This paper appeared in the *Proceedings of ICTAI 2006.*

Finding Crucial Subproblems to Focus Global Search

Susan L. Epstein
Hunter College and The Graduate School of The City University of New York
susan.epstein@hunter.cuny.edu

Richard J. Wallace
Cork Constraint Computation Centre
University College Cork
r.wallace@4c.ucc.ie

Abstract

Traditional global search heuristics to solve constraint satisfaction problems focus on properties of an individual variable that mandate early search attention. If, however, one could predict crucial subproblems *(the portions of a constraint satisfaction problem likely to cause each other particular difficulty) in advance, search could address them first. This paper postulates several types of crucial subproblems, and shows how local search can be harnessed to identify them before global search for a solution. A variety of heuristics and metrics are then used to guide traditional constraint heuristics with those crucial subproblems. On certain classes of structured problems, such search outperforms traditional heuristics by at least an order of magnitude in both time and space.*

1. Introduction

Constraint satisfaction problems (*CSPs*) often have a *backdoor*, a crucial, relatively small subset of variables whose assignment makes the remainder of search nearly trivial [1]. Our premise here is that the backdoor may lie within a *cluster,* a dense, highly-constrained subproblem destined to impede global search. The principal contributions of this paper are two new, efficient local search algorithms to identify such CSP subproblems prior to search, and their heuristic integration with traditional methods to direct global search to a solution. Our approach harnesses the power of propagation within the density of the subproblem, and directs it to the variables upon which constraints press hardest. On certain classes of structured problems, our approach outperforms traditional variable-ordering heuristics by at least an order of magnitude, in both time and space. Even on structured problems, whose uniform edge tightness makes subproblem detection more difficult, we show that crucial subproblems can still reduce solution time by about half.

We investigate four methods approaches to detect crucial subproblems in advance, and then use them to guide search:

• The *clique hypothesis:* Density (the percentage of possible constraints present in the subproblem) alone is the key. Since a *clique* (a complete subgraph) has maximal density, search should exploit clique membership. Thus search should assign values to variables in maximum (largest present) cliques first [2].
• The *tension hypothesis:* Tightness (the degree to which a constraint excludes possible value combinations) alone is the key. The *tension* on a vertex is the average tightness of the edges on which it is incident. Search should assign values to the maximally constrained variables (those incident on the tightest edges) first.
• The *near clique hypothesis:* Problems are unlikely to have very large cliques, because too many constraints with some reasonable tightness are likely to admit no solutions at all. A *near clique* is a subgraph that would be a cliques but for a few missing edges. Search should assign values to variables in maximum near cliques first.
• The *cluster hypothesis:* Together, density and tension predict difficult subproblems. Search should therefore assign values to variables in maximum clusters first.

Precise identification of these subproblems (maximum cliques, near cliques and clusters) is NP-hard. Instead, we estimate them with *variable neighborhood search*, a local search metaheuristic [3]. After detection, we use one or more of these multiple disjoint subproblems in a single problem to guide global search. We show that both clusters and near cliques are considerably give more incisive information, resulting in more and efficient search than do cliques on several classes of problems.

Although arc consistency is more efficiently enforced by solving global subproblems on the fly [4], the identification of such subproblems has thus far been the province of the human modeler. The exploitation of some semantically-based, dense subgraphs has certainly been successful. For example, all-different cliques have their own propagation schemes [5], but they require identification by the modeler. There has also been some domain-specific success for relatively sparse coloring problems [6, 7], and with subsets of natural language constraints (*circles*) that can be solved exhaustively and then combined with branch and bound to optimize sentence translation [8]. Circles, however, have few interactions among them and are predicated upon relatively low density and the prevalence of articulation points in the graph underlying natural language constraints on a sentence. Clusters, in contrast, are purely syntactic, at least from the view of the solver (and perhaps of the modeler as well).

Our primary focus here is on composed problems, which simulate the uneven density and tightness often prevalent in real-world problems such as frequency assignment [9]. On composed problems, the combination of density and constrainedness embodied by a cluster, proves more powerful than either alone. Clusters go undetected by traditional heuristics until they impact search. Although an identified cluster takes priority during search, our approach uses a traditional heuristic to make decisions within a cluster, and to guide search once all cluster variables have been assigned values. Clusters harness more information than cliques, because they consider tightness as well as structure.

The next section provides some fundamental definitions. To identify crucial subproblems prior to global search, we have adapted variable neighborhood search [3], a metaheuristic that systematically changes local search neighborhoods. Section 3 describes the basic variable neighborhood search approach, along with the metrics and heuristics we have devised to apply it and capitalize on crucial subproblems. Subsequent sections describe the experimental design and discuss empirical results and future work.

