Exploiting Decomposition in Constraint Optimization Problems

Matthew Kitching and Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada.
[kitching|fbacchus]@cs.toronto.edu

Abstract. Decomposition is a powerful technique for reducing the size of a backtracking search tree. However, when solving constraint optimization problems (COP's) the standard technique of invoking a separate recursion to solve each independent component can significantly reduce the strength of the bounds that can be applied when using branch and bound techniques. In this paper we present a new search algorithm that can obtain many of the computational benefits of decomposition without having to resort to separate recursions. That is, the algorithm explores a standard OR tree not an AND-OR tree. In this way incremental information gathered from any component can be immediately applied to improve the bounding information for all of the other components. We also discuss how caching and local propagation can be combined with our approach and finally test our methods empirically to verify their potential.

1 Introduction

In this paper we investigate the use of decomposition during search to aid in solving **Constraint Optimization Problems (COPs)**. In particular, we are interested in COPs whose objective function is decomposed into a sum of sub-objectives. This means that when variables are instantiated during search the COP can split into independent components that can be solved separately.

Since backtracking's worst case time complexity is exponential in the number of variables, decomposition into independent components can yield an exponential speedup when applied recursively. More precisely, decomposition can reduce the worst case time complexity from $2^{O(n)}$ to $n^{O(1)}2^{O(w)}$ where n is the number of variables and w is the tree-width of the constraint graph, see, e.g., [7, 1].

The downside to using decomposition in COPs is that it can reduce the effectiveness of the bounding techniques that are essential for solving COPs. In particular, the standard method for exploiting decomposition during search is to invoke a separate recursion for each independent component generated during the search, yielding an AND/OR search tree [7, 14, 10, 8, 12, 9]. Unfortunately, the bounds that can be employed in these separate recursions can be quite weak causing inefficiencies in the search.

In this paper we present an algorithm that is able to exploit decomposition in a standard backtracking search tree (an OR tree). Our algorithm has complete freedom in its variable ordering and is able to switch between working on different components while retaining the space efficiency of depth-first search. In this way incremental information gathered from any component can be used to improve the bounding information for all

of the other components. This can often mean that an entire collection of components can be rejected without ever having to solve any of them to optimality. In addition we discuss how local propagation (soft-arc consistency) [6] can be employed in conjunction with our algorithm, and demonstrate how a fixed tree-decomposition can be flexibly utilized to improve decomposition without having it impose excessive restrictions on the variable ordering.

In the sequel we first present some background on branch and bound and AND/OR search with bounding. Our new algorithm is then presented and some of its properties illustrated. After a discussion of local propagation and our technique for flexibly exploiting a fixed tree-decomposition, we present empirical results demonstrating the potential of our approach.

2 Background

A COP, $\mathcal{C}$, is specified by a tuple $\langle \mathit{Vars}, \mathit{Dom}, \mathit{Cons}, \mathit{Obj} \rangle$, where Vars is a set of variables, for each $V \in \mathit{Vars}$, $\mathit{Dom}[V]$ is a domain for V, Cons is a set of constraints, and Obj is an objective function mapping every complete assignment to Vars to a real value. A *solution* of $\mathcal{C}$ is a complete set of assignments to Vars that *minimizes* Obj and satisfies the constraints in Cons.

The techniques we discuss in this paper are effective on COPs whose constraints and objective function are *decomposable*. In particular, we require that Obj be decomposed into a sum of sub-objectives o_i such that: (1) each o_i (and each constraint in Cons) depends on only a proper subset of the variables in Vars, denoted by $\mathit{scope}(o_i)$; (2) each o_i maps assignments to the variables in $\mathit{scope}(o_i)$ to a real value; and (3) on any complete assignment $\mathcal{A}$ the total objective is the sum of the sub-objectives, $\mathit{Obj}(\mathcal{A}) = \sum_i o_i(\mathcal{A})$.[1]

The constraints in Cons can be treated as additional sub-objectives that map satisfying assignments to 0 and violating assignments to ∞. Thus the problem can be reformulated with a single unified objective $\mathit{Obj} = \sum_i o_i + \sum_{c_j \in \mathit{Cons}} c_j$, and we simply need to minimize this unified objective. Hence, we regard a COP as being defined by a tuple $\langle \mathit{Vars}, \mathit{Dom}, \mathit{Objs} \rangle$ where Objs includes both the original objective sub-functions and the hard constraints. We use the term **objectives** to denote the sub-objectives in Objs.

Let $\mathcal{A}$ be any set of assignments to some of the variables of Vars: we use $\mathit{varsOf}(\mathcal{A})$ to denote the set of variables assigned by $\mathcal{A}$; $\mathit{cost}(\mathcal{A}, \mathcal{C})$ to denote the sum of the costs of all objectives in the COP $\mathcal{C}$ that are fully instantiated by $\mathcal{A}$; and $\mathit{mincost}(\mathcal{C})$ to denote $\mathit{cost}(\mathcal{A}, \mathcal{C})$ of any solution $\mathcal{A}$ to $\mathcal{C}$ (i.e., the optimal objective value achievable in $\mathcal{C}$).

A set of assignments $\mathcal{A}$ reduces the original COP $\mathcal{C}$ to a smaller COP $\mathcal{C}|_{\mathcal{A}}$ whose variables are the variables of $\mathcal{C}$ not assigned in $\mathcal{A}$ ($\mathcal{C}.\mathit{Vars} - \mathit{varsOf}(\mathcal{A})$), and whose objectives are those that contain at least one unassigned variable and are obtained by restricting the original objectives of $\mathcal{C}$ by $\mathcal{A}$. That is, for any objective $o_i \in \mathit{Objs}$ if $\mathit{scope}(o_i) \not\subseteq \mathit{varsOf}(\mathcal{A})$, then the reduction of o_i by $\mathcal{A}$, $o_i|_{\mathcal{A}}$, is the new objective

[1] If the objective function or constraints are not decomposable (and they can often be reformulated in a decomposed form), our techniques will still correct solve the COP, but no computational advantage will be gained from decomposition.

