A Portfolio Approach for Enforcing Minimality in a Tree Decomposition

Daniel J. Geschwender[1,2] Robert J. Woodward[1,2]
Berthe Y. Choueiry[1,2]
Stephen D. Scott[2]

[1,2]Constraint Systems Laboratory
[2]Department of Computer Science and Eng.
University of Nebraska-Lincoln, USA
{dgeschwe|rwoodwar|choueiry|sscott}@cse.unl.edu

TR-UNL-CSE-2016-0003

September 26, 2016

Abstract

Minimality, a highly desirable consistency property of Constraint Satisfaction Problems (CSPs), is in general too expensive to enforce. Previous work has shown the practical benefits of restricting minimality to the clusters of a tree decomposition, allowing us to solve many difficult problems in a backtrack-free manner. We explore two alternative algorithms for enforcing minimality whose performance widely vary from one instance to another. We advocate a fine-grain portfolio approach to dynamically choose, during lookahead, the most appropriate algorithm for a cluster. Our strategy operates by selecting among two algorithms for enforcing minimality and an algorithm that enforces the lowest-level of consistency, which, in our setting, is Generalized Arc Consistency. Empirical evaluation on benchmark problems shows a significant improvement both in terms of the number of instances solved and CPU time.

Contents

1 Introduction

Local consistency techniques are at the heart of Constraint Programming and constitute an invaluable tool for solving Constraint Satisfaction Problems (CSPs). On many problems, enforcing simple consistency properties, such as Generalized Arc Consistency (GAC) [Mackworth, 1977], during backtrack search can be sufficient to reduce the problem to a manageable state. However, some problems are more resilient and require stronger consistency properties to effectively filter them.

In this paper, we focus on constraint minimality as one such consistency property. A constraint is considered minimal if every tuple of the constraint can be extended to a complete solution to the CSP. Enforcing constraint minimality is prohibitively expensive because it involves enumerating many, if not all, of the solutions to the problem. However, it can be applied locally (that is, to a given subproblem) with some success [Freuder and Elfe, 1996; Karakashian *et al.*, 2013]. Following Karakashian et al. [Karakashian *et al.*, 2013], we consider enforcing the property on clusters of the tree decomposition of the problem. This restriction (or localization) to the clusters of a tree decomposition can result in a strong filtering power at a manageable cost.

Previous work has proposed two algorithms for enforcing minimality: PerTuple [Karakashian *et al.*, 2010] and AllSol [Karakashian, 2013]. Our contribution is the development of a portfolio approach for choosing between these two algorithms, as well as identifying when to forego both algorithms and instead use an algorithm for GAC [Mackworth, 1977; Bessière *et al.*, 2005]. Our empirical evaluation shows that such a portfolio can solve significantly more problem instances than GAC, AllSol, or PerTuple alone, and in less runtime on average.

The paper is organized as follows. Section 2 reviews some necessary background material. Section 3 outlines related work. Section 4 discusses enforcing minimality on the clusters of a tree decomposition. Section 5 discusses the construction of the portfolio. Section 6 describes experimental evaluations of the portfolio on benchmark problems. Finally, Section 7 presents our conclusions.

2 Background

Here we provide a summary of the background material relevant to our work.

2.1 The Constraint Satisfaction Problem

The *Constraint Satisfaction Problem* (CSP) is denoted by $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X} = \{x_1, ..., x_n\}$ is a set of n variables, each associated with a finite domain from $\mathcal{D} = \{D_1, ..., D_n\}$. $\mathcal{C} = \{C_1, ..., C_e\}$ is the set of constraints restricting how values may be assigned to variables. Each constraint covers some subset of the variables, known as the scope of the constraint. A solution to a CSP is an assignment to each variable a value from its domain such that all constraints are satisfied. Deciding the existence of a solution is an NP-complete problem.

A constraint C_i is defined by relation R_i over the $scope(C_i)$. In this paper, we consider relations expressed as a set of allowed tuples.[1] Each relation R_i is a subset of the Cartesian product of the domains of the variables in the $scope(C_i)$. Each tuple in the relations represents an assignment of values to the respective variables that is consistent with the constraint. A set of variable assignments constitutes a consistent partial solution if it satisfies all the constraints defined on the variables. The arity of a constraint is the cardinality of its scope. In a binary CSP, the constraints have arity two.

Several graphical representations of a CSP exist. The constraint network of a binary CSP is a graph where the vertices represent the variables and the edges the binary constraints. The constraint network of a non-binary CSP is a *hypergraph*. In the hypergraph, the vertices represent the variables and the hyperedges the scopes of the constraints (Figure 1). In the *primal graph*, the vertices represent the variables, and the edges connect every two variables that appear in the scope of some constraint (Figure 2). In the *dual graph*, the vertices represent the constraints of the CSP, and the edges connect vertices corresponding to constraints whose scopes overlap (see Figure 3).

In fact, the dual graph represents a *dual* CSP (of the original CSP), where the vertices represent dual variables whose domains are the tuples allowed by the corresponding original relations. The edges of the dual graph are equality constraints, indicating that tuples assigned to two dual variables must necessarily agree on the values given to the CSP variables that appear on the edge label. A feature of the dual CSP is that all of its constraints are binary. Finally, the *incidence graph* of a CSP is a bipartite graph where one set contains all the variables and the other all the constraints (Figure 4).

[1] Our approach remains applicable to constraints defined in intension. For example, Bayer et al. enforce domain minimality in the presence of constraints defined in intension and also global constraints [Bayer *et al.*, 2007].

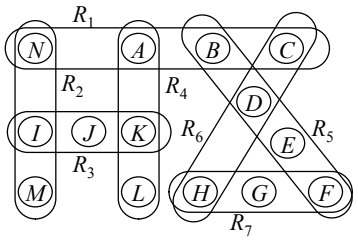

Figure 1: Hyper

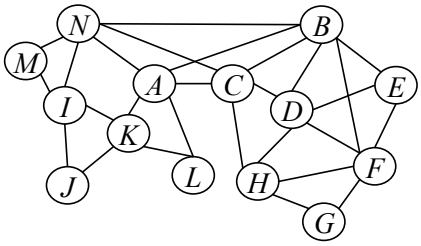

Figure 2: Primal

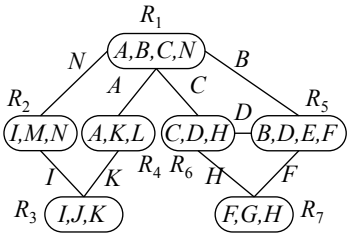

Figure 3: Dual

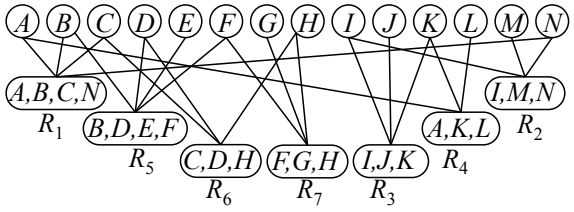

Figure 4: Incidence

An edge connects a variable and constraint if the variable appears in the scope of the constraint. The incidence graph is the same graph used in the hidden-variable encoding [Rossi *et al.*, 1990].

