Cluster-based Modeling for Constraint Satisfaction Problems

Susan L. Epstein[1,2] and Xingjian Li[2]
Department of Computer Science
Hunter College and The Graduate Center of The City University of New York
susan.epstein@hunter.cuny.edu, xli1@gc.cuny.edu

Abstract

Observations on a low-level description of a constraint satisfaction problem are assembled here to learn a higher level, structural model, a *cluster graph*. The model is built from unusually dense and tight subproblems (*clusters)* detected during local search. Heuristics then assemble the clusters into a new, learned representation and use it effectively to guide global search for a solution. A cluster graph provides insight into a large, complex problem, and can be exploited to solve it as much as an order of magnitude faster. Additional learning protects against the occasional inadequacies of local search. The tradeoff between structure learning and performance is also examined.

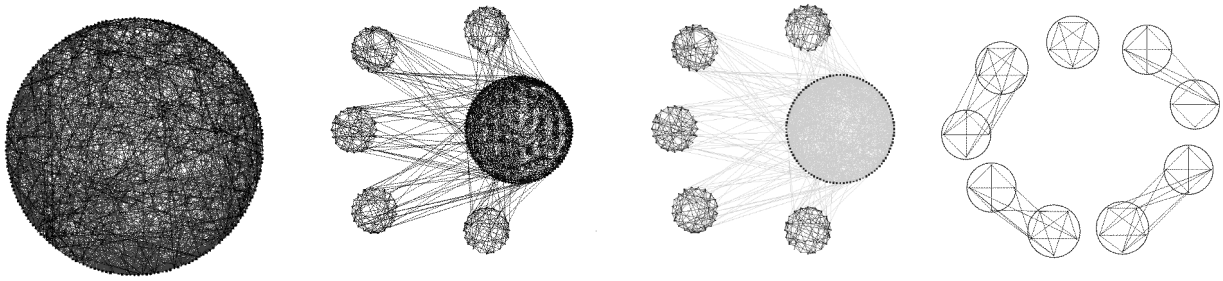
1 Introduction

Many challenging real-world problems can be cast as constraint satisfaction problems (*CSPs*). The thesis of this work is that it is possible to predict and then exploit the most difficult parts of a CSP, both to solve it and to explain it. A *cluster graph* is a structural model that displays relationships among difficult CSP subproblems (*clusters*). The principal result here is that it is feasible to construct a cluster graph for a complex CSP quickly, before search for a solution, and then exploit it effectively to solve the problem. Indeed, the time to construct a cluster graph and then solve with it is faster than search with traditional heuristics by much as an order of magnitude on some classes of difficult CSPs. A cluster graph also provides a higher-level formulation of a problem that is more meaningful for the user.

A binary CSP is a set of variables, each with a domain of values, and a set of *constraints*, each of which restricts how some pair of variables (*neighbors*) can be bound simultaneously. For insight, people often represent a CSP as a graph where each variable is shown as a node and each constraint as an edge between the pair of variables it restricts. For a large, complex CSP, however, the traditional graph (e.g., Figure 1(a)) offers little insight or guidance.

Intuitively, a solver should address the most difficult parts of a problem first. When search for a CSP's solution assigns values to variables one at a time, the order in which the variables are addressed is crucial to search performance. Such search is typically supported by variable-ordering heuristics that prefer variables incident on many edges in the graph. Difficult subproblems, however, are not necessarily characterized by such variables. Figure 1(b) redraws the graph by extracting some variables from the central circle, and Figure 1(c) darkens its more restrictive constraints. This formulation is clear only to the generator, however. Without that knowledge, the problem could not be solved in 30 minutes by a traditional heuristic. Two learning heuristics (described below) solved it in about 127 and 88 seconds. What is really needed is an effective reasoning mechanism that predicts and exploits difficult subproblems. Under the guidance of the cluster graph in Figure 1(d), the approach described here solved the same problem in 3.56 seconds.

