Search on Constraint Satisfaction Problems with Sparse Secondary Structure

Susan L Epstein[1,2] and Xingjian Li[2]

[1] Hunter College and [2]The Graduate Center of The City University of New York
Department of Computer Science
New York, NY 10065 USA
susan.epstein@hunter.cuny.edu, xli1@gc.cuny.edu

Abstract

This paper considers a variety of ways to detect relatively isolated, highly restricted subproblems and then exploit them to guide search for a solution. It introduces a local search method that, prior to search, estimates where such subproblems lie within constraint satisfaction problems. These subproblems are assembled into a secondary structure used with dynamic variable-ordering heuristics to guide search, while learning protects against the occasional inadequacies of local search. On some classes of difficult structured benchmark problems, this approach solves constraint satisfaction problems an order of magnitude faster.

1 Introduction

During search for a solution to a constraint satisfaction problem (*CSP*), the *fail first* principle dictates that one should consider first those variables for which it is difficult to find values that lead to a solution (Smith and Grant, 1998). That has long been the justification for many traditional variable-ordering heuristics that choose one variable at a time during search. The thesis of this work is that fail first extends as well to certain subproblems, sets of variables distinguished by extensive and highly restrictive mutual constraints. The search envisioned here is therefore a hybrid — local, resource-bounded search detects such subproblems, and then a complete search for a solution exploits the *secondary structure* that describes those subproblems and the relationships among them. The principle results of this paper include how to detect a sparse secondary structure quickly prior to search, and how to exploit it to achieve as much as an order of magnitude speedup on certain benchmark problems.

Problems like that in Figure 1 provide insight into how search-ordering heuristics can be misled if they overlook secondary structure. This problem presents a considerable challenge to traditional CSP search-ordering heuristics. Section 2 recounts how one failed to solve it in 30 minutes, and two learning heuristics solved it after 127 and 88 seconds. That is because, as explained there, they "see" only Figure 1(a), the problem's primary structure. Figure 1(b) is a (manual) redrawing of the way the problem generator built the problem, as a set of five subproblems connected to one another only through a much larger subproblem. There is more to this problem than connectivity, however. Figure 1(c) darkens the more restrictive constraints, all of which are found in the smaller subproblems. Given Figure 1(c) in advance, it would still not be obvious how that foreknowledge should be used during search. Furthermore, because it is known only to the problem generator and not to the solver, that knowledge must ultimately be elicited from the problem by a heuristic, not assumed. Our approach used local search to identify the subproblems in Figure 1(d), built the secondary structure in Figure 1(e), and then solved the problem in 3.56 seconds.

After background and related work in Section 2, Section 3 investigates ways to apply foreknowledge like Figure 1(c) to search. Section 4 describes *Foretell*, a local search mechanism to detect heavily constrained, highly interactive

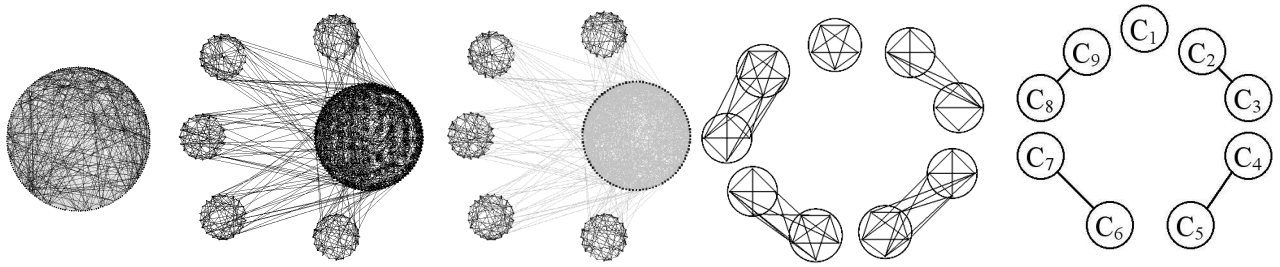

Figure 1: (a) An opaque representation of a composed CSP with 5 satellites of size 20 and a central component of size 100. (b) Redrawing clarifies some relationships. (c) Darker edges represent tighter constraints. (d) Prior to search for a solution, 9 clusters (circled for clarity) reach some portion of every satellite and avoid the central component. (e) Resultant sparse secondary structure.

subproblems like those in Figure 1(d). Section 5 uses the results of Section 3 to hypothesize variable-ordering heuristics that exploit sparse secondary structure like Figure 1(e) based on those subproblems, and Section 6 tests them on several classes of problems, including some standard benchmarks of varying sizes and some challenging real-world problems. The final section discusses the results and the tradeoff between exploration and exploitation.

2 Background and related work

A CSP is a set of variables, each with an associated domain of values, and a set of *constraints*, each of which restricts how some subset of variables (its *scope*) can be bound simultaneously. (Henceforward, CSPs are assumed to be *binary*, that is, all constraints have scope size no more than two.) The *density* of a CSP is the fraction of the possible pairs of variables that are represented as constraints. The *tightness* of a constraint is the percentage of value tuples it excludes from the Cartesian product of the domains of its variables. A *constraint graph*, such as Figure 1(a), represents each variable as a vertex and each constraint as an edge between its pair of variables. *Adjacent* variables have a common constraint and are called *neighbors*. The (static) *degree* of a variable is the number of its neighbors.