2.2 Tree Decomposition

A *tree decomposition* of a CSP is a tree embedding of its constraint network. The tree nodes are *clusters* of variables and constraints from the CSP. A tree decomposition must satisfy two conditions: *a*) each constraint appears in at least one cluster and the variables in its scope must appear in this cluster; and *b*) for every variable, the clusters where the variable appears induce a connected subtree. Many techniques for generating a tree decomposition of a CSP exist [Dechter and Pearl, 1989; Jeavons *et al.*, 1994; Gottlob *et al.*, 1999]. We use an adaption for non-binary CSPs of the tree-clustering technique [Dechter and Pearl, 1989]. First, we triangulate the primal graph of the CSP using the min-fill heuristic [Kjærulff, 1990]. Then, we identify the maximal cliques in the resulting chordal graph using the MaxCliques algorithm [Golumbic, 1980], and use the identified maximal cliques to form the clusters of the tree decomposition. We build the tree by connecting the

clusters using the JoinTree algorithm [Dechter, 2003]. While any cluster can be chosen as the root of the tree, we choose the cluster that minimizes the longest chain from the root to a leaf. Figure 5 shows a triangulated primal graph of the example in Figure 2. The dotted edges (B, H) and (A, I) in

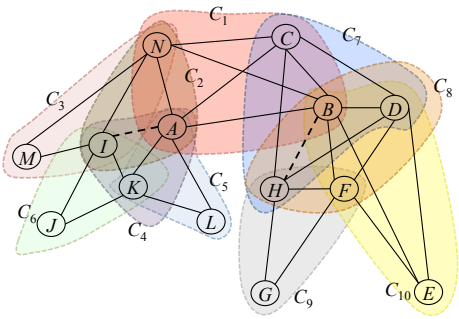

Figure 5: Triangulated primal graph and its maximal cliques

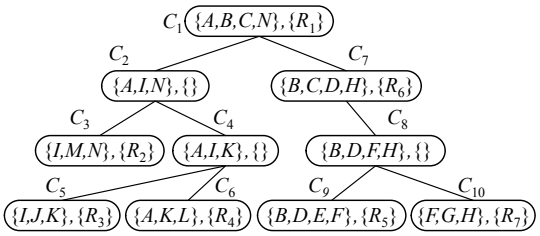

Figure 6: Tree decomposition

Figure 5 are fill-in edges generated by the triangulation algorithm. The ten maximal cliques of the triangulated graph are highlighted with 'blobs.' The resulting tree decomposition is shown in Figure 6.

A *separator* of two adjacent clusters is the set of variables that are associated with both clusters. A given tree decomposition is characterized by its *treewidth*, which is the maximum number of variables in a cluster minus one. The complexity of solving a CSP using a given tree decomposition can be bound in time by the treewidth of the decomposition and in space by the size of its largest separator. The *treewidth of a constraint network* is the minimum treewidth of all its decompositions; computing it is known to be NP-hard [Arnborg, 1985]. In order to guarantee perfect 'message passing' across clusters, one would have to generate a unique constraint over all the variables of a separator, which is prohibitively expensive in space. As an approximation, and in order to enhance constraint propagation between two adjacent clusters, we use the projection schema described by Karakashian et al. [Karakashian *et al.*, 2013]. According to this bolstering strategy, we add to each cluster the projection on the variables of the cluster of all the constraints (from outside the cluster), then we normalize the constraints in the cluster by merging all two constraints where the scope of one is a subset the other's.

2.3 Consistency Properties

We first discuss global consistency properties, then local ones. A CSP is said to be *globally consistent* iff every consistent instantiation of any number of variables can be extended to a complete solution [Dechter, 2003]. This property is also called *decomposability* [Dechter *et al.*, 1991] and strong n-consistency [Freuder, 1978]. It guarantees not only the existence of a solution, but also that the solution can be found in a backtrack-free manner, thus, tractability. *Minimality*, a strictly weaker property than global consistency, requires that any tuple that satisfies a constraint appears in at least one solution to the CSP [Montanari, 1974]. (Thus, the constraints are minimal.) Gottlob argued that although computing the minimal network and solving the minimal CSP are both NP-hard, the former is still useful for knowledge compilation [Gottlob, 2011]. Minimality, which was initially proposed as a property of the constraints of a CSP, can also be used for the variables' domains, which are in fact unary constraints.[2] Finally, *consistency* (i.e., satisfiability) guarantees that the CSP has a solution.

In contrast to global properties, local properties focus on subproblems of a fixed size (e.g., every subset of k variables or every subset of m constraints). Thus, they can usually be efficiently enforced (i.e., in polynomial time) while still yielding substantial filtering of inconsistent values/tuples. *Arc consistency* (AC) is the most common local property: A CSP is AC if every value has a supporting value in all neighboring variables. A similar property for non-binary constraints is *generalized arc consistency* (GAC) [Mackworth, 1977]. Our experiments use the GAC-2001 algorithm [Bessière *et al.*, 2005].

2.4 Algorithms for enforcing minimality

Bayer et al. enforce domain minimality in two different ways: enumerating all solutions or finding a single solution for every variable-value pair in the CSP [Bayer *et al.*, 2007]. They compare the effectiveness of the two algorithms for a spatial-reasoning application where constraints are defined in extension, intension, as well as a global all-diff constraint [Régin, 1994]. Bessiere et al. report a similar algorithm in the context of an interactive configuration application [Bessiere *et al.*, 2013]. Karakashian et al. introduce the PerTuple algorithm, which enforces *any level* of relational consistency up

[2]Domain minimality is also called global inverse consistency [Bessiere *et al.*, 2013].

to constraint minimality (then domain minimality by projection on the domains) [Karakashian *et al.*, 2010; 2013]. They demonstrate the effectiveness of PerTuple for lookahead during search and its ability to solve difficult classes of CSPs in a backtrack-free manner.

The two algorithms, AllSol and PerTuple, proposed by Karakashian [Karakashian, 2013], which both compute the minimal relations, can be thought of as the relational versions of the two domain-minimality algorithms of Bayer et al. [Bayer *et al.*, 2007]. Our portfolio exploits AllSol and PerTuple. To compute the minimal constraints, PerTuple performs a backtrack search on every tuple in every relation, trying to consistently extend it to a tuple in each other relation in the CSP. The process is illustrated in Figure 7. If the search fails, the tuple is removed. Otherwise, the

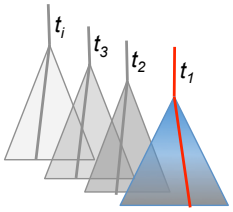

Figure 7: PerTuple performs a new solution search for every tuple

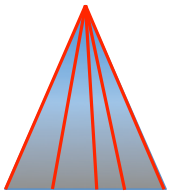

Figure 8: AllSol performs a single exhaustive search, finding all solutions

search stops after finding the first solution. Further, the solution is used as a support structure for all the tuples that appear in the solution, which are marked as 'minimal.' In contrast, AllSol conducts a single backtrack search over the tuples of the relations, finding all the solutions and marking as 'minimal' every tuple that appears in any solution (see Figure 8). If PerTuple is interrupted at any point, any deleted tuple is guaranteed to be inconsistent. However, when interrupted, the effort invested by AllSol is lost. Whereas the space used for storing support structures in PerTuple constitutes a tradeoff between time and space, AllSol does not incur such an overhead.

