On-Demand Bound Computation for Best-First Constraint Optimization

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. Many problems in Artificial Intelligence, such as diagnosis, control, and planning, can be framed as constraint optimization problems where a limited number of leading solutions are needed. An important class of optimization algorithms use A*, a variant of best first search, to guide the search effectively, while employing a heuristic evaluation function that is computed using dynamic programming. A key bottleneck, however, is that significant effort can be wasted precomputing bounds that are not used to generate the leading solutions. This paper introduces a method for solving semi-ring CSPs, based on a variant of A* that generates *on demand*, only those bounds that are specifically required in order to generate a next best solution. On demand bound computation is performed using "lazy", best-first variants of constraint projection and combination operators, and a scheme that coordinates the computation of these operators by exploiting a decomposition of the optimization problem into a tree structure. We demonstrate a significant performance improvement over bound precomputation on randomly generated, semiring-based optimization problems.

1 Introduction

Algorithms for constraint optimization are key to many problems in Artificial Intelligence, such as monitoring, diagnosis, autonomous control, or reconfiguration. An important class of algorithms for constraint optimization finds solutions by searching through the space of possible assignments, guided by a heuristic evaluation function (bound) [16, 13, 11].

Often, the set of solutions is too large to explicitly enumerate all solutions. Instead, for many applications only a limited number of *leading* solutions is required. For instance, in fault diagnosis it might be sufficient to compute the most likely diagnoses that cover most of the probability density space [15, 17]. In planning, it might be sufficient to compute the best plan and a few backup plans in case the best plan cannot be executed.

When only a few solutions are generated, computing heuristics (bounds) for all values prior to the search would be inefficient because only a few of these bounds are typically needed to compute the best solutions. In this paper, we

present a method called *best-first search with on-demand bound computation* (BFOB) that interleaves bound computation and best-first search, such that bounds are computed and assignments are expanded only as required to generate each next best solution. The approach involves a streamed computation analogous to that of distributed database systems [5, 2]. It builds upon best-first variants of the constraint combination and projection operators, and uses a coordination architecture that exploits a decomposition of the optimization problem into a tree structure.

This approach has a complexity that never exceeds those of performing bound computation as a separate pre-processing step, yet it can derive the best solutions faster. The solution improves on previous approaches [13] that compute bounds using a separate pre-processing step, but can also outperform approaches that are based on limited interleaving of bound computation and search [11].

We present the approach in the context of semiring-based constraint optimization problems [3] and tree decompositions [8, 14], a decomposition method that is more general than the bucket elimination framework considered in [13]. Experiments with randomly generated constraint optimization problems indicate dramatic performance gains using on-demand bound function computation.

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. In this case, $\leq_S$ is equal to maximization [4].

Definition 2 (Semiring-based Constraint Optimization Problem). *A semiring-based* 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 $\{0, 1\}$. We assume that variables

x, z, and s are observed to be 1, and variable c is observed to be 0 (therefore, we omit them in the following). 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, but OR gates and XOR gates have a 5% probability of failure. Table 1 shows the resulting constraints for the example, where each tuple is assigned the probability of its corresponding mode.

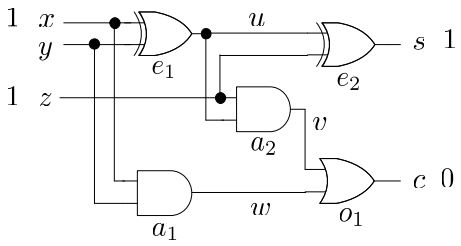

Fig. 1. The full adder example consists of two AND gates, one OR gate and two XOR gates. Variables x, z, s, and c are observed as indicated.

A semi-ring 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 Best-First Optimization using 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 [16, 13, 11]. In the A* framework [7], 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 [10]). *Given a constraint graph G and an ordering on its variables, the* induced graph *G^* 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 [12, 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*

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 $\{u, v, w, x, y, a_1, a_2, e_1, e_2, o_1\}$. The right-hand side of Fig. 2 shows a tree decomposition for the example.

