Set Branching in Constraint Optimization

Matthew Kitching and **Fahiem Bacchus**
Department of Computer Science,
University of Toronto, Canada.
[kitching|fbacchus]@cs.toronto.edu

Abstract

Branch and bound is an effective technique for solving constraint optimization problems (COP's). However, its search space expands very rapidly as the domain sizes of the problem variables grow. In this paper, we present an algorithm that clusters the values of a variable's domain into sets. Branch and bound can then branch on these sets of values rather than on individual values, thereby reducing the branching factor of its search space. The aim of our clustering algorithm is to construct a collection of sets such that branching on these sets will still allow effective bounding. In conjunction with the reduced branching factor, the size of the explored search space is thus significantly reduced. We test our method and show empirically that it can yield significant performance gains over existing state-of-the-art techniques.

1 Introduction

Branch and bound is a powerful algorithm for solving discrete valued Constraint Optimization Problems (COP's). At each node of the search tree it selects an unassigned variable and attempts, in turn, to assign the variable each unpruned value in its domain. Each of these assignments generates a new branch in the search tree, and bounding is used to close those branches that cannot improve the current best solution.

One difficulty with branch and bound is that when the domains of values for the variables are large, the nodes in its search space have a high branching factor and its explored search space grows very quickly even when efficient bounding techniques are used.

A common approach for alleviating this problem is the technique of domain splitting [Dincbas *et al.*, 1988]. In domain splitting the unpruned values of the selected variable V are first split into two sets based on the lexicographical order of the values. Then, instead of branching on the V's individual unpruned values, the search branches on these two sets, reducing the search space's branching factor down to two. Branch and bound descends its search tree until either a bound is violated or the domain of every variable has been reduced to a single value, in which case these values form a new best solution.

Unfortunately, by itself domain splitting does not reduce the size of branch and bound's search space. It simply changes the shape of the search tree making it narrower but potentially deeper. In particular, in the absence of pruning due to bounding, each leaf node generated by domain splitting, where all variable domains have been reduced to single values, lies in one to one correspondence with the leaf nodes generated by branching on individual values, where all variables have been assigned a single value. Domain splitting can only be effective when bounding is able to refute entire sets of values without having to explicitly test each value.

In this paper we present a new approach for exploiting the intuition underlying domain splitting. Our technique, called **set branching**, operates like domain splitting. In particular, it also partitions the unpruned values of a variable into sets and then branches on these sets rather than on individual values. However, instead of arbitrarily splitting the variable's domain based on lexicographic order, we develop a principled technique for clustering the values into sets of similar values. Our technique aims to construct sets such that branching on these sets will generate improvements to the current bounds that are similar to the improvements that would be generated by branching on the individual values in the set. Our method thus allows set branching to work well with bounding so as to reduce the overall size of the *explored* search tree.

We demonstrate the empirical effectiveness of our approach on the RLFAP and Pedigree problem sets showing that our method can yield significantly improvements in performance over other state-of-the-art COP solvers.

2 Background

A discrete **Constraint Optimization Problem**, $\mathcal{P}$, is specified by a tuple $\langle \mathcal{V}, Dom, \mathcal{O} \rangle$, where $\mathcal{V}$ is a set of variables; for each $V \in \mathcal{V}$, $Dom[V]$ is the domain of V; and $\mathcal{O}$ is an objective function that maps every complete assignment of variables to a cost. We concentrate here on objective functions $\mathcal{O}$ expressed as a set of sub-objectives. Each $O \in \mathcal{O}$ is dependent on a subset of variables $scope(O) \subset \mathcal{V}$, and on any complete assignment τ to the variables, $\mathcal{O}(\tau) = \sum_{O \in \mathcal{O}} O(\tau|_{scope(O)})$ (i.e., the sum of the sub-objectives evaluated at the assignments specified in τ). We will use the term **objectives** to denote the sub-objectives $O \in \mathcal{O}$.

The goal of solving a COP is to find an assignment of values to the variables that minimizes $\mathcal{O}$. With minimization,

hard constraints can be modeled by including each hard constraint as an extra sub-objective that maps satisfying tuples to 0 and violating tuples to ∞.

