STRUCTURE-BASED SEARCH TO SOLVE CONSTRAINT SATISFACTION PROBLEMS

by

XINGJIAN LI

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy
The City University of New York

2011

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Professor Susan L. Epstein

Date

Chair of Examining Committee

Professor Theodore Brown

Date

Executive Officer

Professor Amotz Bar-Noy

Professor Melvin Fitting

Professor J. Christopher Beck
(Supervisory Committee)

The City University of New York

ABSTRACT

STRUCTURE-BASED SEARCH TO SOLVE CONSTRAINT SATISFACTION PROBLEMS

by

Xingjian Li

Advisor: Professor Susan L. Epstein

Constraint satisfaction is a paradigm that applies to many challenging real-world tasks with direct practical relevance, such as planning and scheduling. It models these tasks as constraint satisfaction problems (*CSP*s) and then solves them with search. Compared to artificially-generated CSPs, real-world CSPs are more likely to have non-random structure. This dissertation shows the difficulties encountered by traditional and state-of-the-art search techniques when the structure of such a problem is overlooked. Here, a novel hybrid search algorithm for CSPs combines two fundamental search paradigms (global search and local search) to solve structured CSPs. It first uses local search to detect crucial structure (the *structure identification stage*), and then applies that structural knowledge to solve the problem by global search (the *search stage*).

In the structure identification stage, we adapt a local search metaheuristic to find crucial structure within a CSP. This dissertation presents different structure detection methods based on different metrics. Static metrics from the original problem are sometimes available without any cost and are also straightforward to use, but they may not represent the real challenges within the problem. When dynamic metrics reveal the true crucial structure, they provide better guidance for search. On easier problems,

however, this benefit does not justify the additional computational cost of the dynamic metrics.

In the search stage, the structure identified in the previous stage guides global search for a solution. This dissertation presents various approaches to exploit this structure, which is presumably the most difficult area for search. Under this premise, search can either address the identified structure earlier than other areas of the problem, or allow the structure to direct *inference*, a technique that reduces search space for global search.

In addition, this dissertation presents a new visualization tool for binary CSPs and the structures identified by our algorithms. The tool inspires, supports, and verifies the design and improvement of structure-based search. On benchmark and real-world CSPs, structured-based search not only improves search performance, but also provides users with direct explanations and visualization of the inherent challenges in each problem.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Professor Susan L. Epstein. She displays the skills required to be a strong research scientist. The weekly paper-reading lab meeting that she leads prepared me a solid and broad foundation of knowledge on constraint solving and machine learning for rigorous scientific research. More importantly, she allowed me the freedom to develop my own ideas and strengths. Her enthusiasm and optimism encouraged me in the most difficult moments during my studies. I hereby thank Prof. Epstein for her excellent guidance, persistent inspiration and unwavering support throughout the years.

I would like to thank my committee members, Professors Amotz Bar-Noy, J. Christopher Beck and Melvin Fitting, for their scientific advices throughout the course of my project and their valuable comments on my dissertation. I would also like to thank Dr. Richard Wallace from the Cork Constraint Computation Centre for his constructive suggestions and inspirations for my research. Appreciation also goes to the chair of the Computer Science Ph.D. Program, Professor Theodore Brown, for the helpful discussion on my academic achievements in every semester.

I would like to thank all of the past and present members of the Problem Solving and Machine Learning lab in Hunter College, in particular, Tiziana Ligorio, Eric Osisek, Smiljana Petrovic, Xi Yun and Zhijun Zhang, for their helps, suggestions and friendships. It has been a long run and we share a lot of memories: passionate discussions, exciting moments on papers and new ideas, and frustration over unexpected empirical

results. Every day in this lab was a learning and growing experience for me. I will miss every moment of my stay with you guys!

Last but not least, I would like to thank my family. My parents, Weiquan and Jihuang, showed me the way into the world of science in my childhood. I thank them for their unconditional support and encouragement. My son Felix's journey to this world accompanied every minute of my writing. He fills my life with happiness and joy and makes all the hard effort worthwhile. Finally, I want to thank my wife, Yizhou, who always believes in me. Her love, support and patience made me survive this journey over the years.

DEDICATION

To my parents and grandparents
for bringing me to this world and raising me

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Many decision-making problems have restrictions. For example, to avoid wake turbulence, an airplane must take off at least two or three minutes behind an airplane on the same runway. In a Sudoku game, all numbers in the same row must be different. Solutions to these problems require decisions that satisfy their restrictions. Such problems can be modeled naturally as constraint satisfaction problems (*CSP*s). A CSP is a set of *variables* (here with finite and discrete domains) and a set of constraints that restrict simultaneous value assignments of subsets of those variables. A CSP represents decisions with variable assignments and restrictions with constraints. A *solution* of a CSP is an assignment of values to its variables that satisfies all its constraints. A CSP solver explores, systematically or stochastically, the search space of all possible variable assignments to find solutions. Because CSP solution is NP-complete (Mackworth and Freuder, 1993), *search heuristics* to guide exploration of the search space and *inference* to narrow options during search are crucial to the solver's performance.

A general-purpose heuristic may mislead the solver if it overlooks a CSP's inherent non-random structure. Modern algorithms seek to learn about such structure when they encounter it during search. The thesis of this dissertation is that it is possible and beneficial to:

- identify crucial structures within a constraint satisfaction problem before search

- exploit that knowledge to solve the problem
- explain the problem's inherent difficulties with that knowledge.

The novelty of this research is that we adapt a local search metaheuristic to identify crucial structure (clusters) within CSPs. To detect clusters, we use various metrics including constraint tightness and conflict-directed weights on constraints learned during probing. Topological characteristics, such as graph density, are used to detect structure in CSPs. Conflict-directed heuristics maintain and use constraint weights to guide search. They may use probing to improve the quality of constraint weights before search. This research is the first to use constraint tightness or constraint weights learned during probing to detect CSP structure.

The principal results of this research are

- It is possible to rapidly detect CSP structures called *clusters* before search.
- The exploitation of clusters solves CSPs more effectively.

This structure-based search not only outperforms traditional heuristics on challenging CSPs by as much as an order of magnitude, but also offers insight into the nature of these problems.

The remainder of this chapter provides formal definitions, discusses major paradigms for CSP search algorithms and inference, and describes related work on structured CSPs. Chapter 2 introduces a visualization tool, *DrawCSP*. This tool visualizes a *binary* CSP, where each constraint restricts no more than two variables, in two different ways. The structure that *DrawCSP* reveals inspires and directly supports the structure-based search presented here. *DrawCSP* can also visualize the structure identified by our algorithms and, thereby, permits the user to compare detected structure with the problem's original

structure. Visualizations from *DrawCSP* are used throughout this dissertation. Chapters 3 and 4 cover the two steps of structure-based search for CSP: structure identification and search. In Chapter 3, the structure identification algorithm greedily and stochastically builds clusters. We use three different metrics in the greedy algorithm: one readily-available measure of constraint restrictedness and two adaptive measures of the location of constraint violations during search. The adaptive measures require a *probing* stage before the structure identification stage to build clusters. Chapter 4 applies identified structure to search. It explores several search heuristics based on the knowledge detected in the structure identification stage, and investigates how such structural knowledge can be used in inference. Chapter 5 provides empirical results on structure-based search for both benchmark and read-world problems and an application of our structure identification algorithm to a biology problem. The final chapter discusses this research and future work. A glossary of terms precedes the references.

1.1 Constraint satisfaction problems

Formally, a constraint satisfaction problem is represented by a triple $<X, D, C>$, where:

- X is a set of variables, $X = \{X_1,\ldots,X_n\}$.
- The *domain* $D_i \in D$ of variable X_i is a finite and discrete set of its possible values.
- C is a set of constraints. A constraint C_i restricts simultaneous value assignments on its scope, $S_i \subseteq X$.

1.1.1 Preliminaries

An *instantiation* of size z for a problem $<X, D, C>$ assigns values to a subset of z variables in X. If $z < n$, it is a *partial instantiation*; if $z = n$, it is a *full instantiation*. A *solution* to a CSP is a full instantiation that satisfies every constraint. A CSP is said to be *unsolvable* if it does not have a solution, and *solvable* if it has at least one solution. The *search space* of a CSP consists of all partial and full instantiations.

The *arity* of a constraint is the *cardinality*, or size, of its scope. A *unary* constraint is defined on a single variable; a *binary* constraint, on two variables. This dissertation focuses on *binary CSPs*, which have only unary and binary constraints. For illustration and/or inspiration, a binary CSP can be represented as a *constraint graph*, where each variable is a node, each binary constraint is an edge, nodes are labeled by their domains, and each edge is labeled by the relation for that constraint. Figure 1.1 shows the constraint graph for a coloring problem, with one of its solutions.

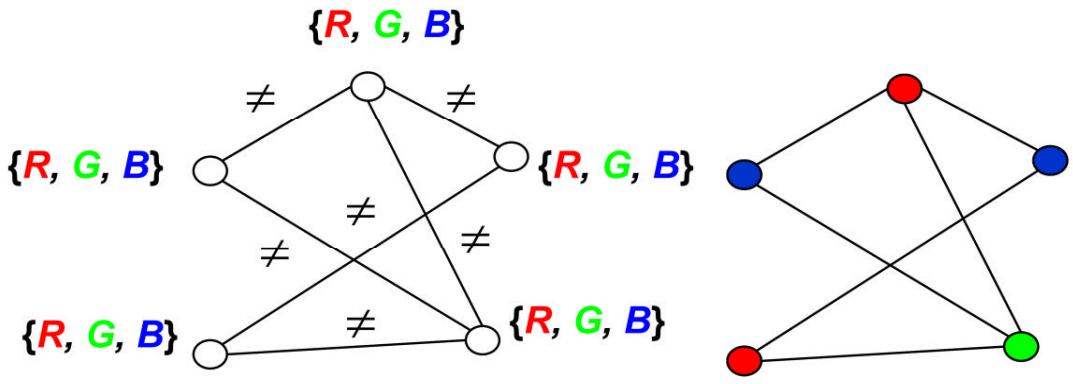


Figure 1.1. The constraint graph of a coloring problem with five variables and six constraints. (a) Each variable is labeled by its domain and each constraint is labeled by its relation (not-equal). (b) A solution to this problem, where no pair of neighbors share the same color.

The *density* $d(P)$ of a binary CSP $P = \langle X, D, C \rangle$ with $|X| = n$ is the fraction of the $\frac{n(n-1)}{2}$ possible pairs of variables restricted by C:

$$d(P) = \frac{2|C|}{n(n-1)} \quad [1]$$

For example, the density of the problem in Figure 1.1 is 0.6. Two variables whose nodes in a constraint graph are joined by an edge are *neighbors*. The *static degree* of a variable is the number of its neighbors. A *subproblem* $\langle X', D', C' \rangle$ of a CSP $\langle X, D, C \rangle$ is a CSP induced by a subset of variables $X' \subseteq X$, where $D' = \{D_i \in D \mid X_i \in X'\}$ and $C' = \{C_i \mid S_i$ is the scope of C_i and $S_i \subseteq X'\}$.

1.1.2 Tightness

Let a binary constraint C_{ij} have scope $\{X_i, X_j\}$, where X_i's domain is D_i and X_j's domain is D_j. If C_{ij} excludes r value pairs from $X_i \times X_j$, then its *tightness* $t(C_{ij})$ is defined as the proportion of possible value pairs C_{ij} excludes:

$$t(C_{ij}) = \frac{r}{|D_i| \times |D_j|} \text{ where } r \leq |D_i| \times |D_j| \quad [2]$$

For example, each of the constraints in Figure 1.1 has tightness 3/9. A *constraint* $C_i \in C$ uses a relation R_i to indicate which tuples in the Cartesian product of the domains of the variables in its scope $S_i \subseteq X$ are acceptable. If C_i is *extensional*, R_i enumerates the simultaneous legal (or forbidden) value tuples. If C_i is *intensional,* R_i is a boolean predicate that returns *true* on simultaneous legal value tuples and *false* on all other tuples. This research deals primarily with *extensional CSPs*, all of whose constraints are

extensional. (Some empirical results on *intensional CSPs*, all of whose constraints are intensional, appear in Chapter 5.)

For intensional C_{ij}, [2] requires a test on every possible tuple. Often, however, an intensional constraint is defined with a predicate (e.g., an inequality or a disjunctive inequality over integer intervals) that permits direct calculation of $t(C_{ij})$. For a constraint C_{xy} with scope $\{x, y\}$ and relation $x + T \leq y$, for some constant $T \in \mathbb{Z}^+$, the domains of x and y are integer intervals $D_x = [L_x, U_x]$ and $D_y = [L_y, U_y]$. Clearly, for $x \in D'$ and any y, C_{xy} is satisfied exactly when $D' = [L_x, y - T]$. If $L_x > U_y - T$, $D' = \varnothing$ for all y, so $r = |D_x||D_y|$ and $t(C_{xy}) = 1$. When $L_x \leq U_y - T$, for each $y \geq L_x + T$ and any $x \in [L_x, y - T]$, C_{xy} is satisfied. Thus the number of pairs of values accepted by C_{xy} is

$$\sum_{y=L_x+T}^{U_y} |D'| = \sum_{y=L_x+T}^{U_y} (y - T - L_x + 1) = \frac{(U_y - T - L_x + 2)(U_y - T - L_x + 1)}{2}$$

Let $M_T = U_y - T - L_x + 1$ if $L_x \leq U_y - T$, 0 otherwise. For intensional C_{xy} of this type

$$t(C_{xy}) = 1 - \frac{(M_T + 1)M_T}{2|D_x||D_y|} \qquad [3]$$

The tightness of an intensional constraint with a disjunctive inequality predicate can be similarly calculated. For a constraint C_{xy} with scope $\{x, y\}$ and relation $(x + T \leq y$ OR $y + W \leq x)$, for some constants $T, W \in \mathbb{Z}^+$, the domains of x and y are integer intervals $D_x = [L_x, U_x]$ and $D_y = [L_y, U_y]$. Because T and W are both positive, either $x + T \leq y$ or $y + W \leq x$, but not both, can be satisfied. Therefore, the number of pairs of values accepted by C_{xy} is the sum of the number of pairs of values accepted by $x + T \leq y$ and the number of pairs of values accepted by $y + W \leq x$. Let

$M_T = U_y - T - L_x + 1$ if $L_x \leq U_y - T$, 0 otherwise and $M_W = U_x - W - L_y + 1$ if $L_y \leq U_x - W$, 0 otherwise. Then the number of pairs of values accepted by C_{xy} is

$$\frac{(M_T + 1)M_T + (M_W + 1)M_W}{2}$$

Therefore, for intensional C_{xy} of this type

$$t(C_{xy}) = 1 - \frac{(M_T + 1)M_T + (M_W + 1)M_W}{2|D_x||D_y|} \quad [4]$$

Because [3] and [4] use no iteration to compute the tightness of C_{xy}, the computation time is independent of $|D_x|$ and $|D_y|$. This is desirable because such variables' domains may be quite large.

1.1.3 Problem classes

A CSP *problem class* is a set of CSPs categorized as similar, such as all CSPs that share the same parameters $<n, k, d, t>$, where n denotes the problem's number of variables, k its maximum domain size, d its density, and t the average of the tightness of its individual constraints. Without further restriction on them, such CSP classes are said to have *random structure* and can be generated by Model B (Gent et al., 2001).

Classes of parameterized CSPs with non-random structure can be built from these random CSPs. For example, a class of artificially-generated *composed* CSPs (Aardal et al., 2003) can be written as

$$<n, k, d, t> s <n', k', d', t'> d'' t'' \quad [5]$$

Each problem in [5] partitions its variables into subproblems, one designated as the *central component* and the others as *satellites*. There are some constraints (*links*) between satellite variables and variables in the central component, but none between variables in

distinct satellites. A CSP in [5] has its central component in <n, k, d, t> and s satellites each in <n', k', d', t'>. In addition, there are l links between the central component and the s satellites in [5]. The link density d'' is the fraction of the nsn' possible link edges present:

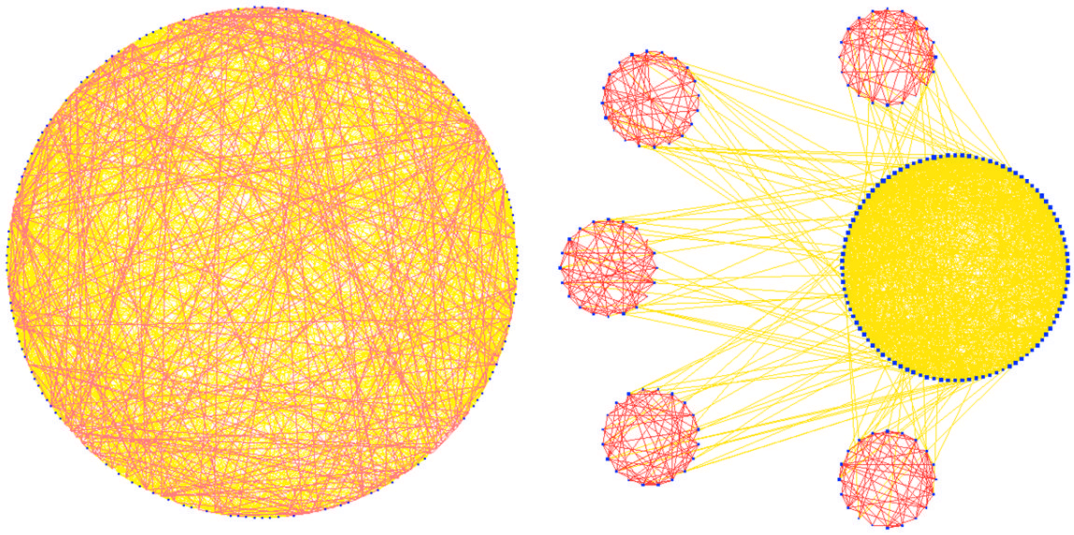
$$d'' = \frac{l}{nsn'} \text{ where } l \le nsn' \qquad [6]$$

The link tightness t'' is the average tightness of all l links.

An example of a class of composed problems, which here we call *Comp*, is

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

It has one uniform tightness (0.05) within its central component, another along its links (again 0.05 in this example), and a third (0.50) within its satellites. Figure 1.2(a) is the



(a) (b)

Figure 1.2. Two constraint graphs for the same *Comp* problem. Edge colors vary from yellow to red as the tightness of the corresponding constraint increases. (a) An uninformative representation. (b) Another constraint graph reveals the problem's structure.

constraint graph of a particular *Comp* problem with its 200 variables and 1257 constraints. Here, the variable labels (domains) and constraint labels (relations) have been omitted to focus on the problem's structure. Edge colors in the figure vary from yellow to red as the tightness of the corresponding constraints increase. When variables are randomly plotted on the circumference of a circle, as in Figure 1.2(a), this problem appears to have random structure. In contrast, Figure 1.2(b) reveals its true structure; it draws the variables in the sparse and loose central component along a large circle and the variables in each dense and tight satellite on separate small circles. *Structured* CSPs, including those in *Comp*, have characteristics that can be exploited by a specialized method to outperform a more general one. They also offer an opportunity to explore the impact and management of difficult subproblems.

1.2 Global search

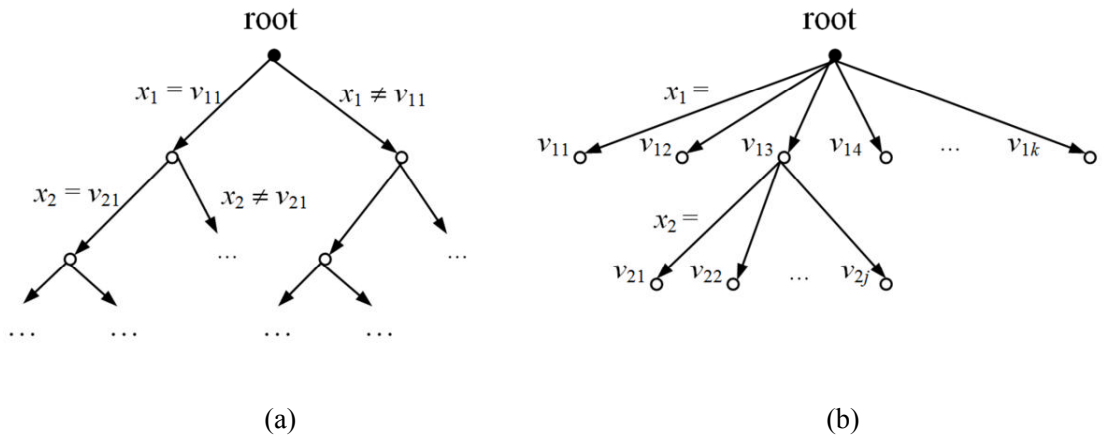
Search for a solution to a CSP moves through the space of its possible instantiations. A *complete search* is always able to find a solution if there is any. Failure by a complete algorithm to find a solution is a proof of the problem's insolvability. A problem is termed solved in this dissertation if global search finds a solution or proves that none exists.

Global search traverses the space of partial and full instantiations, while *local search* is restricted to the space of full instantiations. Global search begins with an *empty instantiation*, where no variable is assigned a value. It traverses the search space systematically, assigning a value to one unbound variable at a time. Global search is complete.

During global search, from the perspective of a given node, an instantiated variable is called a *past variable* and an unbound variable is called a *future variable.* The *dynamic degree* of a variable is the number of its neighbors that are future variables. An instantiation for a future variable is *consistent* if it does not violate any constraint on a past variable. An *inconsistent* instantiation violates one or more constraints.

1.2.1 Branching and backtracking

When global search traverses the search tree, it branches whenever an uninstantiated variable $x \in X$ is instantiated with a value $v \in D_x$. This is equivalent to adding the constraint $x = v$ to the original problem; it results in a subproblem, whose solution becomes part of the solution of the original problem. *Two-way branching* creates a binary search tree where one child node assigns a value to a variable and the other child node removes the same value from the domain of that variable, as in Figure 1.3(a). In *k-way branching* each child node represents a different value assignment for a single variable, as in Figure 1.3(b). All experiments in this dissertation use k-way branching.



(a) (b)

Figure 1.3. (a) Two-way branching and (b) *k*-way branching.

Global search extends the current partial instantiation by assigning a consistent value, if one exists, for the currently-selected variable. A *deadend* is encountered if there is no consistent assignment for the currently-selected variable. Because inference is often used together with global search, a more commonly used definition of deadend is a partial solution that makes some future variable's domain empty during inference. (See Section 1.3 for more details on inference.) In such a case, search returns to some variable instantiated before the current variable, and retracts that assignment. This is a *backtrack*. *Chronological backtracking* revisits each value-assignment in order of recency. Because chronological backtracking can rediscover the same inconsistency repeatedly, it can be outperformed by other backtracking methods, such as *backjumping* that retract to some earlier value-assignment (Gaschnig, 1979; Dechter and Frost, 2002). Nonetheless, chronological backtracking is still commonly used in global search for its simplicity. All experiments in this dissertation use chronological backtracking.

1.2.2 Search heuristics

The order in which variables and values are selected is important. Proper selection of the next variable to instantiate and the value to assign to it can significantly improve search performance (Bessiere and Regin, 1996; Smith and Grant, 1998). Variable-ordering heuristics and value-ordering heuristics provide crucial advice for global search. A variable-ordering heuristic usually follows the *fail-first principle*: "To succeed, try first where you are most likely to fail" (Haralick and Elliot, 1980). The fail-first principle underlies many traditional variable-ordering heuristics intended to speed global search (Gent et al., 1996; Smith, 1999; Bessière et al., 2001). Recent work shows that two important factors underlie most effective variable-ordering heuristics: "immediate failure"

(e.g., the fail-first principle) and "future failure" (Wallace, 2005). An example of the latter is maximum forward degree as a variable-ordering heuristic. It increases the possibility of future failure through inference. (Inference is described in the next section.)

There are many common variable-ordering heuristics. *MinDom* seeks to minimize search tree size by reducing the branch factor; it prefers variables with small *dynamic domains,* the values remaining after inference (Dechter and Meiri, 1989). *MaxDeg* focuses on variables with many constraints; it prefers variables with high dynamic degree. *MinDomDeg* combines the two: it prefers variables that minimize their ratio of dynamic domain size to dynamic degree (Bessiere and Regin, 1996; Smith and Grant, 1998). Although *MinDomDeg* is an effective off-the-shelf variable-ordering heuristic, it can be outperformed, especially on structured CSPs, by ordering heuristics that learn during search. Some heuristics that learn are discussed in Section 1.3.2.

While most variable-ordering heuristics obey the fail-first principle, value-ordering heuristics try to make search succeed. One way to achieve success is to maximize the number of options available. The *promise* of each value, the product of the domain sizes of the future variables after choosing it, can be calculated and used as a heuristic to select values (Geelen, 1992). The promise of a value is actually an upper bound on the number of possible solutions resulting from the instantiation. One common value-ordering heuristic maximizes promise. This research uses only numerical value ordering, which always selects the smallest value first.

1.3 Inference

Because global search traverses a search space systematically to find a solution, reduction of the search space can speed the process. *Inference* seeks to transform a CSP into an equivalent problem with a smaller search space through reasoning. It is, therefore, usually used together with global search. During global search, after every variable assignment, inference propagates the effect of this assignment to nearby unassigned variables and removes values from these variables' domains that are inconsistent with the current assignment.

1.3.1 Inference methods

Inference reasons about the effect of an instantiation on the remainder of a problem. *Forward checking* is a simple inference strategy (Haralick and Elliot, 1980). Immediately after the instantiation of a variable x, forward checking removes all inconsistent values from the domain of every uninstantiated neighbor of x. During forward checking, if some future variable's domain becomes empty (*wipeout*), the instantiation of the current variable is retracted and the domains that were reduced by that instantiation are restored. Since forwarding checking does not explore beyond the immediate neighbors of x, its power is limited.

Arc consistency is a potentially more powerful inference method (Mackworth and Freuder, 1985). Along a binary constraint C_{ij} with scope (X_i, X_j) in a CSP, value $a \in D_i$ is *supported* by value $b \in D_j$ if and only if $X_i = a$ and $X_j = b$ together satisfy C_{ij}. C_{ij} is *arc consistent* if and only if every value in D_i has some support in D_j and every value in D_j has some support in D_i. If every constraint in a CSP is arc consistent, then the CSP is *arc*

consistent. *MAC*-3 is an inference method that maintains a problem's arc consistency during search. Immediately after the instantiation of variable X_i, MAC-3 enqueues the edges from X_i to all its future-variable neighbors, and then checks each element of the queue for domain reduction (Sabin and Freuder, 1997). Whenever this process reduces the domain of any future variable X_k, MAC-3 enqueues every constraint between X_k and its future-variable neighbors, and iterates until its queue is empty. Although arc consistency is more computationally expensive than forward checking, it has been shown to outperform forward checking empirically (Bessiere and Regin, 1996). Therefore, it is often used in CSP search.

1.3.2 Inference-based heuristics

Inference methods are usually combined with global search. Figure 1.4 is the pseudocode for global search with inference. After the instantiation of the current variable, inference can cause a wipeout on a future variable. A history of such conflicts can be used to guide search.

Weighted degree (*wdeg*) is a conflict-directed metric on variables (Boussemart et al., 2004). For *wdeg*, each constraint is associated with a weight, initialized to 1. During

Algorithm 1: Pseudocode for CSP global search

```
until (all variables have values that satisfy all constraints OR
        the variable selected first has an empty domain)
    select a variable
    assign a value to this variable
    infer the impact of this assignment ; *inference*
    if a wipeout occurs, backtrack
if (all variables have values that satisfy all constraints) return solution
else return "unsolvable"
```

Figure 1.4. Pseudocode for global search.

search, if constraint C_{ij} causes a wipeout for X_j, the weight of this constraint is incremented by 1. At any time during search, the weighted degree of a variable X_i is the sum of the weights of all constraints whose scopes include X_i and some future variable.

MaxWdeg is a variable-ordering heuristic that selects a variable with the highest weighted degree (Boussemart et al., 2004). *MaxWdeg* is adaptive; it gradually guides search toward variables whose constraints cause wipeouts. Another adaptive heuristic, *MinDomWdeg*, minimizes the ratio of dynamic domain size to weighted degree (Boussemart et al., 2004). It combines the two important "fail-first" factors: "immediate failure" and "future failure" and is considered a state-of-the-art variable-ordering heuristic. Section 3.2.2 has further details on weighted degree and how to use it to identify conflict-directed structure.

