Algorithms for the Minimal Network of a CSP and a Classifier for Choosing Between Them

Shant Karakashian[1] Robert J. Woodward[1]
Berthe Y. Choueiry[1] Stephen D. Scott
[1] Constraint Systems Laboratory
University of Nebraska-Lincoln, USA
{shantk|rwoodwar|choueiry|sscott}@cse.unl.edu

TR-UNL-CSE-2012-0007

September 5, 2012

Abstract

The minimal constraint network of a constraint satisfaction problem (CSP) is a compiled version of the problem where every tuple in a constraint's relation appears in at least one solution to the CSP. Recently, Gottlob argued that, when a CSP has this property, a number of NP-hard queries can be answered in polynomial time, but he also showed that deciding whether or not a given network is minimal is NP-complete [Gottlob, 2011]. We propose two search-based algorithms for computing the minimal network of a CSP. We investigate the performance of the two algorithms and propose a classifier to select the appropriate algorithm that minimizes the CPU time, using a number of parameters. Our approach constitutes a significant contribution towards the automation of the selection of the appropriate algorithms for computing the minimal network of a CSP. In most cases that we studied, we achieved classifier accuracy of above 90%, which allowed us realize significant time savings.

Contents

1 Introduction

The minimal network of a constraint satisfaction problem is the network where each tuple in the relation of a constraint appears in at least one solution to the problem [Montanari, 1974]. Gottlob argued that, when a CSP has this property, a number of NP-hard queries can be answered in polynomial time, but also showed that deciding whether or not a constraint network is minimal is NP-complete [Gottlob, 2011].

In [Karakashian *et al.*, 2010], we introduced an algorithm for enforcing a relational consistency property called R($*$,m)C, also known as m-wise consistency in relational databases. In essence, the algorithm computes the minimal graph of the problem induced by every set of m relations of the CSP. We showed that this algorithm is particularly effective, when used for full lookahead in a backtrack search, for solving difficult CSPs.

In this paper, we examine the algorithm described in [Karakashian *et al.*, 2010] and use it to compute the minimal network of a CSP. We refer to that algorithm as PERTUPLE. We also explore an alternative search algorithm for enforcing the same property and call it ALLSOL. Both algorithms use backtrack search to verify whether or not a given tuple appears in a solution to the problem. However, the former repeats a 'satisfiability' search (i.e., stopping after finding the first solution), for every tuple in every relation, in the worst case, whereas the latter carries out a single 'solution counting' search (i.e., exploring the entire space).

In order to select which algorithm is more appropriate to minimize the CPU time in a given context, we propose a set of parameters that attempt to characterize the 'interactions' among the relations in the problem. Our set includes the parameter κ introduced by Gent et al. in [Gent *et al.*, 1996]. We use machine learning algorithms to generate classifiers that determine whether to use PERTUPLE or ALLSOL. In most cases that we studied, we achieved classifier accuracy of above 90%, which allowed us realize average time savings of more than 100 seconds.

The contributions of this paper are as follows: the presentation of two algorithms for computing the minimal network of a CSP, identification of CSP parameters that can be computed in polynomial time, and their use in building a decision tree that predicts the appropriate algorithm.

This paper is structured as follows. Section 2 gives background information, Section 3 presents the two algorithms, and Section 4 describes and evaluates the classifier used for predicting the appropriate algorithm. Section 5 discusses related work. Section 6 concludes with future work.

2 Background

A constraint satisfaction problem (CSP) is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints. Each variable $A_i \in \mathcal{X}$ has a finite domain $D_i \in \mathcal{D}$, and is constrained by a subset of the constraints in $\mathcal{C}$. Each constraint $C_i \in \mathcal{C}$ is defined by a relation R_i specified over the *scope* of the constraint, *scope*(C_i), which are the variables to which the constraint applies, as a subset of the Cartesian product of the domains of those variables. The *arity* of a constraint is the cardinality of its scope. A tuple $t_i \in R_i$ is thus a combination of values for the variables in the scope of the constraint that is either allowed (i.e., support) or forbidden (i.e., conflict). In this paper, we consider only allowed tuples. A solution to the CSP is an assignment, to each variable, of a value taken from its domain such that all the constraints are satisfied. Solving a CSP consists of finding one or all solutions.

A CSP can be represented by several types of graphs: in the *hypergraph* of a CSP, as shown in Figure 1, the vertices represent the variables of the CSP and the hyperedges represent the scopes of the constraints. The *primal graph* of a CSP is a graph whose vertices represent the variables and the edges connect every two variables that appear in the scope of some constraint as shown in Figure 2. The *dual graph* of a CSP is a graph whose vertices represent the constraints of the CSP, and whose edges connect two vertices corresponding to constraints whose scopes overlap as in Figure 3. The dual CSP, $\mathcal{P}_D$, is thus a binary CSP where: (1) variables are the constraints of the original CSP; (2) the variables' domains are the tuples of the corresponding relations; and (3) the constraints enforce *equalities* over the shared variables.

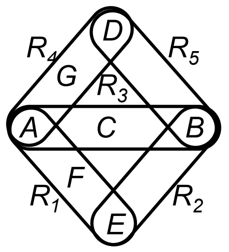

Figure 1: Hypergraph.

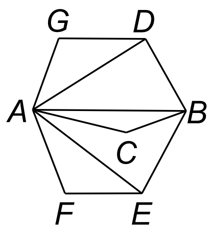

Figure 2: Primal graph.

