TWO APPROACHES TO THE HIERARCHICAL SOLUTION OF CONSTRAINT SATISFACTION PROBLEMS

BY SUNIL KUMAR MOHAN

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Prof. Thomas Ellman

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 1996

ABSTRACT OF THE DISSERTATION

Two Approaches to the Hierarchical Solution of Constraint Satisfaction Problems

by Sunil Kumar Mohan, Ph.D.

Dissertation Director: Prof. Thomas Ellman

Finite Constraint Satisfaction Problems (CSPs) involve assigning values to a finite set of variables from their finite domains to satisfy a finite set of constraints. Scheduling is a common application. All general CSP solution procedures use combinatorial enumeration of variable bindings as a basis for search. CSPs, however, are NP-hard, and general-purpose search algorithms are slow. This research explores the application of decomposition to construct faster CSP solvers. The solution techniques developed in this thesis can be used on general finite n-ary CSPs.

There are three major components to this research:

- *Bottom-up solution*, a framework for solving a CSP through its decomposition. It is aimed at handling decompositions with interacting subproblems, and has been implemented on top of the basic Backtrack and Forward-Check search algorithms.
- Two problem *decomposition algorithms* aimed at reducing redundant constraint checks. These algorithms seek to reduce *artificial dependencies* in search, an artifact of combinatorial enumeration. The resulting decompositions can be solved using bottom-up solution.

- *Global constraint decomposition*, used to transform a CSP into an equivalent hierarchical CSP in which the global constraint has been replaced by a conjunction of smaller arity constraints. This improves the potential for early pruning of combinatorially explosive search.

The problem decomposition techniques are evaluated on random problems and on some sample application domains. They are shown to be generally more beneficial for harder problems. Global constraint decomposition is demonstrated on some sample application domains, and shown to significantly reduce search effort.

The primary contribution of this research is in the form of new problem solving methods for general constraint satisfaction problems which significantly improve performance, particularly for harder problems. It also extends the current understanding of what makes constraint satisfaction problems difficult to solve and where search algorithms spend their effort. On the more general side, this thesis promotes a deeper understanding of the application and benefits of problem decomposition as a problem solving strategy.

Acknowledgements

This research was started under the guidance of Professor Chris Tong, and completed under the guidance of Professor Tom Ellman. I would like to thank Chris and Tom for their patient support, focus, and the many long hours of discussion, on search, within and without.

This research greatly benefitted from the support of Professor Eugene Freuder, who always asked the right question; my thanks for helping me bring this work into sharper focus and placing it in context. I would also like to thank the members of my committee, Professors Bill Steiger and Andrew Gelsey, for their patient reviews and valuable suggestions which helped improve this thesis.

My thanks also go out to the faculty, staff and my fellow students at Hill Center, for making my stay exciting and memorable, and especially Janet Willis and Valentine Rolfe, for making it a home away from home.

An endeavour like this is never a day job. I would like to thank my friends and various roommates, my tennis and basketball buddies, and particularly Arvind, Mike, Bob and Mark, for their companionship.

I especially thank my family, who made it all possible, and my wife Debbie, for her unending love, patience and understanding.

This research was supported by the National Science Foundation (Grants IRI-9017121 and IRI-9021607).

Dedication

To my family and friends, who gave me roots,
And my teachers, who gave me wings.

"And the night shall be filled with music ..."

The day is done, H.W. Longfellow.

Table of Contents

List of Figures

Chapter 1

Introduction

This is a report on research into applying decomposition to aid the solution of constraint satisfaction problems. The decomposition techniques described here are aimed at improving problem solving performance, are automatable, and are applicable to problems with constraints of arbitrary arity, a superset of the much studied class of binary constraint satisfaction problems. This chapter introduces this thesis and its motivation, and gives an overview of the research.

1.1 Constraint Satisfaction Problems

1.1.1 Finite CSPs

A (finite) Constraint Satisfaction Problem (commonly abbreviated as CSP) consists of a finite set of variables, a domain of a finite set of legal values for each variable, and a finite set of constraints expressed on the variables. A solution requires the assignment of a value to each variable, selected from the corresponding domain, such that the set of constraints is satisfied. Constraint satisfaction problems have found application in design (e.g. [52, 39]), planning and scheduling (e.g. [12, 36]). Job-shop scheduling, graph coloring and satisfiability ([24]) are some examples of CSP applications that have also been of considerable theoretical interest through the years.

In job-shop scheduling, for example, the variables are operation start times, and the constraints represent restrictions on resource capacities and operation precedences. In graph coloring, each variable represents a node in the graph, its domain the set of legal colors for that node, and constraints restrict adjacent nodes from taking the same color.

1.1.2 Constraint Satisfaction and Constrained Optimization

The significance of constraint satisfaction problems can perhaps best be understood by comparing it with the well known and related class of optimization problems. The main difference between Constrained Optimization Problems (abbreviated as COPs) and CSPs is the requirement in the former that the solution be optimal with respect to some objective function. Discrete constrained optimization problems further require their solution to be selected from a set of discrete values. The subclass of finite discrete COPs, also called (finite) Integer Programming or IP, is directly related to the class of finite CSPs as defined above. Discrete optimization problems, particularly those with non-linear constraints, are also the most difficult type of optimization problem to solve ([53]). Often the only solution procedure is to simply enumerate all the solutions and then select the optimal. A user may choose to formulate an application as a CSP rather than a COP when

- There is no optimality requirement. In this case any solution satisfying the constraints is sufficient.
- The optimality requirement is inexpressible. The user may then want to inspect several or all solutions to the CSP.
- The optimization problem is too difficult to solve. The user may then be satisfied with selecting a sub-optimal solution from a small set of solutions to the CSP, such as those produced within a set time.

1.1.3 Types of CSPs

Constraint satisfaction problems can be divided into subclasses using several criteria based upon the classifier's interest. There are two such criteria important to this thesis:

1. *Constraint Arity.* Problems where each constraint restricts only up to two variables are called Binary CSPs. Several problem solution techniques and heuristics take advantage of this property (e.g. [18, 10, 47]). The problem solution methods proposed in this thesis are more general and so we shall consider problems not

restricted on the arity of their constraints. These problems are popularly referred to as 'n-ary CSPs'.

2. *Number of Solutions.* Using constraint satisfaction techniques to solve optimization problems usually requires the generation of several solutions. A satisfiability problem, on the other hand, is considered solved when any one solution is found. The decomposition techniques proposed in this thesis will usually provide greater benefit when several or all solutions to a problem are called for. Some situations when this might be necessary are described in section 1.1.2.

1.1.4 An example: Simple floorplanning

To help introduce the ideas in this thesis, we now take a look at an example application. A simple floorplanning problem requires the placement of two rectangular rooms, each of a specified minimum area, into a house represented as a rectangle of specified length and width. The room sides are to be parallel to the house sides so we can represent each room with the coordinates of its bottom-left and top-right corners. The dimensions of the house then define the domain for each corner coordinate, and we will restrict the values to be integral. Four variables are needed to represent the two corners of each room, requiring a total of 8 variables.

$$\begin{aligned} \text{House dimensions} &= L \times W \\ \text{Room } i &= (r_i.x_1, r_i.y_1), (r_i.x_2, r_i.y_2) \\ r_i.x_j &\in \{0 \ldots L\} \\ r_i.y_j &\in \{0 \ldots W\} \end{aligned}$$

The constraints require that the rooms not overlap, and each room to be of a certain area. To avoid repeated generation of the same room, we insert additional constraints restricting the top-right corner of a room to be above and to the right of its bottom-left

corner.

RightOf-r1$(r_1.x_2, r_1.x_1)$	RightOf-r2$(r_2.x_2, r_2.x_1)$
Above-r1$(r_1.y_2, r_1.y_1)$	Above-r2$(r_2.y_2, r_2.y_1)$
HasArea-r1$(r_1.*)$	HasArea-r2$(r_2.*)$
DontOverlap$(r_1.*, r_2.*)$	

We use the informal notation $r_i.*$ to refer to all four variables for room r_i. *DontOverlap* has 8 arguments, four for each room, *HasArea* has four arguments for a room's four corner coordinates, and *Above* and *RightOf* are each of arity two. In the complete problem, there are two instances of each of the *HasArea, Above* and *RightOf* constraints, one for each room. We indicate this with a suffix of the room number on the constraint name. Our problem thus has a total of eight variables and seven constraints. One of the constraints, *DontOverlap*, is global. (An expanded version of this problem is formally described in Chapter 6).

The constraint satisfaction problem then is to find the members of the set P_{f2}. Figure 1.1 shows an instance of the general problem P_{f2} and a sample floorplan layout, with a house of length 9 and width 6, and minimum room areas of 20 each. For the rest of this chapter, references to the problem P_{f2} will be to this particular instance of the parameterized two-room floorplanning problem.

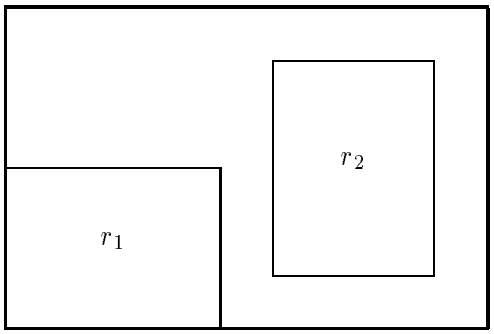
$$\begin{aligned} P_{f2} \quad = \quad \{\langle r_1.*, r_2.*\rangle : & \text{RightOf-r1} \wedge \text{Above-r1} \wedge \text{HasArea-r1} \\ & \wedge \text{RightOf-r2} \wedge \text{Above-r2} \wedge \text{HasArea-r2} \\ & \wedge \text{DontOverlap}\} \end{aligned}$$

In our sample application, we shall assume that the designer of the house will select a floorplan on consulting the buyer and other inexpressible and aesthetic constraints. So the floorplanning problem solver is required to produce all solutions.

1.2 Constraint Satisfaction Problems are Difficult

1.2.1 CSPs and in-tractability

The 3-SAT satisfiability problem, the K-colorability of a graph, and the job-shop scheduling problem are all NP-Complete ([24]). The class of constraint satisfaction



Problem P_{f2}:

House $L = 9,\ W = 6$

Area(r_1) ≥ 20

Area(r_2) ≥ 20

Figure 1.1: The two-room floorplanning problem P_{f2} and a solution.

problems includes all of these, and is NP-Hard. Not surprisingly, every general algorithm for solving CSPs is a clever form of enumerating, or searching through the space of, possible solutions.

Certain types of CSPs have been found to be easily solvable, e.g. systems of linear equations, and arc-consistent graphs of width 1 ([18]). On the other extreme, for example, a CSP with a single (n-ary) constraint expressed on all n variables, each with a domain of size d, is likely to require the enumeration of all d^n candidates to find all solutions. In this thesis, we look at problems in the middle, which are hard, but can benefit from clever search.

There has been some recent interest in identifying particularly hard problem regions, e.g. in graph colorability ([7]) and satisfiability problems ([43]). For our purposes, hard problems have larger, and therefore bushy search trees (see also Chapter 2). The proposed solution techniques try to take advantage of this feature.

1.2.2 Serialized search leads to redundant computation

Most general CSP solving algorithms search for a solution by systematically enumerating the set of possible variable bindings. While different algorithms use different mechanisms for avoiding unnecessary computation, what they share in common is the serial order in which variables are selected for binding, and constraints for evaluation.

The algorithm Backtrack, for example, selects at each step the next variable and binds a new value to it. As soon as all of a constraint's arguments are bound, it is tested. If a constraint fails, or a variable's domain is exhausted, the algorithm backtracks to the last variable from which it can proceed forward again. Backtrack avoids enumerating the complete set of variable bindings (the cartesian product of the variables' domains) when employed on a suitable order of variables in which constraints can be tested early in the serial control order. When a constraint fails, there is no need to extend the current set of bindings since they will not lead to a solution. A more detailed description of Backtrack is given in Chapter 2.

In such enumerative or constructive algorithms, the number of times a constraint may be tested depends not only on the number of unique bindings of the variables occurring as its arguments, but on a possibly expanded set including other variables also bound and rebound before the constraint gets tested. The result is redundant evaluation of constraints on repeated argument bindings. This *artificial dependency* of a constraint on variables not directly affecting its evaluation result is a direct outcome of the serialized nature of combinatorial enumeration.

The degree of artificial serial dependencies in an algorithm instantiation depends upon the problem's constraint topology, coupled with the particular algorithm and the control sequence of variables and constraints. The resulting amount of redundant computation also depends upon the number of solutions to each constraint in the problem, and their interaction along the selected control sequence.

The combinatorial enumeration of variable bindings can be viewed as a tree in which each node represents a binding to a subset of variables, and the children of a node extend that node's bindings to one more variable. The constraints limit the bushiness of this search tree. As problems get harder, a larger number of variable bindings get explored by the search algorithm, and this search tree gets larger and bushier. If a particular problem constraint topology, coupled with an algorithm and control, exhibits artificial serial dependencies, then the likelihood of redundant constraint checking increases as the specific problem gets harder.

Artificial dependencies and redundant constraint checks are discussed in more detail

in Chapters 2 and 3. We now take a look at the occurrence of this phenomenon in our simple floorplanning example.

1.2.3 Solving the simple floorplanning example

The following control sequence can be used with Backtrack to solve our floorplanning problem:

$$\begin{aligned} C_{f2} \;=\; \langle\; & r_1.x_1\ r_1.x_2\ \text{RightOf-r1} \\ & r_1.y_1\ r_1.y_2\ \text{Above-r1 HasArea-r1} \\ & r_2.x_1\ r_2.x_2\ \text{RightOf-r2} \\ & r_2.y_1\ r_2.y_2\ \text{Above-r2 HasArea-r2 DontOverlap}\ \rangle \end{aligned}$$

This control sequence first generates a valid rectangle for room r_1, then a valid rectangle for room r_2, and then ensures that the two rooms do not overlap.

Notice here that the second instances, for room r_2, of the constraints *RightOf, Above* and *HasArea* are artificially dependent upon the variables for room r_1. For example, the constraint RightOf-r2$(r_2.x_2, r_2.x_1)$ depends only upon the variables $\{r_2.x_2, r_2.x_1\}$. However, the four variables for room r_1, $\{r_1.x_1, r_1.y_1, r_1.x_2, r_1.y_2\}$ are also bound before this constraint is evaluated. Figure 1.2 shows all the artificial dependencies in the control sequence C_{f2}.

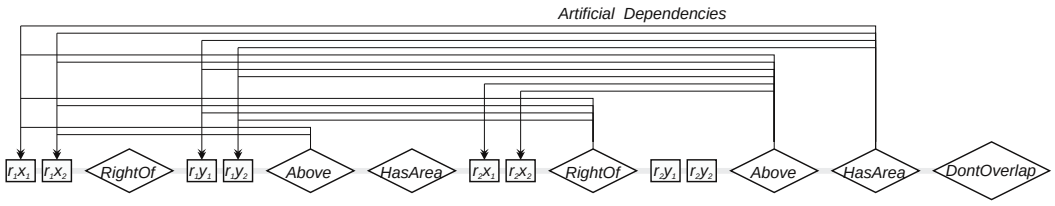


Figure 1.2: The control sequence C_{f2} sets up artificial dependencies between constraints and preceding independent variables.

There are 198 solutions to the problem instance P_{f2} of figure 1.1. Applying Backtrack to the control sequence C_{f2} to solve this problem requires 369,342 constraint

checks, over 94% of which are redundant (figure 1.3). Almost all of the redundant constraint checks occur due to the artificial dependencies of room r_2 constraints on variables of r_1.

Constraint	#Checks	#Redundant	% Redundancy
RightOf-r1	81	0	0.00
Above-r1	1,620	1,584	97.78
HasArea-r1	945	0	0.00
RightOf-r2	10,692	10,611	99.24
Above-r2	213,840	213,804	99.98
HasArea-r2	124,740	123,795	98.24
DontOverlap	17,424	0	0.00
Total	*369,342*	*349,794*	*94.71*

Figure 1.3: Run statistics for Backtrack and control sequence C_{f2} on the problem P_{f2}.

The reason for the redundant computation in this example can be graphically demonstrated by figure 1.4. This figure shows two possible positions for room r_1 explored by Backtrack. The gray area in the figure represents the possible placements for room r_2 that are common to both positions of r_1, thus denoting the area of redundant computations of the above constraints. Each placement of room r_2 in the gray area is repeated after the second position of room r_1, and the corresponding tests of constraints RightOf-r2, Above-r2 and HasArea-r2 are redundant.

1.3 The Thesis: Problem Decomposition

1.3.1 Problem decomposition interrupts serial control

We have seen above that artificial serial dependencies can be a significant source of inefficiency in solving constraint satisfaction problems. The thesis in this research is that problem decomposition can be used to reduce redundant computation by interrupting the serial control order that leads to artificial dependencies.

An extreme case of the applicability of decomposition is in a CSP composed of two completely independent subproblems. This means that the variables of the problem can

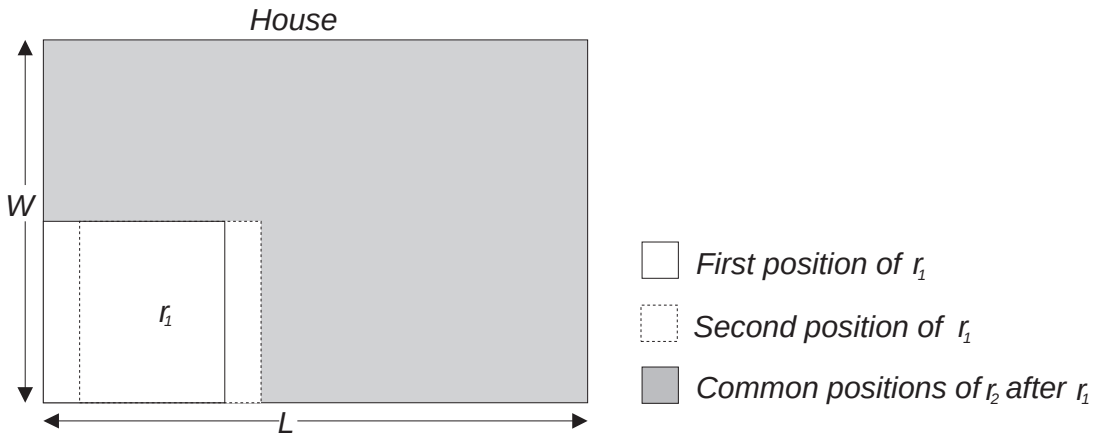


Figure 1.4: Redundant computation due to artificial serial dependencies in the floor-planning example.

be partitioned into two non-trivial disjoint sets such that no constraint in the problem has arguments from both sets. Any Backtrack control order for this problem will set up artificial serial dependencies between the constraints of one partition and the variables of the other; and if the problem is hard, this will cause a significant amount of redundant constraint checking. If the two independent subproblems are solved separately and their solutions cached, generating complete solutions is simply a matter of taking the cartesian-product of the two solution sets, without any further constraint checking. In practice, this 'join' operation can be merged with the solution of the second subproblem.

This idea can be extended to CSPs that are not decomposable into completely independent subproblems. The general procedure proposed in this thesis is as follows:

1. Extract one or more independent subproblems from the given CSP, leaving behind a residual subproblem whose constraints may depend upon the variables of the other subproblems.

2. Solve the independent subproblems using your favourite algorithm, and cache their solutions.

3. Solve the residual subproblem while recalling the cached solutions such that all the problem's variables get instantiated in this step. The solutions of this step

are the solutions to the original CSP.

Small arity constraints near the end of a long control sequence are likely to have a high degree of artificial dependence on preceding variables, with a correspondingly higher potential for redundant computation. Small subproblems, with their smaller control sequences, can greatly reduce the redundant testing of their constraints. When the procedure reaches step 3, several of the problem constraints have already been solved. It is now easier to select a control sequence for the residual that avoids high artificial dependencies in the remaining constraints.

Chapters 3 and 5 of the thesis describe some procedures to use in step 1 to decompose a CSP. Chapter 3 also describes a framework, named *bottom-up solution*, for subproblem solution caching and efficient solution of the residual subproblem. This framework can also be combined with other decomposition methods.

We will defer discussion of the bottom-up solution framework and a detailed description of the decomposition techniques to later chapters. Instead, we now demonstrate the effectiveness of some of these decomposition techniques on our floorplanning problem.

1.3.2 The independent-set decomposition

On examining the floorplanning problem in the previous section, we observed that artificial dependencies led to significant amounts of redundant computation of the constraints *Above-r1*, *RightOf-r2*, *Above-r2* and *HasArea-r2*. When selecting subproblems to extract from this CSP, an obvious choice then is to solve each of these constraints as independent subproblems, thus eliminating their redundant computation. However, they are not all independent, as the constraint *HasArea-r2* shares variables with *Above-r2* and *RightOf-r2*. Since smaller subproblems are cheaper to solve, we select the constraints *Above-r1, RightOf-r2* and *Above-r2*, along with their argument variables, to form the independent subproblems. This yields the bottom-up control sequence $C_{is(f2)}$ shown below.

$$C_{is(f2)} \quad = \quad \langle \; (\; r_1.y_1 \;\; r_1.y_2 \;\; \text{Above-r1} \;)$$

($r_2.x_1$ $r_2.x_2$ RightOf-r2)

($r_2.y_1$ $r_2.y_2$ Above-r2)

($r_1.x_1$ $r_1.x_2$ RightOf-r1

$r_1.y_1'$ $r_1.y_2'$ HasArea-r1

$r_2.x_1'$ $r_2.x_2'$ $r_2.y_1'$ $r_2.y_2'$ HasArea-r2 DontOverlap) $\rangle$

The control sequence $C_{is(f2)}$ contains four components, delimited by parentheses, for the four subproblems obtained after decomposition. The last component is for the residual subproblem in which the cached solutions of the first three subproblems are recalled. This regeneration of values for the cached variables is indicated in the control sequence by a prime in front of the variable names.

This type of decomposition, which we will call an *Independent-set Decomposition*, extracts a set of constraints from the CSP that show high artificial dependencies, and are independent with respect to their arguments. Solving each one individually as a separate subproblem eliminates any redundant testing of these constraints. Figure 1.5 demonstrates how an IS-decomposition removes artificial dependencies on the extracted independent-set of constraints.

The run statistics for the bottom-up solution of $C_{is(f2)}$ using Backtrack are shown in figure 1.6. The total amount of redundant constraint checks is reduced by 225,999, reducing the total cost of solving the problem by over 61%. Notice that this decomposition of the problem does not affect the redundant computation of the constraint *HasArea-r2*.

1.3.3 The degree-of-independence based decomposition

We noted in the previous section that in the floorplanning problem, the room r_2 constraints *RightOf-r2, Above-r2* and *HasArea-r2* are all artificially dependent upon all four r_1 variables in the Backtrack control sequence C. A large number of valid r_1 placements will result in a large amount of redundant evaluation of these constraints. The run statistics in figure 1.3 confirm that most of the redundant computation is in these constraints.

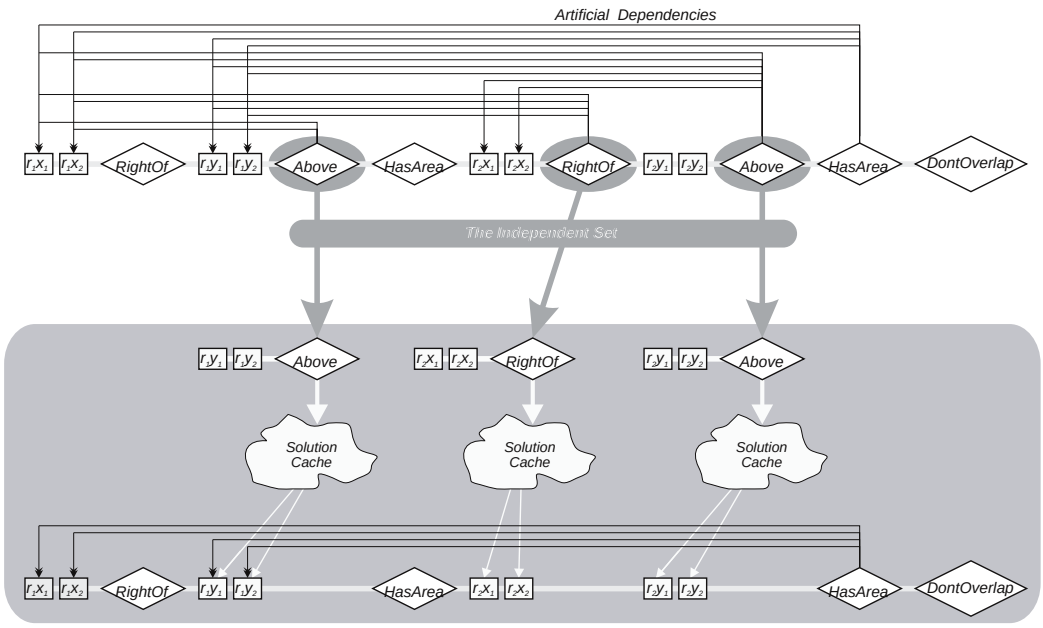


Figure 1.5: An IS-decomposition of C_{f2} eliminates artificial dependencies on a selected set of mutually independent constraints showing high artificial dependencies in C_{f2}.

The independent-set decomposition $C_{is(f2)}$.

Constraint	#Checks	#Redundant	% Redundancy
RightOf-r1	81	0	0.00
Above-r1	36	0	0.00
HasArea-r1	945	0	0.00
RightOf-r2	81	0	0.00
Above-r2	36	0	0.00
HasArea-r2	124,740	123,795	99.24
DontOverlap	17,424	0	0.00
Total	*143,343*	*123,795*	*86.36*

The degree-of-independence based decomposition $C_{doi(f2)}$.

Constraint	#Checks	#Redundant	% Redundancy
RightOf-r1	81	0	0.00
Above-r1	1,620	1,584	97.78
HasArea-r1	945	0	0.00
RightOf-r2	81	0	0.00
Above-r2	1,620	1,584	97.78
HasArea-r2	945	0	0.00
DontOverlap	17,424	0	0.00
Total	*22,716*	*3,168*	*13.95*

Figure 1.6: Run statistics for the Bottom-up Backtrack solution of problem P_{f2} after decomposition.

A simple decomposition technique that breaks this artificial dependency in C_{f2} of r_2 constraints on r_1 variables is to cut the control sequence at a point between the two rooms. The r_1 portion then becomes the independent subproblem, and the remaining portion forms the residual subproblem. This division allows the placement of the recalled variables in the residual after the r_2 constraints, where they do not cause any artificial serial dependencies. The resulting control sequence $C_{doi(f2)}$ is shown below, and the reduction in artificial dependence is demonstrated graphically in figure 1.7.

$$\begin{aligned} C_{\mathrm{DOI}(f2)} \quad = \quad & \langle\, (\ r_1.x_1\ r_1.x_2\ \text{RightOf-r1}\ r_1.y_1\ r_1.y_2\ \text{Above-r1 HasArea-r1}\ ) \\ & (\ r_2.x_1\ r_2.x_2\ \text{RightOf-r2}\ r_2.y_1\ r_2.y_2\ \text{Above-r2 HasArea-r2} \\ & r_1.x_1'\ r_1.x_2'\ r_1.y_1'\ r_1.y_2'\ \text{DontOverlap}\ )\ \rangle \end{aligned}$$

The idea behind this type of decomposition is to divide a given control sequence at a point where it will break the largest number of artificial dependencies (figure 1.7). We will therefore call this technique the *Degree-of-independence based decomposition.*

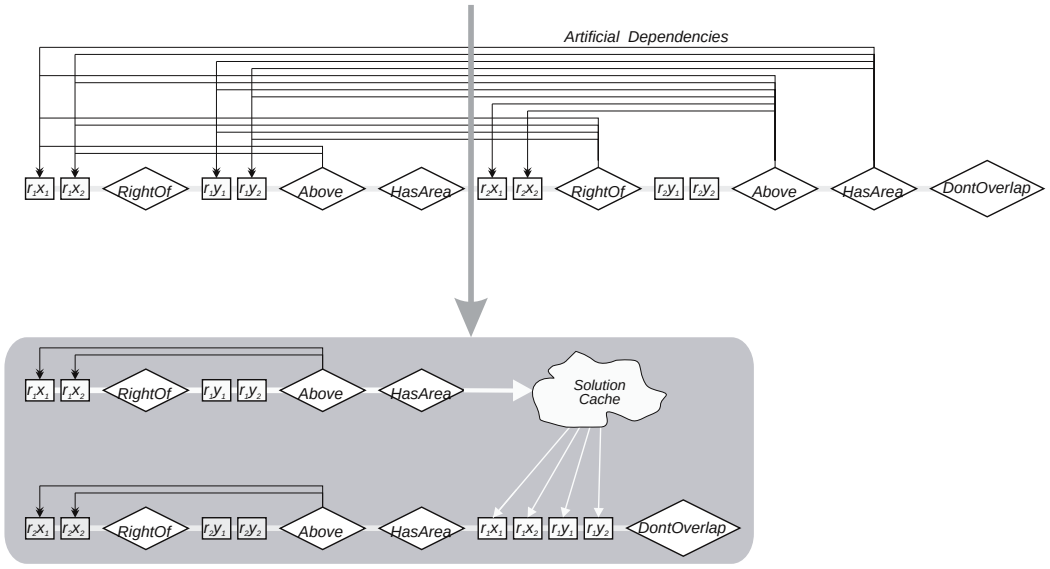


Figure 1.7: DOI-decomposition of C_{f2} cuts the sequence at a point where the largest number of artificial dependencies are broken.

$C_{\mathrm{DOI}(f2)}$ proves to be the most effective Backtrack control sequence for problem P_{f2}

(figure 1.6). The final cost for finding all solutions comes down from over 369,000 to merely 22,716 constraint checks, of which only about 14% are redundant!

1.3.4 Global constraint decomposition

We noted earlier that constraints are used during search to limit the bushiness of the search tree. Clever search algorithms employ various means to take advantage of the constraints in a problem early in the search process in order to reduce the number of nodes explored. A constraint however can only be tested when all of its arguments are instantiated. Consequently, algorithms are unable to take advantage of global constraints.

The second part of this research extends the work on problem decomposition to the decomposition of global (or high-arity) constraints. The procedure is analogous to the decomposition procedure described earlier. A constraint's syntactic formulation is decomposed into subexpressions and a residual combining expression. The subexpressions can then be used to construct smaller arity constraints. Global constraint decomposition produces an abstraction of the original problem. The abstraction can be used with any hierarchical search strategy, such as top-down breadth-first search, and also combined with one of the decomposition techniques described earlier in a bottom-up solution scheme.

We now demonstrate global constraint decomposition on a slightly modified version of the floorplanning problem. To P_{f2} we add another global constraint that requires the two rooms to completely fill the house. Since the rooms are already restricted against overlapping, the new requirement is met by simply constraining the total area of the two rooms to equal the house area. The syntactic expression of the constraint uses the function *Area* which computes the area of a room. We will call the modified floorplanning problem $P_{f2'}$:

$$\begin{aligned} Area(r_i.*) &= (r_i.x_2 - r_i.x_1) * (r_i.y_2 - r_i.y_1) \\ \text{FillHouse}(r_1.*, r_2.*) &\equiv \text{Area}(r_1.*) + \text{Area}(r_2.*) = L \times W \end{aligned}$$

$$\text{HasArea-ri}(r_i.*) \quad \equiv \quad \text{Area}(r_i.*) \geq A_i$$

The *HasArea* constraints can also be expressed in terms of the function *Area* and the problem parameter A_i denoting the minimum area for room r_i.

The addition of the *FillHouse* constraint actually allows a very easy solution of the two-room floorplanning problem which takes advantage of the problem's geometric properties. This thesis is however about general, so called 'weak' problem solving methods.

The control sequence C_{f2} can be extended for $P_{f2'}$ by simply adding the *FillHouse* constraint at the end, just before the *DontOverlap* constraint. This will not change the number of nodes explored, but slightly increase the number of constraint checks required to find all solutions. The modified problem $P_{f2'}$, with the same parameter values as used in P_{f2}, has 6 solutions. It takes Backtrack running on the extended control sequence $C_{f2'}$ 370,034 constraint checks to find all of these solutions. The degree-of-independence based decomposition can also be similarly extended into the control sequence $C_{\text{DOI}(f2')}$ by adding the *FillHouse* constraint at the end. Again, this proves to be a very efficient decomposition (figure 1.8).

Consider the decomposition of the *FillHouse* constraint obtained by abstracting each room to just its area. We introduce two abstract variables z_1 and z_2 which represent the areas of the rooms r_1 and r_2, respectively, and rewrite the *FillHouse* constraint in terms of these variables:

$$\begin{aligned} \text{RC-Z1} \quad &\equiv \quad z_1 = \text{Area}(r_1.*) \\ \text{RC-Z2} \quad &\equiv \quad z_2 = \text{Area}(r_2.*) \\ \text{FillHouse-ac} \quad &\equiv \quad z_1 + z_2 = L \times W \end{aligned}$$

The new *FillHouse-ac* constraint, now completely in terms of the abstract variables, can be viewed as an abstract constraint, and the two constraints *RC-Z1* and *RC-Z2* relating the abstract and original variables can be viewed as refinement constraints. Replacing the *FillHouse* constraint in problem $P_{f2'}$ with the two new variables and the

Backtrack and Bottom-up Backtrack.

	$C_{f2'}$		$C_{\text{DOI}(f2')}$	
Constraint	#Checks	#Redundant	#Checks	#Redundant
RightOf-r1	81	0	81	0
Above-r1	1,620	1,584	1,620	1,584
HasArea-r1	945	0	945	0
RightOf-r2	10,692	10,611	81	0
Above-r2	213,840	213,804	1,620	1,584
HasArea-r2	124,740	123,795	945	0
FillHouse	17,424	0	17,424	0
DontOverlap	692	0	692	0
Total	*370,034*	*349,794*	*23,408*	*3,168*

Bottom-up solution of the hierarchical problem $P_{f2'h}$ using $C_{\text{HSS-4}(f2'h)}$.

Constraint	#Checks	#Redundant	% Redundancy
RightOf-r1	81	0	0.00
Above-r1	1,620	1,584	97.78
HasArea-r1	945	0	0.00
RC-Z1	310	0	0.00
RightOf-r2	81	0	0.00
Above-r2	1,620	1,584	97.78
HasArea-r2	945	0	0.00
RC-Z2	310	0	0.00
FillHouse-ac	225	0	0.00
DontOverlap	692	0	0.00
Total	*6,829*	*3,168*	*46.39*

Comparing the size of the explored search space.

Control Sequence	Number of Nodes	Run-time (mSec)
$C_{f2'}$	263,340	960
$C_{\text{DOI}(f2')}$	33,264	80
$C_{\text{HSS-4}(f2'h)}$	6,645	30

Figure 1.8: Run statistics for the problem $P_{f2'}$, comparing Backtrack, DOI Decomposition and Global Constraint Decomposition.

above three constraints yields the hierarchical problem $P_{f2'h}$.

$$\begin{aligned} P_{f2'h} \quad = \quad \text{Find} \quad & \{\langle r_1.*, r_2.*, z_1, z_2\rangle : \text{RightOf-r1} \wedge \text{Above-r1} \wedge \text{HasArea-r1} \\ & \wedge (z_1 = \text{Area}(r_1.*)) \\ & \wedge \text{RightOf-r2} \wedge \text{Above-r2} \wedge \text{HasArea-r2} \\ & \wedge (z_2 = \text{Area}(r_2.*)) \\ & \wedge \text{FillHouse-ac} \wedge \text{DontOverlap}\} \end{aligned}$$

Several problem-solving strategies can be employed to solve this hierarchical problem. We will demonstrate what is described in Chapter 5 as hierarchical solution strategy number 4 (HSS-4), which implements a bottom-up solution of the hierarchical structure. The control sequence $C_{\text{HSS-4}(f2'h)}$ takes advantage of the mutual independence of the two refinement constraints *RC-Z1* and *RC-Z2* to construct two independent subproblems. Since the refinement constraints are equality relations, their role is inverted into actually computing the value of the abstract variables.

$$\begin{aligned} & C_{\text{HSS-4}(f2'h)} \\ = \quad & \langle\ (\ r_1.x_1\ r_1.x_2\ \text{RightOf-r1}\ r_1.y_1\ r_1.y_2\ \text{Above-r1}\ \text{HasArea-r1}\ [z_1 = \text{Area}(r_1.*)]\ ) \\ & \quad (\ r_2.x_1\ r_2.x_2\ \text{RightOf-r2}\ r_2.y_1\ r_2.y_2\ \text{Above-r2}\ \text{HasArea-r2}\ [z_2 = \text{Area}(r_2.*)]\ ) \\ & \quad (\ z_1'\ z_2'\ \text{FillHouse-ac}\ r_1.*'\ r_2.*'\ \text{DontOverlap}\ )\ \rangle \end{aligned}$$

The residual subproblem (the last component) in $C_{\text{HSS-4}(f2'h)}$ takes advantage of the special subproblem solution caching scheme used in the bottom-up solution framework (see Chapter 3). In this case, the cached solutions to the first two subproblems are indexed on the values of z_1 and z_2 respectively. The early testing of *FillHouse-ac* in the final component of this control sequence filters out rooms whose total area is too small to fill the house. This greatly reduces the rooms on which *DontOverlap* gets tested.

The run statistics for $C_{\text{HSS-4}(f2'h)}$ are shown in figure 1.8. Its performance is even better than the degree-of-independence decomposition because of its clever filtering of room areas. The effectiveness of this strategy is demonstrated in the small size of its explored search space, which is only about 2.5% that of running Backtrack on the original problem!

1.3.5 Repeated structures

A natural extension of the use of the bottom-up solution framework's efficient sub-problem solution caching scheme is in problems where repeated components can be identified. Problem P_{f2} requires the generation of two rooms which, except for the *DontOverlap* constraint, are otherwise completely identical. This makes it possible to generate all possible rooms that satisfy the *Above*, *RightOf* and *HasArea* constraints, and simply reuse as valid r_1 and r_2 placements. This application of the bottom-up solution framework is left out of this research as an area for future study.

1.4 Research Claims and Proposed Evaluation

To summarize, this research proposes some new solution techniques for general constraint satisfaction problems, motivated by the following problems:

- Redundant constraint checks are a significant source of inefficiency in present search algorithms.
- Existing CSP preprocessing and solving techniques are unable to handle or take advantage of general global constraints.

It is the general thesis of this research that decomposition techniques can be applied to CSPs to address these concerns. Specifically, the thesis proposes the following three techniques:

1. A *bottom-up* solution framework for the efficient solution of CSPs in their decomposed form.
2. Two problem decomposition techniques — Independent-Set (IS) decomposition and Degree-of-Independence based (DOI) decomposition — that partition the set of problem constraints to reduce redundant constraint checks.
3. A global constraint decomposition technique that produces a new equivalent problem in which the global constraint is replaced with a conjunction of smaller arity constraints.

The general research claim is that these problem solving techniques are useful and effective in improving problem solving efficiency. The specific research claims are given below.

1. Decomposition by Constraint-Set Partitioning and Bottom-Up Solution:

Claim 1.1. The IS and DOI decomposition techniques, when used with Bottom-Up solution, improve the performance over the basic search algorithm, when looking for all solutions.

Claim 1.2. The IS and DOI decomposition techniques are applicable to a wide and interesting range of problems.

Claim 1.3. The IS and DOI decomposition algorithms are automatable and fast.

The benefit of applying these techniques tends to be greater for problems with a higher potential for redundant computation.

2. Global Constraint Decomposition:

Claim 2.1. Global constraint decomposition is useful in improving problem solving performance beyond basic search.

Claim 2.2. The constraint decomposition procedure is automatable and fast.

Claim 2.3. Global constraint decomposition is applicable to some interesting problems.

Global constraint decomposition is generally applicable only on problems where an intensional representation of its constraints is available. However like the constraint set partitioning techniques, it offers possibilities for improved problem solving performance where none has been available.

The problem decomposition techniques are evaluated on random problems and on some sample application domains. Global constraint decomposition is demonstrated on some sample application domains (floorplanning, corporate decentralization and ship bulkhead design), and shown to significantly reduce search effort.

1.5 Structure of the Thesis

Each chapter in the rest of the thesis has a brief introduction and summary, giving a precis of the main themes and contributions of that chapter.

Chapter 2 sets the foundation and motivation for the rest of this thesis. Constraint Satisfaction Problems are formally introduced there, some well known solution techniques are described, the problem of artificial dependencies and redundant constraint checks, and the difficulty in solving high-arity constraints are discussed in more detail.

Chapter 3 describes the bottom-up solution framework and two problem decomposition algorithms aimed at reducing the number of redundant constraint checks.

Chapter 4 presents an empirical evaluation of the ideas of chapter 3 on random constraint satisfaction problems.

Chapter 5 describes the procedure for constructing and solving hierarchical problems by applying decomposition to global constraints.

Chapter 6 describes three sample application domains, and discusses how their particular characterisitcs are suited to the problem solving techniques proposed in this thesis.

Finally, chapter 7 summarizes the main contributions of this research, and puts it in the context of other research on problem solving and constraint satisfaction problems.

Chapter 2

Constraint Satisfaction Problems

This chapter sets the foundation and motivation for the rest of this thesis. Constraint Satisfaction Problems are formally introduced, some well known solution techniques are described, the problem of artificial dependencies and redundant constraint checks, and the difficulty in solving high-arity constraints are discussed. The notation that will be used in the rest of the thesis is also introduced here.

2.1 Constraint Satisfaction Problems (CSP)

2.1.1 Definition

A *Constraint Satisfaction Problem* (CSP) involves finding one or more values for a fixed finite set of variables (Y), chosen from a fixed finite domain (Domain(Y)), such that they satisfy a fixed and finite set of constraints (R).

$$
\begin{aligned}
P &= \text{Find} \quad \{y \in \text{Domain}(Y) : R(y)\} \\
Y &= \langle X_1, \ldots, X_N \rangle \\
\text{Domain}(Y) &= \text{Domain}(X_1) \times \cdots \times \text{Domain}(X_N) \\
R &= T_1(Y_1) \wedge \cdots \wedge T_K(Y_K) \\
T_i(Y_i) &\subseteq \text{Domain}(Y_i) \\
Y_i &\subseteq Y
\end{aligned}
$$

Figure 2.1: The general model for a Constraint Satisfaction Problem.

Here the CSP P has N variables, whose values are restricted by a conjunction of K constraints (also referred to as tests). Each test constrains the values of arbitrary subsets of the variables Y. A solution to the CSP is a binding to each variable in Y

that satisfies all the constraints.

The domain of the variable set Y (or any subset thereof) is a cartesian product of the domains of the individual variables in the set. The domain of the set of variables of a CSP is also referred to as the problem's search space.

Each constraint T_i is a relation on some subset Y_i of the set of variables Y for the problem. We will refer to Y_i as the set of argument variables (argument set, for short) of the constraint, and the domain of the argument set will also be referred to as the domain, or search space, for the constraint. A constraint with an argument set of size two is called *binary*, and if the argument set is the entire set of variables in the problem, it is called a *global constraint*.

Binding to a variable a value from its domain is also called a labeling. Correspondingly, the problem of constraint satisfaction is also referred to as the Consistent Labeling Problem ([30, 56]).

Several types of Constraint Satisfaction Problems can be identified. The most commonly studied class is that of finding one solution to the CSP ([49, 18] and a summary in [10]). We will also be devoting considerable attention in this thesis to the problem of finding all solutions ([31]). Nadel ([56]) also identifies the problems of counting the number of solutions, and of determining solvability. Others have looked at the issue of partial constraint satisfaction ([20, 23]) in which the attempt is to satisfy as many constraints in the problem as possible.

Another distinguishing criterion classifies CSPs according to the arity of its constraints. Again, the most commonly studied class is that of Binary CSPs, in which each constraint is on exactly two variables. This thesis was motivated by the special issues encountered in dealing with higher arity constraints.

Satisfiability problems, like the famous 3-SAT ([24]), are CSPs with two values in the domain of each variable. The K-Colorability problem of coloring regions in a map (ibid.) is also expressible as a constraint satisfaction problem. CSPs are also NP-Hard ([31]).

2.1.2 An example CSP

Figure 2.2 shows a CSP with five variables and five binary constraints. This CSP has 243 solutions.

$$
\begin{array}{rcl}
P_1 = & \text{Find} & \{y \in \text{Domain}(Y) : R(y)\} \\
\text{where} \quad Y & = & \langle X_1, X_2, X_3, X_4, X_5 \rangle \\
\text{Domain}(X_i) & = & \{\text{a, b, c, d}\} \\
R(Y) & = & T_{13}(X_1, X_3) \wedge T_{14}(X_1, X_4) \wedge T_{24}(X_2, X_4) \\
 & & \wedge\, T_{35}(X_3, X_5) \wedge T_{45}(X_4, X_5)
\end{array}
$$

and the constraints T_{ij} represent the following relations:

$$T_{13} = T_{14} = T_{24} = T_{35} = T_{45} \quad = \quad \{a, b, c\} \times \{a, b, c\}$$

Figure 2.2: Example 1: A simple Constraint Satisfaction Problem.

2.2 Backtrack Search

2.2.1 The Backtrack algorithm

The most well known algorithm for solving a Constraint Satisfaction Problem is *Backtrack* [27, 55]. Given an ordering of variables and constraints for a problem, at each step it either picks a new value binding for the current variable, or evaluates the current constraint on the already bound (or instantiated) variables. If it runs out of values, or a constraint fails, it backtracks to the last variable in the supplied order, binds it to the next value, and proceeds forward again. If it is successful in generating a value for the current variable or evaluating a constraint, it then proceeds to the next item in the supplied ordering.

We will refer to the 'ordering' as a *control sequence* of *nodes*, where each node is either a *generator* for a variable, or a *tester* for a constraint. The backtracking method described above is called Chronological Backtracking. This algorithm, in reference to its architecture, is also sometimes called Generate-and-Test with Chronological Backtracking.

For the problem P, a control sequence C may be denoted by a tuple of $N+K$ nodes

$$C = \langle S_1 S_2 \ldots S_{N+K} \rangle$$

where each node S_i is either a variable generator (briefly just variable, or generator) for X_i, $1 \leq i \leq N$, or a constraint tester (briefly, constraint) for the constraint T_j, $1 \leq j \leq K$. The Backtrack algorithm, given in figure 2.3, takes a valid control sequence as argument. A control sequence is *valid* if all variables referred to by a constraint in the sequence are instantiated by generators occurring before the constraint in the sequence. In this thesis, all the control sequences will be assumed either to be valid, or part of a larger control sequence that is valid.

The algorithm of figure 2.3 is started by the initial call `Backtrack(C, no-bindings, 1)`. The algorithm as shown looks for all solutions. Changing line `(5)` to `EXIT` will transform it to stop after finding the first solution.

Backtrack is the most common and simplest algorithm for solving CSPs. It requires the least amount of memory, compared to the usually faster algorithms Backmark and Forward-Check [31]. Furthermore, it does not require the domains of its variables to be explicitly listed. A variable's generator may actually be a function that calculates the values in its domain.

2.2.2 The search tree

The search space of partial variable bindings explored by Backtrack can be represented in the form of a tree. Each level in the tree corresponds to one of the problem variables, and the N levels are ordered according to the order of variables in the control sequence. A path from the root to a node at level i in the tree represents a particular binding of values to the first i variables in the control sequence. Descendants of the node extend this binding to other variables.

The Backtrack algorithm executes a Depth-First Search of this search tree. It invokes constraints on partial bindings (internal nodes) as soon as all argument variables are bound. When a constraint fails, Backtrack avoids further exploration of that branch.

```
Backtrack(C[1...N + K], x, i)
(1)  if  i < 1  then
(2)       EXIT (no more solutions)
(3)  else  if  i > N + K  then
(4)       Output current bindings x for X as a solution
(5)       GoBack(C, x, i)
(6)  else  case  Type( C[i] )
(7)       Generator for variable X_i:
(8)            x_i =  next value from Dom(X_i)
(9)            if  no more values  then
(10)                Reset Generator for X_i
(11)                GoBack(C, x, i)
(12)           else  Backtrack(C, x, i + 1)
(13)      Tester for constraint T_i:
(14)           Evaluate T_i(x) on current bindings x
(15)           if  T_i fails  then
(16)                GoBack(C, x, i)
(17)           else  Backtrack(C, x, i + 1)
end

GoBack(C[1...N + K], x, i)
(1)  if  i <= 1  then
(2)       EXIT (no more solutions)
(3)  else  Let 1 <= j < i be the index of the last variable Generator before i
(4)       Backtrack(C, x, j)
end
```

Figure 2.3: The Backtrack algorithm for finding all solutions.

At each level, only the branches that satisfy all evaluable constraints are further expanded.

A complete search tree lists all the possible bindings. Clever search algorithms try to reduce the number of nodes actually explored by avoiding as early as possible paths that do not lead to a solution. Different techniques are used to detect dead-end branches, e.g. [31, 62, 10].

2.2.3 The cost of search

Both space (memory) and time requirements of a search algorithm are useful measures of its efficiency. The memory used by most popular search algorithms is bound by a polynomial function on the number of variables and sizes of their domains [31]. Since Constraint Satisfaction Problems are NP-hard, time is usually the more important cost parameter.

When evaluating and comparing different algorithms, the time cost is usually estimated by counting the number of events in a search algorithm. Various researchers have reported on the number of search tree nodes explored, the number of constraint checks, and even the number of backtracks. In the most popular models of CSPs, the domains for each variable are explicitly listed, and the constraints are boolean valued expressions. As such, the evaluation of a constraint is computationally significantly more expensive than generating a value for a variable. Hence counting the number of constraint checks has come to be the most popularly accepted estimate of the time cost of an algorithm. This is also the primary cost parameter used in this thesis.

Usually, each CSP has several valid control sequences, with different associated costs. An optimal control sequence for the CSP of figure 2.2 is

$$C_1 = \langle X_4 X_5 T_{45} X_3 T_{35} X_1 T_{13} T_{14} X_2 T_{24} \rangle$$

This problem was specially contrived to make it easy to calculate the cost of its solution. Each variable has a four element domain, and each constraint has nine solutions. The cost of running Backtrack on C_1 to find all solutions, in terms of the number

of constraint checks, can now be calculated:

$$\begin{aligned}
\text{Cost}(C_1) &= \text{Checks}(C_1) \\
&= \text{Checks}_{C1}(T_{45}) + \text{Checks}_{C1}(T_{35}) + \text{Checks}_{C1}(T_{13}) \\
&\quad + \text{Checks}_{C1}(T_{14}) + \text{Checks}_{C1}(T_{24}) \\
&= 16 + 36 + 108 + 81 + 324 \\
&= 565
\end{aligned}$$

Note that there are 243 solutions to the problem, so the cost has to be greater than 243, since T_{24} alone is tested at least that many times.

2.3 The Problem of Redundant Constraint Checks

A constraint check, occurring during the course of searching for solutions, will be defined *redundant* if that particular constraint has been evaluated earlier on the same argument values (same bindings for its argument variables). This is a source of inefficiency in the search procedure, and the first part of the thesis (chapter 3) focuses on some techniques to reduce redundant constraint checks. Redundancies are a well known source of inefficiency in the solution of CSPs. For example, [62] identify the "redundant discovery of incompatibilities and the redundant elimination of possibilities" as "two major sources of inefficiency in chronological backtracking". The definition of a redundant constraint check used in this thesis is more general, and includes constraint checks that succeed, in addition to those that fail.