2. Background

A constraint satisfaction problem (*CSP*) consists of a set of variables *X*, a set *D* of *domains* for those variables (values which may be assigned to them), and a set *C* of *constraints* on *X*, restrictions on how values may be assigned to subsets of *X.* This discussion is restricted to *binary* CSPs, where each constraint is on a pair of variables. The *constraint graph* of a CSP represents each variable as a vertex and each constraint as an edge between the two vertices whose variables it restricts. The *degree* of a variable is the number of edges on which it is incident in the constraint graph. The *density* of a CSP is the fraction of possible edges in its constraint graph. The *tightness* of a constraint is the fraction of possible value pairs it excludes. A CSP with *fixed tightness* has the same tightness on each of its edges; otherwise it is said to have *varying tightness*.

An *instantiation* of a CSP = <*X, D, C*> assigns values from their respective domains to some subset of *X.* A *consistent instantiation* abides by all the constraints on its instantiated variables. A *full instantiation* assigns a value to every variable, otherwise an instantiation is said to be *partial*. A *solution* to a CSP is a full, consistent instantiation. *Global search* on a CSP assigns a value to one variable at a time, and backs up when a partial instantiation is inconsistent, until it reaches a solution. During global search, a *future variable* is one that is not yet assigned a value. As global search progresses, *constraint propagation* eliminates, from the domains of future variables, values that would conflict with the current partial instantiation. Thus each variable has an original (*static*) domain size and a *dynamic* domain size with respect to the current partial instantiation. Similarly, the *dynamic* constraint graph consists only of the vertices that represent future variables and edges between them. Each variable has a (*static*) degree in the original constraint graph as well as a *dynamic degree* in the dynamic constraint graph.

Although in principle any ordering of the variables for instantiation can lead to a solution, a good ordering speeds search. Many heuristics for variable ordering during global search are based on the premise that density in the constraint graph is important, for example, "maximize the degree." Other heuristics insist that constrainedness is important, such as "minimize the domain size." Particularly successful variable-ordering heuristics combine density and constrainedness, for example, *dom/deg* (minimize the ratio of dynamic domain size to static degree) and *dom/ddeg* (minimize the ratio of dynamic domain size to dynamic degree).

```
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 1. A high-level description of VNS metaheuristic search through *k* neighborhoods. The initial solution, shake algorithm, and local search vary with the application.

3. Local search for crucial subproblems

Although variable neighborhood search (*VNS*) is a metaheuristic that has been successfully applied to a wide range of combinatorial and optimization problems [3], our new work appears to be its most general CSP application. Our description here focuses upon the aspects of VNS that support our particular goals. The key processes are the generation of an initial solution, a local search algorithm, and *shaking* to search within *k* pre-specified neighborhoods for a solution.

Figure 1 provides pseudocode for VNS. (The "variable" in VNS refers to changing neighborhoods, not to variables in the CSP sense.) A neighborhood intentionally delimits the current search space; as VNS iterates, each new neighborhood provides a larger search space. VNS begins with an initial solution (line 1) and terminates on a user-specified stopping condition (e.g., elapsed time) or after shaking in *k* increasingly larger neighborhoods without an improvement (lines 1 and 4). After *best-yet* is initialized, VNS performs local search in its neighborhoods (line 6). Local optima are compared by a metric, *score*. A better local optimum resets *best-yet* and begins again with the first neighborhood (lines 8 and 9); otherwise it proceeds to the next neighborhood (line 10). Shaking (line 5, described further below) randomizes *best-yet* to avoid looping through the same portions of the search space. Shaking shifts search within the current neighborhood. As *index* increases, these neighborhoods become larger and the shaken version of *best-yet* becomes less similar to *best-yet* itself. The VNS algorithm to find a maximum clique in a graph clarifies the details, and provides a foundation for our own new algorithms.

3.1 Finding a maximum clique with VNS

Computations for the maximum clique in a graph $G = <V, E>$ on vertices V with edges E are made with respect to its *complement* $G' = <V, V\times V - E>$. At all times, a 3-part partition of the vertices is maintained: those in the solution, those forbidden to join the solution (the *transversal*), and the remainder (*remaining*).

The VNS algorithm to identify a maximum clique finds an initial subgraph for *best-yet* in line 1 of Figure 1. with the simplicial vertex test (*SVT*) shown in Figure 3 [10]. A vertex is said to be *simplicial* if and only if its neighbors form a clique. The *size* of a simplicial vertex is the size of the clique it forms with its neighbors. Because a vertex of degree zero in G' is clearly a member of any maximum clique in G, SVT begins with all simplicial vertices of size 0 in G' (e.g., vertex 1 in Figure 2a). SVT then adds simplicial vertices from *remaining* in ascending size order (in G′) with *select*, one at a time. Ties are broken by a greedy choice: minimize the degree of the vertex in G' (i.e., maximize the degree in G); subsequent ties are broken at random. After each selection E' is updated. SVT returns a starting point for local search, an initial (possibly empty) solution (e.g., Figure 2b).

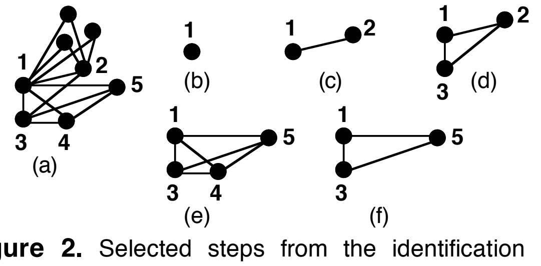

Figure 2. Selected steps from the identification of a maximum clique with VNS from the graph (a). (b) SVT: Begin with a simplicial vertex of size 0. (c)-(d) VND: Greedily add, one at a time, vertices adjacent to every selected vertex. (e) Interchange a pair of adjacent vertices for a selected one. (f) Shake out a randomly-selected vertex. This process repeats; see the text for further details.

