Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning

Ian Miguel

Ph.D.
University of Edinburgh
2001

This book is dedicated to my wife Angela, without whose love
and support this would not have been possible.

Preface

First, I would like to thank my principal supervisor Dr Qiang Shen for all his help, advice and friendship throughout. Many thanks also to my second supervisor Dr Peter Jarvis for his enthusiasm, help and friendship.

I would also like to thank the other members of the Approximate and Qualitative Reasoning group at Edinburgh who have also helped and inspired me.

This project has been funded by an EPSRC studentship, award number 97305803. I would like, therefore, to extend my gratitude to EPSRC for supporting this work.

Many thanks to the staff at Edinburgh University for all their help and support and for promptly fixing any technical problems that I have had.

My whole family have been both encouraging and supportive throughout the completion of this book, for which I am forever indebted.

York,
April 2003

Ian Miguel

Contents

List of Figures	XX
1 Introduction	1
1.1 Solving Classical CSPs	2
1.2 Applications of Classical CSP	3
1.3 Limitations of Classical CSP	6
1.3.1 Flexible CSP	6
1.3.2 Dynamic CSP	7
1.4 Dynamic Flexible CSP	7
1.5 Flexible Planning: a DFCSP Application	8
1.6 Structure	9
1.7 Contributions and their Significance	11
2 The Constraint Satisfaction Problem	13
2.1 Constraints and Constraint Graphs	13
2.2 Tree Search Solution Techniques for Classical CSP	16
2.2.1 Backtrack	17
2.2.2 Backjumping	18
2.2.3 Conflict-Directed Backjumping	19
2.2.4 Backmarking	20
2.2.5 The Backmark Hybrids	21
2.2.6 Dynamic Backtracking	22
2.2.7 Relative Evaluation	23
2.3 Pre-Processing Techniques	23
2.3.1 Arc Consistency	24
2.3.2 Improving Efficiency in Enforcing Arc Consistency	25
2.3.3 Path Consistency	28
2.3.4 K-Consistency	29
2.3.5 Practical Consistency Enforcing	30
2.3.6 Directional Pre-Processing	31
2.4 Hybrid Tree-search Consistency-enforcing Algorithms	32
2.4.1 Partial Arc Consistency	33
2.4.2 Relative Evaluation	34
2.5 Heuristics	35
2.6 Conflict Recording	37

2.7	The Phase Transition in CSPs	38
2.8	Graph-Based Methods	39
2.8.1	The Invasion Procedure	40
2.8.2	The Cycle-Cutset Method	41
2.8.3	Non-separable Components	42
2.8.4	Tree-Clustering	42
2.9	Extending the CSP Framework.....	43
2.9.1	Extending Tree Search	45
2.9.2	Solution via Graph Decomposition.....	46
2.9.3	Additive Flexible CSP	47
2.9.4	Priority Maximisation Flexible CSP	51
2.10	Dynamic Constraint Satisfaction	55
2.10.1	Restriction/Relaxation-based Dynamic Constraint Satisfaction Problems	56
2.10.2	Recurrent Dynamic Constraint Satisfaction Problems .	58
2.10.3	Activity-based Dynamic Constraint Satisfaction Problems	60
2.11	Summary	62
3	Dynamic Flexible Constraint Satisfaction	65
3.1	Towards Dynamic Flexible Constraint Satisfaction	65
3.1.1	Concepts of DFCSP	66
3.2	Examples from the Dynamic Perspective	67
3.2.1	Restriction/Relaxation-based DFCSP	67
3.2.2	Recurrent DFCSP	68
3.2.3	Activity-based DFCSP	70
3.3	A Specific Instance of DFCSP.....	70
3.3.1	The Flexible Component - a Recap	71
3.4	Fuzzy <i>rr</i> DFCSP Solution via Branch and Bound	72
3.5	Fuzzy <i>rr</i> DFCSP Solution via Local Repair.....	72
3.5.1	Local Changes	73
3.5.2	Flexible Local Changes: A Fuzzy <i>rr</i> DFCSP Algorithm	74
3.5.3	FLC Complexity Issues	79
3.6	Fuzzy Arc Consistency	80
3.6.1	The Complexity of Fuzzy Arc Consistency	81
3.6.2	Pre-processing with Fuzzy Arc Consistency	82
3.6.3	Hybrids	82
3.6.4	The Deletion Threshold	84
3.7	Solution Techniques for other DFCSP Instances	85
3.8	An Example	86
3.8.1	Solution of Initial Problem via Branch and Bound	86
3.8.2	Solution of Initial Problem via FLC	87
3.8.3	The Problem Changes	87
3.8.4	Solution of Updated Problem via Branch and Bound ..	88
3.8.5	Solution of Updated Problem via FLC	88

3.9	Summary	89
4	An Empirical Study of Fuzzy <i>rr</i>DFCSPs	91
4.1	The Problems	91
4.2	The Algorithms Studied	93
4.3	Evaluation Criteria	93
4.4	Heuristics Investigated	94
4.4.1	Variable Selection	94
4.4.2	Domain Element Selection	95
4.4.3	Constraint Check Selection	96
4.5	Results: 3-point Satisfaction Scale	96
4.6	Results: 4-point Satisfaction Scale	106
4.7	Results: 5-point Satisfaction Scale	114
4.8	The Utility of Dynamic Information	122
4.9	The Utility of the Deletion Threshold	130
4.10	The Utility of the Constraint Check Ordering Heuristic	136
4.11	The Utility of FLC Variable Selection Heuristics	140
4.12	The Utility of FLC Domain Element Selection Heuristics	148
4.13	Summary	157
5	Dynamic CSP in Domain-independent AI Planning	159
5.1	AI Planning	159
5.1.1	Constraint Satisfaction in Planning	161
5.2	An Overview of Graphplan	161
5.2.1	The Planning Graph	163
5.2.2	Basic Plan Extraction	165
5.2.3	Memoisation	165
5.3	Viewing the Planning Graph as a CSP	166
5.4	Plan Extraction via Dynamic Constraint Satisfaction	166
5.4.1	A Hierarchical Approach	166
5.4.2	Memoisation in the Hierarchical Context	168
5.5	The GP- <i>rr</i> DCSP Algorithm	174
5.5.1	The Top-level Procedure	174
5.5.2	The <code>extract()</code> Procedure	174
5.5.3	The <code>propagateMS()</code> Procedure	176
5.6	Complexity Issues	177
5.7	Avoiding Irrelevant Variables in Memosets Created by Propagation	177
5.8	Focusing the Search	179
5.8.1	Variable Selection	179
5.8.2	Value Selection	180
5.8.3	Constraint Check Selection	181
5.9	Summary	182

6	GP-<i>rr</i>DCSP:	
	Experimental Results	185
6.1	The Logistics Domain	186
6.1.1	The RocketA Problem	186
6.1.2	The RocketB Problem	187
6.1.3	The LogisticsA Problem	189
6.1.4	The LogisticsB and LogisticsC Problems	191
6.1.5	The LogisticsD Problem	193
6.2	The Blocks-world Domain	194
6.2.1	The 12-step Problem	194
6.2.2	The Blocks-world LargeA Problem	195
6.2.3	The Blocks-world LargeB Problem	196
6.3	The Gripper Domain	197
6.4	The Movie Domain	198
6.5	The Grid Domain	199
6.6	Summary	200
7	Flexible Planning Problems & Flexible Graphplan	203
7.1	Background	203
7.2	Flexible Planning Problems	205
7.2.1	Representation	205
7.2.2	The Valuable Package Problem (Continued)	208
7.3	Flexible Graph Expansion	209
7.3.1	Mutual Exclusivity	210
7.3.2	Basic Flexible Graph Expansion	210
7.3.3	Limited Graph Expansion	212
7.3.4	Satisfaction Propagation During Graph Expansion	212
7.4	Flexible Plan Extraction via <i>rr</i> DFCSP	213
7.4.1	The FCSP Viewpoint	213
7.4.2	The Hierarchical Approach Revisited	214
7.4.3	Memoisation for Flexible Plan Synthesis	215
7.4.4	The Valuable Package Problem - Synthesised Plans ...	218
7.5	The FGP Algorithm	219
7.5.1	The Top-level Procedure	219
7.5.2	The FExtract() Procedure	220
7.5.3	Complexity Issues	221
7.6	Summary	222
8	FGP: Experimental Results	225
8.1	The Test Suite	225
8.2	The Test Suite: Plan Synthesis Results	227
8.2.1	The Utility of Limited Graph Expansion and Satisfaction Propagation	229
8.3	The Rescue Problem	230
8.3.1	The Shortest Plan	232

8.3.2	A Five Step Plan	232
8.3.3	A Nine Step Plan: The People are Evacuated	233
8.3.4	An Eleven Step Plan: All People and Possessions Evacuated	234
8.3.5	A Plan with No Compromises	235
8.4	Summary	236
9	Conclusion	239
9.1	A Summary	239
9.1.1	Dynamic Flexible Constraint Satisfaction	239
9.1.2	The Application of Dynamic and Dynamic Flexible CSP to AI Planning	241
9.2	Future Work	242
9.2.1	Enrichment of the DFCSP Matrix	242
9.2.2	Improvement of GP- <i>rr</i> DCSP	243
9.2.3	Further Development of Flexible Graphplan	244
9.2.4	Further Applications	246
9.3	And Finally	250
	Bibliography	251
	References	251
A	Pseudo-code	263
A.1	Backtrack	263
A.2	Backjump	263
A.3	Conflict-directed Backjump	264
A.4	Backmark	265
A.5	Revise()	266
A.6	AC-1()	266
A.7	AC-3()	267
A.8	AC-1/4()	267
A.9	Branch and Bound	268
B	Proofs	271
B.1	Soundness and Completeness of FLC	271
B.2	Soundness and Completeness of GP- <i>rr</i> DCSP	272
B.3	Soundness and Completeness of Flexible Graphplan	274
C	An Empirical Study of Fuzzy <i>rr</i>DFCSPs: Complete Results	279
D	Planning Problems	281
D.1	The Test Suite	281
D.1.1	Domain Operators	281
D.1.2	Problem 1	284
D.1.3	Problem 2	285

D.1.4 Problem 3	286
D.1.5 Problem 4	288
D.1.6 Problem 5	290
D.1.7 Problem 6	292
D.1.8 Problem 7	294
D.1.9 Problem 8	296
D.1.10 Problem 9	299
D.1.11 Problem 10	301
D.1.12 Problem 11	304
D.1.13 Problem 12	306
D.2 The Rescue Problem	309
D.2.1 Domain Operators	309
D.2.2 Problem Specification	311
Index	315

List of Figures

1.1	Increasing Sophistication of Classical Constraint Satisfaction Algorithms	4
1.2	Some Applications of Classical Constraint Satisfaction	5
2.1	Example CSP: Constraint Graph Representation	15
2.2	Dual Graph Representation of Example of Figure 2.1	15
2.3	Generate-and-Test as a Process of Tree Search	16
2.4	Improvement Enabled by Backtrack	17
2.5	Further Improvement Enabled by Backjump	18
2.6	Failure Example for Max-Fail Backjump.....	19
2.7	Attempts to Solve the Problem Specified in Figure 2.6	20
2.8	Savings Made by Backmarking	21
2.9	Enforcing Arc Consistency	24
2.10	Global Effect of Enforcing Arc Consistency Locally	25
2.11	Example for the AC-3 Algorithm	26
2.12	Impact of Enforcing Arc Consistency on Tree Search.....	27
2.13	Effects of Enforcing Path Consistency	29
2.14	Problem Decomposition by a Hybrid Algorithm	32
2.15	Consistency Checks Required by Partial Arc Consistency Algorithms	33
2.16	Impact of a Better Instantiation Order on the Problem of Figure 2.1	36
2.17	The Phase Transition	39
2.18	Solution Graph for the Problem of Figure 2.1	40
2.19	Effects of Instantiating a Cycle Cutset	41
2.20	Non-Separable Components.....	42
2.21	Creating a Chordal Primal Graph	43
2.22	Creating a Tree in the Dual Graph	43
2.23	An Example of Flexible CSP.....	44
2.24	Rating Optimisation by Cost Minimisation for Flexible CSP ..	45
2.25	Subgraph. Optimise with Respect to Assignments to x_i (Adapted from [9])	47
2.26	Solution to the Problem of Figure 2.23 using Additive Methods	48
2.27	The Failure of Simple Summation to Capture Satisfaction Levels.....	49

