5 Experimental Results

We evaluated the performance of BFOB on the task of generating best solutions to random Max-CSP problems. Max-CSP can be formulated as a constraint optimization problem over the c-semiring $(\mathbb{N}_0^+ \cup \infty, \min, +, \infty, 0)$, where the tuples of a constraint $f_j \in F$ have value 0 if the tuple is allowed, and value 1 if the tuple is not allowed. To generate the constraints, we used a binary constraint model with four parameters N, K, C, and T, 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. The tightness of a constraint is the number of tuples having value 1.

We compared the performance of BFOB *relative* to the alternative approach of pre-computing all functions h_{v_i} using dynamic programming pre-processing as described in Sec. 3. We call this alternative algorithm BFTC (for best-first search with tree clustering). BFTC is analogous to the algorithm BFMB described in [13]. Tables 2, 3 and 4 show the results of experiments with three classes of Max-CSP problems, N=15, K=4, C=20 (low density), N=10, K=4, C=15, (medium density), and N=10, K=4, C=20 (medium to high density). In each class, 10 instances where generated for $4 \leq T \leq 8$ and we compared the relative mean runtime of BFOB and BFTC. The comparison does not include the time for computing the tree decomposition of the problem. All experiments were performed using a Pentium 4 CPU and 1 GB of RAM.

Tables 2 to 4 indicate that BFOB leads to significant savings especially when computing best solutions to problems with low constraint tightness and sparse to medium constraint networks. This is consistent with experiments in [13], showing that pre-computing bounding functions is inefficient especially for problems that have many solutions. We are currently working on a comparison of BFOB with BTD$_{val}$ and other algorithms for Max-CSPs to study time and space requirements of dynamic programming with A* search versus branch-and-bound.

6 Related Work and Discussion

As already noted, the algorithm BTD$_{val}$ by Terrioux and Jégou [17] is closest to ours. The approach in [17] is to improve backtracking by recording information (goods) during search. We illustrated how this hybrid approach can also be understood as computing a heuristic using dynamic programming on-demand. An advantage of this perspective is that techniques to approximate heuristics (bounding functions) become applicable in this framework. For instance, Dechter and Rish [8] present a method to decrease the complexity of bound computation by defining an approximate version of dynamic programming called mini-bucket elimination (called mini-clustering [10] for the more general case of tree decompositions). The idea of mini-bucket elimination is to limit the size of the computed functions by restricting their maximum arity to a fixed value z. This is accomplished by partitioning functions $f_1, \ldots, f_k$ that need to be combined into sets $P_1, \ldots, P_m$ called mini-clusters, each having a combined number of variables less than or equal to z. Then the function $(\bigotimes_{i=1}^{k} f_i) \Downarrow_Y$ is bounded by the function $f = \bigotimes_{i=1}^{m} (\bigotimes_{f_j \in P_i} \Downarrow_Y)$ that applies projection early at the level of mini-clusters. The accuracy of the approximation can be controlled by varying the parameter z. The algorithm BFMB(z) in [13] combines mini-clustering and best-first search. Lower values for z lead to loose bounds that are easy to compute, but will guide the search less and therefore necessitate more backtracking in order to find optimal solutions. Kask and Dechter [13] empirically observe an U-shaped performance curve when varying the parameter z, that is, a trade-off between bound accuracy and search. It would be interesting to combine BFOB with approximate bound computation using mini-buckets. This can be accomplished by replacing the scheme of operators and functions (Fig. 4) with an approximate mini-clustering scheme.