Consider the example CSP P_1 of figure 2.2, and the backtracking control sequence C_1

$$C_1 = \langle X_4\, X_5\, T_{45}\, X_3\, T_{35}\, X_1\, T_{13}\, T_{14}\, X_2\, T_{24} \rangle$$

Let the order of generation of values from each variable's domain be $\{a, b, c, d\}$. The first solution found by Backtrack is $\langle X_4, X_5, X_3, X_1, X_2 \rangle = \langle a, a, a, a, a \rangle$. After all the values of X_2 are exhausted, the first backtrack to X_1 occurs. The value b is bound to X_1, which satisfies the constraints T_{13} and T_{14}. Each extension of the new bindings

$\langle a, a, a, b\rangle$ to a binding for X_2 results in a redundant check of the constraint T_{24}, since the value of variable X_4 has not changed since before the backtrack to X_1.

This redundant testing of the constraint T_{24} is a result of the particular serial order implemented in the control sequence C_1. The order in which the variables are bound and constraints tested in C_1 sets up an *artificial serial dependency* of the constraint T_{24} on the variables X_1, X_3 and X_5. So besides changes in the value of X_1, changes in the values of X_3 and X_5 will also cause redundant testing of T_{24}, and their effect, at least in problem P_1, is combinatorially large.

Clearly different control sequences will exhibit different degrees of artificial dependencies. The phenomenon however is not limited to Backtrack. All general (and complete) search algorithms are based on a combinatorial and serial enumeration of variable bindings. It is this serial aspect of the enumeration that sets up artificial dependencies.

We can actually calculate the number of redundant constraint checks in this example. Constraint T_{45} does not have any redundant checks. As there are three values each of X_4 and X_5 that satisfy T_{45}, and there are four values in the domain of X_3, the constraint T_{35} is evaluated redundantly $(3-1)\times(3\times 4) = 24$ times. Similarly, there are nine bindings for $\langle X_4, X_5\rangle$ that satisfy $T_{45} \wedge T_{35}$. For each of these bindings, T_{13} is executed 12 times. All but the first set of 12 are redundant, resulting in $(9-1)\times 12 = 96$ redundant checks of T_{13}. Similarly there are 72 redundant checks of T_{14}, and $(27-1)\times 12 = 312$ redundant checks of T_{24}. The total number of redundant checks in C_1

$$\text{Redundant-Checks}(C_1) = 24 + 96 + 72 + 312 = 504$$

Considering that the total number of constraint checks during execution of C_1 only amounts to 565 (see previous section), this is an astonishingly high figure.

Alternatively, we can calculate the number of unique constraint checks. The constraint T_{45} has 16. Since X_5 gets filtered by T_{45}, and the domain size for X_3 is four, there are $3\times 4 = 12$ unique constraint checks for T_{35}. The same holds for constraints T_{13} and T_{24}. Constraint T_{14} gets both its arguments filtered by other constraints, resulting in only 9 unique checks. The total number of unique (non-redundant) constraint checks

$= 16 + 12 + 12 + 9 + 12 = 61$. The number of redundant constraint checks is, therefore, $565 - 61 = 504$.

Let us define the *context at a node* during the execution of a control sequence as the variable bindings present when that node is invoked during search. We will talk about a variable, or a sequence of nodes, *establishing a context*, and of nodes and in particular constraints *occurring within a context* established by (or of) another portion of the control sequence. Note then that in static search algorithms like Backtrack, the portion of the control sequence succeeding a variable occurs within the context of that variable. Let us also define a variable and a constraint to be *independent* of each other if the variable is not an argument to the constraint. A constraint (say T_{35} in C_1) then gets tested redundantly during search if it occurs within the context of an independent variable (e.g. X_4). We will use this notion of context later to analyze the degree of redundant testing of a constraint.

2.4 Advanced Search Techniques

Researchers have developed clever search algorithms that avoid some of the inefficiencies of Backtrack. We now look at two of these algorithms. Forward Check is a 'look-ahead' search algorithm, and popular for its speed. Dynamic Backtrack is an 'intelligent backtracking' algorithm that has generated some renewed interest.

2.4.1 The Forward-Check algorithm

We saw that although the example CSP's control sequence C_1 described above is an optimal control sequence for Backtrack, it still does a lot of unnecessary work. For example, when constraint T_{24} fails for the first time on the bindings $\langle X_4, X_2 \rangle = \langle a, d \rangle$, the algorithm backtracks and generates a new binding for X_1, and then eventually proceeds to test T_{24} on $\langle X_4, X_2 \rangle = \langle a, d \rangle$ again. A form of look-ahead search called *Forward-Check*, described below, avoids this kind of inefficiency.

The Forward-Check algorithm shown in figure 2.4 is adapted from [31]. The basic principle of Forward-Check is best described by focusing on the ordering of variables

in a control sequence. At each variable in the sequence, a set of future variables is defined, whose values are restricted in conjunction with the current variable through one or more constraints. At each level (of recursion of `LookAheadTreeSearch`), when a new value is bound to the current variable, future variables get their domains trimmed by removing values incompatible with the current bindings. Usually, only a single-level look-ahead is performed (by `CheckForward` in the algorithm of figure 2.4), meaning that each future single variable, together with the current and past variables, completes the set of arguments for some constraint.

```
LookAheadTreeSearch( varsRemaining, bindings, domains )
(1) var = next variable from varsRemaining
(2)  for  each value  in  domains[var]
(3)       bindings[var] = value
(4)       if  var is not last remaining variable  then
(5)            newDomains = CheckForward( var, bindings, domains )
(6)            if  newDomains is not EMPTY-ROW-FLAG  then
(7)                 LookAheadTreeSearch( varsRemaining – var,
                                         bindings, newDomains )
(8)       else  Solution found: bindings
end  LookAheadTreeSearch


CheckForward( var, bindings, domains )
(1)  for  each forward-checkable variable fwdVar
(2)       newDomains[fwdVar] = Empty
(3)       for  each value in domains[fwdVar]
(4)            bindings[fwdVar] = value
(5)            test all forward-checkable constraints on bindings
(6)            if  all constraints pass  then
(7)                 add value to newDomains[fwdVar]
(8)       if  no values added to newDomains[fwdVar]  then
(9)            Return EMPTY-ROW-FLAG
(10)  for  each variable nfwdVar that is not forward-checkable
(11)      newDomains[nfwdVar] = domains[nfwdVar]
(12) Return newDomains
end  CheckForward
```

Figure 2.4: The Forward-Check algorithm for finding all solutions.

The initial call to `LookAheadTreeSearch` is made with *varsRemaining* as an ordered

sequence of all the variables in the problem, *domains* as the provided domains for those variables (after reduction through any unary constraints), and *bindings* empty. The algorithm is static if it always picks the first variable in the ordering *varsRemaining* at each level. Dynamic versions of the algorithm use different criteria to select the next variable to process.

`LookAheadTreeSearch` recurses to as many levels as there are variables in the problem. At each level, a new set of *domains* is generated for the remaining variables. This can require up to $N(N-1)d$ additional locations in memory, for a CSP of N variables, the domain size of each variable bound by d. At the cost of this additional memory requirement, the look-ahead action of Forward-Check often yields a significant speed improvement over Backtrack and most other algorithms (e.g. [31]). However sometimes the memory requirement can be too high, and Backtrack the only feasible algorithm. For this reason, and the simplicity of the algorithm, Backtrack is still widely used, with a lot of techniques devised to improve its performance. Before we look at some of these, we we take a look at the occurrence of redundant constraint checks in Forward-Check.

2.4.2 Redundant constraint checks within Forward-Check

The look-ahead in Forward-Check goes a long way to reduce extra work done by Backtrack, and thus reduces the amount of redundant constraint checks. We can see this by applying Forward-Check to the CSP of Example 1 (figure 2.2). While the same control sequence C_1 is also valid for Forward-Check, it is more instructive to inspect a more explicit representation of the Forward-Check control. We construct an equivalent control sequence using the format $\{$*Variable* — *Forward-Variable*$_1$ *Forward-Constraint*$_1$, ...$\}$. This format makes it easier to analyse the degree of artificial dependence of constraints under Forward-Check:

$$\begin{aligned} C_{1FC} \quad = \quad & \{X_4 - X_5T_{45}, X_1T_{14}, X_2T_{24}\} \\ & \{X_5 - X_3T_{35}\} \\ & \{X_3 - X_1T_{13}\} \\ & \{X_1 - \} \end{aligned}$$

$$\{X_2 - \}$$

Notice that due to look-ahead, constraints get tested at shallower levels in the control sequence, consequently occurring in much smaller contexts. This reduces both the overall cost, as well as the amount of redundant checking. The cost of C_{1FC} is 157, with only 96 redundant constraint checks.

The small proportion of redundant constraint checks in C_{1FC} is a consequence of the constraint topology in the problem. Consider Example 2 of figure 2.5. The subscripts on the constraints indicate their argument variables.

$$\begin{aligned} P_2 = \quad & \text{Find} \quad && \{y \in \text{Domain}(Y) : R(y)\} \\ \text{where} \quad Y \quad & = && \langle X_1, X_2, X_3, X_4, X_5 \rangle \\ \text{Domain}(X_i) \quad & = && \{\text{a, b, c, d}\} \\ R(X) \quad & = && T_{12} \wedge T_{13} \wedge T_{14} \wedge T_{23} \\ & && \wedge T_{25} \wedge T_{34} \wedge T_{45} \end{aligned}$$

where the constraints $T_{ij}(X_i, X_j)$ represent the following relations:

$$\begin{aligned} T_{12} = T_{13} = T_{14} = T_{23} = T_{25} = T_{34} = T_{45} \\ = \quad \{a, b, c\} \times \{a, b, c\} \end{aligned}$$

Figure 2.5: Example 2: Another Constraint Satisfaction Problem.

The control sequence C_{2FC} has a total cost of 427 constraint checks, of which 348 are redundant. The higher cost comes from additional constraints tested at deeper levels in the control.

$$\begin{aligned} C_{2FC} \quad = \quad & \{X_1 - X_2 T_{12}, X_3 T_{13}, X_4 T_{14}\} \\ & \{X_2 - X_3 T_{23}, X_5 T_{25}\} \\ & \{X_3 - X_4 T_{34}\} \\ & \{X_4 - X_5 T_{45}\} \\ & \{X_5 - \} \end{aligned}$$

2.4.3 Dynamic Backtracking

Consider again the example problem P_1 (2.2), and its backtracking control sequence C_1. When T_{24} fails and the domain of X_2 is exhausted, control backtracks to X_1. A new value for X_1 is generated, and eventually T_{24} is tested again. However, the values bound to its argument variables X_2, X_4 have not changed, so until control backtracks all the way back to X_4, T_{24} will continue failing. Each test of T_{24}, after the first run through the domain of X_2, and until X_4 is bound to a new value, is redundant.

Dependency-directed Backtracking ([64]) avoids exactly this type of inefficiency, by identifying X_4 as the culprit variable after X_2 is exhausted, and backtracking directly back to X_4. The ideas of backtracking to culprit variables, and the recording of incompatible bindings as "no-goods" to avoid retesting the same bindings (ibid.), were further developed and presented in an algorithm for solving CSPs, called Intelligent Backtracking ([4]). This was yet further generalized into the *Dynamic Backtracking* algorithm ([26]). See also [62] for a good description of the historical development of the ideas of selective backtracking and conflict recording, and their application to solving CSPs and theorem proving. We will only look at some of the main points in Dynamic Backtracking here, and for a detailed description defer to [26].

Dynamic Backtracking contains two key ideas. Whenever a constraint fails, or a variable runs out of new values to generate, a culprit is chosen as the backtracking target. When control jumps back to the culprit variable, the values bound to other variables are not retracted. After selecting a new binding for the culprit variable, the control jumps back (forward) to the point at which the initial failure had occurred. The effect of this is to make the control sequence a *dynamic* ordering. On failure at a node in the sequence, a culprit node is identified, moved forward in the sequence to a position immediately preceding the node that failed, and then invoking chronological backtracking to the culprit node. Various heuristics can be used to pick the most suitable culprit node. In its most general form, after retracting the current binding to the culprit variable, any unbound variable can be picked for generating a new binding.

The second key idea in Dynamic Backtrack is the maintaining of an *eliminating*

explanation, for each value in a variable's domain that was found to be an incompatible extension to the other current bindings. A value bound to a variable is retracted when it fails to satisfy a constraint evaluated downstream, and the variable was identified as a culprit by the dynamic backtracking mechanism. When this happens an explanation, for the elimination of this value from further consideration, is added to the set of eliminating explanations for that variable. The explanation is (possibly a subset of) the current set of bindings (with which the eliminated value has been identified as being incompatible). A value with an eliminating explanation will not be considered as a possible binding to its variable, as long as future search actions do not retract a binding occurring within that explanation. On backtrack, when the binding for a variable is retracted, in addition to adding explanations for values for that variable, all other explanations (for all other variables) that refer to the retracted binding, are discarded, and those previously eliminated values become available again.

Culprit variables are selected from the eliminating explanations. When a solution is found, to look for more solutions, an eliminating explanation for the last variable bound, containing all the other bindings, is generated. This action mimics the presence of an imaginary global constraint that rejects the solutions found so far.

Dynamic Backtrack techniques can be superimposed onto any basic static search algorithm. This includes both Backtrack and Forward-Check.

2.4.4 Redundant constraint checks within Dynamic Backtrack

We saw earlier in this chapter that the occurrence of constraints in the context of independent variables was the cause of redundant constraint checks. The dynamic nature of backtracking and jumping forward in Dynamic Backtrack reduces the effective context of variables, and thus the potential for redundant constraint checks.

Retaining eliminating explanations in a variable's domain prevents the regeneration of values that had previously lead to a backtrack action. This device further reduces the potential of redundant checks.

There are, however, situations in which redundant constraint checks can still occur in Dynamic Backtrack. Intelligent Backtracking algorithms generally do not attempt

to avoid retesting constraints on bindings that have previously been tested and found to succeed. An eliminating explanation records information about a conflict, and the value it is attached to will not be generated and tested again. However, when one of the current bindings that also occurs in the explanation is retracted, that explanation is discarded. This is perfectly logical, however it now does make that value available as a possible future binding.

The dynamic context of a variable in Dynamic Backtrack is controlled by the culprit identification mechanism. Good mechanisms will prevent the failure of a constraint from causing a backtrack to an independent variable. However, when a higher arity constraint fails, there is the potential for searching its entire domain for solutions. The context of at least one variable in the constraint's argument set is then at least as large as the entire argument set. If there are other smaller arity constraints whose argument sets are subsumed by that of the higher arity constraint, they will be tested within the context of independent variables.

This means that with Dynamic Backtrack, the potential for redundant constraint checks is higher in the presence of mixed arity constraints. In particular, remember that Dynamic Backtrack effectively treats the problem of finding all solutions as if there was a global constraint in the problem. Thus every CSP in this situation has the potential for redundant constraint checks. Other situations where redundant constraint checks can occur in Dynamic Backtrack are discussed in the next chapter.

This ends our discussion of Dynamic Backtrack. In the rest of the thesis, we will only consider Backtrack and Forward-Check where knowledge of the particular basic search algorithm is required. Dynamic Backtrack is a new algorithm, and even its earlier variations have not been studied and experimented with as much as Backtrack and Forward Check. For the purpose of this thesis, we stop at the point of establishing the presence of redundant constraint checks, and identifying high arity constraints (both real and virtual) as the culprit.

2.5 Other Advanced Processing Techniques

Backtrack and Forward-Check are well established algorithms, and researchers have devised several techniques and heuristics for extracting performance from these algorithms. We examine two such techniques, and how they fare in the presence of high arity constraints.

2.5.1 Control sequence variable ordering

Different control sequences for a CSP can produce different performances in both Backtrack and Forward-Check. Even assuming that constraints are tested as soon as all their argument variables are bound, different variable orderings can have different associated costs. Several heuristics have been devised for selecting a good ordering. Among the more well known are the following:

Variable ordering based on connectivity. There are two established heuristics in this class. One counts the number of other variables each is connected to, and orders them in decreasing count ([64]). This heuristic fails, for obvious reasons, in the presence of a global constraint. A similar heuristic minimizes the *width of the ordering* (see next section for definition) ([19]), and fails for the same reason, as with a global constraint all orderings have the same width.

Another variable ordering heuristic, shown to perform well on graph coloring problems, selects an ordering that minimizes the largest constraint bandwidth ([71]). The *bandwidth of a constraint* in a control sequence is the number of variables that have to be bound, after and including the binding of its first argument variable, before it is evaluable. The largest bandwidth for a CSP with a global constraint will always be N (the number of variables in the problem), so this heuristic will not be able to distinguish between different orderings.

Variable Ordering based on Domain Size ([3, 31]). In this heuristic, variables are ordered in increasing domain size. In the absence of any constraints, this ordering will produce a search tree with the smallest number of internal nodes. However, in the presence of constraints of mixed arity, pathologically bad orderings can be selected

where constraints do not become testable until deep levels in the search tree. For example, this will happen when the smaller domain variables do not complete the argument set of any constraint.

Variable ordering based on domain size is also used as a dynamic ordering heuristic with Forward-Check. This works for binary CSPs, where because of the look-ahead, each constraint is evaluable as soon as one of its argument variables is bound. However the presence of higher arity constraints is effectively ignored by the heuristic. Thus the variable ordering strategy fails to include the early solution of higher arity constraints as a goal.

2.5.2 Network consistency and variable clustering

A constraint satisfaction problem can be represented by a Constraint Network ([49]). In the general form, a constraint network is a *constraint (hyper-)graph* ([59]), where for each CSP variable there is a node in the graph, and for each constraint a hyper-arc connecting the nodes in its argument set. Binary CSPs like example problem P_1 of figure 2.2 are represented by a simple constraint graph, where each arc connects at most two nodes (figure 2.6-a). A non-binary CSP can also have a simple graph representation, called a *primal constraint graph*, by replacing each hyper-arc of the constraint graph with a set of simple arcs between each pair of nodes in the constraint's argument set ([11]).

Mackworth ([40]) identifies some network preprocessing techniques, called consistency techniques, which when used before searching for solutions, could help reduce the search by removing some incompatibilities between subsets of the CSP variables. For example, *arc-consistency* removed values from each variable's domain that could not be extended to a solution to some constraint in the problem. Such consistency processing techniques only remove some non-solutions by reducing the domains of the variables. While this can be effective, the reduced problem still has the potential for significant amount of redundant constraint checks for the reasons discussed earlier. Furthermore, the cost of arc consistency rises exponentially on the arity of the constraints.

The network consistency techniques developed for binary CSPs can be applied to

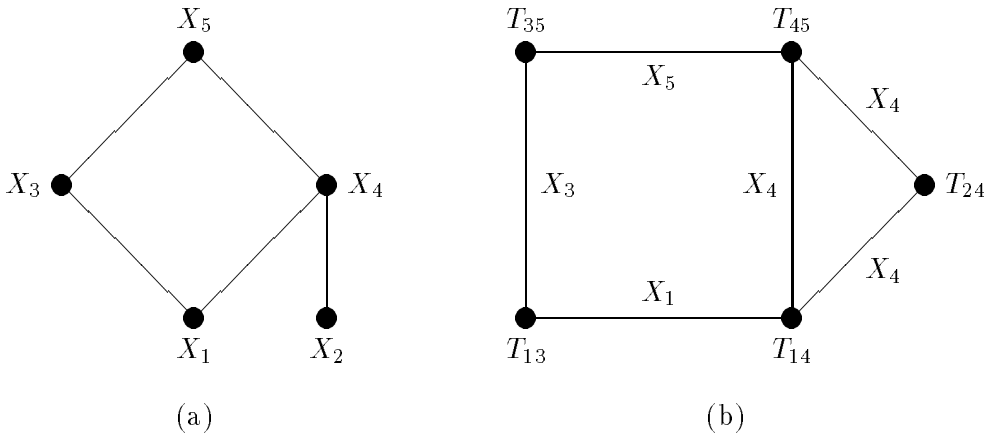


Figure 2.6: Network representations for Example 1: (a) Constraint Graph, (b) Dual Constraint Graph.

higher arity CSPs by transforming the CSP into an equivalent *dual constraint graph* [11]). In the dual graph, each constraint is represented by a node (called a *c-variable*), and there is a simple arc between nodes that share variables in their argument sets. The dual graph for example 1 is shown in figure 2.6-b. The cost of the techniques is then dominated by the number of tuples in the domain of each c-variable, and the cost of generating these tuples. For a c-variable associated with a high arity constraint, this cost can be very high.

Before we look at the next level of development in the employment of network consistency techniques, we need to introduce some definitions and preliminary results.

The *width* of a node in an ordered constraint graph is the number of nodes preceding it in the ordering that it is connected to by some constraints. The width of an ordering is the largest width among its nodes, and the *width of a CSP* is the minimal width over all its orderings.

Freuder generalizes the notion of consistency to *k-ary* consistency, in which consistency is performed on sets of k variables. Arc-consistency is then a 2-ary consistency. Dechter and Pearl further develop this into *directional consistency* [10].

Freuder ([18]) demonstrates that a tree-shaped constraint graph, after performing

arc-consistency, does not require any backtracks to find the first solution. A generalization of this result requires the execution of a level of directional strong consistency along an ordering that is greater than the width of the ordering, for then using that ordering to find the first solution without backtracks ([11]). The drawback, however, is that the cost of directional consistency is exponential in the level of consistency required, which depends upon the largest arity of a constraint in the CSP. The width of a CSP with a global constraint is N (the number of variables).

Another set of techniques seek to take advantage of the backtrack-free properties of tree-shaped constraint graphs by restructuring the given problem. A tree-shaped constraint graph can be induced for non-tree shaped CSPs by partitioning the variables into clusters, and looking at the arcs between these clusters. For non-binary CSPs, the primal constraint graph is used for this step. Consistency techniques performed on the clustered graph, by treating each cluster as a single variable, will render the network backtrack-free. Two examples of such clustering techniques are described in [11] and [28]. For these techniques to work, there should be at least one variable cluster that completely subsumes the largest arity constraint in the problem. And since the cost of consistency is exponential in the size of the cluster, high arity and global constraints present a serious problem for these techniques.

2.6 Analysing the Cost of Backtrack

We observed earlier in the chapter how a control sequence completely determines the operation of Backtrack. We now develop a cost model for solving a CSP using Backtrack. Determining, and even estimating the cost of Backtrack *a-priori* is a very difficult task, and in most cases perhaps impossible ([38, 58]). The purpose of the cost model described here is mostly analytical, to be used in reasoning about different control sequences in later chapters.

Before delving into the model itself, we first formally define some notation. We shall use the following symbols to refer to various components of a CSP:

$$Y, Y_i \quad = \quad \text{a set or tuple of variables}$$

$$\begin{aligned} X, X_i &= \text{a variable} \\ R, R_i &= \text{Set of constraints} \\ T, T_i &= \text{a constraint} \end{aligned}$$

We will use the model of describing a CSP of figure 2.1. The problem P has N variables (X_i) and K constraints (T_i). Sometimes, Y will be used to denote the set of variables of the CSP P, and R its set of constraints. Such specific usage will be obvious from its context.

$$\begin{aligned} Y &= \{X_1, \ldots, X_N\} \\ Y_i &\subseteq Y \\ R &= \{T_1, \ldots, T_K\} \\ R_i &\subseteq R \end{aligned}$$

To help us define the problem's search space and its solutions, we define the following:

$$\begin{aligned} \text{Domain}(Y) &= \text{Set of legal values for variable(s) } Y \\ \text{Dom}(Y) &= |\text{Domain}(Y)| \\ \text{Solnset}(P) &= \{y \in \text{Domain}(Y) : R(y)\} \\ \text{Numsol}(P) &= |\text{Solnset}(P)| \end{aligned}$$

The Domain of a tuple or set of variables is the cartesian product of the domains of the component variables. Thus $\text{Dom}(X_1X_2) = \text{Dom}(X_1) \times \text{Dom}(X_2)$.

We shall also use the definition of a valid control sequence introduced in section 2.2.1, and assume that C is a valid control sequence for the problem P. We can now talk about the solution-set of a sequence as being equivalent to the solution-set to the corresponding CSP.

$$\begin{aligned} \text{Solnset}(C) &= \text{Solnset}(P) \\ \text{Numsol}(C) &= \text{Numsol}(P) \end{aligned}$$

We will often need to look at the behaviour of various portions of a control sequence. We shall denote a control sequence as a concatenation of its components, e.g. $C = \langle C_a C_b \rangle$,

or $C_1 = C_a XTC_b$ (sometimes leaving out the angled brackets for brevity). Note that all complete control sequences will be assumed to be valid, i.e. assumed to bind all argument variables before they are required to test a constraint. Also note that the prefix of any valid control sequence is also a valid sequence.

Finally, we define the following additional functions:

$$\begin{aligned} \text{Args}(C) &= \text{The set or tuple of variables occuring as arguments} \\ &\quad \text{to constraints in the control sequence } C \\ \text{Vars}(C) &= \text{The set or tuple of variables generated} \\ &\quad \text{or bound in control sequence } C \end{aligned}$$

The cost model developed below is an extension of the cost model described in [63], modified to account for CSPs where the cost of a constraint depends upon the size of its search space, and not on the number of its solutions.

2.6.1 Number of solutions of a CSP

Let us define $\text{Numsol}(C_b, Y_a = y)$ to represent the number of solutions to the control sequence C_b when the variable set Y_a is bound to the tuple y. In this way, we can talk about the number of solutions to any subsequence C_b of a control sequence, supplying a binding in the second argument for all the variables required by C_b but not bound by C_b. When none of the variables Y_a occur as generators in the sequence C_b, we refer to the binding $Y_a = y$ as a context for the sequence C_b, established by the variables Y_a (presumably bound in a preceding portion of a complete control sequence). The number of solutions of a control sequence $C = \langle C_a\ C_b \rangle$, $\text{Vars}(C_a) = Y_a$, can be determined as

$$\text{Numsol}(C_a\ C_b) = \sum_{y \in \text{Solnset}(C_a)} \text{Numsol}(C_b, Y_a = y)$$

Let us now define the conditional solution size $\text{AvgNumsol}(C_b, C_a)$ to be the number of solutions to control sequence C_b, in the presence of a set of bindings that solves the sequence C_a, averaged over all solutions to C_a.

$$\text{AvgNumsol}(C_b, C_a) = \underset{y \in \text{Solnset}(C_a)}{\text{Avg}} \text{Numsol}(C_b, Y_a = y)$$

$$\text{where} \qquad Y_a = \text{Vars}(C_a)$$

The number of solutions to a control sequence can now be rewritten as

$$\text{Numsol}(C_a\ C_b) = \text{Numsol}(C_a) \times \text{AvgNumsol}(C_b, C_a)$$

The following identities are defined:

$$\begin{aligned} \text{AvgNumsol}(\langle\rangle, C) &= 1 \\ \text{AvgNumsol}(C, \langle\rangle) &= \text{Numsol}(C) \end{aligned}$$

The smallest possible *valid* control sequence is a single variable. The number of solutions to a variable generator is simply the size of its domain. An empty control sequence will have no solutions.

$$\begin{aligned} \text{Numsol}(x) &= \text{Dom}(x) \\ \text{Numsol}(\langle\rangle) &= 0 \end{aligned}$$

For a constraint T, a variable binding either solves the constraint, or it does not. We define the *solution density* of a constraint to be the average number of solutions per element in its search space.

$$\begin{aligned} \text{Solution Density of } T &= \text{AvgNumsol}(T, Y) \\ &= \text{Numsol}(\langle YT\rangle)/\text{Dom}(Y) \\ \text{where} \qquad & Y = \text{Args}(T) \end{aligned}$$

The solution density for any constraint will be at most one, this high figure achieved when every member of its search space solves the constraint.

We can also talk about the average number of solutions of a constraint in some context. The *conditional solution density* of a constraint T with respect to a context is defined as $\text{AvgNumsol}(T, C)$ when T occurs within the context established by C.

$$\text{AvgNumsol}(T, C) = \text{Numsol}(\langle CT\rangle)/\text{Numsol}(C)$$

The solution density is then a special case of the conditional solution density, where the context is established by generators for its argument variables.

A variable generator is a static quantity and does not change its behaviour with context. The average number of solutions of a variable generator is again simply the size of its domain.

$$\mathrm{AvgNumsol}(X, C) = \mathrm{Dom}(X)$$

Two control sequences will be said to be *independent* when constraints in either sequence do not refer to variables generated in the other. If C_2 is independent of C_1,

$$\mathrm{Args}(C_2) \cap \mathrm{Vars}(C_1) = \emptyset$$

then

$$\mathrm{AvgNumsol}(C_2, CC_1) = \mathrm{AvgNumsol}(C_2, C)$$

Furthermore, if C_1 is also independent of C_2, (and both are valid), i.e.

$$\begin{aligned} \mathrm{Args}(C_1) &= \mathrm{Vars}(C_1) \\ \mathrm{Args}(C_2) &= \mathrm{Vars}(C_2) \\ \mathrm{Args}(C_1) \cap \mathrm{Vars}(C_2) &= \emptyset \\ \mathrm{Args}(C_2) \cap \mathrm{Vars}(C_1) &= \emptyset \end{aligned}$$

then

$$\begin{aligned} \mathrm{AvgNumsol}(C_2, C_1) &= \mathrm{Numsol}(C_2) \\ \mathrm{Numsol}(C_1 C_2) &= \mathrm{Numsol}(C_2 C_1) \\ &= \mathrm{Numsol}(C_1) \times \mathrm{Numsol}(C_2) \end{aligned}$$

Notice that to actually calculate the number of solutions to a problem using this formulation, we will first have to pick a valid control sequence, and then require knowledge of several conditional solution-set sizes or conditional constraint solution densities.

The number of solutions to a set of constraints is independent of the order in which the constraints are evaluated. Similarly, the number of solutions for a problem is independent of which valid control sequence is used to solve the problem. The same is also true of valid permutations of a portion of a control sequence, as every valid subsequence can be said to define a subproblem of the original CSP.

2.6.2 Cost of Backtrack

The cost of running Backtrack to find all solutions, on a control sequence C of M constraints and variables,

$$C = \langle S_1 S_2 \ldots S_M \rangle$$

can be expressed in terms of the cost incurred in each node as follows:

$$\text{Cost}(C) = \text{Cost}(S_1) + \text{Numsol}(S_1)\text{Cost}(S_2) + \cdots + \text{Numsol}(S_1 \cdots S_{M-1})\text{Cost}(S_M)$$

Let the sequence C be the concatenation of two sequences C_1 and C_2, and let the variables occurring in C_1 be Y_1. If we use $\text{Cost}(C, Y_1 = a)$ to be the cost of solving C under variable bindings $Y_1 = a$ then

$$\begin{aligned} C &= \langle C_1 C_2 \rangle \\ \text{Cost}(C) &= \text{Cost}(C_1) + \sum_{\forall a \in \text{Solnset}(C_1)} \text{Cost}(C_2, X_1 = a) \\ &= \text{Cost}(C_1) + \text{Numsol}(C_1) \times \text{AvgCost}(C_2, C_1) \end{aligned}$$

where $\text{AvgCost}(C_2, C_1)$ is defined to be the cost of solving sequence C_2 given a solution to C_1, averaged over all solutions to the sequence C_1. We will often drop the mention of the variables X_1 in the second argument to Cost where they are understood from the context, thus writing $\text{Cost}(C_2, a)$ instead of $\text{Cost}(C_2, X_1 = a)$.

$$\text{AvgCost}(C_2, C_1) = \underset{a \in \text{Solnset}(C_1)}{\text{Avg}} \text{Cost}(C_2, a)$$

Clearly if C_1 and C_2 are independent, the average cost of C_2 is the same as its normal cost.

We will assume that the cost of generating the domain of a variable X, $\text{Cost}(X)$, is a product of the number of values in the domain, and the cost of generating a single value from that domain. We will further assume that this unit generating cost is the same for all variables and domains, and denote it with the symbol k_G. Thus for a variable X,

$$\text{Cost}(X) = \text{Dom}(X) \times k_G$$

In general, each sequence is at least responsible for the cost of the solutions it produces. If we assume the unit cost of all testers to be at least as high as the unit generating cost, then for a control sequence C,

$$\text{Cost}(C) \geq \text{Numsol}(C) \times k_G$$

In some cases we may assume that the average cost of testing any constraint once is the same for all constraints. We will represent this unit testing cost with k_T. We may also sometimes find it convenient to assume that there is no difference in cost of evaluating a constraint under different variable bindings. For a constraint T,

$$\text{AvgCost}(T, C) = \text{Cost}(T) = k_T$$

The main difference between this cost model and that developed by Smith [63] is that here the cost of solving a constraint includes the cost of generating its search space, whereas in Smith's model the cost of solving a conjunct is proportional to the number of its solutions.

As noted above, researchers usually restrict the measure of the cost of solving a CSP to the number of constraint checks required. This is equivalent to choosing the unit costs $k_G = 0$ and $k_T = 1$.

2.6.3 Projections

We will sometimes need to look at the bindings to only a subset of the variables in a solution set. We define $C|Y$ as the the solution set for C *projected onto* the variables Y. It is the set of all bindings to the variables Y that form a part of a complete solution to C. When used as argument in $\text{Numsol}(C|X)$, the function denotes the size of the solution set for sequence C projected onto Y. Let the variables of the control sequence C consist of two disjoint sets Y_1 and Y_2. The number of solutions to C can then also be calculated as follows:

$$\begin{aligned} \text{Vars}(C) &= Y_1 \cup Y_2 \\ \text{Numsol}(C) &= \text{Numsol}(C|Y_1) \times \text{AvgNumsol}(C|Y_2, C|Y_1) \end{aligned}$$

A similar statement can also be made for the number of solutions for constraints.

$$\begin{aligned} \text{Args}(R) &= Y_1 \cup Y_2 \\ \text{Numsol}(R) &= \text{Numsol}(R|Y_1) \times \text{AvgNumsol}(R|Y_2, R|Y_1) \end{aligned}$$

The expression $\text{AvgNumsol}(R|Y_2, R|Y_1)$ denotes, in a set of solutions to the constraints R, the average number of bindings to Y_2 that extend a Y_1 binding to complete solutions. Since every value $R|Y_1$ is a partial solution, it is a part of at least one complete solution:

$$\text{AvgNumsol}(R|Y_2, R|Y_1) \geq 1$$

The same is also true if R was a valid control sequence and Y_1, Y_2 its variables.

The average number of solutions to a constraint set R_1, given a solution to a second set of constraints R_2, only depends upon the variables of the second set also shared by the first. Let $Y_1 = \text{Args}(R_1)$ and $Y_2 = \text{Args}(R_2)$, such that $Y_1 \cap Y_2 = Y_{2a}$ and $Y_2 = Y_{2a} \cup Y_{2b}$. Then

$$\text{AvgNumsol}(R_1, R_2) = \text{AvgNumsol}(R_1, R_2|Y_{2a})$$

This ends our general discussion of the cost of running Backtrack for a CSP. We will use this notation, developed for evaluating the number of solutions and the cost of solving a Constraint Satisfaction Problem (using Backtrack), to support the intuitions motivating the solution procedures proposed in the following chapters, and for deriving heuristics for the application of these procedures to problems.

2.7 Summary

In this chapter we introduced Constraint Satisfaction Problems, and the well known and widely used Backtrack and Forward-Check algorithms for searching for solutions. We identified redundant constraint checks as a major source of inefficiency in both algorithms, and linked them to the testing of constraints in the context of independent variables.

We then looked at some advanced search techniques, and the situations where these too were susceptible to the same source of inefficiency. These situations usually involved

the presence of high arity constraints. We saw that some of the advanced processing techniques, particularly the variable ordering and clustering schemes, fail completely to be useful for CSPs with a global constraint.

Finally we introduced an analytic cost model for running Backtrack to find all solutions to a problem.

In chapters 3 and 4, we look at some decomposition techniques specifically aimed at reducing redundant constraint checks. Then in chapters 5 and 6, we look at the use of abstractions for solving some global constraint CSPs.

Chapter 3

Decomposition and Bottom-Up Solution of CSPs

In chapter 2 we observed that redundant constraint checks were a major source of inefficiency in the search for solutions to constraint satisfaction problems. This chapter describes two problem decomposition techniques aimed specifically at reducing redundant constraint checks, and a 'bottom-up' solution framework for solving a problem through its decomposition. The decompositions are produced by partitioning the problem's set of constraints.

3.1 When do Redundant Constraint Checks Occur

We observed in previous chapters how control sequences for search algorithms set up artificial dependencies of constraints on independent variables, leading to redundant testing of those constraints. We now take a closer look at this phenomenon, using primarily static Backtrack as the search paradigm.

3.1.1 Within the context of independent variables

Consider again the example CSP P_1 of figure 2.2, and its associated control sequence

$$C_1 = \langle X_4\, X_5\, T_{45}\, X_3\, T_{35}\, X_1\, T_{13}\, T_{14}\, X_2\, T_{24} \rangle$$

In the last chapter we saw that when Backtrack is applied to this control sequence to look for all solutions, 504 of the 565 constraint checks are redundant. Remembering that each constraint in the problem restricts its argument variables to three ($\{a, b, c\}$) of the four possible values, the constraint T_{35} in C_1 will get tested $3 \times 3 \times 4 = 36$ times, once for each value of the tuple $\langle X_4 X_5 X_3 \rangle$ satisfying T_{45}. In these, there are

only $3 \times 4 = 12$ unique values for the actual arguments of T_{35}. Thus $36 - 12 = 24$ of the tests must be redundant.

As already discussed, the redundant tests of T_{35} occur because of the presence of the variable X_4 preceding the constraint in the control sequence; a variable that this constraint is actually independent of. We defined the constraint T_{35} as occurring within the *context of independent variables* in the control sequence, causing it to be *artificially serially dependent* upon that variable.

We can determine the number of redundant constraint checks for T_{35} as follows:

$$
\begin{aligned}
C_{1h} &= \langle X_4 X_5 T_{45} X_3 \rangle \\
&\text{Redundant-Checks}_{C1}(T_{35}) \\
&= \text{Numsol}(C_{1h}|X_4X_5X_3) - \text{Numsol}(C_{1h}|X_5X_3) \\
&= (\text{AvgNumsol}(C_{1h}|X_4, C_{1h}|X_5X_3) - 1) \times \text{Numsol}(C_{1h}|X_5X_3) \\
&= (3-1) \times 12 \\
&= 24
\end{aligned}
$$

And in general for a constraint $T(Y_T)$, the number of redundant checks in the sequence $\langle CT \rangle$, where the variables generated (and possibly tested) in C are $Y_I \cup Y_T$, is

$$
\begin{aligned}
\text{Checks}_{\langle CT \rangle}(T) &= \text{Numsol}(C) \\
\text{Redundant-Checks}_{\langle CT \rangle}(T) &= (\text{AvgNumsol}(C|Y_I, C|Y_T) - 1) \times \text{Numsol}(C|Y_T)
\end{aligned}
$$

From this exposition it is clear that the amount of redundant checking of a constraint T depends upon the number of variables independent of T in whose context it occurs, and the restriction of their values by any other constraints preceding T.

Let us define the *size of the context* at a node in a control sequence as the number of values of the preceding bound variables that are active when search control reaches that node. Note that when searching with static control sequences, the size of a context is also the number of solutions to the portion of the control sequence establishing that context. The number of redundant constraint checks of a constraint in a control sequence is then directly proportional to the size of the independent portion of its

context (i.e. the context at that constraint). This is formally defined by the last equation above.

While we have only considered Backtrack here, the last chapter also discussed the effect of independent variable contexts in other search algorithms like Forward-Check and Dynamic Backtrack.

3.1.2 When backtracking reassigns independent variables

When constraints occur in the context of independent variables in a control sequence, it is a good preliminary indication of the possibility of redundant constraint checks. The actual degree of redundancy, as we saw above, is controlled by how tightly the values of those independent variables are constrained before the embedded constraint is tested in the control sequence. When instead of all, only one or a few solutions are needed, additional factors come to affect the degree of redundancy. This is because to find the first solution, not all values of those independent variables might be needed. We now look at another topological property of the control sequence that can serve as an indication of the potential for redundant constraint checks, even when the search algorithm does not look for all solutions.

To extend the analysis to 'one-solution' situations, we have to consider the backtrack behaviour of the search algorithm. In the trivial case where the right value is generated for each variable the first time each generator is invoked, there are no backtracks, and no redundant constraint checks. On the other extreme is the case where each constraint is satisfied on only one of the bindings it gets tested on. There is some backtracking, but no redundant constraint checks (and the problem has only one solution). It is in the middle that hard problems lie, and where redundant constraint checks occur.

Backtracking is caused by the failure of a set of variable bindings to satisfy a constraint. The portion of the control sequence that control must then backtrack to resolve the failure depends upon the argument variables to the failing constraint. Let us define the *scope of a constraint* as that smallest portion of the control sequence immediately preceding the constraint, that contains generators for all the constraint's argument variables. For simplicity, we will restrict ourselves to the Backtrack algorithm, though this

argument can easily be extended to other search algorithms.

When a constraint fails, control may have to backtrack all the way to the beginning of the constraint's scope to search for a solution to the constraint. If this scope contains another constraint, and preceding that some variables that are independent of this embedded constraint, then the potential for redundant checking of this embedded constraint is introduced.

Figure 3.1 shows the scope of the constraint T_{14} in the control sequence C_1. The inner box shows the scope of the constraint T_{13}, which in this case is completely contained within the larger scope of T_{14}. The important point here is that preceding T_{13} in the scope of T_{14} are the variables X_4, X_5, which are independent of the contained constraint T_{13}. In general, for redundant constraint checks caused by the scope of a constraint, the embedded constraint's scope need not be completely contained. All that is needed is the presence of independent variables preceding the embedded constraint, and contained within the scope of the later constraint.

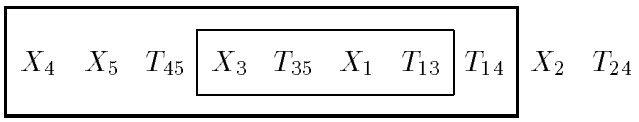


Figure 3.1: T_{13} occurs in the scope of T_{14} in sequence C_1.

When two constraints interact, the resulting combined constraint can often be much tighter than the individual constraints. For instance in C_1, there is the potential for interaction between T_{35} and T_{13}. This interaction acts like a virtual ternary constraint T_{135}, placed after the later constraint in the sequence:

$$\langle X_4\, X_5\, T_{45}\, X_3\, T_{35}\, X_1\, T_{13}\, \mathbf{T_{135}}\, T_{14}\, X_2\, T_{24} \rangle$$

In this way, an interaction between constraints can result in a scope larger than that of either constraint. Constraint T_{13} again occurs embedded within this scope, and in the

context of the embedded independent variable X_5.

When all solutions to a CSP are required the search behaves as if there is a virtual constraint, positioned at the end of the control sequence, that rejects every N-tuple it is tested on, with the side-effect of printing the tuples that solve the CSP. Since this virtual constraint is a global constraint, every constraint of the original CSP occurs in its scope. In general, looking for the first solution requires far less search than looking for all solutions, and the degree of redundancy is also smaller.

The situations described above in which redundant constraint tests occur can also be found in other search algorithms besides Backtrack. The look-ahead nature of Forward-Check reduces the scope of constraints by eliminating the forward variable from the scope (see Chapter 2). Binary constraints get their scope trivially reduced to a single variable. This is why some dynamic ordering heuristics work particularly well for Binary CSPs in Forward-Check [47]. However, higher arity constraints, and constraint interactions, can still have extended scopes, and contain other constraints with the potential for redundant constraint checks.

One of the aims of Dynamic Backtracking is to reduce the scope of a failing constraint to only its argument variables ([26, 64]). This removes the possibility of situations like that in the control sequence C_1, where the constraint T_{35} is embedded within the scope of the constraint T_{14}, even though their argument sets do not intersect. However, when two constraints do have some argument variables in common, one must still occur within the scope of the other. Thus when the outer constraint causes a lot of search, the potential for redundant evaluations of the inner constraint is introduced. The larger the argument set of a constraint, the greater the number of constraints embedded within its scope, and the greater the likelihood of redundant constraint checks. In addition, the same argument for interacting constraints also holds true for Dynamic Backtrack.

3.2 Two Techniques for Reducing Redundant Constraint Checks

We have argued above that redundant constraint checks in a control sequence depend upon the size of the independent portion of the context at each constraint. The number

of variables in a constraint's context that are independent of the constraint provides a crude but readily evaluable measure of this size. We now introduce two techniques that decompose the control sequence (and the corresponding CSP) in order to reduce these independent variable contexts, and thus improve the efficiency of the basic search algorithm. They are based on two different algorithms for partitioning a problem's constraint set.

The decomposition methods are briefly introduced in this section by applying them to the CSP of Example 1 (figure 2.2). Later sections then describe each technique in more detail.

3.2.1 Bottom-Up solution of decomposed CSPs

The job of a problem decomposition method does not end after decomposing a problem into subproblems. It must also suggest a procedure for combining the solutions to the subproblems and handling any interactions, so that the solutions to the original problem can eventually be produced. We start this section by introducing the *bottom-up solution framework* aimed at efficiently handling this role. All the decomposition techniques presented in this thesis are targeted at this framework.

The decomposition techniques described later in this chapter decompose a CSP into a set of one or more mutually independent subproblems, and a residual subproblem. The problem's constraints, along with their argument variables, get divided between the first set of subproblems, with any remaining constraints and variables allocated to the residual. The residual subproblem contains all the interactions between the solutions of the former set that need to be handled to produce solutions to the original problem.

In the *Bottom-Up Solution Framework*, each subproblem is solved individually, and its solutions are cached. The residual subproblem is solved last, and is responsible for producing the actual solutions to the CSP. It must therefore generate all the variables in the CSP. Those variables that it shares with other subproblems are generated from the corresponding cached solution sets, such that only the cached solution bindings are generated.

This bottom-up solution strategy is conveniently represented in a *nested* or *multi-level* control sequence, consisting of control sequences for each individual subproblem, delimited by parentheses. For example, the multi-level sequence C_1' below represents a decomposition of the problem P_1 of Example 1 into two subproblems.

$$\begin{aligned} C_1 &= \langle X_5 X_4 T_{45} X_3 T_{35} X_1 T_{13} T_{14} X_2 T_{24} \rangle \\ C_1' &= \langle (X_5 X_4 T_{45} X_3 T_{35})\ (X_1 X_3' T_{13} X_4' T_{14} X_2 T_{24} X_5') \rangle \end{aligned}$$

The variables X_5, X_4 and X_3 are first instantiated and their solutions to the constraints T_{45} and T_{35} saved in the first subproblem. In the second subproblem, which in this case is also the residual, these solutions are recalled to solve constraints T_{13}, T_{14} and T_{24}. The variables in the second subproblem used to generate the cached solutions of the first are denoted by a prime superscript (e.g. X_3').

We will refer to variables of the type X_3' used to recall values saved at a previous subproblem as *cached variables*. In the general framework, any subproblem besides the residual may also refer to the saved solutions of another. These solution cache references can involve several levels, inducing a partial order on the subproblems. In the control sequence

$$\begin{aligned} C = \langle &(X_1\ X_2\ T_1)\ (X_3\ X_4\ T_2) && \ldots Level1 \\ &(X_1'\ X_2'\ X_5\ T_3) && \ldots Level2 \\ &(X_2''\ X_3''\ T_4\ X_5''\ X_6\ T_5\ X_1''\ X_4'') \rangle && \ldots Level3 \end{aligned}$$

there are three levels, the first containing two subproblems. The levels in a multi-level control sequence are denoted by the number of primes on the cached variables of the control sequences in each level. The first level will have none, and the residual subproblem will always be in the last level. The subproblems within a level can be solved in any order, or even in parallel. A decomposition strategy must specify the level each subproblem belongs in, which is then used to define a partial order for the execution of the subproblems. The decomposition strategies presented in this thesis always generate only two levels.

We shall sometimes refer to the first level as the *top level* in the level hierarchy, and the last level as the *bottom* or *base level*. The levels below the top level will then be

called *lower levels*.

The generators for cached variables are different from those of ordinary variables. Cached variables must be bound to only those values from their domains that are actual solutions to the cached subproblem. Thus X_3' in C_1' is restricted to those values that form part of a solution to the first subproblem. When control reaches X_4', only those values for it are valid that actually extend the *current binding* of X_3' towards a solution to the first subproblem of the sequence. It is the role of the bottom-up solution framework to handle these type of generators efficiently. Section 3.3 describes in more detail how this is accomplished.

So far, no mention has been made of the search algorithm used to solve each subproblem. The reason for this is very simple – any search algorithm may be used. The caching scheme described later in this chapter, used for saving and regenerating the solutions to a subproblem, requires that the algorithm used to solve a subproblem containing cached variables maintain the relative order of those cached variables in the control sequence. So long as this restriction is satisfied, any algorithm may be used to solve the subproblems, and indeed different algorithms may be used for different subproblems. In this thesis however, we shall only consider the use of the same search algorithm for every search problem in a multi-level control sequence. We shall refer to this search algorithm as the *basic* search algorithm. When a basic search algorithm like Backtrack is coupled with bottom-up solution, we shall refer to the overall solution procedure as *bottom-up Backtrack*.

Through the rest of this chapter, unless specifically mentioned otherwise, we will restrict ourselves to improving upon the performance of Backtrack, when employed to search for all solutions to a CSP, by using bottom-up Backtrack. Thus Backtrack will be used for solving each subproblem.

3.2.2 An example of the independent set decomposition

In chapter 2, we looked at the cost and number of redundant checks incurred when employing Backtrack to find all solutions to the problem P_1 of Example 1, using the

control sequence C_1.

$$\begin{aligned} C_1 &= \langle X_4 X_5 T_{45} X_3 T_{35} X_1 T_{13} T_{14} X_2 T_{24} \rangle \\ \text{Checks}(C_1) &= 16 + 36 + 108 + 81 + 324 = 565 \\ \text{Redundant-Checks}(C_1) &= 24 + 96 + 72 + 312 = 504 \end{aligned}$$

We also noticed that the the 96 redundant checks of T_{13} occur because it is placed within the context of the independent variables X_4 and X_5.

The first decomposition technique, applied to the sequence C_1, reduces the number of redundant checks of the constraint T_{13} by simply removing it from its independent context, and solving it separately. This produces the two-level control sequence

$$C_{1a} = \langle (X_3 X_1 T_{13})\ (X_4 X_5 T_{45} X_3' T_{35} X_1' T_{14} X_2 T_{24}) \rangle$$

Remembering that the size of the domain of each variable in P_1 is 4, the number of constraint checks required to solve T_{35} is reduced from 108 to 16. The second subproblem is the residual, and regenerates the cached solutions of the first by including X_1' and X_3' in its control sequence.

For the bottom-up solution of C_{1a} to be cheaper than C_1, we need a solution caching and regeneration scheme that does not (significantly) add to the cost of running Backtrack on each subproblem. Such a scheme is implemented in the bottom-up solution framework introduced above, and described in more detail in section 3.3. Here, we will simply assume the employment of such a scheme when determining the cost (in number of constraint checks) of solving a decomposed control sequence.