Intuitively, AllSol may perform better when solutions are very sparse and searching for even a single solution is a costly process. PerTuple may perform better when solutions are plentiful and costly to enumerate. In practice, their performance is complex and difficult to predict.

3 Related Work

Lately, there has been much research for 'opportunistically' choosing the most appropriate 'method' for solving a CSP. We distinguish two main directions, namely, algorithm portfolio and dynamic selection of consistency properties. While portfolio approaches rely on machine-learning classifiers to select the algorithm best suited to a given instance, the second direction chooses between two or more consistency properties to enforce for lookahead *during* search.

The portfolio approach has roots in the Algorithm Selection Problem [Rice, 1976], which involves selecting the best algorithm to apply to a particular instance to maximize some performance metric. Early work by Gomes and Selman used a portfolio of several algorithms running in parallel to exploit their complementarity [Gomes and Selman, 2001]. Portfolios gained in popularity for both SAT and CSPs through solver competitions, with SATZilla [Xu *et al.*, 2008] winning the SAT Challenge 2012 and CPHydra [O'Mahony *et al.*, 2008] winning the 2008 Constraint Solver Competition. More recent use of portfolio techniques include the solvers Proteus [Hurley *et al.*, 2014] and `sunny-cp2` [Amadini *et al.*, 2015]. Proteus' portfolio selects both a solver and a problem representation, converting the problem to one of several SAT/CSP encodings. `sunny-cp2` selects a schedule of multiple solvers from a portfolio to run in parallel that can cooperate and outperform the individual solvers. Our approach is perhaps the first one to adopt a portfolio strategy for choosing between consistency algorithms.

As for dynamically enforcing different levels of consistency, the earliest approach is perhaps the adaptive-consistency algorithm of Dechter and Pearl [Dechter and Pearl, 1988], which adapts the level of consistency enforced to the number of the parents of a variable in an ordering of the variables to guarantee backtrack-free search. The Adaptive Constraint Engine of Epstein et al. trains on a set of problems and learns a propagation policy of several consistency algorithms and parameters specifying how and when they are applied to the problems [Epstein *et al.*, 2005]. Several approaches have been proposed to choose between two or more levels of consistency during search. Such approaches are probabilistic [Mehta and van Dongen, 2007], heuristic [Stergiou, 2009; Paparrizou and Stergiou, 2012], adaptive [Balafrej *et al.*, 2013; Woodward *et al.*, 2014], or based on multi-armed bandits [Balafrej *et al.*, 2015]. An orthogonal approach was adopted by Woodward et al. where the topology of the dual graph of a CSP is modified before search resulting in

different filtering levels of the same consistency algorithm [Woodward *et al.*, 2011]. Our research is likely the first one to consider algorithms for minimality and to operate dynamically on the clusters of a tree decomposition.

4 Enforcing Minimality in a Tree Decomposition

Karakashian never compared the performance of PerTuple and AllSol during search, but only on individual clusters [Karakashian, 2013] collected from tree decompositions of CSP instances. When exploiting a tree decomposition for lookahead, Karakashian et al. exclusively used PerTuple [Karakashian *et al.*, 2013]. We consider three variations of their basic process:

1. During lookahead, we include the option of whether or not to enforce GAC over the entire CSP prior to processing any given cluster.

2. Every time a cluster is considered for consistency, we can call a specific consistency algorithm, or use a classifier to determine whether to call AllSol or PerTuple on the cluster or to do 'Neither.'

3. Finally, we include an optional 'timeout' setting for processing individual clusters. This timeout interrupts the consistency algorithm currently operating on the cluster when the set threshold is reached. In the case of GAC and PerTuple, the filtering done so far is preserved. For AllSol, it is lost.

FilterClusters (Algorithm 1) implements the above strategies and controls how consistency is enforced and propagated. In addition to the clusters, FilterClusters takes three parameters that implement the above described variations of the basic process. Table 1 lists the parameter settings that yield six algorithms. Other variations were tested, but these were selected to show the most meaningful comparisons.

FilterClusters filters both the domains of the variables and the tuples of the relations. It may be applied as a preprocessing step as well as a lookahead procedure during search. The foreach loop (line 6) processes clusters from the specified *clusterOrder*. In our setting, this ordering corresponds to the MaxCliques ordering of the clusters (see Section 2.2), but we will investigate other priority orderings in the future. The outer while-loop (line 4)

Table 1: Parameter variations of FilterClusters

Algorithm	*classifier*	*interleaveGAC*	*timeout*
AllSol	Always select 'AllSol'	$false$	∞
PerTuple	Always select 'PerTuple'	$false$	∞
AllSol$^+$	Always select 'AllSol'	$true$	1 (s)
PerTuple$^+$	Always select 'PerTuple'	$true$	1 (s)
Random	Randomly select 'AllSol', 'PerTuple', or 'Neither'	$true$	1 (s)
DecTree	Decision tree selects 'AllSol', 'PerTuple', or 'Neither'	$true$	1 (s)

iterates until no further filtering can be achieved. At each pass, the direction of the cluster ordering is reversed to facilitate propagation (line 19). The classifier allows the selection of the most appropriate algorithm on a cluster by cluster basis (line 11). The option to run GAC (i.e., $interleaveGAC = true$) allows 'easy and quick' filtering, which may trigger rapid and effective propagation throughout the problem. The *timeout* option, which specifies a time limit in seconds, ensures that excessive time is not wasted on a single cluster, allowing us to quickly recover from an unfortunate classification error.

5 Building a Portfolio

Our algorithm portfolio must decide which of the two minimality algorithms to enforce on a cluster given that the performance of the two algorithms vary widely. Because both algorithms enforce the same consistency, the portfolio must select the fastest algorithm based on features extracted from the cluster being processed or, when both algorithms are too costly, it must choose to run neither.

