

University of Wollongong - Research Online

Thesis Collection

Title: Solving very large distributed constraint satisfaction problems

Author: Peter Harvey

Year: 2009

Repository DOI:

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following: This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author. Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material.

Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Research Online is the open access repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au



RESEARCH ONLINE

University of Wollongong
Research Online

University of Wollongong Thesis Collection

University of Wollongong Thesis Collections

2009

Solving very large distributed constraint satisfaction problems

Peter Harvey
University of Wollongong

Recommended Citation

Harvey, Peter, Solving very large distributed constraint satisfaction problems, Doctor of Philosophy thesis, School of Computer Science and Software Engineering - Faculty of Informatics, University of Wollongong, 2009. <http://ro.uow.edu.au/theses/3161>

Research Online is the open access institutional repository for the University of Wollongong. For further information contact Manager Repository Services: morgan@uow.edu.au.



RESEARCH ONLINE

NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Solving Very Large Distributed Constraint Satisfaction Problems

A thesis submitted in partial fulfilment of the
requirements for the award of the degree

Doctor of Philosophy

from

University of Wollongong

by

Peter Harvey

Bachelor of Mathematics

Bachelor of Computer Science

School of Computer Science and Software Engineering

2009

CERTIFICATION

I, Peter A. Harvey, declare that this thesis, submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Computer Science and Software Engineering, University of Wollongong, is wholly my own work unless otherwise referenced or acknowledged. The document has not been submitted for qualifications at any other academic institution.

Peter A. Harvey
8 December 2009

Abstract

This thesis investigates issues with existing approaches to distributed constraint satisfaction, and proposes a solution in the form of a new algorithm. These issues are most evident when solving large distributed constraint satisfaction problems, hence the title of the thesis.

We will first survey existing algorithms for centralised constraint satisfaction, and describe how they have been modified to handle distributed constraint satisfaction. The method by which each algorithm achieves completeness will be investigated and analysed by application of a new theorem.

We will then present a new algorithm, Support-Based Distributed Search, developed explicitly for distributed constraint satisfaction rather than being derived from centralised algorithms. This algorithm is inspired by the inherent structure of human arguments and similar mechanisms we observe in real-world negotiations.

A number of modifications to this new algorithm are considered, and comparisons are made with existing algorithms, effectively demonstrating its place within the field. Empirical analysis is then conducted, and comparisons are made to state-of-the-art algorithms most able to handle large distributed constraint satisfaction problems.

Finally, it is argued that any future development in distributed constraint satisfaction will necessitate changes in the algorithms used to solve small ‘embedded’ constraint satisfaction problems. The impact on embedded constraint satisfaction problems is considered, with a brief presentation of an improved algorithm for hypertree decomposition.

Previously published work includes [HG03, HCG05, HCG06a, HCG06b, HCG06c].

This thesis is dedicated to
my dearest wife Emily,
my baby daughter Adelaide,
my parents Keith and Sandra,
and my siblings Sean and Danielle.
I love you. You mean the world to me.

I would like to thank
Professor Aditya Ghose for his guidance,
Chee Fon Chang for his fellowship,
Farzad Salim for his friendship,
and the partners and many friends
who helped me through these last years.

Two weddings, one divorce, and a beautiful baby...
who would have thought it would take this long?

Table of Contents

1	Introduction	1
1.1	Constraint Satisfaction Problem	2
1.2	Distributed Constraint Satisfaction Problem	3
1.3	Centralised vs Distributed Algorithms	4
1.4	Motivation	5
2	Analysis of Completeness Techniques	10
2.1	Common Proof Theorem	11
2.2	Chronological Backtracking	14
2.3	Chronological Backtracking with Reordering	17
2.4	Dynamic Backtracking	20
2.5	Weak Commitment Search	24
2.6	Asynchronous Backtracking	26
2.7	Asynchronous Backtracking with Dynamic Ordering	31
2.8	Asynchronous Weak Commitment Search	34
2.9	Breakout	37
2.10	Distributed Breakout	38
2.11	Total, Partial, Dynamic and Static Orders	39
2.12	Categorisation	40
3	Support-Based Distributed Search	43
3.1	Introduction	43
3.2	Representation	45
3.2.1	Assignments as Arguments	47
3.2.2	Interpretation of Arguments	49
3.3	Solving	52
3.3.1	Asynchronicity with Isgoods	56
3.3.2	Computation of Isgoods	57
3.3.3	Demonstration of Isgoods	59
3.3.4	Postponement of Isgoods	61
3.4	Results	65
3.4.1	Soundness of Algorithm	68
3.4.2	Completeness of Algorithm	69

4	Variations and Relations	71
4.1	Minimising Conflicts	71
4.2	Minimising Communication	74
4.3	Minimising Storage	76
4.4	Relation to Other Algorithms	79
4.4.1	Asynchronous Weak-Commitment Search (AWCS)	79
4.4.2	Distributed Breakout (DBO)	80
4.4.3	Asynchronous Backtracking With Dynamic Ordering (ABT-DO) . .	81
5	Empirical Analysis	82
5.1	Metrics	82
5.1.1	Total vs Non-Concurrent Measures	83
5.1.2	Constraint Checks	84
5.1.3	Nogood Checks	84
5.1.4	Bytes	84
5.1.5	Packets	85
5.1.6	Bytes Per Packet	85
5.1.7	CPU Time	85
5.1.8	Concurrent Checks and Concurrent Traffic	85
5.2	Implementation	86
5.2.1	ABT and ABT-DO Implementation Notes	86
5.2.2	AWCS Implementation Notes	87
5.2.3	SBDS Implementation Notes	88
5.2.4	SBDS Value Selection Heuristic	88
5.3	Problem Sets	90
5.4	Results for Smaller Problems	91
5.5	Results for Larger Problems	99
5.6	Discussion	105
5.6.1	Coordinated Approach	105
5.6.2	Non-Coordinated Approach	106
5.6.3	Unified Approach	107
5.7	Summary	108
6	Multiple Variables Per Agent	109
6.1	Introduction	109
6.2	Hypertree Decompositions	111
6.2.1	General Form	112
6.2.2	Normal Form	113

6.2.3	Reduced Normal Form	114
6.3	Algorithm	118
6.3.1	opt- k -decomp	118
6.3.2	red- k -decomp	120
6.4	Performance	126
6.5	Application Within DisCSP Agents	128
7	Conclusion	129
7.1	Summary	131
7.2	Future Work	132
A	Results	140

List of Figures

1.1	It can never be clear which agent should take the greatest burden of search.	6
1.2	Adding links will increase the amount of communication between agents.	7
1.3	Distributed problems should not be treated in isolation, but in a wider network.	8
2.1	CBT: Making an assignment involves taking from the head of statically ordered list of unassigned variables and appending to the tail of an ordered list of assigned variables	14
2.2	CBT: Backtracking an assignment is the exact inverse, taking from the tail of the assigned variables and appending to the head of an ordered list of unassigned variables	14
2.3	CBT-R: Making an assignment involves choosing a variable from an unordered list of unassigned variables and appending to the tail of an ordered list of assigned variables	18
2.4	CBT-R: Backtracking an assignment is the exact inverse, taking from the tail of the assigned variables and appending to an unordered list of unassigned variables	18
2.5	DBT: ‘Eliminating explanations’ tell us which values are unusable, based on current values for specific assigned variables.	20
2.6	DBT: When all values are eliminated, the last-assigned variable causing any of the eliminations is unassigned, and a new eliminating explanation constructed.	20
2.7	WCS: Variables are assigned and partitioned into ‘current consistent’ and ‘tentative inconsistent’. Tentatively-assigned variables are iteratively given consistent assignments.	24
2.8	WCS: When a tentatively-assigned variable cannot be given a consistent assignment, the current assignment is transformed into a nogood and all assignments become tentative.	24
2.9	ABT: Changes in assignment are broadcast from each agent to each lower-ranked agent.	27
2.10	ABT: Nogoods only contain assignments from higher-ranked agents, and are sent to the lowest-ranked agent.	27
2.11	ABT-DO: Changes in assignment are broadcast to all neighbours regardless of rank.	32

2.12	ABT: Changes in ordering only effect lower-ranked agents, and are sent to all of them.	32
2.13	AWCS: Changes in assignment are broadcast from each agent to each neighbour, regardless of priorities.	34
2.14	AWCS: Nogoods are sent to all involved agents, and the priority of the sender is raised above all its neighbours.	34
2.15	BO: Constraint violations are weighted and summed. Dynamic adjustments of weights allow a simple hill-climbing algorithm to escape local minima. .	37
3.1	Example constraint model and graph	44
3.2	Constraint model and graph for the ring-ordering problem	61
3.3	Constraint model and graph demonstrating cyclic behaviour in SBDS . . .	63
4.1	Constraint model and support graph for a simple 5-node problem	72
5.1	Average feasibility of problem instances for Problem Sets 1 and 2	92
5.2	Constraint checks for Problem Sets 1 and 2. Full results on pages 141 and 159.	92
5.3	Constraint checks for Problem Sets 1 and 2, broken down by feasibility. Full results on pages 142, 143, 160 and 161.	93
5.4	Nogood checks for Problem Sets 1 and 2. Full results on pages 144 and 162.	94
5.5	Network traffic for Problem Sets 1 and 2. Full results on pages 147 and 165.	95
5.6	Numbers of packets for Problem Sets 1 and 2. Full results on pages 150 and 168.	96
5.7	Packet sizes for Problem Sets 1 and 2. Full results on pages 153 and 171. .	97
5.8	Concurrent checks for Problem Sets 1 and 2. Full results on page 192. . . .	98
5.9	CPU time for Problem Sets 1 and 2. Full results on pages 153 and 171. . . .	99
5.10	Average feasibility of problem instances for Problem Set 3	99
5.11	Constraint checks for Problem Set 3. Full results on page 177.	101
5.12	Nogood checks for Problem Set 3. Full results on page 180.	101
5.13	Number of packets for Problem Set 3. Full results on page 186.	101
5.14	Packet sizes for Problem Set 3. Full results on page 189.	103
5.15	Network traffic for Problem Set 3. Full results on page 183.	103
5.16	CPU time for Problem Set 3. Full results on page 189.	103
5.17	Concurrent checks for Problem Set 3, broken down by feasibility. Full results on pages 193 and 194.	104
5.18	Concurrent traffic for Problem Set 3, broken down by feasibility. Full results on pages 193 and 194.	105

6.1	A demonstration of how hypertree decomposition forms a new acyclic problem from a cyclic problem.	111
6.2	Transforming a normal form hypertree decomposition to reduced normal form.	117
6.3	Graphs of the CPU time of opt- k -decomp, and comparisons to its best-case and worst-case complexity functions. Each point represents a single run of opt- k -decomp.	127
6.4	Graphs of the CPU time of red- k -decomp, and comparisons to its best-case and worst-case complexity functions. Each point represents a single run of red- k -decomp.	127
6.5	A comparison of CPU times for random CSP instances. Values are computed as the CPU time for opt- k -decomp divided by the CPU time for red- k -decomp.	128
A.1	Number of constraint checks for all instances in problem set 1	141
A.2	Number of constraint checks for feasible instances in problem set 1	142
A.3	Number of constraint checks for infeasible instances in problem set 1 . . .	143
A.4	Number of nogood checks for all instances in problem set 1	144
A.5	Number of nogood checks for feasible instances in problem set 1	145
A.6	Number of nogood checks for infeasible instances in problem set 1	146
A.7	Network traffic for all instances in problem set 1	147
A.8	Network traffic for feasible instances in problem set 1	148
A.9	Network traffic for infeasible instances in problem set 1	149
A.10	Number of packets for all instances in problem set 1	150
A.11	Number of packets for feasible instances in problem set 1	151
A.12	Number of packets for infeasible instances in problem set 1	152
A.13	Average packet size and CPU time for all instances in problem set 1	153
A.14	Average packet size and CPU time for feasible instances in problem set 1 .	154
A.15	Average packet size and CPU time for infeasible instances in problem set 1 .	155
A.16	Other measures of concurrency for all instances in problem set 1	156
A.17	Other measures of concurrency for feasible instances in problem set 1 . . .	157
A.18	Other measures of concurrency for infeasible instances in problem set 1 . .	158
A.19	Number of constraint checks for all instances in problem set 2	159
A.20	Number of constraint checks for feasible instances in problem set 2	160
A.21	Number of constraint checks for infeasible instances in problem set 2 . . .	161
A.22	Number of nogood checks for all instances in problem set 2	162
A.23	Number of nogood checks for feasible instances in problem set 2	163
A.24	Number of nogood checks for infeasible instances in problem set 2	164
A.25	Network traffic for all instances in problem set 2	165

A.26	Network traffic for feasible instances in problem set 2 .	166
A.27	Network traffic for infeasible instances in problem set 2 .	167
A.28	Number of packets for all instances in problem set 2 .	168
A.29	Number of packets for feasible instances in problem set 2 .	169
A.30	Number of packets for infeasible instances in problem set 2 .	170
A.31	Average packet size and CPU time for all instances in problem set 2 .	171
A.32	Average packet size and CPU time for feasible instances in problem set 2 .	172
A.33	Average packet size and CPU time for infeasible instances in problem set 2 .	173
A.34	Other measures of concurrency for all instances in problem set 2 .	174
A.35	Other measures of concurrency for feasible instances in problem set 2 .	175
A.36	Other measures of concurrency for infeasible instances in problem set 2 .	176
A.37	Number of constraint checks for all instances in problem set 3 .	177
A.38	Number of constraint checks for feasible instances in problem set 3 .	178
A.39	Number of constraint checks for infeasible instances in problem set 3 .	179
A.40	Number of nogood checks for all instances in problem set 3 .	180
A.41	Number of nogood checks for feasible instances in problem set 3 .	181
A.42	Number of nogood checks for infeasible instances in problem set 3 .	182
A.43	Network traffic for all instances in problem set 3 .	183
A.44	Network traffic for feasible instances in problem set 3 .	184
A.45	Network traffic for infeasible instances in problem set 3 .	185
A.46	Number of packets for all instances in problem set 3 .	186
A.47	Number of packets for feasible instances in problem set 3 .	187
A.48	Number of packets for infeasible instances in problem set 3 .	188
A.49	Average packet size and CPU time for all instances in problem set 3 .	189
A.50	Average packet size and CPU time for feasible instances in problem set 3 .	190
A.51	Average packet size and CPU time for infeasible instances in problem set 3 .	191
A.52	Other measures of concurrency for all instances in problem set 3 .	192
A.53	Other measures of concurrency for feasible instances in problem set 3 .	193
A.54	Other measures of concurrency for infeasible instances in problem set 3 .	194

List of Tables

2.1	Classification of algorithms according to their completeness mechanism . .	41
6.1	Complexity results for red- k -decomp	124
6.2	Results of red- k -decomp on successively larger ‘spider webs’ constraint graphs. ‘Cond 2’ and ‘Cond 3’ shows the number of k -vertices discarded due to con- ditions 2 and 3 of RNF. Run-times (in seconds) for <i>init-vertices</i> and <i>init-</i> <i>graph</i> are given as a combined value.	125

Terminology

Constraint satisfaction literature often uses the same term but with differing definitions. The following definitions will be used throughout this thesis.

complete	<p>The terms ‘complete’ and ‘incomplete’ will indicate whether concepts or methods cover all possibilities. An assignment is complete if and only if it provides values for all variables. An algorithm is complete if and only if it provides an answer for all problems. Note that we are using ‘complete’ in the general algorithmic sense, and not to indicate that a constraint satisfaction algorithm considers all possible assignments.</p> <p><i>Example:</i> A solution must be a complete assignment.</p> <p><i>Example:</i> Breakout is an incomplete algorithm.</p>
consistent	<p>The terms ‘consistent’ and ‘inconsistent’ will refer to simple tests that can be conducted with available information. The most common instance of this in constraint satisfaction is to say that a particular combination of values is consistent/inconsistent with the set of constraint. If necessary, an algorithm may redefine what it means to test an assignment for consistency.</p> <p><i>Example:</i> The assignment is first tested for consistency.</p> <p><i>Example:</i> The current assignment may still be inconsistent.</p>
feasible	<p>The terms ‘feasible’ and ‘infeasible’ will refer to more complex determinations made by an algorithm during its execution. This is most often used in constructive search algorithms once they prove, by exhaustive search, that a partial assignment of values to variables cannot be extended into a consistent assignment for all variables. Note that an assignment is feasible if and only if it is a subset of a complete consistent assignment.</p> <p><i>Example:</i> Nogoods record which assignments are infeasible.</p> <p><i>Example:</i> Let T be the set of all feasible assignments.</p>
solvable	<p>The terms ‘solvable’ and ‘unsolvable’ will refer to whether or not a constraint satisfaction problem has a solution. A solution is a complete, consistent assignment of values to variables. By definition, an unsolvable problem has no feasible assignments.</p> <p><i>Example:</i> If $E = \emptyset$, we can conclude the problem is unsolvable.</p> <p><i>Example:</i> Breakout search is only suitable for solvable problems.</p>

Formal Notation

Formulas, algorithms and proofs will attempt to use a consistent lettering and numbering scheme. When no additional information is provided, the following definitions should be assumed.

$\mathcal{V}, \mathcal{C}, \mathcal{D}$	The symbols \mathcal{V} , \mathcal{C} , and \mathcal{D} respectively refer to the variables, constraints, and domain of a given problem. If more than one problem exists we will subscript related symbols according. For example \mathcal{V}_1 , \mathcal{C}_1 and \mathcal{D}_1 .
V, C, D	The letters V , C , and D will refer to <i>subsets</i> of \mathcal{V} , \mathcal{C} , and \mathcal{D} respectively.
s, t	In most instances the letters s and t refer to assignments. An assignment is a function mapping some subset of \mathcal{V} to \mathcal{D} . They should not be assumed to be complete assignments (mapping <i>all</i> of \mathcal{V} to \mathcal{D}) unless explicitly stated.
\hat{s}	We use \hat{s} to denote the set of variables assigned values by s . Formally, if $s : V \rightarrow \mathcal{D}$ then $\hat{s} = V \subseteq \mathcal{V}$. Due to the nature of many constraint algorithms, we will assume that there exists an ‘order of assignment’ for \hat{s} , and will define the symbol for this order as needed.
$s \downarrow_V$	We use $s \downarrow_V$ to denote the assignment s projected on to some $V \subseteq \hat{s}$.
\mathcal{S}	The symbol \mathcal{S} refers to the set of partial assignments for a problem. Note that it does include all complete assignments $s : \mathcal{V} \rightarrow \mathcal{D}$, and the empty assignment $s = \emptyset$.
c	In most instances the letter c is used to refer to a constraint. A constraint is seen as a mapping from the set of assignments to a value T or F. If necessary an index such as i or j may be applied to differentiate between constraints. For example, $c_i, c_j \in \mathcal{C}$.
\hat{c}	We use \hat{c} to denote the scope (set of variables) of a constraint c .
$c(s)$	We use $c(s)$ to denote the evaluation of an assignment $s \downarrow_{\hat{c}}$ by the constraint c .
u, v, w	In most instances the letters u , v and w refer to variables. If necessary an index such as i or j may be applied to differentiate between variables. For example $v_i, v_j \in \mathcal{V}$.
d	In most instances this symbol is used to refer to a value. If necessary an index such as i or j may be applied to differentiate between values. For example $d_i, d_j \in \mathcal{D}$.

Pseudocode Notation

Algorithm pseudocode in this thesis will use some keywords and notation beyond the usual ‘if’, ‘for’, ‘while’ and ‘break’. These are presented below, along with a standardised interpretation for other common keywords such as ‘set’ and ‘let’.

algorithm, procedure	The label ‘algorithm’ is used to refer to the main function of a constraint satisfaction algorithm. Component functions such as backtracking and computing nogoods are labelled ‘procedure’ and are numbered accordingly.
when	<p>The term ‘when’ is used to model event-driven programming commonly found in distributed programs. It is assumed that the program pauses at the beginning of a ‘when’ block until one of the conditions is satisfied, and will not exit the ‘when’ block until none of the conditions are satisfied.</p> <p><i>Example:</i> when an assignment $v \rightarrow d$ is received from a neighbour</p> <p><i>Example:</i> when some random amount of time t has passed</p>
let	<p>The term ‘let’ is used to declare variables, often stating their intent and initial value. This is often also used to declare constants, or to define useful terms to simplify formulas.</p> <p><i>Example:</i> let V be a set of variables, initially empty</p> <p><i>Example:</i> let v be the variable most recently added to \hat{s}</p>
set	<p>The term ‘set’ is used to modify variables, describing the new value that they will take. This is most often used to modify functions, but also can be used in other circumstances.</p> <p><i>Example:</i> set $s(v)$ to a value consistent with the assignments in t</p> <p><i>Example:</i> set N to $N \cup \{n\}$</p>
unset	<p>The term ‘unset’ is used to give a variable <i>no</i> value. This is most often used to remove a particular mapping from a function. Note that ‘unset V’ is different from the ‘set V to \emptyset’. That is, if V is unset then $V \neq \emptyset$.</p> <p><i>Example:</i> unset $s(v)$, for all v appearing in \hat{c}</p> <p><i>Example:</i> unset the eliminating explanation $e(v, d)$</p>
'	The decoration ‘ $'$ ’ is used only in algorithm proofs, and not in algorithm bodies. It refers to the <i>next</i> value of a variable. For example, if s refers to the current variable-value assignment, then s' refers to the variable-value assignment after one step or iteration of the algorithm.

Chapter 1

Introduction

Classically, we consider computers as machines which, given a task description, will complete that task on our behalf. Examples of such tasks include database queries, file reorganisation, and more practical examples such as reserving a room at a hotel. As software developers we assume the ‘task description’ contains all necessary information, and the task of the computer is then relatively straightforward.

Artificial intelligence research has recently introduced the concept of ‘agent-based’ systems [WJ95, RN03, FG96]. Agent-based systems do not assume that the ‘task description’ provided by any user is complete, but will instead rely on collaboration between distributed ‘agents’ to gain sufficient information and complete the task. A common example given in agent literature is: dynamically organise transportation and hotel rooms for conference guests. Such a task demands the collaboration of multiple agents to discover and negotiate room availability, transportation options, and, critically, adapt to unforeseen changes. Each agent in the system will have an inbuilt goal and will interact with other agents to achieve it.

Many of the problems faced by agent-based systems bear a remarkable similarity to that of ‘distributed constraint satisfaction’ or ‘distributed constraint optimisation’. Indeed, an algorithm for distributed constraint satisfaction or optimisation can be seen as a simple protocol for handling logic-based multi-agent negotiation [MSTY05, YD91, ML04]. However, agent-based systems are most often designed so that new agents may join any other in negotiation. In contrast, distributed constraint satisfaction algorithms will most often assume that the set of participating agents is fixed, limiting their utility in agent-based systems. Distributed constraint satisfaction research often appears more focused on *parallelism* within a naturally distributed setting, while ignoring or eliminating the option of dynamic problem expansion or the addition of agents. Within this thesis, we hope to develop a distributed constraint satisfaction algorithm without such restrictions.

This chapter will first provide concrete definitions of the Distributed Constraint Satisfaction Problem, and relate it to agent-based systems. At the completion of this chapter we will

also describe criteria to be satisfied by distributed constraint satisfaction algorithms if they are to be suitable for use in agent-based systems. These criteria will be used to construct and judge our new algorithm, Support-Based Distributed Search.

1.1 Constraint Satisfaction Problem

The Constraint Satisfaction Problem (CSP) can be described as: determine an **assignment** of **values** to **variables** that satisfies a set of **constraints**. Constraint Satisfaction Problems can be used to model a wide range of decision problems by an appropriate choice of values, variables and constraints. For example, abstract problems such as SAT can be modelled as propositional variables, boolean values and logical constraints. Practical problems, such as the decision variant of the travelling salesman problem, are also easily and intuitively modelled.

The Constraint Satisfaction and Optimisation Problem (CSOP) can be described as: determine an assignment of values to variables that satisfies a set of constraints but also optimises an **objective**. This simple extension of CSPs is often used to model problems with preferences, such as determining minimal vehicle routings or minimising makespans for scheduling problems. Although the CSOP generalises the CSP, any algorithm for solving CSPs can be adapted to solve CSOPs by iterative application. This is not always a practical approach, but demonstrates that, on a theoretical level, CSP and CSOPs are equivalent.

For this thesis we will only be concerned with the Constraint Satisfaction Problem, as defined below:

Definition 1 Let \mathcal{V} be a set of variables and \mathcal{D} a set of values (called the domain). An **assignment** is a function $s : V \rightarrow \mathcal{D}$ where $V \subseteq \mathcal{V}$. The **scope** of the assignment is the set $\hat{s} = V$. If $\hat{s} = \mathcal{V}$ then s is a **complete assignment**; otherwise it is a **partial assignment**.

Definition 2 Let S_V be the set of assignments with scope $V \subseteq \mathcal{V}$. A **constraint** is a function $c : S_V \rightarrow \{T, F\}$ evaluating whether an assignment is satisfactory (T) or unsatisfactory (F). The **scope** of the constraint is the set $\hat{c} = V$. We say c is an **n -ary constraint**, where $n = |\hat{c}|$.

Definition 3 A constraint satisfaction problem is a triple $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ where:

- \mathcal{V} is a set of variables
- \mathcal{D} is a domain (finite set of totally ordered values)
- \mathcal{C} is a set of constraints

A **solution** to \mathcal{P} is a complete assignment $s \in S_{\mathcal{V}}$ such that $\forall c \in \mathcal{C}, c(s \downarrow_{\hat{c}}) = T$.

Example. A simple example of a CSP, often quoted in literature, is the map-colouring problem. In this problem we are given a map of the world, and must colour each country in such a way that:

- no two bordering countries have the same colour; and
- no more than four different colours are used in all.

In such a problem, the domain \mathcal{D} is the set of colours, and is limited to just four values. The set of variables \mathcal{V} is used to represent the set of countries, with one variable per country. The set of constraints \mathcal{C} will express the requirements that no two bordering countries receive the same colour. The solution to $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ is therefore a mapping of countries to colours satisfying border requirements. ■

1.2 Distributed Constraint Satisfaction Problem

To describe the Distributed Constraint Satisfaction Problem, we must first describe the concept of **agents**. We do not need to provide a formal description as we do for CSPs - there is much disagreement about the concept of ‘agents’ anyway. Popular definitions are provided in [RN03] and [WJ95]: However, a literature survey conducted in [FG96] resulted in the following definition:

An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

This is a relatively simple definition of an agent that would be agreed to by the majority of researchers. Given this definition of agents, we can describe the simplest form of Distributed Constraint Satisfaction Problem (DisCSP) as:

Solve a constraint satisfaction problem where the control of each variable is distributed among agents.

As a general rule, Distributed Constraint Satisfaction Problems occur in situations that are physically different to classical constraint satisfaction problems. It is the nature of the problem, and *not* the choice of the user, that requires us to invoke ‘distribution’ or ‘agents’ rather than regular parallelisation techniques.

In the simplest forms of DisCSP, each agent controls only a single variable, and the ‘agenda’ of each agent is ‘to find an assignment for its variable that is consistent with the assignments chosen by other agents’. Note also that alternative, constraint-centric or multi-variable definitions of DisCSP exist, but the majority of this thesis will be limited to the

simple ‘one-variable-per-agent’ model of DisCSP. While not ideal, it is always possible to re-encode a ‘multiple-variable-per-agent’ DisCSP instance as a ‘single-variable-per-agent’. One mechanism for efficiently re-encoding such problems will be explored in Chapter 6.

1.3 Centralised vs Distributed Algorithms

It is important to note that, modulo the introduction of agents, an instance of the Distributed Constraint Satisfaction Problem still remains an instance of the Constraint Satisfaction Problem. Indeed, the difference between DisCSP and CSP lies primarily in the addition of algorithmic restrictions and not any fundamental change to the problem itself. For example, whereas a CSP algorithm has access to all information, each variable/agent of a DisCSP is aware only of the constraints associated with itself and will coordinate with other variables to find a solution.

We can differentiate between DisCSP algorithms and CSP algorithms by the following:

- A DisCSP algorithm has a notion of ‘information that is local to a variable/agent’. This often includes current assignments of neighbouring variables, and constraints deduced during the execution of the algorithm. It may also include variable orderings (static [HBQ98, BMM01, YDIK92] or dynamic [Yok95]), and information required for communication (such as the physical location of a neighbouring variable). Variables may exchange local information by use of the communication channels.
- A DisCSP algorithm must only use information local to a variable when making variable-value assignments or similar decisions. A CSP algorithm has access to all information at no cost; a DisCSP algorithm must copy information from one variable to another to be able to use it. There is therefore no ‘global’ information in a DisCSP algorithm, and no global decision process; all decisions must be made using only information local to a variable. This differentiates DisCSP algorithms from parallelised instances of regular CSP algorithms, which can generally assume access to shared memory.
- A DisCSP algorithm’s performance is measured using some form of ‘non-concurrent’ measure [MKRZ02, ZM06d]. Depending on the target problem, it may be assumed that some information is already known to all agents and is not counted in the performance of the algorithm. Examples of information which is predetermined include neighbouring variable identifiers and, in some algorithms, a globally consistent variable ordering or the means to determine one [Ham02].

- A DisCSP algorithm makes decisions for each variable concurrently [YH00b]. In many such algorithms a certain degree of synchronisation is assumed, implemented via usual distributed synchronisation schemes.

We can see that a DisCSP algorithm is a regular CSP algorithm, but merely adhering to certain additional requirements on information handling. Viewing DisCSP algorithms in this way, rather than as a communications protocol, will make analysis and comparison simpler.

1.4 Motivation

We will now describe our motivation for the work presented in this thesis, and especially the issues encountered when solving very large or ‘unbounded’ constraint satisfaction problems. We will begin by discussing a realistic but very large DisCSP that in itself illustrates problems with the DisCSP model and existing algorithms. The problem of unbounded constraint satisfaction will be used as motivation for the development of a new algorithm in Chapter 3.

We have already described how Distributed Constraint Satisfaction Problems are formed when the description and solution procedure of a CSP are separated amongst multiple agents. The distributed environment extends the applicability of CSPs to domains such as distributed scheduling and resource contention. Of particular interest is the use of DisCSPs as models for solving other multi-agent problems, with DisCSP algorithms defining a protocol for agent communication. Common examples include scheduling, task assignment, and limited forms of negotiation where simple decision(s) must be made per agent.

In the following chapter we will also review how a DisCSP can be solved by variants of existing global-search or local-search algorithms. For now, it is sufficient to note that local-search algorithms [ZW02] are incomplete, while global-search [BMM01, Ham02, Yok95, YH00a] make use of a total order over variables, or produce additional connections between agents. We would like to avoid both of these attributes, and for good reasons.

It is well-known that local-search algorithms may be incomplete, even when restricted to feasible problem instances. For example, agents in Distributed Breakout may enter a repeating pattern of behaviour which prevents them from finding a solution to a known-feasible DisCSP [ZW02]. While we can draw inspiration from the heuristics embedded in local-search algorithms, we will limit this thesis to situations where a complete algorithm is required.

For many types of multi-agent problems, it is also preferable if agents are not explicitly granted ‘authority’ over any other. When using a total ordering to establish agent behaviour, those agents with ‘higher’ rank will have more ‘authority’ and are less likely to change value than a ‘lower’ ranking agent. In an anytime environment this results in higher-ranked agents

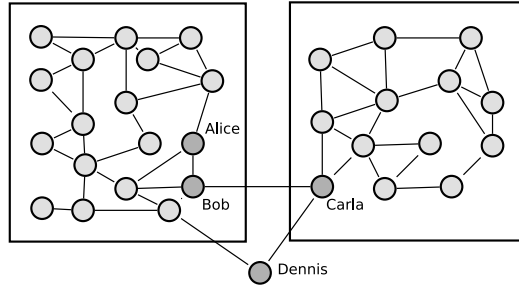


Figure 1.1: It can never be clear which agent should take the greatest burden of search.

being granted more stable answers, and even a potential reduction in concurrency [ZTS04]. While a notion of ‘authority’ may be useful in some situations, we will limit this thesis to situations where explicit ‘authority’ between agents is not desirable.

We also note that it is difficult to add constraints between two previously independent DisCSPs when using a total order. Forming a new connection between independent DisCSPs would require a re-computation of the variable ordering and/or an arbitrary decision on priorities. If a problem is frequently altered by addition of groups of variables, as is likely to occur in large DisCSP networks, this re-computation will become increasingly difficult.

Finally, unbounded addition of new links between agents will result in a DisCSP being effectively reduced to a centralised CSP. Assuming that a DisCSP was created due to the distributed nature of the problem, then merely increasing the amount of communication between nodes cannot be an effective solution. Consider the following examples:

Example. The universities of Pluto and Saturn each use an automated system for scheduling meetings amongst their own staff. Staff give constraints of the form ‘Alice needs to meet with Bob for 2 hours this Wednesday or Thursday’ to agents on their own computers. Individual universities contain a large number of staff with generally sparse connections, so a distributed algorithm is used in which agents communicate directly with each other. Agents are assigned comparable identifiers using finely-tuned schemes specific to each university. These identifiers are chosen to permit backtracking in a distributed global-search algorithm within the university.

Despite best efforts at fairness, a static ordering creates problems between research peers. For example, any trivial change in Alice’s meeting times must always be accepted by Bob. Inversely, Bob may request a change to Alice’s meetings only after exhausting all possible meeting schedules and detecting infeasibility. This problem is distinct from that of preference orderings, and instead relates to stability (Alice’s schedules are more stable than Bob’s).

Worsening matters, Bob at Pluto wishes to arrange a meeting with Carla at Saturn. Their identifiers, while still possibly unique, are not meaningfully comparable for the purpose

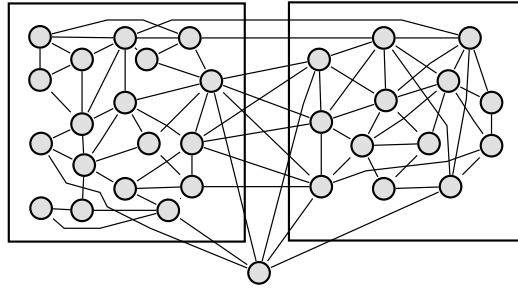


Figure 1.2: Adding links will increase the amount of communication between agents.

of backtracking search. To continue using any existing distributed algorithm we must be able to compare identifiers between agents operating at Pluto and Saturn universities. An example solution is to decide that all Saturn identifiers are ‘greater’ than Pluto identifiers. Unfortunately this would have the same impact on the behaviour of the algorithm as outlined above - meeting schedules for researchers at Pluto would become subservient to those at Saturn. Any changes in meeting times for Saturn researchers, no matter how trivial, must be accepted by Pluto researchers.

Furthermore, any decision for resolving the variable order would require intervention by authorities at each university or the use of a method such as DisAO [BMM01, Ham02]. While this decision could be made for pairs or sets of universities, it does not scale well computationally. For example, if Dennis was an independent researcher he must establish ‘comparability’ with each university and all other researchers. The addition of new researchers frequently raises the possibility of frequent re-computation of variable ordering. ■

Example. A steel manufacturing company has recently installed a system for planning weekly production targets over a one year horizon. This system takes marketing information and machine availability as inputs, and produces approximate targets for each product type. As an isolated system it works well, but is often disassociated from the reality of production on the ground, and so does not necessarily produce truly feasible solutions.

The company has also commissioned software for hour-by-hour plant scheduling with two week horizon. The output of the planning software will form part of the input to this scheduling software. Additionally, it is desirable that any deviations from the weekly plan be communicated back to the planning software. This interaction between scheduler and planner is necessary to ensure that the plan for next week is correctly computed.

Once this interaction has been established, it is expected that a third system will be integrated to perform delivery vehicle routing. This routing package must interact with the scheduling software in a feedback loop, essentially negotiating a feasible manufacturing

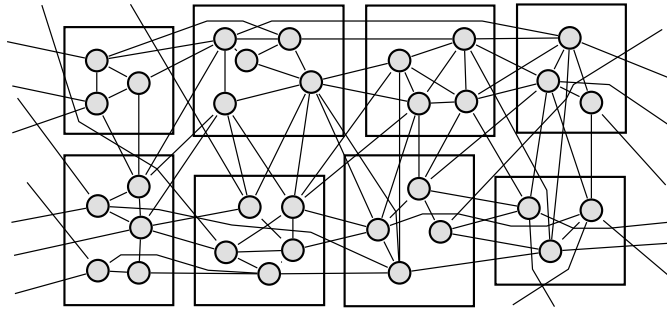


Figure 1.3: Distributed problems should not be treated in isolation, but in a wider network.

schedule and delivery routings. The routing software should not have a direct impact on the planning software, but may implicitly communicate via the scheduling software.

Eventually, this collection of three loosely-coupled software packages will be integrated with the company's rostering software. The rostering software will be expected to interact with all three of the planning, scheduling and routing packages, though in different ways. For example, a yearly plan that reduces a particular product may call for early retirement or retraining of selected personnel. This, in turn, will impact the availability of those personnel in the short term, which then affects the feasibility of the current fortnight's production schedule.

Eventually, it will become desirable to link the planning and scheduling software with the computer systems of various suppliers. Similarly, it will be desirable to link the rostering software with that of various on-site contractors. Of course, the supply and demand for machine parts are necessary will affect and be affected by the yearly plan and weekly schedule.

However, in all of this extensions and system modification, no one software package should be assumed to dominate the others. For example, a steel company cannot forcibly negotiate with a coal mine for the supply of coal; if the coal is not easily available in the quantities desired, then that is the problem of the steel mill. Similarly, a plan may be *possible* by forcing employees to work overtime, but alternative plans should be considered before making such a request.

Further, there is no apparent limit to the number of component systems involved. Such a distributed system could easily encompass the operations of multiple companies. However, it should not be assumed that links should be added between unrelated companies. The company supplying replacement parts to the steel manufacturer should not be negotiating scheduling feasibility with the coal mines. The feasibility issues encountered by a single component system should be handled within a localised manner, rather than passing the issue to other systems. ■

These two examples highlight:

- the problems that arise from using a total order to establish ‘authority’ between agents;
- the problems of maintaining a total order when merging multiple DisCSPs;
- that DisCSPs will *grow* over time, and should not be seen as parallelised CSPs;
- that it is necessary to avoid adding connections between agents within a DisCSP.

The specific difficulties of the above distributed constraint satisfaction problems do not seem to be handled appropriately by existing algorithms. We will attempt to develop an algorithm suitable for the kinds of problems in these examples. We acknowledge that this is only a subset of distributed constraint satisfaction problems, but is one which we feel has not been well addressed by existing algorithms. We can summarise the above issues into the following criteria for development of our algorithm:

- It must not use an explicit notion of ‘authority’ between variables, and so cannot use a total order.
- It must not add links between variables, and so avoids the eventual need for ‘broadcasting’ assignments.
- It must be complete, unlike existing local search algorithms.

Chapter 2

Analysis of Completeness Techniques

In the previous chapter, we mentioned that a (Distributed) Constraint Satisfaction Problem is a form of **decision problem**. We will briefly describe the concepts and terminology used for decision problems, and then apply them to (Distributed) Constraint Satisfaction Problems.

Decision problems are computational problems which map a set of possible inputs to either ‘true’ or ‘false’. For example, a simple decision problem would be: determine whether a collection of numbers contains a duplicate entry. For such a problem, the input would be the collection of numbers, and the output is ‘true’ if and only if a pair of duplicate numbers exists in that collection. It is trivial to develop an algorithm to solve this particular decision problem, though not all are as easy.

Note that it is often also desirable to produce a ‘certificate’ any time an algorithm for a decision problem returns ‘true’. A certificate should provide enough information that we could quickly verify the output. In the case of our simple example problem, a certificate may identify which numbers in the collection are duplicates.

Given the above, we can clearly see that a Constraint Satisfaction Problem is a form of decision problem. The input is any instance \mathcal{P} and the output is ‘true’ if and only if a complete consistent assignment exists. Any complete consistent assignment s would serve as a certificate, though any certificate would suffice.

As the Constraint Satisfaction Problem is a form of decision problem, it is reasonable to use corresponding terminology to describe the correctness of CSP algorithms. We will therefore use the following definitions to describe the soundness, completeness, and correctness of the CSP algorithms that we describe.

Definition 4 *We will say that an algorithm is **sound** if it only terminates with a correct answer. We will say that an algorithm is **complete** if it terminates for all problems. We will say that an algorithm is **correct** if it is both sound and complete.*

In the remainder of this chapter we will demonstrate how algorithms for distributed constraint satisfaction have, in general, evolved as variants of existing centralised algorithms. We will describe the procedures used in both distributed constraint satisfaction algorithms and their centralised ancestors, and identify the process by which they attain, or fail to attain, algorithmic completeness. Note that this chapter will mostly focus on understanding the common and necessary requirements for distributed constraint satisfaction algorithms, and so will assist us in the development of our own distributed constraint satisfaction algorithm.

2.1 Common Proof Theorem

Various methods have been used to prove the correctness of distributed and non-distributed constraint satisfaction algorithms. In general, a proof of soundness is offered first, proving that the algorithm in question does not make a mistake or end in deadlock. This is followed by a proof of completeness, showing that the algorithm eventually terminates regardless of the input.

Many proofs of completeness rely on reduction to a known complete algorithm - often chronological backtracking. However, such proofs do not illustrate the actual mechanism used by an algorithm to attain completeness. While many algorithms reduce to chronological backtracking *eventually*, they may never actually demonstrate such behaviour in practise, and in truth have little or no relation to chronological backtracking.

We will provide a theorem and a resulting common proof structure that uses a more direct or ‘constructive’ approach to proving completeness. Through the resulting proofs we hope to better understand how each algorithm achieves completeness.

Theorem 1 *Assume that an algorithm A has already been proven to be sound, and we want to prove that A is complete. Let I be a set of partial assignments, defined with respect to the current internal state of A as it executes, satisfying the following properties at all times:*

1. *A solution to the constraint satisfaction problem is contained in I .*
2. *For each $s \in I$ where $|\hat{s}| > 1$, there exists $t \in I$ such that $t \subseteq s$ and $|\hat{t}| + 1 = |\hat{s}|$.*
3. *Testing $s \in I$ takes linear-time with respect to the size of the internal state of A .*

If, over time, $|I|$ is convergent to some minimal set satisfying the above conditions, then the algorithm is complete.

Proof. Assume that $|I|$ is convergent to a minimal set as described, but the algorithm A is not complete for some input problem \mathcal{P} . As the algorithm is sound, but not complete, it will never terminate and $|I|$ must eventually reach its lower bound. If I becomes empty then, by

property 1, there is no solution to the problem, and the algorithm has proven that there is no solution to \mathcal{P} .

Alternatively, I must contain at least one solution to the problem (by property 1), and also contains a minimal set of partial assignments (required by property 2). Further, as I is a minimal set satisfying properties 1 and 2, we know that all partial assignments in I are in fact partial *solutions*.

Now, by considering property 2 we know that there exists at least one assignment $s \in I$ such that $|s| = 1$. Searching for such an assignment requires just $|\mathcal{V}| \times |\mathcal{D}|$ tests of whether a singleton assignment is in I , with each test taking equal or less time than A itself. Once an assignment $s \in I$ is found, we can also identify an assignment $s' \supset s$ such that $s \in I$ and $|s'| = |s| + 1$ within the same amount of time. So, finding a complete assignment in I is at most $|\mathcal{V}|^2 \times |\mathcal{D}|$ more than the algorithm's own runtime, and we know that every complete assignment in I is a solution.

We can conclude that, given a known-sound algorithm A and an appropriate definition of a set of partial assignments I , it is possible to produce a solution for \mathcal{P} . Computing the solution takes at most $|\mathcal{V}|^2 \times |\mathcal{D}|$ more than the algorithm's runtime in the *worst* case, which does not alter the (in)tractability of constraint satisfaction problems. \square

All complete search methods work by progressively eliminating portions of the search space. This can be said to be true even of local search techniques that are proven complete on particular classes of problems. In the above theorem, I can be seen as representing those partial assignments that an algorithm has not eliminated from consideration. So, for any given point in the execution of the algorithm, we will say that I is the set of assignments currently **admitted** by an algorithm. The above theorem states that, if a suitable definition of I can be constructed for an algorithm A , and I is convergent to a minimal set, then the algorithm is provably complete. In general, the test of whether a partial assignment is 'currently admitted' should be equivalent to testing whether the partial assignment is 'currently known to be infeasible'. If we construct a definition of I for a given CSP algorithm, matching the requirements of the above theorem, then we will simultaneously prove completeness and, hopefully, gain insight into the mechanism used to achieve it.

Note that Theorem 1 applies equally to distributed and non-distributed constraint satisfaction algorithms. Specifically, the proof will remain valid even if the definition of I for a distributed algorithm violates the conditions outlined in Section 1.2. That is, the proof of 'completeness' is independent of any agent's ability to detect completeness.

Also note that Theorem 1 did not require that a CSP algorithm actually produce a solution to the given problem instance. Rather than demanding that s be output as a 'certificate', we have demanded that enough internal state be maintained by the algorithm to produce s .

Similarity to Other Techniques

Theorem 1 parallels the known technique of using an exponential-time transformation to produce a tractable (but large) problem from an intractable one. In the above theorem, we have effectively stated that a constraint satisfaction algorithm A is complete if:

1. the algorithm's internal state can be used to define a tractable (though potentially very large) decision problem; and
2. over time, the decision problem is refined by the algorithm until it can be used to solve the original, intractable constraint satisfaction problem

That is, an algorithm A is complete if it serves as a transformation from the intractable CSP to a tractable decision problem. While this is not a topic we wish to cover in this thesis, we feel it is worthwhile to point it out.

Also, some readers may see a similarity between Theorem 1 and the concept of model-theoretic semantics for mathematical logics. If we assume that the state of the algorithm corresponds to a logical sentence α , then the set I can be seen as the possible models $M(\alpha)$ for that sentence. An algorithm A can therefore be thought of as a form of logical calculus, progressively building upon the sentence and reducing the set of possible models. This similarity will become more apparent during the remainder of this chapter; proofs of correctness for the definition of I will bear striking similarity to proofs of correctness for model definitions. Again, we note this merely out of interest, and will not focus on this aspect during the thesis.

The remainder of this chapter will look at a number of existing algorithms for constraint satisfaction problems. Each algorithm will be briefly described, with diagrams for the key ideas introduced. The description is followed by a proof of completeness using Theorem 1, and commentary on the algorithm's relationship to preceding algorithms.

2.2 Chronological Backtracking

We will first demonstrate the application of Theorem 1 by proving completeness of Chronological Backtracking (CBT). Chronological Backtracking is a very simple algorithm, progressively extending the current assignment of values to variables. If the current assignment cannot be extended into a solution due to constraint violations, then the algorithm ‘chronologically backtracks’.

A chronological backtrack identifies the most recently assigned variable, and then attempts an untried assignment for that variable. If no such assignment is available, then the variable is left unassigned, and backtracking occurs recursively. If no variables are assigned when backtracking, then no solution exists and the algorithm terminates.

Algorithm 1 provides pseudo-code for Chronological Backtracking. Initially, all variables are placed in a fixed order and are assigned no value (line 1). At each iteration of the algorithm, the current partial assignment s is tested for consistency with the set of constraints \mathcal{C} (line 3). If s was found to be consistent then it is extended by assigning a value for the ‘next’ variable v (lines 4-5). Alternatively, if s was found to be inconsistent then the algorithm ‘backtracks’ to attempt an alternative value (line 7). The algorithm continues iterating until s is extended into a complete consistent assignment (line 2), or until backtrack is impossible.

‘Backtracking’ itself performs an incremental change to the current assignment s , and is described in Procedure 1.1. It assumes, without loss of generality, the existence of a total order on \mathcal{D} . If possible, the value of the most recently assigned variable is incremented (lines 4-5), generating a previously-unseen s . If this is not possible, the variable is unassigned and backtracking is performed recursively (lines 6-7). It is possible that no variables are assigned when backtracking occurs. As backtracking is a sound procedure, this implies that no complete consistent assignment can be found, and the algorithm terminates (line 3).

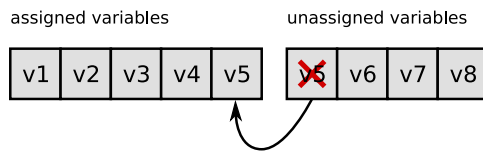


Figure 2.1: CBT: Making an assignment involves taking from the head of statically ordered list of unassigned variables and appending to the tail of an ordered list of assigned variables

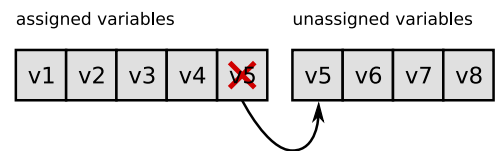


Figure 2.2: CBT: Backtracking an assignment is the exact inverse, taking from the tail of the assigned variables and appending to the head of an ordered list of unassigned variables

It is well established that chronological backtracking (CBT) is sound. That is, CBT never performs a step that would lead to an invalid solution, or to claim that there is no solution when one exists. We therefore need to provide an appropriate definition of I and, using Theorem 1, we can conclude that chronological backtracking is complete.

Algorithm 1 chronological-backtrack-search()

```

1: let  $s$  be the current assignment, initially empty
2: while  $s$  is not a complete and consistent assignment do
3:   if  $s$  is consistent then
4:     let  $v \in \mathcal{V}$  be the least variable (according to  $<$ ) not in  $\hat{s}$ 
5:     set  $s(v)$  to the lowest value for  $v$ 
6:   else
7:     chronological-backtrack ( $s$ )
8:   end if
9: end while

```

Procedure 1.1 chronological-backtrack (s)

```

1: let  $v \in \mathcal{V}$  be the variable most recently added to  $\hat{s}$ 
2: if no such  $v$  exists then
3:   terminate
4: else if there exists a higher value for  $v$  than  $s(v)$  then
5:   set  $s(v)$  to the next higher value for  $v$ 
6: else
7:   unset1 $s(v)$ 
8:   chronological-backtrack ( $s$ )
9: end if

```

We will first assume that a total ordering $<$ on \mathcal{V} is available, and that there exists some natural ordering over \mathcal{D} . From this, we can define a comparator \prec for partial assignments which will be central to our proof of completeness.

Definition 5 *Given the ordering $<$ on \mathcal{V} and assignments $s, t \in \mathcal{S}$, we can say $s \prec t$ iff there exists $v \in \mathcal{V}$ such that:*

- $\forall u \in \mathcal{V}$, if $u < v$ then $s(u) = t(u)$; and
- $v \notin \hat{t}$ or $v \notin \hat{s}$ or $s(v)$ is less than $t(v)$

¹The ‘unset’ command will remove v from \hat{s} . See Notation section for more details.

This definition of \prec may seem complex; it is not an ordering on solutions, and does not appear to encapsulate the methodical elimination of potential solutions. However, it does identify those partial assignments that might be visited by CBT in the future, given the current assignment s . We will use this in the following proof as a means to define the set I , and show it is monotonic decreasing:

Theorem 2 *Chronological Backtracking is complete.*

Proof. Assume that the CBT is sound, but not complete, and let s denote the current assignment during execution. Let $I = \{t \in \mathcal{S} : s = t \vee s \prec t\}$, so that it is defined with respect to the current assignment. It may not be immediately obvious, but I now describes the set of all partial assignments that have not yet been pruned by the algorithm.

We will use $'$ to indicate *next* or *future* values of variables during the execution of an algorithm. So, let s' be the assignment of values to variables in the *next* iteration of the algorithm, and $I' = \{t \in \mathcal{S} : s = t \vee s' \prec t\}$ be the corresponding next set of assignments. We will prove that our definition of I satisfies the conditions of Theorem 1.

1. We must prove that I is convergent to some minimal set while the algorithm executes.

We begin by considering any $t \in I'$ (so either $s' = t$ or there is some $v \in \mathcal{V}$ satisfying $s' \prec t$), and proving that $s \prec t$. We then consider the two possible operations of CBT - extension of s by assigning some new variable, or backtracking on some variable.

First, assume that the current iteration extended s to s' by assigning a new variable $w \in \mathcal{V} - \hat{s}$, using the smallest value possible. If $s' = t \downarrow_{\hat{s}}$, then either $s = t$ or the variable w will satisfy the relation $s \prec t$. If $s' \neq t \downarrow_{\hat{s}}$ then $s' \prec t$, and whichever $v \in \mathcal{V}$ satisfied the relation $s' \prec t$ must also satisfy $s \prec t$. In either case, $t \in I'$ implies that $t \in I$.

Alternatively, assume that the current iteration involved backtracking and incrementing the value for a variable $w \in \hat{s}$, so that $s'(w)$ is greater than $s(w)$. If $s' = t \downarrow_{\hat{s}}$, then w itself will clearly satisfy the relation $s \prec t$. If $s' \neq t \downarrow_{\hat{s}}$ then $s' \prec t$, and whichever $v \in \mathcal{V}$ satisfied the relation $s' \prec t$ must also satisfy $s \prec t$. In either case, $t \in I'$ implies that $t \in I$.

Clearly, regardless of what CBT does in each iteration, $t \in I' \Rightarrow t \in I$, and so $I' \subseteq I$. However, in a backtrack iteration, $s \in I$ but $s \notin I'$, so we can further conclude that $I' \subset I$ after a finite number of iterations. As I is finite and is monotonic decreasing (with respect to set inclusion) over the execution of CBT, it must converge to a minimal set.

2. We must prove that condition 1 of Theorem 1 is satisfied.

Consider the situation where a solution $t \in I$ but $t \notin I'$ due to the change in current assignment from s to s' . By the definition of I and \prec we know that:

- a variable-value assignment was incremented in s' (i.e. we backtracked); and
- $s = t \downarrow_s$ (i.e. the current assignment is a prefix of the solution t)

However, a variable-value assignment will only be incremented in s' if s is found to be infeasible. If t is a solution and $s \subseteq t$, then s must be feasible, and so by contradiction $t \in I'$. We have therefore proven that all solutions in I are also in I' . As I is initially equal to \mathcal{S} it must include all solutions of the problem, and so I' must also include all solutions. Therefore the definition of I will satisfy condition 1 of Theorem 1.

From the above, we have shown that our definition of I is convergent to a minimal set, and satisfies condition 1 of Theorem 1. By the definition of \prec it is also clear that I satisfies conditions 2 and 3 of Theorem 1. From these results and Theorem 1 we can conclude that CBT is complete. \square

The above proof shows that CBT progressively eliminates the set of partial assignments, expressed formally as $I' \subset I$. Unfortunately, the proof itself is not very informative, except to confirm that CBT is monotonic in its pruning of possible assignments. However, the definition of I and \prec provide much greater insight into the functioning of CBT.

Definition 5 can be rewritten less formally. A given t satisfies $s \prec t$, and so is *not* pruned by CBT, if and only if it satisfies one of the following:

1. t is a prefix of s ; or
2. s is a prefix of t ; or
3. s and t share a common prefix, and the next variable outside that prefix is assigned a lesser value by s than by t

The ‘prefix’ requirement on t , and the knowledge that $I' \subset I$, highlights the methodical approach taken by Chronological Backtracking. Any assignment which does not share a prefix with s is immediately eliminated from consideration, and will not be readmitted. CBT is therefore an algorithm which iterates over possible solution prefixes, eliminating sections of the search space.

We can see that the performance of Chronological Backtracking will be dependent on the variable ordering, as backtracking with smaller $|s|$ will reduce $|I|$ more rapidly. This is confirmed in literature where, for CSPs with few or no solutions, a fail-first variable ordering heuristic is known to be highly effective.

2.3 Chronological Backtracking with Reordering

Chronological Backtracking and its completeness proof are highly dependant on the fixed nature of the $<$ ordering on \mathcal{V} . However, it is trivial to adapt our proof structure to algorithms

where the ordering on \mathcal{V} is not fixed. For this purpose, we consider a simple extension to Chronological Backtracking, permitting variable reordering at selected points.

Algorithm 2 chronological-backtrack-search-with-variable-reordering()

```

1: let  $s$  be the current assignment, initially empty
2: while  $s$  is not a complete and consistent assignment do
3:   if  $s$  is consistent then
4:     let  $v \in \mathcal{V}$  be any variable not in  $\hat{s}$ 
5:     set  $s(v)$  to the lowest value for  $v$ 
6:   else
7:     chronological-backtrack ( $s$ )
8:   end if
9: end while

```

Procedure 2.1 chronological-backtrack (s)

```

1: let  $v \in \mathcal{V}$  be the variable most recently added to  $\hat{s}$ 
2: if no such  $v$  exists then
3:   terminate
4: else if there exists a higher value for  $v$  than  $s(v)$  then
5:   set  $s(v)$  to the next higher value for  $v$ 
6: else
7:   unset  $s(v)$ 
8:   chronological-backtrack ( $s$ )
9: end if

```

This algorithm differs from Chronological Backtracking only in the variable selection; rather than using a fixed order, any variable may be chosen to extend the assignment s . Therefore, to prove completeness for Chronological Backtracking with Reordering (CBT-R) we must

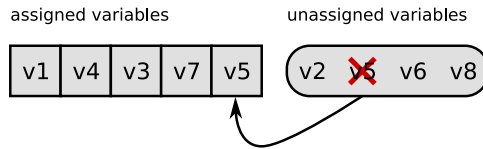


Figure 2.3: CBT-R: Making an assignment involves choosing a variable from an unordered list of unassigned variables and appending to the tail of an ordered list of assigned variables

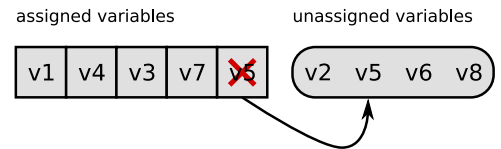


Figure 2.4: CBT-R: Backtracking an assignment is the exact inverse, taking from the tail of the assigned variables and appending to an unordered list of unassigned variables

provide a dynamic definition of the ordering $<$ on \mathcal{V} . Note that this change also requires a minor alteration to the definition of \prec as it is based on $<$.

Definition 6 Assume that s is the current assignment held by the algorithm at any timepoint. We define the ordering $<$ on \mathcal{V} as ‘the order in which variables were most recently added to s , with unassigned variables last’. That is $v_1 < v_2$ iff:

- $v_1, v_2 \in \hat{s}$ and v_1 was added to \hat{s} earlier than v_2 ; or
- $v_1 \in \hat{s}$ and $v_2 \notin \hat{s}$

As with Chronological Backtracking, we can then say $s \prec t$ iff there exists $v \in \mathcal{V}$ such that:

- $\forall u \in \mathcal{V}$, if $u < v$ then $s(u) = t(u)$; and
- $v \notin \hat{t}$ or $v \notin \hat{s}$ or $s(v) < t(v)$

From the above pseudo-code and definitions we can see that Chronological Backtracking with Reordering has the flexibility to progress through the set of solution prefixes in a substantially different way to that of Chronological Backtracking. It is able to, at certain points during execution, alter the ordering of variables and, consequently, alter the definition of \prec . However, it still can be seen to use the same pruning procedure of Chronological Backtracking. Neither of these alterations has any impact on the proof of completeness; using the above definition of \prec , the proof of completeness of CBT-R would be identical to that of CBT.

We can therefore conclude that, despite the addition of variable reordering, CBT-R also has the same core characteristic of CBT. Partial assignments that are not admitted at one iteration will never be readmitted at any subsequent iteration. Essentially, the ‘pruning’ of the search space remains monotonic. However, CBT-R does permit more advanced heuristics for variable ordering, either through online learning or instance-specific information.

We have now demonstrated the application of Theorem 1 to the simplest of constructive search algorithms. We have been able to identify the method by which it achieves completeness by considering how the set of ‘admitted’ assignments is reduced over time. This investigation has been, by necessity, a drawn out process as a means to demonstrate Theorem 1 more completely. However, the following sections will apply the same technique to more advanced algorithms, and help us identify similarities and differences in their approach to completeness.

2.4 Dynamic Backtracking

Dynamic Backtracking (DBT), introduced in [Gin93], is a modified backtracking search algorithm where the ordering of variables may be changed, including some of those which are currently assigned. Such reordering is not possible within CBT or CBT-R, but is made possible in DBT by retaining ‘eliminating explanations’ for each backtracked value assignment. These additional data structures are used to temporarily record the reason, if any, that a particular value cannot be chosen for assignment. This more complex method of backtracking hopes to improve over CBT and CBT-R by retaining as many assigned values as possible [Gin93, JG93, Bak94].

Algorithm 3 describes the main loop of Dynamic Backtracking. Initially, an empty partial assignment s is constructed (line 1). Similarly, the ‘eliminating explanation’ for each variable-value pair are unset² (line 2). At each iteration of the algorithm, a variable-value pair which are not currently eliminated are selected and assigned (lines 4-5). The current partial assignment s is then tested for consistency with the set of constraints (line 6). If s was found to be inconsistent, then a new eliminating explanation is recorded (line 7). At this point, the variable must be backtracked to restore s to a consistent assignment (line 8). As with Chronological Backtracking, the algorithm will continue iterating until s is extended into a complete consistent assignment.

The dynamic backtrack procedure itself, shown in Procedure 3.1, attempts to restore s to a consistent assignment by unassigning a variable and updating any eliminating explanations. The variable v is the focus of the backtrack and is unassigned immediately (line 1). If the entire domain of v has been eliminated, then the backtrack procedure continues (line 2). Backtracking first constructs the set of preceding variables E which are causing the elimina-

²See the Notation page at the start of this thesis for an explanation of ‘unset’

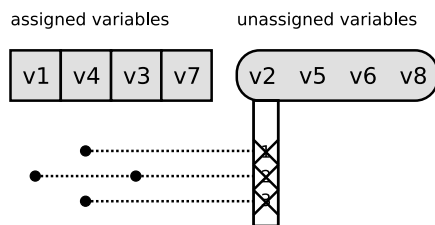


Figure 2.5: DBT: ‘Eliminating explanations’ tell us which values are unusable, based on current values for specific assigned variables.

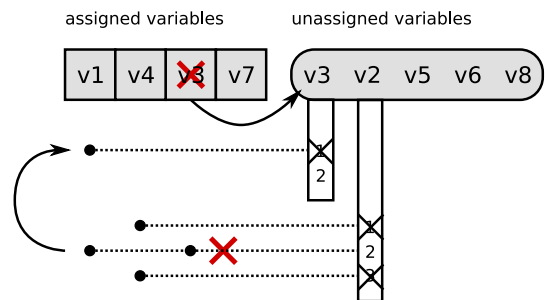


Figure 2.6: DBT: When all values are eliminated, the last-assigned variable causing any of the eliminations is unassigned, and a new eliminating explanation constructed.

tions (line 3). If E is empty, then there is no reason for the elimination of all values except for the constraints themselves, and so the problem is determined to be unsatisfiable (lines 4-6). Otherwise, the most recently assigned variable $w \in E$ is selected as the ‘culprit’, and will be the focus of a recursive backtrack (line 7). Any eliminating explanations that were dependant on w are removed, as they will no longer be relevant after w is unassigned (line 8). A new elimination explanation is constructed for the current value of w , thus ensuring that it does not revisit that value until some preceding variable changes value (line 9). The dynamic backtrack procedure is then called recursively to actually unassign w (line 10).