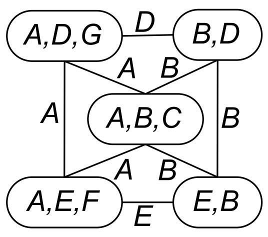

Figure 3: Dual graph.

Montanari [Montanari, 1974] introduced the *minimal network* of a CSP. Stated informally, the network is the one where the relations are as tight as can be, that is, each tuple in a relation can be extended to a solution to the CSP. The formal definition from [Dechter, 2003] is given next.

Definition 2.1. *Given a CSP $\mathcal{P}_0$, let $\{\mathcal{P}_1, \ldots, \mathcal{P}_l\}$ be the set of all networks equivalent to $\mathcal{P}_0$. Then the minimal network M of $\mathcal{P}_0$ is defined by $M(\mathcal{P}_0) = \cap_{i=1}^{l} \mathcal{P}_i$.*

Finally, π, σ and $\bowtie$ denote the relational operators project, select and natural join respectively.

3 Computing the Minimal Network of a CSP

In this section, we present two search-based algorithms for computing the minimal network of a CSP. The first algorithm is called PERTUPLE and the second ALLSOL. As the names indicate, the former conducts a search for a consistent solution for each tuple, and the latter conducts a single backtrack search generating a sufficient number of solutions to cover all the tuples of the minimal network. Below, we informally describe both search algorithms.

3.1 PERTUPLE: Solving Several Satisfiability Problems

As stated in Section 1, PERTUPLE is the algorithm presented in [Karakashian *et al.*, 2010]. The pseudocode is given in Algorithm 1. Given a CSP with e relations, t tuples per relation, where the total number of tuples is $T = et$, PERTUPLE conducts a sequence of backtrack searches on dual CSP, $\mathcal{P}_D$. It proceeds as follows. First, it initializes the tuple marks to 'false'. Then it considers the T tuples in some sequence. For each tuple τ_i of a relation R_i, it conducts in Line 7 a backtrack search in the space defined by the relations in the CSP (i.e., the variables of $\mathcal{P}_D$) excluding R_i. It attempts to find in that space a solution that is consistent with τ_i. As soon as a first solution is found, search is interrupted successfully (i.e., satisfiability search). τ_i and all the tuples appearing in that solution are marked as belonging to the minimal network in Line 9. If no solution is found, τ_i is removed from R_i in Line 8 because it is inconsistent and cannot be in the minimal network. All tuples in the e relations of the CSP are considered unless they have been already marked. PERTUPLE executes at most T backtrack searches, each of worst-case time complexity $\mathcal{O}(t^{e-1})$. The performance of the search is enhanced by forward checking. Checking the consistency of two tuples during search is facilitated by the index-tree data structure, also introduced in [Karakashian *et al.*, 2010].

Algorithm 1: PERTUPLE($\mathcal{P}_D$)

```
  Input: $\mathcal{P}_D$
  Output: Minimal Network of $\mathcal{P}_D$
1 foreach $R_i \in \mathcal{P}_D$ do
2     foreach $\tau_i \in R_i$ do SETMARK($\tau_i, false$)
3 foreach $R_i \in \mathcal{P}_D$ do
4     foreach $\tau_i \in R_i$, do
5         if MARKED($\tau_i$) $= false$ then
6             ASSIGN($R_i, \tau_i$)
              /* Backtrack search for a solution        */
7             $sol$ ←BTSEARCHONESOL($\mathcal{P}_D$)
8             if $sol = false$ then DELETE($\tau_i$)
9             else foreach $\tau_j \in sol$ do SETMARK($\tau_j, true$);
```

3.2 ALLSOL: Solving a single Counting Problem

We introduce the alternative algorithm ALLSOL which conducts a single backtrack search on the e variables of $\mathcal{P}_D$, and its worst-case time complexity is thus $\mathcal{O}(t^e)$. It is outlined in Algorithm 2. First, it initializes the tuple marks to 'false'. Then it proceeds with the single backtrack search by calling BTSEARCHNEXTSOL in Line 5. However, it does not stop after the first solution, it continues in the loop of Line 4 until all the solutions are found. Note that only the first call to BTSEARCHNEXTSOL starts a backtrack search, and the subsequent calls only advance the backtrack search to the next solution. Every time a solution is found, all the tuples in the solution are marked as belonging to the minimal network in Line 7. Like PERTUPLE, ALLSOL uses forward checking. Finally, it deletes all the tuples that were not marked in Line 10.

An important improvement allows us to interrupt search before traversing the entire space (which would be necessary in search for solution counting). After every step in the search and after executing forward checking, the domains of the future 'variables' (in fact, the relations of the original CSP) are considered. If all the 'surviving' tuples are marked as belonging to the minimal network, as well as all the tuples in the current path, then the search resumes from that path as if it was a dead-end. At the end of search, which may or may not cover all solutions, all unmarked tuples are removed from the relations.