The sequence C_{1a} can be further improved by also removing the constraint T_{24} from the residual and solving it separately. This then yields the sequence $C_{is(1)}$:

$$\begin{aligned} C_{is(1)} &= \langle (X_3 X_1 T_{13})\ (X_2 X_4 T_{24})\ (X_4' X_5 T_{45} X_3' T_{35} X_1' T_{14} X_2') \rangle \\ \text{Checks}(C_{is(1)}) &= (4 \times 4) + (4 \times 4) + (3 \times 4 + 9 \times 3 + 27 \times 3) \\ &= (16) + (16) + (12 + 27 + 81) \\ &= 152 \quad \ldots\text{(reduced from 565)} \end{aligned}$$

To calculate the cost of the decomposed problem, we simply add the costs of solving each subproblem. Computing the cost of the residual is complicated by the fact that it has

cached variables. These variables are restricted to generating the solutions to the cached subproblems. The CSP P_1 has been specially contrived to make this computation easy. Each constraint in the problem restricts each of its argument variables to the same 3 of the 4 domain values. Thus there are nine 2-tuple solutions to each constraint.

The dramatic reduction in the cost of solving the problem, from 565 to only 152 constraint checks after decomposition, comes mostly from the reduction in the number of redundant constraint checks of the constraints extracted into separate subproblems in the first level. The original costs of these constraints in the sequence C_1 were 108 and 324 for the constraints T_{13} and T_{24} respectively. The total cost of solving these constraints in $C_{is(1)}$ is $16 + 16 = 32$, yielding a savings of 400 constraint checks. A further savings of 13 constraint checks resulted from the rearrangement of the contexts in the residual subproblem.

This decomposition strategy is called *Independent Set Decomposition* because the subproblems in the first level must be independent of each other (i.e. not share any variables). This is needed for efficient use of the solution caching scheme, and is discussed in more detail later in this chapter.

3.2.3 An example of the degree of independence based decomposition

The *Degree of Independence based Decomposition* (*DOI-Decomposition*) also reduces the context at embedded constraints, in this case by solving the prefix establishing the context in a control sequence as a separate subproblem. Consider once again the constraint T_{13}, embedded within the context of the independent variables X_4 and X_5 in the control sequence C_1. These variables occur in a prefix of C_1 preceding the constraint. DOI-decomposition extracts this prefix into a separate subproblem control sequence, using the remainder of the sequence to build the residual. This yields the two-level sequence $C_{\mathrm{DOI(1)}}$:

$$C_{\mathrm{DOI(1)}} = \langle (X_4 X_5 T_{45} X_3 T_{35})\ (X_3' X_1 T_{13} X_4' T_{14} X_2 T_{24} X_5') \rangle$$

In the residual, the variables cached in the first subproblem are recalled after the originally embedded constraints, thus reducing their independent context. In $C_{\mathrm{DOI(1)}}$,

the constraint T_{13} no longer has an independent variable context. The contexts of the constraints T_{14} and T_{24} are reduced by one independent variable (X_5). We can calculate the cost of the new sequence as before:

$$\begin{aligned} \text{Checks}(C_{\text{DOI}(1)}) &= (4 \times 4 + 9 \times 4) + (3 \times 4 + 9 \times 3 + 27 \times 4) \\ &= (16 + 36) + (12 + 27 + 108) \\ &= 199 \quad \ldots(\text{reduced from } 565) \end{aligned}$$

The savings here come from the reduction of the redundant constraint checks for the constraints T_{13} (total checks reduced from 108 to 12, a factor of 9 representing the number of solutions to the variables X_4, X_5 at T_{13} in C_1), T_{14} (total reduced from 81 to 27) and T_{24} (total reduced from 324 to 108).

The next section contains a description of the solution caching scheme, and in the following two sections we take a closer look at the two decomposition techniques introduced here.

3.3 Solution Caching and Bottom-Up Solution

The purpose of decomposing a CSP into multiple levels is to first solve simple subproblems and then efficiently combine these solutions to solve problems at the next level. The solution caching scheme is responsible for efficient storage and recall of the subproblems' solutions. As was noted earlier, it is important that this storage scheme involve a minimal amount of overhead for the decomposition and bottom-up solution to be useful.

3.3.1 The solution cache tree data structure

We first look at the efficient recall requirements of a solution cache. In the model of Constraint Satisfaction Problems used here, the domain for each variable is presented as a list of values. The list also indicates the order in which the values are generated. A generator then simply steps down the list and produces the next value each time it is invoked, at the same time keeping track of its position in the list relative to the end.

For our cost model where constraint checks dominate the computational search time, the generation of solutions from a cache structure must closely match the operation of ordinary generators. At the same time, the generator for a cached variable must be aware of the set of solutions it is generating from. Furthermore, to avoid generating non-solutions, it must also be aware of the bindings of other variables from the same solution cache.

The Backtrack search algorithm executes a depth-first traversal of a search tree in which the levels are ordered according to the order of the variables in the given control sequence. The solutions to a CSP (subproblem) can also be represented as a search tree (figure 3.2-(a)), though with fewer branches and leaves. When these cached variables are generated in the same order in a later subproblem, the order of generation will also be a depth-first traversal of the solution tree. This suggests the use of a list structure mirroring this depth-first order to store the solutions of a subproblem.

An example of such a solution cache is shown in figure 3.2. The solution tree in figure 3.2-(a) stores the solution set

$$\langle X_3, X_4, X_5 \rangle \in \{\langle a, a, a \rangle, \langle b, a, a \rangle, \langle b, a, b \rangle, \langle b, c, c \rangle\}$$

The corresponding solution cache structure is shown in figure 3.2-(b). We will refer to this data structure as the *solution cache tree*, or just solution cache or solution tree for short. Generating bindings for variables from a solution cache is simply a matter of following the pointers, provided the variables are generated in the same order as they are cached. The cost of such an operation is about as expensive as generating values from an explicitly enumerated list stored as an array.

In the figure, the solution cache facilitates recalling the variables in the order X_3, X_4, X_5. In the second level of the control sequence, after generating value a for variable X_3', the only legal value for X_4' is a.

3.3.2 Building the solution cache tree

During bottom-up solution, the solution tree for each subproblem is built from scratch, growing as each solution is produced during execution of the subproblem. A new

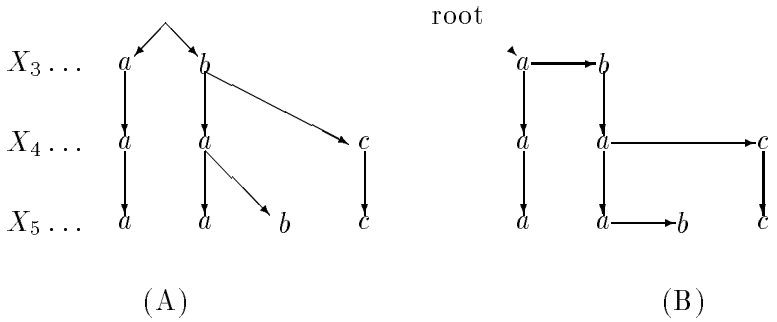


Figure 3.2: Caching the solutions of a subproblem: (a) The solutions in a search tree, (b) Depth-first traversal Solution Cache Tree.

solution is added to the solution tree by first locating the largest path from the first level down that matches a portion of the new solution. In the first level below this path a new node is inserted into the sibling nodes chain for the new value for that variable in the new solution. Any remaining variable values are then added by dropping new nodes below this node. To facilitate the locating of a matching path, the sibling nodes are kept in sorted order (in a linked list).

Since the solution cache stores the variable bindings for the solution set in a depth-first order for later generation, the order in which the variables will be regenerated (in a later subproblem) must be known before building the cache.

Theorem 3.1 *For a CSP (sub)problem of n variables and s solutions, the cost of building the solution tree is bound by $O(s^2 + sn)$, and the size of the solution tree is $O(sn)$ nodes.*

Consider a solution tree containing r solutions. The first step in adding a new solution is to locate a partially matching path starting from the first level. This is done by consulting each node in the first level until a matching node is found. This node will point to its children. These nodes are consulted next; and so on until a level is reached where no matching node can be found. At this point, new nodes are added to

the solution tree.

Each node consulted and eliminated eliminates one solution whose path passes through that node. As there are only r solutions in the tree, at most r nodes can be consulted.

Each solution consists of n values, one for each of the n variables in the subproblem. Thus at most n new nodes can be added to the solution tree with each solution. As there are s solutions, there will be at most sn nodes in the final tree.

The cost of adding a new solution is then at most $r + n$. The total cost of building the solution tree is at most $\sum_{r=0}^{s-1} r + n = s(s-1)/2 + sn$ which is $O(s^2 + sn)$. □

Of the operations involved in building the solution tree, the closest in cost to a constraint test is the consultation of a tree node to check if it is on the path for a new solution. This consultation requires simply comparing the binding of a variable in the new solution with the binding in the tree node. Of the total cost ($s(s-1)/2 + sn$) of building the solution tree, there are at most $s(s-1)/2$ node-value comparisons. These value comparisons are, in general, far simpler than the computation of a constraint expression, which latter could be quite large and complex. Therefore when measuring the cost of a multi-level control sequence employing the solution cache tree to store solutions, we ignore the cost of building the tree as well as the cost of generating from the solution cache tree. However for CSPs where the cost of variable generation needs to be taken into consideration, the cost of building the solution tree might also be significant, depending upon the number of solutions being stored.

An alternative storage scheme can eliminate the cost of node consultations during solution tree construction, albeit at an increase in storage requirements. In this scheme, each node is really a composite structure containing a d-sized vector of pointers, where d is the size of that variable's domain. Figure 3.3 shows an example. The top level in the tree, corresponding to the first variable (X_3 in the figure), contains one node. If the j-th value for that variable is present in the solution cache, the j-th element in the node's vector points to a node at the next level, otherwise it contains a "null" entry. Nodes at the bottom level in the tree indicate the presence of a value (along the path from the root) by a dummy pointer. Using an array instead of a linked list to store

sibling values in the solution cache tree allows testing for the presence of the j-th value in a set of siblings at constant cost irrespective of j and the size of the tree. This decreases the cost of adding a solution to a tree already containing r solutions down to n steps. The total cost of building a solution tree of s solutions then becomes $O(sn)$. However the upper bound on the size of such a tree will be dsn instead of just sn for the linked-list structure. As the number of solutions approaches the maximum possible number (cartesian product of he domains), the vector-node data structure begins to be more efficient, as it does not need to actually store the domain values in each node, unlike the linked-node implementation.

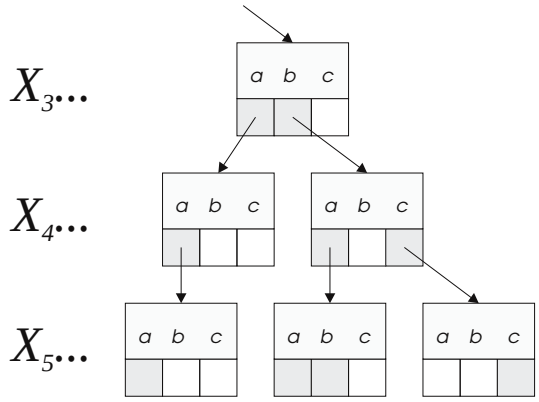


Figure 3.3: An alternative data structure for the solution tree of figure 3.2. Each node contains a domain-sized vector. Shaded elements indicate presence of that value and point to the next level.

If space is not a problem, e.g. for small subproblems or for a subproblem with a relatively small number of solutions, the vector-nodes provide a much more efficient structure for building a solution tree. On the other hand, the cost of generating solutions (figure 3.5) from the solution tree will be almost the same for both data structures (see section 3.6). Unless specifically mentioned, in the rest of the thesis we will assume that the solution tree uses the linked-list structure of figure 3.2-(b).

The size of a solution tree may be compared with the storage overhead required for Forward-Check. At each of $N-1$ levels in an N variable CSP, Forward Check requires

additional storage for the domains of the remaining variables. If the domain size for each variable is bound by d, Forward Check will need up to $(\sum_{r=1}^{N-1} rd) = N(N-1)d/2$ memory locations.

Finally, we note that this solution caching scheme does not provide for two solution caches sharing variables. If two subproblems do indeed share variables, they cannot be solved in parallel. Furthermore, the solution of the second subproblem must result in a solution tree containing both subproblems' variables. This is affected by regenerating all the variables from a solution tree referred to by a subproblem, even if the constraints in the second subproblem do not require all the variables. The partial control sequence $\langle \ldots (X_1 X_2 T_1)\ (X_2' X_3 T_2 X_1') \ldots \rangle$ demonstrates how this is accomplished.

As we have seen above, some cost is incurred each time a solution cache is built. While this cost, like the cost of variable value generation, can be ignored compared to the number of constraint checks, when the number of these cheap operations becomes very large, the total starts becoming significant. In the decomposition strategies proposed below, such overlapping subproblems are avoided, resulting in only two level decompositions, with the residual subproblem forming the second level.

3.4 Decomposition into Maximal Independent Constraint Set

3.4.1 Reducing the independent context at embedded constraints

We now take a closer look at the first of the two decomposition techniques. As has been noted above, the degree of redundant testing of a constraint depends directly upon the size of the independent portion of the context at that constraint in the control sequence. The *Independent Set Decomposition* (*IS-decomposition*) reduces the independent context at a constraint by simply removing the constraint from the control sequence and solving it as a separate subproblem. Only the extracted constraint's argument variables are included in the new subproblem, thus avoiding any artificial dependencies. Figure 1.5 gives a graphical depiction of the reduction in artificial dependencies achieved by IS-decomposition.

While solving a constraint as a separate subproblem avoids any redundant testing,

it also prevents other constraints from pruning the domains of its argument variables, possibly requiring the constraint to be tested on values it would not have been in the original control sequence. So only those constraints must be extracted where the savings in redundancy more than compensates for any increase in their effective search space.

Clearly, the more constraints that can be extracted and solved separately, the greater the total savings. A further savings may also be realized in the residual problem, composed of the unextracted constraints, when the cached variables can be rearranged so as to reduce the independent contexts at those remaining constraints. The IS-decomposition of C_1 for example problem P_1 (see section 3.2) does indeed achieve savings both in the extracted constraints as well as in the reduced executions of the residual constraints, though the former in this case dominates.

Employing the solution cache tree to store the solutions of each subproblem will avoid their regeneration from adding to the cost of solving the decomposed problem. However, as observed in section 3.3, it is generally helpful to keep the number of levels in a decomposition small. For simplicity, we restrict the IS-decomposition to produce only two subproblem levels. Since the residual problem must be in the second subproblem level, all the extracted constraints get placed in the first subproblem level. They therefore need to be variable-independent of each other. This is why the decomposition is called an *Independent Set* decomposition, borrowing the graph-theoretic notion of an independent set of vertices. In the dual graph representation of a CSP (chapter 2, and [11]), constraints are represented by vertices, and extracting an independent set of constraints corresponds directly to selecting an independent set of these vertices.

3.4.2 Selecting a suitable independent constraint set

We noted above that the more constraints that can be extracted, the greater the potential for improving search performance by the decomposition. The problem of finding the largest independent set of a graph is NP-hard ([24]). However the controlling size parameter in the cost of constructing an independent set is the size of the dual constraint graph. This is usually so much smaller compared to the cost of solving the CSP, as to be relatively insignificant.

Since bottom-up solution of a decomposed problem is an enhancement over basic search algorithms, it is natural to expect decomposition to improve upon the performance of a selected control sequence for the problem. Accordingly, we shall assume that one such sequence has been selected as a proposed optimal sequence for the basic search algorithm (e.g. Backtrack), and use that sequence as the basis for further improvement.

Selecting the best set of constraints to extract is not just a matter of selecting the largest independent set. Since most of the cost improvement by this decomposition is expected to come from the reduction in redundant testing of the extracted constraints, we can look for the set of constraints that maximizes the sum of this quantity over the extracted set. Since the extracted constraints must be independent of each other, any choice of a constraint to extract can limit the choice of other constraints that can be extracted along with it. For a CSP with constraint set R and chosen control sequence C, the ideal extracted set will optimize the following quantity over all possible independent sets

$$\underset{Ind\ Set\ R_I \subset R}{\max} \left(\sum_{T \in R_I} \text{Checks}_C(T) - \text{Dom}(\text{Args}(T)) \right)$$

The parameters required to evaluate this formula are not generally available before actually solving the given control sequence. If they can be cheaply estimated to a sufficient accuracy, then this formula will yield the best independent set. However, for most situations this is not likely to be feasible, so we look for heuristic procedures.

In general, lower arity constraints are the most likely candidates for extraction. They will be independent of a larger number of preceding variables in the control sequence, and therefore likely to incur a larger amount of redundant testing. In addition, with a smaller number of argument variables, the search space of each extracted constraint will be smaller, yielding cheaper subproblems.

As we have also seen in our analysis of the example CSP P_1, the first constraint in the control sequence will never have any redundant checks. Constraints deeper in the control sequence will tend to occur in larger independent contexts, especially in hard problems, due to the larger number of variables preceding them. The procedure *ISB* for extracting constraints (figure 3.4) incorporates these heuristics.

```
ISB( C[1...N+K], MaxArity ( = N Default) )
(1) iset = empty
(2) e = N + K + 1
(3) repeat
(4)       Let j = index of last constraint in C before e
(5)       if j not found then Return(iset)
(6)       Let i = index of last variable in C before e
(7)       Find smallest r s.t.:
(8)             i < r ≤ j,
(9)             Arity(C[r]) ≤ MaxArity
(10)            Arity(C[r]) ≤ Arity(C[t]), ∀t, i < t ≤ j
(11)            C[r] is independent of each constraint in iset
(12)      if r found then iset = iset ∪ C[r]
(13)      e = i
(14) end
end
```

Figure 3.4: The ISB procedure for extracting an independent set.

To use ISB, we first use whatever heuristics and techniques we might have to select a good backtracking control sequence (C). This is then used as a base, and ISB tries to improve upon its performance. ISB steps backwards along the control sequence, selecting constraints to add to its growing independent set, making an attempt to pick smaller arity constraints. An optional cap may be supplied to the procedure on the largest arity constraint to include in the independent set. ISB also tries to pick constraints occurring earlier in a consecutive subsequence of constraints, as these would tend to get invoked more often in C.

A complete scan of the control sequence will produce an independent set that is maximal (without necessarily being the largest), and will tend to include constraints occurring at deeper levels in the given control sequence, and specifically excluding the very first constraint of the sequence. To complete the decomposition process, each extracted independent constraint is used to form a subproblem, and the remaining constraints are used to form the residual.

Each extracted constraint's subproblem consists of generators for its argument variables, followed by a tester for the constraint. If the argument variable set of any of the

residual constraints is a subset of that of an extracted constraint, that residual constraint is added to the extracted constraint's subproblem. Finally, the same ordering heuristics used to build the original control sequence C may now be employed to order the variables and constraints in each of the subproblems.

For the example CSP P_1 (figure 2.2), the control sequence C_1 is optimal for Backtrack. Giving this as input to ISB produces the control sequence $C_{is(1)}$. The performance of this IS-decomposition was analyzed in section 3.2.

$$\begin{aligned} C_1 &= \langle X_4 X_5 T_{45} X_3 T_{35} X_1 T_{13} T_{14} X_2 T_{24} \rangle \\ C_{is(1)} &= \langle (X_3 X_1 T_{13})\ (X_2 X_4 T_{24})\ (X'_4 X_5 T_{45} X'_3 T_{35} X'_1 T_{14} X'_2) \rangle \end{aligned}$$

3.4.3 Analysis of the decomposition procedure

The IS-decomposition of a CSP is not guaranteed to improve performance over a simple Backtrack application to the problem. However, we can identify some situations in which the performance is improved. When running Backtrack, each constraint in the consecutive series of testers in the tail of the sequence must be invoked at least as many times as number of solutions produced. This gives an obvious clue to the problem classes where ISB can produce a useful decomposition. Before we look at such a problem class, note that in a binary independent set, all the constraints in the independent set are binary (have two arguments).

Theorem 3.2 *A decomposition induced by a binary independent set of size m, as extracted by the ISB procedure outlined above, will be more efficient (in the number of constraint checks) than the original control sequence when the CSP has more than md^2 solutions, where d is the size of the largest variable domain*

$$Checks_{BT}(C_{is}) < Checks_{BT}(C) \quad when \quad Numsol(C) > md^2$$

when every variable in the given CSP has at least one binary constraint restricting its values.

We will prove this theorem in two parts: first showing that the cost of the extracted independent set is lower in the decomposition, and then establishing that the number

of constraint checks invoked in the residual in the decomposition is no greater than the number of times those constraints would be tested in the original sequence.

Let $C_{is} = \langle C_i\, C_r \rangle$, where C_i consists of the independent-set subproblems, and C_r is the sequence for the residual. Then under Backtrack,

$$\text{Checks}(C_{is}) = \text{Checks}_{C_{is}}(C_i) + \text{Checks}_{C_{is}}(C_r)$$

Part 1. Since each constraint in the independent set is binary, and the size of each variable domain is at most d, the cost of solving the m subproblems in C_i, corresponding to the independent set of constraints, is md^2.

$$\text{Checks}_{C_{is}}(C_i) = md^2$$

Since ISB was used to extract the independent set, one of these constraints (say T_t) must occur in the tail of the control sequence C from which the independent set was extracted. As the problem has more than md^2 solutions, this constraint is invoked at least md^2 times in the original sequence. Using ISB(C) to represent the set of independent constraints extracted by the ISB procedure,

$$\exists T_t \in \text{ISB}(C) \text{s. t.} \text{Checks}_C T_t \geq md^2$$

The cost of solving the independent set in the decomposed sequence is therefore cheaper than its cost in the original sequence.

$$\text{Checks}_{C'}(\text{ISB}(C)) \leq \text{Checks}_C(\text{ISB}(C))$$

Part 2. We now prove that the residual base level does not increase the cost of the remaining constraints.

Let T be any one of the extracted independent constraints, with arguments X_a and X_b. We will prove that extracting T from C does not increase the cost of testing the remaining constraints. Without any loss of generality, we can rewrite C as follows

$$C = \langle C_1 X_a C_2 X_b C_3 T C_4 \rangle$$

where the C_i are possibly empty subsequences of C. Extracting T from C yields the nested sequence C' for the decomposition

$$C' = \langle (X_a X_b T)\ (C_1 X'_a C_2 X'_b C_3 C_4) \rangle$$

We need to show that under Backtrack,

$$\text{Checks}_{C'}(C_1C_2C_3C_4) \leq \text{Checks}_{C}(C_1C_2C_3C_4)$$

Clearly the decomposition does not affect the prefix C_1,

$$\text{Checks}_{C'}(C_1) = \text{Checks}_{C}(C_1)$$

Furthermore, since $\text{Domain}(X'_a) \subseteq \text{Domain}(X_a)$ and $\text{Domain}(X'_b) \subseteq \text{Domain}(X_b)$, the number of times backtrack control will reach C_2 (and C_3) in C' will be at most the same as in C. Therefore,

$$\begin{aligned} \text{Checks}_{C'}(C_2) &\leq \text{Checks}_{C}(C_2) \quad \text{and so} \\ \text{Checks}_{C'}(C_3) &\leq \text{Checks}_{C}(C_3) \end{aligned}$$

Our subproblem solution caching and variable recall strategy ensures that decomposition does not alter the number of solutions of a problem. Therefore

$$\text{Numsol}(C_1X_aC_2X_bC_3T) = \text{Numsol}(C_1X'_aC_2X'_bC_3)$$

from which we can deduce that

$$\text{Checks}_{C'}(C_4) = \text{Checks}_{C}(C_4)$$

We can then conclude that

$$\text{Checks}_{C'}(\text{Constraints}(C) - \text{ISB}(C)) \leq \text{Checks}_{C}(\text{Constraints}(C) - \text{ISB}(C))$$

and therefore

$$\text{Checks}_{BT}(C_{is}) \leq \text{Checks}_{BT}(C)$$

□.

The total size of the solution cache trees for m extracted binary constraints, each tree holding upto s solutions, is $2ms < 2md^2$. For a problem with N variables, the size of the largest independent set is at most $N/2$ (each constraint connects at least two variables). This presents an additional bound on the size of the solution cache of a binary independent set: $2ms \leq m(d^2 + d) \leq N(d^2 + d)/2$ and $2ms \leq Ns$. As a

comparison, Forward Check requires $N(N-1)d/2$ memory locations. For CSPs with large number of variables and a smaller domain size, the Backtrack-based bottom-up solution of an IS-decomposition of the problem could present a viable alternative to plain Backtrack, with less memory overhead than Forward-Check.

3.4.4 Discussion

In general, the suitability of an IS-decomposition depends upon the following factors:

- The number of constraints (the size of) in the independent set.
- The degree of redundant execution of the extracted constraints in the original control sequence.
- The increase in the number of tests of the extracted constraints when tested independently, out of context of the original sequence.

The last two items are also affected by the number of solutions after which search stops.

One general conclusion we can draw from the theorem above is that the larger the number of solutions to the CSP (when looking for all solutions), and the larger the size of the independent set extracted by the ISB procedure, the greater the potential for improving performance over Backtrack.

It is also obvious from the same theorem that the ISB procedure may not be suitable for CSPs with a small number of solutions, or where only one or a few solutions are needed. We can define a variant of the ISB algorithm, which we will call the *ISF procedure* for extracting an independent set, which builds its independent set by scanning *forward* along the given control sequence. We shall see in the experimental results of the next chapter that this procedure produces better independent set decompositions for problems with or requiring a small number of solutions.

Finally, although we have only concentrated on Backtrack, as discussed earlier, indeed any basic search algorithm may be used in the bottom-up solution of a decomposed problem. The solution caches are built according to the order of the cached variables in the residual. The search algorithm used to solve the residual must maintain this

relative order of the cached variables for each solution cache, while no particular order is required between variables from different caches. In fact, since the subproblems in the first level are independent, they can even be solved in parallel.

3.5 Decomposition by Maximizing Degree of Independence

3.5.1 The decomposition strategy

We have seen that the context at a constraint in a control sequence, is defined by the variables and constraints in the prefix portion of the control sequence ending at the last node before the constraint in question. We then observed that the amount of redundant testing of a constraint depends upon the size of the independent portion of its context. There are two obvious ways to reduce a constraint's context by decomposing a control sequence: one is to remove the constraint from the offending context into a separate subproblem. This was used in the IS-decomposition. The alternative is to deal directly with the offending context. We shall now explore this decomposition technique.

In *DOI-decomposition*, a prefix of the given control sequence is selected and extracted into a separate subproblem. The remaining constraints are used to form the residual subproblem. Cached variables are inserted into an ordering of the residual's variables and constraints in positions where they minimize the independent contexts of the residual constraints. This reduces the degree of artificial serial dependence of constraints in the residual subproblem, and results in a savings in the redundant executions of these constraints. The resulting decomposition has two levels, and two subproblems. Figure 1.7 gives a graphical demonstration of the reduction in artificial dependencies obtained by DOI-decomposition.

3.5.2 Selecting a decomposition

The issue then is how to select the best control sequence prefix to yield the most beneficial decomposition. Since the solution-cost improvement comes from the reduction of independent contexts of the constraints in the residual subproblem, there are two factors to maximize:

- The number of variables in the prefix. Larger number of variables will include larger portions of the independent contexts of remaining constraints.
- The number of constraints in the tail. Larger number of residual constraints that get their independent contexts reduced, the greater the reduction in redundant constraint checks.

These two factors oppose each other, since a prefix with a larger number of variables also leaves a smaller tail, with fewer constraints. The best decomposition must find an ideal balance between the two factors. Note that the prefix also contains constraints, and the decomposition does not yield any improvement in their performance. This is because we do not reorder the prefix control sequence after decomposition, based upon the assumption that the original control sequence was selected as optimal for the basic search algorithm, and so its prefix would also be an optimal ordering.

The optimal DOI-decomposition of a control sequence is induced by a division of the sequence into a prefix and a suffix which maximizes the reduction in the number of redundant constraint checks for constraints in the residual subproblem formed from the suffix. For simplicity, we will assume that the relative order of constraints in the suffix is maintained after decomposition. Then for each suffix constraint, the reduction in its redundant testing after decomposition is exactly the size, in the original sequence, of that independent portion of the context at the constraint consisting of variables from the suffix.

We can determine this contribution of a control sequence's prefix to the redundant testing of a constraint in the remaining suffix as follows. Let C_p be a prefix of the control sequence C, and let T be a constraint in the resulting suffix, such that $C = \langle C_p C_a T C_b \rangle$. In addition, let Y_T be the argument set of T, and Y_p the variables bound in C_p. The contribution by C_p to the independent context of T comes from the variables in the set $Y_p - Y_T$. However, some of these variables may be required by the constraints of C_a. We shall use Y_{at} to denote the arguments of those constraints. We now define the *Degree of Independence* $\mathrm{DOI}_C(T, C_p)$ of the constraint T in the control sequence C from its prefix C_p, as the number of redundant tests of T in C directly attributable to the

portion of the context established by the prefix C_p that is independent of the portion of the sequence $\langle C_a\ T\rangle$ following C_p and including T.

With the definitions

$$\begin{aligned} C &= \langle C_p\, C_s\rangle = \langle C_p\, C_{s1}\, T\, C_{s2}\rangle \\ Y_p &= \mathrm{Vars}(C_p) \\ Y_T &= \mathrm{Args}(T) \\ Y_{at} &= \mathrm{Args}(C_a) \end{aligned}$$

and

$$\begin{aligned} \mathrm{Checks}_C(T) &= \mathrm{Numsol}_C(C_p\, C_{s1}) \\ \mathrm{Redundant\text{-}Checks}_C(T) &= \mathrm{Checks}_C(T) - \mathrm{Numsol}_C(C_p\, C_{s1}|Y_T) \end{aligned}$$

we can express $\mathrm{DOI}_C(T, C_p)$, the degree of independence of T from C_p in C, as

$$\begin{aligned} &\mathrm{DOI}_C(T, C_p) \\ &= (\mathrm{AvgNumsol}_C(C_p\, C_{s1}|Y_p - Y_T - Y_{at}) - 1, C_p\, C_{s1}|Y_T) \times \mathrm{Numsol}_C(C_p\, C_{s1}|Y_T) \\ &\mathrm{DOI}_C(C_s, C_p) \\ &= \sum_{T \in C_s} (\mathrm{AvgNumsol}_C(C_p C_{s1}|Y_p - Y_T - Y_{at}, C_p C_{s1}|Y_T) - 1) \times \mathrm{Numsol}_C(C_p C_{s1}|Y_T) \end{aligned}$$

The ideal DOI-decomposition of the control sequence C will then maximize the combined degree of independence $\mathrm{DOI}_C(C_s, C_p)$ of the suffix C_s from the prefix C_p.

Again, while easy to formally specify, the degree of independence is not easy to evaluate, or even estimate. So once again, we resort to heuristic procedures.

First, note that it is generally more useful for the prefix C_p to end in a constraint and the suffix C_s to begin in a variable. If C_s should begin in a constraint, its argument variables would be bound in C_p, and the corresponding residual will need to recall these variables at the beginning of its sequence, thus defeating the purpose of the decomposition. The objective is to build a residual such that recalled variables occur as deep in its sequence as possible, thus reducing the independent contexts of shallower constraints. Adding constraints to the end of the corresponding prefix C_p will also help

reduce the size of the solution cache. This means that a control sequence can only be divided at points between a cluster of constraints and a cluster of variables. With K constraints in the problem, a sequence of $N+K$ nodes can be divided in at most $K-1$ places.

Having restricted our space of possible decompositions, we must now evaluate the suitability of each candidate decomposition. Since the actual combined degree of independence, of a control sequence suffix from the corresponding prefix, is too difficult to determine, we resort to heuristic estimates, and select the decomposition that maximizes this estimate. Three levels of approximation suggest themselves, and are given below in increasing order of precision. We shall continue to use the above model of $C = \langle C_p\, C_a\, T\, C_b \rangle$, where the candidate decomposition extracts the prefix C_p as the subproblem.

1. *Artificial Variable Dependencies.* A crude estimate of the degree of independence of a constraint T from a control sequence C_p is to simply count the number of variables in C_p that T is independent of. We will call this measure the degree of artificial serial dependence, or, the *artificial variable dependence* ($\mathrm{AVD}(T, C_p)$), of the constraint T from the control sequence C_p. We must not, however, forget the portion C_a of the control sequence occurring between T and C_p. The DOI-decomposition procedure recalls the cached variables of the extracted subproblem, formed from the prefix C_p, into the residual problem formed from the control suffix $\langle C_a\, T\, C_b \rangle$. The effective reduction in the artificial dependence of the constraint T after DOI-decomposition is tempered by the cached variables needed for the evaluation of constraints in C_a. The approximate combined degree of independence reduced by a candidate DOI-decomposition, $\mathrm{DOI}'_C(C_s, C_p)$, can be calculated by

$$\begin{aligned} \mathrm{DOI}'_C(T, C_p) &= \mathrm{AVD}(T, C_p) - |Y_{at}| = |Y_p - Y_{at} - Y_T| \\ \mathrm{DOI}'_C(C_s, C_p) &= \mathrm{AVD}(C_s, C_p) = \sum_{T \in C_s = \langle C_a T C_b \rangle} \mathrm{AVD}(T, C_p - Y_{at}) \end{aligned}$$

We can also use artificial variable dependence as a crude estimate of the potential for redundant computations in a control sequence:

$$\mathrm{AVD}(C) \quad = \sum_{T \in C = \langle C_p T C_s \rangle} \mathrm{AVD}(T, C_p)$$

Note that artificial variable dependence is truly a measure only of the topology of the total ordering of a problem's constraint network induced by the control sequence. It does not actually estimate the actual number of redundant constraint checks, for which an estimate of the size of the independent portion of the context at a constraint would be needed. However, it is also a very easy measure to calculate. The next two heuristics attempt to measure the actual number of redundant constraint checks of a constraint.

2. *Sizing the domains of the independent variables.* A more precise estimate of a set of independent variables' contribution to the redundant checking of a constraint can be made by taking the domains of the independent variables into account. For instance, we can use the combined domain size of the independent variables in the prefix as the estimated degree of independence of a suffix constraint from the prefix. This is similar in spirit to the popular variable ordering heuristic used in Backtrack and Forward-Check, selecting a control that orders variables in increasing domain size.

$$\begin{aligned} \mathrm{DOI}'_C(T, C_p) &= \mathrm{Dom}(Y_p - Y_{at} - Y_T) \\ \mathrm{DOI}'_C(C_s, C_p) &= \sum_{T \in C = \langle C_p T C_s \rangle} \mathrm{Dom}(Y_p - Y_{at} - Y_T) \end{aligned}$$

3. *Using estimates of constraint tightness.* The precision of the previous approximation can be further improved by reducing the domain size of the independent variables by a factor that estimates the filtering of those domains be all the constraints preceding T in the control sequence C. Elaborate methods to produce this approximation can use Monte Carlo techniques to estimate each constraint's solution density (proportion of a constraint's domain that satisfies the constraint).

As an example, we now analyze the three possible decompositions of C_1, the Backtrack control sequence for the example problem P_1 of figure 2.2. We shall use artificial

variable dependence as the heuristic estimate of the degree of independence reduced by decomposition. The four possible DOI-decompositions are:

$$
\begin{aligned}
C_1 &= \langle X_4 X_5 T_{45} X_3 T_{35} X_1 T_{13} T_{14} X_2 T_{24} \rangle \\
\text{Checks}_{C_1} &= 16 + 36 + 108 + 81 + 324 = 565 \\
\text{AVD}(C_1) &= 0 + 1 + 2 + 2 + 3 = 8 \\
\text{AVD}_{avg}(C_1) &= \text{AVD}(C_1)/|R| = 8/5 = 1.6
\end{aligned}
$$

$$
\begin{aligned}
C_{\text{DOI(1)}_1} &= \langle (X_4 X_5 T_{45})\ (X_5' X_3 T_{35} X_1 T_{13} X_4' T_{14} X_2 T_{24}) \rangle \\
\text{DOI}_{C_{\text{DOI}(1)_1}} &\approx \text{AVD}(\langle X_3 T_{35} X_1 T_{13} T_{14} X_2 T_{24} \rangle, \langle X_4 X_5 T_{45} \rangle) \\
&= 1 + 1 + 0 + 0 = 2 \\
\text{Checks}C_{\text{DOI(1)}_1} &= (16) + (12 + 36 + 81 + 324) = 469
\end{aligned}
$$

$$
\begin{aligned}
C_{\text{DOI(1)}_2} &= \langle (X_4 X_5 T_{45} X_3 T_{35})\ (X_3' X_1 T_{13} X_4' T_{14} X_2 T_{24} X_5') \rangle \\
\text{DOI}_{C_{\text{DOI}(1)_2}} &\approx 2 + 1 + 1 = 4 \\
\text{Checks}C_{\text{DOI(1)}_2} &= (16 + 36) + (12 + 27 + 108) = 199
\end{aligned}
$$

$$
\begin{aligned}
C_{\text{DOI(1)}_3} &= \langle (X_4 X_5 T_{45} X_3 T_{35} X_1 T_{13} T_{14})\ (X_4' X_2 T_{24} X_5' X_3' X_1') \rangle \\
\text{DOI}_{C_{\text{DOI}(1)_3}} &\approx 3 \\
\text{Checks}C_{\text{DOI(1)}_3} &= (16 + 36 + 108 + 81) + (12) = 253
\end{aligned}
$$

In this case, this approximation works well, and gives a ranking of the actual costs of the three decompositions. The second decomposition is also the sequence $C_{\text{DOI(1)}}$ discussed in section 3.2 above.

The DOI-decomposition C_{DOI} of a control sequence $C = \langle C_p\ C_s \rangle$ can be seen to reduce the artificial dependencies present in C by reducing the artificial variable dependencies of constraints in a suffix portion (C_s) of C. This reduction may be a result of not only the degree of independence of the suffix C_s from the prefix C_p, now eliminated by the decomposition. Rearrangement of nodes in the suffix C_s when building the residual

subproblem may yield further reductions in artificial dependence. Therefore,

$$\mathrm{AVD}(C) - \mathrm{AVD}(C_{\mathrm{DOI}}) \geq \mathrm{AVD}(C_s, C_p)$$

We will use the function AVDr to denote the reduction or difference in AVD between two control sequences

$$\mathrm{AVDr}(C', C) = \mathrm{AVD}(C) - \mathrm{AVD}(C')$$

Often when comparing different decompositions of a sequence C, we will drop the second argument for brevity.

3.5.3 Analysis of the decomposition procedure

The degree of independence based decomposition strategy divides a control sequence into a prefix and a suffix. The prefix is solved as the first subproblem, followed by the residual subproblem formed from the suffix. While different divisions of the given control sequence will result in different bottom-up solution costs, the next theorem establishes an interesting property of the DOI-decomposition solution strategy.

Theorem 3.3 *Decomposing a Backtrack control sequence using DOI-decomposition will never increase the number of constraint checks. That is (using BT to denote Backtrack),* $Checks_{BT}(C') \leq Checks_{BT}(C)$ *for any DOI-decomposition* C' *of* C*, when searching for all solutions.*

Consider a control sequence $C = \langle C_p\ C_s \rangle$, resulting in the DOI decomposition $C' = \langle (C_p)\ (C'_s) \rangle$, where the residual C'_s contains the constraints from C_s. We will assume that the solution cache tree is used to store the solutions of the first subproblem (C_p). The cost of the decomposition is the sum of the cost of solving the two subproblems.

$$\mathrm{Checks}(C') = \mathrm{Checks}(C_p) + \mathrm{Checks}(C'_s)$$

The cost of solving C can also be similarly broken down (see Chapter 2).

$$\mathrm{Checks}(C) = \mathrm{Checks}(C_p) + \mathrm{Numsol}(C_p) \times \mathrm{AvgCost}(C_s, C_p)$$

Clearly then the difference in the costs between the two control sequences is incurred in the solution of the constraints of the suffix C_s. To prove the theorem, we just have to establish that decomposition does not increase this cost component.

We now consider the cost incurred at a constraint T embedded in the suffix C_s, before and after decomposition. The residual C'_s is formed by maintaining the same order of variables and constraints as in C_s, and inserting variables cached in the first subproblem immediately before the constraint that needs them to be bound. Let T be the first constraint in C_s to depend upon some variables bound in C_p. We can then write the argument set of T as $\text{Args}(T) = Y_{Tp} \cup Y_{Ts}$, where the variables Y_{Tp} are the ones bound in C_p. We can now represent the two control sequences as

$$\begin{aligned} C &= \langle C_p \; C_{s1} \; T \; C_{s2} \rangle \\ C' &= \langle (C_p) \; (C_{s1} \; Y'_{Tp} T \; C'_{s2} \rangle \end{aligned}$$

where all the variables in the arguments Y_{Ts} of T are bound in C_{s1}. We will use the primed superscript in Y'_{Tp} to denote that these are really cached variables, restricted to generating the corresponding portions of the solutions to the first subproblem (C_p). Thus

$$\text{Numsol}(Y'_{Tp}) = \text{Dom}(Y'_{Tp}) = \text{Numsol}(C_p | Y_{Tp})$$

The number of tests of T in C' is then

$$\begin{aligned} \text{Checks}_{C'}(T) &= \text{Numsol}_{C'}(C_{s1}) \times \text{AvgNumsol}_{C'}(Y'_{Tp}, C_{s1}) \\ &= \text{Numsol}(C_{s1}) \times \text{Numsol}(C_p | Y'_{Tp}) \end{aligned}$$

where C_{s1} is a self-contained valid control sequence, with no inter-dependence with C_p. Similarly, the prefix C_p is also a self contained valid control sequence, whose solution set does not depend upon whether it occurs in C or in C'. These facts also help in developing an expression for the number of tests of T in C.

$$\begin{aligned} \text{Checks}_{C}(T) &= \text{Numsol}_{C}(C_p) \times \text{AvgNumsol}_{C}(C_{s1}, C_p) \\ &= \text{Numsol}(C_{s1}) \times \text{Numsol}(C_p) \end{aligned}$$

The size of the projection of a set of tuples onto some of its component variables can never exceed the size of the unprojected set, i.e.

$$\text{Numsol}(C_p) \geq \text{Numsol}(C_p|Y_{Tp}) \quad \text{where} \quad Y_{Tp} \subseteq \text{Vars}(C_p)$$

This yields

$$\begin{aligned} \text{Checks}_{C'}(T) &\leq \text{Checks}_{C}(T) \quad \text{and also} \\ \text{Numsol}_{C'}(C_{s1} Y'_{Tp} T) &\leq \text{Numsol}_{C}(C_p\ C_{s1} T) \end{aligned} \tag{3.1}$$

Clearly if we include the possibility that T above may not depend upon any of the variables of C_p, making Y_{Tp} an empty set, the above propositions will still hold true.

Having established the basis, we now take the inductive step. Consider now the constraint T embedded anywhere in C_s and possibly depending upon some of the variables of C_p. We can now represent the two control sequences as follows

$$\begin{aligned} C &= \langle C_p\ C_{s1}\ T\ C_{s2} \rangle \\ C' &= \langle (C_p)\ (C'_{s1}\ Y'_{Tp} T\ C'_{s2} \rangle \end{aligned}$$

this time defining the argument set $\text{Args}(T) = Y_{Tp} \cup Y_{Ts}$, where the variables Y_{Tp} are those arguments of T that are bound in C_p, and not occurring as arguments to any constraints in C_{s1}. In the decomposed sequence, the residual prefix C'_{s1} represents C_{s1} with cached variables inserted as required by the constraints in C_{s1}, according to the rule for building the residual subproblem described in the beginning of this proof. Then,

$$\begin{aligned} \text{Checks}_{C}(T) &= \text{Numsol}_{C}(C_p\ C_{s1}) \qquad \text{and} \\ \text{Checks}_{C'}(T) &= \text{Numsol}_{C'}(C'_{s1}) \times \text{AvgNumsol}_{C'}(Y'_{Tp}, C'_{s1}) \\ &= \text{Numsol}_{C'}(C'_{s1}) \times \text{Numsol}_{C}(C_p|Y_{Tp}) \end{aligned}$$

Making the inductive assumption $\text{Numsol}_{C'}(C'_{s1} Y'_{Tp}) \leq \text{Numsol}_{C}(C_p\ C_{s1})$ based upon the above result 3.1, where $Y_{Tp} \subseteq \text{Vars}(C_p)$,

$$\begin{aligned} \text{Checks}_{C'}(T) &\leq \text{Numsol}(C_p\ C_{s1}) \\ &\leq \text{Checks}_{C}(T) \end{aligned}$$

□.

Note that the proof depends upon the particular ordering chosen for the nodes of the residual subproblem after decomposition. In general, however, if available heuristics can produce a better ordering, it should be used.

3.6 Cost of using the Bottom-Up framework

The two decomposition techniques described in this chapter are both aimed at reducing the number of redundant constraint checks during solution, and rely on the Bottom-Up solution framework to achieve this. While the natural implication is that this approach is best suited for problems in which the cost of testing constraints dominates, other problems can also benefit. (Chapter 6 describes some sample applications where the cost of a constraint test is comparable to the cost of generating a value for a variable.)

In this section, we take a closer look at the implications of using DOI and IS decomposition with Bottom-Up solution on run-time and memory usage.

3.6.1 Run-time

There are two ways in which the Bottom-Up solution framework adds to the run-time of solving a CSP: the time needed to build the solution caches for each independent subproblem, and the time required to generate cached variables (in the residual subproblem, in the case of DOI and IS decomposition).

In section 3.3.2 we measured the cost, in number of steps, of building the solution cache. To summarize, caching s solutions for a subproblem of n variables requires $O(s^2 + sn)$ steps for the linked-list data structure of figure 3.2, and $O(sn)$ steps for the vector-node based data structure of figure 3.3. In either case, it is useful to keep the size (number of variables) of each cached subproblem and the number of such subproblems small. This means that when using IS decomposition, it is better to extract low arity constraints in the independent set, and when using DOI decomposition, it is better to select relatively shallow "cuts".

A primary goal in selecting a data structure for the solution cache was to keep the

```
Generate( v, backtrack-flag )
{Normal generation from an array Domain}
(1)  if  not backtrack-flag
(2)      then  NextValueIndex(v) = 1
(3)  if  NextValueIndex(v) ≤ Dom(v)
(4)      then  newValue = Domain(v)[ NextValueIndex(v) ]
(5)            NextValueIndex(v) += 1
(6)            return  newValue
(7)      else  NextValueIndex(v) = 1
(8)            return  FAIL
end

GenerateFromCache-LL( v, backtrack-flag )
{Generates from the Linked-List cache of figure 3.2}
(1)  if  not backtrack-flag
(2)      then  CurrValueNode(v) = NIL
(3)  if  CurrValueNode(v) == NIL
(4)      then  u = PreviousVariableInCache(v)
(5)            CurrValueNode(v) =
                  NextVariableFirstValueNode( CurrValueNode(u) )
(6)            return  Value( CurrValueNode(v) )
(7)      else  CurrValueNode(v) = NextNode( CurrValueNode(v) )
(8)            if  CurrValueNode(v) ≠ NIL
(9)               then  return  Value( CurrValueNode(v) )
(10)              else  return  FAIL
end

GenerateFromCache-VN( v, backtrack-flag )
{Generates from the Vector-Node cache of figure 3.3}
(1)  if  not backtrack-flag
(2)      then  CurrValueNode(v) = NIL
(3)  if  CurrValueNode(v) == NIL
(4)      then  u = PreviousVariableInCache(v)
(5)            CurrValueNode(v) =
                  NextVariableNode( CurrValueNode(u) )[ NextValueIndex(u) - 1 ]
(6)            NextValueIndex(v) = 2
(7)            return  Domain(CurrValueNode(v))[ 1 ]
(8)      elseif  NextValueIndex(v) ≤ DomainSize(CurrValueNode(v))
(9)            then  newValue = Domain(CurrValueNode(v))[ NextValueIndex(v) ]
(10)              NextValueIndex(v) += 1
(11)              return  newValue
(12)           else  CurrValueNode(v) = NIL
(13)              return  FAIL
end
```

Figure 3.5: Procedures for generating a variable's value.

cost of generating values from the cache low. The algorithms for generating a value from a variable's (original and cached) domain are shown in figure 3.5.

The argument `backtrack-flag` is `TRUE` if the procedure is invoked as a direct result of backtracking in the search control. This argument and the first two lines in each procedure can be omitted if the search algorithm uses chronological backtracking. Initialization for `Generate` requires `NextValueIndex`(v) to be set to 1 for all variables; for `GenerateFromCache-LL` and `GenerateFromCache-VN`, `CurrValueNode`(v) should be set to `NIL`, except in the case of the first variable in each solution cache it should be set to the root node for that cache.

The procedure `Generate`, for generating from an ordinary domain available as an array of values, is the simplest of all three. `GenerateFromCache-VN` is almost identical, with the addition of lines 3 - 7 which are executed once each time a new value is generated for the previous variable. Both algorithms for generating from a solution cache are of similar complexity, and only slightly more complex than `Generate`. Note that in all three cases, a call to the generators returning `FAIL`, indicating that search has run through that variable's active domain in this pass, is quite as expensive as one returning a valid binding. We shall be counting this call when measuring the number of generate events *Gens(C)* for a sequence C.

While DOI and IS decompositions achieve a reduction both in the number of times some of the problem's constraints get tested and the number of times their variables are bound, they also add some solution-cache generate events. It is possible for this added cost to become quite large, especially in the case of DOI decompositions of hard problems if all the cached variables are recalled very deep in the residual subproblem's sequence. (Generally the cached variables will be spread throughout the residual sequence).

The following analysis counts the number of generate events under Backtrack for the simple sequence C_1 and its IS and DOI decompositions (see section 3.2), for the example CSP P_1 (figure 2.2). The cache-generate counts are underlined.

$$
\begin{aligned}
C_1 &= \langle X_4 X_5 T_{45} X_3 T_{35} X_1 T_{13} T_{14} X_2 T_{24} \rangle \\
\text{Gens}(C_1) &= 5 + 20 + 45 + 135 + 405 = 610 \\
C_{is(1)} &= \langle (X_3 X_1 T_{13})\ (X_2 X_4 T_{24})\ (X_4' X_5 T_{45} X_3' T_{35} X_1' T_{14} X_2') \rangle \\
\text{Gens}(C_{is(1)}) &= (5 + 20) + (5 + 20) + (\underline{4} + 15 + \underline{36} + \underline{108} + \underline{324}) \\
&= 65 + \underline{472} \\
C_{\text{DOI}(1)} &= \langle (X_4 X_5 T_{45} X_3 T_{35})\ (X_3' X_1 T_{13} X_4' T_{14} X_2 T_{24} X_5') \rangle \\
\text{Gens}(C_{\text{DOI}(1)}) &= (5 + 20 + 45) + (\underline{4} + 20 + \underline{36} + 135 + \underline{324}) \\
&= 225 + \underline{364}
\end{aligned}
$$

If we estimate the average cost of a generate from cache to be 40% greater than a normal generate (see experiment in section 4.3.2), the total generate cost of $C_{is(1)}$ comes to 725, and of $C_{\text{DOI}(1)}$ to 734, a small increase when compared to the reduction in constraint checks (413 and 366, respectively). Note the placement of X_4' before X_5 in the residual subsequence of $C_{is(1)}$ to keep cache generates low.

Between the two decompositions, $C_{is(1)}$ has the lower cache build cost, requiring two caches of 9 solutions each, both for two variables ($sn = 9 + 9 = 18$), whereas $C_{\text{DOI}(1)}$ requires a single cache of three variables and 27 solutions ($sn = 27 \times 3 = 81$). In either case, the costs of building and generating from the cache are much smaller than the reduction in constraint checks.

In general, care must be taken when selecting a decomposition that the added costs of building the subproblem solution caches and generating from the solution caches are adequately offset by a corresponding reduction in other generate and test events for the decomposition to be useful.