Algorithm 3 dynamic-backtrack-search()

```

1: let  $s$  be the current assignment, initially empty
2: let  $e(v, d)$  be a set of variables preventing  $v$  from taking value  $d$ , initially unset
3: while  $s$  is not a complete and consistent assignment do
4:   let  $v$  be any variable in  $\mathcal{V} - \hat{s}$ 
5:   set  $s(v)$  to any value such that  $e(v, s(v))$  is not set
6:   if  $s$  is inconsistent then
7:     set  $e(v, s(v))$  to those variables in  $\hat{s} - \{v\}$  causing the inconsistency
8:     dynamic-backtrack ( $v$ )
9:   end if
10: end while

```

Procedure 3.1 dynamic-backtrack (v)

```

1: unset  $s(v)$ 
2: if  $e(v, d)$  has been set for all  $d \in \mathcal{D}$  then
3:   let  $E$  be the union of  $e(v, d)$  for all  $d \in \mathcal{D}$ 
4:   if  $E$  is empty then
5:     terminate
6:   end if
7:   let  $w$  be the variable in  $E$  that was most recently added to  $\hat{s}$ 
8:   unset all  $e(u, d)$  where  $u \in \mathcal{V}$ ,  $d \in \mathcal{D}$  and  $w \in e(u, d)$ 
9:   set  $e(w, s(w))$  to  $E - \{w\}$ 
10:  dynamic-backtrack ( $w$ )
11: end if

```

In the above description we represent ‘eliminating explanations’ as a set of variables $e(v, d)$ for each variable v and value d . If $e(v, d)$ is set then it is interpreted as “ v may not take on value d until one of the variables in $e(v, d)$ is unassigned”. We can also represent an eliminating explanation as a nogood:

Definition 7 Given an assignment s , we define the **nogood** n matching $e(v, d)$ as:

$$n = s \downarrow_{e(v, d)} \cup \{v \rightarrow d\}$$

A nogood can be considered to be a partial assignment that cannot be extended into a complete solution. That is, a nogood is a (minimal) infeasible partial assignment. We will use this concept of nogoods in our proof of completeness for Dynamic Backtracking.

Theorem 3 *Dynamic Backtracking is complete.*

Proof. Let s denote the current assignment in an execution of Dynamic Backtracking Search. Let N be the set of nogoods derived from the explanations e and assignment s . Let $I = \{t : \forall n \in N, n \not\subseteq t\}$ be a set of assignments, defined by N . Intuitively, I is all those assignments which Dynamic Backtracking Search has no eliminating explanation.

For this proof we will introduce the set J , representing the set of assignments known to have been visited and eliminated by the algorithm. We will prove that $|J'| > |J|$ and $I \cap J = \emptyset$ following either a new variable assignment or a backtrack. Note that J is a set constructed solely to prove that $|I|$ is convergent, and does not exist in the algorithm explicitly.

1. We must prove that I is convergent to some minimal set.

When an assignment is added to form s' , no explanations are modified, and thus no nogoods are added to N' . Therefore, $I' = I$ and, as no assignment has been eliminated, we will keep $J' = J$. However, assuming that DBT is not complete for the current problem instance, we know that a backtrack must occur within a finite number of additions. We will consider the impact on I' and J' when a backtrack occurs to a variable v , proving that $|J'| > |J|$ and $I' \cap J' = \emptyset$.

For the purposes of this proof, we will let $J' = J \cup \{s\}$ to record that the assignment s has been visited and must never occur again. As v is the variable to be unassigned, the algorithm constructs a new eliminating explanation $e'(v, s(v))$ containing a subset of variables preceding v in \hat{s} . This eliminating explanation will be represented as a new nogood $n' = s \downarrow_{e'(v, s(v)) \cup \{v\}} \in N'$. We will say that the nogood n' *protects* the assignment $s \in J'$ as the relation $n' \subseteq s$ ensures that $s \notin I'$ and thus $J' \cap I' = \emptyset$.

However, each backtrack also has the potential to delete such eliminating explanations, which could conceivably reduce the set of nogoods. Assume that, at some future point in the algorithm, the algorithm backtracks on a variable w in the eliminating explanation $e(v, s(v))$ and so $e'(v, s(v))$ must be unset. This would implicitly remove the nogood n from N' , and so intuitively could expand I' so that $I' \cap J' \neq \emptyset$. However, we know that the backtrack on w will create a new eliminating explanation $e'(w, s(w))$,

and this will cause a new nogood m to be added to N' . We also know from the algorithm pseudo-code that this must be the first backtrack on w since the time that n was created, and w must have always preceded v during the intervening time. Therefore, the nogood m will be a subset of any assignment protected by n , and so will continue to protect them. Note that the new nogood m will not be a subset of the old nogood n' , but this is not necessary to ensure that $I' \cap J' = \emptyset$. We can still be assured that each assignment $t \in J'$ will be protected by some nogood after a backtrack.

The end result is that every backtrack ensures $|J'| = |J| + 1$ while guaranteeing that $J' \cap I' = \emptyset$. As $|I| + |J|$ must have a fixed upper-bound (\mathcal{D}') we are guaranteed that $|I|$ converges to some lower bound.

2. As N contains only assignments that are proven to be infeasible, we know that I satisfies conditions 1 and 2 of Theorem 1. We are also guaranteed that testing membership of I is linear in the size of $|N|$, and so condition 3 is satisfied.

From these results and Theorem 1 we can conclude that DBT is complete. □

As this proof shows, the operation of Dynamic Backtracking is substantially different from either CBT or CBT-R. Both CBT and CBT-R guaranteed that an assignment that was not admitted at one time will never be re-admitted at a future time; DBT does not.

This proof provides substantial insight into the actual operation and expected behaviour of Dynamic Backtracking. On casual inspection it may appear that ‘explanations’ contain substantial information about the search space and eliminate a significant portion of \mathcal{S} . However, the above proof demonstrates that an explanation is *only guaranteed to eliminate the assignment that was current at the time it was created*. If a large current assignment is maintained, then each explanation will eliminate only a very small portion of the search space. Interestingly, part of the motivation for Dynamic Backtracking was to maintain as much of the current assignment as possible. This leads to the *potential* for a slower reduction in $|I|$ than observed in either CBT or CBT-R.

The potentially poor performance of Dynamic Backtracking was observed in [JG93], where it was shown that Dynamic Backtracking performed worse than a simple Chronological Backtracking on graph colouring problems. In [Gin93], the poor performance was attributed to the tendency of Dynamic Backtracking to maintain a large current assignment. Similar analysis was conducted in [Bak94], which also concluded that Dynamic Backtracking is likely to maintain assignments without good heuristic justification. These empirical observations correspond to the expected behaviour from the above proof. If a large current assignment is maintained, then each eliminating explanation runs the risk of being less effective than it may first appear.

2.5 Weak Commitment Search

Weak Commitment Search (WCS), presented in [Yok94] and [YH00b], appears superficially similar to backtracking search procedures. Assignments are attempted and, if found inconsistent, they are ‘backtracked’ and new assignments are tried. However, WCS differs from each of the previous algorithms by backtracking *all* variables and recording a nogood. By using an unbounded nogoods store WCS is sure to never revisit a partial assignment that was previously tried. WCS then takes advantage of the recently failed assignment as a guide for min-conflict value selection. The result is an algorithm which is complete, but can make significant use of the min-conflict heuristic for performance.

Algorithm 4 describes Weak Commitment Search in its entirety. It is interesting to note that it is simpler than any algorithm currently reviewed, and has striking similarities to even simpler algorithms, such as generate-and-test.

Initially, an empty set of nogoods N and an empty partial assignment s are constructed (lines 1-2). A new *complete* assignment t is also constructed (line 3). We will refer to t as the ‘tentative’ assignment, as it has no requirement for consistency. At all times, each variable has a value in just one of s or t .

At every iteration, a variable v is chosen from the current set of tentative assignments (line 5). A value d for v is then selected such that it is consistent with the current partial assignment s and maximises consistency with the current tentative assignment t . Note that ‘consistent’ in this context means ‘consistent with respect to all constraints and all nogoods’, and ‘maximise consistency’ normally refers to maximising the number of satisfied constraints and nogoods.

If such a value can be found, then the assignment is added to the current partial assignment, and the variable removed from the tentative assignment (lines 7-9). If such a value cannot be found, but the current partial assignment is empty, then the problem is unsatisfi-

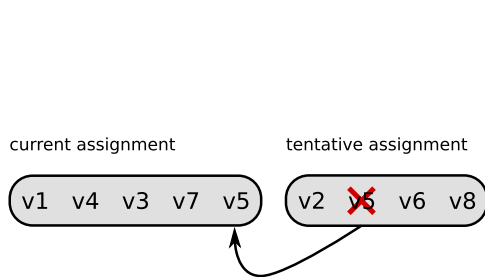


Figure 2.7: WCS: Variables are assigned and partitioned into ‘current consistent’ and ‘tentative inconsistent’. Tentatively-assigned variables are iteratively given consistent assignments.

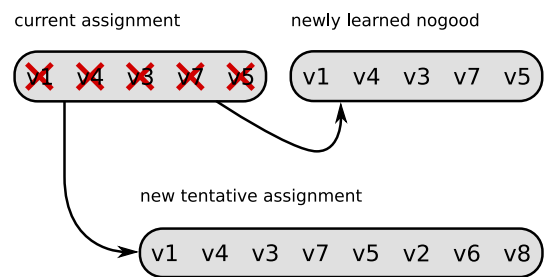


Figure 2.8: WCS: When a tentatively-assigned variable cannot be given a consistent assignment, the current assignment is transformed into a nogood and all assignments become tentative.

able and the algorithm terminates (lines 11-12). Otherwise, the current assignment has been shown to be infeasible and is recorded as a nogood (line 14). The current assignment is appended to the tentative assignment, and then erased (lines 15-16). This final step ensures that WCS has the greatest freedom to assign variables and values on subsequent iterations.

Algorithm 4 weak-commitment-search()

```

1: let  $N$  be a set of nogoods, initially empty
2: let  $s$  be the current assignment, initially empty
3: let  $t$  be a ‘tentative’ assignment, initially complete
4: while  $s \cup t$  is not consistent do
5:   let  $v$  be some variable in  $\hat{t}$ 
6:   let  $d$  be some value for  $v$  consistent with  $s$  and maximising consistency with  $s \cup t$ 
7:   if  $d$  exists then
8:     unset  $t(v)$ 
9:     set  $s(v) = d$ 
10:  else
11:    if  $s$  is empty then
12:      terminate
13:    end if
14:    set  $N$  to  $N \cup \{s\}$ 
15:    set  $t$  to  $t \cup s$ 
16:    set  $s$  to  $\emptyset$ 
17:  end if
18: end while

```

WCS is the first algorithm we have investigated which uses an unbounded nogood store. This makes the proof of completeness relatively trivial, as a monotonic-increasing nogood store ensures a monotonic-decreasing I .

Theorem 4 *Weak Commitment Search is complete.*

Proof. Let s denote the current assignment in an execution of Weak Commitment Search. Let N be the current set of nogoods, and t the set of ‘tentative’ assignments. Let $I = \{t : \forall n \in N, n \not\subseteq t\}$ be a set of assignments, defined by the current set of nogoods. We will prove that $N' \supset N$ following a finite number of iterations.

When an assignment is added to form s' , no nogoods are added to N' . Note that the resultant s' is consistent (according to N) and so $s \notin N'$. Assuming that Weak Commitment Search is not complete for the current problem instance, we know that after at most $|\mathcal{V}|$ iterations we will find no assignment is possible. When no assignment is possible, s is added

to N' , and so $N' \supset N$. By the definition of I we can see that $|I'| < |I|$ and so I is convergent to some minimal set.

As I is determined by the nogoods N , and each nogood is only generated by an inconsistency, we can be sure that I and I' contain all solutions. Further, I clearly contains all partial solutions, and membership is testable in time linear in $|N|$. Therefore I satisfies the conditions of Theorem 1, and we have proven that it will converge. From these results and Theorem 1 we can conclude that WCS is complete. \square

Weak Commitment Search, in some sense, generalises Dynamic Backtracking by permitting changes to all assignments after encountering a new inconsistency. By retaining the previous assignment it also provides more heuristic information for value selection. This directly addresses the problems inherent in Dynamic Backtracking, where heuristic guidance was prevented by the backtracking mechanism. Unfortunately, these additional freedoms are only made possible by the use of an unbounded nogood store.

The unbounded nogood store also ensures that information is not lost as is the case in Dynamic Backtracking. Therefore, WCS does not have the same potential poor performance as DBT, albeit at the cost of maintaining an exponential number of nogoods.

We have now demonstrated the application of Theorem 1 to two advanced centralised search algorithms. We have identified two distinct means for achieving completeness; careful and complex maintenance of memory, and the use of an unbounded nogood store. The advantages and disadvantages of each are made apparent within the corresponding completeness proofs; DBT risks terrible worst-case performance, while WCS risks exponential growth in the nogood store. The following sections will apply the same analysis technique to distributed algorithms.

2.6 Asynchronous Backtracking

We will now introduce our first explicitly distributed algorithm - Asynchronous Backtracking. This algorithm was first presented in [YDIK92], and similar algorithms have been presented in [BMM01, BMBM05, Ham05, Ham02, HBQ98, ZM05b], though in some cases the similarity was only recognised well after publication. A ‘concurrent’ variation, which balances workload by exploring multiple assignments simultaneously was also proposed in [ZM06c]. For space considerations, we will not consider each variant algorithm separately. This review and proof will be treated as sufficient for all such distributed backtracking algorithms, with the exception of ABT-DO.

To facilitate explanation of distributed algorithms we will introduce some commonly used terms, and relate those to our previous description of distributed search algorithms:

- agent** An agent encapsulates all information local to a variable, and makes decisions accordingly. Algorithms such as *asynchronous-backtrack-search* are executed by each agent separately and in parallel. We will assume that each agent owns one variable and each variable is owned by one agent, so will often use the terms interchangeably.
- neighbour** A neighbour is an agent which is connected to the current agent by a communication channel. At the start of each algorithm we assume that agents are connected iff their variables share a constraint. It is possible that an agent opens a communication channel and thus has a new neighbour, though this may only be done if it has the identifier of the agent. It is assumed that, when an agent adds another agent as neighbour, current assignments will be exchanged as if the two agents were always neighbours.
- view/nogoods** An agent holds a partial (and possibly incorrect) set of assignments for other variables, called it's view. Typically, the view will contain the most recent assignments from all neighbours. Similarly, an agent will normally accumulate or generate inferences in the form of nogoods. Each agent must therefore maintain its own, separate store of nogoods.

Within our algorithm description and pseudo-code, we will refer to the agent executing the algorithm as 'this agent' or *self*. Asynchronous Backtracking can then be described as follows. Initially, each agent assigns a value to its variable and communicates that information to neighbours (lines 1-3). When an updated assignment is received from a neighbour, it is incorporated into the current view (lines 5-7). Nogoods which are no longer current are then erased, and a new value is selected for this agent's variable (lines 8-9). When a nogood

variables in order

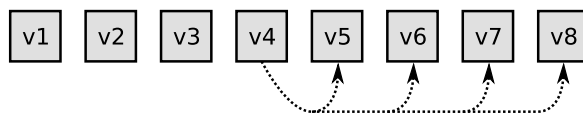


Figure 2.9: ABT: Changes in assignment are broadcast from each agent to each lower-ranked agent.

variables in order

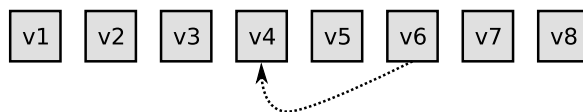


Figure 2.10: ABT: Nogoods only contain assignments from higher-ranked agents, and are sent to the lowest-ranked agent.

is received from a neighbour, it is incorporated into the set of nogoods and a new value is selected for this agent's variable (lines 11-13).

Algorithm 5 asynchronous-backtrack-search ()

```

1: set  $view(self)$  to some value
2: set  $nogoods$  to  $\{\}$ 
3: send a message  $\langle self, view(self) \rangle$  to all neighbours
4: while true do
5:   when a message  $\langle w, e \rangle$  is received
6:     if  $view(w)$  is not  $e$  then
7:       set  $view(w)$  to  $e$ 
8:       set  $nogoods$  to  $\{n \in nogoods : w \notin \hat{n}\}$ 
9:     end if
10:  when a nogood  $n$  is received
11:    set  $nogoods$  to  $nogoods \cup \{n\}$ 
12:    make every agent in  $\hat{n}$  a neighbour
13:  end when
14:   $select\_value()$ 
15: end while

```

Procedure 5.1 select-value ()

```

1: if  $view$  is not consistent then
2:   unset  $view(self)$ 
3:   while no value for  $self$  is consistent with  $view$  do
4:      $compute\_nogood()$ 
5:   end while
6:   set  $view(self)$  to some value making  $view$  consistent
7:   send a message  $\langle self, view(self) \rangle$  to all lower neighbours
8: end if

```

Procedure 5.2 compute-nogood ()

```

1: let  $n$  be a minimal subset of  $view$  causing the domain wipeout
2: if  $n$  is empty then
3:   terminate
4: end if
5: let  $u$  be the lowest variable in  $\hat{n}$ 
6: send  $n$  to  $u$ 
7: unset  $view(u)$ 

```

Selection of a value for the agent's variable is straightforward. First, if *view* is consistent, then the current value needn't be changed (line 1). Otherwise, the agent's variable is unassigned and a search is made for a consistent value (lines 2-3). If no value can be found then it must be caused by the current assignments of higher-ranked variables, and so a nogood is generated (line 4). Until a consistent value can be found, nogoods will continue to be produced. Once a value is found it is assigned, and all lower-ranked neighbours are notified (lines 6-7).

Construction of a nogood is also straightforward. First, the algorithm identifies a minimal set of assignments which are causing the inconsistencies discovered previously (line 1). If there are no assignments causing the inconsistencies, then the problem is unsatisfiable and the algorithm terminates (lines 2-3). Otherwise, the lowest-ranked variable in the nogood is identified (line 5). The current assignment for this variable is removed from the agent view, in the hope that it will permit a consistent assignment (line 6). The constructed nogood is then transmitted to the identified variable, forcing it (or other higher-ranked variables) to change value (line 7).

Theorem 5 *Asynchronous Backtracking is complete.*

Proof. Let s denote the current global assignment to all variables during an execution of Asynchronous Backtracking, so $s(v)$ is simply defined to be that value observed by the agent v itself. Similarly, let N be the global union of all nogoods held by each agent. Let I be the set of consistent partial assignments, defined by the current set of nogoods and constraints. Note that the consistency of a partial assignment can be theoretically determined in polynomial time, though in practise no agent would have access to all the necessary information.

For this proof we will introduce the set J , representing the set of assignments known to have been visited and eliminated by the algorithm. We will prove that $|J'| > |J|$ and $I \cap J = \emptyset$ following either a new variable assignment or a backtrack. Note that J is a set constructed solely to prove that $|I|$ is convergent, and does not exist in the algorithm explicitly.

First, we will expand J during the execution of the algorithm as follows. At any point in time where an agent constructs a nogood n , we will add the current assignment s to J' . As $n \subseteq s$, and the nogood n is newly created, we are guaranteed that $s \notin I'$. Therefore, at any point where an agent constructs a nogood n , we are guaranteed that $|J'| > |J|$ and that $I' \cap J' = \emptyset$. Further, for each assignment $t \in J$, we will say that t is *protected* if some nogood or constraint exists that shows that t is inconsistent.

Second, we note that an agent in Asynchronous Backtracking must detect an inconsistency before being able to change its value. Assume that v is the agent/variable that detects an inconsistency, either by constraints or by the use of nogoods it received from variables

$w > v$. We know that the inconsistency detected by v involves only itself and variables $u < v$, by virtue of the way the algorithm shares information.

Now, if v changes its value as a result of detecting an inconsistency, then there is the possibility that some nogood n held by a variable $w > v$ may be deleted. As with Dynamic Backtracking, this deletion is a cause for concern; n was used to ‘protect’ some assignments in J , and those assignments must remain protected. However, recall that v will *only* change value if it detects an inconsistency. The inconsistency may arise by v receiving a nogood, or by a constraint violation when a variable $u < v$ changed value. In either case, some variable $u \leq v$ is aware of an inconsistency, either by constraint or nogood, that would continue to protect assignments in t that were previously protected by the deleted nogood n . By letting $J' = J$, we are guaranteed that all assignments in J' are protected, and $|J| \leq |J'|$.

We have therefore proven that the construction of a nogood increases the size of J , while a change in assignment will not decrease J . In either case, $I \cap J = \emptyset$. Further, under the assumption that ABT is not complete for the current problem instance, we know that a nogood must be generated eventually. Therefore, after a finite time, $|J| < |J'|$, and as $|I| + |J|$ must have a fixed upper-bound, we are guaranteed that $|I|$ converges to some lower bound.

Also, the definition of I clearly satisfies all conditions of Theorem 1 by virtue of relying only on ‘consistency’. From these results and Theorem 1 we can conclude that ABT is complete. \square

Asynchronous Backtracking may be considered a simple translation of Chronological Backtracking to a distributed setting. Like Chronological Backtracking it depends on a fixed ordering $<$ on \mathcal{V} . However, it also shares much in common with Dynamic Backtracking Search, as evidenced by the above proof. In particular, the set of admitted solutions is not necessarily strictly monotonic decreasing; significant work may be performed and then lost by an aggressive nogood deletion strategy. The relation of Asynchronous Backtracking to the concepts in Dynamic Backtracking have been covered in detail [BMM01, BMBM05].

However, Asynchronous Backtracking will not suffer from the pathological worst case behaviour of Dynamic Backtracking. While work may be lost, the use of a fixed total order on variables ensures that nogoods are inferred over progressively smaller sets of variables. Each agent effectively sees a smaller set of assignments, avoiding the problematic behaviour of DBT whenever a large partial assignment is maintained. Curiously, the end result is that Asynchronous Backtracking produces a similar ‘progression’ through the set of assignments as Chronological Backtracking, while using nogoods in a similar fashion to Dynamic Backtracking.

2.7 Asynchronous Backtracking with Dynamic Ordering

Asynchronous Backtracking with Dynamic Ordering [ZM05b, ZM06a] can be considered a straightforward extension of Asynchronous Backtracking. The primary difference is the ability to reorder variables, similar in many ways to Dynamic Backtracking. We will present the pseudocode for ABT-DO, but will note that the proof of completeness for ABT applies equally well to ABT-DO without any alteration. We have highlighted those sections of ABT-DO which are different from ABT by the use of dotted underlines.

Algorithm 6 asynchronous-backtracking-dynamic-ordering ()

```
1: set  $view(self)$  to some value
2: set  $order$  to some default global ordering over all variables
3: set  $nogoods$  to  $\{\}$ 
4: send a message  $\langle self, view(self) \rangle$  to all neighbours
5: while true do
6:   when a message  $\langle w, e \rangle$  is received
7:     if  $view(w)$  is not  $e$  then
8:       set  $view(w)$  to  $e$ 
9:       set  $nogoods$  to  $\{n \in nogoods : w \notin \hat{n}\}$ 
10:    end if
11:   when a new ordering  $o$  is received
12:     if  $o$  is more up-to-date than  $order$  then
13:       set  $order$  to  $o$ 
14:     end if
15:   when a nogood  $n$  is received
16:     if there is  $u \in \hat{n}$  such that  $order(u) > order(self)$  then
17:       send a message  $\langle self, view(self) \rangle$  to the sender
18:       resend the message  $n$  to  $u$ 
19:     else if  $n$  is out-of-date according to  $view$  then
20:       send a message  $\langle self, view(self) \rangle$  to the sender
21:       discard  $n$ 
22:     else
23:       set  $nogoods$  to  $nogoods \cup \{n\}$ 
24:       make every agent in  $\hat{n}$  a neighbour
25:     end if
26:   end when
27:   select-value ()
28: end while
```

Procedure 6.1 select-value ()

```
1: let tmp be view  $\downarrow \{u:u \text{ is a neighbour and } \text{ordering}(u) \geq \text{ordering}(\text{self})\}$ 
2: if tmp is not consistent then
3:   unset tmp(self)
4:   while no value for self is consistent with tmp do
5:     compute-nogood (tmp)
6:   end while
7:   set view(self) to some value consistent with tmp
8:   set order to some ordering as appropriate
9:   send a message  $\langle \text{self}, \text{view}(\text{self}) \rangle$  to all neighbours
10:  send a message order to all lower agents
11: end if
```

Procedure 6.2 compute-nogood (*tmp*)

```
1: let n be a minimal subset of tmp causing the domain wipeout
2: if n is empty then
3:   terminate
4: end if
5: let u be the variable in  $\hat{n}$  with the smallest ordering(u)
6: send n to u
7: unset view(u)
8: unset tmp(u)
```

Note that ABT-DO uses an explicit and mutable definition of ‘order’ to influence agent behaviour. In the above pseudocode, we have assumed that the *order* is a simple function from variables to numbers. An agent *v* is said to be higher-ranked than *u* if $\text{order}(v) > \text{order}(u)$.

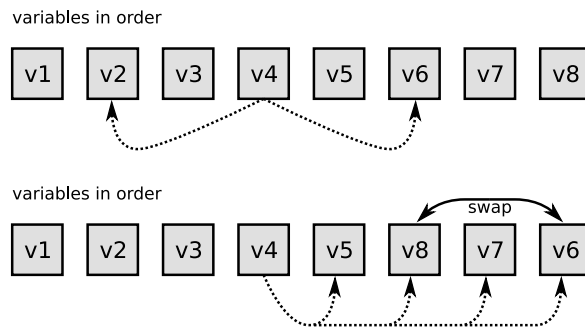


Figure 2.11: ABT-DO: Changes in assignment are broadcast to all neighbours regardless of rank.

Figure 2.12: ABT: Changes in ordering only effect lower-ranked agents, and are sent to all of them.

The key differences between ABT and ABT-DO are shown in Figures 2.11 and 2.12. The first difference is that values are sent to all neighbours rather than all lower-ranked neighbours. This change is necessary as the variable ordering may change at any time. An agent still needs only to maintain consistency with higher-ranked variables, but the definition of ‘rank’ in ABT-DO can be changed.

The second is that a higher-ranked agent can at any time alter the ordering of lower-ranked agents. An effective heuristic is to always place the sender of the most recent nogood as high as possible in the ordering [ZM05b, ZM06a]. This is inspired by the heuristic developed for use in Dynamic Backtracking [Gin93]. However, the updated ordering must be broadcast to all lower-ranked agents, including those unaffected by the change.

It is this last requirement which is most concerning for a Distributed Constraint Satisfaction Algorithm. An ‘order’ message is relative large, containing information on all variables in the DisCSP instance. It contains a total order over all variables, plus ancillary data which acts as a form of time-stamp. Our concern is that each ‘order’ message references every variable in the DisCSP, distributing ‘global’ information about the search state.

For example, it would be a fairly minimal change to make the ‘order’ message contain all current variable assignments along with the rankings. This would merely double the size (in bytes) of an ‘order’ message, which is an inconsequential amount when considering the usual exponential growth in runtimes. At that point, it becomes unclear whether ABT-DO is a ‘distributed’ algorithm in the sense of ABT or AWCS, or is merely passing a global context between agents for solving. None of the reviewed literature appears to have discussed the impact on scalability of using such large messages. This concern will be form part of our empirical analysis and comparison.

Finally, we note that ABT-DO progresses through the solution space in much the same way as ABT. However, a ‘nogood-triggered’ reordering of agents has been shown to significantly improve performance in some measures. To demonstrate this point, both ABT and ABT-DO will be included in our empirical comparisons of DisCSP algorithms.

2.8 Asynchronous Weak Commitment Search

Asynchronous Weak Commitment Search [Yok95, YH00b] is another effective distributed local search algorithm, and comes closest to meeting the requirements we outlined in our motivation. It is considered a distributed instance of Weak Commitment Search and implements the same base principles, though the translation to a distributed setting has resulted in a quite different algorithm.

The Asynchronous Weak Commitment Search algorithm provides each agent with its own view of the committed set of assignments by use of dynamic ‘priority’ values. Each agent has a current assignment and will treat a neighbour as ‘committed’ iff the neighbour has a higher priority. If an agent discovers that the set of committed assignments (from its perspective) is inconsistent it will generate a new nogood. An agent simulates discarding the set of committed assignments by raising its own priority above that of all neighbours. In this way, each agent asynchronously practises Weak Commitment Search.

As with our distributed ABT, we will use *self* to denote the variable handled by this agent, *view* to denote the current agent view, *priority* to denote the priorities of neighbours, and *nogoods* to denote a set of nogoods.

Asynchronous Weak Commitment Search has much the same structure as Asynchronous Backtracking. Initially, each agent assigns a value to its variable and sets its ‘priority’ to a default value (lines 1-3). This information is then communicated to all neighbours (line 4). When an updated assignment and priority is received from a neighbour, it is incorporated into the current view (lines 6-8). A new value is selected for this agent’s variable (line 9). When a nogood is received from a neighbour, it is incorporated into the set of nogoods and a new value is selected for this agent’s variable (lines 11-13).

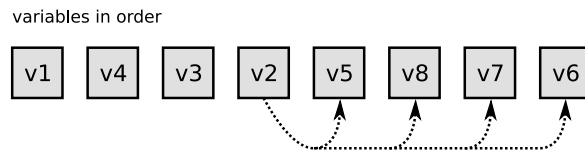


Figure 2.13: AWCS: Changes in assignment are broadcast from each agent to each neighbour, regardless of priorities.

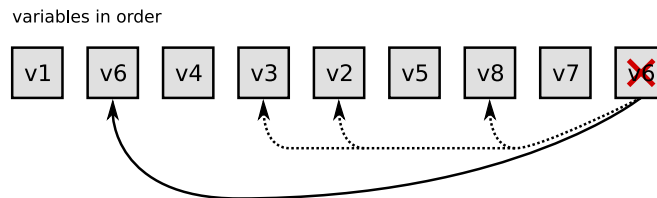


Figure 2.14: AWCS: Nogoods are sent to all involved agents, and the priority of the sender is raised above all its neighbours.

Algorithm 7 asynchronous-weak-commitment-search ()

```
1: set  $view(self)$  to some value
2: set  $priority(self)$  to 1
3: set  $nogoods$  to  $\{\}$ 
4: send a message  $\langle self, view(self), priority(self) \rangle$  to all neighbours
5: while true do
6:   when a message  $\langle w, e, q \rangle$  is received
7:     set  $view(w)$  to  $e$ 
8:     set  $priority(w)$  to  $q$ 
9:      $select-value()$ 
10:  when a nogood  $n$  is received
11:    set  $nogoods$  to  $nogoods \cup \{n\}$ 
12:    make every agent in  $\hat{n}$  a neighbour
13:     $select-value()$ 
14:  end when
15: end while
```

Procedure 7.1 select-value ()

```
1: let  $tmp$  be  $view \downarrow \{u:u \text{ is a neighbour and } priority(u) \geq priority(self)\}$ 
2: if  $tmp$  is not consistent then
3:   unset  $tmp(self)$ 
4:   if no value for  $self$  is consistent with  $tmp$  then
5:      $compute-nogood()$ 
6:     set  $tmp$  to  $\{\}$ 
7:   end if
8:   set  $view(self)$  to some value consistent with  $tmp$  and maximising consistency of  $view$ 
9:   send a message  $\langle self, view(self), priority(self) \rangle$  to all neighbours
10: end if
```

Procedure 7.2 compute-nogood (tmp)

```
1: let  $n$  be a minimal subset of  $tmp$  causing the domain wipeout
2: if  $n$  is empty then
3:   terminate
4: end if
5: set  $priority(self)$  to  $1 + \max\{priority(u) : u \text{ is a neighbour}\}$ 
6: send  $n$  to every agent in  $\hat{n}$ 
```

Selection of a value for the agent's variable is straightforward. First, the neighbours with equal or higher priority are identified and their assignments recorded (line 1). If this subset of *view* is consistent, then the current value needn't be changed (line 2). Otherwise, the agent's variable is unassigned and a search is made for a consistent value (lines 3-4). If no consistent value can be found then one of current assignments of higher-priority variables must change, and so a nogood is generated (line 5). While constructing a nogood the priority of this agent is raised, and so the subset of *view* is updated accordingly (line 6). A consistent value is then assigned, and all neighbours are notified (lines 8-9).

Construction of a nogood is also straightforward. First, the algorithm identifies a minimal set of assignments which are causing the inconsistencies discovered previously (line 1). If there are no assignments causing the inconsistencies, then the problem is unsatisfiable and the algorithm terminates (lines 2-3). The current priority for this agent's variable is raised above that of all neighbours, ensuring that a consistent value can be found (line 5). The constructed nogood is then transmitted to all neighbours, forcing at least one to change value (line 6).

Theorem 6 *Asynchronous Weak Commitment Search is complete.*

Proof. Let s denote the current complete assignment in an execution of Asynchronous Weak Commitment Search, where $s(v)$ is that value observed by v itself. Let N be the union of all nogoods held by each agent. Let $I = \{t : \forall n \in N, n \not\subseteq t\}$ be a set of assignments, defined by the current set of nogoods. We will prove that $N' \supset N$ following a finite number of iterations.

Note that changes in rank are communicated simultaneously with changes in value, and at any timepoint the rankings observed by a single agent form a total order. If an assignment is modified in s' (without the generation of a nogood), the new assignment must be taken into account only by lower-ranked neighbouring agents. Assuming that Asynchronous Weak Commitment Search is not complete for the current problem instance, we know that after at most $|\mathcal{V}|$ iterations an agent will discover that no assignment is possible for its variable. When no assignment is possible, a subset of s is added to N' , and so $N' \supset N$. By the definition of I we can see that $|I'| < |I|$ and so I is convergent to some minimal set.

As I is determined by the nogoods N , and each nogood is only generated by an inconsistency, we can be sure that I and I' contain all solutions. Further, I clearly contains all partial solutions, and membership is testable in time linear in $|N|$. Therefore I satisfies the conditions of Theorem 1, and we have proven that it will converge. From these results and Theorem 1 we can conclude that AWCS is complete. \square

2.9 Breakout

Breakout (BO) was introduced in [Mor93] as a method for escaping local-minima in iterative search. Breakout is normally presented as a hill-climbing algorithm, where the objective is to minimise the weighted sum of constraint violations. If a local minima is encountered, the Breakout algorithm increments the weights of violated constraints in an attempt to ‘break out’. This simple scheme allows Breakout to solve a surprisingly large number of problems very quickly. However, it is unable to detect infeasible problems and may not terminate on feasible problems. We will therefore present the algorithm, but not a proof of completeness.

Algorithm 8 breakout-search()

```

1: let  $s$  be the current assignment, initially complete
2: let  $p(c)$  be the weight of each constraint  $c$ , initially 1
3: while  $s$  is not a consistent assignment do
4:   let  $o$  be the sum of  $p(c)$  for all unsatisfied constraints  $c \in \mathcal{C}$ 
5:   set  $s$  to reduce  $o$  (usually a single value change)
6:   if  $o$  could not be reduced then
7:     for all unsatisfied constraints  $c$  do
8:       set  $p(c)$  to  $p(c) + 1$ 
9:     end for
10:  end if
11: end while

```

While Breakout does lack completeness, and combination with systematic techniques [EF03] could perhaps make it complete, Breakout still has many properties that are desirable. For example, there is no need for a total order amongst variables, ensuring each variable has an approximately equal obligation to change value. Further, it does not create additional connections between variables while executing. These properties will make it an interesting basis for comparison later in the thesis.

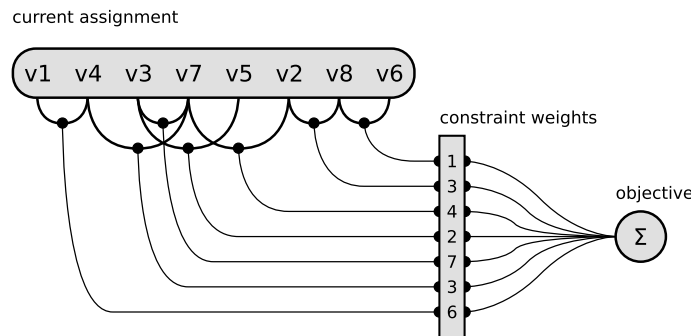


Figure 2.15: BO: Constraint violations are weighted and summed. Dynamic adjustments of weights allow a simple hill-climbing algorithm to escape local minima.

2.10 Distributed Breakout

Distributed Breakout (DBO) was introduced in [YH95] with various extensions provided in [Pet04, ZWXW05, HY05]. It is a straight-forward translation of Breakout to a distributed setting; each agent first announces its value, and then participates in a “bidding” process with its neighbours, trying to win the chance to change its value. A bid is simply the maximal change in constraint satisfaction that the variable can achieve if it were permitted to change value.

If an agent and its neighbours are unable to present bids that improve the weighted sum of satisfied constraints, then the weights of violated constraints are increased. Otherwise, if an agent has the winning bid, then it changes value. In the event of a tie for winning big, any simple tie-breaking mechanism can be used.

Algorithm 9 distributed-breakout-search ()

```
1: let  $priority(c)$  be the weight of each constraint  $c$  involving  $self$ , initially 1
2: set  $view(self)$  to some value
3: while true do
4:   send the assignment  $\langle self, view(self) \rangle$  to all neighbours
5:   receive all assignments  $\langle w, e \rangle$  from neighbours, updating  $view(w)$  as each is received
6:   let  $objective$  be the sum of  $priority(c)$  for all unsatisfied constraints  $c$  involving  $self$ 
7:   let  $movement$  be the best possible change in  $objective$  if  $view(self)$  was changed
8:   send the bid  $\langle self, movement \rangle$  to all neighbours
9:   receive all bids  $\langle w, m \rangle$  from all neighbours
10:  if no bid was negative then
11:    for all unsatisfied constraints  $c$  involving  $self$  do
12:      set  $priority(c)$  to  $priority(c) + 1$ 
13:    end for
14:  else if  $self$  sent the lowest bid (breaking ties using any suitable mechanism) then
15:    set  $self$  to a value that minimises  $objective$ 
16:  end if
17: end while
```

It is clear that this distributed variant of Breakout has maintained the desirable properties outlined earlier. It has no need for a total order amongst variables, distributing work evenly, and does not create additional connections between variables while executing. While these properties ultimately cause Breakout and Distributed Breakout to be incomplete [Mor93, ZW02], it is worthwhile taking the underlying principles as inspiration for our own algorithm.

2.11 Total, Partial, Dynamic and Static Orders

Up until this point we have informally used the terms ‘total’, ‘dynamic’, and ‘static’ to define the variable orderings used by different algorithms. Before continuing, it is essential that the reader understand the distinction between these terms. If not, some key aspects of the thesis may be misunderstood.

A ‘total order’ is a mathematical relation on a set of objects, normally written as \leq . For any three objects a, b, c in the set:

- either $a \leq b$ or $b \leq a$ (totality)
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)
- if $a \leq b$ and $b \leq a$ then $a = b$ (anti-symmetry)

From the above, we can clearly see that ABT, ABT-DO³, and AWCS⁴. all use a total order on variables. At all times, it is possible to compare two variables and determine which is ‘greater’ or of ‘higher rank’. Some form of total order is maintained throughout algorithm execution.

The terms ‘static order’ and ‘dynamic order’ can be understood by contrasting ABT and ABT-DO. In ABT, there is a single, fixed, unchanging, ‘static’ ordering between variables. In ABT-DO, the variable ordering is subject to change or ‘dynamic’. The same is true of AWCS, where the variable ordering can be altered by raising priorities. Note that this does not alter the fact that all three algorithms have and maintain a total order between variables. For an ordering to be ‘dynamic’ merely states that the ordering is malleable.

A ‘partial order’ is also a mathematical relation on a set of objects, similar to a total order. However, it lacks the key requirement that either $a \leq b$ or $b \leq a$ (totality). It may be possible to find two distinct objects a, b in a partially ordered set such that $a \not\leq b$ and $b \not\leq a$. To put it plainly, it may be possible for two objects to be *incomparable*.

All of the reviewed DisCSP algorithms use a ‘total order’ between variables. Part of the claims of this thesis is the development of an algorithm that does not use a ‘total order’ between variables. Indeed, we will argue that the algorithm we develop does not even use a ‘partial order’ as there are no guarantees of anti-symmetry.

This should not be misconstrued as a claim on ‘dynamic ordering’. While the algorithm we develop does have a dynamic concept of ordering, this is obviously not a unique feature.

³In ABT-DO, there may be two orderings being sent to agents simultaneously. However, it is guaranteed that one of those orderings is more authoritative than the other, and so that effectively defines the total order.

⁴In AWCS, it may appear that two variables are of ‘equal rank’. However, AWCS explicitly requires tie-breaking in such cases, and so it does effectively use a total order

2.12 Categorisation

We are able to classify the algorithms surveyed in this chapter into 4 distinct classes based on how they achieve completeness. Curiously, we are able to do this based on two factors: whether $I \subset I'$, and whether an unbounded memory was used. The four classes are as follows:

Class 1 - those algorithms which are not complete. Most local search algorithms belong to this class, and are quite successful in selected domains. Although they provide no completeness guarantees they are, by careful selection of heuristics, able to solve many real-world problems. We have presented Breakout Search and Distributed Breakout Search as representatives of this class.

Class 2 - those algorithms which achieve completeness by a very structured and limiting approach to search. Such methods use bounded memory, yet are highly limited in their flexibility to move within a search space. We have presented Chronological Backtracking, with its Reordering variant, as representatives of this class. Such methods may be very effective if certain choices, such as which variable to assign next, can be determined optimally.

Class 3 - those algorithms which achieve completeness by using unbounded or exponential memory. Most local search algorithms can be made complete by combining some form of flexible backtracking with an unbounded memory. We have presented Weak Commitment Search and Asynchronous Weak Commitment Search as representatives of this class. The unstructured manner in which the search space is explored by these algorithms allows for significant use of heuristics, similar to that usually observed in local search methods.

Class 4 - those algorithms which achieve completeness by careful manipulation of a bounded memory. These algorithms provide more flexibility than simple progression algorithms, without the costs associated with unbounded memories. We have presented Dynamic Backtracking and Asynchronous Backtracking as a representative of this class. Note that the defining attribute of this class of algorithms is the use of memory to manage what is, ultimately, a constructive search.

	CBT	DBT	ABT-DO	WCS	AWCS	BOS	DBOS
Class	2	4	4	3	3	1	1
Distributed Algorithm			•		•		•
Complete Algorithm	•	•	•	•	•		
Monotonic Proof	•			•	•	NA	NA
Global Ordering	•	•	•			NA	NA
Bounded Memory	•	•	•			•	•

Table 2.1: Classification of algorithms according to their completeness mechanism

The above table provides a summary of our algorithm classifications, and highlights particular properties of the surveyed algorithms. The properties of each algorithm are:

- Class - the class of the algorithm as described above
- Distributed Algorithm - whether the algorithm is designed for solving DisCSP
- Complete Algorithm - whether the algorithm is complete
- Monotonic Proof - whether the completeness proof showed that $I \subset I'$
- Global Ordering - whether all agents are aware of the total variable ordering
- Bounded Memory - whether the algorithm uses a non-exponential amount of memory

First, it should be noted that the classifications we have used on the previous page actually correspond to the type of completeness proof and the ‘boundedness’ of memory. This is partly by design, but is an interesting result - by using Theorem 1 in our completeness proofs, and observing the amount of memory required, we can formally classify each algorithm using relatively intuitive categories.

We can also see that, for all complete algorithms we have surveyed, we require either an ‘unbounded’ memory store or a total order on variables. It seems highly unlikely that this requirement will change, though it would be interesting to consider whether such an algorithm would ever be possible. Such a question is outside the scope of this thesis however, and we will simply assume that it is necessary to use either a total order on variables, or an unbounded memory store.

Finally, we will note which of the reviewed algorithms are suitable to use as ‘benchmarks’ for comparing our own algorithm. Note that we have limited our thesis to considering situations which require a complete distributed constraint satisfaction algorithm. Therefore we will omit non-complete and non-distributed algorithms such as Distributed Breakout or Weak Commitment Search from our comparisons. This limits any empirical or analytical comparisons to AWCS, ABT and ABT-DO.

Technically, none of these three algorithms satisfy the criteria outlined in our motivation. For example, all three algorithms require that messages be sent to variables which do not have a pre-existing constraint. Similarly, all three use a total order of some form to determine search behaviour, requiring a notion of ‘authority’ between variables.

The issue of ‘total ordering’ is mitigated in part by the introduction of ‘dynamic’ ordering. AWCS appears to be the most dynamic in this respect, allowing variables to be elevated to the ‘highest-rank’ relatively easily. By changing the variable ordering on a frequent basis, AWCS is able to operate as if there were almost no ordering at all. Unfortunately, it is still the case that lower-ranked agents are unable to raise their rank unless they can produce a nogood.

ABT-DO is less dynamic than AWCS, placing significant restrictions on variable ordering behaviour. For example, ABT-DO guarantees that the ‘highest-ranked’ variable will always remain highest-ranked. It is similarly unclear whether a ‘high-ranked’ variable will ever drop significantly in priority. However, this is already a significant improvement over ABT, where the ‘authority’ between variables is determined before solving begins.

As stated, we also hope to avoid the need for ‘broadcasting’ variable assignments within our new algorithm. AWCS will effectively start ‘broadcasting’ value assignment if the use of nogoods calls for many additional links between variables. ABT-DO explicitly requires ‘broadcasting’ of ordering messages, and may also need additional links between variables to support nogoods. ABT requires ‘broadcasting’ of value assignments from higher-ranked agents to lower-ranked agents.

While none of these algorithms fits our motivating requirements, they are all well-established algorithms for solving DisCSP problems. As such, they will serve as ‘benchmarks’ for comparing the performance of our algorithm. However, it is critical to note that the primary point of our algorithm is to match the motivating criteria. Avoiding a ‘total order’ and ‘broadcasting’ are considered the primary contribution, and not necessarily an improvement in solving performance.

This completes our survey of existing distributed constraint satisfaction algorithms. We have reviewed the most commonly cited distributed algorithms, and related each to their centralised counterparts. This thesis will focus on developing an alternative distributed constraint satisfaction algorithm, designed solely for distributed constraint satisfaction.

Chapter 3

Support-Based Distributed Search

3.1 Introduction

In this chapter we will describe Support-based Distributed Search; an algorithm based on the notion of ‘arguments’. We will first provide a simple example that demonstrates how humans are able to solve a meeting scheduling problem through arguments. We use meeting scheduling as it is a natural class of problems which can be seen as unbounded distributed constraint satisfaction problems.

Example. Alice, Bob, Carla and Dennis are attending a conference and must organise meetings amongst themselves:

- Bob must meet with Carla.
- Bob must meet with Alice before meeting with Carla.
- Dennis must meet with Alice.
- Bob, Carla and Dennis must have a separate group meeting.
- Available times are 1pm, 2pm and 3pm, and double-booking is not allowed.

Communication should only occur between pairs of people who need to meet, and no-one should directly exercise any ‘authority’ over any other. Each person is initially only aware of those meetings that they need to attend, but may become aware of the existence of other meetings. To find the solution they state arguments (proposals and rejections) in turn, providing further detail if two arguments are contradictory or if they find they need to alter a previous argument:

Alice to Dennis \Rightarrow I propose a 1pm meeting
 Dennis to Carla \Rightarrow I propose a 2pm group meeting
 Dennis to Bob \Rightarrow I propose a 2pm group meeting
 Carla to Bob \Rightarrow I propose a 1pm meeting
 Alice to Bob \Rightarrow I propose a 2pm meeting
 Bob to Alice \Rightarrow I have a group meeting at 2pm,
 so I propose a 1pm meeting instead
 Bob to Carla \Rightarrow I completely reject your proposal,
 so I propose a 3pm meeting instead
 Alice to Dennis \Rightarrow I now have another meeting at 1pm,
 so I propose a 3pm meeting instead

The participants can successfully establish a meeting schedule by using such a dialogue. Bob will meet with Alice at 1pm and then Carla at 3pm. Dennis will meet with Alice at 3pm. Finally, Bob, Carla and Dennis will have a group meeting at 2pm. This was achieved while no-one spoke to a person they didn't already know, and no-one exercised explicit authority over anyone else. ■

To be able to use this small meeting scheduling example in a formal context, we will construct a matching distributed constraint satisfaction problem. We first translate the time of each meeting into a separate variable for each participant. Equality constraints are used to ensure meeting times are agreed to by all users. For example, a pair of variables a and b may represent the scheduled time of the meeting between Alice and Dennis. The constraint $a = b$ is interpreted as 'the time Alice decides to meet with Dennis must be the same as the time that Dennis decides to meet with Alice'. Inequality constraints, such as $b \neq c$, ensure that a participant is not involved in two meetings simultaneously.

$$\mathcal{V} = \{a, b, c, d, e, f, g, h, i\}$$

$$\mathcal{D} = \{1\text{pm}, 2\text{pm}, 3\text{pm}\}$$

$$C = \left\{ \begin{array}{ll} a = b & a \neq i \\ b \neq c & c = d \\ c = g & d \neq e \\ d = g & e = f \\ f \neq g & f > h \\ g \neq h & h = i \end{array} \right\}$$

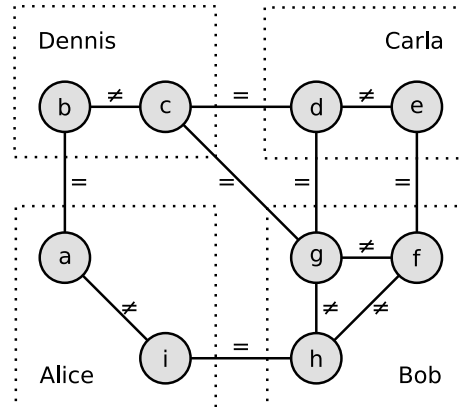


Figure 3.1: Example constraint model and graph

Note that there is significant redundancy in the constraints and variables as presented in Figure 3.1. This is necessary as the constraints upon one person are not automatically known to others. Relaxing this requirement would generate a simpler constraint graph, but would conflict with our aim to solve the problem in a distributed manner. Using this constraint model as an example, we will now define suitable notation for representing the dialogue. The remainder of this chapter will use this notation to develop and demonstrate how *arguments* can form the basis of a distributed constraint satisfaction algorithm.

3.2 Representation

We consider arguments as belonging to two classes: proposals and rejections. Formally we will translate these to the terms ‘isgoods’ and ‘nogoods’.

Definition 8 An *isgood* is an ordered partial assignment for a sequence of connected variables, and so represents a ‘proposal’. For a given constraint satisfaction problem $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$, an isgood is written as a sequence of assignments:

$$I = \langle \langle v_{i_1}, d_{j_1} \rangle, \dots, \langle v_{i_n}, d_{j_n} \rangle \rangle$$

Consider the argument in our example where Bob says to Alice: “I already have a group meeting at 2pm, so I propose a 1pm meeting for us instead”. This is a proposal, and so can be written as an ‘isgood’:

$$\langle \langle g, 2\text{pm} \rangle, \langle h, 1\text{pm} \rangle \rangle$$

This isgood is read as “variable g took on value 2pm, and so h took on value 1pm”. We say that a variable h is **supported** by the variable g in the above isgood as h is the immediate predecessor to g .

Note that variables in an isgood must be connected to their immediate predecessor, and so $\langle \langle d, 2\text{pm} \rangle, \langle h, 1\text{pm} \rangle \rangle$ is not an isgood. Also, for notational convenience, we will use the operator $+$ to represent the appending of a variable assignment to an isgood. For example, $\langle \langle g, 2\text{pm} \rangle, \langle h, 1\text{pm} \rangle \rangle + \langle i, 1\text{pm} \rangle = \langle \langle g, 2\text{pm} \rangle, \langle h, 1\text{pm} \rangle, \langle i, 1\text{pm} \rangle \rangle$.

Definition 9 A *nogood* is an unordered partial assignment which is provably not part of a solution, and so represents a ‘rejection’. For a given constraint satisfaction problem $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$, a nogood is written as a set of assignments:

$$N = \{ \langle v_{i_1}, d_{j_1} \rangle, \dots, \langle v_{i_n}, d_{j_n} \rangle \}$$

Consider the argument in our example where Bob says to Carla: “I reject your proposal, and I propose a 3pm meeting for us instead”. This is a rejection (he must meet Carla before Alice, so 1pm is not a possible meeting time) followed by a proposal. Written in sequence these would be a nogood followed by an isgood:

$$\{\langle e, 1\text{pm} \rangle\} \text{ and } \langle \langle f, 3\text{pm} \rangle \rangle$$

They are read respectively as “variable e cannot take value 1pm” and “variable f took on value 3pm”. In general, a nogood will be accompanied by an isgood as it was in the above example.

Difference between Isgoods and Partial Assignments

It may appear that isgoods are equivalent to the well-known concept of ‘partial assignment’. However, there are two notable differences which are worth highlighting:

- An isgood provides information about the order in which assignments are made. The isgood $\langle \langle g, 2\text{pm} \rangle, \langle h, 1\text{pm} \rangle \rangle$ is not equivalent to $\langle \langle h, 1\text{pm} \rangle, \langle g, 2\text{pm} \rangle \rangle$.
- An isgood must be formed from a chain of variables which are connected by a constraint. It is not always possible to produce an isgood which contains all variables, as such a chain may not exist in the constraint graph.

Consistency of Arguments

We say that a constraint is **satisfied** by an isgood I if the constraint is not explicitly violated by the assignments in I . Testing whether a constraint is satisfied is therefore only possible if all variables appearing in the constraint also appear in I . Similarly, a nogood is satisfied if it is not a subset of the assignments in I . For example, given an isgood $I = \langle \langle g, 2\text{pm} \rangle, \langle h, 1\text{pm} \rangle \rangle$ we know:

- $I + \langle i, 2\text{pm} \rangle$ does not satisfy the constraint $h = i$
- $I + \langle i, 1\text{pm} \rangle$ does satisfy the constraint $h = i$
- I does not satisfy the nogood $\{\langle h, 1\text{pm} \rangle\}$
- I does satisfy the nogood $\{\langle h, 1\text{pm} \rangle, \langle i, 1\text{pm} \rangle\}$

Thus, given a set of constraints and a set of nogoods, we say that an assignment $\langle v, d \rangle$ is **consistent** with respect to an isgood I iff each constraint on v and each nogood is satisfied by $I + \langle v, d \rangle$.

3.2.1 Assignments as Arguments

It is obvious that every isgood is also a partial assignment, yet not all partial assignments are isgoods. We must consider the limitations and ramifications from using isgoods.

An isgood is an ordered sequence of variable-value pairs, and must be formed from a chain of variables. This definition is substantially more restrictive than the notion of assignments, in which no relation between variables is required. Most concerning is that a complete assignment is not representable as an isgood if a complete tour of the constraint graph does not exist. The usefulness of isgoods would be quite limited if we were unable to represent a complete assignment and test for consistency.

To construct a sound algorithm using isgoods, it is useful (and, depending on the algorithm itself, necessary) that every *solution* is able to be represented as a *set of isgoods*. Such a *set* of isgoods must be able to be tested for consistency with all constraints simultaneously.

Definition 10 *Given a complete assignment s for a constraint satisfaction problem $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ we say that a set of isgoods $\{I_1, \dots, I_n\}$ is **representing** s if the assignments in each I_i is a subset of s . We say the set of isgoods is **variable-covering** if every variable appears in at least one isgood. We say the set of isgoods is **constraint-covering** if the scope of every constraint is included in at least one isgood.*

Theorem 7 *Given a complete assignment s for a constraint satisfaction problem $\langle \mathcal{V}, \mathcal{C}, \mathcal{D} \rangle$ we can trivially construct a representing, constraint-covering set of isgoods.*

Proof. We are able to prove this by simply constructing a set of isgoods where each is a sequence representing $s \downarrow \hat{c}_i$. As the scope of each isgood is contained by the scope of a constraint, we are sure that each isgood forms a chain. \square

Example. Consider the constraint satisfaction problem described in Figure 3.1. A possible solution is where $s(a) = s(b) = s(e) = s(f) = 3\text{pm}$, $s(c) = s(d) = s(g) = 2\text{pm}$, and $s(h) = s(i) = 1\text{pm}$. We can construct a representing, variable-covering, and constraint-covering set of isgoods by projecting s on to each constraint in \mathcal{C} .

$$C = \left\{ \begin{array}{ll} a = b & a \neq i \\ b \neq c & c = d \\ c = g & d \neq e \\ d = g & e = f \\ f \neq g & f > h \\ g \neq h & h = i \end{array} \right\} \quad I = \left\{ \begin{array}{ll} \langle (a, 3\text{pm}), (b, 3\text{pm}) \rangle & \langle (a, 3\text{pm}), (i, 1\text{pm}) \rangle \\ \langle (b, 3\text{pm}), (c, 2\text{pm}) \rangle & \langle (c, 2\text{pm}), (d, 2\text{pm}) \rangle \\ \langle (c, 2\text{pm}), (g, 2\text{pm}) \rangle & \langle (d, 2\text{pm}), (e, 3\text{pm}) \rangle \\ \langle (d, 2\text{pm}), (g, 2\text{pm}) \rangle & \langle (e, 3\text{pm}), (f, 3\text{pm}) \rangle \\ \langle (f, 3\text{pm}), (g, 2\text{pm}) \rangle & \langle (f, 3\text{pm}), (h, 1\text{pm}) \rangle \\ \langle (g, 2\text{pm}), (h, 1\text{pm}) \rangle & \langle (h, 1\text{pm}), (i, 1\text{pm}) \rangle \end{array} \right\}$$

Note that this theorem only ensures that each solution can be represented as a set of isgoods that can be tested for consistency with all constraints in \mathcal{C} simultaneously. However, it does not guarantee that the set can be constructed to be tested against all nogoods. If we ever construct a nogood that cannot be covered by a constraint graph tour, then that nogood can *never* be tested against any isgood. We will show later that testing of arbitrary nogoods is not required for SBDS, and that testing for constraint violations is sufficient for soundness.

Also note that many of the isgoods in the above example are redundant. Many isgoods are also supporting the same variable ($\langle(d, 2\text{pm}), (g, 2\text{pm})\rangle$ and $\langle(f, 3\text{pm}), (g, 2\text{pm})\rangle$ for example), which is unnecessary. We can construct a smaller representing and variable-covering set of isgoods as follows:

Theorem 8 *Given a complete assignment s for a constraint satisfaction problem $\langle\mathcal{V}, \mathcal{C}, \mathcal{D}\rangle$, we can construct a representing, variable-covering set of isgoods such that each variable is supported by at most one other.*

Proof. Again, we are able to prove this by simply constructing a directed spanning tree of the constraint graph and constructing isgoods along each edge. Each variable is thus covered, and each variable has just one (or no) support corresponding to its parent in the spanning tree. \square

Example. Consider again the constraint satisfaction problem described in Figure 3.1, with solution $s(a) = s(b) = s(e) = s(f) = 3\text{pm}$, $s(c) = s(d) = s(g) = 2\text{pm}$, and $s(h) = s(i) = 1\text{pm}$. A representing, variable-covering set of isgoods can be constructed by following a spanning tree:

$$\mathcal{C} = \left\{ \begin{array}{ll} a = b & a \neq i \\ b \neq c & c = d \\ c = g & d \neq e \\ d = g & e = f \\ f \neq g & f > h \\ g \neq h & h = i \end{array} \right\} \quad I = \left\{ \begin{array}{ll} \langle(a, 3\text{pm}), (b, 3\text{pm})\rangle & \\ \langle(b, 3\text{pm}), (c, 2\text{pm})\rangle & \langle(c, 2\text{pm}), (d, 2\text{pm})\rangle \\ \langle(c, 2\text{pm}), (g, 2\text{pm})\rangle & \langle(d, 2\text{pm}), (e, 3\text{pm})\rangle \\ & \langle(e, 3\text{pm}), (f, 3\text{pm})\rangle \\ & \langle(f, 3\text{pm}), (h, 1\text{pm})\rangle \\ & \langle(h, 1\text{pm}), (i, 1\text{pm})\rangle \end{array} \right\}$$

Note that it is not necessary for a set of isgoods to be arranged according to a single spanning tree to have unique supports for each variable. Other common possibilities are multiple spanning trees over connected components, or even the introduction of cycles of support.

Finally, we note that it is trivial to extend any variable-covering set of isgoods to a constraint-covering set of isgoods when the CSP is binary. We simply need to check the consistency of each variable-covering isgood, and check consistency with each neighbouring variable. This ensures that it is both possible and efficient to test the consistency of a complete assignment s , even when represented as a variable-covering set of isgoods. We emphasise this point as, in most circumstances, SBDS will work with variable-covering sets of isgoods and not constraint-covering.

3.2.2 Interpretation of Arguments

In our simple meeting example, the participants were cooperative; they attempted to satisfy everyone. Similarly, we will assume that, within a distributed constraint satisfaction algorithm, the satisfaction of all constraints is the goal of all agents. We make this assumption even though each may only be explicitly aware of a limited subset of constraints.

However, the exact means by which agents cooperate depends upon the interpretation of the messages between them. Are they being asked to do something, or asked to test something? We now consider two possible interpretations of isgoods. Our choice of interpretation will guide our algorithm development. As each interpretation refers to the isgood as a ‘proposal’ we will informally use the term ‘accepting’ to refer to those agents which have no objection to the proposal.

1. *An isgood is a proposal to test a subspace.* By this interpretation an isgood describes a portion of the search space (we term this a subspace) which an agent currently believes may contain a solution. To more fully test this subspace an agent sends the isgood to others, requesting their assistance in testing the consistency of that subspace. They in turn refine the subspace with their own assignment and ask other agents to test it further.
2. *An isgood is a proposal to take an assignment.* By this interpretation an isgood describes a partial assignment which an agent believes is a partial solution. By communicating an isgood to others it is attempting to convince them to take on and extend that partial solution.

The differences between these ‘interpretations’ is very subtle, but will be shown as critical in the development of a DisCSP algorithm. We will refer to these interpretations as ‘subspace’ and ‘assignment’ respectively.

The ‘assignment’ interpretation views an isgood as a set of assignments that are being suggested as part of a global search. This, in some sense, distributes control of variable

values to all agents, rather than using a single agent per variable. Under this interpretation an agent may suggest a specific value for variables which have not yet accepted the proposal.

Alice to Dennis \Rightarrow I propose that I meet Bob at 1pm, that I meet you at 3pm, and that you have your group meeting with Bob and Carla at 2pm. Does that sound good?

The ‘subspace’ interpretation views an isgood as a set of assignments limiting the search space, and requesting the help of neighbours in exploring that space. Therefore an agent may only provide assignments for variables which have already accepted the proposal in its current form; all others are asked to help explore instead.

Dennis to Alice \Rightarrow Bob and I agreed to a meeting at 1pm, so I’m proposing you and I meet at 3pm. Can you help explore this possibility?

The difference between these interpretations is important to discuss before development of an algorithm: the interpretation will determine how each agent should cooperate with its neighbours. Either interpretation can be seen as cooperative as each is trying to establish mutual agreement without specific preferences on values. The question must then be asked: which is the more difficult to enact in a distributed system?