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

f_{a1}: *a*1 *w* *y*				f_{a2}: *a*2 *u* *v*				f_{e1}: *e*1 *u* *y*				f_{e2}: *e*2 *u*			f_{o1}: *o*1 *v* *w*			
G	0	0	.99	G	0	0	.99	G	1	0	.95	G	0	.05	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							

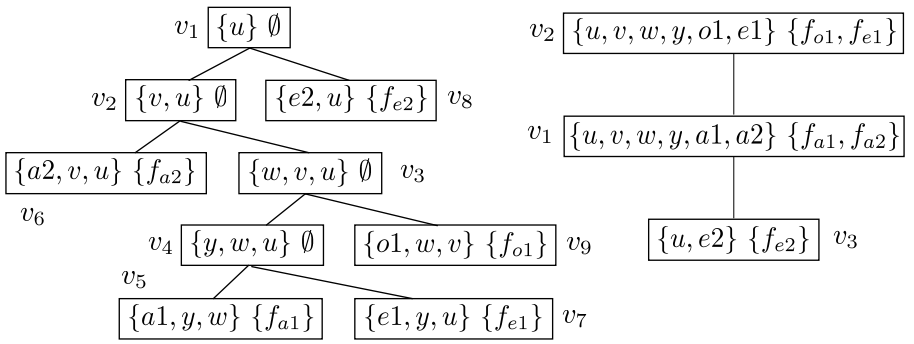

Fig. 2. Bucket tree (*left*) and tree decomposition (*right*) for the example in Fig. 1. The trees show the labels χ and λ for each node.

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 [11]) 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 h_{v_i}. If v_i has a parent node v_j, then v_i sends the constraint $h_{v_i} \Downarrow_{\chi(v_j)}$ to v_j. Assume v_{root} is the root of the tree T. After the dynamic programming algorithm has terminated, $h_{v_{\text{root}}}$ is the solution to the constraint optimization task that consists of COP and the variables of interest $Z = \chi(v_{\text{root}})$.

The functions h_{v_i} computed by the dynamic programming algorithm can be exploited to guide the search for solution assignments to the task where the variables of interest are $Z = X$. In this paper we focus on an A* search strategy that expands partial assignments in best first order on $f = g + h$, where g is the value of the partial assignment, and h is the above evaluation function.

For the case of a bucket tree, Kask and Dechter [13] describe an algorithm that extracts the best assignment by processing the tree top-down, that is, in pre-order. Variables are assigned according to the variable order $\{x_1, \ldots, x_n\}$ 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 of the nodes $c_1, \ldots, c_l$ that are children of $v_1, \ldots, v_i$:

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

Given an extension $x_1 = x_1^0, \ldots, x_i = x_i^0, x_{i+1} = x_{i+1}^0$ of the assignment, then $g^{(i)}(x_1^0, \ldots, x_i^0) \times (h^{(i)} \Downarrow_{x_{i+1}})(x_{i+1}^0)$ is an upper bound (with respect to $\leq_S$) on the value that can be achieved for completing this extended assignment. In addition, if the operator $\times$ is idempotent or has an inverse, then $g^{(i)}$ and $h^{(i)}$ can be incrementally updated when going from variable x_i to x_{i+1} using the recursive scheme defined in [13]. 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 tree decompositions to tree decompositions as follows. Let $p = v_1, \ldots, v_n$ be a pre-order of the tree nodes. The pre-order defines an ordering on groups of variables $G_1, \ldots, G_{|V|} \subseteq X$, by letting

$$G_1 = \chi(v_{\text{root}}),\ G_{i+1} = \chi(v_i) \setminus (G_1 \cup \ldots \cup G_i).$$

For example, for the tree on the right-hand side of Fig. 2, the pre-order v_1, v_2, v_3 induces an ordering on three groups of variables $G_1 = \{u, v, w, y, a1, a2\}$, $G_2 = \{e1, o1\}$, and $G_3 = \{e2\}$.

The principle for deriving bounding functions from bucket trees carries over to tree decompositions, except that the variables are assigned in groups. For example, consider the tree on the right-hand side of Fig. 2 and the case where the variables in the group $G_1 = \{u, v, w, y, a1, a2\}$ have been assigned a value, that is, node v_1 has been traversed. Then $g^{(1)} \otimes h^{(1)}$ with $g^{(1)} = f_{a1} \otimes f_{a2}$ and $h^{(1)} = h_{v_2} \otimes h_{v_3}$ is a bounding function for the values that can be achieved when the assignment is extended by assigning the variables in the group $G_2 = \{e1, o1\}$.

In the following, we develop a demand driven computation of the bounding function h for this general case of tree decompositions.