Algorithm 2: AllSol($\mathcal{P}_D$)

```
    Input: $\mathcal{P}_D$
    Output: Minimal Network of $\mathcal{P}_D$
 1  foreach $R_i \in \mathcal{P}_D$ do
 2      foreach $\tau_i \in R_i$ do SetMark($\tau_i, false$)
 3  $sol \leftarrow false$
 4  while $sol = false$ do
 5      $sol \leftarrow$BTsearchNextSol($\mathcal{P}_D$)
 6      if $sol \neq false$ then
 7          foreach $\tau_i \in sol$ do SetMark($\tau_i, true$)
 8  foreach $R_i \in \mathcal{P}_D$ do
 9      foreach $\tau_i \in R_i$ do
10          if Marked($\tau_i$) $= false$ then Delete($\tau_i$)
```

3.3 Improving Forward Checking

Another practical improvement in this work attempts to reduce the effort necessary for executing forward checking. Given that the size of relations can be large, it becomes important to check the consistency of two tuples without scanning all the relations. We have already mentioned that we use the index-tree data structure for checking the consistency of two tuples from two relations whose scope overlap. Forward checking operates by removing from the 'future' relations those tuples that are not consistent with the current path. We call the tuples that are consistent 'valid' and those that are not 'invalid.'

For a given tuple t_i in a given relation R_i, the index-tree data structure returns all the tuples in an 'adjacent' relation that are consistent with t_i. The set of such tuples includes both valid and invalid tuples because the index-tree is not updated when forward checking invalidates tuples in future relations. Thus, the returned tuples must be scanned and only the ones considered to be valid should be considered. At some point, during the backtrack search, most of the tuples may become invalid. Hence, it may be more efficient to check the valid tuples for consistency with t_i than to check for validity of the consistent tuples returned by the index-tree. For this purpose, each relation keeps a counter of the number of 'valid' tuples and the index-tree data structure keeps a counter of the number of (valid and invalid) tuples consistent with t_i. We compare the two counters, and the smaller set is examined.

3.4 Correctness and Efficiency

The algorithms PERTUPLE and ALLSOL compute the minimal network by finding at most T solutions, T being the total number of tuples in all relations of the minimal CSP. We first show that the bound on the number of solutions found is a tight bound on the minimum number of solutions needed, then we prove this bound.

Theorem 3.1. *Given a CSP, the problem that answers the following question is NP-Complete: is there a set of at most k solutions such that every tuple appears in at least one solution?*

Proof sketch. We reduce Minimum Set Cover [Garey and Johnson, 1979] to this problem in polynomial time. Given a collection $\mathcal{C}$ of subsets of a finite set $\mathcal{S}$ and a positive integer k, a set cover of size k or less exists iff a set of at most $(2 \cdot |\mathcal{S}| + k)$ solutions exists. The reduction is accomplished by constructing a CSP with a variable for each element in $\mathcal{S}$, and domains and relations to have a solution corresponding to each subset in $\mathcal{C}$. The details of the construction are outside the scope of this paper. □

Therefore, we will likely have to find more than the minimum number of solutions necessary for 'covering' the tuples in the minimal network.

Theorem 3.2. *Given $\mathcal{P}_\mathcal{D}$ the dual encoding of a CSP,* PERTUPLE *finds at most T solutions, where T is the total number of tuples in the relations of the minimal network.*

Proof sketch. Clearly PERTUPLE starts a backtrack search for each tuple t_i, after assigning t_i to its corresponding relation, and deletes t_i if no solution is found. Therefore, no tuple that appears in some solution is deleted. Moreover, at most T solutions are found. However, our 'marking' mechanism saves us the effort of search for solutions for those tuples encountered in solutions during search. Therefore, every tuple in the minimal network appears in at least one solution. □

We next prove that ALLSOL also finds at most T solutions and computes the minimal network with the improvement discussed in Section 3.2.

Theorem 3.3. *Given $\mathcal{P}_\mathcal{D}$ the dual encoding of a CSP,* ALLSOL *finds at most T solutions, where T is the total number of tuples in the relations of the minimal network.*

Proof sketch. AllSol marks the tuples that appear in a solution, and deletes all the tuples that were not marked, thus guarantees that all the remaining tuples appear in at least one solution. Moreover, whenever the backtrack search finds a solution, at least one tuple in the solution must be unmarked. Otherwise, the backtrack search will resume as if a dead end was discovered and will not continue to the solution. This is due to the improvement discussed in Section 3.2. Therefore, at least one tuple is marked for each solution found by AllSol, and as a result, there can not be more solutions than marked tuples.

3.5 Rationale for Operating on the Dual Encoding

Computing the minimal network of a CSP requires checking that tuples of the relations appear in some solution to the CSP and removing those that do not. Thus, operating on the dual CSP seems to be a natural choice. The constraints in the dual CSP are equality constraints and enforce that the values of the variables common to the scope of two constraints be the same. Typically, the number and size of the dual constraints are large in practice. We do not explicitly generate the dual constraints and thus, their size or number do not cause any space overhead.

Moreover, given the nature and the number of the dual constraints, a partial look-ahead strategy such as forward checking (FC) on the dual graph proved to be sufficient in practice. We tested the full-lookahead strategy known as 'Maintaining Arc Consistency' (MAC) [Sabin and Freuder, 1994], and it performed significantly worse than FC. Domain sizes is another problem of the dual variables which could be very large. Our algorithms scale well with the domain sizes of the dual variable (i.e., the number of support tuples in the relations) by using the index-tree structures [Karakashian *et al.*, 2010]. We could easily handle relations with 150,000 tuples.