Cluster-based modeling first finds clusters and then exploits them. After background and fundamental definitions in the next section, Section 3 describes *Foretell*, our local



(a) (b) (c) (d)

Figure 1. A cluster graph predicts subproblems crucial to search. (a) An uninformative graph for a CSP places the variables on the circumference of a circle. (b) The hidden structure of the same problem, given perfect knowledge. (c) Darker edges represent tighter constraints in the same problem. (d) A cluster graph displays critical portions of the problem. This graph was detected from (a) by local search.

search mechanism to detect clusters. Section 4 provides cluster graphs for several interesting CSPs. Section 5 provides experimental results for *focus*, our variable-ordering search heuristic that exploits a cluster graph. The final section discusses the role of learning in this approach, and the tradeoffs between structure detection and solution.

2 Background and Related Work

The (static) *degree* of a variable is the number of its neighbors. The *tightness* of a constraint is the percentage of value tuples it excludes from the Cartesian product of the domains of its variables. The *density* of a binary CSP is the fraction of the possible pairs of variables that are represented as constraints. 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, and the *dynamic degree* of a variable is the number of its neighbors that are future variables. A *full instantiation* assigns a value to every variable. A full instantiation that satisfies all the constraints is a *solution*.

Complete search either finds a solution to a CSP or proves that none exists. (If search achieves either, the problem is labeled "solved" here.) *Global search* assigns a value to one variable at a time. After each assignment, *propagation* removes from the domains of the future variables any values shown inconsistent with the current partial instantiation, producing *dynamic domains*. If a dynamic domain becomes empty (a *wipeout*) search *backtracks* (retracts previously assigned values). The experiments reported here use chronological backtracking and MAC-3 propagation to maintain arc consistency (Sabin and Freuder, 1997), but are in no way restricted to either. CSP search is traditionally evaluated by time (in CPU seconds) and by number of *nodes* (partial instantiations) explored.

Many variable-ordering heuristics have been proposed to speed global search (Bessière, Chmeiss and Saîs, 2001; Gent et al., 1996; Smith, 1999). For example, *MinDom* seeks to minimize the branch factor of the search tree; it prefers variables with small dynamic domains. Particularly influential has been the *fail first* principle: address first those variables for which it is difficult to find values that lead to a solution (Haralick and Elliott, 1980). For example, *MaxDeg* focuses upon variables with many constraints; it prefers variables with high dynamic degree. The traditional popular heuristic *MinDomDeg* combines *MinDom* and *MaxDeg*. It prefers variables that minimize the ratio of their dynamic domain size to their dynamic degree.

Another way to fail first is to learn weights to prioritize troublesome variables (Boussemart et al., 2004). This approach initializes the weight of every constraint to 1. Then, each time 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. These learning heuristics are also initially drawn to the central component of the problem in Figure 1, but they eventually recover. *MinDomDeg, MaxWdeg*, and *MinDomWdeg* were the non-cluster heuristic timed on the CSP in Figure 1.

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

A *structured* CSP has characteristics that can be exploited by a specialized method to outperform a more general one. A *composed* CSP is a randomly-generated structured problem that partitions its variables into connected components. One subset is designated as the *central component*; the others are its *satellites*. *Links* (constraints) connect some variables in each satellite to variables in the central component, but there are no constraints between variables in distinct satellites. A class of composed problems

$$<n,k,d,t>\ s\ <n',k',d',t'>\ d''\ t''$$

specifies the central component (n variables, maximum domain size k, density d, and tightness t), s satellites (each with n' variables, maximum domain size k', density d', and tightness t'), and links with density d'' and tightness t''. For example, Figure 1 is an unsolvable problem in

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

Composed CSPs offer an opportunity to explore the impact and management of difficult subproblems. Some, including *Comp,* can respond poorly to traditional CSP ordering heuristics (Bayardo and Schrag, 1996). As indicated by Figure 1(c), the central component is large and easy to satisfy (relatively few, loose constraints) and the links are sparse and loose, but the satellites are denser, with tighter constraints. Traditional variable-ordering heuristics usually begin search in the central component of a *Comp* problem.