2.28	Solution to the Problem of Figure 2.23 using Necessity-based Constraint Satisfaction	53
2.29	A Fuzzy CSP and Associated Search Path	55
2.30	An Evolving Problem Modelled via <i>rrDCSP</i>	56
2.31	An <i>rrDCSP</i> Example, Updated from Figure 2.1	58
2.32	Solution to the Problem Presented in Figure 2.31 via Local Changes.....	59
2.33	Compatibility Constraints for the Car Configuration Example .	62
3.1	Integration of Disparate Extensions of Classical CSP	66
3.2	The Matrix of Possible Instances of DFCSP	67
3.3	Potential Changes to a Flexible Problem Based on Restriction/Relaxation	68
3.4	Restriction/Relaxation-based Dynamic Flexible CSP	68
3.5	Flexible Assignment Choices for Recurrent DFCSP	69
3.6	A Flexible Compatibility Constraint for Car Configuration ...	70
3.7	The Local Changes Algorithm: Example Search Structure	74
3.8	Pseudo-code for <code>FLCSolve()</code>	76
3.9	Pseudo-code for <code>FLCVariables()</code>	76
3.10	Pseudo-code for <code>FLCVariable()</code>	77
3.11	Pseudo-code for <code>FLCRepair()</code>	79
3.12	Enforcing Flexible Arc Consistency via <code>FRevise()</code>	81
3.13	The FAC-3 Algorithm	81
3.14	Sub-problem Composition Under Repair.....	83
3.15	Enhancing <code>FRevise()</code> via a Deletion Threshold	84
3.16	Adapting <code>FLCVariable()</code> to Additive Flexible CSP	85
3.17	Example Problem	86
3.18	Branch and Bound Solution to the Problem Given in Figure 3.17	87
3.19	Example Problem - Part 2	88
3.20	Branch and Bound Solution to the Problem Given in Figure 3.19	89
4.1	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	97
4.2	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$	99
4.3	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$	100
4.4	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.1$, $n = 20$, $con = 0.25$	103
4.5	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$	104
4.6	Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	107

4.7	Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 1$ constraint, $n = 20$, $con = 0.75$	108
4.8	Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 1$ constraint, $n = 40$, $con = 0.25$	109
4.9	Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 0.1$, $n = 20$, $con = 0.25$	111
4.10	Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 0.25$, $n = 20$, $con = 0.25$	112
4.11	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	115
4.12	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 1$ constraint, $n = 20$, $con = 0.75$	116
4.13	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 1$ constraint, $n = 40$, $con = 0.25$	117
4.14	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 20$, $con = 0.25$	119
4.15	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.25$, $n = 20$, $con = 0.25$	120
4.16	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	123
4.17	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.1$, $n = 20$, $con = 0.25$	124
4.18	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$	125
4.19	Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 0.25$, $n = 20$, $con = 0.25$	126
4.20	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 20$, $con = 0.25$	127
4.21	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	131
4.22	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$	132
4.23	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$	133
4.24	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$	134
4.25	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 20$, $con = 0.25$	135
4.26	Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	136
4.27	Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$	137
4.28	Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$	138

4.29	Mean Constraint Checks: $\ell = 4$, $ch = 1$ constraint, $n = 30$, $con = 0.25$	139
4.30	Mean Constraint Checks: $\ell = 5$, $ch = 1$ constraint, $n = 30$, $con = 0.25$	139
4.31	Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	141
4.32	Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$	142
4.33	Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$	144
4.34	Mean Constraint Checks: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$..	145
4.35	Mean Constraint Checks: $\ell = 5$, $ch = 0.1$, $n = 30$, $con = 0.25$..	147
4.36	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$	149
4.37	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$	150
4.38	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$	151
4.39	Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$	153
4.40	Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 30$, $con = 0.25$	155
5.1	Components of A Planning Agent	160
5.2	An Example Problem.	162
5.3	The Load-truck Operator.	162
5.4	Planning Graph $Levels_{0-2}$ for the Problem Given in Figure 5.2. For clarity, only mutual exclusion relations between non-Noop actions are given in $actions_2$ and mutual exclusion relations are not given in $propositions_2$	164
5.5	Plan Extraction as a Hierarchy of $rrDCSPs$	167
5.6	Supporting Several Phases of Plan Extraction	167
5.7	Enforcing Arc Consistency on a Sub-problem	169
5.8	Basic Memoset Construction Following Total Filtering of D_i ..	170
5.9	Avoiding Irrelevant Variables during Memoset Construction ..	171
5.10	Reverse Mapping from a Memoset to a Sub-assignment via Activity Constraints	172
5.11	The Sub-problem Hierarchy under Propagated Repair	173
5.12	Pseudo-code for $GP-rrDCSP()$	175
5.13	Pseudo-code for $extract()$	175
5.14	Pseudo-code for $propagateMS()$	176
5.15	Propagated Irrelevance Testing and Propagation of Constrainedness Information	178
5.16	Variable Selection Based on Parent-node Membership of Partially-Constructed Memosets	180

5.17	Propagating Partial Memoset Information down the Level Hierarchy	181
5.18	Propagating Domain Size Information down the Level Hierarchy	182
6.1	The RocketA Problem	186
6.2	The RocketB Problem	188
6.3	The LogisticsA Problem	189
6.4	The LogisticsB Problem	191
6.5	The LogisticsC Problem	192
6.6	The LogisticsD Problem	193
6.7	The Blocks-world 12Step Problem	194
6.8	The Blocks-world LargeA Problem	196
6.9	The Blocks-world LargeB Problem	197
6.10	The Gripper Problem, Number 1	198
6.11	The Grid Problem, Number 1	200
7.1	Discrete Unary Fuzzy Relation Underlying the Flexible Proposition (<i>valuable ?pkg_i k_j</i>)	206
7.2	General Format of a Flexible Operator	206
7.3	General Format of a Flexible Goal	207
7.4	An Example Problem (Flexible Version)	208
7.5	The Flexible Operator Load-Truck	209
7.6	The Flexible Operator Drive	209
7.7	Flexible Goals for the Valuable Package Problem	209
7.8	Flexible Planning Graph <i>Levels</i> _{1,2} for the Flexible Version of the Problem given in Figure 5.2. For clarity, mutual exclusion relations are given only in <i>actions</i> ₁ . Note that all pairs of non-Noop nodes in <i>actions</i> ₂ except {G-T <i>l</i> _⊥ , L-T <i>l</i> _⊥ } are mutually exclusive.	211
7.9	FGP Hierarchical Plan Synthesis: Flexible Sub-problem Sub-sequence Level	215
7.10	A Flexible Sub-problem Sub-sequence in Detail	215
7.11	Constructing Memosets by Tracing Reduction Explanations...	217
7.12	Memoset Construction Following Reduction of Unary Consistency Degree of all Elements of <i>D_i</i> below <i>l_a</i>	217
7.13	Pseudo-code for FGP()	219
7.14	Pseudo-code for FExtract()	220
8.1	Flexible Problem 2 (of 12) for Comparing Boolean/Flexible Extraction.....	226
8.2	Test Suite of Flexible Problems: Composition	226
8.3	Test Suite of Problems: Cumulative Run-times Utilising Both Flexible and Boolean Operators	227

8.4	Test Suite of Problems: Cumulative Run-times Utilising Flexible Operators without Limited Graph Expansion or Satisfaction Propagation	228
8.5	Test Suite of Problems: Cumulative Run-times Utilising Flexible Operators and Limited Graph Expansion Only	229
8.6	Test Suite of Problems: Cumulative Run-times Utilising Flexible Operators and Satisfaction Propagation Only	230
8.7	Overview of the Rescue Problem	231
9.1	A Simple Behaviour Tree	246
9.2	Constraint Graph Decomposition into Non-separable Components to Form Meta-Variables	249
A.1	The Backtrack Algorithm	263
A.2	The Backjump Algorithm	264
A.3	The Conflict-directed Backjump Algorithm	265
A.4	The Backmark Algorithm	266
A.5	Enforcing Arc Consistency via <code>Revise()</code>	267
A.6	The <code>AC-1()</code> Algorithm	267
A.7	The <code>AC-3()</code> Algorithm	268
A.8	The <code>AC-1/4()</code> Procedure	268
A.9	The Branch and Bound Algorithm	269

1 Introduction

Constraints are a natural means of knowledge representation in many disparate fields. A constraint often takes the form of an equation or inequality, but in the most abstract sense is simply a logical relation among several variables expressing a set of admissible value combinations. The following are simple examples: the sum of two variables must equal 30; no two adjacent countries on the map may be coloured the same; the helicopter is designed to carry only one passenger, although a second can be carried in an emergency; the maths class must be scheduled between 9 and 11am, but it may be moved to the afternoon later in the term. It is this generality and simplicity of structure which underly the ubiquity of the constraint-based representation in Artificial Intelligence.

The process of identifying a solution to a problem which satisfies all specified constraints is termed constraint *satisfaction*. It has emerged as a fundamental technique for inference. Many years of research effort have been spent on developing constraint satisfaction and interest is still growing in these methods as they become increasingly widely applied. The unique structure and properties of constraint satisfaction problems are considered to be deserving of detailed examination in order that they can be exploited in the search for solutions.

Constraint-based reasoning is declarative, allowing the formulation of knowledge without specifying how the constraints should be satisfied. This decoupling of problem and solution technique has allowed the constraint satisfaction problem to be tackled from a number of different perspectives. Hence, a variety of techniques have been developed for finding partial/complete solutions of different kinds.

Despite its inherent simplicity, a constraint-based representation can be used very naturally to express real, difficult problems. For example, the problems of interpreting an image, scheduling a collection of tasks, planning an evacuation procedure, or diagnosing a fault in an electrical circuit can all be viewed as instances of the constraint satisfaction problem. The importance of constraint satisfaction to such an array of different fields has led to the development of new programming languages to support it.

There are, however, some problems that cannot easily be modelled via standard constraint-based techniques. For example, the classical formulation

offers no means to deal with over-constrained problems that admit no ‘perfect’ solution. Consider, for instance, the opening example involving the capacity of the helicopter. In reality, if no other solution could be found a ‘compromise’ solution would be constructed whereby the helicopter would carry a second passenger.

Furthermore, classical constraint satisfaction cannot efficiently support problems whose structure is subject to change. In the opening example concerning the scheduling of the maths class it is noted that the constraint may change. If this situation occurs, the natural approach is to attempt to repair the old solution, disturbing it as little as possible. This, however, is beyond the scope of the standard constraint satisfaction definition.

This book identifies and examines existing extended constraint-based methods which address these limitations, and shows how they can be further combined to solve still more complex problems.

1.1 Solving Classical CSPs

The classical *Constraint Satisfaction Problem* (CSP) [31, 100, 112, 162] involves a fixed, finite set of problem *variables*, each with an associated finite *domain* of potential values. A set of *constraints* range over the variables, specifying the allowed combinations of *assignments* of values to variables. In order to solve a classical CSP, it is necessary to find one or all assignments to all the variables such that all constraints are simultaneously satisfied. A constraint satisfaction solution procedure must find one/all such assignments or prove that no such solution exists.

It has been shown that the classical CSP is NP-complete [104]. Therefore, solution techniques for classical CSP, although many and varied, generally employ some form of search to explore the space of possible variable assignments (see Figure 1.1a). A number of such search procedures have been developed, the most basic being the well known *Backtrack* search [13]. Efficiency is crucial in solving the complex combinatorial problems to which CSP methods are typically applied. Therefore, constraint satisfaction algorithms have become increasingly sophisticated in their attempts to minimise search effort. The standard measures of search effort by which CSP algorithms are judged are the size of the space explored and the number of constraint *checks*, the latter being an examination of a constraint to see whether a particular combination of assignments is allowable.