A *partial instantiation* is a set of value assignments to some of a CSP's variables from their respective domains. A *future* variable is one unbound in the current partial instantiation. A *full instantiation* assigns a value to every variable. A full instantiation that satisfies all the constraints is a *solution*. The solution paradigm used here is *global search*, complete search that finds a solution or proves that none exists. (In either case, the problem is labeled "solved."). (Local search on problems with some non-random structure, like those addressed here, can be ineffective (Gent et al., 1999).) In global search, one variable at a time is selected and assigned some value from its domain.

After each value assignment, *propagation* removes from the domains of the future variables all values it shows inconsistent with the current partial instantiation, producing *dynamic domains*. During search, a variable's *dynamic degree* is the number of its neighbors that are future variables. A simple propagation method, *forward checking* (*FC*), removes from the dynamic domains of the neighbors of a just-bound variable any values inconsistent with its own value. MAC-3, which maintains *arc consistency* (*AC*), does more work than FC: it enqueues the edges to all the unvalued neighbors of the just-bound variable, and checks each element of the queue for domain reduction (Sabin and Freuder, 1997). Whenever a variable's domain is reduced, MAC-3 enqueues every constraint between that variable and its unvalued neighbors. If a domain becomes empty (a *wipeout*) search backtracks, *retracting* previously assigned values.

The experiments described here initialized problems with AC, and used chronological backtracking and MAC-3, but are in no way restricted to them. They evaluate CSP search by time (in CPU seconds) and number of *nodes* (partial instantiations) explored. Any difference cited here was statistically significant at the 95% confidence level under a one-tail *t*-test.

The fail first principle underlies many traditional variable-ordering heuristics intended to speed global search (Bessière, Chmeiss and Saîs, 2001; Gent et al., 1996; Smith, 1999). *MinDom* seeks to minimize the branch factor of the search tree; it prefers variables with small dynamic domains. *MaxDeg* focuses on variables with many constraints; it prefers variables with high dynamic degree. *MinDomDeg*, a traditional favorite, combines *MinDom* and *MaxDeg*. It prefers variables that minimize the ratio of their dynamic domain size to their dynamic degree.

Some CSP heuristics learn during search (Boussemart et al., 2004; Refalo, 2004). Such learning summarizes difficulties that arise as a result of assignments made to variables at the top of the search tree. In contrast, the work reported here seeks to predict which variables should be assigned first, and learns weights for constraints as a support rather than a central focus. As in (Boussemart et al., 2004), it initializes the weight of every constraint to 1. Whenever propagation along a constraint induces a wipeout, the weight of that constraint is incremented by 1. The variable-ordering heuristic *MaxWdeg* maximizes the *weighted degree* of a variable, the sum of the weights of the constraints between it and its future-variable neighbors. Alternatively, *MinDomWdeg* minimizes the ratio of dynamic domain size to weighted degree. *MinDomDeg*, *MaxWdeg*, and *MinDomWdeg* were the three heuristics timed for the problem in Figure 1.

Other variable-ordering heuristics respond to structure detected in the constraint graph. A CSP whose graph is an arc-consistent tree can be solved without backtracking (Freuder, 1982; Freuder, 1994). SAT problems generated with unsatisfiable large cyclic cores have stumped many proficient SAT solvers (Hemery et al., 2006). To reduce a cyclic CSP to a tree, a solver could first identify and then address some heuristic approximation of the (NP-hard) minimal cycle cutset (Dechter, 1990). Cycle-ridden problems like those addressed here, however, have cycle cutsets far too large to provide effective guidance. Most related work on elaborate structural features that might facilitate search ignores the tightness of individual constraints and is primarily theoretical or incurs considerable computational overhead unjustifiable on easy problems (e.g., (Cohen and Green, 2006; Freuder, 1985; Gompert and Choueiry, 2005; Gottlob, Leone and Scarcello, 1999; Gyssens, Jeavons and Cohen, 1994; Pearson and Jeavons, 1997; Samer and Szeider, 2006; Weigel and Faltings, 1999; Zheng and Choueiry, 2005)).

The heavily constrained, highly interactive subproblems our algorithms identify and exploit in a CSP are called *clusters*. "Cluster" has been used elsewhere to describe aggregations of data, portions of a solution space (Kroc, Sabharwal and Selman, 2008; Mézard, Parisi and Zecchina, 2002), or relatively isolated, dense areas in a graph (van Dongen, 2000). Other work on clusters as subproblems assumed an acyclic metastructure and addressed two classes

of artificial problems, half the size of those studied here, and offered no structural description or explanation (Razgon and O'Sullivan, 2006).

With respect to a given search algorithm, many CSPs have a relatively small *backdoor*, a set of variables whose correct assignment makes the rest of the search relatively trivial (Williams, Gomes and Selman, 2003). Certification of a backdoor requires examination of the problem's entire search tree, however. Clusters are intended to predict enough of the backdoor to give global search guided by traditional heuristics a considerable advantage, and to explain search failure. Unlike (Hemery et al., 2006; Junker, 2004), cluster-based explanations are available whether or not a problem has a solution.

Many of the experiments here are on composed problems. A *composed* CSP partitions its variables into $s + 1$ connected subsets: s *satellites* of uniform size and a *central component*. Every constraint in a composed problem is either a *link* (between a satellite variable and a central-component variable) or joins two variables in the same subset. There are no edges between satellites. Let $\langle n,k,d,t\rangle$ be a class of CSPs each of which has n variables, maximum domain size k, density d, and tightness t. Then $\langle n,k,d,t\rangle$ s $\langle n',k',d',t'\rangle$ d'' t'' specifies a class of composed problems, each of which has a central component described by $\langle n,k,d,t\rangle$, s satellites in $\langle n',k',d',t'\rangle$, and links with density d'' and tightness t''.

