Cluster Graphs as Abstractions for Constraint Satisfaction Problems

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

[1]Hunter College and [2]The Graduate School 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

In a constraint satisfaction problem, the tightness of an individual constraint only describes the influence that the variables within its scope have on one another. Clusters provide a broader view; they are dense, tight subproblems within a problem. A set of clusters for a problem and the links between them provide an abstraction of it. That abstraction can be used to guide search, to curtail inference, and to provide explanations to the user. This work is a hybrid of global and local search, where local search creates an abstraction and then global search exploits it. Heuristics reference clusters to order variables and to propagate more thoughtfully with respect to them. Results are provided on a variety of challenging benchmark problems.

Introduction

The thesis of this work is that an appropriate abstraction of a constraint satisfaction problem (*CSP*) can provide guidance for a solver and insight for a user. Ideally, that abstraction would facilitate search for a solution or be used to prove that no solution exists, to reduce inference during search, and to support concise, pertinent explanations for a user. The principal results of this paper demonstrate these ideas with *clusters*, particularly dense and tight subproblems detected quickly prior to search. For some challenging CSPs, cluster-based abstraction proves a powerful approach to search and explanation.

One traditional representation of a CSP is as a graph. Formally, a CSP is a set of variables, each with a domain of values, and a set of constraints that restrict how those variables can be bound simultaneously. In a binary CSP, each constraint addresses no more than two variables. A *constraint graph* for a binary CSP represents each variable as a vertex and each constraint as an edge between the pair of vertices for the variables it restricts.

An automated graph-drawing program, however, is unlikely to offer much insight into the nature of a CSP. A program that plotted one problem's 200 variables along the circumference of a circle, for example, produced Figure 1(a). The problem's *density* (fraction of possible edges included) obscures any meaningful information. A human-guided rearrangement of the vertices in that graph produced Figure 1(b), which in some sense reveals the problem's "shape." It consists of small subgraphs connected to one another only through another, larger subgraph.

Many traditional CSP variable-ordering heuristics direct search for a solution to variables with high degree in the graph. (Two vertices with an edge between them in the graph are *neighbors*, and the number of neighbors a vertex has is its *degree.*) Such search on the problem in Figure 1 would begin in the larger subgraph — and it would fail, because it ignores tightness.

The *tightness* of a constraint is the fraction of possible

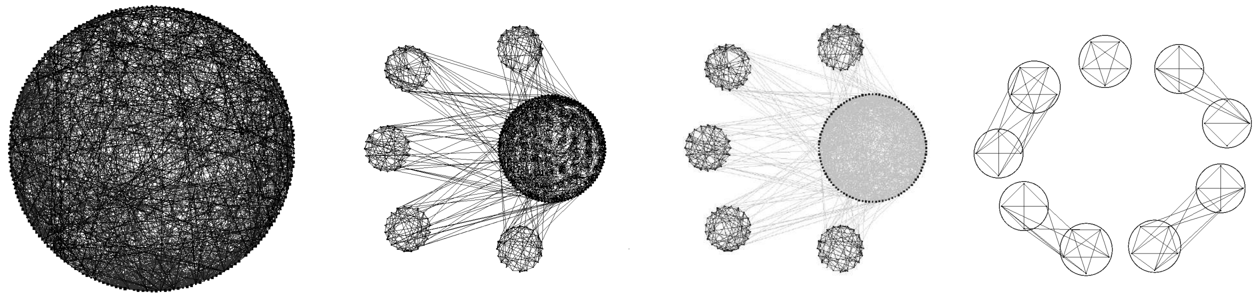
(a) (b) (c) (d)

Figure 1. For the same CSP (a) an uninformative constraint graph plots variables on the circumference of a circle while (b) another constraint graph reveals some relationships. (c) Darker edges represent tighter constraints. (d) Detected from (a) by local search, this cluster graph is an abstraction that highlights each critical subproblems with a circle for clarity.

assignments to its variables that it excludes. Non-uniform constraint tightness dictates that search cannot rely on shape alone. In Figure 1(c), tighter edges are drawn darker. Clearly the edges within the smaller subgraphs are tighter than the now faint edges that lead out from them and tighter than the edges within the larger subgraph.