Algorithm 1: Branch and Bound

1 **BB** $(\mathcal{C}, \text{UB})$
  /* *Return bounds* $(\mathcal{C}.lb, \mathcal{C}.ub)$ *on* $mincost(\mathcal{C})$. *If* $mincost(\mathcal{C}) < \text{UB}$, *then return exact bounds* $\mathcal{C}.lb = mincost(\mathcal{C}) = \mathcal{C}.ub$. *Else return bounds such that* $\text{UB} \leq \mathcal{C}.lb \leq mincost(\mathcal{C}) \leq \mathcal{C}.ub$. */
2 **begin**
3   $(\mathcal{C}.lb, \mathcal{C}.ub) = getBounds(\mathcal{C})$
4   **if** $(\mathcal{C}.lb < \text{UB} \wedge \mathcal{C}.lb \neq \mathcal{C}.ub)$ **then**
5   **choose** (a variable $V \in \mathcal{C}.Vars$)
6   **foreach** $d \in Dom[V]$ **do**
7       $\text{UB} = \min(\text{UB}, \mathcal{C}.ub)$
8       $\Delta^d = cost(V = d, \mathcal{C})$
9       $(lb^d, ub^d) =$ **BB** $(\mathcal{C}|_{V=d}, \text{UB} - \Delta^d)$
10       $\mathcal{C}.ub = \min(\mathcal{C}.ub, ub^d + \Delta^d)$
11     $\mathcal{C}.lb = \max(\mathcal{C}.lb, \text{MIN}_{d \in Dom[V]}\ lb^d + \Delta^d)$
12   **return** $(\mathcal{C}.lb, \mathcal{C}.ub)$
13 **end**

function with $scope(o_i|_{\mathcal{A}}) = scope(o_i) - varsOf(\mathcal{A})$ and on any set of assignments α to the variables in $scope(o_i|_{\mathcal{A}})$ we have that $o_i|_{\mathcal{A}}(\alpha) = o_i(\mathcal{A} \cup \alpha)$.

Branch and Bound: (Alg. 1) is a standard technique for solving COPs using backtracking search. It works by building up partial variable assignments in a depth-first manner using bounding to prune the search space. Each recursion is passed a COP $\mathcal{C}$ (a reduction of the original COP by the current set of assignments) and an upper bound UB. It tries to compute $mincost(\mathcal{C})$, subject to the condition that it can abort its computation as soon as it can conclude that $mincost(\mathcal{C}) \geq \text{UB}$.

The computation starts with obtaining valid bounds on $mincost(\mathcal{C})$ (various approximations can be used). If it is possible that $mincost(\mathcal{C}) < \text{UB}$ (i.e., $\mathcal{C}.lb < \text{UB}$) and $mincost(\mathcal{C})$ is not already known (i.e., $\mathcal{C}.lb \neq \mathcal{C}.ub$), then the computation of $mincost(\mathcal{C})$ can proceed. For any variable $V \in \mathcal{C}.Vars$ we know that $mincost(\mathcal{C})$ must be achieved by assigning V one of its values, and we can try each of these values in turn. The minimal cost for $\mathcal{C}$ under the assignment $V = d$ is the sum of Δ^d, the cost contributed by any objectives of $\mathcal{C}$ that are fully instantiated by $V = d$ (line 8), and $mincost(\mathcal{C}|_{V=d})$. Hence, to achieve a total cost for $\mathcal{C}$ of less than UB under $V = d$, we must achieve a cost less than $\text{UB} - \Delta^d$ for $\mathcal{C}|_{V=d}$ (line 9). After the recursive call we know that $mincost(prob)$ can be no greater than the returned ub^d plus Δ^d, so we can reset $\mathcal{C}.ub$ to this value if it provides a tighter bound. We can also update the desired bound for the next value to be the minimum of what was already required, UB, and the current upper bound for $\mathcal{C}$, $\mathcal{C}.ub$ (line 7). That is, we force the rest of the search to achieve an even better value for $\mathcal{C}$. After trying all values, we know that both the initially estimated lower bound, $\mathcal{C}.lb$, and the minimum of the lower bounds, $lb^d + \Delta^d$, achieved under the different values of V are valid lower bounds for $\mathcal{C}$. Hence, we can use the tightest (maximum) of these as a new lower bound $\mathcal{C}.lb$. Note that when all variables have been assigned the recursion must stop. In particular, the passed $\mathcal{C}$ will be the empty COP and will have exact bounds $\mathcal{C}.lb = 0 = \mathcal{C}.ub$.

Branch and Bound with Decomposition: (Alg. 2) A more recent technique used in solving COPs (and other types of constraint problems) is *search with decomposition* (or

Algorithm 2: AND-OR Decomposition with Branch And Bound

```
1  AND-OR-Decomp (κ, UB)
   /* On entry (κ.lb, κ.ub) must be valid bounds on mincost(κ). If mincost(κ) < UB, then compute
      exact bounds κ.lb = mincost(κ) = κ.ub. Else compute bounds such that
      UB ≤ κ.lb ≤ mincost(κ) ≤ κ.ub.                                              */
2  begin
3     if (κ.lb < UB ∧ κ.lb ≠ κ.ub) then
4        choose (a variable V ∈ κ.Vars)
5        foreach d ∈ Dom[V] do
6           UB = min(UB, κ.ub)
7           Δ^d = cost(V = d, κ)
            // Start new Processing for decomposition.
8           K^d = toComponents(κ|_{V=d})
9           foreach κ^d ∈ K^d do
10             (κ^d.lb, κ^d.ub) = getBounds(κ^d)
11          foreach κ^d ∈ K^d while (Σ_{κ^d ∈ K^d} κ^d.lb < UB − Δ^d) do
12             UB_{κ^d} = UB − Δ^d − Σ_{κ' ∈ K ∧ κ' ≠ κ^d} κ'.lb
13             AND-OR-Decomp (κ^d, UB_{κ^d})
14          (lb^d, ub^d) = Σ_{κ^d ∈ K}(κ^d.lb, κ^d.ub)
            // End new Processing for decomposition.
15          κ.ub = min(κ.ub, ub^d + Δ^d)
16       κ.lb = max(κ.lb, MIN_{d ∈ Dom[V]} lb^d + Δ^d)
17 end
```

AND/OR search) [7, 14, 10, 8]. As variable assignments are made during backtracking search, the COP can become separated into smaller independent COPs called **components**. These components are COPs that share no variables and hence they can be solved independently of each other. For example, if $\mathcal{C}$ has the objectives $o_1(V_1, V_2, V_3)$ and $o_3(V_3, V_4, V_5)$ then the assignment $V_3 = d$ will split $\mathcal{C}$ into two components, the first containing the variables V_1 and V_2, and the objective $o_1|_{V_3=d}$ while the second contains the variables V_4 and V_5, and the objective $o_2|_{V_3=d}$. Setting the variables of one component has no effect on the other. AND/OR search works by invoking a separate recursion for each component κ generated during search.

