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,‡], and Stephen D. Scott[2]

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

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.

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) [27], 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 [13, 23]. Following Karakashian et al. [23], 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.

* 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.

Previous work has proposed two algorithms for enforcing minimality: PerTuple [24] and AllSol [22]. 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 [27, 7]. 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 discusses enforcing minimality on the clusters of a tree decomposition. Section 4 discusses the construction of the portfolio. Section 5 describes experimental evaluations of the portfolio on benchmark problems. Finally, Section 6 presents our conclusions.

2 Background

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. 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 represent an assignment of values to the respective variables that is consistent with the constraint.

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. In the *primal graph*, the vertices represent the variables, and the edges connect every two variables that appear in the scope of some constraint. In the *dual graph*, the vertices represent the constraints of the CSP, and the edges connect vertices corresponding to constraints whose scopes overlap. Finally, the *incidence graph* of a CSP is a bipartite graph where one set contains all the variables and the other all the constraints. 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 [33].

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 [11,

21, 17]. We use an adaption for non-binary CSPs of the tree-clustering technique [11]. First, we triangulate the primal graph of the CSP using the min-fill heuristic [26]. Then, we identify the maximal cliques in the resulting chordal graph using the MaxCliques algorithm [14], 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 [9]. In order to enhance constraint propagation between adjacent clusters, we use the projection schema described by Karakashian et al. [23].

A CSP is *arc consistent* (AC) if every value has a supporting value in all neighboring variables. A similar property for non-binary constraints is *generalized arc consistency* (GAC) [27]. Our experiments use the GAC-2001 algorithm [7]. *Minimality* requires that any tuple that satisfies a constraint appears in at least one solution to the CSP [29].

The two algorithms, AllSol and PerTuple, proposed by Karakashian [22], both compute the minimal relations. PerTuple performs a backtrack search on every tuple in every relation, trying to consistently extend it a tuple in each other relation in the CSP. If the search fails, the tuple is removed. Otherwise, the 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. 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.

Related work on computing minimality includes: [5, 6, 16]; portfolio approaches: [32, 15, 37, 30, 20, 2]; adaptive consistency: [10, 12, 28, 34, 31, 3, 36, 4, 35].

3 Enforcing Minimality in a Tree Decomposition

Karakashian never compared the performance of PerTuple and AllSol during search, but only on individual clusters [22] collected from tree decompositions of CSP instances. When exploiting a tree decomposition for lookahead, Karakashian et al. exclusively used PerTuple [23]. 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 after processing any given cluster.
2. Every time a cluster is considered for consistency, we can consistently 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 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 process. Table 1 lists the parameter settings that yield six algorithms.