Given the difficulty of solving classical CSPs, much of the research into constraint satisfaction has gone into finding ways to reduce the effort required. For example, a class of *pre-processing* algorithms [12, 98, 116] has been developed which simplify the problem prior to search. A process known as *enforcing consistency* [98] is used which examines the problem structure, making explicit constraints implied by, but not actually present in, the original problem formulation (see Figure 1.1b). The new constraints may manifest

in terms of the removal of elements (which have been shown not to form part of any solution to the problem) from certain domains, the removal of some tuples in existing constraints, or the creation of entirely new constraints. Identifying such ‘hidden’ constraints saves time during subsequent search by restricting the size of the search space which must be explored.

However, a ‘once only’ pre-processing phase can have only a limited effect on search effort. The most successful CSP algorithms combine sophisticated tree search with a certain amount of consistency enforcing at each search node to form *hybrids* [121, 127]. Hybrid tree search/consistency enforcing algorithms take advantage of the fact that as search progresses, parts of the problem become ‘fixed’ as variable assignments are made. Hence, simpler sub-problems are formed which may be examined by a consistency enforcing technique in exactly the same way as in a pre-processing phase (see Figure 1.1c). This method quickly uncovers ‘dead ends’ in the search where the remaining sub-problem is shown to be insoluble. Much search effort is therefore saved as the algorithm can immediately try a different path towards a solution.

Heuristics also play an important role in reducing search effort, as presented in Figure 1.1. Ordering the three basic operations of a constraint satisfaction algorithm: variable selection, value selection and the order in which constraint checks are made, can have a substantial impact on efficiency. The guiding principle on which most CSP heuristics are based is to focus on the most constrained area of the problem, minimizing the branching factor of the search [136]. Heuristics have also evolved from early attempts which simply examine the size of each variable’s domain to more sophisticated techniques which consider a range of factors such as the structure of the problem and the *tightness* of the constraints. The tightness of a constraint is the proportion of possible assignment combinations to the constrained variables disallowed by the constraint.

1.2 Applications of Classical CSP

The inherent simplicity of a constraint-based representation is the foundation for its potentially extremely wide applicability. A variety of different problems (see Figure 1.2) can, when formulated appropriately, be seen as instances of the classical CSP which can then be solved using the techniques described briefly in the previous section and in greater detail in chapter 2.

An early application of constraint satisfaction techniques appears, for example, in the field of machine vision. Waltz presents a method for producing semantic descriptions given a line drawing of a scene, leading to the Waltz algorithm [173]. In this case, variables are picture junctions which have potential values consisting of possible interpretations as corners, and constraints specify that each edge in the drawing must have the same interpretation at

Fig. 1.1. Increasing Sophistication of Classical Constraint Satisfaction Algorithms

both of its ends. Other examples of constraint satisfaction in machine vision can be found in [18, 87, 99, 117, 135, 147].

Another area that involves extensive use of constraint satisfaction is that of AI planning. Graphplan [14] reduces classical domain-independent planning [175] to the solution of a CSP. By this method, large efficiency gains were made as compared to previous state-of-the-art planning algorithms. A related approach [83] converts a planning problem into the solution of a propositional satisfiability (SAT) problem [70, 178]. SAT is the problem of deciding if there is an assignment to the variables in a propositional formula that make the formula true, and is a restriction of classical CSP. Several specialised search techniques for this type of problem have been developed [27, 146] which can then be used indirectly to solve the planning problem. Further applications to planning are to be found in [81, 154]. Chapter 5 contains more detail on this topic.

The field of scheduling also directly employs constraint satisfaction techniques. Fox presented the ISIS system for the scheduling of factory job-shops [52]. It is capable of representing a variety of constraint types, including organisational goals and physical constraints. Techniques have been developed to support both the cumulative [7] and disjunctive cases [6], where tasks can and cannot share resources respectively. Other CSP applications to scheduling are presented in [24, 126, 134].

Fig. 1.2. Some Applications of Classical Constraint Satisfaction

Classical CSP also plays a central role in the field of systems simulation [89, 107, 148]. This application requires the construction of a tree of possible system behaviours given an initial system state and some constraints. Starting from a particular system state and given rules specifying how individual variables' states may change, CSP techniques are used to construct efficiently all potential next system states, thus growing the required behaviour tree.

Embedding constraint satisfaction techniques into programming languages has proven to be very successful [169]. Constraint logic programming (CLP) enhances traditional declarative programming languages such as Prolog [78] to harness the power of CSP techniques. While searching a database of facts, a CLP program makes use of constraints to prune a significant proportion of the search tree. This results in large efficiency gains to the extent that CLP programs can rival custom solutions developed using imperative programming languages such as C. Other examples are to be found in [17, 165, 166].

The list of applications of CSP is as broad as it is deep. Further examples include belief maintenance [28, 32, 39, 102], puzzles [121], timetabling [48], temporal reasoning [1, 133, 155, 161], graph problems [21, 51, 103, 164], structural design [74], assigning system components to tasks [163], and circuit analysis [153]. Further discussion of the applications of classical CSP may be found in [169].

1.3 Limitations of Classical CSP

Classical CSP has a long and successful history of practical application. As the techniques of classical constraint satisfaction have been applied to more complex real problems, however, it has become increasingly clear that the classical formulation is insufficient. There are two main areas of weakness: firstly, the inability to deal with *flexibility* in the problem description in case no ‘perfect’ solution exists, and secondly the ability to deal gracefully with changes to the problem structure.

In the case of both limitations identified above, a requirement is exhibited for an extension to classical CSP in order to cope gracefully with the demands made by more realistic problems. Such extensions do exist, albeit separately, and are discussed briefly in the following sections, and in more detail in Chapter 2.

1.3.1 Flexible CSP

Classical constraint satisfaction techniques support only *hard* constraints specifying exactly the allowed variable assignment combinations. Hard constraints are *imperative* (a valid solution must satisfy all of the constraints) and *inflexible* (constraints are either wholly satisfied or wholly violated). In reality, of course, problems rarely exhibit this rigidity of structure. It is common for there to be *flexibility* inherent which can be used to overcome over-constrainedness (i.e. a problem that admits no solution) by indicating where a sensible compromise can be made.

In order to address this weakness, classical constraint satisfaction techniques have been extended to incorporate different types of flexible constraints. The flexible constraints used attempt to model the ‘soft’ constraints often found in real problems. The different methods of extending classical CSP to model flexibility are known under the umbrella term of flexible CSP (FCSP) [42, 58].

One common example is the use of *priorities* or weights attached to each constraint indicating its relative importance. In the over-constrained case, where a compromise-free solution does not exist, if only the constraints with the lowest importance are *relaxed* (i.e. removed from the problem) the likelihood of finding a soluble problem with a useful solution is maximised. For example, the GARI system [37] provides weighted pieces of advice in the process of producing plans for machining parts given their drawings. If there is no ‘perfect’ solution, then the least important pieces of advice can be ignored.

Further applications of flexible CSP include: flexible job-shop scheduling [41], management of over-constrained resource scheduling problems [8], and control in the wine industry [137].

1.3.2 Dynamic CSP

A further limitation of classical CSP is in its assumption of a *static* problem. That is, once the sets of variables, domains and constraints have been defined, they are fixed for the duration of the solution process. In reality, of course, problems can be subject to change either as a solution is being constructed, or while a constructed solution is in use. Consider a large scale scheduling problem, such as scheduling astronomical observations on the Hubble Space Telescope [114]. Not only is the initial problem extremely difficult to solve (e.g. tens of thousands of observations must be scheduled each year, subject to a great variety of constraints), but it also changes continually: new observations may be submitted at any time for scheduling. Classical constraint satisfaction can deal at best only clumsily with this situation, considering the changed problem as an entirely new problem which must be solved from scratch.

A special case also exists in which the problem structure changes not due to external factors, but as a result of choices made in partially solving the problem. Consider a simple example involving configuring a new car for a customer. If the customer wants a sun-roof, further choices must be made concerning its type (glass or metal?) and operation (manual or electric?). The sub-choices are meaningless without the selection of a sun-roof in the first place. Classical CSP can only be coerced into solving this type of problem by explicating all ways in which the problem structure can change based on tentative sub-solutions.

To address these types of problem, the techniques of *dynamic* constraint satisfaction (DCSP) have also been developed. In order to model problems which change over time, constraints are added to and removed from the current problem description as necessary [32]. Specialised techniques are then employed to re-use as much of the solution or partial solution obtained for a problem before it changed with respect to the new problem state [166, 114, 167]. In order to model problems whose state changes based on the assignments of one or more variables, special constraints are introduced which serve to activate sub-parts of the problem structure given certain assignments [115].

1.4 Dynamic Flexible CSP

The two types of extension to classical CSP briefly described above: flexible CSP and dynamic CSP respectively, separately offer significant advances in the range of problems that CSP techniques can solve. Unfortunately, current dynamic constraint satisfaction research is founded almost exclusively on classical CSP, unable to take advantage of flexible constraints in a dynamic environment, and flexible CSP research is limited to static problems. Little has been done to combine dynamic and flexible constraint satisfaction in

order to maintain the benefits of both individual approaches to solve more complex problems.

The principal contribution of this book is in presenting the combination of these two previously disparate extensions, exploring techniques for coping with dynamic flexible CSPs (DFCSPs [110]). Based on a systematic review of classical, dynamic, and flexible CSP techniques, a matrix of possible instances of DFCSP is identified. Each instance combines particular individual methods for dynamic and flexible CSP and is suitable for modelling a different type of problem.

One such instance, combining restriction/relaxation-based dynamic CSP [32] and fuzzy flexible CSP [42] is investigated in detail, both in terms of solution techniques and the structure of the problems produced. The individual dynamic and flexible components are powerful and have already been successful in their own right. Hence, this particular instance of DFCSP is chosen as being sufficiently general to find a range of uses and as an indicator of the utility of DFCSP.

In this book, two techniques are developed to solve this type of DFCSP. The first is a novel integrated algorithm, Flexible Local Changes. The soundness and completeness of this algorithm is established, and an analysis of its time and space complexity is performed. The second is a heuristic enhancement of a standard branch and bound [93] approach to solving flexible CSP. In addition, an extensive empirical analysis is made of the structure and properties of this type of DFCSP and the performance of the two solution techniques.

1.5 Flexible Planning: a DFCSP Application

The second principal contribution of this book is in the application of DFCSP techniques to the field of AI Planning [175]. This field is an active and long-established research area with a wide applicability to such tasks as automating data-processing procedures [23], game-playing [152], and large-scale logistics problems [177]. Solving these types of problems demands a highly efficient approach due to their size and complexity. However, as per classical CSP, classical AI planning is unable to support flexibility in the problem description. Providing such an ability is an important step forward for the real-world utility of planning research.

Preparatory to the development of the flexible planner, a novel approach to classical plan synthesis in the Graphplan framework [14] is defined using dynamic CSP techniques [113]. It is based on the hierarchical decomposition of the initial problem into sequences of sub-problems. A dynamic CSP algorithm is then employed to efficiently solve these sequences. Empirical tests on benchmark problems show that this algorithm performs competitively against the current state-of-the-art solvers, and in some cases shows a performance gain.

Using the above non-flexible planner as a basis, a flexible planning system is developed. Firstly, the classical AI Planning problem description is extended to incorporate flexible constraints. For flexible plan synthesis, a hierarchical decomposition of the problem is made, similar to that used in the classical case. Hence, plan synthesis is reduced to the solution of a hierarchy of DFCSPs. Flexible Graphplan (FGP), a new flexible planner, is developed to solve such flexible planning problems. A test suite of flexible problems engineered to exhibit different structural properties is created, as well as a more complex benchmark flexible problem. FGP's performance on these flexible problems is analysed in the book.

