Submitted to: Journal of Heuristics, Special Issue on Soft Constraints, 2005

A* Search for Soft Constraints Bounded by Tree Decompositions

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. Some of the most efficient methods for solving soft constraints are based on heuristic search using an evaluation function that is mechanically generated from the problem. However, if only a few best solutions are needed, significant effort can be wasted pre-computing heuristics that are not used during search. Recently, a scheme for depth-first branch-and-bound search has been proposed that avoids the problems of pre-computation by interleaving search with the generation of heuristics using tree decomposition and dynamic programming. In this paper, we extend this idea to A* search, which has the advantage of expanding a minimal number of search nodes to find optimal solutions, and allows to generate solutions in best-first order. The approach uses tree decomposition and dynamic programming to generate only those heuristics that are specifically required to generate a next best solution. The time complexity of the approach is thus optimal among all search algorithms having access to the same heuristics, while its space complexity is bounded by structural parameters of the constraint graph (induced width) in the worst case, and is even lower in the average case.

1 Introduction

Many problems in Artificial Intelligence, such as monitoring, diagnosis, planning, configuration, and autonomous control, can be framed as constraint optimization problems [14]. In order for these applications to meet real-time requirements, an optimal solution should be generated as fast as possible. In order for these applications to be robust, generating one best solution is often not enough; instead, a best solution and possibly a limited number of next best solutions need to be generated. For instance, in fault diagnosis, the goal is to compute the most likely diagnoses that cover most of the probability density space [12, 17]. Likewise, in planning, it might be necessary to compute a least-cost plan and also some backup plans in case the best plan cannot be executed.

A* search [7] allows to generate solutions to constraint optimization problems in best-first order. It uses a lower bound g for the partial assignment made so far, and an optimistic estimate h of the value that can be achieved when extending the assignment to all variables. At each point in the search, A* expands the

assignment with the best combined value of g and h. A* is run-time optimal as it visits a minimal (of all search algorithms having access to the same heuristics) number of search nodes to generate the best solution [2]. However, its memory requirements can make the approach infeasible.

[9, 10] have proposed a scheme for combining A* search with a scheme for computing heuristics from a structural decomposition of the problem into a hierarchy of subproblems (tree). In this case, the memory requirements of A* can be bounded by structural parameters of the constraint graph (limited induced width). The method consists of a pre-computation phase that computes heuristic values using dynamic programming on the tree, and a search phase that guides search using the pre-computed values. However, if only a few best solutions are needed, then the method can waste significant effort pre-computing heuristics that are not used during search.

More recently, [8, 15] have proposed an approach to interleave search with the computation of values using dynamic programming on a tree decomposition of the problem. In the following, we call this approach *demand-driven* heuristics computation in order to distinguish it from pre-computation of heuristics as in [9, 10]. The method thus benefits from the complexity bounds provided by the structural decomposition, while avoiding the problems of pre-computation of heuristics and thus allowing a much smaller average-case complexity. However, the method in [15] is based on depth-first branch-and-bound search and not A* search.

In this paper, we extend the ideas in [9, 10, 15] and present an algorithm for demand-driven heuristics computation for A* search. The approach interleaves A* search with dynamic programming on a tree decomposition of the problem, performing dynamic programming on the tree only to an extend that is specifically required to generate a next best solution. Thus, as in [9, 10], the approach benefits from the optimal time complexity of A* search, while its worst-case memory requirements are bounded by structural parameters of the constraint graph. However, due to the demand-driven computation similar to [8, 15], the average case memory requirements are typically much lower than those of the approach in [9, 10]. A key step of our approach that allows us to combine A* search with demand-driven heuristics computation is a dual problem formulation that treats constraints as variables and tuples of constraints as domain values. We present the approach in the context of valued constraint satisfaction problems (VSCSPs) [14], a general framework for soft constraints with totally ordered preferences. We illustrate the performance of our algorithm with experimental results on randomly generated problems.