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
```

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)

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 look-ahead procedure during search. The foreach loop (line 8) processes clusters from the specified *clusterOrder*. In our setting, this ordering corresponds to the MaxCliques ordering of the clusters, but we will investigate other priority orderings in the future. The outer while-loop (line 6) iterates until no further filtering can be achieved. At each pass, the direction of the cluster ordering is reversed to facilitate propagation (line 20). The classifier allows the selection of the most appropriate algorithm on a cluster by cluster basis (line 9). 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 recover from classification errors.

4 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.

Inspired by features that appeared in the literature [25, 1], 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, minimum, maximum, and entropy (*). For some features, we report total sum (†) or $\log_{10}$ of mean (‡).

- *CSP parameters*: number of variables; number of relations; number of tuples per relation†*; domain size†*; arity of relations*; tightness of relations*; relational linkage‡*.
- *Dual-graph parameters*: density; vertex degree*; vertex eccentricity*; vertex-clustering coefficient*.
- *Incidence-graph parameters*: density; vertex degree*; vertex eccentricity*.
- *Primal-graph parameters*: density; vertex degree*; vertex eccentricity*; vertex-clustering coefficient*.

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 from each of the 175 benchmarks or taking all the clusters of a benchmark when it has less than 70 clusters. We ran both AllSol and PerTuple on every cluster, while recording (for each cluster) a set of features as well as the runtimes of the algorithms. Figure 1 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.

After collecting data from 9362 individual clusters, we trained a decision tree using the J48 algorithm of the Weka machine learning software suite [18]. 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. 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 CPU time on i of AllSol and PerTuple, respectively.

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

We designed this weighting scheme to emphasize instances where the execution times differ greatly, both in the ratio and in the difference of their values. We assigned a weight of 20 to the 'Neither' instances after empirical testing.

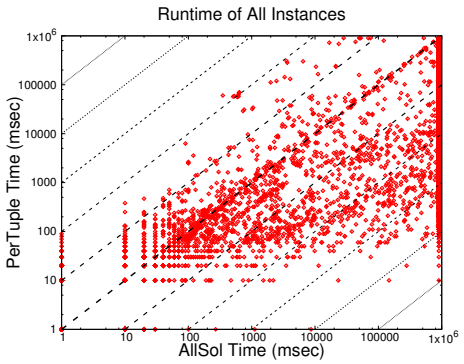

Fig. 1. Distribution of algorithm runtimes on single clusters

As a preliminary evaluation, we performed ten-fold cross validation using the collected instances. Our classifier achieved an unweighted accuracy of 80.1% and a weighted accuracy of 90.8%, which indicates the classifier is correctly handling the more heavily weighted instances.

5 Experimental Evaluation

In our evaluation, we use the six algorithms obtained by setting the parameters of FilterClusters alongside GAC for real-full lookahead [19] in a backtrack search on a set of 1055 instances taken from 42 benchmarks from the XCSP library.[1] Our search procedure terminates after finding the first solution and uses the dom/deg *dynamic* ordering heuristic. Although dom/wdeg [8] 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 1, this value strikes a good balance between completing clusters and not spending excessive time on any one cluster.

Table 2 summarizes the results (top) and provides detailed results per benchmark (bottom). 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.

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

Table 2. Experimental results of the seven solvers run on 1055 benchmark instances. We consider only instances where at least one solver completes. 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.

	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

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× 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 2 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.

6 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 [23].

References

1. 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.
2. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A Multicore Tool for Constraint Solving. In *Proc. of IJCAI 2015*, pages 232–238. AAAI Press, 2015.
3. Amine Balafrej, Christian Bessiere, Remi Coletta, and El Houssine Bouyakhf. Adaptive Parameterized Consistency. In *Proc. of CP 2013*, pages 143–158, 2013.
4. Amine Balafrej, Christian Bessiere, and Anastasia Paparrizou. Multi-Armed Bandits for Adaptive Constraint Propagation. In *Proc. of IJCAI 2015*, 2015.
5. 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.
6. Christian Bessiere, Hélène Fargier, and Christophe Lecoutre. Global Inverse Consistency for Interactive Constraint Satisfaction. In *Proc. of CP 2013*, volume 8124 of *LNCS*, pages 159–174, 2013.
7. 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.
8. Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proc. of ECAI 2004*, pages 146–150, 2004.
9. Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
10. Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1988.
11. Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.
12. Susan Epstein, Richard Wallace, Eugene Freuder, and Xingjian Li. Learning Propagation Policies. In *Proc. of the Second International Workshop on Constraint Propagation and Implementation*, pages 1–15, 2005.
13. Eugene C. Freuder and Charles D. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proc. of AAAI 1996*, pages 202–208, 1996.
14. Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press Inc., 1980.
15. Carla P. Gomes and Bart Selman. Algorithm Portfolios. *Artificial Intelligence*, 126(1):43–62, 2001.
16. Georg Gottlob. On Minimal Constraint Networks. In *Proc. of CP 2011)*, volume 6876 of *LNCS*, pages 325–0339. Springer, 2011.
17. Georg Gottlob, Nicola Leone, and Francesco Scarcello. A Comparison of Structural CSP Decomposition Methods. In *Proc. of IJCAI 1999*, pages 394–399, 1999.
18. 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.
19. Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.
20. Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *Proc. of CPAIOR 2014*, pages 301–317, 2014.

21. Peter G. Jeavons, David A. Cohen, and Marc Gyssens. A Structural Decomposition for Hypergraphs. *Contemporary Mathematics*, 178:161–177, 1994.
22. Shant Karakashian. *Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition*. PhD thesis, CSE, UNL, Lincoln, NE, May 2013.
23. 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.
24. Shant Karakashian, Robert Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *Proc. of AAAI 2010*, pages 101–107, 2010.
25. 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.
26. U. Kjærulff. Triagulation of Graphs - Algorithms Giving Small Total State Space. Research Report R-90-09, Aalborg University, Denmark, 1990.
27. Alan K. Mackworth. On Reading Sketch Maps. In *Proc. of IJCAI 1977*, pages 598–606, 1977.
28. Deepak Mehta and Marc R.C. van Dongen. Probabilistic Consistency Boosts MAC and SAC. In *Proc. of IJCAI 2007*, pages 143–148, 2007.
29. Ugo Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Sciences*, 7:95–132, 1974.
30. 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.
31. Anastasia Paparrizou and Kostas Stergiou. Evaluating Simple Fully Automated Heuristics for Adaptive Constraint Propagation. In *Proc. of IEEE ICTAI 2012*, pages 880–885, 2012.
32. John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.
33. Francesca Rossi, Charles Petrie, and Vasant Dhar. On the Equivalence of Constraint Satisfaction Problems. In *Proc. of ECAI 1990*, pages 550–556, 1990.
34. Kostas Stergiou. Heuristics for Dynamically Adapting Propagation in Constraint Satisfaction Problems. *AI Communications*, 22(3):125–141, 2009.
35. 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.
36. Robert J. Woodward, Anthony Schneider, Berthe Y. Choueiry, and Christian Bessiere. Adaptive Parameterized Consistency for Non-Binary CSPs by Counting Supports. In *Proc. of CP 2014*, volume 8656 of *Lecture Notes in Computer Science*, pages 755–764, 2014.
37. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32(1):565–606, 2008.