Given a COP $\mathcal{P} = \langle \mathcal{V}, Dom, \mathcal{O} \rangle$ a **Domain Reduced COP**, $\mathcal{P}|_{Dom^*} = \langle \mathcal{V}, Dom^*, \mathcal{O}^* \rangle$, is generated by reducing the variable domains of $\mathcal{P}$ to the new sets specified in Dom^*, where $Dom^*[V] \subseteq Dom[V]$ for all $V \in \mathcal{V}$. The variables $\mathcal{V}$ of $\mathcal{P}|_{Dom^*}$ are the same as those in $\mathcal{P}$, and its objectives, $\mathcal{O}^*$, are the objectives of $\mathcal{O}$ but now defined only over tuples of values contained in Dom^*.

Branch and Bound is a standard technique for solving COP's using backtracking search. A *global upper bound* (**GUB**) is maintained which represents the cost of the best known solution to the problem. At each node of the search tree, branch and bound selects an unassigned variable V and branches on each unpruned value in $Dom[V]$. Branching on the individual values of a variable is called **value branching**. After assigning a value to V, a lower bound on the cost of any complete assignment extending the current assignment, is calculated. If the bound is greater than GUB, branch and bound backtracks, since no optimal solution can be found in subtree below the current assignment.

3 Domain Splitting and Set Branching

When the variables of the COP have large domains the branching factor of branch and bound's search tree becomes large, and even when efficient bounding techniques are used the search space it must explore often becomes impractically large. In this section, the set branching algorithm and its precursor, domain splitting, are described. These algorithms aim to reduce the search space's branching factor.

Domain Splitting is a technique for improving branch and bound search [Dincbas *et al.*, 1988]. At each node of the search tree, domain splitting selects a variable V, and partitions V's unpruned values into two sets, L and R, based on the lexographic order of the values. That is, all values in the set L are lexicographically less than any value in the set R. To test whether or not L leads to an optimal solution, a domain reduced COP is created by pruning the values of R from V's domain. If the lower bound of the domain reduced COP is larger that GUB, then no optimal solution can exist using any value in L. If the bound is not exceeded, search descends down the search tree until either a bound is violated or the domain of every variable is reduced to a single value, in which case a new solution is found. After testing L, R is tested in a similar manner.

Set Branching (weak assignments) is a technique originally developed for Constraint Satisfaction Problems (CSP) that partitions the values of a domain into sets, and branches on each set in a fashion similar to that employed by domain splitting [Larrosa, 1997; Silaghi *et al.*, 1999]. The approach presented here differs from this previous work in two ways. The most important difference is that previous work has utilized problem specific algorithms for partitioning the values into sets. Here we present a generic algorithm for accomplishing this. The second difference is that previous work has addressed satisfaction problems not optimization problems.

Consider the COP $\mathcal{P} = \langle \mathcal{V}, Dom, \mathcal{O} \rangle$. The standard branch and bound algorithm first selects a variable V and then branches on each unpruned value $a \in Dom[V]$. Suppose that instead of branching on each value, the unpruned values of $Dom[V]$ are *partitioned* into k disjoint sets, $\mathcal{S} = [S_1, ..., S_k]$. $\mathcal{S}$ is called a **branching set** for V. Set branching branches on each set S_i by pruning the values in all other sets $S_j, i \neq j$ from $Dom[V]$.

Algorithm 1 shows the set branch (**SB**) algorithm. SB is a recursive function with one parameter, $\mathrm{CurrentDom}$, that is an array of variable domains indexed by the variables of the COP. The current domain of each variable V is specified by $\mathrm{CurrentDom}[V]$. If $|\mathrm{CurrentDom}[V]| = 1$, then V is considered to be assigned the single value in $\mathrm{CurrentDom}[V]$, otherwise V is unassigned. The vector $\mathrm{CurrentDom}$ defines the domain reduced COP $\mathcal{P}|_{\mathrm{CurrentDom}}$.