Composed problems were inspired by the hard problems in (Bayardo and Schrag, 1996), and have appeared both in solver competitions and as benchmarks (Lecoutre, 2009). They can be randomly generated so that satellites are particularly dense and tight, the central component and links are relatively sparse and loose, and central-component variables have generally higher degrees than satellite variables. Figure 1 is from the class of composed CSPs *Comp:*

<100, 10, 0.15, 0.05> 5 <20, 10, 0.25, 0.50> 0.12, 0.05

where all variables have domain size 10. Only some problems in *Comp* have a solution. *Comp*'s parameters were chosen so that the individual satellites are far more difficult to solve than the central component, but the degrees of the central-component variables are higher, and therefore more attractive to traditional variable-ordering heuristics. Other classes of problems are investigated in Section 6 as well.

3 How to exploit structural foreknowledge

This section explores the power of foreknowledge about difficult subproblems to guide search. The approaches it tests are not ultimately allowable as variable-ordering heuristics. Rather they gauge how well knowledge about structure supports search, and how best to use that knowledge. Results appear in Table 1.

Standard variable ordering heuristics did poorly on *Comp*. *MinDomDeg* was immediately drawn to the central component and solved only 2 of 50 problems within the time limit. Because links in *Comp* are so few and loose, wipeouts began fairly deep in the search tree, after at least 36 variables had been bound. Retraction only led *MinDomDeg* to repair its partial instantiation of the central component, while the true difficulties lay elsewhere, in the satellites. Both *MaxWdeg* and *MinDomWdeg* initially suffered from the same attraction to the central component. After enough experience within the satellites, they eventually recovered and solved all the problems. Learning lacks the foresight clusters are intended to provide.

Now, consider how heuristics might exploit foreknowledge about the problem. Assume one was given the structure shown in Figure 1(c), and believed that the satellites contained the backdoor. In that case, preference for satellite variables should speed search. Rather than discard traditional variable-ordering heuristics, however, each approach investigated here makes satellites a priority and then breaks ties with *MinDomDeg*. Each approach was given 30 minutes to solve each problem.

The next experiments seek to exploit perfect structural foreknowledge. The variable-ordering heuristic *satellite* examines whether mere presence in a satellite is sufficient to warrant prioritization. This approach binds all 100 satellite variables first, in a random order, and then uses *MinDomDeg* on the central component. On a single run, *satellite* never solved any problem within 30 minutes. (Given its lack of promise, this is the only randomized heuristic that was tested only once. All other non-deterministic experiments here report on an average of 10 runs.)

The variable-ordering heuristic *stay* addresses entire satellites first, one at a time in a random order, before it selects any variable from the central component. *Stay* selects a satellite at random, binds all its variables, and then proceeds to another randomly chosen satellite. Within a satellite and within the central component, *stay* breaks ties with *MinDomDeg*. Guided by the satellites, *stay* with *MinDomDeg* yields dramatically improved results over the traditional heuristics; it averages less than 3 seconds and a 96% smaller search tree than *MinDomWdeg*.

Given that noteworthy improvement, and the fact that the satellites may only estimate the backdoor of a *Comp* problem, the next approach binds only some of the variables in each satellite. If MAC-3 is in use, for example, there would appear to be little point in "finishing" a satellite once it is reduced to only a pair of variables (with at most a single edge between them); *until-2* selects a differ-

Table 1. On 50 *Comp* problems, mean and standard deviation for nodes and CPU seconds, including time to find clusters. Search heuristics appear above the line. Search methods with perfect knowledge (below the line) are not legitimate heuristics because they apply foreknowledge available only to the problem generator, not the search engine. *Until*-11 is therefore only a target.

Heuristic	***Time***		***Nodes***	
MinDomDeg	1728.157	(355.877)	285751.970	(61368.701)
MaxWdeg	123.000	(128.580)	20817.640	(22954.165)
MinDomWdeg	83.580	(38.964)	12519.360	(5811.370)
satellite	No problems solved		—	—
stay	2.848	(3.584)	511.922	(416.345)
until -11	1.612	(1.866)	398.776	(244.112)

ent satellite at that point. The generalization of this approach, *until-i*, instantiates variables within a randomly chosen satellite until all but *i* variables are bound, and then moves on to another randomly chosen satellite. (*Stay* is equivalent to *until*-0.) Within a selected satellite and later, within the central component and any "leftover" satellite variables, *until-i* also uses *MinDomDeg*. We tested a range of values: $i = 2,3,\ldots, 15$.

Surprisingly, search need not stay long in a given satellite. For *Comp*, where the satellites are of size 20, the clear winner was *until*-11, that is, search can address as few as 40% of the variables in a satellite before safely moving on to the next one. In contrast, the variable-ordering heuristic *satellite-i*, which randomly chooses satellite variables that are not among the last *i* future variables in their satellite, performed poorly. (Data omitted.) As *i* increases, *satellite-i* becomes more like *MinDomDeg* alone. Clearly, known satellites speed the solution of *Comp* problems when search addresses them one at a time. The next section describes how knowledge about such dense, tight substructures can be detected automatically, prior to search.