2 Valued Constraint Satisfaction Problems

Definition 1 (Valued Constraint Satisfaction Problem [14]). *A* valued constraint satisfaction problem *(VCSP) consists of a tuple* (X, D, F) *with variables* $X = \{x_1, \ldots, x_n\}$*, finite domains* $D = \{d_1, \ldots, d_n\}$*, constraints* $F = \{f_1, \ldots, f_m\}$*, and a valuation structure* $(E, \leq, \oplus, \bot, \top)$*. The constraints* $f_j \in F$ *are*

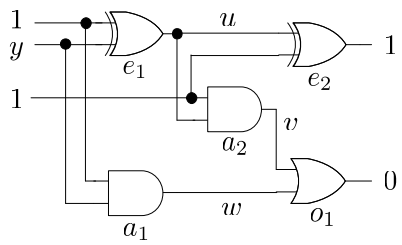

Fig. 1. The full adder example consists of two AND gates, one OR gate, and two XOR gates. Variables x, z, s, and c are observed as indicated

functions defined over var(f_j) $\subseteq X$ and map assignments to var(f_j) to values in E. The set E is totally ordered by $\leq$ with a minimum element $\bot \in E$ and a maximum element $\top \in E$, and $\oplus$ is an associative, commutative, and monotonic operation with identity element $\bot$ and absorbing element $\top$.

The set of valuations E expresses different levels of constraint violation, such that $\bot$ means satisfaction and $\top$ means unacceptable violation. The operation $\oplus$ is used to combine (aggregate) several valuations. A constraint is *hard*, if all its valuations are either $\bot$ or $\top$.

For example, the problem of diagnosing the full adder circuit in Fig. 1 can be framed as a VCSP with variables $X = \{u, v, w, y, a_1, a_2, e_1, e_2, o_1\}$. Variables u to y describe boolean signals and have domain $\{0, 1\}$. Variables a_1 to o_1 describe the mode of each gate, which can either be G (good), or U (unknown failure). The VCSP has five constraints f_{a1}, f_{a2}, f_{e1}, f_{e2}, f_{o1}, one for each gate in the circuit. Each constraint expresses that if the gate is G then it correctly performs its boolean function, and if it is U then it is broken in an unknown way and no assumption is made about its behavior. The valuation structure captures the likelihood of being in a mode, and is $([0, 1], \geq, \cdot, 1, 0)$ (with $\cdot$ being multiplication over the real numbers). We assume Or-gates and Xor-gates have a .95 probability of being G, and a .05 probability of being U, whereas And-gates have a .99 probability of being G and a .01 probability of being U. Table 1 shows the constraints for the example, where each tuple is assigned the probability of its corresponding mode.

Definition 2 (Combination and Projection). *Let f and g be two constraints defined over var(f) and var(g), respectively. Let $t \downarrow_Y$ denote the restriction of an assignment t to a subset Y of its variables. Then,*

1. *The* combination *of f and g, denoted $f \oplus g$, is the constraint over* $\mathrm{var}(f) \cup \mathrm{var}(g)$ *that maps each t to the value $f(t \downarrow_{\mathrm{var}(f)}) \oplus g(t \downarrow_{\mathrm{var}(g)})$;*
2. *The* projection *of f onto a set of variables Y, denoted $f \Downarrow_Y$, is the constraint over $Y \cap \mathrm{var}(f)$ that maps each t to the value* $\min\{f(t_1), f(t_2), \ldots, f(t_k)\}$, *where $t_1, t_2, \ldots, t_k$ are all the assignments for which $t_i \downarrow_Y = t$.*

Given a VCSP and a subset $Z \subseteq X$ of variables of interest, a *solution* is an assignment t with value $((\bigoplus_{j=1}^{m} f_j) \Downarrow_Z)(t)$. In particular, for $Z = \emptyset$, the solution is the value α^* of an assignment with minimum constraint violation, that is, $\alpha^* = (\bigoplus_{j=1}^{m} f_j) \Downarrow_\emptyset$. For the full adder circuit example, α^* is 0.044, corresponding to a single failure either of the Or-gate or of the first Xor-gate.