A *cluster graph* is an abstraction that highlights dense, tight subproblems in a CSP. After a discussion of related work, this paper describes the cluster-detection algorithm that produced the imperfect but incisive cluster graph in Figure 1(d). Subsequent sections describe how clusters are exploited to guide global search, to explain the nature of a problem to the user, and to control inference during search. The final sections demonstrate performance improvements with cluster graphs and discuss current work.

Related Work

A cluster graph identifies subproblems that are dense and tight *before* search for a solution. A cluster graph selects elements of the original graph and groups them together. The variables and constraints not explicit in a cluster graph have influenced its formation (via pressure, described in the next section). Thus a cluster graph captures a kind of *fail-first* (Smith and Grant, 1998) metastructure that anticipates and confronts difficulties. This approach differs, therefore, from methods that relax, remove, or soften constraints. As used here, "cluster," does not refer to aggregations of data, sets of solutions in a search space, or relatively isolated, dense areas in a graph (van Dongen, 2000).

Most structure-based work in CSP has focused upon the identification and exploitation of tractable structures, such as trees (Mackworth and Freuder, 1985), acyclic graphs (Dechter and Pearl, 1987), tree decomposition (Dechter and Pearl, 1989), hinges (Gyssens, Jeavons and Cohen, 1994), and other complex structures (Gompert and Choueiry, 2005). Unlike clusters, however, that work ignores tightness along individual constraints.

With respect to a given search algorithm, the *backdoor* of a CSP is a set of variables that, once assigned values, make the remainder of search trivial (Ruan, Horvitz and Kautz, 2004). A backdoor is typically less than 30% of the variables, but its identification before search is NP-complete. Recent work suggested that both static and dynamic properties should be considered during search for a backdoor (Dilkina, Gomes and Sabharwal, 2007). The formation of a cluster graph prior to search considers both static (initial) shape and potential (dynamic) changes in domain size. A cluster graph would, ideally, contain the backdoor, but no claim is made here that it does so.

An abstraction can be used to simplify a problem temporarily (e.g., (Sacerdoti, 1974)). Its solution is then gradually revised to accept additional problem detail, until the revision solves the original problem. Because a cluster graph is applied in collaboration with the traditional graph, not as a replacement for it, no re-solution is necessary.

Elsewhere we have shown that, given 30 minutes per problem, a heuristic that merely prioritized variables in the smaller subgraphs failed to solve any problem like that in Figure 1 (Epstein and Li, 2009). Thus, individual edge tightness cannot be the sole consideration. That work also investigated a variety of ways to use perfect knowledge, such as that in Figure 1(c), to solve such problems. We showed there that it is important to explore one subproblem at a time. Individual edge tightness overlooks the synergy among a set of variables with many mutual constraints, a synergy that clusters are designed to anticipate.

Cluster Detection

Cluster detection uses *VNS* (Variable Neighborhood Search), a local search metaheuristic (Hansen, Mladenovic and Urosevic, 2004). (The "variable" in VNS refers to changing neighborhoods in the graph, not to CSP variables.) The original VNS application remains the state of the art for maximum clique detection within a graph. (A *clique* is graph with an edge between every pair of distinct vertices.)