The subspace interpretation emphasises the autonomy of neighbouring agents, always permitting them to determine their own value. The subspace interpretation also encourages an ad-hoc form of distributed backtracking as agents make only incremental extensions to a proposal. However, the ‘subspace’ interpretation suffers problems as agents would be prevented from expressing the entire description of their current subspace if doing so would prescribe a value for a variable. This can lead to cyclic behaviour, as no agent would ever be able to present a global view:

Alice to Bob \Rightarrow Dennis and I agreed to a meeting at 1pm,
 so I'm proposing you and I meet at 3pm.
 Can you help explore this possibility?
 Bob to Dennis \Rightarrow Alice and I agreed to a meeting at 3pm,
 so I'm proposing you and I meet at 1pm.
 Can you help explore this possibility?
 Dennis to Alice \Rightarrow Bob and I agreed to a meeting at 1pm,
 so I'm proposing you and I meet at 3pm.
 Can you help explore this possibility?
 Alice to Bob \Rightarrow Dennis and I agreed to a meeting at 3pm,
 so I'm proposing you and I meet at 1pm.
 Can you help explore this possibility?
 ad infinitum ...

In contrast, the above cyclic behaviour would be easily handled by the 'assignment' interpretation as cycles of dependant meeting times can (potentially) be expressed within a single argument. However, the 'assignment' interpretation, if taken to the extreme, allows for the distribution of control of variable values to the neighbours of each agent. Presumably, this would require additional transfer of information, such as constraints, which is not desirable in general.

However, a limited form of 'assignment' interpretation is possible: each agent may only propose assignments for their own variable or previously proposed assignments for other variables. Consider the following example:

Alice to Dennis \Rightarrow I propose that I meet Bob at 1pm, that I meet you
 at 3pm, and you previously proposed to have your group meeting with
 Bob and Carla at 2pm. Does that sound good?

As we will see in the following section, this limited form of the 'assignment' interpretation seems most appropriate for a multi-agent algorithm. It preserves the autonomy of neighbouring agents, ensuring that no agent is burdened with deciding the value for others. However, the cyclic behaviour seen in the subspace interpretation has the potential to be eliminated.

3.3 Solving

Using the above notation, interpretation, and the dialogue of our example as a guide, we will now describe a distributed search algorithm called Support-Based Distributed Search (SBDS). In this algorithm agents cooperate by proposing assignments for their own variables, and possibly for their neighbours. In this design, they will:

- send and receive proposals (isgoods) and rejections (nogoods)
- attempt to convince neighbours to accept their proposals
- reject a proposal from a neighbour if it is inconsistent
- justify their variable assignment by the proposal from one neighbour
- communicate only with agents for which they share a constraint

To achieve this, each agent records the most recent proposals that are exchanged with neighbouring agents, and an unbounded store of received nogoods. Unlike other distributed algorithms, SBDS does not collapse all information from neighbours into a consistent ‘agent view’. Instead, the isgood received from just one neighbour is chosen as justification for the current assignment. Together, the ‘supporting’ isgood and current assignment will form our ‘agent view’. Formally, the information stored by each agent is:

- *self* - a reference to the agent itself
- *sent*(*v*) - last isgood sent to each neighbouring agent *v*
- *recv*(*v*) - last isgood received from each neighbouring agent *v*
- *nogoods* - set of all nogoods ever received
- *support* - the neighbour chosen for our ‘agent view’
- *view* - current agent view (*recv*(*support*) plus an assignment to our own variable)

To simplify the algorithm description, we will also use the following shorthand:

- the phrase ‘*d* is consistent with *I*’ indicates that the isgood $I + \langle self, d \rangle$ is consistent with the constraints and *nogoods* known by the current agent.
- the phrase ‘*I* is extended with *d*’ indicates that any previous value for *self* in *I* is removed, and the tuple $\langle self, d \rangle$ is appended to *I*.
- the phrase ‘strength of *I*’ describes the length of an isgood once it has been extended with some value *d*, formally defined as $|I| = |scope(I) \cup \{self\}|$

Note that this definition of ‘strength’ is used extensively within the algorithm.

Algorithm 10 *support-based-distributed-search ()*

```
1: set support to self
2: set recv(self) to  $\langle \rangle$ 
3: set nogoods to  $\{ \}$ 
4: while a choice of value is consistent with the empty isgood  $\langle \rangle$  do
5:   compute-nogood(v) for each neighbour v
6:   select-view()
7:   compute-isgood(v) for each neighbour v
8:   when a nogood N is received
9:     set nogoods to nogoods  $\cup \{N\}$ 
10:  when an isgood I is received from some neighbour v
11:    set recv(v) to I
12:  end when
13: end while
```

Procedure 10.1 *compute-nogood (v)*

```
1: if recv(v) is set, and no choice of value is consistent with recv(v) then
2:   let N be an inconsistent subset of recv(v)
3:   send N to v
4:   set recv(v) to  $\langle \rangle$ 
5:   if support = v then
6:     set support to self
7:   end if
8: end if
```

Procedure 10.2 *select-view ()*

```
1: choose any neighbour v and value d such that:
    • v is equal to support, or  $\lceil \text{recv}(v) \rceil > \lceil \text{recv}(\text{support}) \rceil$ ; and
    • d is the lowest value consistent with recv(v); and
    • for all u where  $\lceil \text{recv}(u) \rceil > \lceil \text{recv}(v) \rceil$ , d must be consistent with recv(u).
2: set support to v
3: set view to recv(v) + (self, d)
```

Procedure 10.3 *compute-isgood* (v)

```
1: let cycling be true iff self and  $v$  are the first two variables in  $recv(support)$ 
2: let conflicting be true iff  $\lceil sent(v) \rceil < \lceil recv(v) \rceil$  and  $view(this)$  is inconsistent with  $recv(v)$ 
3: let updating be true iff  $sent(v)$  is  $\langle \rangle$  or is not a tail of  $view$ 
4: if cycling then
5:   if  $view \not\leq sent(v)$ , or  $sent(v)$  is not consistent then
6:     send  $view$  to  $v$ 
7:     set  $sent(v)$  to  $view$ 
8:   end if
9: else
10:  let maximum be the longest tail of  $view$  that can be sent to  $v$ 
11:  let preferred be initially 0
12:  if updating then set preferred to  $\lceil recv(v) \rceil + 1$ 
13:  if conflicting then set preferred to  $\max(preferred, \lceil recv(v) \rceil + 1)$ 
14:  if preferred  $> 0$  then
15:    let  $I$  be a tail of  $view$  such that  $\lceil I \rceil = \min(preferred, maximum)$ 
16:    if  $I \neq sent(v)$  then send  $I$  to  $v$ 
17:    set  $sent(v)$  to  $I$ 
18:  end if
19: end if
```

The *main* loop of our algorithm alternates between computing and sending messages to neighbours, and receiving messages from neighbours. In each iteration:

- the *compute-nogood* procedure is called to test the consistency of the most recently received isgoods of each neighbour, and sends nogoods when appropriate
- the *select-view* procedure is called to select a *support* and *view*, effectively selecting the variable's value
- the *compute-isgood* procedure is called to test if the agent needs to update each neighbour, sending isgoods when appropriate
- when a nogood is received, it is added to the current store of nogoods
- when an isgood is received, it is recorded as the most recently received isgood of the respective neighbour

The *compute-nogood* procedure tests whether the isgood received by a given neighbour is consistent within itself, regardless of our current value. If the isgood is inconsistent, then a nogood is generated, sent to the neighbour, and the record of the most recently received isgood is discarded. The nogood may be any inconsistent subset of the assignments in the

isgood, but a minimal nogood is generally more effective. An interesting effect of this procedure is that nogoods are formed only from variables in a sequence, rather than ‘all neighbours’ as occurs in AWCS. Note that if a nogood is sent to our current *support* neighbour, then the *support* is changed to *self*.

The *select-view* procedure considers possible ways to use each of the isgoods received from neighbours, and produces a new *support/view* pair. This function expresses a set of criteria that must be satisfied for the *support/view* pair - precise implementation is left to the implementer. The criteria state:

- The list of candidate neighbours includes the current *support*, and all neighbours which would provide stronger isgoods.
- For each candidate, only one possible *view* is considered; that which assigns to *self* the lowest possible consistent value.
- Only those candidates which are capable of defeating or agreeing with all neighbours, based on this restricted choice of value, can be chosen.

Note that the restriction on the list of candidates ensures that an agent never chooses a ‘weaker’ neighbour as a new *support*. As will be demonstrated later, monotonic increasing strength of *view* is critical to the completeness of the algorithm. Also, the *view* is determined only by the *support*’s isgood, which is also critical to completeness.

The *compute-isgood* procedure determines the suitability and content of isgoods to be transmitted to neighbours. Three boolean variables are first determined:

- *cycling* is true if and only if the algorithm has detected cyclic behaviour. A cycle is formed when there is a mutual dependence of *support* between a sequence of connected agents. A cycle is detected when the values of *self* and the neighbour *v* are both prescribed, in that order, by the isgood of the current *support*.
- *updating* is true if and only if no isgood was previously sent to the neighbour *v*, or the last isgood contains information that is out-of-date. Intuitively, if *updating* is true, then a new isgood *must* be sent to correct the information.
- *conflicting* is true if and only if there is a conflict between *self* and the neighbour *v*, and the isgood received from *v* is stronger than what was sent. So, if *conflicting* is true, then a stronger isgood must be sent in an attempt to defeat the *recv(v)*.

If *cycling* is true, then the entire *view* (including the prescribed value for the neighbour) should be sent and the procedure terminates. Note that the transmission of the entire *view* will be postponed if a ‘greater’ isgood was sent previously. The details and necessity for postponement, and the concept of ‘greater’, are explained later.

If *cycling* is false, then the agent will need to decide what strength of isgood should be sent. The *maximum* strength isgood that can be sent is normally the entire current *view*. However, if *view* contains some reference to the neighbour v , then the *maximum* strength may need to be less. For example, if the *view* is $\langle (t, 1), (u, 1), (v, 1), (w, 1) \rangle$ then the strongest possible isgood is $\langle (v, 1), (w, 1) \rangle$. While the strength of *view* is 4, the *maximum* allowable strength is just 2.

After computing the maximum strength, the agent must determine a *preferred* strength. If *updating* is true, then the procedure will *prefer* an isgood that is stronger than the last sent. If *conflicting* is true, then the procedure will *prefer* an isgood that is stronger than the last received. Finally, if *preferred* is greater than 0, the agent will construct an appropriate strength isgood and send it.

We will discuss unique and interesting aspects of this algorithm in the following sections. Particular attention will be paid to the computation of isgoods, resolving issues with asynchronous environments, and resolving cyclic behaviour.

3.3.1 Asynchronicity with Isgoods

Support-Based Distributed Search is an asynchronous algorithm, allowing each agent to send messages at any time. Further, each agent may use different measurements of time and may communicate at different rates. This contrasts with some other distributed algorithms where agents either operate in strict synchronisation or use timestamps to identify ‘current’ messages. Guaranteeing global synchronisation of agents in large networks of may be very expensive, and so we have avoided such schemes in SBDS. It is worthwhile describing how this can be achieved.

SBDS uses self-contained messages between agents, ensuring that the actions of each agent are not dependent on the exact current state of neighbours. Each message contains all information that is necessary for correct interpretation, regardless of information from the same or other sources. This is apparent in the way that, for example, an agent’s *view* is computed.

In all previous algorithms, the *view* contains the current assignments of an agent’s neighbours and is used in determining the new assignment for the agent’s variable. Agents attempt to construct a *view* of the current global assignment by combining messages from multiple neighbours. In doing so, each agent implicitly assumes that its current *view* represents a consistent assignment of all observed variables; this may or may not be true. It is the responsibility of each agent to construct nogoods if it determines that assignment is infeasible. However, this behaviour may result in the construction of spurious nogoods for assignments that are already known to be inconsistent.

SBDS takes an alternative approach, constructing the current *view* using information from at most one neighbouring agent. By doing so, each agent will only respond to partial assignments that have been verified as consistent by relevant agents. The potential for spurious or redundant nogoods is therefore significantly reduced.

3.3.2 Computation of Isgoods

The *compute-isgood* draws heavily from human behaviours, plus inspiration from both argumentation and other DisCSP algorithms. In this section we will explain the rationale, and give examples of isgoods it would produce. The key requirements that must be satisfied when computing an isgood are as follows:

- An isgood must be crafted in response to a particular neighbour; different arguments must be presented to different neighbours. There are many ‘human’ reasons for this in the real world (privacy, lying for advantage, etc), but none are necessary features of SBDS. However, in both the real world and in SBDS, an argument which already includes a neighbour cannot be used to later convince that neighbour. While cyclic arguments are permitted, self-reinforcing arguments are not, and so arguments may be truncated to suit the recipient.
- Information transmitted with an isgood must be kept up-to-date; a change in value of any variable must be communicated to all agents which were aware of the previous value. Further, changes to values should be accompanied by a stronger isgood when possible. This is common practise in the real world, where a ‘change of mind’ should be accompanied by a stronger supporting argument. In most cases, this is merely a courtesy, but it also helps identify mutual dependencies between neighbours. The repercussions of *not* increasing the strength of isgoods is explored in detail in a later section.
- Conflicts may only be resolved by the use of a stronger isgood; whichever agent is able to present a stronger isgood wins. We differentiate between ‘conflicts’ pertaining to the value of the agent’s own variable, and ‘disagreements’ pertaining to the value of other agent’s variables. It is not required that disagreements about the value of third-party variables be resolved. It is merely required that pairs of agents resolve direct conflicts between themselves.¹

¹Note that there is a passing similarity here to the mediation mechanism of the recent Asynchronous Partial Overlay algorithm [GM07, ML06]. However, APO performs conflict resolution by first centralising the problem rather than maintaining a distributed approach. APO has not been included in our literature review for that reason, and was not a source of inspiration in the development of SBDS.

Consider the following examples of isgoods computed by g for each of its neighbours c , d , f and h . Assume that g has chosen d as *support*, with a *view* of $\langle(e, 2\text{pm}), (d, 1\text{pm}), (g, 1\text{pm})\rangle$.

	Received	Sent	Computed
c	$\langle(b, 3\text{pm}), (c, 2\text{pm})\rangle$	$\langle(g, 1\text{pm})\rangle$	$\langle(e, 2\text{pm}), (d, 1\text{pm}), (g, 1\text{pm})\rangle$
f	$\langle(f, 1\text{pm})\rangle$	$\langle(g, 1\text{pm})\rangle$	
h	$\langle(h, 1\text{pm})\rangle$	$\langle(g, 3\text{pm})\rangle$	$\langle(d, 1\text{pm}), (g, 1\text{pm})\rangle$
d	$\langle(e, 2\text{pm}), (d, 1\text{pm})\rangle$	$\langle(d, 2\text{pm}), (g, 3\text{pm})\rangle$	$\langle(d, 1\text{pm}), (g, 1\text{pm})\rangle$

We can explain the computation of each isgood in turn.

- The isgood that was received from neighbour c specified that $c = 2\text{pm}$. This conflicts directly with g 's current *view*, as there is a constraint specifying $c = g$. As it conflicts with the current *view*, and is stronger than the previously sent isgood, g must provide a stronger isgood in response. The strength of the isgood from c , as measured by g , is 3, so an isgood of at least that strength if required.
- The isgood that was received from neighbour f does not conflict with g 's current *view*. Further, the isgood that was previously sent to neighbour f does not contain any information that is not in g 's current *view*. As there is no conflict and the previously sent isgood is up-to-date, no isgood needs to be computed or sent to f .
- The isgood that was received from neighbour h does not conflict with g 's current *view*. However, the isgood that was previously sent to neighbour h states that $g = 3\text{pm}$, which is not correct according to g 's current *view*. As the previously sent isgood is not up-to-date, an isgood with strictly greater strength must be computed.
- The isgood that was received from neighbour d forms the basis for g 's current *view*. However, the isgood that was previously sent to neighbour d states that $g = 3\text{pm}$, which is not correct according to g 's current *view*. As the previously sent isgood is not up-to-date, an isgood with strictly greater strength should be computed. However, the computed isgood for d is the maximal isgood that could be sent.

3.3.3 Demonstration of Isgoods

The following table and text describe a possible execution of Support Based Distributed Search. Each line of the table presents a message transmitted from an agent to selected neighbours. To reduce the size of the table we have combined lines which involve identical messages, instead listing all agents that the message was transmitted to. To increase clarity we have also assumed that agents transmit in half-duplex with discrete time windows, though SBDS does not require this. Each ‘iteration’ is separated by a line for readability.

From	To	Argument
<i>a</i>	<i>b, i</i>	$\langle\langle a, 1\text{pm} \rangle\rangle$
<i>c</i>	<i>b, d, g</i>	$\langle\langle c, 1\text{pm} \rangle\rangle$
<i>e</i>	<i>d, f</i>	$\langle\langle e, 1\text{pm} \rangle\rangle$
<i>b</i>	<i>a</i>	$\langle\langle b, 1\text{pm} \rangle\rangle$
<i>b</i>	<i>c</i>	$\langle\langle a, 1\text{pm} \rangle, \langle b, 1\text{pm} \rangle\rangle$
<i>d</i>	<i>c, g</i>	$\langle\langle d, 1\text{pm} \rangle\rangle$
<i>d</i>	<i>e</i>	$\langle\langle c, 1\text{pm} \rangle, \langle d, 1\text{pm} \rangle\rangle$
<i>f</i>	<i>e</i>	$\{\langle e, 1\text{pm} \rangle\}$
<i>f</i>	<i>e, g, h</i>	$\langle\langle f, 3\text{pm} \rangle\rangle$
<i>e</i>	<i>f</i>	$\langle\langle e, 3\text{pm} \rangle\rangle$
<i>e</i>	<i>d</i>	$\langle\langle f, 3\text{pm} \rangle, \langle e, 3\text{pm} \rangle\rangle$
<i>c</i>	<i>b</i>	$\langle\langle c, 2\text{pm} \rangle\rangle$
<i>c</i>	<i>d, g</i>	$\langle\langle b, 1\text{pm} \rangle, \langle c, 2\text{pm} \rangle\rangle$
<i>g</i>	<i>c, d, f, h</i>	$\langle\langle g, 2\text{pm} \rangle\rangle$
<i>i</i>	<i>a, h</i>	$\langle\langle i, 2\text{pm} \rangle\rangle$

In the first iteration, *a*, *c* and *e* propose their initial values (all chose 1pm). Note that other agents could communicate at the same time (e.g. *g* and *h*) but to simplify the explanation we have limited the presentation of arguments.

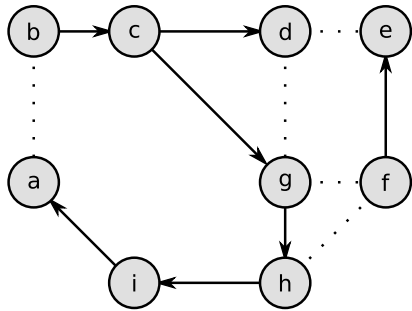
In the second iteration, *b*, *d* and *f* respond to these proposals. In the first case, *b* accepts the proposed time of 1pm from *a* and communicates an agreeable value to *a*. As *c* proposed a contradictory value, *b* provides a stronger isgood as a counter-proposal. At the same time, *d* accepts the proposed time of 1pm from *c* and communicates that to its neighbours. Finally, *f* rejects the proposal from *e* outright with a nogood, and proposes instead a time of 3pm.

In the third iteration, *e* and *c* respond to the counter-proposals. Given the stronger proposal from *f*, *e* must change its current value. When updating its neighbour *d*, *e* must provide a stronger argument. Similarly, *c* provides stronger proposals to *d* and *g*.

In the fourth iteration *g* and *i* announce their values for the first time, choosing values supported by *c* and *a* respectively.

From	To	Argument
d	c	$\langle(d, 2\text{pm})\rangle$
d	g	$\langle(c, 2\text{pm}), (d, 2\text{pm})\rangle$
d	e	$\langle(b, 1\text{pm}), (c, 2\text{pm}), (d, 2\text{pm})\rangle$
h	g	$\langle(h, 1\text{pm})\rangle$
h	i	$\langle(g, 2\text{pm}), (h, 1\text{pm})\rangle$
i	h	$\langle(i, 1\text{pm})\rangle$
i	a	$\langle(h, 1\text{pm}), (i, 1\text{pm})\rangle$
a	i	$\langle(a, 3\text{pm})\rangle$
a	b	$\langle(i, 1\text{pm}), (a, 3\text{pm})\rangle$
b	a	$\langle(b, 3\text{pm})\rangle$
b	c	$\langle(a, 3\text{pm}), (b, 3\text{pm})\rangle$
c	d, g	$\langle(b, 3\text{pm}), (c, 2\text{pm})\rangle$

Solved: $a, b, e, f = 3\text{pm}$; $c, d, g = 2\text{pm}$; $h, i = 1\text{pm}$



In the fifth iteration, d accepts the proposed time of 2pm from c . Note the increasing (and varying) lengths of isgoods as d communicates with different neighbours. Similarly, h accepts the proposed time of 2pm from g and chooses its first assignment accordingly.

In the sixth iteration, i changes its *support* from a to h , and changes its value accordingly. This change is then communicated to a . Note that this is the first communication from i to a .

In the seventh iteration, a is forced to choose between b and i as *support*. As a had no previous support it is free to choose either (if it previously had used b as *support* it would have retained it). In this instance, a chooses to use i as *support* and change value accordingly.

The final iterations consist of propagating this change to b (which changes its value), and then to c . Note that c does not change value, but still must communicate the change of b 's value to d and g .

It is interesting to consider the support relations once the algorithm is completed. We can see, for example, that g has been used as a support for c , and b for c . The variables b, c, d, g, h, i and a form a rooted tree of support relations.

However, e and f have been able to construct a solution which is independent but consistent with that of the other variables. The independence of variables was fortuitous in this instance, but will be explored in more detail in Chapter 4.

3.3.4 Postponement of Isgoods

As described previously, *compute-isgood* has the option of ‘postponing’ arguments (see lines 4-10 of Procedure 10.3). This section will explain the rationale for postponement.

Algorithms for distributed constraint satisfaction face a common difficulty; it is impossible to construct an algorithm that is self-stabilising, uniform, and still complete. This was proven in [CDK99], showing that no such algorithm is capable of solving the ring-ordering problem. The proof can be summarised as follows:

A distributed constraint satisfaction algorithm is self-stabilising if it may start from any initial assignment, and is guaranteed to converge to a consistent solution. A collection of agents are uniform if there is no means to distinguish between them. The 6-node ring-ordering problem defines a ring of 6 variables v_0, \dots, v_5 with constraints $(v_i + 1) \bmod 6 = v_{(i+1) \bmod 6}$, illustrated in Figure 3.2.

If each variable holds the same initial state (say, $v_i = 0$ for all i), then:

1. Each variable has the same constraints.
2. Each variable has the same internal state.
3. Assuming each variable operates at the same speed (fair scheduling of CPU time), then each will make the same decision.
4. By simultaneously making the same decision, each variable will be given the same value.
5. Therefore, each variable will always have the same constraints and internal state, and the algorithm will never terminate with a solution.

This proof may seem to be partially invalidated by the guarantee in SBDS that neighbours will not communicate with each other simultaneously (see lines 4-5 of *compute-isgood*). However, a slight variation of this proof was applied to the Distributed Breakout algorithm, which also prevents simultaneous communication.

$$\mathcal{V} = \{v_0, v_1, v_2, v_3, v_4, v_5\}$$

$$\mathcal{D} = \{0, 1, 2, 3, 4, 5\}$$

$$C = \left\{ \begin{array}{l} (v_0 + 1) \bmod 6 = v_1 \\ (v_1 + 1) \bmod 6 = v_2 \\ (v_2 + 1) \bmod 6 = v_3 \\ (v_3 + 1) \bmod 6 = v_4 \\ (v_4 + 1) \bmod 6 = v_5 \\ (v_5 + 1) \bmod 6 = v_0 \end{array} \right\}$$

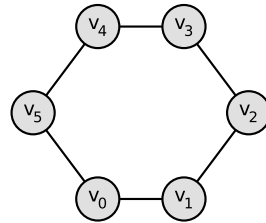


Figure 3.2: Constraint model and graph for the ring-ordering problem

The proof as presented in [ZW02] is remarkably similar to the proof from [CDK99]: an 8-node ring, with constraints that neighbours must hold different values, and a domain of just two values, was used to induce identical and simultaneous behaviour in all agents. The proof of incompleteness for Distributed Breakout relies on the possibility that two distinct partial solutions may exist in an implicit form on different nodes, and that the system oscillates between them. Unfortunately, this behaviour can also be seen in SBDS.

Consider a simple 4-node ring problem with variable $\mathcal{V} = \{a, b, c, d\}$, domain $\{0, 1\}$, and constraints $\{a = b, b \neq c, c = d, d \neq a\}$:

From	To	Argument
a	b, d	$\langle (a, 0) \rangle$
c	b, d	$\langle (c, 0) \rangle$
b	c	$\langle (a, 0), (b, 0) \rangle$
d	a	$\langle (c, 0), (d, 0) \rangle$
b	a	$\langle (b, 0) \rangle$
d	c	$\langle (d, 0) \rangle$
a	b	$\langle (d, 0), (a, 1) \rangle$
c	d	$\langle (b, 0), (c, 1) \rangle$
a	d	$\langle (d, 0), (a, 1) \rangle$
c	b	$\langle (b, 0), (c, 1) \rangle$
b	c	$\langle (d, 0), (a, 1), (b, 1) \rangle$
d	a	$\langle (b, 0), (c, 1), (d, 1) \rangle$
b	a	$\langle (a, 1), (b, 1) \rangle$
d	c	$\langle (c, 1), (d, 1) \rangle$
a	b	$\langle (c, 1), (d, 1), (a, 0) \rangle$
c	d	$\langle (a, 1), (b, 1), (c, 0) \rangle$
a	d	$\langle (d, 0), (a, 1) \rangle$
c	b	$\langle (b, 0), (c, 1) \rangle$
b	c	$\langle (c, 1), (d, 1), (a, 0), (b, 0) \rangle$
d	a	$\langle (a, 1), (b, 1), (c, 0), (d, 0) \rangle$
b	a	$\langle (a, 0), (b, 0) \rangle$
d	c	$\langle (c, 0), (d, 0) \rangle$
a	b	$\langle (b, 1), (c, 0), (d, 0), (a, 1) \rangle$
c	d	$\langle (d, 1), (a, 0), (b, 0), (c, 1) \rangle$
a	d	$\langle (d, 0), (a, 1) \rangle$
c	b	$\langle (b, 0), (c, 1) \rangle$

Initially, variables a and c (at opposite ends of the 4-node ring) choose and announce their values. As they have identical state, both choose 0.

In the next iteration, b and d each receive isgoods from a and c , and take one as their *support*. Assume that b takes a as *support*, and d takes c as *support*, so that each take the value 0. Due to the constraints, b now conflicts with c , and d now conflicts with a , so each sends stronger isgoods to those neighbours.

In the third iteration, a and c receive those isgoods from d and b respectively. Whereas they previously had *self* as *support*, they are now forced to choose their neighbours d and b . Simultaneously and independently, a and c change their values to 1.

In the fourth iteration, b and d receive the new (and stronger) isgoods from a and c . This induces b and d to again change value. Again, this is simultaneous and independent.

Eventually, two complete solutions exist in the ring, and each variable oscillates between them in turn.

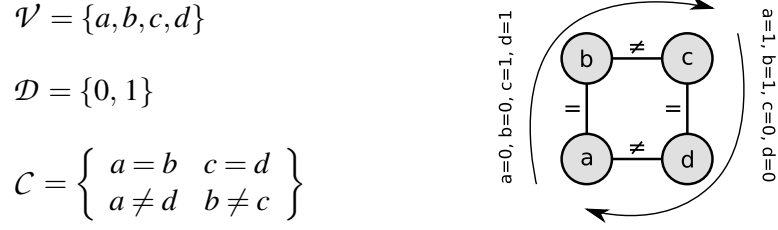


Figure 3.3: Constraint model and graph demonstrating cyclic behaviour in SBDS

If left unchecked, SBDS is able to maintain multiple ‘competing’ assignments simultaneously, and propagate those assignments in a cycle indefinitely. This behaviour occurs because each agent is acting identically; a mechanism is therefore required to selectively alter the behaviour of agents. Note, however, that a requirement in the design of SBDS was that there be no overarching means to differentiate between agents, for fear of forcing unfair levels of effort. The solution provided by SBDS is:

1. Detect any cycle of agents that are depending on each other for support. As isgoods are monotonic increasing in length, some isgoods must eventually list all agents that are participating in such a cycle. We know a cycle exists if an agent receives an isgood that prescribes a value for its variable, and then uses that isgood to construct its view.
2. Identify those isgoods propagated within the cycle that can be safely and temporarily eliminated from consideration. To achieve this, we must define an ordering such that partial assignments are comparable when they have the same scope. Note that the ordering of partial assignments may be completely different for different scopes.
3. Postpone propagation of identified isgoods, ensuring their elimination from the cycle. By using the ordering to identify which isgoods to postpone, we can guarantee that at least one will continue to propagate in the cycle. This selective postponement mechanism ensures that the cycle is eventually resolved.

The use of ‘postponement’ is best described by continuing the example execution of the problem in Figure 3.3. Assume that assignments are ordered so $\{(a,0), (b,0), (c,1), (d,1)\}$ is less than $\{(a,1), (b,1), (c,0), (d,0)\}$. The ordering of other assignments for the variables $\{a,b,c,d\}$ is irrelevant for this example, but would be defined in an actual system. The execution of SBDS described on the previous page would then continue as follows:

From	To	Argument
<i>b</i>	<i>c</i>	$\langle (c, 1), (d, 1), (a, 0), (b, 0) \rangle$
<i>d</i>	<i>a</i>	$\langle (a, 1), (b, 1), (c, 0), (d, 0) \rangle$
<i>b</i>	<i>a</i>	$\langle (a, 0), (b, 0) \rangle$
<i>d</i>	<i>c</i>	$\langle (c, 0), (d, 0) \rangle$
<i>a</i>	<i>b</i>	$\langle (b, 1), (c, 0), (d, 0), (a, 1) \rangle$
<i>c</i>	<i>d</i>	$\langle (d, 1), (a, 0), (b, 0), (c, 1) \rangle$
<i>a</i>	<i>d</i>	$\langle (d, 0), (a, 1) \rangle$
<i>c</i>	<i>b</i>	$\langle (b, 0), (c, 1) \rangle$
<i>b</i>	<i>c</i>	postponed transmission
<i>d</i>	<i>a</i>	$\langle (a, 0), (b, 0), (c, 1), (d, 1) \rangle$
<i>b</i>	<i>a</i>	$\langle (a, 1), (b, 1) \rangle$
<i>d</i>	<i>c</i>	$\langle (c, 0), (d, 0) \rangle$
<i>a</i>	<i>b</i>	$\langle (b, 0), (c, 1), (d, 1), (a, 0) \rangle$
<i>a</i>	<i>d</i>	$\langle (d, 1), (a, 0) \rangle$
<i>b</i>	<i>a</i>	$\langle (a, 0), (b, 0) \rangle$

Solved: $a, b = 0; c, d = 1$

The first and second iterations show the systems initially cycling between two solutions.

In the third iteration, *b* has received an isgood $\langle (b, 1), (c, 0), (d, 0), (a, 1) \rangle$ from *a*. However, the partial assignment represented by this isgood is ‘lesser’ than that which *b* previously sent to *c*. As the previously sent isgood is still consistent, and is ‘greater’ than what would be otherwise sent, *b* postpones sending messages to *c*. The postponement effectively blocks propagation of the ‘lesser’ partial assignment.

In the fourth iteration *b* receives the ‘greater’ partial assignment from *a* again. As *b* never propagated the ‘lesser’ partial assignment to *c*, no more messages are sent.

The algorithm terminates after the fifth iteration, with all agents agreeing on the solution.

Note that *b* still propagates the ‘lesser’ isgood to *a*, as there is no need to postpone such a message. Postponement is only applied in limited circumstances. Specifically, when *b* knows that a ‘greater’ isgood exists in the cycle of agents, and that either:

- the ‘greater’ isgood will eventually be received and used as *view* (as above); or
- the ‘greater’ isgood will be found inconsistent, and a nogood will be received; or
- the cycle of agents will be broken, and so nothing was lost by postponing.

Such limitations are required for the proof of completeness, which will be described later.

3.4 Results

With Support-Based Distributed Search, we have attempted to construct an algorithm which:

- has no need for ‘authority’ between variables, effectively avoiding the need for a total order on variables.
- does not add links between variables, and so avoids the eventual need for ‘broadcasting’ assignments.
- addresses the risk of cyclic behaviour exhibited by local search algorithms.

The fact that we do not add links between variables is evident from the algorithm itself. Similarly, we note the absence of any total ordering over the variables, avoiding any notion of ‘authority’. Each isgood establishes an order over variables within a local context in the form of a sequence in which assignments were made. This is necessary for the introduction of nogoods in the style of Dynamic Backtracking [Gin93, BMM01]. However, the combination of these local orders does not necessarily end in the construction of a total order over variables. In the general case, SBDS only constructs a ‘partial pre-order’ over variables, which is a very weak form of relationship.

We have also provided a novel method to address cyclic behaviour which plagues distributed local search algorithms [ZW02]. We must prove that cyclic behaviour has been eliminated, rendering SBDS sound and complete.

Lemma 1 *Eventually no new nogoods will be generated.*

Proof. Each agent keeps all nogoods it ever receives. A nogood is sent when a received isgood is found to be inconsistent, ensuring that the isgood will never be received twice from the same source. As the set of possible isgoods must be finite, eventually no new nogoods will be generated. \square

Note that it appears possible to use a nogood-deletion policy derived from that of Dynamic Backtracking [Gin93], though caution must be taken. Dynamic Backtracking has just one single variable ordering at any one time, allowing for nogoods to be deleted while guaranteeing that some information is always retained. It is common for SBDS to have multiple conflicting variable orderings and to contain cycles, and so information can be lost permanently if a nogood is deleted. To prevent information loss, it is possible to annotate a nogood with the variable ordering that was in use at the time of the nogood construction. Using this annotation it is possible to apply the nogood-deletion policy of Dynamic Backtracking safely, though the impact on algorithm performance has not been tested.

Lemma 2 *If no new nogoods are generated, then eventually the length of view will become stable for each agent.*

Proof. Let $W_i \subseteq \mathcal{V}$ be the set of variables whose *view* has length greater than or equal to $i \in \mathbb{Z}$. We will prove that any decrease in $|W_i|$ must be preceded by an increase in $|W_j|$, where $j < i$.

First, we note that an agent will never willingly reduce the length of its *view*, as per the requirements of *select-view*. So, in the usual case, $|W_i|$ will be monotonic increasing, for all i . However, in limited circumstances an agent may receive a shorter isgood from its *support*, and so the length of its *view* could be forced to decrease. Such events are rare, but they can occur whenever a cycle of supporting agents is formed.

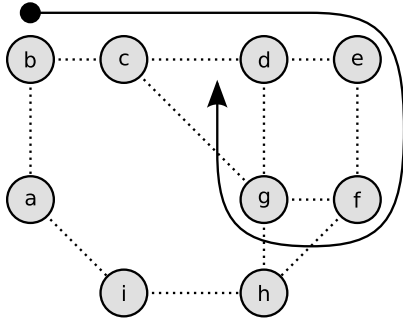
Let us assume that some agent v receives a shorter isgood I from its current *support*, and so v is forced to choose a shorter *view*. Let i be the length of v 's old *view*, and j be the length of v 's new *view*, respectively. The new, shorter *view* for v will obviously decrease each $|W_k|$, where $j < k \leq i$.

However, for v to have received the shorter isgood I , some agent w must have formed a cycle by changing its *support*. Note that w will only have selected a new *support* if it could increase the length of its own *view*, as per the requirements of *select-view*. Also note that the newly-formed cycle cannot involve more than j agents, else there would have been no reason to reduce the length of v 's *view*. Therefore, the length of w 's new *view* must then be less than or equal to j , but is certainly longer than its old *view*.

So, if an agent v is forced to reduce the length of its *view*, then there must be some preceding agent w which increased the length of its *view*. Further, w 's new *view* is guaranteed to be no longer than v 's new *view*. Therefore, the term $|W_1| \cdot |W_2| \cdot |W_3| \dots$ must increase lexicographically over time. As the term is bounded above, we can conclude that the length of *view* must eventually become stable for each agent. \square

Corollary 1 *If no new nogoods are generated, then eventually the support will become fixed for each agent.*

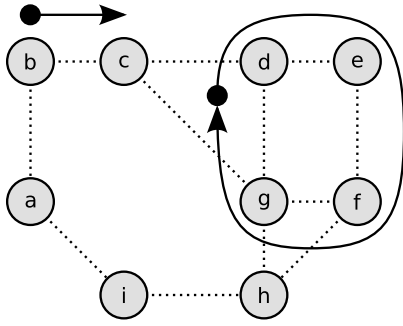
The above proof is best illustrated with an instance of our meeting scheduling example. Consider the following table showing possible *views* for each variable. A diagram showing the direction of the *support* relation is also provided. In the diagram, c has chosen b as support, d has chosen c as support, etc as indicated by the black arrow.



Example *view* held by each variable

b :	$\langle (b, 1pm) \rangle$
c :	$\langle (b, 1pm), (c, 2pm) \rangle$
d :	$\langle (b, 1pm), (c, 2pm), (d, 2pm) \rangle$
e :	$\langle (b, 2pm), (c, 1pm), (d, 1pm), (e, 2pm) \rangle$
f :	$\langle (b, 2pm), (c, 1pm), (d, 1pm), (e, 2pm), (f, 2pm) \rangle$
g :	$\langle (c, 1pm), (d, 1pm), (e, 2pm), (f, 2pm), (g, 1pm) \rangle$

In the above table, b , c and d have recently changed value, so the *view* held by g contains contradictory values to that held by d . However, g can provide a stronger argument to d in the form of an isgood $\langle (e, 2pm), (f, 2pm), (g, 1pm) \rangle$. As the argument provided by g is stronger than that provided by c , it forces d to choose g as a new *support* and update it's *view* accordingly. Following this choice, and after a few iterations, we can have the following situation:

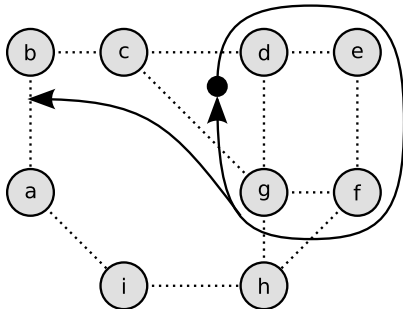


Example *view* held by each variable

b :	$\langle (b, 1pm) \rangle$
c :	$\langle (b, 1pm), (c, 2pm) \rangle$
d :	$\langle (e, 2pm), (f, 2pm), (g, 1pm), (d, 1pm) \rangle$
e :	$\langle (f, 2pm), (g, 1pm), (d, 1pm), (e, 2pm) \rangle$
f :	$\langle (g, 1pm), (d, 1pm), (e, 2pm), (f, 2pm) \rangle$
g :	$\langle (d, 1pm), (e, 2pm), (f, 2pm), (g, 1pm) \rangle$

The *view* held by each of d , e , f and g were affected by d 's choice. Critically, the length of *view* for f and g decreased. However, the minimum $\lceil view \rceil$ for affected variables has increased from 3 to 4. This demonstrates in a concrete way the mechanism described in the proof of Lemma 2.

Note that, in the above table, the *view* held by c is shorter and contradictory to that held by d and g . Therefore c and subsequently b would change their choice of *support* and *view*, giving us the following:



Example *view* held by each variable

b :	$\langle (g, 1pm), (c, 1pm), (b, 2pm) \rangle$
c :	$\langle (e, 2pm), (f, 2pm), (g, 1pm), (c, 1pm) \rangle$
d :	$\langle (e, 2pm), (f, 2pm), (g, 1pm), (d, 1pm) \rangle$
e :	$\langle (f, 2pm), (g, 1pm), (d, 1pm), (e, 2pm) \rangle$
f :	$\langle (g, 1pm), (d, 1pm), (e, 2pm), (f, 2pm) \rangle$
g :	$\langle (d, 1pm), (e, 2pm), (f, 2pm), (g, 1pm) \rangle$

While we do not argue that these examples provide any theoretical results, they do illustrate the proof of Lemma 2. With these results, a formal proof of soundness and termination (completeness) can now be presented.

Please note that neither of the proofs of Lemmas 1 and 2 accounted for or were affected by the postponement mechanism used in SBDS. While they offer proof that the length of *view* and value of *support* eventually become stable, they do not prove that it terminates with a correct answer. The proofs of soundness and completeness for SBDS will utilise Lemmas 1 and 2, but must also address the correctness of the postponement mechanism.

3.4.1 Soundness of Algorithm

Definition 11 *We will say that two variables u and v are in **agreement** if their respective views do not contain different assignments.*

Theorem 9 *Support-Based Distributed Search is sound.*

Proof. SBDS has no explicit notion of termination if the problem is satisfiable. However, it is possible for all agents to reach a state of ‘rest’, which we will treat as equivalent to termination for satisfiable problems, even if there exists no way to verify that all agents are resting. We must now prove that SBDS will not terminate unless the problem is proven unsatisfiable, or all agents are in a state of agreement with no unsatisfied constraints.

1. If SBDS states that the problem is unsatisfiable then it must have derived the empty nogood. As nogood derivation is obviously sound, we can conclude that SBDS will only say a problem is unsatisfiable if that is true.
2. Assume that agent u is not in agreement with agent v on the value of some variable w (it is possible that $u = w$ or $v = w$ but is not necessary). We know that both u and v obtained their current value of w through chains of agents which intersect at w .

If no agent postpones the transmission of a new *view* (see *compute-isgood*, Procedure 10.3) then each agent will update neighbours on the current value of w . As they are sending an update, the agents will not be at ‘rest’ and the algorithm will not have terminated.

If an agent in the chain postpones the transmission of a new *view* then it effectively causes a neighbour to be at rest until the conditions causing the postponement are cleared. However, using a total ordering on isgoods ensures that not all agents participating in the same cycle will postpone simultaneously. As at least one agent must continue to propagate the ‘greatest’ *view*, the algorithm has not terminated.

3. Assume that agent u is in conflict with agent v , and that the *view* held by u is stronger than or equal to that held by v . The *compute-isgood* procedure guarantees that $\lceil \text{sent}(v) \rceil > \lceil \text{recv}(v) \rceil$, and so, from the perspective of v , $\lceil \text{recv}(u) \rceil > \lceil \text{view} \rceil$. This condition will force v to change *view* to $\text{recv}(u)$, and so the algorithm has not terminated.

From the above, we know that SBDS will not terminate until the problem is proven unsatisfiable, or all agents are in agreement and no pair of agents are in conflict. Therefore, SBDS is sound. \square

3.4.2 Completeness of Algorithm

To prove completeness, we can use the proof system described in Chapter 2. We must provide a definition for a set of partial assignments I as a function of the current state of the algorithm, satisfying the following properties:

1. A solution to the constraint satisfaction problem is contained in I .
2. For each $s \in I$ where $|\hat{s}| > 1$, there exists $t \in I$ such that $t \subseteq s$ and $|\hat{t}| + 1 = |\hat{s}|$.
3. Testing $s \in I$ takes linear-time with respect to the size of the internal state of SBDS.

Theorem 10 *Support-Based Distributed Search is complete.*

Proof. Assume that SBDS is not complete. Let s denote the current complete assignment in an execution of SBDS, where $s(v)$ is that value observed by v itself. Let N be the union of all nogoods held by each agent. Let $I = \{t : \forall n \in N, n \not\subseteq t\}$ be a set of assignments, defined by the current set of nogoods. We will prove that $N' \supset N$ following a finite number of iterations.

Clearly, if a nogood is generated by any agent in an iteration, then $N' \supset N$. Assume instead that no new nogoods are ever generated. From Corollary 1 we know that the scope of *view* for each agent will eventually stabilise. As agents may only base their own value selection on their *view*, and we have assumed that SBDS does not terminate, we know that agents must be involved in a cycle. However, we have shown that the postponement mechanism will eliminate cycles in a finite number of steps. So to maintain the assumption that SBDS does not terminate, we can conclude that a new nogood must eventually be generated and $N' \supset N$. By the definition of I we can see that $I' \subset I$ and so I is convergent to some minimal set.

As I is determined by the nogoods N , and each nogood is only generated by an inconsistency, we can be sure that I and I' contain all solutions. Further, I clearly contains all partial solutions, and membership is testable in time linear in $|N|$. Therefore I satisfies the conditions of Theorem 1, and we have proven that it will converge. From these results and Theorem 1 we can conclude that SBDS is complete. \square

In the following chapter, we will consider some variants of SBDS. This will necessitate additional proofs of completeness, and will allow for interesting comparisons with other algorithms.

Chapter 4

Variations and Relations

In the previous chapter, the SBDS algorithm was introduced and described. We presented the simplest form of the algorithm, but our experience (from preliminary experimentation) is that it will perform poorly on even small problems. In this chapter we will investigate variations of SBDS that utilise heuristic guidance, and discuss how these modified forms of SBDS relate to other algorithms.

4.1 Minimising Conflicts

Minimising conflicts in value selection (normally shortened to min-conflicts [MJPL92]) is a well known heuristic used to improve algorithm performance. It allows an algorithm to focus on search subspaces most likely to contain a solution. We would like to include this heuristic within SBDS.

The min-conflict heuristic is generally applied when determining the value for a single variable. The value is chosen in such a way as to reduce the number of known or expected constraint violations. In some instances the constraints are dynamically or statically weighted, and this is included in min-conflict calculations [Mor93, VT95].

All existing algorithms for DisCSP either use min-conflicts directly, or can be modified to use min-conflicts. Such modifications are trivial when the mechanism used by an algorithm for completeness places no restrictions on value selection. However, our proof of completeness for SBDS made the statement “agents may only base their own value selection on their *view*”. Limiting the use of min-conflict to the current *view* makes no sense, as we are guaranteed consistency within the *view*. Adding a simple min-conflict heuristic to SBDS is a non-trivial task.

We will first demonstrate that SBDS is incomplete with a naive addition of min-conflict. We will change the second criteria of *select-view* so that it permits consideration of min-conflicting values.

Procedure 11.1 *select-view ()* with free choice of value

- 1: choose any neighbour v and value d such that:
 - v is equal to *support*, or $\lceil \text{recv}(v) \rceil > \lceil \text{recv}(\text{support}) \rceil$
 - d is any value consistent with $\text{recv}(v)$
 - d is consistent with each $\text{recv}(u)$, where $\lceil \text{recv}(u) \rceil > \lceil \text{recv}(v) \rceil$
 - 2: set *support* to v
 - 3: set *view* to $\text{recv}(v) + (\text{self}, d)$
-

By widening the possible choices of value, we allow for a variable to use the min-conflict heuristics. While this seems the most obvious method to allow for the inclusion of min-conflict, a simple proof of incompleteness is possible. Consider a 5-node problem with variables $\{a, b, c, d, e\}$, domain $\{0, 1\}$, and constraints $\{a = b, b \leq c, c = d, d = e, e = c\}$. Assume that a is the support for b , b is the support for c , and so on. Assume that all views are maximal, so that *view* of e is $\langle (a, 0), (b, 0), (c, 0), (d, 0), (e, 0) \rangle$. Consider what happens if a and b temporarily change from 0 to 1 and back, in the process forcing c to change value from 0 to 1:

1. initially $c = 1$, though $d = 0$ and $e = 0$
 2. c has value 1, so d is forced to change value to 1
 3. e has value 0, so c chooses to change value to 0
 4. d has value 1, so e is forced to change value to 1
 5. c has value 0, so d is forced to change value to 0
 6. e has value 1, so c chooses to change value to 1
 7. d has value 0, so e is forced to change value to 0
 8. c has value 1, so d is forced to change value to 1
 9. ad infinitum ...
-

$$\mathcal{V} = \{a, b, c, d, e\} \quad \mathcal{D} = \{0, 1\}$$

$$\mathcal{C} = \left\{ \begin{array}{lll} a = b & b \leq c & c = d \\ d = e & e = c & \end{array} \right\}$$

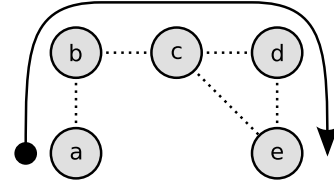


Figure 4.1: Constraint model and support graph for a simple 5-node problem

The above proof is similar to that used to prove incompleteness in Distributed Breakout [HY05, ZW02]. In the proof of incompleteness for Distributed Breakout, it was shown how the value of each variable in a ring problem could be effectively determined by the value of a single neighbour. Similarly, in the above proof of incompleteness for SBDS, the value of e is determined by d , the value of d is determined by c , and the value of c is determined by e . In both proofs, cyclic behaviour is created by carefully setting initial values for variables, causing variables to oscillate between alternative assignments.

This issue of cyclic dependencies has been resolved in existing distributed constraint satisfaction algorithms by limiting the use of min-conflict. For example, in AWCS, min-conflict is only applied when a variable with a higher priority has changed value, or when a nogood is generated. As the priority of variables forms a total order, this restriction ensures that cyclic dependencies never occur. Unfortunately, the reliance on a total ordering prevents SBDS from using this restriction directly. However, a similar approach can be constructed in SBDS by permitting limited use of the min-conflict heuristic after the scope of *view* changes:

Procedure 11.2 *select-view ()* with restricted choice of value

- 1: choose any neighbour v and value d such that:
 - v is equal to *support*, or $\lceil \text{recv}(v) \rceil > \lceil \text{recv}(\text{support}) \rceil$
 - d is the lowest value consistent with $\text{recv}(v)$, or d is equal to $\text{view}(\text{self})$, or the scope of *view* changed recently
 - d is consistent with each $\text{recv}(u)$, where $\lceil \text{recv}(u) \rceil > \lceil \text{recv}(v) \rceil$
 - 2: set *support* to v
 - 3: set *view* to $\text{recv}(v) + (\text{self}, d)$
 - 4: record whether the scope of *view* has now changed
-

Note that, as the first criteria remains unchanged, the monotonic-increasing nature of *view* will not be affected by the modifications presented above. Therefore, Lemmas 1 and 2, Corollary 1 and Theorem 9 (soundness) continue to hold. However, the proof of completeness requires small modifications.

Theorem 11 *Support-Based Distributed Search is complete, even if we permit discretionary selection value on each change of the scope of view.*

Proof. Assume that SBDS is not complete. Let s denote the current complete assignment in an execution of SBDS, where $s(v)$ is that value observed by v itself. Let N be the union of all nogoods held by each agent. Let $I = \{t : \forall n \in N, n \not\subseteq t\}$ be a set of assignments, defined by the current set of nogoods. We will prove that $N' \supset N$ following a finite number of iterations.

Clearly, if a nogood is generated by any agent in an iteration, then $N' \supset N$. Assume instead that no new nogoods are ever generated. From Corollary 1 we know that the scope of *view* for each agent will eventually stabilise. Assuming the algorithm has not terminated, then some variables are continuing to change value without changing the scope of *view*. As the scope has not changed, the value selection for each agent will eventually be based solely on the contents of their *view*, rather than the values of all neighbours. We can conclude that the agents must be involved in a cycle, but now the cycle is restricted to only involving agents within the *view*. However, we have shown that the postponement mechanism will eliminate such cycles in a finite number of steps. So to maintain the assumption that SBDS does not terminate, we can conclude that a new nogood must eventually be generated and $N' \supset N$. By the definition of I we can see that $I' \subset I$ and so I is convergent to some minimal set.

As I is determined by the nogoods N , and each nogood is only generated by an inconsistency, we can be sure that I and I' contain all solutions. Further, I clearly contains all partial solutions, and membership is testable in time linear in $|N|$. Therefore I satisfies the conditions of Theorem 1, and we have proven that it will converge. From these results and Theorem 1 we can conclude that SBDS is complete. \square

We have now established a mechanism by which we can introduce heuristics into SBDS. We explicitly included the ability to retain our previous value as, within constructive search algorithms, the reuse of previously assigned values is a commonly used heuristic, and can be superior to min-conflict in some instances. We have also allowed SBDS to use value selection heuristics for a limited time, dependent on the definition of ‘recently’ in line 1. The optimal choice of heuristic will be considered in the following chapter.

4.2 Minimising Communication

We have proven the ability of SBDS to support limited application of heuristics for value selection. Another heuristic, used extensively in local search algorithms, is to permit changes to early decisions. More accurately, local search algorithms are designed to minimise the cost incurred when changing early decisions.

Currently, SBDS is capable of changing values of variables regardless of the time of assignment. This clearly identifies SBDS as a form of local search. However, there is significant cost involved in changing early decisions; agents are required to update neighbours of changed information. Clearly, the less information propagated to neighbours, the lower the cost of updating information, and so the lower the cost of changing early decisions. It is worth considering methods to reduce the length of isgoods, thus reducing the amount of in-

formation propagated to neighbours. However, as shown in the previous sections, care must be taken to avoid disturbing completeness or, now, support for value selection heuristics.

A simple modification of *compute-isgood* is possible, and will permit each agent the option to skip increasing the length of isgoods for a finite number of steps.

Procedure 11.3 *compute-isgood* (v) with isgood reduction

```

1: let cycling be true iff self and  $v$  are the first two variables in  $recv(support)$ 
2: let conflicting be true iff  $\lceil sent(v) \rceil < \lceil recv(v) \rceil$  and  $view(this)$  is inconsistent with  $recv(v)$ 
3: let updating be true iff  $sent(v)$  is  $\langle \rangle$  or is not a tail of view
4: if cycling then
5:   if  $view \not\leq sent(v)$ , or  $sent(v)$  is not consistent then
6:     send view to  $v$ 
7:     set  $sent(v)$  to view
8:   end if
9: else
10:  let maximum be the longest tail of view that can be sent to  $v$ 
11:  let maintaining be true iff  $sent(v)$  was strengthened recently
12:  let discarding be true iff view was forced to change due to a nogood
13:  let preferred be initially 0
14:  if updating then set preferred to  $\lceil recv(v) \rceil$ 
15:  if not maintaining and not discarding then increase preferred by 1
16:  if conflicting then set preferred to  $\max(preferred, \lceil recv(v) \rceil + 1)$ 
17:  if preferred > 0 then
18:    let  $I$  be a tail of view such that  $\lceil I \rceil = \min(preferred, maximum)$ 
19:    if  $I \neq sent(v)$  then send  $I$  to  $v$ 
20:    set  $sent(v)$  to  $I$ 
21:  end if
22: end if

```

Line 11 allows the agent to defer strengthening the isgood it sends to a neighbour if it has already done so ‘recently’. The definition of ‘recently’ is assumed to indicate a finite number of iterations or perhaps a random coin toss. Line 12 allows for agent to again defer strengthening the isgood if a nogood was created that forced a change in *view*. This condition is normally triggered directly when an agent receives a nogood, but may be triggered indirectly if an agent’s *support* received a nogood. One effective technique is to set a ‘discarded’ flag whenever an agent’s *view* is impacted by a new nogood, and then pass this ‘discarded’ flag to neighbours. Each neighbour can then pass the ‘discarded’ flag to their neighbours if appropriate.

This change has no impact on Lemmas 1 and 2. Nor does it impact the postponement mechanism; the scope of *view* for cycling agents is still monotonic increasing, albeit at a slower rate. With this change, SBDS remains sound and complete, but reduces the amount of information propagated to neighbours by slowing the rate of growth of isgoods.

Note, however, that empirical evaluations of SBDS will require us to define ‘recently’ and ‘forced to change’ in more concrete terms, and there are trade-offs. If we limit the growth of isgoods too much, then the postponement mechanism may take a long time to trigger. If we allow too much growth of isgoods, then the algorithm will begin sending unnecessary information.

4.3 Minimising Storage

A common method for achieving completeness in local search algorithms is to simply retain an ‘unbounded’ collection of nogoods. Assuming the algorithm never becomes trapped in an infinite cycle, it can use the collection of nogoods as a means to monotonically eliminate portions of the search space. Unfortunately, this method may require a significant amount of storage space as the algorithm accumulates more nogoods. Dynamic Backtracking Search uses elimination explanations, which are fundamentally identical to the concept of nogoods, but guarantees a polynomial bound on the number of eliminating explanations by deleting any explanation that is no longer valid.

It is impossible for SBDS to use precisely the same mechanism as DBS. SBDS has no global total order, which DBS uses implicitly to guarantee that some eliminating explanations will never be deleted. However, we can achieve a similar effect by adding a new annotation to every nogood.

Assume that every nogood N is annotated with a non-negative integer p indicating the ‘inference steps’ of the nogood. We will use the notation N_p for a nogood N annotated with p . We will now modify SBDS so that a nogood:

1. is created with an annotation greater than the nogoods used in its inference; and
2. is deleted if it was used in deriving a new nogood, and has an annotation less than some fixed bound P .

Algorithm 12 *support-based-distributed-search* () with nogood deletion

```
1: set support to self
2: set recv(self) to  $\langle \rangle$ 
3: set nogoods to  $\{ \}$ 
4: set P to some finite positive integer
5: while a choice of value is consistent with the empty isgood  $\langle \rangle$  do
6:   compute-nogood(v) for each neighbour v
7:   select-view()
8:   compute-isgood(v) for each neighbour v
9:   when a nogood  $N_p$  is received
10:    set nogoods to nogoods  $\cup \{N_p\}$ 
11:   when an isgood I is received from some neighbour v
12:    set recv(v) to I
13:   end when
14: end while
```

Procedure 12.1 *compute-nogood* (*v*) with nogood deletion

```
1: if no choice of value is consistent with recv(v) then
2:   let N be an inconsistent subset of recv(v)
3:   let used be a set of nogoods used to prove recv(v) was inconsistent
4:   let p be the minimal annotation on the nogoods in used, or 0 if used is empty
5:   let usedp be the subset of used with annotations greater than p
6:   set nogoods to nogoods - usedp
7:   send  $N_{p+1}$  to v
8:   set recv(v) to  $\langle \rangle$ 
9:   if support = v then
10:    set support to self
11:   end if
12: end if
```

These modifications reduce the number of nogoods stored by SBDS by allowing a bounded number of nogood deletions. This parallels the behaviour of Dynamic Backtrack Search, but avoids using a total order to bound nogood deletions. It also requires a new proof of completeness.

Theorem 12 *Support-Based Distributed Search, with bounded nogood deletions, is complete.*

Proof. Assume that SBDS is not complete. Let s denote the current complete assignment in an execution of SBDS, where $s(v)$ is that value observed by v itself. Let N_p be the union of all nogoods with annotation greater than or equal to p , so that N_0 is all nogoods held by all agents. Let $I_p = \{t : \forall n \in N_p, n \not\subseteq t\}$ be a set of assignments, and so is defined by the set of nogoods N_p . We will define I as I_0 , and prove that it converges to a minimal set. Note that $I = I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots \subseteq I_P$, and so to prove convergence of I it is sufficient to prove that any I_p is convergent.

We will first prove that, for SBDS to be incomplete, it must continue to generate nogoods. Assume that no new nogoods are ever generated. From Corollary 1 we know that the scope of *view* for each agent will eventually stabilise. As agents may only base their own value selection on their *view*, and we have assumed that SBDS does not terminate, we know that agents must be involved in a cycle. However, we have shown that the postponement mechanism will eliminate cycles in a finite number of steps. So to maintain the assumption that SBDS does not terminate, we can conclude that a new nogood must eventually be generated.

Now, in the construction of a nogood, it is possible that other nogoods are deleted. If a nogood with annotation $p + 1$ is generated in an iteration, then $N'_{p+1} \supset N'_{p+1}$, but it is possible that $N'_p \not\supseteq N_p$. There is one exception to this, as it is always the case that $N'_p \supseteq N_p$ in every iteration. For each iteration of SBDS, let q be the smallest non-negative integer such that $N'_q \supseteq N_q$ for all future iterations. We know that $0 \leq q \leq P$ and, by definition, q must be monotonic decreasing over the execution of SBDS.

Now, a nogood is only ever deleted if a new nogood (with higher annotation) was produced. As q was defined to be the smallest suitable integer such that $N'_q \supseteq N_q$, we know that a new nogood must be added to N_q in a later iteration, and so eventually $N'_q \supset N_q$. Further, as q is bounded below by 0 and was shown to be monotonic decreasing, it will eventually converge to some fixed value. Therefore, after a finite number of iterations of SBDS we know that q becomes fixed, and there is some N_q which is (strictly) monotonic increasing (and so convergent). As $N_0 \supseteq N_q$, we can conclude that N_0 is also convergent (though not necessarily monotonic increasing.) to some maximal set. The set of partial assignments I is determined by N_0 , and so we can conclude that I is convergent to a minimal set.

As I is determined by the nogoods N_0 , and each nogood is only generated by an inconsistency, we can be sure that I and I' contain all solutions. Further, I clearly contains all partial solutions, and membership is testable in time linear in $|N_0|$. Therefore I satisfies the conditions of Theorem 1, and we have proven that it will converge. From these results and Theorem 1 we can conclude that SBDS, with bounded nogood deletions, is complete. \square

4.4 Relation to Other Algorithms

Support-Based Distributed Search has been introduced using human concepts such as ‘proposals’ and ‘rejections’. However, from a purely computational perspective, many parallels can be drawn to other algorithms.

4.4.1 Asynchronous Weak-Commitment Search (AWCS)

Support-Based Distributed Search operates on similar problems and will be empirically compared to Asynchronous Weak-Commitment Search. It is therefore worthwhile comparing and contrasting SBDS with the Weak-Commitment Search family of algorithms.

AWCS	SBDS
A variable is forced to choose a value consistent with neighbours which have higher priorities. A variable will try to choose a value consistent with all neighbours.	A variable is forced to choose a value consistent with neighbours which have stronger (incrementally revealed) views. A variable will try to choose a value consistent with all neighbours.
Each variable communicates its current value to its neighbours. Variables may communicate with any other variable that they have been made aware of with nogoods. The current view for a variable is determined by the combination of values from all neighbours with higher priorities.	Each variable communicates its current value, and those which it takes as support, to its neighbours. Variables may communicate only with neighbours. The current view for a variable is determined by the values provided by a single neighbour with a stronger or equal view.
Inconsistent assignments between multiple variables are explicitly detected and recorded using nogoods.	Inconsistent assignments between multiple variables are explicitly detected and recorded using nogoods, but only if those variables are simultaneously mentioned in an isgood.

The only similarities between AWCS and SBDS are the use of nogoods, the min-conflict heuristic, and the identification of a subset of neighbours which are ‘stronger’ or ‘higher’. SBDS places significant restrictions on which variables can communicate with each other, and on the construction of nogoods. Where AWCS combines the values provided by all neighbours, SBDS chooses a single neighbour for its view.