In order to train a classifier for the portfolio, we collected a large data set of runtimes for both algorithms. We took instances from 175 benchmarks and broke them down into clusters of a tree decomposition. We then sampled 9362 individual instances from these clusters, either randomly selecting 70 clusters (70 provided a manageable set of training data to collect) from each of the 175 benchmarks or taking all the clusters of a benchmark when it has

Algorithm 1: FilterClusters(*clusterOrder*, *classifier*, *interleaveGAC*, *timeout*)

```
   Input: clusterOrder,classifier, interleaveGAC, timeout
   Output: Entire problem is GAC with potentially minimal clusters
 1 didFiltering ← true
 2 passDidFiltering ← true
 3 consistent ← true
 4 (consistent, didFiltering) ← GAC()
 5 if consistent = false then return false
 6 while passDidFiltering do
 7     passDidFiltering ← false
 8     foreach cluster ∈ clusterOrder do
 9         algo ← Classify(cluster, classifier)
10         if algo ='AllSol' then
11             (consistent, didFiltering) ← AllSol(cluster, timeout)
12         else if algo ='PerTuple' then
13             (consistent, didFiltering) ← PerTuple(cluster, timeout)
14         else didFiltering ← false
15         if consistent = false then return false
16         if didFiltering then passDidFiltering ← true
17         if interleaveGAC and didFiltering then
18             (consistent, didFiltering) ← GAC()
19             if consistent = false then return false
20     clusterOrder ← Reverse(clusterOrder)
21 if interleaveGAC = false then
22     (consistent, didFiltering) ← GAC()
23     if consistent = false then return false
24 return true
```

less than 70 clusters. We ran both AllSol and PerTuple on every cluster, while recording (for each cluster) a set of features (see Section 5.1) as well as the runtimes of the algorithms. Figure 9 shows the runtime distribution of the training instances. Although there are substantially more instances favoring PerTuple, AllSol does have its niche of instances on which it completes in up to two orders of magnitude faster.

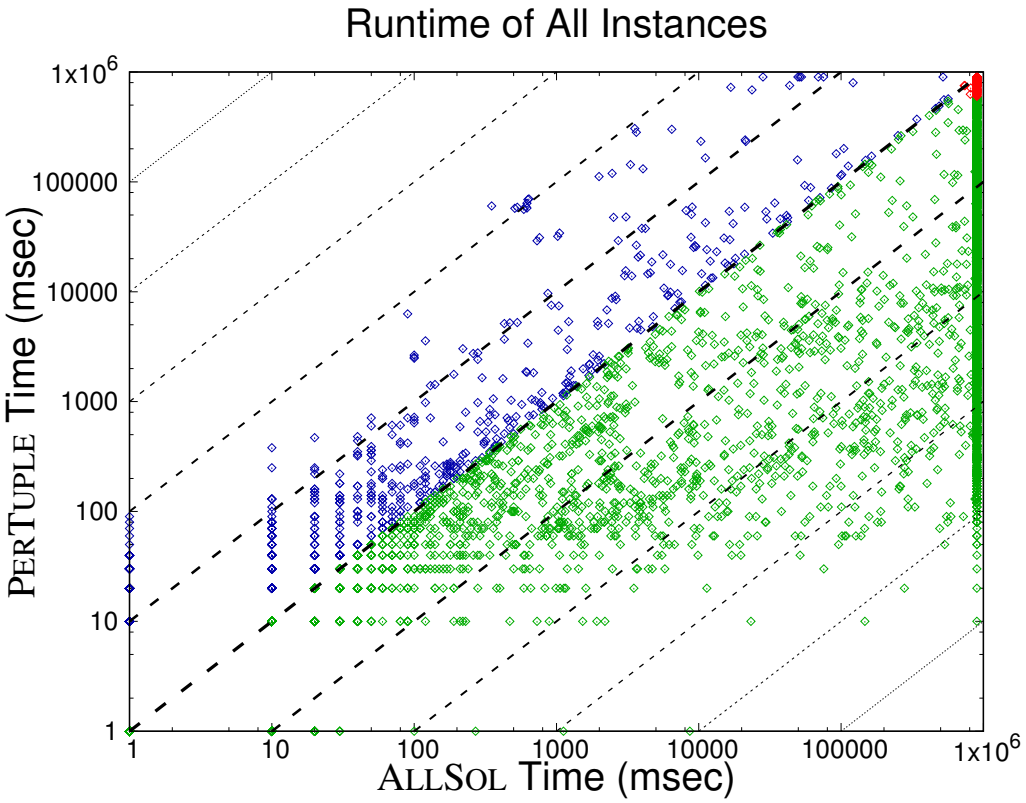

Figure 9: Distribution of algorithm runtimes on single clusters

5.1 Features

Inspired by features that appeared in the literature [Karakashian *et al.*, 2012; Amadini *et al.*, 2014], we identified a selection of 73 features that attempt to capture the constraint-network structure and the relation properties of a problem instance. The majority of the features that we collect are aggregations of many data points. In general, we aggregate using the mean, *coefficient of variation* (CV), minimum, maximum, and *entropy*. The coefficient of variation is the ratio of the standard deviation to the mean. The entropy of a multiset $\mathcal{X} = \langle X, m \rangle$ (the set X is the possible values in $\mathcal{X}$ and for all $x \in X$, $m(x)$ is the multiplicity of x in $\mathcal{X}$) is calculated by $H(\mathcal{X}) = -\sum_{x \in X} \frac{m(x)}{|\mathcal{X}|} \log\left(\frac{m(x)}{|\mathcal{X}|}\right)$. For the number of tuples and values, we also report a total sum. For relational linkage (defined below), we report the

$\log_{10}$ of the mean as well. We first list the collected features, then explain them:

- *CSP parameters*: number of variables; number of relations; number of tuples per relation (total, mean, CV, min, max, entropy); domain size (total, mean, CV, min, max, entropy); arity of relations (mean, CV, min, max, entropy); tightness of relations (mean, CV, min, max, entropy); relational linkage ($\log_{10}$ mean, mean, CV, min, max, entropy).

- *Dual-graph parameters*: density; vertex degree (mean, CV, min, max, entropy); vertex eccentricity (mean, CV, min, max, entropy); vertex-clustering coefficient (mean, CV, min, max, entropy).

- *Incidence-graph parameters*: density; vertex degree (mean, CV, min, max, entropy); vertex eccentricity (mean, CV, min, max, entropy).

- *Primal-graph parameters*: density; vertex degree (mean, CV, min, max, entropy); vertex eccentricity (mean, CV, min, max, entropy); vertex-clustering coefficient (mean, CV, min, max, entropy).

The first five features are obvious. The tightness is the ratio of conflicting tuples over the product of the domain sizes of the variables in the scope of the constraint. The relational linkage estimates how likely a tuple at the overlap of two relations is to appear in a solution, and is computed as follows. For every two relations R_i, R_j, let $V_{ij} = scope(R_i) \cap scope(R_j)$. $\forall R_k, scope(R_k) \supseteq V_{ij}, \forall x \in scope(R_k) \setminus V_{ij}$, the relational linkage of every tuple $t \in \pi_{V_{ij}}(R_i \bowtie R_j)$ is computed as $min_{R_k}\left(\frac{|\sigma_t(R_k)|}{\prod_x |domain(x)|}\right)$ where $\bowtie, \sigma, \pi$ are the relational operators join, selection, and projection, respectively.

We consider structural properties of three types of graphs (i.e., the dual, primal, and incidence), collecting the density, degree, eccentricity, and (except for the incidence graph) clustering coefficient. Eccentricity of a vertex is the longest shortest-path to another vertex. Its clustering coefficient is the edge density of the vertex's induced neighborhood (excluding edges incident to itself).