Local search begins with an initial solution and then seeks to move it toward some goal with respect to some metric. Figure 2 provides pseudocode for VNS. It repeat-

```
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. *k* = 10 was adopted from (Hansen, Mladenovic and Urosevic, 2004).

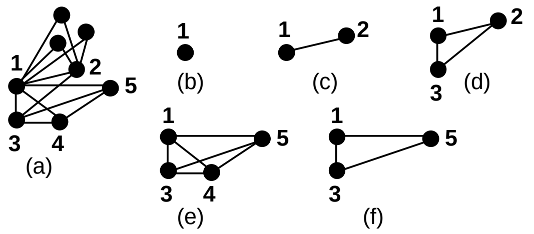

Figure 3. Selected VNS steps to find a maximum clique in graph (a). (b) A starting vertex. (c)-(d) Greedy steps add vertices adjacent to every selected vertex, one at a time. (e) A swap replaces vertex 2 with vertices 4 and 5. (f) VNS for *index* = 1 shakes out one randomly selected vertex. See the text for further details.

edly calls (in line 6) a greedy local search algorithm within a *neighborhood* in the graph, nodes adjacent to some of *best-yet*. The subgraph returned by local search is scored, but only the highest-scoring subgraph is retained. That top-scored subgraph is then shaken (i.e., *index* randomly chosen variables are deleted) to shift search to a new neighborhood, and the shaken graph is resubmitted as a starting point for local search. VNS terminates when it has searched the maximum number of neighborhoods or under some user-specified stopping condition, typically time.

Figure 3 is a sample of steps that might occur during VNS search to find a maximum clique in a simple graph. The initial solution is a vertex that is a neighbor of every vertex in the graph; the local search metric is subset size. VNS adds one vertex adjacent to every vertex in the growing subgraph. (This is the greedy step; ties are broken on maximum degree in the original graph.) When greedy steps are no longer possible, local search swaps out one vertex for a pair of adjacent vertices that are also adjacent to every other vertex in the subgraph, as in Figure 2(e). Eventually neither greedy steps nor swaps can be found. Then the subgraph is returned to VNS, scored, stored if it is the best so far, and then shaken before local search resumes.

Our cluster detection algorithm, *Foretell*, adapts VNS to detect multiple subgraphs, and redefines routines for the initial solution, the *score* metric, and local search. The initial solution is all variables of maximum possible degree in the graph. *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, an algorithm was devised to speed an approximation for the first term in that series, corrected to avoid bias in favor of variables with high degrees or large domains. Let V_i be a variable with domain size D_i and neighbors N_i. Let t_{ik} denote the tightness of the constraint between V_i and $V_k \in N_i$. Then the approximate pressure on V_i is defined by the constraints upon it as:

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

Foretell calculates the initial pressure on every variable, and takes as an initial solution (in line 1 of Figure 2) a vertex that is the neighbor of every vertex in the graph (often an empty set). *Foretell*'s greedy step maximizes pressure (instead of degree, as in Figure 3), 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.