3.6 Qualitative Comparison of PerTuple and AllSol

Consider a network of e relations, and t tuples per relation. In order to compute the minimal network, PerTuple solves $O(et)$ times a *satisfiability* problem of size $O(t^{e-1})$. Thus, its time complexity is $O(et^e)$. In contrast, AllSol solves once a *solution counting* problem of size $O(t^e)$, and its time complexity is $O(t^e)$. Relating the worst-case time complexities of the two algorithms, their behaviors may be more clearly characterized thanks to the phase transition phenomenon observed on CSPs [Cheeseman *et al.*, 1991].

First, let us notice that ALLSOL and PERTUPLE differ in two main aspects: (1) the cost of each backtrack search, and (2) the number times a new search is started. ALLSOL starts a single search, but searches the entire space. PERTUPLE starts a search once for each tuple in the problem, but each search stops after finding the first solution. Now, back to the phase-transition. According to that macro-characterization of CSPs,

- When a problem instance is located in the area where the existence of a solution is highly likely, solutions abound and are easy to find. In those conditions, each call to PERTUPLE is likely to terminate successfully and quickly. Even with repetitive calls to search, PERTUPLE remains quick. On the other hand, although it is sweeping only once through the search space, ALLSOL is likely to easily get 'overwhelmed,' enumerating the large number of solutions. In that area, PERTUPLE is likely significantly more efficient than ALLSOL.

- When a problem instance is located in the area where the existence of a solution is highly unlikely, a search procedure with decent lookahead is likely to effectively prune the tree, quickly terminating the search. Even though PERTUPLE starts many more searches than ALLSOL does, both algorithms are likely to quickly traverse the same 'barren' space and their performance is comparable.

- The difference between the two algorithms arises around the area of the phase transition. An instance in that area is likely to have many 'almost' solutions [Cheeseman *et al.*, 1991]. ALLSOL traverses the space once, may struggle to find the few solutions, if any, as one expects to be the case at the phase transition. The real misfortune is for PERTUPLE, because it may have to repeat the same costly process for every tuple in each relation, which may render totally unusable in practice.

In summary, while PERTUPLE is likely to be quite cheap more often than ALLSOL, when it encounters instances around the phase transition, it is unlikely to terminate even when ALLSOL does. The experiments reported below confirm the above interpretation.

4 Building a Hybrid Solver

As stated above, we expect, grossly speaking, the two algorithms to be 'complementary' in terms of their effectiveness in practice despite the fact that, obviously, there are problems too hard for either algorithm, and others easy for both. Our goal is to build a hybrid solver that, adaptively, chooses the 'best' algorithm to use or, at least, avoids the algorithm that does not terminate. The hybrid solver consists of the two algorithms (ALLSOL and PERTUPLE), a set of parameters to compute for each problem instance given in input (Section 4.2), and a 'quick' but discriminating classifier (Section 4.3). The hybrid solver computes the values of the parameters, gives them to the classifier, which determines whether to use PERTUPLE or ALLSOL. Below, we describe the sample data, the problem parameters and features used, and the classifiers built. We then discuss the evaluation of the two resulting hybrid solvers on the benchmarks used to build the classifiers and on randomly generated problems that were not part of the training data.

4.1 Data Used for Building the Classifiers

We drew the sample data from 1,616 instances from 61 benchmarks of the CSP Solver Competition.[1] Because the ultimate goal of this research endeavor is to compute the minimal network of each cluster of a tree decomposition of a CSP [Dechter, 2003], we generated a tree decomposition of each problem instance, and considered each cluster in the tree decomposition as an independent problem instance. The characteristics of the instances extracted from the benchmarks and those used are shown in Table 1.

We computed the minimal network of all 65,894 instances extracted using PERTUPLE and ALLSOL, and recorded the time taken by each algorithm. Neither algorithm consistently outperformed the other, but PERTUPLE was faster for more instances than ALLSOL was (10,283 versus 3,791). We chose to ignore all instances on which the execution of the two algorithms differed by less than 256 milliseconds, which we estimate to be, in our context, an insignificant time difference. Typically, the ignored instances are 'easily' solved by both algorithms or not solved by both. In this section, when we say 'solved' we mean computed the minimal network within the time limit of 30 minutes. To avoid overshadowing the differences between the two algorithms caused by the benchmark distribution, we partitioned the 8,319 remaining instances into two sets. $\mathcal{P}$ is the set of instances

[1]http://www.cril.univ-artois.fr/CPAI08/

Table 1: Summary of data used.

Original Data	
Number of instances drawn from benchmarks	65,894
Number of instances solved by ALLSOL	32,702
Number of instances solved by PERTUPLE	37,379

Data Used in Study ($\lvert\delta_{\text{time}}\rvert \geq 256$ **msec**)				
Timeout per instance	30 minutes			
Number of instances solved by ALLSOL: $\mathcal{A}$	3,618			
Number of instances solved by PERTUPLE: $\mathcal{P}$	8,295			
Number of instances in $\mathcal{A} \setminus \mathcal{P}$	24			
Number of instances in $\mathcal{P} \setminus \mathcal{A}$	4,701			
Total number of instances used: $\mathcal{A} \cup \mathcal{P}$	8,319			
	Min	**Max**	**Avg**	**Median**
Number of variables	4	145	20.40	15
Domain size	2	1001	85.16	20
Number of relations	2	947	63.35	35
Arity of relations	2	16	3.90	3
Number of tuples per relation	1	150,000	7,187.00	992

on which PERTUPLE was faster than ALLSOL by more than 256 milliseconds; $\mathcal{A}$ is the set on which ALLSOL runs faster than PERTUPLE by more than 256 milliseconds.