1.6 Structure

The remainder of this book is structured as itemised below:

- Chapter 2: *The Constraint Satisfaction Problem*. This chapter contains a detailed review of the current state of the art of constraint satisfaction techniques. It begins with a formal description of classical constraint satisfaction problems and the methods for representing and reasoning with such problems. The wide variety of solution techniques for solving classical CSPs are then discussed in detail. The limitations of classical CSP are addressed, concentrating in particular on the need to support flexible constraints and an evolving problem structure. A review of existing extensions to the classical framework to support the former or the latter is then presented.
- Chapter 3: *Dynamic Flexible Constraint Satisfaction*. This chapter discusses the further evolution of constraint satisfaction techniques to support simultaneously flexible constraints and dynamic changes to the problem structure: dynamic flexible CSP (DFCSP). A matrix of possible DFCSP instances is defined via the combination of the different available dynamic and flexible extensions to classical CSP. A specific representative instance, fuzzy *rrDFCSP*, combining fuzzy constraint satisfaction and restriction/relaxation based dynamic CSP is considered in detail and two basic solution algorithms, one based on branch and bound [93] techniques and the other, Flexible Local Changes, on an existing dynamic CSP algorithm [167] are developed for this type of problem. A small worked example illustrates the utility of both algorithms.
- Chapter 4: *An Empirical Study of fuzzy rrDFCSPs*. This chapter contains an extensive empirical analysis of the structure and properties of fuzzy *rrDFCSPs*. A large set of random problem sequences are generated, varying five different parameters to alter their composition. Hybrid variants of the algorithms developed in Chapter 3 are then tested on the random problems, as well as the utility of various heuristics and efficiency-improving techniques. In particular, the results show the utility of using

dynamic flexible CSP algorithms as opposed to coercing algorithms capable of supporting flexibility only into solving the dynamic sequences.

- Chapter 5: *Dynamic CSP in Domain-independent AI Planning*. Domain-independent AI Planning is introduced in this chapter as an application domain for both dynamic and dynamic flexible CSP. The recent trend of reducing the planning problem to a CSP in order to make efficiency gains is continued. A novel solution technique, GP-rrDCSP, based on applying restriction/relaxation-based dynamic CSP techniques to the standard Graphplan [14] framework is developed and described in detail.
- Chapter 6: *GP-rrDCSP: Experimental Results*. An investigation is presented in this chapter into the utility of the GP-rrDCSP AI planner, as compared with state of the art planners on benchmark problems. Both basic and enhanced versions of GP-rrDCSP are tested over five different planning domains. The enhanced version performs particularly well, with both versions being competitive across the problem set. The results therefore establish GP-rrDCSP as a solid foundation upon which to build a flexible planning system in the following chapters.
- Chapter 7: *The Flexible Planning Problem and Flexible Graphplan*. This chapter defines the novel flexible planning problem, which extends classical domain-independent planning to support soft constraints. Fuzzy constraint satisfaction underlies this definition, allowing the association of subjective truth degrees with propositions and satisfaction degrees with plan operators and goals. Flexible plan synthesis is then supported via fuzzy rrDFCSP, generalising the dynamic CSP methods used for plan synthesis (as developed in Chapter 5) to create Flexible Graphplan (FGP).
- Chapter 8: *FGP: Experimental Results*. An evaluation of Flexible Graphplan is presented in this chapter. Initially, a specially constructed test suite of problems is used in order to examine the relationship between the structure of flexible planning problems and their expected solution difficulty. This test suite is also used to evaluate the utility of various enhancements to the basic FGP algorithm. Secondly, the novel flexible benchmark Rescue problem is introduced and used to evaluate FGP's performance on a more complex example.
- Chapter 9: *Conclusions*. This chapter contains conclusional remarks and details of important future work. Examples of future work include the exploration of other instances of DFCSP, leading to the eventual development of a generalised framework capable of supporting many such instances.
- Appendices: The four appendices respectively contain companion pseudocode, proofs of the soundness and completeness of the new algorithms developed in this book, full results for the empirical study performed in Chapter 4, and the details of the test suite and Rescue problem used in Chapter 8.

1.7 Contributions and their Significance

The main contributions of this book and their significance are as follows:

- Literature review: A systematic review of the constraint satisfaction literature is made, encompassing classical, flexible and dynamic techniques. To the best of the author's knowledge, this is the first systematic review of the CSP literature for several years.
- Identification of a matrix of dynamic flexible constraint satisfaction problems (DFCSPs): these problem types enable the modelling of complex problems, combining flexible constraints with a dynamic problem specification, previously beyond the reach of constraint satisfaction techniques.
- Development of DFCSP Solution methods: Principally, this involves the creation of a novel algorithm, Flexible Local Changes (FLC), to solve fuzzy *rr*DFCSPs, a particular representative instance of DFCSP. This is the first algorithm to address such problems. Furthermore, a heuristic enhancement of branch and bound is developed also to solve fuzzy *rr*DFCSPs. It is claimed that these algorithms are easily adaptable to solving other instances of DFCSP.
- Empirical analysis of fuzzy *rr*DFCSPs: Almost 40,000 random problem instances were generated, varying five parameters to alter their composition. This is the first such study of the structure and properties of this type of problem and also the first evaluation of the performance of variants of FLC and the branch and bound algorithm.
- Development of GP-*rr*DCSP, a new classical domain-independent planner: within the popular Graphplan framework, it hierarchically decomposes the original problem and uses *rr*DCSP techniques to solve the resultant sequences of sub-problems. This algorithm offers a new perspective on the solution of classical planning problems and is found to be competitive with the state of the art via experimental analysis.
- Creation of the Flexible Planning Problem: This is an extension of the classical planning problem using fuzzy sets. Through the association of satisfaction degrees with operators and truth degrees with propositions, this type of problem allows a tradeoff between making compromises to satisfy the plan goals to a certain extent in a short number of steps, versus reducing the number of compromises made but extending the length of the plan. This is the first specification of a planning problem which allows such a tradeoff to take place.
- Development of Flexible Graphplan (FGP): This is a novel algorithm to solve flexible planning problems. Based on the GP-*rr*DCSP algorithm, it too relies on the hierarchical decomposition of the initial problem into dynamic sequences of sub-problems. These flexible sub-problems are then solved using the FLC algorithm. FGP is the first algorithm to be able to solve flexible planning problems, and contains the first practical use of the FLC algorithm.

2 The Constraint Satisfaction Problem

This chapter presents a formal description of the classical constraint satisfaction problem, the methods used to represent and reason with such problems, and the solution techniques used to solve them. Appendix A contains more detailed pseudo-code descriptions of some of these fundamental approaches for completeness. The reader is directed to these descriptions as appropriate.

Also presented are extensions to the classical constraint satisfaction framework which address, though separately, two weaknesses it exhibits: the assumption of a static problem, and the use of inflexible, imperative constraints. The following chapter then describes how CSP techniques may be developed further by combining approaches to both these weaknesses.

To illustrate the strengths and weaknesses of the techniques described, a common type of example is used throughout this chapter. A series of simple problems will be used consisting of the assignment of a letter to each of several problem variables in order to make a word. This assignment must be made such that constraints between the variables are satisfied. The exact structure of such a problem will be described in tandem with classical CSP itself initially to form a concrete example of the ideas presented.

2.1 Constraints and Constraint Graphs

A classical constraint satisfaction problem (CSP) involves a finite set of n variables, $X = \{x_1, \dots, x_n\}$, each with an associated finite domain, D_i , describing its potential values. A variable, x_i , is *assigned* one of the values from D_i . Table 2.1 presents the four variables in the example problem and their associated domains.

Variable	Domain
x_1	$\{P, K, B\}$
x_2	$\{U, L, I\}$
x_3	$\{V, U, N\}$
x_4	$\{K, J, E, B\}$

Table 2.1. Variables and Associated Domains for the Example Problem

Constraint	Definition
$c(x_1, x_2)$	$x_1 < x_2$
$c(x_1, x_3)$	$x_1 < x_3$
$c(x_1, x_4)$	$x_1 < x_4$
$c(x_2, x_3)$	$x_2 < x_3$
$c(x_2, x_4)$	$x_2 < x_4$
$c(x_3, x_4)$	$x_3 > x_4$

Table 2.2. Constraints for the Example Problem

A *partial assignment* to X is an assignment to one or more $x_i \in X$, where the number of variables assigned is less than n , and a *complete assignment* to X is an assignment to every $x_i \in X$. A set of constraints, C , ranges over these variables. A classical (hard) constraint $c(x_i, \dots, x_j) \in C$ specifies a subset of the Cartesian product $D_i \times \dots \times D_j$, indicating variable assignments that are compatible with each other. These constraints are imperative (each one *must* be satisfied) and inflexible (they are fully satisfied or fully violated). Table 2.2 presents the constraints that exist in the example problem. They specify a lexicographic ordering such that, for example, $x_1 < x_2$ requires that the assignment to x_1 is strictly lexicographically less than that to x_2 .

A *solution* to a classical CSP is a complete assignment to the problem variables satisfying all constraints simultaneously. A CSP may contain several solutions, and the task for a constraint-based problem-solver is to find one or all of these. Furthermore, a classical CSP is assumed to be static, precluding any changes to X , D , or C after their initial specification.

A CSP can be represented graphically as a *constraint network* [31]. The structure of the network is then used to guide CSP solution techniques. A CSP containing at most binary constraints (i.e. those involving two variables) may be viewed as a *constraint graph*. Each node represents a problem variable, and constraints are represented by labelled, directed arcs. For each $\text{arc}(i, j)$, there exists $\text{arc}(j, i)$, since $c(x_i, x_j) = c(x_j, x_i)$ due to the fact that $D_i \times D_j$ defines the same set of assignments as $D_j \times D_i$. Figure 2.1 presents a constraint graph representation of the example problem. For pictorial clarity, only arcs $\text{arc}(i, j)$ are shown, where $i < j$.

It is possible to represent k -ary constraints (i.e. those involving k variables) in several ways, with the different representations suited to different tasks. The *primal* graph representation [36] is a generalised constraint graph in that nodes represent variables, and an arc connects any two variables that are related by a constraint. It is used as part of some graph-based solution techniques (see, for example, Section 2.8.4). In the *dual* graph representation [36, 156], each node represents a particular constraint and has an associated domain of allowed assignment tuples, i.e. combinations of variable assignments which satisfy the constraint. Arcs connect any two nodes which share a common variable or variables. It is specified that tuples at either end of an arc must assign the same domain elements to the shared variables. A dual

Fig. 2.1. Example CSP: Constraint Graph Representation

graph representation of the example problem is presented in Figure 2.2. Using the dual graph representation, arbitrary k -ary CSP can be solved using techniques developed for binary constraint graphs [31].

An intermediate representation, known as the *hidden variable* encoding [156] contains both dual variable nodes and nodes representing individual variables. Constraints between each node representing a variable x_i and the dual nodes representing constraints involving x_i specify that the assignment to the variable must match the assignment to x_i in a tuple assigned to the dual variable. The hidden variable encoding also allows the application of binary CSP techniques, but has the advantage of enabling search using the original variables. This can be especially important when dealing with higher arity non-binary constraints which are cumbersome in a pure dual graph representation. Comparisons between the different non-binary encodings have been made and methods for translations between them have been explored [156].

Fig. 2.2. Dual Graph Representation of Example of Figure 2.1

Fig. 2.3. Generate-and-Test as a Process of Tree Search

2.2 Tree Search Solution Techniques for Classical CSP

The search for a solution to a CSP may be viewed as tree traversal. Each node in the search tree corresponds to an individual variable assignment. The *instantiation order* is the order in which variable assignments are made during search. Assuming initially that assignments are made in a fixed order following the numbering scheme of the variables, the i th level of the tree represents choices made for the i th variable in the instantiation order. At a particular *level*, i , of the instantiation order, *past* variables have indices less than i and have been assigned values. *Future* variables have indices greater than i and do not yet have assignments. The term *depth* is used to describe the distance from the root (level 0) which represents the empty set of assignments. Each branch represents a partial assignment. When the leaves of the tree are reached, a complete assignment has been made. A path from the root node to a leaf node represents a particular complete assignment.