4 *Foretell* finds secondary structure

Intuitively, *Foretell*, the cluster finder described here, assembles sets of tightly related variables whose domains are likely to reduce during search. *Foretell* was inspired by the state-of-the-art work for both speed and accuracy on the DIMACS maximum clique problems (Hansen, Mladenovic and Urosevic, 2004). A *clique* is a maximally dense graph, that is, one with all possible edges between its variables. Let a *near-clique* be a clique with a few missing edges. *Foretell* searches for subproblems that are tight near cliques, where the tightness of a subproblem is the product of the tightness of the constraints that it includes. Note, for example, the missing edges in the clusters of Figure 1(d).

Foretell is based on Variable Neighborhood Search (*VNS*). (The "variable" in VNS refers to changing neighborhoods, not to CSP variables.) VNS is a local search meta-heuristic that succeeds on a wide range of combinatorial and optimization problems (Hansen and Mladenovic, 2003). VNS works outward from an initial solution (Figure 2, line 1) in a relatively small neighborhood in a graph through *k* pre-specified, increasingly large neighborhoods (lines 2–3). Each neighborhood restricts the current options; as VNS iterates, each new neighborhood provides a larger search space. Within a neighborhood, Variable Neighborhood Descent (*VND*) is a local search that tries to improve the current solution (*best-yet*) according to a metric, *score*. A better local optimum resets *best-yet* and returns to the first neighborhood (lines 6–9); otherwise search proceeds to the next neighborhood (lines 10–11). *Shaking* (line 5) shifts search within the current neighborhood and randomizes the current *best-yet* to explore different portions of the search space. As *index* increases, the neighborhoods become larger so that the shaken version of *best-yet* becomes less similar to *best-yet* itself. The user-specified stopping condition (line 4) is either elapsed time or movement through some number of increasingly larger neighborhoods without improvement.

```
1  best-yet ← initial-solution
2  index ← 1
3  neighborhood ← neighborhood(index)
4  until stopping condition or index = k
5     unless index = 1, best-yet ← shake(best-yet, index)
6     local-optimum ← local-search(best-yet, neighborhood)
7     If score(local-optimum) > score(best-yet)
8        then   best-yet ← local-optimum
9               index ← 1
10       else   index ← index + 1
11    neighborhood ← neighborhood(index)
```

Figure 2. A high-level description of VNS meta-heuristic search through *k* neighborhoods. The initial solution, the *score* metric, and the local search routine vary with the application.

VND greedily extends *best-yet* from *neighborhood*. Once its greedy steps are exhausted, VND repeatedly interchanges one element of its current solution for two elements in *neighborhood*. In the search for a maximum clique, for example, VND swaps out some variable *v* in *best-yet* for two adjacent variables that are not neighbors of *v* and were not in *best-yet*, but are neighbors of all the other variables in *best-yet*. Ties are broken greedily, that is, to maximize the variable's degree. An alternative produced by VND replaces *best-yet* only if it outscores it.

Foretell adapts VNS to detect multiple subgraphs that are clusters. It relies on the *pressure* on a variable *v*, the probability that, given all the constraints upon it, when one of *v*'s neighbors is assigned a value, at least one value will be excluded from *v*'s domain. Precise calculation of the series that defines pressure is computationally expensive. Instead, we devised an algorithm to quickly approximate the first term in that series, corrected to avoid bias in favor of variables with high degrees or large domains. For variable V_i with domain size D_i, neighbors N_i and constraint with tightness t_{ik} between V_i and $V_k \in N_i$, the approximate pressure on V_i, given the constraints on it, is

$$p(V_i) = \frac{1}{\text{degree}(V_i)} \sum_{V_k \in N_i} \frac{\begin{pmatrix} (D_i - 1) \cdot D_k \\ (1 - t_{ik}) D_i \cdot D_k \end{pmatrix}}{\begin{pmatrix} D_i \cdot D_k \\ (1 - t_{ik}) D_i \cdot D_k \end{pmatrix}} \qquad [1]$$

A cluster's *score* is the ratio of the product of its number of variables and density to its average edge tightness. No cluster is returned unless it has at least 3 variables.

To find multiple clusters in a problem, *Foretell* finds a first cluster, removes those variables, and then iterates to find the next cluster among the remaining variables and their constraints. Clusters are typically (but not always) detected in decreasing size order. Because this is local search, some variation is expected from one pass to the next. The maximum neighborhood index was taken from the original work on maximum cliques: the minimum of 10 and the

current cluster size (Hansen, Mladenovic and Urosevic, 2004).

For our purposes, the *secondary structure* of a CSP is itself a graph (e.g., Figure 1(e)), where each cluster is represented as a node and each set of constraints between variables in two clusters as a single edge. A *sparse* secondary structure is one with few edges in it. Next we develop heuristics for CSPs with sparse secondary structure.

5 Exploiting sparse secondary structure

This section seeks to exploit clusters detected automatically by *Foretell*, much the way foreknown satellites were used in Section 3. *Foretell* never found a cluster larger than 6 variables in a *Comp* problem; instead it found multiple (disjoint) clusters in individual satellites, clusters that covered satellites only partially. The primary question thus becomes how best to exploit clusters. Is it, for example, better to shift from one cluster to another during search, or to solve them one at a time? And if one at a time, in what order should the clusters be considered? Perhaps one would address the cluster that at the moment is the tightest. The true *dynamic tightness of a cluster* is the ratio of the number of tuples that satisfy its unbound variables under the current partial instantiation to the product of their dynamic domain sizes. That is too expensive to calculate repeatedly, as is a dynamic version of pressure in (1). Instead, a cluster's dynamic tightness is estimated here as the ratio of the product of the current domain sizes of those variables to the product of their original domain sizes.