Algorithm 1: Set Branch Algorithm

```
1  SB(CurrentDom)
2  begin
3      if getBound(CurrentDom) ≥ GUB then
4          return
5      if ∀V.|CurrentDom[V]| = 1 then
6          GUB:= getBound(CurrentDom)
7          return
8      choose (a variable V where |CurrentDom[V]| ≥ 2)
9      S = partition CurrentDom[V] into k sets, k ≥ 2
10     foreach S_i ∈ S do
11         CurrentDom[V] = S_i
12         SB(CurrentDom)
13 end
```

SB immediately backtracks if it determines that $\mathcal{P}|_{\mathrm{CurrentDom}}$ has no solution with cost lower than GUB. The function call $getBound(\mathrm{CurrentDom})$ calculates a lower bound for $\mathcal{P}|_{\mathrm{CurrentDom}}$. If the domain size for every $V \in \mathcal{V}$ is 1, then every variable has been assigned a value and the search is at a leaf node. SB then updates the GUB and backtracks ($getBound$ when called on singleton domains returns the exact value of the corresponding complete assignment). Otherwise, a variable is selected from the set of unassigned variables (those variables V with more than one value in $\mathrm{CurrentDom}[V]$).

Line 9 partitions the values in $\mathrm{CurrentDom}[V]$ into k disjoint sets, where $k \geq 2$. By forcing k to be greater than 2, SB ensures that the domain of at least one variable is reduced in each recursive call; thus, SB will eventually exceed the GUB or reach a leaf. In Lines 10–12, SB loops through the k disjoint sets, and recursively calls itself on each one. The loop is iterated k times, where k is the number of sets created in Line 9. In standard branch and bound there is an equivalent loop which is iterated $|Dom[V]|$ times. Thus, SB reduces the branching factor of the node from $|Dom[V]|$ to k.

However, as pointed out above, to attain a net reduction in the search space it is not sufficient to simply reduce the branching factor. We must also be able to obtain good bounding from $getBound(\mathrm{CurrentDom})$ to avoid descending too deeply in the search tree.

4 Obtaining Good Bounds

To obtain good bounding from set branching we must examine more closely how set branching affects the bounds that could be computed by $getBound$. Consider two values a and b in $\mathrm{CurrentDom}[V]$. The assignments $V \leftarrow a$ and $V \leftarrow b$ generate two different reduced COPs P_a and P_b. When standard value branching is used $getBound$ will be applied to compute bounds for these two reduced COPs. If P_a and P_b are structurally similar, then $getBound$ will compute similar bounds for both problems irrespective of the specific bounding technique used by $getBound$.

Intuitively, if P_a and P_b are similar and generate similar bounds, then a and b should not be distinguished. That is, a and b should be placed in the same set S_i by SB. Then, when SB branches on S_i, the resultant reduced COP, $P_{\{a,b\}}$, will likely be structurally similar to both P_a and P_b, and $getBound$ should generate bounds for $P_{\{a,b\}}$ similar to the bounds it would have generated for either P_a or P_b. Thus, set branching on S_i can yield similar bounds without forcing a choice between a and b. Our approach, then will be to identify values in V's domain that would be treated similarly by $getBound$, and place these similar values into the same set during set branching.

In order to formalize these intuitions into a specific algorithm we must focus on the details of a particular technique for implementing $getBound$. Here we examine the technique of **enforcing local consistency** [Larrosa and Schiex, 2003], a powerful and popular technique employed to implement $getBound$ when solving COPs.

Let O_{ij} denote a binary objective between the variables V_i and V_j where $O_{ij}(a, b)$ is the cost of assigning a to V_i and b to V_j, and let O_i denote a unary objective for variable V_i where $O_i(a)$ is the cost of assigning a to V_i. If the COP does not initially specify a unary objective O_i for variable V_i we add an initially null unary objective O_i to the COP mapping all values of V_i to zero. Finally, a zero arity lower bound objective $O_\emptyset$ is added to the COP with initial value zero. When using local consistency for lower bounding a COP, $getBound$ simply returns the current value of $O_\emptyset$.

Enforcing local consistency in COP's involves modifying the COP into an equivalent problem by flowing value towards $O_\emptyset$. The modifications ensure that $O_\emptyset$ always remains a lower bound on the minimal value that the COP can achieve. In addition to generating a lower bound, local consistency can also prune infeasible values from the variable domains, thus improving the efficiency of branch and bound's search.