4 On-Demand Bound Computation

A semiring-based COP will typically have many different solutions (if $Z = X$, the number of solutions is $\mid D_1 \mid \cdot \mid D_2 \mid \cdot \ldots \cdot \mid D_n \mid$). In practice, many of these solutions are uninteresting. When only a few best solutions are required, computing bounding functions for all assignments is wasteful, since typically a large percentage of the bounds are not needed in order to compute the best solutions. The key to capturing this intuition formally is the following monotonicity property of c-semirings, which is an instance of preferential independence [6]:

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 in best-first search, rather than considering all possible expansions of a node, it is sufficient to consider only the expansion with the best value, while keeping a reference to its next best sibling. This is sufficient because all other expansions cannot lead to solutions that have a better value with respect to the order $\leq_S$. The constraint-based A* algorithm in [17] exploits this principle in order to significantly limit the nodes created at each expansion step. In this paper we generalize upon this approach in order to avoid computing bounds on all possible expansions of a node, instead computing only a bound on the best expansion of a node.

Based on the above, the key idea pursued in this paper is to interleave best-first search and the computation of a bounding function h, such that h is computed 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). BFOB exploits the above expansion scheme within the on-demand computation of the evaluation function h itself, as well as during the best-first search phase.

In the following we develop an algorithm for best-first, incremental computation of h using a tree-structured network. The approach is similar to pipelined, on-demand computation in distributed database systems [5, 2]. It consists of developing best-first variants of the constraint combination and projection operators, and connecting them through objects called *streams* [1]. Each constraint is viewed as a stream that generates consistent assignments. Each constraint in the network operates as a consumer of the steams of its children, and produces a stream for its parent. The elements of each stream are constructed only as needed: if a consumer attempts to access an element of its stream that has not yet been constructed, the stream will only perform the work necessary to produce the required element.

Figure 3 shows a coordination network of functions and operators corresponding to the tree decomposition in Fig. 2. Bold boxes correspond to given constraints, whereas the other boxes correspond to constraints that need to be computed. Function h_{v_1} can be computed incrementally from the network by applying the constraint operations only partially, that is, to the relevant subsets of the tuples of the constraints. Consider the best tuple of the function f_{e2}, which is $\langle e2 = G, u = 0\rangle$ 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 $\langle u = 0, v = 0, w = 0, y = 0, a1 = G, a2 = G\rangle$ with value .98 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_{e2}. 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 h_{v_1}, $\langle u = 0, v = 0, w = 0, y = 0, a1 = G, a2 = G\rangle$ 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.

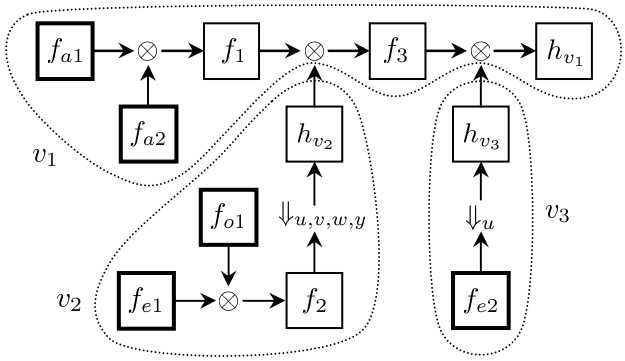

Fig. 3. Computational scheme for the tree decomposition in Fig. 2. The circled fragments correspond to the nodes v_1, v_2 and v_3 of the tree.

We formalize the concept of processing constraints partially, in best-first order, based on a decomposition of each function into a partition of subfunctions, where each subfunction is a set representing either enumerated or unenumerated tuples of the function. A best-first decomposition of a function corresponds to a partition of the function's tuples into a sequence of elements, such that the values of tuples in an element are better (with respect to $\leq_S$) than those of all elements that follow:

Definition 8 (Function Decomposition). *A* best-first decomposition *of a function f is a sequence of subfunctions $f_1, f_2, \ldots, f_k$ such that*

1. $(f_j \Downarrow_\emptyset) \leq_S f_i(t)$ *for all t where* $f_i(t) \neq \mathbf{0}$, $1 \leq i < j \leq k$;
2. $f(t) = f_1(t) + f_2(t) + \ldots + f_k(t)$;
3. $f_i \otimes f_j \equiv \mathbf{0}$ *for* $1 \leq i < j \leq k$.