A simple (yet computationally very expensive) approach to solving a CSP is known as *Generate and Test*. Each possible combination of variable assignments is generated and subsequently tested to see whether any of the specified constraints are violated. In the worst case, the number of complete assignments to be generated and tested is the size of the cartesian product of all variable domains. This brute force method is very expensive on problems of any reasonable size.

Figure 2.3 presents the search tree produced by Generate and Test in finding the solution $\{x_1 = B, x_2 = I, x_3 = V, x_4 = K\}$ to the example problem. Nodes are annotated with the domain values that were assigned to the current variable at this point in the search. The search path is indicated by the dashed, directed line. For clarity, a failed assignment to x_4 of each of the elements in D_4 has been replaced by a triangle. The inefficiency stems from the fact that no consistency checks are made before the leaf nodes are reached (i.e. a complete assignment has been made).

Fig. 2.4. Improvement Enabled by Backtrack

2.2.1 Backtrack

Chronological backtracking (or *Backtrack*, see Section A.1) improves this initial scheme, viewing the search process as the incremental extension of partial solutions [13]. Backtrack makes assignments sequentially such that x_i is instantiated to extend a partially complete assignment involving past variables, i.e. $x_1, \dots, x_{i-1}$. The constraints dictate the subset of D_i that can be assigned to x_i given previous variable assignments. If a constraint check fails against a past variable, the current partial assignment is not extended further. If all elements of D_i fail consistency checks (a so-called *dead-end* or *wipeout*), then the algorithm *backtracks*, making a new choice for x_{i-1} . If there are no new choices for x_{i-1} , it backtracks until a variable is found with another consistent assignment or the CSP is shown to have no solution.

Backtracking may also be viewed as tree traversal. Figure 2.4 presents the search tree generated by Backtrack in producing the same solution to the problem in Figure 2.1 as that found by generate and test. Black and white nodes represent successful and failed assignments at the current point in the search respectively. The improvement in terms of computational effort can be seen immediately. Backtrack prunes subsections of the search space by rejecting a partial assignment as soon as it is clear that it cannot yield a solution to the CSP. By analogy, Generate and Test always extends each search branch to a complete assignment before making any constraint checks. However, the worst case time complexity of Backtrack is exponential in the number of variables [97]. It is inefficient in that it usually performs more operations than are necessary to find a solution, as is discussed in the next section.

Fig. 2.5. Further Improvement Enabled by Backjump

The Thrashing Problem

A major factor contributing to the poor behaviour of Backtrack is that it suffers from *thrashing*: repeated failure of the search process due to the same reason [63]. Here is a general example: x_g and x_i are related via a constraint such that when x_g is assigned a particular value, the constraint disallows *all* potential values for x_i . Backtrack fails at level i for each element of D_i . Further, this failure will be repeated for every combination of variable assignments to the variables in the intermediate levels, h , where $g < h < i$. The degree of thrashing worsens as the number of variables in the intermediate levels, and the size of their associated domains increases.

Thrashing is present in Backtrack's attempt to solve the example problem (see Figure 2.4). It is caused here because of $c(x_1, x_4)$: when $x_1 = P$, or $x_1 = K$ there is no element of D_4 that can be assigned to x_4 such that the constraint will be satisfied. This is not discovered by Backtrack until many of the possible assignment combinations to x_2 , x_3 and x_4 have been tried.

2.2.2 Backjumping

Several improvements to the core Backtrack algorithm have been proposed to alleviate thrashing. One approach is to 'jump' back to the level that disallows any assignment to the current variable. This lifts the restriction that the algorithm must unwind the set of instantiations sequentially, potentially saving much work. The *Backjump* algorithm [63] (see Section A.2) embodies this approach. Figure 2.5 describes how Backjump solves the problem of Figure 2.1. Thrashing is avoided by jumping back to the cause of the inconsistency: from x_4 to x_1 .

Fig. 2.6. Failure Example for Max-Fail Backjump

Problems with Backjump

By jumping back to the maximum level *checked* against, Backjump can only make one jump backwards when attempting to resolve inconsistencies. The level, say h , jumped back to always has a *returnDepth* of $h - 1$ because the current assignment to x_h must have passed all consistency checks for the search to have proceeded forwards to level i . Hence, if there are no further consistent values to be assigned to x_h , the algorithm resorts to chronological backtracking. If D_h is exhausted, there must be a previous assignment which is in conflict with levels h and i . It is not necessarily the case that level $h - 1$ is the conflicting level, so thrashing occurs until the conflicting variable is re-assigned.

One remedy is to record the deepest *failure* level and jump back to that, but this leads to an incomplete algorithm. Consider the CSP presented in Figure 2.6 and the search tree generated by Backjump altered to record the deepest level at which a constraint check failed (Figure 2.7a). The algorithm fails to find the solution. Elements $\{E, F\} \in D_6$ conflict with the current assignments to x_4 and x_2 respectively. The algorithm jumps back to the deepest (x_4), but no alternative consistent assignment exists. Element $\{D\} \in D_4$ is in conflict with the current assignment to x_1 . The algorithm thus tries to find a new value for x_1 and fails. The fact that x_6 was also in conflict with x_2 was discarded when the jump was made from x_6 to x_4 . Therefore, the second jump from x_4 to x_1 jumps over x_2 and accidentally prunes the solution.

2.2.3 Conflict-Directed Backjumping

Conflict-directed Backjump (CBJ, see Section A.3) [127] improves Backjump further. Multiple backward jumps are enabled by recording the *Conflict Set* ($cSet_i$ at level i) of past variables that *failed* constraint checks with the current variable. Jumps are made to the deepest variable in this set. When jumping from i to h , $cSet_h$ is set to the union of the conflict sets at levels i and h . A further jump, if necessary, is made to the deepest variable in conflict with either x_i or x_h , avoiding the problems encountered by max-fail Backjump.

Fig. 2.7. Attempts to Solve the Problem Specified in Figure 2.6

Figure 2.7b presents the search tree for CBJ when solving the problem in Figure 2.6. The critical point follows the initial jump back from level 6 to level 4. Where max-fail Backjump took into account only those variables in conflict with x_4 , CBJ remembers that x_2 was also in conflict with x_6 and jumps back to level 2 as the deepest conflicting assignment.

2.2.4 Backmarking

Gaschnig [63] also proposed the *Backmark* algorithm (see Section A.4), which minimises the execution of redundant constraint checks in conjunction with chronological backtracking. A redundant check tests whether a set of variable assignments satisfy a constraint when the result of this check is already known. First, Backmark avoids performing constraint checks that must fail: if an assignment previously failed against a past variable whose instantiation has not changed, the same assignment must fail again. Secondly, an assignment that succeeded against a set of past variables whose instantiations are unchanged must satisfy repeated consistency checks against the same past assignments.

In order to make use of these observations, two arrays are necessary:

- **mark**[] is dimensioned by the number of variables, n , and by d , the size of the largest variable domain. $mark[i, a]$ contains the most shallow level at which a consistency check failed for the assignment of $d_{i_a} \in D_i$ to x_i .
- **minBackupLevel**[i] records the most shallow level at which an assignment has changed since the last assignment at level i .

Backmark generates the same search path as Backtrack, but makes fewer constraint checks. Consider Figure 2.8, which presents the search tree generated by Backmark for the problem given in Figure 2.1. This figure contains

Fig. 2.8. Savings Made by Backmarking

examples of both ‘must fail’ and ‘must succeed’ situations. For example in the region marked ‘1’ in the figure, when Backmark assigns the elements $\{V\}$, $\{U\}$, or $\{N\}$ to x_3 , consistency checks against x_1 must succeed since they did on the previous occasion these assignments were made, and x_1 has not been re-assigned since ($\text{minBackupLevel}[3] = 2$). Similarly, in the region marked ‘2’ in the figure, when Backmark attempts to assign the elements $\{K\}$, $\{J\}$, $\{E\}$, or $\{B\}$ to x_4 , consistency checks against x_1 must fail, since they did previously, and the assignment to x_1 has not changed ($\text{minBackupLevel}[4] = 2$ at this point).

2.2.5 The Backmark Hybrids

Backmark may be combined with more powerful versions of Backtrack to create Backmarkjump (BMJ) and Backmark-Conflict-Directed Backjump (BM-CBJ) respectively [127]. This is possible because Backmark reduces the number of consistency checks, whilst Backjump and CBJ reduce the number of search tree nodes to be explored. BMJ was first suggested in [121] and subsequently presented in [127]. It maintains the information required by *both* base algorithms: in addition to $\text{mark}[]$ and $\text{minBackupLevel}[]$, BMJ also maintains $\text{maxCheckLevel}[]$ so that a jump backwards may be performed in the event of a domain wipeout. Similarly, BM-CBJ [127] maintains the conflict set required for conflict-directed Backjump.

Unfortunately, an apparently attractive pair of algorithms can sometimes perform more consistency checks than the original Backmark. This is because Backmark was originally designed for use with chronological backtracking. When Backjump or CBJ jump over a level i , any remaining elements of D_i are not tested against previous assignments. The $\text{minBackupLevel}[i]$ is deprived of the information that it would have learnt from the consistency

checks made against these untested domain elements, hence the sometime behaviour degradation.

Kondrak and van Beek [88] extend *minBackupLevel*[] to two dimensions to solve this problem. The extended array is dimensioned by the number of variables, n , and d , the size of the largest variable domain. A particular entry $[i, j]$ in this array stores the index of the most shallow level whose assignment has changed since x_i was last assigned the j th element of D_i . This method maintains the maximum amount of ‘backup’ information. The extended *minBackupLevel*[] array is used as the foundation of Kondrak and van Beek’s two hybrids BMJ2 and BM-CBJ2, both of which always perform better than Backmark (in terms of the number of constraint checks).

2.2.6 Dynamic Backtracking

Jumping or stepping back over a level erases the work done by a problem solver in determining a consistent assignment at that level. In many cases, levels backtracked over may have nothing to do with the inconsistency that the algorithm is trying to resolve (e.g. the thrashing behaviour described in Section 2.2.1). Therefore, potentially useful work is needlessly discarded in the search for the real cause of the inconsistency.

Ginsberg [68] presented *Dynamic Backtracking* (DBT) as an approach to this problem. Erasing information which is irrelevant to the current inconsistency is due to the nature of the algorithms presented thus far which must unwind assignments sequentially until the inconsistency can be resolved. DBT dispenses with a fixed order of instantiation, instead choosing variables to assign values to in a dynamic manner (hence the name). This dynamic ordering is beneficial when an inconsistency is detected. DBT effectively changes the instantiation order such that the level responsible for the inconsistency becomes the deepest level, leaving intact all uninvolved variable assignments.

For each $d_{i_a} \in D_i$, DBT maintains a set of *eliminating explanations* which record the reason why d_{i_a} is currently excluded from consideration. These explanations are defined in terms of the current assignments to past variables. When a jump back over x_i to x_h is necessary, DBT retains the current assignment to x_i and all of its eliminating explanations except those involving x_h (whose assignment is about to change).

The use of eliminating explanations is similar to the *nogoods* mentioned in the context of conflict recording (see Section 2.6). The prohibitive space complexity that is incurred by storing all derived nogood information is avoided by DBT. Eliminating explanations are kept for only the *current* assignments to other variables; once an assignment changes, all explanations involving it are discarded. Hence, DBT represents a compromise in the tradeoff between space and time complexity.