The left-hand side of Table 2 reports the number of instances solved from each set ($\mathcal{A}$ and $\mathcal{P}$) by each algorithm (ALLSOL and PERTUPLE). The right-hand side of the table reports the corresponding average CPU times in seconds. To compute the average, we consider only the instances solved by *both* algorithms (i.e., 456 instances from $\mathcal{A}$ and 3,135 instance from $\mathcal{P}$). On the instances solved by both

Table 2: Number of instances solved and the corresponding average times.

#Instances in	**solved by**. . .			**Average CPU** (sec)	
	ALLSOL	PERTUPLE	Both	ALLSOL	PERTUPLE
$\mathcal{A}$	483	459	459	**31.51**	58.66
$\mathcal{P}$	3,135	7,836	3,135	190.9	**8.08**

algorithms, ALLSOL is 46% faster than PERTUPLE on the instances in $\mathcal{A}$, and while PERTUPLE is 96% faster than ALLSOL on the instances in $\mathcal{P}$. Incidentally, the average time for PERTUPLE is small (8.08 seconds) on the particular subset of instances in $\mathcal{P}$ that were solved by ALLSOL (3,135). The average time of PERTUPLE on all the instances of $\mathcal{P}$ (8,295) is in fact much larger (76.22 seconds) as reported in Table 4. Table 2 shows that ALLSOL and PERTUPLE clearly outperform each other in their respective 'niche' (here, the instance sets $\mathcal{A}$ and $\mathcal{P}$ respectively). In practice, we need to determine from the outset which algorithm to use, which motivates us to build a classifier with machine learning techniques.

4.2 Parameters and Features

The topology of the constraint network (e.g., degree of a variable) and the definitions of the constraints (e.g., tightness of a relation) heavily impact the performance of the algorithms for solving CSPs (PERTUPLE) and counting their solutions (ALLSOL). We suspect that the relative performance of ALLSOL and PERTUPLE is also affected by the density of solutions in the space. Thus, we considered the following CSP parameters:

1. κ is a known parameter to predict that an instance is at the phase transition [Gent *et al.*, 1996]. It is defined for CSPs as $\kappa = -\frac{\sum_{R \in \mathcal{C}} log_2(1-p_R)}{\sum_{x \in \mathcal{X}} log_2(domain(x))}$, where p_R is the tightness of the constraint.

2. `relLinkage` is an approximate measure of how a 'tuple at the overlap of two relations' is likely to appear in a solution. We propose to compute it as follows. For every two relations R_i, R_j, let $V_{ij} = scope(R_i) \cap scope(R_j)$. $\forall R_k, scope(R_k) \supseteq V_{ij}, x \in scope(R_k) \setminus V_{ij}$, `relLinkage` for every tuple $t \in \pi_{V_{ij}}(R_i \bowtie R_j)$ is computed as $\prod_{R_k} \frac{|\sigma_t(R_k)|}{\prod_x |domain(x)|}$,

3. `tupPerVvp` is the sum of all tuples in which a given variable-value pair vvp appears, $\sum_{R_i \in \mathcal{R}} |\sigma_{vvp}(R_i)|$.

4. `tupPerVvpNorm` is the value of `tupPerVvp` normalized to the size of each relation, $\sum_{R_i \in \mathcal{R}} \frac{|\sigma_{vvp}(R_i)|}{|R_i|}$.

5. `tupPerVvpNormProd` is similar to `tupPerVvpNorm` using the product instead of the sum, $\prod_{R_i \in \mathcal{R}} \frac{|\sigma_{vvp}(R_i)|}{|R_i|}$.

6. `relPerVar` is the number of relations per variable v, $|\{R_i \mid v \in scope(R_i)\}|$, which is its degree in the primal graph.

For a given CSP instance, each parameter yields a set of numbers, which we combine into a single value using different statistical aggregations to obtain the following 12 features for training our classifiers:

1. κ
2. $\log_2(avg(\texttt{relLinkage}))$
3. $\log_2(stDev(\texttt{relLinkage}))$
4. $stDev(\texttt{relLinkage})/avg(\texttt{relLinkage})$
5. $stDev(\texttt{tupPerVvp})/avg(\texttt{tupPerVvp})$
6. $avg(\texttt{tupPerVvpNorm})$
7. $stDev(\texttt{tupPerVvpNorm})$
8. $stDev(\texttt{tupPerVvpNormProd})$
9. $stDev(\texttt{tupPerVvpNormProd})/avg(\texttt{tupPerVvpNormProd})$
10. $avg(\texttt{relPerVar})$
11. $stDev(\texttt{relPerVar})$
12. $stDev(\texttt{relPerVar})/avg(\texttt{relPerVar})$

We originally considered 34 combinations of CSP parameters (e.g., product of domain sizes, relations sizes, the entropy of constraint definitions) and ways to aggregate the corresponding values (e.g., sums and products, their ratios and logarithms, averages, and standard deviations). After constructing different decision trees produced by the learning algorithms used (i.e., C4.5 and Random Forest, see Section 4.3), the above-listed 12 features appeared constantly at the top levels of the produced trees. It is commonly acknowledged by the machine learning community that the features appearing at the top levels of decision trees are likely the most significant ones. Thus, we settled with this set of 12 features.