```
solution ← all vertices of degree zero in G′
while vertices remain in transversal and simplicial
  vertices appear in remaining
  v ← select(remaining)
  solution←solution ∪ {v}
  transversal← transversal ∪ neighbors(v)
  remaining ←remaining – v – neighbors(v)
  E′ ← E′ – all edges to v or to neighbors(v)
return solution
```

Figure 3. SVT, the initialization procedure in VNS search for a maximum clique.

Once SVT terminates, the VNS algorithm to identify a maximum clique uses local search to expand it. When local search can proceed no farther, the algorithm shakes the subgraph by removing some number of randomly-selected vertices from it and begins local search again. Examples of the algorithm's behavior appear in Figure 2.

The local search algorithm to identify a maximum clique M is Variable Neighborhood Descent (*VND*), summarized in Figure 4 [10]. One vertex at a time, it greedily extends *best-yet* with a vertex adjacent to every vertex already in M (e.g., vertices 2 and then 3 in Figure 2), using the same tie-breaking rules as SVT. Next, it may be possible to enlarge M by swapping some vertex w in M for two adjacent vertices that are not neighbors of w, are not presently in M, and are neighbors of all vertices in $M - w$ (e.g., vertices 4 and 5 for 2 in Figure 2). VND repeatedly performs such *interchanges* in an attempt to identify a larger clique in the same neighborhood. When interchange can proceed no further, shaking in VNS removes some number of randomly-selected vertices from M (e.g., for *index* = 1, vertex 4 in Figure 2) and thereby shifts search within the current neighborhood. For both VNS and VND the maximum neighborhood index k is set to the minimum of 10 and $|M|$ dynamically. The important differences between VND and VNS are their termination conditions and greedy extension rather than shaking.

We extended VNS to seek disjoint cliques in G as follows. When VNS returns M, we remove M from G and repeat the entire process with VNS to identify the next maximum clique in $G - M$.