DBT does not, however, maintain a uniform improvement over existing techniques. This is because it is sometimes *desirable* to erase intermediate information. In comparisons using random propositional satisfiability problems

and employing standard heuristics (see Section 2.5), Baker shows that DBT performs worse than Backjump by a factor that is exponential in the size of the problem [5]. The problem is that the variable assignments that DBT does not discard often have no justification (according to the heuristics) once an inconsistency is found. Maintaining these variable assignments therefore degrades the operation of the heuristic. In light of this, Baker proposes a method of selectively erasing variable assignments that, following a detected inconsistency, no longer have a heuristic justification.

2.2.7 Relative Evaluation

It is difficult to pick one ‘best’ algorithm from those presented. There have been several attempts to compare a number of them empirically [5, 63, 80, 127]. The means of comparison is usually the number of constraint checks made and/or the number of nodes in the search tree. These measures fail to take into account the overhead of individual algorithms in terms of maintaining data structures. To try to quantify this overhead, some attempts have been made to use run time also as a means of comparison [69, 127].

Kondrak and van Beek [88] have made a purely theoretical comparison of tree search algorithms. The comparison was based on both the number of search tree nodes explored and number of consistency checks. In terms of the former, Backtrack and Backmark (which has no effect on the number of search nodes) are unsurprisingly the worst. The most efficient were CBJ and its Backmark hybrids. In terms of consistency checks, Backtrack was again the worst algorithm, whilst the hybrids of Backmark were seen to be the best, in particular BM-CBJ2.

2.3 Pre-Processing Techniques

Pre-processing a CSP creates an *equivalent* (i.e. with the same solution set) but simpler problem. Tree search can solve the resulting problem with much less effort than would otherwise be required. The original CSP is modified as follows. The underlying mechanism used is *constraint propagation*: a local group of constraints is used to deduce information, which is recorded as changes to the problem. This update forms the basis for further deductions, hence the result of any change is gradually propagated through the entire CSP. Basic pre-processing consists of *filtering* domain elements that cannot take part in any solution to the problem, essentially adding and/or enforcing unary constraints, $c(x_i)$. More advanced pre-processing adds or modifies higher arity constraints.

Node consistency holds if for each $d_{i_a} \in D_i$, $c(x_i)$ is satisfied. To enforce local node consistency, all elements of D_i which do not satisfy $c(x_i)$ are removed. Global node consistency may be enforced in one pass: unary constraints affect single variables, so a local change cannot affect other variables.

Fig. 2.9. Enforcing Arc Consistency

The following sections describe methods for enforcing higher consistency levels. Although, for presentational simplicity, the techniques are presented as operating on binary CSPs, they are applicable to k -ary CSP - for example by making use of the dual graph or hidden-variable representations described in Section 2.1.

2.3.1 Arc Consistency

The logical next step in terms of pre-processing is to consider a single $arc(i, j)$ (see Section 2.1). Local *arc consistency* is defined as follows: both nodes x_i , and x_j must be node consistent, and for each $d_{i_a} \in D_i$, there is a corresponding element $d_{j_b} \in D_j$ such that $c(x_i, x_j)$ is satisfied by $x_i = d_{i_a}$, $x_j = d_{j_b}$. The element $d_{j_b} \in D_j$ *supports* $d_{i_a} \in D_i$. Global arc consistency holds if this condition exists between all pairs of variable nodes connected by an arc. The process of enforcing local arc consistency is embodied by **Revise()** (see Section A.5).

Figure 2.9 describes how arc consistency is enforced on the two arcs, $arc(1, 4)$ and $arc(4, 1)$ that form the constraint $c(x_1, x_4)$ from the example problem. Figures 2.9a and b present $arc(1, 4)$ before and after arc consistency is enforced. Elements P and K are removed since there are no corresponding elements in D_4 such that $c(x_1, x_4)$ could be satisfied. As can be seen, arc consistency is *directional*: the fact that $arc(1, 4)$ is consistent does not mean that $arc(4, 1)$ is also. A further operation is required to enforce consistency on the latter arc, as presented in Figure 2.9c. The element B is removed, leaving only elements in D_4 for which corresponding elements in D_1 can be found such that $c(x_1, x_4)$ may be satisfied.

Global arc consistency is obtained by enforcing arc consistency on each arc in the problem. However, **Revise()** cannot simply be applied sequentially to every arc in the problem to achieve global arc consistency, since each arc involves two variables; enforcing local arc consistency might remove the

Fig. 2.10. Global Effect of Enforcing Arc Consistency Locally

support and so violate the consistency of a separate arc, as presented in Figure 2.10. Initially, $\text{arc}(1, 3)$ and $\text{arc}(3, 1)$ are arc consistent (Figure 2.10a). $\text{Revise}(1, 2)$ is then called, which has the effect of removing the element $\{D\}$ from D_1 . This removes the support for element $\{D\}$ in D_3 , violating the consistency of $\text{arc}(3, 1)$ (Figure 2.10b).

The simplest method of achieving global arc consistency is to iterate ‘passes’ of the revision of every arc in the problem until there is no change in the constraint graph following a particular pass. This approach is embodied by the algorithm AC-1 (see Section A.6). The inefficiency of this approach is clear: all arcs are re-revised regardless of the possibility of their becoming arc inconsistent. A local change to $\text{arc}(i, j)$ can only affect the arc consistency of $\text{arcs}(h, i)$, as per the example of Figure 2.10.

Waltz’s [173] algorithm enforces arc consistency in one pass through the constraint arcs. Arc consistency is achieved by making a local subset of the arcs consistent, then performing the revisions required to maintain consistency when a new node is added to the subset. The Waltz algorithm is sometimes named as AC-2 [98]. AC-3 [98] (see Section A.7) is a generalisation of AC-2 which allows arcs to be revised in any order.

AC-3 is a popular, efficient method of achieving global arc consistency. As an example, an application of AC-3 to the problem of Figure 2.1 is described in Figure 2.11. There are no unary constraints in this problem, and it is assumed that arcs are selected from the queue in a first-in first-out manner. The globally arc consistent state of the constraint graph is presented in Figure 2.12a, and the subsequent application of Backtrack search is given in Figure 2.12b. In this simple case, global arc consistency entirely forestalls any form of thrashing.

2.3.2 Improving Efficiency in Enforcing Arc Consistency

Arc consistency is the most popular pre-processing type due to its potential for dramatically reducing the workload of subsequent tree search via relatively modest amounts of computation. AC-3 has a worst-case time complexity upper bound of $O(em^3)$ [97]; e is the number of constraint arcs, m is the

- 1 $Q = \{arc(1, 2), arc(1, 3), arc(1, 4), arc(2, 3), arc(2, 4), arc(3, 4), arc(2, 1), arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3)\}$
- 2 *Revise*(1, 2)
 - D_1 is unchanged
 - $Q = \{arc(1, 3), arc(1, 4), arc(2, 3), arc(2, 4), arc(3, 4), arc(2, 1), arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3)\}$
- 3 *Revise*(1, 3)
 - D_1 is unchanged
 - $Q = \{arc(1, 4), arc(2, 3), arc(2, 4), arc(3, 4), arc(2, 1), arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3)\}$
- 4 *Revise*(1, 4)
 - $D_1 = \{B\}$ ($\{P, K\}$ removed)
 - Add $arc(2, 1), arc(3, 1)$ to Q (both already present)
 - $Q = \{arc(2, 3), arc(2, 4), arc(3, 4), arc(2, 1), arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3)\}$
- 5 *Revise*(2, 3)
 - D_2 is unchanged
 - $Q = \{arc(2, 4), arc(3, 4), arc(2, 1), arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3)\}$
- 6 *Revise*(2, 4)
 - $D_2 = \{I\}$ ($\{U, L\}$ removed)
 - Add $arc(1, 2), arc(3, 2)$ to Q (latter already present)
 - $Q = \{arc(3, 4), arc(2, 1), arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3), arc(1, 2)\}$
- 7 *Revise*(3, 4)
 - D_3 is unchanged
 - $Q = \{arc(2, 1), arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3), arc(1, 2)\}$
- 8 *Revise*(2, 1)
 - D_2 is unchanged
 - $Q = \{arc(3, 1), arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3), arc(1, 2)\}$
- 9 *Revise*(3, 1)
 - D_3 is unchanged
 - $Q = \{arc(4, 1), arc(3, 2), arc(4, 2), arc(4, 3), arc(1, 2)\}$
- 10 *Revise*(4, 1)
 - $D_4 = \{K, J, E\}$ ($\{B\}$ removed)
 - Add $arc(2, 4), arc(3, 4)$ to Q
 - $Q = \{arc(3, 2), arc(4, 2), arc(4, 3), arc(1, 2), arc(2, 4), arc(3, 4)\}$
- 11 *Revise*(3, 2)
 - D_3 is unchanged
 - $Q = \{arc(4, 2), arc(4, 3), arc(1, 2), arc(2, 4), arc(3, 4)\}$
- 12 *Revise*(4, 2)
 - $D_4 = \{K, J\}$ ($\{E\}$ removed)
 - Add $arc(1, 4), arc(3, 4)$ to Q (latter already present)
 - $Q = \{arc(4, 3), arc(1, 2), arc(2, 4), arc(3, 4), arc(1, 4)\}$
- 13 *Revise*(4, 3)
 - D_4 is unchanged
 - $Q = \{arc(1, 2), arc(2, 4), arc(3, 4), arc(1, 4)\}$
- 14 *Revise*(1, 2)
 - D_1 is unchanged
 - $Q = \{arc(2, 4), arc(3, 4), arc(1, 4)\}$
- 15 *Revise*(2, 4)
 - D_2 is unchanged
 - $Q = \{arc(3, 4), arc(1, 4)\}$
- 16 *Revise*(3, 4)
 - D_3 is unchanged
 - $Q = \{arc(1, 4)\}$
- 17 *Revise*(1, 4)
 - D_1 is unchanged
 - $Q = \{\}$

Fig. 2.11. Example for the AC-3 Algorithm

maximum domain size. Improving this bound has been the focus of much research effort. AC-4 [116] improves the worst-case bound to an optimal $O(em^2)$ by enumerating the support for each domain element. A *counter* is associated with an arc-label pair, $(arc(i, j), d_{i_a})$, where $d_{i_a} \in D_i$. The counter's value equals the number of elements in D_j that support d_{i_a} . For each d_{j_b} , a set S_{j_b} is constructed which contains a set of pairs (i, d_{i_a}) , such that d_{j_b} supports $d_{i_a} \in D_i$.

Fig. 2.12. Impact of Enforcing Arc Consistency on Tree Search

AC-4 first constructs the aforementioned data structures, then filters domain elements. A list, L , is maintained of domain elements to be removed which is initialised with the domain elements that lack any support. Domain filtering progresses by selecting $d_{j_b} \in D_j$ from L , and removing it from D_j . S_{j_b} is then examined and the counter of each arc-label pair, $(arc(i, j), d_{i_a})$ such that $d_{j_b} \in D_j$ supported $d_{i_a} \in D_i$ is decremented. All domain elements that consequently lack support (the counter reached 0 for a particular arc) are added to L for removal.

AC-5 [38] is a generic arc consistency algorithm which depends on two procedures, `ArcCons` and `LocalArcCons` which are specified but whose actual implementation is left open. AC-3 and AC-4 can be represented by a specific implementation of these two procedures. AC-5 can also be implemented to produce an algorithm with a worst-case bound $O(em)$ for restricted constraint types.

Although AC-4 has optimal worst-case time complexity, on *average* it performs less well than AC-3 [170]. It also has a high space complexity, which is a drawback for large problems. Hence, it is common for AC-3 to be used in preference to AC-4, especially for large problems [11]. AC-6 [11] maintains the optimal worst-case time complexity of AC-4, but improves the space complexity. Whilst AC-4 records all support for a domain element, AC-6 checks for only one supporting value per domain element per constraint. Further support is looked for only when the current support is filtered out.