Most structure-based work in CSP has focused upon the identification and exploitation of tractable (Dechter and Pearl, 1989; Gyssens, Jeavons and Cohen, 1994; Mackworth and Freuder, 1985) and complex structures (Gompert and Choueiry, 2005). Unlike clusters, however, that work ignores tightness along individual constraints, the crucial distinction between the satellites and the central component in a *Comp* problem.

3 Cluster Detection with *Foretell*

The cluster finder *Foretell* gathers information in advance about 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 graph with all possible edges between distinct variables.) *Foretell* searches for large, dense, tight subproblems that either are cliques or, but for a few missing edges, would be cliques. (Note the missing edges in the clusters of Figure 1(d).)

```
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);VND
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 the non-deterministic VNS meta-heuristic search through k neighborhoods. The initial solution, the *score* metric, and the local search routine vary with the application.

Foretell is based on Variable Neighborhood Search (*VNS*), a local search meta-heuristic that succeeds on a wide range of combinatorial and optimization problems (Hansen and Mladenovic, 2003). (The "variable" in VNS refers to changing neighborhoods, not to CSP variables.) VNS works outward from an initial solution (Figure 2, line 1) in a relatively small neighborhood in a graph through k prespecified, 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, local search 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.

Foretell, adapts VNS to detect multiple subgraphs, and redefines routines for the initial solution, the *score* metric, and local search. For these, *Foretell* relies on the notion of *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, our algorithm quickly approximates the first term in that series, corrected to avoid bias in favor of variables with high degrees or large domains. For a constraint with tightness t between variables V_1 and V_2 with domain sizes D_1 and D_2, respectively, *Foretell* calculates the initial pressure on variable V_1 as:

$$p(V_1) = \frac{1}{\text{degree}(V_1)} \sum_{c \in C \text{ on } V_1} \frac{\binom{(D_1 - 1) \cdot D_2}{(1-t) D_1 \cdot D_2}}{\binom{D_1 \cdot D_2}{(1-t) D_1 \cdot D_2}} \qquad [1]$$

Foretell's initial solution (in line 1 of Figure 2) is a vertex that is the neighbor of every vertex in the graph (often an empty set). Its greedy step maximizes pressure and uses pressure to break ties during swaps as well. Since we seek large, tight, closely related subproblems, *Foretell* scores a cluster according to its number of variables, its density, and the average tightness of its constraints. After each cluster is found, *Foretell* removes the variables within it from consideration, and seeks a new cluster among the variables that remain. Ties unbroken by maximum pressure, are broken by maximum degree, and then, if need be, at random. The minimum acceptable cluster is a clique of size 3, but not every cluster is a clique.

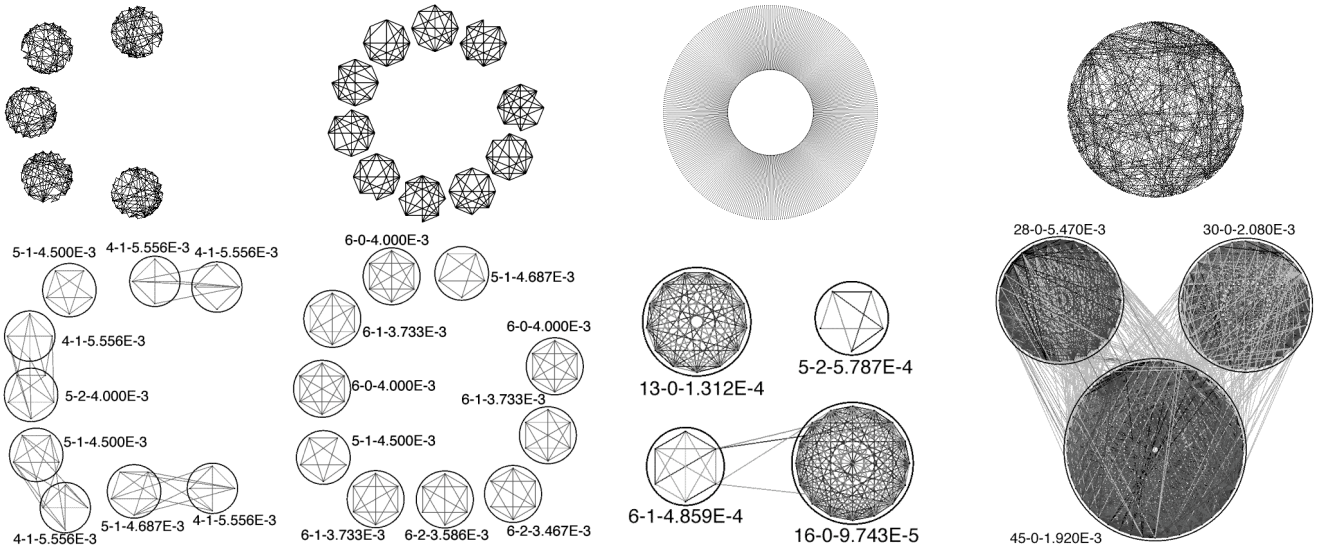


Figure 3. Tight edges (top) and cluster graphs (bottom) for different CSPs display different secondary structures. Each cluster is drawn within a circle; tighter edges are darker. Cluster labels are number of variables, number of additional edges needed to make it a clique, and *Foretell* score. (a) *Comp* problem (b) 25-10-20 problem (c) RLFAP scene 11 with tightness ≥ 0.3 and (d) the driverlogw-08cc problem.

4 Cluster Graphs

Foretell identifies clusters, and the program then joins them with any constraints that connect their variables in the original CSP to automatically generate cluster graphs. Applied to a broad range of benchmark problems taken from (Lecoutre, 2009), this approach produces cluster graphs that make explicit a variety of *secondary structure* among the clusters themselves.