1.4 Local search

Global search is constructive; it builds solutions by iteratively extending partial solutions systematically. In contrast, *local search* begins with a *candidate solution*, a full instantiation that may or may not be a solution, and moves to another candidate solution in the search space. Local search is incomplete because it does not traverse the search space systematically. Figure 1.5 provides pseudocode for local search.

1.4.1 Neighborhood

The *distance* between two candidate solutions is the number of variable to which they assign different values. The *k-neighborhood* of a candidate solution *s* includes *s* and all the candidate solutions within distance *k* of *s*. Any candidate solution in *s*'s neighborhood

Algorithm 2: Pseudocode for CSP local search

```
current-solution ← Generate-Initial-Full-Instantiation
until current-solution violates no constraint
   best-neighbor ← Find-best-neighbor(current-solution)
   if best-neighbor is better than current-solution
   then current-solution ← best-neighbor
   else Escape-local-optimum
return current-solution
```

Figure 1.5. Pseudocode for local search.

is *s*'s *neighbor*. In local search, each move is determined by a decision based on only *local knowledge*, the information inherent in the neighborhood of the candidate solution being investigated.

Given a neighborhood, local search uses an evaluation function to determine which neighbor is the most improved candidate solution. An *evaluation function* for a problem is a mapping of candidate solutions onto the real numbers, where solutions map onto the global maximum. Given an evaluation function, local search moves greedily: it chooses the neighbor that maps onto a maximum for the neighborhood and is better than the current candidate solution. Given an evaluation function *f*, a search space *S* and a neighborhood relation $N \subseteq S \times S$, a candidate solution *s* is a *local optimum* if and only if for all $s' \in N(s)$, $f(s') \leq f(s)$ and *s* is not a solution. A *strict local optimum* is a *local optimum* if and only if for any $s' \in N(s)$, $f(s') < f(s)$ (Hoos and Stutzle, 2005).

When a local optimum is reached, search can only wander among local optima or remain at the strict local optimum candidate solution. This phenomenon is called *stagnation*. A local search strategy that uses a *k*-neighborhood can avoid local optima under the *complete neighborhood relation* when $k = n$, because its local optima are also

global optima. Finding the best neighbor in the complete neighborhood is equivalent to solving the CSP and is therefore NP-complete.

Because local search does not traverse the search space systematically, it can become trapped at a local optimum, and some points in the search space may be inaccessible from the start state. Nonetheless, local search often finds CSP solutions much faster than systematic search. For example, a local search algorithm found solutions to the million-queens problem (Minton et al., 1990). In some real-time problems, (e.g., game playing), search is time-limited and local search may be more efficient. In addition, when the time limit is reached and a systematic search is aborted, no full solution is available at all, since some variables are not instantiated. A local search algorithm, however, always has a candidate solution and could return the best solution it has found so far. Local search normally requires less space than systematic search because it remembers few visited states.

1.4.2 Intensification and diversification

The best-performing local search algorithms use randomization to generate candidate solutions initially and to move between states during search (Hoos and Stutzle, 2005). These algorithms do *stochastic local search* (*SLS*). *Iterative best improvement* always chooses the neighbor with the highest score from the evaluation function. Maximizing the improvement in every step increases the efficiency of the algorithm. At the same time, however, the possibility of getting trapped in a local optimum is also increased, which significantly reduces the algorithm's performance. This is the trade-off between diversification and intensification.

Intensification is the extent to which some limited portion of the search space is visited; *diversification* is the extent to which different areas of the search space are visited. Depending upon the topology of the search space, a greedy search strategy could intensify or diversify as it follows the cost gradient. A typical example of intensification is to move to a candidate solution with a higher evaluation in local search. A diversification strategy seeks reasonable coverage of the search space to reduce the likelihood of entrapment in local optima. One example of diversification is *random walk*, which moves to an arbitrary candidate solution without consideration of its quality. Pure diversification, such as applying random walk throughout search, is very inefficient (Hoos and Stutzle, 2005).

The combination of diversification and intensification typically outperforms both pure strategies (Hoos and Stutzle, 2005). A larger neighborhood is an effective way to avoid local optima. For example, during each local search step of *iterative first improvement*, the algorithm evaluates the neighboring candidate solutions s' in a particular order, and selects the first s' for which $f(s') > f(s)$. The order in which the neighbors are evaluated has a significant effect on the performance of an iterative first improvement strategy. If a fixed neighbor evaluation order is used, and one begins with the same initial candidate solution, the same local optimum will be encountered. Thus, a random neighbor evaluation order is usually used. Iterative first improvement not only leads to some diversification, but avoids the time required to evaluate all neighboring candidate solutions as well.

1.4.3 Variable Neighborhood Search

Local search can also be improved by changing neighborhoods during search. The local optima of a neighborhood are not necessarily also local optima in another neighborhood. The larger the neighborhood a candidate solution checks, the better the chance to improve the current candidate solution and the less likely it is to get trapped in local optima. Variable Neighborhood Search (*VNS*) is a metaheuristic that uses multiple neighborhoods of increasing size (Hansen et al., 2004). Figure 1.6 provides pseudocode for VNS, a non-deterministic search through k neighborhoods.

VNS succeeds on a wide range of combinatorial and optimization problems (Hansen and Mladenovic, 2003). Figure 1.6 is a general algorithm that uses VNS to search. It works outward from an initial subset of variables (Figure 1.6, line 1) in a relatively small neighborhood in a graph through a sequence of k pre-specified, increasingly large neighborhoods (lines 2–3). An example of such a sequence could be the 1-neighborhood, 2-neighborhood, and so on of a set of variables in a graph. Let $N_k(X)$ be the

Algorithm 3: VNS on *k* neighborhoods

```
1  best-yet ← initial-subset
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 ← VND(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 1.6. Pseudocode for Variable Neighborhood Search.

k-neighborhood of a vertex X in a graph. $N_k(X)$ is recursively defined by:

$N_1(X)$ = set of all neighbors of X

$$N_k(X) = N_{k-1}(X) \cup \left(\bigcup_{Y \in N_{k-1}(X)} N_1(Y) \right)$$

Similarly, the *k-neighborhood of a set of vertices S* is defined by:

$$N_k(S) = \bigcup_{X \in S} N_k(X)$$

Each neighborhood restricts the current options; as VNS iterates, each new neighborhood provides a larger search space.

Within a neighborhood, Variable Neighborhood Descent (*VND*) is a local search algorithm that tries to improve the current subset (*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) 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 large neighborhoods without improvement. The initial subset, the score metric, and the local search routine vary with the application. VNS is used to find structure in a CSP in Chapter 3.

VND greedily extends *best-yet* within *neighborhood* (Figure 1.7). Once its greedy steps are exhausted, VND repeatedly interchanges one element of its current subset for two elements in *neighborhood* and breaks ties greedily (line 5). For example, a *clique* is a maximally dense graph, that is, one with all possible edges between its variables. In the search for a maximum clique, VND swaps out some variable *v* in *best-yet* for two

Algorithm 4: VND, local search algorithm called by VNS