The method uses two atomic transformations that are both guaranteed preserve equivalence. The first transformation is a *unary transformation*. Let α be the minimum cost $O_i(a)$ for every $a \in Dom[V_i]$. If $\alpha > 0$ then it can be added to $O_\emptyset$ and subtracted from every $O_i(a), a \in Dom[V_i]$.

The second transformation is a *binary transformation*. Consider two objectives O_{ij} and O_i, and a value $a \in Dom[V_i]$. Let α be the minimum cost of $O_{ij}(a, b)$ over all $b \in Dom[V_j]$. If $\alpha > 0$ then it can be added to $O_i(a)$ and subtracted from $O_{ij}(a, b)$ for all $b \in Dom[V_j]$. This is called a flow into $O_i(a)$.

It is useful to divide the process of enforcing local consistency into two stages. The first stage, called *simplification*, occurs when the domain of a variable V_i is reduced at a node by branching. During simplification, there is a unary transformation from O_i to $O_\emptyset$, and for every objective O_{ij}, $\forall b \in Dom[V_j]$, there is a binary transformation from $O_{ij}(a, b)$ into $O_j(b)$. The second stage, called *propagation*, repeatedly applies unary and binary transformations to the simplified problem, with the goal of flowing value into $O_\emptyset$ where it can returned to the search by $getBound$.

When value assignments are used, the simplification stage is extremely powerful in finding strong bounds for the problem. When the assignment $V_i \leftarrow a$ is made, the domain of V_i is reduced to a single value. In this case, the entire cost of $O_i(a)$ is immediately added to $O_\emptyset$. In addition, the entire cost of the edge $O_{ij}(a, b)$ is added to $O_j(b)$ for each $b \in Dom[V_j]$. These large binary transformations often allow additional binary and unary transformations that further increase $O_\emptyset$ during the propagation stage.

However, suppose that instead of reducing a variable domain to a single value, as when value branching is employed, the domain of a variable is simply reduced, as when set branching is employed. In this case, the simplification stage can be far less effective. For example, consider the case where the domain of V_i is reduced to two values, a and b, with unary costs $O_i(a) = 10$ and $O_i(b) = 0$. In this case, no unary transformation can occur during simplification, since α, the minimum cost in O_i, is not greater than zero. In this case we see that $getBound$ would compute two very different values under these two assignments.

Since analyzing the effects of propagation on different values is complex, and likely to be computationally expensive, we focus instead on analyzing the effects of simplification on different values. This will give us partial information about how different values would be treated by $getBound$.

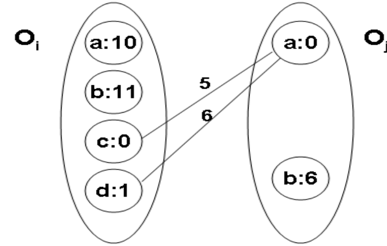

Figure 1: Objectives O_i, O_j, and O_{ij}.

Simplification and Set Branching. Suppose variable V_i is selected for set branching, $Dom[V_i] = \{a, b, c, d\}$, and the COP $\mathcal{P}$ has the objective functions represented in Figure 1. In the diagram edges between two values indicate the cost of O_{ij} on that pair of values, while omitted edges indicate that O_{ij} has zero cost on that pair. The unary objectives are represented as the costs listed with each value.

Figure 2 shows the COP's that are created after making value assignments with the individual values of V_i and applying simplification to the problem. The lower bound objective $O_\emptyset$ is increased by 10, 11, 0, and 1 respectively from the unary transformation made during simplification. In addition, a binary transformation is made during simplification which increases $O_j(a)$ by 0, 0, 5, and 6 respectively. The fig-