The variable-ordering heuristic *tight* selects a variable from the (estimated) dynamically tightest cluster. Search guided by *tight*, however, could shift from one cluster to another, and therefore from one satellite to another in *Comp*, the way the poorly-performing *satellite* did. The improvement produced by *stay* in Table 1 therefore inspired heuristics that treat one cluster at a time. *Concentrate* chooses a cluster at random, selects variables from it until all of them are bound, and then selects the next cluster at random. In contrast, *focus* selects the (estimated) dynamically tightest cluster, selects variables from it until all of them are bound, and then uses estimated dynamic tightness to select the next cluster. *Concentrate-i* and *focus-i* are analogous to *until-i*; they instantiate within a cluster until all but *i* of its variables have been bound. In all these heuristics, if clusters have the same maximum tightness, ties are broken by maximum dynamic cluster size.

Given the vagaries of local search, one cannot expect VNS to produce an adequate set of clusters every time. Rather than allot substantial time to VNS (which should ultimately find adequate clusters that way), *MinDomWdeg* supports cluster-guided search as a tiebreaker. It is very slightly slower than *MinDomDeg* but it provides backup if Foretell's local search is simply "unlucky." Learning is there to help, although it is rarely necessary.

Table 2. Cluster-guided search speeds traditional heuristics on *Comp* by more than an order of magnitude. Average and standard deviation are shown for nodes and time in CPU seconds, including time for cluster detection. Data above the line is repeated from Table 1. Except for *MinDomDeg*, every method solved every problem. *Focus* is statistically significantly better (in bold) than all the heuristics tested. *Until*-11 is a target, not a legitimate heuristic; it applies foreknowledge about structure available only to the problem generator, not to the search engine.

Heuristic	Time		Nodes	
MinDomDeg	1728.157	(355.877)	285751.970	(61368.701)
MaxWdeg	123.000	(128.580)	20817.640	(22954.165)
MinDomWdeg	83.580	(38.964)	12519.360	(5811.370)
until-11	1.612	(1.866)	398.776	(244.112)
tight	4.705	(6.252)	505.296	(718.029)
concentrate	5.461	(5.628)	836.434	(876.539)
focus	**4.311**	(2.411)	**497.964**	(324.327)
focus-1	5.267	(3.215)	516.406	(425.739)
focus -2	8.713	(22.442)	1371.338	(2765.681)

6 Results

In the following experiments, each heuristic had 30 minutes to solve each problem. Data for all non-deterministic algorithms, including those involving clusters, is reported as an average across 10 runs. For the heuristics that use cluster detection, time *e* is allocated to VNS per cluster. Thus a problem in which *s* clusters were detected could require up to *se* time. The total VNS time required to find clusters is included in all timing data.

On *Comp* problems, *Foretell* finds clusters that form a secondary structure remarkably like foreknowledge. It found between 6 and 19 clusters per problem, all of sizes 3 to 6. It found at least one cluster in every satellite in every problem on every run. A typical result appears in Figure 1(d). With $e = 0.2$ seconds per cluster, VNS search time averaged 2.111 seconds per problem, 45% of the total time.

Clusters guide search in *Comp* effectively, as shown below the line in Table 2. *Concentrate*'s weaker performance clearly indicates that the order in which clusters are addressed is important. Unlike *stay*, however, *focus* appears to need to finish a cluster to produce its best performance. Essentially, by $i = 3$, both *concentrate-i* and *focus-i* dete-

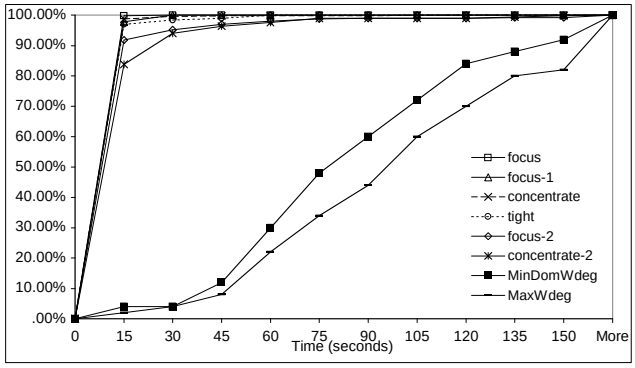

Figure. 3. Cumulative percentage of 50 *Comp* problems solved. Solvers were allocated 30 minutes per problem. Because it solved only 2 of the problems, *MinDomDeg* was omitted.

riorate to *MinDomWdeg*. (Data omitted.) A full graphic comparison appears in Figure 3.

The strongest cluster-based search heuristic, *focus* with *MinDomWdeg*, was also tested on all the composed problems on the benchmark website (Lecoutre, 2009), where there are 10 problems per class. The upper portion of Table 3 reports those results. *MinDomDeg* could solve only 9 of these problems. That and the search tree sizes for *MinDomWdeg* suggest that these benchmarks are easier than *Comp*. On six classes, *focus* once again provided an order of magnitude speedup. Note that, for a fixed central component size, *focus* scales about linearly with problem size.