4.3 Building the Classifiers

To build the classifier, we used ‘off-the-shelf’ learning algorithms, the sample instances described in Section 4.1, the values of the set of features listed in Section 4.2 on the sample data, and the CPU times for solving the sample instances with both algorithms (i.e., PERTUPLE ALLSOL).

- *Learning algorithms.* We experimented with ten different learning algorithms from the open-source data-mining tool Weka [Hall *et al.*, 2009]. The two algorithms that yielded the best results were J48 and RF, which are Java implementations of C4.5 [Quinlan, 1993] and Random Forests [Breiman, 2001], respectively. In our experiments, we used the default parameters for each algorithm (e.g., ten trees for RF). The advantage of C4.5 is that it outputs a single decision tree, which when limited to around 20 nodes, seemed to provide a good trade-off between classification precision and ‘transparency’ to a human user. We tuned the C4.5 algorithm to output heavily pruned trees by reducing the pruning confidence to one percent.

- *The feature sets.* We evaluated two feature sets: the set of 12 features listed in Section 4.2, and a subset of it consisting of the features #1, #4, #5 and #8. The four features of the latter consistently appeared at the top three levels of the decision trees that we constructed on ten different partitions of the training set. Thus, they are likely the most significant ones.

- *Classes.* We classified the data into two classes: the first class is for the instances on which PERTUPLE is faster than ALLSOL by more than 256 milliseconds, and the second class is for the instances on which ALLSOL is faster than PERTUPLE by more than 256 milliseconds.

- *Training data ($\mathcal{T}$).* At the training stage, we used data from the partitions $\mathcal{A}$ and $\mathcal{P}$. We generated the training data, denoted by $\mathcal{T}$, by including all the instances of $\mathcal{A}$ and sampling a maximum of 30 instances from $\mathcal{P}$ for every benchmark represented in it. To select the 30 instances from $\mathcal{P}$, we chose the 15 instances with the largest time difference between ALLSOL and PERTUPLE, randomly selecting the rest from the remaining instances in the benchmark. We sampled instances from $\mathcal{P}$ instead of including all of them in order to balance and number of instances in each class. We balanced the number of instances so that the classifier does not bias one class over the other in its attempt to reduce the overall error rate.

We evaluated various configurations of the learning algorithms according to the transparency of classification process and the error rate. We consider the six configurations listed in Table 3.

Table 3: Main learning algorithms and configurations tested.

Classifier	Learning Algorithm	#Trees	Avg. #Nodes	Setting	#Features	Avg. Error Rate
DT1	**C4.5**	1	27.20	**Heavy pruning**	**12**	0.07
DT2	C4.5	1	65.00	Default pruning	12	0.06
RF1	**Random Forests**	10	174.42	**Default**	**12**	0.05
DT3	C4.5	1	23.40	Heavy pruning	4	0.08
DT4	C4.5	1	45.80	Default pruning	4	0.05
RF2	Random Forests	10	166.78	Default	4	0.04

We partitioned the training set $\mathcal{T}$ described above into ten partitions, and cross-validated each configuration by testing each partition on a classifier trained on the other nine partitions. Only the decision trees produced by C4.5 with heavy pruning were deemed to be transparent enough for readability. The number of nodes and the error rates reported in Table 3 are the averages across all ten folds of the cross-validation.

As for the classification error-rate, we performed a paired t-test and found no statistically significant difference between the classifiers produced by C4.5, under default pruning, using the set of 12 features and the set of four features (DT2 versus DT4). Also, we observed no statistically significant difference between the classifiers produced by C4.5 using the 12 feature set with pruning and the default pruning (DT1 and DT2).

However, we discovered that applying heavy pruning on C4.5 with the four feature set increases the classification error, as well as changing from the 12 feature set to four feature set does, on the heavily pruned trees, with more than 98% confidence. Moreover, we did not find any statistically significant difference between Random Forests and C4.5 for both four and 12 feature sets. Therefore, we chose to use the set with 12 features for the rest of the analysis and for generating the production classifier since it is human readable and performs as well or better than the others.

4.4 The Hybrid Solvers

We propose two hybrid solvers: SOLVER$_{C4.5}$ and SOLVER$_{RF}$ based on each of the two classifiers DT1 and RF1 of Table 3.

As described above, at the training stage, in order to avoid biasing the classifier while exploiting all the data available, we partitioned $\mathcal{T}$ into ten partitions, and did a cross-validation by testing each partition using the classifier trained on the other nine partitions. Subsequently, in an experiment separate from the cross-validation, we trained a 'production classifier' on all the instances in $\mathcal{T}$, and used it to evaluate the instances in $\mathcal{P}$ that were not included in $\mathcal{T}$. Therefore, all the instances in $\mathcal{A}$ and $\mathcal{P}$ are validated with unbiased classifiers. The decision tree of the production classifier output by C4.5 is given in Figure 4.