4.4.2 Distributed Breakout (DBO)

In contrast, Support-Based Distributed Search shares much in common with modified hill-climbing algorithms such as Breakout and Distributed Breakout. With trivial changes, SBDS can be made vulnerable to the same incompleteness problems of such algorithms. In many ways, SBDS can be considered an extension of such algorithms to provide completeness. Consider the following table, describing the main characteristics of Breakout, and corresponding characteristics in SBDS:

DBO	Support-Based Distributed Search
At each iteration, conflicts are handled by increasing the weights of constraints, or by choosing an alternative value. Constraint weights are unbounded. Each variable chooses a value which is max-consistent with weights of constraints.	At each iteration, conflicts are handled by increasing the strength of isgoods, or by choosing an alternative value. Strength of isgoods are bounded. Each variable chooses a value which is consistent with the maximal received isgood.
It is possible for multiple solutions to be propagated in a cycle of variables, causing an infinite loop and incompleteness. Inconsistent assignments between multiple variables are never explicitly detected and recorded.	It is possible for multiple solutions to be propagated in a cycle of variables, but the cycle will eventually be detected and resolved, maintaining completeness. Inconsistent assignments between multiple variables are explicitly detected and recorded, but only if those variables are simultaneously mentioned in an isgood.

As can be seen above, there are significant parallels between the Breakout family of algorithms and SBDS. Most notably, the strength of an isgood is a reasonable parallel to constraint weights. SBDS is only to provide completeness by:

- placing a bound on the strength of isgoods (similar to constraint weights)
- being able to use nogoods as increasing isgood strength provides more information
- using the increasing amount of information in each isgood to resolve cycles

It is interesting to note that similar ‘arc weighting schemes’ refer to constraint weights as a means to represent knowledge about the search space [Fra97, Fra96]. Indeed, this is exactly what the isgoods of SBDS provide.

4.4.3 Asynchronous Backtracking With Dynamic Ordering (ABT-DO)

ABT-DO is the second algorithm which we will be empirically compared in the following chapter. The primary characteristic of ABT-DO is the ability to dynamically reorder variables without losing the polynomial-size nogood store of ABT. This ability is dependant on maintaining and dynamically updating a total order over all agents, with strict rules on re-orderings. In many ways ABT-DO is very similar to Dynamic Backtracking, but modified to function in a asynchronous and distributed setting.

ABT-DO	SBDS
The current view for a variable is an ordered sequence of variables and their assignments. A variable is forced to choose a value consistent with its view.	The current view for a variable is an ordered sequence of variables and their assignments. A variable is forced to choose a value consistent with its view.
At any time there is only one, global ordering of variables. The ordering of variables is total, and so cannot contain cycles.	At any time there are multiple, local orderings of variables (normally not contradictory). The ordering of variables is not total, and so can contain cycles.
Nogoods are issued to variables in the order prescribed by the current view. Nogoods are removed when a new, superior nogood is found (made possible by having a total order).	Nogoods are issued to variables in the order prescribed by the current view. Nogoods are never removed (this rule can be weakened if the algorithm bounds the number of such removals).
The relative positions of lower-ranked variables can be arbitrarily altered by any higher-ranked agents. No variable can elevate its own rank over higher-ranked variables.	The rankings of variables are chosen by the variables themselves, limited by the arguments presented by neighbours. A variable can attempt to elevate its own rank over others, but it may not be successful.
Conflicts are handled by making lower-ranked variables choose a new value, or construct a new nogood. Min-conflict is not used by higher-ranked variables.	Conflicts are handled by increasing argument strength and so altering the local ordering, or by constructing a new nogood. Min-conflict should be used whenever possible.

Chapter 5

Empirical Analysis

In this chapter we will present a number of results that compare SBDS, AWCS, ABT and ABT-DO on different problem classes. These results should identify the appropriateness of each algorithm to solving different types of DisCSP instances. However, it should be noted that this comparison is limited to randomly generated problem instances only. The impact of problem structure on each DisCSP algorithm has not been presented due to space considerations. Appendix A contains a large collection of tables and graphs if further details are necessary.

5.1 Metrics

When comparing distributed constraint satisfaction algorithms, there are a number of possible metrics that can be used. To allow for a broad comparison of algorithms, we use the following four measures of algorithm performance:

- Constraint Checks
- Nogood Checks
- Bytes
- Packets

While these performance measures can be useful to compare different DisCSP algorithms, they provide an incomplete picture of their behaviour. To provide further information, we also use the following measures:

- CPU Time
- Bytes Per Packet
- Concurrent Checks

- Concurrent Traffic

To ensure that the results are interpreted correctly, we provide a brief description of each measure below.

5.1.1 Total vs Non-Concurrent Measures

The total value of any of the above measures is simply defined as the accumulated results from each agent. For example, the total number of bytes sent by the algorithm is the sum of the number of bytes sent by each agent. The ‘non-concurrent’ value can be best described as the ‘longest chain of sequential operations necessary for the algorithm to function’. A non-concurrent measure is generally preferable when analysing distributed algorithms as we can presume the number of available CPUs scales in the number of agents.

For a full discussion of ‘non-concurrent’ counts see [ZM06d], with some earlier work in [MKRZ02]. We will only briefly describe how a ‘non-concurrent’ value is maintained during algorithm execution:

1. Each agent holds a ‘non-concurrent’ value for each measure. Each operation performed by an agent causes it to increase the appropriate ‘non-concurrent’ value(s).
2. When an agent sends a message to a neighbour, it attaches its ‘non-concurrent’ values. When the receiving agent uses that message, it must compare its own ‘non-concurrent’ values with those in the message and keep the greater of the two.
3. The ‘non-concurrent’ value of a measure is defined as ‘the maximum non-concurrent value held by any agent for that measure’.

Note that it is critical that a receiving agent only update its ‘non-concurrent’ values when it first makes use of a message. A message which is received, but never used, does not influence the ‘non-concurrent’ results of an algorithm. We say that an agent ‘uses’ a message only when the agent processes the message in any way after initial reception, evaluation, and storage.

For example, an agent in AWCS may receive and store a nogood, but does not actually ‘use’ it until the nogood is checked against the current view. Note that if the nogood is found to be consistent it is still considered to have been ‘used’, despite having no impact on the decision-making process.

However, if a message is received, evaluated, and then discarded it will never be ‘used’. For example, ABT-DO may receive a nogood and immediately discard it for being out-of-date, without having said to have ‘used’ it.

5.1.2 Constraint Checks

A ‘constraint check’ is a test of whether a constraint c is satisfied by some assignment s . We presume that some mechanism is used within each agent to avoid checking a constraint c when the relevant variables are not in \hat{s} . For example, agents in ABT, ABT-DO and AWCS do not perform a ‘constraint check’ if the constraint involves variables which are ranked beneath them. Similarly, agents in SBDS do not perform a ‘constraint check’ if the constraint involves variables which are not present in an isgood.

5.1.3 Nogood Checks

A ‘nogood check’ is customarily defined as ‘any test of whether a nogood matches an assignment s ’. However, this definition is not sufficient as the handling of nogoods in ABT and ABT-DO are quite different from that of AWCS and SBDS:

- The nogood store of ABT and ABT-DO should be able to maintain the ‘current set of nogoods’ very cheaply. This store should answer the question ‘is the value v currently eliminated by any current nogood?’ in constant time. Each time such a question is asked is considered a ‘nogood check’.
- The nogood store of AWCS and SBDS is monotonic increasing in size, and does not maintain a ‘current set of nogoods’. A simple array of nogoods can become very expensive to check, and so we have used an efficient pattern-matching tree instead. Using such a tree structure complicates the question of when a ‘nogood check’ has been performed. For this thesis, we count a nogood n as having been ‘checked’ if, while attempting to test the assignment s , we encounter a branch which was created when adding the nogood n to the tree.

As a result of these differences, the reader should not try to compare nogood checks between AWCS/SBDS and ABT/ABT-DO. The numbers from each algorithm are presented together only for convenience, and to emphasise the difference in approach.

5.1.4 Bytes

To determine the number of bytes sent by each algorithm, we use the following sizes for messages:

- A ‘nogood message’ by any algorithm is 4 bytes, plus 8 bytes for every assignment in the nogood (source variable, and the nogood itself).
- A ‘value message’ by AWCS is 12 bytes (source variable, new value, and new priority).
- A ‘value message’ by ABT or ABT-DO is 8 bytes (source variable, and new value).

- An ‘ordering message’ by ABT-DO is 4 bytes, plus 4 bytes for every variable in the CSP (source variable, and the ordering itself).
- An ‘isgood message’ by SBDS is 8 bytes, plus 8 bytes for every assignment in the argument (source variable, discard flag, and the isgood itself).

5.1.5 Packets

It is assumed that a message consists of a single packet of data transmitted from a source to a single destination. For example, if AWCS constructs a nogood involving 4 variables, then AWCS will send 4 distinct packets of information. Similarly, when ABT-DO sends an updated ordering message to all lower agents, it will send one distinct packet to each receiving agent. As nogoods are the only packet type which is common to all algorithms, we will present two separate results - ‘nogood packets’ and ‘other packets’.

5.1.6 Bytes Per Packet

The average packet size can be very informative when looking at the scaling behaviour of algorithms. Some algorithms appear to have varying packet sizes as we scale one or two problem parameters. This measure can identify possible scaling issues in each algorithm.

5.1.7 CPU Time

All experiments were performed on a single core of a 3GHz Intel Core 2 Duo PC running Linux, and using Sun’s Java 1.6 Server VM. The CPU times reported are total for the simulation of the entire algorithm, including some small amount of instrumentation overhead. While efforts have been made to ensure all algorithms are equally efficient, the CPU time measure should not be taken as a real indication of algorithm performance. At most, we can use this measure to comment on the scaling behaviour of each algorithm.

5.1.8 Concurrent Checks and Concurrent Traffic

It can be interesting to look at the difference between ‘non-concurrent constraint checks’ and ‘total constraint checks’. For example, a small difference indicates that few agents are performing checks simultaneously, and that the algorithm is fundamentally sequential. A large difference may indicate that the algorithm is highly concurrent (though it may equally indicate that the algorithm is highly inefficient). We define ‘concurrent checks’ for each algorithm as ‘total constraint and nogood checks’ divided by ‘non-concurrent constraint and nogood checks’.

Similarly, we define ‘concurrent traffic’ as being ‘total traffic’ divided by ‘non-concurrent traffic’. This measure can be seen as indicating the amount of traffic being sent or received simultaneously within an algorithm.

It should be noted that neither of these measures has been presented in literature that was reviewed for this thesis and are fairly novel. While they provide some indication of how much work is being performed simultaneously, it is critical to note that this measure does not indicate algorithm performance. These measures can at most support other claims, or validate expectations of scaling behaviour.

5.2 Implementation

All experiments were performed using algorithm code implemented in Java. A very simple ‘simulator’ was used in place of a real distributed cluster of nodes. The simulator is admittedly very primitive, with all messages delivered instantaneously.¹ All calls to an algorithm’s ‘main’ procedure are deferred, and so an agent will likely receive multiple messages before making any decisions. As a result, each agent has always received the latest information before making decisions about value assignments, nogoods, etc.

All algorithms make use of a simple ‘brute force’ nogood resolution technique, with no variable-selection heuristics. This technique only guarantees that a nogood is minimal in terms of set inclusion, but not cardinality. It is also assumed that each agent caches the information used to detect a domain wipeout, and so no additional constraint/nogood checks are needed to perform resolution.

Finally, it should be noted that none of the algorithm implementations have been reviewed by third parties. The results presented in the following sections should be taken as indicative, and not as a conclusive comparison of algorithm performance. We will therefore limit the majority of our conclusions to algorithm scaling behaviour, rather than direct numerical comparisons.

5.2.1 ABT and ABT-DO Implementation Notes

ABT-DO was implemented by following the pseudo-code in Chapter 2, which was derived from [ZM06a]. However, the following implementation details may affect the experiment results. All efforts were made to improve algorithm performance while not deviating from the intention of the authors.

¹We acknowledge that instantaneous message delivery is a limitation of this study. The impact of message delays has been pointed out in [ZM06e, ZM06b].

- When checking consistency of an assignment s , we first check the nogood store before checking the constraint store.
- Each agent always uses the smallest possible value as its assignment. No value selection heuristics such as min-conflict are used.
- ‘Value’ messages are only sent after a variable actually changes value, or because a nogood was out-of-date.
- ‘Value’ messages are sent to all neighbours, regardless of their position in the ordering.
- ‘Ordering’ messages are only sent if a new ordering is computed, and are sent separately from ‘value’ messages. This significantly reduces network traffic.
- ‘Ordering’ messages are sent to all lower-ranked agents, whether they are neighbours or not. All agents are aware of the initial ordering.
- ‘Ordering’ messages are not sent to higher-ranked agents. Sending the ordering message to higher-ranked agents would reduce the number of constraint checks for ABT-DO, but would also significantly increase the total network traffic.
- ‘Nogood’ messages may contain variables which are unknown to the recipient agent. An agent can establish a new ‘neighbour’ relationship by the mutual exchange of ‘value’ messages.
- An agent may receive multiple nogoods before *select-value()* is called. Therefore, each agent must record the set of neighbours from which it has received a nogood, and take care to send a ‘value’ message to each of those agents.
- If an agent receives a nogood, then it alters the ordering to maximise the rank of the sending agent. This is called the nogood-triggered heuristic [ZM05b, ZM06a].

Note that our implementation of ABT-DO does not send an ‘ordering’ message unless the ordering has changed. ABT was therefore implemented by using the same code as ABT-DO and maintaining a simple static ordering.

5.2.2 AWCS Implementation Notes

AWCS was implemented by following the pseudo-code in Chapter 2, which was derived from [YH00b]. However, the following implementation details may affect the experiment results. Again, all efforts were made to improve algorithm performance while not deviating from the intention of the authors.

- When checking consistency of an assignment s , we first check the constraint store before checking the nogood store.
- Value selection is only triggered by receiving a value from a higher-ranked neighbour.

5.2.3 SBDS Implementation Notes

SBDS was implemented by following the pseudo-code in Chapter 3 and the modified pseudo-code in Chapter 4. However, the following implementation details may affect the experiment results.

- When checking consistency of an assignment s , we first check the constraint store before checking the nogood store.
- Each agent is allowed to switch its value and support 10 times without increasing the strength of its view, in accordance with Section 4.1.
- Each agent is allowed to send equal-strength arguments to its neighbour approximately 10 percent of the time, using a weighted coin flip in accordance with Section 4.2.

Note that the choices of ‘10 times’ and ‘10 percent’ are mostly arbitrary. A number of possible values were tested, with very mixed results depending on the problem class. A larger study would be necessary to examine the behaviour of SBDS with different settings and on different problem classes.

5.2.4 SBDS Value Selection Heuristic

In most DisCSP algorithms, an agent has only one or two possible definitions of ‘world view’. As a result, there is only a small selection of ‘intuitive’ value selection heuristics. However, each isgood received by an SBDS agent constitutes a possible world view, increasing the number of possible value selection heuristics. In the previous chapter we described how to use value selection heuristics with SBDS, but did not fully describe any heuristic. In this section we will briefly describe a variety of value selection heuristics, and then attempt to ‘tune’ SBDS for a sample problem class.

For each received isgood i , an SBDS agent can quickly determine whether a value v is consistent by consulting its constraints and nogoods. The only strict requirement of an SBDS agent is that it choose a consistent value and support that is stronger than any conflicting isgoods from neighbours. However, it seems sensible to choose a value which is consistent with the most isgoods, or inconsistent with the least isgoods. We can therefore derive a number of value selection heuristics:

Number Support - Choose a value that maximises the number of consistent isgoods. This heuristic attempts to minimise the number of neighbours that need to change their value and/or support. The hope is to avoid ‘cascading’ changes where a small change forces many other agents to change their value.

Total Support - Choose a value that maximises the combined strengths of consistent isgoods. This heuristic also attempts to avoid cascading changes, but weights each isgood according to its ‘strength’. We assume that the ‘strength’ of an isgood is roughly correlated with the computational effort tied up in its construction.

Minimum Against - Choose a value that minimises the maximum strength of any inconsistent isgood. This heuristic attempts to select a value which has a lower risk of being defeated by a neighbour. The assumption is that neighbours who have sent weaker isgoods are more likely to accept a change in value.

Maximum Strength - Choose a value that maximises the maximum strength of a consistent isgood. This heuristic also attempts to select a value with a low risk of being defeated. In this case, it is presumed that use of the strongest argument is the best mechanism to achieve agreement amongst neighbours.

Same Support - Choose the same support as was previously held. This heuristic attempts to maintain the same support, even if that requires a change in value. The assumption is that frequent changes are disruptive and waste computational effort.

Same Value - Choose the same value as was previously held. This heuristic attempts to select the same value, even if that requires a change of support. The assumption again is that frequent changes are disruptive and waste computational effort.

It is of course possible to chain the above value selection heuristics together. For example, an agent can attempt to maintain the same support, and ‘tie-break’ alternative value assignments by maximising the number of consistent isgoods.

We evaluated different combinations of heuristics by considering problems with 15 variables, 15 values, 50 constraints, and a constraint tightness of 0.5. Most problems of this class are solvable, but are relatively difficult. We do not present detailed results here as we wish to focus on the scaling behaviour of SBDS and other DisCSP algorithms rather than a comparison of value selection heuristics. The rest of this chapter will assume that SBDS is using the following sequence of heuristics for value selection:

1. Number Support
2. Same Support
3. Same Value
4. Minimum Against
5. Maximum Strength
6. Total Support

5.3 Problem Sets

Many existing DisCSP reviews take a few representative problem configurations and then compare algorithm performance over different levels of ‘constraint tightness’. However, the primary motivation of this thesis was large DisCSP instances. Therefore, we are more interested in comparing algorithm performance as we increase the size of the problem, regardless of the constraint tightness.

Specifically, we would like to see how each algorithm scales as we increase the number of variables, or the domain size, or the number of constraints. Therefore, the three parameters we are interested in scaling are:

- **Domain** - the number of values in the domain of each variable
- **Degree** - the average number of constraints on each variable
- **Variables** - the number of variables or agents in the problem instance

Note that ‘constraint tightness’ does not appear as a problem parameter. In actual fact, each experiment we conduct will use 41 different constraint tightness values (from 10% to 90%) and 400 different random seeds. In this way, we hope to provide a representative selection of feasible and infeasible problems, while concentrating our analysis on scaling behaviour. In cases where algorithm behaviour is substantially different on feasible or infeasible instances, we will present those results separately.

Also note that we use the notion of ‘degree’ and not the more familiar ‘constraint density’ to define our problem classes. ‘Degree’ appears to be more appropriate if we accept that the number of connections in a distributed system is scaled linearly in response to the number of nodes.² The ‘constraint density’ can be easily computed as $\frac{\text{Degree}}{\text{Variables}-1}$. A problem instance with 30 variables and a degree of 12 would have $30 \times 12/2 = 180$ constraints, or a density of roughly 0.414.

We can then define three distinct problems sets of interest:

	Variables	Degree	Domain
Problem Set 1	30	4	3 to 17
Problem Set 2	30	3 to $12\frac{1}{3}$	5
Problem Set 3	10 to 150	4	5

²A new telephone exchange technology would be tested on its ability to support an increasing number of residences, with the assumption that each residence will connect to an average of 5 other residences in a day. If 10 residences make 50 calls, then 20 residences would make $\frac{20}{10} \times \frac{50}{10} = 100$ calls, and not $\frac{50}{10 \times 9} \times (20 \times 19) = 211\frac{1}{9}$.

Within each problem set we have used 15 different parameter configurations. For example, Problem Set 1 consists of problems instances with 30 variables, degree of 4, and then one of 15 possible values for domain size. To provide reasonable accuracy in our results, we have then performed 16,400 runs of each algorithm on each parameter configuration. This consists of the 400 different random seeds, and 41 different values of constraint tightness. To be clear on the total number of experiments run per problem set, consider the following:

$$\begin{array}{rcl}
15 \text{ parameter values} & \times & \\
4 \text{ algorithms} & = & 60 \text{ configurations per problem set} \\
400 \text{ random seeds} & \times & \\
41 \text{ tightness values} & = & 16,400 \text{ experiments per configuration} \\
16,400 \text{ experiments} & \times & \\
60 \text{ configurations} & = & 984,000 \text{ experiments per problem set}
\end{array}$$

For each problem set, we will present a set of highlighted results with commentary, followed by a conclusion. For the most part, the commentary will focus on the scaling behaviours of each algorithm, with less emphasis on direct performance comparisons. As such, the majority of graphs will be presented with a logarithmic vertical axis. If two algorithms have the same scaling behaviour (i.e. differentiated only by a constant factor), then they should present almost the same curve. A comprehensive set of result tables are presented in Appendix A.

5.4 Results for Smaller Problems

In general, the scaling behaviour for each algorithm was very similar on problem sets 1 and 2. The instances in these problem sets can also be characterised as relatively ‘small’ with just 30 variables. Therefore, we will analyse the results of those problem sets together.

First, it should be noted that different percentages of the results are ‘infeasible’ depending on the problem parameters used. Figure 5.1 shows the percentage of problems which were ‘feasible’ for each parameter value. This should be considered when reading our analysis.

Observation 1 *For problems with high constraint density or domain size, ABT/ABT-DO performed far fewer constraint checks than AWCS or SBDS. AWCS has similar totals to SBDS, but SBDS performed many more constraint checks concurrently.*

Figure 5.2 illustrates the total and non-concurrent constraint checks for each algorithm over different parameter values. We should reiterate that each graph uses a logarithmic axis as this allows for an easier comparison of algorithm scaling behaviour. For example, all

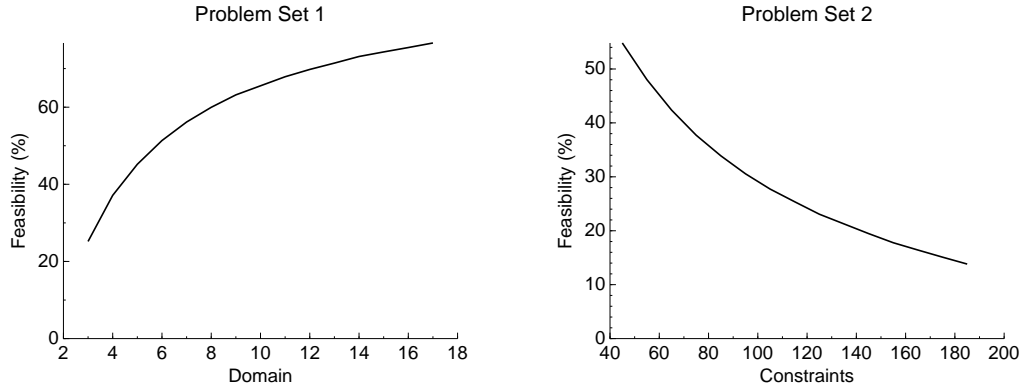


Figure 5.1: Average feasibility of problem instances for Problem Sets 1 and 2

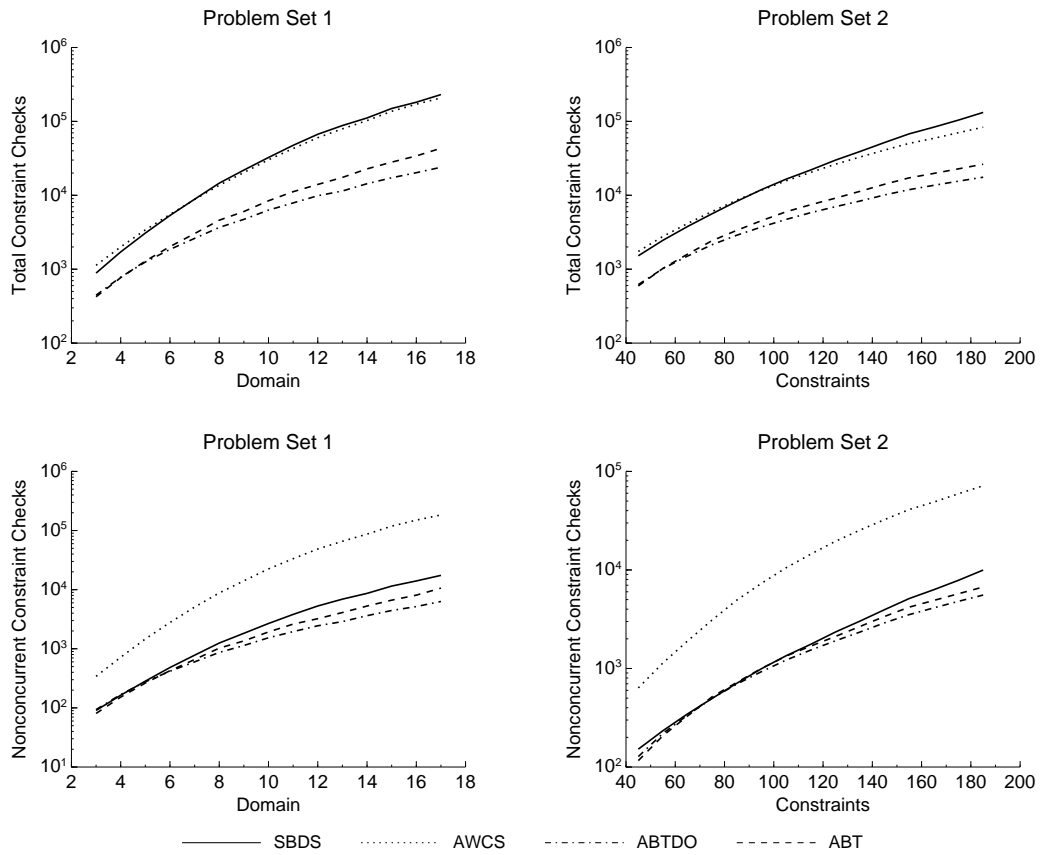


Figure 5.2: Constraint checks for Problem Sets 1 and 2. Full results on pages 141 and 159.

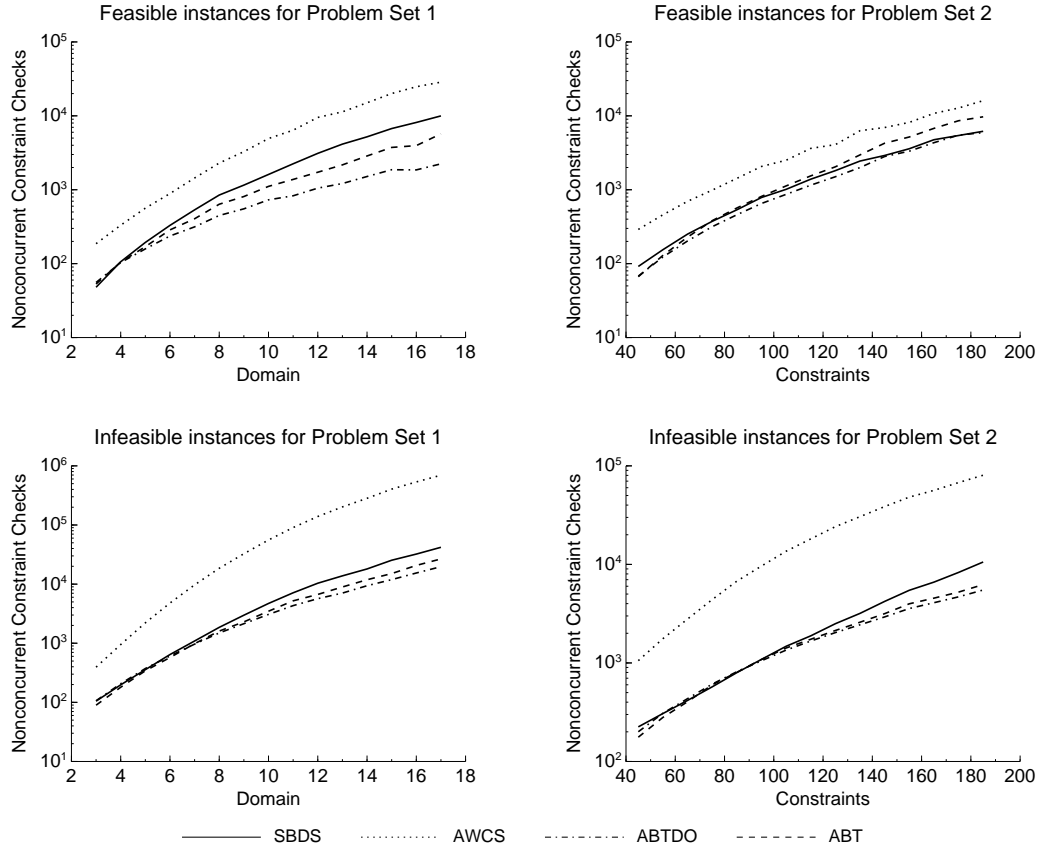


Figure 5.3: Constraint checks for Problem Sets 1 and 2, broken down by feasibility. Full results on pages 142, 143, 160 and 161.

graphs in Figure 5.2 show that ABT-DO produces lower numbers of constraint checks than the other algorithms we reviewed. However, it is more important that the gap between ABT-DO and other algorithms is slowly increasing, indicating better scaling behaviour.

We have also intentionally presented ‘total’ and ‘non-concurrent’ constraint checks together. For the majority of problem instances, SBDS and AWCS performed very similarly in terms of total constraint checks. However, SBDS performed far fewer non-concurrent checks, indicating that SBDS takes greater advantage of concurrency than AWCS. As a result, SBDS is almost competitive with ABT and ABT-DO in terms of non-concurrent constraint checks.

Observation 2 *There is significant variation in the relative performance of AWCS when differentiating between feasible and infeasible instances.*

Figure 5.3 shows that AWCS performs relatively worse on infeasible instances when compared to any other algorithm. The majority of algorithms maintain their relative positions when comparing feasible and infeasible positions.

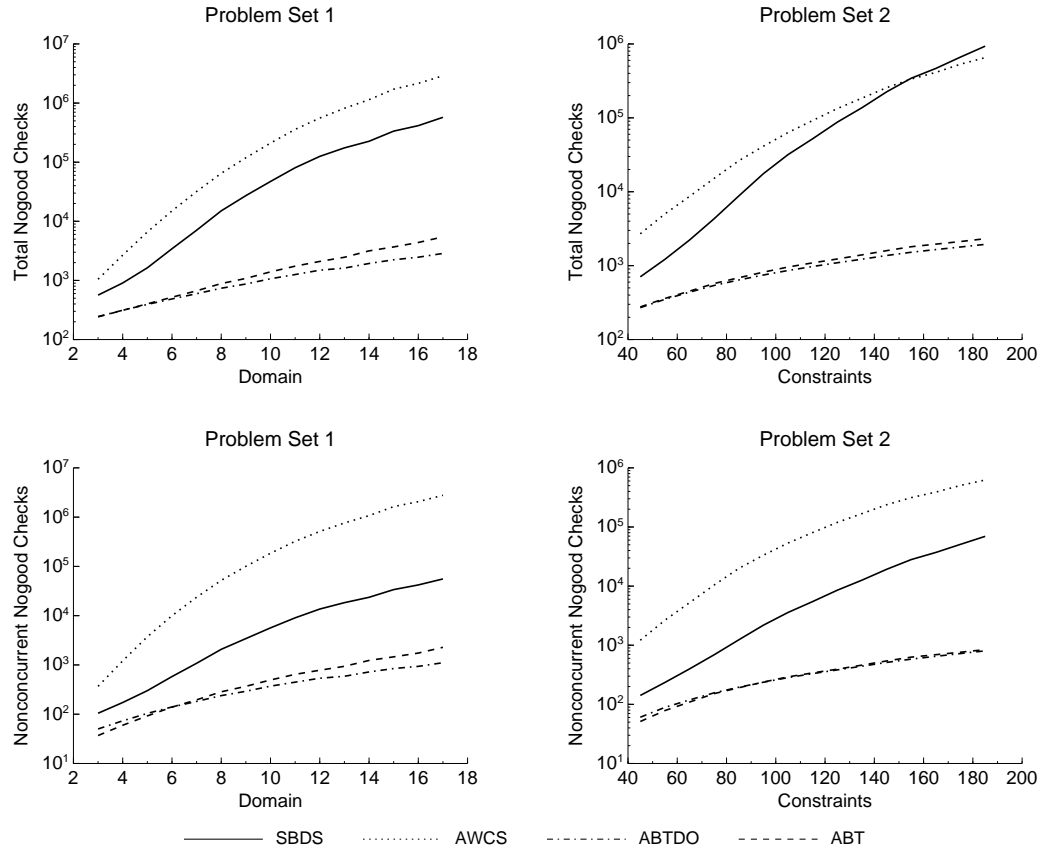


Figure 5.4: Nogood checks for Problem Sets 1 and 2. Full results on pages 144 and 162.

Observation 3 *SBDS appears to have a lower reliance on nogood checks when compared directly to AWCS. However, there are mixed results when considering scaling behaviour.*

Figure 5.4 illustrates the differing emphasis placed by each algorithm on the use of nogood checks. It appears that both SBDS and AWCS have relatively large number of nogood checks when compare with ABT-DO and ABT. This should be expected as both SBDS and AWCS utilise an unbounded nogood store, while ABT-DO and ABT maintain a ‘current’ set. In most cases, SBDS performs far fewer non-concurrent nogood checks than AWCS.

Comparing only SBDS and AWCS, we can see that SBDS shows slightly poorer scaling behaviour as we increase constraint density. However, SBDS clearly scales better as we increase domain size, possibly assisted by the use of other conflict resolution techniques. For example, increasing isgood sizes and the heavy use of the min-conflict heuristic may allow SBDS to avoid the need for nogoods.

Observation 4 *As we scale up the domain size or constraint density, AWCS and SBDS will probably produce more network traffic than ABT or ABT-DO. However, ABT-DO used the*

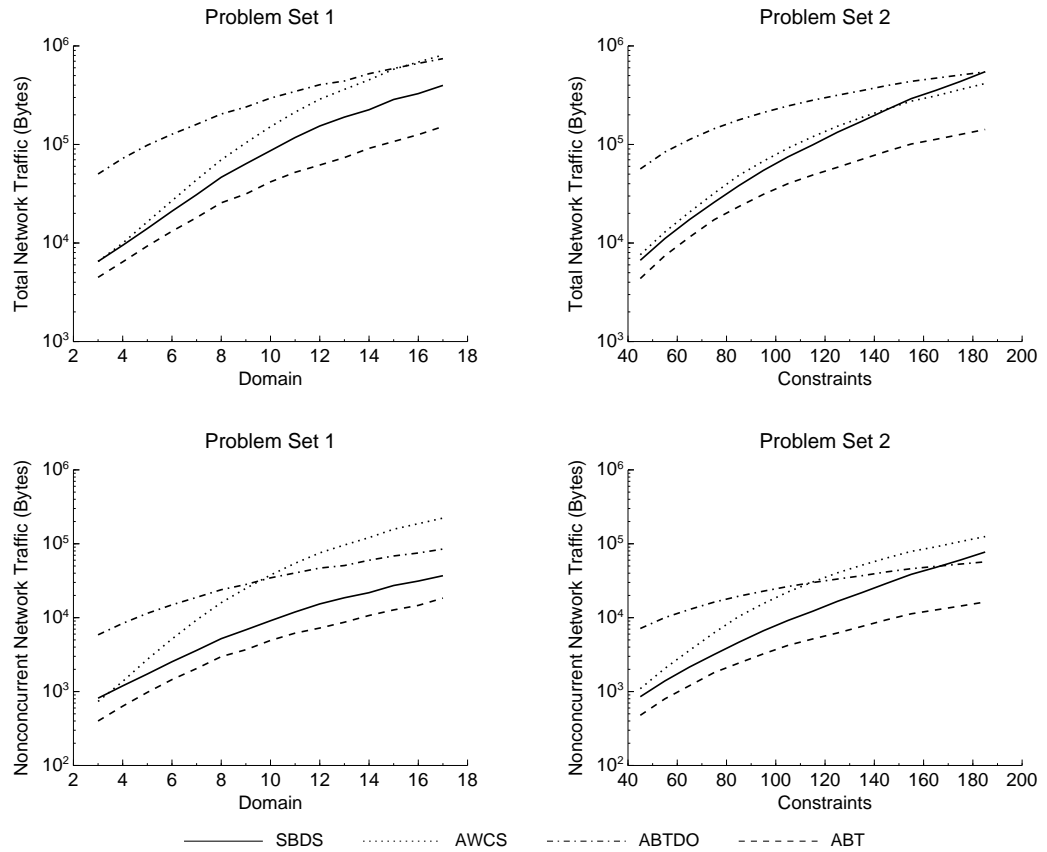


Figure 5.5: Network traffic for Problem Sets 1 and 2. Full results on pages 147 and 165.

most traffic amongst the instances we tested, particularly on problems with small domains or low density.

Figure 5.5 illustrates very different algorithm behaviours in terms of total network traffic. For example, we can see that ABT-DO generated the most network traffic on almost all of the problem instances we considered. However, AWCS and SBDS demonstrated poorer scaling behaviours, suggesting that they will likely exceed ABT-DO when given larger domains or high constraint density. Overall, ABT was the most efficient in terms of network traffic, presumably aided by its small packet sizes.

For a more complete picture of network usage patterns, it is useful to also look at ‘non-concurrent network traffic’. Non-concurrent network traffic is a better measure of performance if we presume that (a) the communication network backbone is extremely fast; but (b) the network interface of each agent is relatively slow. That is, if we assume that the greatest communication bottleneck is the individual network interfaces of the agents, and not the network backbone itself. An algorithm which sends all messages via a central ‘mediator’ agent would likely have very high non-concurrent network traffic. An algorithm which

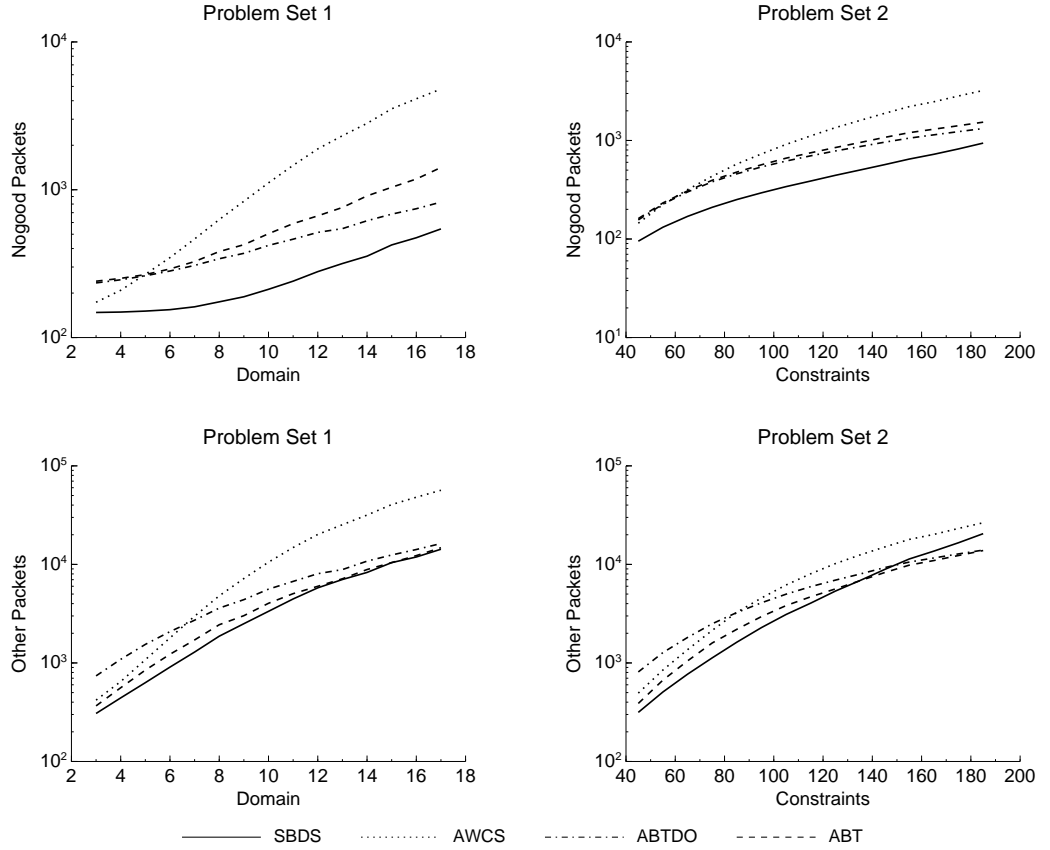


Figure 5.6: Numbers of packets for Problem Sets 1 and 2. Full results on pages 150 and 168.

uses fairly independent communication between agents would have lower non-concurrent network traffic.

In this case, there is little difference in the scaling results; ABT-DO and ABT both scale better than AWCS or SBDS regardless of whether we use ‘total’ or ‘non-concurrent’ measures. However, by comparing the ‘total’ and ‘non-concurrent’ measures we can see AWCS performs more of its communication in a sequential fashion than other algorithms. This matches our previous statements on AWCS and the sequential nature of its constraint checks.

Observation 5 *SBDS produced the least nogood packets for all domain sizes and constraint densities that we considered, but ABT-DO may have better scaling behaviour. ABT and ABT-DO have better scaling behaviour when considering other packet types.*

Figure 5.6 illustrates the number of packets sent by each algorithm. It is clear that AWCS scales poorly as the domain size is increased, sending almost exponentially more nogoods. It also appears that SBDS sent the least nogoods of all algorithms, underlining our previous observation that SBDS does not rely on nogood checks for conflict resolutions.

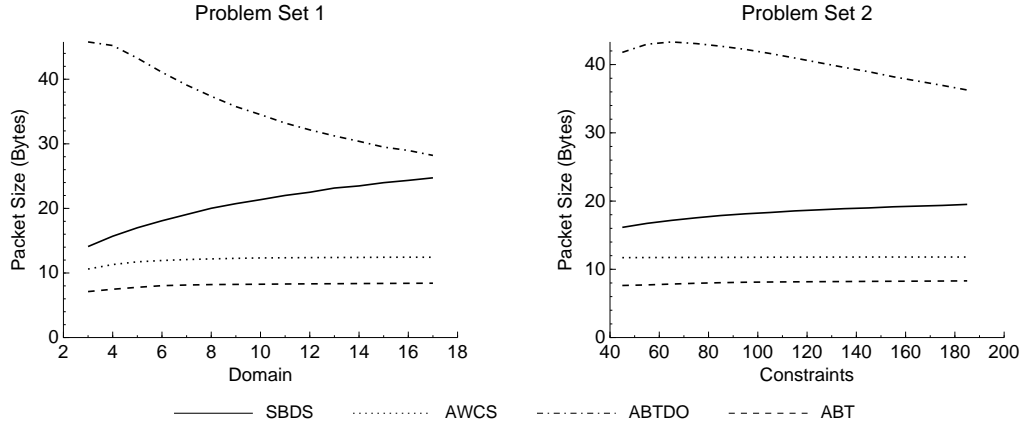


Figure 5.7: Packet sizes for Problem Sets 1 and 2. Full results on pages 153 and 171.

However, the scaling behaviour of ABT-DO suggests that it may actually send less nogoods than SBDS when considering a problem with a very large domain size. Also, the majority of packets are not nogoods, but relate to other parts of the algorithm (e.g. value or ordering messages). As a result, ABT and ABT-DO send less packets in total than either of AWCS or SBDS for most of the problem instances that we considered.

Observation 6 *ABT-DO has larger packet sizes, presumably due to the size of ‘ordering’ messages.*

Figure 5.7 highlights the rather large average packet sizes used by ABT-DO. The packet sizes of ABT-DO are unsurprising when we consider that an ‘ordering’ message includes information on every variable in the constraint network. These unusually large messages would also explain why ABT-DO generates so much network traffic on otherwise simple problems. The slow decrease in packet size for ABT-DO may be explained a slow decline in the relative frequency of ‘ordering’ packets on larger problem instances.

The slow increase in packet sizes for SBDS is most likely explained by the increasing length of isgoods. The more effort required to solve a problem, the more isgoods which are exchanged, and so the average length of an isgood must necessarily increase. For extremely difficult problems with small numbers of variables, it is possible that SBDS will exceed the average packet size of ABT-DO.

Observation 7 *As expected, most algorithms show a relatively stable ‘concurrent checks’ as we scale the number of values. However, there is a decrease in ‘concurrent checks’ as we increase the number of connections between variables.*

Figure 5.8 illustrates the new ‘concurrent checks’ measure described earlier in this chapter. For problems with an increasing domain size, we would expect and do observe very little

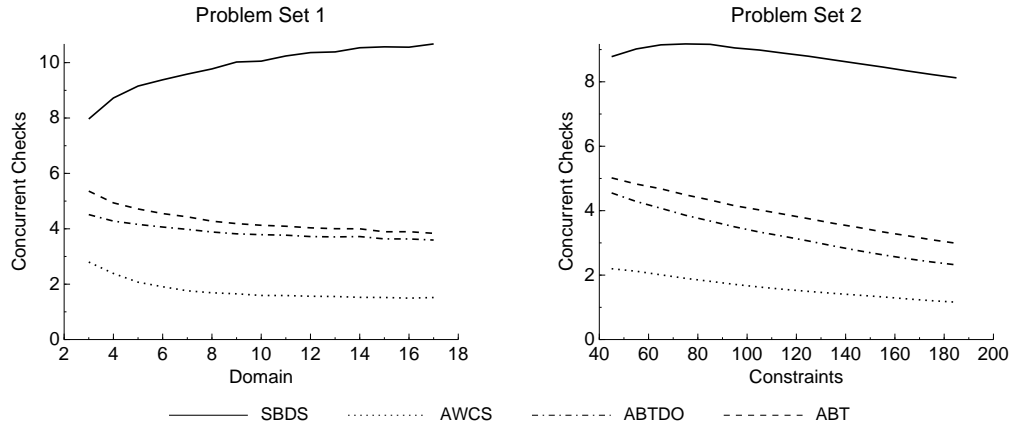


Figure 5.8: Concurrent checks for Problem Sets 1 and 2. Full results on page 192.

change in concurrency. However, an increase constraint density produces more connections between variables, and so we see a slow decline in concurrency.

While this measure is not appropriate for comparing performance, it is still interesting to note that ABT has greater ‘concurrency’ than ABT-DO. This may be explained by the fact that ABT-DO sends ‘ordering’ messages to all lower neighbours, effectively synchronising their non-concurrent counts. However, it is equally possible that dynamic-ordering reduces thrashing, and so increases overall efficiency.

SBDS has significantly higher concurrency, but this is most likely an indication of massive inefficiency rather than good performance. Many agents in SBDS hold identical nogoods and isgoods, and so multiple agents may perform identical constraint and nogood checks simultaneously. Algorithms such as AWCS, ABT and ABT-DO use a total ordering and so can mostly ensure that agents do not perform duplicate work.

Observation 8 *ABT-DO clearly dominates in CPU time, followed by ABT. SBDS performs somewhere between AWCS and ABT depending on problem parameters.*

Figure 5.9 shows our implementation of ABT-DO to be more efficient than SBDS and AWCS on problems with larger domains or higher density. SBDS is more competitive when increasing domain size than constraint density, but ABT-DO still shows better overall scaling behaviour. AWCS is a particularly poor performer when increasing domain size, presumably due to the increasing workload of maintaining many nogoods.

This simply confirms the majority of the above observations - SBDS and AWCS do not scale as well as ABT-DO to problems with large domains or high density.

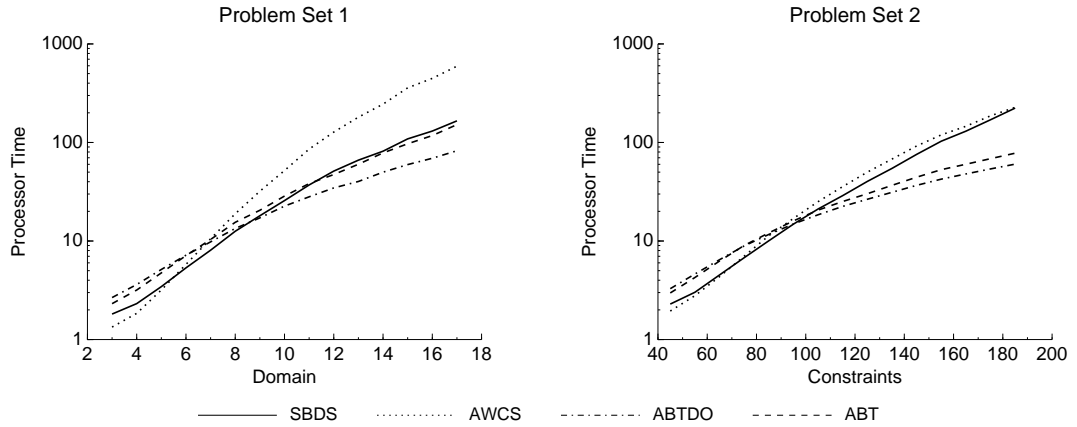


Figure 5.9: CPU time for Problem Sets 1 and 2. Full results on pages 153 and 171.

5.5 Results for Larger Problems

The third problem set demonstrates algorithm behaviour on progressively larger problems, but with a low ‘degree’ and small domain size. As the ratio of constraints to variables remains constant, the average feasibility remains stable (see Figure 5.10). This problem set is quite different to the previous two, and is perhaps most representative of the problems described in the motivation of this thesis.

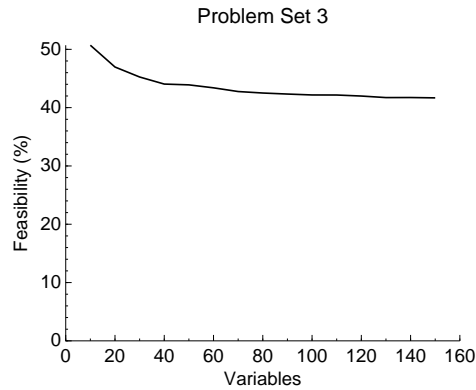


Figure 5.10: Average feasibility of problem instances for Problem Set 3

Observation 9 *Most algorithms scale similarly in terms of non-concurrent constraint checks. However, AWCS performs significantly more non-concurrent constraint checks than other algorithms.*

Figure 5.11 is a stark departure from the results of the previous two problem sets. As we increase the number of variables, we see similar scaling behaviour between ABT-DO and

SBDS. The number of non-concurrent constraint checks performed by SBDS and ABT-DO is nearly identical. The results of ABT appear to have been affected by a number of outlier runs, presumably due to some kind of thrashing.

AWCS again appears to operate in a sequential manner, with significantly more non-concurrent checks than any other algorithm. Surprisingly, AWCS still appears to scale similarly to both SBDS and ABT-DO.

Observation 10 *AWCS scales better in terms of nogood checks than SBDS. However, both scale similarly in terms of non-concurrent nogoods as SBDS demonstrates greater concurrency.*

Figure 5.12 illustrates algorithm scaling behaviour in terms of the number of nogood checks. It appears that SBDS will perform fewer nogood checks than AWCS for smaller problems, but exceeds AWCS for larger problems. Similarly to previous results, SBDS demonstrates greater concurrency by having a low number of non-concurrent nogood checks.

Also, the number of nogood checks performed by ABT-DO is very small due to its maintenance of ‘current nogoods’. However, ABT shows a sudden rise in the number of total and non-concurrent nogood checks, perhaps caused by some kind of ‘thrashing’.

Observation 11 *Beyond a certain problem size, most algorithms scale similarly if we only consider nogood packets. However, ABT and ABT-DO have poorer scaling behaviour in terms of other packets. SBDS produced significantly less packets than ABT or ABT-DO for all problem sizes.*

Figure 5.13 shows algorithm scaling behaviours in terms of network packets. It appears that all algorithms have similar scaling behaviours in terms of the number of nogood packets, but only once we exceed a particular problem size. ABT-DO shows worse scaling for other packets, perhaps caused by the broadcasting of ‘ordering’ messages to an increasing number of recipients. Comparing the results from all algorithms, SBDS sends significantly fewer packets of any type.

While not clearly shown in these graphs, it appears that ABT-DO, AWCS and SBDS all exhibit a linear increase in the number of nogood messages. ABT demonstrates linear scaling on small instances but becomes non-linear on larger instances, perhaps caused by ‘thrashing’ on some instances.

Observation 12 *ABT-DO has linearly-increasing packet sizes, presumably due to the use of ‘ordering’ messages. Other algorithms maintain a consistent packet size when increasing the network size.*

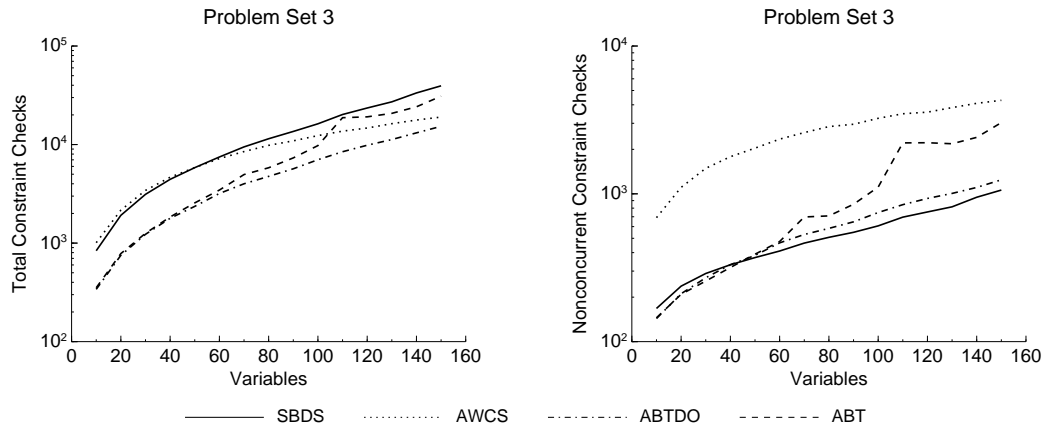


Figure 5.11: Constraint checks for Problem Set 3. Full results on page 177.

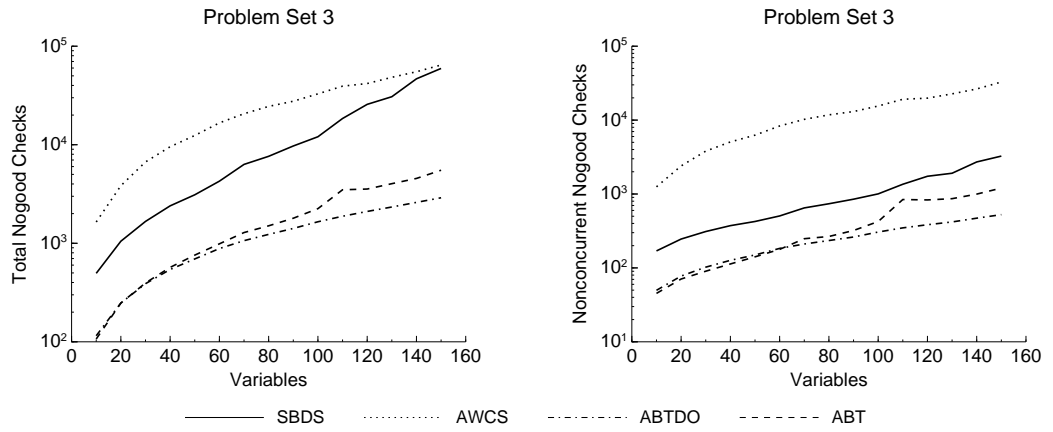


Figure 5.12: Nogood checks for Problem Set 3. Full results on page 180.

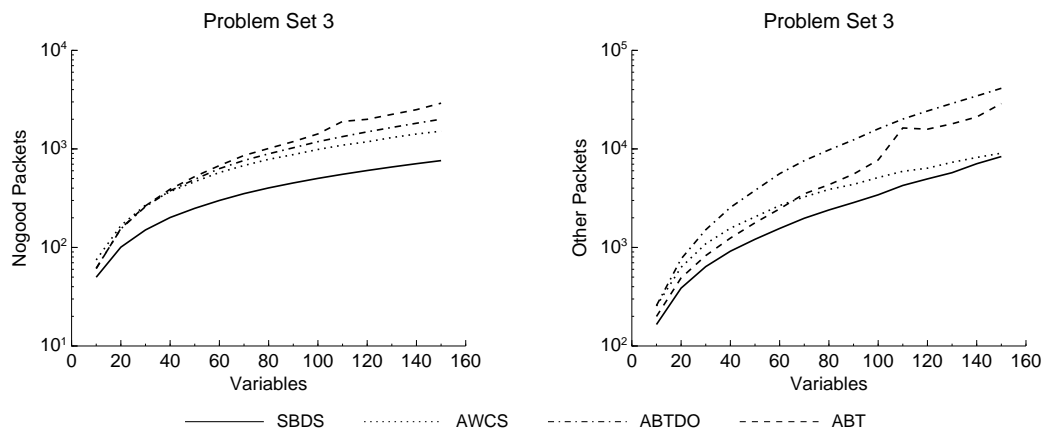


Figure 5.13: Number of packets for Problem Set 3. Full results on page 186.

Figure 5.14 illustrates a unique issue of ABT-DO. ABT-DO uses ‘ordering’ messages which are linear in the number of agents in the network, but no other algorithm reviewed for this thesis uses such a data structure. As a result, ABT-DO is the only algorithm to demonstrate a significantly increasing packet size as the number of variables is increased.

This would appear to be a major drawback when applying ABT-DO (in the form presented in this thesis) to ‘very large’ constraint networks with many variables. ABT-DO requires that each agent must know every other agent’s position in a total order. Avoiding this kind of ‘global’ structure was part of the motivation for this thesis.

While not shown clearly, SBDS also demonstrates a slowly increasing packet size. Presumably this is caused by an increase in the length of isgood and nogoods. However, this growth is very small, and barely noticeable on the graphs above.

Observation 13 *ABT-DO scales poorly in terms of network traffic when compared to any other algorithm. This applies for both feasible and infeasible problem instances.*

Figure 5.15 again emphasises the poor scaling behaviour of ABT-DO on large problem instances. As seen previously, the total number of nogoods generated by ABT-DO scales linearly in the number of variables. Also, the size of an ‘ordering’ message scales linearly due to its inherent structure, and is sent to all lower-ranked agents. As a result, ABT-DO appears to be scaling cubically, whereas other algorithms appear to scale quadratically.

Observation 14 *In a reversal from previous results, ABT and ABT-DO have the worst scaling behaviour in terms of CPU time.*

Figure 5.16 emphasises the poor scaling behaviour of our ABT and ABT-DO implementations on large problem instances. This is somewhat surprising, as ABT and ABT-DO were significantly more efficient for large domains or high constraint densities. In the case of ABT-DO, it may be the case that significant processor time is spent managing the increased network traffic.

Observation 15 *All algorithms show a near-linear increase in ‘concurrency’ of constraint and nogood checks as we scale the number of variables. SBDS clearly performs more operations in parallel.*

For a problem set with an increasing number of variables, we would expect a corresponding increase in ‘concurrent checks’. Figure 5.17 illustrates the new ‘concurrent checks’ measure described earlier in this chapter, broken down by feasibility. As expected, an increase in the number of agents produces an increase in the number of concurrent constraint or nogood checks. This is true regardless of whether we are considering feasible or infeasible instances.

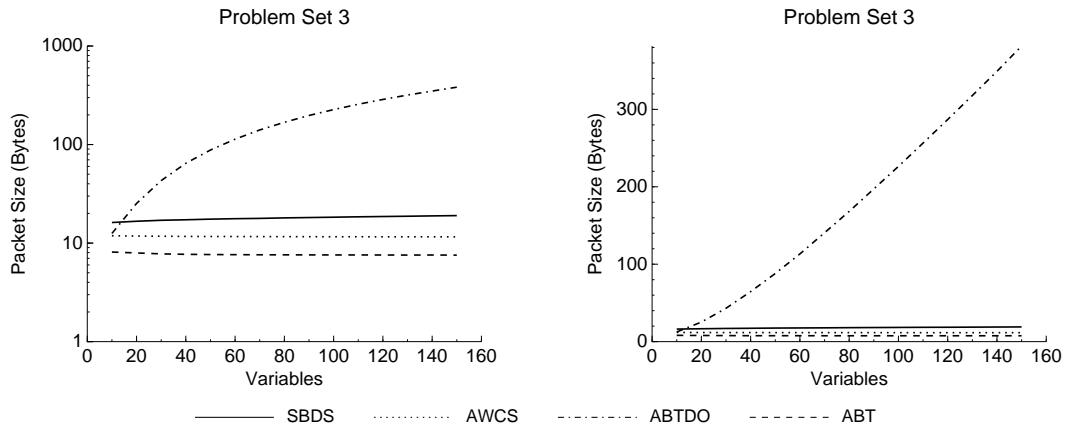


Figure 5.14: Packet sizes for Problem Set 3. Full results on page 189.

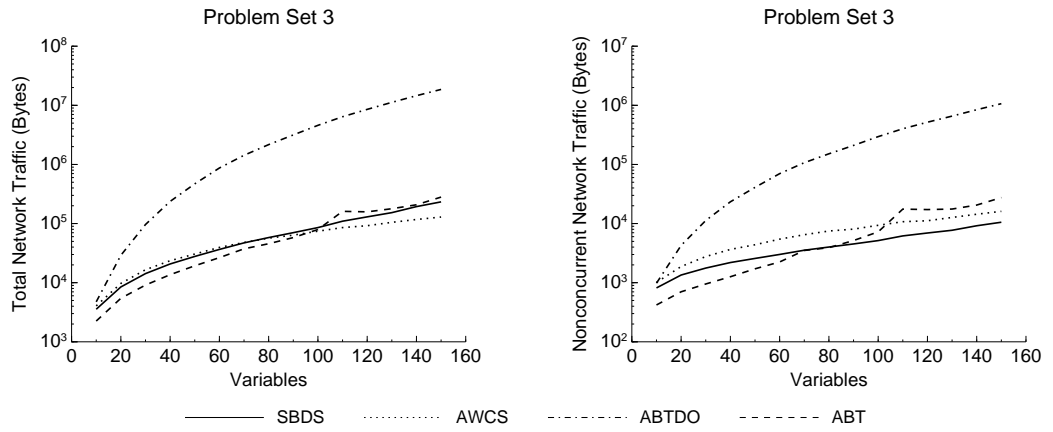


Figure 5.15: Network traffic for Problem Set 3. Full results on page 183.

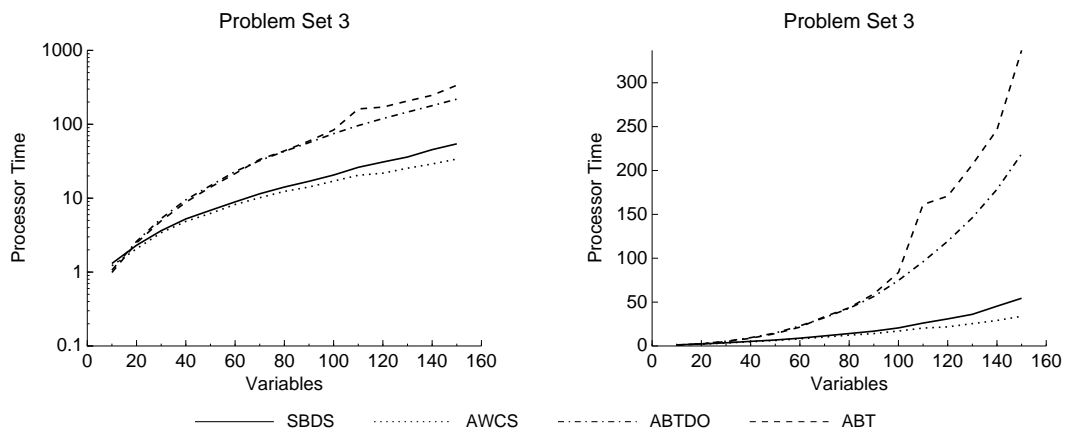


Figure 5.16: CPU time for Problem Set 3. Full results on page 189.