The first category of secondary structure is a set of isolated clusters, or pairs of clusters, in the cluster graphs for composed problems. The cluster graph in Figure 3(a) for a *Comp* problem is suggestive of its tightest edges. Because satellites, with density 0.25, are far from cliques, quite often more than one cluster lies in the same satellite. Thus some clusters are linked. Figure 3(b) shows a similar secondary structure for another composed CSP, designated 25-10-20 in (Lecoutre, 2009) or, in our notation,

<25,10,0.667,0.15> 10 <8,10,0.786,0.5> 0.01, 0.05.

The larger satellite density (0.786) encourages the formation of somewhat larger clusters, but typically leaves too few edges to form two clusters in one satellite. Thus its clusters are isolated from one another.

Clusters are not always composed from only the tightest edges. *RLFAP* is scene 11 of the radio link frequency problems (Cabon et al., 1999). Its many constraints vary dramatically in tightness. Figure 3(c) shows only the tightest, with the 650 variables on two concentric circles; this is a bipartite graph. The cluster graph is considerably more informative. *Foretell* found the same clusters of size 6 and 13 on every run. Note that only two clusters are connected to one another, and that the size-13 cluster is not one of them. These 40 variables are the crux of RLFAP, the part that makes its rapid solution possible. Only 18 of the tightest edges appear in the cluster graph at all, and only two are in the top-priority cluster. We also tested driverlog problems from (Lecoutre, 2009), originally from a planning competition. The one in Figure 3(d) displays a more closely coupled secondary structure. Other driverlog problems also display a path-like secondary structure in their cluster graphs.

Table 1. On 50 *Comp* problems, search with perfect knowledge (above the line) must remain in a satellite to succeed.. *Until*-11 is a target, not a legitimate heuristic; it applies foreknowledge available only to the problem generator, not the search engine. Without perfect knowledge (below the line), cluster-guided search improved the performance of 3 traditional heuristics by more than an order of magnitude. Mean and standard deviation are shown for nodes and CPU seconds, including time *Foretell* uses to find clusters.

Heuristic	Time		Nodes	
satellite	No problems solved			
stay	2.848	(3.584)	511.922	(416.345)
until -11	1.612	(1.866)	398.776	(244.112)
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)
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)

5 Experiments in Cluster-guided Search

This section exploits clusters detected automatically by *Foretell*. All problems were initialized with AC and propagated with MAC-3. Given the non-determinism in many of these approaches, all results are averaged across 10 runs.

The first set of experiments, on *Comp*, tested both traditional variable-ordering heuristics and *perfect-knowledge approaches* that are not ultimately allowable as heuristics because they use information about problem generation to search. The latter gauge how well perfect knowledge about structure supports search, and how best to use that knowledge. Rather than discard traditional variable-ordering heuristics, the perfect knowledge approaches use *MinDomDeg* as a tiebreaker. Each variable ordering had 30 minutes to solve each problem. Results appear in Table 1.