Composed problems offer an opportunity to explore the impact and management of difficult subproblems. Their uniformity, however, is unlike real-world problems where tightness and density vary broadly. This raises questions about the presence and usefulness of sparse, cluster-based secondary structure in other difficult problems. We tested *RLFAP* problems (data for radio broadcasting (Cabon et al., 1999)) and driver-log problems from (Lecoutre, 2009). Most were easy enough for *MinDomDeg*; RLFAP scene 11 and driverlogw-08cc and driverlogw-08c provided more of a challenge. *Foretell* detected sparse secondary structure on all three, and Table 3 shows that, *focus* was statistically significantly faster on them. Observe that, despite some small variation in *Foretell*'s output from one run to the next on scene 11, the search tree size was always the same.

7 Discussion

7.1 Detecting clusters

Density and tightness are synergistic. Earlier work (Epstein and Wallace, 2006) tested them separately on classes of smaller, considerably easier composed problems, ones that even *MinDomDeg* could solve. A heuristic that prioritized variables by tightness roughly halved *MinDomDeg*'s search time. A heuristic that prioritized variables by density (with VNS-based near-clique detection) consumed about a third of the time. When combined in an earlier version of *Foretell*, however, density and tightness did an order of magnitude better, and produced nearly backtrack-free search trees (Epstein and Wallace, 2006). On smaller composed problems there with one or two satellites, clusters did not harm performance, and they sometimes improved it.

Methods to detect tight, dense subproblems must not only be incisive, they must also scale. Every real-world problem that we tested (i.e., all the RLFAP and driver problems) contained clusters that *Foretell* found fairly quickly. On *Comp*, 0.2 seconds per call to find a cluster sufficed. For RLFAP scene 11, which has roughly 3 times as many variables as a *Comp* problem, 0.6 seconds per cluster sufficed. This suggests that *Foretell* scales linearly.

Composed problems are built to confound traditional heuristics in a particular way, while real-world problems are merely difficult. Not surprisingly, the performance improvements on real-world problems (below the line in Table 3) are noteworthy, but less dramatic. The insights provided by the secondary structure, however, could prove meaningful to a user. For example, people who know Scene 11 well may be interested in *Foretell*'s identification, on every run, of the same clusters (sizes 5, 6, 13, and 16), only two of which are linked in the constraint graph.

Although the experiments described here are on binary CSPs, in principle there is nothing in *Foretell* that would restrict it to binary problems. As long as there is some estimate of the tightness of a constraint, it is possible to estimate the pressure on a variable and to detect sets of mutually constrained, variables by local search. The swap and score functions would require only some modification.

7.2 Why *focus* works

Variable-ordering heuristics usually do not consider persis-

Table 3. Preference for variables in clusters improves search. Numerical identifiers are benchmark problems from (Lecoutre, 2009). For example, 25-1-80 is <25, 10, 0.67, 0.15> 10 <8, 10, 0.79, 0.50> 0.01, 0.05 here. Data for *Foretell* includes number of clusters, average cluster size, and maximum cluster size, averaged over 10 runs. Mean and standard deviation over 10 runs is provided for *focus.* All improvements over *MinDomWdeg* are statistically significant; order of magnitude improvements are in bold.

				MinDomWdeg		***Foretell's clusters***			***Focus Time***		***Focus Nodes***	
Problem	***d***	***t′***	***d″***	***Time***	***Nodes***	***Count***	***Size***	***Max***	μ	σ	μ	σ
25-1-2	0.667	0.65	0.010	1.007	553.00	1.01	5.770	5.77	**0.019**	0.003	**41.40**	1.363
25-1-25	0.667	0.65	0.125	0.913	465.70	2.30	5.597	5.90	**0.042**	0.021	**41.60**	1.287
25-1-40	0.667	0.65	0.200	1.097	473.80	5.00	5.372	6.40	**0.073**	0.016	**41.50**	1.210
25-1-80	0.667	0.65	0.010	0.951	308.00	5.60	5.281	6.08	0.262	0.246	94.50	71.805
25-10-20	0.667	0.50	0.010	2.485	670.10	10.17	5.197	5.58	0.882	0.466	192.07	149.883
75-1-2	0.216	0.65	0.003	3.330	1171.70	1.00	5.690	5.69	**0.044**	0.005	**91.60**	1.504
75-1-25	0.216	0.65	0.042	3.289	1084.40	5.40	5.242	6.46	**0.146**	0.121	**91.40**	1.287
75-1-40	0.216	0.65	0.067	2.972	960.90	4.60	5.292	5.80	**0.153**	0.142	**91.30**	1.275
75-1-80	0.216	0.65	0.133	2.317	595.20	9.09	4.864	5.90	0.365	0.167	181.40	21.687
Comp	0.150	0.50	0.120	83.580	12519.40	11.00	4.309	5.15	**4.311**	2.411	**497.96**	324.327
RLFAP scene 11	—	—	—	58.034	2777.00	38.10	7.912	16.00	**51.133**	1.285	**1557.00**	0.000
Driverlogw 08cc	—	—	—	134.281	4200.00	3.00	34.333	45.00	**87.842**	3.712	**2983.70**	14.100
Driverlogw 08c	—	—	—	149.449	4136.00	3.00	34.333	45.00	**83.622**	3.406	**2815.30**	3.900

tence in a "geographic area" of a problem. Nonetheless, that was clearly *satellite*'s mistake —even with foreknowledge about *Comp*, it *satellite hopped*, that is, it failed to address enough variables in the same satellite consecutively. *Stay* forbade satellite hopping and resulted in a considerable improvement. Analogously, *cluster hopping* occurs when a heuristic fails to address enough variables in the same cluster consecutively. Because constraints within a cluster are selected for above average tightness, once any variable in a cluster has been bound, propagation is likely to reduce the domains of the other variables in that cluster. As a result, variables in a partially-instantiated cluster are more likely to have smaller domain sizes and make their cluster even more attractive to cluster-guided search. *Tight* was permitted to cluster hop, while *focus* and *concentrate* both explicitly forbade it. *Focus* does better, however, because it uses knowledge about clusters to select one.