```
Until all variables have values that satisfy all constraints or
some variable has an empty domain
    **Assign** a value to a variable
    **Infer** the impact of that assignment *propagation*
    If a wipeout occurs, **backtrack**
```

Figure 4. Pseudocode for CSP global search.

Cluster-guided Search

Cluster-guided search is a hybrid of local and global search. As in Figure 4, *global search* (henceforward, *search*) iteratively selects a variable, assigns it a value, and then *propagates*, to infer the impact of that assignment on the domains of *future variables* (those not yet assigned a value). Inference calculates *dynamic domains* as it temporarily removes from future variables' domains any values inconsistent with the current assignments. If any dynamic domain becomes empty (a *wipeout*), search *backtracks*, that is, retracts one or more assignments. (The work reported here uses chronological backtracking, but is in no way limited to it.) This search is complete and correct, but often intractable on CSPs with many variables or large domains. A problem is *solved* when search finds either a solution or a proof that none exists.

A variable-ordering heuristic can speed search by directing it to the most troublesome variables first. A traditional favorite, *MinDomDeg*, prefers variables with a small ratio of dynamic domain size to *forward degree* (number of neighbors that are future variables). Given any of the challenging problems used here, however, *MinDomDeg* could rarely solve it in 30 minutes. Thus, we gauge problem difficulty with *MinDomWdeg*, a heuristic that learns weights (Boussemart et al., 2004). Initially every constraint has weight 1. Then, whenever an assignment propagated along that constraint creates a wipeout, the weight of that constraint is increased by one. The *weighted degree* of a variable is the sum of the weights of the constraints that reference it. *MinDomWdeg* prefers variables with a small ratio of dynamic domain size to weighted degree.

Cluster-guided search first uses local search in *Foretell* to create a cluster graph like Figure 1(d), and then directs global search for a solution with *focus*, a cluster-oriented variable-ordering heuristic. *Focus* restricts search to one cluster at a time, and references *MinDomWdeg* to break ties within that cluster. Clusters are prioritized for *focus* according to the product of the ratios of dynamic domain size to original domain size for all future variables in the cluster. This value dynamically estimates the extent to which tuples have been eliminated as possible cluster solutions, another application of the fail-first principle.

Structure in Cluster Graphs

We have applied cluster-guided search to benchmark problems taken from (Lecoutre, 2009). In each case *Foretell*

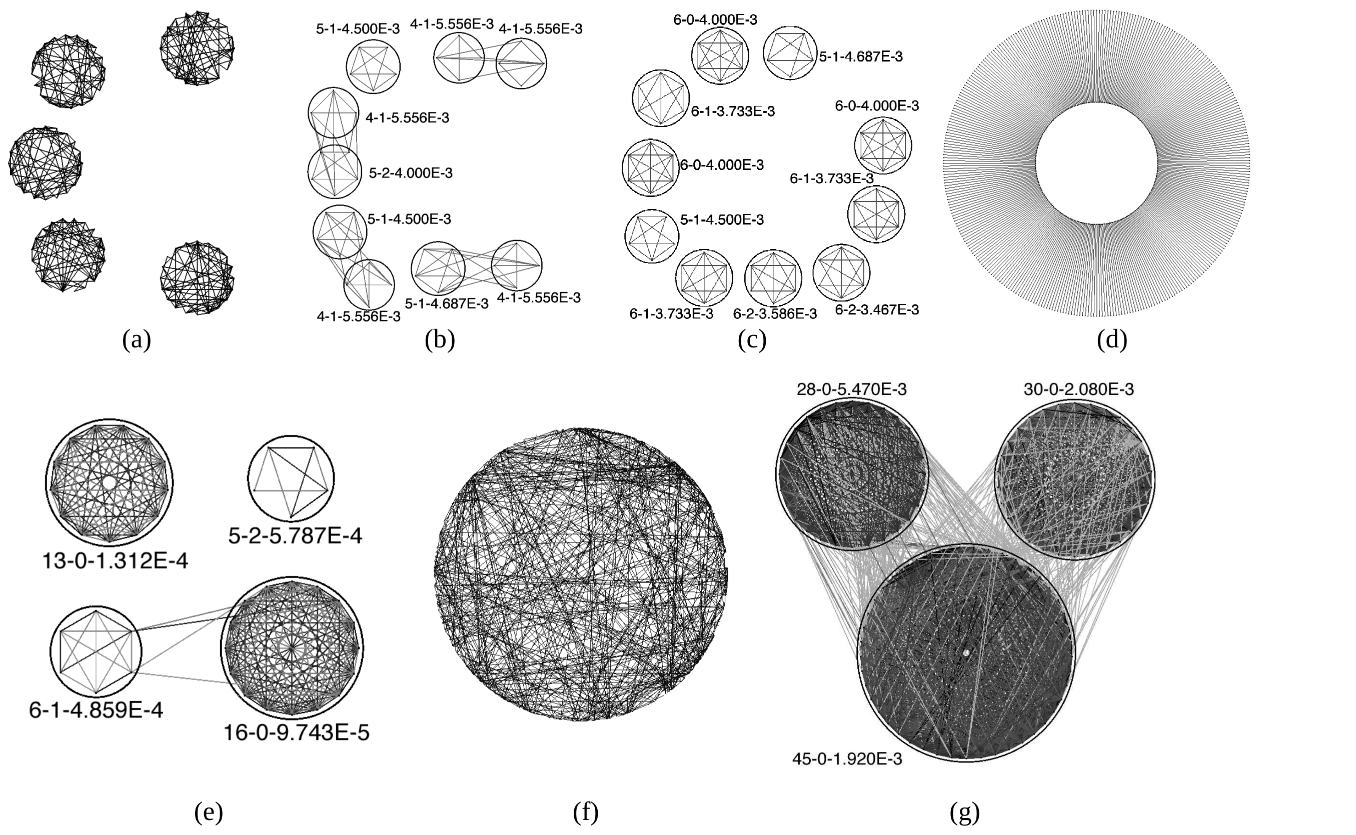

Figure 5. Tight edges and cluster graphs for different kinds of CSPs produce different secondary structures. Each cluster is drawn within a circle, and tighter edges are darker. The label gives the cluster's number of variables, the number of additional edges needed to make it a clique, and *Foretell*'s score for it. (a) Tight edges in a typical *Comp* problem and (b) its cluster graph. (c) The cluster graph for a 25-10-20 problem. (d) Edges in RLFAP scene 11 with tightness ≥ 0.3 and (e) the cluster graph for RLFAP. (f) The tightest edges in a driverlog problem and (g) its cluster graph.

identified clusters, and the program joined them with any constraints that connected their variables in the original CSP to form a cluster graph. *Foretell* is non-deterministic because shaking in VNS is by random selection. For that reason, every experiment using *Foretell* was repeated 10 times and averaged. Although the 10 cluster graphs for any given CSP were very similar to one another, the cluster graphs for different kinds of CSPs were quite different. Figure 5 includes examples of these structures, where the vertices for each cluster are displayed within a circle.

The first category of secondary structure is seen in 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 <*n,k,d,t*> be a class of CSPs each of which has *n* variables, maximum domain size *k*, density *d*, and tightness *t*. Then <*n,k,d,t*> *s* <*n′,k′,d′,t′*> *d″ t″* specifies a class of composed CSPs, each with a central component described by <*n,k,d,t*>, *s* satellites in <*n′,k′,d′,t′*>, and links with density *d″* and tightness *t″*. Figure 1 is a problem in *Comp*:

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

Comp contains both satisfiable and unsatisfiable instances.