First, we tested three perfect-knowledge approaches. *Satellite* prefers satellite variables, and binds central-component variables last. It never solved any problem. *Stay* repeatedly selects a satellite at random and binds all its variables. It too binds central-component variables last. *Until-i* selects a satellite at random, binds all but *i* of its variables, and then selects another satellite at random. (*Stay* is equivalent to *until*-0.) It binds central-component variables and any "leftover" satellite variables last. We tested i =2, 3,…, 15 on *Comp*; i = 11 was the best. Given perfect knowledge about satellites of size 20, search can address as few as 9 satellite variables before moving on to the next satellite.

Next we tested three traditional approaches *MinDomDeg* solved only 2 problems within the time limit. Inspection indicates that it was immediately drawn to the central component of *Comp*. Because links there are so few and loose, wipeouts did not occur until fairly deep in the search tree, after at least 36 variables had been bound. After retraction, *MinDomDeg* repaired its partial instantiation by re-solving part of the central component, when the true difficulties lay elsewhere, in the satellites. Although the learning heuristics initially suffered from the same attraction to the central component, they both solved all the problems eventually, *MaxWdeg* in about 2 minutes on average, and *MinDomWdeg* in about 84 seconds. Nonetheless, they both lack the foresight that the cluster graph is intended to provide.

The three perfect-knowledge experiments suggested several variable-ordering heuristics without perfect knowledge, fail-first orderings that address clusters first, one at a time. True *cluster tightness* (the ratio of the number of tuples that satisfy its future variables to the product of their dynamic domain sizes) is too expensive to calculate dynamically. It is estimated instead over its future variables as the product of the ratios of their dynamic domain sizes to original domain sizes. Furthermore, given the vagaries of local search and uncertainty about how much time to allocate to VNS, *MinDomWdeg* supports cluster-guided search as a tiebreaker. Thus, learning is there to help, although it is rarely necessary. Each of the following cluster-guided search heuristics breaks ties by maximum dynamic cluster size. *Tight* selects a variable from the (estimated) dynamically tightest cluster. Search guided by *tight* could shift from one cluster to another, and therefore from one satellite to another in *Comp*,

Table 2: Clusters improve performance. Classes above the line are composed, with central component density *d,* link density *d″*, and satellite tightness *t′*. Cluster-guided search uses both *Foretell* and *focus.* Time is in CPU seconds

				MinDomWdeg		*Foretell's clusters*			*Focus*	
Problem	***d***	***t′***	***d″***	***Time***	***Nodes***	***Count***	***Size***	***Max***	***Time***	***Nodes***
25-10-20	0.667	0.50	0.010	2.485	670.10	10.17	5.197	5.58	0.882	350.00
25-1-80	0.667	0.65	0.010	0.951	308.00	5.60	5.281	6.08	0.262	94.50
75-1-80	0.216	0.65	0.133	2.317	595.20	9.09	4.864	5.90	0.365	181.40
25-1-2	0.667	0.65	0.010	1.007	553.00	1.01	5.770	5.77	0.019	41.40
25-1-25	0.667	0.65	0.125	0.913	465.70	2.30	5.597	5.90	0.042	41.60
25-1-40	0.667	0.65	0.200	1.097	473.80	5.00	5.372	6.40	0.073	41.50
75-1-2	0.216	0.65	0.003	3.330	1171.70	1.00	5.690	5.69	0.044	91.60
75-1-25	0.216	0.65	0.042	3.289	1084.40	5.40	5.242	6.46	0.146	91.40
75-1-40	0.216	0.65	0.067	2.972	960.90	4.60	5.292	5.80	0.153	91.30
Comp	0.150	0.50	0.120	83.580	12519.40	11.00	4.309	5.15	4.311	497.96
RLFAP scene 11	—	—	—	58.034	2777.00	38.10	7.912	16.00	51.133	1557.00
Driverlogw 08cc	—	—	—	134.281	4200.00	3.00	34.333	45.00	87.842	2983.70
Driverlogw 08c	—	—	—	149.449	4136.00	3.00	34.333	45.00	83.622	2815.30