In Alg. 2 components are represented by a data structure κ that is created and destroyed as Alg. 2 performs its search. κ is defined by some subset of the objectives of the original input problem, $\kappa.Objs$, that have been reduced by some set of assignments $\kappa.\mathcal{A}$ sufficient to disconnect these objectives from the rest of the problem. The variables of κ, $\kappa.Var$, are all of the unassigned variables of these objectives (thus $varsOf(\kappa.\mathcal{A}) \cup \kappa.Vars = \bigcup_{o \in \kappa.Objs} scope(o)$). Also note that κ is a COP so we can evaluate $cost(\mathcal{A}', \kappa)$ for any set of assignments $\mathcal{A}'$. κ also contains fields $\kappa.lb$ and $\kappa.ub$ used to store bounds on $mincost(\kappa)$.

The search starts with the call $\kappa = \mathcal{C}$ and with $\kappa.lb \leq mincost(\kappa) \leq \kappa.ub$, i.e., a request to solve the original problem with valid bounds on the optimal cost. Each recursion solves a single component κ created by the current set of assignments. To solve the component κ we try all values of one of its variables V, reducing κ by each possible assignment. The reduced component, $\kappa|_{V=d}$, is first separated into a set of sub-components $\mathcal{K}^d$ and a data-structure κ_d is created for each of these sub-components

(line 8). Each sub-component is solved independently (line 13). We know that in order to achieve a total cost of less than UB for κ under the assignment $V = d$, the sum of the lower bounds over all components in $\mathcal{K}$ must be less than $\text{UB} - \Delta^d$ (where Δ^d is the immediate cost of making the assignment $V = d$). Thus each sub-component $\kappa^d \in \mathcal{K}$ must achieve a value of no greater than $\text{UB} - \Delta^d$ minus the sum of the lower bounds of all of the *other* sub-components in $\mathcal{K}$ (line 12). Since each recursive call updates the lower bound of a sub-component in $\mathcal{K}$, we can abort the solving of these sub-components whenever the sum of their lower bounds exceeds $\text{UB} - \Delta^d$ (line 11).

Once we have finished with the value $V = d$ all of the data structures in $\mathcal{K}^d$ can be deleted—so that the space requirements of the algorithm remain polynomial. On the other hand, during search with decomposition the same component can be encountered many times. Thus it is natural to cache the computed bounds for these components so that when they are encountered again we can use these better bounds to optimize the next attempt at solving the component. Cache look up can occur inside of the function $getBounds$ (line 10) where the better bounds (perhaps exact bounds) stored in the cache can be retrieved.

3 Decomposition without Separate Recursions

Although the above use of decomposition with bounding gains computational advantage from breaking the problem into independent sub-problems, it suffers from a weakening of the bounding information it can utilize.

Consider solving a component κ under $\text{UB} = 100$. Say that we branch on variable V making the assignment $V = d$, and that this adds zero to the cost ($\Delta^d = 0$) while breaking κ into five components $\kappa_1, \ldots, \kappa_5$. If $mincost(\kappa_i) = 25$ this value for V must eventually be rejected as under $V = d$, κ can only achieve a minimal cost of 125. Say further that for each of the κ_i, our estimated lower bounds, $\kappa_i.lb$, is 10. We can see that the upper bound applied when solving κ_1 will be $100 - 4 * 10 = 60$, and the search will be forced to solve κ_1 to optimality. This will update $\kappa_1.lb$ to 25, and yield an upper bound of $100 - (3 * 10 + 25) = 45$ for solving κ_2. Thus the search will also be forced to solve κ_2 to optimality. The bound for κ_3 will then be 30, and κ_3 must be solved to optimality. The bound for κ_4 will be 15 and now the search can terminate before solving κ_4, after which $\sum_i \kappa_i.lb > 100$ and we can reject $V = d$. Computing the optimal value for κ_1–κ_3 can be very expensive, and it could be that some much shallower search of all of the components could have served to move their lower bounds to 20 or higher so that $V = d$ could be rejected without having to solve any of them to optimality. Thus we see that although we are solving simpler problems, the bounds we can exploit in these problems are weaker.

The key contribution of this paper is to demonstrate how the computational benefits of decomposition can be obtained without having to perform separate recursions for each component. Instead our method exploits the ideas originally presented in [1] for counting problems where the benefits of decomposition are obtained in a regular backtracking search tree (OR-tree). We extend the ideas of [1] in a non-trivial way so that bounding can be exploited.