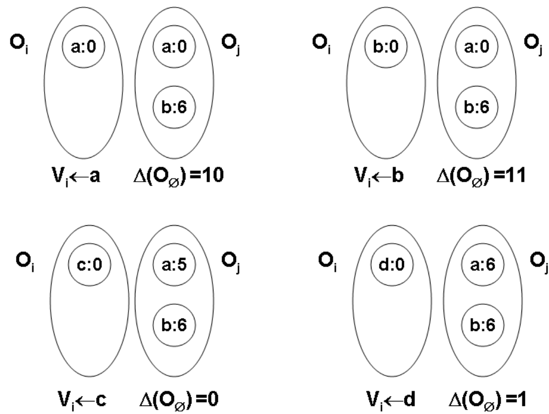

Figure 2: Objectives after value branching on a, b, c and d

ure shows that the COPs that arise after branching on a and b and then performing simplification are quite similar, as are the COPs that arise after branching on c and d.

The intuition described above would indicate that set branching on $\{a, b\}$ should generate similar bounds as the two problems created by branching on these values individually. Similarly for set branching on $\{c, d\}$. The result of simplification after branching on these two sets, $S_{i1} = \{a, b\}$ and $S_{i2} = \{c, d\}$ is shown in Figure 3. The figure shows that in this example our intuition is correct.

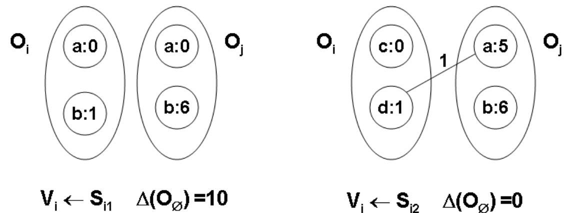

Figure 3: Objectives after set branching on S_{i1} and S_{i2}

Because the propagation phase operates on the simplified problem, we would expect that it also will generate similar increases to the lower bound $O_\emptyset$ on similar simplified problems. However, it can be that small changes in the simplified problem can be exaggerated into profound differences during propagation. Nevertheless, similarity under simplification is a reasonable heuristic as is borne out by our experiments.

We say that a set of values are **similar** if the transformations allowed during the simplification stage are similar regardless of whether set branching or value branching is used on the values in a set. The **set branching ratio** is the ratio of the size of the branching set to the number of values in the domain of the variables. Set branching is most effective when the set branching ratio is low while the similarity is high.

4.1 Calculating Effective Branching Sets

Effective set branching hence requires finding branching sets that partition the variable's domain into a relatively small number of sets (yielding a low set branching ratio) where each set contains similar values (yielding high similarity). In this section we propose a heuristic algorithm for constructing such branching sets.

For each variable V_i, the clustering algorithm builds a dissimilarity matrix Dis where Dis(a, b) represents the dissimilarity between value a and value b after simplification on the respective reduced problems takes place. SB uses this matrix to cluster the values of the variable into sets by trying to place highly dissimilar values into different sets. Each cell Dis(a, b) of the dissimilarity matrix is initialized to zero and is then updated with the following two rules:

Rule 1.
difference := $abs(O_i(a) - O_i(b))$
if (difference > 0)
  Dis(a,b) := Dis(a,b) + difference
  Dis(b,a) := Dis(b,a) + difference

Rule 1 updates Dis(a, b) if $O_i(a) \neq O_i(b)$. Consider the following example: let $O_i(a) > O_i(b)$, let $\alpha = O_i(a) - O_i(b)$, and suppose our clustering algorithm places a in S_q and b in S_r. Suppose also that SB branches on S_q. If $q = r$ (i.e., a and b are placed in the same set) then the maximum unary transformation from O_i into $O_\emptyset$ is $O_i(b)$. If $q \neq r$, then the maximum unary transformation from O_i into $O_\emptyset$ is $O_i(a)$. Thus, the unary transformation allowed if $q \neq r$ may be up to α greater than if $q = r$. This unary transformation is not assured however. It is possible that even if $q \neq r$ and a and b are placed in different sets, a third value $c \in S_q$ may stop any unary transformation from O_i.