the way the (inadequate) *satellite* did. Like *stay*, *concentrate* selects a cluster at random, binds all its variables, and then goes to the next randomly chosen cluster. *Focus* selects the (estimated) dynamically tightest cluster, binds all its variables, and then uses estimated dynamic tightness to select the next cluster. *Focus-i* is analogous to *until-i*; it instantiates within a cluster until all but *i* of its variables have been bound. We applied these variable-ordering heuristics to *Comp*, including total VNS time required to find clusters in all timing data. *Focus* was the most successful; it solved every problem in under 16.720 seconds. In contrast, *MinDomWdeg* solved only 4% in that time. Because *Foretell's* clusters are about one fourth the size of *Comp's* satellites, it proves safer to bind an entire cluster, that is, *i* should be 0.

The second set of experiments is reported in Table 2. It tested the strongest cluster-based search heuristic, *focus* with *MinDomWdeg*, on all the composed problems on the benchmark website (Lecoutre, 2009), where there are 10 problems per class. *MinDomDeg* could solve only 9 of these problems, all in the first 3 classes. The search tree sizes for *MinDomWdeg* suggest that these benchmarks are easier than *Comp*. On the harder classes, *focus* once again provided an order of magnitude improvement. Because composed problems are more uniform in density and tightness than real-world problems, we also tested *RLFAP* and the driverlog problems w-08cc and w-08c from (Lecoutre, 2009). On both, *focus* was statistically significantly faster.

6 Discussion

Earlier work demonstrated that both density and tightness are necessary to predict difficult subproblems (Epstein and Wallace, 2006). It also showed that cluster graphs do not harm performance on smaller composed problems, and may improve it. When the ratio of satellite size to central component size is high, or when a satellite's tightness does not warrant prioritization, clusters are less useful. The very rare presence here of a central component variable in a cluster did not negatively impact cluster-guided search in *Comp*.

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 a 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, *MinDomWdeg* proves that the problem in Figure 1 is unsolvable, but the trace is fairly opaque. *Until*-11 is somewhat more informative: it searched within only two satellites before it reported insolvability. Each of its 10 proofs was different, but most of them used no more than 8 variables in 2 satellites, about 4% of the 200 variables. On the same problem, *focus* offers a more concise and meaningful explanation with 3 clusters.

The tradeoff between task performance and learning is addressed by a final set of experiments (Table 3) that varied *Foretell*'s time allocation per cluster on RLFAP. Under 300 ms., clusters were smaller. At 2000 ms., *Foretell* always

Table 3: The tradeoff between exploration and exploitation averaged across 10 runs on RLFAP. Allocated and actual times per cluster are in milliseconds; search time, time consumed by *Foretell* to find all clusters, and total time are in seconds. Statistics include the average and range of the number of clusters on those runs, their average and maximum size, and their coverage (fraction of variables included in the cluster graph). All cluster search time is included in the total time to solution and includes *Foretell* time.

		Cluster statistics					*Cluster-guided search (times in sec.)*			
Time per cluster (ms.)			***Count***	***Average***	***Max***			***Search***	***Foretell***	***Total***
Allocated	***Actual***	***Count***	***range***	***size***	***size***	***Coverage***	***Nodes***	***time***	***time***	***time***
300	395.318	55.100	7 - 65	6.886	15.600	55.80%	1616.100	35.457	21.782	57.239
400	479.895	38.100	36 - 41	7.912	16.000	44.33%	1557.000	32.849	18.284	51.133
500	573.561	39.600	5 - 69	7.468	14.800	43.49%	1519.000	50.859	22.713	73.572
600	621.343	31.005	5 - 65	7.548	15.700	34.42%	1655.000	43.691	36.031	74.696
800	821.202	46.600	17 - 65	7.769	16.000	53.24%	1532.800	43.269	38.268	81.537