Algorithm 3: Decomposition and Bounding in a Standard Backtracking Tree

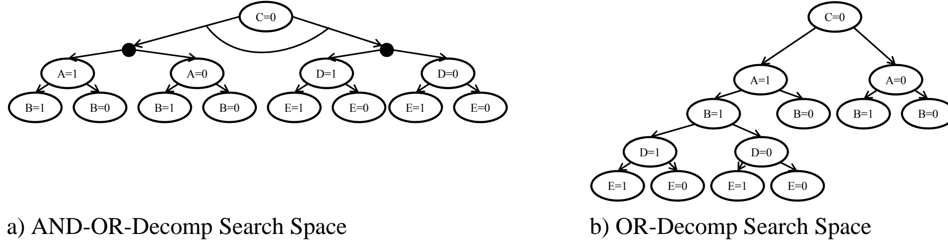
```
1  OR-Decomp (𝒦, UB)
   /* On entry each κ ∈ 𝒦 must have valid bounds (κ.lb, κ.ub). If Σ_{κ∈𝒦} mincost(κ) < UB
      then compute exact bounds for every κ ∈ 𝒦, κ.lb = mincost(κ) = κ.ub. Else
      compute valid bounds on the components in 𝒦 such that UB ≤ Σ_{κ∈𝒦} κ.lb.      */
2  begin
3      if (Σ_{κ∈𝒦} κ.lb < UB ∧ Σ_{κ∈𝒦} κ.lb ≠ Σ_{κ∈𝒦} κ.ub) then
4          choose (a component τ ∈ 𝒦 with τ.lb ≠ τ.ub and a variable V ∈ τ.Vars)
5          AddConstraint(τ, τ.ub)
6          foreach d ∈ Dom[V] while Σ_{κ∈𝒦} κ.lb < UB do
7              UB = min(UB, Σ_{κ∈𝒦} κ.ub)
8              Δ^d = cost(V = d, τ)
9              𝒦^d = toComponents(τ|_{V=d})
10             foreach κ^d ∈ 𝒦^d do
11                 (κ^d.lb, κ^d.ub) = getBounds(κ^d)
12             𝒦' = (𝒦 − τ ∪ 𝒦^d)
13             OR-Decomp (𝒦', UB − Δ^d)
14             (lb^d, ub^d) = Σ_{κ^d∈𝒦^d} (κ^d.lb, κ^d.ub)
15             τ.ub = min(τ.ub, ub^d + Δ^d)
16         RemoveConstraint(τ, τ.ub)
17         τ.lb = max(τ.lb, MIN_{d∈Dom[V]} lb^d + Δ^d)
18 end
```

Our new algorithm is shown in Alg. 3. Like Alg. 1 the algorithm takes as input the entire remaining problem. However, instead of regarding the input as being a single reduced COP ($\mathcal{C}$), the input has been broken up into a set of components $\mathcal{K}$. The aim is to solve all of the components in $\mathcal{K}$ (compute bounds such that $\forall\kappa \in \mathcal{K} : \kappa.lb = \kappa.ub$ or equivalently $\sum_{\kappa\in\mathcal{K}} \kappa.lb = \sum_{\kappa\in\mathcal{K}} \kappa.ub$), subject to the condition that we can give up on the computation once we have concluded that the combined cost of these components is greater than or equal to the passed upper bound UB (i.e., when $\sum_{\kappa\in\mathcal{K}} \kappa.lb \geq \text{UB}$).

If neither of these conditions have been met, some unassigned variable is chosen, and all of its values are tried. On each instantiation the component, τ, containing V might be split up into a new collection of components. These replace τ in the recursive call (line 12), and as in Alg. 1 the upper bound is updated to account for the cost of making the assignment $V = d$. Note that unlike Alg. 2 we pass all of the remaining components to the recursive call (line 13)—we do not recurse just on a single component. Thus the sub-tree search below this invocation can choose to branch on variables from any component in any order—it is not constrained to branch only on the remaining variables of a single passed component as in Alg. 2.

On return from the search below the newly generated components in $\mathcal{K}^d$ will potentially have had their bounds updated, and we can update the upper bound of τ (line 15). After trying all of the values for V we can update the lower bound of τ (line 17).

Note that in the search below the other components in $\mathcal{K}$ can be branched on, and can have their bounds updated. Thus we may obtain sufficient information to abort the for loop before trying all of the values of V. This motivates the **while** test during the for



a) AND-OR-Decomp Search Space b) OR-Decomp Search Space

Fig. 1. Search Space of AND/OR decomposition and OR decomposition

loop (line 6). Note also that if any of these components are solved, i.e., if exact bounds on their value are computed, we will never branch on them again: the branch variable must be from an unsolved component (line 4). That is, if under V's first value we solve some component $\kappa' \in \mathcal{K}$, then in the sub-trees generated by all of V's other values we will never branch on any of the variables of κ' again. This is where decomposition is exploited. If all of the components of $\mathcal{K}$ beside τ are solved (which will occur if $\text{UB} - \Delta^d > \sum_{\kappa \in \mathcal{K} \wedge \kappa \neq \tau} mincost(\kappa)$) then Alg. 3 will obtain all of the computational benefits of decomposition.

Example 1. Consider a COP $\mathcal{C}$ with two objectives $o_1(A, B, C) = A + B + C$ and $o_2(C, D, E) = C + D + E$ and where all of the variables have domain $\{0, 1\}$. Thus $mincost(\mathcal{C}) = 0$ is obtained when all variables have been set to zero. Also suppose that $getBounds$ always returns the minimum and maximum values for the remaining reduced sub-COP. Say that we first branch on $C = 0$ which splits the problem into two components $\kappa_1 = \{o_1(A, B, C = 0)\}$ and $\kappa_2 = \{o_2(C = 0, D, E)$.

If Alg. 1 is used always trying the value 1 before 0, it can be demonstrated that the search below $C = 0$ will attempt 20 variable assignments. In contrast the search by Alg. 2, shown in Fig 1.a is smaller attempting only 12 assignments. It is able to detect that the problem consists of two independent components and solve them independently. Alg. 3, shown in Fig 1.b also searches only 12 nodes. It also exploits decomposition but in a different search tree. In particular, under the left most instantiation of the variables A and B, κ_2 is solved exactly. Hence, the search need never branch on D and E again until it tries a different value for C.