Composed problems can be designed to mislead traditional CSP search heuristics that prefer the higher-degree variables in the readily-solved central component. Later, when conflicts arise within a satellite, search backtracks to the central component, although the true difficulties lie in the satellites. Composed problems' relatively dense, tight satellites are isolated from one another. Drawing the tightest edges would produce Figure 5(a); the cluster graph in Figure 5(b) is suggestive of the same structure. (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 5(c) shows a similar secondary structure for a problem from the class designated 25-10-20 by (Lecoutre, 2009), and here by

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

Its higher satellite density (0.786) encourages the formation of somewhat larger clusters, and typically leaves behind too few edges to form a second cluster in the same satellite. Thus these clusters are isolated from one another. Clusters are not always composed from the tightest edges in the graph, however. *RLFAP* here is scene 11 of the radio link frequency problems (Cabon et al., 1999). RLFAP's many constraints vary dramatically in their tightness. Figure 5(d), with the variables on two concentric circles, shows that the tightest constraints form a bipartite graph. The cluster graph is considerably more informative. *Foretell* finds the identical clusters of size 6 and 13 on every run. Figure 5(e) shows the four clusters with the highest priority for *focus* (including the two of sizes 6 and 13) from a typical run. Note that only two of the clusters are connected, and 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. Of particular interest is the fact that only 18 of the tight edges in Figure 5(d) appear in Figure 5(e) at all, and only two are in the top-priority cluster.

The driver problems (driverlogw-08cc-sat_ext and driverlogw-08c-sat_ext) have the same 408 variables and 9321 constraints with identical tightness; they differ only in the values permitted by the constraints. Given their identical graphs and tightness, one would expect *Foretell* to produce the same cluster graphs, and so it does. Indeed, on every run on both problems *Foretell* found only three large clusters, all of which were cliques. Figure 5(f) shows the tightest edges in that problem; Figure 5(g) shows the clusters *Foretell* finds. This structure is a path among three cliques, and behaves, as we shall see, quite differently from the less connected structures of the others.

Clusters and Inference

A cluster graph provides information that can be harnessed to guide inference. Inference methods can be characterized along a spectrum by the effort they exert. More inference does not always result in more domain reduction — it is often faster to risk and retract mistakes than to anticipate them. Inference after every assignment, as in Figure 4, is called *consistency maintenance.* The simplest method, *forward checking* (*FC*), removes from the dynamic domains of the neighbors of a just-bound variable any values inconsistent with its newly assigned value. The MAC-3 algorithm to maintain *arc consistency (AC)* does more work: it initially enqueues the edges to all the unvalued neighbors of the just-bound variable, and checks each element of the

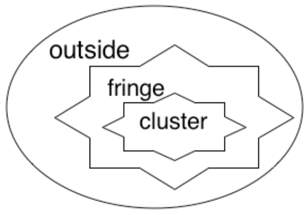

Figure 6. Propagation regions delineated with respect to a cluster.

queue for domain reduction (Sabin and Freuder, 1997)). Whenever a variable's domain is reduced, MAC-3 enqueues the constraints between that variable and its unvalued neighbors. *ACR-k* takes a stance between FC and MAC-3 (Epstein et al., 2005). It begins with the same initial queue as MAC-3, but subsequently enqueues only constraints on variables whose dynamic domains lose at least *k*% of their values. (The R is for "response.") Intuitively, higher values for *k* make ACR lazier.

Cluster-based inference considers where other variables lie with respect to the clusters. As in Figure 6, each cluster *C* in problem *P* delineates a *fringe* (variables in *P* – *C* within *width* edges of some variable in *C*), and an *outside* (*P* – *C* – *fringe(C)*), as shown in Figure 6. The question then becomes how to select propagation methods for the cluster, the fringe, and the outside.

To begin, we generated classes of small, not necessarily solvable CSPs similar to the clusters *Foretell* finds. The densest possible graph is a clique. Intuitively, a near clique is a subgraph that is a few edges short of a clique. A near clique is defined recursively. A clique on 3 vertices is a near clique and, given a near clique *NC* on *v* vertices with *m* missing edges, the addition of a new vertex to *NC* also forms a near clique if and only if Δm, the increase in the number of missing edges, conforms to:

$$\Delta m < \frac{v}{2} + \frac{m}{v-1}$$

For *n* > 3, this requires

$$m \le \left\lfloor \frac{v-1}{2} \right\rfloor$$

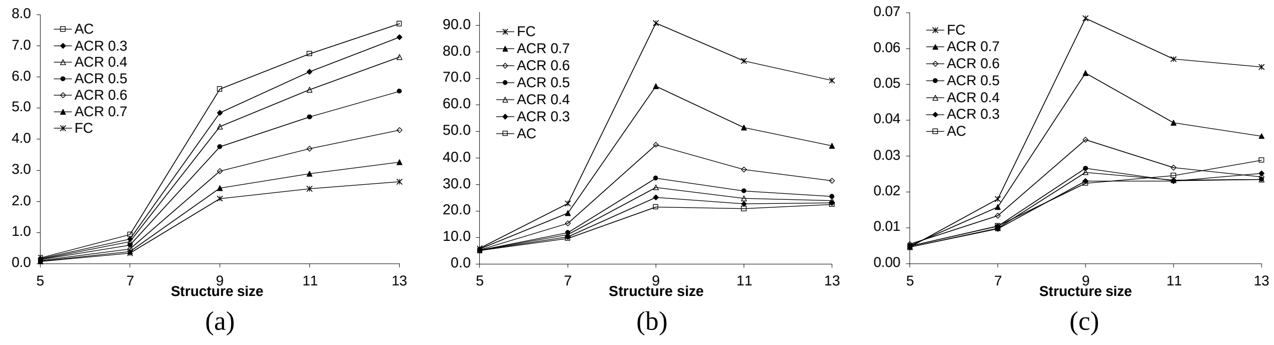

Figure 7. Number of (a) checks (b) expanded nodes and (c) CPU seconds required to solve cluster-like graphs of various sizes under 7 different propagation methods. Lazier propagation does fewer checks, expands more nodes, and is sometimes faster.

To simulate *Foretell*'s clusters, we generated classes of CSPs that were near cliques of sizes 5 to 13 with edge tightness 0.5, and solved them with *MinDomDeg* in separate runs that maintained consistency with FC, MAC-3, or a MAC-3-like version of ACR-*k* for *k* from 0.3 to 0.7. The results are shown in Figure 7. AC is warranted while clusters are of size no more than 7, but on larger simulated clusters it is statistically significantly slower than ACR-0.4. ACR-0.4 also showed low variation in performance on the simulated clusters.

The structure of a cluster graph suggests the design of cluster-based inference methods. Let *c/w/f/o* be a cluster-based method that propagates within a cluster with method *c*, within its fringe of width *w* with method *f*, and outside with method *o*. For *Comp*, the clusters in Figure 5(b) are small; that mandates AC propagation within them. Because *Comp* clusters are often linked, even a fringe of width 1 may reach other clusters. Thus AC is a wise approach within the fringe as well. Once outside the clusters, propagation can afford to be lazy. Thus a reasonable cluster-based propagation method for *Comp* is as AC/2/AC/FC. RLFAP's cluster graph is different; the clusters are larger, so that propagation within clusters of ACR 0.4 is reasonable. The relative isolation of the 4 crucial clusters there suggests propagation in the fringe with ACR-0.6, but FC is expected to be safe in the rest of the graph. This produces the method ACR-0.4/1/ACR-0.6/FC. Finally, the large clusters in the driver problems are so closely connected that it may only be reasonable to attempt ACR-0.4/1/ACR-0.5/FC.

Experimental Design and Results

These experiments were run with *ACE* (the Adaptive Constraint Engine), a test-bed for CSP solution (Epstein, Freuder and Wallace, 2005). Because ACE is a research tool that gathers extensive data, it is highly informative but not honed for speed. Performance is therefore reported here both as elapsed CPU time in seconds and as number of nodes in the search tree. To control for the vagaries of local search, performance for any experiment with *Foretell* was averaged across 10 trials for each problem. On each problem, *Foretell* was given some number of milliseconds per cluster, and identified as many clusters as it could until a call to VNS failed. All cited differences are statistically significant at the 95% confidence level on a one-tailed *t*-test. Table 1 lists the number, average size, and maximum size of the clusters detected with *Foretell* prior to search.

All but the last of the classes above the line in Table 1 are composed problems from (Lecoutre, 2009). A problem described there by *a-b-c* denotes a central component with *a* variables and *b* satellites of 8 variables each. All variables have domain size 10, constraint tightness 0.150 within the central component and tightness 0.050 on each link. Constraint density within a satellite is always 0.786. Clusters are often readily detected in non-composed CSPs as well. Table 1 includes RLFAP and the driver problems.

The difficulty of a class of problems is gauged here by the resources *MinDomWdeg* required to solve it. Both *MinDomWdeg* and cluster-guided search solved every problem within 30 minutes. For cluster-guided search, time includes the time used by *Foretell* to detect clusters. The results support the premise that a cluster graph addresses the hardest parts of a problem. Far fewer incorrect assignments were made under cluster-guided search.

In *Comp, Foretell* found at least one cluster in every satellite in every problem on every run. The cluster graph in Figure 1(d) is typical of the result. The structure of the composed problems in the other classes, however, is deliberately obscured. Nonetheless, *Foretell*'s output matches the descriptions provided for those problems.

Cluster-based inference was added to cluster-guided search and tested on all the problems in Table 1. On all the classes of composed problems there was little room for