5.2 Classifier

After collecting data from 9362 individual clusters, we trained a decision tree using the J48 algorithm of the Weka machine learning software suite [Hall

et al., 2009]. Using the 9362 clusters as a training set, we labeled each cluster 'AllSol' when AllSol was the fastest, 'PerTuple' when PerTuple was fastest, and 'Neither' when neither algorithm completed within ten minutes. Further, we weighted our instances to increase the importance of instances with a large difference in runtimes, computing the weight of an instance i using Equation (1) where $allSol(i), perTuple(i)$ are the running time on i of AllSol and PerTuple, respectively.

$$weight(i) = \left\lceil \left| \log_{10} \left(\frac{allSol(i)}{perTuple(i)} \right) \right| \cdot \left| \log_{10} \left(|allSol(i) - perTuple(i)| + 0.01 \right) \right| \right\rceil \tag{1}$$

We designed this weighting scheme to give more importance to instances where the execution times differ greatly, both in the ratio and in the difference of their values. After weighting the 'AllSol' and 'PerTuple' instances, we computed their average weight to be 6. We then chose a weight of 20 for the 'Neither' instances to emphasize the importance of skipping costly clusters.

Table 2 shows the results of evaluating our classifier (trained on the weighted data) by performing a ten-fold cross validation on both the weighted and unweighted (all weight 1) instances. The F-Measure ranges from 0 to 1, with 1 being perfect classification. It is calculated by $F = \frac{2TP}{2TP+FP+FN}$ where TP (true positives) is the number of instances of the given class that are correctly labeled, FP (false positives) is the number of instances from other classes that are incorrectly labeled as the given class, and FN (false negatives) is the number of instances of the given class that were incorrectly labeled as other classes. The fact that the results of the weighted data set

Table 2: Performance of the decision tree classifier

	weighted	**unweighted**
Accuracy	90.8%	80.1%
F-Measure		
'AllSol'	0.50	0.40
'PerTuple'	0.89	0.85
'Neither'	0.93	0.93

are better than those of the unweighted data set clearly indicates that the classifier is correctly handling the more important instances. The values of

the F-Measure of 'PerTuple' and 'Neither' are both near 1, however, that of 'AllSol' is quite small, which can likely be explained by the relatively reduced number and small weights of the 'AllSol'-labeled instances.

6 Experimental Evaluation

In our evaluation, we use the six algorithms obtained by setting the parameters of FilterClusters alongside GAC for real-full lookahead [Haralick and Elliott, 1980] in a backtrack search on a set of 1055 instances taken from 42 benchmarks from the XCSP library, which includes a mix of random, academic, and real-world instances.[3] Our search procedure terminates after finding the first solution and uses the dom/deg dynamic ordering heuristic. Although dom/wdeg [Boussemart *et al.*, 2004] can improve performance across the board, FilterClusters is not yet equipped to take advantage of this heuristic. Our experiments run on a cluster computer with Intel Xeon E5-2650 v3 2.30GHz processors. Search is allowed to run for two hours (7200 sec) and given 12 GB of memory. To account for load variations on the cluster computer, we measure instruction count and convert it to runtime using a standardized measure of instructions per cycle and clock speed. We use a timeout of 1 second per cluster because, based on the data from the 9362 clusters shown in Figure 9, this value strikes a good balance between completing clusters and not spending excessive time on any one cluster.

6.1 Results

Figure 10 shows the number of instances completed by each solver as the time increases. The ordering of the lines generally remains consistent throughout the time axis and the instance completion gap expands proportionally. Exceptions include: GAC starts ahead on easy problems but has fewer completions on difficult instances and Random steadily rises as the stabilizing effect of the per-cluster timeout becomes stronger over a longer total runtime.

Table 3 summarizes the results (top) on 1055 benchmark instances and provides detailed results per benchmark (bottom). In this table,

- We consider only instances where at least one solver completes.

[3] `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

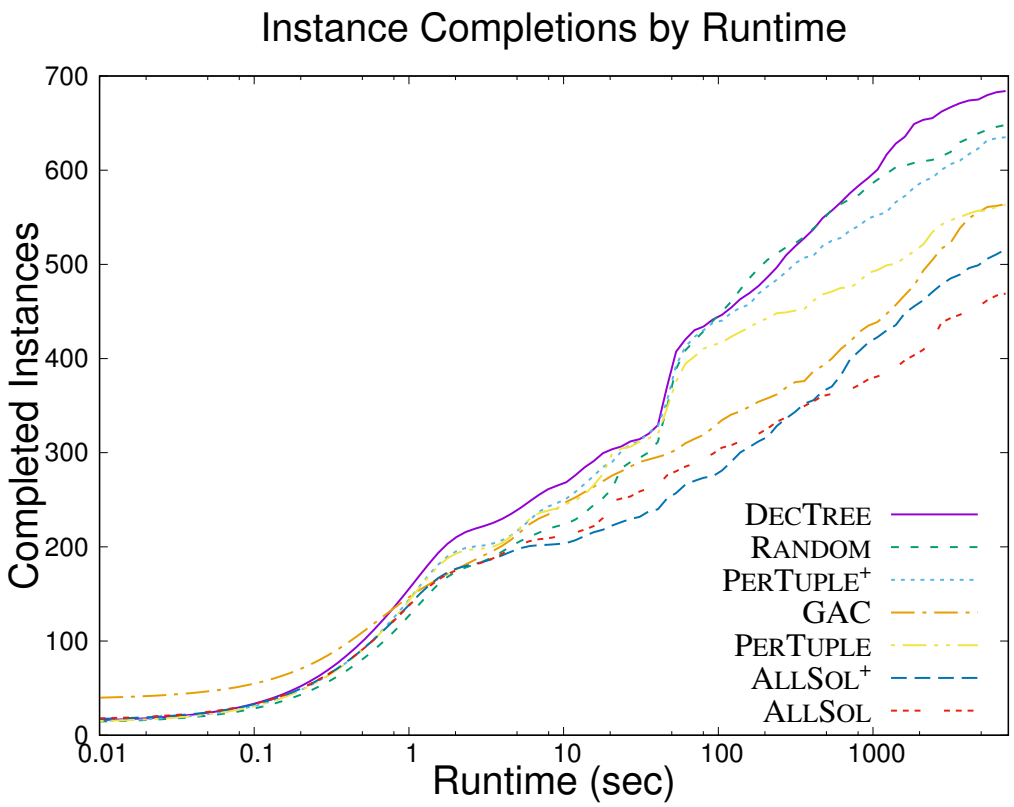

Figure 10: Instance completions over time

- The number of instances per benchmark is denoted by 'solved by one/total'.
- 'Data Summary' gives the number of completions, average, and total CPU time. The lower sections give the number of completions and average CPU time per benchmark.
- '>' indicates that at least one instance did not complete. '⋆' indicates that at least one instance caused a memout. The '>' and '⋆' are omitted from 'Data Summary' for readability.
- Columns 'A%,' 'P%,' and 'N%' indicate DecTree's average percentage of selecting 'AllSol,' 'PerTuple,' and 'Neither,' respectively.
- The columns do not always sum to 100 due to rounding.