In problems with expensive constraints, generate events will only contribute a small proportion of the overall cost of solving a problem, and so might not be an important factor in selecting a decomposition. This is true in the case of problems where constraints require a table look-up to evaluate an argument binding, e.g. consulting an ephemeris file to determine the position of a space body at a given time ([51]), and the

problems of Chapter 4.

3.6.2 Memory Overhead

The Bottom-Up solution framework provides a means to reduce constraint checks at the expense of some extra storage space required to store the subproblem solution caches. We now take another look at this cost.

In section 3.3.2 we saw that a solution cache for n variables and s solutions will require $O(sn)$ memory space. For the hardest problems, the value of s can rise exponentially with increasing n. When this memory requirement exceeds the available RAM in a computer, suddenly the temporal cost of swapping memory blocks from hard disk becomes a factor in the cost of solving a decomposition.

DOI and IS decompositions will usually have very different memory requirements for the same problem. The DOI decomposition achieving the greatest reduction in artificial dependence in a large CSP will usually involve an independent subproblem of a fairly large number of variables; and if the problem is "hard" (large search tree), this subproblem will also have a large number of solutions. An IS decomposition, on the other hand, will usually consist of a large number of independent subproblems, where the size of each subproblem is fairly small. Therefore generally, IS decompositions will usually have a smaller memory requirement.

While not a large problem itself, the IS and DOI decompositions of the example CSP P_1 discussed above provide a good demonstration of this fact. The decomposition $C_{\mathrm{DOI}(1)}$ requires a solution cache for 27 solutions in 3 variables, requiring $3+9+27 = 39$ nodes of a linked-list cache. The IS decomposition $C_{is(1)}$ on the other hand has two independent subproblems, each with 9 solutions in 2 variables, requiring a total of only 24 nodes.

While the memory requirement for a DOI decomposition depends, in the worst case, exponentially upon the size (number of variables) of the independent subproblem (size of the prefix when cutting a simple sequence), for an IS decomposition it depends exponentially on the arity and linearly on the number of the extracted constraints in the independent set. In section 3.4.3 we observed that an independent set of m

binary constraints will require at most $m(d^2 + d)$ linked-list nodes, where $m \leq N/2$ for an N variable problem. As a comparison, the non-hierarchical Backtrack algorithm would require N memory locations to store active partial solutions, and non-hierarchical Forward-Check could require as much as $N(N-1)d/2$ memory locations to store all the active domains. For example, a binary CSP with 100 variables and a maximum domain size of 5 could need 24,750 memory locations for Forward-Check, but at most only 4,500 memory locations (each linked-list node taking 3 memory locations – see figure 3.2) for IS decomposition under Backtrack. A DOI decomposition of the same problem, with an independent subproblem of only 6 variables, could require as much as 46,875 memory locations if all the possible bindings solved the subproblem.

So IS decomposition with Backtrack can actually prove to be a more viable alternative to serial Forward-Check for very large problems. In fact using a combination of both a shallow DOI decomposition and an IS decomposition should provide the fastest solution of very large CSPs, while still keeping memory requirements low.

3.7 General Discussion of the Two Decomposition Techniques

We now note some general principles, heuristics and intuitions applicable to both decomposition strategies. First, note that the estimated degree of independence suggested above in the selection of a DOI-decomposition can also be used to estimate the quality of an IS decomposition. The same estimates can also be used to compare the suitabilities of the two decompositions. However, the various estimated measures proposed above do remain just that — estimates, and should be used with caution.

In the proof of the above theorem for DOI-decomposition, we inserted cached variables in the residual immediately before the dependent constraint. In the example decompositions of example problem P_1 (see section 3.2) however we inserted the cached variable before any variable generators immediately preceding the dependent constraint (e.g. $\langle \ldots (X_5' X_3 T_{35} \ldots) \rangle$). This does not alter the applicability of the theorem, and also helps to reduce the generator overhead, as the cached variables have already had their domains trimmed by previous subproblems, and also generating from a cache is more

expensive than an ordinary generate.

The DOI-decomposition may be applicable to CSPs for which the ISB and ISF procedures do not produce a usable decomposition. The ideal CSP for IS-decomposition is likely to have several extractable small arity constraints. The DOI-decomposition does not depend upon the presence of small arity constraints. However, if a problem has only global constraints, or if its control sequences do not exhibit any artificial dependencies, neither decomposition strategy can be applied. Some of these ideas will be tested in the experiments in the next chapter.

3.8 Summary

In this chapter, we examined two new problem decomposition methods, applied to CSPs, that directly address the problem of redundant computations of constraints. Both methods produce a two-level decomposition that can be solved using the bottom-up solution framework also presented in this chapter. The general strategy is as follows:

1. Extract one or more independent subproblems from the given CSP, leaving behind a residual subproblem whose constraints may depend upon the variables of the other subproblems.

2. Solve the independent subproblems using your favourite algorithm, and cache their solutions.

3. Solve the residual subproblem while recalling the cached solutions such that all the problem's variables get instantiated in this step. The solutions of this step are the solutions to the original CSP.

The problem decompositions are based on partitioning the problem's constraint set, and the two procedures differ on how this partitioning is achieved.

The decomposition methods do not rely on being able to produce a completely independent decomposition of the given problem, and the bottom-up solution framework is devised to handle this situation efficiently. The framework itself is general, and can be used by other decomposition procedures.

For each decomposition procedure, we were able to analyze some situations where their employment would benefit the problem. In particular, we found that using DOI-decomposition can never make the problem worse.

Chapter 4 presents an empirical evaluation of these ideas on randomly generated constraint satisfaction problems. Chapter 6 then applies these decomposition techniques on some sample application domains.

Chapter 4

Experiments with Decomposition by Constraint-Set Partitioning

In this chapter we investigate the utility of the DOI and IS decomposition strategies on randomly generated constraint satisfaction problems. The performance of the decompositions is charted against various problem size parameters.

4.1 Experiment Objectives

The main goal of these experiments is to understand the performance behaviour of DOI and IS decompositions as applied to 'n-ary' CSPs. Specifically, the goals are:

- Evaluate AVD reduction (AVDr) as a heuristic for selecting a good decomposition.
- Compare the performances of DOI and IS-decompositions.
- Test the hypothesis that the relative benefit of using decomposition increases with harder problems and larger AVD reductions.
- Chart the degree of benefit from decomposition against various problem size parameters.
- Evaluate the efficiency of the Bottom-Up solution framework.

4.2 Experiment Methodology

An empirical evaluation of these goals was performed on randomly generated problems. The general experiment procedure, which was completely automated, was as follows. Each experiment described in the next section tested a small number of problem-sets. Each set of 60 problems was generated by the random problem generator based upon

a set of parameter values describing problem properties. A simple control sequence was then automatically constructed for each problem using some ordering heuristics. Then DOI, ISB and ISF decomposition procedures were applied to this control sequence, yielding some multi-level control sequences. The declarative representation of each problem produced by the random problem generator, together with all the control sequences, was then automatically translated into a C program. The C program was then compiled with each of three search algorithms and executed. The execution produced run statistics for every control sequence, and these are collected and analyzed in the next section.

A necessary limiting factor in all experiments on random problems is the number of random problems tested. After some initial experiments in Lisp, the search algorithms were finally implemented in C with a large improvement in performance. Nevertheless, several decisions had to be made to limit the scope of the experiments due to limited computing resources. Thus each experiment tested only three or four problem-sets, representing that many different values for the problem-property parameters. Each problem-set was limited to 60 problems. The problems themselves were kept small, with number of variables ranging from 6 to 9, and domain size ranging from 6 to 8.

4.2.1 Random generation of CSPs

The random problem generator is a Lisp program that produces a declarative representation of a randomly generated CSP based upon a set of specified problem characteristics. The problem generator has been programmed to construct a set of constraints of arbitrary arity and one global constraint for each problem.

Constraint satisfaction problems with constraints of arbitrary arity can be classified along many problem size and complexity dimensions. The particular dimension parameters used in the random problem generator for these experiments are described below. Some parameter values were chosen to limit the physical size of the problem which directly affected the size of the executable. In each experiment, for each set of parameter values, a set of 60 CSPs was generated.

Number of variables (N). These ranged from 6 to 9 in the experiments. This number was kept small because of limited computing resources.

Domain size (d). Each variable in a problem was given the same domain size. Values used ranged from 5 to 8.

Number of constraints ($K_{min} \leq K \leq K_{max}$). A range was used to make it easier for the problem generator to generate constraints of random arities. The argument set of each constraint was guaranteed unique within the problem, and the resulting constraint graph connected all the variables.

Constraint tightness (c). Constraints were assigned a list of solutions by the problem generator such that c was the proportion of each constraint's domain that satisfied that constraint. The problem generator attempted to give each constraint the same tightness, subject to the limit s_{max} on the number of solutions to each constraint. For each constraint T_i with argument set Y_i,

$$\text{Numsol}(T_i)/\text{Dom}(Y_i) = \text{Numsol}(T_i)/d^{|Y_i|} \approx c$$

subject to the restriction $\text{Numsol}(T_i) \leq s_{max}$.

Maximum number of constraint solutions (s_{max}). For a particular constraint tightness, as constraint arity increases, the number of solutions to each constraint also increases. As this directly affects the physical size of the problem, a maximum value of $s_{max} = 9000$ was used. This value allowed a maximum tightness of 0.6 at arity 6 and domain size 5. Without this limit, the storage space required for the list of solutions to a constraint of arity 6, tightness 0.6 and domain size 8 would have exceeded 6 MBytes!

Maximum constraint arity (k_{max}). A maximum constraint arity of 6 was used to keep the physical problem size small. Without this limit, the limit on the number of solutions to each constraint would force the tightness of high arity constraints to decrease to very small values. The single exception to this rule was that each problem was also given one global constraint.

Constraint arity profile (b). Constraints were generated with a minimum arity of 2, and upto a specified maximum arity (k_{max}). The probability of generating a particular arity was dictated by the constraint arity profile as follows. The probability that a constraint had arity k was controlled by the formula

$$p(k-1) = b \times p(k), \qquad \text{for } 2 \leq k \leq k_{max}$$

For example, a value of $b = 1$ would generate each arity with equal likelihood. The global constraint in each problem was excepted from this rule.

Minimum number of solutions (S_{min}). In an attempt to generate problems of widely varying difficulties, each problem was given a specified minimum number of solutions. How much the actual number of solutions exceeded the minimum depended upon the interaction between constraints. This interaction was completely random.

This was the only parameter allowed to change within a problem-set. Each set of 60 problems was generated to contain 10 problems for each of the following minimum number of solutions: 1, 5, 25, 125, 625, 3125.

4.2.2 Problem sets

The experiments described in the next section investigate how utility of the decomposition strategies changes with changes in problem parameters. In each experiment, one problem parameter was selected to vary over a particular range of values. For example the number of variables N was varied in one experiment while keeping other parameters constant. However the set of parameters accepted by the random problem generator used here do not completely cover all the dimensions of variability in CSPs. In particular, there is no control on the interaction between the constraints in a problem. In order to prevent this omission from skewing results, a set of problems was generated for each set of problem parameter values, and the performance results averaged.

Each problem-set in the experiments consisted of 60 problems, ten each for the following six values of minimum number of solutions:

$$S_{min} \in \{1, 5, 25, 125, 625, 3125\}$$

Since all other parameters in each set were kept at a chosen constant value, we can briefly denote each set with the list of these values. For example,

$$N = 9,\ d = 5,\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$$

represents a set of problems of 9 variables and between 9 and 12 constraints (inclusive), each variable with a domain of size 5, and each constraint with a tightness of 0.2. The arity of the constraints in each problem in the set are chosen with uniform randomness ($b = 1.0$) between the values 2 and 6 (also inclusive), except for the single global constraint assigned to each problem. The number of solutions to each constraint is limited to a maximum of 9000. Each problem-set tested in the experiments is generated independently from other problem-sets.

The experiment descriptions given below will reveal that the random problem generator was not as succesful in producing a broad spectrum of problem difficulties for algorithms that stopped search after the first solution. There has been some interest in the research community in finding regions of hard problems. This work has not so far been extended to 'n-ary' CSPs.

4.2.3 Search Algorithms

Every problem set was tested with and without decomposition on three basic search algorithms: Backtrack and Forward-Check looking for all solutions, and Backtrack looking for the first solution. The primary focus of this thesis is on problems where all or many solutions are required. Unless explicitly mentioned, all discussion below will refer to these problems. For convenience, we will use *Backtrack-1* to denote the Backtrack algorithm that stops search after finding the first solution.

The Backtrack algorithm was included in the experiments because it is the simplest and most well known search algorithm, and also has the lowest memory requirement. Forward-Check was selected because it is the fastest known algorithm in widespread use (e.g. [31]).

Three additional search algorithms — Dynamic Forward-Check using the "fail-first" heuristic ([31]), Backmarking (ibid.) and Backjumping ([8]) — were also considered, but

eliminated by early experiments for not performing as well as simple Forward-Check. The fail-first heuristic advocates selecting the variable with the smallest active domain as the variable to bind next. While this works well for binary CSPs, it fails badly on n-ary problems because the heuristic ignores the topology of the constraint network, and has a tendency to defer higher arity constraints towards the end of the control sequence. Backjumping is an attempt to avoid generation of values for independent variables when trying to satisfy a failed constraint. Again early experiments showed that this algorithm was rapidly decaying to ordinary chronological backtracking for two reasons: failure of high arity constraints and searching for solutions beyond the first solution to the CSP, both of which tended to mark all preceding variables as backjump destinations. Finally, early experiments verified the conclusion in [31] that Backmarking generally did not perform as well as Forward-Check.

4.2.4 Control Sequences

The simple control sequence used for basic search was generated using a combination of the following heuristics:

- Minimize the total depth and bandwidth of constraints in the sequence ([47, 71]).
- Order adjacent constraints on increasing arity.

This simple control sequence was also used as the basis for DOI, ISF and ISB decompositions. All the decompositions were tested, and the experiments rated the performance of the decompositions against the simple sequence as the performance benchmark.

For each problem, the ISF and ISB procedures produced up to two different IS-decompositions of the benchmark simple control sequence. Constraints in the independent set were limited to an arity of half the number of variables in the problem. So for some problems the decomposition procedure did not produce an IS-decomposition.

The DOI-decomposition procedure produced a set of decompositions by cutting the benchmark sequence at various points. The cuts were restricted to points immediately following the last constraint-node before a variable.

In the rest of this chapter, we will denote the benchmark control sequence with the symbol C_0, and use C^0_{DOI} for the best DOI-decomposition of C_0, and C_{ISB} and C_{ISF} for the ISB and ISF-decompositions of C_0 respectively.

4.2.5 Heuristic measures

The reduction in Artificial Variable Dependence (AVDr) obtained by each decomposition of the benchmark control sequence was evaluated for all problems. The significance of this value as a heuristic measure of the utility of the decomposition is discussed in the next section, and also in the experiments of Chapter 6. For a decomposition C, $\mathrm{AVDr}(C) = \mathrm{AVD}(C_0) - \mathrm{AVD}(C)$, where C_0 is the benchmark sequence for that problem.

The calculation of AVD and AVDr for a control sequence follows the procedure described in section 3.5.2. For a control sequence C,

$$\mathrm{AVD}(C) \quad = \sum_{T \in C = \langle C_p T C_s \rangle} \mathrm{AVD}(T, C_p)$$

While effective for use with Backtrack, this calculation does not take into account the look-ahead step in Forward-Check. All constraints in Forward-Check are tested in the look-ahead step, and when this forward checked constraint occurs at a depth greater than one level below the current level of search, the size of the context at that constraint will be smaller than the corresponding size under Backtrack. Thus the same control sequence can actually have a smaller AVD in Forward-Check than in Backtrack. The AVD calculations used in the experiments are Backtrack AVDs, and so only approximate (and are an upper bound on) the true Forward-Check AVDs.

4.2.6 Performance and problem hardness

In most of the experiments described below, the cost of using each control sequence with each search algorithm has been measured in the number of tests (constraint-checks) needed to solve that problem. The number of tests has been generally accepted as a more portable measure of cost than run-time, allowing comparisons to be made

between different algorithms on problems executed within different computing environments. Furthermore, reducing constraint checks is a primary focus for the decomposition schemes tested here.

The ultimate aim for most problem-solving performance improvement schemes is, of course, to produce solutions quicker, and one set of experiments was performed to measure run times for the different algorithms (section 4.3.2). To get consistent results, these experiments needed to be run on dedicated computer. Only one problem set was tested due to the difficulty in obtaining sole usage time on a computer in a multi-user environment. The number of constraint checks is used as the measure of performance in all other experiments.

To measure the benefit obtained by using a particular decomposition, the experiments compare the relative performances of the different control sequences. The relative performance of a decomposition strategy is the ratio of the number of tests needed to solve the corresponding sequence to the number of tests needed by the benchmark sequence (Tests(C)/Tests(C_0)). Thus a relative performance of 40% means that that control sequence required only two-fifth the number of tests needed by the benchmark sequence.

Except when evaluating heuristic measures for selecting the best decomposition, the best performing DOI-decomposition in each problem is used to represent the DOI decomposition strategy. This helps define the goal performance that a heuristic selection strategy should try to achieve.

Several graphs in the next section chart the results of the experiments by mapping the relative performance of a decomposition strategy against basic problem difficulty or 'hardness'. A problem's *basic hardness* is measured in the number of nodes searched by the benchmark control sequence using one of the basic search algorithms. We will use *normalized hardness*, the proportion of the full enumeration search tree examined by the problem's benchmark sequence, as a measure of how close a problem is to approaching the maximum possible difficulty for problems in that size class. For a control sequence C and algorithm A, on a problem with N variables each with a domain size d,

$$\begin{aligned} \text{Basic-Hardness}_A(C) &= \text{Nodes}_A(C) \\ \text{Normalized-Hardness}_A(C) &= \text{Nodes}_A(C)/(\sum_{i=1}^{N} d^i) \end{aligned}$$

Thus for problems near 100% hardness, running the benchmark sequence required the specified search algorithm to investigate almost every possible combination of values for its variables. Note that Forward-Check may exceed this maximum number, because its look-ahead mechanism may cause it to look at some nodes twice. Normalized hardness is important when observing decomposition behaviour because the degree of redundant testing in a problem depends more on the fullness of its search tree than on an absolute number of nodes. A fixed number of nodes can belong to a very long thin tree which will have a minimal amount of redundant testing, if any.

Availability of computing resources for running the experiments was the limiting factor in the size of problems used in the experiments. The largest problems tested belonged to the set

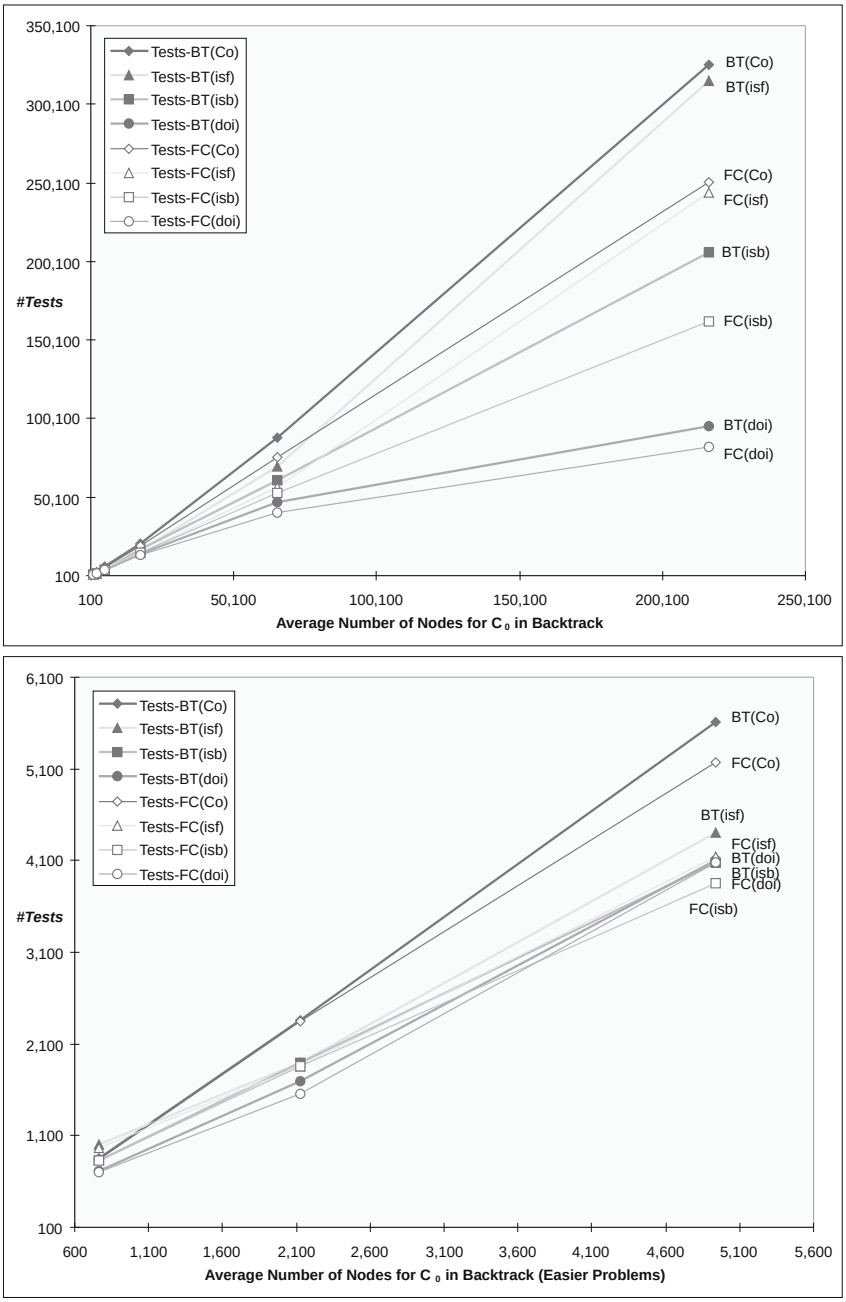
$$N = 9,\ d = 5,\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$$

(e.g. figure 4.10). As has been mentioned above, a range of problem difficulties was generated. The hardest problems in this set required Backtrack to explore over a million nodes and over two million constraint checks. The hardest problem required 2,404,030 nodes (a basic hardness of 98%) and 4,264,717 constraint checks.

4.3 The Experiments

4.3.1 Performance improvement through decomposition

Figure 4.1 compares the performance of all three decomposition strategies with basic search, using Backtrack and Forward-Check to look for all solutions, for one problem-set. The problems have been grouped into six hardness ranges, distributed logarithmically along the hardness spectrum, based upon the number of nodes needed by Backtrack to solve each problem. Except for the three problems averaging about 700



Problem-set: $(N = 6,\ d = 8,\ c = 0.2,\ 6 \leq K \leq 8,\ b = 1.0)$

Figure 4.1: Comparing the performance of decomposition with basic search. The second graph shows more detail on the easier problems.

nodes and grouped at the easiest end of the spectrum, the remaining 57 problems of the problem-set are divided fairly evenly among the rest of the hardness ranges. The DOI decomposition lines in the figure are for the best DOI decomposition for each problem, averaged for each hardness range.

The first and obvious observation to make is that there is a fairly linear relationship between the number of nodes and number of tests for the simple control sequences, with Forward-Check giving better performance than Backtrack. In fact Forward-Check also yields better performance than Backtrack for the decomposed multi-level sequences.

The second fact obvious from the figure is that DOI-decomposition almost always yields the best performing control sequence, especially when used with Forward-Check. What is also interesting is that DOI-decomposition with Backtrack performs much better than basic Forward-Check, and running bottom-up Forward-Check on the DOI-decomposition only gives a small improvement in performance over bottom-up Backtrack.

Between the two IS-decomposition algorithms, ISB can be seen to yield the better performance, and competes well with DOI decomposition in the medium difficulty range. The second graph on easier problems shows that ISF-decomposition actually works well only for some of the easier problems, and its relative performance degrades as the basic problem gets harder.

In comparison with ISB, the ISF decomposition algorithm tends to reduce redundancies in constraints tested at shallower levels in the basic (benchmark) sequence's search tree. So for large independent sets, ISF might even perform better than ISB and DOI for some problems. As problems get harder, most of the redundancies start shifting to constraints occurring at deeper levels, and ISB-decompositions start performing better.

DOI-decomposition focuses entirely on constraints in the lower portion of the search tree. As the number of nodes in these levels rises exponentially with depth, the (best) DOI-decomposition actually improves in relative performance. This can be seen in the leveling off of the DOI performance curve with increasing problem difficulty.

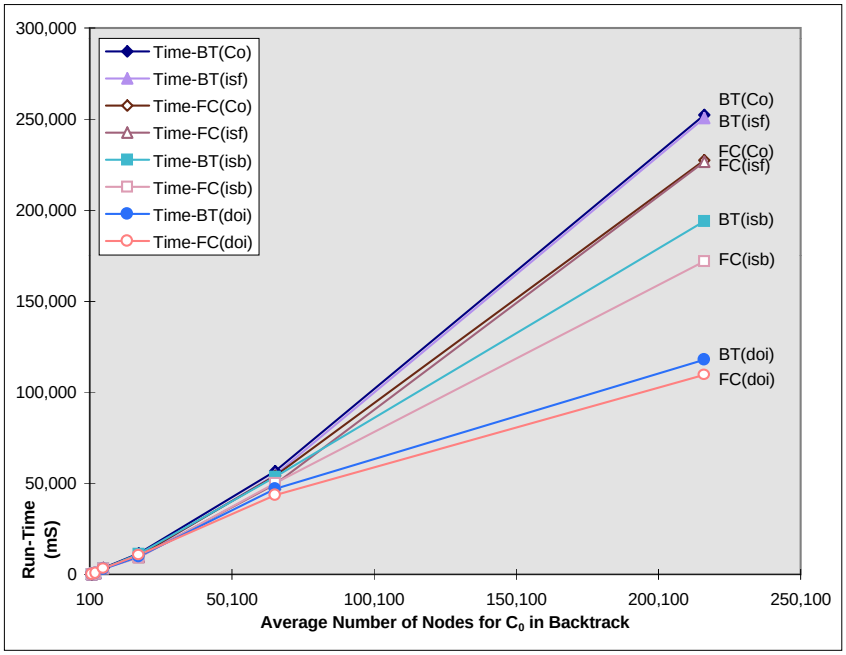
4.3.2 Run-time performance

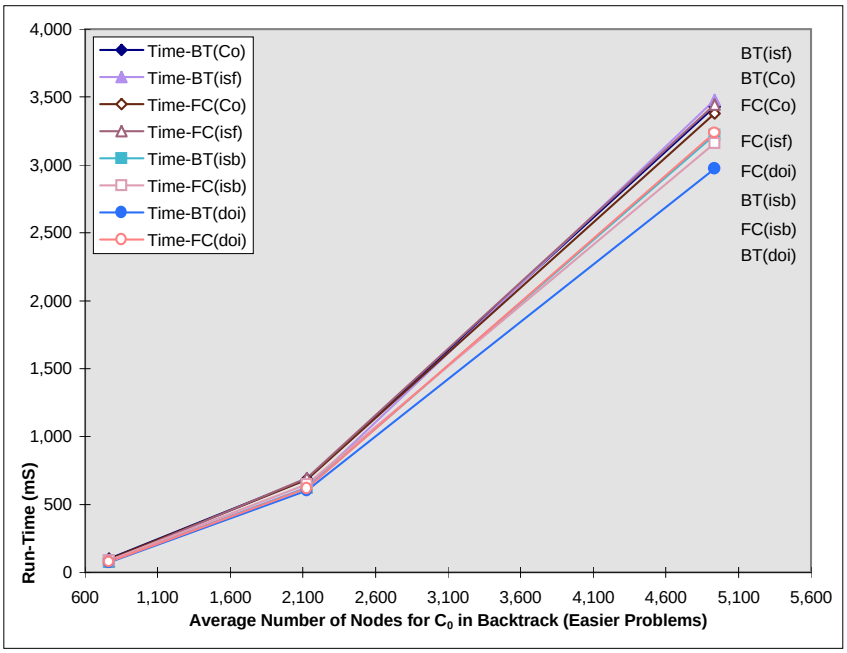
Figure 4.2 shows the run-time performance of the same problems, control sequences and algorithms charted in figure 4.1. It is obvious on comparing the two charts that the relative order in performance of the different search techniques is the same, particularly on the harder problems, whether measuring performance in constraint checks or run time. It can also be observed that the degree of benefit offered by decomposition is smaller when measuring run time (the performance curves for decomposition are steeper when charting run times). The run-time curves for decomposition include the cost of building and generating values from the solution caches.

Figure 4.3 shows the results of an experiment to measure the run-time cost of building and generating from a solution cache. Two sets of experiments were conducted, for domain sizes 5 and 10, respectively. Dummy problems of different number of variables and no constraints were run, and total times measured under Backtrack for all the generate events, building the complete solution cache tree and generating from the solution cache. Since there were no constraints, every value assignment to the variables was a solution. An extra generate event was counted each time the end of a variable's domain was reached; the event detected the end of the domain and performed a reset for the next generate event. (This is why the number of generate events is larger than the number of nodes in the solution tree). The simple linked-node version of the solution cache tree was used (see figure 3.2-(b)).

Figure 4.3-(a) compares the unit cost of a simple generate event to that of generating from the solution cache. Generating from a solution cache, it seems, takes about 40-45% more time than a simple generate. In the implementation tested here, a simple generate event generates the value contained in the next element of an array housing the variable's domain. A solution cache generate, on the other hand, follows a link to the next sibling node and produces the value found there. The procedures used for generating values are shown in figure 3.5.

The cost of building the complete solution cache tree is shown in figure 4.3-(b). Comparing these costs with the total cost of simple generates shows that for domain





Problem-set: $(N = 6,\ d = 8,\ c = 0.2,\ 6 \leq K \leq 8,\ b = 1.0)$

Figure 4.2: Comparing the run-time performance of decomposition with basic search. The second graph shows more detail on the easier problems.

(A): Comparing the cost of using the Solution Cache Tree with cost of simple generate.

Dom Size	Nbr Vars	Nbr Solutions	Nbr of Generates	Time(mS) Total	Time(uS) Unit	Nbr of Recalls	Time(mS) Total	Time(uS) Unit
5	6	15,625	23,436	30	1.28	23,436	50	2.13
5	7	78,125	117,186	150	1.28	117,186	223	1.90
5	8	390,625	585,936	740	1.26	585,936	1,080	1.84
5	9	1,953,125	2,929,686	3,750	1.28	2,929,686	5,600	1.91
10	5	100,000	122,221	160	1.31	122,221	226	1.85
10	6	1,000,000	1,222,221	1,640	1.34	1,222,221	2,340	1.91

(B): The cost of building the Solution Cache Tree.

Dom Size	Nbr Vars	Nbr Solutions	Nbr of Nodes	Time(mS) Total
5	6	15,625	19,530	140
5	7	78,125	97,655	830
5	8	390,625	488,280	4,340
5	9	1,953,125	2,441,405	24,280
10	5	100,000	111,110	1,200
10	6	1,000,000	1,111,110	14,120

Figure 4.3: The cost of building and generating from a linked-list cache.

size 5, the cache build cost is 3 to 4 times greater, and this factor increases to 6 when the domain size is doubled. Almost all of this extra cost is incurred in the tests needed, on adding each new solution to the cache, to determine where to 'hang' the new solution in the tree. These tests look for the occurrence of the largest prefix of the new solution in the existing solution cache. The remaining suffix is then hung underneath the last node of the prefix. Remember from chapter 3 that these tests accounted for $O(s^2)$ steps of the total build cost, and that these tests are largely eliminated in the vector-node version of the solution cache tree. Even in the linked-node version of the cache, the overhead of building the solution cache will be a lot smaller for smaller number of solutions. There is also room for improvement in the actual code implementing the solution cache used in these experiments.

Constraint Name	Nbr Vars	Dom Size	Nbr of Gens	Time(mS) Total	Time(uS) Unit
-	8	5	585,936	720	1.23

Constraint Name	Nbr Vars	Dom Size	Nbr of Gens	Time(mS) Total	Nbr of Tests	Time(mS) Test	Time(mS) Unit	Ratio Test/Gen
c2	8	5	585,936	3,050	390,625	2,330	5.96	4.85
c3	8	5	585,936	15,630	390,625	14,910	38.17	31.06
c4	8	5	585,936	106,790	390,625	106,070	271.54	220.98
c5	8	5	585,936	948,080	390,625	947,360	2,425.24	1,973.66
c6	8	5	585,936	1,491,190	390,625	1,490,470	3,815.60	3,105.14
cx2x1	8	5	585,936	1,030	390,625	310	0.79	0.65
adjhside	8	5	585,936	1,030	390,625	310	0.79	0.65
minarea	8	5	585,936	1,170	390,625	450	1.15	0.94
dontovlap	8	5	585,936	1,240	390,625	520	1.33	1.08

Figure 4.4: Comparing the costs of constraint testing and generating.

The final experiment in this group measures the average cost of testing the constraints of arities 2 through 6 used in the problems of this chapter, and some of the constraints from the floorplanning domain described in a later chapter. The results are shown in figure 4.4, with a comparison against the average time for a normal variable value generate. The experiment was based on dummy problems similar to those used

in the previous experiment.

The constraints *ci* are the arity i table-lookup constraints of tightness 0.2 used in random problems tested in this chapter. Their average cost increases with arity as the size of the table increases. This is an important factor that should be considered when selecting a decomposition, and is ignored by the AVD heuristic for selecting a DOI decomposition.

As a comparison, this experiment also measured the average cost of testing some constraints from the floorplanning domain of Chapter 6. While the table-lookup constraints were much more expensive to test than a generate action, the floorplanning constraints were of comparable (mostly cheaper) cost. As will be seen in Chapter 6, the decomposition techniques were still effective in reducing both run-time and constraint-checks for this problem.

While the benefit of decomposition coupled with Bottom-Up solution will generally be greater in problems where constraint tests are significantly more expensive than generates, other problems can also benefit, though the cache build and generate overhead costs must be considered when selecting a decomposition for those problems.

4.3.3 AVD reduction by decomposition

		Averages		
AVD(C_0)	Count	AVDr(C^0_{DOI})	AVDr(C_{ISB})	AVDr(C_{ISF})
3	5	1.4	1.2	1.0
4	4	1.8	1.7	1.7
5	13	2.2	2.1	1.6
6	19	2.3	2.5	2.1
7	6	2.7	3.8	1.6
8	5	4.0	2.8	2.0
9	4	2.5	4.5	2.8
10.5	4	3.8	4.8	3.0

Problem-set: ($N = 6,\ d = 8,\ c = 0.2,\ 6 \leq K \leq 8,\ b = 1.0$)

Figure 4.5: AVD reduction by decomposition.

The table in figure 4.5 gives the average AVD reduction achieved by the three decomposition strategies with increasing artificial variable dependence in the benchmark control sequence. The AVD reduction achieved by ISF-decomposition increases much more slowly than the AVDr values for ISB and DOI decompositions. This is a result of the ISF procedure's selecting constraints for the independent set starting from the beginning of the benchmark control sequence.

Figure 4.8 shows that higher AVD reductions generally lead to better relative performance for each decomposition. However, even though ISB decomposition yields the highest average AVD reduction of 4.8, DOI decompositions usually have the better relative performance. This is because smaller reductions in artificial dependence yield bigger performance improvements for constraints at deeper levels in the search tree.

4.3.4 Selecting the best DOI-decomposition

The procedure for DOI-decomposition used in these experiments produces a decomposition for a problem by cutting its benchmark sequence C_0 at a point such that the last node in the resulting prefix is a constraint, and the first node in the resulting suffix is a variable generator. There will generally be several such decompositions available for each simple control sequence. The problem then is how to select the decomposition that will give the best performance.

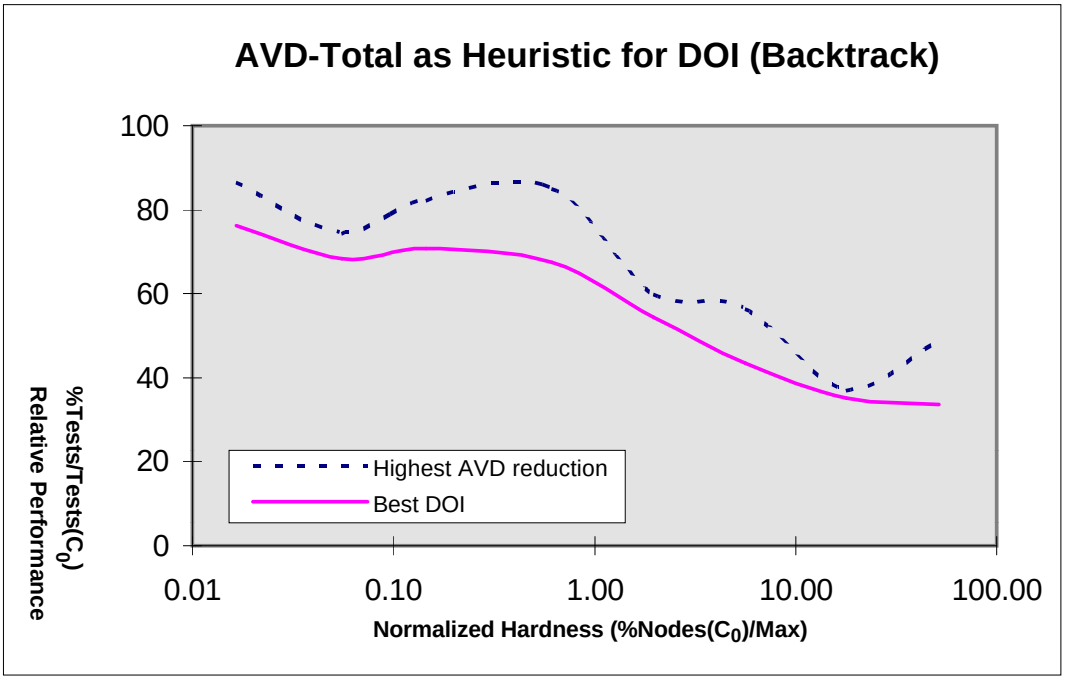
Figure 4.6 compares the relative performance of the best DOI-decomposition with the DOI-decomposition with the highest AVD reduction for problems in the problem-set

$$N = 9,\ d = 5,\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$$

The values have been averaged over the normalized hardness ranges

$$0.01\% - 0.03\%,\ 0.03\% - 0.1\%,\ \ldots,\ 30\% - 100\%$$

The average difference between the performance of the DOI-decomposition with the highest AVD reduction and the best is of 9.8 percentage points, with a median of 4.4 and standard deviation of 12.8. Thus selecting the decomposition giving the largest reduction in AVD over the benchmark sequence gives a fairly close approximation to the performance of the best possible DOI-decomposition, but it is not perfect.



Problem-set: ($N = 9,\ d = 5,\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$)

Figure 4.6: Utility of AVD reduction as a heuristic for selecting a DOI-decomposition.

	Tests(C_{DOI}) in Backtrack (ranked)	
Correlations	*All problems*	*Norm.-Hardness* $\geq 3\%$
Ranked AVDr(C_{DOI})	-0.69	-0.85
Nbr. Prefix Vars in C_{DOI} (ranked)	0.38	-0.55

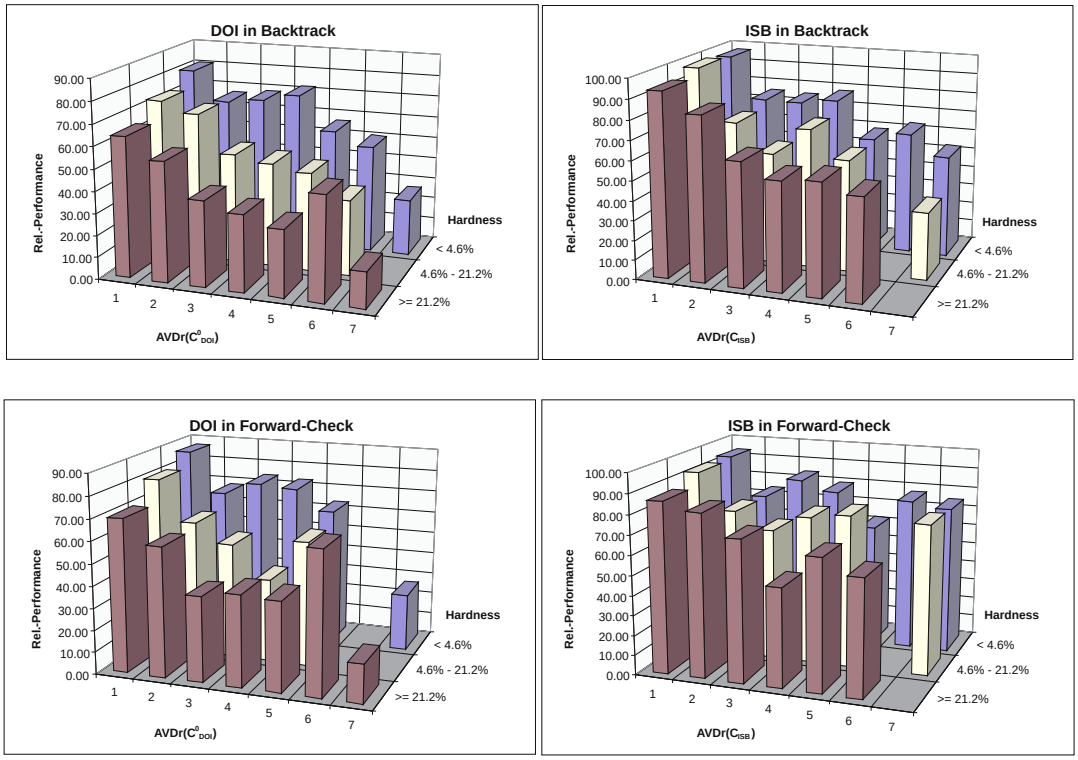
Problem-set: ($N = 9,\ d = 5,\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$)

The table correlates the ranks within each problem of AVD reductions and number of variables in the prefix with the number of tests for DOI-decompositions. Lowest values within each problem are assigned the rank 1, and the rank is incremented by 1 for each higher value.

Figure 4.7: More on heuristics for selecting the best DOI-decomposition of C_0.

Figure 4.7 shows that there is a fairly high correlation, particularly for harder problems, between the amount of AVD reduction obtained by a DOI-decomposition and how close it is to being the best DOI-decomposition of that problem's C_0. It also shows that for the harder problems, DOI-decompositions obtained from deeper cuts in the benchmark sequence tend to perform better. This suggests that a combination of the amount of AVD reduction and depth of cut (number of variables in the prefix) would make a good heuristic for selecting the best DOI-decomposition for a problem. The use of this heuristic is discussed in more detail in Chapter 6.

4.3.5 Problem hardness



Averaged over problem sets: $N = 6,\ d = 5,\ c \in \{0.2, 0.4, 0.6\},\ 9 \leq K \leq 12,\ b = 1.0$

Figure 4.8: Decomposition performance with changes in AVDr and normalized problem hardness.

Figure 4.8 demonstrates that the performance of

DOI-decomposition improves with increases in both AVD reduction and problem hardness, in both Backtrack and Forward-Check. ISB-decomposition does show a trend towards better performance with increases in AVD reduction, but not with increases in problem hardness.

Other experiments described here confirm that DOI-decomposition does indeed show better performance as the basic problem gets harder. They also confirm Theorem 3.3, in that the Backtrack performance of a DOI-decomposition never exceeds that of the benchmark simple sequence the decomposition was derived from. While performance improvement with DOI-decomposition on the easiest problems is slight, experiments showed that the relative performance on the hardest problems was as low as under 20% (i.e. more than 80% better than the benchmark), for both Backtrack and Forward-Check.

ISB-decomposition, on the other hand, offers no benefit on the easy problems, with relative performance falling quickly below 100% as problem difficulty increases, and then leveling out into a bowl-shaped curve. The relative benefit of ISB-decomposition actually starts to decrease for the hardest problems.

As a problem gets more difficult and approaches 100% normalized hardness, its search tree gets fuller. After a certain point in the middle of the hardness region, the search tree starts becoming rapidly bottom-heavy. As the number of nodes in the lower levels of the search increases exponentially, so does the number of redundant tests of artificially dependent constraints in those levels. Most of the number of redundant tests in a hard problem with such constraints (e.g. a binary-CSP like P_1) occur in these lower regions. DOI-decomposition gets its performance improvement by reducing artificial dependencies of constraints in the original sequence's suffix, whereas ISB-decomposition usually benefits constraints dispersed through the length of the sequence. This is why DOI-decomposition performs so well for the really hard problems, and the relative performance of ISB-decomposition shows a trend towards the worse for those problems.

For example, in the control sequence C_1 for the binary CSP P_1, (section 2.3), 312

of the 504 total redundant tests under Backtrack occur at constraint T_{12}, the last constraint in the control sequence. Any decomposition strategy that does not reduce the artificial dependencies of this constraint will not make a big impact on performance.

The problems produced by the random problem generator tended to be very easy for the one-solution search of Backtrack-1. For this reason, very little trend information could be derived from them. In the rest of this chapter, one-solution search experiments will be mentioned only when they show an interesting phenomenon.

4.3.6 Number of variables

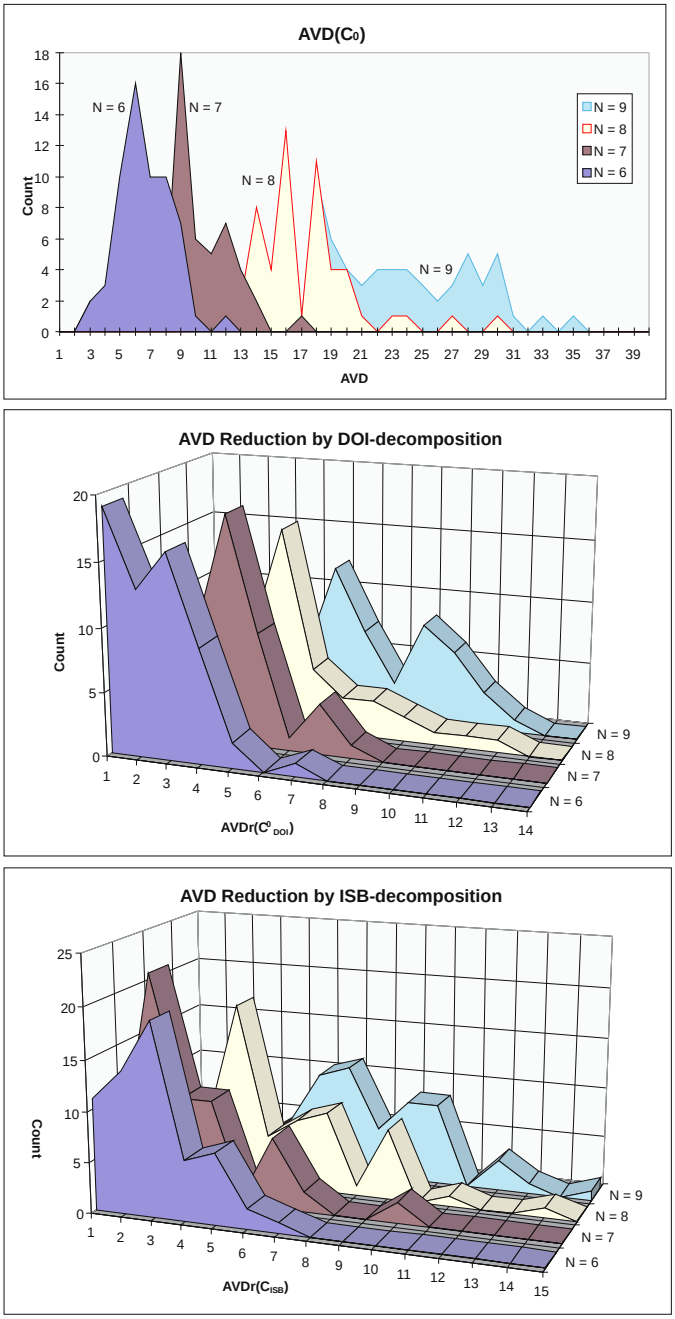
Problems with larger number of variables are likely to have larger degrees of artificial dependence, as each constraint can now dependent artificially on more variables. Figure 4.9 confirms that AVD values increase with increasing number of variables. The figure also shows that the increase in AVD of the benchmark sequence also results in a larger AVD reduction obtained by both DOI and IS decompositions.

Figure 4.10 shows the performance of DOI and ISB decompositions as they change with number of problem variables and problem difficulty, averaged in the following normalized hardness ranges:

$$\ldots,\ 0.1\% - 0.3\%,\ 0.3\% - 1.0\%,\ 1.0\% - 3.0\%,\ \ldots$$

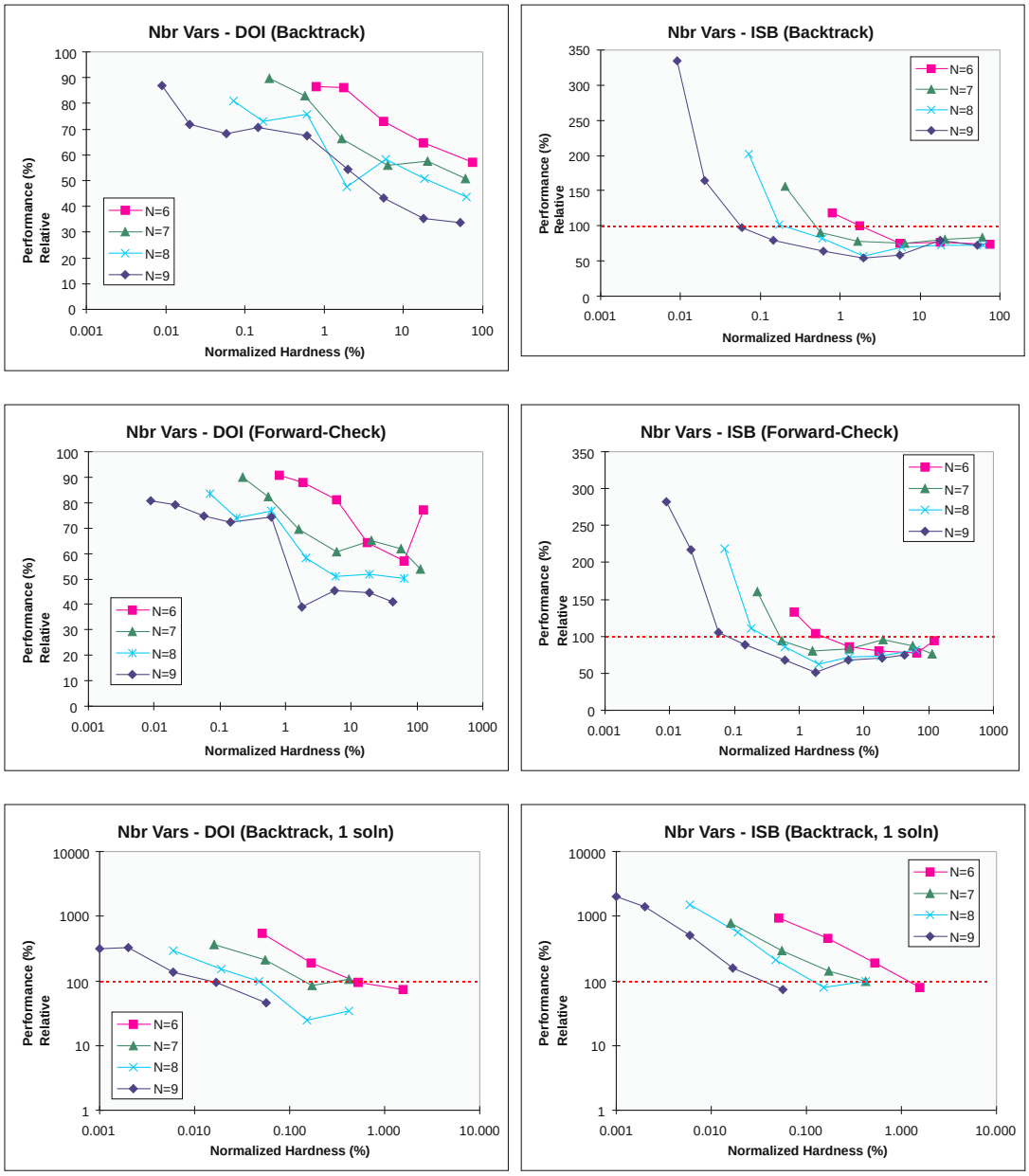
The first thing to note in these performance curves is their general shape, which is as discussed in the previous subsection. Secondly, both decomposition strategies show a general trend towards better performance as problems get harder, for each of the three search algorithms. This is to be expected from the tendency towards higher AVD reductions achieved by these decomposition strategies for problems with larger number of variables.