Rule 2.
foreach $V_j \in \mathcal{V}$ such that O_{ij} is a binary objective
  foreach $k \in Dom[V_j]$
    difference := $abs(O_{ij}(a, k) - O_{ij}(b, k))$
    if (difference > 0)
      Dis(a,b) = Dis(a,b) + difference/$|Dom[V_j]|$
      Dis(b,a) = Dis(b,a) + difference/$|Dom[V_j]|$

Rule 2 updates the dissimilarity matrix of V_i for each variable V_j it appears in a binary objective O_{ij} with. For each value $k \in Dom[V_j]$, it may be possible to perform a binary transformation into $O_j(k)$ and the rule evaluates the difference between this transformation for two values a and b in $Dom[V_i]$. Consider the following example: let $O_{ij}(a, k) > O_{ij}(b, k)$, let $\alpha = O_{ij}(a, k) - O_{ij}(b, k)$, and suppose that the clustering algorithm places a in S_q and b in S_r. Suppose also that SB branches on S_q. Consider the binary transformations made by simplification. If $q = r$ (a and b are placed in the same set), then the maximum binary transformation from O_{ij} into $O_j(k)$ is $O_{ij}(b, k)$. If $q \neq r$, then the maximum binary transformation from O_{ij} into $O_j(k)$ is $O_{ij}(a, k)$. Thus, the binary transformation allowed when $q \neq r$ may be up to α greater than the transformation allowed when $q = r$. As with rule 1, this binary transformation is not assured even if a and b are placed in different sets.

The total value of Dis(a, b) is computed by applying rule 1 to every pair of values in $Dom[V_i]$, and rule 2 to every pair of values in $Dom[V_i]$ and every value of V_j for all V_j that appear in a binary objective with V_j. For example, consider the COP represented by Figure 1. The value of Dis(a,c) is 12.5 (rule 1 increments Dis(a,c) by 10 and rule 2 increments Dis(a,c) by 2.5). The complete dissimilarity matrix generated is shown in Figure 4.

	a	b	c	d
a	0.0	1	12.5	12
b	1	0.0	13.5	13
c	12.5	13.5	0.0	1.5
d	12	13	1.5	0.0

Figure 4: Dissimilarity matrix for COP of Figure 1

Algorithm 2, **MakeCluster** uses the computed dissimilarity matrix to cluster the values into disjoint sets. MakeCluster uses a global parameter λ which is a threshold parameter. The input to the MakeCluster is the set ValueSet, initially including all values in $\mathrm{CurrentDom}[V_i]$.

Algorithm 2: Make Cluster Algorithm

```
1  MakeCluster (ValueSet)
2  begin
3      if |ValueSet| = 1 then
4          addSetToFinalList(ValueSet) return
5      (Set1, Set2, score) := valuecluster(ValueSet)
6      if score > λ then
7          addSetToFinalList(ValueSet)
8          return
9      MakeCluster (Set1)
10     MakeCluster (Set2)
11 end
```

Alg. 2 calls the valuecluster function, which splits $\mathrm{ValueSet}$ into two sets, Set1 and Set2 and returns a score. If the score is above λ, then the algorithm assumes the values in $\mathrm{ValueSet}$ are similar to one another and adds $\mathrm{ValueSet}$ to the final list of sets that SB will use to set branch. If the score is below λ, MakeCluster is called recursively on Set1 and Set2 in order to further subdivide the values until they become singleton sets or are added to the final list of sets.

Each row and each column in the dissimilarity matrix represents a single value in $Dom[V_i]$. The valuecluster function first eliminates all rows and columns from the dissimilarity matrix that do not represent a value in ValueSet.

Valuecluster then calculates a rowscore for each row s as follows. The cells of the row are partitioned into two clusters (high and low) based their cellscore. The clustering is done using a bimodal thresholding technique [Otsu, 1979]. The mean of the high cluster is the average cellscore of cells in the high cluster and the mean of the low cluster is the average cellscore of cells in the low cluster. The rowscore for row s is equal to the mean of the low cluster divided by the mean of the high cluster. As the ratio approaches one, the values are presumed to be more similar.

After calculating the rowscore for every row, valuecluster selects the row with the lowest rowscore. It then returns two sets; the first set contains the values represented by the cells in the high cluster of the selected row, while the second set contains the values represented by the cells in the low cluster of the selected row. Valuecluster also returns the rowscore of the selected row. If the rowscore is less than some threshold percentage λ, then the partition is accepted and the algorithm is called recursively. Otherwise, the partition is not accepted (meaning that the values in the set are deemed to be similar to one another), $\mathrm{ValueSet}$ is added to the completed set, and Alg. 2 returns.