Table 1: At the 95% confidence level, *focus* outperforms *MinDomWDeg* on these problem classes. Order of magnitude improvements over *MinDomWdeg* in **bold**. Classes above the line are composed, with central component density *d*, satellite tightness *t′*, and link density *d″*. Time is in CPU seconds. Data for *Foretell* includes number of clusters, average cluster size, and maximum cluster size, averaged across 10 runs. Data for *focus* is mean and standard deviation over 10 runs *focus.*

				MinDomWdeg		***Foretell's clusters***			***Focus Time***		***Focus Nodes***	
Problem	***d***	***t′***	***d″***	***Time***	***Nodes***	***Count***	***Size***	***Max***	μ	σ	μ	σ
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
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
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
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
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
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

improvement over cluster-guided search, and none appeared. On RLFAP, however, cluster-based inference further improved the performance of cluster-guided search, reducing it to 49.294 seconds, a statistically significant improvement. As Figure 7 anticipated, the laziness of ACR-*k* engendered more mistakes than AC, so there was no concomitant reduction in nodes. The success of cluster-based inference on RLFAP suggests that *Foretell*'s clusters cover enough of the backdoor so that FC suffices for the "outside." (This improvement is not attributable solely to FC; FC alone is dramatically slower on this problem.) On the driver problem, cluster-based inference did not improve cluster-guided search. We suspect that this is because the secondary structure is markedly different.

Discussion

Foretell's key parameter is how much time to devote to the detection of any single cluster. It has no prior knowledge about how many clusters lie within a problem, nor about how many might be necessary to solve it. We have found empirically that too few clusters provide inadequate guidance, but that too many clusters require *focus* to do too much computation to pick its next target.

Consider, for example, the experiments in Table 2, where *Foretell* was allocated some fixed amount of time to find each cluster in RLFAP. (At 200 ms. per cluster, clusters were rarely found at all; data omitted.) At 300 ms., the average size of the clusters was smaller than at the other times, which suggests that 300 ms. was not enough time to build clusters substantial enough to guide search.

As *Foretell*'s time allocation increased, the number of clusters it found increased. By 2000 ms. the same largest cluster was found consistently. There was little difference between the cluster graphs and none in the resultant search tree size observed in the experiments for 1000 and 2000 ms. Total time for search, however, increased because the increased allocation allowed *Foretell* more iterations through the loop in Figure 2, during which it tinkered more with whatever cluster it found, as indicated by the second column in Table 2. Observe that, if *Foretell* dawdles during local search (line 6 in Figure 2), it is possible to exceed the allocated time on average. An allocation greater than time per cluster indicates that *Foretell* has done all it can.

One way to think about Table 2 is that as complete a cluster graph as possible may provide an important explanation for the user. In that case, one should iteratively increase the time allocation until the time per cluster is smaller than the allocation and a consistent number of clusters is found. Another approach to Table 2 is computational, informed by two surprising observations. First, on all 10 runs for RLFAP with 400 ms., cluster-guided search averaged only 2 errors (retracted 2 assignments) in its first 300 assignments, and none at all in the last 350 (i.e., outside the cluster graph). Second, although *Foretell* sometimes delivered 10 different cluster graphs from 10 runs under the same time allocation, *focus* used them the same way, that is, the tightest, largest clusters dominated and the standard deviation in the search tree size was 0. Thus the difference between the 400 ms. experiment and the 2000 ms. experiment was that a few variables were treated differently. The 2000 ms. experiment errs somewhat more and earlier (8 retractions about 60 deep in the tree). 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 work therefore addresses additional termination conditions for *Foretell* (line 4 in Figure 2). These include an overall VNS time limit, a Luby-like adaptive cutoff for allocations on successive clusters (Luby, Sinclair and Zuckerman, 1993) , and a limit on the percentage of variables that may be included in either an individual cluster or the entire cluster graph.