```
best-yet ←greedy-extension (best-yet)
index ← 1
neighborhood ← neighborhood(index)
  until index = k
    local-optimum ← interchange(best-yet, neighborhood)
    If score(local-optimum) > score( best-yet)
      then  best-yet ← local-optimum
            index ← 1
      else  index ← index + 1
    neighborhood ← neighborhood(index)
```

Figure 4. A high-level description of VND, the local search algorithm called by VNS in Figure 1.

3.2 Finding near cliques with VNS

To find near cliques with VNS, our implementation uses SVT to initialize a near clique *N* as if search were for a clique, and shakes the current near clique as cliques are shaken. There are, however, two important changes: one to *select* in SVT and the other to *interchange* in VND. When adding a vertex to *N* in the greedy extension step, if no vertex in *remaining* is a neighbor of every vertex in *N*, *select* takes a vertex whose inclusion in *N* will introduce the fewest missing edges (e.g., one missing edge for vertex 4 in Figure 5). During search for a near clique on *v* vertices that is missing *m* edges, we prefer large, dense subgraphs. This is implemented with the scoring function:

$$score = v \cdot \left(1 - \frac{m}{C_{v,2}}\right) \qquad [1]$$

where $C_{v,2}$ is the number of edges to form a clique on *v* vertices. Thus, a new vertex that is missing Δm edges should be added only if

$$(v+1)\left(1 - \frac{m + \Delta m}{C_{v+1,2}}\right) > v\left(1 - \frac{m}{C_{v,2}}\right) \qquad [2]$$

Condition [2] can be shown to reduce to

$$\Delta m < \frac{v}{2} + \frac{m}{v-1} \qquad [3]$$

Thus, to maintain *N*'s near clique density, a selected vertex can be missing no more than the number of edges indicated in condition [3].

Our VNS algorithm to find a near clique permits the same interchange used for cliques: the replacement of *w* in *N* by a pair of adjacent vertices that are neighbors to all of $N - w$. If no such *w* exists, subject to the restriction in condition [3], the near clique algorithm will accept as a replacement for *w* in *N,* a pair of non-adjacent vertices that are neighbors to all of $N - w$ (e.g., 5 and 6 for 2 in Figure 5). This introduces one missing edge. Although other possibilities (e.g., a 3 for 2 interchange) might respect condition [3], in the interest of speed we restricted interchange to these two possibilities. Our implementation iterates to identify disjoint near cliques until no near clique of size at least 3 can be found.

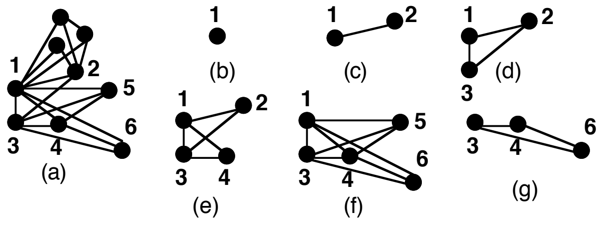

Figure 5. Selected steps from the identification of a near clique with VNS from graph (a). (b) SVT: Begin with a simplicial vertex of size 0. (c) - (d) VND: Greedily add vertices adjacent to every selected vertex. (e) Add a vertex that introduces one missing edge. (f) Interchange a pair of non-adjacent vertices for a selected one. (g) Shake out two randomly-selected vertices. This process also repeats.

3.3 Clusters

Clusters are near cliques whose vertices have above average tension on them. Because they are computed prior to any search, clusters do not appear in graphs with uniform edge tightness. Our VNS algorithm to identify clusters is based upon our algorithm for near cliques. Before any search, our algorithm calculates the tightness of each edge in the graph and initializes the (static) tension of a vertex to the average tightness on its edges. As propagation reduces the domains, the number of pairs of possible values excluded by each edge changes. The *dynamic tension* of each future variable *v* considers only the edges still present in the graph and is estimated by

$$tension(v) = 1 - \frac{\prod_{w \in neighbors(v)} |\tilde{D}_w|}{\prod_{w \in neighbors(v)} |D_w|} \qquad [4]$$

where $\tilde{D}_w$ is the dynamic domain size of the neighbor *w* of *v*, and D_w. is its original domain size.

Constrainedness impacts local search here in three ways. First, SVT is modified to prefer simplicial vertices with above average tension and to exclude vertices of degree 1. Second, selection and extension in VND are greedy first with respect to tension, and only then with respect to minimum degree in G'. (We also tested greediness with respect first to degree and then to tension, and found that it performs less well. Data omitted here.) Finally, the scoring metric for "better" clusters in VND prefers large, dense, highly-constrained subgraphs, that is,

$score(S) = |S| \cdot \text{density}(S) \cdot \text{tension}(S)$.

4. Testing the hypotheses

All experiments were conducted with *ACE* (the Adaptive Constraint Engine) on solvable problems [11]. Each hypothesis was tested as a variable-ordering heuristic that preferred its designated vertices and broke ties with *dom/deg*. For example, the cluster hypothesis was tested by preferring future variables in an identified cluster, and, if there was more than one such variable, those with the lowest *dom/deg* value. (We chose to break ties with *dom/deg* rather than *dom/ddeg* in the interest of speed; subsequent testing suggested no advantage to *dom/ddeg* as a tie breaker.) Any further ties were resolved by lexical ordering. Values were assigned in lexical order and arc consistency was maintained with the *MAC*-3 algorithm with *d*-way branching [12]. (2-way branching performed the no differently.) For both classes of random problems, each heuristic was allocated 20000 steps (variable selections and value assignments) to solve a problem. The benchmark in every class is variable ordering with *dom/ddeg* and lexical value assignment. The termination condition for all subproblem calls (to find a clique, a near clique, or a cluster) was a fixed time limit of 0.1 seconds. For repeated calls on a single problem (e.g., to find a sequence of *c* cliques) this could incur a cost of up to $c \cdot 0.1$ seconds.