Example 2. However, sometimes bounding in Alg. 3 can interfere with independently solving the components in $\mathcal{K}$. Say that $\mathcal{K}$ contains two components κ_1 and κ_2, where κ_1 contains only a single unassigned variable V that has values $\{a, b, c\}$. If Alg. 3 first branches on V, then after each value for V it will attempt to solve $\mathcal{K}' = \{\kappa_2\}$ ($\mathcal{K}^d$ will be a empty set of components since V is κ_1's final value). Dependent on Δ^d we will be trying to solve κ_2 under different, perhaps too stringent bounds. For example, if $mincost(\kappa_2) = 10$, $\text{UB} = 15$, and $\Delta^a = 7$, $\Delta^b = 6$, and $\Delta^c = 3$, the attempts to solve κ_2 under both $V = a$ and $V = b$ will fail (although we will increase $\kappa_2.lb$). Only when $V = c$ will we try to solve κ_2 under a bound that is greater than $mincost(\kappa_2)$. As the number of variables in κ_1 increases, these repeated attempts to solve κ_2 can multiply.

Some savings can, however, occur since each solution attempt can tighten the bounds on κ_2. Nevertheless, a multiplicative effect can occur destroying independence.

Thus on the positive side Alg. 3 can interleave the solving of the current components by branching on variables from different components at each recursion. This can produce refined bounding information that can be sufficient to refute a whole collection of components without ever having to solve any component to optimality. This can be accomplished while still obtaining many of the benefits of decomposition. On the negative side however, bounding can sometimes interfere with the benefits of decomposition, as illustrated in the example above.

Fortunately, there are a couple of simple ideas that can remove the worst of the negative effects of bounding on decomposition. The first idea is to force a component to be solved once and for all, if the search continues to return to it. The second idea is to force the complete solution of a component if that component is sufficient small. In the example above, the first method would solve κ_2 (i.e., find $mincost(\kappa_2)$) after having returned to it some number of times; while the second method would solve κ_1 (finding that $V = c$ is the correct assignment to make) before advancing to κ_2 since κ_1 is small. In our implementation we did not find an effective way of utilizing the first idea: any fixed count of how often the search can return to a component before forcing it to be solved sometimes degraded performance. The second idea of forcing the solution of a component when it is small was effective, and in our implementation we forced the solution of any component whose variables had a product domain size of 20 or less.

Local Bounding: There is one further aspect of Alg. 3: the two lines **AddConstraint** and **RemoveConstraint** that bracket the for loop over V's values. The intuition for these lines is that the current component τ has an upper bound $\tau.ub$ that is initialized when τ was first added to $\mathcal{K}$ and is updated after each value for V has been attempted (line 15). Thus in the search below it is never effective to instantiate the variables of $\tau.Vars$ to values that will cause τ to achieve a value greater than $\tau.ub$. Note that this can happen even though the global bound of $\sum_{\kappa \in \mathcal{K}} .lb < \text{UB}$ remains valid. We have found that the easiest way to enforce this local bound on the settings of $\tau.Vars$ is to post a constraint on the search below. Note that the strength of this constraint increases as we obtain tighter bounds on $\tau.ub$.

For example, say that τ contains the objectives $o_1(A, B, C)$, $o_2(A, E, F)$, and $o_3(F, G)$ and that $\tau.ub = 10$. If we branch on $A = a$ with $\Delta^a = 3$ then sometime later on $F = f$ with $\Delta^f = 3$, we will have broken τ into three sub-components: $\kappa_1 = \{o_1(A = a, B, C)\}$, $\kappa_2 = \{o_2(A = a, E, F = f)\}$, and $\kappa_3 = \{o_3(F = f, G)\}$, and accumulated an immediate cost of $\Delta^a + \Delta^f = 6$. This means that if $\kappa_1.lb + \kappa_2.lb + \kappa_3.lb > (10 - 6)$ we can immediately backtrack to the deepest point a variable of τ has been instantiated (in this case to undo the assignment $F = f$). That is, under the assignments $A = a$ and $F = f$, τ cannot achieve its optimal value—it is already exceeding a known upper bound on its optimal value. More formally, we require that in the subtree below if S is the set of components that have been generated from τ, and A is the set of assignments that have been made to variables of τ, then $\sum_{\kappa \in S} \kappa.lb + cost(A, \tau) \leq \tau.ub$.

There are other ways of implementing this local bound condition, but utilizing a hard constraint is a simple method. There are also potentially other ways the local bounds could be used, including, e.g., using more sophisticated propagation of the

added constraint. In our current implementation we are only checking this constraint and backtracking when it is violated.
Formal Results: It can be proven that Alg. 3 is correct.

Theorem 1. *If on entry to Alg. 3* $\forall \kappa \in \mathcal{K}.\big(\kappa.lb \leq mincost(\kappa) \leq \kappa.ub\big)$, *then if* $\text{UB} > \sum_{\kappa \in \mathcal{K}} mincost(\kappa)$ *on return* $\forall \kappa \in \mathcal{K}.\big(\kappa.lb = mincost(\kappa) = \kappa.ub\big)$. *On the other hand under the same entry conditions, if* $\text{UB} \leq \sum_{\kappa \in \mathcal{K}} mincost(\kappa)$, *then on return* $\text{UB} \leq \sum_{\kappa \in \mathcal{K}} lb(\kappa)$.

This theorem can be proved by induction on the total number of variables in the set of components in $\mathcal{K}$. The base case is when there are no variables in $\mathcal{K}$, i.e., $\mathcal{K}$ is empty. The inductive case is straight-forward. This theorem means that Alg.3 correctly solves the initial input COP, $\mathcal{C}$, as long as the initial bounds on $mincost(\mathcal{C})$ are valid—any valid bounds will work, but tighter bounds can yield smaller search trees.

The space requirements of Alg. 3 are also worth looking at. It can be noted that after the value $V = d$ has been tried, all newly generated components ($\mathcal{K}^d$) can be discarded. If the input COP $\mathcal{C}$ has n variables, then there can be at most n components in $\mathcal{K}$ (each component must contain at least one variable), and we can descend a path of at most length n. Thus at most $O(n^2)$ space is ever needed to store the active components during the algorithm's operation, above and beyond the space initially needed to represent the input problem $\mathcal{C}$.

The algorithm's performance can be considerably enhanced by remembering previously encountered components in a cache. Thus after new bounds on τ have been computed, at line 17, these bounds (perhaps exact) can be stored in the cache and reused whenever τ is encountered again in the search. Caching is an important part of our implementation and we have utilized the template techniques described in [10] to make its use more efficient. Note, that the cache serves only to improve the algorithm's performance, it is not required for the algorithm's correctness.