A cluster graph provides an explanation of where the difficulties lie in a CSP. Figure 5(b) focuses attention on the satellites, but the solution with *focus* is even more descriptive: it searches only within three of those clusters and proves that there is no solution with only 12 variables (out of 200). This provides a more satisfying explanation than either a search tree rooted at a single node or a collection of edge weights.

RLFAP and the driverlog problems demonstrate that a problem need not have satellites to have clusters. On small-world problems, for example, almost every variable is

Table 2: Average results of 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.

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

quickly shown to lie in some cluster. Having clusters, however, does not justify directing computational resources to *Foretell*. On easy problems, it is faster to use *MinDomWDeg* or even *MinDomDeg*. Clusters are not detected dynamically, during search, because *Foretell* does not find clusters in order of either tightness or size. To identify a good starting point, *focus* must therefore choose among a set of clusters. This static but predictive perspective serves search well.

Cluster-based propagation is still in an early design phase. Because cluster sizes vary in real-world problems, ACR seems a wise choice unless the problem has uniformly small clusters. Cluster-based propagation should be further tailored to the metastructure of the cluster graph, including cluster size, domain size, variance in internal edge tightness, and the number and tightness of inter-cluster edges.

We see no impediment to adapting this approach for non-binary constraints. Real-valued domains present a different challenge, one we believe to be surmountable through the methods planned for large domains. There are problems in which *Foretell* cannot find any clusters at all. Other structures, such as lengthy cycles, can create search difficulty without local density (Markstrom, 2006). Something similar may be operative in these problems.

No solver, human or machine, has an efficient way to "see" Figure 1(c) perfectly without knowledge about the problem generator. A cluster graph is a prediction of significant metastructure, an abstraction of a CSP that focuses on the hard subproblems, the ones where global search is likely to fail. Those clusters can be used not only to focus attention during search but also to inform propagation and to provide insight into the nature of the problem in a user-friendly representation.

Acknowledgements

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

References

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.

Dechter, R. and J. Pearl 1987. The cycle-cutset method for improving search performance. In *Proceedings of Third Conference on Artificial Intelligence Applications*, 224-230.

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

Dilkina, B., C. P. Gomes and A. Sabharwal 2007. Tradeoffs in the Complexity of Backdoor Detection. In *Proceedings of CP-2007*, 256-270. Providence RI, Springer.

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., E. C. Freuder, R. M. Wallace and X. Li 2005. Learning Propagation Policies. In *Proceedings of Second International Workshop on Constraint Propagation and Implementation*, 1-15. Sitges, Spain.

Epstein, S. L. and X. Li 2009. Cluster-based Modeling for Constraint Satisfaction Problems. In *Proceedings of IJCAI Workshop on Learning Structural Knowledge from Observations (StrucK-09)*, Pasadena, CA, AAAI Press.

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.

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

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

Luby, M., A. Sinclair and D. Zuckerman 1993. Optimal speedup of Las Vegas algorithms. *Information Processing Letters* 47: 173–180.

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.

Markstrom, K. 2006. Locality and Hard SAT-Instances. *JSAT* 2: 221-227.

Ruan, Y., E. Horvitz and H. Kautz 2004. The Backdoor Key: A Path to Understanding Problem Hardness. In *Proceedings of AAAI-2004*, 124-130. San Jose, CA, AAAI Press.

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

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. 5(2): 115-135.

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

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