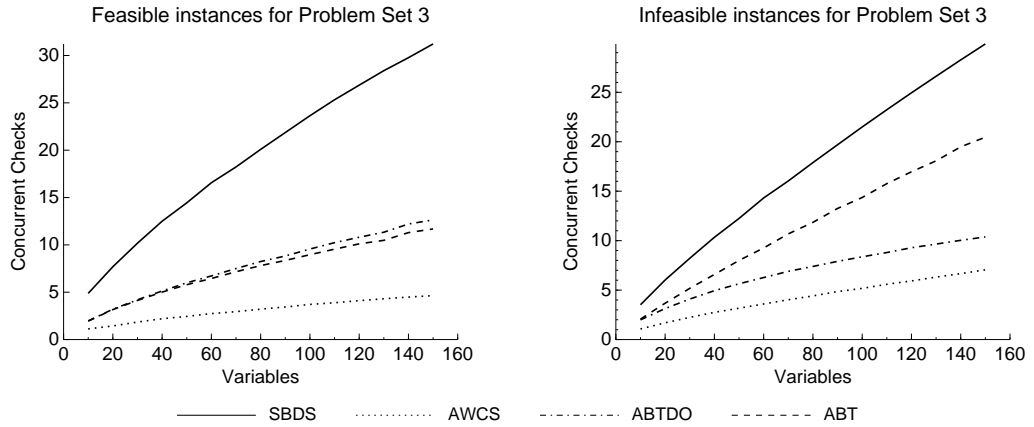


Figure 5.17: Concurrent checks for Problem Set 3, broken down by feasibility. Full results on pages 193 and 194.

The increase in concurrency is perhaps most evident in SBDS. On average, around one fifth of all agents appear to be checking a nogood or constraint ‘concurrently’. This can be compared with AWCS, where around one thirtieth of all agents are operating concurrently. While this is not an indication of performance, it may indicate that AWCS is not using all agents maximally.

There is also a surprising difference between ABT and ABT-DO. For feasible instances there were similar levels of concurrency for ABT and ABT-DO, which was the expected result. However, ABT showed significantly higher levels of concurrency for infeasible instances. It is unclear whether this is because ABT-DO is more efficient than ABT, or whether the use of ordering messages are forcing more synchronisation between agents in ABT-DO.

Observation 16 *ABT-DO shows a surprisingly low ‘network concurrency’ for feasible instances as we increase the number of variables. There is a significant contrast between feasible instances and infeasible instances.*

Similar to the previous results, we would expect an increasing number of variables to produce an increase in ‘network concurrency’. However, Figure 5.18 indicates that ABT-DO does not see a significant increase in network concurrency. This may suggest that fewer agents are responsible for more of the total network traffic.

We can only speculate that this is an inherent property of ABT-DO’s ordering mechanism. Agents higher up the hierarchy are more likely to receive nogood messages, and therefore are more likely to send ordering messages. At the same time, agents which are already high up in the hierarchy are less likely to be moved lower in the hierarchy. As a result, those agents at the top of the hierarchy would become a bottleneck for network traffic, and reduce

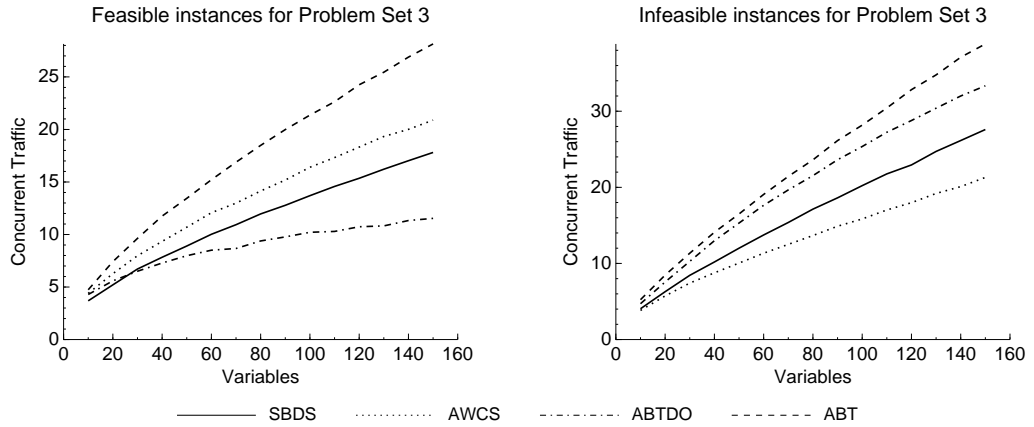


Figure 5.18: Concurrent traffic for Problem Set 3, broken down by feasibility. Full results on pages 193 and 194.

network ‘concurrency’. This bottleneck may not be fully realised on infeasible instances as the algorithm would terminate relatively quickly.

5.6 Discussion

In problem sets 1 and 2, we considered small problems with progressively increasing domain size or constraint density. Almost all results showed ABT-DO as either the most efficient amongst the algorithms reviewed, or scaling better than other algorithms. Based on these experiments alone, we would have concluded that ABT-DO is the most effective DisCSP algorithms of those we compared.

However, the motivation of this thesis is for DisCSP instances with ‘large’ (potentially unbounded) numbers of variables. Problem set 3 demonstrated that the network and computational requirements of ABT-DO scale poorly in the number of variables, despite low domain size and constraint density. This result can perhaps be ascribed to underlying differences between the algorithms.

5.6.1 Coordinated Approach

In ABT-DO, every agent is aware of its position in a global hierarchy at all times. As a result, all agents can contribute to the solving process in a very efficient and ‘coordinated’ manner. The term ‘coordinated’ is used here very loosely, but it seems the best word to describe the behaviour of ABT-DO.

For example, a constraint would never be checked by two agents simultaneously in ABT-DO, as the role of ‘constraint checker’ is determined by the global ordering. In a similar

fashion, it would be very rare for two agents to derive the same nogood, as each agent is checking constraints for a different set of variable assignments. ABT-DO would be unable to maintain a bounded set of ‘current’ nogoods without this ‘coordination’ between agents.

Unfortunately, the global ‘ordering’ messages used to maintain this ‘coordination’ does not seem appropriate for a distributed algorithm. The network requirements scale significantly as the number of nodes in the network is increased. At the very least, a replacement ‘coordination’ mechanism would be necessary if ABT-DO were to handle larger numbers of variables. Ultimately, ABT-DO appears to spend more time managing the ordering than in solving the problem.

5.6.2 Non-Coordinated Approach

In contrast, SBDS can be said to represent a ‘non-coordinated’ approach. The notion of ‘arguments’ provides some semantics for each agents behaviour, without referring to some form of global hierarchy or context. As a direct result, SBDS appears to scale very well as we increase the number of variables.

AWCS can also be said to have a ‘non-coordinated’ approach, as no agent is aware of its position in a global context. AWCS does not need to manage global information and demonstrates reasonable scaling behaviour as we increase the number of variables. However, experiments have repeatedly shown that AWCS performs many of its operations in a ‘sequential’ manner, limiting its ability to take advantage of concurrency. AWCS therefore produces a higher number of ‘non-concurrent’ operations which, in a distributed setting, translates to relatively poor performance.

Unfortunately, the approaches used by AWCS and SBDS have many significant drawbacks. Each algorithm must have an unbounded nogood store to prevent the possibility of cyclic behaviour. Also, the lack of coordination tends to waste computational resources, as the same constraint checks are performed by multiple agents simultaneously.

SBDS has somewhat avoided the issue of an unbounded nogood store, using ‘arguments’ where possible to resolve any conflicts. This claim is supported by the fact that SBDS sends significantly less nogoods than any other algorithm on all problem sizes we tested. That said, ABT and ABT-DO performed far fewer nogood checks, which may be a more important measure depending on the situation.

In any case, both AWCS and SBDS are very wasteful in terms of computational resources. Both presented consistently high numbers of constraint checks and nogood checks. We cannot avoid the fact that SBDS and AWCS place a greater load on the agents in total than either ABT or ABT-DO. This was only mitigated by the fact that SBDS performs many of its constraint and nogood checks in parallel.

5.6.3 Unified Approach

From these experiments and analysis, we hope to derive some future directions for algorithm development.

It is clear that the unbounded nogood stores of AWCS and SBDS are a significant deficiency. Many nogoods are held and checked indefinitely, even if they are no longer necessary. The number of nogood checks performed by both algorithms is significantly higher than ABT or ABT-DO, with few exceptions. A single nogood check is presumably more expensive than a constraint check, as demonstrated in our CPU time comparisons.

While it may seem absurd, the waste of computational resources is perhaps not a great concern. In problem sets 1 and 2, we saw that SBDS performed the most constraint checks, but almost equalled ABT and ABT-DO in terms of non-concurrent constraint checks. Whether an algorithm must be efficient in both total and non-concurrent terms may depend on the context.

Finally, it appears that ‘coordination’ amongst agents provides significant gains for problem instances with high constraint density or large domains. On such problems, ABT-DO would be superior to either AWCS or SBDS. However, we have also demonstrated that explicit methods of ‘coordination’ do not scale well as we increase the number of agents. SBDS would be a better choice of algorithm for large, sparse problem instances.

It would be worthwhile investigating whether ABT-DO can be modified to provide coordination without the overhead of explicit ‘ordering’ messages. Alternatively, it may be possible that agents of SBDS can avoid or delay some computational work based on the contents of their received ‘arguments’.

5.7 Summary

We have compared four DisCSP algorithms (AWCS, SBDS, ABT and ABT-DO) in terms of scaling behaviour. We have also performed some limited comparison of raw performance numbers such as constraint checks, CPU time, etc. Our main conclusions are:

1. In the form presented in this thesis, ABT-DO is inappropriate for problems with large numbers of variables. The use of ordering messages, first commented upon in Section 2.7, generates a significant amount of network traffic. While an explicit total ordering provides significant benefits for small, hard problem instances, it is not appropriate to use such a data structure with a larger number of variables. DisCSP algorithms with less demands on ‘coordinated’ behaviour (for example, AWCS and SBDS) are much more appropriate for such problem instances.
2. Despite the use of an unbounded nogood store, SBDS is not highly dependant on the use of nogoods. It has been shown to be relatively conservative in the creation of nogoods with consistently fewer nogood packets than any other algorithm. This result can be ascribed to the use of isgoods to resolve inconsistencies rather than nogoods. Unfortunately, having an unbounded nogood store means that the low number of nogoods does not translate to a low number of nogood checks.
3. SBDS appears to perform significant amounts of redundant work. While ‘concurrency’ would normally be interpreted as a positive feature in a distributed algorithm, in this case it is most likely an indication of wasted computation. However, without the use of a total order, it may be difficult to avoid redundant work in future versions of SBDS.
4. SBDS has been shown to be competitive with all of the other algorithms we considered. However, the performance of each algorithm is highly dependant on problem parameter values, and what metric is considered most important.

Chapter 6

Multiple Variables Per Agent

6.1 Introduction

The majority of this thesis has focused on the development of the distributed constraint satisfaction algorithm SBDS. This algorithm is, effectively, a protocol for exchanging information between agents. Until this point, we have not considered the possibility of having multiple variables per agent and the resulting constraint satisfaction process *within* the agent.

In this chapter we will describe a method for solving any local CSPs within an agent, allowing us to present the combined set of variables as a single variable. We will begin with a review of definitions for (non-distributed) constraint satisfaction problems, and then describe the ‘hypertree decomposition’ method for solving CSPs. At the end of this chapter we will identify why this method is perhaps more useful than classical algorithms for constraint satisfaction.

First, we require some additional definitions. We know that for each constraint in \mathcal{C} we define its **scope** to be the set of variables which it constrains. If a constraint’s scope consists of only two variables it is called **binary**. The **primal graph** of a CSP is defined as a graph where the nodes correspond to the variables and an edge exists between two nodes iff there exists a constraint between the corresponding variables. The **hypergraph** of a CSP is defined as a tuple $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ with the set of nodes \mathcal{V} corresponds to the set of variables and the set of hyperedges \mathcal{C} corresponds to the set of constraint scopes.

In general, constraint satisfaction problems (CSPs) are NP-complete and thus intractable, but certain classes of CSPs have been found to be tractable due to their structure (represented by their primal or hyper graph). Methods of identifying and solving these classes of tractable CSPs are obviously of importance to the constraints (and by extension, the artificial intelligence) community.

The simplest example of a class of tractable CSPs are those which are acyclic (it is known that binary acyclic CSPs can be solved in linear time [Dec92, Fre82]). A binary CSP is said to be acyclic if there are no loops in its corresponding primal graph. A general (non-binary) CSP is acyclic iff the primal graph is chordal (cycles longer than 3 arcs have chords) and the maximal cliques of the primal graph are the edges of the hypergraph. Given these definitions of acyclicity, we are also able to recognise a CSP as being acyclic in linear time.

Another example of a scheme for recognising tractable CSPs involves identifying the biconnected components of a CSP [Fre85]. A biconnected component of a CSP is a maximal set of variables which remains connected after the removal of any single variable. If the size of the largest biconnected component is k , then the CSP can be solved in time exponential in k , but polynomial in the size of the problem. Thus, the class of CSPs denoted by having biconnected components no larger than a fixed k are tractable.

Several more general schemes have been developed to recognise other classes of tractable CSPs, based upon varying notions of graph or hypergraph **width**. By [GLS00], we know that a CSP with a width of k for any scheme can be solved in polynomial time, where the value of k is the index of the polynomial (naturally, for any scheme, a CSP with a width of 1 (the minimum width) is acyclic, and so can be solved in linear time). The actual technique to be used to solve the CSP in polynomial time is defined by the scheme itself, usually in the form of a **decomposition** or **transformation**.

For each scheme there exists the decision problem of determining whether a given CSP has k width or less. This decision problem itself can at times be intractable, as was shown in [GLS99b] for query-width. In addition, for each scheme there exists the problem of finding a decomposition for a given CSP (if one exists). A decomposition describes the transformation from a cyclic CSP to an acyclic CSP. For example, the transformation of CSPs using biconnected components involves solving each biconnected component separately (which takes time exponential in the size of the component), and then combining the solutions (polynomial time).

Of the proposed schemes with tractable decision problems, hypertree decompositions have been shown to generate the lowest widths [GLS99b, GLS00]. That is, for a maximum width k , the hypertree decomposition scheme is guaranteed to recognise more CSPs as tractable than any other known scheme with a tractable decision problem. Further, for a given CSP and maximum width k , an optimal hypertree decomposition can be found in polynomial time, where the index of the polynomial is a function of k . This has been proven, with *opt- k -decomp* and *cost- k -decomp* [LMS02] being two examples of algorithms which find an optimal decomposition in polynomial time.

One unfortunate aspect of creating hypertree decompositions using *opt- k -decomp* is the size of the index of the time complexity function. The time complexity of *opt- k -decomp* is

approximately $O(|C|^{2k}|\mathcal{V}|^2)$ for large $|C|$. Although the index of $|\mathcal{V}|$ may be smaller for specific classes of CSPs, the index of $|C|$ is unavoidable in *opt-k-decomp* and *cost-k-decomp*. We will present a new algorithm *red-k-decomp* which has worst-case time complexity identical to *opt-k-decomp* and *cost-k-decomp*, but best-case time complexity of $O(|C|^k|\mathcal{V}| + |C|^2|\mathcal{V}|)$. We argue that many CSPs are in the form required to exhibit near-best-case complexity, and so provides a very efficient method for finding solutions to small CSPs.

6.2 Hypertree Decompositions

The basis of the hypertree decomposition scheme is to join together small collections of constraints until the dual-encoding of the CSP is acyclic. Once the dual-encoding becomes acyclic, we can apply known techniques such as directed arc consistency [Wal95] to solve the CSP in polynomial time. As the solutions to the dual-encoding of a CSP are equivalent to the solutions of the original CSP, this is an acceptable transformation.

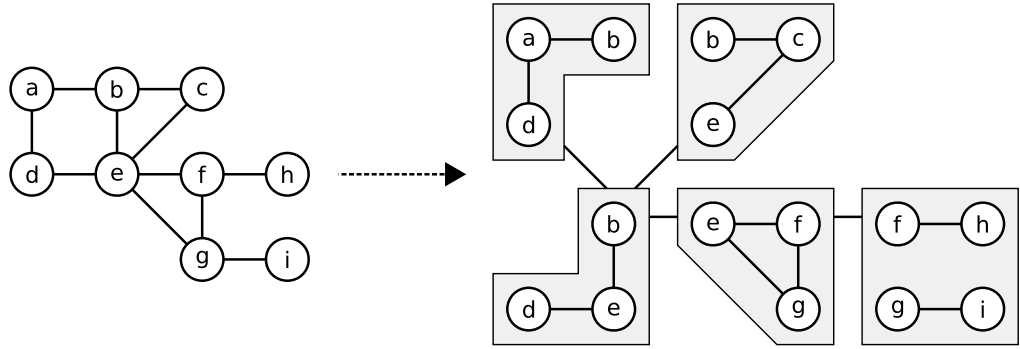


Figure 6.1: A demonstration of how hypertree decomposition forms a new acyclic problem from a cyclic problem.

To define hypertree decompositions (and for the rest of this chapter), we need only represent a CSP by its hypergraph, ignoring the domains of constraints and variables. To easily describe hypertree decompositions, the following notation is defined for the hypergraph $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ with $v \in \mathcal{V}$ a vertex (or variable) and $c \in \mathcal{C}$ an edge (or constraint):

$$\begin{array}{ll} \text{var}(c) = c & \text{variables in } c \\ \text{con}(v) = \{c \in \mathcal{C} : v \in \text{var}(c)\} & \text{constraints **covering** } v \\ \text{adj}(v) = \bigcup_{c \in \text{con}(v)} \text{var}(c) & \text{variables **adjacent** to } v \end{array}$$

Each of the single-element forms of *var*, *con*, and *adj* can be extended to sets by using set union in the usual way. For example, given a set $X \subseteq \mathcal{C}$, $\text{var}(X) = \bigcup_{c \in X} \text{var}(c)$.

Further, we provide parameterised definitions of components used in the definition of hypertree decompositions. Given a set $V \subseteq \mathcal{V}$, and $x, y \in \mathcal{V}$ we say that x and y are $[V]$ -connected if there exists a sequence of adjacent variables (a path), none of which are contained in V , which starts at x and ends at y . A set $X \subseteq \mathcal{V}$ is said to be $[V]$ -connected if every pair $x, y \in X$ are $[V]$ -connected. The set $X \subseteq \mathcal{V}$ is a **[V]-component** if it is $[V]$ -connected and maximal.

It should be noted that the concept of $[V]$ -components will be central to the definitions of hypertree decomposition. By careful selection of a set of variables V we will be able to ‘break’ or ‘decompose’ our CSP into separate $[V]$ -components. This intuition should be kept in mind for the remainder of the definitions.

6.2.1 General Form

A hypertree decomposition consists of a rooted tree with labelling functions assigning constraints and variables to nodes of the tree. Precisely how a hypertree decomposition is used is detailed later. We will provide here the most general definition of hypertree decompositions. For a more complete treatment of hypertree decompositions and their relation to other decomposition techniques see [GLS99b, GLS00].

Definition 12 A *rooted tree* is a pair $T = (N, E)$ where N are the nodes of the tree, and $E \subseteq N \times N$ are the directed edges. T_n indicates the subtree of T with root $n \in N$, $\text{nodes}(T)$ indicates the nodes of T , and $\text{edges}(T)$ indicates the directed edges of T . A node n is the parent of m if $\langle n, m \rangle \in \text{edges}(T)$.

Definition 13 A *hypertree decomposition* of a hypergraph $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ consists of a triple $\langle T, \chi, \lambda \rangle$, where $T = (N, E)$ is a rooted tree and χ and λ are functions defined over N . The functions χ and λ map each node n to a subset of variables $\chi(n) \subseteq \mathcal{V}$ and a subset of constraints $\lambda(n) \subseteq \mathcal{C}$, subject to certain restrictions:

1. Each constraint must be covered by at least one node of the hypertree.
2. Each variable must induce a connected subtree of the hypertree.
3. For each node, $\chi(n)$ is covered by the combined scopes of the constraints in $\lambda(n)$.
4. For each node, $\chi(n)$ must include any variable which is both covered by the constraints $\lambda(n)$, and present in any child node.

These statements can be expressed formally as:

1. $\forall c \in \mathcal{C}, \exists n \in N$ such that $\text{var}(c) \subseteq \chi(n)$
2. $\forall v \in \mathcal{V}, (\{n \in T : v \in \chi(n)\}, E)$ is a connected subtree

3. $\forall n \in N, \chi(n) \subseteq \text{var}(\lambda(n))$
4. $\forall n \in N, \text{var}(\lambda(n)) \cap \chi(T_n) \subseteq \chi(n)$, where $\chi(T_n) = \bigcup_{m \in \text{nodes}(T_n)} \chi(m)$.

The **width** of a hypertree decomposition $\langle T, \chi, \lambda \rangle$ is defined as the maximum of $|\lambda(n)|$ for all $n \in \text{nodes}(T)$. The **hypertree-width** of a CSP is the minimum width for all hypertree decompositions of that CSP. If the width of a hypertree decomposition is equal to the hypertree-width of the CSP, we say the decomposition is **optimal**.

Definition 14 A *complete hypertree decomposition* of $\mathcal{H} = \langle \mathcal{V}, \mathcal{C} \rangle$ is any hypertree decomposition $\langle T, \chi, \lambda \rangle$ where for every constraint $c \in \mathcal{C}$ there exists a node n of the hypertree such that $\text{var}(c) \subseteq \chi(n)$ and $c \in \lambda(n)$.

A complete hypertree decomposition $\langle T, \chi, \lambda \rangle$ describes the transformation from the original cyclic CSP to a new acyclic CSP. Each node n of the hypertree represents a single variable in the new CSP, with its domain defined as the solutions to the subproblem $\langle \lambda(n), \chi(n), \mathcal{D} \rangle$. In other words, the labels $\chi(n)$ and $\lambda(n)$ of each hypertree node n are used as follows:

1. The set of constraints $\lambda(n)$ from the original CSP are solved as a subproblem.
2. The set of solutions of the subproblem are projected over the variables $\chi(n)$.
3. The set of projected solutions are used as the domain for a variable in the new acyclic CSP.

An arc between nodes of the hypertree represents a constraint between their corresponding variables in the new CSP. The form of the constraint is the same as that used in the dual-encoding of a CSP. The new CSP is obviously acyclic as its primal graph corresponds to the tree T .

6.2.2 Normal Form

The definitions of hypertree decompositions given so far are very general. In [LMS02] a more restrictive definition is given which simplifies the search process for a hypertree decomposition, but does not prevent us from finding an optimal hypertree decomposition.

Definition 15 A hypertree decomposition is in **normal form** if, for any pair of nodes $\langle n, m \rangle \in \text{edges}(T)$, there is exactly one $[\chi(n)]$ -component C satisfying all of:

1. $\chi(T_m) = C \cup (\chi(n) \cap \chi(m))$
2. $\chi(m) \cap C \neq \emptyset$
3. $\forall c \in \lambda(m), \text{var}(c) \cap \text{adj}(C) \neq \emptyset$

$$4. \chi(m) = \text{adj}(C) \cap \text{var}(\lambda(m))$$

It should be noted that it is trivial to convert any hypertree decomposition to a complete hypertree decomposition without increasing its width. Given the definition of normal form for hypertree decompositions we can derive χ from λ , which makes the task of generating a hypertree decomposition simpler. This particular result is used advantageously in $\text{opt-}k\text{-decomp}$.

6.2.3 Reduced Normal Form

The original definition of hypertree decompositions allowed for very poor decompositions. Using normal form it was ensured that certain very poor decompositions were never considered, which led to the development of the $\text{opt-}k\text{-decomp}$ algorithm. We now introduce further restrictions on the form of hypertree decompositions, removing even more possible decompositions. The aim of these restrictions is to allow certain assumptions to be made early in $\text{opt-}k\text{-decomp}$, reducing the size of data structures and thus time complexity.

Definition 16 A hypertree decomposition $\langle T, \chi, \lambda \rangle$ is in **reduced normal form (RNF)** if it is in normal form and for every $n \in \text{nodes}(T)$:

1. The constraints in $\lambda(n)$ do not completely cover the constraints of n 's parent node.
2. Every constraint c in $\lambda(n)$ contains a variable which is covered by no other constraint in $\lambda(n)$, and which is adjacent to a variable not covered by c .
3. If there is only one $[\text{var}(\lambda(n))]$ -component then n is a leaf node.

These statements can be expressed formally as:

1. $\forall \langle m, n \rangle \in \text{edges}(T), \text{var}(\lambda(n)) \not\supseteq \text{var}(\lambda(m))$
2. $\forall c \in \lambda(n), \exists v \in \text{var}(c), \text{con}(v) \cap \lambda(n) = \{c\} \wedge \text{adj}(v) \neq \text{var}(c)$
3. $\forall n \in \text{nodes}(T), \text{if } (\mathcal{V} - \text{var}(\lambda(n))) \text{ is a } [\text{var}(\lambda(n))]\text{-component, then } |\text{nodes}(T_n)| = 1$

The definition of RNF places limitations on possible values for $\chi(n)$ which can be used in the hypertree decomposition, and where they may appear in the tree. The guiding intuition of RNF is that only nodes which split the problem into multiple components may appear in the body and root of the hypertree decomposition. As a result, nodes which do not split the problem into multiple components may appear only in the leaves of the hypertree.

The following three lemmas (and their proofs) indicate how to transform a hypertree decomposition in normal form to a hypertree decomposition in reduced normal form. The associated figures provide a good intuition of reduced normal form by describing why it removes many decompositions from consideration.

Lemma 3 *Let $\langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form. It is possible to transform $\langle T, \chi, \lambda \rangle$ to also satisfy condition 1 of RNF, without increasing the width.*

Proof. Assume there exists a pair $\langle n, m \rangle \in \text{edges}(T)$ such that $\text{var}(\lambda(n)) \subseteq \text{var}(\lambda(m))$. The following transformation will delete the child m and modify the parent n to satisfy condition 1 for n :

Procedure 13.1 *fix-condition-1*

- 1: set $\lambda(n)$ to $\lambda(m)$.
 - 2: set $\text{edges}(T)$ to $\text{edges}(T) + \{\langle n, r \rangle : \langle m, r \rangle \in E\} - \{\langle m, r \rangle \in E\} - \{\langle n, m \rangle\}$
 - 3: set $\text{nodes}(T)$ to $\text{nodes}(T) - \{m\}$
-

After the transformation, we can recompute $\chi(n)$ by the definition of normal form. Note that the width of the decomposition does not increase, as the size of $\lambda(n)$ did not increase beyond that of $\lambda(m)$ (which was already in the decomposition). \square

Lemma 4 *Let $\langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form and satisfying condition 1 of RNF. It is possible to transform $\langle T, \chi, \lambda \rangle$ to also satisfy condition 2 of RNF, without increasing the width.*

Proof. Assume $n \in \text{nodes}(T)$ is a node which does not satisfy condition 2 of RNF. Let $c \in \lambda(n)$ be the constraint with smallest $|\text{var}(c)|$ and no variable $v \in \text{var}(c)$ which satisfies both $\text{con}(v) \cap \lambda(n) = \{c\}$ and $\text{adj}(v) \neq \text{var}(c)$. The following transformation will remove the constraint from n (effectively satisfying condition 2 for n):

Procedure 13.2 *fix-condition-2*

- 1: set $\lambda(n)$ to $\lambda(n) - \{c\}$
 - 2: **if** $\text{var}(c) \not\subseteq \text{var}(\lambda(n))$ **then**
 - 3: create new node $m \in \text{nodes}(T)$
 - 4: set $\text{edges}(T)$ to $\text{edges}(T) + \{\langle n, m \rangle\}$
 - 5: set $\lambda(m)$ to $\{c\}$
 - 6: **end if**
-

If $\text{var}(c) \subseteq \text{var}(\lambda(n))$, then a new node need not be created (it may be added later when completing the hypertree decomposition). Alternatively, if $\text{var}(c) \not\subseteq \text{var}(\lambda(n))$, the newly created node m will satisfy condition 1 of RNF (c is the smallest constraint which could be chosen, and so $\text{var}(c) \not\subseteq \text{var}(\lambda(n))$). Again, after the transformation we let $\chi(n)$ and $\chi(m)$ be computed by the definition of normal form. Note that the width of the decomposition can only have decreased as a result of this transformation.

Also note that the node n may still break condition 2 of RNF, requiring repeated applications of this transformation. As there are a finite number of constraints in $\lambda(n)$, these repetitions will terminate. See Figure 6.2 for an example of this transformation. \square

Lemma 5 *Let $\langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form, satisfying conditions 1 and 2 of RNF, with $|\text{nodes}(T)| \geq 3$. It is possible to transform $\langle T, \chi, \lambda \rangle$ to also satisfy condition 3 of RNF, without increasing the width.*

Proof. For a hypertree decomposition in normal form and satisfying condition 1 of RNF, any node n where only one $[\text{var}(\lambda(n))]$ -component exists must be either a leaf or root node (proof omitted). If n is a leaf node then no transformation of the hypertree is necessary.

If, however, n is the root node, then it must have a single child node m , and there must exist more than one $[\text{var}(\lambda(m))]$ -component. A transformation of the hypertree can then be performed as follows.

Procedure 13.3 *fix-condition-3*

- 1: let m be the (only) child of the root node n .
 - 2: **for all** $R \subseteq \lambda(n)$ where $\text{var}(R) - \text{var}(\lambda(m))$ is connected and R is maximal **do**
 - 3: create new node $r \in \text{nodes}(T)$
 - 4: set $\text{edges}(T)$ to $\text{edges}(T) + \{\langle m, r \rangle\}$
 - 5: set $\lambda(r)$ to R
 - 6: **end for**
 - 7: set $\text{edges}(T)$ to $\text{edges}(T) - \{\langle n, m \rangle\}$
 - 8: set $\text{nodes}(T)$ to $\text{nodes}(T) - \{n\}$
-

After the transformation we let $\chi(m)$ and $\chi(r)$ (for all new nodes r) be computed by the definition of normal form. This transformation ensures any node n with only one $[\text{var}(\lambda(n))]$ -component is a leaf node. See Figure 6.2 for an example of this transformation. \square

From Lemmas 3, 4, and 5 we can show that most CSPs with a hypertree decomposition in normal form have an equivalent or better hypertree decomposition in reduced normal form. The best intuition of CSPs which may not have a reduced normal form hypertree decomposition is those whose optimal hypertree decompositions have only two nodes.

Theorem 13 *If all optimal normal form hypertree decompositions of $\mathcal{H} = \langle \mathcal{V}, C \rangle$ have more than three nodes, then there exists an optimal reduced normal form hypertree decomposition.*

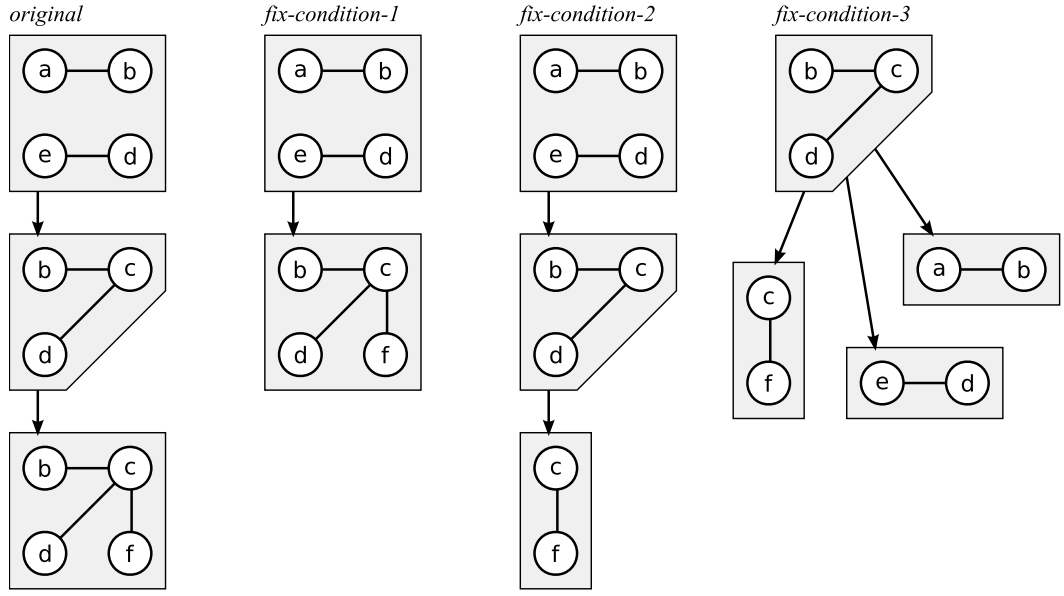


Figure 6.2: Transforming a normal form hypertree decomposition to reduced normal form.

Proof. By Lemmas 3 and 4 we know that for any optimal normal form hypertree decomposition $\langle T, \chi, \lambda \rangle$ we can derive a new optimal normal form hypertree decomposition $\langle T', \chi', \lambda' \rangle$ which obeys conditions 1 and 2 of RNF. By our assumptions, $|nodes(T')| \geq 3$, and so by Lemma 5 we can derive an optimal reduced normal form hypertree decomposition. \square

It is possible to construct CSPs which do not have a hypertree decomposition in reduced normal form. However, by Theorem 13 we know they have very small numbers of nodes in their optimal hypertree decompositions, and so are unlikely to benefit from hypertree decompositions. Finally, we present a theorem which will form an important part of our optimised algorithm for computing hypertree decompositions.

Theorem 14 *If a hypertree decomposition $\langle T, \chi, \lambda \rangle$ is in reduced normal form, and for a node $m \in nodes(T)$ there exists only one $[var(\lambda(m))]$ -component, then $var(\lambda(m))$ is connected.*

Proof. By condition 3 of RNF, m must be a leaf node. Let n be the parent of m , so there exists a single $[var(\lambda(n))]$ -component C satisfying the conditions of normal form for m . By condition 3 of normal form, every constraint in $\lambda(m)$ must intersect C . Additionally, as m is a leaf node, $C \subseteq var(\lambda(m))$. As C is connected, we can conclude that $var(\lambda(m))$ is connected. \square

6.3 Algorithm

6.3.1 opt- k -decomp

The opt- k -decomp algorithm [LMS02] achieves two things: it determines if a hypergraph has a hypertree decomposition with width less than k and, if the answer is yes, finds a hypertree decomposition with the smallest width possible. We can use the definition of reduced normal form to improve opt- k -decomp, but first need an intuitive understanding of how it achieves its task.

The algorithm makes use of the concept of a **k -vertex**. A set of constraints $R \subseteq C$ is called a k -vertex iff $|R| \leq k$, where $k \in \mathbb{N}$. As each k -vertex R is a set of constraints, we know that it covers a set of variables $\text{var}(R)$. Therefore, if chosen correctly, it can be used to split (or decompose) a CSP into independent $[\text{var}(R)]$ -components.

The opt- k -decomp algorithm encapsulates k -vertices and their resulting decompositions within a directed acyclic graph. All nodes of this graph are pairs of the form (R, C) where R is a k -vertex and $C \subseteq \mathcal{V}$, but have different intuitions depending on their placement:

- A **problem node** (R, C) produced by opt- k -decomp effectively describes a subproblem that needs to be decomposed further; R is a k -vertex that was used for an initial decomposition, and C is a remaining $[\text{var}(R)]$ -component. The set C effectively describes the variables of the ‘problem’, while R indicates that ‘this problem was created when some larger set of variables was decomposed by R ’. The special root node (\emptyset, \mathcal{V}) of the graph is also a problem node.
- A **decomposition node** (R, C) represents the fact that R can be used to decompose the set C . For each problem node, the resultant graph should contain at least one child decomposition node to describe how it could be decomposed. Similarly, for each decomposition node, there may exist child problem nodes to represent parts of the CSP which require further decomposition.

The opt- k -decomp algorithm constructs the graph of nodes as a means to represent the relationship between ‘using a k -vertex to decompose a problem’ and the resultant ‘problem components’. Once the graph of nodes is constructed, it is relatively straightforward and efficient to construct a hypertree decomposition. The algorithm fragment for constructing the graph can be briefly described as:

Algorithm 14 Outline of *opt-k-decomp*

```
1: initialise the list of  $k$ -vertices
2: initialise the set of problem nodes, including a node representing the entire CSP
3: for each problem node do
4:   for each  $k$ -vertex do
5:     if the current  $k$ -vertex decomposes the problem node then
6:       create a new decomposition node holding the current  $k$ -vertex
7:       add an arc from the current problem node to the new decomposition node
8:       for each component of the new decomposition node do
9:         find the problem nodes formed from decomposing the current component with
           the current  $k$ -vertex
10:      add an arc from the current decomposition node to the found problem node
11:     end for
12:   end if
13: end for
14: end for
```

It is possible to execute *opt-k-decomp* with a large value of k , and so decompose and solve a wider selection of CSPs. However, it is not practically feasible due to the exponential runtime of *opt-k-decomp* in terms of k . The worst-case complexity of *opt-k-decomp* is $O(|C|^{2k}|\mathcal{V}|^2)$ (for large values of $|C|$). The $|C|^{2k}$ term provides the greatest difficulty for even relatively small values of k (like 3 or 4).

To understand where this worst-case complexity term originates, note that the two outer loops iterate independently over data structures whose length is linear in the number of k -vertices. Also note that the total number of k -vertices is linear with respect to $|C|^k$. Blindly accepting any k -vertex as a candidate for a decomposition node can thus be blamed for the poor worst-case complexity of *opt-k-decomp*.

The best-case complexity of *opt-k-decomp* is not noticeably better than the worst-case complexity. Figure 6.3(a) presents CPU time data for executions of *opt-k-decomp* (with $k = 3$) on random CSPs with 20 variables and increasing numbers of constraints. Figure 6.3(b) presents the execution times divided by the worst-case complexity function for *opt-k-decomp*. The ratio between CPU times and worst-case complexity values converges to a finite value as the number of constraints increases, suggesting that average-case and worst-case complexity are the same.

6.3.2 red- k -decomp

We now present the algorithm red- k -decomp. As it is similar to opt- k -decomp (plus checks related to RNF) we will state, without proof, that the algorithm is correct. That is, if there exists a hypertree decomposition in reduced normal form with width less than or equal to a fixed value k , then it will be found by red- k -decomp and it will be optimal.

As in opt- k -decomp, this algorithm constructs a graph representing possible hypertree decompositions of the hypergraph $\langle \mathcal{V}, C \rangle$. To construct the graph it makes use of two distinct lists of k -vertices, V_b and V_l (see *init-vertices*). The graph itself consists of nodes (separated into problem nodes N_n and decomposition nodes N_d , plus a root node) and directed edges (E , pointing towards the root node).

The nodes retain the same intuition as that described for opt- k -decomp, and so are pairs of the form (R, C) where $R \subset C$ and $C \subseteq \mathcal{V}$. Edges are directed and, as described earlier, the meaning of an edge will differ depending upon the type of node. An edge $\langle (R, C), (R', C) \rangle$, where $(R, C) \in N_n$ or $(R, C) = (\emptyset, \mathcal{V})$, indicates that $(R', C) \in N_d$ is a possible decomposition. An edge $\langle (R', C), (R', C') \rangle$ where $(R', C) \in N_d$ indicates that $(R', C') \in N_n$ is a ‘subproblem’ caused if C is decomposed by R . Edges are not created between pairs of nodes in N_n , or between pairs in N_d .

The algorithm is therefore a simple sequence of procedure calls. The body of the algorithm, and the details of each procedure, are given below:

Algorithm 15 *red- k -decomp*

```

input  $\mathcal{H} = \langle \mathcal{V}, C \rangle$ 
init-vertices
init-graph
build-graph
calc-width( $\emptyset, \mathcal{V}$ )
if width( $\emptyset, \mathcal{V}$ )  $\leq \infty$  then
    extract-decomposition( $\emptyset, \mathcal{V}$ )
    output  $\langle T, \chi, \lambda \rangle$ 
else
    output fail
end if
```

init-vertices: Generates the list of k -vertices to be used in decomposing the problem. Each generated k -vertex R is checked against condition 2 of RNF. This is done efficiently by generating the set of variables which are covered by only one constraint in R and

Procedure 15.1 *init-vertices*

set V_b and V_l to be empty lists
for all $R \subseteq C$ where $0 < |R| \leq k$ **do**
 $t := \{v \in \text{var}(R) : |\text{con}(v) \cap R| = 1\}$
 if $\forall c \in R, \exists v \in \text{var}(c)$ such that $v \in t \wedge \text{adj}(v) \not\subseteq \text{var}(R)$ **then**
 if there are multiple $[R]$ -components **then**
 $V_b := V_b + R$
 else if $\text{var}(R)$ is connected **then**
 $V_l := V_l + R$
 end if
 end if
end for

Procedure 15.2 *init-graph*

set N_d, N_n and E to be empty lists
for all $R \in V_b$ **do**
 $N_d := N_d + (R, \mathcal{V})$
 $E := E + \langle (R, \mathcal{V}), (\emptyset, \mathcal{V}) \rangle$
 for all $C \subseteq \mathcal{V}$ where C is a $[\text{var}(R)]$ -component **do**
 $N_n := N_n + (R, C)$
 $E := E + \langle (R, C), (R, \mathcal{V}) \rangle$
 end for
end for

verifying that each constraint in R contains at least one which is adjacent to a $[\text{var}(R)]$ -component. Each k -vertex R is then placed into V_b (those with more than one $[\text{var}(R)]$ -component) or V_l (all others where $\text{var}(R)$ is connected), or are discarded. Generating the lists of vertices is no worse than $O(|C|^k |\mathcal{V}|)$ time.

init-graph: Initialises a graph structure $\langle N_n \cup N_d \cup \{(\emptyset, \mathcal{V})\}, E \rangle$ which will eventually contain all possible reduced normal hypertree decompositions. The initialisation of the graph involves construction of all nodes in N_n , and construction of required nodes in N_d to link them to the root node. Initialisation of N_n and N_d is straightforward, and takes $O(|V_b| |\mathcal{V}|)$ time. The number of elements of N_n is bounded by $|V_b| |\mathcal{V}|$, with each element taking $|\mathcal{V}|$ space.

build-graph: Makes additions to the graph structure representing possible reduced normal hypertree decompositions. Primarily, this function adds nodes to N_d , connecting them with existing nodes in N_n . Adding nodes to N_d works as follows:

Procedure 15.3 *build-graph*

```
for all  $(R, C) \in N_n$  do
   $e := \bigcup_{v \in C} \text{adj}(v)$ 
   $a := \bigcup_{v \in e} \text{con}(v)$ 
   $r := e \cap \text{var}(R)$ 
  for all  $R' \in V_b \cup V_l$  do
    if  $R' \subseteq a$  and  $\text{var}(R) \not\subseteq \text{var}(R')$  and  $\text{var}(R') \cap C \neq \emptyset$  and  $r \subseteq \text{var}(R')$  then
       $N_d := N_d + (R', C)$ 
       $E := E + \langle (R', C), (R, C) \rangle$ 
      for all  $(R', C') \in N_n$  where  $C' \subseteq C$  do
         $E := E + \langle (R', C'), (R', C) \rangle$ 
      end for
    end if
  end for
end for
```

1. Each node in $(R, C) \in N_n$ is taken in turn.
2. Each k -vertex $R' \in V_b \cup V_l$ is checked to see if it decomposes (R, C) . For each k -vertex R' found to decompose (R, C) :
 - A new node (R', C) is added to N_d .
 - An arc linking (R', C) to (R, C) is added to E .
 - For each node $(R', C') \in N_n$ representing a ‘subproblem’ of the decomposition, an arc linking (R', C') to (R', C) is added to E .

It should be noted that the checks to see if R' decomposes (R, C) , and the search for all subproblems $(R', C') \in N_n$ each have time complexity $O(|\mathcal{V}|)$ at worst. Also, for each time a k -vertex $R' \in V_l$ is processed, we know there are no nodes $(R', C') \in N_n$. Thus the time complexity of *build-graph* is $O(|N_n||V_b \cup V_l||\mathcal{V}|)$ at worst. The size of the list N_d is bounded by $|N_n||V_b \cup V_l|$.

calc-width: Calculates the best possible width for a node (recursively). If the node is in N_n or is (\emptyset, \mathcal{V}) , then it’s width is calculated as the minimum of the widths of all possible ‘decompositions’, or ∞ if there are no decompositions. If the node is in N_d , it’s width is calculated as the maximum of $|R|$ and the widths of all it’s ‘problem’ nodes. Processing each node in N_d takes $O(|\mathcal{V}|)$ time at worst; all other nodes take $O(|V_b|)$. Thus the time complexity of *calc-width* is $O(|N_d||\mathcal{V}| + |N_n||V_b| + |V_b|)$ at worst.

Procedure 15.4 *calc-width* (R, C)

```
if  $width(R, C)$  is not defined then
  if  $(R, C) \in N_d$  then
     $width(R, C) := |R|$ 
    for all  $\langle (R, C'), (R, C) \rangle \in E$  do
       $calc-width(R, C')$ 
       $width(R, C) := \max(width(R, C), width(R, C'))$ 
    end for
  else
     $width(R, C) := \infty$ 
    for all  $\langle (R', C), (R, C) \rangle \in E$  do
       $calc-width(R', C)$ 
       $width(R, C) := \min(width(R, C), width(R', C))$ 
    end for
  end if
end if
```

Procedure 15.5 *extract-decomposition* (R, C)

```
choose a predecessor  $(R', C) \in N_d$  of  $(R, C)$  with  $width(R', C) = width(R, C)$ 
create a new node  $n \in nodes(T)$ 
 $\lambda(n) := R'$ 
 $\chi(n) := adj(C) \cap var(\lambda(n))$ 
for all predecessors  $(R', C') \in N_n$  of  $(R', C)$  do
   $m := extract-decomposition(R', C')$ 
   $edges(T) := edges(T) + \langle n, m \rangle$ 
end for
return  $n$ 
```

extract-decomposition: Generates the hypertree decomposition $\langle T, \chi, \lambda \rangle$ from the graph structure built by *init-graph* and *build-graph*. A minimum weighted tree is extracted recursively from the graph structure, starting with the root node (\emptyset, \mathcal{V}) . For each problem node (R, C) given to *extract-decomposition*, a minimum-weighted decomposition node (R', C) is chosen. A new node $n \in nodes(T)$ is created for (R', C) , with $\lambda(n) = R'$ and $\chi(n) := adj(C) \cap var(\lambda(n))$. Each of the problem nodes (R', C') linked to (R', C) is then processed recursively, and the results of their decomposition linked to n . A bound on the time complexity of *extract-decomposition* is the time complexity of *calc-width*, though in practise it is significantly faster.

From the above explanations and pseudo-code of *init-vertices*, *init-graph* and *build-graph* we can determine that worst-case time complexity for red- k -decomp is $O(|C|^{2k}|\mathcal{V}|^2)$. Note that for our complexity analysis we assume that a list append (denoted by $+$) takes constant time. To determine the best-case complexity of red- k -decomp, we consider the class of CSPs satisfying:

1. $|V_b \cup V_l|$ increases linearly with respect to $|C|$. As the singleton sets $\{c\} \subseteq C$ are each elements of V_l , we know that $|C|$ is the minimum possible size of $|V_b \cup V_l|$. That a class of CSPs exists where $|V_b \cup V_l|$ increases linearly with respect to $|C|$ has been proven by example in Table 6.2.
2. For any k -vertex R , the number of $[R]$ -components is bounded by a fixed value. This bounds the size of N_n to a fixed multiple of $|V_b|$, rather than $|V_b||\mathcal{V}|$.

From these assumptions, the best-case complexity of red- k -decomp can be shown to be $O(|C|^k|\mathcal{V}| + |C|^2|\mathcal{V}|)$. A summary of the complexity results are presented in Table 6.1. We omit results for *calc-width* and *extract-decomposition*.

Although the worst-case complexity of red- k -decomp is $O(|C|^{2k}|\mathcal{V}|^2)$, experiments show it's average-case complexity is significantly less. We can even identify a fairly broad class of CSPs which can be decomposed with the best-case complexity $O(|C|^k|\mathcal{V}| + |C|^2|\mathcal{V}|)$. We have found that classes of CSPs which exhibit best-case complexity usually have repeating patterns (i.e. where an increment in the number of variables and constraints yields a graph with a similar topology, only larger).

As proof that best-case complexity is achievable, we present the performance results from a general class of CSPs that still obeys our assumptions for best-case complexity. Further investigation of this data also provides us with more information on the effectiveness of reduced normal form. For example, Table 6.2 shows that, as the number of variables and

Worst Case	Time Complexity	Space Complexity
<i>init-vertices</i>	$ C ^k \mathcal{V} $	$ C ^k$
<i>init-graph</i>	$ V_b \mathcal{V} ^2$	$ V_b \mathcal{V} ^2$
<i>build-graph</i>	$ N_n V_b \cup V_l \mathcal{V} $	$ N_n V_b \cup V_l $
Overall	$ C ^{2k} \mathcal{V} ^2$	$ C ^{2k} \mathcal{V} + C ^k \mathcal{V} ^2$
Best-Case	Time Complexity	Space Complexity
<i>init-vertices</i>	$ C ^k \mathcal{V} $	$ C $
<i>init-graph</i>	$ V_b \mathcal{V} $	$ V_b \mathcal{V} $
<i>build-graph</i>	$ N_n V_b \cup V_l \mathcal{V} $	$ N_n V_b \cup V_l $
Overall	$ C ^k \mathcal{V} + C ^2 \mathcal{V} $	$ C ^2 + C \mathcal{V} $

Table 6.1: Complexity results for red- k -decomp

$ \mathcal{V} $	$ \mathcal{C} $	Cond 2	Cond 3	$ V_l $	$ V_b $	$ N_n $	init	build
10	15	60	20	350	145	290	0.00	0.02
20	35	235	3855	2735	350	700	0.03	0.72
30	55	415	21220	5620	520	1040	0.11	2.57
40	75	595	60580	8510	690	1380	0.34	5.05
50	95	775	129940	11400	860	1720	0.82	8.34
60	115	955	237300	14290	1030	2060	1.69	12.32
70	135	1135	390660	17180	1200	2400	3.15	17.17
80	155	1315	598020	20070	1370	2740	5.37	22.77
90	175	1495	867380	22960	1540	3080	8.62	29.00
100	195	1675	1206740	25850	1710	3420	13.18	36.29
110	215	1855	1624100	28740	1880	3760	19.28	44.24
120	235	2035	2127460	31630	2050	4100	27.41	52.92
130	255	2215	2724820	34520	2220	4440	37.67	62.22

Table 6.2: Results of red- k -decomp on successively larger ‘spider webs’ constraint graphs. ‘Cond 2’ and ‘Cond 3’ shows the number of k -vertices discarded due to conditions 2 and 3 of RNF. Run-times (in seconds) for *init-vertices* and *init-graph* are given as a combined value.

constraints increase, the number of possible k -vertices generated by *init-vertices* increase polynomially. This is an unavoidable aspect of red- k -decomp, and will be present in all CSPs. Further, for this particular class of CSPs we can remove only a linearly increasing number of k -vertices by checking for redundant edges.

However, the number of k -vertices R which are neither suitable for branch nodes (have more than one $[R]$ -component) or as leaf nodes ($\text{var}(R)$ is not a connected set) increases polynomially with the same index as the number of possible k -vertices. By rejecting such k -vertices, the size of $|V_b \cup V_l|$ is reduced significantly. Further, the number of nodes in N_n is precisely twice that of the number of k -vertices in V_b , ensuring that $|N_n|$ also increases linearly with respect to the size of the CSP.

That CSPs with repeating structures can be decomposed with best-case or near-best-case complexity is an interesting result, as many CSPs can exhibit repeating structure. Although, for a given repeating structure, it is possible to develop a heuristic which decomposes it in less time than red- k -decomp, it is more useful to have a general method. A general method like red- k -decomp allows for CSPs with mostly repeating structure, but which can have minor, non-repeating elements.

6.4 Performance

We have shown that the definition of reduced normal form provides us with the ability to remove redundant k -vertices with the aim of reducing the length of the outer loops of $\text{opt-}k\text{-decomp}$. Utilising Theorem 14 and condition 2 of RNF we can remove many k -vertices from consideration before encountering the outer loops. Also by identifying k -vertices with multiple components (and assuming the first node in a hypertree decomposition must always have more than one child) we can limit the number of problem nodes created.

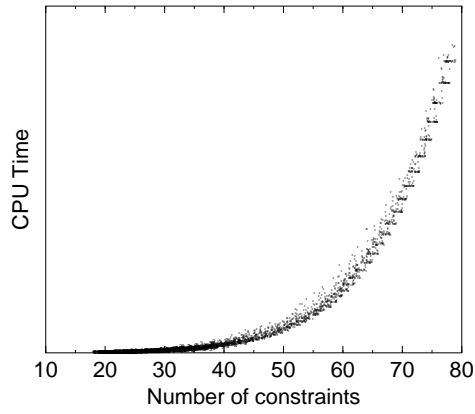
Specifically, best-case complexity of $\text{red-}k\text{-decomp}$ has been shown to be $O(|\mathcal{C}|^k|\mathcal{V}| + |\mathcal{C}|^2|\mathcal{V}|)$ (a class of CSPs has been identified which achieves best-case complexity by ensuring the number of k -vertices remaining for consideration increases linearly with the number of constraints). Unfortunately the worst-case complexity of $\text{red-}k\text{-decomp}$ remains at $O(|\mathcal{C}|^{2k}|\mathcal{V}|^2)$.

For more general CSPs, we can show (but not prove) that an implementation of $\text{red-}k\text{-decomp}$ tends to have best-case complexity rather than worst-case. Figure 6.4(a) shows recorded CPU times of this implementation executed on random binary CSPs with 20 variables, with $k = 3$. Figure 6.4(b) shows that the ratio between worst-case complexity and recorded CPU time tends to zero as the number of constraints increase. This indicates that average-case complexity is less than the theoretical worst-case. It is possible that the actual worst-case complexity is less than $O(|\mathcal{C}|^{2k}|\mathcal{V}|^2)$.

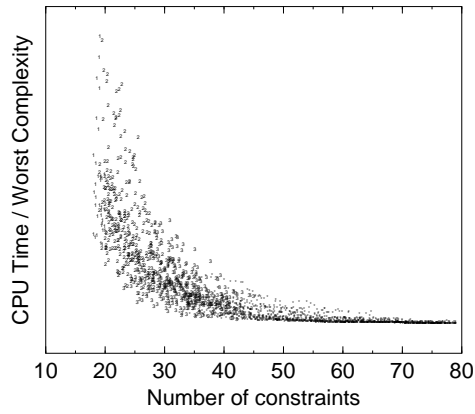
Figure 6.4(c) also shows that the ratio between best-case complexity and recorded CPU time changes only slightly as the number of constraint increases, appearing to converge to a fixed value. While this is not conclusive evidence, it does indicate that the average-case complexity of the modified $\text{opt-}k\text{-decomp}$ algorithm may be very close to the best-case complexity. We expect that many classes of CSPs will be decomposable with near-best-case complexity.

Finally, as a comparison between the original and modified $\text{opt-}k\text{-decomp}$, we present a graph of the ratios of CPU execution times. Each line represents a class of CSP (described by using a fixed random seed, and progressively adding more constraints). The main observations with respect to this graph is that:

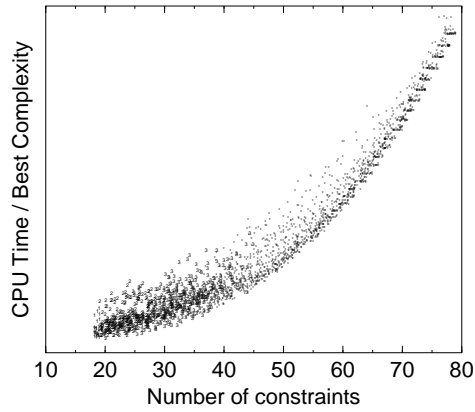
1. At no time does $\text{red-}k\text{-decomp}$ take more time to decompose a CSP than the original $\text{opt-}k\text{-decomp}$.
2. As the number of constraints increases beyond a certain threshold, the modified $\text{opt-}k\text{-decomp}$ can become significantly faster. For one sample point, the modifications provide a 1000-fold increase in speed.



(a)

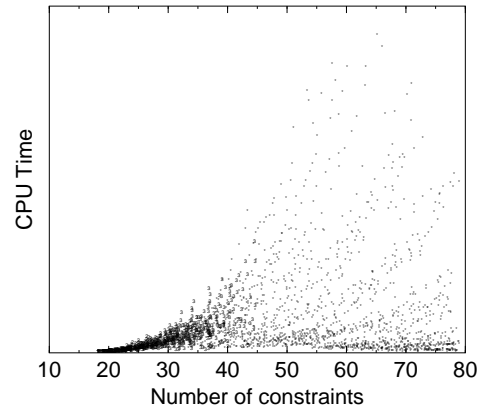


(b)

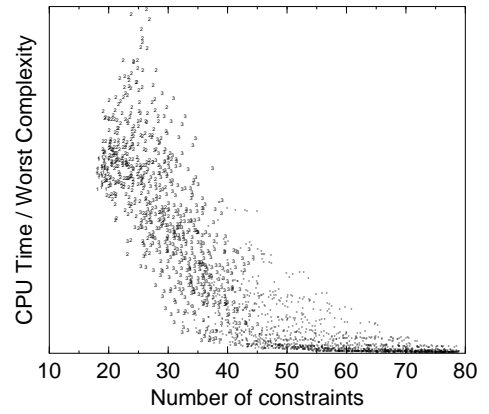


(c)

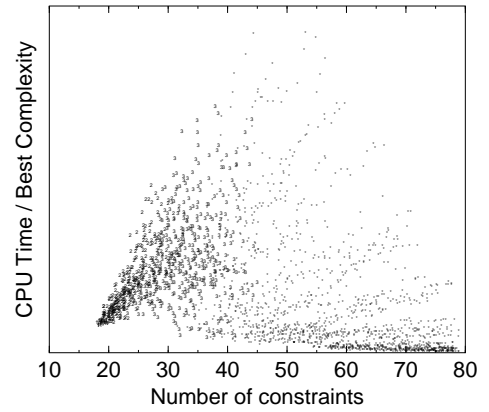
Figure 6.3: Graphs of the CPU time of $\text{opt-}k\text{-decomp}$, and comparisons to its best-case and worst-case complexity functions. Each point represents a single run of $\text{opt-}k\text{-decomp}$.



(a)



(b)



(c)

Figure 6.4: Graphs of the CPU time of $\text{red-}k\text{-decomp}$, and comparisons to its best-case and worst-case complexity functions. Each point represents a single run of $\text{red-}k\text{-decomp}$.

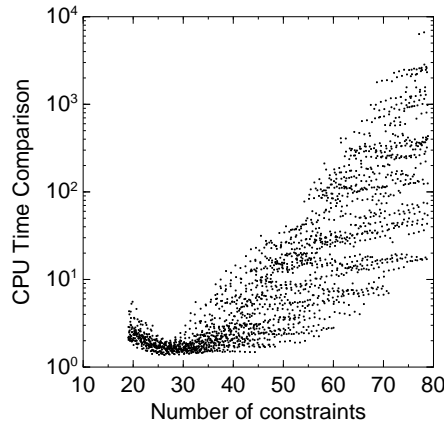


Figure 6.5: A comparison of CPU times for random CSP instances. Values are computed as the CPU time for *opt-k-decomp* divided by the CPU time for *red-k-decomp*.

6.5 Application Within DisCSP Agents

We have presented a new algorithm for finding optimal hypertree decompositions, similar to *opt-k-decomp* but with significantly lower worst-case complexity. The new algorithm is particularly well suited for large CSPs with repetitive structure as the restrictions of reduced normal form (on which *red-k-decomp* is based) reduce the number of *k*-vertices. Although *red-k-decomp* offers improvements over *opt-k-decomp*, it is still unsuitable for very large CSPs with high hypertree-width. In such cases, the cost of *init-vertices* will significantly outweigh the cost of any other part of *red-k-decomp*. However, this method is of particular interest when constructing a distributed constraint satisfaction system.

Individual agents often have small multi-variable problems to solve internally. Such agents are most often modelled as single-variable agents, where the domain of the single variable is equal to the set of solutions to the multi-variable problem. In doing so, DisCSP algorithms are assuming that agents are able to efficiently and rapidly solve their internal constraint satisfaction problems.

Hypertree decompositions were initially developed as a means for conducting efficient queries of relational databases. They provide an efficient means to transform a set of constraints into a set of solutions, which is a necessary step if multiple variables are to be represented as a single variable. Therefore, hypertree decompositions make an appropriate choice of solution method for any CSPs embedded inside an agent of a DisCSP. This component of a distributed constraint satisfaction system is often overlooked during research on DisCSP algorithms, but is a necessary component in the construction of a fully-functional DisCSP *system*. While decomposition techniques such as these are normally applied only to small problems, we argue that they are also of significant value when solving large DisCSP.

Chapter 7

Conclusion

This thesis, ultimately, attempts to address the problems of using distributed constraint satisfaction within agent-based systems. Therefore, we began this thesis by reviewing the definition of DisCSP and CSP, and their relationship to agent-based systems (Chapter 1). We established a number of desirable criteria for any distributed constraint satisfaction algorithms, and determined to accept a distributed algorithm if it:

- has no need for ‘authority’ between variables, effectively avoiding the need for a total order on variables.
- does not add links between variables, avoiding the eventual need for ‘broadcasting’ assignments.
- is complete, avoiding the risk of cyclic behaviour exhibited by local search.

We developed these criteria by considering two example problems that involve a potentially unlimited number of naturally-distributed participants. These criteria are arguably necessary for any distributed constraint satisfaction algorithm to be used in the wider ‘agent-based’ domain as, ultimately, an agent-based system involves an unbounded number of participants. We do not believe that any existing distributed constraint satisfaction algorithm satisfies all of these criteria.

In Chapter 2 we established a new and general method for proving completeness of distributed and centralised constraint satisfaction algorithms. A completeness proof produced using this method will necessarily describe how the algorithm ‘prunes’ the set of consistent assignments. We believe this method provides more insight into each algorithm, and allows for relatively direct comparisons of their operation - something which was not previously possible. This proof method was then used during our review of prominent distributed constraint satisfaction algorithms presented.

As a result of our review, we noted that existing algorithms either require authority between variables, add links, or are not complete, and so fail the above criteria. More importantly, we achieved an in-depth understanding of how each algorithm achieves completeness;

a critical task given that we wish to develop our own algorithm. Using our proofs, we could formally categorise the reviewed algorithms into 4 relatively intuitive classes:

- Those algorithms which are not complete. Most local search algorithms belong to this class, and are quite successful in selected domains.
- Those algorithms which achieve completeness by a very structured and limiting approach to search. Such methods use bounded memory, yet are highly limited in their flexibility to move within a search space.
- Those algorithms which achieve completeness by using unbounded or exponential memory. Most local search algorithms can be made complete by combining some form of flexible backtracking with an unbounded memory.
- Those algorithms which achieve completeness by careful manipulation of a bounded memory. These algorithms provide more flexibility than simple progression algorithms, without the costs associated with unbounded memories.