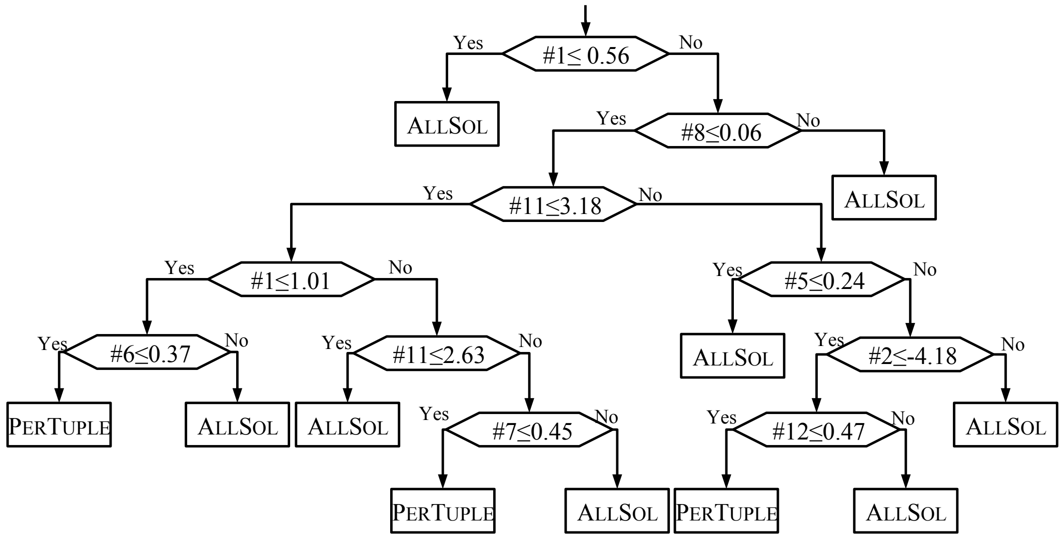

Figure 4: Decision tree of SOLVER$_{C4.5}$.

Table 4 lists, to the left, the number of instances solved by each algorithm (ALLSOL, PERTUPLE, SOLVER$_{C4.5}$ and SOLVER$_{RF}$), as well as the number of instances solved by all four algorithms. Because ALLSOL solves so few instances from rows 2 and 3, the numbers of instances solved by all four solvers are significantly reduced (from around 8,000 down to around 3,000, that is less than half the instances). When we next compute the average CPU times of each solver on the instances computed by all four solvers (at the right of Table 4), we ignore the impact of ALLSOL altogether in rows 2 and 3 in order to maintain a decent number of instances on which to compute the averages (i.e., 7,771 and 8,230).

Table 4: Comparing the performance of all four algorithms.

		#Instances solved by					Average CPU (sec)				
	Partition	AllSol	PerTuple	Solver$_{C4.5}$	Solver$_{RF}$	All solvers	#Instances	AllSol	PerTuple	Solver$_{C4.5}$	Solver$_{RF}$
1	$\mathcal{A}$	483	459	477	479	459	459	(ideal) 31.51	58.66	39.87	36.11
2	$\mathcal{P}$	3,135	7,836	7,784	7,785	3,135	7,771	-	(ideal) 77.26	79.42	79.00
3	$\mathcal{A} \cup \mathcal{P}$	3,618	8,295	8,261	8,264	3,594	8,230	-	76.22	77.21	76.60

On the instances solved by all algorithms, Solver$_{RF}$ is on average 38% faster than PerTuple alone on the instances in partition $\mathcal{A}$, and only 15% slower than the ideal algorithm. On the instances in partition $\mathcal{B}$, Solver$_{C4.5}$ and Solver$_{RF}$ solved more than twice the number of instances solved by AllSol, while being slower than the ideal algorithm by less than 3%. The ideal algorithm is chosen by the (non-existent) perfect solver that always knows which algorithm is faster. In conclusion, we note that the performance of either of our hybrid algorithms are comparable to that of the ideal situation.

The benchmark data used to build then validate our classifiers are rather structured data. One may rightfully worry that the features we selected, which attempt to capture the characteristics of the structure of a CSP, and our classifiers trained on structured data, may lose their 'edge' when used on 'amorphous' instances such as randomly generated CSPs. For this reason, we tested our two hybrid solvers Solver$_{4.5}$ and Solver$_{RF}$ on three sets of random CSPs (model B) generated in a window around the phase transition. The hybrid solvers Solver$_{4.5}$ and Solver$_{RF}$ use the production classifiers trained on the benchmark data in set $\mathcal{T}$, i.e. not trained on any instance from the three sets of random CSPs. This is also to test how well the classifiers and solvers generalize to new CSP. The problem sets' characteristics and the average times on the instances solved by both AllSol and PerTuple are shown in Table 5. We notice that both hybrid solvers were able to achieve times faster than both AllSol and PerTuple when taken individually, and consequently outperform the algorithms that they are choosing from.

In Table 6, we compare the performance of our 'production' solvers Solver$_{RF}$ and Solver$_{C4.5}$ on the benchmark data as well as the randomly generated instances, summarizing the following results:

- *Fatal* indicates the number of 'fatal' decisions corresponding to choosing

Table 5: Randomly generated CSPs.

	Set I	**Set II**	**Set III**
Number of variables	10	30	75
Domain size	10	6	5
Number of relations	100	75	120
Number of tuples per relation	[100,900]	[22,194]	[12,112]
Constraint arity	3		
Total number of instances	1000		
Number of instances solved by ALLSOL	997	733	383
Number of instances solved by PERTUPLE	1,000	990	365
Number of instances solved by both	997	733	335
On the instances solved by both PERTUPLE and ALLSOL			
Average time of ALLSOL in sec.	134.87	88.37	172.24
Average time of PERTUPLE in sec.	119.79	131.68	302.90
Average time of SOLVER$_{RF}$ in sec.	**58.06**	88.37	172.24
Average time of SOLVER$_{C4.5}$ in sec.	58.25	**85.53**	172.24

the wrong solver (ALLSOL or PERTUPLE), that is, choosing a solver that does not complete within the time threshold over another that does.