This clustering algorithm takes time $O(|\mathcal{V}|Dom^3)$ at every node, making it disadvantageous to run Alg. 2 at every node. In our SB implementation, the algorithm is run as a preprocessing step for every variable of the problem, and the resulting sets are stored. After a variable is selected in Line 8 of Alg. 1, if the variable's domain is greater than some threshold size, the stored set is used to partition the remaining domain.

4.2 Extensions

The Dissimilarity matrix presented is developed for COP's with binary constraints. However, the same concept on which the dissmiliarity matrix was based can be employed for other constraint problems. For example, a dissimilarity matrix for ternary COPs can be made by comparing the way in which values of a variable change the unary objectives of remaining variables in each ternary objective. Although the details of this algorithm are not discussed in this paper, we implemented a ternary version of the clustering technique and tested it on the Pedigree problems in the following section.

Our method has also been presented in the context of the particular bounding technique of enforcing local consistency, a powerful bounding method that is commonly applied. To adapt our approach to another bounding technique we would need to develop new method for computing the dissimilarity matrix—the rest of our method would still operate as before. In particular, the bounding technique would have to be examined so as to develop a way of estimating the dissimilarity between alternate values in terms of their effect on the bounds generated.

5 Experimental Results

Our set branch approach was tested on two benchmarks. All tests were performed on 2.66 GHz machines with 8GB of RAM and limited to a timeout of 1200 seconds.

Our first experiments were with the **Radio Link Frequency Assignment Problem** (RLFAP) shown in Table 1. These problems involve assigning frequencies to a set of radio links in such a way that the links can operate together without noticeable interference. The RLFAP instances were cast as binary weighted CSP's [Cabon *et al.*, 1999]. The benchmark family includes 23 problems. Problems that could not be solved by any the solvers (within 1200 sec.) are not listed.

Table 1 shows the results of three different algorithms on the RLFAP problems. All of the algorithms were implemented on top of the state-of-the art Toolbar system [Bouveret *et al.*, 2008], and all employ the EDAC local consistency bounding technique. **BB** is a branch and bound algorithm; **DS** is the domain splitting algorithm which splits the variable domains in half based on the value's lexicographical order in the original encoding; and **SB** is our set branch technique that utilizes Alg. 2 to cluster the variable values into sets.

The algorithms were tested with a Dom/WDeg heuristic where the next variable chosen is the uninstantiated variable with highest current domain size divided by the variable's weighted degree. The weighted degree is the sum of the expected values of the objectives the variable appears in. The values of the chosen variable V are attempted in the order lowest unary cost first (i.e., lowest $O(V)$ value). In the case of **DS** and **SB**, the sets were ordered by the minimum unary

Instance	**BB**	**DS**	**SB**
CELAR6-SUB0	**0.17**	0.22	0.22
CELAR6-SUB1-24	2.68	**2.02**	3.18
CELAR6-SUB1	42.6	35.9	**16.5**
CELAR6-SUB2	16.0	36.4	**13.4**
CELAR6-SUB3	224.9	203.8	**63.7**
CELAR6-SUB4-20	10.3	3.3	**2.7**
CELAR6-SUB4	> 1200	> 1200	**519.4**
CELAR7-SUB0	**0.05**	0.13	0.13
CELAR7-SUB1-20	**0.14**	0.6	**0.14**
CELAR7-SUB1	4.9	3.9	**3.7**
CELAR7-SUB2	42.1	36.0	**33.7**
CELAR7-SUB3	425.4	**283.9**	393.6
CELAR7-SUB4-22	6.67	7.72	**5.17**
graph07	24.2	> 1200	**13.9**
graph12	9.6	29.0	**9.3**
scen-9	8.1	96.6	**5.1**
scen-10	**0.71**	> 1200	1.3

Table 1: RLFAP Problems: solving times in CPU seconds

cost of the values in the set and ties were broken by selecting the smallest set first. **DS** and **SB** defaulted to ordinary value branching when the variable's current domain size was smaller than $\frac{1}{6}$ the original domain size. The times shown include the preprocessing time required by **SB** to compute its branching.