Table 3: Experimental results of the seven solvers

	GAC	AllSol	PerTuple	AllSol$^+$	PerTuple$^+$	Random	DecTree			
Data Summary										
#Completed 770/1055	550	472	567	514	633	643	**685**			
Average CPU time	2,471.6	3,075.3	2,081.9	2,789.4	1,622.7	1,427.4	**1,121.3**			
Sum of CPU time	1,900,653.4	2,364,878.9	1,601,010.4	2,145,062.1	1,247,840.7	1,097,633.8	**862,259.9**			
Benchmark	Hybrid solvers are best							A%	P%	N%
aim-100 21/24	17 >1,857.1	11 >3,984.0	20 >631.0	11 >3,741.4	21 559.3	16 >1,754.1	21 **512.4**	0	99	1
aim-200 17/24	8 >3,942.3	2 >6,494.0	8 >3,815.9	6 >5,208.7	10 >3,166.8	3 >5,990.8	14 **>1,647.3**	0	92	8
cmpsd-25-1-25 10/10	0 >7,200.0	9 >720.1	10 11.7	10 53.7	10 18.4	10 **0.1**	10 **0.1**	0	100	0
cmpsd-25-1-40 10/10	0 >7,200.0	8 >1,440.1	10 36.4	10 121.6	10 50.0	10 **0.1**	10 **0.1**	0	100	0
cmpsd-25-1-80 10/10	4 >4,445.6	6 >3,046.1	10 24.2	10 205.2	10 33.8	10 **2.2**	10 6.7	0	100	0
cmpsd-25-10-20 10/10	6 >2,892.7	0 >7,200.0	9 >2,208.1	0 >7,200.0	10 2,821.4	10 **404.8**	9 >2,202.6	0	96	4
cmpsd-75-1-25 10/10	0 >7,200.0	8 >1,440.4	8 >1,440.5	10 212.8	10 217.5	10 **3.1**	10 11.9	0	92	8
cmpsd-75-1-40 10/10	0 >7,200.0	6 >2,880.4	6 >2,880.4	10 611.5	10 454.0	10 **5.6**	10 64.4	0	93	7
cmpsd-75-1-80 10/10	3 >5,040.0	1 >6,480.1	2 >5,761.1	9 >2,241.4	10 1,176.6	10 **15.9**	10 123.5	0	99	1
cril 6/8	3 >3,968.4	3 >3,605.2	3⋆>3,604.8	3 >3,606.0	4⋆>**2,459.2**	4⋆>2,999.9	3 ⋆>3,604.9	2	66	32
ehi-90 100/100	84 >2,372.2	43 >4,456.8	72 >2,103.5	28 >5,259.4	81 >1,484.3	100 **61.2**	100 136.5	0	98	2
GC-hos 10/14	6 >2,882.3	0⋆>7,200.0	3⋆>5,129.9	2⋆>6,360.4	7 ⋆>3,401.4	8⋆>2,693.5	8⋆>**2,309.2**	0	98	2
GC-full-ins 24/41	17 >2,105.7	4⋆>6,004.0	17⋆>2,440.0	8⋆>5,266.7	18 ⋆>2,146.5	15⋆>3,008.7	22⋆>**1,010.9**	0	99	1
GC-mug 8/8	4 >3,600.0	6 >2,182.2	6 >2,156.0	8 47.8	8 **41.5**	4 >3,600.0	8 102.6	0	97	3
pseudo-aim 42/48	25 >2,917.5	20 >3,867.3	28 >2,406.8	24 >3,676.8	37 >1,054.0	28 >2,515.5	42 **265.4**	0	94	6
QCP-15 15/15	10 >3,023.7	2 >6,241.1	2 >6,241.4	2 >6,250.4	3 >6,041.4	8 >3,973.8	15 **533.3**	0	80	20
rand-8-20-5 20/20	19 >1,532.7	3 >6,551.8	0 >7,200.0	18 >2,333.3	3 >6,811.0	20 **587.8**	20 **605.2**	35	59	7
rlfapGraphsMod 11/12	5 >3,975.5	4 >4,582.2	5 >4,180.5	7 >4,015.7	9 >1,878.8	11 **843.3**	8 >2,043.0	0	88	12
rlfapScens11 7/12	0 >7,200.0	3 >4,199.1	4 >3,373.2	5 >3,528.1	7 **1,016.4**	6 >1,371.2	1 >6,183.0	15	60	25
rlfapScensMod 13/13	7 >3,323.4	8 >3,103.4	9 >2,316.1	8 >3,227.5	10 >2,008.9	12 >**1,249.1**	10 >2,227.1	7	81	12
No clear winner										
aim-50 24/24	24 **0.6**	24 6.2	24 2.3	24 53.9	24 **0.7**	24 4.7	24 **0.6**	0	100	0
cmpsd-25-1-2 10/10	0 >7,200.0	10 **0.1**	10 **0.1**	10 **0.1**	10 **0.1**	10 **0.1**	9 >720.1	0	94	6
cmpsd-75-1-2 10/10	0 >7,200.0	10 **0.5**	10 **0.6**	10 **0.5**	10 **0.6**	10 **0.6**	10 **0.6**	0	86	14
hanoi 5/5	5 **1.8**	5 **2.5**	5 **2.5**	5 **2.5**	5 **2.5**	5 **2.5**	5 **2.5**	0	100	0
knights 11/19	10 >**1,098.3**	7 >2,716.0	8 >2,485.4	10 >**1,144.4**	10 >**1,138.0**	10 >**1,128.6**	10 >**1,131.3**	0	64	36
modRenault 50/50	27 >3,439.2	50 **2.1**	50 **3.1**	50 **2.4**	50 3.2	50 **2.6**	48 >290.7	12	84	3
rand-10-20-10 20/20	20 3.7	20 **1.0**	20 **1.1**	20 **1.0**	20 **1.1**	20 **1.1**	20 **1.3**	0	100	0
ssa 7/8	6 >**1,029.3**	6 >**1,058.6**	6 >**1,058.8**	6 >**1,052.3**	6 >**1,052.5**	5 >2,058.1	6 >**1,065.3**	0	96	4
Basic solvers are best										
dag-rand 25/25	25 2,467.6	25 **21.0**	25 **21.7**	25 45.2	25 38.2	25 24.7	24 >1,423.5	3	96	1
dubois 7/13	7 **1,959.6**	6 >2,191.7	7 2,099.1	6 >2,175.1	6 >2,085.8	5 >3,388.8	6 >2,457.6	0	100	0
GC-reg-fpsol 8/37	6 >**1,814.4**	4⋆>4,237.4	4⋆>4,238.9	2⋆>5,407.9	2 ⋆>5,408.1	2⋆>5,408.1	2 ⋆>5,408.2	0	99	1
GC-reg-inithx 7/32	5 >**2,129.3**	4⋆>3,915.8	2⋆>5,160.2	2⋆>5,159.6	2 ⋆>5,160.1	2⋆>5,160.2	2 ⋆>5,160.1	0	100	0
GC-reg-mulsol 13/49	9 >**2,218.0**	9 >2,928.4	9⋆>2,928.9	5 >4,440.2	5 ⋆>4,440.5	5⋆>4,440.5	5 ⋆>4,440.5	0	99	1
GC-reg-zeroin 8/31	6 >**1,801.6**	5 >3,249.9	5⋆>3,251.2	3 >4,519.0	3 ⋆>4,519.5	3⋆>4,519.5	3 ⋆>4,519.6	0	90	10
GC-sgb-book 23/26	18 >1,818.3	19 >1,657.5	23 **256.2**	16 >3,106.7	22 >737.5	20 >1,321.6	22 >657.4	0	94	6
GC-sgb-games 4/4	2 >3,600.2	2 >3,600.2	4 **26.0**	2 >3,600.2	4 46.4	3 >1,999.1	4 46.4	0	99	1
GC-sgb-miles 13/42	11 >**1,411.7**	9⋆>2,749.8	8⋆>3,109.3	6⋆>3,902.3	6 ⋆>3,883.6	7⋆>3,488.3	7 ⋆>3,652.3	0	87	13
GC-sgb-queen 14/50	10 >**2,619.2**	6 >4,676.4	7⋆>3,874.8	3 >5,852.4	6 ⋆>4,421.7	6⋆>4,315.4	9 ⋆>3,415.8	0	76	24
haystacks 8/51	5 >2,786.7	7 >1,055.8	8 **228.6**	5 >2,700.9	7 >1,043.3	5 >2,716.0	7 >934.3	0	100	0
marc 10/10	10 **16.8**	10 253.6	0⋆>7,200.0	10 1,321.7	0 ⋆>7,200.0	0⋆>7,200.0	0 ⋆>7,200.0	-	-	-
os-taillard-4 29/30	27 >**887.8**	2 >6,704.7	2 >6,704.7	21 >2,427.0	24 >2,967.6	23 >1,876.7	23 >2,681.4	15	83	1
tightness0.9 99/100	99 **352.6**	85 >1,946.3	98 >489.7	84 >1,950.5	98 >561.8	98 >741.6	98 >549.5	0	99	0