3 Tree Decomposition

An important class of algorithms for constraint optimization finds solutions by searching through the space of possible assignments, guided by a heuristic evaluation function. In the following, we focus on the approach of automatically generating evaluation functions from solutions to smaller subproblems of the original problem. This idea underlies many of the known most efficient search algorithms, such as branch-and-bound with mini-bucket elimination (BBMB) [10], best-first search with mini-bucket elimination (BFMB) [9], Russian Doll search (RDS) [16], and backtracking with tree decompositions (BTD) [8, 15].

A problem can be broken down into smaller subproblems ("clusters") by decomposing the constraint hypergraph H, which associates a node with each variable x_i, and a hyperedge with the variables $\mathrm{var}(f_j)$ of each constraint f_j.

Definition 3 (Tree Decomposition [6, 11]). *A* tree decomposition *for a problem (X, D, F) is a triple (T, χ, λ), where $T = (V, E)$ is a rooted tree, and χ, λ are labeling functions that associate with each node (cluster) $v_i \in V$ two sets $\chi(v_i) \subseteq X$ and $\lambda(v_i) \subseteq F$, 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).*

In addition, we demand that the constraints appear as close to the root of the tree as possible, that is,

3. *If $var(f_j) \subseteq \chi(v_i)$ and $var(f_j) \not\subseteq \chi(v_k)$ with v_k the parent of v_i, then $f_j \in \lambda(v_i)$.*

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					

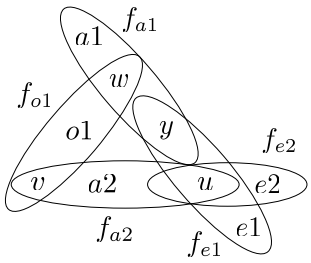

Fig. 2. Hypergraph for the example in Fig. 1.

Fig. 2 shows the hypergraph for the example, and Fig. 3 shows two possible tree decompositions.

The *separator* of a node, denoted $\text{sep}(v_i)$, is the set of variables that v_i shares with its parent node v_j: $\text{sep}(v_i) = \chi(v_i) \cap \chi(v_j)$. For convenience, we define $\text{sep}(v_{\text{root}}) = \emptyset$. Intuitively, $\text{sep}(v_i)$ is the set of variables that connects the subproblem rooted at v_i with the rest of the problem:

Definition 4 (Subproblem). *For a VCSP and a tree decomposition (T, χ, λ), the* subproblem rooted at v_i *is the VCSP that consists of the constraints and variables in v_i and any descendant v_k of v_i in T, with variables of interest sep(v_i).*

For a tree node v_i, we denote solutions to the subproblem rooted at v_i by $h(v_i)$. The subproblem rooted at v_{root} is then identical to the problem of finding α^* for the original COP.

The benefit of a tree decomposition is that each subproblem needs to be solved only once (possibly involving re-using its solutions); the optimal solutions can be obtained from optimal solutions to the subproblems using dynamic programming. Thus, the complexity of constraint solving is reduced to being exponential in the size of the largest cluster only.

4 Generating Search Heuristics from Decompositions using Dynamic Programming

Kask and Dechter [9, 10] show how the solutions $h(v_i)$ to subproblems can be exploited to guide the search for solutions to the original constraint optimization problem with variables of interest X.

In order to exploit the decomposition during search, the variables must be assigned in an order that is compatible with the tree, namely by first assigning the variables in a cluster before assigning the variables in the rest of the subproblems rooted in the cluster. This is called a *compatible order* in [8]. For example, for both trees shown in Fig. 1, a compatible order is $u, v, w, y, a_1, a_2, e_1, e_2, o_1$.

In [9, 10], the approach to guide search by solutions to subproblems is presented for a so-called *bucket trees*. A bucket tree is a specialization of a tree