The results show that **SB** is the best performing of the three algorithms, beating the other algorithms on 12 out of 17 problems. It also never does much worse than **BB** or **DS**, except on CELAR7-SUB3 where it takes >100 seconds more than **DS**. Finally, it is able to solve CELAR6-SUB4 on which the other algorithms failed. In sum, on these problems, **BB** solved 16 problems in 2018 seconds, **DS** solved 14 problems in 4340 seconds, and our approach **SB** solved 17 problems in 1085 seconds.

We also examined the number of nodes expanded by the three different algorithms. **SB** generally explored a smaller search tree to solve the problem indicating that our heuristic for constructing branching sets was effective.

To test the robustness of our clustering algorithm we ran the RLFAP benchmarks an second time with the values for each variable randomly permuted before search began. The results for **DS** were significantly worse, since the initial ordering of the values probably embedded some structural information that was lost after permutation. However, **BB** and **SB** both displayed only minor changes in their performance (due to some differences in tie breaking).

We also experimented with **Pedigree** problems which try to decide if Mendelian errors exist in various pedigrees. The Pedigree instances were cast as weighted CSP's [Sanchez *et al.*, 2008]. The benchmark family includes 23 problems. Again we only list those problems solvable within the timeout by at least one of our tested algorithms.

Table 2 shows the results. **DS** performs very poorly on these problems. **BB** on the other hand performs quite well, but still **SB** is able to solve two problems not solvable by **BB**.

Instance	**BB**	**DS**	**SB**
eye	**0.01**	6.52	0.13
pedck-60-L1	**0.01**	0.11	0.03
pedck-60-L2	**1.0**	13.64	1.3
pedck-60-L12	**0.01**	120.72	0.04
pedck-350	> 1200	> 1200	**1.22**
pedck-350-L2	> 1200	> 1200	**0.84**
pedck-350-L3	671.63	> 1200	**0.88**
saudiarabia	**1.69**	3.1	160.73
wijsmanguo	11.3	> 1200	**1.43**

Table 2: Pedigree problems

6 Conclusions and Future Work

This paper demonstrated that variable values can be effectively clustered into sets so that branching on these sets can yield useful speedups over branching on individual values in branch and bound search. The key contribution of the paper is a domain independent algorithm for clustering values into sets in a way that preserves effective bounding. Although our technique was developed with a particular bounding method in mind, it should be possible to use the same intuitions to develop clustering algorithms more tailored to other more recent bounding methods, e.g., [Cooper *et al.*, 2008]. We also believe that the same intuitions could lead to effective domain independent set branching algorithms for satisfaction problems (CSPs).

References

[Bouveret *et al.*, 2008] 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.

[Cabon *et al.*, 1999] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.

[Cooper *et al.*, 2008] Martin C. Cooper, Simon de Givry, Martí Sánchez, Thomas Schiex, and Matthias Zytnicki. Virtual arc consistency for weighted csp. In *Proceedings of the AAAI National Conference (AAAI)*, pages 253–258, 2008.

[Dincbas *et al.*, 1988] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language chip. In *FGCS*, pages 693–702, 1988.

[Larrosa and Schiex, 2003] Javier Larrosa and Thomas Schiex. In the quest of the best form of local consistency for weighted csp. *IJCAI*, pages 239–244, 2003.

[Larrosa, 1997] Javier Larrosa. Merging constraint satisfaction subproblems to avoid redundant search. In *IJCAI*, pages 424–433, 1997.

[Otsu, 1979] N. Otsu. A threshold selection method from gray level histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9:62–66, 1979.

[Sanchez *et al.*, 2008] M. Sanchez, S. de Givry, and T. Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1), 2008.

[Silaghi *et al.*, 1999] M. Silaghi, D. Sam-haroud, and B. Faltings. Intelligent domain splitting for csps with ordered domains. In *International Conference on Principles and Practice of Constraint Programming*, pages 488–489, 1999.