While each problem-set covered a fairly broad spectrum of difficulties when searching for all solutions, the single-solution search problem tended to be relatively easy, going upto only about 2,000 nodes in the search tree compared to over a million for the hardest problem when looking for all solutions. Both decomposition strategies proved beneficial only for problems requiring more than about 300 nodes in their C_0 search



The Y-axis (Count) shows number of problems.
Problem sets: $N \in \{6, 7, 8, 9\}$, $d = 5,\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$

Figure 4.9: Changes in AVD(C_0) and AVD reductions by decomposition — changing the number of variables.



Problem sets: $N \in \{6, 7, 8, 9\},\ d = 5,\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$

Figure 4.10: The performance of DOI and ISB-decompositions — changing the number of variables.

tree. Problems with larger number of variables needed to be harder to get benefit from decomposition. While the degree of benefit obtainable from decomposition for these problems was quite small, it is still expected that as the one-solution problem gets harder, the degree of benefit will also increase.

4.3.7 Domain size

In general, as increases in domain size lead to increased bushiness of the search tree, the number and proportion of redundant constraint checks should also increase, with a corresponding improvement in decomposition performance. Consider the following simple situation in control sequence C_3:

$$C_3 = \langle X_1\ X_2\ T_1(X_1, X_2)\ X_3\ X_4\ T_2(X_3, X_4)\rangle$$

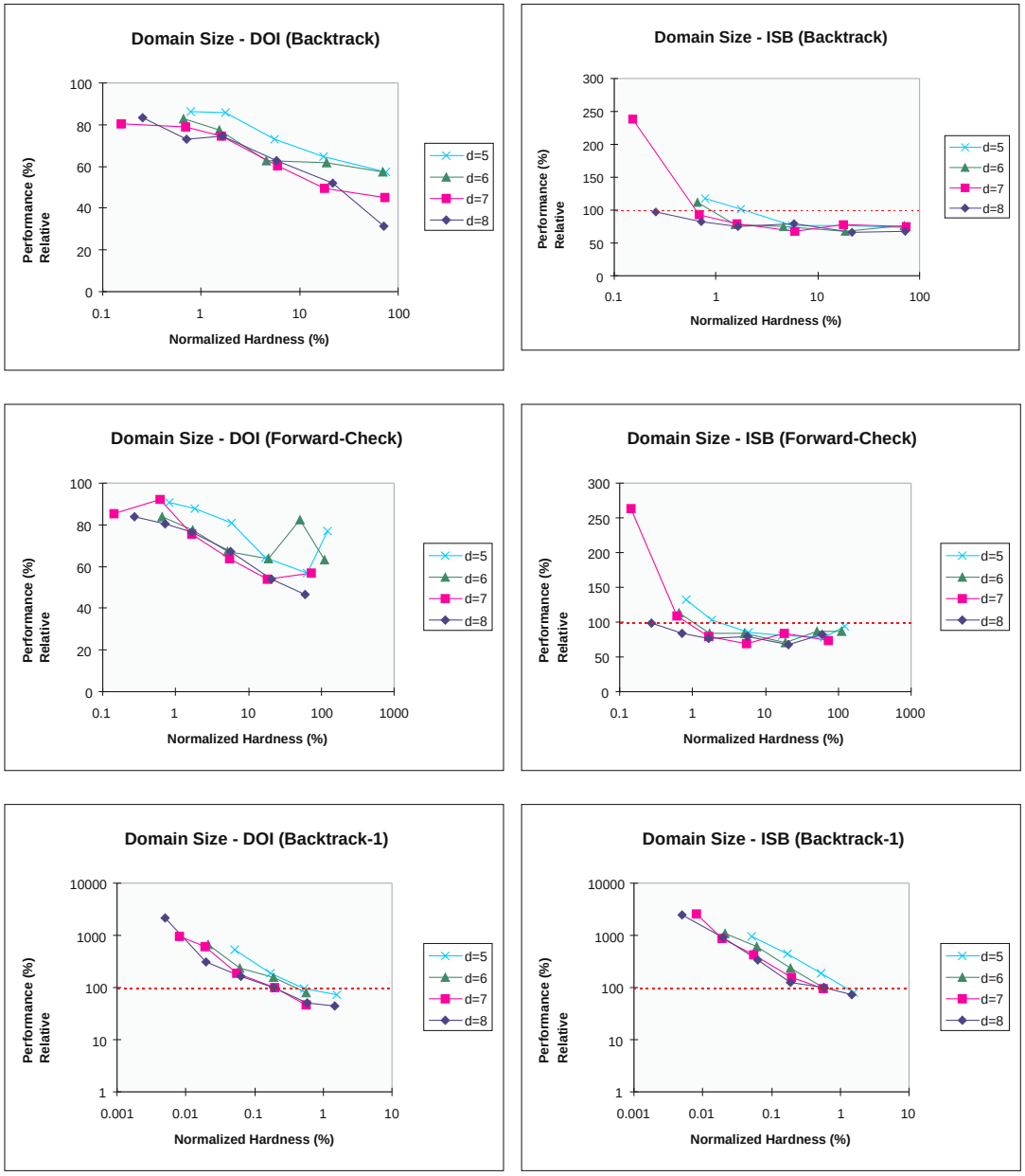
If both constraints have a tightness of 0.2, then a domain size of 5 results in 5 solutions for either constraint. The artificial dependence of T_2 on variables X_1 and X_2 results in $(5-1)\times 25 = 100$ redundant tests, $100/(25+5\times 25) = 66\%$ of the total number of tests needed to solve C_3 using Backtrack. Increasing the domain size to 6 gives each constraint 7 solutions, and constraint T_2 $(7-1)\times(7\times 7) = 294$ redundant tests, $294/(49+7\times 49) = 75\%$ of the total number of constraint checks. While the number of redundant tests has almost tripled, the proportion increased only slightly.

Compare this with a change in the amount of artificial dependence. Let us add another argument variable to T_1, increasing T_2's AVD from 2 to 3.

$$C_4 = \langle X_1\ X_2\ X_4\ T_1(X_1, X_2, X_4)\ X_3\ X_4\ T_2(X_3, X_4)\rangle$$

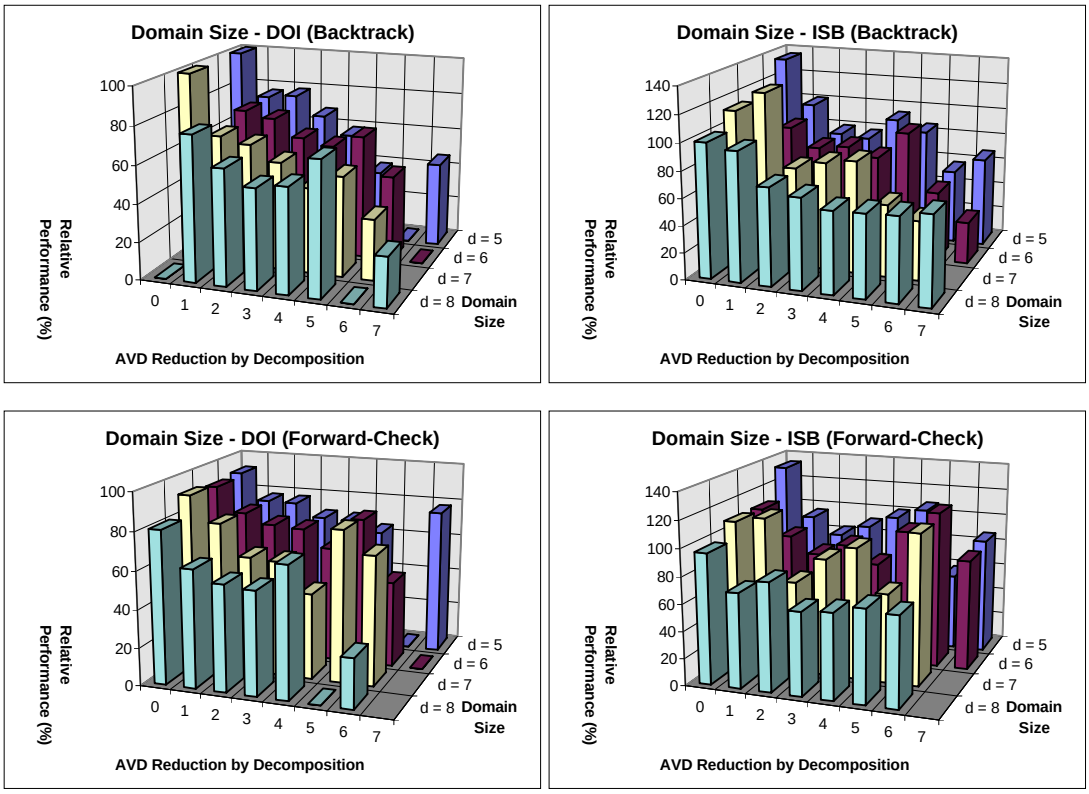
Keeping the domain size at 5, T_1 now has 25 solutions, and T_2 now has 600 redundant constraint tests, 80% of the total number of tests needed to solve C_4. Increases in AVD (e.g. by increasing the number of variables) has a bigger effect on the number and proportion of redundant tests.

Both these examples represent, of course, very simple situations where there are no interactions between constraints. Constraint interactions tend to decrease the combined number of solutions, and so also the number of redundant tests.



Problem sets: $N = 6,\ d \in \{5,6,7,8\},\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$

Figure 4.11: The performance of DOI and ISB-decompositions — changing the domain size.



Problem sets: $N = 6,\ d \in \{5, 6, 7, 8\},\ c = 0.2,\ 9 \leq K \leq 12,\ b = 1.0$

Figure 4.12: Decomposition performance with changes in AVDr and domain size.

The performance graphs in figure 4.11 show that increasing the domain size from 5 to 8 does lead to better relative performance of DOI decompositions, but small increments do not produce significant changes. The overall trend is on a smaller scale when compared to performance improvements from increasing the number of variables (figure 4.10).

The performance of ISB-decomposition shows an even smaller change in relative performance with increases in domain size, with the performance at harder problems not changing significantly.

Figure 4.12 shows that increases in AVD reduction yield much bigger improvements in performance than increasing the domain size, for problems in the same hardness range.

4.3.8 Number of constraints

Increasing the number of constraints while keeping the number of variables the same should result in control sequences with higher AVD. However this need not necessarily translate into higher redundant testing, as due to constraint interaction, the same number of variables in each control sequence prefix will now have fewer solutions.

The performance charts in figure 4.13 show that problems with 8-10 constraints are generally the least amenable to improvement by decomposition. After 6-8 constraints, the inter-constraint interaction becomes strong, making the search tree much narrower. Problems with 6-8 constraints show better performance improvements in DOI-decomposition than those with 4-6 constraints. ISB-decomposition performance is less predictable.

4.3.9 Constraint tightness

In general, the effect of loosening the constraint tightness on redundant tests, and on performance improvement with decomposition, should be similar to that of increasing the domain size. The performance graphs of figure 4.14 however do not show any significant correlation between constraint tightness and relative performance of decompositions. Part of this is probably due to the use of independent problem-sets for the

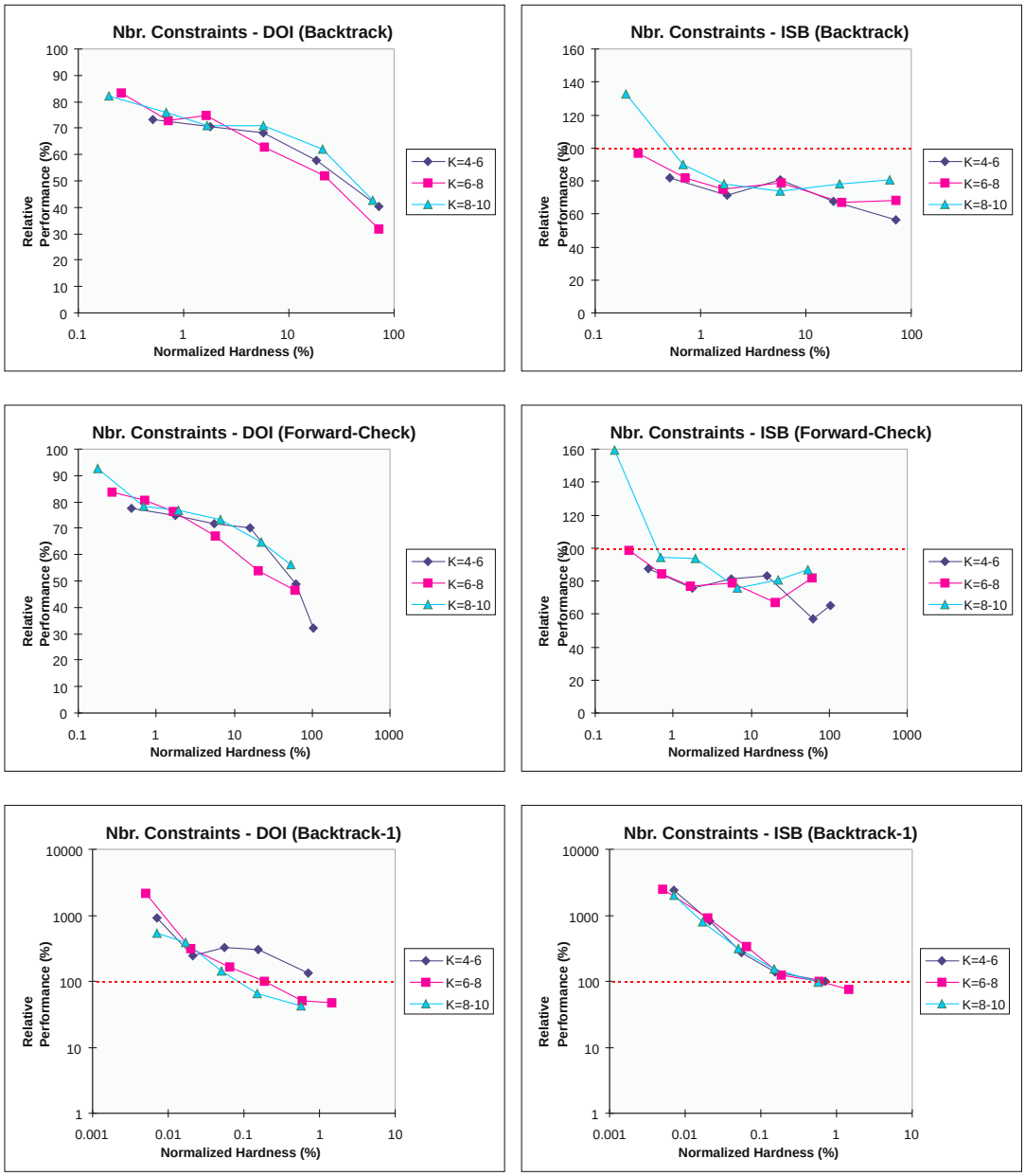


Figure 4.13: The performance of DOI and ISB-decompositions — changing the number of constraints.

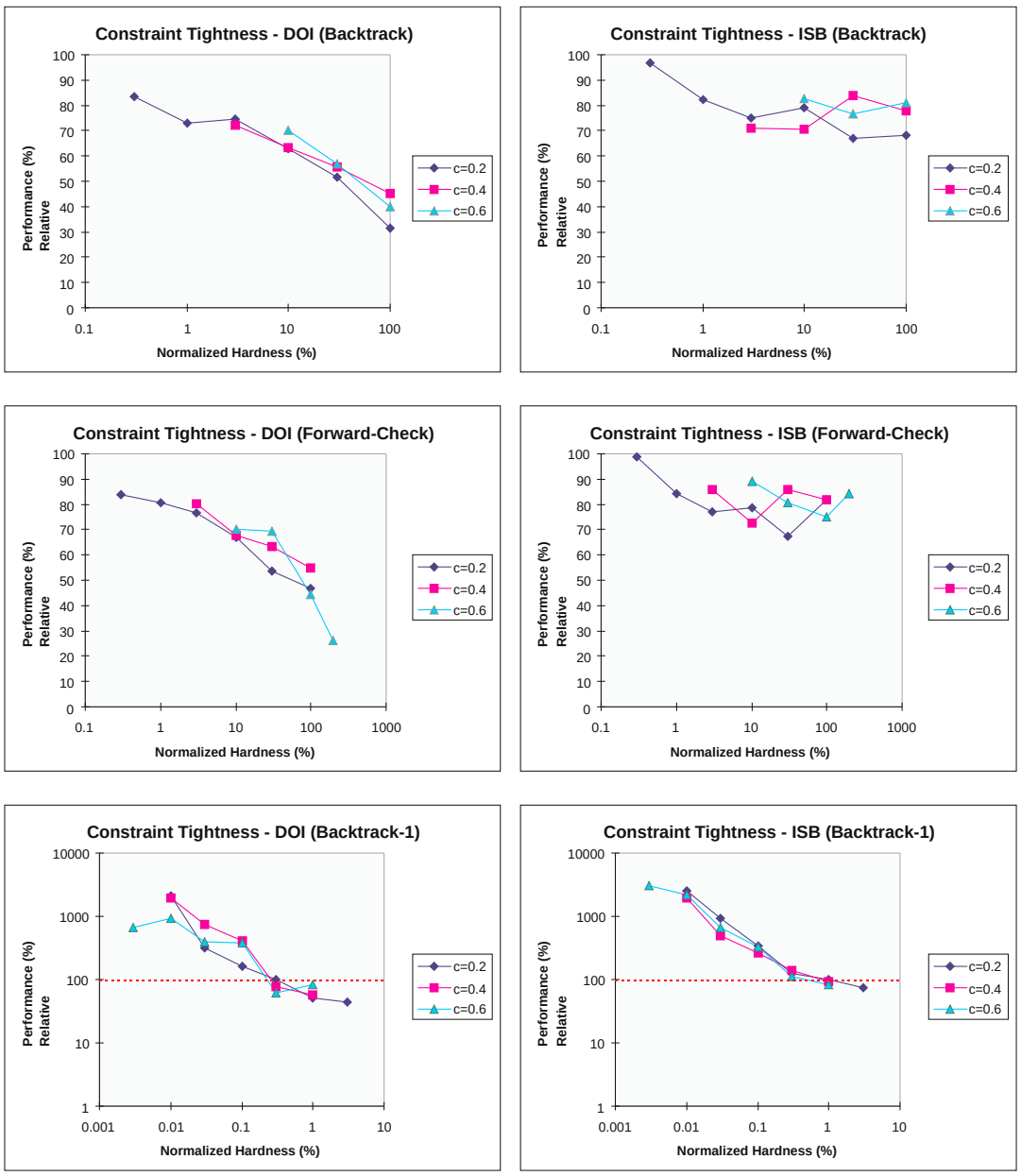


Figure 4.14: The performance of DOI and ISB-decompositions — changing the constraint tightness.

three tightness values. If the constraint topology had been kept the same in the three problem sets, the results might have been different. This is an area that needs further investigation.

4.4 Summary

The main conclusion of this empirical study is that DOI decomposition offers the most potential for performance improvement in CSPs with constraints of varying and arbitrary arity ('nary' CSPs). In harder problems, the best DOI decomposition of a simple control sequence tends to be the one offering a higher AVD reduction and also the one dividing the original sequence at a deeper point. A bottom-up Backtrack solution of DOI-decompositions was also seen to perform much better than simple Forward-Check.

Generally, decompositions were seen to offer better relative performance on problems of greater basic difficulty, and when they yielded greater reductions in artificial variable dependence when compared to the original simple control sequence the decomposition was derived from.

Chapter 5

Decomposing Global Constraints

So far, we have investigated techniques for decomposing a CSP by partitioning its set of constraints, and then employing a bottom-up procedure to solve the resulting hierarchical system. This solution procedure helps reduce redundant constraint checks. However, like other search algorithms, even these techniques fail to take advantage of global constraints to reduce search effort.

Since global constraints can be evaluated only when all variables have been instantiated, search algorithms cannot use them to prune portions of the search space without complete enumeration. In this chapter we directly address the issue of making global constraints more useful to search algorithms.

The techniques described here produce a hierarchical system by decomposing the intentional representation of a global constraint. This decomposition is used to define an abstraction on the original search space. In the resulting hierarchical system the decomposed global constraint is replaced with a set of smaller arity constraints, which search procedures can use for early pruning. This procedure of transforming a CSP into a hierarchical CSP system is an outcome of previous research on search space abstraction, described in [45, 48, 46, 68].

For completeness, this chapter begins by reviewing search space abstraction. Then follows a description of the procedure for constructing a hierarchical CSP system by decomposing a global constraint. Following that is a discussion of alternative solution procedures for the hierarchical CSP, and their impact on the selection of an appropriate abstraction function for improving the eventual problem solving performance. The next chapter demonstrates the use of global constraint decomposition on some application domains, and its result in significantly reducing problem solving effort.

5.1 Search Space Abstraction

As has already been discussed in Chapter 2, constraint satisfaction problems can be difficult to solve. In the worst case they are NP-hard, and so can require an exponential search effort. In particular, the complexity of a CSP depends upon the arity of its constraints, with global constraints often being the most expensive.

Hierarchical problem solvers cope with large and complex search spaces by developing a solution in stages. First an abstract solution is developed in a smaller and simpler version of the domain, which is then used to focus attention on a narrow region of the complex search space; the net effect is a more efficient problem solver. In this section we will formalize the notion of search space abstraction and abstraction levels as applied to constraint satisfaction problems.

5.1.1 Defining an abstraction level

The notion of Abstraction Levels was first introduced in the AI community in GPS [54] and ABSTRIPS [60]. An *abstraction level* is a complete and self-contained simpler version of the original problem, with an abstracted search space, and abstracted constraints. Each abstract solution refines into a set of candidates, called its *refinement*, in the *Lower* or *Base Level* search space. To solve a problem, a hierarchical problem solver first solves an abstracted problem in the abstract level's smaller search space. It then searches for solutions to the original problem in the refinement of the abstract solutions. The idea is to mark large portions of the base level search space as not worth searching while working in the smaller abstract search space. The hierarchical search will be more efficient than non-hierarchical search when the combined cost of searching the abstract search space and the refinement space is smaller than that of searching the original space directly.

We now formalize this notion of an abstract level for CSPs. Consider the general CSP system as introduced in Chapter 2:

$$P = \text{Find} \quad \{y \in \text{Domain}(Y) : R(y)\}$$

Here R is the problem constraint, and $\text{Domain}(Y)$ the search space. Let P_a be another

CSP with constraint Q and variables Z,

$$P_a = \text{Find} \quad \{z \in \text{Domain}(Z) : Q(z)\}$$

and related to P through the mapping

$$\forall y \in \text{Domain}(Y), \exists z \in \text{Domain}(Z) \quad f(y) = z$$

We define the mapping f as the *abstraction mapping* between the two search spaces.

The two CSPs combine to form a *hierarchical system* P_H on which a hierarchical problem solver, as described above, can operate.

$$\begin{aligned} P_H &= \langle P_a, P' \rangle \\ P_a &= \text{Find} \quad \{z \in \text{Domain}(Z) : Q(Z)\} \\ P' &= \text{Find} \quad \{y \in \text{Domain}(Y) : R(y) \wedge F(y)\} \\ \text{where} \qquad F(y) &\equiv f(y) \in \text{Solnset}(P_a) \end{aligned}$$

The CSP subsystem P' is derived from P, by adding the constraint that the solutions to P' also map to some solution of P_a through the abstraction mapping f. In such a hierarchical system we will refer to P' as the *base level* and P_a as the *abstract level*. The constraint F is the *refinement constraint*, and denotes the steps in the hierarchical solver where, after solving the abstract problem Q, it searches in that portion of the base space that lies within the abstract solutions' refinement. Note that this is still a declarative representation, and nothing is indicated of how such a system might actually be solved. The refinement constraint is only a constraint, and a solver may take advantage of it as it pleases.

Notice that the hierarchical system actually defines an expanded CSP system:

$$\begin{aligned} P_h &= P' \bowtie P_a \\ &= \text{Find} \quad \{\langle z, y \rangle \in \text{Domain}(Z) \times \text{Domain}(Y) : Q(z) \wedge R(y) \wedge F(y, z)\} \\ \text{where} \qquad F(y, z) &\equiv (f(Y) = Z) \end{aligned}$$

We have been a little free with our use of the refinement constraint symbol F; its specific usage will always be obvious from the context. We shall continue to use this

representation of the hierarchical system as, by combining the abstract and base levels into one CSP, it actually gives us more flexibility in choice of search procedures. We shall also continue, even in this expanded representation, to refer to the new CSP as a hierarchical CSP, and its variables and constraints as belonging to the abstract or base levels.

Each solution to P_h, or P_H, will be a tuple $\langle z, y \rangle$. If the hierarchical system is constructed as described above, each such solution will also contain a solution y to P as a subtuple. This is true because the hierarchical system contains in the base level the original CSP P as a component. The only change made to P is the addition of the refinement constraints, and adding constraints without changing the variables and their domains can only reduce the number of solutions, not change them. Of course, so far, there is no guarantee that the hierarchical system will have any solutions at all that can satisfy all of the abstract, base level and refinement constraints.

For the hierarchical system to be effective, a (correct and complete) hierarchical solver should be able to find all the solutions to P (and no more). This means that every solution to R in Domain(Y) must map through the abstraction mapping to some solution to Q in Domain(Z). Formally,

$$\begin{aligned} \text{Solnset}(R, Y) &\subseteq f^{-1}(\text{Solnset}(Q, Z)) \\ \text{or} \qquad R(y) &\rightarrow Q(f(y)) \end{aligned}$$

We shall refer to this as the *Hierarchical Completeness* condition, and refer to the hierarchical systems P_H and P_h as hierarchically complete with respect to the CSP P.

Given a CSP P, and another (abstract) CSP P_a related to P through an abstraction mapping f, a hierarchical system can be constructed by first deriving the base level P' from P by adding refinement constraints. The refinement constraints represent the inverse of the abstraction mapping. As shown above, the hierarchical CSP system can be an ordered two-level system of distinct CSPs, or a single extended CSP. In either case, if the system is constructed as described above is hierarchically complete, each solution to the hierarchical system will contain one of the solutions to the original CSP P, and every solution to P will be contained in some solution to the hierarchical system. If the

abstraction mapping is functional, there will be a one-to-one correspondence between the solutions to the hierarchical CSP and the original CSP.

Of course every abstraction function will not yield a hierarchical system that is easier to solve than the original problem. We will look at selecting an abstraction function for performance improvement later in this chapter.

5.1.2 An example

Consider the following CSP P_2:

$$\begin{aligned} P_2 \quad = \quad & \text{Find} \quad \{\langle X_1 X_2 X_3 X_4 \rangle \in \{1 \ldots 10\}^4 : \\ & \qquad ((X_1 + X_2) \times (X_3 + X_4) = c_1) \\ & \qquad \wedge ((X_1 + X_2) > c_2)\} \\ \text{where} \qquad & c_1, c_2 \text{ are constants.} \end{aligned}$$

This example has two constraints: $((X_1 + X_2) \times (X_3 + X_4) = c_1)$ is a completely global constraint, and $((X_1 + X_2) > c_2)$ is a binary constraint.

We define an abstract level P_{2a},

$$\begin{aligned} P_{2a} \quad = \quad & \text{Find} \quad \{\langle W_1 W_2 \rangle \in \{1 \ldots 20\}^2 : (W_1 \times W_2 = c_1)\} \\ \text{where} \qquad & Q \equiv (W_1 \times W_2 = c_1) \\ & Z = \langle W_1 W_2 \rangle \\ & \text{Domain}(W_1) = \text{Domain}(W_2) = \{1 \ldots 20\} \end{aligned}$$

and the abstraction mapping between the base level and P_{2a}:

$$\begin{aligned} f \quad &= \quad \langle f1, f2 \rangle \\ f_1(Y) \quad &= \quad X_1 + X_2 \quad = \quad W_1 \\ f_2(Y) \quad &= \quad X_3 + X_4 \quad = \quad W_2 \end{aligned}$$

Notice that f is actually a composite function, consisting of the two functional terms f_1 and f_2, which correspond to the two abstract variables W_1 and W_2. We will call f the *abstraction function*, and f_i the *component abstraction functions*.

To construct a hierarchical system we first derive the base level by adding to P_2 the following refinement constraints derived from the abstraction function:

$$\begin{aligned} X_1 + X_2 &= W_1 \\ X_3 + X_4 &= W_2 \end{aligned}$$

This then yields the following expanded hierarchical CSP:

$$\begin{aligned} \text{Find} \quad & \{\langle W_1 W_2 X_1 X_2 X_3 X_4 \rangle \in \{1 \ldots 20\}^2 \times \{1 \ldots 10\}^4 : \\ & (W_1 \times W_2 = c_1) \\ & \wedge ((X_1 + X_2) \times (X_3 + X_4) = c_1) \wedge (X_1 + X_2 > c_2) \\ & \wedge (X_1 + X_2 = W_1) \wedge (X_3 + X_4 = W_2)\} \end{aligned}$$

An inspection of this hierarchical CSP reveals that the original global constraint of P_2 is redundant (in the third line above), and can be removed. The final hierarchical CSP P_{2h} is shown below.

$$\begin{aligned} P_{2h} = \text{Find} \quad & \{\langle W_1 W_2 X_1 X_2 X_3 X_4 \rangle \in \{1 \ldots 20\}^2 \times \{1 \ldots 10\}^4 : \\ & (W_1 \times W_2 = c_1) \\ & \wedge (X_1 + X_2 > c_2) \\ & \wedge (X_1 + X_2 = W_1) \wedge (X_3 + X_4 = W_2)\} \end{aligned}$$

It can be easily verified that this system is hierarchically complete.

5.2 Building Hierarchical CSPs with Global Constraint Decomposition

The hierarchical completeness condition can be used to define a space of possible abstractions. Decomposing a global constraint by syntactic decomposition of its intentional expression lies within this search space. In this section we look at how a hierarchical system may be constructed that decomposes global constraints into a conjunction of constraints of smaller arity.

5.2.1 HiT: General weak method for exploring abstractions

The hierarchical completeness condition described above can be used to construct an abstract level from a CSP. The procedure, which we will call *HiT* for *Hierarchical Transformation*, is as follows. The first step is to find a constraint $t(Y)$ such that

$$R(Y) \rightarrow t(Y)$$

In the second step, t is decomposed into the abstraction mapping f and the abstract problem Q,

$$t = Q \circ f, \qquad \text{or} \qquad t(Y) = Q(f(Y)).$$

This is easier if we restrict f to be a functional mapping, or more appropriately, an *abstraction function*. This means that for each base level element $y \in \text{Domain}(Y)$, there is exactly one abstract element $z \in \text{Domain}(Z)$ which is its unique abstraction, such that $f(y) = z$.

Having obtained the abstraction function and abstract problem, in step three the abstract level CSP is now constructed. The abstract level search space is defined as the range of the abstraction function f. The abstract function itself might actually be a tuple of component functions. The number of variables in the abstract level (size of the tuple Z) will be the same as the number of component abstraction functions in f (which is at least one).

$$\begin{aligned} P_a &= \{z \in \text{Domain}(Z) : Q(z)\}, \qquad \text{where} \\ |Z| &= |f| \\ Z &= \langle W_1, \ldots, W_m \rangle \\ f &= \langle f_1, \ldots, f_m \rangle \\ \text{Domain}(Z) &= \text{Range}(f) = \text{Range}(f_1) \times \cdots \times \text{Range}(f_m) \end{aligned}$$

In step four, we construct the hierarchical CSP, by combining the original problem P, the abstracted problem P_a, and refinement constraints obtained by inverting the abstraction function.

$$P_h = \{\langle z, y \rangle \in \text{Domain}(Z) \times \text{Domain}(Y) :$$

$$Q(z) \wedge F(y,z) \wedge R(y)\}$$

$$F(y,z) \equiv f(y) = z$$

The final step, five, is to examine other constraints in the original problem and, using the substitution W_i for $f_i(Y)$, attempt to move them to the abstract level. The reason behind this step is that hierarchical solvers usually search in the abstract level before generating base level variables. Making constraints testable on abstract variables allows them to be tested earlier in the search tree, increasing their pruning power.

Theorem 5.1 *The hierarchical CSP P_h constructed by the HiT procedure is equivalent to the original CSP P, i.e.,*

1. $\forall \langle z, y \rangle \in Domain(Z) \times Domain(Y),\ \langle z, y \rangle \in Solnset(P_h) \rightarrow y \in Solnset(P)$, *and,*
2. $\forall y \in Domain(Y),\ y \in Solnset(P) \rightarrow \exists z \in Domain(Z), (f(y) = z) \wedge (\langle z, y \rangle \in Solnset(P_h))$

The theorem states that there is a one-to-one correspondence between the solutions to the CSP P and the constructed hierarchical CSP P_h, each solution to the former contained in exactly one solution to the latter.

The first part of the theorem proves the hierarchical system P_h correct, with respect to the problem P. Consider any tuple $\langle z, y \rangle$ that solves P_h, as defined in the left hand side of statement (i) of the theorem. We know that $y \in$ Domain(Y), and from the definition of P_h, we know that $R(y)$ is true. Thus y solves P.

The second part of the theorem proves that P_h is hierarchically complete w.r.t. P. We know from the definition of the abstraction function f that for every solution y to P, there is a unique value z such that $f(y) = z$. From the definition of P_h, we also know that this z is a member of the abstract search space, and satisfies Q. Thus for each y that solves P, there is a unique $z \in$ Domain(Z) such that $R(y) \wedge F(y,z) \wedge Q(z)$ is true. Therefore, $\langle z, y \rangle$ solves P_h.

□.

The procedure for constructing a hierarchical system described above also defines a space of possible abstractions and hierarchical systems. The constraint t used as the

basis for abstraction, and its decomposition into the abstraction function and abstract problem, may be derived from the definition of the original CSP P using rules of logic and mathematics, and any domain knowledge that may be available about the given problem.

5.2.2 Decomposing global constraints

According to the model of constraint satisfaction problems introduced in chapter 2, the problem constraint R in the CSP P may be expressed as the conjunction of K individual constraints $T_1, \ldots, T_K$. We also know that $T_1 \wedge \cdots T_i \cdots \wedge T_K \rightarrow T_i$. Thus any of these individual constraints can be used as the constraint t, the basis of abstraction in step one of the procedure HiT for constructing a hierarchical system described above. This allows us to select a global constraint from the problem and decompose it into the abstraction function f and the abstract constraint Q. Using the construction procedure defined above will produce a hierarchical CSP guaranteed to be equivalent to the original CSP in its solutions.

In the hierarchical system P_h constructed in this manner, the conjunction of the refinement constraints and the abstract constraint will be logically equivalent to the constraint t that was used as the basis for abstraction. Therefore, the constraint t may be safely removed from the final hierarchical system.

An example of this procedure is the derivation of the system P_{2h} from the example CSP P_2 defined in the previous section. The derivation is discussed in more detail below.

5.2.3 An example

We now apply the procedure HiT for constructing a hierarchical CSP to our example problem P_2. In step one, we have to select a constraint t for decomposition. We will select the global constraint $(X_1 + X_2) \times (X_3 + X_4) = c_1$. In step two of our procedure, we decompose this into $f_1(X_1, X_2) + f_2(X_3, X_4) = c_1$. This yields a composite abstraction function with two component abstraction functions f_1 and f_2. The results of the first

two steps are summarized below.

$$\begin{aligned} t: &\quad ((X_1 + X_2) \times (X_3 + X_4) = c_1) \\ Q: &\quad (W_1 \times W_2 = c_1) \\ f: &\quad \langle f_1, f_2 \rangle \qquad \text{where} \\ f_1(Y) &= X_1 + X_2 = W_1 \\ f_2(Y) &= X_3 + X_4 = W_2 \end{aligned}$$

In step three, we define the abstract level problem P_{2a}. The abstract search space is defined as the range of the abstraction function, which in this case will have two variables W_1 and W_2. The abstract constraint Q obtained from the decomposition in step two is added to the abstract problem.

$$P_{2a} = \{\langle W_1 W_2 \rangle \in \text{Range}(f_1) \times \text{Range}(f_2) : W_1 \times W_2 = c_1\}$$

In this case, we will rely upon the presence of some domain knowledge to inform us about the relevant ranges of the two component abstraction functions.

$$\begin{aligned} \text{Range}(f_1) = \text{Range}(f_2) &= \{1 \ldots 20\} \\ P_{2a} &= \{\langle W_1 W_2 \rangle \in \{1 \ldots 20\}^2 : W_1 \times W_2 = c_1\} \end{aligned}$$

Step four is to construct the hierarchical CSP. For this, we now define the refinement constraints by simply inverting each component abstraction function. These are added to P_2 and the original global constraint removed to yield our base level P_2'.

$$\begin{aligned} P_2' = \{Y \in \{1 \ldots 10\}^4 : &\ (X_1 + X_2 > c_2) \\ &\wedge F_1(X_1, X_2, W_1) \wedge F_2(X_3, X_4, W_2)\} \end{aligned}$$

where the two refinement constraints F_1 and F_2 are defined as

$$\begin{aligned} F_1(X_1, X_2, W_1) &\equiv (X_1 + X_2 = W_1) \\ F_2(X_3, X_4, W_2) &\equiv (X_3 + X_4 = W_2) \end{aligned}$$

Combining the abstract and base level CSPs results in the extended hierarchical CSP P_{2h}:

$$P_{2h} = \{\langle W_1 W_2 X_1 X_2 X_3 X_4 \rangle \in \{1 \ldots 20\}^2 \times \{1 \ldots 10\}^4 :$$

$$(W_1 \times W_2 = c_1)$$
$$\wedge\, (X_1 + X_2 > c_2)$$
$$\wedge\, F_1(X_1, X_2, W_1) \wedge F_2(X_3, X_4, W_2)\}$$

This system can be further improved by noting that the abstract variable W_1 can be substituted into the binary constraint $(X_1 + X_2 > c_2)$ for the expression $X_1 + X_2$. This is possible from the definition of the abstract component function $f_1(Y) = X_1 + X_2 = W_1$. This allows us to move this constraint completely into the abstract level. This completes the final step in applying our procedure HiT to construct a hierarchical CSP by decomposing a global constraint from the original CSP.

$$\begin{aligned} P_{2h} \quad = \quad & \{\langle W_1 W_2 X_1 X_2 X_3 X_4\rangle \in \{1 \ldots 20\}^2 \times \{1 \ldots 10\}^4 : \\ & (W_1 \times W_2 = c_1) \wedge (W_1 > c_2) \\ & \wedge\, F_1(X_1, X_2, W_1) \wedge F_2(X_3, X_4, W_2)\} \end{aligned}$$

The original global constraint of P_2 is of arity 4. In the hierarchical system P_{2h} constructed from the above abstraction function, the global constraint is removed, and replaced with two additional variables, the abstract constraint Q of arity 2, and the two refinement constraints of arity 3. This reduction in arity, as we will see later, is often enough to speed up the solution of the problem. If we look at the sizes of the domains of the constraints, the possibilities of solution performance improvement become more obvious:

$$\begin{aligned} |\text{Domain}(P)| &= \text{Dom}(Y) = 10^4 = 10{,}000 \\ |\text{Domain}(Q)| &= \text{Dom}(Z) = 20^2 = 400 \\ |\text{Domain}(F_1)| = |\text{Domain}(F_2)| &= 20 \times 10 \times 10 = 2{,}000 \end{aligned}$$

Thus both in terms of the number of variables, and especially in terms of the size of its domain, the original constraint t has been decomposed and *localized* through the use of an abstraction level.

An important factor in this localization or decomposition of the global constraint is the composite nature of the abstraction function. There will be as many refinement constraints (F_i) as number of components (f_i) in the abstraction function f. The arity of the abstract constraint Q, which is also the number of abstract variables W_i, will also be the same as the number of components in the abstract function. Furthermore, each refinement constraint F_i must refer to one abstract variable W_i, and some base level variables.

In P_{2h}, the base level variables are divided between the two component abstraction functions. As there are two component abstraction functions, there are also two abstract variables, and the arity of the abstract constraint is also two. This also produces two refinement constraints F_1 and F_2 which, in this case, are both of arity three.

A larger number of abstract variables would result in a smaller arity of refinement constraints, and a smaller number of abstract variables would result in a smaller arity of the abstract constraint. A balance is thus needed for achieving optimal performance. The actual selection of an optimal abstraction function is very difficult and usually impossible, as it requires complete knowledge of interaction between existing and new constraints. In the next section we discuss the solution of the hierarchical system, and some of the factors that affect its performance, resulting in guidelines for selecting a good abstraction function.

5.3 Solving the Hierarchical System

So far in this chapter, we have discussed how a CSP with a global constraint can be used as the basis for defining an alternative but equivalent CSP which replaces the global constraint with a series of smaller arity constraints. We now discuss some solution strategies that take advantage of the new CSP's special hierarchical structure. We will look at four different hierarchical solution strategies, HSS-1 through HSS-4, two of which employ simple search, and the other two take advantage of the bottom-up search procedure developed earlier in this thesis. They are presented below as strategies for building a hierarchical solver rather than well defined algorithms, in roughly

increasing order of complexity (of the procedure, not problem solving effort) as applied to the example CSP P_3. Unless specifically mentioned otherwise, in each case we will assume that all solutions to the problem need to be found. The performance of the four approaches is summarized in figure 5.1.

5.3.1 The example CSP P_3

The problem P_3 is simply P_2 with only the global constraint:

$$\begin{aligned} P_3 &= \text{Find} \quad \{Y \in \{1 \ldots 10\}^4 : \\ &\qquad ((X_1 + X_2) \times (X_3 + X_4) = c_1)\} \\ Y &= \langle X_1 X_2 X_3 X_4 \rangle \\ \text{where} &\quad c_1 \text{ is a constant.} \end{aligned}$$

Valid values for the constant c_1 in this parameterized CSP are in the range $\{4, \ldots, 400\}$. We will use this as our running example for its simplicity.

To find all solutions, Backtrack applied to any control sequence, and for any value of c_1, will require 10,000 constraint checks. $S_{3.0}$ shown below is an example of such a sequence

$$S_{3.0} = \langle X_1 \ X_2 \ X_3 \ X_4 \ R(Y) \rangle$$

where $R(Y)$ is the single global constraint in P_3. Forward-Check requires the same number of constraint checks, but with some additional memory overhead. Figure 5.1 shows the result of running Backtrack on P_3 for four different values of the constant c_1.

5.3.2 HSS-1: Simple search on the expanded hierarchical CSP

A hierarchical version of P_3 can be constructed using the procedure described in the previous section. Using the same abstraction function as was used for P_2,

$$\begin{aligned} f &= \langle f_1, f_2 \rangle \\ f_1(Y) &= X_1 + X_2 \\ f_2(Y) &= X_3 + X_4 \end{aligned}$$

yields the hierarchical system

$$\begin{aligned} P_{3h} \quad = \quad & \{\langle W_1 W_2 X_1 X_2 X_3 X_4 \rangle \in \{2 \ldots 20\}^2 \times \{1 \ldots 10\}^4 : \\ & Q(W_1, W_2) \\ & \wedge F_1(X_1, X_2, W_1) \wedge F_2(X_3, X_4, W_2)\} \end{aligned}$$

where the abstract constraint Q is defined as

$$Q(W_1, W_2) \quad \equiv \quad (W_1 \times W_2 = c_1)$$

and the two refinement constraints F_1 and F_2 are defined as

$$\begin{aligned} F_1(X_1, X_2, W_1) \quad & \equiv \quad (X_1 + X_2 = W_1) \\ F_2(X_3, X_4, W_2) \quad & \equiv \quad (X_3 + X_4 = W_2) \end{aligned}$$

Note that we use $\{2 \ldots 20\}$ as the range of our component abstraction functions, f_1 and f_2, instead of $\{1 \ldots 20\}$ as 1 can never occur as the sum of two integers greater than zero. This reduces the abstract search space size from 400 to 361. Again, it is easily verified that the two CSPs P_3 and P_{3h} are equivalent.

Any of the popular simple search algorithms can be applied to solve P_{3h}; projecting the solutions obtained on to the variables of P_3 will yield the solutions to P_3. We consider here the algorithm Backtrack, as applied to the control sequence $S_{3.1}$.

$$S_{3.1} = \langle W_1 \; W_2 \; Q(W_1, W_2) \; X_1 \; X_2 \; F_1(X_1, X_2, W_1) \; X_3 \; X_4 \; F_2(X_3, X_4, W_2) \rangle$$

In constructing a control sequence, any trusted heuristic may be used to order the variables and constraints. To keep with the principle of hierarchical solution, sequence $S_{3.1}$ is ordered to place the abstract level nodes before base level variables and constraints. In addition, the bandwidth of the refinement constraints in the base level portion is kept to a minimum by grouping together those variables. The general heuristic followed here is to test the refinement constraints as early as possible.

The results of applying Backtrack to $S_{3.1}$ are summarized in the figure 5.1. The first constraint in $S_{3.1}$ is the abstract constraint Q, so there are at least $\mathrm{Dom}(W_1 W_2) = 361$ constraint checks in each case. For $c_1 = 1$, the non-existence of any solutions is actually

detected in the abstract level portion of the control sequence itself, requiring only the 361 constraint checks. The total number of constraint checks required decreases with the number of solutions to Q found in the prefix portion of $S_{3.1}$. This is typical of the behavior of Backtrack. The hierarchical problem's sequence $S_{3.1}$ performs better than $S_{3.0}$ because of the smaller arity of the constraints in the former, allowing early testing, and early pruning of the search space. The number of generate events is also depicted in the figure for reference.

		$c_1 = 200$	$c_1 = 100$	$c_1 = 256$	$c_1 = 1$
Nbr. of solutions		89	18	25	0
BT on $S_{3.0}$	*checks*	10,000	10,000	10,000	10,000
	gens[1]	12,221	12,221	12,221	12,221
HSS-1 *BT on* $S_{3.1}$	*checks*	2,061	1,561	961	361
	gens	2,457	1,852	1,126	400
	#solns to Q	3	2	1	0
HSS-2 *BT on* $S_{3.2}$	*checks*	3,300	2,900	2,400	1,900
	gens	3,815	3,331	2,726	2,121
	#solns to Q	14	10	5	0
HSS-3 *BU-BT on* $S_{3.3}$	*checks*	2,061	1,561	961	361
	gens	5,415	4,031	2,326	621
	#solns to Q	3	2	1	0
HSS-4 *BU-BT on* $S_{3.4}$	*checks*[2]	361 + 200	361 + 200	361 + 200	361 + 200
	gens	1,368	1,138	1,138	1,042
	#solns to Q	3	2	1	0

KEY:
BT = Backtrack
BU-BT = Bottom-up Backtrack
checks = number of constraint checks
gens = number of generate events

NOTES:

1. The end of each variable's domain is detected by an extra generate event returning failure. This causes the generator to restart from the beginning of the domain on its next invocation.

2. The total number of computations shown in the number of constraint checks for $S_{3.4}$ is actually the sum of the number of true constraint checks and the number of times a computation of the value for W_1 or W_2 was required.

Figure 5.1: A comparison of different hierarchical solutions of P_3

5.3.3 Selecting an abstraction function

The problem P_{3h} differs from P_3 in two additional variables, each with a domain size of 19, and in having three constraints instead of one, though each of smaller arity. Searching in the abstract space is an additional overhead. Furthermore, in the control sequence $S_{3.1}$, the entire base level portion $(X_1X_2F_1(X_1, X_2, W_1)X_3X_4F_2(X_3, X_4, W_2))$ is executed once for each abstract solution (i.e. solution to Q). On the other hand, pruning of possibilities occurs much earlier in the control sequence due to the smaller arity of the abstract and refinement constraints.

A hierarchical solution procedure is effective when, by expending a small amount of effort in the abstract search space, it can reject large portions of the base search space for not containing a solution. The mapping from members of the abstract space to subsets of the base space is defined by the inverse of the abstraction function, and represented in our hierarchical CSPs by the refinement constraints. An ideal hierarchical CSP system must then have the following two characteristics. Through searching in the abstract space and testing abstract constraints, it must filter out a sizable portion of the abstract search space that maps into a very large portion of the base search space. In addition, after determining the abstract solutions, it must restrict its search in the base space as closely as possible to only the refinement of these abstract solutions.

This suggests the following desirable features in an abstraction function and the resulting hierarchical CSP:

- *Small abstract search space.* The smaller the abstract search space, the lower the additional search overhead. However, too small an abstract search space can be ineffective when practically every member is an abstract solution because it contains a base level solution in its refinement, thus removing the pruning power of the abstract level. Furthermore, since every base level tuple must have an abstraction (in our model), the size of each abstract solution's refinement gets larger as the abstract search space gets smaller.

- *Tight abstract constraint.* Smaller number of abstract solutions, by implication, hopefully lead to a smaller subset of the base space that will need to be searched.

- *Moving other constraints to the abstract level.* An example of this was presented in the previous section, in the derivation of the hierarchical system P_{2h}. If, after defining the abstract level, other constraints can be reformulated entirely in terms of the abstract level variables, they can help reduce the abstract level search effort, and also the number of abstract solutions.

- *Many component abstraction functions.* We define the *refinement zone* of a refinement constraint, a component abstraction function, or equivalently of the corresponding abstract variable, as the subset of base level variables in the argument set of that constraint. Having a large number of component abstraction functions results in a larger number of refinement constraints, hopefully each of smaller refinement zones, thus reducing the portion of the base level generated outside of the abstract solutions' refinement. Larger number of component abstraction functions should be translatable into frequent and early testing of refinement constraints. Unfortunately, it also results in a larger number of abstract level variables, and thus a potentially larger abstract search space with a higher arity abstract constraint.

- *Minimum overlap between refinement zones.* The purpose of this is the same as in the previous point. An abstraction function contains all the base level variables in its argument set. These are divided between the component abstraction functions, into the corresponding refinement zones. The greater the variable independence between these zones, the smaller the bandwidth of each refinement constraint in the base level portion of a control sequence. This results in earlier invocation of each refinement constraint, with smaller backtrack potential. (See, for example, the discussion on the connection between constraint bandwidth and backtrack overhead in [71]).