Happily, remaining in a cluster in *Comp* is likely to encourage remaining in any additional clusters within the same satellite. In problems that are not composed, any other region with sufficiently dense and/or tight connections to a cluster should also have the domains of its variables reduced when those of the related cluster are instantiated. In this way, cluster-guided search results in a sequence of decisions that persist in a particularly constrained region of the graph.

Reducing a subproblem to an arc-consistent tree (which always has at least one solution) would make it safe to continue on to another subproblem. Binding w variables in a subproblem of size s with density d, leaves a tree only if

$$d\binom{s-w}{2} \le s-w-1 \text{ ,that is, } w \ge s-\frac{2}{d}$$

(Of course, this only makes a tree possible, not certain.) For *Comp* satellites, $s = 20$ and $d = 0.25$, so that there is no possibility of a tree unless $w \ge 12$, that is, we have bound 12 variables and 8 remain (*until*-8). Our empirical results, however, show that *until*-11 minimizes both time and nodes (using a one-tail t-test at the 95% confidence level). This ability to leave behind a (necessarily) cyclic subgraph is probably attributable to propagation. The occasional retraction back to a "finished" satellite proved less costly than binding a few more variables in the current satellite before moving on to the next one. Because clusters in *Comp* average $s = 4.309$, however, $w \ge 0{,}309$, that is, only *focus*–0 is safe, which is exactly what our results indicate.

7.3 Clusters and search

The performance of perfect foreknowledge on *Comp*, as embodied by *until-11*, is the gold standard. *Until-11* knows a superset of the backdoor and exploits it. Inspection indicates that the backdoor is probably no more than 35 variables for a *Comp* problem. The last retraction on a *Comp* problem under *focus* was at a node where an average of 15.588 variables had been bound, with a maximum of 62.

The two driverlog problems differ in the tuples they allow, but they have the same primary structure and the same tightness on their constraints. On every run, *Foretell* found the identical sparse secondary structure in the two problems, that is, exactly the same clusters among the 408 variables: 3 nodes with 2 edges. That *focus* improves search on them both confirms its ability to manage secondary structure dynamically.

Clusters explain why a problem is difficult to solve or has no solution at all. The variables in a cluster mutually constrain one another in an intense, highly restrictive way. A user confronted with an unsolvable real-world problem could use clusters to reconsider its specifications, or at least to understand why it is unsolvable. For example, *until-11* searched within at most 3 satellites before it reported the insolvability of Figure 1. To demonstrate the insolvability of the *Comp* problem in Figure 1, *focus* bound only 12 variables drawn from 3 clusters found by *Foretell*. Those 3 clusters, two of size 5 and one of size 4, provide a concise and meaningful explanation.

We have also investigated the tradeoff between exploration (here, local search to detect secondary structure) and exploitation (use of that structure for variable ordering). Too much time devoted to *Foretell* before search wastes CPU cycles; too little may not explore the problem sufficiently. For an extensive secondary structure analysis, e should be so large that *Foretell* averages less time searching for clusters than the total allotted to it, that is, it should leave the loop in the algorithm in Figure 2 because of the index. To solve a CSP, however, we have found that a far smaller e (0.2 – 0.6 seconds) suffices.

Several enhancements are currently under development. Not every problem needs *Foretell*. The "right" clusters are not necessarily many, but incisive, so *Foretell* could partition less than the entire problem. Other problems have more dense secondary structures and may therefore respond better to other cluster-based orderings. Finally, *focus* might attend more closely to learned weights before all the cluster variables have been bound.

All the experiments reported here ran within *ACE*, the Adaptive Constraint Engine (Epstein, Freuder and Wallace, 2005). ACE is a highly modular and flexible research tool that collects substantial data; it is not honed for speed. Nonetheless, the concomitant reductions in checks and nodes searched suggest that clusters will accelerate other, more agile solvers as well. For an easy problem, no clusters are necessary, and any reasonable amount of time spent on cluster detection will have no noteworthy impact. For more challenging problems, however, cluster-guided search substantially accelerated search with off-the-shelf heuristics on problems with sparse secondary structure. Given their acuity and explanatory ability, clusters are a worthwhile preprocessing step.

Acknowledgments. ACE is a joint project with Eugene Freuder and Richard Wallace of the Cork Constraint Computation Centre. Thanks go to Pierre Hansen for helpful discussions on VNS. This work was supported in part by the National Science Foundation under awards IIS-0811437 and IIS-0739122.

References

Bayardo, R. J. J. and R. Schrag 1996. Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances. In *Proceedings of CP-1996*, 46-60. Cambridge, Springer Verlag.

Bessière, C., A. Chmeiss and L. Saîs 2001. Neighborhood-based Variable Ordering Heuristics for the Constraint Satisfaction Problem. In *Proceedings of CP2001*, 565-569. Berlin, Springer Verlag.