In Table 3, we place the solvers into two categories. On one hand, the basic solvers, which include GAC, AllSol, and PerTuple. On the other hand, hybrid solvers, which include AllSol$^+$, PerTuple$^+$, Random, and DecTree. We compute the average CPU time only over instances completed by at least one of the solvers. Both timeouts and memouts (memory out) are considered 7200 seconds. Due to the randomness of Random, we perform ten runs for each instance and report the median.

Overall, it is clear that DecTree outperforms all solvers both in terms of the number of completed instances, and average and sum CPU time. It solves instances 2.2 times faster than GAC on average, and completes 135 more instances than GAC out of the 1055 tested.

Random is surprisingly competitive with DecTree. This fact is largely due to the stabilizing effect of the per-cluster timeout, which minimizes the time loss from poor classification decisions. We ran an experiment to assess the extent of this effect. We compared the performance of Random and DecTree with no per-cluster timeout. Random completes only 484 instances whereas DecTree completes 649, with an average CPU time across all instances completed by at least one solver of 2,955.9 seconds and 1,413.9 seconds, respectively. Thus, DecTree makes substantially better decisions than a random choice.

The lower sections of Table 3 break down the performance of the seven solvers by benchmark. We identify three categories: benchmarks where the hybrid solvers outperform all others (top), those where hybrid and basic solvers perform equally well (middle), and finally those on which the basic solvers perform best (bottom). For each benchmark, we format in bold the smallest average runtime and all runtimes within 1 second or 5% of the best time.

In the top category, DecTree and Random are generally the best, but are outperformed by PerTuple$^+$ on two benchmarks. The benchmarks in the middle category seem to be solvable relatively fast by most solvers and have few timeouts (except GAC). The basic solvers outperform the others on the benchmarks in the bottom-most category. Those benchmarks tend to be memory intensive: indeed PerTuple, PerTuple$^+$, and DecTree have many memouts.

GAC never has memouts because it uses relatively light support structures. AllSol has a handful of memouts due to the added relations to bolster propagation at the separators. PerTuple uses more memory than AllSol for data structures to record and maintain tuple supports, and suf-

fers from more memouts than AllSol. In our current implementation, all solvers that can call PerTuple prepare for the *option* of running PerTuple by creating the support data-structures for the tuples, at startup. As a result, the memout occurs before the solver has even had the option to avoid calling PerTuple. In the bottom-most category (i.e., the basic solvers), several benchmarks, such as marc,[4] have huge memory requirements allowing GAC to outperform the hybrid solvers.

The right-most columns in Table 3 (i.e., A%, P%, and N%) show, as a percentage, the average number of times the classifier returns the labels 'AllSol,' 'PerTuple,' and 'Neither,' respectively.[5] Although DecTree heavily favors 'PerTuple,' it chooses 'AllSol' and 'Neither' when necessary. The three benchmarks cmpsd-25-1-25|40|80 exhibit an interesting anomaly: DecTree selects 100% 'PerTuple.' One would expect the performance of DecTree to be close to that of PerTuple$^+$. In fact, the former significantly outperforms the latter. This difference is due to non-deterministic cutoffs resulting from the per-cluster timeout. Indeed, a close examination of the results reveals that, in PerTuple$^+$, a couple of instances hit the cutoff prior to reaching a 'vital' filtering stage.

6.2 Shapley Value

As an additional evaluation method, we calculate the Shapley Value [Shapley, 1952; Fréchette *et al.*, 2016] of the seven solvers. This calculation determines each solver's contribution to a hypothetical portfolio constructed from all considered solvers. The Shapley Value provides a means of comparing and ranking the solvers. The calculation requires a scoring mechanism to evaluate a given coalition of solvers. In this case, we assign a solver a score of $1 - \frac{CPU\,time}{time\,limit}$ per instance. We assume that the coalition of solvers is an oracle portfolio and always selects the best solver. Thus, it takes the maximum score for each instance from among all solvers present. The total score for a coalition is the sum of the coalition's score on each instance. A solver's contribution to a coalition is the change in the coalition's score when including the solver. The Shapley value of an individual solver is its average contribution to every possible permutation of every possible subset of the

[4]Upon examination, the unique large cluster of marc would have been correctly labelled 'Neither' and solved backtrack free by GAC.

[5]The percentage values are computed for each instance, then averaged across all the instances in a given benchmark.

solvers. Table 4 shows the Shapley values of the seven solvers. DecTree contributes the most to this hypothetical portfolio, followed by GAC. GAC scores highly because it is able to do well on instances on which the other solvers do poorly, whereas the other solvers overlap more in their strengths.

Table 4: Shapley values of the seven solvers

Solver	**Shapley Value**
DecTree	**134.8**
GAC	126.8
Random	120.9
PerTuple$^+$	109.3
PerTuple	96.9
AllSol$^+$	78.8
AllSol	75.4

7 Conclusions

We advocate a portfolio method for enforcing constraint minimality on the clusters of a tree decomposition, making minimality even more beneficial in practice by selectively applying it during problem solving. We provide three improvements in the application of constraint minimality: a classifier for choosing when to run AllSol, PerTuple, or neither, the use of GAC prior to every cluster being processed, and a timeout mechanism to prevent getting stuck on a single cluster. Our approach yields more problem completions and faster runtimes than lookahead with a simple GAC, PerTuple, or AllSol.