Usually, a small abstract search space and many refinement constraints are conflicting requirements, and a balance is needed. In the procedure HiT for constructing a hierarchical CSP described in the previous section, the abstract search space was defined from knowledge about the ranges of the component abstraction functions. These

component abstraction functions might themselves be complex expressions, and precision in the knowledge of their ranges becomes important. Restricting the range of a function to only those values that can actually be produced through the given variable domains, rather than simply taking a contiguous range of values from the smallest to the largest member of the range, can greatly reduce the size of the resulting abstract search space. See the floorplanning problem domain, described in the next chapter, for an example.

Section 5.4 describes the actual heuristics included in an implementation of HiT.

5.3.4 HSS-2: Simple search on a reduced hierarchical CSP

We discussed above the importance of a small abstract search space, and its restriction to values that do actually map to some generatable values of the base level variables. The domain of an abstract variable might still contain some values that cannot be mapped to valid values for its refinement zone variables in the base level. If problem P_3 for example had the additional constraint $(X_1 > 6)$, all values of W_1 less than seven in the corresponding version of the hierarchical CSP P_{3h} would not map to a valid value for the base level tuple $\langle X_1, X_2 \rangle$, given the component abstraction function f_1 requiring $W_1 = X_1 + X_2$.

Reducing the hierarchical CSP by choosing not to use one (or more) of the component abstraction functions can help reduce the possibilities of unmappable abstract values, and also reduce the abstract search space. A reduced version of P_{3h} can be defined by dropping the component abstraction function f_1 and its corresponding abstract variable W_1:

$$\begin{aligned} P_{3h2} \quad = \quad & \{\langle W_2 X_1 X_2 X_3 X_4 \rangle \in \{2 \ldots 20\} \times \{1 \ldots 10\}^4 : \\ & Q(X_1, X_2, W_2) \\ & \wedge F_2(X_3, X_4, W_2)\} \end{aligned}$$

where the abstract constraint Q is defined as

$$Q(X_1, X_2, W_2) \equiv ((X_1 + X_2) \times W_2 = c_1)$$

and the single remaining refinement constraint F_2 is defined as

$$F_2(X_3, X_4, W_2) \equiv (X_3 + X_4 = W_2)$$

The control sequence for this system then becomes

$$S_{3.2} = \langle X_1\ X_2\ W_2\ Q(X_1, X_2, W_2)\ X_3\ X_4\ F_2(X_3, X_4, W_2)\rangle$$

The abstract search space in P_{3h2} is a square root of the size of the abstract search space in P_{3h}. The problem P_{3h2} is of course derived from P_3, which does not have any constraints that could cause abstract values to be unmappable into the base level. The only benefit to be hoped for in reducing P_{3h} to P_{3h2} is then in the reduced overhead of searching the abstract search space. However this does not come without a price. The size of the domain of the first constraint in the control sequence $S_{3.2}$ is now $10 \times 10 \times 19 = 1900$, as compared to $19 \times 19 = 361$ for the control sequence $S_{3.1}$ for problem P_{3h}. While using a reduced hierarchical version of the problem reduces the size of the abstract search space, in this case, it also increases the level at which the first constraint gets tested. As can be seen in figure 5.1, this actually causes the reduced abstraction level hierarchical solver to show worse performance results than solving $S_{3.1}$. The next chapter presents an example in the floorplanning domain where the reverse is true.

5.3.5 HSS-3: Breadth-first search between levels

There are two possibilities in solving a problem with an abstraction level. The first is to find an abstract solution, then search in its refinement in the base level, and then go back to the abstract level to look for the next abstract solution and repeat the process until the entire space is exhausted. This procedure can be called *depth-first search between levels*, and is the solution procedure HSS-1 described above, implemented for P_{3h} in the control sequence $S_{3.1}$. An alternative procedure first finds all the abstract solutions, and then organizes its search in the base level in their combined refinements. We will call this procedure *breadth first search between levels*, and it can be implemented by applying DOI-Decomposition to the control sequence used by HSS-1, and solving the decomposed control sequence using a bottom-up search algorithm.

In general, any control sequence for the hierarchical problem may be selected for DOI-decomposition, and decomposed at any suitable point along the sequence. Details about applying DOI-decomposition to a problem and its subsequent bottom-up solution have been discussed in the previous two chapters. To implement breadth-first search between levels, we must take a simple control sequence that implements depth-first search between levels (by placing all abstract level nodes before any base level node), and place the decomposition point between the (subsequences for) the two levels. This results in a subproblem consisting entirely of abstract variables and constraints, and a residual subproblem that recalls these abstract solutions and combines them with base level variables and constraints through the refinement constraints.

For our example CSP P_3, we take its hierarchical version P_{3h} constructed for the first solution procedure described above, and decompose the control sequence $S_{3.1}$ as follows:

$$\begin{aligned} S_{3.3} \;=\; & \langle (W_1\ W_2\ Q(W_1, W_2)) \\ & \quad (X_1\ X_2\ W_1'\ F_1(X_1, X_2, W_1')\ X_3\ X_4\ W_2'\ F_2(W_2', X_3, X_4)) \rangle \end{aligned}$$

The performance results for Bottom-up Backtrack applied to $S_{3.3}$ are summarized in figure 5.1. As expected for DOI-decomposition, since the base level does not contain any constraints independent of the abstract variables (they are all refinement constraints), the number of constraint checks required for $S_{3.3}$ is the same as that for $S_{3.1}$. However, this will not usually be the case for problems that, unlike P_3, have more than one constraint. The number of generate events required to solve $S_{3.3}$, though much smaller than in the non-hierarchical case, is significantly larger than for the previous two hierarchical solution strategies. While similar in most respects to $S_{3.1}$, the bottom-up sequence $S_{3.3}$ recalls the cached values for abstract variables W_1 and W_2 in the base level. The amount of this additional generate overhead is controlled by the number of solutions to the portion of the base level sequence before these variables are recalled (e.g., $\langle X_1\ X_2 \rangle$ before W_1').

5.3.6 HSS-4: Bottom-Up evaluation of the decomposed constraint

The fourth approach to solving a hierarchical CSP that we will discuss here inverts the roles of the refinement constraints and the abstracted constraint. The purpose of a refinement constraint, as used for example in HSS-1, is to restrict base level search to the refinement of an abstract solution. This role is very similar to the role of the solution cache tree as used in Bottom-Up search (see Chapter 3), where recalled values of cached variables are restricted to those actually participating in solutions to a previous level. We can combine these two roles into a bottom-up evaluation of the decomposed global constraint. The procedure is demonstrated here by applying it to the example CSP P_3 and its corresponding hierarchical system P_{3h}.

The first step in the procedure, after constructing the hierarchical CSP P_{3h}, is to construct a subproblem for each component abstraction function, linking each value of the corresponding abstract variable to the values of the base variables it is computed from. The linkage is achieved in each subproblem by first generating the base level variables, and then computing the corresponding value of the abstract variable using the abstraction function. Each combination of generated and computed variables is saved as a solution to the subproblem, and cached in the solution cache tree. Note that explicit knowledge of the ranges of the component abstraction functions (which are also the domains for the abstract variables) is not needed here.

For P_{3h}, this yields two subproblems, one for each abstract variable:

$$\text{For } f_1: \quad (X_1\ X_2\ [W_1 = f_1(X_1, X_2)])$$

$$\text{For } f_2: \quad (X_3\ X_4\ [W_2 = f_2(X_3, X_4)])$$

The base level first recalls the values of the cached abstract variables, filters them through the abstract constraint Q, and then simply recalls the corresponding saved values of the base level variables, filtering these through any remaining base level constraints.

For P_{3h}, this results in the following three-part, two-level, bottom-up control sequence:

$$S_{3.4} \quad = \quad \langle (X_1\ X_2\ [W_1 = f_1(X_1, X_2)])$$

$$(X_3\ X_4\ [W_2 = f_2(X_3, X_4)])$$
$$(W_1'\ W_2'\ Q(W_1, W_2)\ X_1'\ X_2'\ X_3'\ X_4')\rangle$$

The values of W_1 and W_2 are computed in the first two subproblems, and cached along with the corresponding values of the base level variables as solutions to those subproblems. In the residual or base level subproblem, only those values of the (cached) abstract variables are passed by the prefix that satisfy the abstract constraint Q. The rest of the residual level then proceeds to recall those cached values of X_1, X_2 and X_3, X_4 that match the selected values of W_1 and W_2 respectively. In this case there are no more constraints remaining to be tested. Control sequence $S_{3.4}$ inverts the refinement constraints F_1 and F_2 by computing the corresponding component abstraction functions f_1 and f_2 in separate subproblems. The abstract constraint itself is actually tested in the residual or base level subproblem.

The generators for variables W_1 and W_2 in the first two subproblems do not generate from a listed domain, but instead compute the value for their variable. We will call generators of this type *functional generators*.

Note that the first two subproblems are actually variable-independent. This allows their solution cache trees to also be independent. We discussed in Chapter 3 that this is a desirable property, as the solution trees of interdependent subproblems need to be combined into one larger tree, resulting in additional computational overhead that might prove to be too high for the bottom-up approach to be beneficial. We were able to construct independent subproblems for the two abstract variables in P_{3h} as the corresponding component abstraction functions were also variable independent. This might not always be possible. The Corporate Decentralization application domain described in the next chapter presents a case where all the component abstraction functions are not variable independent. The solution there is to use a reduced form of bottom-up evaluation of the decomposed constraint, similar in principle to HSS-2 using simple search on a reduced hierarchical CSP.

The performance results for Bottom-up Backtrack applied to $S_{3.4}$ are summarized

in figure 5.1. As the two computed variables W_1 and W_2 actually require the evaluation of an expression (the corresponding component abstraction function), each such computation is really equivalent to a constraint check. This is reflected in the table in figure 5.1. Each subproblem requires $\mathrm{Dom}(X_i)^2$ computations of a functional generator, totaling to 200. Each computed abstract variable can take any of 19 possible values, thus requiring a minimum of 361 checks of the constraint Q in the residual subproblem. There aren't any other constraints in the problem, so that is also the total number of constraint checks.

In the figure, we see that $S_{3.4}$ fares better than the other solution approaches to P_3 for all values of the problem parameter c_1 except when $c_1 = 1$ and there are no solutions. For this value of c_1, the absence of any solutions is entirely detectable in the abstract level of the problem, and so HSS-1 and $S_{3.1}$ produces the best performance. For the other values of the problem parameter c_1, sequence $S_{3.4}$ offers much better performance.

There are two main reasons for the success of strategy HSS-4 on the hierarchical problem P_{3h}. The first reason for the success of $S_{3.4}$ is one that it shares with sequences $S_{3.1}$ and $S_{3.3}$, namely the small size of the abstract search space. The sizes of the ranges of both component abstraction functions F_i (19) are much smaller than the corresponding domain sizes ($\mathrm{Dom}(X_i)^2 = 10^2 = 100 \gg 19$). This brings the cost of solving the abstracted constraint Q down to only 361. Note that the cost of solving Q in this strategy is exactly the same as in the sequences $S_{3.1}$ and $S_{3.3}$.

The second reason for the success of $S_{3.4}$ is where it differs from these other two strategies. Sequence $S_{3.4}$ uses the bottom-up framework's efficient solution caching and recall scheme to prune away values of X_i which do not correspond to solutions to Q, with very little additional overhead. The only overhead here is the cost of solving the first two subproblems, which comes to a total of only 200 constraint test-equivalent computations. Other than that, the only cost in the problem is that of solving the abstract constraint Q.

In the next chapter, we will see an application of HSS-4 to the floorplanning problem where the problem of what would otherwise have been a large abstract search space is

overcome by the ability to use other problem constraints to trim the number of abstract variable values that get cached.

5.4 Implementation

5.4.1 HiT

The procedure for constructing a hierarchical problem from a given CSP is implemented with some heuristics in a program suite also called HiT. The first phase of the system operates on a problem and a selected global constraint, and produces a list of possible abstraction functions, ranked using the following heuristics, listed in their order of importance:

1. Arity(Q) < Arity(F_i). The arity of the new abstract constraint should be smaller than the highest refinement constraint arity. The aim in this heuristic is to keep the abstract problem as easy to solve as possible.

2. Decompositions with smaller arity refinement constraints are ranked higher.

3. Decompositions with independent refinement constraints are ranked higher. Decompositions with a greater number of mutually independent abstraction functions are ranked higher. The last two heuristics attempt to keep the refinement phase of the hierarchical solver efficient.

The final selection of an abstraction function, as well as the constraint to decompose, is left to the user. The abstractions that HiT produces are presently limited to simple syntactic decompositions of the global constraint expression.

The second phase of HiT, when given a CSP, an abstraction function, some domain knowledge on function ranges, and some domain rules for matching equivalent terms in constraint expressions, implements the full HiT procedure for constructing a hierarchical CSP. The term equivalency rules are used to detect other constraints that can be moved to the abstract level, during step five of HiT. A final component of the HiT suite is capable, in its present version, of producing control sequences that implement hierarchical solution strategies HSS-1 and HSS-3.

The next chapter describes some example domains, and the decision process used in deciding which hierarchical solution strategy to use to solve the given problem.

5.4.2 Hierarchical solution strategies

The basic goal of Global Constraint Decomposition is to transform a given CSP into a new and equivalent CSP in which one or more of the original global constraints have been replaced with a set of smaller arity constraints. This hierarchical CSP can be solved using any CSP solution procedure. The hierarchical solution strategies described above are four examples which take special advantage of the new problem's hierarchical structure.

Solution strategies HSS-1 and HSS-2 use simple control sequences, and any basic search algorithm can be used to execute these sequences.

Solution strategies HSS-3 and HSS-4 both require bottom-up solution. The bottom-up framework has so far been implemented on top of the Backtrack and Forward-Check basic search algorithms. Strategy HSS-4 also requires functional generators which compute the value of the abstract variable in the upper level subproblems. Only the Backtrack implementation of bottom-up solution currently supports functional generators.

5.5 Summary

In this chapter, we have presented a formalization of abstraction levels as used to construct hierarchical CSP systems. We used the formalization to define the procedure HiT for constructing a hierarchical problem from a given CSP by decomposing a global constraint in the problem. This was demonstrated on an example problem. Finally, we looked at four hierarchical search strategies for solving the resulting hierarchical system, their relative benefits and disadvantages, and how to select a suitable abstraction function.

The general idea of applying abstraction levels or spaces to constraint satisfaction problems described here, and previously developed in [45, 48, 46, 68], has been generalized by Ellman and described in [14, 15]. In Ellman's terms, the procedure HiT

constructs a hierarchical CSP by defining a "necessary approximate symmetry" on the original problem's search space. This necessary approximation is defined by the abstraction function and the hierarchical completeness condition, requiring that if a tuple of variable bindings from the base level search space is a solution to the original set of problem constraints R, then its abstraction is a solution to the abstract constraint Q. This is the same as the "upward solution property" of a hierarchical system as defined by Knoblock for planning domains in [37].

The hierarchical solution approaches discussed here are demonstrated again on more complex examples of CSPs in the next chapter.

Chapter 6

Some Application Case Studies

This chapter presents three case studies of the application of the hierarchical solution techniques discussed in previous chapters to three simple application domains. The domains have been selected for their particular topological properties. The first example, floorplanning, takes benefit of both the CSP decomposition techniques as well as Global Constraint Decomposition. The second problem, ship bulkhead design, is an example where only DOI-decomposition is applicable. The last example domain is that of decentralizing a corporation's departments, and displays a structure where only global constraint decomposition can be applied. We will examine in detail the applicability of each of the three decomposition techniques to these applications, discuss the reasons behind the selection of each solution strategy, and analyze their success.

6.1 A Brief Note on the Experiment Methodology

The aim of this chapter is to give examples of the decision processes needed to select one of the decomposition procedures proposed in this thesis. The applicability and suitability of each type of decomposition is discussed for each case. This evaluation is tested in actual test runs of the control sequences produced by each decomposition, comparing their performance with each other and with a 'benchmark' simple control sequence.

6.1.1 The basic search algorithms

Three sets of experiments were run for all control sequences. The first two sets used Backtrack and Forward-Check to look for all solutions. The multi-level control sequences were tested using bottom-up versions of the two search algorithms. HSS-4

control sequences require a special type of *functional* generator to compute the abstract variable values in the top-level subproblems. Current implementation of the bottom-up framework in Forward-Check does not support functional generators.

The third experiment set used Backtrack to look for one solution. Problem solving to find the first solution is not the primary focus of this thesis. Except where the problems are hard, predicting performance can be very erratic because of the strong dependence on the order in which values from each domain are generated. This makes the proper evaluation of alternative control sequences very difficult.

6.1.2 The benchmark simple control sequence

The benefits of employing hierarchical solution are evaluated and tested for each problem against a basic non-hierarchical control sequence. A hierarchical technique is considered useful only if it can offer better performance than this benchmark control sequence.

The performance benchmark is obtained by testing a simple, single-level control sequence on Backtrack and Forward-Check. This sequence is selected using a combination of the following heuristics:

- Minimize the total depth and bandwidth of constraints in the sequence ([47, 71]).
- Order adjacent constraints in decreasing order of estimated tightness ([63]).
- Order variables in increasing domain size.

In some cases, actual experimentation was used to select the fastest control sequence among apparently equally suitable alternatives.

6.1.3 The decompositions

This thesis proposes three types of decompositions:

1. *DOI-decomposition.* The decomposition procedure operates on a preselected simple control sequence, and uses heuristics to evaluate alternative DOI-decompositions

of that sequence. Each decomposition is obtained by cutting the given simple control sequence at a different point, dividing it into a prefix and a suffix.

2. *IS-decomposition.* There are two decomposition procedures, *IS-Forward (ISF)* and *IS-Back (ISB)*, for extracting independent sets from a given simple control sequence. Sometimes they produce the same decomposition. Heuristics are used to evaluate the potential benefit of either decomposition.

3. *Global constraint decomposition.* The HiT procedure operates on a global constraint and, using heuristics, suggests the most suitable abstraction to use. The abstraction is used to decompose the global constraint and construct a hierarchical version of the problem. There are four different hierarchical solution strategies, *HSS-1 – HSS-4*, that can then be used to solve the hierarchical problem.

The first two types of decompositions partition the problem's set of constraints. They are aimed at reducing the artificial serial dependencies set up in a simple control sequence. Reduction in artificial variable dependence (AVD) is the heuristic measure used here to evaluate the utility of each decomposition. Its use as a decision factor is described in Chapters 3 and 4, and in the floorplanning example below. The simple control sequence used as performance benchmark is also the control sequence used by these decomposition procedures for constructing their subproblem hierarchy.

For global constraint decomposition, HiT's heuristics are relied upon to produce an abstraction function. The factors used to decide between the different hierarchical solution strategies are discussed in each case study, with the first case study, of the floorplanning application domain, giving the most detailed discussion.

While the decomposition procedures are implemented in autonomous programs which incorporate all the heuristics, their best use is as providers of a small set of primary alternatives, with heuristic measures and other data that can then be used by a human to make the final selection. This methodology is used in each of the case studies in this chapter.

One of the goals of these case studies was to measure the performance of the decomposition algorithms themselves. Tests revealed that all the decomposition algorithms

took less than 50 mSec real time (on a SPARC 5) for each problem. The general conclusion is that the decomposition algorithms are sufficiently efficient.

6.1.4 Performance statistics

The test runs report the number of constraint tests, number of nodes generated and the total run time for each control sequence. The tests were run on a Sun SPARC 5 and SPARC 10. Since these are multi-tasking systems, the run-times are not reliable indicators of relative performance, so the number of constraint checks is used as the primary performance measure.

The total number of solutions cached in the top-level subproblems is also reported for the multi-level control sequences. The larger this number, the greater the overhead cost in building the solution cache.

6.2 Floorplanning

6.2.1 Problem description and model

Floorplanning problems are encountered in space planning and design tasks ranging from the design of buildings to the layout of VLSI circuits. The problem selected here is a simple version which nonetheless is non-trivial, and serves to demonstrate the power of the decomposition techniques proposed in this thesis. Similar problems have been studied for example in [13, 17, 16].

This floorplanning problem is an expansion of the problem described in the first chapter. There is a rectangular house of specified length and width, which must be filled with four rooms, each of a specified minimum area. The rooms are also rectangular, with their sides parallel to the house sides. A solution is a layout giving the position and sizes of all four rooms (figure 6.1).

To model this domain, we will represent each room (e.g., r_1) with the cartesian coordinates of its bottom-left $(r_1.x_1, r_1.y_1)$ and top-right $(r_1.x_2, r_1.y_2)$ corners. The house length (L) and width (W) then determine the limits of the domains of the x and y coordinates. In the experiments described below, we restrict the coordinates to

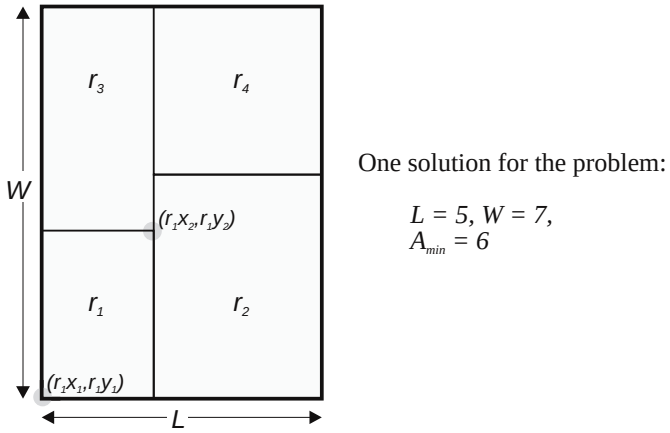


Figure 6.1: A layout in the four-room floorplanning problem.

be integral, and use a house length of 5 and width 7. This gives each x coordinate a domain size 5, and each y coordinate a domain size 7. There are a total of 16 variables in the problem, four for each room.

The constraints in the problem ensure that each room meets the specified minimum area (*MinArea*), no two rooms overlap (*DontOverlap*), and the rooms fill the house (*FillHouse*). In our experiments, we will use the same minimum area (A_{min}) for all four rooms. An additional constraint on each room ensures that it is adjacent to some house side (*AdjSide*). Though redundant, this constraint helps to limit the search for a solution. A final set of constraints, restricting the second corner of each room to be above and to the right of the first, is used to prevent generating each room instance twice. This gives a total of 23 constraints in the problem. A brief representation of the model is shown below, and the complete model may be examined in the appendix. (Note: $r_i.*$ is used to represent the list of all four variables $(r_i.x_1, r_i.y_1, r_i.x_2, r_i.y_2)$ for room i).

$$
\begin{aligned}
P_{f4} \quad = \quad & \{\langle r_1.*, r_2.*, r_3.*, r_4.*\rangle : \\
& \bigwedge_{i=1}^{4} (\text{RightOf}_i(r_i.x_2, r_i.x_1) \wedge \text{Above}_i(r_i.y_2, r_i.y_1) \wedge \\
& \qquad \text{MinArea}_i(r_i.*) \wedge \text{AdjSide}_i(r_i.*))
\end{aligned}
$$

$$\wedge \bigwedge_{1 \le i < j \le 4} \text{DontOverlap}_{ij}(r_i.*, r_j.*)$$
$$\wedge \text{FillHouse}(r_1.*, r_2.*, r_3.*, r_4.*) \}$$

where

$$\begin{aligned} r_i.x_1 &\in \{0, \ldots, L-1\} \\ r_i.x_2 &\in \{1, \ldots, L\} \\ r_i.y_1 &\in \{0, \ldots, W-1\} \\ r_i.y_2 &\in \{1, \ldots, W\} \end{aligned}$$

6.2.2 Basic search strategy

A combination of the heuristics of ordering variables in increasing domain size, and minimizing the depth and bandwidth of constraints produce what has been found (so far) to be the fastest simple control sequence for this problem. In a consecutive sequence of constraint nodes, the constraints are ordered in decreasing order of estimated tightness.

$$\begin{aligned} C_{f4} = \langle\, & r_1.x_1\ r_1.x_2\ \text{RightOf}_1\ r_1.y_1\ r_1.y_2\ \text{Above}_1\ \text{AdjSide}_1\ \text{MinArea}_1 \\ & r_2.x_1\ \ldots\ \text{AdjSide}_2\ \text{DontOverlap}_{12}\ \text{MinArea}_2 \\ & r_3.x_1\ \ldots\ \text{AdjSide}_3\ \text{DontOverlap}_{13}\ \text{DontOverlap}_{23}\ \text{MinArea}_3 \\ & r_4.x_1\ \ldots\ \text{AdjSide}_4\ \text{DontOverlap}_{14}\ \text{DontOverlap}_{24} \\ & \qquad \text{DontOverlap}_{34}\ \text{MinArea}_4\ \text{FillHouse}\,\rangle \end{aligned}$$

$$\begin{aligned} \text{AVD}(C_{f4}) &= 136 \\ \text{AVD}_{avg}(C_{f4}) &= 5.91 \end{aligned}$$

Eight of the 23 constraints in the problem are binary, and another eight are quarternary, causing the high artificial variable dependence in the control sequence C_{f4}. The *DontOverlap* constraint restricting pairs of rooms from overlapping needs to be tested on each of the six unique room-pair combinations possible among four rooms.

In this control sequence, with a single level look-ahead, Forward-Check will not

improve upon Backtrack's performance. In fact, Forward-Check's more complicated control will actually take more time and space than simple Backtrack.

6.2.3 Constraint-set partitioning

The high AVD figure of 136 for C_{f4}, and the strong component structure in the control sequence as well as the problem, strongly suggest that the system should benefit from the constraint-set partitioning decomposition techniques. There are seven possible DOI-decompositions of C_{f4}, corresponding to the seven internal constraint clusters preceding a variable in the control sequence. Both independent-set extraction algorithms of Chapter 3 produce the same independent-set, selecting the seven downstream instances of the constraints *Above*, and *RightOf*.

Evaluation of possible DOI-decompositions.

Seq	C_{f4} Cut Before	Prefix #Vars	Suffix #Constrs	*AVD(suffix,prefix)* Average	Total	Highest
$C_{\mathrm{DOI}(f4)_1}$	$r_1.y_1$	2	22	0.09	2	2
$C_{\mathrm{DOI}(f4)_2}$	$r_2.x_1$	4	19	0.63	12	4
$C_{\mathrm{DOI}(f4)_3}$	$r_2.y_1$	6	18	0.56	10	6
$C_{\mathrm{DOI}(f4)_4}$	$r_3.x_1$	8	14	2.00	28	8
$C_{\mathrm{DOI}(f4)_5}$	$r_3.y_1$	10	13	1.69	22	10
$C_{\mathrm{DOI}(f4)_6}$	$r_4.x_1$	12	8	6.00	48	12
$C_{\mathrm{DOI}(f4)_7}$	$r_4.y_1$	14	7	5.43	38	14

Evaluation of possible IS-decompositions.
Both algorithms produce the same independent-set decomposition.

Seq	Independent Set Size	#Vars	Max Arity	*AVD(ISet,C_{f4})* Average	Total	Highest
$C_{\mathrm{ISB}(f4)}$	7	14	2	8.00	56	14

Figure 6.2: Evaluation of possible constraint-set decompositions of C_{f4}.

A comparative evaluation of the different decompositions of C_{f4} is shown in figure 6.2. We continue to use artificial variable dependence (AVD) as the heuristic measure of the degree of redundant constraint testing in a control sequence. The tables give for each decomposition scheme the total, average and highest reduction in AVD among all affected constraints, when compared against the original sequence C_{f4}. For

each DOI-decomposition, the tables also give the depth at which the cut was made (in number of prefix variables) and the number of constraints in the resulting suffix. For the IS-decomposition, the tables give the number of constraints in the independent set. In both cases, that latter number is also the number of constraints whose AVD is potentially reduced.

DOI and IS decomposition procedures try to improve performance of a basic simple control sequence by reducing the artificial dependencies of constraints in that sequence. The approximation made in using AVD, as the measure of the degree of redundant testing of a constraint, is that a constraint will get more redundant tests if a control sequence causes it to artificially depend upon a larger number of independent variables. The AVD reduction resulting from a decomposition is thus an estimate of the degree of performance improvement offered by that decomposition.

Generally, the selected decomposition should then maximize a combination of the total AVD reduction, the number of constraints affected, and the depth at which greater constraint AVD reductions occur. The depth is important because in the hardest problems for simple search, deeper levels in the search tree will have a larger number of nodes. If constraints at those levels exhibit artificial serial dependence, they will tend to get a larger amount of redundant testing, and therefore offer a larger potential for improvement. Conversely, in the case of DOI-decomposition, cutting the simple search sequence at a deeper level can yield a costly independent subproblem. If the original control sequence was difficult, longer prefixes of the sequence will have a larger number of solutions, increasing the solution caching overhead when solving the prefix as the independent subproblem.

The AVD reduction in a DOI-decomposition occurs among constraints in the suffix used to build the residual subproblem. The table in figure 6.2 shows that as the depth of the DOI-decomposition cut in C_{f4} increases, so does the highest AVD reduction and, for the most part, also the total AVD reduction resulting from the decomposition. The reason is obvious: low arity constraints at deeper levels in the control sequence will have greater artificial dependence. The actual shape of the AVD envelope is determined by the constraint topology of the problem. Control sequence C_{f4} has binary and

quarternary constraints at regular intervals throughout the length of the sequence.

Even though the seventh DOI-decomposition, control sequence $C_{\mathrm{DOI}(f4)_7}$, has the largest maximum reduction in AVD of any constraint, the preceding sequence $C_{\mathrm{DOI}(f4)_6}$ appears as the clear winner in our heuristic evaluation, due to its significantly higher total and average AVD reduction figures affecting a slightly larger number of constraints.

From the evaluation table, the IS-decomposition also looks very promising, offering at 56 the highest total AVD reduction, 8 more than the best DOI decomposition, and affecting 7 constraints, only one less than $C_{\mathrm{DOI}(f4)_6}$. The major difference between the two decompositions is that $C_{\mathrm{DOI}(f4)_6}$ gets all its AVD reductions at room 4 constraints, occurring deep within C_{f4}. The IS-decomposition $C_{\mathrm{ISB}(f4)}$ gets 36 of its total 56 AVD reductions from constraints tested before room 4 in the original sequence. As all its constraints are binary, the IS-decomposition should have a smaller solution caching overhead, but also a smaller performance improvement, than the DOI-decomposition $C_{\mathrm{DOI}(f4)_6}$.

6.2.4 Global constraint decomposition

The floorplanning problem P_{f4} has a single global constraint, whose task is to ensure that the 4 rooms completely fill the house. With the rest of the problem formulation ensuring that all rooms are inside the house, and that they do not overlap each other, all this constraint has to do is to make sure that the total room area is the same as the house area. This formulation is given below:

$$\mathrm{FillHouse}(r_1.*, r_2.*, r_3.*, r_4.*) \equiv \sum_{i=1}^{4} (r_i.x_2 - r_i.x_1) \times (r_i.y_2 - r_i.y_1) = L \times W$$

This constraint's syntax follows the room-based component structure in the problem, and HiT's heuristics (section 5.3) produce the following abstraction function, abstracting each room into its area:

$$\begin{aligned} f &= \langle f_1, f_2, f_3, f_4 \rangle \\ f_i(r_i.*) &= (r_i.x_2 - r_i.x_1) \times (r_i.y_2 - r_i.y_1) \end{aligned}$$

Substituting each component abstraction function f_i with the abstract variable z_i produces the abstracted global constraint *FillHouse-ac* , and refinement constraints F_i,

$$\begin{aligned} \text{FillHouse-ac}(z_1, z_2, z_3, z_4) &\equiv z_1 + z_2 + z_3 + z_4 = L \times W \\ F_i(z_i, r_i.*) &\equiv z_i = (r_i.x_2 - r_i.x_1) \times (r_i.y_2 - r_i.y_1) \end{aligned}$$

The four abstraction components are variable independent, and the arity requirement Arity(FillHouse-ac) $= 4 <$ Arity$(F_i) = 5$ of the abstraction selection heuristics is satisfied.

We can compute the domain of the abstract variable z_i from the range of the component functions f_i, which turns out to be the same in all four cases. The size of this set for the problem when house length and width are 5 and 7, respectively, is 24.

$$\begin{aligned} z_i \in \{&1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, \\ &15, 16, 18, 20, 21, 24, 25, 27, 28, 30, 32, 35\} \end{aligned}$$

The area abstraction of each room maps a set of 420 different possible valid room instances into a set of 24 possible areas. This is a very good contraction, and yields an abstract search space size of only $24^4 = 331,776$. The resulting hierarchical problem P_{f4h} is easy to construct, and a brief description is given below. Using the room-area abstraction allows the *MinArea* constraint to be moved into the abstract level, and help further reduce abstract level search effort.

$$\begin{aligned} P_{f4h} = \{\langle &z_1, z_2, z_3, z_4, r_1.*, r_2.*, r_3.*, r_4.* \rangle : \\ &\ldots \textit{ the new abstract constraints, } \ldots \\ &\wedge \bigwedge_{i=1}^{4} \text{MinArea-ac}_i(z_i) \\ &\wedge \text{FillHouse-ac}(z_1, z_2, z_3, z_4) \\ &\ldots \textit{ the refinement constraints, } \ldots \\ &\wedge \bigwedge_{i=1}^{4} F_i(z_i, r_i.*) \\ &\ldots \textit{ and the remaining base level constraints } \ldots \\ &\wedge \bigwedge_{i=1}^{4} (\text{RightOf}_i(r_i.x_2, r_i.x_1) \wedge \text{Above}_i(r_i.y_2, r_i.y_1) \wedge \text{AdjSide}_i(r_i.*)) \end{aligned}$$

$$\wedge \bigwedge_{1 \le i < j \le 4} \text{DontOverlap}_{ij}(r_i.*, r_j.*) \; \}$$

All four hierarchical solution strategies discussed in section 5.3 can be applied to this hierarchical problem. Strategy HSS-1 constructs the simple control sequence $C_{\text{HSS-1}(f4h)}$ by placing the abstract level nodes before the base level nodes. The refinement constraints are inserted among the base level nodes such that they are the first constraint of that arity to be tested in each constraint cluster.

$$\begin{aligned} C_{\text{HSS-1}(f4h)} \;=\; \langle\, & z_1 \text{ MinArea-ac}_1 \;\ldots\; z_4 \text{ MinArea-ac}_4 \text{ FillHouse-ac} \\ & r_1.x_1 \; r_1.x_2 \text{ RightOf}_1 \; r_1.y_1 \; r_1.y_2 \text{ Above}_1 \; F_1 \text{ AdjSide}_1 \\ & r_2.x_1 \;\ldots\; F_2 \text{ AdjSide}_2 \text{ DontOverlap}_{12} \\ & r_3.x_1 \;\ldots\; F_3 \text{ AdjSide}_3 \text{ DontOverlap}_{13} \text{ DontOverlap}_{23} \\ & r_4.x_1 \;\ldots\; F_4 \text{ AdjSide}_4 \text{ DontOverlap}_{14} \text{ DontOverlap}_{24} \text{ DontOverlap}_{34} \,\rangle \end{aligned}$$

Hierarchical solution strategy HSS-3's control sequence $C_{\text{HSS-3}(f4h)}$ is simply a DOI-decomposition of $C_{\text{HSS-1}(f4h)}$, obtained by cutting it just before the first base level variable $r_1.x_1$. Strategy HSS-2 uses a reduced abstraction, obtained by dropping one of the abstraction components. In $C_{\text{HSS-2}(f4h)}$ we drop the component abstraction function f_1 abstracting room 1 into its area. The resulting control sequence is constructed on the same principles used for $C_{\text{HSS-1}(f4h)}$.

$$\begin{aligned} C_{\text{HSS-2}(f4h)} \;=\; \langle\, & r_1.x_1 \; r_1.x_2 \text{ RightOf}_1 r_1.y_1 \; r_1.y_2 \text{ Above}_1 \text{ AdjSide}_1 \text{ MinArea}_1 \\ & z_2 \text{ MinArea-ac}_2 \;\ldots\; z_4 \text{ MinArea-ac}_4 \text{ FillHouse-ac3}(z_2, z_3, z_4, r_1.*) \\ & r_2.x_1 \;\ldots\; F_2 \text{ AdjSide}_2 \text{ DontOverlap}_{12} \\ & r_3.x_1 \;\ldots\; F_3 \text{ AdjSide}_3 \text{ DontOverlap}_{13} \text{ DontOverlap}_{23} \\ & r_4.x_1 \;\ldots\; F_4 \text{ AdjSide}_4 \text{ DontOverlap}_{14} \text{ DontOverlap}_{24} \text{ DontOverlap}_{34} \,\rangle \end{aligned}$$

Finally, control sequence $C_{\text{HSS-4}(f4h)}$ is an implementation of solution strategy HSS-4, constructed by building subproblems to compute each component abstraction. The values of the four abstract variables z_i are computed using the component abstraction

functions f_i in four separate and independent subproblems. The abstracted *FillHouse-ac* constraint is then tested in the residual subproblem at the base level, as the first constraint in that level. This decomposition of the control sequence also brings the artificial dependencies down.

$$
\begin{aligned}
C_{\text{HSS-4}(f4h)} \;=\; \langle (\; & r_1.x_1 \; r_1.x_2 \; \text{RightOf}_1 \; r_1.y_1 \; r_1.y_2 \; \text{Above}_1 \; \text{AdjSide}_1 \\
& [z_1 = f_1(r_1.*)] \; \text{MinArea-ac}_1(z_1) \;) \\
& \quad \vdots \\
& (\; r_4.x_1 \; r_4.x_2 \; \text{RightOf}_4 \; r_4.y_1 \; r_4.y_2 \; \text{Above}_4 \; \text{AdjSide}_4 \\
& [z_4 = f_4(r_4.*)] \; \text{MinArea-ac}_4(z_4) \;) \\
& (\; z'_1 \; z'_2 \; z'_3 \; z'_4 \; \text{FillHouse-ac} \\
& r_1.x'_1 \; r_1.x'_2 \; r_1.y'_1 \; r_1.y'_2 \\
& r_2.x'_1 \; \ldots \; r_2.y'_2 \\
& \text{DontOverlap}_{12} \\
& r_3.x'_1 \; \ldots \; r_3.y'_2 \\
& \text{DontOverlap}_{13} \; \text{DontOverlap}_{23} \\
& r_4.x'_1 \; \ldots \; r_4.y'_2 \\
& \text{DontOverlap}_{14} \; \text{DontOverlap}_{24} \; \text{DontOverlap}_{34} \;) \rangle
\end{aligned}
$$

$$
\text{AVD}(C_{\text{HSS-4}(f4h)}) \;=\; 64
$$

Heuristics for selecting a good abstraction have been discussed in Chapter 5. The general rule for selecting between possible hierarchical solution strategies should be as follows. If HSS-4 is applicable, it should be the first choice, especially when the set of abstract (in this case computed) variables have a very small (when compared to the size of the base level search space being abstracted) number of solutions reach the base level. In $C_{\text{HSS-4}(f4h)}$, the size of the abstract search space is already very small (only 331,776, compared with over 31 billion for the base level after all *Above* and *RightOf* constraints are satisfied). Testing *Above*, *RightOf* and *MinArea-ac* in the subproblems further trims the number of values of z_i that reach the base level. All this makes the

number of times *FillHouse-ac* gets tested in the base level very small, making HSS-4 more efficient.

When HSS-4 is not suitable, the next strategy to consider is HSS-2. If dropping one of the component abstraction functions can greatly reduce the size of the remaining abstract search space, and the constraints on the base level variables of the dropped component are known to be very tight, then HSS-2 becomes suitable. Our floorplanning problem is very symmetrical, and not a very good candidate for this strategy. If, for example, room 1 had been required by the problem to be square, only five of the possible 24 areas in the test problem would be valid. Dropping f_1 from the abstraction would then yield better performance.

Deciding between HSS-1 and HSS-3 is really a question of whether or not to apply DOI-decomposition. Potential reduction in artificial dependencies, and the expected number of abstract solutions are the deciding factors. The rearrangement of abstract variables in the residual base level in $C_{\text{HSS-3}(f4h)}$ yields an AVD reduction of 6 over the sequence $C_{\text{HSS-1}(f4h)}$.

6.2.5 Performance and analysis

The set of 13 control sequences was tested on four instances of the floorplanning problem, with all four rooms required to have the same minimum area A_{min}, using the following parameter values

$$\begin{aligned} L &= 5 \\ W &= 7 \\ A_{min} &\in \{8, 7, 6, 5\} \end{aligned}$$

Decreasing the minimum room area from 8 to 5 increased the number of solutions to the problem from 0 to 5,328. The test results are shown in figures 6.3, 6.4, 6.5 and 6.6. The best performing control sequence in each decomposition category is highlighted in the tables. The graph of figure 6.6 gives a visual comparison of these sequences.

The clear performance winner when looking for all solutions is the HSS-4 sequence $C_{\text{HSS-4}(f4)}$. The reasons for this success are two-fold. The abstraction, in spite of its

Seq	Amin = 8		NumSol = 0		Amin = 7		NumSol = 576	
	Tests	Nodes	NumChSol	Time (S)	Tests	Nodes	NumChSol	Time (S)
C_{f4}	14,168,920	7,143,730	0	20	22,205,424	11,137,900	0	30
$C_{DOI(f4)1}$	14,168,234	7,143,506	15	19	22,204,738	11,137,676	15	30
$C_{DOI(f4)2}$	14,009,620	7,102,180	136	19	22,040,224	11,093,980	141	30
$C_{DOI(f4)3}$	14,012,309	7,095,776	2,040	19	22,043,038	11,091,026	2,115	30
$C_{DOI(f4)4}$	10,398,790	5,683,670	2,524	15	17,739,934	9,442,201	2,988	25
$C_{DOI(f4)5}$	10,461,179	5,857,628	37,860	16	17,813,923	9,608,794	44,820	26
$C_{DOI(f4)6}$	5,182,294	3,788,259	5,550	10	6,496,932	5,298,786	9,672	14
$C_{DOI(f4)7}$	5,320,333	4,013,188	83,250	11	6,738,021	5,584,162	145,080	16
$C_{IS(f4)}$	7,928,856	4,601,679	157	15	12,476,200	7,126,974	157	22
$C_{HSS-1(f4)}$	5,701,349	4,080,556	0	9	14,683,233	10,513,126	0	22
$C_{HSS-2(f4)}$	6,268,262	4,688,526	0	13	15,320,652	11,214,196	0	30
$C_{HSS-3(f4)}$	4,675,406	4,849,786	16	11	10,994,339	11,380,726	68	26
$C_{HSS-4(f4)}$	168,783	252,429	544	0.6	330,556	554,454	564	1.3

Seq	Amin = 6		NumSol = 2,160		Amin = 5		NumSol = 5,328	
	Tests	Nodes	NumChSol	Time (S)	Tests	Nodes	NumChSol	Time (S)
C_{f4}	171,680,242	83,735,920	0	231	365,078,832	176,353,510	0	492
$C_{DOI(f4)1}$	171,679,556	83,735,696	15	231	365,078,146	176,353,286	15	492
$C_{DOI(f4)2}$	171,470,202	83,675,770	179	231	364,846,372	176,289,040	198	492
$C_{DOI(f4)3}$	171,473,966	83,673,626	2,685	231	364,850,611	176,283,506	2,970	492
$C_{DOI(f4)4}$	160,510,812	79,371,784	7,436	219	349,217,542	170,118,057	10,548	475
$C_{DOI(f4)5}$	160,696,001	79,840,355	111,540	222	349,480,531	170,885,788	158,220	480
$C_{DOI(f4)6}$	24,326,834	26,201,581	88,632	68	44,522,380	51,342,425	191,958	134
$C_{DOI(f4)7}$	26,541,923	26,549,692	1,329,480	282	49,320,619	51,055,124	2,879,370	1,729
$C_{IS(f4)}$	98,532,058	52,827,444	157	169	211,023,328	111,002,109	157	362
$C_{HSS-1(f4)}$	103,864,717	74,121,746	0	158	247,888,324	177,166,696	0	378
$C_{HSS-2(f4)}$	104,723,039	75,106,813	0	201	248,775,386	178,276,522	0	476
$C_{HSS-3(f4)}$	81,983,685	85,121,966	200	195	197,568,228	205,773,196	448	471
$C_{HSS-4(f4)}$	1,962,807	3,521,359	716	8.3	4,371,780	7,927,785	792	18.9

Figure 6.3: Run statistics for P_{f4} and P_{f4h} — looking for all solutions with Backtrack.

Seq	Amin = 8 Tests	Nodes	NumSol = 0 NumChSol	Time (S)	Amin = 7 Tests	Nodes	NumSol = 576 NumChSol	Time (S)
C_{f4}	14,168,920	6,418,800	0	29	22,205,424	10,008,978	0	39
$C_{DOI(f4)1}$	14,168,234	6,418,587	15	29	22,204,738	10,008,761	15	39
$C_{DOI(f4)2}$	13,968,618	6,339,844	136	29	21,998,137	9,927,751	141	40
$C_{DOI(f4)3}$	13,971,307	6,341,987	2,040	29	22,000,951	9,931,718	2,115	39
$C_{DOI(f4)4}$	9,988,550	4,607,895	2,524	23	17,255,337	7,887,430	2,988	33
$C_{DOI(f4)5}$	10,050,939	4,742,328	37,860	24	17,329,326	8,041,718	44,820	34
$C_{DOI(f4)6}$	4,593,844	2,322,839	5,550	11	5,487,083	2,905,389	9,672	14
$C_{DOI(f4)7}$	4,731,883	2,579,431	83,250	13	5,728,172	3,330,618	145,080	19
$C_{IS(f4)}$	7,928,856	3,711,400	157	34	12,476,200	5,771,973	157	46
$C_{HSS-1(f4)}$	5,701,349	3,671,899	0	19	14,683,233	9,456,318	0	42
$C_{HSS-2(f4)}$	6,268,262	4,246,981	0	19	15,320,652	10,123,625	0	43
$C_{HSS-3(f4)}$	5,988,327	2,954,094	16	14	13,500,426	6,627,958	68	30

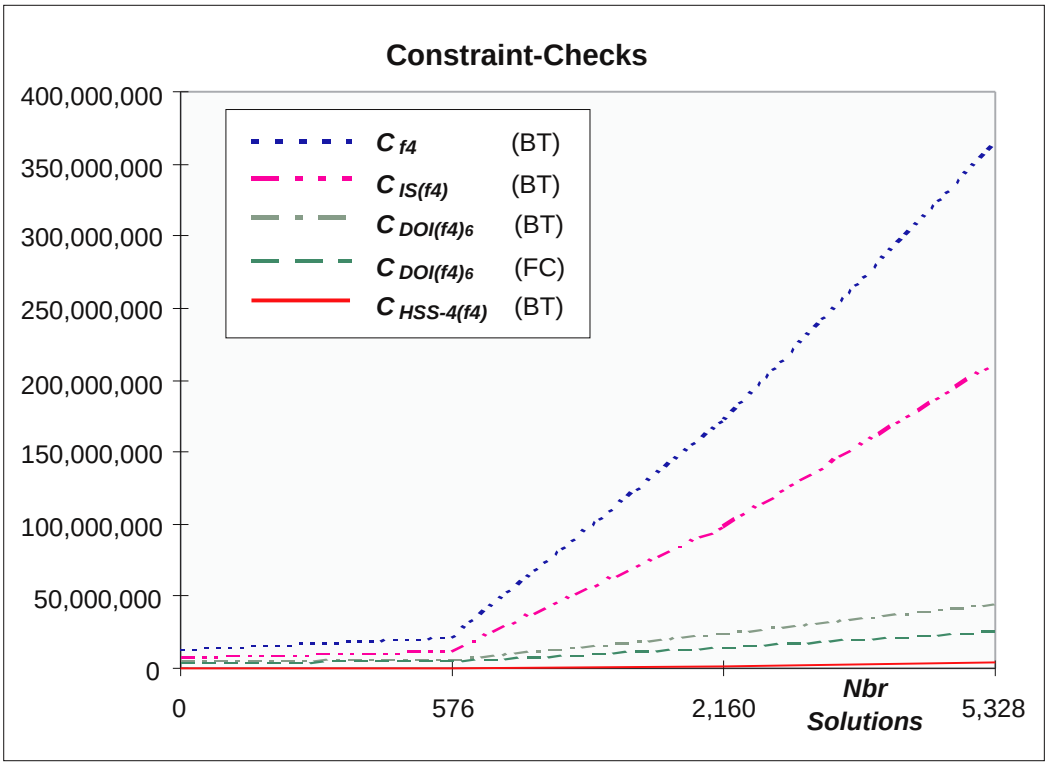
Seq	Amin = 6 Tests	Nodes	NumSol = 2,160 CSolns	Time (mS)	Amin = 5 Tests	Nodes	NumSol = 5,328 CSolns	Time (mS)
C_{f4}	171,680,242	75,233,711	0	303	365,078,832	158,444,041	0	639
$C_{DOI(f4)1}$	171,679,556	75,233,495	15	300	365,078,146	158,443,829	15	637
$C_{DOI(f4)2}$	171,420,997	75,137,579	179	301	364,794,570	158,345,061	198	639
$C_{DOI(f4)3}$	171,424,761	75,142,290	2,685	302	364,798,809	158,346,626	2,970	643
$C_{DOI(f4)4}$	159,315,569	70,135,387	7,436	293	347,565,530	151,359,562	10,548	634
$C_{DOI(f4)5}$	159,500,758	70,582,251	111,540	288	347,828,519	152,077,223	158,220	622
$C_{DOI(f4)6}$	15,629,551	11,243,710	88,632	72	25,851,212	22,193,174	191,958	161
$C_{DOI(f4)7}$	17,844,640	14,496,133	1,329,480	200	30,649,451	28,574,473	2,879,370	683
$C_{IS(f4)}$	98,532,058	43,128,856	157	346	211,023,328	90,746,669	157	726
$C_{HSS-1(f4)}$	103,864,717	66,635,775	0	305	247,888,324	159,266,104	0	719
$C_{HSS-2(f4)}$	104,723,039	67,580,837	0	287	248,775,386	160,344,334	0	681
$C_{HSS-3(f4)}$	105,336,369	50,637,430	200	233	258,304,258	123,685,220	448	576

Figure 6.4: Run statistics for P_{f4} and P_{f4h} — looking for all solutions with Forward-Check.