4.1 The problem classes

A *composed problem* consists of a subgraph called its *central component* joined to one or more subgraphs called *satellites*. (These problems were inspired in part by the hard manufactured problems of [13].) A class of composed graphs stipulates the features of its central component and its satellites separately, along with the density and the tightness of the edges (*links*) between them. In all our problem classes, links were generated under the same conditions, with density 0.115 and tightness 0.05. (Satellites are not connected to one another.) The class notation <*n, k , d, t*> indicates *n* variables, maximum domain size *k*, density *d*, and tightness *t.* Thus the problem class <22, 6, 0.6, 0.1> 1 <8, 6, 0.72, 0.45> has 30 variables, each with domain size 6, separated into a central component of 22 variables with loose constraints and one satellite of 8 variables that is somewhat more dense and significantly tighter. We use randomly-generated solvable problems in a range of these classes to explore the vulnerabilities of various heuristics:

• Class A: <22, 6, 0.6, 0.1> 1 <8, 6, 0.72, 0.45> Variable-ordering heuristics that prefer large degrees should be misled here, because variable degree in the central component will average about twice that in the satellite. As a result, search will remain in the central component for a considerable time. Moreover, given the low tightness on the links and on the edges within the central component, search will rarely backtrack until it begins to select variables from the satellite, quite deep in the tree. Variable-ordering heuristics that prefer small domains will also flounder on these problems, because low tightness in the central component and along the links to the satellite will rarely eliminate any domain values. Only satellite domains are likely to reduce.

• Class B: <22, 6, 0.6, 0.1> 2 <8, 6, 0.72, 0.45> problems are similar to those in class A, but have 2 satellites, so that even if the first satellite is solved, traditional heuristics could still be deflected into the central component. They also have 38 rather than 30, variables, with satellite vertices less in the minority.

• Class C: <22, 6, 0.5, 0.05> 1 <8, 6, 0.8, 0.5> problems further exaggerate the difference in tightness between the central component and its satellite.

• Class D: <22, 6, 0.6, 0.2> 1 <8, 6, 0.72, 0.4> problems have a central component and satellite with the same density as those in class A, but more similar tightness.

• Class E: <22, 6, 0.58, 0.23> 1 <8, 6, 0.64, 0.23> problems have a central component and a satellite with the same tightness, but the satellite is slightly more dense, which should subject its variables to greater tension.

• Class F: <20, 7, 0.58, 0.28> 1 <10, 7, 0.58, 0.05> problems have a central component and a satellite with the same density, but a loosely constrained satellite. Search should address the central component first.

• Class G: <15, 7, 0.6, 0.36> 1 <15, 7, 0.6, 0.05> problems have a central component and a satellite of equal size and the same density, but the central component is tighter, so search should address it first.

• Class H: <10, 7, 0.58, 0.05> 1 <20, 7, 0.58, 0.28> problems have a small central component and a larger, equally dense, tighter satellite. Search should address the satellite first. In principle there is no distinction between classes F and H, but lexical order would otherwise prefer the central component, which is larger and tighter in F.

We emphasize that our solver had no access to any information that would indicate that a particular problem was composed. To the solver, these problems all "look" as if they are simply ordinary (i.e., non-composed) CSPs; only our own inspection has access to the component-satellite partitioning.

4.2 Detecting crucial subproblems

We identified crucial subproblems in 100 random problems from each of the composed problem classes, using the original VNS clique-finding algorithm and our adaptations, described in Section 3. Results appear in Table 1. The clique finder is known to outperform many other algorithms intended for the same task, and to perform on a par with the best known local search algorithm as long as the density of the graph is at least 70% [10]. Although our

Table 1. Detected crucial subproblems, averaged across 100 composed problems. For cliques and near cliques, this table gives the size of the maximum subproblem identified in the graph and the median number of disjoint subproblems. Only a single cluster was calculated; median cluster size and its location in the composed problem are provided.

	Cliques		**Near cliques**		**Clusters**	
Class	Max	No.	Max	No.	SIze	Location
A	6.52	5	9.62	4	7	Satellite
B	6.48	6	9.61	4	7	Satellite
C	6.07	5	8.83	4	8	Satellite
D	6.99	5	10.10	4	7	Satellite
E	7.28	5	10.60	3	10	Central
F	6.36	6	9.49	3	9	Central
G	6.19	5.5	9.48	3	9	Central
H	6.33	6	9.56	4	9	Satellite

graphs are of lower density, the algorithm still performed well. Overall, our implementation found fewer but larger near cliques than cliques in problems from a given class. Both cliques and near cliques were found either within the central component or within a single satellite; none ever straddled a central component and a satellite at once.