In the example above and for the scope of this paper, we consider only best-first decompositions that correspond to partitions into single tuples, that is, subfunctions $f_1, f_2, \ldots, f_k$ where each $f_i(t) \neq \mathbf{0}$ for at most one t.

The streams that hold the constraints are simply lists with a number of procedures to manipulate them. List entries e consist of a subfunction $f(e)$ and a value $v(e) \in A$. In the case where the constraint is the input to a combination

operator, entries also have an index $i(e)$ pointing to an entry in the other input's stream.

Conceptually, each stream is a list of entries organized in decreasing order according to their value $v(e)$. Function pop() selects and deletes an entry from the stream with the next best value. Function push() inserts an entry into a stream. The set of streams are pipelined; functions producer() and consumer() return the preceding (producing) and succeeding (consuming) operator of the stream, respectively. producer() returns nil for streams corresponding to given constraints, and consumer() returns nil for the stream corresponding to the constraint at the root of the tree.

Based on the stream object, the two functions nextBestProj() and nextBestComb() shown in Fig. 4 implement best-first variants of the constraint operators $\Downarrow$ and $\otimes$, respectively. Function nextBestProj() pops a subfunction from its input stream and computes its projection. It then checks whether the resulting tuple already exists on the output stream. If it does not already exist, it is a next best tuple and pushed onto the output stream. Function nextBestComb() processes two input streams input1 and input2 and combines the entries in input1 with the entries in input2. Each entry e_i in input1 points to an entry d_j in input2, and its value $v(e_i)$ is an upper bound on the value $v = (f(e_i) \otimes f(d_j)) \Downarrow_\emptyset$ that can be achieved for the combination.

When nextBestComb() combines two entries e_i and d_j, three cases can occur, depending on the value v of the result and the value $v(e_{i+1})$ of the next best entry e_{i+1} in input1:

1. If $v = \mathbf{0}$, then the result can be ignored;
2. If $v(e_{i+1}) <_S v$, then the result $f(e_i) \otimes f(d_j)$ is a tuple of the output stream. However, it is not necessarily a next best tuple, and therefore put on hold in the stream input1 (this is accomplished by setting its pointer $i()$ to zero);
3. If $v \leq_S v(e_{i+1})$, then the result $f(e_i) \otimes f(d_j)$ is a next best tuple of the output and pushed onto the output stream.

In addition, a next best sibling of the e_i is generated that points to the next entry on stream input2. An optimization is possible for the special case where the constraint of input2 is defined over a subset of the variables of the constraint of input1. In this case (known as semi-join), each tuple of input1 can combine with at most one tuple of input2, hence, no next best sibling of e_i needs to be generated.

Functions nextBestProj() and nextBestComb() are based on functions at(), next() and insert(), shown in Fig. 5. The function at(i) accesses the i-th best subfunction of a stream and generates it, if necessary, using function next(). Function next() calls the constraint operator that produces the stream to generate a next best entry for the stream. Function insert() is a helper function that computes the value of entries and pushes them onto streams.

Initially, constraints are decomposed into subfunctions and inserted at the inputs (leafs) of the scheme. The function at the root of the scheme, $h_{v_{\text{root}}}$, can then be constructed incrementally in best-first order by calling at(1), at(2), etc. for the root stream.

```
function nextBestProj(op)
  while at(input(op,1)) ≠ nil do
    e1 ← pop(input(op))
    s ← f(e1) ⇓_{var(output(op))}
    for i ← 1 to | output(op) | do
      if s ≤_S f(at(output(op), i)) then goto while end if
    end for
    insert(output(op),s)
    return true
  end while
  return false


function nextBestComb(op)
  while at(input1(op,1)) ≠ nil do
    e1 ← pop(input1(op))
    if i(e1) = 0 then
      insert(output(op),f(e1))
      return true
    end if
    if at(input2(op),i(e1)) ≠ nil then
      s ← f(e1) ⊗ f(at(input2(op), i(e1)))
      if not(var(input2) ⊆ var(input1)) or (s ≡ 0) then
        insert(input1(op),f(e1),i(e1) + 1)
      end if
      if (s ≡ 0) then goto while end if
      if at(input1(op,1)) ≠ nil then
        if s ⇓ ∅ <_S v(at(input1(op, 1))) then
          push(input1(op),s,s ⇓_∅,0)
          goto while
        end if
      end if
      insert(output(op),s)
      return true
    end if
  end while
  return false
```