	Amin = 8	NumSol = 0			Amin = 7	NumSol = 1		
Seq	Tests	Nodes	CSolns	Time (mS)	Tests	Nodes	CSolns	Time (mS)
C_{f4}	14,168,920	7,143,730		19,200	3,313	1,755		0
$C_{DOI(f4)_1}$	14,168,234	7,143,506	15	19,120	26,484	13,040	15	40
$C_{DOI(f4)_2}$	14,009,620	7,102,180	136	18,980	5,426	3,609	141	10
$C_{DOI(f4)_3}$	14,012,309	7,095,776	2,040	19,010	35,819	22,559	2,115	70
$C_{DOI(f4)_4}$	10,398,790	5,683,670	2,524	15,120	250,927	149,809	2,988	410
$C_{DOI(f4)_5}$	10,461,179	5,857,628	37,860	15,970	404,357	327,998	44,820	1,310
$C_{DOI(f4)_6}$	5,182,294	3,788,259	5,550	9,870	5,343,307	2,812,487	9,672	7,550
$C_{DOI(f4)_7}$	5,320,333	4,013,188	83,250	11,440	5,697,198	3,334,537	145,080	10,920
$C_{IS(f4)}$	7,928,856	4,601,679	157	13,980	1,569	1,117	157	0
$C_{HSS\text{-}1(f4)}$	5,701,349	4,080,556	0	8,740	2,464	1,860	0	10
$C_{HSS\text{-}2(f4)}$	6,268,262	4,688,526	0	12,760	2,446	1,835	0	10
$C_{HSS\text{-}3(f4)}$	4,675,406	4,849,786	16	11,320	109,115	115,319	68	220
$C_{HSS\text{-}4(f4)}$	168,783	252,429	544	600	6,222	5,009	564	20

	Amin = 6	NumSol = 1			Amin = 5	NumSol = 1		
Seq	Tests	Nodes	CSolns	Time (mS)	Tests	Nodes	CSolns	Time (mS)
C_{f4}	3,649,093	1,740,885		4,950	8,602	4,054		10
$C_{DOI(f4)_1}$	411,970	193,543	15	560	1,898	986	15	0
$C_{DOI(f4)_2}$	3,470,832	1,696,168	179	4,700	105,500	50,088	198	140
$C_{DOI(f4)_3}$	416,563	204,430	2,685	590	11,558	10,657	2,970	40
$C_{DOI(f4)_4}$	3,622,373	1,822,087	7,436	5,050	478,455	276,057	10,548	840
$C_{DOI(f4)_5}$	1,031,049	771,343	111,540	3,210	757,196	669,723	158,220	3,460
$C_{DOI(f4)_6}$	13,531,645	7,320,218	88,632	20,760	19,315,090	10,447,273	191,958	30,730
$C_{DOI(f4)_7}$	16,742,419	11,282,063	1,329,480	74,890	25,251,614	17,481,791	2,879,370	204,120
$C_{IS(f4)}$	2,126,914	1,094,955	157	3,660	12,270	5,979	157	20
$C_{HSS\text{-}1(f4)}$	218,413	158,559	0	330	1,744	1,308	0	0
$C_{HSS\text{-}2(f4)}$	1,817,720	1,301,910	0	8,330	1,724	1,283	0	0
$C_{HSS\text{-}3(f4)}$	1,638,356	1,712,034	200	3,890	166,971	175,498	448	340
$C_{HSS\text{-}4(f4)}$	8,294	8,830	716	20	6,243	5,044	792	20

Figure 6.5: Run statistics for P_{f4} and P_{f4h} — looking for one solution with Backtrack.



BT = Backtrack *FC = Forward-Check*

Only $C_{\mathrm{DOI}(f4)_6}$ is affected by Forward-Check.

Figure 6.6: A comparison of the fastest control sequences in each category, looking for all solutions, for problem P_{f4}.

rather small search space, is very effective in pruning out non-solutions, and drastically reduces the arity of the global constraint (arity 16) down to five constraints of maximum arity 5. Secondly, the HSS-4 solution strategy of partitioning the problem's variables into different subproblems according to the four component abstraction functions also helps in reducing artificial dependencies, down to 64 from 136 for C_{f4}. These factors combine to make this sequence the most efficient. Even the total number of cached solutions, at less than a thousand, is very small.

As was expected, solution strategy HSS-2 does not improve upon HSS-1. As it turns out, its actual performance is only slightly worse. Strategy HSS-3's use of DOI-decomposition in $C_{\text{HSS-3}(f4)}$ does indeed improve upon the single-level HSS-1 sequence $C_{\text{HSS-1}(f4)}$. This is to be expected when the decomposition achieves a reduction in artificial dependence. The efficiency of the abstract level is demonstrated here in the small number of abstract solutions ($16 - 471$) needed to be cached by the bottom-up solution procedure.

The four different instances of the problem tested differ on the value of the minimum area for each room. This quantity controls the number of solutions, and makes the choice of the room-area abstraction very fortunate. In fact the absence of any solutions when $A_{min} = 8$ is completely detectable in the abstract level.

Among the constraint-set decomposition alternatives, the DOI-decomposition implemented in $C_{\text{DOI}(f4)_6}$ has the best performance, again as expected and discussed earlier in this section. Interestingly enough, the DOI-decompositions are the only sequences to actually improve their performance with Forward-Check. This will not generally be the case, and is a result of the particular constraint topology in the problem.

Figure 6.6 shows that as decreasing the value of A_{min} increases the number of solutions, it also makes the problem harder. This graph compares the performance of the basic benchmark sequence C_{f4} with the independent-set decomposition, best DOI-decomposition and HSS-4 sequences. In all three of these, the use of problem decomposition achieves a reduction in artificial serial dependence. In such cases, as the problem gets harder, the performance of the hierarchical sequence gets better relative to the benchmark simple sequence C_{f4}.

Finding one solution for floorplanning is a relatively easy problem in this model, except of course when there are no solutions. The simple control sequence C_{f4} is very effective, because of a fortunate value ordering used during search. When a constraint-set decomposition is used to look for the first solution, the bottom-up solution procedure still searches for all solutions to the upper-level subproblems. The most efficient decompositions will generally be those with simpler upper-level subproblems. In the case of the floorplanning problem P_{f4}, the independent-set decomposition has a set of seven binary subproblems in the upper level, with a combined search effort of $7 \times (5 \times 7) = 245$ constraint checks. The IS-decomposition $C_{\mathrm{ISB}(f4)}$, along with the second DOI-decomposition $C_{\mathrm{DOI}(f4)_2}$ which achieves an AVD reduction of 12 with only four variables in the prefix, offer the most consistent performance improvement. Among the abstractions, strategy HSS-1 offers the most consistent performance, presumably because of a fortunate value ordering in the abstract variables' domains, and because the rest of this control sequence follows the original C_{f4} very closely.

6.3 Corporation Decentralization

6.3.1 Problem description and model

This problem[1] models the situation of a large corporation that wishes to move some of its departments out of the main city to other locations, in order to take advantage of distributed marketing, cheaper housing and tax incentives. However, moving departments to different cities will increase communication costs between departments. The corporate management wishes to find a relocation that provides an overall benefit above a specified minimum, and which will also be acceptable to all its company presidents. The analysis department of the corporation has been asked to provide all relocation schemes that will provide the specified minimum benefit, to the company presidents for their vote.

In our problem model, we will assign one variable to each department. The value

[1] This problem is adapted from problem 12.10 in [70]. An expanded version of the original problem is described here.

of each variable in a solution will be the new location of that department in one relocation scheme. The corporation (c_0 in our model), owns two companies (c_1 and c_2). The first company has three departments ($c_1.d_1, \ldots, c_1.d_3$), and the second company has four departments ($c_1.d_1, \ldots, c_1.d_4$). The corporate headquarters c_0 communicates with its companies through the company headquarters $c_1.d_1$ and $c_2.d_1$. All other significant communications are only between the departments within either company. The corporate structure and communication links are shown in figure 6.7.

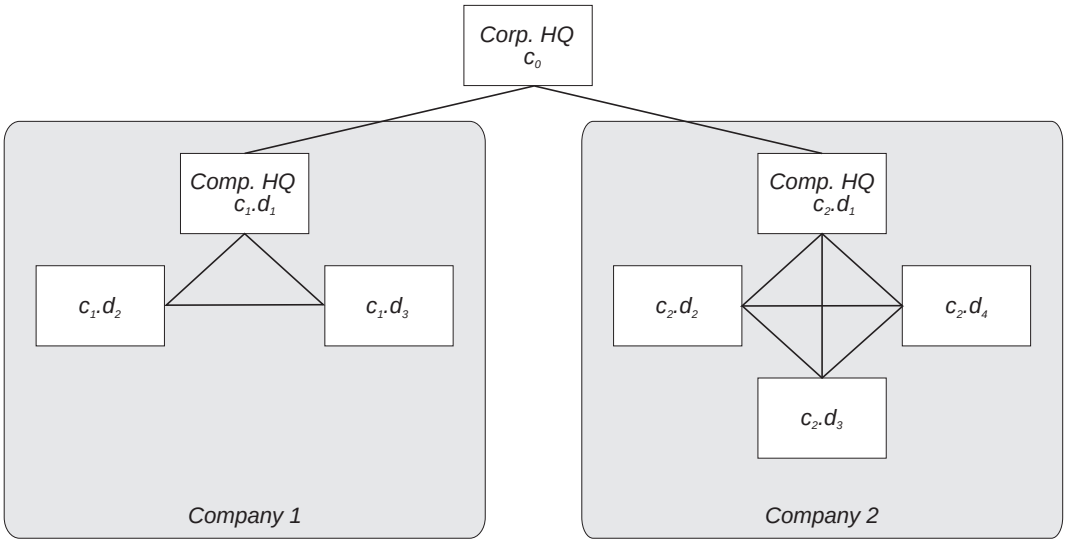


Figure 6.7: The eight departments of the corporation of problem P_{dc}, with their communication links.

As part of its decentralization strategy, the corporation is targeting four other locations besides the main city, and wants to limit the number of departments in any city to four. The management has priced the benefits and costs for all possible locations of each department, and these are shown in the tables in figure 6.8.

The model then has eight variables, one for each department in the corporation, and each variable has a domain of five cities.

There are two types of constraints in the model. The first type (*CityLimit* in the model) restricts the final number of departments in each city to four. For early pruning, a serial control sequence can begin testing this constraint after five department variables

Benefit of moving a department ($\times$\$1000).
All departments are presently in City 1.

	Cities				
Department	1	2	3	4	5
c_0	0	400	400	300	300
$c_1.d_1$	0	600	800	400	600
$c_1.d_2$	0	200	300	200	200
$c_1.d_3$	0	400	800	600	400
$c_2.d_1$	0	600	800	400	600
$c_2.d_2$	0	200	300	200	200
$c_2.d_3$	0	400	800	600	400
$c_2.d_4$	0	400	800	600	400

Inter-department communication volume.

Department	c_0	$c_1.d_1$	$c_1.d_2$	$c_1.d_3$	$c_2.d_1$	$c_2.d_2$	$c_2.d_3$	$c_2.d_4$
c_0		3	0	0	3	0	0	0
$c_1.d_1$			5	7	0	0	0	0
$c_1.d_2$				14	0	0	0	0
$c_1.d_3$					0	0	0	0
$c_2.d_1$						7	7	5
$c_2.d_2$							10	15
$c_2.d_3$								12
$c_2.d_4$								

Inter-city communication costs, per unit volume ($\times$\$1000).

City	1	2	3	4	5
1	10	13	9	10	13
2		5	14	5	13
3			5	9	14
4				5	14
5					5

Figure 6.8: Moving benefits and communication costs in the planned corporation decentralization.

have been instantiated. For this purpose, we include in the model four instances of the city-limit constraint, operating on 5, 6, 7 and 8 variables respectively. The number of variables is indicated in the suffix on the constraint name (e.g. $CityLimit_5$ takes 5 variables as arguments).

The second constraint type enforces the solution to meet a specified overall minimum benefit amount. Since the final benefit can only be calculated after all the departments are assigned a city, this requirement is implemented as a global constraint (*MinBenefit* in the model). An abbreviated problem formulation is displayed below, and the complete model is given in the appendix.

$$\begin{aligned} P_{dc} &= \{\langle c_0\, c_1.d_1\, c_1.d_2\, c_1.d_3\, c_2.d_1\, c_2.d_2\, c_2.d_3\, c_2.d_4\rangle \in \{1,2,3,4,5\}^8 : \\ &\qquad \text{CityLimit}_5 \wedge \text{CityLimit}_6 \wedge \text{CityLimit}_7 \wedge \text{CityLimit}_8 \wedge \text{MinBenefit}\} \end{aligned}$$

6.3.2 Basic search strategy

With the variable domains and constraint topology in the problem, the popular control ordering heuristics do not recommend any one particular order. Note that the same topological features will also prevent Forward-Check from providing any improved performance over Backtrack. The control sequence C_{dc} appears to be optimal for solution with simple search algorithms. The *MinBenefit* constraint is expected to be much tighter than the last *CityLimit* constraint, so it is tested earlier.

$$\begin{aligned} C_{dc} &= \langle c_1.d_1\, c_1.d_2\, c_1.d_3\, c_2.d_1\, c_2.d_2\, \text{CityLimit}_5 \\ &\qquad c_2.d_3\, \text{CityLimit}_6\, c_2.d_4\, \text{CityLimit}_7\, c_0\, \text{MinBenefit}\, \text{CityLimit}_8\rangle \\ \text{AVD}(C_{dc}) &= 0 \end{aligned}$$

This problem has a rather small constraint-to-variable ratio with only 5 constraints on 8 variables. The constraints are also of a high arity, the smallest being of arity 4. The *CityLimit* constraints are not very restrictive, so in the sequence, the tighter *MinBenefit* is tested before $CityLimit_8$.

6.3.3 Constraint-set partitioning

This problem has a very interesting topology. There are no artificial dependencies in the problem, for any simple control sequence including C_{dc} (thus giving a combined artificial-variable-dependence of zero). This prevents any of the constraint-set partitioning techniques of Chapter 3 from being of any use here.

6.3.4 Global constraint decomposition

We next turn our attention to the global constraint *MinBenefit*. The first hopeful indication is the presence of a company-based component structure in the corporation modeled in the problem. The *MinBenefit* constraint also reflects this component structure, when viewed as constraining the sum of the benefits of moving each of the corporation's companies. A formulation of the constraint is given below, with its subexpressions arranged to accentuate this component structure. The constant B_{min} in the constraint is the minimum benefit required for the corporation to approve any relocation scheme. (Consult the appendix for a detailed representation of the constraint).

$$\begin{aligned}\text{MinBenefit}(*) \quad = \quad & \text{FinalBenefit}_{c_1}(c_1.*) + \text{FinalBenefit}_{c_2}(c_2.*) \\ & + \text{FinalBenefit}_{c_0}(c_0, c_1.d_1, c_2.d_1) \geq B_{min}\end{aligned}$$

where the functions *FinalBenefit* subtract the inter-department communication costs from the total benefits of moving each department. For example,

$$\begin{aligned}&\text{FinalBenefit}_{c_1}(c_1.*) \\ &= \quad \text{Benefit}_{c_1.d_1}(c_1.d_1) + \text{Benefit}_{c_1.d_2}(c_1.d_2) + \text{Benefit}_{c_1.d_3}(c_1.d_3) \\ &\quad - \text{CommVolume-}c_1.d_1\text{-}c_1.d_2 \times \text{CommCost}(c_1.d_1, c_1.d_2) \\ &\quad - \text{CommVolume-}c_1.d_1\text{-}c_1.d_3 \times \text{CommCost}(c_1.d_1, c_1.d_3) \\ &\quad - \text{CommVolume-}c_1.d_2\text{-}c_1.d_3 \times \text{CommCost}(c_1.d_2, c_1.d_3)\end{aligned}$$

and

$$\begin{aligned}&\text{FinalBenefit}_{c_0}(c_0, c_1.d_1, c_2.d_1) \\ &\quad = \quad \text{Benefit}_{c_0}(c_0)\end{aligned}$$

$$- \text{CommVolume-}c_0\text{-}c_1.d_1 \times \text{CommCost}(c_0, c_1.d_1)$$

$$- \text{CommVolume-}c_0\text{-}c_2.d_1 \times \text{CommCost}(c_0, c_2.d_1)$$

The most suitable abstraction suggested by HiT's heuristics (section 5.3) requires three abstract variables, one each for the three *FinalBenefit* function instances in *MinBenefit*.

$$\begin{aligned} z_0 &= \text{FinalBenefit}_{c_0}(c_0, c_1.d_1, c_2.d_1) \\ z_1 &= \text{FinalBenefit}_{c_1}(c_1.d_1, c_1.d_2, c_1.d_3) \\ z_2 &= \text{FinalBenefit}_{c_2}(c_2.d_1, c_2.d_2, c_2.d_3, c_2.d_4) \end{aligned}$$

Using this decomposition of *MinBenefit* will abstract it into $(z_0+z_1+z_2 \geq B_{min})$. Notice that the abstraction function component for z_0 shares the variables $c_1.d_1$ and $c_2.d_1$ with the abstraction components for the other two abstract variables. Furthermore, notice that none of the other constraints can be abstracted using this (or any other) decomposition of *MinBenefit*.

If we now use any of the solution strategies HSS-1, HSS-2 or HSS-3 (see section 5.3) with this abstraction, we have to pre-compute the domains of the three abstract variables by computing the ranges of the three *FinalBenefit* terms in the global constraint. The sizes of these domains are given below.

$$\begin{aligned} \text{Dom}(z_0) &= 25 \\ \text{Dom}(z_1) &= 89 \\ \text{Dom}(z_2) &= 422 \\ \text{Dom}(z_1\ z_2\ z_3) &= 938,950 \end{aligned}$$

The sharing of base-level variables between the three abstraction functions causes the resulting abstract search space to be much larger than the original problem search space $(938,950 > 5^8 = 390,625)$. This is despite the fact that the the range of each component abstraction function is much smaller than its domain (e.g. $\text{Range}(\text{FinalBenefit}_{c_0}) = 25 \ll \text{Dom}(\text{FinalBenefit}_{c_0}) = 125$), an otherwise desirable property in abstraction functions.

The large abstract search space would make solving the abstract level alone more difficult than the original problem, and is unacceptable. The problem does not have any other constraints that can be re-expressed as abstract constraints on the variables z_i and thus help to prune the abstract search effort. This rules out strategies HSS-1 and HSS-3.

Solution strategy HSS-2 of using a reduced abstract level does not help here either, because of the variable dependence between the three component abstraction functions. If we dropped, say, z_0 to form the reduced abstract level, the corresponding abstracted constraint would now have to be tested on the variables $(z_1, z_2, c_0, c_1.d_1, c_2.d_1)$. As this would be the first constraint in the control sequence, its domain size would be 4,694,750, which is even larger than the size of the unreduced abstract search space!

So now we are left with HSS-4. Hierarchical solution strategy HSS-4, requires that for maximum efficiency the component abstractions be independent of each other. If the component abstractions are not independent, solving the corresponding subproblems will require more than two levels in the bottom-up framework, which could significantly add to the overhead of caching subproblem solutions. The best such arrangement is in the three-level sequence C^0_{dc} shown below.

$$
\begin{aligned}
C^0_{dc} \;=\; & \langle (c_1.d_1\ c_1.d_2\ c_1.d_3\ [z_1 = \text{FinalBenefit}_{c_1}(c_1.*)]) \\
& (c_2.d_1\ c_2.d_2\ c_2.d_3\ c_2.d_4\ [z_2 = \text{FinalBenefit}_{c_2}(c_2.*)]) \\
& (c_0\ c_1.d'_1\ c_2.d'_1\ [z_0 = \text{FinalBenefit}_{c_0}(c_0, c_1.d_1, c_2.d_1)] \\
& \ z'_1\ z'_2\ c_1.d'_2\ c_1.d'_3\ \text{CityLimit}_5 \\
& \ c_2.d'_2\ \text{CityLimit}_6\ c_2.d'_3\ \text{CityLimit}_7\ c_2.d'_4\ \text{CityLimit}_8\ ) \\
& (z''_0\ z''_1\ z''_2\ \text{Abstr-MinBenefit} \\
& \ c_1.d''_1\ c_1.d''_2\ c_1.d''_3\ c_2.d''_1\ c_2.d''_2\ c_2.d''_3\ c_2.d''_4\ )\rangle
\end{aligned}
$$

The third subproblem in C^0_{dc} depends upon variables of the previous two problems. This dependency results in all the problem's variables being recalled to generate a combined solution cache, as the bottom-up framework does not allow the recall of a solution cache to be distributed between a number of subproblems. While C^0_{dc} has intelligently moved all the *CityLimit* constraints into the third subproblem, it still has a residual in which,

in accordance with the HSS-4 strategy, the abstract constraint is tested. C^0_{dc} is plainly exceedingly inefficient.

There is however an alternative — implement a combination of the ideas in strategies HSS-2 and HSS-4. The main idea in hierarchical solution strategy HSS-2 was to reduce the abstract search space overhead by dropping one of the component abstraction functions. If we drop z_0 from the abstraction, the remaining component abstraction functions are now variable independent, and can be used in HSS-4. The resulting control sequence C'_{dc} effectively merges the last two subproblems of C^0_{dc} into a two-level system:

$$\begin{aligned} C'_{dc} \quad = \quad & \langle (c_1.d_1\ c_1.d_2\ c_1.d_3\ [z_1 = \text{FinalBenefit}_{c_1}(c_1.*)]) \\ & (c_2.d_1\ c_2.d_2\ c_2.d_3\ c_2.d_4\ [z_2 = \text{FinalBenefit}_{c_2}(c_2.*)]) \\ & (c_0\ c_1.d'_1\ c_2.d'_1\ z_1\ z_2\ \text{Abstr-MinBenefit} \\ & \ c_1.d'_2\ c_1.d'_3\ \text{CityLimit}_5\ c_2.d'_2\ \text{CityLimit}_6 \\ & \ c_2.d'_3\ \text{CityLimit}_7\ c_2.d'_4\ \text{CityLimit}_8\ )\rangle \end{aligned}$$

where

$$\begin{aligned} & \text{Abstr-MinBenefit}(c_0, c_1.d_1, c_2.d_1, z_1, z_2) \\ & \quad \equiv \quad \text{FinalBenefit}_{c_0}(c_0, c_1.d_1, c_2.d_1) + z_1 + z_2 \ \geq \ B_{min} \end{aligned}$$

We were able to implement this reduced form of HSS-4 for this problem because of the fortunate nature of the interdependency between the three component abstraction functions — only one had to be dropped to make the remaining mutually independent. Performance results with control sequences C_{dc} and C'_{dc} are discussed below.

6.3.5 Performance and analysis

Six sets of experiments were performed for six different values of B_{min}

$$B_{min} = 3000, 3450, 3700, 3850, 4000, 5000$$

Increasing the value of B_{min} decreased the number of solutions from 20,229 to 0 by a factor of about 10 in each step. The difficulty in the problem of finding all solutions

using C_{dc} did not change much with the B_{min}. As the number of solutions decreased, the problem became slightly easier, but quickly leveled off. The difficulty of finding one solution did steadily increase as the number of solutions decreased. The reason for this behaviour is that the problem difficulty is really being controlled by only one constraint — the global *MinBenefit* constraint, which is where the parameter B_{min} is used.

Backtrack was the only basic search algorithm on which the two control sequences were tested. The problem topology would not have benefited from Forward-Check. Test results are shown in the graphs of figure 6.9.

The reduced HSS-4 implementation of global constraint decomposition in C'_{dc} does indeed improve performance. The number of nodes and constraint checks are decreased to about half, and the run-time shows an improvement of a factor of about 4. The larger improvement in run-time is achieved because decomposition of the *MinBenefit* constraint produces abstraction functions with simpler expressions which are easier to compute.

If we examine the control sequence C'_{dc}, we notice that the abstracted constraint *Abstr-MinBenefit* depends upon the variables $(c_0, c_1.d_1, c_2.d_1, z_1, z_2)$. The abstract variables z_1 and z_2 themselves are computed from the original problem variables $c_1.*$ and $c_2.*$. There are no constraints tested in the upper level subproblems where the abstract variables are computed, and in the base level, *Abstr-MinBenefit* is the first constraint to be tested. So what is it then that keeps the difficulty of solving *Abstr-MinBenefit* from overwhelming the cost of using C'_{dc}, in fact making it an easier control sequence than C_{dc} ?

The entire power of C'_{dc} comes from the nature of the abstraction function used. The function $Finalbenefit_{c1}$ abstracts its domain of 125 3-tuples down to a set of 89 unique values, and $Finalbenefit_{c2}$ abstracts its domain down from 625 down to only 422. This gives *Abstr-MinBenefit* a domain size of $5 \times 89 \times 422 = 187,790$. The original constraint *MinBenefit* had a maximum domain size of $5^8 = 390,625$, more than twice as large. And in C_{dc}, it got tested on most of this domain, since the *CityLimit* constraints are very loose. For example, $CityLimit_5$ rejects only one of the 3,125 unique values of its arguments.

The sequence C'_{dc} achieves a large reduction in the number of times the abstracted global constraint is tested, and also a large reduction in the number of times the *CityLimit* constraints are tested, as they are now tested downstream from the *Abstr-MinBenefit* constraint. To achieve this reduction, it incurs an overhead of only $5^3+5^4 =$ 750 constraint checks in the upper level subproblems. The total number of cached solutions is also 750, too small for its caching overhead to affect the overall performance.

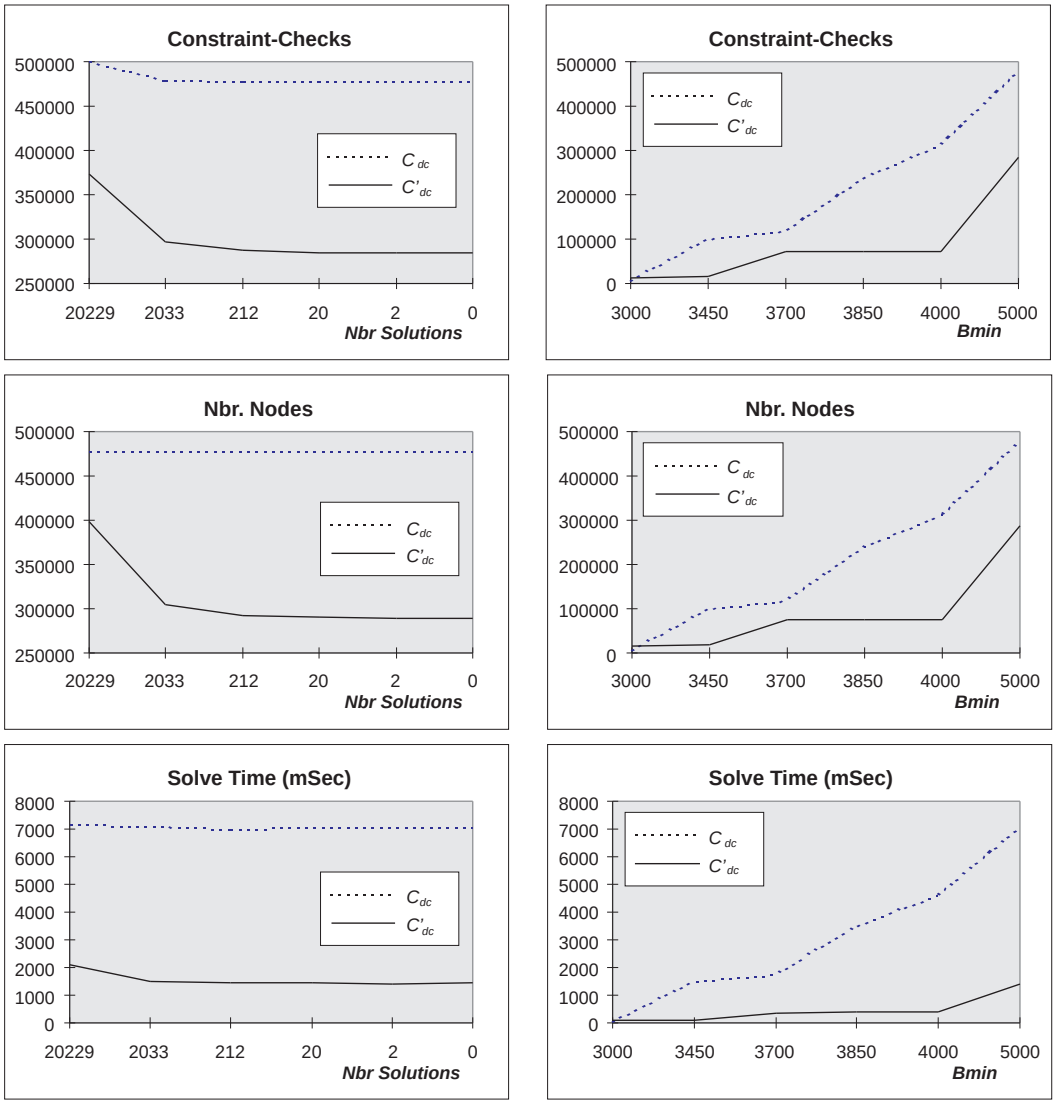
6.4 Corrugated Bulkhead Design

6.4.1 Problem description and model

Bulkheads form the walls of liquid holding internal compartments in tankers. One of the designs used for a vertical transverse bulkhead consists of three vertically stacked corrugated panels, with the corrugations also oriented vertically (figure 6.10). With the heights of the panels determined by configurations of neighboring walls, the remaining design parameters determine the size and shape of the corrugations, and the thicknesses of the three panels. The design goal is to minimize the total weight of the three panels for a specified weight handling capacity.

The problem modeled here is an adaptation from a description published in [1]. The original continuous non-linear optimization problem is posed here as a discrete constraint satisfaction problem (e.g. for a situation where manufacturing can only produce a discrete set of measurements), with the objective function translated into a constraint restricting the weight of the bulkhead to be below a specified value. The domains of the variables have been narrowed for practical experimental considerations. In addition, due to some ambiguity in the original description, it was necessary to adjust the coefficients in some of the constraints in order to ensure a solution. In all other respects, the problem modeled here is syntactically and topologically identical to the one described in the reference.

There are six variables in the problem, with domain sizes ranging from 5 to 100. There are also 13 constraints, of which six are binary, six are quarternary, and one global. An abbreviated problem model is shown below, and presented in more detail in



Looking for all solutions. *Looking for one solution.*

Figure 6.9: Run statistics for P_{dc} using Backtrack.

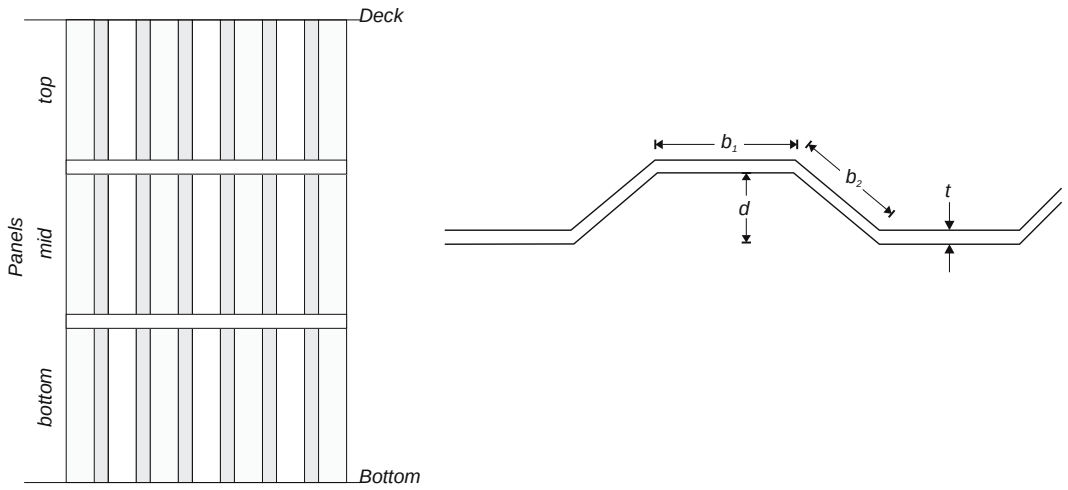


Figure 6.10: Vertical corrugated transverse bulkhead.

the appendix.

$$\begin{aligned} P_{sb} \quad = \quad & \{\langle b_1 b_2 d t_t t_m t_b \rangle : c_1(b_1, b_2, d, t_t, t_m, t_t) \wedge c_2(b_1, b_2, d, t_t) \\ & \wedge c_3(b_1, b_2, d, t_m) \wedge c_4(b_1, b_2, d, t_b) \\ & \wedge c_5(b_1, b_2, d, t_t) \wedge c_6(b_1, b_2, d, t_m) \wedge c_7(b_1, b_2, d, t_b) \\ & \wedge c_8(b_1, t_t) \wedge c_9(b_2, t_t) \wedge c_{10}(b_1, t_m) \wedge c_{11}(b_2, t_m) \\ & \wedge c_{12}(b_1, t_b) \wedge c_{13}(b_2, t_b)\} \end{aligned}$$

where

$$\begin{aligned} \text{Flange width } b_1 \quad & \in \quad \{55.0, 55.1, \ldots, 64.9\} \\ \text{Web length } b_2 \quad & \in \quad \{55.0, 55.1, \ldots, 64.9\} \\ \text{Corrugation depth } d \quad & \in \quad \{35.0, 35.1, \ldots, 39.9\} \\ \text{Top panel thickness } t_t \quad & \in \quad \{1.00, 1.05, \ldots, 1.20\} \\ \text{Middle panel thickness } t_m \quad & \in \quad \{1.00, 1.05, \ldots, 1.20\} \\ \text{Bottom panel thickness } t_b \quad & \in \quad \{1.00, 1.05, \ldots, 1.20\} \end{aligned}$$

6.4.2 Basic search strategy

The control sequence C_{sb} chosen for Backtrack and Forward-Check minimizes the total constraint depth and bandwidth:

$$\begin{aligned} C_{sb} &= \langle t_t\, b_1\, c_8\, b_2\, c_9\, t_m\, c_{10}\, c_{11}\, t_b\, c_{12}\, c_{13}\, d\, c_2\, c_5\, c_3\, c_6\, c_4\, c_7\, c_1 \rangle \\ \mathrm{AVD}(C_{sb}) &= 0+1+2+2+3+3+2+2+2+2+2+2+0 \\ &= 23 \\ \mathrm{AVD}_{avg}(C_{sb}) &= 23/13 \;=\; 1.77 \end{aligned}$$

The problem has a relatively high constraint to variable ratio of 2.17. Since more than half the constraints (7) have an arity of greater than half the number of variables (arity ≥ 3) in the problem, most of the constraints get tested relatively deep in the control sequence. The hardness of the problem is controlled by only six constraints ($c_8 \ldots c_{13}$), less than half the total number of constraints.

6.4.3 Constraint-set partitioning

The control sequence C_{sb} has, at 23, a fairly high total artificial variable dependence (AVD) but, because of the relatively large number of constraints, the average artificial variable dependence of each constraint is a moderate 1.77. The highest AVD of 3 occurs at constraints c_{12} and c_{13}, tested after the fourth variable t_b in the sequence.

For DOI-decomposition, the sequence C_{sb} can be cut at four different places, based on the occurrence of constraints before the third, fourth, fifth and sixth variables. The AVD evaluations for each cut are shown in the table in figure 6.11. The fifth column of the table gives the largest reduction in artificial variable dependence among all constraints affected by the decomposition.

The third and fourth cuts appear to be the most suitable DOI-decompositions based on the heuristic artificial variable dependence measures. The fourth cut has the highest Total-AVD (7), and the second largest Average-AVD (0.78). The third cut has the second highest Total-AVD (6) and the largest Average-AVD (0.86), and it affects more constraints than the fourth cut.

Evaluation of possible DOI-decompositions.

Seq	C_{sb} Cut Before	Prefix #Vars	Suffix #Constrs	*AVD(suffix,prefix)* Average	Total	Highest
$C_{\mathrm{DOI}(sb)_1}$	b_2	2	12	0.08	1	1
$C_{\mathrm{DOI}(sb)_2}$	t_m	3	11	0.45	5	2
$C_{\mathrm{DOI}(sb)_3}$	t_b	4	9	0.78	7	3
$C_{\mathrm{DOI}(sb)_4}$	d	5	7	0.86	6	2

Evaluation of possible IS-decompositions.

Seq	Indep-set Constraints	*AVD(IS,C_{sb})* Average	Total	Highest
$C_{\mathrm{ISB}(sb)}$	c_{11}, c_{12}	2.50	5	3
$C_{\mathrm{ISF}(sb)}$	c_9, c_{10}	1.50	3	2

Figure 6.11: Evaluation of possible constraint-set decompositions of C_{sb}.

Both IS-decomposition algorithms, IS-Back and IS-Forward, produce independent sets composed of two binary constraints. As is usually the case, IS-Back selects its constraints from deeper levels in the control sequence, with consequently higher artificial variable dependences. In such situations, the IS-Back decomposition should be more suitable when the problem is particularly hard, or when looking for all solutions. Conversely, the IS-Forward decomposition should be more suitable when looking for just one solution in a problem of moderate difficulty.

Between the two types of decompositions, $C_{\mathrm{DOI}(sb)_3}$ achieves a significantly higher AVD reduction of 7, compared to 5 achieved by $C_{\mathrm{ISB}(sb)}$, and also affects some deeper constraints in C_{sb}, so it should be a better choice.

6.4.4 Global constraint decomposition

The problem has a single global constraint, c_1, whose formulation is shown below. To better demonstrate the constraint decomposition process, the representation has been arranged to partition the variables between subexpressions as much as possible.

$$c_1(*) \quad = \quad (b_1 + b_2) \times (495t_t + 385t_m + 315t_b) < 1400 \times (b_1 + \sqrt{b_2^2 - d^2})$$

The only decomposition of this constraint into non-trivial subexpressions with a

minimum overlap of variables is into the following components,

$$\begin{aligned} z_1 &= b_1 + b_2 \\ z_2 &= 495t_t + 385t_m + 315t_b \\ z_3 &= b_1 + \sqrt{b_2^2 - d^2} \end{aligned}$$

yielding the abstract constraint ($z_1 z_2 / z_3 < 1400$). A quick analysis proves this to be a very unsuitable abstraction for the following reasons:

1. The size of the abstract search space is actually larger than the original search space ($\mathrm{Dom}(z_1 z_2 z_3) \approx 200 \times 125 \times 5000 = 125,000,000$, compared to 62,500,000). This is a result of both the sizes of ranges of the particular expressions and the overlap of variables between two of the expressions.

2. None of the other constraints can be abstracted using this decomposition, and so they cannot be used to reduce search in the abstract level.

3. Dropping one of the component abstractions for use with solution strategy HSS-2 or HSS-4 (as in the case of the decentralization problem) also does not help here because the size of the abstracted constraint's domain in the reduced hierarchical problem is still very large (unlike, e.g., in the case of the corporation decentralization problem).

Other nontrivial decompositions of the global constraint c_1 show even worse degrees of variable overlap. The difficulty in finding a suitable abstraction comes in a large part from the problem having a small number of variables, and the domain not having a component structure that could be reflected in the formulation of the global constraint. Such a component structure was found in the other case studies, and we selected decompositions that separated out portions of the constraint using variables from different components.

6.4.5 Results and discussion

The various control sequences for the problem were tested using both Backtrack and Forward-Check, and the performance statistics are shown in figure 6.12. Problem P_{sb}

has 1,382 solutions, and it is fairly difficult to find all of them. Backtrack, for example, examined almost 13 million of a total possible 64,238,290 nodes in C_{sb}'s search tree. Forward-Check was more effective, requiring about 28 million fewer constraint checks than Backtrack. Domain value ordering in this problem was found to have very little effect on the effort required to locate the first solution.

Control Seq	Total Cached Solns	All (1,382) Solutions						One Solution		
		Backtrack			Forward-Check			Backtrack		
		Tests	Nodes	Time (S)	Tests	Nodes	Time (S)	Tests	Nodes	Time (S)
C_{sb}	0	67,409,712	12,749,755	851	39,597,156	16,017,972	528	72,365	14,869	930
$C_{DOI(sb)_1}$	500	67,360,212	12,702,855	850	50,328,693	16,078,009	668	21,476	4,585	280
$C_{DOI(sb)_2}$	50,000	66,007,328	14,120,660	860	34,064,273	13,128,335	495	129,480	67,859	3,950
$C_{DOI(sb)_3}$	182,050	48,892,574	12,708,044	631	20,338,422	15,230,108	320	610,343	327,089	12,570
$C_{DOI(sb)_4}$	230,780	41,065,261	17,147,137	524	17,909,040	16,819,440	342	7,941,495	3,937,964	97,410
$C_{ISB(sb)}$	577	66,214,832	12,085,740	848	39,412,236	15,877,067	531	72,791	15,687	940
$C_{ISF(sb)}$	908	67,110,712	12,656,805	850	50,330,693	16,080,416	668	15,493	4,004	210

Figure 6.12: Run statistics for P_{sb} and P_{sbh}.

Our earlier analysis picked $C_{\mathrm{DOI}(sb)_3}$ to be the overall winner, predicting it to offer better performance than both $C_{\mathrm{DOI}(sb)_4}$ and both the IS-decompositions. Actual experiments reveal that while the DOI-decomposition $C_{\mathrm{DOI}(sb)_3}$ does indeed improve upon C_{sb}'s performance and provide better performance than the IS-decompositions, the overall winner when looking for all solutions is actually the DOI-decomposition with the deepest cut and largest number of cached subproblem solutions. This is true because the original problem, in C_{sb}, was hard enough that most of the redundant testing occurred in the deepest level constraints. Sequence $C_{\mathrm{DOI}(sb)_4}$ produces a greater AVD reduction (of 2 each) at constraints c_2 and c_5 tested after the last variable in C_{sb} than $C_{\mathrm{DOI}(sb)_3}$.

The better performance of the IS-Forward decomposition $C_{\mathrm{ISF}(sb)}$ when looking for one solution does follow earlier prediction. The performance of DOI-decompositions when looking for one solution steadily decreases with the length of the prefix on which the decomposition was based.

6.5 Lessons and Conclusions

The experiments in applying decomposition techniques to solve the problems in this chapter and those of chapter 4 are useful for extracting some general guidelines.

6.5.1 Constraint-set decomposition

The case studies in this chapter demonstrate the power of DOI-decompositions in improving performance, and the utility of the artificial variable dependence (AVD) heuristic measure. DOI-decompositions are more effective than IS-decompositions in reducing artificial dependencies in constraints at deeper levels in the search tree where, in hard problems, most of the cost is incurred. This makes it important, when selecting between alternative decompositions, to consider not only the total reduction in artificial dependencies, as measured by the AVD heuristic, but also the depth in the original control sequence at which these reductions are being achieved.

In the sizes of the problems encountered here, the cost of building the solution cache tree was not a significant overhead. In fact it was the cost of generating values from the solution cache which appeared to dominate the bottom-up solution portion of the cost of solving a multi-level control sequence. Acknowledging that there should be some scope for improvement in the implementation of the bottom-up framework, we may note that smaller solution caches will also lead to smaller cache generate events.

6.5.2 Global constraint decomposition

Whenever applicable, a carefully selected abstraction combined with the bottom-up solution strategy of HSS-4 offers the greatest potential in improving performance. In general, the most important factor for a beneficial application of global constraint decomposition appears to be the ability to select an abstraction function that produces a small abstract search space, with a minimum (preferably none) number of base level variables common to the abstraction function components. This is not surprising, and has already been mentioned in section 5.3. The main new heuristic that emerges from the example problems here is the desirability of a component structure in the domain,

and in the syntactic formulation of the global constraint. Such was the case in the floorplanning and decentralization problems, and a component structure was lacking in the bulkhead design problem.

6.6 Summary

We examined in this chapter how the particular topological structure of a problem affects the choice of a decomposition technique. The three problems studied here represented three different possibilities:

- The *floorplanning* domain displayed a strong component structure, which was reflected in one of the global constraints. It also exhibited a propinsity for high degrees of artificial serial dependence in its control sequences. As a result, the problem benefited from all three techniques — DOI-decomposition, IS-decomposition and global constraint decomposition.

- The *corrugated bulkhead design* problem had only 6 variables, but their domains were large. This permitted DOI-decomposition to take advantage of the artificial dependencies inherent in the problem, but the other two techniques failed to be of any use.

- The *corporation decentralization* problem did not exhibit any artificial dependencies in its control sequence. The global constraint did display a component structure in its global constraint, however that decomposition did not produce independent component abstractions. A blend of hierarchical solution strategies HSS-2 and HSS-4 was successful in handling this situation.

Chapter 7

Discussion and Related Work

This chapter evaluates the extent to which this research supports the claims of this thesis, summarizes the major contributions, places this work in the context of related research, and suggests directions for extending this research.

7.1 Evaluation of Thesis Claims

7.1.1 Problem decomposition and Bottom-Up solution

The motivation for looking at decomposition techniques for solving constraint satisfaction problems came from noticing that

- redundant constraint checks form a significant portion of the cost of solving many problems, and that
- these redundancies are a direct result of artificial dependencies set up by the serial nature of the combinatorial enumeration used as the basis for search algorithms.

◇ *Claim 1.1*: **Decomposition improves performance**

In the first part of the thesis we demonstrated how applying DOI and IS decomposition to a simple control sequence could reduce the degree of its artificial serial dependence. The experiments of Chapters 4 and 6 confirmed that this reduction led to an improvement in performance.

The utility of both DOI and IS decomposition techniques was demonstrated in Chapter 4 in experiments on sets of randomly generated 'n-ary' CSPs. Measuring performance in the number of constraint checks, we saw that these decomposition techniques were particularly useful when searching for all solutions to a problem, and their utility

increased with the basic difficulty of the underlying problem, and with the amount of reduction in artificial variable dependence achieved by the decomposition. In general, DOI-decomposition was seen to yield the most consistent performance improvements.

The experiments of Chapter 4 did show that when searching for one solution, the performance of the decomposed multi-level control sequences improved in problems of increasing difficulty. However these experiments were not able to generate problems of high difficulty, and we saw decomposition produce an actual improvement in performance over basic search in only some of the more difficult problems generated.

We next demonstrated the utility of problem decomposition in some application case studies in Chapter 6. Performance improvement here was measured both in the number of constraint checks and in run time. These experiments demonstrated the existence of some interesting problems where problem decomposition and bottom-up solution can be usefully applied. They also provided case studies of how to select between and apply decompositions on a given problem.

In chapter 4, we also tested the efficiency of the Bottom-Up solution framework, by estimating the time needed to build a solution cache structure, and generate values from the cache. Together with the analysis of Chapter 3, we observed that both cache building and cache generating can become significant factors affecting run-time performance of a decomposed problem. Chapter 3 also described an alternative cache data structure which greatly reduces the cache build time. Run-time measurements in experiments in Chapters 4 and 6 showed that significant run-time performance improvement can be obtained for problems in addition to reducing redundant constraint checks.

The storage cost of caching solutions to subproblems in a decomposition based solution strategy can sometimes be of concern. When the subproblems are large, and have a large number of solutions, the memory requirements can exceed a computer's RAM, causing disk swapping which is slow. Such judgements have to be made by the user before deciding upon a decomposition strategy. Section 3.6 discusses these issues, and suggests some strategies for tackling very large CSPs.

◇ *Claim 1.2*: **Decomposition is applicable to a wide and interesting range of problems**

The random problem and application case study experiments described earlier in the thesis demonstrate that, unlike many other advanced constraint network processing techniques, the IS and DOI decomposition procedures are not limited in their applicability to binary CSPs. This is very significant, because most applications are not easily modeled in only binary constraints. We also demonstrated that in the more difficult problems, where the need for performance improvement is greater, these decomposition techniques actually perform better.

The run-time experiments described in Chapters 4 and 6 also showed that decomposition could achieve significant performance improvement both for problems whose constraints were expensive to test and for problems with constraints that are relatively cheap to test, when compared with the average cost of a generate action.

◇ *Claim 1.3*: **The decomposition techniques are automatable and fast**

Both DOI and IS decomposition procedures were implemented in Lisp, and tests performed on the problems of Chapter 6 showed that run times on a SPARC 5 were better than 50 mSec. This was faster than the time needed to actually solve the problems by several orders of magnitude. This will generally be true, as the cost of producing a decomposition depends upon the syntactic size of the problem (number of variables and constraints).

The decomposition procedures produce a set of alternative decompositions for each problem. While simple heuristics can be used to automatically select a suitable candidate, the best results will be obtained by a more detailed analysis of the heuristic measures, of the type used in the case studies of Chapter 6. These more complex decision procedures can also probably be implemented into a type of "expert system". The current implementation does not extend to this level.

7.1.2 Global constraint decomposition

We discussed in Chapters 2 and 5 that CSP solution algorithms are unable to use global constraints to prune search. This led to the work on global constraint decomposition, where the central idea is to decompose a global constraint into a conjunction of smaller arity constraints. This decomposition reformulates the original problem by adding

an abstraction level. Even though this increases the size of the overall search space, hierarchical solution procedures that take advantage of the hierarchical structure in the reformulated problem can produce improved performance on a suitably chosen constraint decomposition.

◇ *Claim 2.1*: **Global constraint decomposition is useful**

The utility of global constraint decomposition was demonstrated on two application case studies in Chapter 6. The floorplanning problem yielded dramatic results, with performance improvements of up to two orders of magnitude. The performance improvement in the second corporate decentralization problem was smaller in proportion, but equally significant because of the unwieldy topological structure of the problem's constraint network.

◇ *Claim 2.2*: **The constraint decomposition procedure is automatable and fast**

The global constraint decomposition procedure has been implemented in a Lisp program called HiT. The program will take a problem and a global constraint, and automatically produce a decomposition and reformulated hierarchical problem using the heuristics described in Chapter 5. The speed of HiT is comparable to that of the DOI and IS decomposition procedures.

◇ *Claim 2.3*: **Global constraint decomposition is applicable to some interesting problems**

Global constraint decomposition is a syntactic reformulation of the problem, produced by processing an intensional representation of a problem constraint. The hierarchical transformation will not benefit all problems. The situations where it does benefit are partially encoded in the decomposition heuristics used in HiT, and discussed in more detail in Chapters 5 and 6.

However, this technique is aimed at providing a type of benefit not provided by any other technique, and when applicable, its benefit can be quite dramatic, as in the two cases in Chapter 6. Such problems will usually model a domain displaying an inherent component structure.

7.2 Contributions

The contributions made by this research to the field of problem solving can be described at two levels. At the most practical level, the thesis describes some new problem solving procedures for constraint satisfaction problems that offer better performance than currently popular methods. At a more theoretical level, this research extends the understanding of the benefits of problem decomposition.

Some of the specific major contributions of this research are:

- An analysis of the reasons for redundant testing of constraints during problem solving, and identification of the culprit — serial combinatorial enumeration of the search space sets up artificial serial dependencies between constraints and independent variables. (Chapters 2 and 3).

- A demonstration of how problem decomposition can be used to interrupt and remove artificial serial dependencies. (Chapter 3).

- A bottom-up solution framework, that can be used with any problem decomposition scheme, for efficiently resolving interactions and combining the solutions to different subproblems to produce solutions to the composite problem. (Chapter 3).

- Two new types of problem decompositions, DOI and IS decomposition, that are applicable to CSPs, and aimed specifically at reducing redundant constraint checks. Efficient procedures for producing these types of decompositions. (Chapter 3).

- A constraint decomposition and reformulation procedure that allows search algorithms to take advantage of global constraints to prune search. (Chapter 5).

- New hierarchical search strategies that use decomposition to take advantage of the hierarchical structure in a CSP. (Chapter 5).

7.3 Related Work

Several areas of research on solving constraint satisfaction problems were examined in detail in Chapter 2, when setting the motivation for this thesis. This section summarizes those discussions, and describes some other work.

The main motivations for this thesis were the occurrence of redundant constraint tests as an inefficiency in all the major basic search algorithms, and the inability of these algorithms to take advantage of global constraints to reduce search. We will examine other developments in the field within this context.

7.3.1 Basic search algorithms for CSPs

Backmarking ([25, 31]) was perhaps the first search algorithm proposed for CSPs that avoids retesting constraints on previously tested argument bindings. It keeps track of which variables have not had their values changed by a backtrack action, and will not retest constraints that depend only on those variables. Once a variable's binding changes, the rebinding of a variable to an old value by further backtracking is not detected. While it does improve performance over plain Backtrack, Forward-Check (ibid.) has generally been found to be more efficient (this was supported by tests made during this research) so it was not included in the experiments for this thesis.

The efficiency of Forward-Check comes from using look-ahead to detect future conflicts, and thus avoiding further extension of those search paths. Backtracking action is just like Backtrack, and no attempt is made to avoid redundant tests.