Clusters were larger than cliques in the same graphs, With one exception, a cluster was found either in the central component or in a satellite, but did not straddle them both. We inspected the identified clusters carefully. In classes A, D, H and for the most part B, they were always restricted to the satellite. (One class-B problem had a cluster in its first satellite that included a single vertex from its central component; another class-B problem had a cluster of size 2. Both problems were solved without difficulty.) Class-C problems had clusters restricted to the satellite and often included all of it. In classes F and G the cluster was always restricted to the central component, as expected. Only class E offered a surprise; although we had expected the slightly higher density of the satellite to force the cluster into the satellite, it never did.

Table 2. Performance results averaged across 100 solvable composed problems, with a limit of 20000 solution steps. Steps and checks have been rounded to the nearest integer. Noteworthy results are in bold.

Class	Method	Solved	Time		Checks
A	*dom/ddeg*	85%	6.59	3609	229621
	Tension	92%	4.22	1955	136623
	Cliques	98%	0.97	473	51350
	Near cliques	98%	1.42	626	52440
	Clusters	**100%**	**0.21**	**62**	**4660**
B	*dom/ddeg*	99%	0.82	395	22553
	Tension	100%	0.51	228	17587
	Cliques	100%	0.34	105	10556
	Near cliques	97%	1.83	828	70398
	Clusters	**100%**	**0.33**	**93**	**9984**
C	*dom/ddeg*	90%	5.25	2408	277444
	Tension	90%	4.40	2257	225325
	Cliques	97%	1.60	730	84873
	Near cliques	98%	1.25	461	46082
	Clusters	**100%**	**0.16**	**62**	**3854**
D	*dom/ddeg*	100%	0.47	127	17253
	Tension	100%	0.29	137	18561
	Cliques	100%	0.28	105	16461
	Near cliques	100%	0.58	113	17812
	Clusters	100%	0.31	121	21287
E	*dom/ddeg*	100%	0.35	119	32870
	Tension	100%	0.35	116	33153
	Cliques	100%	0.38	118	33858
	Near cliques	100%	0.67	117	32999
	Clusters	100%	0.55	121	35936
F	*dom/ddeg*	100%	0.31	111	32141
	Tension	100%	0.33	109	32017
	Cliques	100%	0.37	111	33462
	Near cliques	100%	0.72	109	32547
	Clusters	100%	0.45	105	30633
G	*dom/ddeg*	99%	1.75	360	142370
	Tension	99%	0.93	293	111311
	Cliques	92%	9.49	1773	702746
	Near cliques	100%	0.65	111	33496
	Clusters	**100%**	**0.38**	**70**	**9766**
H	*dom/ddeg*	100%	0.36	104	29735
	Tension	100%	0.29	103	29663
	Cliques	100%	0.34	104	30122
	Near cliques	100%	0.70	103	29720
	Clusters	100%	0.47	103	29465

4.3 Using crucial subproblems in global search

The results of our hypothesis testing appear in Table 2. We tested the tension hypothesis with a heuristic that preferred vertices of maximum tension, as estimated by the expression in [4]. On the composed problems, tension always performed at least as well as *dom/ddeg*; it solved more problems or solved as many faster. Nonetheless, tension failed to solve problems in classes A, C and G; on class B it was slower than some other successful methods. On classes D, E, F and H, however, this simple approach was among the best methods tested.

We tested the clique hypothesis with a heuristic that preferred variables in the largest disjoint cliques of size at least 3. For example, if a problem had identified disjoint cliques of sizes 6, 5, 5, 4, 3, and 3 (with 4 vertices in no identified clique at all), then the prioritized subsets of variables would be of size 6, 10, 4, and 6. Merely selecting variables in large cliques did not prove effective, however. (Data omitted.) Our most successful approach sorted cliques in descending order by their estimated tightness on their remaining *active variables* (future variables with a dynamic domain that included more than one value). Our theory was that the assignment of a value to an active variable would have considerable influence on the other active variables in its clique. Because repeated calculation of the tuples supported within a subgraph S is too costly, we estimate tightness as:

$$tightness(S) = 1 - \frac{\prod_{v \in S} |\tilde{D}_v|}{\prod_{v \in S} |D_v|} \qquad [5]$$

where $\tilde{D}_v$ is the dynamic domain size of variable v in S, and D_v is its original domain size. On some composed problem classes (A, B, C, D and H), cliques solved at least as many problems as *dom/ddeg*, and solved them faster. Nonetheless, on classes A, C and G cliques failed to find a solution within the step limit. Without knowledge about tension, the clique hypothesis appears vulnerable. Indeed, it failed to solve 7 problems that *dom/ddeg* could solve in class G, and clearly floundered about there on many others. This is likely because the satellite and the central component in G have virtually identical structure, and cliques do not consider tightness.

We tested the near clique hypothesis with a heuristic that preferred variables in the largest near cliques of size at least 3. Merely selecting variables in large near cliques did not prove effective, however. (Data omitted.) Our most successful approach sorted near cliques in descending order by tightness on their future variables, using the estimate in [5]. The performance of near cliques on composed problems was uneven. On the relatively easy composed classes (D, E, F, and H), near cliques were about twice as slow as cliques, despite virtually no difference in number of steps or checks. Near cliques solved fewer class B problems than cliques did, and solved the same number of class A problems more slowly. Near cliques solved one more class C problem than cliques did. On composed class G, near cliques solved all the problems but clusters did better.