Fig. 4. Best-first variants of constraint projection and constraint combination.

```
function at(stream,i)
  if i ≤| stream | then
    return stream[i]
  else
    if next(stream) then
      return at(stream,i)
    end if
  return nil


function next(stream)
  op ← producer(stream)
  if op ≠ nil then
    case op
      proj: return nextBestProj(op)
      comb: return nextBestComb(op)
    end case
  end if
  return false


function insert(stream,f,optional i ← 1)
  op ← consumer(stream)
  if op ≠ nil then
    case op
      comb:
        if at(Input2(op),i) ≠ nil then
          push(stream,f,f ⇓∅ ×v(at(Input2(op), i)), i)
        end if
    end case
  end if
  push(stream,f,f ⇓∅)
  return
```

Fig. 5. Functions for accessing and generating i-th best entries of streams and inserting entries into streams.

The best-first variants of the constraint operators have the same complexity as their counterparts $\Downarrow$ and $\otimes$. Therefore, apart from overhead due to additional data structures, on-demand function computation is not computationally more complex than at-once function computation:

Theorem 1 (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 both for the algorithm in Sec. 3 and the algorithm in Fig. 4 and 5, $h_{v_{\text{root}}}$ can be computed 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.

5 Best-First Optimization with On-Demand Bounds

Figure 6 presents algorithm BFOB (best-first search with on-demand bound computation). It maintains a list of search nodes and expands best elements of this list. Variables are assigned in groups following a pre-order traversal of a tree decomposition as described in Sec. 3. When expanding an element, instead of computing all possible expansions of an element, only the best expansion is computed. Bounds are computed on-demand for the best expansion and its next best sibling using the function at() described in Sec. 4. A more detailed description of the expansion scheme can be found in [17].

Algorithm BFOB
Input: A COP (X, D, F) over a c-semiring.
Output: The optimal assignments to X in best-first order.
Initialize: Compute tree decomposition (T, χ, λ) to obtain scheme of streams and operators. Compute variable groups $G_1, \ldots, G_{|V|}$ from pre-order traversal of T. Decompose constraints into subfunctions and enqueue the subfunctions in streams. Insert entry with empty assignment, node index $i = 1$, expansion index $j = 1$ into list.
Search: While not out of time and list not empty:
Pop entry e with best value from list. If node index $i >\mid V \mid$, then output assignment as next best solution and goto search.
Expand: (Generate best expansion and next best sibling):
- Best expansion: Compute tuple $s1 = \text{at}(\text{stream}(v_i), j)$. If $s1$ is not nil then create new assignment $x_1 = x_1^0, \ldots, x_k = x_k^0$ that extends the assignment of e by assigning the variables in G_i the values of $s1$. Create new entry with the expanded assignment, node index $i + 1$, and expansion index $j = 1$. Put new entry on list with value $g^{(i)}(x_1^0, \ldots, x_k^0) \times h^{(i)} \Downarrow_\emptyset$ as defined in Sec. 3.
- Next best sibling: Compute tuple $s2 = \text{at}(\text{stream}(v_i), j + 1)$. If $s2$ is not nil, create new entry with assignment of e, node index i, and expansion index $j + 1$. Put new entry on list with value $g^{(i-1)}(x_1^0, \ldots, x_{k-|G_i|}^0) \times v(s2) \times h^{(i)} \Downarrow_\emptyset$ as defined in Sec. 3. Goto search.

Fig. 6. Algorithm best-first search with on-demand bound computation.

Table 2. Results for random Max-CSPs, sparse networks (5 instances).

T	C	N	K	BFPB (sec)	BFOB (sec)
256 (25%)	20	20	32	0.15	0.046
512 (50%)	20	20	32	0.18	0.078
768 (75%)	20	20	32	0.19	0.14

Table 3. Results for random Max-CSPs, medium networks (5 instances).

T	C	N	K	BFPB (sec)	BFOB (sec)
256 (25%)	30	20	32	6.4	0.34
512 (50%)	30	20	32	10.1	2.6
768 (75%)	30	20	32	12.3	6.9

6 Experimental Results

We evaluated the performance of BFOB on solving the task of generating a single best solution to Max-CSP problems where $Z = \emptyset$. Experiments were performed using randomly generated 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 random, 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 that have value 1.