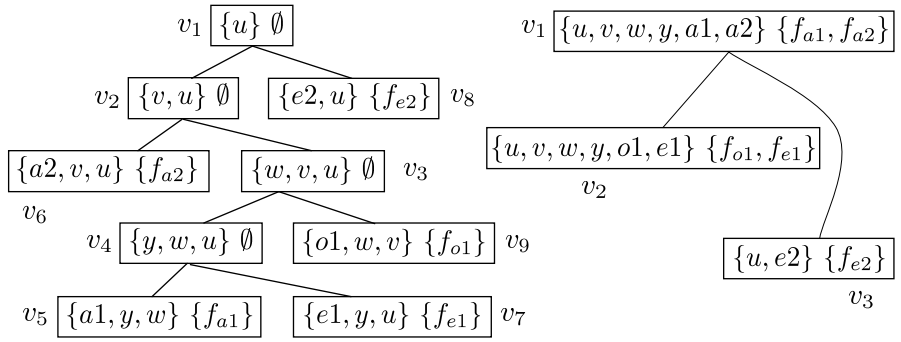

Fig. 3. Bucket tree (*left*) and tree decomposition (*right*) for the example in Fig. 1. The trees show the labels χ and λ for each node.

decomposition where there is one node (cluster) for each variable in the problem, and thus (provided that the clusters of the tree are traversed in a fixed order, such as depth-first left-first), there exists exactly one compatible order.

Assume the compatible variable order of the bucket tree is $x_1 \prec \ldots \prec x_n$. Consider a point in the search where the current assignment is $x_1 \leftarrow x_1^0, \ldots, x_i \leftarrow x_i^0$. 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:

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

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

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

Then $g^{(i)} \oplus h^{(i)}(x_1^0, \ldots, x_i^0)$ is the best value achievable when completing this assignment.

For example, consider the bucket tree on the left-hand side of Fig. 3 for the case 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 the idea of deriving bounding functions for search from bucket trees to tree decompositions. Consider again an assignment $x_1 \leftarrow x_1^0, \ldots, x_i \leftarrow x_i^0$. 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:

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

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 solutions to subproblems of nodes $c_1, \ldots, c_l$ that are children of $v_1, \ldots, V_p(x_i)$, projected on $x_1, \ldots, x_i$:

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

For example, consider the tree on the right-hand side of Fig. 3 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 an (exact) bounding function for the value that can be achieved when completing this assignment.

Kask and Dechter [9, 10] present two search algorithms BBMB and BFMB that exploit this idea of mechanically generating search heuristics from a decomposition of the problem (they also present a way to approximate the heuristics; we will return to this issue in Section 8). The algorithms proceed in two separate phases. First, a pre-computation phase computes the functions $h(v_i)$ (solutions to the subproblems), and then in a second phase, search is guided using the pre-computed values. BBMB uses branch-and-bound search, whereas BFMB uses best-first (A*) search. It is shown in [9] that, given enough memory, the best-first search variant can outperform branch-and-bound by a factor of 5-10.

5 Demand-Driven Heuristics Computation

However, using a separate phase to pre-compute all functions $h(v_i)$ can be wasteful, because typically only a fraction of the heuristic values will be needed to generate a best solution to the problem. The cost of pre-processing can be avoided by interleaving the dynamic programming and the search phases with each other. In the following, we call this approach *demand-driven* heuristics computation in order to distinguish it from pre-computation of heuristics as in [9, 10].

Recently, [8, 15] have proposed an algorithm called BTD (backtracking with tree decompositions) that achieves interleaving of depth-first branch-and-bound search with dynamic programming on tree decompositions. BTD assigns variables along a compatible order, beginning with the variables in $\chi(v_{\mathrm{root}})$. Inside a cluster v_i, it proceeds like classical branch-and-bound, taking into account only the constraints $\lambda(v_i)$ of this cluster. Once all variables in the cluster have been assigned, BTD considers its children (if there are any). Assume v_j is a child of v_i. BTD first checks if the restriction of the current assignment to the variables in $\mathrm{sep}(v_j)$ has previously been computed as a solution to the subproblem rooted at v_j. If so, the value of this solution (called a "good") is retrieved and combined with the value of the current assignment, thus preventing BTD from solving the same subproblem again (called a "forward jump" in the search). Otherwise, BTD solves the subproblem rooted at v_j for the current assignment to $\mathrm{sep}(v_j)$ and the current upper bound, and records the solution as a new good. Its value is combined with the value of the current assignment, and if the result is below the upper bound, BTD proceeds with the next child of v_i.

In fact, the goods that the BTD algorithm computes correspond to a partial construction of the solutions to the subproblems $h(v_i)$. BTD thus benefits from the complexity bounds of dynamic programming on tree decompositions, while avoiding the problem of performing dynamic programming prior to search and thus allowing a much smaller average-case complexity. It has been shown [8] that BTD can outperform BBMB by several orders of magnitude.