Learning from these algorithms, we then introduced a new algorithm called Support-Based Distributed Search (Chapter 3). SBDS was inspired by and derived from observations of how networks of humans are able to solve real-world distributed constraint satisfaction problems. A simple meeting scheduling problem was considered, with a hypothetical ‘dialogue’ between humans who attempt to solve the problem. SBDS was designed to parallel human behaviour as much as possible, and so we mapped the dialogue into a formal notation. As a result, SBDS can be said to use a simple form of ‘argumentation’ to negotiate value assignments between agents.

To represent some of our ‘arguments’ it was necessary to introduce the concept of ‘isgoods’. An isgood is an ordered partial assignment satisfying all constraints within its scope, and where successive variables are neighbours within the original constraint graph. We chose to interpret an isgood as ‘a message from one agent to another, requesting assistance in further exploring a potential partial solution’. When combined with nogoods, SBDS represents a primitive executable argumentation system.

As with all distributed constraint satisfaction algorithms, there was a high potential for infinite cycles in SBDS. We addressed cycles by ensuring:

- agents do not propose arbitrary values for variables outside their direct control.
- agents propose successively stronger isgoods.
- agents postpone ‘lesser’ isgoods if they suspected a cycle of isgoods was underway.

This is a particularly novel method for resolving cycles, and is unique to SBDS. It requires a total order over isgoods but not over agents, and so does not impose any kind of clear

authority between agents. Using this ‘postponement’ technique, we can prove that SBDS is complete. Note that if any part of this technique is not correctly applied, then SBDS will become incomplete.

Initial empirical evaluation of SBDS showed that it had particularly bad performance, most of which could be ascribed to its inflexibility in choosing values. We therefore constructed a simple variation on SBDS that permits a limited amount of heuristic value selection, minimising conflicts and/or change of variable values. Similarly, we constructed another variant of SBDS that permits agents to propose some number of equally-strong arguments, effectively reducing the amount of communication between agents.

We have qualitatively compared SBDS to other distributed algorithms. We showed that SBDS has little in common with Asynchronous Weak Commitment search, sharing just the concept of ‘nogoods’. Surprisingly, SBDS has much in common with the incomplete Distributed Breakout Algorithm. Indeed, SBDS can be cast as a form of Breakout that becomes increasingly more systematic in its behaviour by sharing more information between agents. Backtracking search algorithms have been particularly inspirational in the development of SBDS, and this is apparent in our qualitative comparison with ABT-DO. SBDS can, in some sense, be considered a parallelised form of a backtracking search algorithm, with runtime development of agent coalitions.

We have quantitatively compared SBDS to AWCS, ABT, and ABT-DO. We showed that SBDS:

- does not scale well on problems with larger domains or higher constraint density.
- scales significantly better than the reviewed algorithms when increasing the number of variables.
- is much less likely to generate nogoods than the reviewed algorithms.

Finally, as part of a wider work on distributed constraint satisfaction problems, we have provided an improved method for enumerating all solutions to a small, local problem instance. This is a necessary component of any distributed constraint satisfaction algorithm, if it were to be deployed in a realisation situation.

7.1 Summary

We have demonstrated that Support-Based Distributed Search satisfies the criteria outlined at the start of the thesis. By using ‘arguments’ between agents, and by providing each agent with a choice of actions, SBDS ensures that no agent is arbitrarily granted authority over any other. Arguments, in the form of isgoods and nogoods, also provide a mechanism for agents to become aware of other variable’s values without introducing links.

We have compared Support-Based Distributed Search to a variety of distinct algorithms for constraint satisfaction, and noted that it has similarities to all, but no direct ancestor. It is therefore a unique algorithm, and has been constructed solely for distributed constraint satisfaction. It is the first algorithm for distributed constraint satisfaction which, to our knowledge, was constructed without a clear centralised ancestor.

We have also provided new proofs of completeness for a variety of centralised and distributed constraint satisfaction algorithms. These proofs use a common base theorem, and provide insight into how each iteration of a given algorithm help establish completeness.

7.2 Future Work

This thesis suggests a number of new avenues for research, either to address the shortcomings of SBDS, or to address the entire field of DisCSP algorithms:

- It seems possible to construct a distributed algorithm that uses polynomial space, but does not use a total order over agents. SBDS nearly achieved this, but there was no straightforward mechanism to maintain completeness in the face of nogood deletions. It would be worthwhile to see if such a mechanism could be developed, or if it could be proven that no such algorithm is possible.
- A comprehensive evaluation of distributed constraint satisfaction algorithms should be conducted, similar to the comparison we have conducted here. It would be interesting to compare each algorithm using both large domains and large numbers of variables.
- We must acknowledge that, while SBDS allows for introduction of new agents and constraints, it does not allow for their change or removal. Algorithms for handling an altered DisCSP should be developed if DisCSP is to serve as a model for ongoing agent negotiation.
- SBDS was not developed to preserve privacy, though individual agents are allowed significant discretion in which information they reveal. It would be interesting to consider a variation of SBDS that preserves privacy and determine the performance tradeoffs.

Bibliography

- [Bak94] Andrew B. Baker. The hazards of fancy backtracking. In AAAI, pages 288–293, 1994.
- [BC03] Edmund K. Burke and Patrick De Causmaecker, editors. Practice and Theory of Automated Timetabling IV, 4th International Conference, PATAT 2002, Gent, Belgium, August 21-23, 2002, Selected Revised Papers, volume 2740 of Lecture Notes in Computer Science. Springer, 2003.
- [BM04] Valerie Barr and Zdravko Markov, editors. Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA. AAAI Press, 2004.
- [BMBM05] Christian Bessière, Arnold Maestre, Ismel Brito, and Pedro Meseguer. Asynchronous backtracking without adding links: a new member in the abt family. Artif. Intell., 161(1-2):7–24, 2005.
- [BMM01] Christian Bessière, Arnold Maestre, and Pedro Meseguer. Distributed dynamic backtracking. In Toby Walsh, editor, CP, volume 2239 of Lecture Notes in Computer Science, page 772. Springer, 2001.
- [CDK99] Zeev Collin, Rina Dechter, and Shmuel Katz. Self-stabilizing distributed constraint satisfaction. Chicago J. Theor. Comput. Sci., 1999, 1999.
- [Dec92] Rina Dechter. Constraint Networks. In Stuart C. Shapiro, editor, Encyclopedia of Artificial Intelligence, volume 1. Addison-Wesley Publishing Company, 1992.
- [EF03] Carlos Eisenberg and Boi Faltings. Making the breakout algorithm complete using systematic search. In Georg Gottlob and Toby Walsh, editors, IJCAI, pages 1374–1375. Morgan Kaufmann, 2003.
- [FG96] Stan Franklin and Arthur C. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In Jörg P. Müller, Michael Wooldridge, and Nicholas R. Jennings, editors, ATAL, volume 1193 of Lecture Notes in Computer Science, pages 21–35. Springer, 1996.
- [FMG05] Boi Faltings and Santiago Macho-Gonzalez. Open constraint programming. Artif. Intell., 161(1-2):181–208, 2005.

- [Fra96] Jeremy Frank. Weighting for godot: Learning heuristics for gsat. In AAAI/IAAI, Vol. 1, pages 338–343, 1996.
- [Fra97] Jeremy Frank. Learning short-term weights for gsat. In IJCAI (1), pages 384–391, 1997.
- [Fre82] Eugene C. Freuder. A Sufficient Condition for Backtrack-free Search. JACM, 29(1):24–32, January 1982.
- [Fre85] Eugene C. Freuder. A Sufficient Condition for Backtrack-bounded Search. JACM, 32(4):755–761, October 1985.
- [GF03] Tamás D. Gedeon and Lance Chun Che Fung, editors. AI 2003: Advances in Artificial Intelligence, 16th Australian Conference on Artificial Intelligence, Perth, Australia, December 3-5, 2003, Proceedings, volume 2903 of Lecture Notes in Computer Science. Springer, 2003.
- [Gin93] Matthew L. Ginsberg. Dynamic backtracking. J. Artif. Intell. Res. (JAIR), 1:25–46, 1993.
- [GLS99a] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI), pages 394–399. Morgan Kaufmann, 1999.
- [GLS99b] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pages 21–32. ACM Press, May 31 - June 2 1999.
- [GLS00] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. Artificial Intelligence, 124(2):243–282, December 2000.
- [GM07] Tal Grinshpoun and Amnon Meisels. Compapo: A complete version of the apo algorithm. In IAT, pages 370–376. IEEE Computer Society, 2007.
- [Ham02] Youssef Hamadi. Interleaved backtracking in distributed constraint networks. International Journal on Artificial Intelligence Tools, 11(2):167–188, 2002.
- [Ham05] Youssef Hamadi. Conflicting agents in distributed search. International Journal on Artificial Intelligence Tools, 14(3):459–476, 2005.

- [HBQ98] Youssef Hamadi, Christian Bessière, and Joël Quinqueton. Distributed intelligent backtracking. In ECAI, pages 219–223, 1998.
- [HCG05] Peter Harvey, Chee Fon Chang, and Aditya K. Ghose. Practical application of support-based distributed search. In ICTAI, pages 34–38. IEEE Computer Society, 2005.
- [HCG06a] Peter Harvey, Chee Fon Chang, and Aditya Ghose. Simple support-based distributed search. In Luc Lamontagne and Mario Marchand, editors, Canadian Conference on AI, volume 4013 of Lecture Notes in Computer Science, pages 159–170. Springer, 2006.
- [HCG06b] Peter Harvey, Chee Fon Chang, and Aditya Ghose. Support-based distributed search: A new approach for multiagent constraint processing. In Nicolas Maudet, Simon Parsons, and Iyad Rahwan, editors, ArgMAS, volume 4766 of Lecture Notes in Computer Science, pages 91–106. Springer, 2006.
- [HCG06c] Peter Harvey, Chee Fon Chang, and Aditya Ghose. Support-based distributed search: a new approach for multiagent constraint processing. In Hideyuki Nakashima, Michael P. Wellman, Gerhard Weiss, and Peter Stone, editors, AAMAS, pages 377–383. ACM, 2006.
- [HG03] Peter Harvey and Aditya Ghose. Reducing redundancy in the hypertree decomposition scheme. In ICTAI, pages 474–481. IEEE Computer Society, 2003.
- [HY05] Katsutoshi Hirayama and Makoto Yokoo. The distributed breakout algorithms. Artif. Intell., 161(1-2):89–115, 2005.
- [JG93] Ari K. Jónsson and Matthew L. Ginsberg. Experimenting with new systematic and nonsystematic search techniques. In Proceedings of the AAAI Spring Symposium on AI and NP-Hard Problems, Stanford, CA, 1993.
- [KM05] E. Kaplansky and A. Meisels. Distributed personnel scheduling - negotiation among scheduling agents. Annals of Operations Research, 2005. Accepted, July 2005.
- [LMS02] Nicola Leone, Alfredo Mazzitelli, and Francesco Scarcello. Cost-Based Query Decompositions. In Proceedings of SEBD 2002, 2002.
- [MJPL92] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. Artificial Intelligence, 58(1-3):161–205, 1992.

- [MK02] Amnon Meisels and Eliezer Kaplansky. Scheduling agents - distributed timetabling problems(disttp). In Burke and Causmaecker [BC03], pages 166–180.
- [MKRZ02] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. 2002.
- [ML04] Roger Mailler and Victor R. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In AAMAS, pages 438–445. IEEE Computer Society, 2004.
- [ML06] Roger Mailler and Victor R. Lesser. Asynchronous partial overlay: A new algorithm for solving distributed constraint satisfaction problems. J. Artif. Intell. Res. (JAIR), 25:529–576, 2006.
- [Mor93] Paul Morris. The breakout method for escaping from local minima. In AAAI, pages 40–45, 1993.
- [MSTY05] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artif. Intell., 161(1-2):149–180, 2005.
- [Pet04] Adrian Petcu. Heuristics for the distributed breakout algorithm. In Mark Wallace, editor, CP, volume 3258 of Lecture Notes in Computer Science, page 804. Springer, 2004.
- [PF05] Adrian Petcu and Boi Faltings. Dpop: A scalable method for multiagent constraint optimization. In IJCAI 05, pages 266–271, Edinburgh, Scotland, Aug 2005.
- [RN03] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.
- [TS98] John Thornton and Abdul Sattar. Using arc weights to improve iterative repair. In AAAI/IAAI, pages 367–372, 1998.
- [vB05] Peter van Beek, editor. Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings, volume 3709 of Lecture Notes in Computer Science. Springer, 2005.
- [VT95] C. Voudouris and E. Tsang. Guided local search, 1995.

- [Wal95] Richard Wallace. Directed Arc Consistency Processing. In Manfred Meyer, editor, Constraint Processing, Selected Papers, volume 923 of Lecture Notes in Computer Science. Springer, 1995.
- [WF05] Richard J. Wallace and Eugene C. Freuder. Constraint-based reasoning and privacy/efficiency tradeoffs in multi-agent problem solving. Artif. Intell., 161(1-2):209–227, 2005.
- [WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. Knowledge Engineering Review, 10(2):115–152, 1995.
- [YD91] Makoto Yokoo and Edmund H. Durfee. Distributed constraint optimization as a formal model of partially adversarial cooperation. Technical Report CSE-TR-101-91, Ann Arbor, MI 48109, 1991.
- [YDIK92] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In ICDCS, pages 614–621, 1992.
- [YDIK98] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. IEEE Trans. Knowl. Data Eng., 10(5):673–685, 1998.
- [YH95] Makoto Yokoo and Katsutoshi Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. In Victor Lesser, editor, Proceedings of the First International Conference on Multi-Agent Systems. MIT Press, 1995.
- [YH98] Makoto Yokoo and Katsutoshi Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In ICMAS, pages 372–381. IEEE Computer Society, 1998.
- [YH00a] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems, 3(2):185–207, 2000.
- [YH00b] Makoto Yokoo and Katsutoshi Hirayama. The effect of nogood learning in distributed constraint satisfaction. In ICDCS, pages 169–177, 2000.
- [Yok94] Makoto Yokoo. Weak-commitment search for solving constraint satisfaction problems. In AAAI, pages 313–318, 1994.

- [Yok95] Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In Ugo Montanari and Francesca Rossi, editors, CP, volume 976 of Lecture Notes in Computer Science, pages 88–102. Springer, 1995.
- [YSH02] Makoto Yokoo, Koutarou Suzuki, and Katsutoshi Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. In Pascal Van Hentenryck, editor, CP, volume 2470 of Lecture Notes in Computer Science, pages 387–401. Springer, 2002.
- [YSH05] Makoto Yokoo, Koutarou Suzuki, and Katsutoshi Hirayama. Secure distributed constraint satisfaction: reaching agreement without revealing private information. Artif. Intell., 161(1-2):229–245, 2005.
- [ZM05a] R. Zivan and A. Meisels. Asynchronous backtracking for asymmetric discsps. In Proc. 6th Intern. Workshop on Distributed Constraint Reasoning (DCR-05), pages 148–160, July, 2005.
- [ZM05b] Roie Zivan and Amnon Meisels. Dynamic ordering for asynchronous backtracking on discsps. In van Beek [vB05], pages 32–46.
- [ZM06a] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. Constraints, 11:179–197, 2006.
- [ZM06b] R. Zivan and A. Meisels. Message delay and asynchronous discsp search. Archives of Control, 16:221–242, 2006.
- [ZM06c] Roie Zivan and Amnon Meisels. Concurrent search for distributed csp. Artif. Intell., 170(4-5):440–461, 2006.
- [ZM06d] Roie Zivan and Amnon Meisels. Generic run-time measurement for discsps search algorithms. IOS Press, IOS Press, 2006.
- [ZM06e] Roie Zivan and Amnon Meisels. Message delay and discsp search algorithms. Ann. Math. Artif. Intell., 46(4):415–439, 2006.
- [ZTS03] Lingzhong Zhou, John Thornton, and Abdul Sattar. Dynamic agent ordering in distributed constraint satisfaction problems. In Gedeon and Fung [GF03], pages 427–439.
- [ZTS04] Lingzhong Zhou, John Thornton, and Abdul Sattar. Dynamic agent-ordering and nogood-repairing in distributed constraint satisfaction problems. In Barr and Markov [BM04].

- [ZW02] Weixiong Zhang and Lars Wittenburg. Distributed breakout revisited. In AAAI/IAAI, pages 352–, 2002.
- [ZWXW05] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. Artif. Intell., 161(1-2):55–87, 2005.

Appendix A

Results

The following sets of tables and figures are the results of experiments conducted in Chapter 5. A total three problem sets were considered, with each problem set focusing on scaling behaviour for a specific problem parameter (domain size, constraint degree, or number of variables). As each problem set contains both feasible and infeasible instances, we present the results of each set separately for ‘feasible’, ‘infeasible’, and ‘all’ instances.

As there are many ways to compare distributed constraint satisfaction algorithms, we have needed to consider a wide variety of possible measures. The measures presented here are:

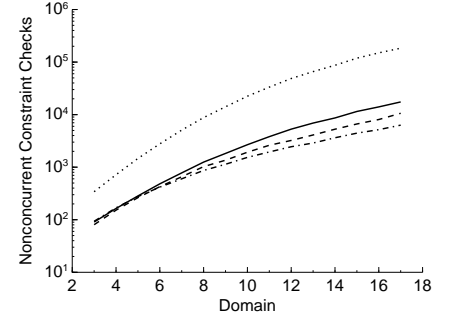
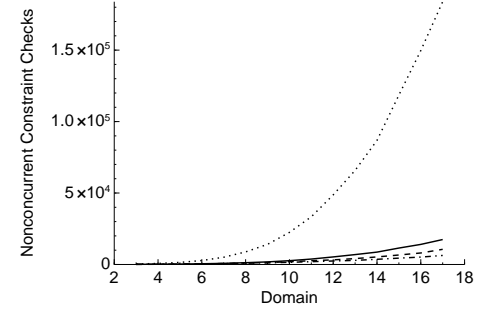
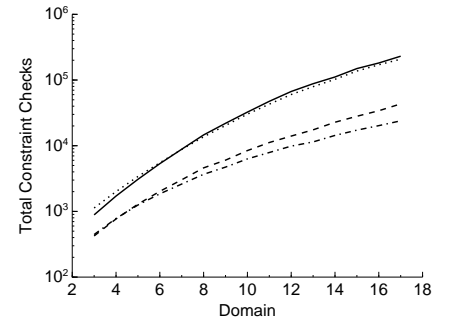
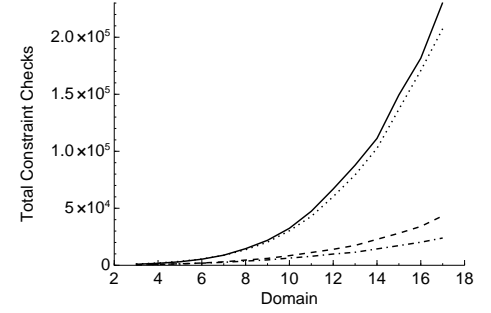
- Total and Non-Concurrent Constraint Checks
- Total and Non-Concurrent Nogood Checks
- Total and Non-Concurrent Network Traffic (Bytes)
- Total and Non-Concurrent Number of Packets
- Average Packet Size
- Processor Time
- Concurrent Checks
- Concurrent Traffic

For each measure and algorithm we present numerical results, a graph of results, and a log-scaled graph of results. Each presentation of data is targeted at a different need of the reader, though this does produce a rather large set of tables and graphs. For highlighted results and commentary, see Chapter 5.

Figure A.1: Number of constraint checks for **all instances** in **problem set 1**.

Total Constraint Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	886	1,128	445	424
4	1,715	2,003	781	767
5	3,078	3,402	1,251	1,288
6	5,333	5,508	1,853	2,021
7	8,843	8,771	2,608	3,031
8	14,591	13,671	3,671	4,599
9	21,915	20,322	4,722	6,031
10	32,469	30,255	6,305	8,434
11	47,408	42,911	7,882	11,188
12	66,890	60,198	9,851	14,013
13	87,816	79,272	11,426	17,440
14	111,298	102,573	14,341	22,766
15	149,405	136,858	17,253	28,099
16	181,598	170,760	20,212	34,053
17	230,473	207,487	23,905	43,337

Nonconcurrent Constraint Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	91	342	94	80
4	164	720	168	152
5	283	1,466	276	264
6	482	2,778	417	426
7	779	5,070	600	658
8	1,251	8,785	866	1,025
9	1,826	14,093	1,136	1,357
10	2,676	22,445	1,536	1,926
11	3,811	33,367	1,950	2,611
12	5,295	48,682	2,449	3,223
13	6,905	65,831	2,883	4,103
14	8,652	86,814	3,627	5,279
15	11,504	118,522	4,445	6,641
16	14,048	149,669	5,180	8,065
17	17,448	183,867	6,314	10,597

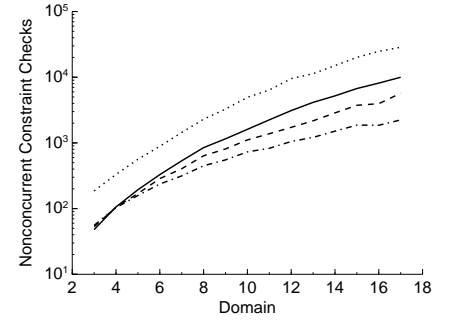
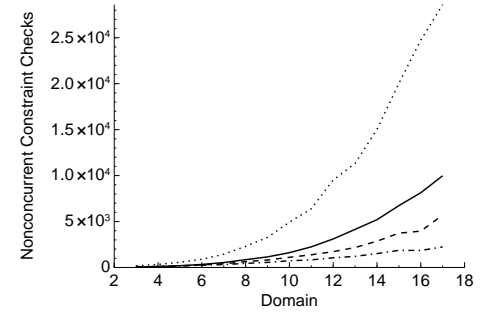
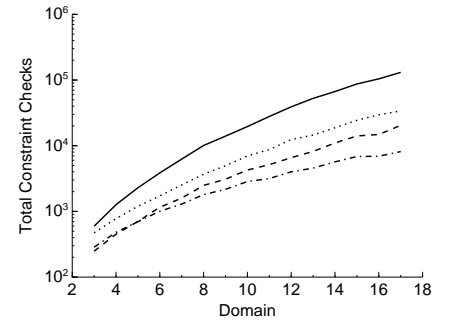
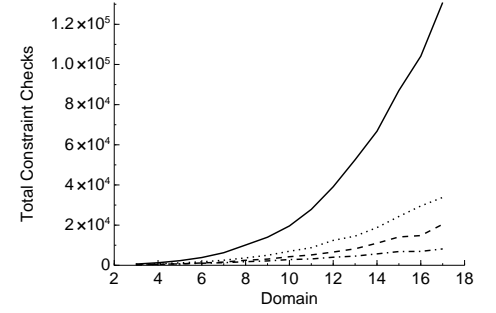


———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.2: Number of constraint checks for **feasible instances** in **problem set 1**.

Total Constraint Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	597	473	285	248
4	1,267	775	480	450
5	2,291	1,197	698	704
6	3,855	1,724	1,006	1,158
7	6,260	2,534	1,295	1,595
8	10,085	3,684	1,800	2,487
9	13,984	4,961	2,168	3,089
10	19,579	6,984	2,846	4,262
11	27,853	8,760	3,160	5,144
12	39,090	12,389	4,010	6,577
13	52,553	14,514	4,560	8,200
14	66,768	18,782	5,678	11,008
15	87,033	24,314	6,843	14,101
16	104,122	29,535	6,908	14,803
17	130,858	33,808	8,143	20,423

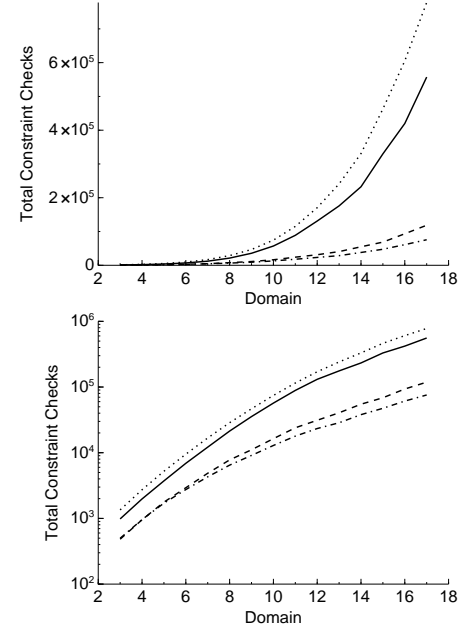
Nonconcurrent Constraint Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	48	186	56	53
4	106	331	103	105
5	194	563	158	170
6	329	887	237	286
7	534	1,442	315	405
8	847	2,297	448	637
9	1,155	3,275	551	810
10	1,609	4,932	730	1,111
11	2,243	6,392	828	1,379
12	3,108	9,513	1,052	1,726
13	4,146	11,324	1,214	2,169
14	5,200	15,034	1,509	2,864
15	6,728	20,033	1,865	3,737
16	8,137	24,747	1,856	3,957
17	9,987	28,597	2,240	5,708



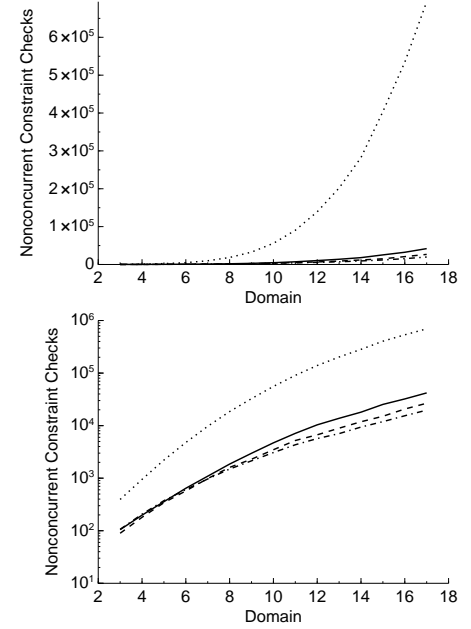
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.3: Number of constraint checks for **infeasible instances** in **problem set 1**.

Total Constraint Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	983	1,349	499	483
4	1,979	2,729	960	954
5	3,727	5,221	1,708	1,770
6	6,893	9,501	2,747	2,931
7	12,151	16,757	4,290	4,869
8	21,341	28,637	6,474	7,764
9	35,524	46,684	9,104	11,078
10	56,980	74,520	12,886	16,370
11	88,741	115,138	17,868	23,972
12	131,067	170,564	23,335	31,180
13	175,915	241,104	28,582	40,531
14	232,557	330,810	37,940	54,795
15	329,945	462,833	47,403	68,643
16	419,801	605,390	61,156	93,296
17	557,295	778,083	75,690	118,618



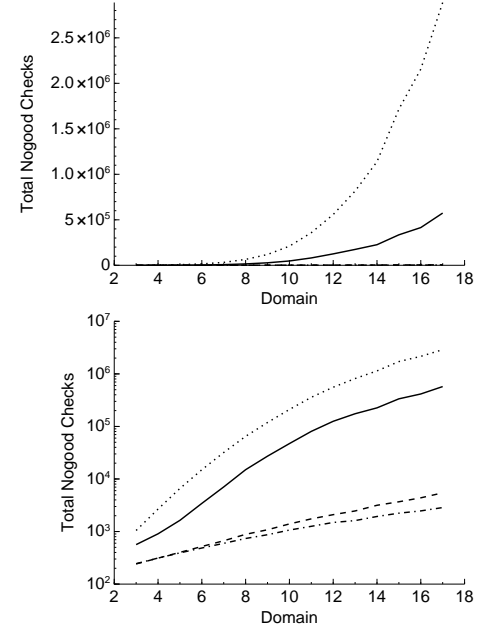
Nonconcurrent Constraint Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	105	394	106	90
4	198	949	207	180
5	357	2,210	373	341
6	643	4,775	608	573
7	1,093	9,713	966	983
8	1,856	18,507	1,492	1,607
9	2,977	32,658	2,140	2,298
10	4,705	55,759	3,069	3,476
11	7,126	90,418	4,324	5,216
12	10,344	139,101	5,674	6,679
13	13,797	202,048	7,054	8,936
14	18,050	282,334	9,397	11,858
15	25,326	403,787	11,919	15,052
16	32,220	534,124	15,411	20,706
17	41,927	693,983	19,698	26,660



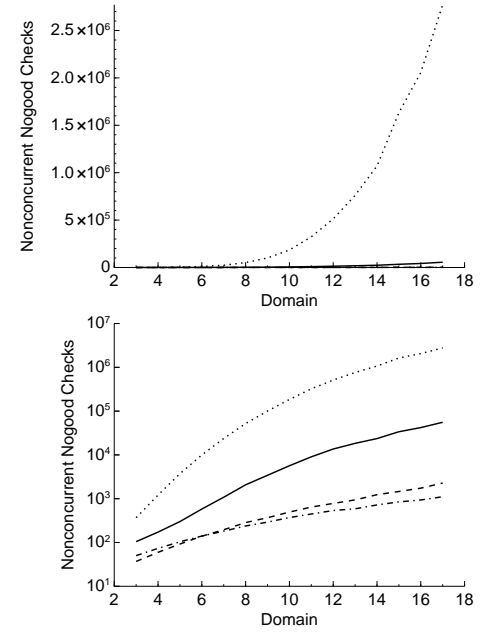
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.4: Number of nogood checks for **all instances** in **problem set 1**.

Domain	SBDS	AWCS	ABTDO	ABT
3	564	1,049	245	242
4	905	2,674	313	312
5	1,634	6,606	393	401
6	3,431	14,986	487	520
7	7,070	31,894	596	668
8	15,022	64,821	737	885
9	27,174	118,708	871	1,082
10	47,184	210,129	1,067	1,395
11	80,624	358,020	1,258	1,745
12	125,246	557,559	1,487	2,095
13	174,525	815,534	1,615	2,464
14	225,932	1,138,985	1,937	3,163
15	335,099	1,716,317	2,233	3,682
16	414,503	2,155,446	2,473	4,377
17	573,634	2,888,263	2,863	5,452



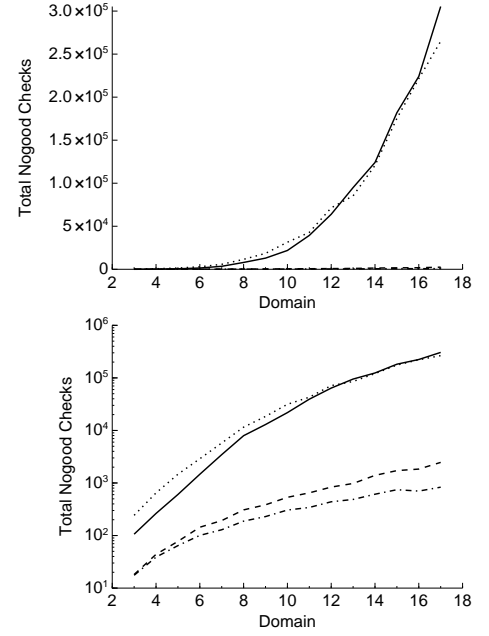
Domain	SBDS	AWCS	ABTDO	ABT
3	105	368	50	37
4	173	1,211	73	60
5	300	3,726	104	92
6	581	9,975	140	139
7	1,081	23,710	182	196
8	2,074	52,093	238	285
9	3,422	100,320	292	368
10	5,648	184,576	371	495
11	9,044	323,471	447	644
12	13,619	512,735	538	781
13	18,367	760,904	588	939
14	23,605	1,070,684	721	1,236
15	33,766	1,632,892	846	1,458
16	42,029	2,056,608	935	1,742
17	55,628	2,772,270	1,110	2,280



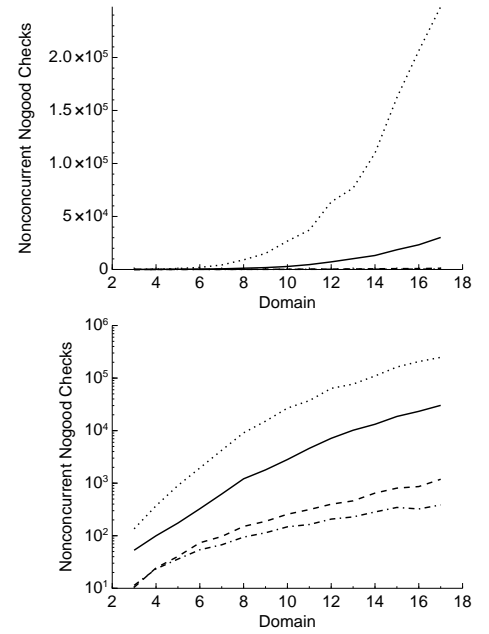
— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.5: Number of nogood checks for **feasible instances** in **problem set 1**.

Domain	SBDS	AWCS	ABTDO	ABT
3	106	243	18	18
4	263	648	39	44
5	604	1,466	64	78
6	1,465	2,892	101	144
7	3,464	5,772	129	193
8	7,945	11,532	188	308
9	12,963	18,484	230	384
10	21,843	31,425	306	534
11	39,353	42,852	342	649
12	63,902	71,389	439	837
13	94,904	85,314	486	980
14	124,093	120,651	612	1,397
15	181,908	175,227	745	1,722
16	224,068	221,494	702	1,840
17	305,284	264,866	832	2,465



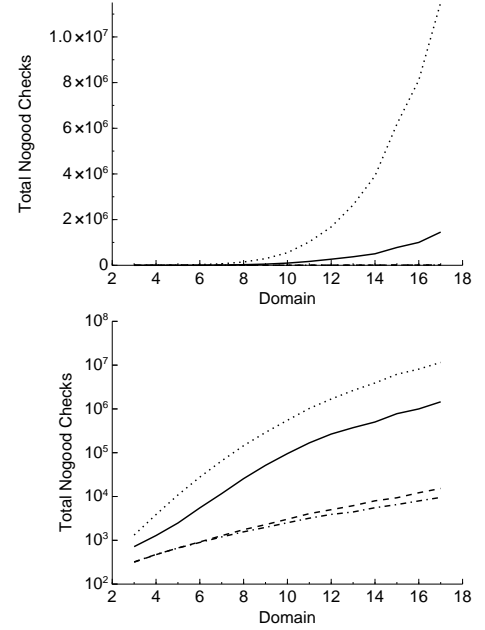
Domain	SBDS	AWCS	ABTDO	ABT
3	53	135	11	10
4	100	372	23	24
5	174	904	36	40
6	326	1,942	54	73
7	622	4,234	67	97
8	1,213	9,089	94	150
9	1,792	15,143	114	188
10	2,818	26,794	148	256
11	4,600	37,172	163	315
12	7,152	63,725	207	398
13	10,175	76,804	228	462
14	13,207	109,818	284	648
15	18,600	162,125	345	806
16	23,274	206,374	323	862
17	30,231	247,910	386	1,188



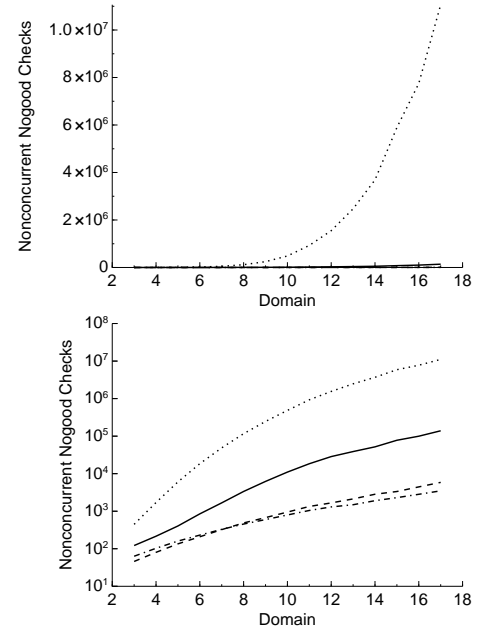
———— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.6: Number of nogood checks for **infeasible instances** in **problem set 1**.

Domain	SBDS	AWCS	ABTDO	ABT
3	718	1,320	321	317
4	1,285	3,871	475	470
5	2,483	10,845	664	668
6	5,506	27,751	894	917
7	11,686	65,338	1,194	1,277
8	25,624	144,672	1,561	1,750
9	51,562	290,707	1,971	2,279
10	95,372	550,051	2,517	3,032
11	167,862	1,024,572	3,193	4,062
12	266,857	1,679,861	3,907	5,000
13	373,443	2,640,383	4,437	6,172
14	503,246	3,912,812	5,545	7,974
15	778,521	6,179,949	6,543	9,358
16	1,000,003	8,107,325	7,922	12,184
17	1,454,053	11,507,018	9,535	15,265



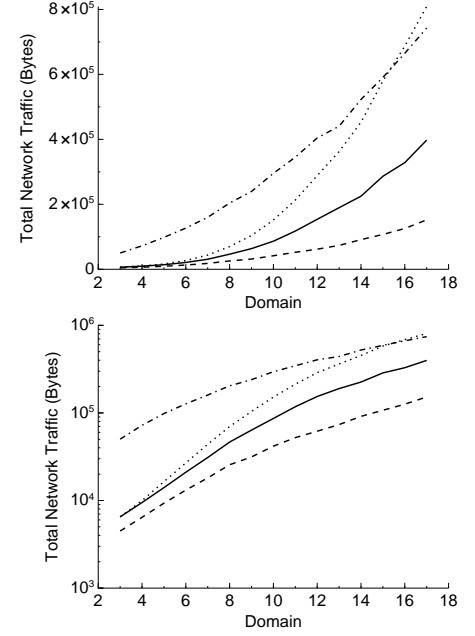
Domain	SBDS	AWCS	ABTDO	ABT
3	122	446	63	46
4	216	1,706	103	80
5	404	6,053	160	135
6	850	18,453	231	209
7	1,669	48,645	328	323
8	3,365	116,533	455	488
9	6,219	246,494	598	676
10	11,031	484,699	795	948
11	18,438	928,968	1,046	1,340
12	28,548	1,549,252	1,301	1,665
13	38,834	2,470,498	1,488	2,130
14	51,922	3,687,975	1,911	2,836
15	77,665	5,892,840	2,296	3,348
16	99,690	7,750,837	2,819	4,452
17	138,952	11,065,654	3,488	5,866



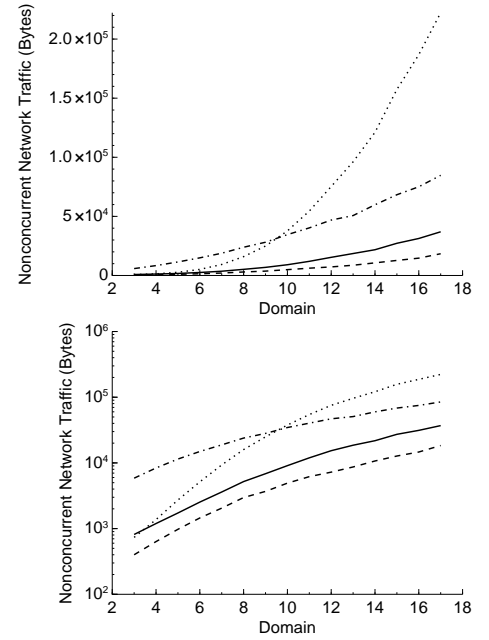
SBDS
 AWCS
 ABTDO
 ABT

Figure A.7: Network traffic for **all instances in problem set 1**.

Domain	SBDS	AWCS	ABTDO	ABT
3	6,501	6,491	50,144	4,481
4	9,487	9,979	72,239	6,436
5	14,027	16,397	98,228	9,319
6	21,045	26,824	126,938	13,163
7	30,964	43,839	160,087	18,123
8	46,475	69,838	203,772	25,573
9	63,690	103,934	239,628	31,470
10	86,439	151,601	295,461	41,703
11	117,613	212,947	345,130	52,208
12	153,940	288,249	404,334	61,811
13	189,569	362,319	440,514	73,567
14	225,036	452,671	522,797	91,320
15	286,338	579,658	590,632	107,186
16	328,273	687,107	664,922	125,690
17	397,557	808,993	742,783	152,180



Domain	SBDS	AWCS	ABTDO	ABT
3	814	736	5,853	400
4	1,194	1,373	8,374	634
5	1,728	2,722	11,450	976
6	2,533	5,145	14,917	1,453
7	3,609	9,306	18,816	2,049
8	5,197	15,945	23,884	2,970
9	6,851	24,933	28,124	3,683
10	9,083	37,704	34,566	4,920
11	11,954	54,490	40,264	6,176
12	15,343	75,262	46,881	7,237
13	18,590	96,165	50,687	8,680
14	21,768	121,100	59,815	10,692
15	27,246	157,391	68,324	12,725
16	31,286	186,823	75,035	14,652
17	36,951	222,320	84,730	18,388

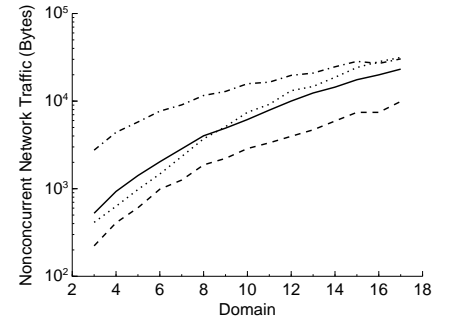
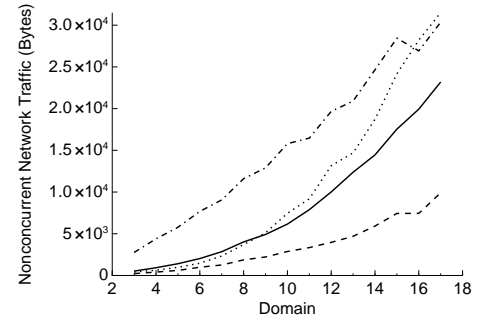
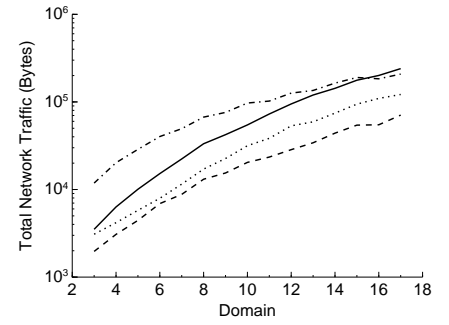
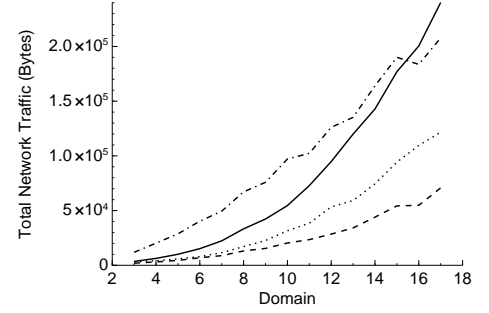


———— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.8: Network traffic for **feasible instances** in **problem set 1**.

Total Network Traffic (Bytes)				
Domain	SBDS	AWCS	ABTDO	ABT
3	3,512	3,086	11,839	1,966
4	6,323	4,182	20,246	3,073
5	10,063	5,783	28,720	4,443
6	15,188	7,939	40,250	6,900
7	22,336	11,524	49,476	8,802
8	33,297	17,105	67,087	13,109
9	42,282	22,603	75,893	15,444
10	54,587	31,604	97,166	20,404
11	72,821	38,401	102,321	23,449
12	94,805	53,338	126,072	28,496
13	119,912	59,219	135,206	34,144
14	142,843	74,446	164,037	43,994
15	176,936	94,267	190,104	54,290
16	200,430	109,570	183,580	54,820
17	240,154	121,665	207,740	70,735

Nonconcurrent Network Traffic (Bytes)				
Domain	SBDS	AWCS	ABTDO	ABT
3	524	413	2,758	223
4	931	630	4,400	408
5	1,412	978	5,775	606
6	2,028	1,472	7,681	987
7	2,852	2,330	9,060	1,256
8	4,020	3,715	11,599	1,871
9	4,913	5,109	12,897	2,217
10	6,160	7,446	15,795	2,872
11	7,884	9,188	16,467	3,341
12	10,014	13,155	19,668	3,966
13	12,378	14,695	20,904	4,698
14	14,454	18,719	24,655	5,926
15	17,540	24,165	28,444	7,437
16	19,935	28,205	26,917	7,443
17	23,188	31,494	30,352	9,907

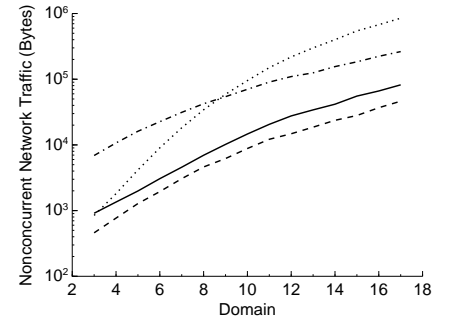
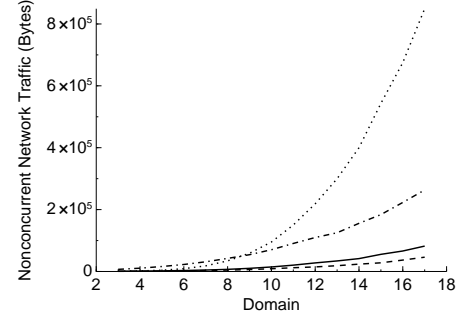
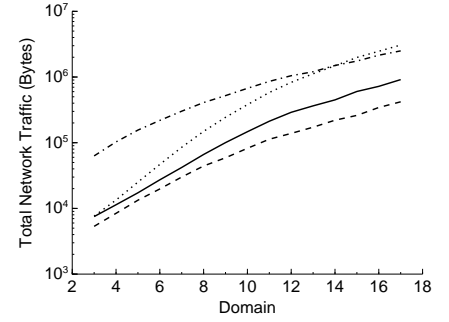
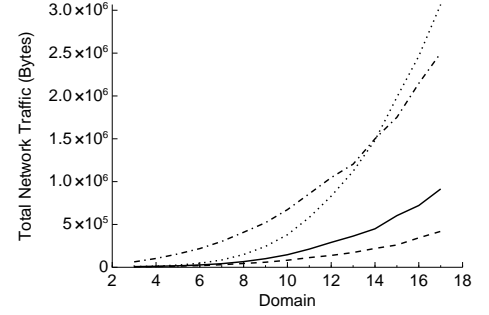


———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.9: Network traffic for **infeasible instances** in **problem set 1**.

Total Network Traffic (Bytes)				
Domain	SBDS	AWCS	ABTDO	ABT
3	7,507	7,638	63,041	5,328
4	11,356	13,405	102,959	8,423
5	17,295	25,150	155,549	13,339
6	27,226	46,756	218,429	19,773
7	42,011	85,212	301,704	30,056
8	66,217	148,858	408,591	44,251
9	100,429	243,508	520,619	58,974
10	147,008	379,854	672,647	82,217
11	212,292	582,096	858,649	113,031
12	290,451	830,530	1,046,690	138,717
13	363,593	1,119,776	1,203,492	172,085
14	448,852	1,482,912	1,500,018	220,231
15	603,008	1,985,549	1,750,726	260,392
16	721,331	2,464,521	2,146,289	343,795
17	913,974	3,067,099	2,500,579	419,751

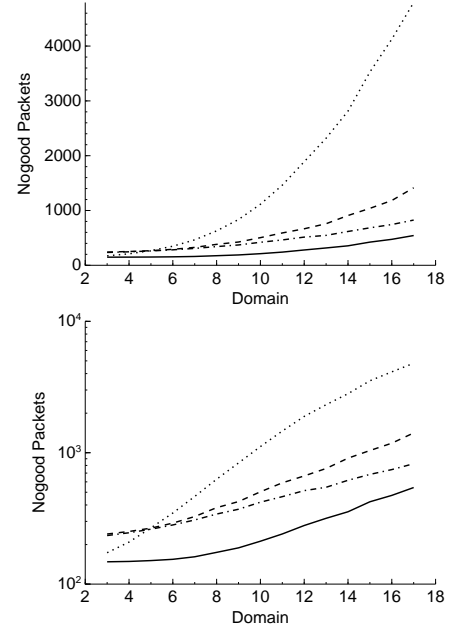
Nonconcurrent Network Traffic (Bytes)				
Domain	SBDS	AWCS	ABTDO	ABT
3	912	845	6,895	459
4	1,350	1,812	10,721	768
5	1,989	4,160	16,130	1,281
6	3,065	9,022	22,554	1,945
7	4,577	18,238	31,306	3,065
8	6,960	34,272	42,293	4,616
9	10,177	58,953	54,255	6,198
10	14,643	95,260	70,272	8,815
11	20,559	150,301	90,593	12,171
12	27,643	218,634	109,701	14,789
13	34,111	299,761	125,114	18,629
14	41,683	399,975	155,587	23,674
15	55,338	543,269	183,835	28,040
16	66,182	674,983	223,122	36,839
17	82,104	849,248	263,383	46,251



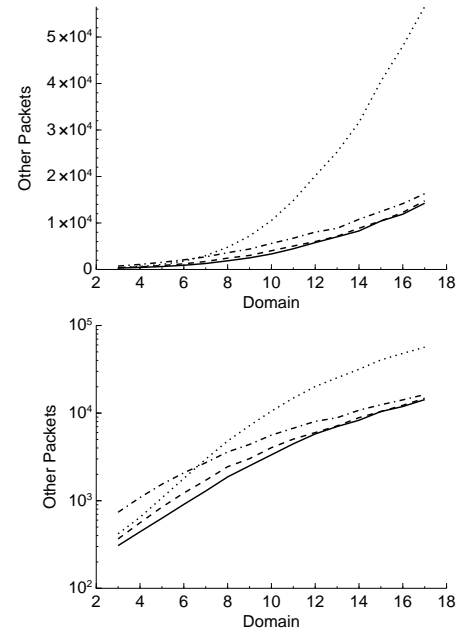
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.10: Number of packets for **all instances** in **problem set 1**.

Nogood Packets				
Domain	SBDS	AWCS	ABTDO	ABT
3	148	173	234	241
4	149	209	246	252
5	151	268	262	267
6	154	348	282	291
7	161	465	308	327
8	174	628	341	382
9	189	835	371	425
10	212	1,114	420	504
11	241	1,463	463	590
12	279	1,889	515	666
13	317	2,319	545	759
14	355	2,813	617	908
15	423	3,526	683	1,039
16	473	4,125	744	1,182
17	543	4,795	825	1,415



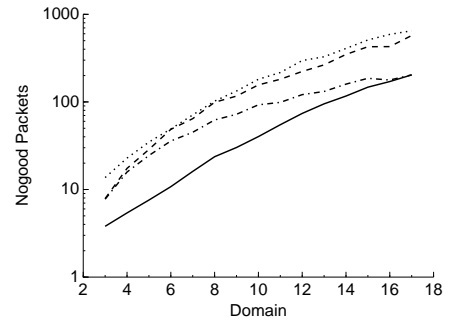
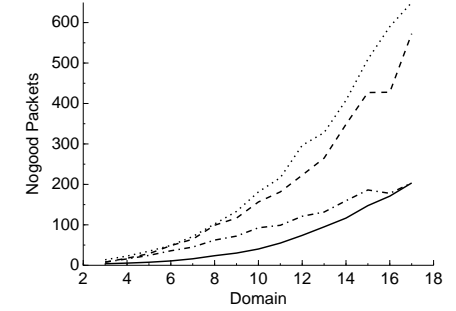
Other Packets				
Domain	SBDS	AWCS	ABTDO	ABT
3	307	419	739	367
4	442	647	1,088	560
5	630	1,079	1,542	846
6	908	1,796	2,078	1,224
7	1,290	2,981	2,718	1,711
8	1,867	4,800	3,606	2,442
9	2,502	7,190	4,389	3,017
10	3,341	10,537	5,608	4,031
11	4,458	14,837	6,722	5,049
12	5,779	20,125	8,044	5,996
13	7,022	25,271	8,852	7,143
14	8,290	31,654	10,793	8,875
15	10,419	40,470	12,465	10,430
16	11,889	48,042	14,167	12,279
17	14,254	56,521	16,292	14,774



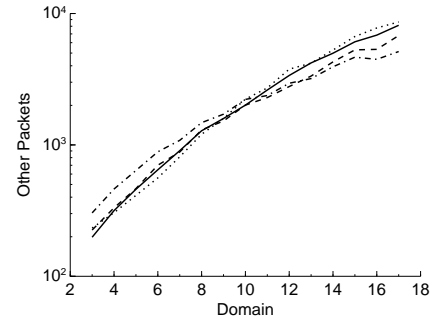
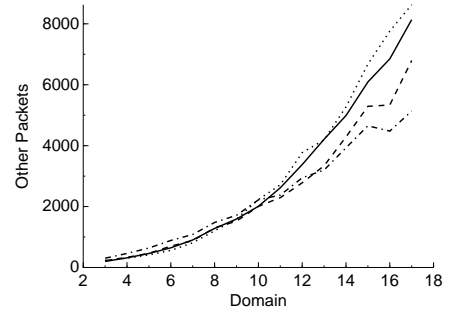
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.11: Number of packets for **feasible instances** in **problem set 1**.

Nogood Packets				
Domain	SBDS	AWCS	ABTDO	ABT
3	4	14	8	8
4	5	23	16	17
5	8	35	24	29
6	11	49	36	49
7	16	71	45	64
8	24	102	62	99
9	30	133	72	117
10	40	181	93	157
11	55	217	99	181
12	74	297	121	223
13	95	328	132	265
14	117	408	160	348
15	147	509	186	427
16	171	590	178	428
17	204	650	204	573



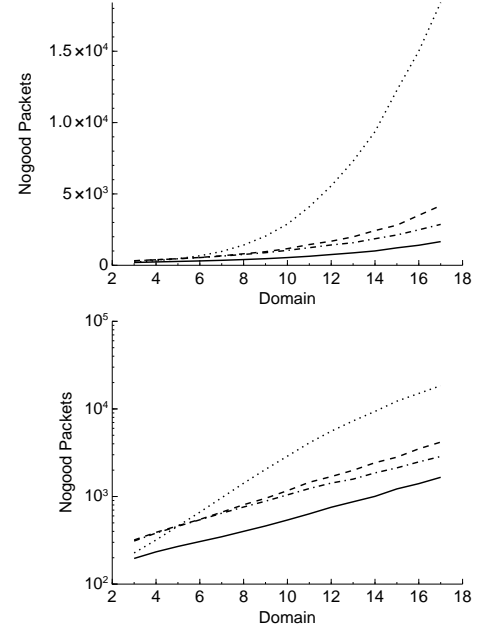
Other Packets				
Domain	SBDS	AWCS	ABTDO	ABT
3	198	233	304	225
4	316	305	462	332
5	460	413	638	465
6	647	561	886	701
7	903	811	1,083	883
8	1,282	1,205	1,477	1,299
9	1,589	1,592	1,714	1,521
10	2,009	2,230	2,220	2,008
11	2,617	2,715	2,376	2,293
12	3,378	3,771	2,945	2,780
13	4,209	4,189	3,200	3,340
14	4,988	5,274	3,930	4,286
15	6,092	6,678	4,652	5,291
16	6,849	7,762	4,478	5,337
17	8,138	8,628	5,136	6,794



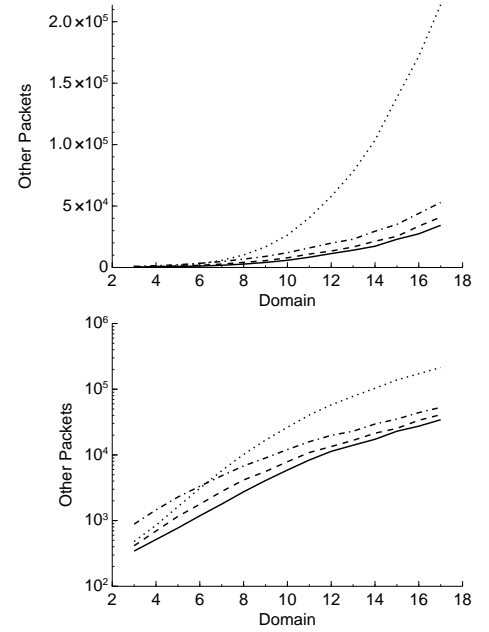
———— SBDS AWCS - · - · - ABTDO - - - - ABT

Figure A.12: Number of packets for **infeasible instances** in **problem set 1**.

Nogood Packets				
Domain	SBDS	AWCS	ABTDO	ABT
3	196	227	310	319
4	233	319	382	390
5	269	460	458	463
6	306	663	542	547
7	347	969	644	663
8	400	1,417	759	806
9	461	2,041	885	954
10	539	2,888	1,043	1,166
11	634	4,098	1,233	1,454
12	754	5,566	1,423	1,688
13	871	7,295	1,580	1,992
14	1,005	9,363	1,861	2,433
15	1,221	12,265	2,124	2,811
16	1,403	15,006	2,486	3,503
17	1,658	18,413	2,866	4,181



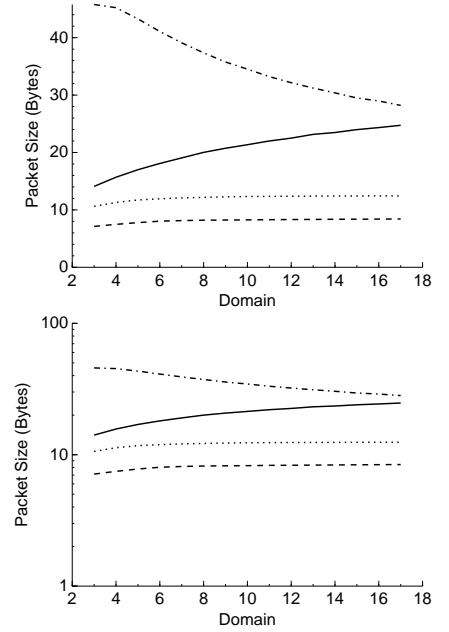
Other Packets				
Domain	SBDS	AWCS	ABTDO	ABT
3	344	482	885	415
4	516	849	1,458	695
5	771	1,629	2,286	1,160
6	1,183	3,099	3,336	1,777
7	1,784	5,760	4,812	2,772
8	2,744	10,187	6,795	4,155
9	4,069	16,797	8,981	5,584
10	5,876	26,338	12,051	7,878
11	8,350	40,475	15,914	10,879
12	11,323	57,878	19,817	13,418
13	14,052	77,955	22,977	16,647
14	17,281	103,510	29,487	21,376
15	22,943	138,347	35,094	25,316
16	27,383	172,004	43,988	33,643
17	34,320	213,864	52,943	40,992



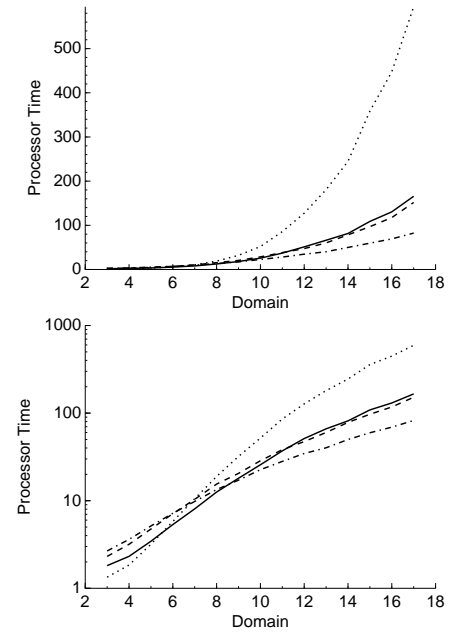
———— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.13: Average packet size and CPU time for **all instances in problem set 1**.

Packet Size (Bytes)				
Domain	SBDS	AWCS	ABTDO	ABT
3	14	11	46	7
4	16	11	45	7
5	17	12	43	8
6	18	12	41	8
7	19	12	39	8
8	20	12	37	8
9	21	12	36	8
10	21	12	35	8
11	22	12	33	8
12	23	12	32	8
13	23	12	31	8
14	23	12	30	8
15	24	12	29	8
16	24	12	29	8
17	25	12	28	8



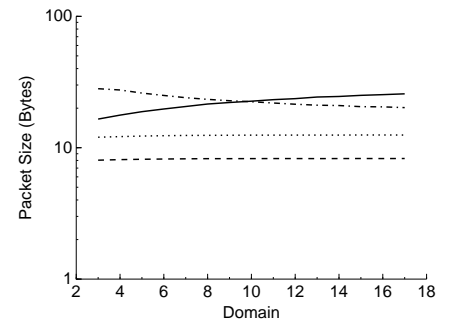
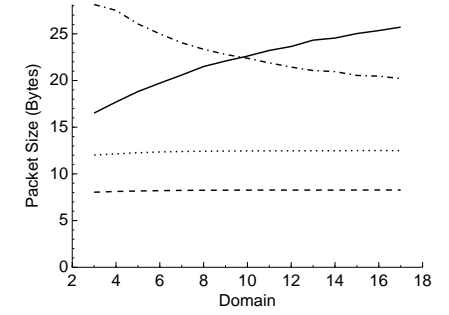
Processor Time				
Domain	SBDS	AWCS	ABTDO	ABT
3	2	1	3	2
4	2	2	4	3
5	3	3	5	5
6	5	6	7	7
7	8	11	10	10
8	13	19	13	15
9	18	32	17	21
10	26	52	23	29
11	37	86	28	38
12	51	127	35	47
13	66	181	40	60
14	82	245	50	78
15	109	358	60	97
16	130	447	69	117
17	166	595	82	152



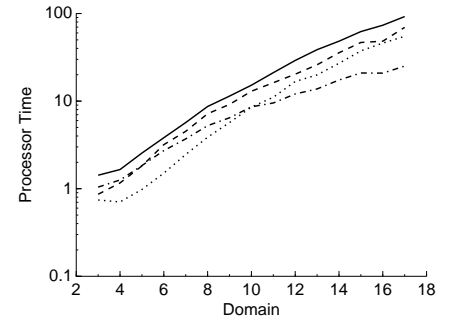
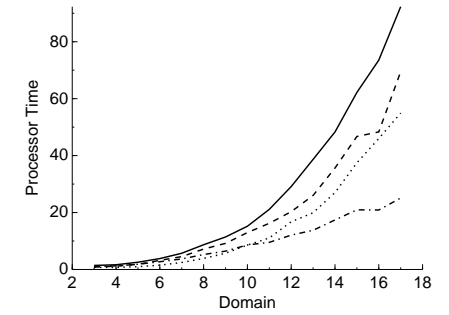
SBDS
 AWCS
 ABTDO
 ABT

Figure A.14: Average packet size and CPU time for **feasible instances** in **problem set 1**.