4 Local Propagation

An important technique when solving COPs is local propagation or soft-arc consistency, developed in a number of previous works, e.g., [13, 8, 6]. This technique works by "sweeping" values from the sub-objectives to a zero-arity sub-objective. The value of the zero-arity objective can then be used as a lower-bound on the COP's value, and to prune the variable domains.

The technique works by adding to the original COP unary sub-objectives $o_i(V_i)$, one for each variable, and a zero-arity objective $0()$ (none of these added objectives affect decomposition). Sweeping (enforcing soft-arc consistency) moves value into the zero-arity objective.

Two sweeping transformations are employed. First, values can be swept between a unary objective $o_1(V)$ and any binary objective involving V, e.g., $o_2(V, X)$. In particular, if for $a \in Dom[V]$ we have that $\min_{b \in Dom[X]} o_2(a, b) = \alpha > 0$, then we can sweep α from o_2 into o_1: for all $b \in Dom[X]$ we reset $o_2(a, b) = o_2(a, b) - \alpha$, and $o_1(a) = o_1(a) + \alpha$. Intuitively, if o_2 yields a value of at least α when $V = a$, then we can move α into the unary objective over V adding it to the unary cost of

$V = a$. Similarly, we can sweep a value from o_1 into o_2. If $o_1(a) = \alpha > 0$: we reset $o_2(a,b) = o_2(a,b) + \alpha$ for all $b \in Dom[X]$, and $o_1(a) = 0$. Second, values can be swept from a unary objective $o_1(V)$ into the zero-ary objective $0()$: if $\min_{a \in Dom[V]} o_1(a) = \alpha > 0$ we can reset $0() = 0() + \alpha$ and $o_1(a) = o_1(a) - \alpha$ for all $a \in Dom[V]$. These two types of transformations are equivalence preserving in the sense that the updated COP has an unchanged minimum cost.

Local propagation can be added to the three previously specified algorithms as follows. For Alg. 1 we set Δ^d on line 8 so that it is equal to all of the costs that have been swept to $0()$ as a result of applying local propagation after the assignment $V = d$, and we invoke the algorithm recursively (line 9) on $\mathcal{C}|_{V=d}$ after local propagation has been applied to the reduced problem (thus the bounds computed at line 3 are with respect to a problem that has already been modified by local propagation). Similarly to add local propagation to Alg. 2, we set Δ^d on line 7 to be the total value swept to $0()$ *from variables of the component* κ, and break $\kappa|_{V=d}$ into components (line 8) after local propagation has been performed.

Finally, to add local propagation to our new algorithm (Alg. 3), we again set Δ^d (line 8) to be the total value swept to $0()$ as a result of applying local propagation after the assignment $V = d$, and break $\kappa|_{V=d}$ into components (line 9) after local propagation has been performed. In addition, to accommodate local bounding we enforce the constraints added at line 5 by ensuring that in the search below the sum of the lower bounds of all of the components generated from τ plus the total value swept to $0()$ from variables of τ always remains $\leq \tau.ub$.

Caching: Local propagation can also interfere with caching. With caching, we store the bounds computed on components that arise during search and reuse these bounds if the component reappears in the search. However, the next time the component appears local propagation might have moved a different amount of value into or out of the component as compared to the previous time the component was encountered. This can invalidate the cached bounds.

To exploit caching in the presence of local propagation we must make the bounds independent of the current propagation before we store them in the cache, and adjust these bounds to account for the current propagation when we retrieve them from the cache. In [8] a technique was developed for accomplishing this when a fixed tree-decomposition is used to guide the search. With a fixed decomposition the components that will arise during search can be predicted ahead of time. In Alg. 3, however, the order in which the variables are instantiated is unconstrained—i.e., a fixed tree-decomposition is not used. Rather, components are detected dynamically whenever they are created by the instantiated variables. Nevertheless, we were able to generalize the techniques of [8] so that we can compute the value that has flowed into and out of the components as they are generated during search. Using these flows the cached bounds can be adjusted so that they are made independent of the context when they are to be stored in the cache, and made compatible with the current context when retrieved from the cache.

5 Tree Decompositions

A commonly used technique for exploiting decomposition during search is to compute a tree-decomposition $\mathcal{T}$ for the constraint graph prior to search. $\mathcal{T}$ is a tree where each node is labeled by a set of variables of the COP (these labels satisfy certain conditions, see, e.g., [1]). We then force the variable ordering of the backtracking search to follow $\mathcal{T}$ by requiring that it always branch on an unassigned variable from an active node of $\mathcal{T}$. Initially only the root of $\mathcal{T}$ is active, and once all variables in an active node have been assigned all of its children become active nodes. By then forcing the variable ordering to follow the tree-decomposition the components that will appear during search can be determined before the search commences. This reduces the overhead of detecting and caching components. The other advantage of computing a tree-decomposition prior to search is that more expensive algorithms can be used that can better analyze how to effectively decompose the COP.

We have specified our algorithms as using arbitrary variable orderings. When these orderings are not following a fixed tree-decomposition detecting and caching components during search is more expensive. However, such fully dynamic variable orderings can yield small search trees. In [1] it was proved that for some problem instances fully dynamic variables orderings can yield a super-polynomial speedup over variable orderings forced to follow *any* fixed tree-decomposition. Empirical evidence has also been given that despite the higher overheads, search with decomposition can perform better with dynamic variable orderings in COPs [15].

With the additional flexibility for variable ordering provided by our algorithm we have found that a hybrid approach can be very effective. In this hybrid we compute a tree-decomposition and try to follow it. However, we allow the algorithm to deviate from the ordering dictated by the tree-decomposition if an alternative variable looks particularly promising. In particular, using a heuristic to score the variables, we impose an additional penalty on any variable that would violate the ordering imposed by the tree-decomposition. However, if that variable's heuristic score (measuring the merit of branching on it next) is high enough it can overcome the penalty and cause the search to make a different decision than that dictated by the tree-decomposition.