6 A* Search with Demand-Driven Heuristics

In the following, we extend the idea of demand-driven heuristics computation from depth-first branch-and-bound search to the case of A* search. A* search [7] uses a lower bound g for the partial assignment made so far, and an optimistic estimate h of the value that can be achieved when extending the assignment to all variables. At each point in the search, A* expands the assignment with the best combined value of g and h. Given the same heuristic, A* search is faster than branch-and-bound because it can be shown to expand an optimal number of search nodes to find the best solution [2].

Combining demand-driven heuristics computation with A* search will thus result in an algorithm with a time complexity that is optimal among all search algorithms having access to the same heuristics. In addition, its space complexity is bounded by dynamic programming and, as we have seen for BTD, typically much lower in the average case.

The main step of extending demand-driven heuristics computation to A* search is to limit the expansion of an A* search node to expanding the next best child only, instead of expanding all children of the node. This in turn allows to limit the computation of heuristics to compute a value for the next best child only, as illustrated in Fig. 4.

Proposition 1. *If $g_0 \leq g_1$ for $g_0, g_1 \in E$, then for $h_0 \in E$, $g_0 \oplus h_0 \leq g_1 \oplus h_0$.*

Proof. Because $\oplus$ distributes over min, $\min(g_0 \oplus h_0, g_1 \oplus h_0) = h_0 \oplus \min(g_0, g_1)$. Because $g_0 \leq g_1$, $\min(g_0, g_1) = g_0$. Thus, $\min(g_0 \oplus h_0, g_1 \oplus h_0) = g_0 \oplus h_0$.

Proposition 1 is an instance of preferential independence [1, 17] and implies that in A* search, if the value of a successor node is better than or equal to the values of all its siblings, then the siblings cannot immediately lead to solutions that have a better value. Consequently, 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 (see Fig. 4). Exploiting preferential independence can significantly limit the number of nodes created at each expansion step in A* search. Here, we exploit the idea in the following way: if the expansion of a search node can be limited to expanding the next best child only, then also the computation of a heuristics can be limited to computing a heuristic value for the next best child only.

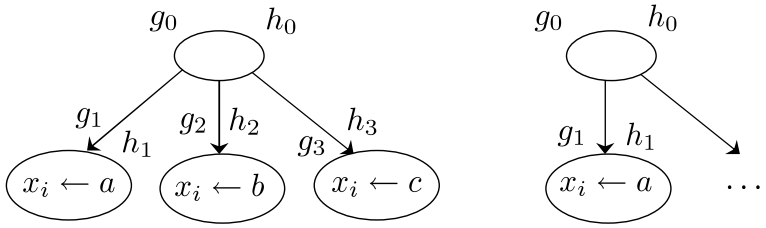

Fig. 4. Instead of expanding all children of a search node at once *(left)*, preferential independence can be exploited to limit expansion to the next best child only *(right)*. Consequently, at each expansion step, it is sufficient to compute a heuristic value for the next best child only.

Dual Problem Reformulation. However, in order to apply this incremental expansion scheme, we need to know in advance which child of the search node has the next best valuation $g \oplus h$. Unfortunately, in a valued CSP it is not easy to establish such an order among the search nodes a priori, because valuations are defined for partial assignments (per constraint) rather than for assignments to individual variables as in the search tree.

One way to solve this problem is to switch to a *dual representation* of the valued CSP, which treats constraints as variables, and tuples of constraints as their possible domain values. In the dual representation, there are only unary soft constraints, and binary hard constraints corresponding to equality of shared variables (see [13]). Thus, in the dual representation, the desired immediate relationship exists between assigning a variable – corresponding to assigning a tuple to a unary constraint – and obtaining a valuation g.

Approximating the Heuristics. However, which child has the best value might then still be affected by the value of the heuristics h; if the heuristic value h is better for some child than for another, then this child might become the next best child, even if its value g is worse.