Intelligent Backtracking techniques, including selective backtracking and dynamic backtracking ([64, 4, 26, 62]), generally try to avoid retesting variable bindings previously found to have resulted in a conflict. While this can be extended to avoiding retesting of successful bindings, the limiting factor is the history recording mechanism (see Chapter 2).

The solution cache tree in the bottom-up solution procedure proposed in Chapter 3 can be seen as an efficient history data structure. This bottom-up framework was successfully implemented on top of both Backtrack and Forward-Check. It can be

easily extended to other algorithms that follow a static control order.

7.3.2 Control ordering techniques

Several of these were also discussed in Chapter 2. Current ordering heuristics based upon constraint topology ([64, 19, 71]) tend to select an ordering with an optimal control sequence property (e.g. width) whose value is bounded from below by the highest constraint arity in the problem. In problems with global constraints, every control sequence will have the same value for these properties. Some alternative heuristics are suggested in [47] and used in this thesis.

7.3.3 Network consistency techniques

Network consistency techniques preprocess a constraint network and remove values from variables' domains that are found to lead to a conflict in a small subset of the problem. Arc consistency ([40]) is the most popular, and uses a single constraint as the problem subset. A value can be removed from a variable's domain if there is a constraint on that variable that cannot be satisfied with the variable bound to that value.

It is very easy to perform this test for binary constraints. A candidate value for elimination is tested against all values for its co-dependent variable for the given binary constraint. If any one of these tests fails, the candidate value gets removed from its variable's domain. The cost of performing arc-consistency on a constraint, however, depends exponentially on its arity. Unfortunately, this cost rises very rapidly as the arity of the constraint increases.

Solving a set of independent constraints before tackling the residual subproblem in IS-decomposition is very similar to arc consistency. The main difference between them is in how the constraints are used in what might be called the preprocessing step. Arc consistency simply reduces the domain of each variable, so that each value in the filtered domain will belong to some solution of the processed constraints. In IS-decomposition, the actual tuples that solve each of the independent constraints are stored. This is usually more efficient for finding all solutions, because after arc consistency filters the variables' domains, their cross-product might still contain values that

violate conjunctions of constraints. Where arc consistency is particularly suitable for problems requiring one solution [18, 19, 10] IS-decomposition might be viewed as an efficient adaptation of arc consistency for problems requiring all solutions.

7.3.4 Trees of variable clusters

In [18] Freuder showed that after performing arc consistency, finding the first solution for a problem with a tree-shaped constraint network does not require any backtracking. This has led to decomposition techniques that divide a CSP into subproblems (variable clusters) connected by a tree-shaped residual subproblem [11, 28].

The decomposition is solved by first solving the subproblems, saving their solutions, and then performing directional arc-consistency on the tree-shaped residual, treating each cluster-subproblem as a single variable. The first solution for the problem can now be found without any further backtracking. The problem with this approach is twofold. First, in the worst case, the cost of solving each subproblem is exponential in the number of its variables. Second, in a problem with a high arity constraint, it must either go into the residual (making residual consistency expensive) or in one of the clusters. If the constraint is global, this approach completely fails to produce useful decompositions. Tree clustering also fails to produce any decompositions for constraint networks that are complete graphs.

The Adaptive Consistency algorithm described in [10] is shown to induce the same clusters produced by the triangulation scheme of [11]. Both algorithms take an 'm-ordering' of variables as argument. Given the same variable ordering, Adaptive Consistency coordinates the solution of the subproblems produced by Tree Clustering by enforcing local consistency between them while it is being executed. Its efficiency comes from merely performing local consistency within each subproblem instead of solving it. The resulting network can now produce one solution without any further backtracks.

Adaptive Consistency is not suited for finding all solutions for the same reasons as arc consistency, as discussed in the previous subsection. Tree clustering is also very inefficient. However, these inefficiencies are removed by applying Bottom-Up Backtrack to the subproblems produced by Tree Clustering.

Consider again the problem of figure 2.2. The variable ordering in C_1 is an m-ordering:

$$C_1 \text{ variables } = \langle X_4 X_5 X_3 X_1 X_2 \rangle$$

Tree clustering on C_1 produces the clusters $(X_4 X_2)$, $(X_4 X_3 X_1)$, and $(X_5 X_4 X_3)$. Since these clusters are not variable-independent, the order in which they are solved is important. The following is an efficient three-level control sequence:

$$\begin{aligned} C_{TC} &= \langle (X_2 X_4 T_{24})(X_1 X_3 T_{13} X_4' T_{14} X_2')(X_3'' X_5 T_{35} X_4'' T_{45} X_2'' X_1'') \rangle \\ \text{Checks}(C_{TC}) &= 16 + 16 + 27 + 12 + 27 \;=\; 98 \end{aligned}$$

This decomposition turns out to be more efficient than the DOI and IS-decompositions of this problem.

The subproblems produced by Tree Clustering will tend to have a high degree of independence, since each subproblem will tend to cluster dependent components together. However, the larger each clique, the larger the cost of solving the corresponding subproblem. If the constraint graph for a problem is complete, as in the case of a global constraint, Tree Clustering will combine the whole problem into one cluster, whereas both the decomposition strategies proposed here will produce a usable decomposition.

The Cyclic-Clustering algorithm described in ([35]) combines the ideas of cycle-cutset ([9]) and tree-clustering, but fails on problems that induce complete constraint graphs for the same reasons.

7.3.5 Cross-product representation of the search space

A set of partial solution tuples can often be efficiently represented as a smaller set of cross-products, where each cross-product is represented as a set of values for each participating variable. For example, the set $\{\langle a\, c\rangle, \langle a\, d\rangle, \langle b\, c\rangle, \langle b\, d\rangle\}$ can be efficiently represented as the cross-product $\{a, b\} \times \{c, d\}$. This representation can be more efficient than the solution trees described in Chapter 4, particularly when the tuples being represented cover a large proportion of the corresponding (partial) search space. The cross-product representation (CPR) and its use in modified versions of the Backtrack (BT-CPR) and Forward-Check (FC-CPR) algorithms is described in [33, 32].

The BT-CPR algorithm, for example, maintains a set of cross-products, representing the valid partial solutions found so far, that are repeatedly extended, trimmed and merged until all the variables are covered and all non-solutions eliminated. At each iteration, BT-CPR extends a cross-product from this set by adding to it a new variable. A new extension is created for each value in the new variable's domain. Non-solutions are then eliminated from each extended cross-product by checking for consistency of the old variables' values against the new variable's value. Inconsistent values are discarded, and the entire cross-product is discarded if one of the variables loses all its values. All constraints on the new variable are tested, as needed, on the singleton value of the new variable and the values of the old variables. A trimmed cross-product gets merged with another if they differ on the values of only one variable.

The elimination of values in the trim step is exactly like that in arc consistency procedures, and BT-CPR can be viewed as performing directional arc consistency on incrementally larger prefix subproblems of the given CSP. The difference is that BT-CPR further decomposes the CSP on its domains as defined by each cross-product, and applies consistency to each decomposition separately. The way the domain decompositions are created causes this incremental consistency application to actually yield solutions after the final iteration.

The power of CPR comes from the fact that member tuples of any subset of variables in a cross-product can be generated without paying any attention to other variables in that cross-product. This eliminates artificial serial dependence, and therefore redundant testing of (binary) constraints within a cross-product. However a particular tuple may occur in more than one cross-product, so redundant testing is not completely eliminated from the overall search procedure. The purpose of the merge step mentioned above is exactly to reduce the number of cross-products, and its effect on reducing redundant testing increases when problems get harder.

Higher arity constraints do create a problem in the trim step. When a cross-product extension makes a higher arity constraint active, old variables have to be tested against other old variables in combination with the new variable. This implies that unless the result of each constraint test is somehow saved (at potentially prohibitive cost in space),

each constraint of arity $n > 2$ will have to be tested $n - 1$ times on each tuple in its search space within the cross-product, in order to perform inconsistent value elimination on all the old variables dependent upon the new variable through that constraint. This introduces redundant constraint checks, although in this case the cause is not artificial serial dependence. The possibility of two higher arity constraints sharing some old variables in their argument set further increases the redundant generation of values. An actual comparison of the costs of the CPR and Bottom-Up algorithms and how they vary with problem hardness is an interesting direction for further study.

The solution tree structure of Chapter 4 also achieves a degree of "merging", in this case of prefixes of the cached solution tuples in its tree representation, thus requiring less space than simply storing each tuple separartely. However, in general it is a less efficient storage mechanism than the cross-product representation.

7.3.6 Other decomposition techniques

A stable set is a set of nodes not directly interconnected by an arc in a graph. The nodes of every graph can be partitioned into stable sets, which can be organized to form the levels of a 'pseudo-tree'. Using the intuition that variables in a stable set can be bound independently, Freuder ([22]) describes an improvement on basic Backtrack for finding one solution that takes advantage of a partitioning of the CSP into stable-sets of variables. The worst-case cost of this algorithm is d^p, where p is the number of levels in the pseudo-tree. This is then used as a motivation for a decomposition procedure which produces subproblems interconnected by a residual graph with a small number of stable sets.

As in the tree-clustering techniques, the limiting factor in stable-set based processing of a CSP is the size of the largest clique in the (primal) constraint graph. A pseudo-tree will have at least that many levels. Both DOI and IS-decompositions, on the other hand, have no trouble producing non-trivial decompositions in complete constraint graphs, or in problems with global constraints so long as there are other smaller arity constraints. These too are, however, limited into producing subproblems whose size can be no smaller than the smallest constraint arity in the problem.

The fundamental motivation for both tree-clustering and stable-set based decomposition is to reduce the number of variables for which a potentially complete cross-product of values needs to be examined to solve the problem. The motivation for DOI and IS-decompositions is to reduce artificial serial dependence of constraints in order to reduce redundant testing of those constraints.

The decomposition procedures described in Chapter 3 are "complete" in that the decomposition preserves a CSPs solution set. The improved performance is obtained by reducing the number of redundant constraint checks. Decompositions that do not preserve completeness can also be used to reduce overall search effort. An example is described in ([21]), where an inferred disjunctive constraint is used to decompose the CSP into a set of disjunctive subproblems such that if the original problem has a solution, then so does one of the subproblems. Problem solving performance is improved by reducing the combination of domain values tested for variables involved in the disjunctive constraint. Such decomposition techniques are useful when any single solution to the CSP will suffice.

7.3.7 Repeated substructures

We noted in Chapter 3 that the bottom-up solution framework can be extended to take advantage of repeated subgraphs in a constraint network, by first solving the subgraph, and then reusing its solutions at each point an instance of the subgraph occurred in the problem. This could be applied, for example, to the 4-room floorplanning problem of Chapter 6, where the subproblem of generating a valid room occurs four times in the problem.

HiT extends this approach to examining the intensional expressions of constraints for occurrence of abstraction function components. Such constraints can be reformulated by substituting in appropriate abstract variables, thus changing the scope of the constraint, and also avoiding recomputation of that subexpression.

Taking advantage of common subexpressions is a well known technique for optimizing relational database queries. It has been applied within a single query (e.g. [29]) and across groups of multiple queries (e.g. [34, 61]). The motivation there is to reduce

repeated access to data stored on slow media. This is analogous to the motivation in CSPs of reducing the computational cost of testing constraints.

7.3.8 Constraint Relaxation

Montanari and Rossi ([50]) generalize constraint network processing techniques into the notion of *constraint relaxation*. A CSP is solved by a series of applications of relaxation rules. If a decomposition strategy is used, then each relaxation step solves one subproblem. They introduce the notion of *perfect relaxation*, where each relaxation rule needs to be applied only once in a solution strategy. This is usually the case when solving by decomposition, and both the DOI and IS decomposition procedures produce perfect relaxation strategies.

The authors also demonstrate how the productions in a context-free graph grammar can be mapped to relaxation rules in a perfect solution strategy for (constraint-) graphs in the grammar's language. The cost of using such a solution strategy increases linearly in the number of production rule applications needed to derive the constraint graph. The constant factor in this linear bound depends on the size of the largest production, which ultimately depends exponentially upon the highest constraint arity in the network. While this is not surprising, the result is useful in associating complexity bounds with classes of CSPs. The authors do not directly address the issue of constructing an optimal or good strategy for solving a given CSP.

In this thesis, we have taken the reverse approach, where a CSP is given to be solved without any knowledge about how it was produced. The IS and DOI decomposition procedures are two methods for constructing useful perfect relaxation strategies, aimed directly at reducing redundant constraint checks. The bottom-up solution framework can be used as an efficient framework for implementing a perfect decompositional relaxation strategy, another point not addressed in the above paper.

7.3.9 Abstraction levels

GPS ([54]) was probably the first AI problem solver to make explicit use of an abstraction level. Sacerdoti ([60]) further developed the idea into a hierarchy of abstraction

spaces in the system ABSTRIPS. The abstraction levels produced by HiT are very similar to these earlier forms.

The general idea of using abstraction levels in constraint satisfaction problems was previously developed in [45, 48, 46, 68]. The research in this thesis builds upon this work, and makes the motivation of seeking search-pruning benefit from global constraints more explicit. Two of the hierarchical solution strategies described here, HSS-2 and HSS-4, are also new.

Ellman describes a more general formulation of the use of abstraction levels in solving constraint satisfaction problems in [14, 15]. In Ellman's terms, the procedure HiT constructs a hierarchical CSP by defining a "necessary approximate symmetry" on the original problem's search space. This necessary approximation is defined by the abstraction function and the hierarchical completeness condition, requiring that if a tuple of base-level variable bindings is a solution to the original set of problem constraints R, then its abstraction is a solution to the abstract constraint Q. This is the same as the "upward solution property" of a hierarchical system as defined by Knoblock for planning domains in [37].

7.3.10 Hierarchical solution in CSPs

Several application domains exhibit a component based hierarchical structure. This structure manifests itself in a CSP model in several different ways. In the floorplanning domain of Chapter 6, the components are rooms, and each room is represented by four variables in the CSP. There are four rooms in the problem, and this component structure is reflected in the intensional expressions of the constraints. HiT's global constraint decomposition was able to take advantage of this syntactic structure of the global constraint in constructing a useful abstraction for the problem.

An alternative approach to the modelling and use of component structures in a CSP is described in [44]. This paper advocates the explication of the hidden structure of compound objects into tuples of variables, each representing a primitive component. The variable domains are also expressed in sets of corresponding value-tuples representing predefined component configurations. The size of such a set will be much

smaller than simply taking the cartesian product of all the possible component values, yet a least-commitment search algorithm can take advantage of the tuple substructure by making partial commitments to component variable values. The paper also describes a least-commitment approach that takes advantage of a taxonomic organization of components. This is very similar to the least-commitment search used in MOLGEN ([66, 65]).

A third approach to exploiting hierarchical structure in a problem domain is described in [41]. The Hierarchical Arc Consistency (HAC) algorithm takes advantage of a taxonomic organization of the domain of each variable induced by inherent structure in the application. The algorithm becomes efficient when the taxonomic structure partitions each variable's domain into subsets that "share consistency properties" allowing them to be treated as a unit. The larger such subsets, the more efficient the taxonomy for HAC. Another similar approach to using domain taxonomies is described in ([5, 6]).

While these hierarchical and least-commitment based approaches also use abstractions to reduce search, the source of the abstractions is the application domain itself. Global constraint decomposition uses a syntactic procedure for abstraction based upon the problem formulation, and it is aimed directly at global constraints.

7.3.11 Automated development of problem solving programs

The research described in this thesis was done within a larger program on Knowledge-Based Software Development at Rutgers ([67, 68]). All the decomposition algorithms described in this research have been automated, and are capable of taking a declarative representation of a CSP and producing a multi-level control sequence implementing either a DOI or IS decomposition, or producing a control sequence and a new hierarchical problem that decomposes a global constraint.

There has been other work on automated programming in the field of CSPs, e.g. [2] describing techniques for problem reformulation, [69] describing a solution-repair approach, and [42] describing the automated formulation of search heuristics.

7.4 Future Directions

This thesis opens up several interesting areas for further research. Some of these are listed below.

7.4.1 Extending the scope of this research

The following topics address some of the limitations of this research, and extend the utility and understanding of the problem solving techniques proposed in this thesis.

- *More efficient solution cache structure.* Experiments in Chapter 6 revealed that the current implementation of the bottom-up solution framework is inefficient in building and storing the solution cache. While the vector-node implementation of the solution cache tree removes most of the cost of building the cache, the space requirements are not improved.

  There are situations where the solution cache tree structure can be made more space efficient. For example, a fuller cache tree is more efficiently representable by storing values that do not occur. Relation factoring (e.g. [57]) can also be used to compress the solution cache.

- *Upgrading the Forward-Check implementation of bottom-up solution.* The current implementation treats divisions between subproblems as hard boundaries. This limitation can be removed, allowing the look-ahead step to test constraints that are outside the current subproblem.

- *When is IS-decomposition better than DOI-decomposition*? There are clearly problems where IS-decomposition produces a more efficient control sequence than DOI-decomposition (e.g. problem P_1 in section 3.2.2). The answer might involve the arity of the constraints in the problem, and the size of the independent set.

- *What is the relationship between a problem's constraint arity profile and decomposition performance*? More experiments need to be done to examine this relationship.

- *Hard problems for single-solution search.* The claim is that decomposition should be just as useful for hard problems when looking for one solution as when looking for all solutions. The search tree for a problem does not change its shape based upon the number of solutions. So a hard problem will have a fuller and bushier search tree regardless of the number of solutions examined. This is exactly the situation where DOI and IS decomposition techniques should benefit the most. The experiments of Chapter 4 were unable to confirm this because the problems generated by the random problem generator tended to be very easy for single-solutions search.

- *A solution cache structure not dependent on recall order.* This will be useful for search algorithms based on a dynamic control order, e.g. Dynamic Backtracking.

7.4.2 New directions

This research also suggests possibilities for improving other applications and problem solving algorithms. Some of these are:

- *Applying decomposition techniques to optimization tasks.* Branch and bound optimization algorithms are essentially constraint satisfaction algorithms with a heuristic function for pruning search. The hierarchical solution techniques proposed in this thesis should be of benefit here.

- *Control ordering to minimize artificial serial dependencies.* Since redundant tests are caused by artificial serial dependencies, even simple control sequences that minimize artificial variable dependence (as an example heuristic) should offer improved performance. Some of these heuristic techniques are described in [47], and were used in this thesis to construct the simple control sequence for each test problem.

- *Comparing Bottom-Up with CPR generalized for n-ary CSPs.* The cross-product representation (CPR) based algorithms [33, 32] also attack the redundant testing of constraints, using what are effectively incrementally extended domain decompositions of the CSP (see previous section for a brief description). While primarily

aimed at binary CSPs, it should be possible to extend them to problems with arbitrary arity constraints, and compare their performance with the Bottom-Up solution procedures described here.

- *Storing extensional constraints in a solution cache tree.* This was suggested by Freuder[1] as an efficient scheme for storing constraints available to the problem as a list of satisfying tuples. This could reduce or even eliminate the cost of testing such constraints.

- *Improving the explanation mechanism of Dynamic Backtracking.* As described in [26], Dynamic Backtracking only keeps track of "no-goods" — value combinations that violate some constraint. This allows the algorithms to avoid generating and testing these value combinations again. This can be extended to also storing values that have been previously tested and found to succeed, and use it to further reduce redundant testing. For this to be successful, the cost of looking up an "explanation" will have to be small.

- *Other decomposition algorithms for use with the bottom-up framework.* The previous section gives an example where the decomposition produced by a tree-clustering algorithm ([11]) produced a three-level control sequence for the problem P_1 that proved to be faster than both the IS and DOI decompositions.

- *"Beam search" bottom-up solution for finding one solution.* When only one or a small number of solutions are needed, a beam-search approach might be more fruitful even with bottom-up solution. In this solution strategy, control would shift to the next subproblem after a pre-specified number of solutions to the current subproblem are found. This would reduce the time spent in solving the subproblem, and in producing solutions that are not going to be used.

- *Recognizing existing hierarchical structure in a CSP.* The hierarchical solution strategies of Chapter 5 exploit the hierarchical structure built into a CSP produced by HiT. Recognizing the presence of a similar structure in other CSPs will allow

[1] Personal communication.

these strategies to be used on problems where global constraint decomposition is not applicable.

Appendix A

Application Domain Models

A.1 Floorplanning

```
(defconstant fp4
      `(
               ;;  Variable Names
         (VARS r1x1 r1y1 r1x2 r1y2              ; room 1
               r2x1 r2y1 r2x2 r2y2              ; room 2
               r3x1 r3y1 r3x2 r3y2              ; room 3
               r4x1 r4y1 r4x2 r4y2 )            ; room 3


               ;;  Domains for the Vars
         (DOMAINS (r1x1 ,(INTSEQ 0 9))          ; room 1
                  (r1y1 ,(INTSEQ 0 9))
                  (r1x2 ,(INTSEQ 1 10))
                  (r1y2 ,(INTSEQ 1 10))
                  (r2x1 ,(INTSEQ 0 9))          ; room 2
                  (r2y1 ,(INTSEQ 0 9))
                  (r2x2 ,(INTSEQ 1 10))
                  (r2y2 ,(INTSEQ 1 10))
                  (r3x1 ,(INTSEQ 0 9))          ; room 3
                  (r3y1 ,(INTSEQ 0 9))
                  (r3x2 ,(INTSEQ 1 10))
                  (r3y2 ,(INTSEQ 1 10))
                  (r4x1 ,(INTSEQ 0 9))          ; room 4
```

```
          (r4y1 ,(INTSEQ 0 9))
          (r4x2 ,(INTSEQ 1 10))
          (r4y2 ,(INTSEQ 1 10)) )

        ;;  Constraints (-name- -arg_vars- -expr-) ...
(CONSTRAINTS
                ;;  Rooms inside House, x1, y1, x2, y2
                ;;  r1
        (rinhouse-x1-r1  (r1x1)
                (< (var r1x1) *fp-house-length*)
        )
        (rinhouse-y1-r1  (r1y1)
                (< (var r1y1) *fp-house-width*)
        )
        (rinhouse-x2-r1  (r1x2)
                (<= (var r1x2) *fp-house-length*)
        )
        (rinhouse-y2-r1  (r1y2)
                (<= (var r1y2) *fp-house-width*)
        )
                ;;  Rooms inside House, x1, y1, x2, y2
                ;;  r2
        (rinhouse-x1-r2  (r2x1)
                (< (var r2x1) *fp-house-length*)
        )
        (rinhouse-y1-r2  (r2y1)
                (< (var r2y1) *fp-house-width*)
        )
        (rinhouse-x2-r2  (r2x2)
                (<= (var r2x2) *fp-house-length*)
```

```
)
(rinhouse-y2-r2  (r2y2)
        (<= (var r2y2) *fp-house-width*)
)
        ;;  Rooms inside House, x1, y1, x2, y2
        ;;  r3
(rinhouse-x1-r3  (r3x1)
        (< (var r3x1) *fp-house-length*)
)
(rinhouse-y1-r3  (r3y1)
        (< (var r3y1) *fp-house-width*)
)
(rinhouse-x2-r3  (r3x2)
        (<= (var r3x2) *fp-house-length*)
)
(rinhouse-y2-r3  (r3y2)
        (<= (var r3y2) *fp-house-width*)
)
        ;;  Rooms inside House, x1, y1, x2, y2
        ;;  r4
(rinhouse-x1-r4  (r4x1)
        (< (var r4x1) *fp-house-length*)
)
(rinhouse-y1-r4  (r4y1)
        (< (var r4y1) *fp-house-width*)
)
(rinhouse-x2-r4  (r4x2)
        (<= (var r4x2) *fp-house-length*)
)
(rinhouse-y2-r4  (r4y2)
```

```
        (<= (var r4y2) *fp-house-width*)
)

        ;; Room has positive Area (x2 > x1, y2 > y1)
        ;; r1
(cx2x1-r1  (r1x1 r1x2)
        (< (var r1x1) (var r1x2))
)
(cy2y1-r1  (r1y1 r1y2)
        (< (var r1y1) (var r1y2))
)
        ;; Room has positive Area (x2 > x1, y2 > y1)
        ;; r2
(cx2x1-r2  (r2x1 r2x2)
        (< (var r2x1) (var r2x2))
)
(cy2y1-r2  (r2y1 r2y2)
        (< (var r2y1) (var r2y2))
)
        ;; Room has positive Area (x2 > x1, y2 > y1)
        ;; r3
(cx2x1-r3  (r3x1 r3x2)
        (< (var r3x1) (var r3x2))
)
(cy2y1-r3  (r3y1 r3y2)
        (< (var r3y1) (var r3y2))
)
        ;; Room has positive Area (x2 > x1, y2 > y1)
        ;; r4
(cx2x1-r4  (r4x1 r4x2)
```

```
        (< (var r4x1) (var r4x2))
)
(cy2y1-r4  (r4y1 r4y2)
        (< (var r4y1) (var r4y2))
)

        ;; Room adjacent to House Side
        ;; r1
(adj.h.side-r1 (r1x1 r1y1 r1x2 r1y2)
        (OR  (=  (var r1x1)  0)
             (=  (var r1y1)  0)
             (=  (var r1x2)  *fp-house-length*)
             (=  (var r1y2)  *fp-house-width*)
        )
)
        ;; Room adjacent to House Side
        ;; r2
(adj.h.side-r2 (r2x1 r2y1 r2x2 r2y2)
        (or  (=  (var r2x1)  0)
             (=  (var r2y1)  0)
             (=  (var r2x2)  *fp-house-length*)
             (=  (var r2y2)  *fp-house-width*)
        )
)
        ;; Room adjacent to House Side
        ;; r3
(adj.h.side-r3 (r3x1 r3y1 r3x2 r3y2)
        (or  (=  (var r3x1)  0)
             (=  (var r3y1)  0)
             (=  (var r3x2)  *fp-house-length*)
```

```
                (=  (var r3y2)  *fp-house-width*)
        )
)
        ;; Room adjacent to House Side
        ;; r4
(adj.h.side-r4 (r4x1 r4y1 r4x2 r4y2)
        (or  (=  (var r4x1)  0)
                (=  (var r4y1)  0)
                (=  (var r4x2)  *fp-house-length*)
                (=  (var r4y2)  *fp-house-width*)
        )
)

        ;; Room has min area
        ;; r1
(min.area-r1  (r1x1 r1y1 r1x2 r1y2)
        (>=  (*  (- (var r1x2) (var r1x1))      ; r1 len
                 (- (var r1y2) (var r1y1)) )    ; r1 wid
                *fp-min-area-1*
        )
)
        ;; Room has min area
        ;; r2
(min.area-r2  (r2x1 r2y1 r2x2 r2y2)
        (>=  (*  (- (var r2x2) (var r2x1))      ; r2 len
                 (- (var r2y2) (var r2y1)) )    ; r2 wid
                *fp-min-area-2*
        )
)
        ;; Room has min area
```

```
        ;; r3
(min.area-r3  (r3x1 r3y1 r3x2 r3y2)
        (>=  (*  (- (var r3x2) (var r3x1))      ; r3 len
                 (- (var r3y2) (var r3y1)) )    ; r3 wid
             *fp-min-area-3*
        )
)
        ;; Room has min area
        ;; r4
(min.area-r4  (r4x1 r4y1 r4x2 r4y2)
        (>=  (*  (- (var r4x2) (var r4x1))      ; r4 len
                 (- (var r4y2) (var r4y1)) )    ; r4 wid
             *fp-min-area-4*
        )
)

        ;; Rooms Don't Overlap
        ;;   ri above rj   OR ri below rj  OR
        ;;   ri rightof rj OR ri leftof rj
        ;; r1 r2
(dont.ovlap-r1r2  (r1x1 r1y1 r1x2 r1y2
                   r2x1 r2y1 r2x2 r2y2)
        (OR  (>= (var r1y1) (var r2y2))
             (<= (var r1y2) (var r2y1))
             (>= (var r1x1) (var r2x2))
             (<= (var r1x2) (var r2x1))
        )
)
        ;; Rooms Don't Overlap
        ;; r1 r3
```

```
(dont.ovlap-r1r3  (r1x1 r1y1 r1x2 r1y2
                   r3x1 r3y1 r3x2 r3y2)
        (OR  (>= (var r1y1) (var r3y2))
             (<= (var r1y2) (var r3y1))
             (>= (var r1x1) (var r3x2))
             (<= (var r1x2) (var r3x1))
        )
)
        ;; Rooms Don't Overlap
        ;; r1 r4
(dont.ovlap-r1r4  (r1x1 r1y1 r1x2 r1y2
                   r4x1 r4y1 r4x2 r4y2)
        (OR  (>= (var r1y1) (var r4y2))
             (<= (var r1y2) (var r4y1))
             (>= (var r1x1) (var r4x2))
             (<= (var r1x2) (var r4x1))
        )
)
        ;; Rooms Don't Overlap
        ;; r2 r3
(dont.ovlap-r2r3  (r2x1 r2y1 r2x2 r2y2
                   r3x1 r3y1 r3x2 r3y2)
        (OR  (>= (var r2y1) (var r3y2))
             (<= (var r2y2) (var r3y1))
             (>= (var r2x1) (var r3x2))
             (<= (var r2x2) (var r3x1))
        )
)
        ;; Rooms Don't Overlap
        ;; r2 r4
```

```
(dont.ovlap-r2r4  (r2x1 r2y1 r2x2 r2y2
                   r4x1 r4y1 r4x2 r4y2)
        (OR  (>= (var r2y1) (var r4y2))
             (<= (var r2y2) (var r4y1))
             (>= (var r2x1) (var r4x2))
             (<= (var r2x2) (var r4x1))
        )
)
        ;; Rooms Don't Overlap
        ;; r3 r4
(dont.ovlap-r3r4  (r3x1 r3y1 r3x2 r3y2
                   r4x1 r4y1 r4x2 r4y2)
        (OR  (>= (var r3y1) (var r4y2))
             (<= (var r3y2) (var r4y1))
             (>= (var r3x1) (var r4x2))
             (<= (var r3x2) (var r4x1))
        )
)

        ;; Rooms Cover House
        ;;   sum of areas = house area
(rooms.cover.house (r1x1 r1y1 r1x2 r1y2
                    r2x1 r2y1 r2x2 r2y2
                    r3x1 r3y1 r3x2 r3y2
                    r4x1 r4y1 r4x2 r4y2)
    (=  (+  (*  (- (var r1x2) (var r1x1))       ;area r1
                (- (var r1y2) (var r1y1)) )
            (*  (- (var r2x2) (var r2x1))       ;area r2
                (- (var r2y2) (var r2y1)) )
            (*  (- (var r3x2) (var r3x1))       ;area r3
```

```
                                (- (var r3y2) (var r3y1)) )
                            (*  (- (var r4x2) (var r4x1))       ;area r4
                                (- (var r4y2) (var r4y1)) )
                        )
                        (*  *fp-house-length*  *fp-house-width*)
                    )
                )
        )
      )
)
```

A.2 Corporation Decentralization

```
(defconstant dc
      '(
                ;;  Variable Names

         (VARS     c0
        c1d1 c1d2 c1d3
        c2d1 c2d2 c2d3 c2d4
     )

                ;;  Domains for the Vars

         (DOMAINS (c0   ,(INTSEQ 1 5))        ; 5 cities {1..5}

          (c1d1 ,(INTSEQ 1 5))
          (c1d2 ,(INTSEQ 1 5))
          (c1d3 ,(INTSEQ 1 5))

          (c2d1 ,(INTSEQ 1 5))
```

```
    (c2d2 ,(INTSEQ 1 5))
    (c2d3 ,(INTSEQ 1 5))
    (c2d4 ,(INTSEQ 1 5))
)

          ;;  Constraints (-name- -arg_vars- -expr-) ...
(CONSTRAINTS
          ;;  Upto 4 depts in each city
          ;;
   (CityLimit.5  (c1d3 c1d2 c1d1 c2d4 c2d3)
       (city-limit 1 5)          ; Call general fn
   )

          ;;  Upto 4 depts in each city
          ;;
   (CityLimit.6  (c1d3 c1d2 c1d1 c2d4 c2d3 c2d2)
       (city-limit 1 6)          ; Call general fn
   )

          ;;  Upto 4 depts in each city
          ;;
   (CityLimit.7  (c1d3 c1d2 c1d1 c2d4 c2d3 c2d2 c2d1)
       (city-limit 1 7)          ; Call general fn
   )

          ;;  Upto 4 depts in each city
          ;;
   (CityLimit.8  (c1d3 c1d2 c1d1 c2d4 c2d3 c2d2 c2d1 c0)
       (city-limit 1 8)          ; Call general fn
   )
```

```
    ;;  Objective Function - Maximize Benefit
    ;;  SUM(i)
    ;;     Benefit(dept i in city di)
    ;;    - SUM(j=1..i-1)
    ;;        InterDeptComm(dept i, dept j)
    ;;        * UnitCommCost(city di, city dj)
    ;;
(MinBenefit  (c0 c1d1 c1d2 c1d3 c2d1 c2d2 c2d3 c2d4)

  (<  *ec-min-benefit*

    (+        ;; Final benefit in moving company 1
    (- (+ (benefit 2 (var c1d1))
          (benefit 3 (var c1d2))
          (benefit 4 (var c1d3))
       )
       (* (inter-dept-comm 2 3)
          (unit-comm-cost (var c1d1) (var c1d2)))
       (* (inter-dept-comm 2 4)
          (unit-comm-cost (var c1d1) (var c1d3)))
       (* (inter-dept-comm 3 4)
          (unit-comm-cost (var c1d2) (var c1d3)))
    )

        ;; Final benefit in moving company 2
    (- (+ (benefit 5 (var c2d1))
          (benefit 6 (var c2d2))
          (benefit 7 (var c2d3))
          (benefit 8 (var c2d4))
```

```
                )
                (* (inter-dept-comm 5 6)
                   (unit-comm-cost (var c2d1) (var c2d2)))
                (* (inter-dept-comm 5 7)
                   (unit-comm-cost (var c2d1) (var c2d3)))
                (* (inter-dept-comm 5 8)
                   (unit-comm-cost (var c2d1) (var c2d4)))
                (* (inter-dept-comm 6 7)
                   (unit-comm-cost (var c2d2) (var c2d3)))
                (* (inter-dept-comm 6 8)
                   (unit-comm-cost (var c2d2) (var c2d4)))
                (* (inter-dept-comm 7 8)
                   (unit-comm-cost (var c2d3) (var c2d4)))
             )

                 ;; Final benefit in moving company 0
             (- (benefit 1 (var c0))
                (* (inter-dept-comm 1 2)
                   (unit-comm-cost (var c0) (var c1d1)))
                (* (inter-dept-comm 1 5)
                   (unit-comm-cost (var c0) (var c2d1)))
             )

             )
           )
         )
         )
       )
)
```

A.3 Corrugated Bulkhead Design

```
(defconstant sb
      `(
;;  Variable Names
(VARS b1 b2 d ttop tmid tbot)
;;  Domains for the Vars
(DOMAINS (b1   ,(REALSEQ 55.0 64.9  0.1))
 (b2   ,(REALSEQ 55.0 64.9  0.1))
 (d    ,(REALSEQ 35.0 39.9  0.1))
 (ttop ,(REALSEQ  1.0  1.2  0.05))
 (tmid ,(REALSEQ  1.0  1.2  0.05))
 (tbot ,(REALSEQ  1.0  1.2  0.05)) )
;;  Constraints (-name- -arg_vars- -expr-) ...
(CONSTRAINTS
;; Objective Function
(c1 (b1 b2 d ttop tmid tbot)
    (< (* (+b1b2 (VAR b1) (VAR b2))
   (+ (* 495 (VAR ttop))
      (* 385 (VAR tmid))
      (* 315 (VAR tbot))
   )
)
(* 1400
   (+ (VAR b1)
      (SQRT (- (SQR (VAR b2))  (SQR (VAR d))))
   )
)
    )
)
```

```
;; --- Section Modulus Constraints ---

;; 1/6.b2.d.ttop + 1.36xb1.d.ttop
;;  - 40x[ b1 + (b2^2 - d^2)^0.5 ] >= 0
(c2 (b1 b2 d ttop)
    (<= 0
(+ (* 0.67 (* (VAR b2) (VAR d) (VAR ttop)))
   (* 1.36 (* (VAR b1) (VAR d) (VAR ttop)))
   (NEG (* 40.0 (+ (VAR b1)
   (SQRT (- (SQR (VAR b2))
    (SQR (VAR d))))
)
)
   )
)
    )
)
;; 1/6.b2.d.tmid + 1.36xb1.d.tmid
;;  - 40x[ b1 + (b2^2 - d^2)^0.5 ] >= 0
(c3 (b1 b2 d tmid)
    (<= 0
(+ (* 0.67 (* (VAR b2) (VAR d) (VAR tmid)))
   (* 1.36 (* (VAR b1) (VAR d) (VAR tmid)))
   (NEG (* 40.0 (+ (VAR b1)
   (SQRT (- (SQR (VAR b2))
    (SQR (VAR d))))
)
)
   )
)
```

```
    )
)
;; 1/6.b2.d.tbot + 1.36xb1.d.tbot
;;  - 40x[ b1 + (b2^2 - d^2)^0.5 ] >= 0
(c4 (b1 b2 d tbot)
    (<= 0
(+ (* 0.67 (* (VAR b2) (VAR d) (VAR tbot)))
   (* 1.36 (* (VAR b1) (VAR d) (VAR tbot)))
   (NEG (* 40.0 (+ (VAR b1)
   (SQRT (- (SQR (VAR b2))
    (SQR (VAR d))))
)
)
   )
)
    )
)
;; --- Moment of Inertia Constraints ---

;; 1/12.b2.d^2.ttop + 0.68xb1.d^2.ttop
;;  - 130x[ b1 + (b2^2 - d^2)^0.5 ]^(4/3) >= 0
(c5 (b1 b2 d ttop)
    (<= 0
(+ (* 0.08 (* (VAR b2) (SQR (VAR d)) (VAR ttop)))
   (* 0.68 (* (VAR b1) (SQR (VAR d)) (VAR ttop)))
   (NEG (*  130.0
    (Pwr1.33
(+ (VAR b1)
   (SQRT (- (SQR (VAR b2))
    (SQR (VAR d))))
```

```
)
    )
)
   )
)
    )
)
;; 1/12.b2.d^2.tmid + 0.68xb1.d^2.tmid
;;  - 130x[ b1 + (b2^2 - d^2)^0.5 ]^(4/3) >= 0
(c6 (b1 b2 d tmid)
    (<= 0
(+ (* 0.08 (* (VAR b2) (SQR (VAR d)) (VAR tmid)))
   (* 0.68 (* (VAR b1) (SQR (VAR d)) (VAR tmid)))
   (NEG (*  130.0
    (Pwr1.33
(+ (VAR b1)
   (SQRT (- (SQR (VAR b2))
    (SQR (VAR d))))
)
    )
)
   )
)
    )
)
;; 1/12.b2.d^2.tbot + 0.68xb1.d^2.tbot
;;  - 130x[ b1 + (b2^2 - d^2)^0.5 ]^(4/3) >= 0
(c7 (b1 b2 d tbot)
    (<= 0
(+ (* 0.08 (* (VAR b2) (SQR (VAR d)) (VAR tbot)))
```

```
   (* 0.68 (* (VAR b1) (SQR (VAR d)) (VAR tbot)))
   (NEG (*  130.0
    (Pwr1.33
(+ (VAR b1)
   (SQRT (- (SQR (VAR b2))
    (SQR (VAR d))))
)
    )
)
   )
)
    )
)
;; --- Plate thickness Constraints ---

;; ttop - 0.012xb1 >= 0.12
(c8 (b1 ttop)
    (<=  0.12
 (-  (VAR ttop) (* 0.012 (VAR b1)))
    )
)
;; ttop - 0.012xb2 >= 0.12
(c9 (b2 ttop)
    (<=  0.12
 (-  (VAR ttop) (* 0.012 (VAR b2)))
    )
)
;; tmid - 0.015xb1 >= 0.12
(c10 (b1 tmid)
    (<=  0.12
```

```
 (-  (VAR tmid) (* 0.015 (VAR b1)))
    )
)
;; tmid - 0.015xb2 >= 0.12
(c11 (b2 tmid)
    (<=  0.12
 (-  (VAR tmid) (* 0.015 (VAR b2)))
    )
)
;; tbot - 0.017xb1 >= 0.12
(c12 (b1 tbot)
    (<=  0.12
 (-  (VAR tbot) (* 0.017 (VAR b1)))
    )
)
;; tbot - 0.017xb2 >= 0.12
(c13 (b2 tbot)
    (<=  0.12
 (-  (VAR tbot) (* 0.017 (VAR b2)))
    )
)
 )
    )
)
```

References

[1] Jerome Bracken and Garth P. McCormick. *Selected Applications of Nonlinear Programming.* John Wiley and Sons, 1968.

[2] Wesley Braudaway. Automated synthesis of constrained generators. In *Automating Software Design.* AAAI Press, 1991.

[3] C. A. Brown and P. W. Purdom Junior. How to search efficiently. In *Proceedings of the 7th International Joint Conference on AI*, pages 588–594, 1981.

[4] Maurice Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1), February 1981.

[5] Yves Caseau. Abstract interpretation of constraints over an order-sorted domain. In *Proceedings of the ILPS*, San Diego, 1991.

[6] Yves Caseau and Jean-Francois Puget. Constraints on order-sorted domains. In *Proceedings of the European Conference on AI*, 1994.

[7] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the International Joint Conference on AI*, pages 331–337, Sydney, Australia, 1991.

[8] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[9] Rina Dechter and Judea Pearl. The cycle-cutset method for improving search performance in ai applications. In *Proceedings of the Third IEEE conference on AI applications*, pages 224–230, Orlando, 1987.

[10] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[11] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 35, 1989.

[12] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car sequencing problem in clp. In *Proceedings of the ECAI*, pages 290–295, 1988.

[13] Charles M. Eastman. Automated space planning. *Artificial Intelligence*, 4:41–64, 1973.

[14] Thomas Ellman. Abstraction via approximate symmetry. In *Proceedings of the International Joint Conference on AI*, pages 916–921, 1993.

[15] Thomas Ellman. Synthesis of abstraction hierarchies for constraint satisfaction by clustering approximately equivalent objects. In *Proceedings of the International Machine Learning Workshop*, 1993.

[16] U. Flemming, C. A. Baykan, R. F. Coyne, and M. S. Fox. Hierarchical generate-and-test vs constraint-directed search. In J. S. Gero, editor, *AI in Design*, pages 817–838. Kluwer Academic Publishers, 1992.

[17] Ulrich Flemming, Robert F. Coyne, Timothy Glavin, Hung Hsi, and Michael D. Rychener. A generative expert system for the design of building layouts (final report). Technical Report EDRC 48-15-89, Engineering Design Research Centre, CMU, Pittsburgh, PA, 1989.

[18] Eugene C. Freuder. A sufficient condition for backtrack-free search. *JACM*, 29(1):24–32, 1982.

[19] Eugene C. Freuder. A sufficient condition for backtrack-bounded search. *JACM*, 34(4):755–761, 1985.

[20] Eugene C. Freuder. Partial constraint satisfaction. In *Proceedings of the 11th International Joint Conference on AI*, pages 278–283, 1989.

[21] Eugene C. Freuder and Paul D. Hubbe. Using inferred disjunctive constraints to decompose constraint satisfaction problems. In *Proceedings of the 13th International Joint Conference on AI*, pages 254–260, Chambery, France, 1993.

[22] Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proceedings of the International Joint Conference on AI*, 1985.

[23] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence (special volume: Constraint-Based Reasoning)*, 58(1-3):21–70, 1992.

[24] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[25] J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proceedings of the International Joint Conference on AI*, MIT, Cambridge, MA, August 1977.

[26] Matthew L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1:25–46, 1993.

[27] Solomon W. Golumb and Leonard D. Baumert. Backtrack programming. *JACM*, 12(4):516–524, 1965.

[28] Marc Gyssens, Peter G. Jeavons, and David A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, ?, 1994.

[29] P. V. Hall. Optimization of a single relational expression in a relational data base system. *IBM Journal of Research and Development*, (20) 3, 1976.

[30] R. M. Haralick, L.S. Davis, A. Rosenfeld, and D. L. Milgram. Reduction operations for constraint satisfaction. *Information Science*, 14:199–219, 1978.

[31] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[32] Paul D. Hubbe. Cross product representation of the constraint satisfaction problem search space. Technical Report 92-17, Dept of Computer Science, University of New Hampshire, September 1992. Master's Thesis.

[33] Paul D. Hubbe and Eugene C. Freuder. An efficient cross-product representation of the constraint satisfaction problem search space. In *Proceedings of the National Conference on AI*, pages 421–428, 1992.

[34] M. Jarke. Common subexpression isolation in multiple query optimization. In D. Reiner W. Kim and D. Batory, editors, *Query Processing in Database Systems*. Springer-Verlag, 1984.

[35] Philippe Jegou. Cyclic-clustering: a compromise between tree-clustering and the cycle-cutset method for improving search efficiency. In *Proceedings of the European Conference on AI*, pages 369–371, Stockholm, 1990.

[36] M. D. Johnston. Spike: Ai scheduling for nasa's hubble space telescope. In *Proceedings of the Sixth Conference on AI Applications*. IEEE, 1990.

[37] Craig A. Knoblock, J. Tennenberg, and Q. Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth NCAI*, Anaheim, CA, 1991.

[38] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, January 1975.

[39] S. C-Y. Lu and G. Wilhelm. Applying constraint-based reasoning to geometric tolerancing. In J. S. Gero, editor, *Applications of AI in Engineering V*, pages 37–54. Computational Mechanics Publications with Springer-Verlag, 1990.

[40] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[41] Alan K. Mackworth, J. A. Mulder, and W. S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction. *Computational Intelligence*, 1(3):118–126, 1985.

[42] Steven Minton. Integrating heuristics for constraint satisfaction problems: A case study. In *Proceedings of the National Conference on AI (AAAI-93)*, 1993.

[43] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of sat problems. In *Proceedings of the National Conference on AI (AAAI-92)*, pages 459–465, 1992.

[44] Sanjay Mittal and F. Frayman. Making partial choices in constraint reasoning problems. In *Proceedings of the National Conference on AI (AAAI-87)*, pages 631–636, Seattle, WA, August 1987.

[45] Sunil Mohan. Constructing hierarchical solvers for constraint satisfaction problems. AI/Design Working Paper 137, Department of Computer Science, Rutgers University, New Brunswick, NJ, May 1989. Thesis Proposal.

[46] Sunil Mohan. Constructing hierarchical solvers for functional constraint satisfaction problems. In *Working Notes, AAAI Spring Symposium on Constraint-Based Reasoning*, pages 147–153, Stanford University, Palo Alto, CA, March 1991.

[47] Sunil Mohan. The trouble with n-ary. In *Proceedings of the Workshop on AI Approaches to Modelling and Scheduling Manufacturing Processes, at the 1994 IEEE Conference on Tools with AI.* IEEE, November 1994.

[48] Sunil Mohan and Chris Tong. Automatic construction of a hierarchical generate-and-test algorithm. In *Proceedings of the International Machine Learning Workshop*, Ithaca, NY, 1989.

[49] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[50] Ugo Montanari and Francesca Rossi. Constraint relaxation may be perfect. *Artificial Intelligence*, 48:143–170, 1991.

[51] Nicola Muscettola, Barney Pell, Othar Hansson, and Sunil Mohan. Automating mission scheduling for space-based observatories. In *Robotic Telescopes*, volume 79 of *ASP Conference Series*, pages 148–166. Astronomical Society of the Pacific, San Francisco, California, 1995.

[52] B. A. Nadel and J. Lin. Automobile transmission design as a constraint satisfaction problem: a collaborative research project with ford motor co. In *Proceedings of the AAAI-90 Workshop on Constraint Directed Reasoning*, August 1990.

[53] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons, 1988.

[54] J. C. Shaw Newell, Alan and H. A. Simon. Report on a general problem-solving program. In *Information Processing - UNESCO Proceedings*, pages 256–264. UNESCO, June 1959.

[55] Nils J. Nilsson. *Principles of AI.* Morgan Kaufmann, 1980.

[56] B. A. Nudel. Consistent-labeling problems and their algorithms: expected-complexities and theory-based heuristics. *Artificial Intelligence (special issue on Search and Heuristics)*, 21(3 and 4):135–178, March 1983. Also in book: *Search and Heuristics*, North Holland, Amsterdam, 1983.

[57] Mark Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[58] P. W. Jr. Purdom and C. A. Brown. An analysis of backtracking with search rearrangement. *SIAM Journal of Computing*, 12(4):717–733, 1983.

[59] F. Rossi and U. Montanari. Exact solution in linear time of networks of constraints using perfect relaxation. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 394–399. Morgan Kaufmann, 1989.

[60] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[61] Timos K. Sellis. Global query optimization. In *ACM-SIGMOD*, pages 191–205, 1986.

[62] Murray Shanahan and Richard Southwick. *Search, Inference and Dependencies in AI*. Ellis Horwood Ltd., Chichester, England, 1989.

[63] David E. Smith. *Controlling Inference*. PhD thesis, Dept of Computer Science, Stanford University, 1985.

[64] R. M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[65] Mark J. Stefik. Planning and meta-planning (MOLGEN: Part 2). *Artificial Intelligence*, 16:141–169, 1981.

[66] Mark J. Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16:111–140, 1981.

[67] C. Tong. KBSDE: an environment for developing knowledge-based design tools. In T. Dietterich, editor, *Proceedings of the Workshop on Knowledge Compilation*, pages 127–138, Oregon State University, September 1986.

[68] Christopher Tong, Wesley Braudaway, Sunil Mohan, and Kerstin Voigt. Reformulating constraints for compilability and efficiency. In *Proceedings of the Workshop on Change of Representation and Problem Reformulation*, April 1992. Pacific Grove, CA.

[69] Kerstin Voigt and Chris Tong. Automating the construction of patchers that satisfy global constraints. In *Proceedings of the International Joint Conference on AI*, 1989.

[70] H. P. Williams. *Model Building in Mathematical Programming*. John Wiley and Sons, second edition, 1985.

[71] Ramin Zabih. Some applications of graph bandwidth to constraint satisfaction problems. In *Proceedings of the National Conference on AI (AAAI-90)*, pages 46–51, 1990.

Vita

Sunil K. Mohan

1980 B.S. in Electrical Engineering, Delhi College of Engineering, Delhi University, New Delhi, India.

1980-81 Attended the M.Tech program in Computer Science at the Indian Institute of Technology, Delhi, India.

1988-89 Project Leader, JYACC, Inc., New York City, NY.

1989 S. Mohan and C. Tong. Automatic construction of a hierarchical generate-and-test algorithm. International Machine Learning Workshop 1989.

1989 M.S. Computer Science, Rutgers University, NJ.

1989-93 Project Manager, Mercantile Software Systems, Piscataway, NJ.

1992 C. Tong, W. Braudaway, S. Mohan, and K. Voigt. Reformulating constraints for compilability and efficiency. Workshop on Change of Representation and Problem Reformulation, Pacific Grove, CA, April 1992.

1994 Consultant, Accurate Information Systems, Eatontown, NJ, and Center for Computer Aids in Industrial Productivity, Rutgers University, NJ.

1994- Computer Scientist, NASA Ames Research Center, Moffett Field, CA.

1995 N. Muscettola, B. Pell, O. Hansson, S. Mohan. Automating mission scheduling for space-based observatories. Robotic Telescopes, ASP Conference Series, Vol. 79.

1996 Ph.D. in Computer Science.