AC-6 does not use the arc-label counters maintained by AC-4. S_{j_b} now contains the set of pairs (i, d_{i_a}) such that $d_{j_b} \in D_j$ is the first element of D_j that supports $d_{i_a} \in D_i$ on $arc(i, j)$. The initialisation process searches for the minimum support required for each domain element, and adds to L any elements which lack this support. Subsequently, for each $d_{i_a} \in D_i$ on L alternative support is required for any element, $d_{j_b} \in D_j$ that depended on it. New support is found by checking forward from d_{i_a} in D_i until another

element which supports d_{j_b} is found. If there is no new support, d_{j_b} is added to L and filtered.

The *AC-inference* schema [12] exploits knowledge about the properties of constraints to reduce the cost of consistency checking by *inferring* support for domain elements. A specific instance of this schema, AC-7, makes use of a property common to *all* binary constraints: *bi-directionality*. That is, $d_{i_a} \in D_i$ supports $d_{j_b} \in D_j$ if and only if $d_{j_b} \in D_j$ supports $d_{i_a} \in D_i$. Hence, if support is found in one direction it can be inferred in the other. This algorithm is also the basis of *Lazy Arc Consistency* [142], which specifically aims to detect global inconsistency of the problem with minimum effort.

2.3.3 Path Consistency

Path Consistency [98] is a more sophisticated level of consistency. Local path consistency holds if, given a path of length l through the nodes $(x_1, \dots, x_l)$, for all values $d_{1_a} \in D_1$ and $d_{l_b} \in D_l$ such that $c(x_1)$, $c(x_l)$, and $c(x_1, x_l)$ hold, there is a sequence of values: $d_{2_c} \in D_2, \dots, d_{l-1_c} \in D_{l-1}$ such that $c(x_2), \dots, c(x_{l-1}), c(x_1, x_2), c(x_2, x_3), \dots, c(x_{l-1}, x_l)$ hold.

Global path consistency holds if any pair of assignments which are consistent with a direct constraint between two variables are also consistent with all paths between the two nodes. If every path of length 2 in a *complete* constraint graph is path consistent, then path consistency holds globally [117]. In a complete graph, a constraint arc connects each possible pair of nodes. An incomplete graph can be made complete by creating ‘True’ constraints that allow all possible combinations of assignments between two variables. Path consistency is enforced by modifying the direct constraint between the start and end points of a path to disallow all assignment pairs which violate the required condition. ‘True’ constraints may be updated, effectively adding a new explicit constraint to the problem.

Figure 2.13 presents an example which is part of the example problem given in Figure 2.1. Figure 2.13a describes the initial state of this sub-problem. Consider the assignment $\{x_1 = P, x_3 = U\}$, which satisfies the constraint $c(x_1, x_3)$ between the two variables. There exists no element of D_2 that, when assigned to x_2 , can satisfy the constraints amongst all three variables, hence this pair of assignments can not be part of any solution. This situation is avoided by modifying $c(x_1, x_3)$ such that the assignment pair $\{x_1 = P, x_3 = U\}$ is disallowed irrespective of the variable instantiation order. The updated problem is presented in Figure 2.13b. Note that enforcing arc consistency does *not* disallow this assignment pair.

The first path consistency algorithm was presented in [117], and is equivalent to PC-1 [98]. Mackworth [98] improved the efficiency of PC-1 using a revision-based method, as per AC-3, to create PC-2. PC-3 [116] improves efficiency via the enumeration of support, as per AC-4. PC-3 is, however, not completely correct. PC-4 [76] is a correct version. Further improvements can be found in [25]. Path consistency removes more of the workload of subsequent

Fig. 2.13. Effects of Enforcing Path Consistency

tree search, but with a significant computational overhead. The potential improvements must be weighed against this overhead.

2.3.4 K-Consistency

More complex pre-processing consists of modifying or adding k -ary constraints to disallow inconsistent assignment combinations [26, 98, 116]. *K-consistency* [54] is a general theory of consistency levels. A CSP is k -consistent if, given assignments to $k - 1$ variables satisfying all constraints among them, a value may be chosen for any k th variable satisfying constraints among all k variables. *Strong k -consistency* is defined as j -consistency for all $j \leq k$ [55]. Node and arc consistency correspond to strong 1 and 2-consistency respectively.

Consistency levels higher than 2-consistency typically involve adding or modifying at least binary constraints (as was seen when considering path/3-consistency in the previous section), and are significantly more expensive to enforce than node or arc consistency. A sufficient consistency level may be enforced such that no backtracking is required, i.e. for *backtrack-free* search [55]. It is first necessary to define some terms relating to the structure of a constraint graph. An *ordered constraint graph* is an arrangement of the constraint graph nodes in a fixed linear order, equivalent to maintaining a fixed instantiation order during tree search. The *width* at an ordered node is the number of constraint arcs that link it back to a previous node in the order. The width of an ordering is the maximum width of any of its nodes, and the width of a constraint graph is the minimum width of all its orderings. The following theorem [55] specifies the necessary conditions for backtrack-free search:

- There exists a backtrack-free search order for a CSP if the level of strong consistency is greater than the width of the graph.
- A search order is backtrack-free if the strong consistency level is higher than the width of the corresponding ordered constraint graph.

Consider, for example, a tree-structured constraint graph. If this type of graph can be made arc consistent, then subsequent search will be backtrack-free. Freuder provides algorithms to determine the width of a constraint graph and to enforce k -consistency [55]. In addition Cooper [26] presents a worst-case optimal algorithm to enforce k -consistency in a constraint graph. Unfortunately, these algorithms are very time consuming. Usually, it is much more expensive to enforce the necessary conditions for backtrack-free search than it is to apply even Backtrack to the original CSP. In addition, as noted above, to enforce a level of above 2-consistency it is often necessary to add new constraints to the constraint graph. The width of the constraint graph must then be re-determined to see if a higher level of consistency is required, and so on. It is this high cost that makes it doubtful whether, overall, it is worth using this process to eliminate backtracking search.

2.3.5 Practical Consistency Enforcing

The cost of enforcing a level of consistency equivalent to path-consistency or higher has led to the development of ‘practical’ consistency enforcing techniques which are stronger (i.e. filter more domain elements) than arc consistency, but do not alter the constraint graph. This is an advantage since, as noted in [29], even enforcing path-consistency can often result in a complete constraint graph. Hence, there are many more constraints to check as well as a sharp increase in space complexity.

One example of such a technique is *inverse* consistency. A problem is k -inverse consistent if a single assignment can be extended to a consistent assignment involving $k - 1$ variables [53]. Any domain element failing this test can be removed. Various levels of inverse consistency can be enforced, with the cost of doing so increasing polynomially with k . A specialised variant of this techniques is known as *neighbourhood inverse consistency*[57], which adapts the level of inverse consistency enforced on a particular variable’s domain to the number of immediately adjacent variables (i.e. those connected to the current variable in the constraint graph).

A second example is *singleton* arc consistency [130]. This technique is based on the following observation: if it is true that the assignment of a domain element d_{i_j} to x_i is consistent, then the sub-problem formed by restricting D_i to d_{i_j} can be made arc consistent. Any domain element for which this property does not hold can be filtered.

Experimental analysis made on random problems [29] revealed that singleton arc consistency offers a good ratio between effort required and amount

of filtering achieved. Also, neighbourhood inverse consistency was found to be very powerful in terms of filtering domain elements, but also very time-consuming.

2.3.6 Directional Pre-Processing

To improve the efficiency of pre-processing a CSP, Dechter and Pearl [35] proposed *directional* pre-processing algorithms that exploit the order of instantiation that a tree search algorithm will subsequently use. This class of algorithms therefore avoids enforcing consistency on many constraint arcs that are not necessary for the search process. Consider the ordered constraint graph corresponding to an instantiation order: it is sufficient to ensure that at any node the domains of any future variables it is linked to contain at least one value that is consistent with the current assignment. Assignments inconsistent with previous (instantiated) nodes will be avoided automatically by Backtrack.

This observation leads naturally to the development of a directional arc consistency algorithm (DAC). Variables are processed in the reverse order to which they will be instantiated by Backtrack, ensuring that all arcs that lead to the current level from past variables are made arc consistent. DAC revises each arc in the forward direction only once: for arc consistency on $\text{arc}(h, i)$ to be lost some values would have to be filtered from D_i . This will never happen since arc consistency forward from level i has already been achieved; no further values will be removed from D_i . This is a significant advantage over full arc consistency algorithms that must re-revise arcs that are possibly affected by the removal of any domain value. Indeed, when applied to a tree structured constraint graph, DAC will render the problem backtrack-free in $O(m^2)$ [30], leading to a solution procedure of time complexity $O(nm^2)$.

Directional arc consistency extends naturally to directional path consistency (DPC). DPC directionally enforces path consistency for paths of length 2, thus ensuring path consistency for the whole constraint graph, with respect to the instantiation order. This process can continue towards k -consistency by enforcing just enough consistency at each node to ensure a backtrack-free search (*Adaptive Consistency* [35]). ADAPT also processes the nodes in reverse instantiation order. The width of each node is examined and the appropriate amount of consistency with connected past variables is enforced.

DAC, DPC, and ADAPT were compared experimentally against Backtrack and Backjump [34]. Although the worst case analysis of Backtrack is exponential, the average case is much better. It is therefore not generally cost-effective to do even the restricted pre-processing offered by ADAPT or DPC. However, DAC appears to perform the right amount of pre-processing to ameliorate the performance of Backtrack and will generally perform better even than Backjump.

Fig. 2.14. Problem Decomposition by a Hybrid Algorithm

2.4 Hybrid Tree-search Consistency-enforcing Algorithms

A *hybrid* CSP algorithm [121, 127] embeds a consistency-enforcing component within a tree search algorithm. The motivation is to produce a technique which retains the power of each component approach to create a more powerful algorithm. At each search node, consistency is enforced with respect to the current variable assignments. As further assignments are made, the problem is divided into simpler sub-problems (since more of the original variables have fixed assignments) within which consistency is enforced (Figure 2.4). Here, the original problem has been split into two sub-problems where the first is committed to the assignment $x_1 = d_{1-1}$, and the second is committed to $x_1 = d_{1-2}$. Backtracking occurs when any variable domain becomes empty as a result of consistency enforcing based on the current assignments: the sub-problem is inconsistent given these assignments. When a hybrid backtracks from x_i to x_h , it must return the state of the problem, in terms of constraints and variable domains, to the state it was in before the last assignment to x_i i.e. before any consistency processing was done at level h .

The tradeoff is between pruning more nodes from the search tree versus performing an increasing amount of work per node. There is some debate as to how much consistency enforcing is cost effective at each search node. It was thought to be not cost-effective to even apply arc consistency as part of a hybrid algorithm [90, 121], hence the development of *partial* arc consistency procedures (see below). More recently, however, it has become clear that this is not the case [138] - especially for more difficult problems.

This section discusses several hybrid algorithms employing different levels of sophistication for both the tree search and consistency enforcing components. Due to their nature, hybrid algorithms require an understanding of both consistency enforcing and tree search techniques. The reader is referred to Sections 2.2 and 2.3 for the necessary background to this section.

Fig. 2.15. Consistency Checks Required by Partial Arc Consistency Algorithms

2.4.1 Partial Arc Consistency

Nadel [121] names four partial arc consistency procedures $AC-i$, where i is one of the fractions $1/5$, $1/4$, $1/3$, or $1/2$, with the fraction approximating the degree of consistency enforced. Each procedure may be combined with tree search and returns **True** if a domain wipeout occurs and **False** if not. Since $AC-1/5$ is a reformulation of Backtrack, it is not discussed. Figure 2.15 (adapted from [121]) presents the consistency checks made by three partial arc consistency algorithms. A complete graph consisting of the four variables x_1, x_2, x_3 , and x_4 is assumed. In each case, the checks follow an assignment to x_1 .