Packet Size (Bytes)				
Domain	SBDS	AWCS	ABTDO	ABT
3	17	12	28	8
4	18	12	28	8
5	19	12	26	8
6	20	12	25	8
7	21	12	24	8
8	21	12	23	8
9	22	12	23	8
10	23	12	22	8
11	23	12	22	8
12	24	12	21	8
13	24	12	21	8
14	25	12	21	8
15	25	12	21	8
16	25	12	20	8
17	26	12	20	8



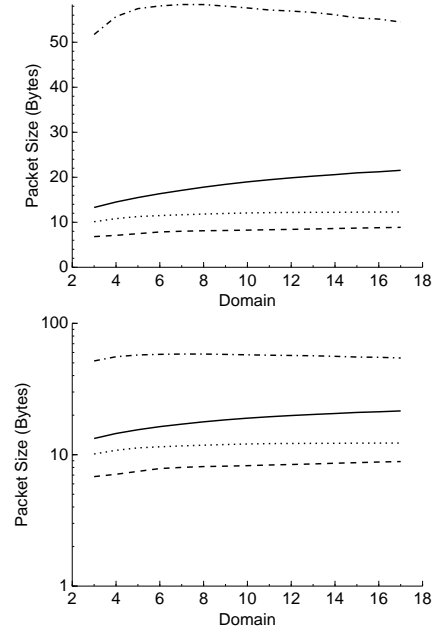
Processor Time				
Domain	SBDS	AWCS	ABTDO	ABT
3	1	1	1	1
4	2	1	1	1
5	3	1	2	2
6	4	2	3	3
7	6	2	4	5
8	9	4	5	7
9	11	6	6	9
10	15	9	9	13
11	21	11	10	16
12	29	17	12	20
13	39	20	14	26
14	48	27	17	36
15	62	37	21	47
16	74	46	21	48
17	92	55	25	69



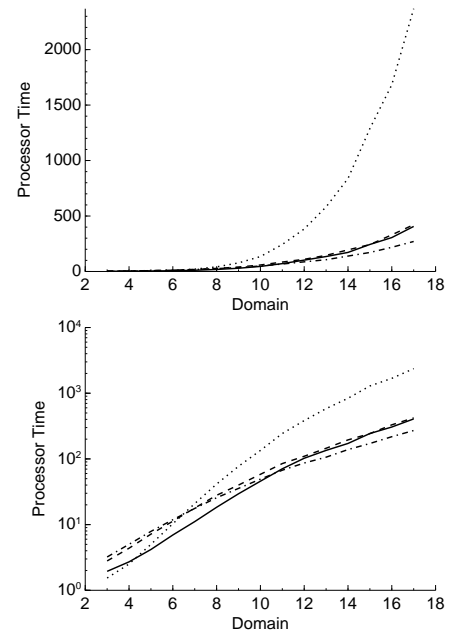
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.15: Average packet size and CPU time for **infeasible instances** in **problem set 1**.

Packet Size (Bytes)				
Domain	SBDS	AWCS	ABTDO	ABT
3	13	10	52	7
4	15	11	56	7
5	15	11	57	7
6	16	11	58	8
7	17	12	58	8
8	18	12	58	8
9	18	12	58	8
10	19	12	58	8
11	19	12	57	8
12	20	12	57	8
13	20	12	57	9
14	21	12	56	9
15	21	12	55	9
16	21	12	55	9
17	22	12	54	9



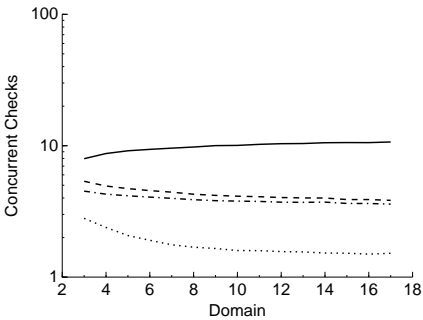
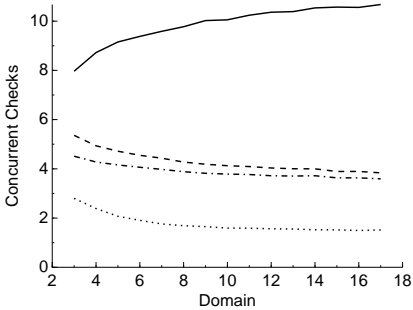
Processor Time				
Domain	SBDS	AWCS	ABTDO	ABT
3	2	2	3	3
4	3	3	5	4
5	4	5	8	7
6	7	10	12	11
7	11	21	18	18
8	18	41	26	28
9	29	77	36	40
10	46	134	49	58
11	71	244	67	85
12	102	383	87	110
13	135	582	106	145
14	172	838	139	195
15	244	1,288	172	243
16	306	1,683	218	330
17	406	2,369	271	422



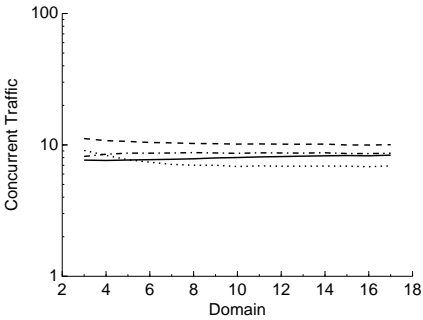
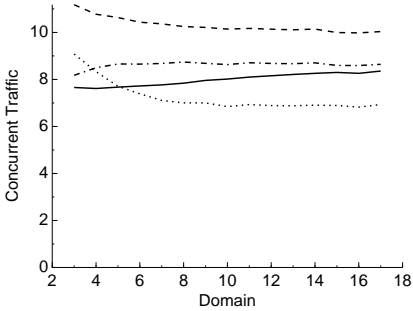
———— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.16: Other measures of concurrency for **all instances in problem set 1**.

Concurrent Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	8	3	5	5
4	9	2	4	5
5	9	2	4	5
6	9	2	4	5
7	10	2	4	4
8	10	2	4	4
9	10	2	4	4
10	10	2	4	4
11	10	2	4	4
12	10	2	4	4
13	10	2	4	4
14	11	2	4	4
15	11	2	4	4
16	11	1	4	4
17	11	2	4	4



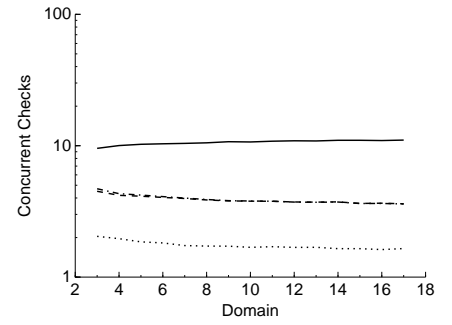
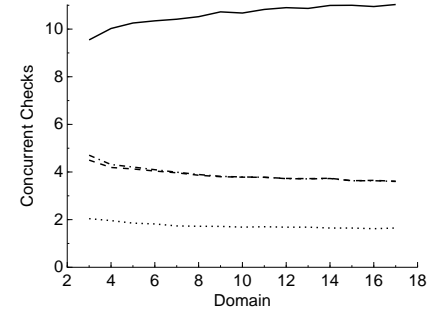
Concurrent Traffic				
Domain	SBDS	AWCS	ABTDO	ABT
3	8	9	8	11
4	8	8	8	11
5	8	8	9	11
6	8	7	9	10
7	8	7	9	10
8	8	7	9	10
9	8	7	9	10
10	8	7	9	10
11	8	7	9	10
12	8	7	9	10
13	8	7	9	10
14	8	7	9	10
15	8	7	9	10
16	8	7	9	10
17	8	7	9	10



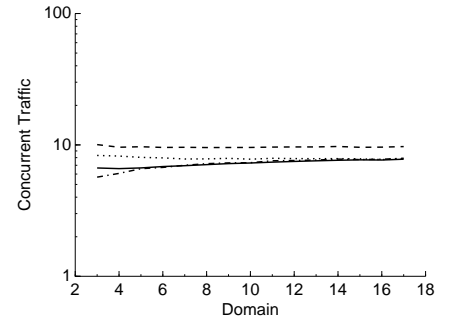
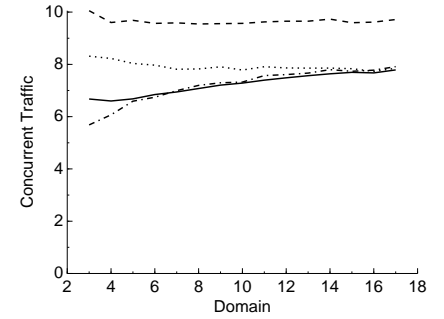
———— SBDS AWCS - . - . - . ABTDO - - - - - ABT

Figure A.17: Other measures of concurrency for **feasible instances** in **problem set 1**.

Concurrent Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	10	2	5	4
4	10	2	4	4
5	10	2	4	4
6	10	2	4	4
7	10	2	4	4
8	11	2	4	4
9	11	2	4	4
10	11	2	4	4
11	11	2	4	4
12	11	2	4	4
13	11	2	4	4
14	11	2	4	4
15	11	2	4	4
16	11	2	4	4
17	11	2	4	4



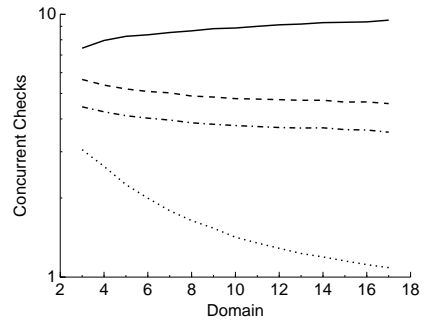
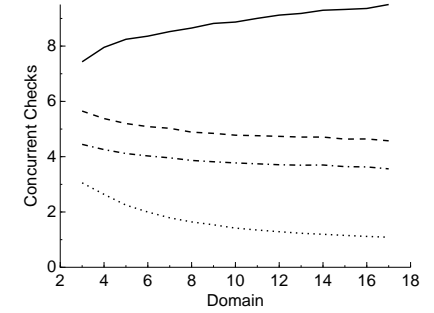
Concurrent Traffic				
Domain	SBDS	AWCS	ABTDO	ABT
3	7	8	6	10
4	7	8	6	10
5	7	8	7	10
6	7	8	7	10
7	7	8	7	10
8	7	8	7	10
9	7	8	7	10
10	7	8	7	10
11	7	8	8	10
12	7	8	8	10
13	8	8	8	10
14	8	8	8	10
15	8	8	8	10
16	8	8	8	10
17	8	8	8	10



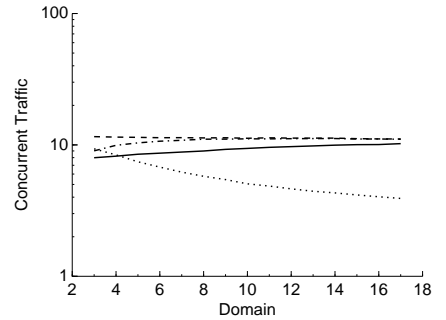
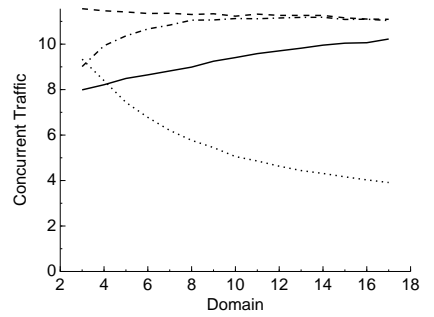
———— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.18: Other measures of concurrency for **infeasible instances** in **problem set 1**.

Concurrent Checks				
Domain	SBDS	AWCS	ABTDO	ABT
3	7	3	4	6
4	8	3	4	5
5	8	2	4	5
6	8	2	4	5
7	9	2	4	5
8	9	2	4	5
9	9	2	4	5
10	9	1	4	5
11	9	1	4	5
12	9	1	4	5
13	9	1	4	5
14	9	1	4	5
15	9	1	4	5
16	9	1	4	5
17	10	1	4	5



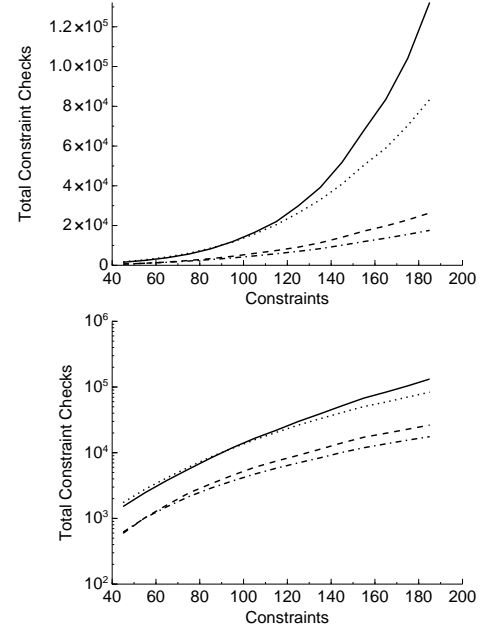
Concurrent Traffic				
Domain	SBDS	AWCS	ABTDO	ABT
3	8	9	9	12
4	8	8	10	11
5	8	7	10	11
6	9	7	11	11
7	9	6	11	11
8	9	6	11	11
9	9	5	11	11
10	9	5	11	11
11	10	5	11	11
12	10	5	11	11
13	10	4	11	11
14	10	4	11	11
15	10	4	11	11
16	10	4	11	11
17	10	4	11	11



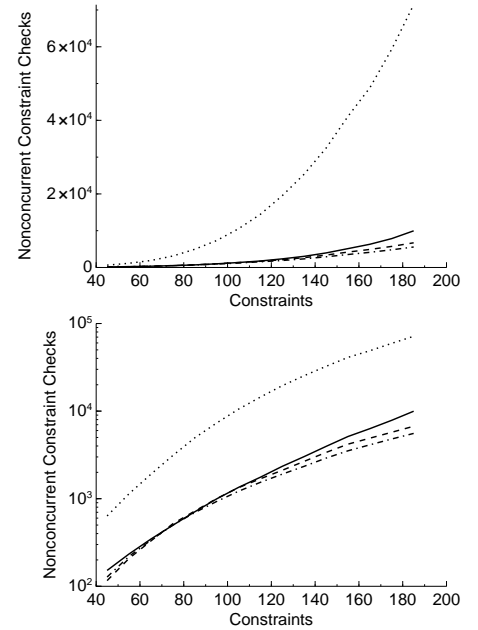
———— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.19: Number of constraint checks for **all instances** in **problem set 2**.

Constraints	SBDS	AWCS	ABTDO	ABT
45	1,519	1,741	619	593
55	2,463	2,776	1,017	1,025
65	3,787	4,148	1,506	1,599
75	5,633	6,036	2,145	2,433
85	8,296	8,632	2,882	3,352
95	11,920	11,736	3,722	4,539
105	16,539	15,712	4,720	5,984
115	22,023	20,439	5,819	7,454
125	29,884	26,317	7,010	9,138
135	39,122	32,959	8,361	11,323
145	51,843	40,897	10,103	14,107
155	67,934	50,431	11,933	17,219
165	83,520	58,990	13,639	19,798
175	104,180	70,360	15,535	22,820
185	132,228	83,371	17,578	26,317



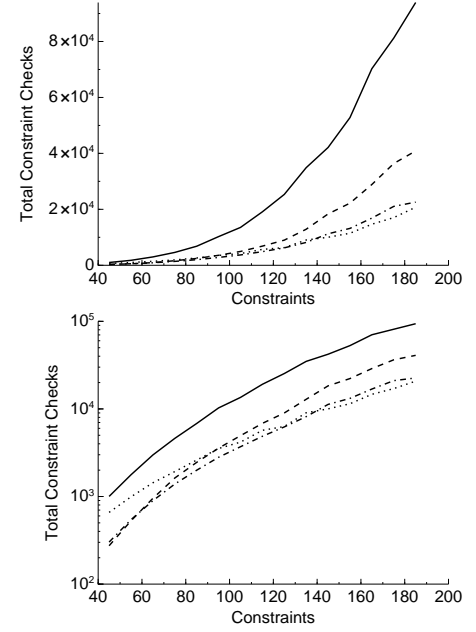
Constraints	SBDS	AWCS	ABTDO	ABT
45	152	636	128	116
55	234	1,138	220	208
65	346	1,904	336	330
75	497	3,111	499	517
85	709	4,950	694	723
95	994	7,315	930	1,002
105	1,347	10,524	1,216	1,337
115	1,759	14,455	1,538	1,688
125	2,344	19,528	1,908	2,113
135	3,027	25,367	2,344	2,669
145	3,956	32,446	2,908	3,375
155	5,131	41,077	3,523	4,194
165	6,305	48,866	4,123	4,906
175	7,858	59,360	4,805	5,738
185	9,968	71,434	5,561	6,716



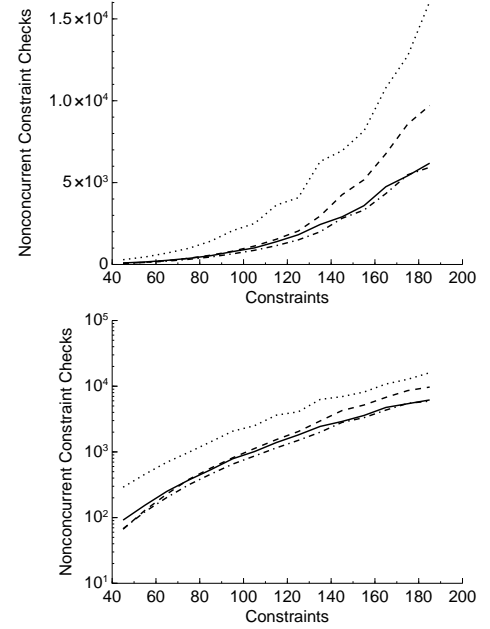
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.20: Number of constraint checks for **feasible instances** in **problem set 2**.

Constraints	SBDS	AWCS	ABTDO	ABT
45	1,002	659	302	274
55	1,770	991	544	528
65	2,981	1,435	896	954
75	4,611	1,927	1,387	1,626
85	6,822	2,608	1,996	2,454
95	10,264	3,530	2,805	3,566
105	13,538	4,178	3,701	4,949
115	19,110	5,697	4,869	6,842
125	25,333	6,297	6,234	8,995
135	34,904	9,039	8,170	12,879
145	42,114	10,019	11,260	18,347
155	52,793	11,456	13,195	22,218
165	70,288	14,682	16,865	28,875
175	81,224	17,055	21,048	36,380
185	93,932	20,755	22,584	40,886



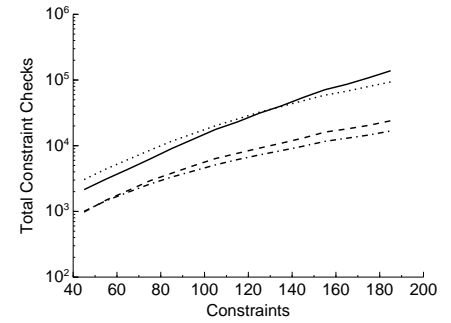
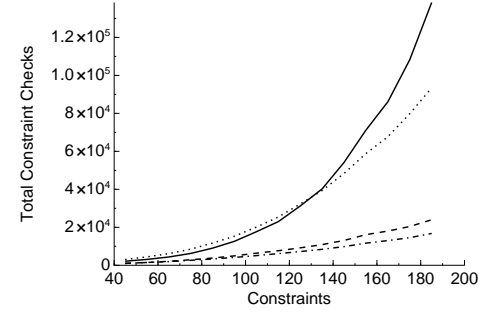
Constraints	SBDS	AWCS	ABTDO	ABT
45	92	290	68	67
55	155	459	123	129
65	251	697	203	228
75	374	987	316	382
85	535	1,425	458	566
95	786	2,064	651	812
105	1,012	2,505	866	1,127
115	1,389	3,629	1,153	1,541
125	1,814	4,078	1,501	2,043
135	2,443	6,290	1,996	2,962
145	2,910	6,947	2,819	4,259
155	3,591	8,155	3,323	5,172
165	4,748	10,787	4,330	6,764
175	5,418	12,752	5,482	8,578
185	6,184	16,048	5,935	9,709



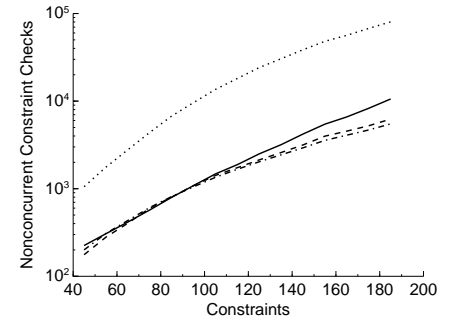
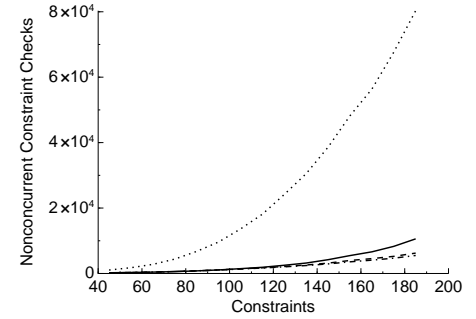
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.21: Number of constraint checks for **infeasible instances** in **problem set 2**.

Total Constraint Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	2,146	3,053	1,003	979
55	3,102	4,423	1,455	1,483
65	4,379	6,143	1,954	2,073
75	6,251	8,524	2,604	2,922
85	9,051	11,720	3,336	3,813
95	12,649	15,347	4,126	4,967
105	17,690	20,138	5,111	6,381
115	23,014	25,449	6,142	7,661
125	31,249	32,319	7,243	9,181
135	40,261	39,418	8,413	10,903
145	54,194	48,359	9,823	13,082
155	71,207	58,855	11,660	16,139
165	86,121	67,698	13,005	18,014
175	108,262	79,843	14,554	20,408
185	138,371	93,415	16,775	23,980



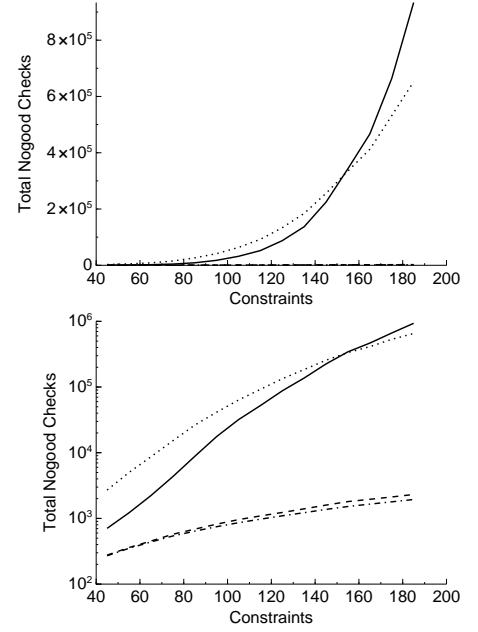
Nonconcurrent Constraint Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	224	1,054	200	176
55	307	1,764	309	280
65	416	2,791	434	404
75	572	4,396	610	599
85	798	6,757	815	803
95	1,086	9,626	1,052	1,085
105	1,475	13,601	1,350	1,417
115	1,884	18,135	1,669	1,738
125	2,503	24,161	2,030	2,134
135	3,184	30,519	2,438	2,590
145	4,209	38,608	2,929	3,161
155	5,463	48,194	3,566	3,983
165	6,611	56,351	4,082	4,541
175	8,292	67,651	4,684	5,232
185	10,575	80,319	5,501	6,236



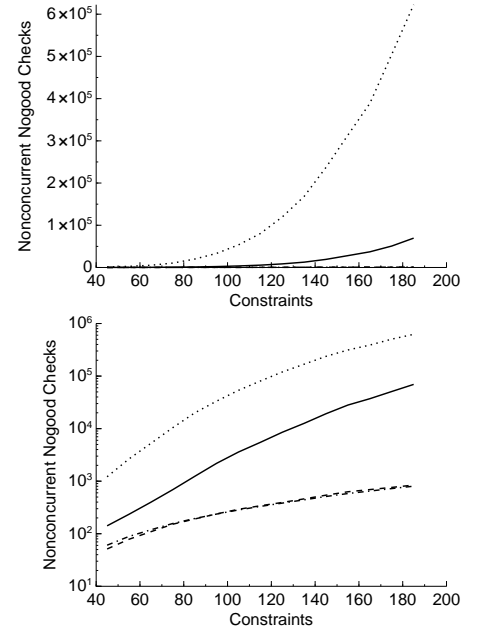
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.22: Number of nogood checks for **all instances** in **problem set 2**.

Total Nogood Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	706	2,698	270	276
55	1,218	5,013	351	360
65	2,223	8,656	437	452
75	4,326	15,169	539	573
85	8,778	26,308	640	686
95	17,585	41,347	747	816
105	31,855	63,364	859	959
115	52,202	92,184	973	1,090
125	87,563	133,582	1,094	1,231
135	137,147	184,456	1,223	1,400
145	224,706	256,023	1,371	1,596
155	343,608	335,308	1,519	1,804
165	467,001	412,333	1,651	1,954
175	664,203	531,735	1,789	2,120
185	933,871	652,475	1,940	2,311



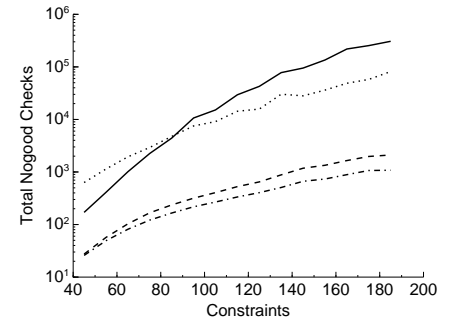
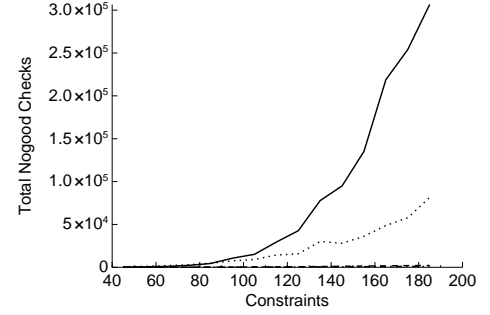
Nonconcurrent Nogood Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	141	1,214	61	51
55	235	2,655	89	79
65	399	5,246	119	109
75	690	10,436	157	152
85	1,237	19,915	195	191
95	2,185	33,375	237	238
105	3,593	53,590	283	292
115	5,489	80,496	331	341
125	8,534	119,750	383	394
135	12,614	168,443	442	461
145	19,252	237,402	512	542
155	28,141	313,884	582	629
165	37,132	388,796	650	692
175	50,831	504,900	723	762
185	69,486	622,463	803	844



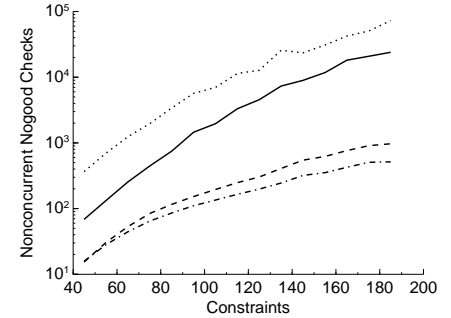
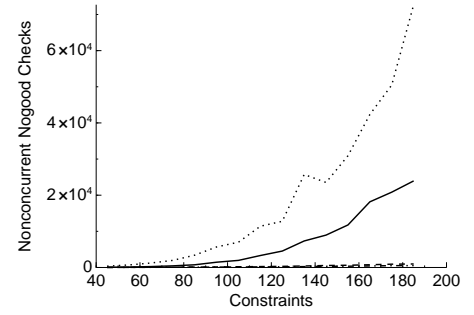
—— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.23: Number of nogood checks for **feasible instances** in **problem set 2**.

Constraints	SBDS	AWCS	ABTDO	ABT
45	171	631	26	27
55	411	1,141	49	57
65	1,013	1,943	81	103
75	2,258	2,931	122	167
85	4,410	4,751	165	236
95	10,650	7,515	218	318
105	15,160	9,096	268	406
115	29,474	14,309	335	526
125	42,643	15,756	404	647
135	77,801	30,186	507	881
145	94,898	28,105	665	1,179
155	135,233	36,164	735	1,343
165	218,856	48,731	890	1,649
175	254,098	57,777	1,068	1,974
185	306,586	81,507	1,079	2,107



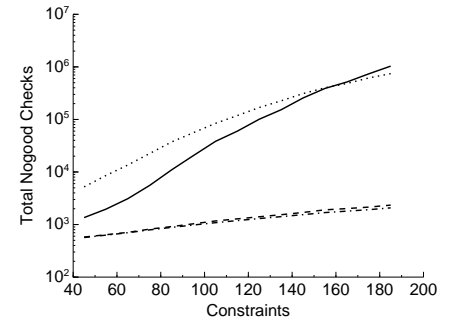
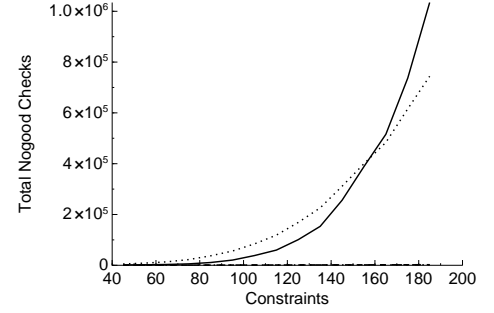
Constraints	SBDS	AWCS	ABTDO	ABT
45	69	366	16	15
55	133	699	28	31
65	254	1,240	44	53
75	446	1,970	65	84
85	748	3,405	86	116
95	1,455	5,677	111	153
105	1,959	6,999	135	195
115	3,310	11,411	165	248
125	4,550	12,778	198	303
135	7,336	25,727	245	409
145	8,962	23,516	320	547
155	11,762	30,928	353	622
165	18,165	42,341	426	763
175	20,841	50,492	507	907
185	23,950	72,705	514	970



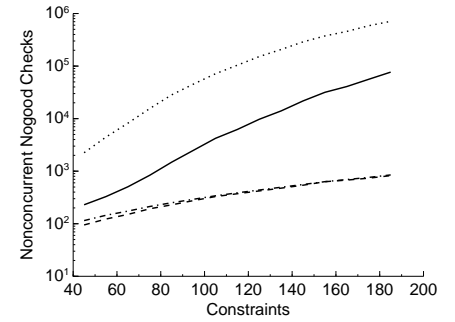
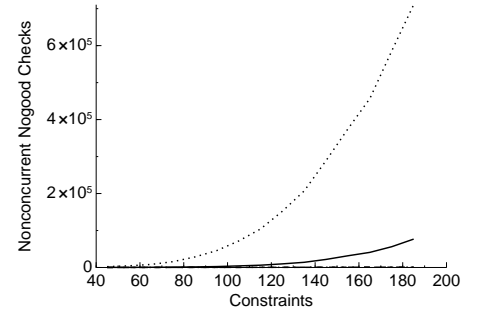
———— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.24: Number of nogood checks for **infeasible instances in problem set 2**.

Total Nogood Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	1,356	5,205	566	578
55	1,964	8,589	629	640
65	3,112	13,592	699	709
75	5,579	22,580	792	818
85	11,017	37,360	884	916
95	20,637	56,237	981	1,035
105	38,259	84,189	1,086	1,171
115	59,927	118,651	1,190	1,282
125	101,032	168,910	1,301	1,406
135	153,173	226,115	1,416	1,540
145	256,077	311,104	1,542	1,697
155	388,652	399,973	1,688	1,903
165	515,775	483,802	1,800	2,014
175	737,129	616,056	1,918	2,146
185	1,034,491	744,060	2,079	2,344



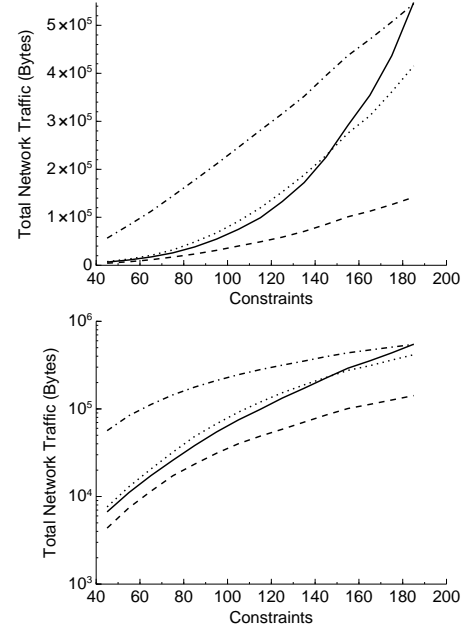
Nonconcurrent Nogood Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	230	2,244	115	95
55	329	4,461	145	123
65	505	8,192	174	151
75	838	15,563	213	193
85	1,487	28,379	251	229
95	2,506	45,565	293	276
105	4,219	71,468	340	329
115	6,229	103,976	388	372
125	9,729	151,824	438	421
135	14,039	206,982	495	475
145	21,739	289,093	558	541
155	31,681	375,049	632	630
165	40,860	456,894	694	678
175	56,164	585,742	762	737
185	76,791	710,647	850	823



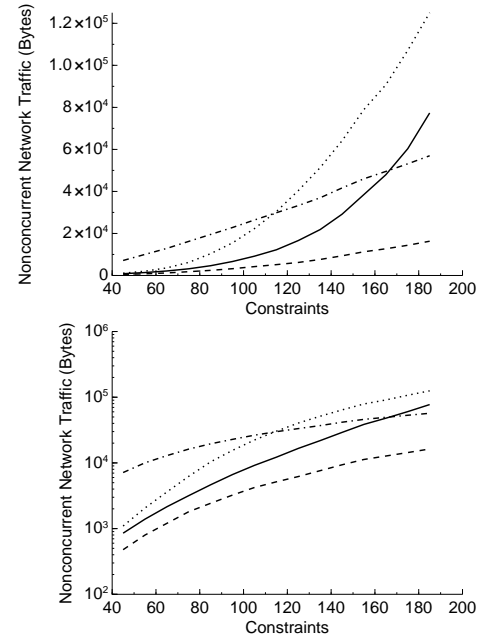
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.25: Network traffic for **all instances** in **problem set 2**.

Total Network Traffic (Bytes)				
Constraints	SBDS	AWCS	ABTDO	ABT
45	6,679	7,607	56,469	4,360
55	11,110	12,950	83,858	7,447
65	17,326	20,599	112,311	11,490
75	25,918	32,035	144,650	17,239
85	38,219	48,568	177,551	23,428
95	54,601	67,733	211,204	31,187
105	75,037	91,911	246,016	40,110
115	99,031	119,801	281,368	48,937
125	132,931	152,907	315,910	58,602
135	172,338	188,428	352,307	70,686
145	225,357	228,815	395,539	85,220
155	291,772	275,279	437,104	101,130
165	354,235	311,592	470,994	112,694
175	436,672	361,413	507,173	126,498
185	547,581	415,424	544,716	141,880



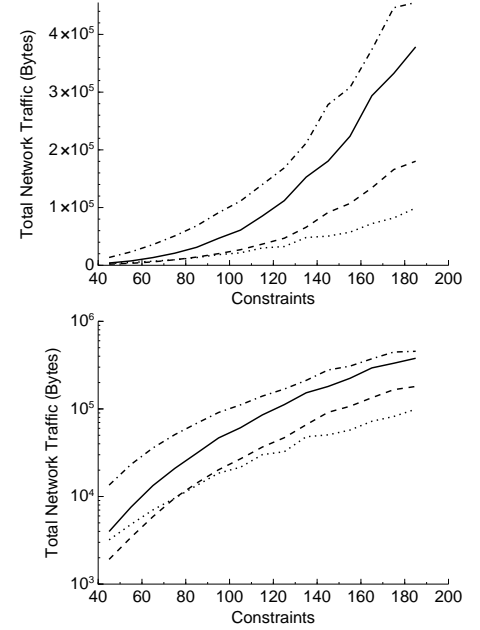
Nonconcurrent Network Traffic (Bytes)				
Constraints	SBDS	AWCS	ABTDO	ABT
45	852	1,095	7,132	478
55	1,394	2,059	10,044	798
65	2,150	3,613	12,916	1,203
75	3,177	6,256	16,242	1,811
85	4,647	10,395	19,535	2,432
95	6,662	15,536	22,845	3,242
105	9,189	22,338	26,407	4,228
115	12,220	30,419	29,864	5,150
125	16,629	40,377	33,275	6,201
135	21,864	51,334	36,981	7,613
145	29,157	64,147	41,658	9,356
155	38,559	79,002	46,026	11,289
165	47,950	90,773	49,513	12,723
175	60,339	107,156	53,131	14,370
185	77,346	125,048	56,970	16,300



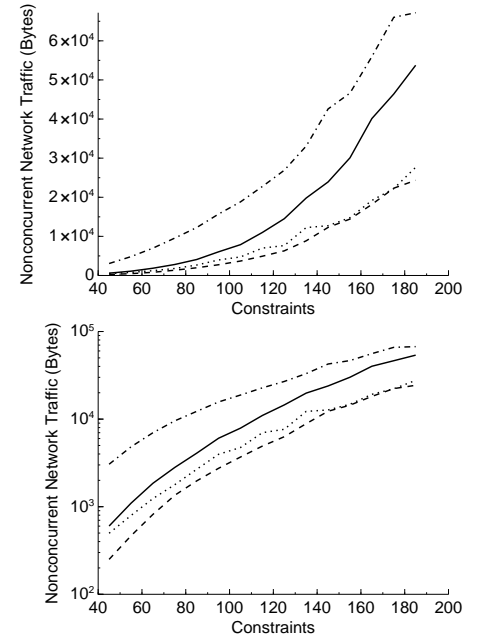
———— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.26: Network traffic for **feasible instances** in **problem set 2**.

Constraints	SBDS	AWCS	ABTDO	ABT
45	3,987	3,202	13,505	1,915
55	7,573	4,805	23,251	3,418
65	13,320	7,025	35,866	5,883
75	20,952	9,578	50,910	9,605
85	31,195	13,365	68,456	14,244
95	46,685	18,481	91,130	20,214
105	60,988	21,613	111,250	26,922
115	85,491	30,061	140,009	36,690
125	111,983	32,483	168,693	46,933
135	152,816	48,143	211,741	65,991
145	180,453	50,536	278,431	91,506
155	223,508	57,334	307,967	107,179
165	293,765	72,222	373,742	134,227
175	332,386	82,036	446,200	165,892
185	378,338	99,241	455,272	180,337



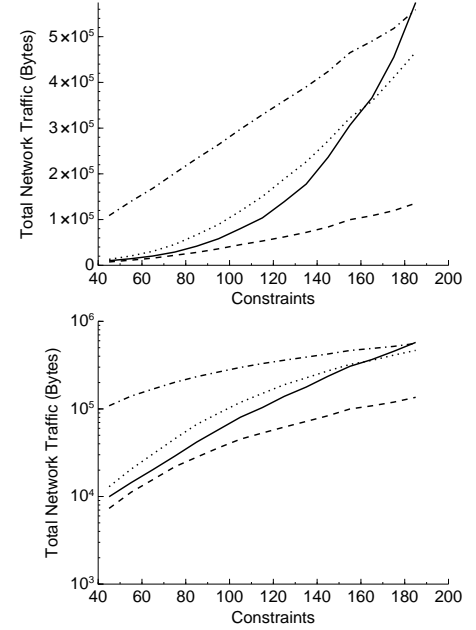
Constraints	SBDS	AWCS	ABTDO	ABT
45	602	497	3,053	250
55	1,096	794	4,824	465
65	1,850	1,237	6,989	817
75	2,809	1,785	9,559	1,351
85	4,083	2,672	12,266	1,969
95	6,055	3,965	15,712	2,744
105	7,887	4,745	18,824	3,692
115	11,002	6,994	22,762	4,920
125	14,520	7,652	26,908	6,296
135	19,803	12,248	33,090	8,870
145	23,910	12,732	42,576	12,293
155	30,077	14,774	46,629	14,432
165	40,132	19,179	55,946	18,173
175	46,404	22,209	66,007	22,318
185	53,752	27,664	67,189	24,338



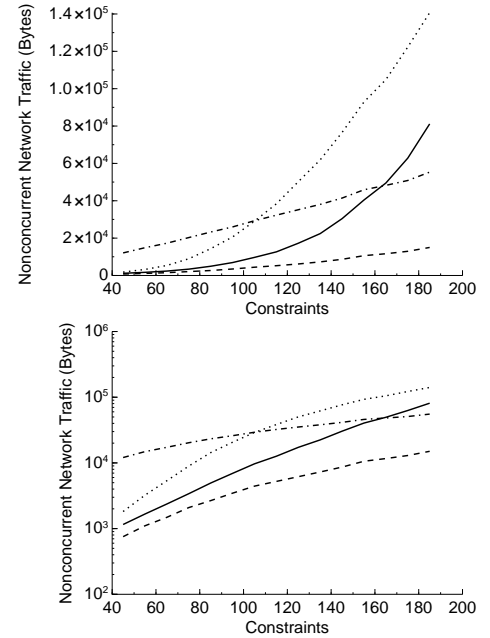
———— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.27: Network traffic for **infeasible instances** in **problem set 2**.

Constraints	SBDS	AWCS	ABTDO	ABT
45	9,944	12,952	108,594	7,326
55	14,376	20,470	139,816	11,166
65	20,272	30,580	168,518	15,613
75	28,924	45,633	201,409	21,862
85	41,819	66,614	233,477	28,135
95	58,085	89,409	264,050	36,017
105	80,426	118,887	297,731	45,171
115	103,632	150,301	329,411	53,099
125	139,212	189,015	360,051	62,101
135	177,609	226,311	390,265	71,953
145	236,208	271,900	423,841	83,701
155	306,528	322,392	465,019	99,822
165	366,121	358,642	490,110	108,462
175	455,216	411,116	518,021	119,490
185	574,728	466,141	559,063	135,711



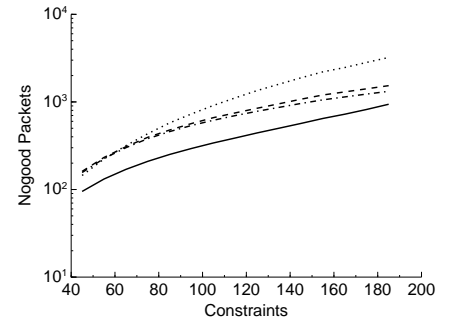
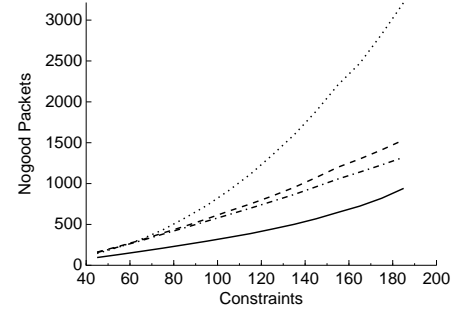
Constraints	SBDS	AWCS	ABTDO	ABT
45	1,155	1,821	12,081	754
55	1,669	3,228	14,864	1,105
65	2,370	5,361	17,274	1,487
75	3,400	8,964	20,289	2,089
85	4,936	14,354	23,261	2,669
95	6,930	20,629	25,984	3,461
105	9,689	29,090	29,316	4,434
115	12,634	38,380	32,278	5,229
125	17,262	50,190	35,184	6,172
135	22,420	61,889	38,031	7,274
145	30,425	76,573	41,436	8,646
155	40,392	92,886	45,895	10,609
165	49,486	104,846	48,248	11,651
175	62,817	122,268	50,841	12,956
185	81,131	140,669	55,331	15,010



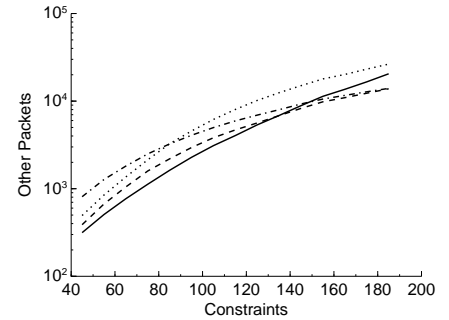
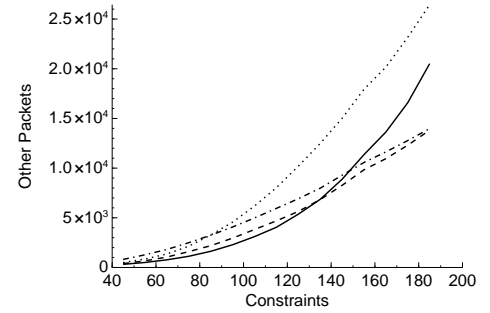
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.28: Number of packets for **all instances in problem set 2**.

Nogood Packets				
Constraints	SBDS	AWCS	ABTDO	ABT
45	95	145	156	161
55	132	223	226	232
65	170	317	300	308
75	210	433	381	394
85	252	577	459	478
95	294	731	539	567
105	340	913	618	662
115	388	1,114	698	750
125	443	1,349	782	848
135	501	1,597	867	953
145	569	1,885	961	1,074
155	647	2,208	1,056	1,200
165	725	2,473	1,140	1,301
175	821	2,829	1,229	1,413
185	941	3,215	1,322	1,534



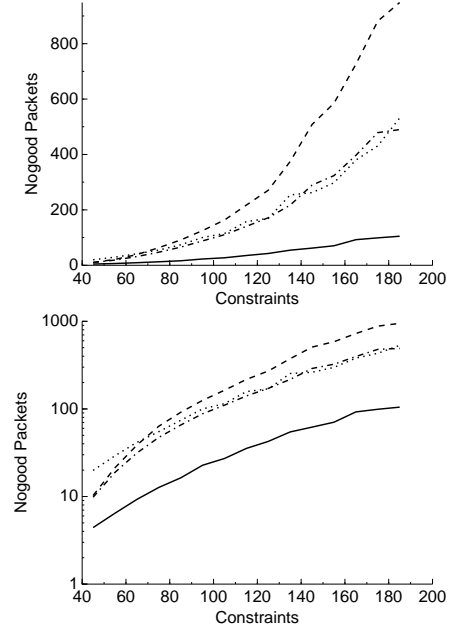
Other Packets				
Constraints	SBDS	AWCS	ABTDO	ABT
45	315	494	810	388
55	507	848	1,275	670
65	770	1,363	1,815	1,049
75	1,125	2,140	2,497	1,595
85	1,625	3,267	3,248	2,194
95	2,286	4,563	4,077	2,953
105	3,104	6,184	4,987	3,822
115	4,046	8,032	5,949	4,692
125	5,362	10,192	6,919	5,640
135	6,858	12,477	7,978	6,830
145	8,879	15,012	9,279	8,263
155	11,356	17,936	10,568	9,833
165	13,632	20,159	11,629	10,955
175	16,584	23,176	12,775	12,311
185	20,513	26,446	13,980	13,817



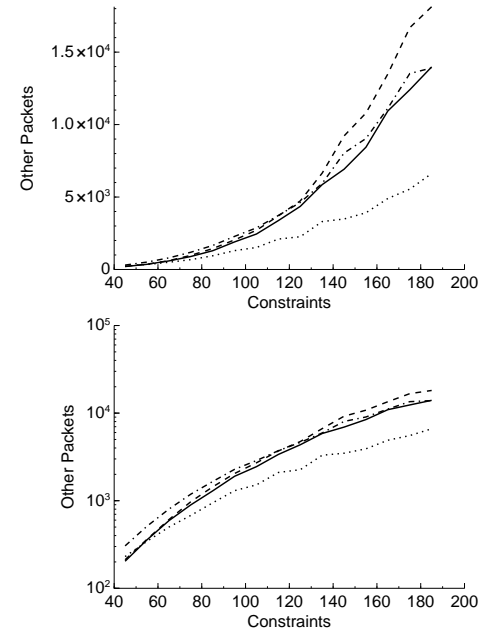
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.29: Number of packets for **feasible instances** in **problem set 2**.

Nogood Packets				
Constraints	SBDS	AWCS	ABTDO	ABT
45	4	20	10	10
55	6	29	19	21
65	9	42	31	39
75	13	55	47	63
85	16	74	66	91
95	23	100	88	125
105	27	115	110	163
115	35	158	141	217
125	43	169	172	271
135	55	254	217	375
145	62	262	290	508
155	71	298	324	585
165	92	380	397	726
175	99	431	479	881
185	105	532	490	949



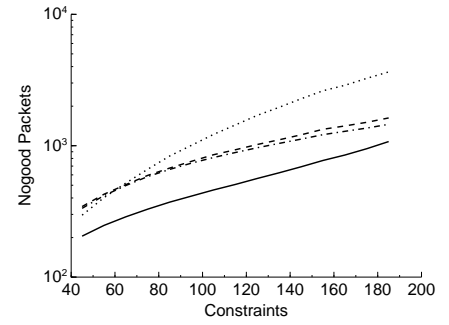
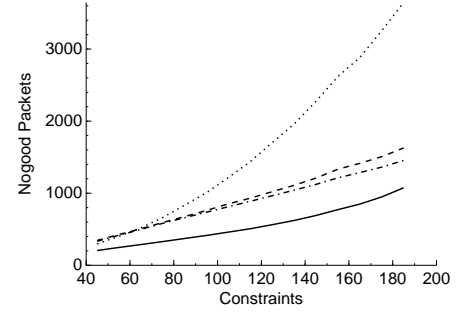
Other Packets				
Constraints	SBDS	AWCS	ABTDO	ABT
45	204	230	305	212
55	357	344	511	363
65	592	500	801	608
75	896	680	1,188	974
85	1,295	946	1,656	1,439
95	1,901	1,302	2,283	2,044
105	2,449	1,518	2,873	2,708
115	3,375	2,100	3,727	3,700
125	4,354	2,260	4,629	4,733
135	5,856	3,305	5,939	6,649
145	6,920	3,484	8,033	9,217
155	8,441	3,915	9,037	10,797
165	10,942	4,901	11,131	13,493
175	12,401	5,545	13,519	16,686
185	13,982	6,622	13,906	18,132



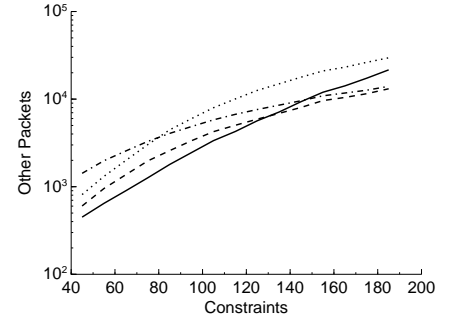
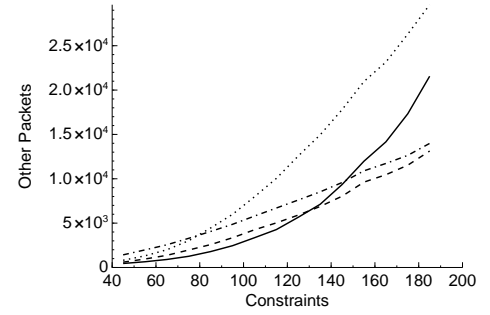
———— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.30: Number of packets for **infeasible instances** in **problem set 2**.

Nogood Packets				
Constraints	SBDS	AWCS	ABTDO	ABT
45	205	296	334	344
55	248	402	417	427
65	288	520	498	506
75	329	663	583	595
85	372	835	661	676
95	414	1,008	737	761
105	461	1,219	813	853
115	507	1,440	888	931
125	563	1,703	965	1,021
135	622	1,960	1,042	1,110
145	691	2,277	1,123	1,211
155	772	2,620	1,214	1,333
165	849	2,884	1,287	1,414
175	949	3,255	1,362	1,507
185	1,075	3,646	1,455	1,628



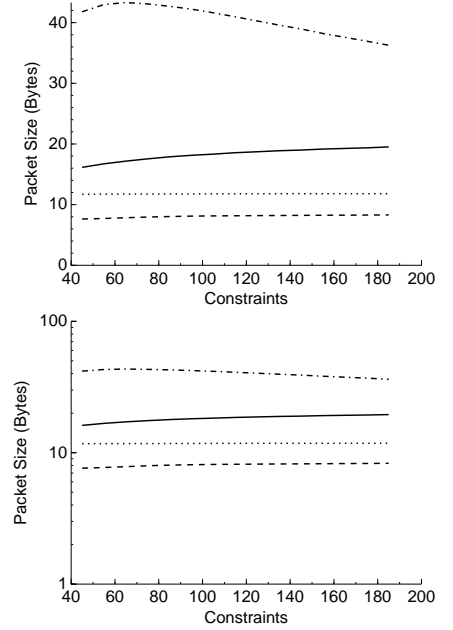
Other Packets				
Constraints	SBDS	AWCS	ABTDO	ABT
45	450	814	1,422	603
55	646	1,313	1,980	954
65	900	1,998	2,561	1,373
75	1,264	3,025	3,289	1,971
85	1,794	4,457	4,064	2,581
95	2,455	5,998	4,866	3,353
105	3,356	7,974	5,798	4,250
115	4,274	10,048	6,704	5,029
125	5,664	12,570	7,605	5,911
135	7,128	14,953	8,529	6,879
145	9,352	17,799	9,579	8,032
155	11,986	20,967	10,899	9,624
165	14,161	23,158	11,727	10,456
175	17,328	26,313	12,642	11,533
185	21,561	29,626	13,992	13,124



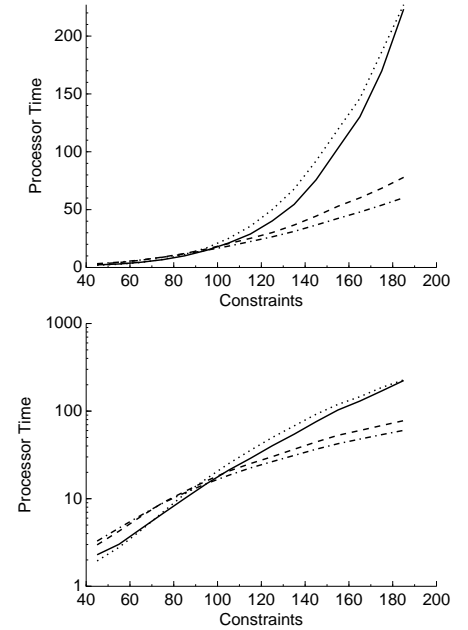
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.31: Average packet size and CPU time for **all instances in problem set 2**.

Packet Size (Bytes)				
Constraints	SBDS	AWCS	ABTDO	ABT
45	16	12	42	8
55	17	12	43	8
65	17	12	43	8
75	18	12	43	8
85	18	12	43	8
95	18	12	42	8
105	18	12	42	8
115	19	12	41	8
125	19	12	40	8
135	19	12	40	8
145	19	12	39	8
155	19	12	38	8
165	19	12	38	8
175	19	12	37	8
185	20	12	36	8



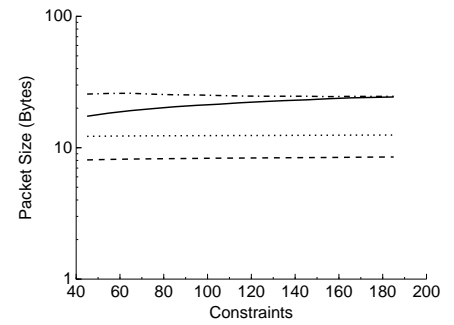
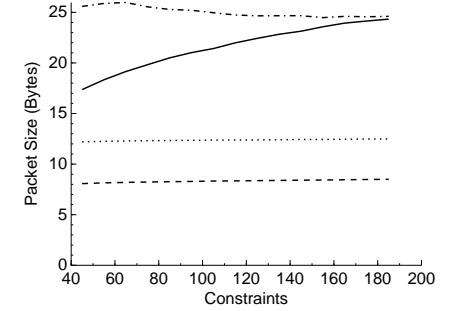
Processor Time				
Constraints	SBDS	AWCS	ABTDO	ABT
45	2	2	3	3
55	3	3	5	4
65	5	4	6	6
75	7	7	9	9
85	10	11	12	12
95	15	17	15	16
105	21	25	18	21
115	29	36	22	25
125	40	50	27	30
135	55	68	31	37
145	76	92	37	44
155	103	119	42	53
165	130	146	48	60
175	170	186	54	68
185	223	227	60	78



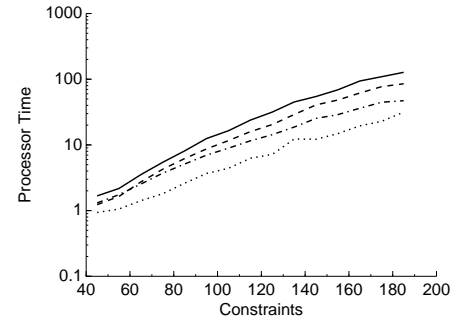
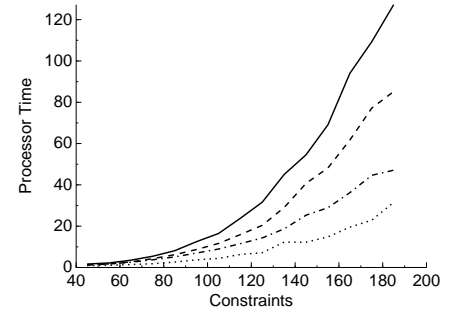
———— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.32: Average packet size and CPU time for **feasible instances** in **problem set 2**.

Packet Size (Bytes)				
Constraints	SBDS	AWCS	ABTDO	ABT
45	17	12	26	8
55	18	12	26	8
65	19	12	26	8
75	20	12	26	8
85	21	12	25	8
95	21	12	25	8
105	21	12	25	8
115	22	12	25	8
125	22	12	25	8
135	23	12	25	8
145	23	12	25	8
155	24	12	24	8
165	24	12	25	8
175	24	12	25	8
185	24	12	25	8



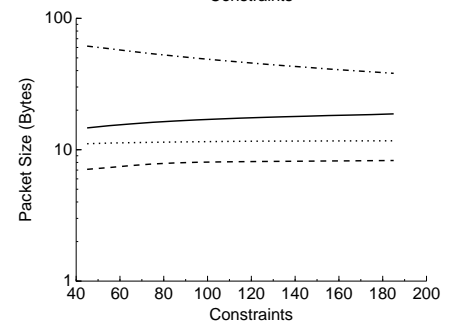
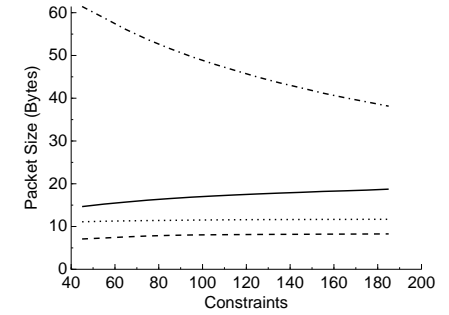
Processor Time				
Constraints	SBDS	AWCS	ABTDO	ABT
45	2	1	1	1
55	2	1	2	2
65	4	1	3	3
75	5	2	4	4
85	8	3	5	6
95	13	4	7	9
105	16	4	9	12
115	24	6	12	16
125	32	7	14	20
135	45	12	19	29
145	55	12	25	41
155	69	15	29	48
165	94	20	36	62
175	109	23	45	77
185	127	32	47	85



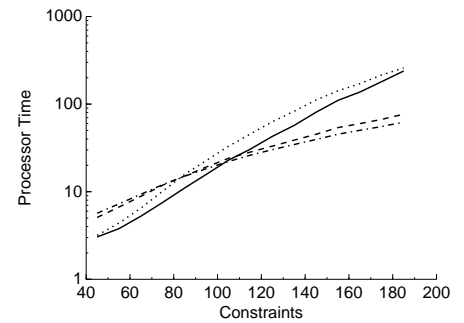
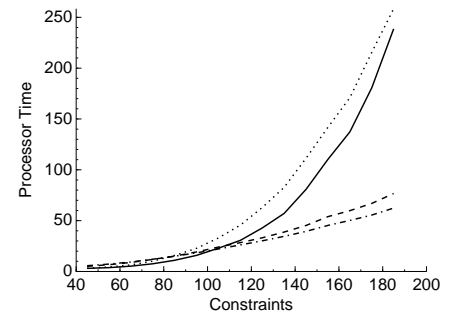
— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.33: Average packet size and CPU time for **infeasible instances** in **problem set 2**.

Packet Size (Bytes)				
Constraints	SBDS	AWCS	ABTDO	ABT
45	15	11	61	7
55	15	11	59	7
65	16	11	56	8
75	16	11	54	8
85	17	11	52	8
95	17	11	50	8
105	17	12	48	8
115	17	12	46	8
125	18	12	45	8
135	18	12	44	8
145	18	12	42	8
155	18	12	41	8
165	18	12	40	8
175	19	12	39	8
185	19	12	38	8



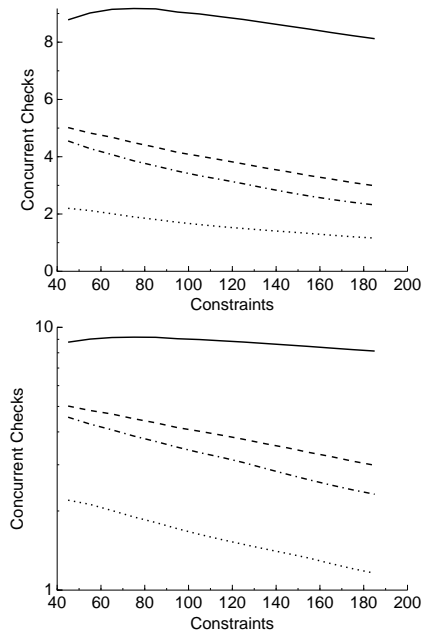
Processor Time				
Constraints	SBDS	AWCS	ABTDO	ABT
45	3	3	6	5
55	4	4	7	7
65	5	6	9	9
75	8	10	12	12
85	11	16	15	15
95	16	23	18	19
105	23	33	22	24
115	31	46	26	28
125	43	63	30	33
135	57	83	35	39
145	81	111	39	45
155	110	142	45	54
165	137	171	50	60
175	181	216	56	67
185	239	258	62	77



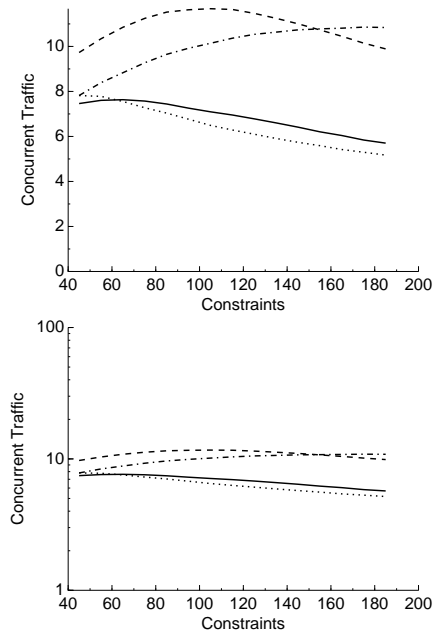
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.34: Other measures of concurrency for **all instances in problem set 2**.