```
1  best-yet ← greedy-extension (best-yet)
2  index ← 1
3  neighborhood ← neighborhood(index)
4  until stopping condition or index = k
5      local-optimum ← interchange(best-yet, neighborhood)
6      if score(local-optimum) > score(best-yet)
7      then best-yet ← local-optimum
8          index ← 1
9      else index ← index + 1
10     neighborhood ← neighborhood(index)
```

Figure 1.7. Pseudocode for Variable Neighborhood Descent.

adjacent variables that are not neighbors of *v* and were not in *best-yet,* but are neighbors of all the other variables already in *best-yet*, and breaks ties with maximum variable degree. An alternative produced by VND replaces *best-yet* only if it outscores it.

Figure 1.8 is a sample of steps that might occur during a call to VND during search for a maximum clique in a simple graph (Epstein and Wallace, 2006). The initial subset is a vertex that is a neighbor of every vertex in the graph, and the local search metric is subset size. VND 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 1.8(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.

Local search is incomplete; it cannot prove a CSP is unsolvable, as real-world problems often are. This research first exploits local search to consider the $O(2^n)$ set of

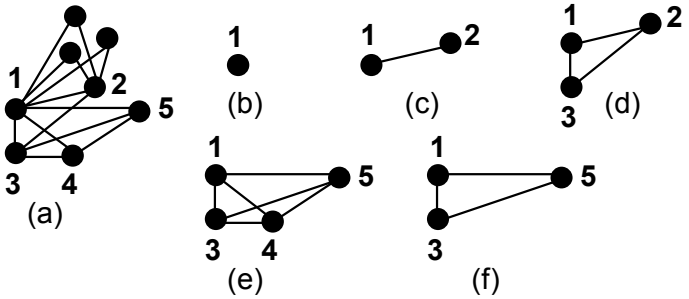


Figure 1.8. Selected VND steps to find a maximum clique in graph (a). (b) A starting vertex. (c)-(d) Greedy steps add vertices adjacent to every selected vertex, one at a time. (e) A swap replaces vertex 2 with vertices 4 and 5. (f) VNS for *index* = 1 shakes out one randomly selected vertex (Hansen et al., 2004). See the text for further details.

possible subproblems in a CSP, and then guides global search with the outcome of local search to solve the problem.

1.5 Related work

Research has long recognized the importance of structure to CSP search, for both propagation and decision making. Variable-ordering heuristics respond to structure detected in the constraint graph. A CSP whose graph is connected and acyclic (i.e., a tree) can be solved without backtracking when search enforces arc-consistency (Freuder, 1982; Mackworth and Freuder, 1985; Freuder, 1994). In a constraint graph, a *cycle cutset* is subset of variables whose removal results in an acyclic graph (Dechter and Pearl, 1987; Dechter, 1990). After a cycle cutset S is identified, the solver first solves the subproblem P' induced by S and then extends this solution of P' to the remaining tree-structured CSP, which will be backtrack-free when arc consistency is enforced. Because the solver still needs to solve P', the smaller the cycle cutset, the greater the performance improvement.

The identification of a minimal cycle cutset is itself NP-hard. While there are efficient approximations for minimal cycle cutset (Pearl, 1988; Dechter, 1990), cycle-ridden problems like those addressed in this dissertation have cycle cutsets far too large to provide practical and effective guidance.

Satisfiability problems are special CSPs, although for practical reason they are treated and solved differently. A *satisfiability problem* (*SAT*) is a CSP whose variables all have binary domain $D = \{0, 1\}$ (Cook, 1971). A *literal* is a boolean variable or the *negation* (logical NOT) of a boolean variable. A SAT constraint is a *disjunction*, a sequence of logical ORs of literals. A constraint is satisfied if and only if it evaluates to 1. A SAT problem asks whether there is a truth assignment for all the variables such that all its constraints (or equivalently the *conjunction*, logical AND of all its constraints) are satisfied. 2*SAT* (SAT with only binary constraints) can be solved in polynomial time, but *k*SAT for $k > 2$ is NP-complete. Like CSPs, the structure of SAT problems also affects search performance. SAT problems generated with unsatisfiable large cyclic cores have stumped many proficient SAT solvers (Hemery et al., 2006).

In addition to trees and acyclic graphs, other related work tries to use elaborate structural features to facilitate search. (Dechter and Pearl, 1989) decomposed a CSP into a tree of cliques of variables. This method first solved all the cliques and then solved the join tree, where each clique was treated as a singleton variable, in a backtrack-free manner. (Cohen and Green, 2006) generalized structural decomposition of *hypergraphs*, constraint graphs for non-binary CSPs, with *typed guarded decomposition*. This approach decomposed a CSP to *guarded blocks*, subproblems of the original CSP, that formed a join tree. Each guarded block is associated with a *type*, an algorithm that is able to solve

the block in polynomial time. The framework of typed guarded decomposition facilitates different ways to solve CSPs with limited interaction between typed and guarded blocks. (Samer and Szeider, 2006) studied the impact of various parameters on tractable CSPs that are tree-decomposable. (Gyssens et al., 1994) decomposed the hypergraph of a CSP into a join tree. Each node of the tree is a 1-*hinge*, a set of constraints. Every pair of adjacent nodes in the tree shares exactly one constraint. This work further generalized the decomposition with *k-hinge*, where adjacent nodes share *k* constraints. Other structural approaches include conversion of a CSP to a maximum clique problem through the CSP's microstructure (Jégou, 1993), conversion of a CSP to a maximal independent set problem (Gompert and Choueiry, 2005), offline compilation of CSPs into minimal synthesis trees for search (Weigel and Faltings, 1999), and comparison of various decomposition techniques on crossword puzzle problems (Zheng and Choueiry, 2005). These approaches, however, are primarily theoretical, or require considerable computational overhead unjustifiable on easy problems. All these approaches focus on the topological structure of constraint graphs of CSPs, but they all ignore the tightness of individual constraints, the significant way in which constraint graphs and ordinary graphs differ.

The key concept in this dissertation, *clusters*, describes heavily constrained, highly-interactive subproblems in a CSP. "Cluster" has been used elsewhere to describe aggregations of data. As a form of unsupervised learning, *clustering* groups nearby entities, based on some distance metric, together. Such clustering has been successfully apply clustering to human face comparison, gene identification, key sentence extraction, and search for easily accessible cities by air (Frey and Dueck, 2007). PageRank (Brin and

Page, 1998), initially introduced for Web search algorithms, is well defined on any graph. It captures structure relations between vertices and has been used for graph clustering (Chung and Tsiatas, 2010). In (Kroc et al., 2008), a cluster is a maximal group of neighboring solutions (in a 1-neighborhood) in the solution graph of a graph coloring problem. (van Dongen, 2000) uses "clusters" for relatively isolated, dense areas in a graph. (Razgon and O'Sullivan, 2006) contracted "clusters" (groups of variables) in a CSP so that the problem's constraint graph becomes acyclic (e.g., a tree, of clusters). That work offered no structural description or explanation and addressed only two classes of artificially-generated problems, much smaller than the artificial and real-world problems studied in this dissertation.

In this thesis, a *cluster graph* is the constraint graph of the subproblem induced by the variables in the detected clusters. It includes dense and tight subproblems in the constraint graph of the original problem. Variables and constraints not explicit in a cluster graph have nonetheless influenced its formation (via pressure, described in Chapter 3). Thus a cluster graph captures a kind of fail-first metastructure that anticipates and directs search attention to difficulties. This approach differs, therefore, from methods that relax, remove, or soften constraints. Clustering in unsupervised learning tries to group together vertices that are close, by some definition of pair-wise distance, while clustering for a CSP in this research seeks to group together variables that are more likely to reduce each other's domains through inference during search.

Contention was initially introduced by (Sadeh, 1991; Beck et al., 1997) to measure the extent to which activities compete for the same resource over the same time in jobshop scheduling problems. Contention has also represented areas where wipeouts

frequently occur during search for CSP solutions (Grimes and Wallace, 2007). The latter definition of contention shows the nature of all types of constraints during search. It is, therefore, more general than the earlier definition, which was only for constraints in a specific kind of CSP. In this research, we use the later definition of contention.

With respect to a given search algorithm, the *backdoor* of a CSP is a set of variables that, once assigned values, makes the remainder of search trivial (Willimans et al., 2003; Ruan et al., 2004). A backdoor is typically less than 30% of the variables in a problem, 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 et al., 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. Unlike (Junker, 2004; Hemery et al., 2006), cluster-based explanations are available whether or not a problem has a solution.

1.6 Summary

This chapter has defined constraint satisfaction problems and provided background knowledge for research on CSPs. This includes common global search, local search and inference algorithms to solve constraint satisfaction problems. The word "search" hereafter means global search unless otherwise specified. The local search metaheuristic Variable Neighborhood Search was emphasized because this research adapts it to solve CSPs. In addition, this chapter reviewed related work that uses structure to improve CSP search.

The novelty of this research is that we adapt a local search metaheuristic, Variable Neighborhood Search, to identify clusters, heavily constrained and highly-interactive subproblems within CSPs. We use various metrics, including constraint tightness and conflict-directed weights on constraints learned during probing, as the metrics to select variables to build clusters. Previous research has only used topological features, such as graph density, to build structure. Conflict-directed heuristics maintain and use constraint weights to guide search. They may use probing to improve the quality of constraint weights before search. The algorithms presented in this dissertation are the first to use either constraint tightness or constraint weights learned during probing for CSP structure detection. The next chapter introduces a novel visualization tool for binary CSPs, one that motivates and supports the research in this dissertation.

Chapter 2

Visualization for structured CSPs

Algorithms and heuristics to solve CSPs have drawn both inspiration and guidance from visualization of the problems (Cambazard and Jussien, 2006). For research on structured CSPs, any clue that suggests the unknown structure of the problem can be useful. Constraint graphs are abstractions of the original problems. They provide straightforward insight of their structure. Visualization of identified structure further supports the design and improvement of search heuristics that exploit such structure. Together, they improve users' understanding of the internal structure so that they may reformulate the problem if it proves too costly to solve. Here we present a tool, *DrawCSP*, which visualizes both the constraint graph and identified structure for binary CSPs. The structure observed in *DrawCSP* output inspired, supported and verified the design of the structure-guided search research presented in this dissertation. The first section introduces *DrawCSP* and the visualization of constraint graphs. The second section describes how to visualize identified structure within a CSP with *DrawCSP*.

2.1 DrawCSP visualizes binary CSPs

It is trivial to display the structure of a CSP as simple as that in Figure 1.1. As solvers tackle larger and more challenging problems with non-random structure, however, such

visualizations become more opaque. Figure 1.2(b), the constraint graph of a *Comp* problem when structural knowledge is given, is very different from Figure 1.2(a), the constraint graph for the same problem without that knowledge. Together they demonstrate the importance of structural knowledge and proper visualization.

As examples of real-world CSPs, consider the driverlog problems from the Third International Planning Competition (Long and Fox, 2002; Lecoutre, 2009). Each problem involves sets of drivers, trucks, locations, and packages. The goal is to deliver packages to different locations, and have the drivers and trucks finish at specified destinations. The traditional constraint graph in Figure 2.1(a), for example, plots 650 points for one driverlog problem along the circumference of a circle, and includes 17447 lines, each of which joins a pair of vertices. This picture offers little information about the structure of the problem. Figures 2.1(b)-2.1(d) are products of the program *DrawCSP*. They offer more striking and more useful visualizations.

DrawCSP is a small and portable CSP visualization program, written in C++ with OpenGL and the OpenGL Utility Toolkit (GLUT). *DrawCSP* accepts a CSP in XCSP format (Roussel and Lecoutre, 2008), which is essentially a variant of the Extensible Markup Language **(**XML). *DrawCSP* uses the XML parser from (Berghen, 2009). There are two drawing modes; the constraint graph mode that produced Figures 2.1(a), 2.1(b) and 2.1(d), and the adjacency matrix mode that produced Figure 2.1(c).

In the constraint graph mode, *DrawCSP* plots variables either on one circle or on two concentric circles. Edge colors vary from yellow to magenta as the tightness of the corresponding constraint increases, that is, darker edges represent tighter constraints. The two-circle arrangement displays odd-numbered variables on the outer circle and even-

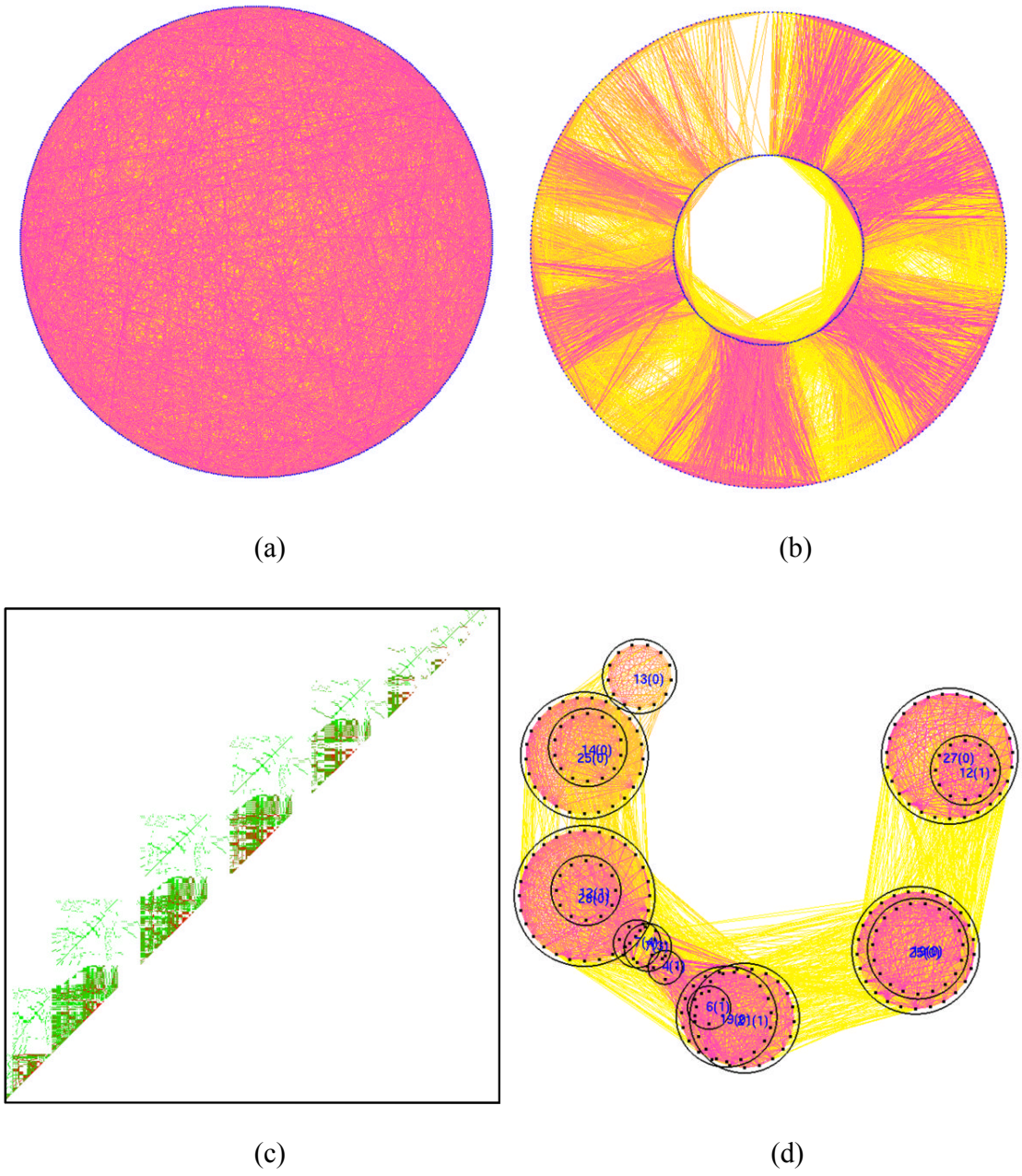
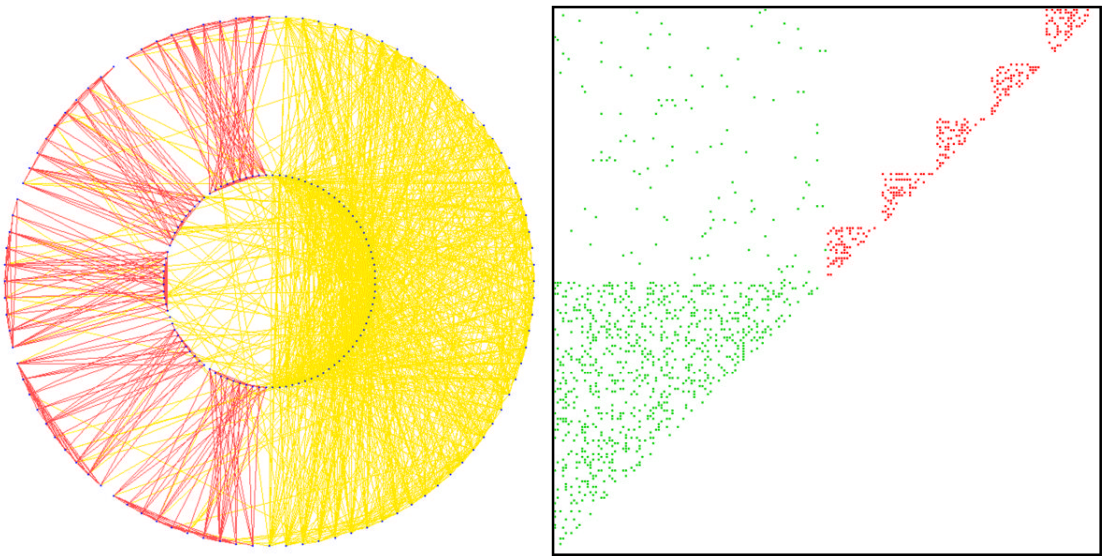


Figure 2.1. For the same driverlog CSP (a) an uninformative constraint graph plots variables on the circumference of a circle while (b) another constraint graph reveals its structure. (c) The adjacency matrix of the same graph (d) Subproblems, identified by local search, that significantly improve global search performance.



(a) (b)

Figure 2.2. *DrawCSP* displays a CSP from the class *Comp*: (a) the structure of the problem, with two concentric circles, showing a large, loose central component on the right and five small but tight satellites on the left (b) the adjacency matrix constraint graph shows the central component at the lower left, the satellites in red along the diagonal at the upper right and, in the upper left, the loose links from the central component to its satellites, which are not connected to each other.

numbered variables on the inner circle. It also sometimes displays constraints between geometrically close variables more effectively than the single circle arrangement. (See, for example, Figures 2.1(b) and 2.2(a).) The user can also manually move variables on the screen with a mouse, and variables' coordinates can be saved to files for future re-display. Figures 2.3(a-c) show an example of a black hole CSP (Gent et al., 2007) in single circle formation, in double circle formation, and the same CSP after manual rearrangement, which reveals a large tree structure formed by its tightest constraints. The purpose of manual variable rearrangement is to allow better visualization of the structure of the problem in case neither the single circle arrangement nor the double circle arrangement provides directly visible structure.

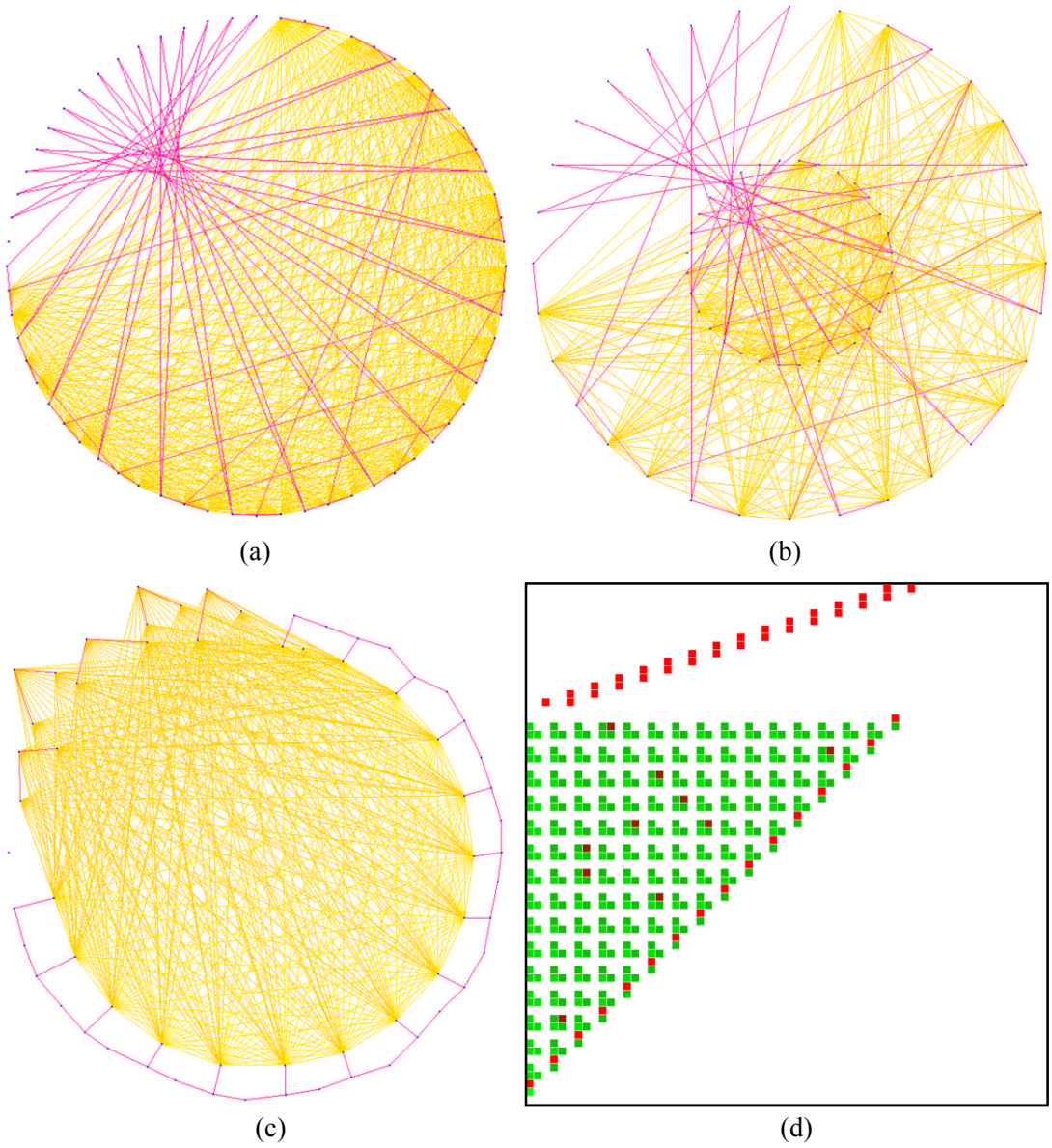
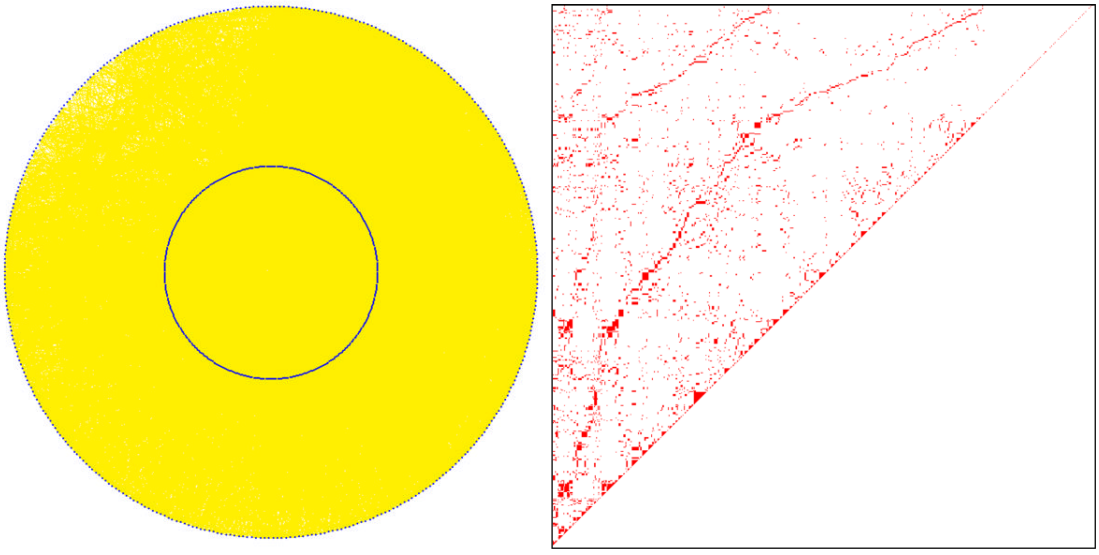


Figure 2.3. *DrawCSP* displays constraint graphs for BlackHole-4-4-e-0_ext.xml with variables arranged (a) on a single circle or (b) on two circles and (c) after manual arrangement to reveal its structure, which includes a large tree and 5 short paths of length 2. (d) BlackHole-4-4-e-0_ext.xml displayed by *DrawCSP*'s adjacency matrix mode.

In *DrawCSP*'s adjacency matrix mode, a constraint with scope variables numbered x and y, $x < y$, is displayed with coordinates (x, y). Here, for visibility, red points denote tighter constraints and green points denote looser ones. Manual rearrangement of constraints is



(a) (b)

Figure 2.4. Examples of DrawCSP (a) Constraint graph on two concentric circles for a large optical network problem and (b) its adjacency matrix.

not allowed in matrix mode. Although a constraint graph represents a CSP, it can be difficult to detect the actual structure when the number of constraints is large or the density of the CSP is high. Because constraints do not overlap on the adjacency matrix presentation, an adjacency matrix may better clarify relationships and thereby provide more structural information. Figure 2.3(d), the adjacency matrix of the same problem shown in Figures 2.3(a-c), shows the problem's large tree of tight constraints without any manual re-arrangement. Figures 2.4(a) and (b) are, respectively, the double-circle constraint graph and the adjacency matrix of a large optical network problem with 778 variables and 12876 constraints. No structure is visible in its constraint graph, but Figure 2.4(b) suggests that this problem has approximately three long paths through its variables.

One limitation of *DrawCSP* is that the layout of the displayed constraint graph relies upon the numbering of variables in the original problem. In Figure 2.2, had the problem

generator not numbered the central component variables from X_1 to X_{100} and the variables from the five satellites from X_{101} to X_{120}, X_{121} to X_{140}, X_{141} to X_{160}, X_{161} to X_{180}, and X_{181} to X_{200}, respectively, the problem's constraint graph or adjacency matrix as visualized by *DrawCSP* may have appeared different, and the structure of the problem may not have been directly visible. In fact, the constraint graph or the adjacency matrix of a problem will be visibly different after the numbers for its variables are shuffled. This limitation of *DrawCSP* demonstrates the importance of structure detection in search: perfect structural knowledge should not be relied upon, nor necessarily available from variable numbering. The key structure detection algorithms in this research (presented in Chapter 3) do not rely on any perfect knowledge, including variable numbering.

2.2 Visualization, identified structure, and search

Recall that for our purposes, a cluster is a heavily constrained, highly-interactive subproblem in a CSP. During search, our solver generates data that describes the number of clusters, their sizes and their member variables. *DrawCSP* visualizes these clusters from this data.

If *DrawCSP* is in constraint graph mode and a cluster formation file is provided, it draws only the clusters, with the radius of each cluster proportional to the number of variables it contains. (See Figure 2.5 for examples.) The center of each cluster's circle is the geometric center of all the vertices it includes, with their coordinates computed from what would have been their single-circle locations. Given a cluster Q with variables $\{X_1, X_2, ..., X_n\}$ whose coordinates are (x_1, y_1), (x_2, y_2), …, (x_n, y_n), respectively, the

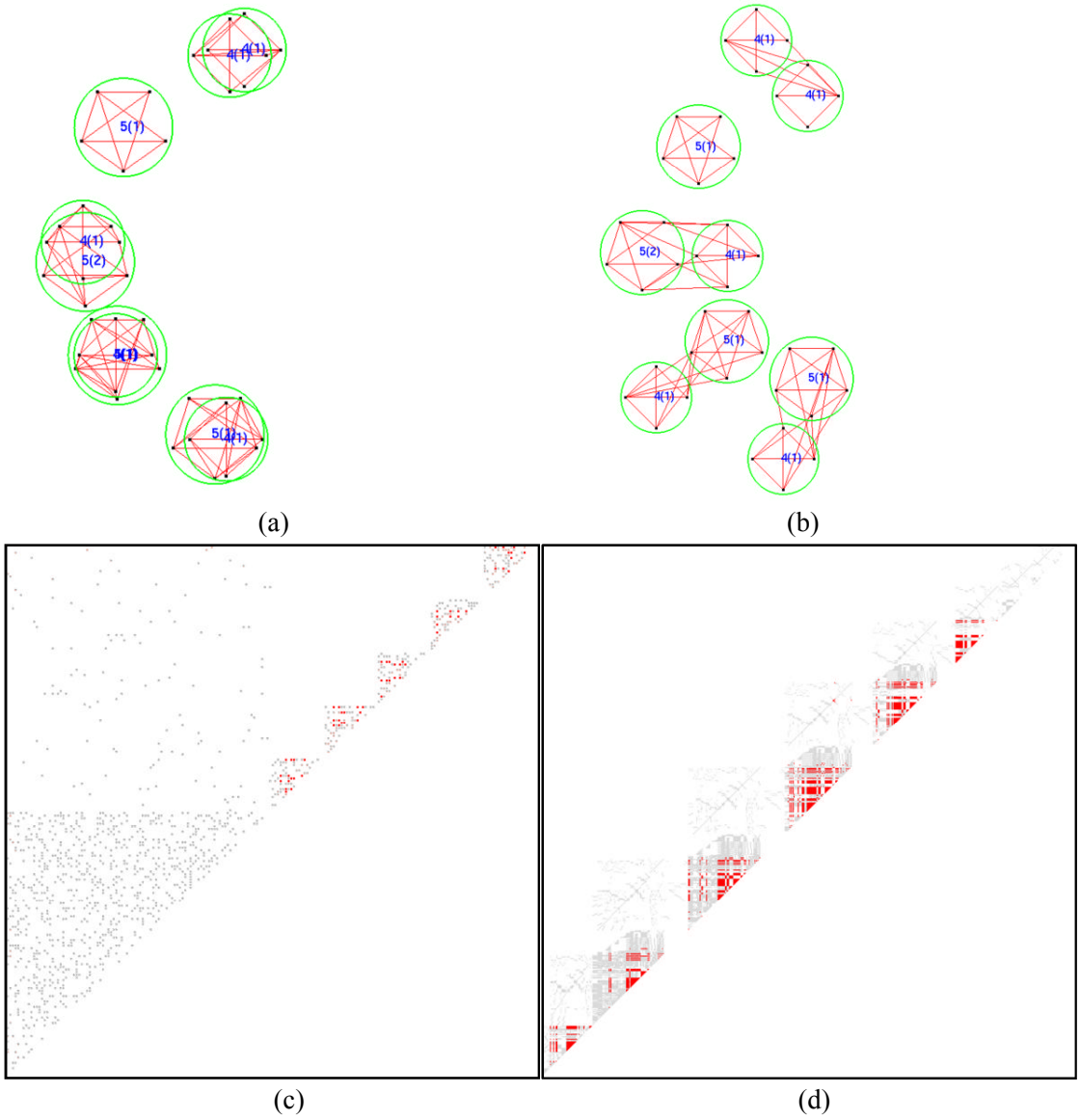


Figure 2.5. *DrawCSP* displays identified clusters: (a)-(c) for the problem in Figure 2.2, and (d) for the problem in Figure 2.1. Each cluster is circled; the label $x(y)$ denotes that it includes x variables and that it needs y additional edges to become a clique. (a) Direct output from *DrawCSP*, where the five groups of clusters correspond to the five satellites. (b) The same nine clusters, manually rearranged. (c) Cluster variables are shown in red in the adjacency matrix. The red dots correspond to the variables shown in (a) and (b). (d) Red dots correspond to the cluster variables shown in Figure 2.1(d).

coordinates of the geometric center of the circle that represents Q are $(\frac{1}{n}\sum_{i=1}^{n} x_i, \frac{1}{n}\sum_{i=1}^{n} y_i)$. The geometric center is selected to preserve the relative positions of the subproblems, as presented by clusters, in the entire CSP graph. In this way, the clusters shown in Figure 2.1(d), for example, closely represent their corresponding subproblems in Figure 2.1(b). Supported manual rearrangement of clusters includes translation and rotation. For example, Figure 2.5(b) is rearranged from Figure 2.5(a).

If *DrawCSP* is in adjacency matrix mode and a cluster formation file is provided, it plots constraints that are inside clusters in red and all out-of-cluster constraints in light grey. (See Figures 2.5(c) and (d) for examples.)

2.3 Summary

DrawCSP is a visualization tool for binary CSPs. Researchers can use it before search to display a problem's structure through its constraint graph. They can also use it after search to verify the structure that search just detected and used. *DrawCSP* can be used together with visualization tools such as CP-Viz (Simonis et al., 2010), which does postmortem analysis of the search tree and invariant validation of search states, to better understand how the problem is solved. Like CP-Viz, *DrawCSP* does not interact with search. It is, however, possible and interesting to combine *DrawCSP* and a CSP solver, in a passive way, to observe the process of search. The next chapter presents one of the two key contributions of this dissertation: the identification of structure in CSPs.

Chapter 3

Identification of structure

Typical variable-ordering heuristics prefer variables of high degree in the constraint graph. Heuristics that focus on these variables can be misled because difficult subproblems are not necessarily characterized by such variables. What is really needed is an effective reasoning mechanism that predicts and exploits the difficult subproblems, some of which may be overlooked by degree-based heuristics, such as *MinDomDeg*.

The presence of a backdoor in a CSP suggests that some portions of a problem are easy and that attention should, therefore, initially be directed to the remainder of the problem. The structure formed by these crucial sets of variables provides important information: search ordering heuristics, propagation methods, and an explicit representation of the most critical portions of the problem, which may have made it unsolvable. Identification of a backdoor, however, requires extensive exploration of the search space (Dilkina et al., 2007).

Clusters, a kind of structural knowledge identified before search, suggest a possible backdoor. Consider, for example, the driverlog problem in Figure 2.1. Its traditional constraint graph in Figure 2.1(a) shows little information about any possible structure. Figure 2.1(b) uses the variable-number assignment provided by the competition and plots the variables on two concentric circles. With the odd-numbered variables on the outer circle and even-numbered variables on the inner circle, it reveals roughly six major,

tightly-connected subproblems (magenta edges) with looser (yellow) connections between subproblems. Figure 2.1(d) shows the clusters detected by the work reported here, structure detected before search for a solution begins. Each cluster is drawn within a circle for clarity. There are 15 clusters, in 6 groups, which closely correspond to the 6 major subproblems in Figure 2.1(b). Including the time to detect clusters, search guided by this structural knowledge solves this problem in less than 40% of the time that two state-of-the-art adaptive heuristics, *MinDomWdeg* and *MaxWdeg*, use. On benchmark problems, such as the *Comp* instance in Figure 1.2 and Figure 2.2, the speedup of search, guided by the detected structure in Figure 2.5(a-c), over these two heuristics is more than an order of magnitude, while the traditional *MinDomDeg* fails to solve the problem within 30 minutes.

The two previous chapters introduced CSPs and showed their structures through the CSPs' constraint graphs. To find out whether structural knowledge is useful, the first section of this chapter explores available structural foreknowledge. The second section presents *Foretell*, the algorithm that adapts VNS to identify crucial structure. The last section shows some interesting structure identified by *Foretell*.

3.1 Exploitation of structural foreknowledge

This section explores the power of perfect foreknowledge about difficult subproblems to guide search. The approaches it tests are not ultimately allowable as variable-ordering heuristics because perfect foreknowledge is only available from the problem generator for artificially generated problems. Rather these heuristics gauge how well knowledge about structure supports search, and how best to use that knowledge. Results appear in

Table 3.1. All experiments reported in this dissertation were run in *ACE* (the Adaptive Constraint Engine), a constraint solver and test-bed for CSP search methods (Epstein et al., 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. All cited differences are statistically significant at the 95% confidence level under a one-tailed *t*-test.

Table 3.1 demonstrates how perfect foreknowledge about the structure of *Comp* problems speeds solution. Each variable-ordering heuristic had 30 minutes to solve each of 50 *Comp* problems (lines 1–3 in Table 3.1). Inspection indicates that *MinDomDeg* was immediately drawn to the central component of *Comp* because variables in the central component generally have larger degrees than those in the satellites. Since the links 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 every retraction, *MinDomDeg* repeatedly made the