The approach that we use to solve this problem is to choose an heuristic for A* that is the same for all children of a node. Consequently, if we sort the tuples of the constraints (values in the dual representation) according to their valuations, then we will know in advance the order of the children just based on their g value. In the dual representation, we can find such an heuristic by simply dropping the hard (equality) constraints from consideration in the heuristics; this heuristic is clearly admissible (optimistic), and has the desired property that it is equal for all children of a node.

Distributing the Search Tree. We make one last step that consists of avoiding to maintain an explicit A* search tree for each tree node (cluster). This will not reduce the size of the search tree, but allows for processing it in a

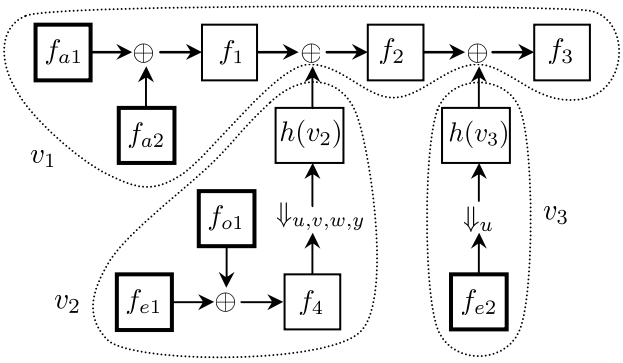

Fig. 5. Computational scheme for the tree decomposition in Fig. 3. The circled fragments correspond to the nodes v_1, v_2 and v_3 of the tree.

distributed way. It 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). Thus, computation can proceed independently not only for each subtree, but also within each tree node (though this is not exploited in our current implementation). For instance, consider again node v_1 of the tree decomposition in Fig. 3. In order to find a tuple for this node, the constraints $f_{a1}, f_{a2}, h(v_2)$ and $h(v_3)$ 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 $h(v_2)$. The resulting scheme is illustrated in Fig. 5. In Fig. 5, 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.

6.1 The BFOB Algorithm

The resulting algorithm can then be understood as best-first, incremental variants of the constraint operators. We illustrate the algorithm by first walking through an example. For instance, a best tuple of the function f_3 in Fig. 5 is computed as follows: Consider the best tuple of the function f_{e2}, which is $\langle e_2 \leftarrow \mathrm{G}, u \leftarrow 0\rangle$ with value .95 (first tuple of f_{e2} in Table 1). The projection of this tuple on u, which is $\langle u \leftarrow 0\rangle$ with value .95, is necessarily a best 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. 6. Function BFOB for best-first search with on-demand bound computation.

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 $\langle u \leftarrow 0, v \leftarrow 0, w \leftarrow 0, y \leftarrow 0, a_1 \leftarrow \mathrm{G}, a_2 \leftarrow \mathrm{G}\rangle$ 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 $\langle u \leftarrow 1, v \leftarrow 0, w \leftarrow 0, y \leftarrow 0\rangle$ 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 $\langle u \leftarrow 0, v \leftarrow 0, w \leftarrow 0, y \leftarrow 1\rangle$ 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 $\langle u \leftarrow 0, v \leftarrow 0, w \leftarrow 0, y \leftarrow 0\rangle$ 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}, $\langle u \leftarrow 0, v \leftarrow 0, w \leftarrow 0, y \leftarrow 0, a_1 \leftarrow \mathrm{G}, a_2 \leftarrow \mathrm{G}\rangle$ 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, and it is not necessary to construct the constraints $h(v_2)$ and $h(v_3)$ completely.