6 Experimental Results

We implemented the three algorithm described above. We have additionally added local propagation to these algorithms, and for Alg. 2 and Alg. 3 we include caching of previously solved components. We have tested these algorithms on both weighted-CSP (wCSP) problems and Most Probable Explanation (MPE) problems from Bayesian networks.

The following specific versions of these algorithms were tested: (1) **BB** which is Alg. 1 with FDAC local propagation implemented in the state-of-the-art solver Toolbar [3]; (2) **AND/OR** which is Alg. 2 with FDAC local propagation using a variable ordering that follows a fixed tree-decomposition; (3) **OR-Decomp+T** which is our Alg. 3 with FDAC local propagation using a variable ordering that follows a fixed tree-decomposition; (4) **OR-Decomp+D** which is Alg. 3 with FDAC local propagation using a heuristically guided dynamic variable ordering; and (5) **OR-Decomp+G**

which is Alg. 3 with FDAC local propagation using a hybrid variable ordering that follows a fixed tree-decomposition but can opportunistically branch on other variables if they have high enough heuristic score. It should be noted that although OR-Decomp+T follows a fixed tree-decomposition it still has more flexibility in its variable ordering than AND/OR following a fixed tree-decomposition. This added flexibility arises from the fact that OR-Decomp+T can branch on a variable from any active node of the tree decomposition, AND/OR search on the other hand must commit to a particular node n of the tree-decomposition and branch on all variables in the subtree below n before being able to branch on any variable in the labels of n's siblings.

The heuristic score used for in the variable ordering decisions[2] is the an adaptation of the Jerslow heuristic that has previously been used for solving COPs [3]. Previous work has found that this heuristic tends to be more effective than simpler heuristics based on domain size or variable degree. Intuitively, in COPs this heuristic considers both the variable's domain size and the average cost of the objective functions the variable appears in.

All algorithms utilized a value ordering determined by the unary objectives used during local propagation. That is, the values for variable V_i were ordered by lowest unary cost $o_i(V_i)$. For those algorithms that utilized a tree decomposition, these decompositions were computed using a min-fill algorithm [11]. The AND/OR search ordered its components so as to solve the largest component first. All experiments were run with 600 second timeouts, and were conducted on 2.66GHx machines with 8GB of memory. In our experiments we found that the space used in caching never exceeded available memory, so we did not have to prune the cache during search. The following four benchmarks were tested.

The **Radio Link Frequency Assignment Problem** (RLFAP) assigns frequencies to a set of radio links in such a way that all the links may operate together without noticeable interference. The RLFAP instances were cast as binary wCSP's [5]. The benchmark family includes 6 problems.

The **Earth Observing Satellites** (SPOT5) problems select from a set of candidate photographs a subset such that some imperative constraints are satisfied and the total importance of the selected photographs is maximized. The problems have been formulated as wCSP's with binary and ternary constraints in the SPOT5 benchmark [2]. The benchmark family consists of 20 problems.

The **GridNetworks** (Grid) problems involve computing the setting of the variables in a Bayes Net that have maximum probability (an MPE problem). The net is a $N \times N$ grid with CPT's that have been filled with values that were either randomly (uniformly) chosen from the interval (0,1) or were randomly assigned 0 or 1. The problem instances have N ranging between 10 and 38, with 90% of the CPTs entries were 0 or 1 [17]. The benchmark family consists of 13 problems.

ISCAS-89 circuits are a benchmark used in formal verification and diagnosis. The problem set has been converted into n-ary wCSPs [4]. The benchmark family consists of 10 problems.

---

[2] Even in those algorithms where the variable ordering must follow a fixed tree-decomposition there are typically a range of variables in the active nodes that can be branched on; the choice of which of these variable to branch on next is determined by the heuristic score.

Table 1 summarizes the number of problems solved by the various algorithms form the various benchmarks. Of the 49 total problem instances **BB** solved 22 problems across the four benchmarks; **AND/OR** solved 22 problems; **OR-BBDecomp+T** solved 23 problems, showing an improvement by dtrack over standard AND/OR search; and **OR-BBDecomp+G** solved the most problems at 24.

Benchmark	**RLFAP (6)**	**Spot5 (20)**	**Grids (13)**	**Iscas89 (10)**	**Total (49)**
BB	5	4	4	9	22
AND/OR	5	5	3	9	22
OR-Decomp+T	5	5	3	10	23
OR-Decomp+D	5	4	4	9	22
OR-Decomp+G	5	5	4	10	24

Table 1. Number of Problems Solved (600 second timeout)

The RLFAP instances have small tree-width that can be calculated quickly. Hence decomposition techniques offer a much lower theoretical time bounds than standard branch and bound search. However, we found that many of the instances benefit more from added flexibility in variable ordering than from decomposition.

In particular, as shown in Table 2 **BB**, **OR-Decomp+D**, and **OR-Decomp+G** all perform well on these benchmarks, since they allow for the largest variable ordering freedom. **AND/OR** and **OR-Decomp+T** both perform poorly on the benchmarks, although **OR-Decomp+T** took less total time to solve all instances since it allows more freedom of variable selection than **AND/OR**.

Instance	**BB**	**AND/OR**	**OR-Decomp+T**	**OR-Decomp+D**	**OR-Decomp+G**
CELAR6-SUB0	0.16	0.03	0.21	0.33	0.21
CELAR6-SUB1-24	2.64	13.2	8.27	2.95	3.35
CELAR6-SUB1	41.3	94.96	57.14	43.42	55.16
CELAR6-SUB2	15.42	101.21	135.19	16.15	16.2
CELAR6-SUB3	239.65	446.28	430.98	222.67	228.61

Table 2. RLFAP Instances - Time in Seconds (600 second timeout)

The Spot5 instances shown in Table 3 are solved efficiently by algorithms exploiting decomposition. For example, **BB** and **OR-Decomp+D** could not solve one of the instances that was solved by all other approaches. Although **OR-Decomp+D** can decompose problems, the problem does not decompose quickly unless a tree decomposition is used to guide search toward decompositions. **AND/OR**, **OR-Decomp+T**, and **OR-Decomp+G** all used decomposition effectively on the Spot5 instances.