We compared BFOB against an alternative algorithm that pre-computes all functions h_{v_i} prior to search (as described in Sec. 3). We call this alternative algorithm BFPB. BFPB is analogous to the algorithm BFMB in [13], extended from bucket trees to tree decompositions. Tables 2 and 3 show the results of experiments with two classes of Max-CSP problems, N=20, K=32, C=20, 256 $\leq T \leq$ 768 and N=20, K=32, C=30, 256 $\leq T \leq$ 768. In each class, the search space is $K^N = 2^{100}$. On each instance, we ran BFOB and BFPB and compared their runtime. The runtime does not include the time for computing a tree decomposition of the problem. All experiments were performed using a Pentium 4 CPU and 1 GB of RAM.

Tables 2 and 3 show that BFOB outperforms BFPB in all cases. The time savings can be dramatic especially for 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 larger problems and a more thorough comparison that includes the memory allocated. The implementation used to derive the results above is limited to problems with a relatively small number of variables and constraints due to an inefficient implementation of tree decomposition.

7 Related Work and Discussion

We presented on-demand bound computation for the case of exact bounds, that is, for the case where search is actually an enumeration (backtrack-free). However, even computing a best tuple of an exact bounding function from a tree decomposition can be exponential in the tree width w. Dechter and Rish [9] present a method to decrease the complexity of dynamic programming by defining an approximate version called mini-bucket elimination (called mini-clustering [11] for the more general case of tree decompositions). The idea 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. BFOB can be combined with our approximate bound computation simply by replacing the scheme of operators and functions (Fig. 3) with an approximate mini-clustering scheme. Some preliminary experimentation indicates that augmenting BFMB(z) with on-demand heuristics computation can turn its U-shaped performance curve into a more J-shaped curve.

In this paper, we focused on best-first search. Branch-and-Bound is an alternative, anytime search algorithm that searches the space of assignments depth-first. It requires less memory than best-first search, but is less efficient in terms of the number of search nodes expanded [7]. BBMB(z) [13] is a variant of BFMB(z) for Branch-and-Bound based on bucket trees. BBBT(z) [13] 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, BBBT(z) interleaves dynamic programming and search. Unlike BFOB, 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 [11]. Since the algorithm presented in this paper is essentially an improvement of BFMB(z), BBBT(z) cannot 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 on-demand bound computation within the Branch-and-Bound search paradigm.

8 Summary and Conclusion

Focusing on leading solutions is an important requirement in many applications. In this paper, we presented a semiring-based constraint optimization algorithm called BFOB that guides best-first search by bounds computed using tree decompositions and dynamic programming. BFOB computes bounds on-demand and only to an extent that it is necessary in order to generate a next best solution. It exploits preferential independence to limit the expansion of search nodes to the best expansion and to compute bounding functions partially for the next best expansion and its next best sibling only. On-demand bound function computation is a key to optimally interleave heuristic computation and search and can lead to considerable performance gains.

References

[1] Abelson H., Sussman, G., Sussman, J.: Structure and Interpretation of Computer Programs. MIT Press (1996)
[2] Babcock, B., et al.: Models and Issues in Data Stream Systems. Proc. ACM Symp. on Principles of Database Systems (PODS) (2002)
[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] Ceri, S., Pelagatti, G.: Distributed Databases. McGraw-Hill (1984)
[6] Debreu, C.: Topological methods in cardinal utility theory. In: Mathematical Methods in the Social Sciences, Stanford University Press (1959)
[7] Dechter, R., Pearl, J.: Generalized Best-First Search Strategies and the Optimality of A*. Journal of the ACM **32** (3) (1985) 505–536
[8] Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence **38** (1989) 353–366
[9] Dechter, R., Rish, I.: A scheme for approximating probabilistic inference. Proc. UAI-97 (1997) 132–141
[10] Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113** (1999) 41–85
[11] Dechter, R., Kask, K., Larrosa, J.: A General Scheme for Multiple Lower Bound Computation in Constraint Optimization. Proc. CP-01 (2001)
[12] Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. Artificial Intelligence **124** (2) (2000) 243–282
[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] Verfaillie, G., Lemaître, M., Schiex, T.: Russian doll search. Proc. AAAI-96 (1996) 181–187
[17] Williams, B., Ragno, R.: Conflict-directed A* and its Role in Model-based Embedded Systems. Journal of Discrete Applied Mathematics, to appear.