As a continuation of our approach, we plan to use a classifier that estimates the runtime of a consistency algorithm and the amount of filtering it could achieve in order to dynamically set-up the timeout threshold. Such a classifier would allow us to allocate more time for running the algorithm when significant filtering can be expected and less time when there is little prospect for filtering. We also want to extend our approach to estimating the memory overhead for running a given consistency algorithm and for selecting the most appropriate bolstering schema at the separators [Karakashian *et al.*, 2013].

Acknowledgements: Experiments were conducted at the Holland Computing Center facility of the University of Nebraska. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1041000 and NSF Grant No. RI-111795 and RI-1619344. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation.

References

[Amadini *et al.*, 2014] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An Enhanced Features Extractor for a Portfolio of Constraint Solvers. In *Proc. of ACM SAC 2014*, pages 1357–1359. ACM, 2014.

[Amadini *et al.*, 2015] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A Multicore Tool for Constraint Solving. pages 232–238. AAAI Press, 2015.

[Arnborg, 1985] Stefan A. Arnborg. Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability–A Survey. *BIT*, 25:2–23, 1985.

[Balafrej *et al.*, 2013] Amine Balafrej, Christian Bessiere, Remi Coletta, and El Houssine Bouyakhf. Adaptive Parameterized Consistency. pages 143–158, 2013.

[Balafrej *et al.*, 2015] Amine Balafrej, Christian Bessiere, and Anastasia Paparrizou. Multi-Armed Bandits for Adaptive Constraint Propagation. In *Proc. of IJCAI 2015*, 2015.

[Bayer *et al.*, 2007] Kenneth M. Bayer, Martin Michalowski, Berthe Y. Choueiry, and Craig A. Knoblock. Reformulating CSPs for Scalability with Application to Geospatial Reasoning. In *Proc. of CP 2007*, volume 4741 of *LNCS*, pages 164–179. Springer, 2007.

[Bessière *et al.*, 2005] Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, and Yuanlin Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[Bessiere *et al.*, 2013] Christian Bessiere, Hélène Fargier, and Christophe Lecoutre. Global Inverse Consistency for Interactive Constraint Satisfaction. In *Proc. CP 2013*, volume 8124 of *LNCS*, pages 159–174, 2013.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proc. ECAI 2004*, pages 146–150, 2004.

[Dechter and Pearl, 1988] Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1988.

[Dechter and Pearl, 1989] Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.

[Dechter *et al.*, 1991] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Epstein *et al.*, 2005] Susan Epstein, Richard Wallace, Eugene Freuder, and Xingjian Li. Learning Propagation Policies. pages 1–15, 2005.

[Fréchette *et al.*, 2016] Alexandre Fréchette, Lars Kotthoff, Tomasz Michalak, Talal Rahwan, Holger H. Hoos, and Kevin Leyton-Brown. Using the Shapley Value to Analyze Algorithm Portfolios. In *Proc. AAAI 2016*, 2016.

[Freuder and Elfe, 1996] Eugene C. Freuder and Charles D. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proc. of AAAI 1996*, pages 202–208, 1996.

[Freuder, 1978] Eugene C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21 (11):958–966, 1978.

[Golumbic, 1980] Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press Inc., 1980.

[Gomes and Selman, 2001] Carla P. Gomes and Bart Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.

[Gottlob *et al.*, 1999] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A Comparison of Structural CSP Decomposition Methods. In *Proc. of IJCAI 1999*, pages 394–399, 1999.

[Gottlob, 2011] Georg Gottlob. On Minimal Constraint Networks. In *Proc. of CP 2011)*, volume 6876 of *LNCS*, pages 325–0339. Springer, 2011.

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[Haralick and Elliott, 1980] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hurley *et al.*, 2014] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry OSullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *Proc. of CPAIOR 2014*, pages 301–317. 2014.

[Jeavons *et al.*, 1994] Peter G. Jeavons, David A. Cohen, and Marc Gyssens. A Structural Decomposition for Hypergraphs. *Contemporary Mathematics*, 178:161–177, 1994.

[Karakashian *et al.*, 2010] Shant Karakashian, Robert Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *Proc. AAAI 2010*, pages 101–107, 2010.

[Karakashian *et al.*, 2012] Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Algorithms for the Minimal Network of a CSP and a Classifier for Choosing Between Them. Technical Report TR-UNL-CSE-2012-0007, CSE, University of Nebraska-Lincoln, Lincoln, NE, 2012.

[Karakashian *et al.*, 2013] Shant Karakashian, Robert Woodward, and Berthe Y. Choueiry. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In *Proc. of AAAI 2013*, pages 466–473, 2013.

[Karakashian, 2013] Shant Karakashian. *Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition*. PhD thesis, CSE, UNL, Lincoln, NE, May 2013.

[Kjærulff, 1990] U. Kjærulff. Triagulation of Graphs - Algorithms Giving Small Total State Space. Research Report R-90-09, Aalborg University, Denmark, 1990.

[Mackworth, 1977] Alan K. Mackworth. On Reading Sketch Maps. In *Proc. of IJCAI 1977*, pages 598–606, 1977.

[Mehta and van Dongen, 2007] Deepak Mehta and Marc R.C. van Dongen. Probabilistic Consistency Boosts MAC and SAC. In *Proc. of IJCAI 2007*, pages 143–148, 2007.

[Montanari, 1974] Ugo Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Sciences*, 7:95–132, 1974.

[O'Mahony *et al.*, 2008] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. In *Proc. of the Irish Conference on AI and Cognitive Science*, pages 210–216, 2008.

[Paparrizou and Stergiou, 2012] Anastasia Paparrizou and Kostas Stergiou. Evaluating Simple Fully Automated Heuristics for Adaptive Constraint Propagation. In *Proc. of IEEE ICTAI 2012*, pages 880–885, 2012.

[Régin, 1994] Jean-Charles Régin. A filtering algorithm for constraints of difference in constraint satisfaction problems, 1994.

[Rice, 1976] John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[Rossi *et al.*, 1990] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the Equivalence of Constraint Satisfaction Problems. In *Proc. of ECAI 1990*, pages 550–556, 1990.

[Shapley, 1952] Lloyd S. Shapley. A Value for n-Person Games. Technical report, DTIC Document, 1952.

[Stergiou, 2009] Kostas Stergiou. Heuristics for Dynamically Adapting Propagation in Constraint Satisfaction Problems. *AI Communications*, 22(3):125–141, 2009.

[Woodward *et al.*, 2011] Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In *Proc. of AAAI 2011*, pages 112–119, 2011.

[Woodward *et al.*, 2014] Robert J. Woodward, Anthony Schneider, Berthe Y. Choueiry, and Christian Bessiere. Adaptive Parameterized Consistency for Non-Binary CSPs by Counting Supports, 2014.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.