A major difference of our approach to the algorithm in [17] is that we use best-first (A*) search instead of branch-and-bound. A* search is faster than branch-and-bound, but it requires more memory. BBMB(z) [13] is a variant of BFMB(z) for branch-and-bound based on bucket trees. BBBT(z) [10] extends BBMB(z) to tree decompositions. Each time a variable needs to be assigned during search, BBBT(z) solves the single-variable optimization problem ($Z = \{x_i\}$) for all unassigned variables. That is, like BFOB and BTD$_{val}$, BBBT(z) interleaves dynamic programming and search. Unlike BFOB and BTD$_{val}$, BBBT(z) can dynamically change the variable order and prune domains during search. However, BBBT(z) does not compute bounds incrementally on-demand, but instead starts a fresh dynamic programming phase at each search node. This can lead to redundant computations, and therefore BBBT(z) and BBMB(z)/BFMB(z) do not dominate each other [10]. Since the algorithm presented in this paper is essentially an improvement of BFMB(z), we expect that BBBT(z) does not dominate BFOB, either. However, variable reordering based on smallest domain size as in BBBT(z) is not possible in BFOB because the values of variables are only partially known. An interesting direction for future work would be to evaluate the impact of the techniques described in Sec. refsec:OnDemandBound within the branch-and-bound search paradigm.

7 Summary and Conclusion

Focusing on leading solutions is an important requirement in many applications. A* search can generate best solutions faster than branch-and-bound search, but needs good heuristics in order to be practically feasible. We presented an algorithm called BFOB that guides A* search using bounds computed using tree decompositions and dynamic programming. BFOB interleaves A* search and dynamic programming to compute bounds on-demand and only to an extent that is necessary in order to generate a next best solution. This hybridization combines the benefits of A* search with the complexity bounds of dynamic programming on trees to generate leading solutions more efficiently than classical methods for best-first search.

References

[1] Babcock, B., et al.: Models and Issues in Data Stream Systems. Proc. ACM Symp. on Principles of Database Systems (PODS) (2002)
[2] Bayardo, R., Miranker, D.: An optimal backtrack algorithm for tree-structured constraint satisfaction problems. Artificial Intelligence **71** (1994) 159–181
[3] Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based Constraint Solving and Optimization. Journal of ACM, **44** (2) (1997) 201–236
[4] Bistarelli, S., et al.: Semiring-based CSPs and Valued CSPs: Frameworks, Properties, and Comparison. Constraints **4** (3) (1999) 199–240
[5] Debreu, C.: Topological methods in cardinal utility theory. In: Mathematical Methods in the Social Sciences, Stanford University Press (1959)
[6] Dechter, R., Pearl, J.: Generalized Best-First Search Strategies and the Optimality of A*. Journal of the ACM **32** (3) (1985) 505–536
[7] Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence **38** (1989) 353–366
[8] Dechter, R., Rish, I.: A scheme for approximating probabilistic inference. Proc. UAI-97 (1997) 132–141
[9] Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113** (1999) 41–85
[10] Dechter, R., Kask, K., Larrosa, J.: A General Scheme for Multiple Lower Bound Computation in Constraint Optimization. Proc. CP-01 (2001)
[11] Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. Artificial Intelligence **124** (2) (2000) 243–282
[12] Jégou, P., Terrioux, C.: Hybrid backtracking bounded by tree-decomposition of constraint networks. Artificial Intelligence **146** (2003) 43–75
[13] Kask, K., Dechter, R.: A General Scheme for Automatic Generation of Search Heuristics from Specification Dependencies. Artificial Intelligence **129** (2001) 91–131
[14] Kask, K., et al.: Unifying Tree-Decomposition Schemes for Automated Reasoning. Technical Report, University of California, Irvine (2001)
[15] de Kleer, J.: Focusing on Probable Diagnoses. Proc. AAAI-91 (1991) 842–848
[16] Schiex, T., Fargier, H., Verfaillie, G.: Valued Constraint Satisfaction Problems: hard and easy problems. Proc. IJCAI-95 (1995) 631–637
[17] Terrioux, C., Jégou, P.: Bounded Backtracking for the Valued Constraint Satisfaction Problems. Proc. CP-03 (2003)
[18] Williams, B., Ragno, R.: Conflict-directed A* and its Role in Model-based Embedded Systems. Journal of Discrete Applied Mathematics, to appear.