The performance of clusters on the composed problems was solid; it was the only method that solved every problem in every class within the step limit. We tested the cluster hypothesis with a heuristic that preferred variables in the single largest cluster of size at least 2. In classes E, F and H, clusters were slower than *dom/ddeg*. The slightly tighter satellites in class E did not, it turned out, warrant more attention, so the computation of a cluster was unnecessary for efficient search. In class F, both *dom/ddeg* and clusters focused search in the central component, so again the time devoted to VNS was unnecessary. Similarly, in class H, both approaches focus on the satellite.

5. Discussion

We do not claim that a cluster is a backdoor, but on our composed problems it functions as if it contains one. On composed problems *dom/ddeg* becomes mired in areas of the graph that will not prove fruitful, whereas clusters succeed because they identify a highly-constrained, dense subgraph in advance.

The implementations for cliques, near cliques, and clusters find crucial subproblems quickly and accurately. There was no appreciable difference is search decisions when each subproblem call was allocated 10 seconds instead of the 0.1 second used for the data presented here. When cliques and near cliques can both solve all the problems in a composed problem class, near cliques are always slower. This is because VNS extends its search as it discovers larger objects, and near cliques are about 50% larger than cliques in these problems. Judged against an exponential algorithm on hundreds of both random and composed CSPs, our VNS implementation found a maximum clique every time. On about 3% of the problems, however, that clique was detected second, and was one larger than the first.

On a larger set of Class-A problems, Table 3 compares the performance of our methods to other popular variable-ordering heuristics: minimize the dynamic domain size (*min ddom*), maximize the original degree (*max deg*), *dom/deg*, *dom/ddeg*, maximize the weighted degree (*wdeg*), and minimize the ratio of dynamic domain to weighted degree (*dom/wdeg*). (The weighted degree of a variable is the sum of the dynamic edge weights, as computed in [14], on which it is incident.) The edge weight heuristics perform well, but they still took more time, more steps, and more constraint checks because they learn from failure. In a composed class where the size difference between the central component and its satellite(s) was further exaggerated, edge weights would take longer to learn and the edge weight heuristics would be slower still. The work in [7] uses extensive local search in sparse coloring problems to establish edge weights with which to identify hard subproblems prior to global search, but this too is a form of learning. In contrast, clusters do not have to err and retract values in order to succeed. Indeed, they solved 136 of the 400 Class-A problems with no retractions at all.

Table 3. Average performance of alternative heuristics on a larger set (400) of class-A problems.

Method	Solved	Time	Steps	Checks
min ddom	95.50%	1.12	566.70	40108.78
max deg	83.75%	3.73	1928.88	87962.25
dom/deg	82.25%	4.00	1999.98	118754.01
dom/ddeg	91.25%	2.17	1069.06	77710.32
wdeg	100.00%	0.31	79.97	6159.94
dom/ wdeg	100.00%	0.29	84.04	6804.63
Tension	92.25%	7.41	1885.09	139161.67
Cliques	97.75%	2.27	636.10	61612.08
Near cliques	97.75%	2.64	584.79	52142.08
Clusters	**100.00%**	**0.22**	**61.81**	**4636.89**

As corroborating evidence, ACE, the program used here to test the heuristics, can also be used to learn weights for known heuristics. Highly-weighted heuristics are those that succeeded in solving problems in a particular class. For class-A problems, the learning version of ACE determined that *min ddom, max deg*, and *dom/deg* all performed worse than random decisions, that is, that they are in effect, anti-heuristics for this problem class [15].

Table 4. Performance on 100 geometric problems where clusters are inapplicable.

Heuristic	Solved	Time	Steps	Checks
dom/ddeg	100%	9.89	424.76	329008.53
Tension	100%	7.20	359.95	325046.60
Cliques	100%	8.89	530.08	411413.87
Near cliques	100%	5.83	319.53	258948.42

In a structured CSP with uniform tightness, clusters cannot be computed, but near cliques are effective. Consider, for example, a *geometric CSP*, constructed by scattering a set of points at random on the Cartesian plane — each point becomes a variable in the problem; constraints are formed among any pair of variables within a some specified distance of each other, with additional constraints added to connect the constraint graph [16], which is ridden with tightly connected sets of vertices. We generated 100 geometric problems with 50 variables domain size 10, resultant density 0.4, and tightness 0.82. The 6.05 cliques in these problems averaged 12.65 variables; the 5.39 near cliques averaged 14.28 variables. As Table 4 indicates, all three of our applicable methods solved these problems in 20000 steps, and did so faster, despite the preliminary time required to detect crucial subproblems.