Table 3.1. On 50 Comp problems, mean and standard deviation for nodes and CPU seconds, including time to find clusters. Search heuristics appear above the line. Search methods with perfect knowledge (below the line) are not legitimate heuristics because they apply structural foreknowledge available only to the problem generator, not the search engine. *Until*-11 is therefore only a target.

	Heuristic	**Time, μ (σ)**		**Nodes, μ (σ)**	
1	*MinDomDeg*	1728.157	(355.877)	285751.970	(61368.701)
2	*MaxWdeg*	123.000	(128.580)	20817.640	(22954.165)
3	*MinDomWdeg*	83.580	(38.964)	12519.360	(5811.370)
4	*satellite*	No problem solved			
5	*stay*	2.848	(3.584)	511.922	(416.345)
6	*until* -11	1.612	(1.866)	398.776	(244.112)

same mistake: it tried to solve the central component first, although the true difficulties lie elsewhere, in the satellites. As a result, *MinDomDeg* solved only 2 of 50 problems within the time limit. Both learning heuristics, *MaxWdeg* and *MinDomWdeg,* initially suffered from the same attraction to the central component. They eventually recovered and solved all the problems through learning, which would have been unnecessary had they attended initially to structure and constraint tightness. Learning lacks the foresight clusters are intended to provide.

Now consider how heuristics might exploit perfect foreknowledge about the problem. Assume one was given the structure shown in Figure 1.2(b), and believed that the satellites contained the backdoor. The variable-ordering heuristics in lines 4-6 of Table 3.1 were designed to exploit the satellites. In that case, preference for satellite variables should speed search. Rather than discard traditional variable-ordering heuristics, however, each approach investigated here makes satellites a priority and then breaks ties with *MinDomDeg.* Each approach was given 30 minutes to solve each problem.

The variable-ordering heuristic *satellite* examines whether mere presence in a satellite is sufficient to warrant prioritization. On a *Comp* problem, this approach selects all 100 satellites variables first, in random order, and then orders the remainder with *MinDomDeg*. Its failure (line 4) implies that mere presence in a satellite is insufficient to warrant prioritization. (Given its lack of promise, this is the only randomized heuristic that was tested only once. All other non-deterministic experiments here report on an average of 10 runs.) The variable-ordering heuristic *stay* addresses entire satellites, in a random order, one at a time before it selects any variable in the central component. *Stay*

uses *MinDomDeg* within a satellite and within the central component. Line 5 in Table 3.1 shows the dramatically improved results. Guided by the satellites, *stay* with *MinDomDeg* requires only about 4% of the time and nodes used by *MinDomWdeg*.

Given that noteworthy improvement, and the fact that the satellites may only estimate the backdoor of a *Comp* problem, the next approach binds only some of the variables in each satellite. If MAC-3 is in use, for example, there would appear to be little point in "finishing" a satellite once it is reduced to only a pair of variables (with at most a single edge between them); *until*-2 selects a different satellite at that point. The generalization of this approach, *until-i*, instantiates variables within a randomly chosen satellite until all but *i* variables have been bound, and then moves on to another randomly chosen satellite. (*Stay* is equivalent to *until*-0.) Within a selected satellite and later, within the central component and any "leftover" satellite variables, *until-i* also uses *MinDomDeg.* We tested a range of values: $i = 2, 3, \ldots, 15$.

Surprisingly, search need not stay long in a given satellite. For *Comp,* where the satellites are of size 20, the clear winner was *until*-11, that is, search can address as few as 40% of the variables in a satellite before safely moving on to the next one. In contrast, the variable-ordering heuristic *satellite-i,* which randomly chooses satellite variables that are not among the last *i* future variables in their satellite, performed poorly. (Data omitted.) As *i* increases, *satellite-i* behaves more like *MinDomDeg* alone does. (Section 6.3 considers why $i = 11$ was so successful.)

Clearly, structural foreknowledge is critical to search performance here. Known satellites dramatically improve the solution of *Comp* problems when search addresses

them one at a time. The following sections describe how knowledge about such crucial substructures can be detected automatically, prior to search.

3.2 Identification of clusters with *Foretell*

Foretell assembles sets of closely related variables whose domains are likely to be reduced by each other during search. Typically these clusters are cliques or near cliques. (Intuitively, a *near clique* is a clique with a few missing edges. A more formal definition appears in Section 4.2.) *Foretell* was inspired by VNS' state-of-the-art speed and accuracy on the DIMACS maximum clique problem (Hansen et al., 2004).

Recall that VND greedily applies a metric (variable degree) to find a maximum clique. *Foretell* instead uses a metric that forms large, tight, heavily-connected subsets of variables. This section describes three metrics, and the three variants of *Foretell*. The first uses static information; the next two use knowledge learned from search.

3.2.1 *Foretell* based on tightness

The tightness of a constraint relates closely to how often domain reduction actually occurs along that constraint during search. Based on tightness, we can define the *pressure* on a variable X_i to be the probability that, given all the constraints upon it, when one of X_i's neighbors is assigned a value, at least one value will be excluded from X_i's domain.

$$\text{pressure}(X_i) = \text{P}\left(\text{Num}(\text{domain-reduction}_{X_i}) \geq 1 \;\middle|\; \text{some neighbor assignment}\right) \qquad [7]$$

To avoid the expensive calculation of exact pressure, we approximate variable pressure as the probability that exactly one value is removed from this variable's domain due to neighbor assignments. For variable X_i with domain size $|D_i|$, neighbors N_i and a constraint with tightness t_{ik} between X_i and $X_k \in N_i$, the approximate pressure pressure′(X_i) on X_i given the constraints on it, is

$$\begin{aligned}\text{pressure}'(X_i) &= \mathrm{P}\left(\text{Num}(\text{domain-reduction}_{X_i}) = 1 \;\middle|\; \text{some neighbor assignment}\right) \\ &= \sum_{X_k \in N_i} \frac{\begin{pmatrix} (|D_i|-1)\cdot|D_k| \\ (1-t_{ik})|D_i|\cdot|D_k| \end{pmatrix}}{\begin{pmatrix} |D_i|\cdot|D_k| \\ (1-t_{ik})|D_i|\cdot|D_k| \end{pmatrix}}\end{aligned} \quad [8]$$

The denominator in equation [8] counts the number of different possible constraints on X_i and its neighbors given their domains and the tightness t_{ik} on scope $\{X_i, X_k\}$ for some neighbor X_k of X_i. The numerator repeats this computation but assumes X_i's domain has been reduced by one value. The fraction is thus the probability that the constraint C_{ik} is still satisfiable after X_i's domain has been reduced by one value. To speed computation, we estimate the probability that X_i's domain is reduced by one value as the result of some neighbor's assignment as the sum of the probability that X_i's domain is reduced by one value across all X_i's neighbors.

We further adjust equation [8] to avoid bias in favor of variables with high degrees:

$$p(X_i) = \frac{1}{\text{degree}(X_i)} \sum_{X_k \in N_i} \frac{\begin{pmatrix} (|D_i|-1)\cdot|D_k| \\ (1-t_{ik})|D_i|\cdot|D_k| \end{pmatrix}}{\begin{pmatrix} |D_i|\cdot|D_k| \\ (1-t_{ik})|D_i|\cdot|D_k| \end{pmatrix}} \quad [9]$$

Foretell relies on the approximate pressure $p(X_i)$ in equation [9] (instead of degree, as

Table 3.2. Examples of estimated pressure on variable X_i with two different senarios.

	t_{ij}	t_{ik}	$p(X_i)$
Scenario 1	0.3	0.3	$p(X_i)=\frac{1}{2}\left(\frac{\binom{90}{70}}{\binom{100}{70}}+\frac{\binom{90}{70}}{\binom{100}{70}}\right) \approx 1.74\times10^{-6}$
Scenario 2	0.3	0.7	$p(X_i)=\frac{1}{2}\left(\frac{\binom{90}{70}}{\binom{100}{70}}+\frac{\binom{90}{30}}{\binom{100}{70}}\right) \approx 1.15\times10^{-2}$

maximum clique search did in Section 1.4.3) to greedily select variables. For example, let variable X_i have two neighbors X_j and X_k and let $D_i = D_j = D_k = 10$. Table 3.2 shows the estimated pressure, $p(X_i)$, on X_i with two scenarios, identical but for the tightness on C_{ik}. In this example, when t_{ik} is higher, the pressure on X_i becomes significantly greater.

Foretell defines its own score function for VNS: the *tightness* of a subproblem $P' = <X', D', C'>$ is the ratio of the product of the subproblem's size $|X'|$ and density $d(P')$ to the sum of the relevant domain products of its constraints:

$$tightness(P') = \frac{|X'|d(P')}{\sum_{C_{ij}\in C'}\left(|D_i||D_j|\right)} \qquad [10]$$

Note, for example, the missing edges in the clusters of Figure 2.5(a). Using the same example from Table 3.2 and assuming X_i, X_j and X_k form a cluster and there is no constraint between X_j and X_k, [10] computes this cluster's tightness, *tightness*(P'), as $\frac{3}{10^2+10^2}\frac{2}{3} = 0.01$. This *tightness* function on a subproblem serves as the score function on line 7 of the VNS algorithm in Figure 1.6 and on line 6 of the VND algorithm in Figure

1.7. Tightness is defined only for a subproblem with at least three variables. *Foretell* only returns as clusters subproblems with defined tightness. A parameter, *cutoff*, determines how much time *Foretell* may devote to the detection of a single cluster.

To find multiple clusters in a problem, *Foretell* finds a first cluster, removes those variables and all constraints whose scopes include them, and then iterates to find the next cluster among the remaining variables and their constraints. To select variables for a cluster, ties on maximum pressure are broken by maximum degree, and then, if need be, at random. Clusters are typically (but not always) detected in decreasing size order. Because this is local search, some variation is expected from one pass to the next. The maximum neighborhood index was taken from the original work on maximum cliques: the minimum of 10 and the current cluster size (Hansen et al., 2004).

3.2.2 *Foretell* based on edge weights

The variable-ordering heuristic *MinDomWdeg* (introduced in Section 1.3.2) learns to guide search into areas where wipeouts are more likely to happen. Here the weights that *MinDomWdeg* uses in learning are called *edge weights* (Boussemart et al., 2004). Every constraint has an edge weight, which is initialized to 1. *MinDomWdeg* maintains edge weights during inference. When the assignment of variable X_i causes a wipeout on variable X_j through constraint C_{ij} with scope (X_i, X_j), the edge weight ew(C_{ij}) on C_{ij} is incremented by 1. Intuitively, *MinDomWdeg* increases the edge weight between the direct culprit X_i and the direct victim X_j of the latest wipeout. The *weighted degree* of a variable X_i is then defined as

$$wdeg(X_i) = \sum_{X_j \in N_i \text{ and } X_j \text{ is unassigned}} \text{ew}(C_{ij}) \qquad [11]$$

A wipeout indicates the conflict between the culprit and the victim. A constraint with a high edge weight or a variable with a high weighted degree is *contentious*. More broadly, an area in the constraint graph of a problem where variables with high weighted degrees gather is also *contentious*. Because there is little contention at the beginning of search, the weighted degree of a variable is essentially the variable's degree before any wipeout happens in the variable's vicinity. Therefore, early variable decisions made by *MinDomWdeg* are similar to those made by *MinDomDeg*. Thus *MinDomWdeg* makes the most important choices (i.e. the first few variable selections) with hardly any learned knowledge (Grimes and Wallace, 2007). To overcome this limitation, Grimes and Wallace introduced *probing*, a sequence of short and quick restarts that explores the search space before an unlimited search for a solution. During probing, edge weights are carried over from one restart to the next.

On artificially generated CSPs with random structure, two probing strategies appear to be effective: random probing and weight freeze during search. *Random probing* updates edge weights but selects variables randomly during probing. If edge weights direct heuristics (as, for example, they do in *MinDomWdeg*), contentious areas are more likely to be visited, which could in turn further increase the contentiousness of the same area. Random probing eliminates the bias of weight learning during probing, and thereby reduces positive feedback in some local contentious areas. After random probing, learned weights are no longer updated (*weight freeze*) during search. Weight freeze protects the global contention learned in random probing from bias produced in search.

Grimes and Wallace's work shows that there exist some contentious areas, in the constraint graph, where domain wipeouts are more likely to happen. The distribution of

contention in the constraint graph is represented by the set *EW* of edge weights on all constraints. The same distribution can also be equivalently represented by the set *WD* of weighed degrees on all variables by using equation [11]. *MinDomWdeg* uses WD to greedily select variables: it always chooses the variable with the highest weighted degree.

Assume edge weights are frozen after probing, so that *MinDomWdeg* uses the same static WD for all variable decisions. Assume WD includes two contentious sets of variables, A and B, such that variables in the same set are densely connected by constraints with high edge weights, and variables between the two sets are loosely connected by constraints with low edge weights. Let variable $X_i \in A$ have the highest weighted degree among all variables in the problem, and let $X_j \in B$ have the second highest weighted degree. Then *MinDomWdeg* selects and assigns X_i first as search begins. For the next variable decision, *MinDomWdeg* chooses X_j and thus forces search to hop from one contentious area to the other, which caused the difficulties for the heuristic *satellite* in Table 3.1. Indeed, the only information available to *MinDomWdeg* is a list of variables ordered by their *weighted degrees*; it is entirely unaware of the kind of structural knowledge detected by *Foretell*.

Because X_i and X_j are in different contentious areas, the assignment of a value to X_j after X_i is less likely to cause a domain wipeout than the assignment of another variable in the same contentious set A with X_i. Hopping from one contentious area to another may postpone domain wipeouts, counter to the fail-first strategy. Therefore, although *MinDomWdeg* greedily selects contentious variables, it does not strictly follow the fail-first principle. To achieve more domain wipeouts in situations similar to the above, we need to use structural knowledge.

Foretell-EW adapts *Foretell* to use the distribution of contention to detect structural knowledge. Before identifying clusters, *Foretell-EW* probes the problem to learn edge weights. Then, instead of relying on pressure as the metric to greedily select variables, *Foretell-EW* uses the weighted degrees of variables as the metric to select variables for clusters. Unlike *Foretell*, *Foretell-EW* does not consider either pressure or constraint tightness. *Foretell-EW* produces densely connected clusters, each of which represents a set of variables with high contention among then.

3.2.3 *Foretell* based on influence weights

Consider the situation during search where the assignment of variable X_i invokes inference to maintain arc consistency, and during this inference, variable X_k has its domain wiped out by the constraint C_{jk}. *MinDomWdeg* treats variable X_j as the culprit, so the edge weight on constraint C_{jk} is incremented. That is one explanation for the contention. Another explanation, however, is that X_i is the true, if more remote, culprit since the domain wipeout of X_k was a consequence of X_i's assignment. Figure 3.1 illustrates the difference between the two explanations.

Formally, we associate an integer *influence weight* with each constraint, a new conflict-directed metric. The influence weight of a constraint is initialized to 1. During search, if the assignment of X_i after inference causes a wipeout at X_j, the influence weight iw(C_{ij}) of the constraint C_{ij} is incremented by 1. If X_i and X_j have no constraint between them, an *invisible edge* IC_{ij} between them is created and its weight is initialized to 2. At any time during search, the *influence degree* of a variable X_i is the sum of the influence weights of all constraints and all invisible edges whose scopes include X_i and some future variable, as shown in [12].

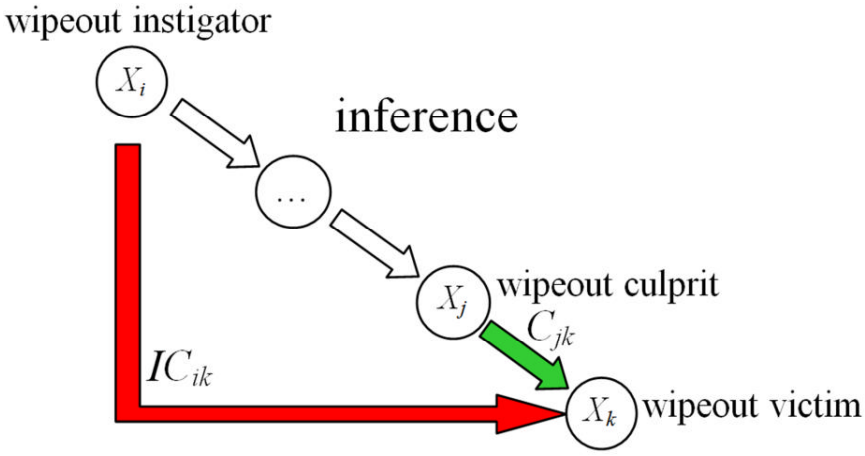


Figure 3.1. Inference invoked by the assignment of X_i causes a domain wipeout of X_k. *MinDomWdeg* increases the *edge weight* on constraint C_{jk}, the constraint between the direct culprit and the victim of this wipeout. *MinDomInfdeg* increases the *influence weight* on constraint C_{ik}, the constraint between the initial culprit and the victim of the wipeout. An *invisible edge* IC_{ik} is created if there is no constraint between X_i and X_k. Constraints are not directional. The arrows indicate only the direction in which inference proceeds.

$$infdeg(X_i) = \sum_{X_j \in N_i \text{ and } X_j \text{ is unassigned}} \text{iw}(C_{ij}) + \sum_{X_j \in N_i \text{ and } X_j \text{ is unassigned}} \text{iw}(IC_{ij}) \qquad [12]$$

During structure detection, invisible edges are treated like constraints. During search, the new variable-ordering heuristic *MinDomInfdeg* uses the influence weights of variables to guide search: it minimizes the ratio of dynamic domain size to influence degree. Variables with higher influence degrees instigate wipeouts on other variables more frequently. Invisible edges are not used during inference.

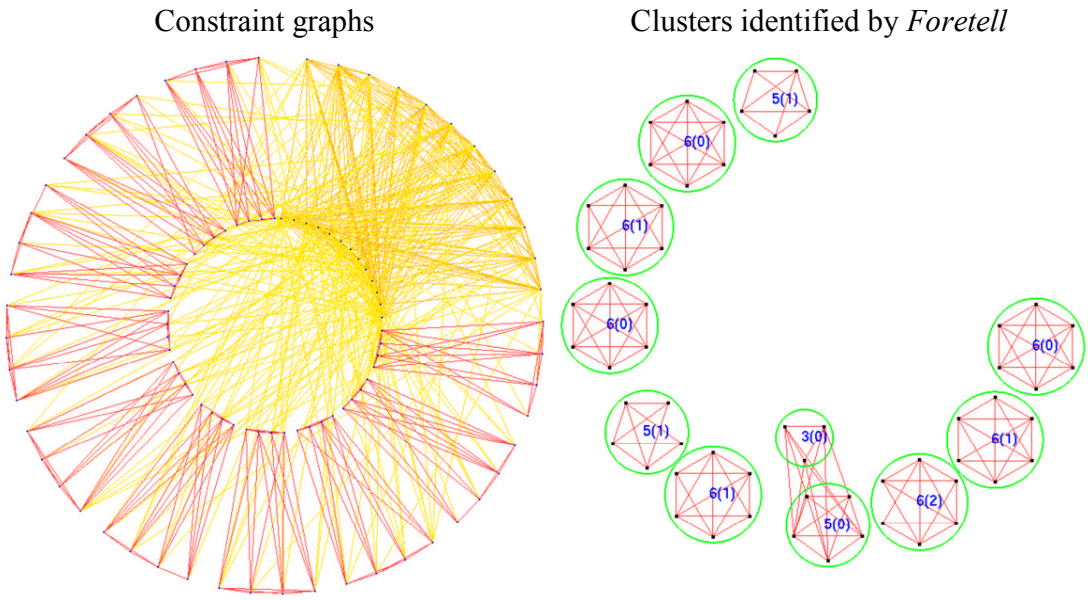
Foretell-IW adapts *Foretell* to use influence weights to detect structural knowledge. Similar to *Foretell-EW*, *Foretell-IW* first probes the problem before search to calculate influence degrees and then uses them to greedily select variables. While *Foretell-EW* captures contentious structure that is directly related to domain wipeout, *Foretell-IW* tries to include the instigators of domain wipeouts in the structure.

3.3 Cluster structure identified in CSPs

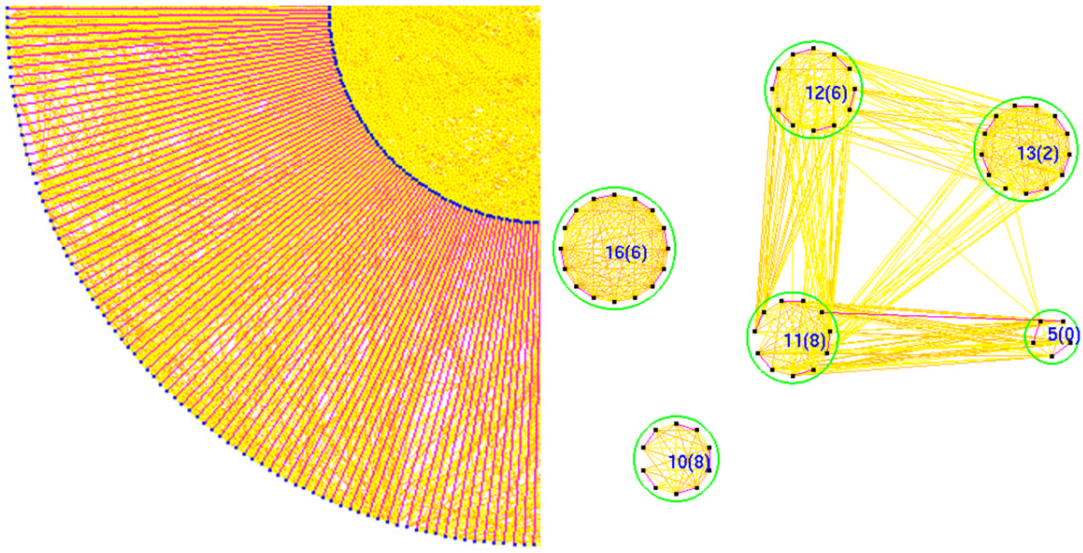
We apply *Foretell* to benchmark problems from (Lecoutre, 2009). A *cluster graph* is the constraint graph of the subproblem induced by the variables in detected clusters. Under the same time cutoff, the cluster graphs produced by different runs on the same CSP are extremely similar to each other, despite *Foretell*'s non-determinism. For different kinds of CSPs, however, the cluster graphs are quite different. Figure 2.1 and Figure 2.5 showed the *Foretell*-detected structure of a driverlog problem and a *Comp* instance. Figure 3.2 shows two more examples of structure detected in constraint graphs.

Figure 3.2(a) is for a composed CSP from the class <25,10,0.667,0.15.>10 <8,10,0.786,0.50> 0.01, 0.05 (designated 25-10-20 by (Lecoutre, 2009)). Its satellite density (0.786) is higher than that of Comp, however. That encourages the formation of somewhat larger clusters, and typically leaves behind too few edges to form a second cluster in the same satellite. As a result, all but one satellite is represented by a single cluster, and clusters in different satellites are isolated from one another.

Clusters edges may also be predominantly loose constraints, as in the cluster graph for rlfap_scene11.xml, scene 11 of the radio link frequency problems (Cabon et al., 1999). The Radio Link Frequency Assignment Problem (*RLFAP*) involves hundreds of radio broadcasting facilities with limited broadcasting frequencies. If the frequencies of two geographically close broadcasting facilities are too close numerically, their signals will interfere with one another and communication will be distorted. Thousands of such pairs of broadcasting facilities in France are susceptible to interference. An RLFAP problem represents radio broadcasting facilities as variables, and restricts their frequencies with constraints between pairs of variables. The domain of an RLFAP variable is the set of



(a) composed-25-10-20-1_ext.xml



(b) rlfap_scene11.xml

Figure 3.2. Constraint graphs (left) and identified structures (right) for two problems. Clusters in the graphs on the right are circled for clarity. The label $x(y)$ indicates that a cluster includes x variables and requires y additional edges to make it a clique. For detail, only the lower-left quarter of the constraint graph for rlfap_scene11.xml is shown.

radio frequencies that a facility can be assigned. A solution of an RLFAP CSP is equivalent to the establishment of interference-free communication. Scene11 is the most difficult solvable RLFAP problem. This problem's 4103 constraints vary dramatically in their tightness. There is a tight constraint between every pair X_i and X_{i+1} where i is an even number, $0 \leq i \leq 678$. These tight constraints form a bipartite graph, so that the double-circle constraint graph resembles a sun with rays of light, one quarter of which is shown in Figure 3.2(b). Its clusters are connected to one another primarily by loose constraints; its variables remain bipartite on tight constraints. The cluster graph of this 680-variable problem includes only 67 variables, and, as the next chapter shows, makes rapid solution possible.

3.4 Summary

This chapter has introduced the structure detection algorithm *Foretell*, which adapts VNS to find crucial structure in CSPs. Motivated by the possible performance improvement from perfect structural foreknowledge, we presented three different structure-detection methods. *Foretell* uses constraint tightness to identify dense and tight structure. *Foretell-EW* and *Foretell-IW* use a probing stage to learn weights that represent the distribution of contention in a problem and then infer structure from those weights instead of tightness. *Foretell-EW* focuses on contention that directly causes domain wipeout. *Foretell-IW* focuses on contention that indirectly causes domain wipeout. Table 3.3 summarizes the three different versions of *Foretell* introduced in this chapter. With structure detected, we are now ready to exploit it during search for a solution.

Table 3.3. Differences among *Foretell*, *Foretell-EW* and *Foretell-IW* in the three steps of search

	Tightness-based structure detection	**Weight-based structure detection**	
Steps in search	*Foretell*	*Foretell-EW*	*Foretell-IW*
1. Preparation for search	Compute *pressure* for every variable based on constraint tightness	Probe to collect edge weights on constraints	Probe to collect influence weights on constraints
2. Structure detection metrics used to select variables to build clusters	*pressure*	Weighted degree (*wdeg*)	Influence degree (*infdeg*)
3. Structure-guided search	To select variables, search first uses a cluster-based heuristic to select a cluster (described in Chapter 4) and then uses a variable-ordering heuristic to select variables inside the selected cluster		

Chapter 4

Search guided by identified structures

The primary question now becomes how best to exploit clusters once they have been identified. Is it, for example, better to solve them one at a time, or to allow a shift to another before the current cluster is entirely bound? And if one at a time, in what order should the clusters be visited? The first section of this chapter seeks to exploit, in different ways, clusters detected automatically by the structure identification algorithms shown in the previous chapter.

Inference may reduce the search space because it maintains the consistency of the domains of constrained variables dynamically. More inference, however, does not always result in more domain reduction — it is sometimes faster to risk and retract from inconsistency than to anticipate it. Given the learned structure of the problem and our premise that clusters are more important than the rest of the problem during search for a solution, the second section of the chapter explores how structure-based inference might spend time more wisely than traditional inference methods.

4.1 Cluster-based variable-ordering heuristics

When *Foretell* returns a set of clusters, the solver needs some metric to exploit them in an orderly manner. Following the fail-first principle, one would ideally address the cluster

that at the moment is the tightest. The true *dynamic tightness of a cluster* $Q \subseteq X$ is the ratio of the number of tuples that do not satisfy its unbound variables under the current partial instantiation to the product of their dynamic domain sizes:

$$dynamic\text{-}tightness(Q) = 1 - \frac{\left|\text{satisfying-assignments}(Q)\right|}{\prod_{X_i \in Q} \left|\text{original-domain}(X_i)\right|} \qquad [13]$$

[13] cannot be obtained until all the solutions of the cluster are found and is thus too expensive to compute. The dynamic tightness of a cluster Q is estimated here as the ratio of the product of the current domain sizes of those variables to the product of their original domain sizes:

$$estimated\text{-}dynamic\text{-}tightness(Q) = 1 - \frac{\prod_{X_i \in Q} \left|\text{dynamic-domain}(X_i)\right|}{\prod_{X_i \in Q} \left|\text{original-domain}(X_i)\right|} \qquad [14]$$

According to [14], the estimated-dynamic-tightness of a cluster is in [0, 1]. The larger the estimated dynamic tightness, the smaller the search space of the cluster and, therefore, the more likely it is to have no solution.

We postulate several variable-ordering heuristics here. The heuristic *tight* selects a variable from the tightest cluster. Search guided by *tight,* however, could shift from one cluster to another, and therefore from one subproblem (e.g., a satellite in *Comp*) to another, the way the poorly-performing *satellite* did. The improvement produced by *stay* in Table 1.1 therefore inspired heuristics that treat one cluster at a time. The heuristic *concentrate* chooses a cluster at random, selects variables from it until all of them are bound, and then selects the next cluster at random. In contrast, *Focus* selects the (estimated) dynamically tightest cluster, selects variables from it using a traditional variable-ordering heuristic (e.g., *MinDomWdeg*) until all of them are bound, and then

uses estimated dynamic tightness to select the next cluster. Note that *tight*, *concentrate* and *Focus* only select clusters (groups of variables), not variables. A traditional variable-ordering heuristic is required to select variables within the selected cluster of interest. The heuristics *concentrate-i* and *focus-i* are analogous to *until-i* in Section 3.1; they instantiate within a cluster until all but *i* of its variables have been bound before search can move to a different cluster or bind any non-cluster variables. In all these heuristics, if clusters have the same maximum tightness, ties are broken by maximum dynamic cluster size, the number of unbound variables in a cluster.

In the experiments in this section, each heuristic had 30 minutes to solve each problem in a class of 50 mixed solvable and unsolvable *Comp* problems. Data for all non-deterministic algorithms, including those involving clusters, is reported as an average across 10 runs. For the heuristics that use cluster detection, time cutoff *e* is allocated to VNS per cluster. Thus a problem in which *s* clusters were detected could require up to *se* time. (Because elapsed time is tested only at the end of a loop iteration, it is possible to slightly exceed *se* in practice.) The total VNS time required to find clusters is included in all search time data.

On *Comp* problems, *Foretell* found clusters that form a structure remarkably like foreknowledge. It found between 6 and 19 clusters per problem, all of sizes 3 to 6. It found at least one cluster in every satellite in every problem on every run. A typical result appears in Figure 2.5(a) and (b). With $e = 0.2$ seconds per cluster, VNS search time averaged 2.11 seconds per problem, 49% of the total time.

Table 4.1. By more than an order of magnitude, cluster-guided search speeds up traditional heuristics that were allocated 30 minutes to solve 50 *Comp* CSPs. Average and standard deviation are shown for nodes and time in CPU seconds, including time for cluster detection. Data above the line is repeated from Table 3.1. Except for *MinDomDeg*, every method solved every problem. *Focus* is statistically significantly better (in **bold**) than all the heuristics tested. Recall that *until*-11 is a target, not a legitimate heuristic; it applies perfect foreknowledge about structure available only to the problem generator, not to the search engine.

Heuristic	Time, μ (σ)		Nodes, μ (σ)	
MinDomDeg	1728.16	(355.88)	285751.97	(61368.70)
MaxWdeg	123.00	(128.58)	20817.64	(22954.17)
MinDomWdeg	83.58	(38.96)	12519.36	(5811.37)
until-11	1.61	(1.87)	398.78	(244.11)
tight	4.71	(6.25)	505.30	(718.03)
concentrate	5.46	(5.63)	836.43	(876.54)
Focus	**4.31**	(2.41)	**497.96**	(324.33)
focus-1	5.27	(3.22)	516.41	(425.74)
focus-2	8.71	(22.44)	1371.34	(2765.68)

Clusters guide search in *Comp* effectively, as shown below the line in Table 4.1. *Concentrate*'s weaker performance clearly indicates that the order in which clusters are addressed is important. Unlike *stay*, however, *Focus* appears to need to finish a cluster to produce its best performance. Essentially, by $i = 3$, both *concentrate-i* and *focus-i* deteriorate to *MinDomWdeg*. (Data omitted.) A full graphic comparison (Figure 4.1) indicates that even *focus*-2 and *concentrate*-2 solve most *Comp* problems far more quickly than the learning heuristics *MaxWdeg* and *MinDomWdeg* do alone. Also, *concentrate* solves more *Comp* problems than *tight* in 45 seconds or less. One possible explanation is that in these CSPs *Foretell's* clusters are all of roughly equal importance, so that *Concentrate* benefits from a refusal to shift from one cluster to another. As more

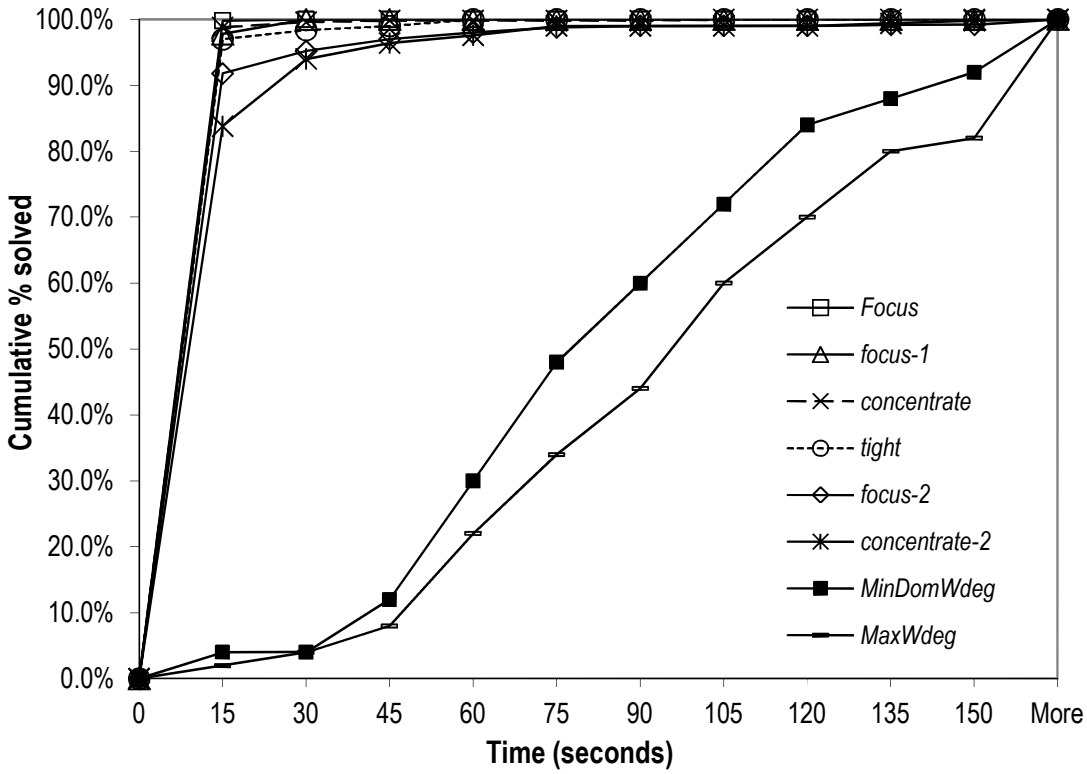


Figure 4.1. Cumulative percentage of 50 *Comp* problems solved. Solvers were allocated 30 minutes per problem. Because it solved only 2 of the problems, *MinDomDeg* was omitted.

clusters are solved, however, it becomes important to instantiate within tight clusters, so that *tight* would then have an advantage. *Focus* combines the best of both approaches.

Given the non-determinism of local search, one cannot expect VNS to produce an adequate set of clusters every time. Rather than allot substantial time to VNS (which should ultimately find adequate clusters that way), we used *MinDomWdeg* to select individual variables within a cluster. *MinDomWdeg* is slightly slower than *MinDomDeg* at selecting variables inside a cluster, but it also provides backup if *Foretell*'s local search is simply "unlucky." Learning is there to help, although it is rarely necessary.

4.2 Cluster-based inference

Structural knowledge also provides information that can be harnessed to guide inference. Inference methods can be characterized along a spectrum by the effort they expend. Stronger inference usually consumes more time, but it does not guarantee more domain reduction. There is a trade-off between the time that inference could save and the time that inference itself consumes. Sometimes it is faster to recover from a mistake than to anticipate it. Inference after every assignment, as in Algorithm 1 (Figure 1.4), is called *consistency maintenance.* FC and MAC-3 (as described in Section 1.3.1) are commonly used to maintain consistency. FC enforces a lower level of consistency by only propagating the effect of a variable assignment to its neighbors. MAC-3 does considerably more work to propagate that effect to the entire problem. *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 (100*k*)% of their values after each instantiation. (The R is for "response.") Intuitively, higher values for *k* make ACR lazier. ACR behaves more and more similarly to MAC-3 as *k* becomes smaller.

With the structural knowledge detected by *Foretell*, it becomes possible to fine tune consistency enforcement. *Cluster-based inference* considers where other variables lie with respect to the clusters. Each cluster Q in problem $P = \langle X, D, C \rangle$ delineates a *fringe* (variables in $X - Q$ within *width* edges of some variable in Q), and an *outside* ($X - Q - fringe(Q)$), as shown in Figure 4.2. The question then becomes how to select propagation methods for the cluster, the fringe, and the outside.

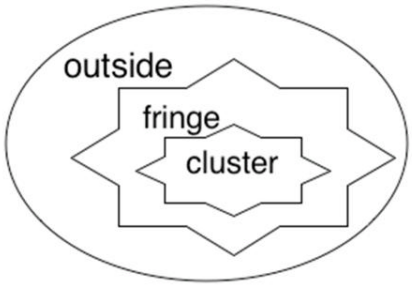


Figure 4.2. Propagation regions delineated with respect to a cluster.

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 formally defined recursively as follows:

- A clique on 3 vertices K_3 is a near clique.
- Given a near clique $NC = \langle V, E \rangle$ with missing edges $m = \frac{|V|(|V|-1)}{2} - |E|$

a new near clique $NC' = \langle V \cup \{v\}, E \cup \{e_1, e_2, \ldots, e_k\} \rangle$ can be constructed from NC by the introduction of a new vertex v and k new edges $e_1, e_2, \ldots, e_k$ to NC such that the increase Δm in the number of missing edges conforms to

$$\Delta m < \frac{|V|}{2} + \frac{m}{|V|-1} \qquad [15]$$

For $|V| > 3$, [15] requires

$$m \leq \left\lfloor \frac{|V|-1}{2} \right\rfloor \qquad [16]$$

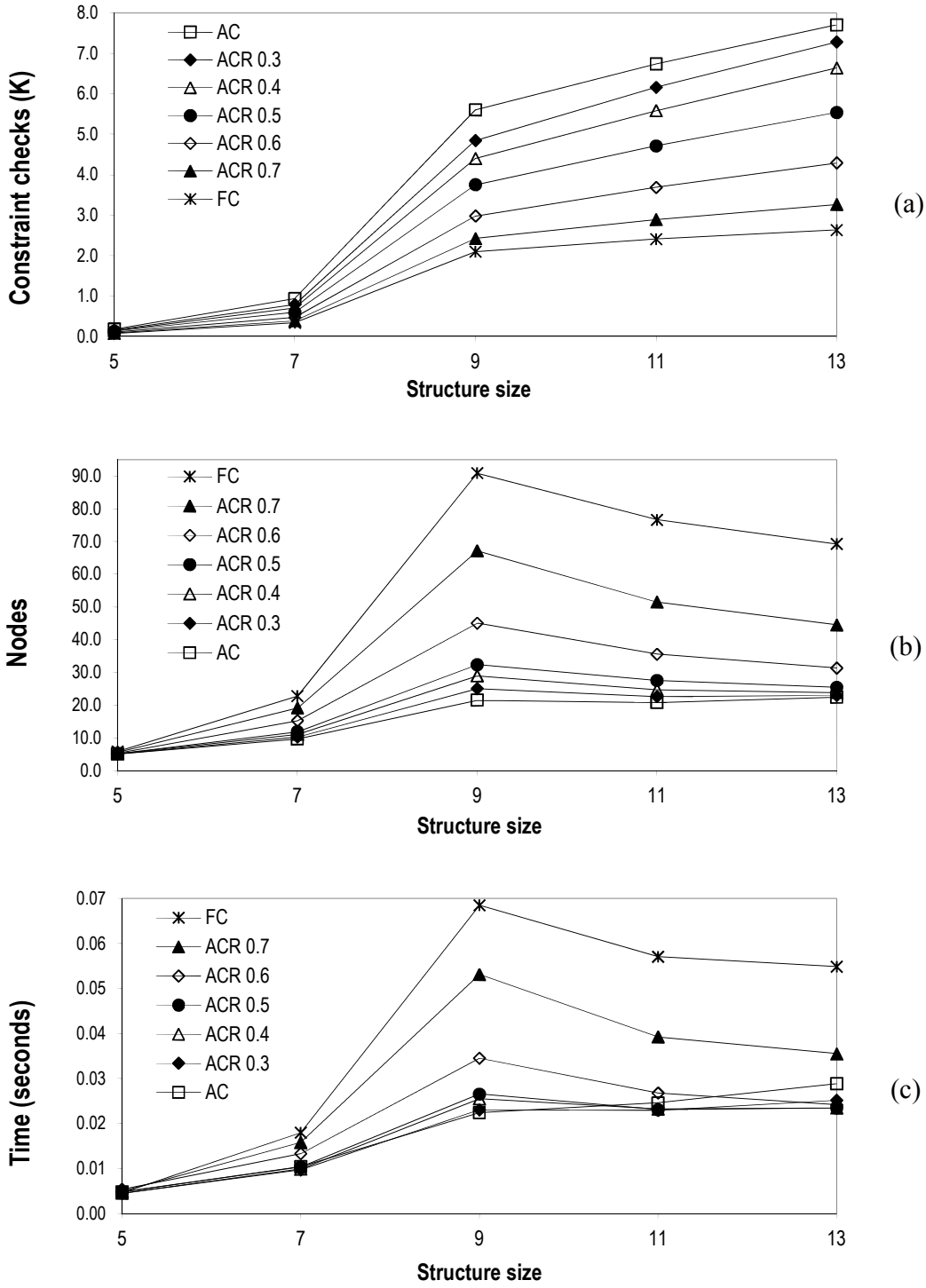


Figure 4.3. Number of (a) checks, (b) expanded nodes, and (c) CPU seconds required to solve cluster-like graphs of various sizes under 7 different inference methods. Lazier propagation does fewer checks, expands more nodes, and is sometimes faster.

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 appear in Figure 4.3. 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 2.5(a) 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 lazier. Thus a reasonable cluster-based propagation method for composed CSPs is as AC/2/AC/FC.

Unlike the satellites in composed CSPs, subproblems in real-world instances are often highly connected to other subproblems, as suggested by Figures 2.1(d) and 3.2(d). To study the effect of structure-based inference on real-world instances, we built artificial problems with similar, but simplified, structure like the driverlog-09 problem in Figure 2.1. These simplified problems include only variables in a pair of designated clusters of interest and constraints that are between these variables. We picked clusters of maximum size in this simplification because they are usually selected in the beginning of search. In a simplified problem, there are two types of edges. An *inside edge* connects two variables

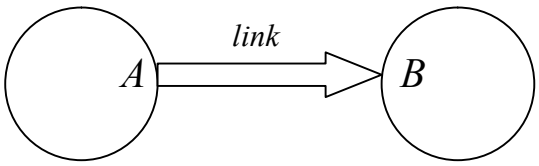


Figure 4.4. An example of the simplified real-world problem. Only one link edge, between variable *A* in the left cluster and variable *B* in the right cluster, is shown. Other links are omitted.

in the same cluster while a *link edge*, as shown in Figure 4.4, connects two variables from different clusters.

We designed a new cluster-based inference method: *Cluster-FC*. Because the clusters that *Foretell* detects usually have high density, FC would be a sufficient inference method inside a cluster. *Cluster-FC* uses a queue (similar to the one used in MAC) to keep track of edges that need to be propagated. The difference between this method and the standard FC is what happens after inference traverses a link edge. For example, suppose inference from *A* in one cluster to *B* in another cluster (shown in Figure 4.4) reduced *B*'s domain. *Cluster-FC* would then enqueue all other constraints between *B* and an unassigned variable. *Cluster-FC* does more inference work than *FC*; it revisits those enqueued constraints while *FC* does not. The motivation of the extra work is to lead inference across links into clusters where more reduction is likely to occur.

Experiments showed that *Cluster-FC* achieved backtrack free search on the simplified problems. We tried it on three real-world problems, the RLFAP scene11 problems and the two largest driverlog problems (08c and 09). We used *Cluster-FC* on *inside* and *link edges* and MAC-3 on all other edges. We call this new inference method *CFC-AC*. The empirical results of cluster-based inference are shown in Section 5.1.

4.3 Summary

This chapter has presented two ways to use the structure identified by *Foretell* in search. Structure-based variable-ordering heuristics try to solve the detected structure before the rest of the problem. We have shown that, among the different heuristics that address the clusters, *Focus*, which selects the tightest cluster and solves it before it moves to any new cluster, performs the best. The structure detected by *Foretell* also makes it possible to fine tune consistency enforcement. In general, stronger consistency enforcement is effective in solving clusters, while FC or lazier forms of MAC-3 may be sufficient for other areas of the constraint graph. The structure detection algorithms in Chapter 3 and the structure-based search heuristics for variable-selecting and inference in this chapter are the key contributions of this dissertation.

Chapter 5

Empirical results

This chapter presents empirical results on the structure-based search heuristics described in Chapters 3 and 4. It uses the state-of-the-art adaptive variable-ordering heuristic *MinDomWdeg* as the baseline. Variations on *Foretell* (detailed in Chapter 3) are used to detect crucial substructure. We use only the strongest structure-based variable-ordering heuristic, *Focus* (from Section 4.1), which solves clusters one at a time based on their estimated dynamic tightness. MAC-3 is used in all the following experiments except those with cluster-based inference. All experiments break ties lexically.

Table 5.1 provides an outline. As the sections progress easier problems are replaced with more difficult ones to demonstrate more advanced techniques. Section 5.1 shows results with tightness-based *Foretell*. Section 5.2 compares the impact on structure-based

Table 5.1. Outline of structure-based CSP search methods presented in Chapter 5.

Section	Structure detector	Metric used to detect structure	Variable Heuristic	Probing	Restart	Weight freeze
5.1, 5.2	*Foretell*	tightness	*MinDomWdeg*	–	–	–
5.2	*Foretell*	tightness	*MinDomInfdeg*	–	–	–
5.3.2	*Foretell-EW*	edge weight	*MinDomWdeg*	random	random	yes
5.3.2	*Foretell-IW*	influence weight	*MinDomInfdeg*	random	random	yes
5.3.3	*Foretell*	tightness	*MinDomWdeg*	heuristic	reasoned	no
5.3.3	*Foretell*	tightness	*MinDomWdeg*	random	reasoned	no
5.3.3	*Foretell*	tightness	*MinDomInfdeg*	heuristic	reasoned	no
5.3.3	*Foretell*	tightness	*MinDomInfdeg*	random	reasoned	no
5.3.4	*Foretell-EW*	edge weight	*MinDomWdeg*	heuristic	reasoned	no
5.3.4, 5.3.5	*Foretell-IW*	influence weight	*MinDomInfdeg*	heuristic	reasoned	no

search of the influence-weight-based variable-ordering heuristic *MinDomInfdeg* to that of *MinDomWdeg*. Section 5.3 explores *Foretell-EW* (edge-weight-based *Foretell*) and *Foretell-IW* (influence-weight-based *Foretell*) with probing restarts to collect information on contention. Finally, Section 5.4 describes an interesting application of *Foretell* to a biological problem, whose goal is to cluster similar proteins.

Because some experiments were on different platforms, results of the same setting on experiment may differ from one table to the next. The results in the upper sections of Tables 5.2 and 5.3 were produced by Macintosh Common Lisp (MCL) 5.1 on a PowerMac with two 2.5GHz G5 processors and 4GB of memory running OS X 10.4. All other results were produced by Clozure Common Lisp (CCL) 1.4 on a Mac Pro with one quad-core 2.93GHz Xeon processor and 8GB of memory running OS X 10.6. Search time and nodes are reported in all tables. Because search nodes are platform or solver independent, it is reasonable to compare the numbers of search nodes we report here to those reported in other literature, as what we are going to do in Section 5.3.4.

5.1 Tightness-based *Foretell*

Tightness-based *Foretell* (described in 3.2.1) relies on constraint tightness to estimate variable pressure, which is then used by *Foretell* to detect structure. The experiments in this section compare *MinDomWdeg* with the strongest cluster-based search heuristic: *Foretell* and *Focus* with *MinDomWdeg*, where *MinDomWdeg* is used to solve each cluster as it is chosen by *Focus*. To control for the non-determinism of local search, every experiment with *Foretell* and its variations is reported as an average across 10 trials for each problem. On each problem, *Foretell* was given some number of milliseconds (ms.)

per cluster, and identified as many clusters as it could until a call to VNS failed when the time was up.

Algorithms were tested on all the classes of composed problems on the benchmark website (Lecoutre, 2009), where there are 10 problems per class. A problem described there by *a-b-c* denotes a central component with *a* variables, *b* satellites of 8 variables each and *c* links. All variables have domain size 10, with constraint tightness 0.150 within the central component and 0.050 on each link. Constraint density within a satellite is always 0.786. The structure of the composed problems in these classes is deliberately obscured. Nonetheless, the structure detected by *Foretell* represents the structural descriptions provided for those problems.

The upper sections of Table 5.2 and Table 5.3 report results on composed problems. Table 5.2 lists the number, average size, and maximum size of the clusters detected with *Foretell* prior to search. In 30 minutes each, *MinDomDeg* could solve only 9 of those 90 benchmark problems. That and the search tree sizes for *MinDomWdeg* suggest that these benchmarks are easier than *Comp*. On six benchmark classes, *Foretell* + *Focus* once again provided an order of magnitude speedup. Note that, for a fixed central component size, our approach scales about linearly with problem size.

Clusters are often readily detected in CSPs for real-world problems too. The lower sections of Table 5.2 and Table 5.3 include RLFAP, driverlog, and Taillard jobshop problems. Scene11 is the most difficult RLFAP; scene11-f10 is a modified scene11 problem with its highest 10 radio frequencies (domain values) removed. This modification makes the problem unsolvable and considerably more difficult than the original problem. The jobshop problems (os-taillard) are intensional CSPs, whose

Table 5.2. Classes in the upper section are composed CSPs, with central component density *d*, satellite tightness *t′*, and link density *d″*. Data for *Foretell* includes number of clusters, average cluster size, maximum cluster size, percentage of variables covered by clusters, averaged across 10 runs, and percentage of total search time devoted to *Foretell*.

				# of	*Foretell's clusters*				
Problems	***d***	***t′***	***d″***	***CSPs***	***Count***	***Size***	***Max***	***Coverage%***	***Time%***
25-10-20	0.667	0.50	0.010	10	10.17	5.20	5.58	50.37	2.87
25-1-80	0.667	0.65	0.010	10	5.60	5.28	6.08	89.60	3.01
75-1-80	0.216	0.65	0.133	10	9.09	4.86	5.90	53.23	0.14
25-1-2	0.667	0.65	0.010	10	1.01	5.77	5.77	17.66	1.45
25-1-25	0.667	0.65	0.125	10	2.30	5.60	5.90	39.03	3.51
25-1-40	0.667	0.65	0.200	10	5.00	5.37	6.40	81.36	1.16
75-1-2	0.216	0.65	0.003	10	1.00	5.69	5.69	6.86	2.07
75-1-25	0.216	0.65	0.042	10	5.40	5.24	6.46	34.09	2.23
75-1-40	0.216	0.65	0.067	10	4.60	5.29	5.80	29.32	5.86
Comp	0.150	0.50	0.120	50	11.00	4.31	5.15	23.71	1.68
scene11	—	—	—	1	5.20	13.02	18.00	9.96	1.02
scene11_f10	—	—	—	1	13.00	11.95	17.80	22.85	3.01
driverlogw-09	—	—	—	1	12.40	17.43	30.10	33.25	9.44
driverlogw-08cc	—	—	—	1	4.00	31.63	46.00	31.01	1.37
driverlogw-08c	—	—	—	1	4.00	31.68	46.00	31.06	1.57
driverlogw-04	—	—	—	1	18.80	9.01	23.80	62.28	23.98
driverlogw-02	—	—	—	1	18.00	8.01	15.80	47.90	6.83
os-taillard-4-95-0	—	—	—	1	5.00	3.20	4.00	100.00	0.05
os-taillard-4-95-1	—	—	—	1	5.00	3.18	3.90	99.38	0.00

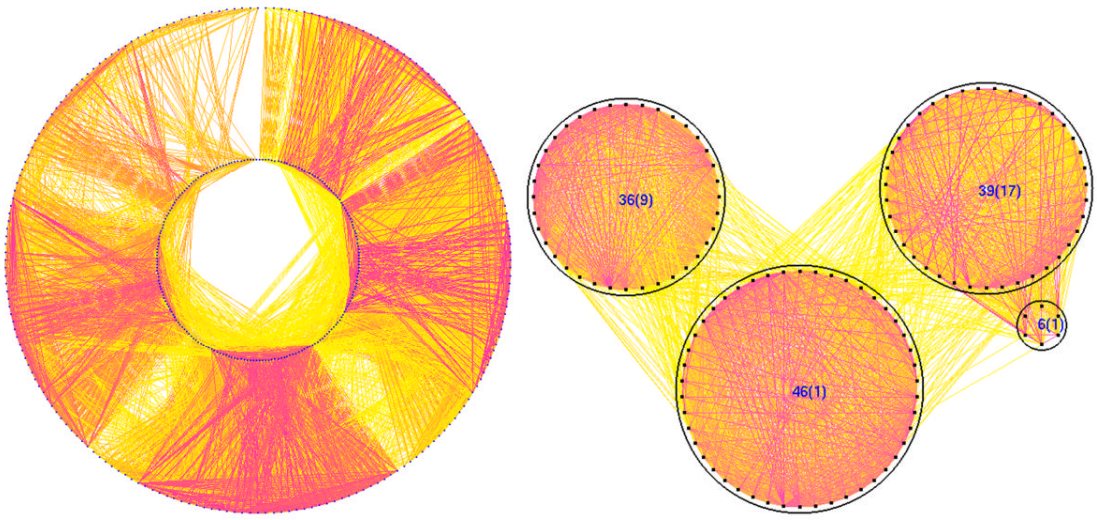
Table 5.3. At the 95% confidence level, *Foretell+Focus+MinDomWdeg* (*FFM*) outperforms *MinDomWdeg* (baseline) on the classes/problems reported in Table 4. Order of magnitude improvements over *MinDomWdeg* are in boldface. Time is in CPU seconds. Data for *FFM* is mean and standard deviation over 10 runs. *FFM* time includes the time to detect clusters.

				Baseline	*FFM Time*		*Baseline*	*FFM Nodes*	
Problems	***n***	***k***	***|C|***	***Time***	***μ***	***σ***	***Nodes***	***μ***	***σ***
25-10-20	105	10	620	2.49	0.88	0.47	670.10	192.07	149.88
25-1-80	33	10	302	0.95	0.26	0.25	308.00	94.50	71.81
75-1-80	83	10	702	2.32	0.37	0.17	595.20	181.40	21.69
25-1-2	33	10	224	1.01	**0.02**	0.00	553.00	**41.40**	1.36
25-1-25	33	10	247	0.91	**0.04**	0.02	465.70	**41.60**	1.29
25-1-40	33	10	262	1.10	**0.07**	0.02	473.80	**41.50**	1.21
75-1-2	83	10	624	3.33	**0.04**	0.01	1171.70	**91.60**	1.50
75-1-25	83	10	647	3.29	**0.15**	0.12	1084.40	**91.40**	1.29
75-1-40	83	10	662	2.97	**0.15**	0.14	960.90	**91.30**	1.28
Comp	200	10	1257	83.58	**4.31**	2.41	12519.40	**497.96**	324.32
scene11	680	44	4103	36.97	13.12	0.06	2810.00	980.40	1.27
scene11_f10	680	34	4103	123.14	35.46	0.06	8768.00	2644.00	0.00
driverlogw-09	650	12	17447	532.50	199.85	17.95	15987.00	5752.90	489.70
driverlogw-08cc	408	11	9312	89.05	43.97	0.40	4880.00	2597.50	0.00
driverlogw-08c	408	11	9312	88.52	38.70	0.45	4820.00	2304.70	0.00
driverlogw-04	272	11	3876	6.01	3.17	0.38	751.00	350.10	51.85
driverlogw-02	301	8	4055	16.11	5.61	1.42	1862.00	649.30	189.61
os-taillard-4-95-0	16	182	48	28.58	6.13	0.09	303.00	95.00	0.00
os-taillard-4-95-1	16	220	48	255.43	142.76	2.30	4721.00	1577.80	10.12

constraints are defined by predicates. Since their predicates are disjunctive inequalities over integer intervals, ACE solves those inequalities (as described in Section 1.1.2) to calculate the constraint tightness that *Foretell* requires. For cluster-based search, time includes the time to calculate tightness and the time used by *Foretell* to detect clusters. The difficulty of a class of problems is gauged here by the resources *MinDomWdeg* required to solve it. On all real-world CSPs, cluster-based search significantly outperforms *MinDomWdeg*, at the 95% confidence level, on both time and nodes. These results support the premise that clusters detected by *Foretell* address the hardest parts of a problem. Far fewer incorrect assignments were made under cluster-based search.

The two driverlog-08 problems (08c and 08cc) differ in the tuples they allow, but they have the same primary structure and the same tightness on their constraints. On every run, *Foretell* found the identical sparse secondary structure in the two problems, that is, exactly the same clusters among the 408 variables: 3 large nodes (2 single-cluster nodes and 1 two-cluster node) with 2 edges (Figure 5.1(b)). That *Focus* improves search on them both confirms its ability to manage structure dynamically.

Not every CSP will benefit from *Foretell.* On easy problems, finding clusters may be informative but unnecessary. On unstructured problems, *Foretell* is likely to find few clusters, so it may be merely a preparatory digression. (If *Foretell* finds no clusters at all, *Focus* simply reverts to *MinDomWdeg.*) On problems with uniform tightness, clusters are either cliques or a few edges shy of them, but the algorithms remain applicable. In the work reported in this section, on average 4.56% of the total problem-solving time was devoted to *Foretell. Foretell* used a much larger percentage on the driverlog problems, however, possibly because their structure is so complex (See Figures 2.1(b) and 5.1(a).).



(a) (b)

Figure 5.1. The (a) constraint graph and the (b) identified structure for the driverlogw-08c problem: 3 large nodes (2 single-cluster nodes and 1 two-cluster node) with 2 edges.

Such overhead is well justified, given that the total search time, including the time *Foretell* consumes, is still reduced by at least 47%.

Table 5.2 also shows cluster *coverage*, the percentage of variables included in clusters. The average coverage varies with problem type. The jobshop problems are almost entirely covered by clusters, not surprising given that every variable participates in two cliques of size 4. Although these two problems have the fewest variables among those addressed here, their large domains make them challenging. *Foretell*, however, is unaffected by large domains once tightness is calculated. The jobshop clusters, detected by *Foretell* in less than 0.5 seconds, again reduce search effort effectively. Coverage of the benchmark composed problems relates to the number of their links (the last value in their names). As the density of links increases, the likelihood that *Foretell* will include central component variables in clusters rises.

Cluster-based inference (AC/2/AC/FC for composed CSPs and CFC-AC for real-world CSPs) was added to cluster-guided search and tested on every problem in Tables 5.2 and 5.3. On all the classes of composed problems, there was little room for improvement over cluster-guided search, and none appeared. On RLFAP-scene11, however, CFC-AC further reduced search time by 2% more than the results in Table 5.3, which used only MAC-3; it also reduced constraint checks by 7%. Both reductions are significant at the 95% confidence level. The success of cluster-based inference on RLFAP-scene11 suggests that *Foretell*'s clusters are areas where backtracks frequently happen so that FC suffices for the "inside." (This improvement is not attributable solely to FC; FC alone is dramatically slower on this problem.) On the driverlog problems, cluster-based inference did not improve cluster-guided search. This unexpected result shows how hard it is to apply cluster-based inference to real problems. The simplified driverlog problems, described in Section 4.2, may not adequately represent the original driverlog problems, so that the experience on the simplified problems does not transfer to the original ones. It would also be interesting to study why CFC-AC works for RLFAP-scene11. Although *Focus* successfully uses *Foretell*'s clusters to guide variable-selection during search, the way that these clusters could best support inference may be entirely different. The improvement achieved on RLFAP-scene11 with CFC-AC suggests that cluster-based inference merits further investigation.

5.2 *MinDomWdeg* vs. *MinDomInfdeg*

MinDomInfdeg was introduced in Section 3.2.3. Similarly to *MinDomWdeg*, it learns about contention, areas where wipeout frequently happens, and as search proceeds, it

guides search to such locations. The difference between *MinDomWdeg* and *MinDomInfdeg* is which constraint is considered the cause of the wipeout. As shown in Figure 3.1, *MinDomWdeg* credits the *wipeout culprit*, the constraint that directly caused the wipeout, while *MinDomInfdeg* credits the *wipeout instigator*, the constraint between the variable whose assignment instigated the inference process that eventually led to the wipeout and the variable whose domain is wiped out.