1000	889.542	63.300	63 - 65	7.059	16.000	65.40%	1519.000	36.536	56.308	92.844
2000	1323.429	63.000	63 - 63	7.100	16.000	65.78%	1519.000	33.541	83.376	116.917

finds 63 clusters, and finds the same largest cluster consistently. In the experiments for 1000 and 2000 ms., there is little difference among the cluster graphs and none ($\sigma = 0$) in the resultant search tree size. Total time for search, however, increases because the increased allocation allows *Foretell* more iterations through the loop in Figure 2, during which it tinkers more with the clusters it finds, as indicated by the second column in Table 2. Observe that, if *Foretell* dawdles during local search, it can exceed the allocated time on average. An allocation greater than time per cluster indicates that *Foretell* has done all it can.

There are two ways to think about Table 2. First, if one requires an explanation for the user, the most complete cluster graph can be produced by iteratively increasing the time allocation until it exceeds time per cluster and a consistent number of clusters is found. The second way is informed by some surprising observations. All the errors made on any 400 ms. run were within the cluster graph, and only 2 errors were within the first 300 assignments. Moreover, despite differences in the cluster graphs under the same time allocation, *focus* uses them the same way, that is, the tightest, largest clusters dominate and the standard deviation in the search tree size is 0. Thus the difference between 400 and 2000 ms. is a few variables treated differently. This suggests that effective search does not require the most extensive possible cluster graph, just enough of it to direct search to the hardest subproblems first. Current investigation therefore includes additional termination conditions for *Foretell*.

Although these experiments are on binary CSPs, in principle there is nothing in *Foretell* that restricts it to binary problems. All the experiments reported here ran in *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 (data omitted) 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, cluster-guided search outperformed off-the-shelf heuristics here, even those that learn, in both time and nodes. Given their acuity and explanatory ability, cluster-guided search is a worthwhile CSP search technique, one that learns structural knowledge and then applies it.

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. *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. *CP2001*, 565-569. Berlin, Springer Verlag.

Boussemart, F., F. Hemery, C. Lecoutre and L. Sais 2004. Boosting systematic search by weighting constraints. *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.

Dechter, R. and J. Pearl 1989. Tree Clustering For Constraint Networks. *Artificial Intelligence* 38: 353-366.

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. *ICTAI-2006*, 151-159. Washington, D.C., IEEE.

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. *CP'99*, 179-193. Cambridge, MA, Springer Verlag.

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

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.

Haralick, R. M. and G. L. Elliott 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14: 263-314.

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

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

Lecoutre, C. 2009. "Benchmarks in XCSP 2.1." from http://www.cril.univatois.fr/~lecoutre/research/benchmarks/benchmarks.html.

Mackworth, A. K. and E. C. Freuder 1985. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence* 25(1): 65-74.

Sabin, D. and E. C. Freuder 1997. Understanding and Improving the MAC Algorithm. *CP-97*, 167-181.

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

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