- *Saved* indicates the number of correct decisions corresponding to choosing a solver that does complete within the time threshold over another that does not. The number instances 'saved' justify the efforts of this research. While the large numbers of 'saved' instances in the benchmark data can be justified by the structure of the CSPs and the fact that the classifiers were trained on similar data, the large numbers of 'saved' data on Sets II and III justifies our endeavor by demonstrating how well our system generalizes to new types of CSPs.

- *Average savings* indicates how much time on average is saved per instance by the hybrid solver on the instances solved by both ALLSOL and PERTUPLE. Clearly both hybrid solvers yielded positive savings in all cases.

- *Classification error* indicates the rate of bad choices made by each hybrid solver. A bad choice is when the chosen solver does not solve and the other does, or the chosen solver is slower than the other. The highest error rate is in Set II, which has resulted in high number of fatal instances. In this set we observe the biggest difference between $\text{SOLVER}_{C4.5}$ and SOLVER_{RF}, and the former seems to be better than the latter.

Table 6: Comparing the two new hybrid solvers.

	#Instances		Average savings	Classification
	Fatal	**Saved**	(sec)	**error**
Benchmarks (8,319 instances)				
SOLVER_{RF}	**55**	**4,670**	**154.26**	0.04
$\text{SOLVER}_{C4.5}$	58	4,667	151.46	0.04
Set I				
SOLVER_{RF}	0	3	**138.53**	0.00
$\text{SOLVER}_{C4.5}$	0	3	138.15	0.04
Set II				
SOLVER_{RF}	188	69	43.30	0.40
$\text{SOLVER}_{C4.5}$	**65**	**192**	**48.97**	**0.26**
Set III				
SOLVER_{RF}	3	75	130.66	0.08
$\text{SOLVER}_{C4.5}$	3	75	130.66	0.08

5 Related Work

The "algorithm selection problem" was discussed at length by Rice [Rice, 1976] and has recently witnessed a surge of successful implementations under the label of "algorithm portfolio." An excellent historical review of the topic can be found by Xu et al. [Xu *et al.*, 2008]. Those authors introduced SATzilla, a wildly successful portfolio algorithm for solving SAT problems. SATzilla uses 48 features, computed from 16 parameters of SAT problems, to choose between seven SAT solvers. We use 12 attributes, computed from five CSP parameters, to choose between two algorithms. Our choice of attributes is sufficient for our task, which is simpler than SATzilla's. Importantly, the time for extracting and computing the features in our case is negligible compared to the time taken by computing the minimal network using either algorithm.

6 Conclusions and Future Work

In this paper we proposed two search-based algorithms for computing the minimal network of a CSP. We identified CSP parameters that can be computed in polynomial time to characterize the difficulty of a CSP instance and the degree of interaction among the various components of the instance. We used those parameters to build two classifiers that predict the appropriate algorithm given a problem instance. We evaluated our solvers and demonstrated their benefits on benchmark and on randomly generated problems, demonstrating their good performance on problems unrelated to the ones on which the classifiers were built. In most cases that we studied, we achieved classifier accuracy of above 90%, which allowed us realize average time savings of more than 100 seconds.

For future work, we plan to replace the algorithm for enforcing the relational consistency property R($*$,m)C introduced in [Karakashian *et al.*, 2010] with one of the hybrid solvers. We also plan to predict if both algorithms are not suitable to solve an instance within the time limit. The choice of using none is useful when we are computing the minimal network of a CSP decomposed into clusters, where we solve each cluster separately and then propagate the effects to one another. If we can predict that a cluster is too expensive to solve, we can delay it, solve the other clusters, and then propagate the effects of the neighbors to the difficult cluster. The propagation may likely simplify its complexity to the extent in which one of the algorithms may be suitable to solve it.

Acknowledgments

Experiments were conducted on the equipment of the Holland Computing Center at UNL. Robert Woodward was partially supported by a B.M. Goldwater Scholarship and by a National Science Foundation (NSF) Graduate Research Fellowship grant number 1041000. This research is supported by NSF Grant No. RI-111795.

References

[Breiman, 2001] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[Cheeseman *et al.*, 1991] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proc. of the 12th IJCAI*, pages 331–337, 1991.

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

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[Gent *et al.*, 1996] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The Constrainedness of Search. In *Proc. of the 13th AAAI Conference on Artificial Intelligence (AAAI 96)*, pages 246–252, 1996.

[Gottlob, 2011] Georg Gottlob. On Minimal Constraint Networks. In *Proceedings of 17th International Conference on Principle and Practice of Constraint Programming (CP 11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 325–339, 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. *SIGKDD Explorations Newsletter*, 11:10–18, November 2009.

[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. of the 24th AAAI Conference on Artificial Intelligence*, pages 101–107, 2010.

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

[Quinlan, 1993] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

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

[Sabin and Freuder, 1994] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of the 11 th ECAI*, pages 125–129, Amsterdam, The Netherlands, 1994.

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