In the following experiments, we compare four search heuristics: *MinDomWdeg*, *MinDomInfdeg*, *Foretell+MinDomWdeg* and *Foretell+MinDomInfdeg* on a class of artificially generated structured CSPs (*Comp*) and a class of real-world CSPs (driverlog). The last two heuristics should not be confused with *Foretell-EW* or *Foretell-IW* (described in 3.2.2 and 3.2.3), which use weights for structure detection. Experiments in this section, similar to those in the previous section, use *Foretell*, which uses constraint tightness for structure detection.

Table 5.4 shows the results on the class of *Comp*, with mixed solvable and unsolvable instances. Horizontal comparisons indicate that heuristics based on *MinDomInfdeg* are more efficient. Statistically, *MinDomInfdeg* (36.46s) is faster than *MinDomWdeg* (47.60s) with $p = 0.00022$. In turn, *Foretell+MinDomInfdeg* (1.35s) is faster than *Foretell+MinDomWdeg* (1.55s) with $p = 0.0054$. Heuristics based on *MinDomInfdeg* produce smaller search tree sizes and require fewer checks than corresponding heuristics based on *MinDomWdeg*. The differences in nodes and checks between the two *Foretell*-enhanced versions are, however, not statistically significant. Vertical comparisons in Table 5.4 show that structure-based search can dramatically

Table 5.4. *MinDomWdeg*-based heuristics and *MinDomInfdeg*-based heuristics solve 100 mixed solvable and unsolvable *Comp* instances. Time is in CPU seconds. Results with *Foretell* are averaged over 10 runs. At the 95% confidence level, *MinDomInfdeg* is significantly faster than *MinDomWdeg* and *Foretell+MinDomInfdeg* is the fastest heuristic. Note that in the top half of the table, but not in the bottom half, nodes are in thousands and checks are in millions.

Comp	***MinDomWdeg***			***MinDomInfdeg***		
	Time	Nodes	Checks	Time	Nodes	Checks
μ	47.60	12.74K	11.94M	36.46	12.21K	10.97M
σ	23.92	6.51K	5.48M	19.88	6.85K	5.44M
Comp	***Foretell+MinDomWdeg***			***Foretell+MinDomInfdeg***		
	Time	Nodes	Checks	Time	Nodes	Checks
μ	1.55	241.07	242.40	1.35	228.00	228.43
σ	1.46	417.32	426.92	1.15	398.72	414.25

improve search performance by at least an order of magnitude. Among the four heuristics, *Foretell+MinDomInfdeg* is significantly faster than the other three methods.

Table 5.5 shows the results of the same four search heuristics on a set of benchmark driverlog problems. Without *Foretell*, *MinDomWdeg* and *MinDomInfdeg* perform similarly on these seven problems. With *Foretell*, however, there are differences. Three of the four easier problems (01c, 04c and 05c) do not actually need *Foretell*. *MinDomWdeg* or *MinDomInfdeg* perform best on them; adding *Foretell* significantly slows search. For 02c and the three hardest driverlog problems (08c, 08cc and 09), however, the structure detected by *Foretell* proves to be extremely helpful. When used with *MinDomWdeg*, *Foretell* reduces the search effort by at least half. *MinDomInfdeg* benefits even more from *Foretell*. Alone, *MinDomInfdeg* is slower on the three hardest problems than *MinDomWdeg*, but *Foretell+MinDomInfdeg* outperforms the time of *Foretell+MinDomWdeg* by 6%, 27%, 37% and 29% for the 02c, 08c, 08cc and 09

Table 5.5. *MinDomWdeg*-based heuristics and *MinDomInfdeg*-based heuristics solve driverlog problems. Time is in CPU seconds. Results of *Foretell* are averaged over 10 runs. As the problem gets harder, the advantage of structure-based search with *MinDomInfdeg* becomes larger. Best performance on each problem is in **bold**.

driverlog	***MinDomWdeg***		***Foretell+MinDomWdeg***	
	Time	Nodes	Time	Nodes
01c	**0.08**	**73.00**	0.33	73.10
04c	6.22	751.00	7.41	721.00
05c	**16.58**	**1391.00**	41.79	2817.30
02c	16.51	1862.00	6.34	554.30
08c	88.52	4820.00	38.70	2304.70
08cc	89.05	4880.00	43.97	2597.50
09	532.50	15987.00	199.85	5752.90
driverlog	***MinDomInfdeg***		***Foretell+MinDomInfdeg***	
	Time	Nodes	Time	Nodes
01c	**0.08**	**73.00**	0.35	73.30
04c	**4.86**	**607.00**	6.52	641.00
05c	19.41	1692.00	35.09	2544.60
02c	15.46	1782.00	**5.94**	**565.20**
08c	96.31	5522.00	**28.18**	**1852.00**
08cc	103.25	5917.00	**27.67**	**1824.00**
09	544.63	15297.00	**141.41**	**4391.70**

problems, respectively. Compared with the state-of-the-art results for *MinDomWdeg*, *Foretell+MinDomInfdeg* solves these problems in about one third the time.

Figure 5.2 plots the time and nodes reported in Table 5.5. It also includes the standard deviations of the two *Foretell* runs. Although *Foretell* is non-deterministic, the standard deviations of the two *Foretell* runs are orders of magnitude smaller than their corresponding means. The improvement by structure-based search is, therefore, consistent and robust.

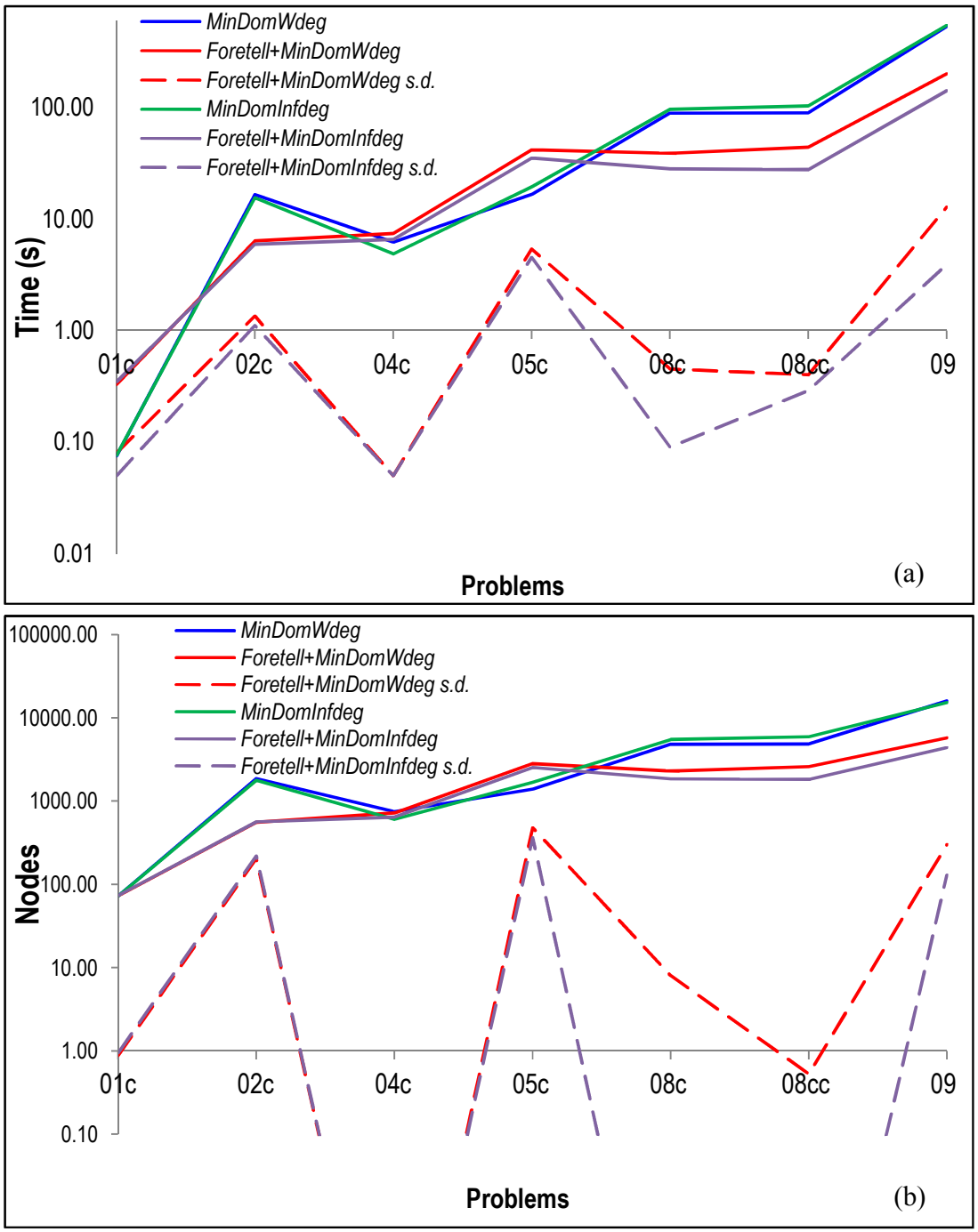


Figure 5.2. Logarithmic plot for (a) time and (b) nodes of the four heuristics in Table 5 on the seven driverlog problems. The standard deviations of search nodes by *Foretell+MinDomWdeg* and *Foretell+MinDomInfdeg* on 04c, 08c and 08cc are zero so they cannot be displayed on a logarithmic plot.

5.3 Weight-based versions of *Foretell*

Foretell-EW and *Foretell-IW*, use probing (short and quick restarts) to collect weights before search. Then they use the weights, which represent the distribution of contention, to detect structure. Search after either weight-based version of *Foretell* is identical to search after tightness-based *Foretell*, where *Focus* and *MinDomWdeg* (for *Foretell-EW*) or *MinDomInfdeg* (for *Foretell-IW*) work together to solve the problem. This section shows the results of weight-based versions of *Foretell* on a class of modified RLFAP problems. Scene11-f*x* is a modified scene11 problem with its highest *x* radio frequencies (domain values) removed. Although the original scene11 is solvable, all modified RLFAP scene11 problems are unsolvable. All of the modified RLFAP scene11 problems are considerably harder than the original scene11. Among them, scene11-f1 is the hardest and scene11-f12 is the easiest.

5.3.1 How much probing is enough?

The first experiment studies how much probing effort is necessary. Here *A*×*B* probing

Table 5.6. Probing effort, search effort and total effort on rlfap-scene11-f8.xml. Row header *A*×*B* denotes a probing regime of *A* restarts, each of which allows *B* retractions. Time is in CPU seconds. Best measurements are in **boldface**. Statistically, there is no significant difference among the five rows.

Probing effort	Probing time		Search time		Search nodes		Total nodes		Total time	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
25×50	29.92	1.15	328.29	52.49	18988.60	3061.00	**20479.40**	3075.65	**358.21**	52.74
50×25	39.94	2.86	324.31	89.70	18933.80	4647.34	20806.40	4664.65	364.25	90.28
50×50	53.63	1.51	326.26	93.06	18964.30	4775.71	21609.30	4754.12	379.89	92.52
25×100	48.16	0.72	341.41	39.52	19674.50	2440.50	22122.80	2424.51	389.57	39.14
100×25	66.04	3.03	**312.90**	42.68	**17906.40**	2292.56	21053.20	2282.75	378.94	42.41

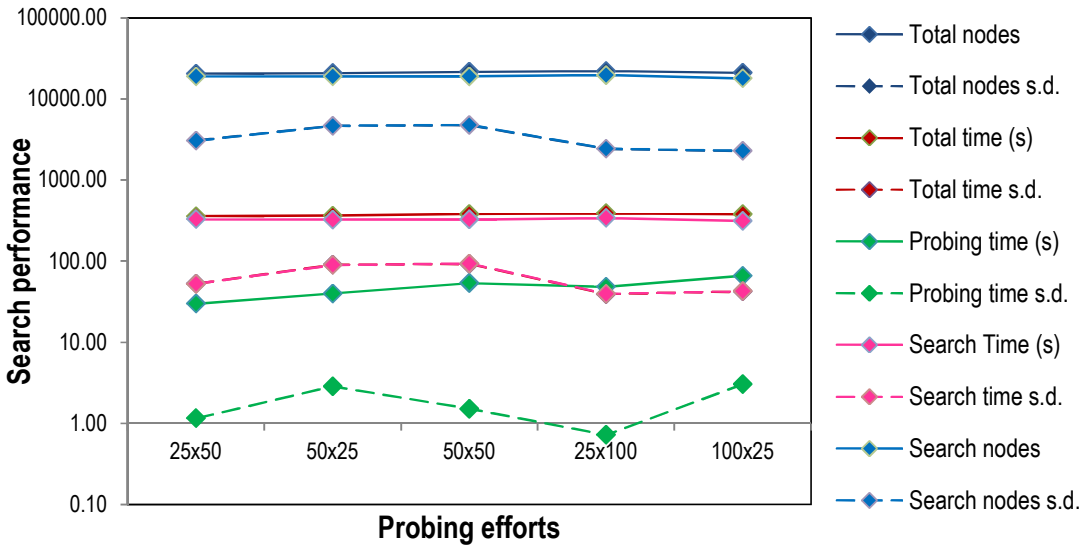


Figure 5.3. The impact of probing on rlfap-scene11-f8.xml.

denotes *A* probing restarts with *B* retractions allowed in each. Scene11-f8 was chosen for this experiment because it is moderately difficult. The results are shown in Table 5.6 and plotted in Figure 5.3. There are three groups of columns (for probing, search performance, and total measurements) in Table 5.6. With more restarts, the collected weights are likely to represent the contention distribution better, so that the time and nodes used in search are reduced. Indeed, 100×25 produces the best search performance after probing. This reduction, however does not justify the extra cost incurred by probing; 25×50 has the best overall performance even though it spends more time and nodes in search. Compared to scene11-f8, which is a relatively easy modified scene11 problem, more difficult modified scene11 problems may need the higher-quality weights learned in longer probing. To balance the performance and the quality of the contention distribution obtained during probing, subsequent experiments on the modified scene11 problems used 50×50 probing.

5.3.2 Uninformed approach

As noted in Section 3.2.2, previous work on random CSPs shows that unbiased probing achieved the best performance (Grimes and Wallace, 2007). In other words, probing should try to find global contention and avoid local contention as much as possible. In unbiased probing, every probing restart randomly selects its root variable, and random variable-ordering is used during probing. We call this *random restart* and *random probing*. To prevent the weights learned during probing from contamination by any local contention that may be experienced during the final search, weights are frozen after probing. Search only uses the weights, but does not update them further (*weight freeze*). The experiment in 5.3.1 used this *uninformed approach*, which combines random restart, random probing, and weight freeze.

Experiments in this section try the uninformed approach on all the modified scene11 problems. In Table 5.7, we test every modified scene 11 problems with three pairs of methods. The baseline pair is *MinDomWdeg* and *MinDomInfdeg*. They do not use probing or any structure-based method. Because these two heuristics are both deterministic, experiments with this pair require only one run. The second pair adds probing to the baseline: *MinDomWdeg* with probing and *MinDomInfdeg* with probing. The uninformed approach helps to highlight global contention with weights. During search, the corresponding heuristic uses the collected contention information to guide search. The last pair uses structure-based search. After probing, weight-based *Foretell* (*Foretell-EW* or *Foretell-IW*) uses the collected contention information to detect structure, which then guides variable selection during search together with the learned weights. The results of the experiments with the last two pairs are averaged over 10 runs, because the

Table 5.7. Uninformed approach on modified scene11 problems where *fx* is more difficult for smaller values of *x*. Time is in CPU seconds. Search is terminated after 7200 seconds. Results on scene11-f1, scene11-f2 and scene11-f3 are omitted because no heuristic solved them within the time limit. Best measurements are in **boldface.**

		Heuristics	Solve rate	Time		Nodes	
				μ	σ	μ	σ
1	f4	*MinDomWdeg*	0/1	No solution			
2		*MinDomInfdeg*	0/1	No solution			
3		*MinDomWdeg* with probing	1/10	7087.41	356.09	460.53K	20.22K
4		*MinDomInfdeg* with probing	0/10	No solution			
5		*Foretell-EW*	1/10	7142.13	183.03	531.43K	25.99K
6		*Foretell-IW*	**2/10**	**6377.09**	**1735.13**	**588.89K**	**165.31K**
7	f5	*MinDomWdeg*	0/1	No solution			
8		*MinDomInfdeg*	0/1	No solution			
9		*MinDomWdeg* with probing	0/10	No solution			
10		*MinDomInfdeg* with probing	1/10	6882.87	1002.89	513.65K	71.72K
11		*Foretell-EW*	0/10	No solution			
12		*Foretell-IW*	**2/10**	**5948.23**	**2639.09**	**520.20K**	**234.41K**
13	f6	*MinDomWdeg*	0/1	No solution			
14		*MinDomInfdeg*	0/1	No solution			
15		*MinDomWdeg* with probing	1/10	6527.21	2127.57	448.95K	145.93K
16		*MinDomInfdeg* with probing	2/10	6710.76	1033.05	499.41K	77.24K
17		*Foretell-EW*	2/10	6512.20	2037.89	493.15K	155.79K
18		*Foretell-IW*	**6/10**	**4777.35**	**2989.91**	**441.49K**	**279.30K**
19	f7	*MinDomWdeg*	0/1	No solution			
20		*MinDomInfdeg*	0/1	No solution			
21		*MinDomWdeg* with probing	5/10	6153.31	1447.42	429.41K	108.08K
22		*MinDomInfdeg* with probing	**10/10**	**2835.43**	**1414.09**	**204.47K**	**101.20K**
23		*Foretell-EW*	6/10	6206.35	1382.00	464.21K	101.94K
24		*Foretell-IW*	9/10	3489.03	1765.75	320.83K	159.91K
25	f8	*MinDomWdeg*	0/1	No solution			
26		*MinDomInfdeg*	1/1	5195.98	–	274.61K	–
27		*MinDomWdeg* with probing	9/10	3452.27	1.91K	232.74K	129.77K
28		*MinDomInfdeg* with probing	9/10	1825.65	1.97K	122.95K	126.78K
29		*Foretell-EW*	9/10	3020.05	1763.26	215.54K	141.45K
30		*Foretell-IW*	**10/10**	**379.89**	**92.52**	**21.61K**	**4.75K**
31	f10	*MinDomWdeg*	1/1	123.14	–	8768	–
32		*MinDomInfdeg*	**1/1**	**113.56**	–	**8243**	–
33		*MinDomWdeg* with probing	10/10	784.26	1.88K	59.02K	144.16K
34		*MinDomInfdeg* with probing	9/10	927.76	2.21K	57.24K	131.53K
35		*Foretell-EW*	4/10	1385.49	703.41	176.94K	97.04K
36		*Foretell-IW*	4/10	1422.60	694.56	185.01K	96.72K
37	f12	*MinDomWdeg*	1/1	112.20	–	8163	–
38		*MinDomInfdeg*	**1/1**	**103.75**	–	**7939**	–
39		*MinDomWdeg* with probing	9/10	950.19	2.20K	62.75K	143.01K
40		*MinDomInfdeg* with probing	10/10	705.56	1.76K	54.86K	139.00K
41		*Foretell-EW*	10/10	286.35	203.35	30.97K	27.49K
42		*Foretell-IW*	10/10	228.95	146.09	22.51K	18.32K

uninformed approach is non-deterministic.

For the easier problems, such as scene11-f10 and scene11-f12, probing is simply too expensive. The two baseline approaches outperformed more sophisticated approaches. As the problems became more difficult, however, probing proved worth the effort. For instances more difficult than scene11-f10, the two baseline approaches performed poorly: *MinDomWdeg* failed to solve any of them, while *MinDomInfdeg* could solve only scene11-f8.

With probing, the performance and the number of trials that solved each problem (*solve rate*) of the two baseline heuristics improved significantly. *MinDomInfdeg* with probing performed best among all the approaches on scene11-f7. Search with structural knowledge on scene11-f6 and scene11-f7 did not guarantee performance improvement: *Foretell-EW* performed similarly to *MinDomWdeg* with probing. *Foretell-IW*, however, significantly outperformed *MinDomInfdeg* with probing on scene11-f4, scene11-f5, scene11-f6 and scene11-f8. As a matter of fact, *Foretell-IW* is the best-performing approach on those four problems.

In general, Table 5.7 shows that on the modified RLFAP instances, with the uninformed approach, structure-based search improves performance on more difficult instances. It also shows that *MinDomInfdeg* is a more effective heuristic than *MinDomWdeg* on those instances. For easier problems, baseline approaches suffice.

The six methods in Table 5.7 begin to show inconsistency on problems more difficult than scene11-f10. For problems more difficult than scene11-f7, none of them solves the problems consistently. On the most difficult problems (scene11-f3, scene11-f2 and scene11-f1), all six methods fail. This questions the effectiveness of the uninformed

approach for real-world problems with non-random structure. Grimes and Wallace's results were only for artificially-generated CSPs with random structure. The next section adopts a more informed approach to probing.

5.3.3 Informed approach

Here we allow weight learning to bias what is learned initially. Instead of probing for global contention, biased probing finds the strongest local contention from the heuristic's initial decisions. It uses *reasoned restart*, where the first probing restart begins with the weighted heuristic's most favored variable, the second probing restart begins with the weighted heuristic's second most favored variable, and so on. During *heuristic probing*, weights are not only collected, but also used to guide search in these restarts. We call this *heuristic probing*. Moreover, in search after probing, weights are not frozen; they are continually updated and used. This *informed approach* combines reasoned restart and heuristic probing without weight freeze in search. Figure 5.4 highlights the differences

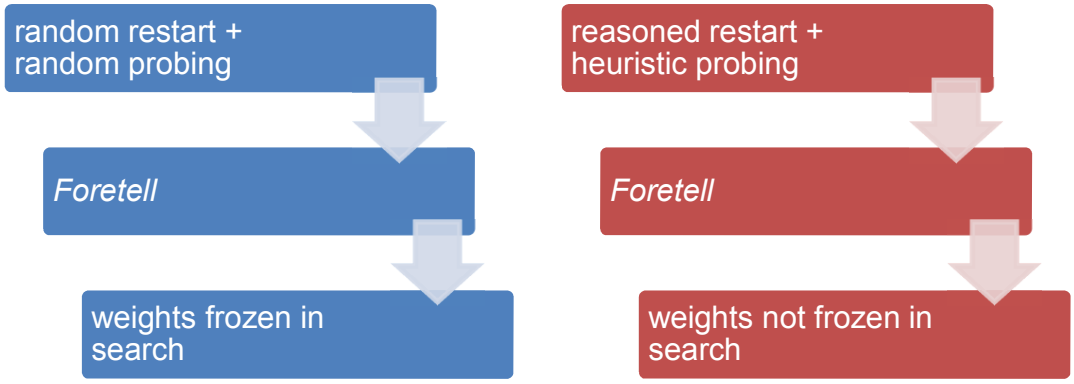


Figure 5.4. Algorithmic components of the uninformed approach and the informed approach to structure-based search.

between the uninformed approach and the informed approach to structure-based search.

The next experiment compares the performance of random probing and heuristic probing under reasoned restart on two moderately difficult problems: scene11-f7 and scene11-f8. We tested four methods: the two baseline heuristics *MinDomWdeg* and *MinDomInfdeg* with and without the help of *Foretell*, the tightness-based structure detector. Results with random probing are averaged over 10 runs here; heuristic probing is deterministic and therefore only needed one run.

Table 5.8 shows the results with 50×50 probing. It is obvious that the bias introduced by heuristic probing dramatically improves search performance. Similarly, comparison of random restart with random probing and *MinDomInfdeg* on these two problems (lines 20 and 26 in Table 5.7) to reasoned restart with random probing and *MinDomInfdeg* (line 2 in Table 5.8) and comparison of random restart with random probing and *MinDomWdeg* (lines 19 and 25 in Table 5.7) to reasoned restart with random probing with *MinDomWdeg* (line 4 in Table 5.8), it is obvious that reasoned restart is superior to random restart.

Table 5.8. Heuristic probing (h.p.) outperforms random probing (r.p.) under reasoned restart on scene11-f7 and scene11-f8. Time is in CPU seconds. Results of random probing are averaged over 10 runs. All probing is 50×50.

	Methods	probing	F8		F7	
			Time	Nodes	Time	Nodes
1	*MinDomInfdeg*	h.p.	**65.12**	**3.2K**	**135.25**	**8.03K**
2		r.p.	201.24	15.42K	291.54	22.35K
3	*MinDomWdeg*	h.p.	**67.23**	**3.4K**	**124.90**	**7.69K**
4		r.p.	285.71	20.68K	509.77	36.21K
5	*Foretell+MinDomInfdeg*	h.p.	**68.66**	**3.21K**	**124.31**	**8.03K**
6		r.p.	223.55	18.04K	833.32	64.79K
7	*Foretell+MinDomWdeg*	h.p.	**71.44**	**3.4K**	**123.45**	**7.69K**
8		r.p.	245.63	18.26K	495.19	36.70K

5.3.4 Weight-based *Foretell* on modified RLFAPs

The previous section demonstrated the importance of bias in weight learning on structured (i.e., non-random) problems. The next experiment, reported in Table 5.9, tests the two versions of weight-based *Foretell* (*Foretell-EW* and *Foretell-IW*) on all the modified scene11 problems under the same 50×50 probing. In the baseline setting we use the informed approach without *Foretell*. Because the informed approach is deterministic, the baseline requires only one run; experiments with *Foretell-EW* and *Foretell-IW* are averaged over 10 runs. Table 5.9 shows the empirical results of the informed approach.

Table 5.10 compares our results with the results of MACR and MACER$^+$ (Mehta et al., 2009) on the four most difficult problems (scene11-f1 though scene11-f4). MACR is simply global search with MAC and restart. MACR and MACER$^+$ use MAC-3 and restart geometrically with factor = 1.5 and initial resource limit = 30 retractions. From one restart to the next, MACER$^+$ maintains representations of as-yet-unvisited search space. It extracts inconsistent visited search space and its unsolvable cores to avoid revisiting failed subtrees and repeated proofs of unsolvability of those cores. MACER$^+$ is the best performing method for these modified RLFAP problems in that work. Comparisons of *Foretell* with MACR and MACER$^+$ required a large resource limit to ensure that all problems were solved. Because of the differences between solvers and platforms, we appropriately compare only search nodes among methods. (The search time of MACR and MACER$^+$ reported in (Mehta et al., 2009) is significantly shorter than the search time of our methods with the constraint solver *ACE*. See Section 6.7 for further explanation.)

For the easy problems in this set (scene11-f12, scene11-f10 and scene11-f8), there is no need to use either version of weight-based *Foretell*. The weights that are collected in

Table 5.9. Performance of the two weight-based versions of *Foretell* with the informed approach. Time is in CPU seconds. Best results are in **boldface**.

MinDom Infdeg	Probing only		*Foretell-IW*			
	Time	Nodes	Time		Nodes	
			μ	σ	μ	σ
f1	174.85K	13019.10K	**136.77K**	6.70K	12940.15K	180.00K
f2	46.23K	**3554.74K**	**36.43K**	0.43K	3560.98K	9.62K
f3	15.56K	1150.34K	**12.55K**	132.90	**1129.79K**	11.28K
f4	4.27K	**326.27K**	**3.42K**	4.14	**326.27K**	0
f5	1.10K	**80.49K**	**0.94K**	32.55	81.76K	3.21K
f6	203.16	**13.69K**	**187.47**	7.55	13.72K	0.52K
f7	159.38	10.01K	**131.56**	11.89	**8.56K**	1.23K
f8	**66.43**	**3.19K**	67.27	0.69	3.19K	0
f10	47.49	**2.35K**	151.43	0.77	9.20K	0
f12	**45.03**	**2.45K**	228.95	146.09	22.51K	18.32K
MinDom Wdeg	Probing only		*Foretell-EW*			
	Time	Nodes	Time		Nodes	
			μ	σ	μ	σ
f1	165.41K	11735.05K	156.90K	334.95	**11722.86K**	43.37K
f2	56.96K	4178.26K	52.95K	67.46	4174.10K	1.88K
f3	16.84K	1206.78K	15.66K	44.25	1206.94K	2.99K
f4	4.99K	361.82K	4.65K	32.85	362.58K	2.67K
f5	1.26K	88.65K	1.18K	14.74	87.98K	1.10K
f6	334.87	21.21K	306.20	18.19	19.18K	1.31K
f7	157.63	9.42K	161.48	1.56	9.42K	0
f8	67.89	3.45K	75.29	1.36	3.44K	1.20
f10	**46.63**	2.44K	178.30	0.14	9.85K	0
f12	45.94	2.47K	112.35	1.51	6.11K	0

Table 5.10. The informed approach outperforms MACR and MACER$^+$ (Mehta et al., 2009) on the number of search nodes. Best results are in **boldface**. Standard deviations of structure-based search are shown in Table 5.9.

Nodes	*MinDomInfdeg* with probing	*MinDomWdeg* with probing	*Foretell-IW*	*Foretell-EW*	MACR	MACER$^+$
f1	13019.10K	11735.05K	12940.15K	**11722.86K**	45423.63K	29094.86K
f2	**3554.74K**	4178.26K	3560.98K	4174.10K	19882.05K	7833.26K
f3	1150.34K	1206.78K	**1129.79K**	1206.94K	4164.34K	2824.17K
f4	**326.27K**	361.82K	**326.27K**	362.58K	1553.78K	995.11K

probing are strong enough to guide search. As the problems become more difficult, however, the benefits of structure appear. *Foretell-IW* is the fastest method among the four on problems from scene11-f1 through scene11-f7; *Foretell-EW* outperforms its baseline on problems from scene11-f1 through scene11-f6. For these difficult problems, both weight-based versions of *Foretell* produce stable results; their standard deviations are at least an order of magnitude smaller than their means. Moreover, *Foretell-IW* significantly outperforms *Foretell-EW* on all problems but scene11-f12. In Table 5.10, on the four most difficult problems (the only scene11-f*x* problems reported on with MACR and MACER$^+$), influence-weight-based methods use less than half the nodes used by MACER$^+$. Edge-weight-based methods perform similarly. This indicates that weight-based *Foretell* makes better choices during search.

5.3.5 Focus of attention

Table 5.9 shows that both weight-based versions of *Foretell* significantly improve search speed over their respective baselines. There is, however, no improvement on search nodes with *Foretell-IW* or *Foretell-EW*. Inspection showed that sometimes a weight-based *Foretell* used exactly the same number of nodes in search as its heuristic with *Foretell* had used in the baseline experiment, but that weight-based *Foretell* was significantly faster. (The row for f4 with *MinDomInfdeg* in Table 5.9 is an example.)

Table 5.11. The breakout of time used by *MinDomInfDeg* and *Foretell-IW* on rlfap-scene11-f7. Time is in CPU seconds.

TIME	Total	*Foretell*	Probing	Search	Variable-selection	Value-selection
MinDomInfDeg	142.04	0	58.54	83.50	29.03	0.23
Foretell-IW	128.78	3.11	58.72	66.94	12.98	0.23
difference	13.26	-3.11	-0.18	16.56	16.05	0.00

We studied a similar case on an easier problem, scene11-f7. Table 5.11 analyzes the time used in the solver to solve this problem. The *Foretell-IW* run is about 13 seconds faster than the baseline run. *Foretell-IW* spends 3.11 seconds to detect structure, but that then saves 16.56 seconds in search. The second to the last column in Table 5.11 shows that this 16-second advantage of *Foretell-IW* comes from faster variable selection.

How variables are selected during search is clearly important. A heuristic first calculates a score for each candidate variable and then selects the variable with minimal (or maximal) score. For n candidate variables, score calculation uses $O(n)$ time. Variable selection that maximizes or minimizes that score also uses $O(n)$ time, but can go to $O(n\log n)$ time if it is necessary to sort the candidates by their scores. Therefore, variable-selection time is directly affected by the number of candidate variables. For the baseline run, $n = 680$ when search begins. For the *Foretell-IW* run, however, search begins with $n = 20$, the size of the first selected cluster.

Because weight-based *Foretell* builds structure using weights, it is possible that *Foretell* only prioritizes the variables, and that the actual order in which variables are selected is the same as this order with the baseline heuristic. This is why weight-based *Foretell* may not reduce the number of search nodes. *Foretell-IW* can, however, significantly improve search speed by selecting the next variable to assign much faster, because search is able to focus its attention approximately, on a cluster, a much smaller set of candidate variables.

5.4 An application of *Foretell*

Previous sections in this chapter have shown how the structure detected by *Foretell* can successfully guide CSP search. This section applies *Foretell* to an entirely different domain, a fruit fly protein clustering problem. In contrast to constraint satisfaction, there is no solution in the protein clustering problem; the clusters detected by *Foretell* are the final results that we seek. This is a more general application of *Foretell* to cluster a weighted graph.

The problem is presented as a protein-protein pair-wise interaction network: a protein pair indicates that two proteins interact with each. An integer is associated with each protein pair and its value is the confidence level, based on experimental evidence, of this interaction. The goal of this problem is to group or cluster proteins that are functionally similar together. Because search and probing are irrelevant and undefined in this context, this section uses only tightness-based *Foretell*.