Purely random problems should not display structure. Nonetheless, we tested our methods on two challenging classes of them: <50,10, 0.38, 0.8> with fixed edge tightness and <50,10, 0.38, 0.8> with varying edge tightness. Although they are larger than the composed problems, these random problem had smaller maximum cliques and near cliques (5.89 and 5.86 variables, respectively). They also had more of these small subproblems: a median of 10 cliques and 7 near cliques. Clusters, by definition, cannot be detected in fixed random problems. In random varying problems, no cluster was detected in one problem; in another the cluster was size 4; in all the other problems cluster size ranged from 6 to 9 and averaged 7.53, again smaller than those in the composed problems.

The performance results on random problems indicate that when structure is not intrinsic to a problem, clusters are ineffective. On random fixed problems, cliques and near cliques solved a few more problems than *dom/ddeg*, but tension solved 47% more problems than *dom/ddeg* within the allotted number of steps. Surprisingly, tension made less of an impact (21% more solved problems) on random varying problems. Our explanation is that, although tension appears to be more of a factor in varying problems, a high-tension vertex may still not have enough neighbors to warrant attention very early in search. Density still matters. Nonetheless, clusters performed poorly. Inspection indicates that the average degree of vertices in the clusters tended to be substantially lower than the average degree in the near cliques. That, combined with the fact that only a single cluster was used, accounted for their poor performance. The moral here is that if a cluster is present because the problem is structured, it can be detected and exploited; if it is not, what passes for a cluster may be misleading.

Other graph-based approaches to solving CSPs have sought to decompose the constraint graph to facilitate search [17-21]. None of them, however, addressed the tightness of the constraints in any way. Clusters derive a substantial advantage because they consider more than mere structure.

Future work includes variations on the use of crucial subproblems and specialized propagation to work with these subproblems. Clusters could also support search for all solutions to a problem, and provide an explanation to the user of the particular difficulties that are likely to arise before a solution is attempted.

Acknowledgements: We are indebted to Pierre Hansen for his enthusiasm about VNS and to Gene Freuder, whose breadth of knowledge and curiosity about cliques inspired this work. Thanks also to Tiziana Ligorio, Saad Mneimneh, and Smiljana Petrovic for many interesting discussions. This work was supported in part by NSF IIS-0328743, PSC-CUNY, and Science Foundation Ireland under Grant 00/PI.1/C075.

References

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

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

[3] Hansen, P. and N. Mladenovic, Variable Neighborhood Search, in *Handbook of Metaheuristics*, F. W. Glover and G. A. Kochenberger, Editors. 2003, Springer: Berlin.

[4] Bessière, C. and J.-C. Régin. Enforcing arc consistency on global constraints by solving subproblems on the fly. In the *Proceedings of Fifth International Conference on Principles and Practice of Constraint Programming (CP-99)*. 1999. p. 103-117.

[5] van Hoeve, W. J. and I. Katriel, Global Constraints, in *Handbook of Constraint Programming*, F. Rossi, P. Van Beek and T. Walsh, Editors. 2006: Prague.

[6] Hallodórsson, M. M. and H. C. Lau, Low-degree Graph Partitioning via Local Search with Applications to Constraint Satisfaction, Max Cut, and Coloring. *Journal of Graph Algorithms and Applications*, 1997. 1(3): p. 1-13.

[7] Eisenberg, C. and B. Faltings. Using the Breakout Algorithm to Identify Hard and Unsolvable Subproblems. In the *Proceedings of Principles and Practice of Constraint Programming CP-2003, LNCS 2833*. 2003: Springer Verlag. p. 822-826.

[8] Beale, S., S. Nirenburg and K. Mahesh. HUNTER-GATHERER: Three Search Techniques Integrated for Natural Language Semantics. In the *Proceedings of AAAI-96*. 1996. Portland: AAAI. p. 1056-1061.

[9] Aardal, K. I., S. P. M. van Hoesel, A. M. C. A. Koster, C. Mannino and A. Sassano, Models and solution techniques for frequency assignment problems. *4OR: A Quarterly Journal of Operations Research*, 2003. 1(4): p. 261-317.

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

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

[12] Mackworth, A. K., Consistency in Networks of Relations. *Artificial Intelligence*, 1977. 8: p. 99-118.

[13] Bayardo, R. J. J. and R. Schrag. Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances. In the *Proceedings of Second International Conference on Principles and Practice of Constraint Programming CP-1996*. 1996. p. 46-60.

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

[15] Petrovic, S. and S. L. Epstein. Learning Weights for Heuristics that Solve Constraint Problems. In the *Proceedings of Workshop on Learning to Search at AAAI-2006*. 2006. Boston.

[16] Johnson, D. B., C. R. Aragon, L. A. McGeooh and C. Schevon, Optimization by Simulated Annealing: An experimental evaluation; Part 1, Graph partitioning. *Operations Research*, 1989. 37(865-892).

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

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

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

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

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