Boussemart, F., F. Hemery, C. Lecoutre and L. Sais 2004. Boosting systematic search by weighting constraints. In *Proceedings of ECAI-2004*, 146-149. IOS Press.

Cabon, R., S. De Givry, L. Lobjois, T. Schiex and J. P. Warners 1999. Radio Link Frequency Assignment. *Constraints* 4: 79-89.

Cohen, D. A. and M. J. Green 2006. Typed Guarded Decompositions for Constraint Satisfaction. In *Proceedings of CP2006*, 122-136. Nantes, Springer Verlag.

Dechter, R. 1990. Enhancement schemes for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence* 41: 273-312.

Epstein, S. L., E. C. Freuder and R. J. Wallace 2005. Learning to Support Constraint Programmers. *Computational Intelligence* 21(4): 337-371.

Epstein, S. L. and R. J. Wallace 2006. Finding Crucial Subproblems to Focus Global Search. In *Proceedings of ICTAI-2006*, 151-159. Washington, D.C., IEEE.

Freuder, E. C. 1982. A Sufficient Condition for Backtrack-Free Search. *JACM* 29(1): 24-32.

Freuder, E. C. 1985. A Sufficient Condition For Backtrack-Bounded Search. *JACM* 32(4): 755-761.

Freuder, E. C. 1994. Exploiting Structure in Constraint Satisfaction Problems. In *Proceedings of Constraint Programming: NATO ASI*, 54-79. Parnu, Estonia, Springer.

Gent, I., E. MacIntyre, P. Prosser, B. Smith and T. Walsh 1996. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'99*, 179-193. Cambridge, MA, Springer Verlag.

Gent, I. P., H. H. Hoos, P. Prosser and T. Walsh 1999. Morphing: Combining Structure and Randomness. In *Proceedings of AAAI-1999*, 654-660. Orlando, AAAI.

Gompert, J. and B. Y. Choueiry 2005. A Decomposition Techniques For CSPs Using Maximal Independent Sets And Its Integration With Local Search. In *Proceedings of FLAIRS-05*, 167-174. Clearwater Beach, FL, AAAI Press.

Gottlob, G., N. Leone and F. Scarcello 1999. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence* 124(2): 243-282.

Gyssens, M., P. G. Jeavons and D. A. Cohen 1994. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence* 66(1): 57-89.

Hansen, P. and N. Mladenovic 2003. Variable Neighborhood Search. *Handbook of Metaheuristics*. Glover, F. W. and G. A. Kochenberger. Berlin, Springer**:** 145-184.

Hansen, P., N. Mladenovic and D. Urosevic 2004. Variable neighborhood search for the maximum clique. *Discrete Applied Mathematics* 145: 117-125.

Hemery, F., C. Lecoutre, L. Sais and F. Boussemart 2006. Extracting MUCs from Constraint Networks. In *Proceedings of ECAI-2006*, 113-117. Riva del Garda.

Junker, U. 2004. QuickXplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of AAAI-04*, 167-172.

Kroc, l., A. Sabharwal and B. Selman 2008. Counting Solution Clusters in Graph Coloring Problems Using Belief Propagation In *Proceedings of NIPS-08*, 873-880. Vancouver.

Lecoutre, C. 2009. "Benchmarks in XCSP 2.1 — XML representation of CSP/WCSP/QCSP instances." from http://www.cril.univatois.fr/~lecoutre/research/benchmarks/benchmarks.html.

Mézard, M., G. Parisi and R. Zecchina 2002. Analytic and Algorithmic Solution of Random Satisfiability Problems. *Science* 297(5582): 812 - 815.

Pearson, J. and P. G. Jeavons 1997. A Survey of Tractable Constraint Satisfaction Problems. London, Royal Holloway University of London.

Razgon, I. and B. O'Sullivan 2006. Efficient Recognition of Acyclic Clustered Constraint Satisfaction Problems. In *Proceedings of 11th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming at CSCLP 2006*.

Refalo, P. 2004. Impact-based search strategies for constraint programming. In *Proceedings of CP 2004*, 557-571. Toronto.

Sabin, D. and E. C. Freuder 1997. Understanding and Improving the MAC Algorithm. *Principles and Practice of Constraint Programming*. Berlin, Springer Verlag**:** 167-181.

Samer, M. and S. Szeider 2006. Constraint Satisfaction with Bounded Treewidth Revisited. In *Proceedings of CP2006*, 499-513. Nantes, Springer Verlag.

Smith, B. A. and S. A. Grant 1998. Trying Harder to Fail First. In *Proceedings of ECAI 1998*, 249-253.

Smith, B. M. 1999. The Brélaz Heuristic and Optimal Static Orderings. In *Proceedings of CP'99*, 405-418. Alexandria, Virginia, Springer Verlag.

van Dongen, S. 2000. Graph Clustering by Flow Simulation, University of Utrecht.

Weigel, R. and B. Faltings 1999. Compiling Constraint Satisfaction Problems. *Artificial Intelligence* 115: 257-287.

Williams, R., C. Gomes and B. Selman 2003. On the Connections between Heavy-tails, Backdoors, and Restarts in Combinatorial search. In *Proceedings of SAT 2003*, 222-230.

Zheng, Y. and B. Y. Choueiry 2005. Applying Decomposition Methods to Crossword Puzzle Problems. In *Proceedings of CP 2005*, 874. Sitges, Spain, Springer Verlag.