Instance	**BB**	**AND/OR**	**OR-Decomp+T**	**OR-Decomp+D**	**OR-Decomp+G**
1502	0.05	0.02	0.06	0.04	0.06
29	1.66	0.01	0.04	0.03	0.03
404	89.18	0.7	1.2	1.49	1.83
54	0.18	0.01	0.02	0.04	0.03
503	Timeout	0.23	1.19	Timeout	40.19

Table 3. Spot5 Instances - Time in Seconds (600 second timeout)

The Grid problems also benefit greatly from flexibility in variable ordering. **OR-Decomp+D** and **OR-Decomp+G** are both extremely effective since they exploit decomposition in the problem while still allowing complete dynamic variable ordering. Neither **AND/OR** nor **OR-Decomp+T** could solve instance 90-24-1.

Instance	**BB**	**AND/OR**	**OR-Decomp+T**	**OR-Decomp+D**	**OR-Decomp+G**
90-10-1	0	1.01	0.85	0.03	0.06
90-14-1	0.02	17.74	1.74	0.12	0.26
90-16-1	0.23	364.07	89.93	1.08	4.45
90-24-1	455.11	Timeout	Timeout	3.26	10.82

Table 4. Grid Instances - Time in Seconds (600 second timeout)

ISCAS'89 instance also benefit from more flexible variable ordering, but decomposition is also effective. The two algorithms that solved the most ISCAS'89 instances are **OR-Decomp+T** and **OR-Decomp+G**. **OR-Decomp+T** could solve one instance not solved by **AND/OR**.

Instance	**BB**	**AND/OR**	**OR-Decomp+T**	**OR-Decomp+D**	**OR-Decomp+G**
c432	0.23	0.13	129.76	0.58	5.17
c499	0.09	0.09	0.23	0.39	0.43
c880	0.3	0.28	0.82	1.33	1.54
s1196	0.13	0.13	256.03	0.57	0.65
s1238	0.11	0.12	0.5	0.55	0.63
s1423	1.64	1	3.23	0.89	1.04
s1488	0.17	0.16	0.53	0.88	1.01
s1494	Timeout	0.16	0.51	Timeout	1.0
s386	3.38	0.01	0.36	12.39	0.06
s953	0.09	Timeout	0.05	3.38	1.07

Table 5. ISCAS'89 Instances - Time in Seconds (600 second timeout)

Finally, we compared **OR-Decomp+G** with Best First search using static mini buckets (**BF**) [16]. This algorithm explores an AND/OR tree, but does so in a best first manner rather than in a depth-first manner. Thus it can need considerably more space, which unlike caching is required for correctness. So we see, e.g., that it could not solve any of the RLFAP problems due to its space requirements. However, it was also able to solve some problems not solvable by OR-Decomp. It is difficult to assess the cause of this difference however, since the bounding technique of mini-buckets is quite distinct from the bounding technique of local propagation. Thus it is hard to say if these results (except for the RFLAP results) are due to better bounding or due to the differences in the search used by the algorithms.

Benchmark	**RLFAP**	**Spot5**	**Grids**	**Iscas89**	**Total**
OR-Decomp+G	5	5	4	10	24
BF	0	5	8	10	23

Table 6. Number of instances solved (600 second timeout)

7 Conclusions and Future Work

Constraint Optimization Problems can benefit greatly from both dynamic variable ordering and decomposition. Unfortunately the recursive nature of current decomposition techniques forces search to solve only one active component at a time. In this paper, we have introduced a novel search method that is able to exploit decomposition while at the same time allowing complete freedom to branch on any unassigned variable of any active component. We also introduced a new variable ordering algorithm which guides search toward decomposition, but still allows for the flexibility to choose any variable.

References

1. F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and Complexity Results for #SAT and Bayesian Inference. In *44th Symposium on Foundations of Computer Science (FOCS)*, pages 340–351, 2003.
2. E. Bensana, M. Lemaitre, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3):293–299, 1999.
3. S. Bouveret, S. de Givry, F. Heras, J. Larrosa, E. Rollon, M. Sanchez, T. Schiex, G. Verfaillie, and M. M.Zytnicki. Max-csp competition 2007. In *Proceedings of the Second International CSP Solver Competition*, pages 19–21, 2008.
4. F. Brglez, D. Bryan, and K. Kozminski. Combinatorial Profiles of Sequential Benchmark Circuits. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pages 1229–1234. IEEE, 1989.
5. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.
6. M. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 68–73, 2007.
7. A. Darwiche. Recursive conditioning. *Artif. Intell.*, 126(1-2):5–41, 2001.
8. S. de Givry, T. Schiex, and G. Verfaillie. Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In *AAAI*, pages 22–27, 2006.
9. P. Jégou, S. Ndiaye, and C. Terrioux. Dynamic heuristics for backtrack search on tree-decomposition of csps. In *IJCAI*, pages 112–117, 2007.
10. M. Kitching and F. Bacchus. Symmetric component caching. In *IJCAI*, pages 118–124, 2007.
11. U. Kjaerulff. Triangulation of graphs - algorithms giving small total state space, Technical Report R90-09. Technical report, Department of Computer Science, University of Aalborg, March 1990.
12. J. Larrosa, P. Meseguer, and M. Sánchez. Pseudo-tree search with soft constraints. In *ECAI*, pages 131–135, 2002.
13. J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.
14. R. Marinescu and R. Dechter. And/or branch-and-bound for graphical models. In *IJCAI*, pages 224–229, 2005.
15. R. Marinescu and R. Dechter. Dynamic orderings for and/or branch-and-bound search in graphical models. In *ECAI*, pages 138–142, 2006.
16. R. Marinescu and R. Dechter. Best-first and/or search for graphical models. In *Proceedings of the AAAI National Conference (AAAI)*, pages 1171–1176, 2007.
17. T. Sang, P. Beame, and H. Kautz. Performing bayesian inference by weighted model counting. In *AAAI*, pages 475–482, 2005.