The conversion of such an interaction network to a graph is straight-forward. For each protein, we create a vertex. For each pair of interacting proteins, we create an edge, whose weight is the confidence level of the interaction. Because we designed and implemented *Foretell* for CSPs, the remainder of this section treats this problem as if it were a constraint graph. Thus we refer to proteins as variables and to interactions as constraints. Because variable domains in this context are undefined, we create a dummy domain of size 1 for each variable. The tightness of a constraint is the normalized confidence level of the interaction it represents. The constraint graph from this fruit fly protein interaction network has 11,408 variables and 759,580 constraints. Both the number of variables and the number of constraints are orders of magnitude larger than

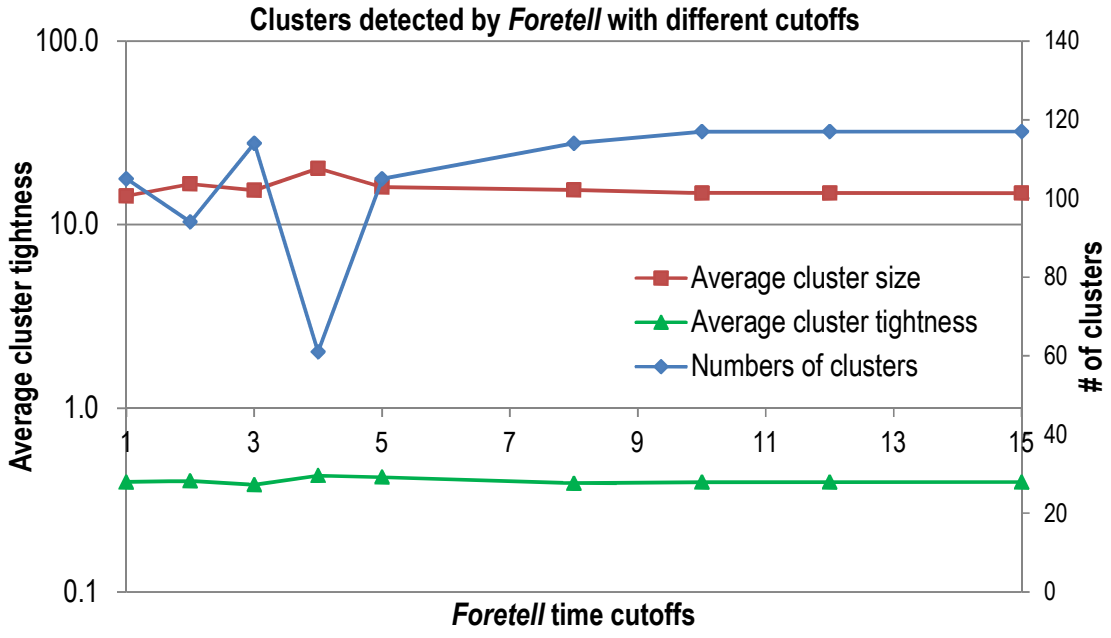


Figure 5.5. Number of clusters, average cluster size and tightness under different *Foretell* cutoffs.

those of the largest CSP used in this dissertation, the driverlog 09 problem with 650 variables and 17,447 constraints.

5.4.1 First approach

The conversion of the problem to CSP format makes it possible to use *Foretell* directly for clustering. The necessary change is the metric that *Foretell* uses to select variables greedily. Because *pressure* is undefined for this problem, we use the sum of the tightness going into each variable in accordance with the problem's semantics. Thus, *Foretell* greedily selects the variable with maximum tightness sum, and breaks ties on maximum variable degree.

We tried a range of cutoffs for search for a single cluster, from 1 to 15 minutes. Figure 5.5 plots the number of clusters and average cluster size and tightness that

Table 5.12. Clusters under time allocations in minutes/cluster search. Size and density are averaged over clusters. *Foretell* cutoffs are in minutes. Total time is in hours.

***Foretell* cutoff**	**All clusters**				**Total time**
	Number of clusters	**Variables**	**Size**	**Density**	
3	114	1758	15.42	0.92	5.92
4	61	1236	20.26	0.95	3.61
10	117	1738	14.86	0.93	15.39

emerged in our experiments at different *Foretell* cutoffs (in minutes) for one cluster. One-minute and two-minute runs are unstable; *Foretell* needed more time to return consistent results. Although cutoffs above 8 minutes allowed *Foretell* to work harder on each cluster, it did not appreciably change the coverage of the graph. Identical sets of clusters were found when the cutoff was 10 minutes, 12 minutes, and 15 minutes. While the clusters detected with a 3-minute cutoff were similar to those detected with an 8-minute or longer cutoff, the clusters detected with a 4-minute cutoff were considerably different: fewer but larger.

Table 5.12 summarizes the clusters detected by *Foretell*. The results of all three cutoffs proved remarkably stable, that is, they formed identical or nearly identical clusters on every run. Three-minute runs produced 114 clusters on 1758 of the 11,408 variables, with average size 15.42. Four-minute runs produced 61 clusters on 1236 variables. The 4-minute clusters were larger, with average size 20.26. We suspect that some of the smaller 3-minute clusters were incorporated into the larger 4-minute ones, leaving fewer variables of interest to *Foretell*. Ten-minute runs produced clusters similar to those from three-minute runs.

Further analysis shows that *Foretell* is selective. Under any cutoff, it never included in its clusters more than 15.45% of the variables and 3.32% of the edges. Moreover,

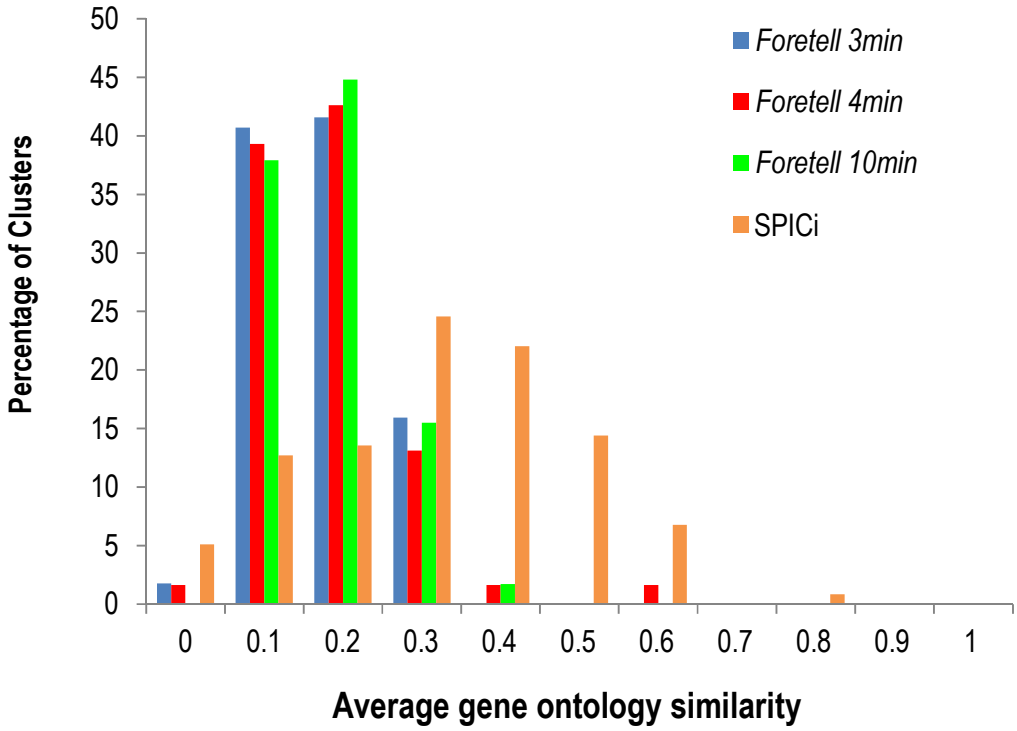


Figure 5.6. Distributions of average gene ontology similarity of clusters detected by *Foretell* and SPICi.

Foretell formed dense clusters. The density of the entire problem is 1.17%, but most clusters are either cliques or just a few edges shy of a clique, with an average density of about 92%. Thus *Foretell* found subsets of the variables that interact strongly with one another.

Finally, we compare the clustering results of *Foretell* with SPICi, a state-of-the-art clustering algorithm specifically designed for clustering proteins in protein-protein interaction network (Jiang and Singh, 2010). Gene ontology similarity is a metric that measures the functional similarity between two proteins. The quality of a cluster, therefore, can be measured by the average pairwise gene ontology similarity. Figure 5.6 shows the clustering quality of the three *Foretell* runs in Table 5.12 and SPICi.

Apparently, proteins clustered by SPICi are more functionally similar than those clustered by *Foretell*. The average similarity of SPICi clusters is 0.369. The average similarities of *Foretell*'s clusters are 0.227, 0.232, and 0.221 for the 10-minute, 4-minute, and 3-minute runs, respectively. Biological analysis, however, indicates that the clusters detected by *Foretell* are quite different from those detected by SPICi (Lei Xie, personal communication). For example, the largest cluster from the 10-minute *Foretell* run includes 87 proteins that are highly involved in forming protein complex (false discovery rate p-value < 0.05). This cluster is worthy of further biological investigation (Lei Xie, personal communication). Researchers can, therefore, use the clusters from *Foretell* to generate unique and testable hypothesis that may lead to new biological discovery.

5.4.2 Adaptation of *Foretell* to cluster protein interaction network

The above approach used the original *Foretell*, whose score function was designed specifically for CSPs, to cluster protein-protein interaction network. Given a *cluster Q*, let $|Q|$ be its size, C_Q be the set of constraints in Q, NC be the number of edges of a clique of size $|Q|$, $d(Q) = |C_Q|/NC$ be the density of Q, and D_i be the domain of variable i. The *Foretell* score for Q is

$$score(Q) = \frac{|Q| \cdot d(Q)}{\sum_{C_{ij} \in C_Q} |D_i||D_j|} = \frac{|Q| \cdot |C_Q|}{NC \sum_{C_{ij} \in C_Q} |D_i||D_j|} \qquad [17]$$

For this particular problem, $|D_i| = 1$ for all variables i and j, so [17] becomes

$$score(Q) = \frac{|Q| \cdot |C_Q|}{NC \sum_{C_{ij} \in C_Q} |D_i||D_j|} = \frac{|Q| \cdot |C_Q|}{NC \sum_{C_{ij} \in C_Q} 1} = \frac{|Q| \cdot |C_Q|}{NC \cdot |C_Q|} = \frac{|Q|}{NC} = \frac{2|Q|}{|Q|(|Q|-1)} = \frac{2}{(|Q|-1)} \qquad [18]$$

A non-trivial CSP never has singleton domains for all its variables because such a problem can be quickly checked for solvability. Thus, [18] would not occur for a CSP. When *Foretell* uses this score function for this biology problem, however, it basically selects clusters by their sizes and ignores other metrics. It is therefore possible to improve this score function to better fit this problem. We defined a new score function for a cluster as the product of the size, the density and the average tightness of the cluster,

$$score'(Q) = |Q| \cdot \frac{NAE}{NC} \frac{T}{NAE} = \frac{2T}{(|Q|-1)}, \text{ where } T = \sum_{C_{ij} \in C_Q} tightness(C_{ij}) \qquad [19]$$

[19] favors clusters with larger T and smaller size. Because larger clusters are likely to be more meaningful than small ones, an alternative score function drops the density factor. [20] favors clusters with larger size and higher average tightness.

$$score''(Q) = |Q| \cdot \frac{T}{NAE} \qquad [20]$$

We examined both new score functions for *Foretell*; the results were nearly identical on this fruit-fly protein-protein interaction network. This is because the densities of the clusters detected by *Foretell* were usually close to 1, so [19] and [20], therefore, produced similar scores. We, therefore, only report *Foretell* results with [19].

Figure 5.7 shows how distributions on average cluster tightness change when *Foretell* uses [19] instead of [18]. First, *Foretell* finds far more clusters using the new score function. The most important change is at the far right side of this figure: more clusters with average tightness greater than 0.8 are detected. The average gene ontology similarities of Foretell clusters are now comparable to those of the SPICi clusters. The average gene ontology (*GO*) similarity score distributions of the top thirty largest clusters are shown in Figure 5.8 for three GO categories: biological process, cellular component,

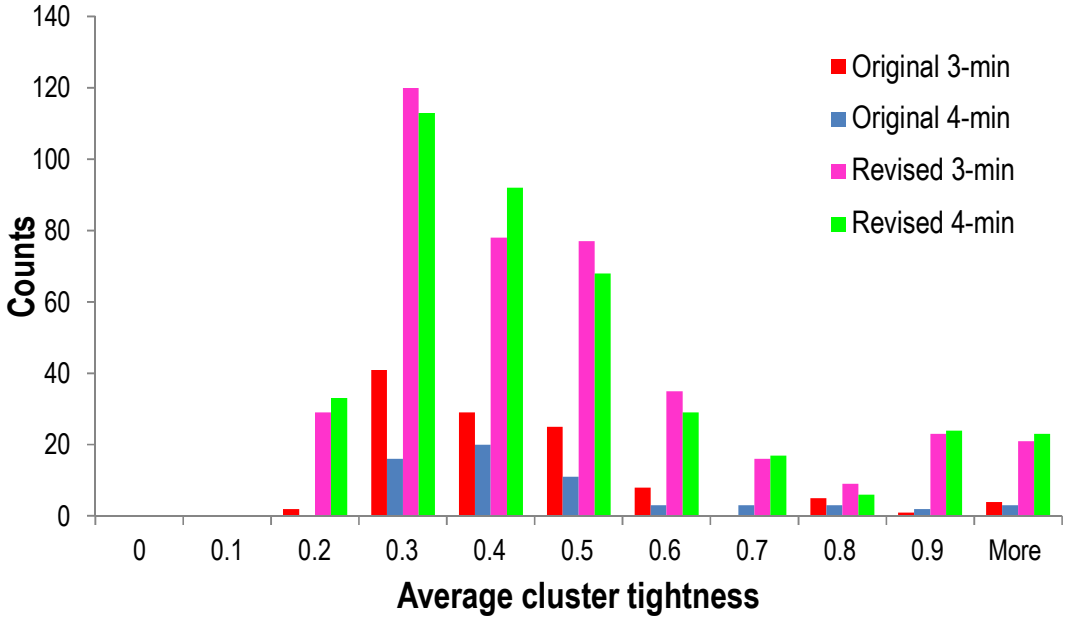


Figure 5.7. Distributions of average cluster tightness by *Foretell* using original [17] and revised [19] score functions.

and molecular function. Compared with the *Foretell* peaks in Figure 5.6, the new *Foretell* results in Figure 5.7 overlap closely with SPICi's result. Analysis shows that SPICi has the advantage of finding larger clusters while *Foretell*'s clusters have higher average tightness. Because biologically meaningful clusters are more likely to be both large and have high average tightness, the combination of the results of *Foretell* and SPICi can both prove informative to biologists.

Future work on the application of *Foretell* to cluster biological data will incorporate *a priori* biological domain knowledge. For example, we could predefine a set of biologically-meaningful variables (e.g., disease-causing genes) and have *Foretell* use them as seeds from which to build clusters of variables that are tightly connected to them.

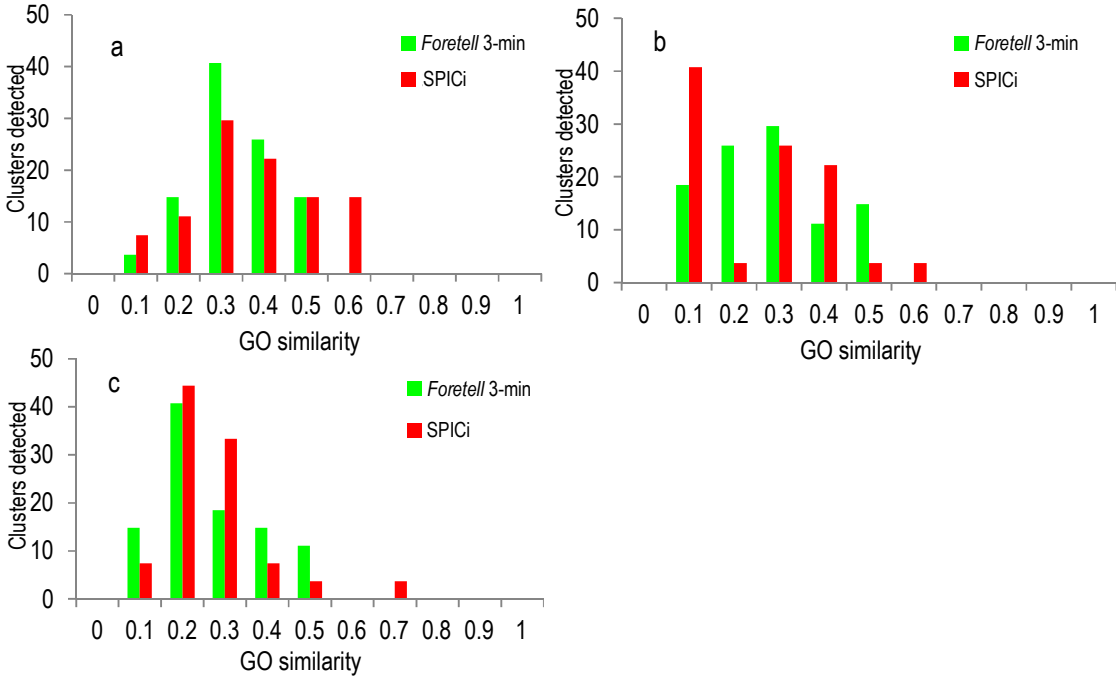


Figure 5.8. Distributions of cluster average gene ontology similarity for (a) biological process, (b) cellular component, and (c) molecular function, respectively, by SPICi and a 3-minute run of *Foretell* using *score'*(c) from [19]. X-axis values are GO similarity scores (higher is better) and y-axis values count clusters.

5.5 Summary

This chapter has presented empirical results for structure-based search on various problems. The structure identification algorithm *Foretell* can use either constraint tightness or weights collected during probing to build structure. First, tightness-based *Foretell* was shown to significantly improve search performance on both artificially generated problems with non-random structure and real-world problems. Then we explored weight-based versions of *Foretell* that worked with *MinDomWdeg* or *MinDomInfdeg* on a class of modified RLFAP problems. We showed that structure-based

search with the informed approach performed far better than the same search with the uninformed approach. *Foretell-IW* was in general the best method to solve the modified RLFAP problems. Finally, we showed a more general application of *Foretell* to cluster biological data. With some semantic adjustments, *Foretell* produced results comparable to those of a state-of-the-art biological clustering algorithm.

Chapter 6

Discussion and conclusion

This dissertation has presented a structure detection algorithm, *Foretell*, that adapts Variable Neighborhood Search to identify crucial substructures in constraint graphs. The adaptation employs new metrics, explores termination conditions, and searches repeatedly within the same problem. The principal result of this work is that structure detected by *Foretell* can guide search on CSPs with non-random structure and thereby significantly improve search performance. *Foretell* can also be used as a clustering algorithm, and has produced comparable results to a state-of-the-art clustering algorithm for a biology problem. In this chapter, we discuss this work and propose future work.

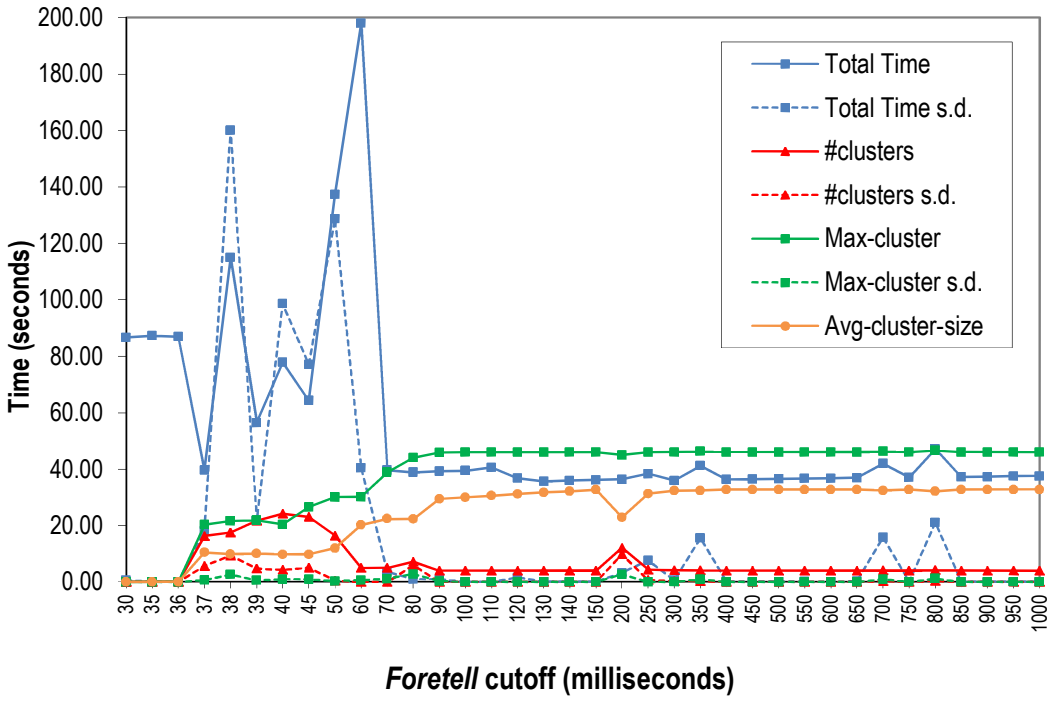
6.1 Early work

Density and tightness are synergistic. Earlier work (Epstein and Wallace, 2006) tested this approach on classes of smaller, considerably easier composed problems, ones that even the structure-unaware *MinDomDeg* could solve. A heuristic that prioritized variables by tightness roughly halved *MinDomDeg*'s search time. A heuristic that prioritized variables by density (with VNS-based near clique detection) consumed about a third of the search time. When combined using an earlier version of *Foretell,* however, density and tightness did an order of magnitude better, and produced nearly backtrack-

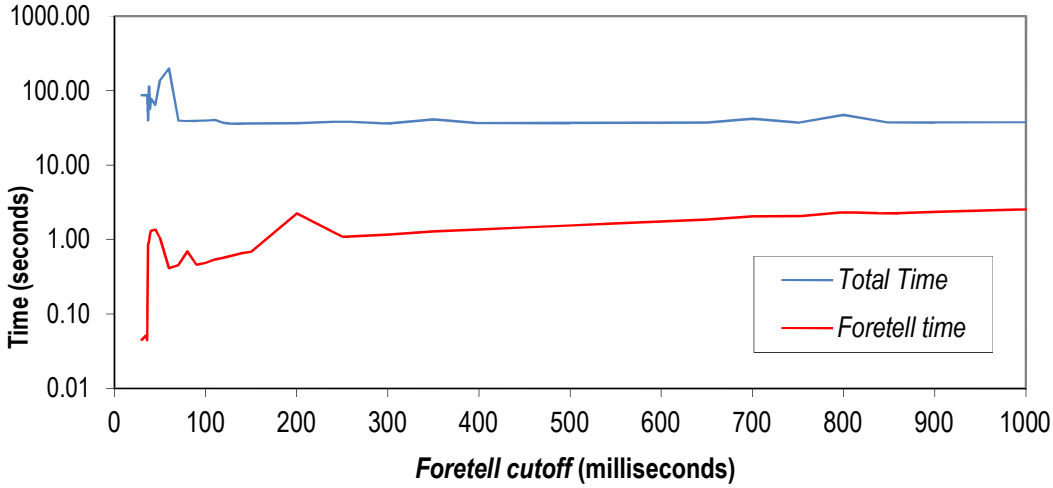
free search trees (Epstein and Wallace, 2006). On smaller composed problems with one or two satellites in the earlier study, the time that was spent to detect clusters was short and it did not reduce overall search performance, and cluster-guided search sometimes improved it. While earlier work and that reported here both use the same local search mechanism to find crucial substructure, they differ in two significant ways: the key measurement used to guide local search and the problems tested. The earlier work used *tension*, the dynamic reduction in the domains of the neighbors of a variable, while the tightness-based *Foretell* in Section 3.2.1 uses a more effective metric, *pressure* as defined in equation [9], an estimate of the probability that a variable's domain will be reduced when its neighbor is assigned a value. The earlier paper presented the initial *Foretell* and experimented only on smaller composed problems with extensional constraints generated by the authors. This dissertation improves the versions of *Foretell* based on static metrics, introduces new versions based on dynamic metrics, and emphasizes both large, real-world and benchmark problems, whose scale is closer to that of real-world problems. Together with *DrawCSP*, these problems provide considerable insight into the role of probing and structure in search for a solution.

6.2 Cluster detection with *Foretell*

Foretell's key parameter, *cutoff*, determines how much time to devote to the detection of any single cluster. *Foretell* has no prior knowledge about how many clusters lie within a problem or about how many might be necessary to solve it. We have found empirically that too few clusters provide inconsistent guidance, and that, as *cutoff* is increased,



(a)



(b)

Figure 6.1. Average results of 10 runs on driverlogw-08c. (a) The relationships among *cutoff*, the time allocated to find one cluster, and total time, number of detected clusters, maximum cluster size and average cluster size. Standard deviations are shown for total time, number of clusters and maximum cluster size. Total time includes both time to find all clusters and search time to solution. Note that the x-axis scale is not uniform. (b) The relationship between *Foretell* time and total time in logarithmic scale.

Foretell finds more consistent sets of clusters from one run to the next. Consider, for example, the experiments on driverlog-08c reported in Figure 6.1, where each data point represents 10 runs with a single *cutoff* value. At 30, 35 and 36 ms. per cluster, *cutoff* was too small — *Foretell* found no clusters, and the resultant performance was effectively that of *MinDomWdeg*. We then tested small *cutoff* values further. Values between 37ms and 80ms produced large variations in both the identified cluster graphs and the resultant search performance. *Foretell* found as many as 29 clusters on some runs at 38ms and 45ms. The best run, however, with *cutoff* = 37ms, found 18 clusters and solved the problem in 18.82 seconds, about half the time reported in Table 5.3. While 37ms and other *cutoff* values in that range do not perform well on average, the structure that led to the 18.82 second solution merits further study, and may ultimately motivate new heuristics able to support consistent performance at this level. Larger *cutoff* values produced more stable *Foretell* results.

As *cutoff* increases, the number of detected clusters stabilizes and decreases. Given more than 80 ms. the same largest cluster in driverlog-08c was found consistently. In general, there is no difference among the identified cluster graphs (Figure 5.1(b)) and search performance beyond 80 ms. Because total time is the time *Foretell* uses to find all clusters plus the time to search for a solution, increasing *cutoff* can increase total time. As Figure 6.1(b) shows, however, there is a wide range of cutoff values (from 37ms to 1000ms), where the time *Foretell* actually consumes is lower than this maximum allocation. Indeed, for that range of values, the cost of structure detection by *Foretell* is at least an order of magnitude lower than the cost of search.

In general, smaller *cutoff* values, although of interest for structure analysis and the design of new heuristics, are too aggressive to produce consistent performance. Larger cutoff values lead to consistent cluster graphs and benefit users who seek to understand the nature of these problems and to solve them effectively. Future work could address additional termination conditions for *Foretell* (line 4 in Figure 1.6). These include a Luby-like adaptive cutoff (Luby et al., 1993) for time allocation on successive clusters, and a limit on the percentage of variables that may be included in either an individual cluster or in the entire cluster graph.

6.3 Why *Focus* works

Variable-ordering heuristics usually do not consider persistence in a "geographic area" of a problem. Nonetheless, that was clearly *satellite*'s mistake — even with perfect foreknowledge about *Comp*, it *satellite hopped*, that is, it failed to address enough variables in the same satellite consecutively. *Stay* forbade satellite hopping and resulted in a considerable improvement. Analogously, *cluster hopping* occurs when a heuristic fails to address enough variables in the same cluster consecutively. Because constraints within a cluster are selected for above average tightness, once any variable in a cluster has been bound, propagation is likely to reduce the domains of the other variables in that cluster. As a result, variables in a partially-instantiated cluster are more likely to have smaller domain sizes and make their clusters even more attractive to cluster-guided search. *Tight* was permitted to cluster hop, while *Focus* and *Concentrate* both explicitly forbade it. *Focus* does better, however, because it uses knowledge about clusters to select one.

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

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

$$d\binom{s-w}{2} \leq s-w-1 \text{ ,that is, } w \geq s-\frac{2}{d} \qquad [21]$$

(Of course, this only makes a tree possible, not certain.) For *Comp* satellites, $s = 20$ and $d = 0.25$, so that there is no possibility of a tree unless $w \geq 12$, that is, we have bound 12 variables and 8 remain (*until*-8). Our empirical results, however, show that *until*-11 minimizes both time and nodes (using a one-tail t-test at the 95% confidence level). This ability to leave behind a (necessarily) cyclic subgraph is probably attributable to propagation. The occasional retraction back to a "finished" satellite proved less costly than binding a few more variables in the current satellite before moving on to the next one. Because clusters in *Comp* average $s = 4.309$, however, $w \geq 0.309$, that is, only *Focus*–0 is safe, which is exactly what our results indicate.

6.4 Clusters and search

The performance of perfect foreknowledge on *Comp*, as embodied by *until*-11, is the gold standard. *Until*-11 knows a superset of the backdoor and exploits it. Inspection indicates that the backdoor is probably no more than 35 variables for a *Comp* problem. The last retraction on any *Comp* problem under *Focus* was at a node where an average of 15.59 variables had been bound, with a maximum of 62.

The structural knowledge learned provides an explanation of where the difficulties lie in a CSP. Figure 2.5(a) focuses attention on the satellites, but the solution with *Focus* is even more descriptive: it searched within at most 3 satellites before it reported the insolvability of the problem in Figure 1.2. To demonstrate that, *Focus* bound only 12 variables (out of 200) drawn from 3 clusters found by *Foretell*. Those 3 clusters, two of size 5 and one of size 4, provide a concise and 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. Having clusters, however, does not justify directing computational resources to *Foretell*. Empirical results in Chapter 5 have repeatedly shown that it is faster to use simple heuristics, such as *MinDomWdeg* or even *MinDomDeg*, on easier problems. 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.

Both composed problems and real-world problems have non-random structure and varying tightness. Traditional heuristics like *MinDomDeg* perform poorly on composed

problems because of the difference in degree and tightness within the satellites and the central component. Composed problems provide an elegant, if artificial, argument for the need to consider tightness during search. Moreover, compared to real-world problems, composed problems offer a known structure that allows us to study (and aspire to) performance given perfect structural knowledge. (This is the point of the study of *Comp* in Section 3.1.) Moreover, composed problems' simpler structure makes it easier to monitor the entire search process and to understand the differences among various search regimens. What was learned from *Comp* applies to other artificial structured classes too, as shown in Chapter 5. While composed problems are built to confound traditional heuristics in a particular way, real-world problems are merely difficult. Not surprisingly, the performance improvements on real-world problems are noteworthy, but less dramatic. The insights provided by the structure detected by *Foretell*, however, could prove meaningful to a user. For example, people who know RLFAP Scene 11 well may be interested in *Foretell*'s detection of just 5 clusters, which include only 67 variables out of 680. Three of those clusters form a triangle in the cluster graph (Figure 3.2(b)).

Search with *Foretell* is a two-stage process: first, *Foretell* seeks clusters and then clusters guide search for a solution. We separated structure detection from structure guidance because, during search, retractions that back out of a cluster could demand frequent calls to find new clusters. This overhead could be expensive and reduce search performance. Thus *Foretell* is called only before search, and the structural knowledge learned by *Foretell* remains static during the subsequent search for a solution. This strategy appears to be effective for the problems we have tested.

Early departure from clusters during search on *Comp*, as with the *focus-i* heuristics in Section 4.1, has been shown to be less effective than *Focus*, a heuristic which entirely forbids early departure. The difference between *until-i* and *focus-i* is the structural knowledge on which they rely. *Until-i* uses perfect knowledge, which covers the entire satellite, but *focus-i* only considers the cluster(s) that partially cover the underlying satellite. Because of this partial coverage of satellites by clusters, *Focus* needs all the structural knowledge it can muster to achieve its best performance improvement. Future work may include how early departure from a cluster affects search on real-world instances, and the relation between subproblem coverage by clusters (where perfect structural knowledge about subproblems is available) and performance.

Methods to detect tight, dense subproblems must not only be incisive, they must also scale. Every real-world problem that we tested (i.e., all the RLFAP and driver problems) contained clusters that *Foretell* found fairly quickly. The *Foretell cutoff* for RLFAP scene11, which has roughly three times as many variables as a *Comp* problem, is about three times as large as the *Foretell cutoff* for *Comp*. This suggests that *Foretell* scales linearly with the number of variables in the problem.

6.5 Why weight-based *Foretell* works

Weight-based *Foretell* can discover structural knowledge that cannot be detected by tightness-based *Foretell*. The tightness of a constraint only shows the probability that a value-pair is acceptable to the constraint. Two constraints with the same tightness, however, can behave differently when they create wipeouts. For example, let constraint C_{ij} and constraint C_{pq} with respective scopes $\{X_i, X_j\}$ and $\{X_p, X_q\}$ have the same

tightness $t = 0.9$ (10% value-pairs accepted), and let $D_i = D_j = D_p = D_q = \{1, 2, \ldots, 10\}$. In an artificial (i.e., generated) problem, the 10 accepted value-pairs in constraint C_{ij} are randomly selected among the 100 possible value-pairs. Consider, in contrast, an intensional real-world constraint C_{pq}, $X_p > X_q + 5$, which also accepts 10 value-pairs, but those value-pairs, {(7, 1), (8, 1), (8, 2), (9, 1), (9, 2), (9, 3), (10, 1), (10, 2), (10, 3), (10, 4)}, are clearly not randomly selected from $|D_p| \times |D_q|$. Constraints in real-world problems are more often like C_{pq}. This kind of subtle difference between constraint C_{ij} and constraint C_{pq} is not evident from their tightness. Consequently, tightness-based structure cannot foresee the kind of conflicts that arise from these two constraints during inference. Probing, however, collects more explicit information through weights. Weight-based structure, therefore, reveals the true nature of contention better than tightness.