Fig. 6 shows the pseudocode of BFOB (best-first search with on-demand bound computation). 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

```
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. 7. Best-first variants of constraint projection and constraint combination.

a tuple t with value v to the constraint. BFOB() is based on the two functions nextBestProj() and nextBestComb() shown in Fig. 7, that implement best-first variants of the constraint operators $\Downarrow$ and $\oplus$, respectively.

Function nextBestProj() in in Fig. 7 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. 7 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 $v_1 \oplus v_2$ (which is optimistic because it does not take into account the hard constraints).

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 best tuple of the function $f_{v_{\text{root}}}$ at the root of the scheme, and thus the optimal solution of the VCSP, can then be obtained by calling BFOB($f_{v_{\text{root}}}$,1).

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

Demand-driven heuristics computation is not computationally more complex than classical dynamic programming methods for best-first search described in Section 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 demand-driven 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%

7 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 [10]. 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 [10], showing that pre-computing bounding functions is inefficient especially for problems that have many solutions. We are working on a comparison of BFOB with BTD and other algorithms for Max-CSPs to study time and space requirements of dynamic programming with A* search versus branch-and-bound.

8 Related Work and Discussion

Our algorithm is an adaption of the algorithm BTD by Terrioux and Jégou [15] to the case of A* search. The view in [15] is to improve backtracking by recording information (goods) during search; we illustrated how this hybrid approach can be understood as demand-driven computation of a heuristic using dynamic programming. One benefit of this new perspective is that techniques to approximate heuristics (bounding functions) become applicable in this framework. For instance, Dechter and Rish [4] 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 in [4] 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 [9, 10] 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 [10] 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. 5) with an approximate mini-clustering scheme.

A major difference of our approach to the algorithm in [15] 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) [10] is a variant of BFMB(z) for branch-and-bound based on bucket trees. BBBT(z) [5] 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, 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 [5]. 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 extend BFOB such that dynamic programming is not performed on the level of individual tuples, but on sets (blocks) of tuples that have the same valuation.

9 Summary and Conclusion

Focusing on a few best solutions is an important requirement in many applications. A* search can generate solutions to soft constraints in best-first order, and faster than branch-and-bound search as it expands a minimal number of search nodes. However, its memory requirements can make A* search infeasible. We presented an algorithm called BFOB that guides A* search for valued CSPs by 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. It thus combines the benefits of A* search with a space complexity that is bounded by structural parameters of the constraint graph, and is even lower in the average case.

References

[1] Debreu, C.: Topological methods in cardinal utility theory. In: Mathematical Methods in the Social Sciences, Stanford University Press (1959)
[2] Dechter, R., Pearl, J.: Generalized Best-First Search Strategies and the Optimality of A*. Journal of the ACM **32** (3) (1985) 505–536
[3] Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence **38** (1989) 353–366
[4] Dechter, R., Rish, I.: A scheme for approximating probabilistic inference. Proc. UAI-97 (1997) 132–141
[5] Dechter, R., Kask, K., Larrosa, J.: A General Scheme for Multiple Lower Bound Computation in Constraint Optimization. Proc. CP-01 (2001)
[6] Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. Artificial Intelligence **124** (2) (2000) 243–282
[7] Hart, P. E., Nilsson, N. J., and Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Sys. Sci. Cybern. **SSC-4** (2) (1968) 100–107.
[8] Jégou, P., Terrioux, C.: Hybrid backtracking bounded by tree-decomposition of constraint networks. Artificial Intelligence **146** (2003) 43–75
[9] Kask, K., and Dechter, R.: Mini-Bucket Heuristics for Improved Search. Proceedings of Uncertainty in AI (UAI-99) (1999)
[10] Kask, K., Dechter, R.: A General Scheme for Automatic Generation of Search Heuristics from Specification Dependencies. Artificial Intelligence **129** (2001) 91–131
[11] Kask, K., et al.: Unifying Tree-Decomposition Schemes for Automated Reasoning. Technical Report, University of California, Irvine (2001)
[12] de Kleer, J.: Focusing on Probable Diagnoses. Proc. AAAI-91 (1991) 842–848
[13] Larrosa, J., Dechter, R.: On the Dual Representation of Non-binary Semiring-based CSPs. Working papers of the Soft Constraints Workshop (SOFT-00), Singapore (2000)
[14] Schiex, T., Fargier, H., Verfaillie, G.: Valued Constraint Satisfaction Problems: hard and easy problems. Proc. IJCAI-95 (1995) 631–637
[15] Terrioux, C., Jégou, P.: Bounded Backtracking for the Valued Constraint Satisfaction Problems. Proc. CP-03 (2003)

[16] Verfaillie, G., Lemaitre, M., and Schiex, T.: Russian Doll Search for Solving Constraint Optimization Problems. In Proc. of the 13th National Conference on Artificial Intelligence (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.