Concurrent Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	9	2	5	5
55	9	2	4	5
65	9	2	4	5
75	9	2	4	4
85	9	2	4	4
95	9	2	3	4
105	9	2	3	4
115	9	2	3	4
125	9	1	3	4
135	9	1	3	4
145	9	1	3	3
155	8	1	3	3
165	8	1	3	3
175	8	1	2	3
185	8	1	2	3



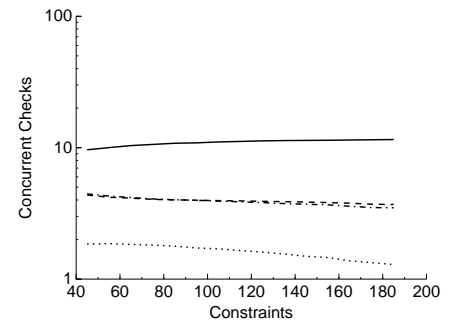
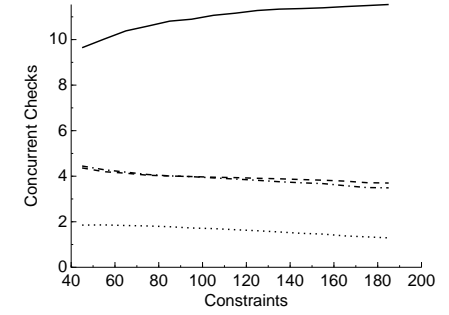
Concurrent Traffic				
Constraints	SBDS	AWCS	ABTDO	ABT
45	7	8	8	10
55	8	8	8	10
65	8	8	9	11
75	8	7	9	11
85	7	7	10	12
95	7	7	10	12
105	7	6	10	12
115	7	6	10	12
125	7	6	11	11
135	7	6	11	11
145	6	6	11	11
155	6	6	11	11
165	6	5	11	10
175	6	5	11	10
185	6	5	11	10



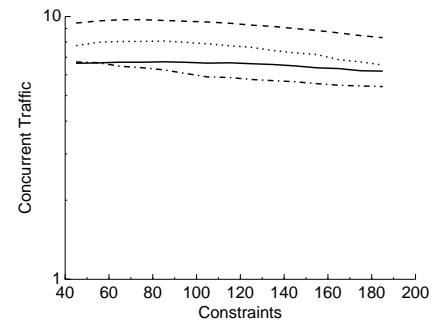
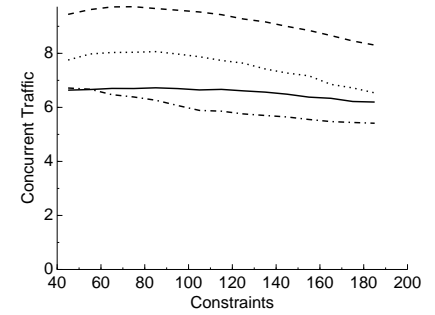
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.35: Other measures of concurrency for **feasible instances** in **problem set 2**.

Concurrent Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	10	2	4	4
55	10	2	4	4
65	10	2	4	4
75	11	2	4	4
85	11	2	4	4
95	11	2	4	4
105	11	2	4	4
115	11	2	4	4
125	11	2	4	4
135	11	2	4	4
145	11	1	4	4
155	11	1	4	4
165	11	1	4	4
175	11	1	4	4
185	12	1	3	4



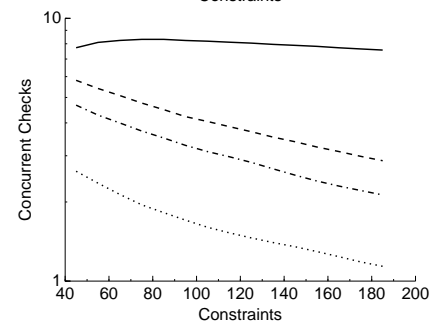
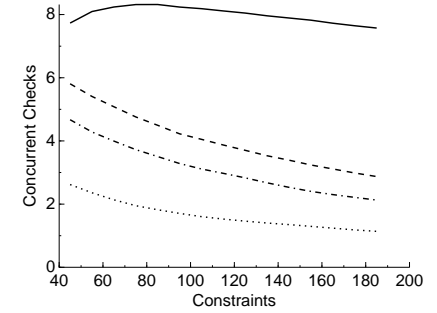
Concurrent Traffic				
Constraints	SBDS	AWCS	ABTDO	ABT
45	7	8	7	9
55	7	8	7	10
65	7	8	6	10
75	7	8	6	10
85	7	8	6	10
95	7	8	6	10
105	7	8	6	10
115	7	8	6	9
125	7	8	6	9
135	7	7	6	9
145	6	7	6	9
155	6	7	6	9
165	6	7	5	9
175	6	7	5	8
185	6	7	5	8



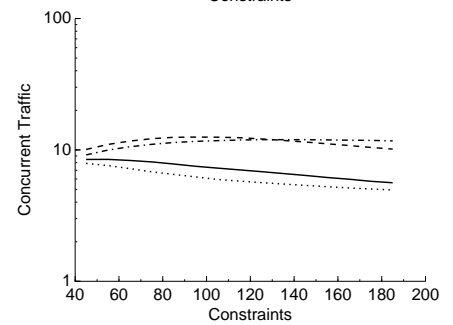
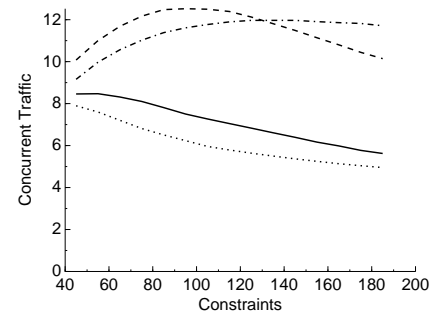
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.36: Other measures of concurrency for **infeasible instances** in **problem set 2**.

Concurrent Checks				
Constraints	SBDS	AWCS	ABTDO	ABT
45	8	3	5	6
55	8	2	4	5
65	8	2	4	5
75	8	2	4	5
85	8	2	4	4
95	8	2	3	4
105	8	2	3	4
115	8	2	3	4
125	8	1	3	4
135	8	1	3	4
145	8	1	3	3
155	8	1	2	3
165	8	1	2	3
175	8	1	2	3
185	8	1	2	3



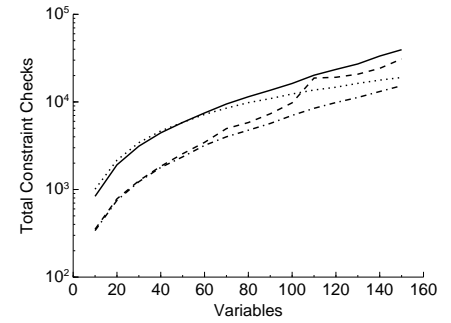
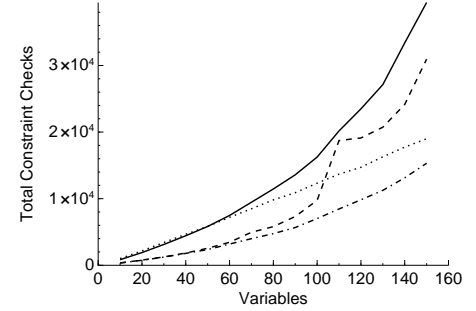
Concurrent Traffic				
Constraints	SBDS	AWCS	ABTDO	ABT
45	8	8	9	10
55	8	8	10	11
65	8	7	11	12
75	8	7	11	12
85	8	7	11	12
95	8	6	12	13
105	7	6	12	12
115	7	6	12	12
125	7	6	12	12
135	7	6	12	12
145	6	5	12	11
155	6	5	12	11
165	6	5	12	11
175	6	5	12	10
185	6	5	12	10



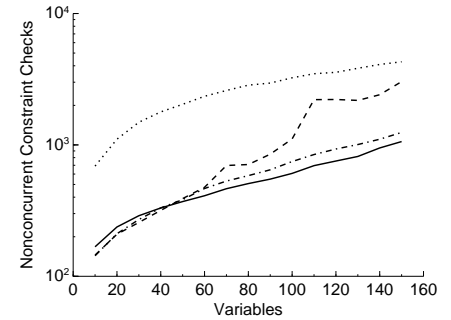
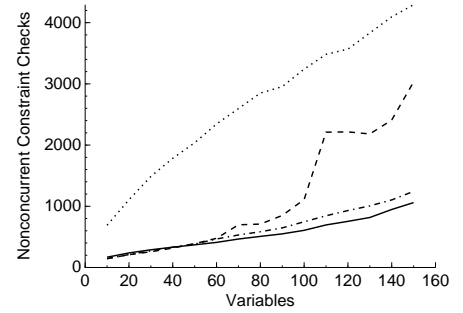
———— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.37: Number of constraint checks for **all instances** in **problem set 3**.

Total Constraint Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	837	1,009	339	352
20	1,914	2,161	752	783
30	3,125	3,427	1,228	1,258
40	4,438	4,659	1,785	1,841
50	5,829	5,834	2,359	2,558
60	7,484	7,202	3,186	3,453
70	9,471	8,490	3,999	4,959
80	11,455	9,820	4,752	5,811
90	13,584	10,900	5,672	7,322
100	16,251	12,320	7,019	9,759
110	20,176	13,683	8,455	18,731
120	23,490	14,717	9,837	19,120
130	27,128	16,287	11,251	20,735
140	33,394	17,742	13,138	24,155
150	39,459	19,002	15,325	30,951



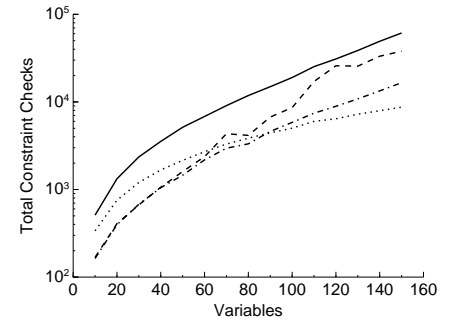
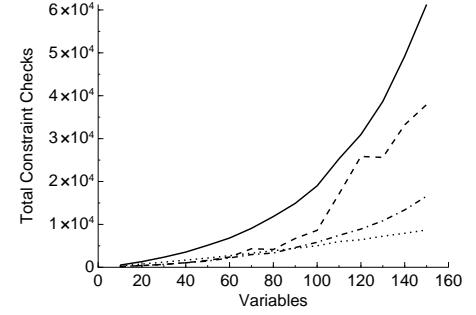
Nonconcurrent Constraint Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	168	690	143	145
20	237	1,106	211	209
30	289	1,487	270	257
40	331	1,784	332	317
50	371	2,038	383	390
60	410	2,344	466	476
70	464	2,596	531	697
80	508	2,850	584	709
90	549	2,955	647	850
100	607	3,241	747	1,110
110	695	3,482	845	2,211
120	754	3,568	930	2,216
130	817	3,832	1,006	2,182
140	948	4,096	1,104	2,409
150	1,060	4,294	1,244	3,032



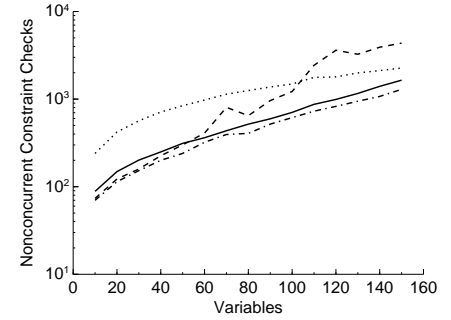
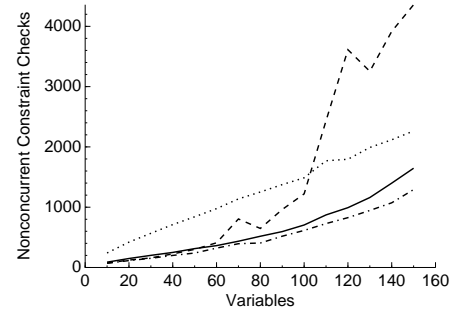
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.38: Number of constraint checks for **feasible instances** in **problem set 3**.

Variables	SBDS	AWCS	ABTDO	ABT
10	510	338	163	168
20	1,327	755	397	407
30	2,354	1,203	679	668
40	3,541	1,675	1,049	1,062
50	5,134	2,152	1,456	1,602
60	6,819	2,702	2,176	2,371
70	9,078	3,298	2,961	4,328
80	11,830	3,857	3,323	4,151
90	14,908	4,439	4,602	6,714
100	18,993	5,016	5,839	8,613
110	25,289	5,992	7,443	17,098
120	30,949	6,418	8,912	25,851
130	38,667	7,242	10,835	25,606
140	49,227	7,926	13,380	33,219
150	61,266	8,702	16,577	37,942



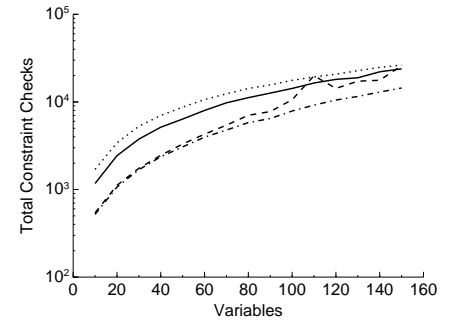
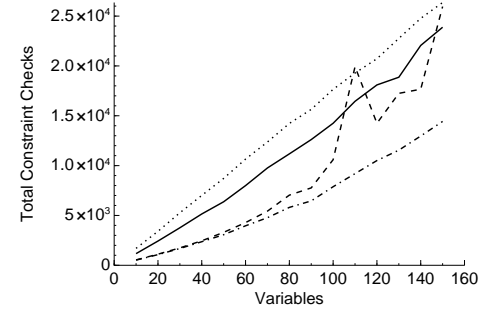
Variables	SBDS	AWCS	ABTDO	ABT
10	88	240	70	74
20	149	421	114	122
30	201	566	153	159
40	249	712	200	225
50	312	840	240	300
60	362	976	321	409
70	435	1,137	394	804
80	518	1,252	406	648
90	595	1,374	519	962
100	705	1,487	615	1,221
110	872	1,770	728	2,434
120	992	1,795	827	3,612
130	1,160	1,990	946	3,254
140	1,399	2,118	1,073	3,915
150	1,646	2,264	1,293	4,359



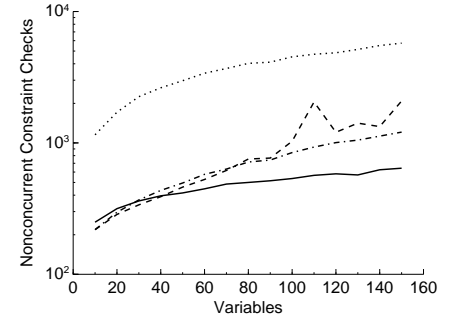
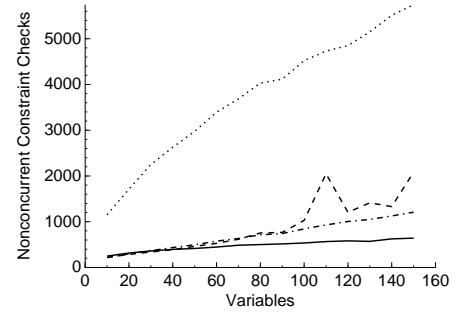
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.39: Number of constraint checks for **infeasible instances** in **problem set 3**.

Variables	SBDS	AWCS	ABTDO	ABT
10	1,172	1,700	521	541
20	2,433	3,406	1,066	1,116
30	3,762	5,266	1,682	1,746
40	5,144	7,007	2,365	2,454
50	6,374	8,717	3,066	3,307
60	7,993	10,653	3,961	4,282
70	9,765	12,368	4,774	5,430
80	11,177	14,228	5,809	7,038
90	12,613	15,644	6,458	7,767
100	14,251	17,648	7,880	10,594
110	16,448	19,290	9,193	19,922
120	18,092	20,723	10,506	14,248
130	18,867	22,761	11,549	17,248
140	22,055	24,771	12,966	17,663
150	23,879	26,362	14,431	25,956



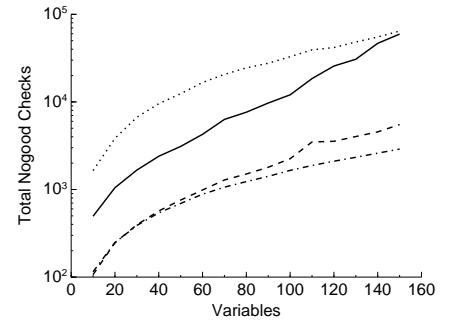
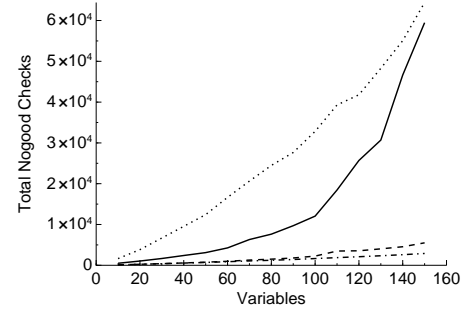
Variables	SBDS	AWCS	ABTDO	ABT
10	249	1,152	219	218
20	316	1,713	297	286
30	362	2,249	368	338
40	395	2,627	435	390
50	416	2,975	495	461
60	447	3,394	577	527
70	486	3,685	633	617
80	500	4,030	715	754
90	515	4,116	741	767
100	535	4,520	844	1,030
110	566	4,730	930	2,048
120	582	4,851	1,005	1,205
130	570	5,151	1,050	1,414
140	625	5,512	1,127	1,331
150	642	5,744	1,209	2,084



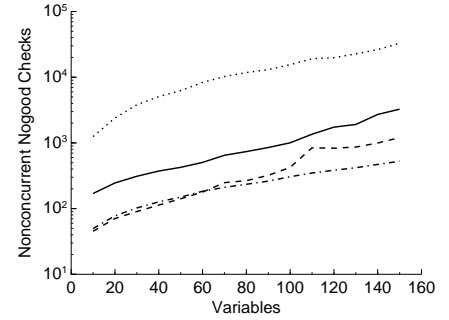
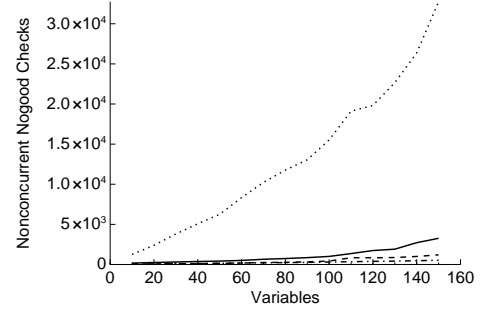
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.40: Number of nogood checks for **all instances** in **problem set 3**.

Total Nogood Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	495	1,645	115	107
20	1,050	3,836	247	247
30	1,667	6,687	389	396
40	2,398	9,540	542	569
50	3,104	12,415	693	758
60	4,265	16,653	883	987
70	6,316	20,640	1,063	1,285
80	7,631	24,477	1,231	1,504
90	9,707	27,647	1,415	1,794
100	12,051	32,831	1,652	2,247
110	18,457	39,307	1,880	3,491
120	25,700	41,795	2,104	3,553
130	30,695	48,281	2,334	4,025
140	46,639	55,062	2,601	4,549
150	59,474	64,415	2,895	5,509



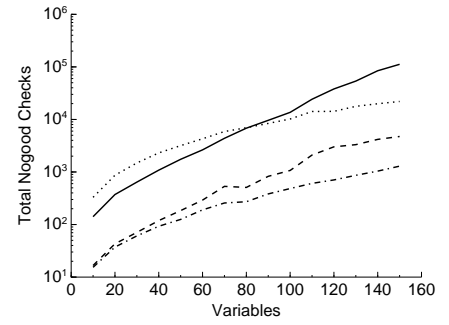
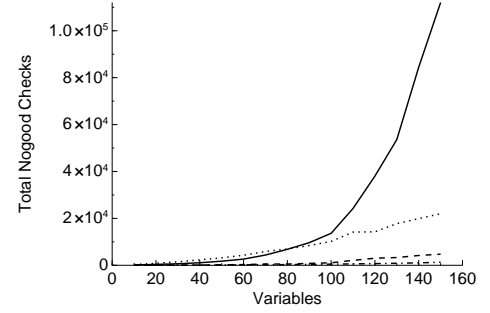
Nonconcurrent Nogood Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	170	1,250	50	45
20	246	2,397	78	71
30	311	3,805	102	90
40	373	5,065	127	113
50	424	6,252	150	141
60	505	8,334	183	178
70	648	10,228	211	248
80	738	11,765	234	266
90	850	13,055	262	321
100	1,003	15,520	306	424
110	1,355	19,111	348	841
120	1,738	19,816	384	832
130	1,911	22,663	418	865
140	2,717	26,372	470	997
150	3,260	32,751	526	1,202



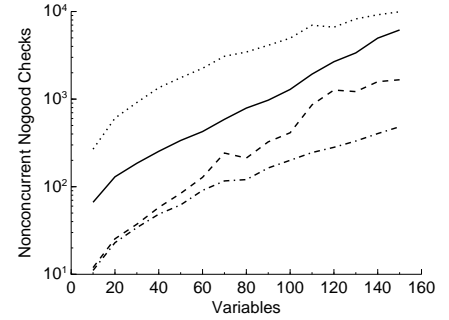
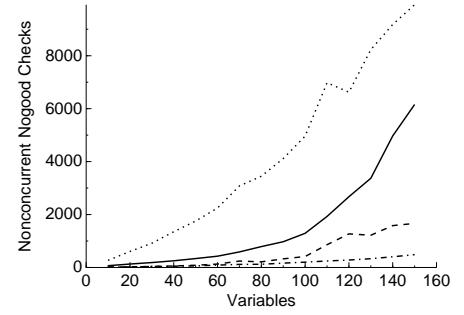
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.41: Number of nogood checks for **feasible instances** in **problem set 3**.

Total Nogood Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	141	330	15	17
20	372	868	37	43
30	642	1,485	61	71
40	1,083	2,305	93	119
50	1,745	3,179	125	184
60	2,633	4,286	190	292
70	4,371	5,886	257	533
80	6,816	6,953	272	509
90	9,629	8,403	383	827
100	13,629	10,194	484	1,069
110	24,221	14,165	608	2,102
120	38,047	14,262	706	3,013
130	53,623	17,762	864	3,311
140	84,458	19,874	1,040	4,174
150	112,038	21,997	1,291	4,735



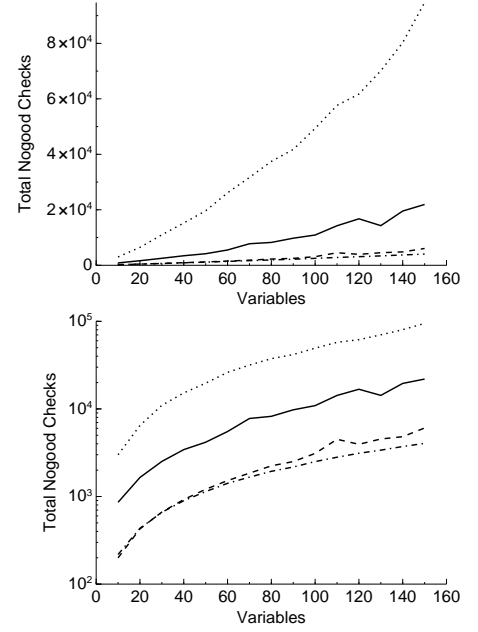
Nonconcurrent Nogood Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	66	269	11	12
20	130	606	23	25
30	185	915	34	37
40	254	1,349	49	58
50	337	1,754	62	84
60	426	2,249	91	128
70	588	3,082	116	242
80	790	3,447	120	212
90	973	4,115	164	325
100	1,290	4,962	200	413
110	1,929	6,981	246	860
120	2,674	6,621	282	1,270
130	3,370	8,232	332	1,218
140	4,970	9,180	404	1,582
150	6,157	9,928	484	1,664



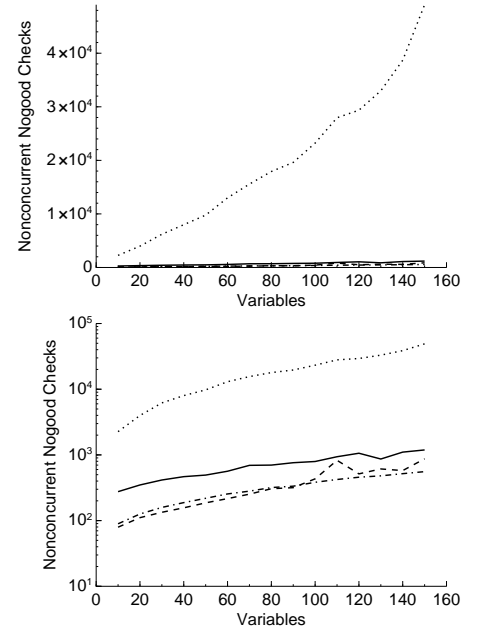
———— SBDS AWCS -.-.-.- ABTDO - - - - ABT

Figure A.42: Number of nogood checks for **infeasible instances** in **problem set 3**.

Total Nogood Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	859	2,997	218	200
20	1,650	6,464	434	428
30	2,514	10,987	660	664
40	3,432	15,232	895	923
50	4,168	19,645	1,138	1,208
60	5,517	26,134	1,414	1,520
70	7,770	31,662	1,665	1,847
80	8,234	37,433	1,940	2,239
90	9,763	41,775	2,172	2,504
100	10,900	49,343	2,504	3,105
110	14,255	57,636	2,808	4,504
120	16,763	61,723	3,115	3,943
130	14,282	70,128	3,386	4,536
140	19,552	80,264	3,719	4,818
150	21,922	94,726	4,042	6,062



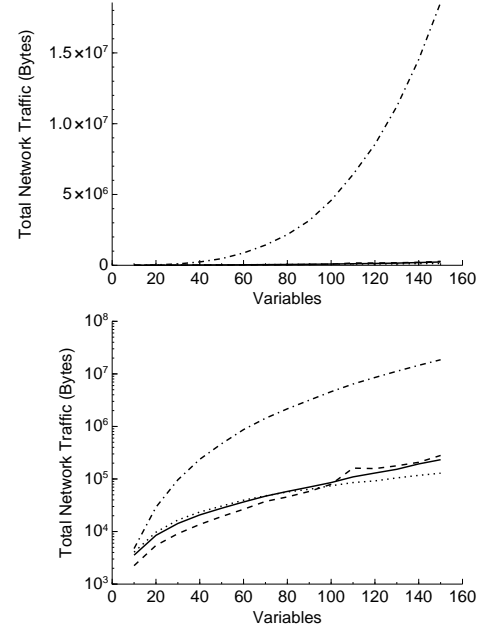
Nonconcurrent Nogood Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	276	2,259	89	79
20	349	3,983	126	111
30	415	6,194	159	134
40	466	7,989	188	157
50	493	9,774	219	186
60	565	12,998	255	216
70	693	15,566	281	253
80	700	17,915	319	306
90	760	19,618	334	317
100	794	23,220	383	432
110	937	27,954	422	828
120	1,060	29,365	457	514
130	866	32,993	480	612
140	1,103	38,686	518	578
150	1,190	49,060	556	872



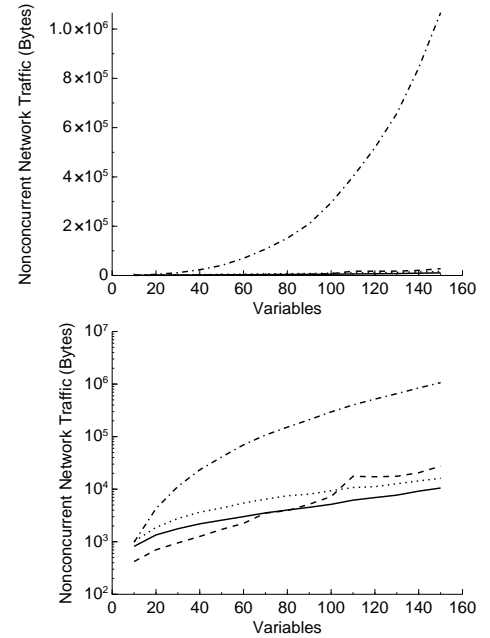
———— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.43: Network traffic for **all instances** in **problem set 3**.

Variables	SBDS	AWCS	ABTDO	ABT
10	3,553	4,048	4,739	2,242
20	8,460	9,710	29,080	5,450
30	14,237	16,549	96,389	9,117
40	20,738	23,550	236,762	13,644
50	27,804	30,491	469,813	19,323
60	36,488	39,430	869,732	26,514
70	47,186	47,916	1,428,238	37,417
80	57,991	56,740	2,165,938	45,638
90	70,232	63,744	3,159,177	57,955
100	85,545	74,357	4,583,661	79,322
110	109,573	85,506	6,412,481	160,975
120	129,820	91,912	8,533,433	157,856
130	153,089	104,949	11,220,678	178,237
140	193,406	117,112	14,535,484	207,951
150	232,585	128,903	18,558,394	280,664



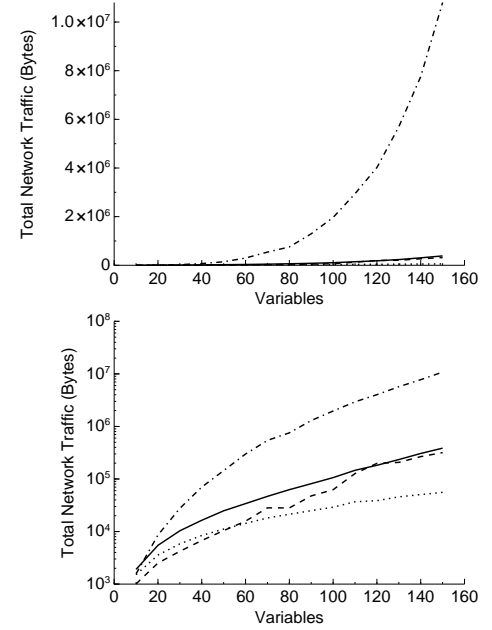
Variables	SBDS	AWCS	ABTDO	ABT
10	816	1,009	980	420
20	1,348	1,878	4,269	701
30	1,768	2,767	11,257	952
40	2,191	3,654	23,310	1,261
50	2,573	4,382	40,950	1,717
60	3,003	5,447	69,842	2,240
70	3,541	6,471	106,967	3,506
80	4,005	7,485	151,473	3,914
90	4,520	8,048	209,825	5,164
100	5,157	9,345	295,291	7,233
110	6,185	10,753	400,998	17,540
120	6,921	11,162	518,529	17,244
130	7,701	12,687	657,376	17,592
140	9,184	14,364	842,629	20,536
150	10,543	16,105	1,066,457	27,512



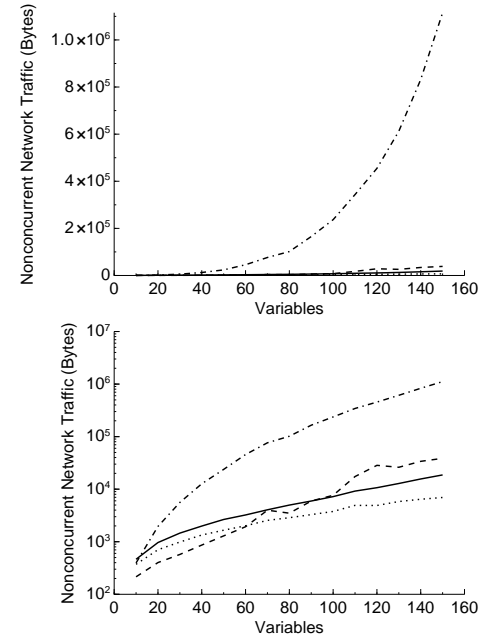
— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.44: Network traffic for **feasible instances in problem set 3**.

Total Network Traffic (Bytes)				
Variables	SBDS	AWCS	ABTDO	ABT
10	1,910	1,513	1,568	1,010
20	5,469	3,560	8,578	2,517
30	10,353	5,833	27,670	4,212
40	16,386	8,406	70,091	6,746
50	24,799	11,126	144,529	10,420
60	34,044	14,277	298,770	15,783
70	46,740	18,122	542,628	28,204
80	62,690	21,265	755,095	28,151
90	81,418	24,987	1,295,906	47,640
100	106,328	29,025	1,966,601	62,464
110	146,335	36,830	2,942,462	127,066
120	182,481	38,728	4,020,167	198,310
130	234,513	45,671	5,694,405	206,773
140	305,767	50,493	7,755,991	266,697
150	386,338	55,664	10,803,077	317,505



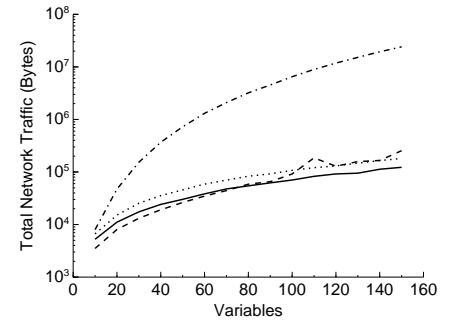
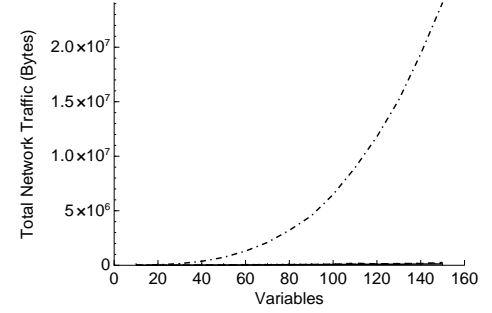
Nonconcurrent Network Traffic (Bytes)				
Variables	SBDS	AWCS	ABTDO	ABT
10	464	372	396	215
20	966	705	1,955	401
30	1,460	987	5,592	573
40	1,987	1,341	12,660	864
50	2,650	1,655	23,775	1,290
60	3,232	2,015	45,662	1,930
70	4,050	2,550	76,304	3,983
80	4,989	2,867	101,672	3,493
90	5,949	3,282	165,191	5,957
100	7,221	3,765	236,859	7,654
110	9,197	4,913	343,430	17,389
120	10,665	4,891	454,799	28,427
130	12,839	5,822	612,423	26,100
140	15,628	6,432	831,948	33,909
150	18,667	6,949	1,115,584	38,622



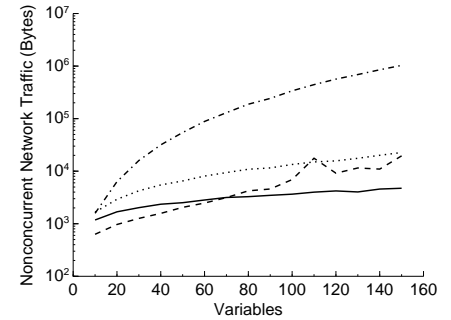
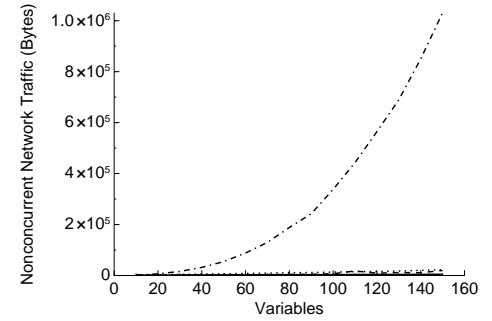
———— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.45: Network traffic for **infeasible instances** in **problem set 3**.

Total Network Traffic (Bytes)				
Variables	SBDS	AWCS	ABTDO	ABT
10	5,244	6,655	7,999	3,509
20	11,110	15,157	47,238	8,048
30	17,447	25,408	153,198	13,172
40	24,163	35,467	367,913	19,073
50	30,157	45,649	724,445	26,292
60	38,361	58,715	1,307,472	34,740
70	47,518	70,174	2,089,874	44,300
80	54,517	82,967	3,208,996	58,567
90	62,022	92,199	4,527,126	65,527
100	70,386	107,422	6,492,572	91,618
110	82,772	120,994	8,942,285	185,696
120	91,706	130,405	11,800,025	128,577
130	94,802	147,383	15,176,606	157,809
140	112,934	164,825	19,390,952	165,877
150	122,744	181,238	24,100,222	254,345



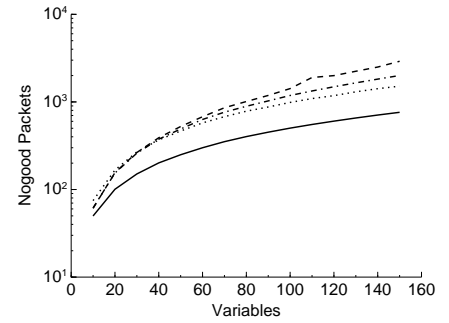
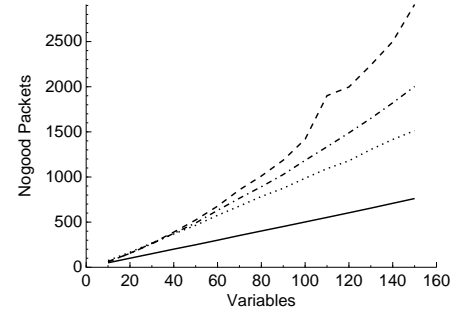
Nonconcurrent Network Traffic (Bytes)				
Variables	SBDS	AWCS	ABTDO	ABT
10	1,178	1,665	1,581	630
20	1,686	2,916	6,318	967
30	2,023	4,238	15,940	1,264
40	2,351	5,474	31,690	1,573
50	2,513	6,516	54,395	2,051
60	2,826	8,078	88,380	2,478
70	3,161	9,400	129,875	3,150
80	3,277	10,898	188,293	4,226
90	3,470	11,547	242,594	4,583
100	3,652	13,415	337,913	6,926
110	3,988	15,011	442,968	17,649
120	4,210	15,701	564,655	9,150
130	4,024	17,600	689,555	11,502
140	4,569	20,045	850,279	10,959
150	4,738	22,647	1,031,351	19,575



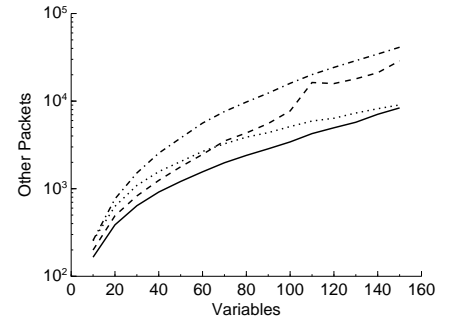
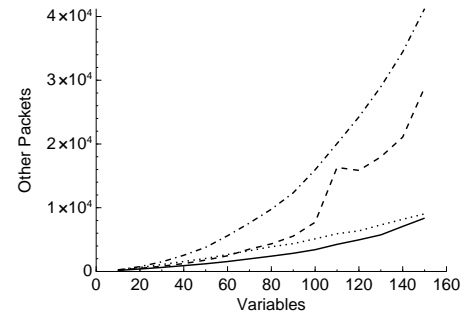
— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.46: Number of packets for **all instances** in **problem set 3**.

Nogood Packets				
Variables	SBDS	AWCS	ABTDO	ABT
10	50	75	62	61
20	101	167	154	156
30	151	269	260	265
40	201	369	376	390
50	249	465	491	524
60	300	576	630	677
70	352	678	765	858
80	402	784	892	1,011
90	451	875	1,027	1,186
100	502	986	1,185	1,419
110	553	1,092	1,334	1,901
120	604	1,180	1,489	1,997
130	655	1,304	1,650	2,239
140	709	1,415	1,820	2,500
150	761	1,513	2,003	2,910



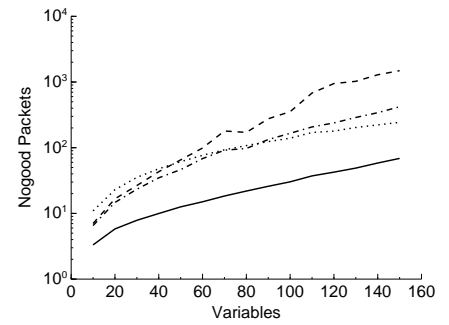
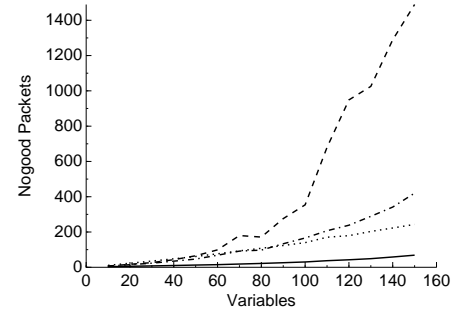
Other Packets				
Variables	SBDS	AWCS	ABTDO	ABT
10	165	255	257	199
20	387	629	765	491
30	639	1,090	1,512	827
40	915	1,567	2,546	1,244
50	1,209	2,044	3,788	1,779
60	1,559	2,665	5,607	2,464
70	1,978	3,258	7,599	3,505
80	2,399	3,878	9,768	4,333
90	2,852	4,368	12,343	5,552
100	3,420	5,124	15,955	7,724
110	4,255	5,923	20,062	16,314
120	4,959	6,370	24,249	15,855
130	5,739	7,310	28,976	17,972
140	7,071	8,186	34,447	21,082
150	8,367	9,044	41,222	28,855



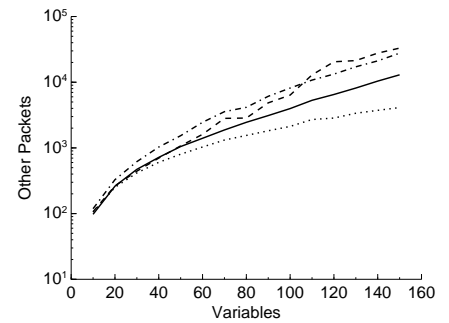
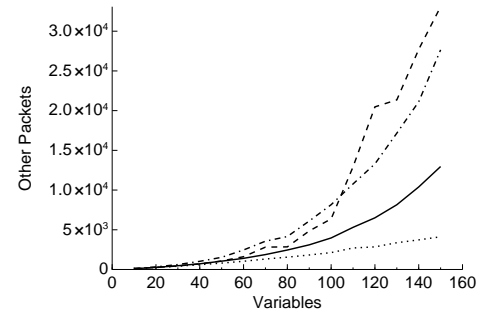
———— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.47: Number of packets for **feasible instances** in **problem set 3**.

Nogood Packets				
Variables	SBDS	AWCS	ABTDO	ABT
10	3	11	7	7
20	6	23	15	17
30	8	35	23	27
40	10	48	35	43
50	13	61	46	65
60	15	77	68	99
70	18	93	92	180
80	22	108	97	171
90	26	124	133	276
100	30	140	166	354
110	37	170	207	680
120	42	179	238	949
130	49	203	289	1,025
140	58	223	341	1,290
150	69	243	420	1,489



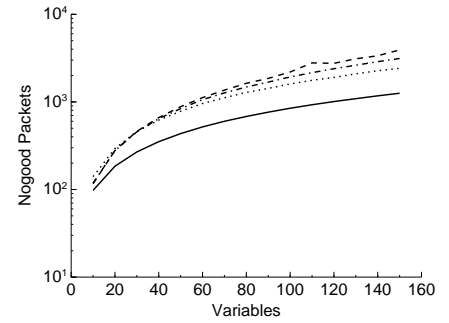
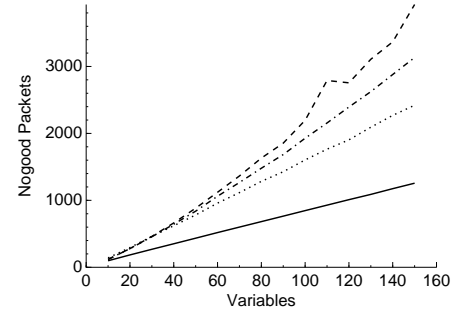
Other Packets				
Variables	SBDS	AWCS	ABTDO	ABT
10	98	106	119	107
20	262	252	328	264
30	472	416	618	443
40	717	604	1,030	700
50	1,049	803	1,534	1,074
60	1,401	1,033	2,454	1,611
70	1,874	1,317	3,564	2,815
80	2,453	1,548	4,149	2,848
90	3,104	1,824	6,099	4,849
100	3,969	2,125	8,155	6,371
110	5,302	2,705	10,769	12,918
120	6,501	2,846	13,295	20,476
130	8,151	3,370	17,163	21,352
140	10,392	3,729	21,120	27,708
150	12,961	4,116	27,654	33,085



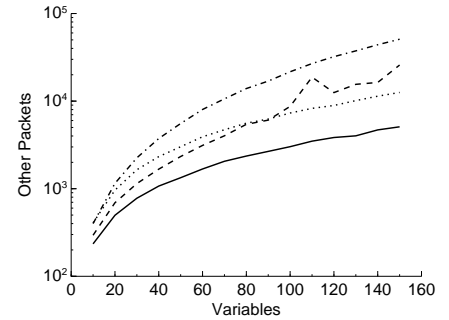
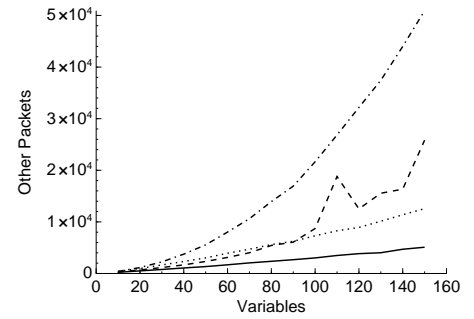
———— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.48: Number of packets for **infeasible instances** in **problem set 3**.

Nogood Packets				
Variables	SBDS	AWCS	ABTDO	ABT
10	98	140	118	116
20	185	294	278	279
30	268	462	456	461
40	352	621	644	662
50	435	781	839	883
60	518	958	1,061	1,121
70	601	1,116	1,267	1,365
80	683	1,284	1,480	1,631
90	764	1,427	1,684	1,855
100	846	1,604	1,928	2,196
110	928	1,765	2,156	2,791
120	1,010	1,904	2,394	2,756
130	1,089	2,092	2,624	3,108
140	1,174	2,269	2,880	3,368
150	1,256	2,421	3,133	3,926



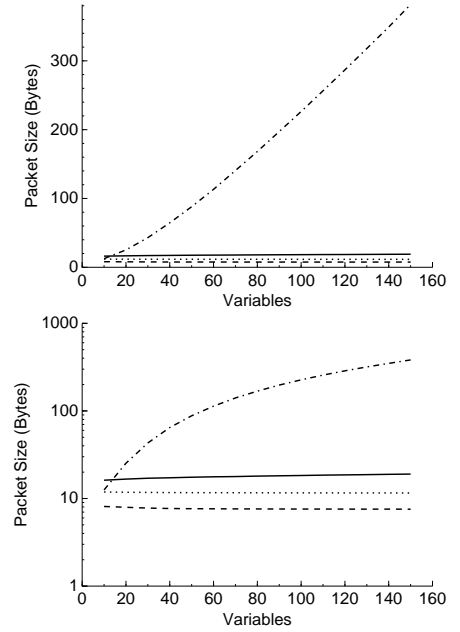
Other Packets				
Variables	SBDS	AWCS	ABTDO	ABT
10	235	408	400	294
20	498	963	1,151	691
30	778	1,647	2,251	1,144
40	1,072	2,325	3,739	1,673
50	1,335	3,016	5,553	2,331
60	1,679	3,916	8,024	3,119
70	2,056	4,709	10,614	4,020
80	2,359	5,600	13,923	5,430
90	2,667	6,236	16,927	6,067
100	3,019	7,311	21,644	8,710
110	3,491	8,268	26,837	18,790
120	3,843	8,921	32,178	12,511
130	4,013	10,130	37,432	15,553
140	4,693	11,379	43,991	16,336
150	5,085	12,567	50,918	25,833



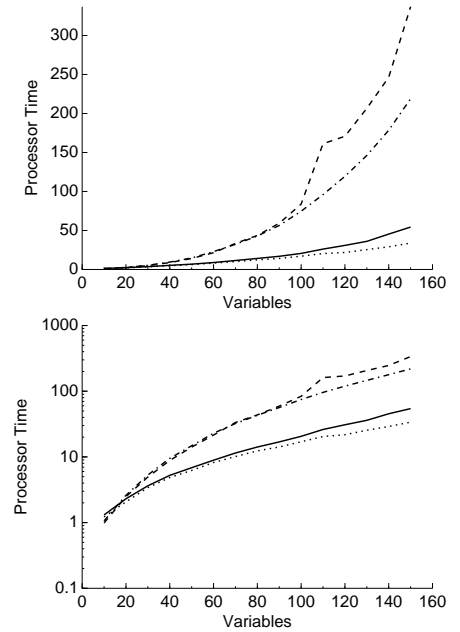
———— SBDS AWCS -.-.-.-.- ABTDO - - - - - ABT

Figure A.49: Average packet size and CPU time for **all instances in problem set 3**.

Packet Size (Bytes)				
Variables	SBDS	AWCS	ABTDO	ABT
10	16	12	13	8
20	17	12	25	8
30	17	12	43	8
40	17	12	65	8
50	17	12	88	8
60	18	12	113	8
70	18	12	141	8
80	18	12	168	8
90	18	12	197	8
100	18	12	226	8
110	18	12	256	8
120	19	12	287	8
130	19	12	318	8
140	19	12	349	8
150	19	12	382	8



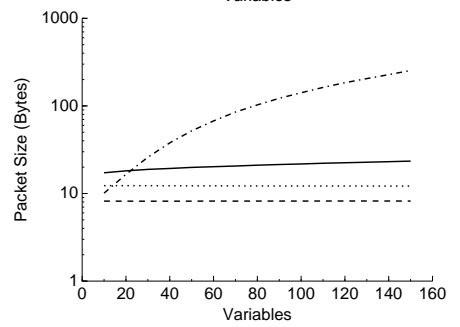
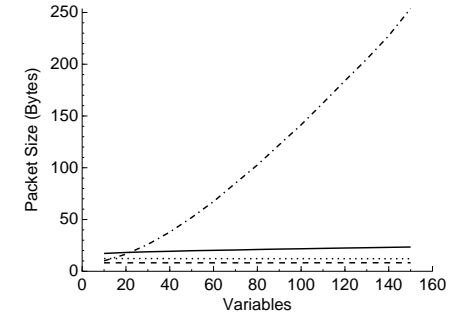
Processor Time				
Variables	SBDS	AWCS	ABTDO	ABT
10	1	1	1	1
20	2	2	3	2
30	4	3	5	5
40	5	5	9	9
50	7	6	15	14
60	9	8	23	22
70	11	10	32	33
80	14	12	43	44
90	17	14	56	59
100	21	17	75	84
110	26	20	96	161
120	31	22	119	171
130	36	25	146	207
140	45	29	179	246
150	54	34	219	337



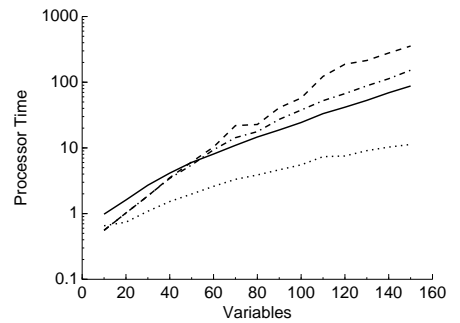
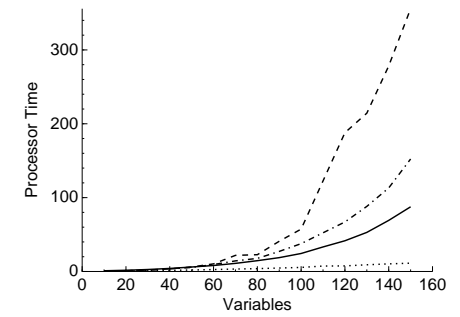
———— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.50: Average packet size and CPU time for **feasible instances** in **problem set 3**.

Packet Size (Bytes)				
Variables	SBDS	AWCS	ABTDO	ABT
10	17	12	10	8
20	18	12	17	8
30	19	12	26	8
40	19	12	38	8
50	20	12	52	8
60	20	12	67	8
70	21	12	85	8
80	21	12	103	8
90	21	12	122	8
100	22	12	141	8
110	22	12	163	8
120	22	12	184	8
130	23	12	205	8
140	23	12	228	8
150	23	12	254	8



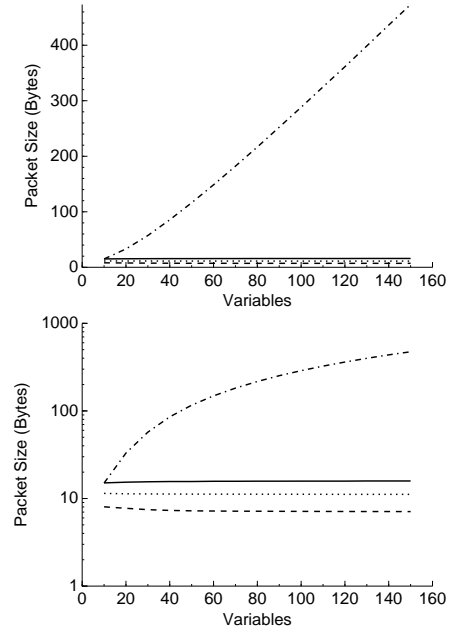
Processor Time				
Variables	SBDS	AWCS	ABTDO	ABT
10	1	1	1	1
20	2	1	1	1
30	3	1	2	2
40	4	2	3	4
50	6	2	5	6
60	8	3	9	10
70	11	3	14	22
80	15	4	18	23
90	19	5	27	41
100	24	6	38	57
110	33	7	52	123
120	42	8	67	188
130	53	9	88	214
140	69	10	113	278
150	88	11	152	356



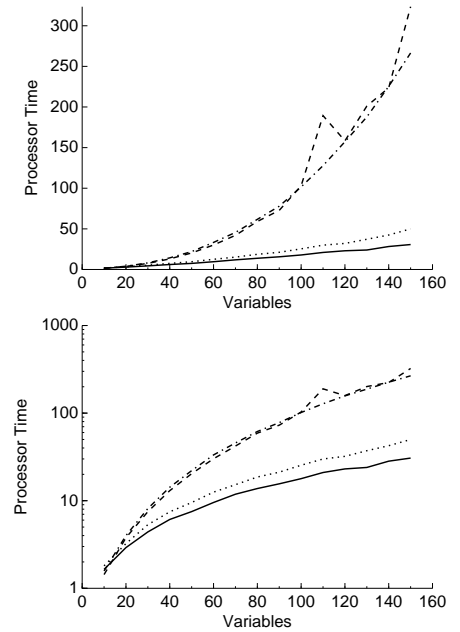
———— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.51: Average packet size and CPU time for **infeasible instances** in **problem set 3**.

Packet Size (Bytes)				
Variables	SBDS	AWCS	ABTDO	ABT
10	15	11	15	8
20	15	11	33	8
30	16	11	57	7
40	16	11	86	7
50	16	11	116	7
60	16	11	149	7
70	16	11	182	7
80	16	11	217	7
90	16	11	252	7
100	16	11	288	7
110	16	11	325	7
120	16	11	361	7
130	16	11	399	7
140	16	11	436	7
150	16	11	473	7



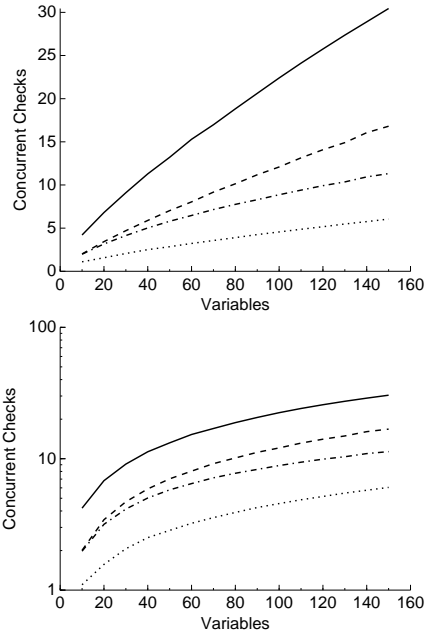
Processor Time				
Variables	SBDS	AWCS	ABTDO	ABT
10	2	2	2	1
20	3	3	4	4
30	4	5	8	7
40	6	7	14	13
50	8	10	22	20
60	10	13	33	30
70	12	15	46	42
80	14	19	62	59
90	16	21	78	73
100	18	25	102	103
110	21	30	128	189
120	23	32	157	158
130	24	37	188	201
140	28	42	225	224
150	31	50	267	323



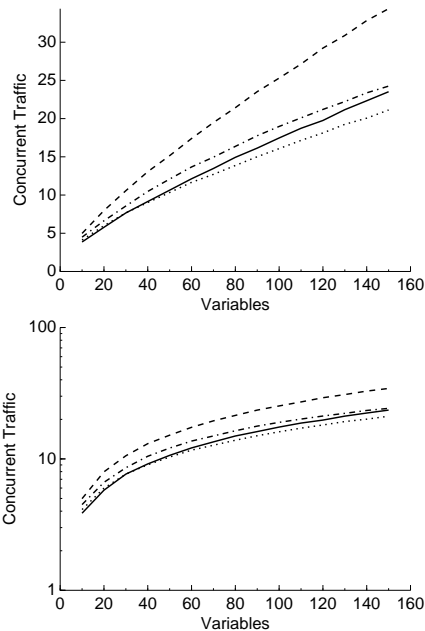
———— SBDS AWCS -.-.-.- ABTDO - - - - - ABT

Figure A.52: Other measures of concurrency for **all instances in problem set 3**.

Concurrent Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	4	1	2	2
20	7	2	3	3
30	9	2	4	5
40	11	3	5	6
50	13	3	6	7
60	15	3	6	8
70	17	4	7	9
80	19	4	8	10
90	21	4	8	11
100	22	5	9	12
110	24	5	9	13
120	26	5	10	14
130	27	5	10	15
140	29	6	11	16
150	30	6	11	17



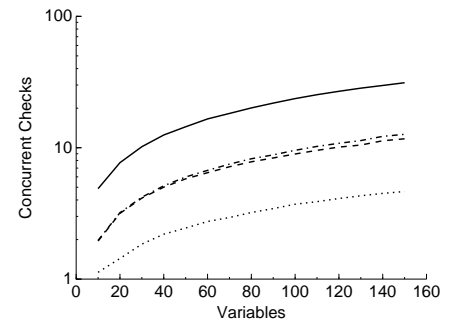
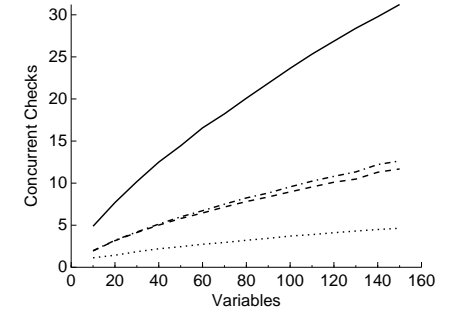
Concurrent Traffic				
Variables	SBDS	AWCS	ABTDO	ABT
10	4	4	4	5
20	6	6	7	8
30	8	8	9	11
40	9	9	10	13
50	11	10	12	15
60	12	12	14	17
70	13	13	15	19
80	15	14	16	21
90	16	15	18	24
100	17	16	19	25
110	19	17	20	27
120	20	18	21	29
130	21	19	22	31
140	22	20	23	33
150	24	21	24	34



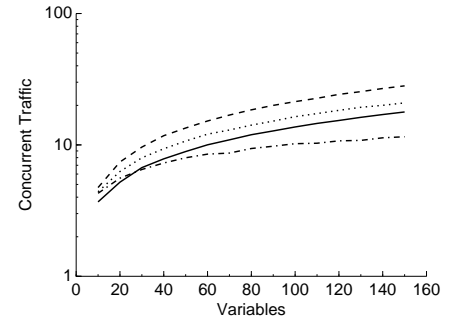
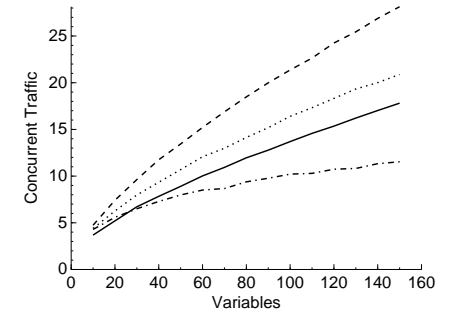
———— SBDS AWCS - - - - - ABTDO - . - . - ABT

Figure A.53: Other measures of concurrency for **feasible instances** in **problem set 3**.

Concurrent Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	5	1	2	2
20	8	1	3	3
30	10	2	4	4
40	12	2	5	5
50	14	2	6	6
60	17	3	7	6
70	18	3	7	7
80	20	3	8	8
90	22	3	9	8
100	24	4	10	9
110	25	4	10	10
120	27	4	11	10
130	28	4	11	10
140	30	4	12	11
150	31	5	13	12



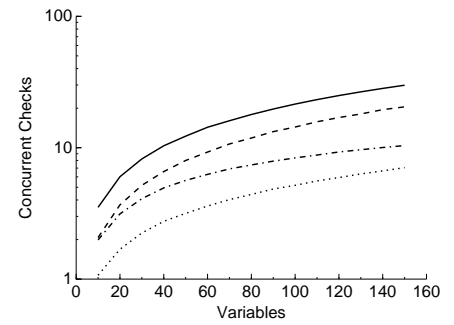
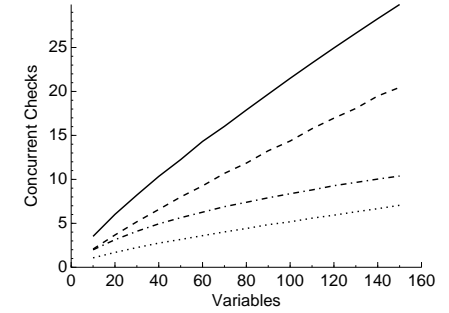
Concurrent Traffic				
Variables	SBDS	AWCS	ABTDO	ABT
10	4	4	4	5
20	5	6	6	7
30	7	8	7	10
40	8	9	7	12
50	9	11	8	13
60	10	12	9	15
70	11	13	9	17
80	12	14	9	18
90	13	15	10	20
100	14	16	10	21
110	15	17	10	23
120	15	18	11	24
130	16	19	11	25
140	17	20	11	27
150	18	21	12	28



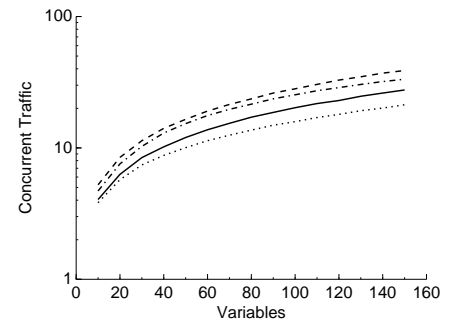
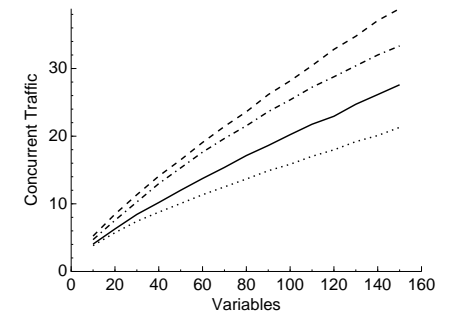
———— SBDS AWCS - - - - - ABTDO - - - - - ABT

Figure A.54: Other measures of concurrency for **infeasible instances** in **problem set 3**.

Concurrent Checks				
Variables	SBDS	AWCS	ABTDO	ABT
10	4	1	2	2
20	6	2	3	4
30	8	2	4	5
40	10	3	5	7
50	12	3	6	8
60	14	4	6	9
70	16	4	7	11
80	18	4	7	12
90	20	5	8	13
100	21	5	8	14
110	23	6	9	16
120	25	6	9	17
130	27	6	10	18
140	28	7	10	19
150	30	7	10	20



Concurrent Traffic				
Variables	SBDS	AWCS	ABTDO	ABT
10	4	4	5	5
20	6	6	8	8
30	8	7	10	11
40	10	9	13	14
50	12	10	15	16
60	14	11	18	19
70	15	13	20	21
80	17	14	22	24
90	19	15	24	26
100	20	16	25	28
110	22	17	27	30
120	23	18	29	33
130	25	19	30	35
140	26	20	32	37
150	28	21	33	39



———— SBDS AWCS - - - - - ABTDO - - - - - ABT