Both Table 5.7 and Table 5.9 show that *Foretell-IW* is in general superior to *Foretell-EW*. To understand influence-weight-based structure's superiority, consider Figure 6.2, which shows the difference between edge weights and influence weights after probing on scene11-f6. Each row of graphs plots the same data; on the left variables appear on two concentric circles, and on the right they appear on a single circle. (Section 2.1 has more details about these two representations.) The variables in modified RLFAP problems have smaller domains than those of the original problem, but the constraint graphs themselves are identical to those of the original problem. The two-circle plots emphasize the bipartite nature of tight constraints in this problem; the single-circle plots conceal these relationships so that other structure becomes more visible. The first row, Figures 6.2(a) and (b), shows the constraint graph for this problem. The second and the third rows show contention information collected by *MinDomWdeg* and *MinDomInfdeg*,

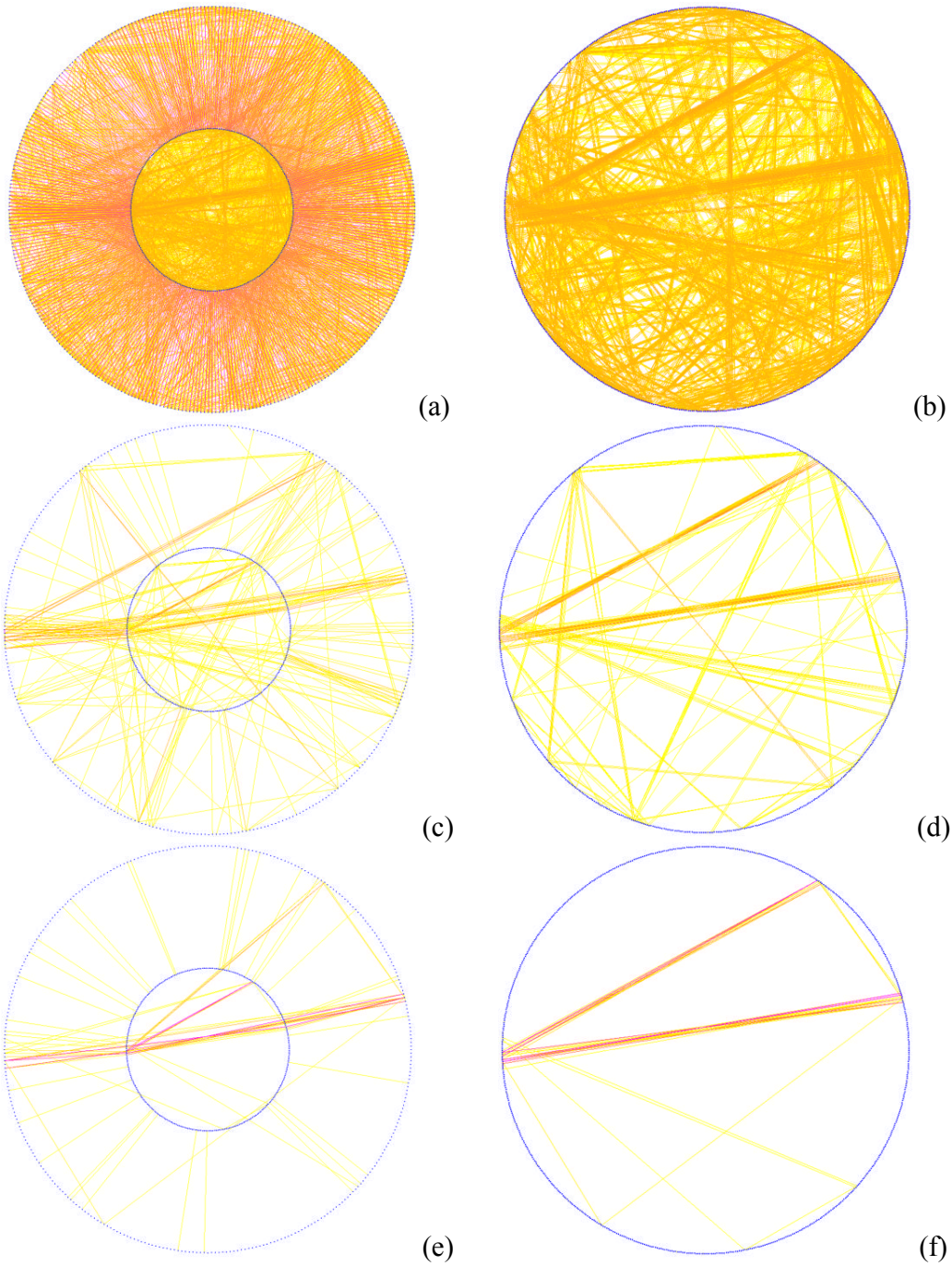


Figure 6.2. The double-circle (a, c, e) and the single-circle (b, d, f) constraint graphs for the rlfap-scene11-f6 problem. Constraints with (a)-(b) tightness, (c)-(d) edge weights (>1) learned after probing, and (e)-(f) influence weights (>1) learned after probing. Darker edge color (magenta) indicates higher tightness in (a) and (b) and higher weights in (c) – (f). Compared with wipeout-culprit edges, wipeout-instigated edges are fewer but have higher weights.

respectively. Similar to the coloring used here for constraint graphs, edge colors vary from yellow to magenta as the weights of the corresponding constraint increases, that is, darker edges represent constraints that caused more wipeouts. Only constraints with weights greater than 1 are shown in Figure 6.2 (c) through (f). There are two major differences between the second and the third rows: there are far fewer wipeout-instigated constraints than wipeout-culprit constraints, and the influence weights on some constraints are much higher than the edge weights on the same constraints. In other words, the contention information represented by influence weights is more condensed. If influence weights indeed identify the important constraints in contention and that set is smaller than the set of high-weighted edges, when search focuses its attention on the smaller set, it accelerates. (Section 5.3.5 has more details about this.) This may be a reason that *MinDomInfdeg*-based methods outperform *MinDomWdeg*-based methods.

6.6 Future work

The structure-based search introduced in this dissertation works well with CSPs that have certain kinds of constraint graphs, such as the driverlog problems and the RLFAP problems. There are, however, problems where *Foretell* cannot find any clusters at all or *Focus* cannot improve search performance with the detected structure. For problems with tree-like structure (e.g., Figures 2.3 and 2.4) it would be necessary to modify *Foretell* and *Focus* to detect that and use it to guide search. Other structures, such as lengthy cycles, can create search difficulty without local density (Markstrom, 2006). In general, domain-specific knowledge may be required for both *Foretell* and *Focus* to apply structure-based search to problems with other kinds of structures.

Not every problem needs *Foretell*. The "right" clusters are not necessarily many, but incisive, so *Foretell* could partition less than the entire problem. Other problems have more dense structures and may therefore respond better to other cluster-based orderings. For CSPs with more complex cluster graphs (e.g., groups of clusters connected by a path in Figures 2.1(d) and 5.1), future work includes the abstraction and detection of metastructure in a cluster graph, and the design of high-level heuristics that prioritize groups of clusters rather than individual ones.

In this research, we term a problem solved if a solution is found or its unsolvability is proved. We therefore terminated the solver as soon as it returned a solution. Users sometimes seek all solutions. Because clusters detected by *Foretell* have highly interactive and constrained variables, it is likely that there are fewer solutions for the subproblem induced by a cluster's variables than there are solutions for the entire problem, that is, a solution for the clusters alone may appear in more than one global solution. Therefore, after the solver binds all the cluster variables, it may become easier to find multiple global solutions by randomizing its variable-ordering heuristic.

Although the experiments described here are on binary CSPs, we see no obvious impediment to adapting this approach for non-binary constraints. Weight-based *Foretell* is not affected because non-binary CSPs can be probed to collect weights in the same way. For tightness-based *Foretell*, as long as there were some estimate of the tightness of a constraint, it would be possible to estimate the pressure on a variable and to detect sets of mutually-constrained variables by local search. The swap and score functions would require only some modification.

Because 2SAT problems are binary CSPs, it would be possible to compare the performance of structure-based search with that of SAT solvers. With the extension of *Foretell* to non-binary CSPs, structure-based search can be used to solve general SAT problems, which are non-binary CSPs with the same binary domain for every variable.

Other work introduces the concept of *impact*, the influence of search space reduction, for every variable (Refalo, 2004), and uses a matrix-based representation to visualize pair-wise impacts between variables (Cambazard and Jussien, 2006). On artificial instances with tighter subproblems, this visualization shows such structure. It would be interesting to compare the contention structure learned by *Foretell* before search and the impact structure learned during search.

Ways to allocate time between probing and search are also worthy of future study. Easier problems do not need any probing, while harder problems may need more probing. Identification of an appropriate *cutoff* for *Foretell* is a similar challenge. Parallelization may help to resolve these issues. One could assign structure detection, probing and search to different processors with proper inter-processor communication and synchronization.

The successful adaptation of *Foretell* to cluster biological data in Section 5.4 suggests that *Foretell* can be used a pure graph clustering algorithm as long as the graph has weights on its edges. It would be interesting to compare *Foretell* with other clustering algorithms on large networks. Our experience in Section 5.4 tells us that proper adaption of *Foretell* with domain knowledge is critical to its performance.

6.7 Conclusion

ACE, the solver that this work used, is not honed for speed. Nonetheless, the concomitant reductions in checks and nodes searched, which are platform and solver independent, suggest that clusters will accelerate other, more agile solvers as well. For an easy problem, no clusters are necessary, and any reasonable amount of time spent on cluster detection will have no noteworthy impact. For more challenging problems, however, structure-based search substantially accelerated search with off-the-shelf heuristics. No solver, human or machine, has an efficient way to "see" Figure 2.1(d) perfectly without knowledge about the problem's semantics.

Given its acuity and explanatory ability, clustering with *Foretell* is a worthwhile preprocessing step. In general, *Foretell* predicts significant structure even where global search is likely to fail. A user confronted with an unsolvable real-world problem could use that structure to reconsider the problem's specifications, or at least to understand why a problem is difficult to solve or has no solution at all. We focused here on structure-based search for CSPs, but *Foretell* alone can be applied to any data that can be represented as a weighted graph and needs clustering.

The thesis of this dissertation is that it is possible and beneficial to

- identify crucial structures for solution in a constraint satisfaction problem before search
- exploit that knowledge to solve the problem
- explain the problem's inherent difficulties with that knowledge.

The novelty of the approach taken here is that *Foretell* adapts a local search metaheuristic, Variable Neighborhood Search, to identify cluster within CSPs. The

algorithms presented in this dissertation are the first to use either constraint tightness or constraint weights learned during probing as the metrics to select variables to build clusters

The major contributions of this research are

- *DrawCSP*, a visualization tool for binary CSPs
- tightness-based and weight-based versions of *Foretell* for structure detection in CSPs
- cluster-based variable-ordering heuristics to guide search with detected structure.

The principal results of this research are

- Rapid detection of *CSP* structures called *clusters* before search is possible.
- The exploitation of clusters solves problems more effectively.

Finally, the conclusion we draw from this dissertation is that structure-based search not only improves search performance, but also provides insight into the nature of the problem in a user-friendly representation.

Glossary

ACE: Adaptive Constraint Engine

ACR-k: inference strategy that begins with the same initial queue as MAC-3, but subsequently enqueues only constraints on variables whose dynamic domains lose at least (100*k*)% of their values

Central component: see Composed CSP

CFC-AC: inference strategy that uses *Cluster-FC* on constraints that inside or between clusters and MAC-3 on all other edges

Cluster: set of densely connected variables whose domains are likely to reduce during search

Cluster-FC: inference strategy that uses a queue (similar to the one used in MAC) to keep track of edges that need to be propagated. When *Cluster-FC* infers along a link edge and reduces the domain of the variable on the other side of the link, it enqueues all other constraints between that variable and an unassigned variable. *Cluster-FC* does more inference work than *FC*; it revisits those enqueued constraints while *FC* does not.

Cluster graph: constraint graph of the subproblem induced by the variables in the detected clusters

Comp: class of artificially-generated composed CSPs with parameters $<100,10,0.15,0.05>5<20,10,0.25,0.50>0.12,0.05$

Composed CSP: class of artificially-generated composed CSPs with parameters $<n,k,d,t>s<n',k',d',t'>d'',t''$, where $<n,k,d,t>$ represents its central component, $<n',k',d',t'>$ represents its s satellites and d'' and t'' represent the density and tightness of the links between the central component and satellites

Concentrate: variable-ordering heuristic that randomly selects a cluster and forces search to select variables from the selected cluster until it is solved

Concentrate-i: variable-ordering heuristic that randomly selects a cluster and forces search to select variables from the selected cluster until all but *i* variables have been bound

Cutoff: time that *Foretell* may devote to the detection of a single cluster

CSP: Constraint Satisfaction Problem

DrawCSP: visualization tool for binary CSPs

Dynamic degree: degree of an unassigned variable after all its assigned neighbors have been removed from the problem

Edge weight: weights on constraints that are used and maintained by weighted-degree-based heuristics during search

Focus: variable-ordering heuristic that always selects the tightest cluster and forces search to select variables from the selected cluster until it is solved

Focus-i: variable-ordering heuristic that always selects the tightest cluster and forces search to select variables from the selected cluster until all but *i* variables have been bound

Foretell: algorithm that adapts VNS to identify crucial structure in CSPs

Foretell-EW: adaptation of *Foretell* that uses the distribution of contention, reflected by edge weights learned in probing before search, to detect structural knowledge

Foretell-IW: adaptation of *Foretell* that uses the distribution of contention, reflected by influence weights learned in probing before search, to detect structural knowledge

Forward checking (*FC*): inference strategy that, immediately after the instantiation of a variable *x*, removes all inconsistent values from the domain of every uninstantiated neighbor of *x*

Future variable: variable that has not yet been assigned a value during search

Heuristic probing: probing that updates constraints weights and uses them to guide search

Influence weight: weights on constraints that are used and maintained by influence-degree-based heuristics during search

Influence degree (*Infdeg*): conflict-directed metric on variables. At any time during search, the influence degree of a variable X_i is the sum of the influence weights of all constraint and invisible constraint whose scopes include X_i and some future variable

Informed approach: weight learning strategy that uses heuristic probing and reasoned restart, and continues to update weights during search

Invisible constraint: edge, created to connects the wipeout instigator and the wipeout victim if no such constraint already exists (See Figure 3.1 for more details)

MAC-3: inference strategy that maintains a problem's arc consistency during search

MinDom: variable-ordering heuristic that prefers variables with the smallest dynamic domains

MinDomDeg: variable-ordering heuristic that prefers variables that minimize their ratio of dynamic domain size to dynamic degree

MinDomWdeg: variable-ordering heuristic that prefers variables that minimize the ratio of dynamic domain size to weighted degree

MaxDeg: variable-ordering heuristic that prefers variables with the largest dynamic degrees

MaxWdeg: variable-ordering heuristic that prefer variables with the highest weighted degrees

Pressure: probability that, given all the constraints upon it, when one of a variable's neighbors is assigned a value, at least one value will be excluded from this variable's domain

Probing: a sequence of short and quick restarts that explores the search space before an unlimited search for solution

Promise: value-ordering heuristic prefers values that maximize the product of the domain sizes of the future variables after choosing this value

Random probing: probing that updates constraint weights but selects variables randomly

Random restart: probing that restarts with randomly selected variables

Reasoned restart: probing that restarts with variables that are favored by variable-ordering heuristic

Satellite (heuristic): variable-ordering heuristic based on perfect structural foreknowledge. It randomly selects satellite variables before any central-component variable is assigned.

Satellite (structural component): see Composed CSP

Stay: variable-ordering heuristic based on perfect structural foreknowledge. It addresses entire satellites, in a random order, one at a time before it selects any variable in the central component. It uses *MinDomDeg* within a satellite and within the central component. *Stay* is equivalent to *until*-0.

Tension: dynamic domain reductions of the neighbors of a variable during search

Tight: cluster-selecting heuristic that always selects the tightest cluster

Uninformed approach: weight learning strategy that uses random probing, random restart and freezes weights during search

Until-i: variable-ordering heuristic based on perfect structural foreknowledge. It instantiates variables within a randomly chosen satellite until all but *i* variables have been bound, and then moves on to another randomly chosen satellite.

Variable Neighborhood Search (*VNS*): search metaheuristic that uses multiple neighborhoods of increasing size in local search

Weighted degree (*Wdeg*): conflict-directed metric on variables. At any time during search, the weighted degree of a variable X_i is the sum of the edge weights of all constraints whose scopes include X_i and some future variable.

Wipeout: when a variable's domain becomes empty during inference

Bibliography

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

Beck, J.C., A.J. Davenport, E.M. Sitarski and M.S. Fox, 1997. Beyond contention: Extending texture-based scheduling heuristics. In Proceedings of *The Fifteenth National Conference on Artificial Intelligence (AAAI-1997)*, pp. 233-240, Providence, Rhode Island.

Berghen, F.V., 2009. *Small, simple, cross-platform, free and fast C++ XML parser*, http://www.applied-mathematics.net/tools/xmlParser.html

Bessière, C., A. Chmeiss and L. Saîs, 2001. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In Proceedings of *Principles and Practice of Constraint Programming (CP2001)*, pp. 565-569, Paphos, Cyprus.

Bessiere, C. and J. Regin, 1996. Mac and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In Proceedings of *Principles and Practice of Constraint Programming (CP1996)*, pp. 61-75, Cambridge, Massachusetts.

Boussemart, F., F. Hemery, C. Lecoutre and L. Sais, 2004. Boosting systematic search by weighting constraints. In Proceedings of *the Sixteenth European Conference on Artificial Intelligence (ECAI-2004)*, pp. 146-150, Valencia, Spain.

Brin, S. and L. Page, 1998. The anatomy of a large-scale hyertextual web search engine. *Computer Netowrks and ISDN Systems*, 30: 107-117.

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

Cambazard, H. and N. Jussien, 2006. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11: 295-313.

Chung, F. and A. Tsiatas, 2010. Finding and visualizing graph clusters using pagerank optimization. In Proceedings of *the 7th Workshop on Algorithms and Models for the Web Graph (WAW 2010)*, Stanford University, California.

Cohen, D.A. and M.J. Green, 2006. Typed guarded decompositions for constraint satisfaction. In Proceedings of *Principles and Practice of Constraint Programming (CP2006)*, pp. 122-136, Nantes, France.

Cook, S.A., 1971. The complexity of theorem-proving procedures. In Proceedings of *The Third Annual ACM Symposium on Theory of computing*, Shaker Heights, Ohio.

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

Dechter, R. and D. Frost, 2002. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136: 147-188.

Dechter, R. and I. Meiri, 1989. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In Proceedings of *IJCAI-89*, pp. 271-277, Detroit, Michigan.

Dechter, R. and J. Pearl, 1987. The cycle-cutset method for improving search performance in ai applications. In Proceedings of *The Third IEEE on AI Applications*, pp. 224-230, Orlando, Florida.

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 *Principles and Practice of Constraint Programming (CP2007)*, pp. 256-270, Providence, Rhode Island.

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

Epstein, S.L. and R.J. Wallace, 2006. Finding crucial subproblems to focus global search. In Proceedings of *The IEEE International Conference on Tools with Artificial Intelligence (ICTAI-2006)*, pp. 151-159, Washington, D.C.

Freuder, E.C., 1982. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1): 24-32.

Freuder, E.C., 1994. Exploiting structure in constraint satisfaction problems. In Proceedings of *Constraint Programming: NATO Advanced Science Institute Series*, pp. 54-79, Parnu, Estonia.

Frey, B.J. and D. Dueck, 2007. Clustering by passing messages between data points. *Science*, 315: 972-975.

Gaschnig, J., 1979. Performance measurement and analysis of certain search algorithms, In *Technical Report CMU-CS-79-124*. Carnegie Mellon University, Pittsburgh, Pennsylvania.

Geelen, P.A., 1992. Dual viewpoint heuristics for binary constraint satisfaction problems. In Proceedings of *The Tenth European Conference on Artificial Intelligence (ECAI '92)*, pp. 31-35, Vienna, Austria.

Gent, I., C. Jefferson, T. Kelsey, I. Lynce, I. Miguel, P. Nightingale, B.M. Smith and S.A. Tarim, 2007. Search in the patience game 'black hole'. *AI Communications*, 20(3): 211-226.

Gent, I., E. MacIntyre, P. Prosser, B. Smith and T. Walsh, 1996. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In Proceedings of *Principles and Practice of Constraint Programming (CP1996)*, pp. 179-193, Cambridge, Massachusetts.

Gent, I.P., E. MacIntyre, P. Prosser, B. Smith and T. Walsh, 2001. Random constraint satisfaction: flaws and structure. *Constraints*, 6: 345-372.

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 *The Eighteenth International FLAIRS Conference (FLAIRS-2005)*, pp. 167-174, Clearwater Beach, Florida.

Grimes, D. and R.J. Wallace, 2007. Learning to identify global bottlenecks in constraint satisfaction search. In Proceedings of *The Twentieth International FLAIRS Conference (FLAIRS-2007)*, pp. 592-598, Key West, Florida.

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*. In Handbook of metaheuristics, F. W. Glover and G. A. Kochenberger, (Eds.). Springer, Berlin, pp: 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. Elliot, 1980. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14: 263-313.

Hemery, F., C. Lecoutre, L. Sais and F. Boussemart, 2006. Extracting MUCs from constraint networks. In Proceedings of *The Seventeenth European Conference on Artificial Intelligence (ECAI-2006)*, pp. 113-117, Riva del Garda, Italy.

Hoos, H.H. and T. Stutzle, 2005. *Stochastic local search: Foundations and applications*. Morgan Kaufmann, San Francisco, California.

Jégou, P., 1993. Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In Proceedings of *The Eleventh National Conference on Artificial Intelligence (AAAI-1993)*, pp. 731-736, Washington, DC.

Jiang, P. and M. Singh, 2010. SPICi: A fast clustering algorithm for large biological networks. *Bioinformatics*, 26(8): 1105-1111.

Junker, U., 2004. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In Proceedings of *The Nineteenth National Conference on Artificial Intelligence (AAAI-2004)*, pp. 167-172, San Jose, California.

Kroc, l., A. Sabharwal and B. Selman, 2008. Counting solution clusters in graph coloring problems using belief propagation In Proceedings of *The Twenty-Second Annual Conference on Neural Information Processing Systems (NIPS-2008)*, pp. 873-880, Vancouver, Canada.

Lecoutre, C., 2009. *Benchmarks - XML representation of CSP instances*, http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html

Long, D. and M. Fox, 2002. *The third international planning competition*, http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/long03a-html/node37.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.

Mackworth, A.K. and E.C. Freuder, 1993. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59: 57-62.

Markstrom, K., 2006. Locality and hard SAT-instances. *Journal on Satisfiability, Boolean Modeling and Computation*, 2: 221-227.

Mehta, D., B. O'sullivan, L. Quesada and N. Wilson, 2009. Search space extraction. In Proceedings of *Principles and Practice of Constraint Programming (CP2009)*, pp. 608-622, Lisbon, Portugal.

Minton, S., M. Johnston, A.B. Philips and P. Laird, 1990. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In Proceedings of *AAAI*, pp. 17-24, Boston, Massachusetts.

Pearl, J., 1988. *Prbabilistic reasoning in intelligent systems*. Morgan Kaufmann, San Francisco, California.

Razgon, I. and B. O'Sullivan, 2006. Efficient recognition of acyclic clustered constraint satisfaction problems. In Proceedings of *The Eleventh Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP2006)*, Caparica, Portugal.

Refalo, P., 2004. Impact-based search strategies for constraint programming. In Proceedings of *Principles and Practice of Constraint Programming (CP2004)*, pp. 556-571, Toronto, Canada.

Roussel, O. and C. Lecoutre, 2008. *XCSP 2.1: A format to represent CSP/QCSP/WCSP instances*, http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html

Ruan, Y., E. Horvitz and H. Kautz, 2004. The backdoor key: A path to understanding problem hardness. In Proceedings of *The Nineteenth National Conference on Artificial Intelligence (AAAI-2004)*, pp. 124-130, San Jose, CA.

Sabin, D. and E.C. Freuder, 1997. Understanding and improving the mac algorithm. In Proceedings of *Principles and Practice of Constraint Programming (CP1997)*, pp. 167-181, Linz, Austria.

Sadeh, N., 1991. Lookahead techniques for micro-opportunistic job-shop scheduling, Ph.D. thesis, Carnegie-Mellon University.

Samer, M. and S. Szeider, 2006. Constraint satisfaction with bounded treewidth revisited. In Proceedings of *Principles and Practice of Constraint Programming (CP2006)*, pp. 499-513, Nantes, France.

Simonis, H., P. Davern, J. Feldman, D. Mehta, L. Quesada and M. Carlsson, 2010. A generic visualization platform for CP. In Proceedings of *Principles and Practice of Constraint Programming (CP2010)*, pp. 460-474, St Andrews, UK.

Smith, B.M., 1999. The brélaz heuristic and optimal static orderings. In Proceedings of *Principles and Practice of Constraint Programming (CP1999)*, pp. 405-418, Alexandria, Virginia.

Smith, B.M. and S.A. Grant, 1998. Trying harder to fail first. In Proceedings of *The 13th European Conference on Artificial Intelligence (ECAI-98)*, pp. 249-253, Brighton, UK.

van Dongen, S., 2000. Graph clustering by flow simulation, Ph.D. thesis, University of Utrecht.

Wallace, R.J., 2005. Factor analytic studies of CSP heuristics. In Proceedings of *Principles and Practice of Constraint Programming (CP2005)*, pp. 712-726, Sitges, Spain.

Weigel, R. and B. Faltings, 1999. Compiling constraint satisfaction problems. *Artificial Intelligence*, 115: 257-287.

Willimans, R., C.P. Gomes and B. Selman, 2003. On the connections between heavy-tails, backdoors, and restarts in combinatorial search. In Proceedings of *The Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, Portofino, Italy.

Zheng, Y. and B.Y. Choueiry, 2005. Applying decomposition methods to crossword puzzle problems. In Proceedings of *Principles and Practice of Constraint Programming (CP2005)*, pp. 874, Sitges, Spain.