The $AC-1/4()$ algorithm (see Section A.8) corresponds to the procedure **Check-Forward()** [77] as the core part of the popular Forward Checking algorithm. $AC1/3$ corresponds to the **Partial-Look-Future()** component of Haralick and Elliot's [77] Partial Lookahead algorithm. The $arcs(j, k)$, where $i \leq j < k \leq n$, i is the current search level, are revised once in ascending order. For each x_j , support is checked for each $d_{j_b} \in D_j$ in the domains of each variable in x_j 's own future (see Figure 2.15b). $AC-1/2$ (also known as **Look-Future()** as part of Haralick and Elliot's Full-Lookahead algorithm) enforces more consistency by revising all the constraint arcs concerning the current and future variables (see Figure 2.15c). Each arc is revised once, in the instantiation order.

Nadel [121] also provides various algorithms to achieve full arc consistency at each node of the search tree. *Really Full Lookahead* uses a combination of $AC-1/4$ and $AC-3$ on future nodes only, exploiting the fact that past variables are instantiated by the search process, effectively reducing their domains to a single element.

When combining consistency enforcing techniques and tree search it is necessary to maintain the data structures required by both the arc revision procedure and the tree search algorithm. Prosser [127] presents such a framework using an array to keep track of domain changes made by arc revision, which must be undone when backtracking.

2.4.2 Relative Evaluation

Haralick and Elliot [77] compared hybrids including Forward Checking, Partial Lookahead and Full Lookahead against Backmark and Backtrack using the n -queens problem.¹ Forward Checking is shown to be the most efficient for this problem. Nadel [121] also uses the n -queens as well as the *confused* n -queens problem (each queen must attack every other) in an empirical comparison between Forward Checking, Partial Lookahead, Full Lookahead, Really Full Lookahead, Backtrack, Backjump and Backmark. Backmark and Forward Checking were the most efficient algorithms for both problems.

In creating hybrids involving more sophisticated tree search components, Prosser [127] was able to measure their performance against less sophisticated hybrids as well as ‘pure’ tree search algorithms. The ZEBRA² problem was used to compare Backtrack, CBJ, Backmark, BMJ, BM-CBJ, Forward Checking, FC-BJ and FC-CBJ in terms of consistency checks and run time. In common with previous results, hybrids involving Forward Checking were the most successful. Specifically, FC-CBJ was clearly the most efficient algorithm for this problem, as might be expected given its combined power of enforcing partial arc consistency *and* avoiding thrashing.

A theoretical evaluation made by Kondrak and van Beek [88] examined the performance of Forward Checking, FC-BJ and FC-CBJ against pure tree search algorithms and hybrids of Backmark. Their results show that, with respect to the number of search nodes visited, FC-CBJ is ranked the best. However, in terms of consistency checks, their algorithm BM-CBJ2 was shown to be the most efficient. Sabin and Freuder [138] compared their MAC algorithm (which maintains full arc consistency at each search node via the AC-4 algorithm) against Forward Checking. Particular attention was paid to difficult problems, where MAC often performed at least an order of magnitude better.

As with all CSP algorithms, the structure of a problem is crucial to the ability of a hybrid algorithm to solve it. It is therefore difficult to choose one overall ‘best’ combination of enforcing consistency and tree search. Although many sets of tests have been carried out, as described above, an exhaustive comparison on a wide range of problems is needed to determine the best hybrid algorithm in the general case.

¹ The n -queens problem involves placing n queens on an $n \times n$ chess board such that no queen is attacking any other.

² The ZEBRA problem is defined on 25 variables: 5 houses, 5 brands of cigarettes, 5 nationalities, 5 pets and 5 drinks. Various constraints are specified such as ‘The Englishman lives in a red house’.

2.5 Heuristics

Certain operations performed by a constraint-based problem solver, may be ordered heuristically such that, generally, less work is required to find a solution. These operations are: the order in which variables are assigned, the order in which domain elements are assigned to the current variable, and the order in which consistency checks are made. For many years the *fail-first* principle, i.e. attempting to establish failure in the form of a dead end in the search as soon as possible, was thought to be a sound strategy for heuristics to follow. More recently, this has been refuted. Smith and Grant [150] created increasingly sophisticated heuristics to select the variable most likely to fail first. They found that early failure was encouraged, keeping the search branch length low, but that so many failures were encountered that the overall search effort increased dramatically. An alternative hypothesis is that of focusing on the most constrained part of the problem [136]. In so doing, the branching factor of the search is minimised, reducing overall search effort.

The variable instantiation order can have a marked effect on efficiency. Consider, for example, Figure 2.16 which describes how Backtrack solves the example problem with the order of instantiation reversed. Various heuristics exist to choose an order which will have a favourable effect on subsequent search. These fall into two categories: *static* variable ordering heuristics examine the input problem prior to search and produce a fixed instantiation order which is used throughout; *dynamic* variable ordering heuristics examine the state of the problem during search, selecting the ‘best’ variable (in terms of hopefully reducing search effort) to instantiate next. In general, this results in different orders of instantiation in the different branches of the search tree. Dynamic variable ordering heuristics are most effective when used with hybrid tree search and consistency enforcing algorithms (see Section 2.4) since they can take into account the changes in sub-problem structure following consistency enforcing.

A fixed instantiation order is required by algorithms such as Backmark, or where directional pre-processing is used (see Sections 2.2.4 and 2.3.6). The *maximum-degree* heuristic [34] constructs an instantiation order with variables involved in the most constraints (i.e. with the highest *degree* in the constraint graph) placed highest. The *minimum-width* heuristic [55] enforces the instantiation order with the minimum width (see Section 2.3.4). The *maximum cardinality* [34] heuristic chooses the first variable at random and adds variables to the instantiation order by picking one which is connected to the largest group among the variables already selected. Fixed instantiation order heuristics were tested on random problems [34]. The maximum cardinality heuristic proved to have the most beneficial effect.

The simplest dynamic variable ordering heuristic is to continually select the variable with the smallest remaining domain [132, 157]. Haralick and Elliot [77] present evidence supporting this heuristic in tests involving the *n*-queens problem. It was also found to be particularly effective in conjunc-

Fig. 2.16. Impact of a Better Instantiation Order on the Problem of Figure 2.1

tion with CBJ [61]. A slightly more refined approach is embodied by the Brelaz heuristic [19], which also selects the smallest remaining domain, but breaks ties by choosing the variable with the largest degree in the future sub-problem. Given a fully connected graph (as per the example problem), the Brelaz heuristic reduces to the simple smallest-domain heuristic. On sparse and moderately densely connected random constraint problems, the Brelaz heuristic has been shown to substantially outperform the smallest-domain heuristic [66]. It has the advantage over the more sophisticated techniques described below that it is simple to calculate.

The *Rho* heuristic [66] is, for example, a more sophisticated (and more costly) dynamic variable ordering heuristic. It seeks to branch into the sub-problem with the maximum solution density, ρ . A related heuristic seeks to branch into the sub-problem with the largest number of expected solutions [66]. This heuristic tends to maximise the solution density and follow the smallest-domain heuristic. Also, the *Kappa* heuristic [67] seeks to branch into a sub-problem which minimises a measure, κ , of constrainedness. Many of these heuristics approximate κ [67], which, although powerful, is expensive to calculate precisely. On random problems, these three heuristics were found to scale better than the simple smallest-domain or Brelaz heuristics [66].

Value-based heuristics order the assignment of values to a variable. If a solution exists it can be found immediately by picking the ‘right’ assignment at each stage. One heuristic seeks an approximation to the most constrained value by selecting values that appear in the fewest allowed constraint tuples [104]. By contrast, *Lookahead value ordering* [62] selects the value that conflicts least with future domain elements in an attempt to maximise the chance of finding a solution on the current search branch. Geelen [64] combines a variable and value ordering heuristic to select the most promising value for the least promising variable. Prosser [129] examined the behaviour

of the smallest-domain, Brelaz, Geelen and Kappa heuristics to discover how each distributed effort at different depths of the search tree. The more informed heuristics seemed to spend most time at a shallow depth. Once the first few ‘hard to satisfy’ variables had been consistently assigned the rest of the assignments could be made straightforwardly. Equally, on insoluble problems, the insolubility was detected by looking at the few most constrained variables.

It is of obvious benefit to perform as few constraint checks as possible. For a consistent assignment, the order of checks is irrelevant - all required checks are made before the assignment is shown to be consistent with the current partial assignment. If an assignment is inconsistent, much work can be avoided if a constraint check that fails is performed early. One technique chooses the constraint that disallows the most assignments (i.e. is the most constraining) for the current variable. For the n -queens problem, Haralick and Elliot [77] show that it is beneficial to perform checks against the variable representing the previous column first since this constraint disallows the most assignments to the current variable.

2.6 Conflict Recording

During search, a *dead end* occurs when a consistent assignment cannot be found for the current variable. That is, the assignments to $x_1, \dots, x_{i-1}$ form a *conflict set* with x_i . This information is used by recording one or more constraints (often termed as *nogoods*) which aid future search by disallowing the inconsistencies that led to this dead end. Consider Figure 2.1 and the search tree generated by Backtrack (Figure 2.4). At the first dead end, the set $CS = \{x_1 = P, x_2 = U, x_3 = V\}$ conflicts with x_4 . Recording CS is only useful if the CSP will be queried again: this exact assignment set will not re-occur during this search. However, CS probably contains subsets in conflict with x_i . *Minimal conflict sets* [20] contain no other conflict sets and are responsible for the current conflict. The search process benefits if these minimal sets are prevented from re-occurring.

Conflict recording can be viewed as a form of *learning* [30]: useful information uncovered by the problem solver is recorded for later use. *Deep learning* extracts all possible information from a dead end, finding all minimal conflicts. It is embodied by *Dependency-Directed Backtracking* [153] and is commonly used in *truth-maintenance systems* (e.g. [39]). Deep learning is relatively expensive: each dead end presents a new opportunity to add constraints. Hence, a prohibitively large proportion of the search space is recorded.

Shallow learning [30] limits the amount of work done at a dead end, say x_i , by removing all variable assignments from the conflict set which are irrelevant (i.e. consistent with all possible assignments) to x_i . *Graph-based shallow learning* uses the constraint graph to test (incompletely) for irrelevancy: x_h

is irrelevant to x_i if the variables are not adjacent in the graph. Since the example given in Figure 2.1 is completely connected, this approach does not rule out any elements of the conflict set. *Value-based learning* removes *all* irrelevant variables from the conflict set, while *Jump-back learning* uses the conflict set recorded by the CBJ algorithm to avoid extra work [60].

The *Stubbornness* approach [144] makes unnecessary mistakes to extract the maximum information. Adding too many constraints, however, hinders search by increasing the number of constraint checks required. Recorded conflicts should therefore have a high probability of being useful to further search. To be useful, a conflict must re-occur, allowing a greater amount of pruning of the search tree. To accomplish this, Dechter [30] suggests limiting the size of the conflicts recorded. *First-order learning* records conflicts concerning just one variable. *Second-order learning* adds constraints of up to two variables. First- and second-order learning can be used independently of the depth of learning. The NR_i algorithms [143] (where i is the order of conflict recording used) justify recorded conflict sets in terms of the constraints responsible for the conflict. This is especially useful in a dynamic context (see Section 2.10), where a conflict will persist in an evolving problem if its justification continues to exist.

Dechter [30] compared shallow and deep, first and second order learning. On easy problems learning has little effect, but on more difficult problems large gains in performance are seen. It is not uniformly the case that the strongest form of learning gave the most benefit. In some cases the overheads outweigh the benefits. Frost and Dechter [60] compared graph-based shallow learning, value-based shallow learning, jump-back learning and deep learning on random problems. The conflict recording techniques were particularly effective on large or hard instances that provide more dead ends from which conflict sets can be extracted. Jump-back learning was found, on average, to be the most effective scheme.

Schiex and Verfaillie [143] compared several variants of the NR_i algorithms against naive Backtrack, Forward Checkings, and shallow learning using the Zebra problem. In this study, the NR_i clearly perform the best in terms of consistency checks made and time spent. Furthermore, when solving the same problem again but retaining the conflicts established in the first attempt at solution, the justified conflicts recorded by the NR_i algorithms led to a substantially larger decrease in search effort than for shallow learning.

2.7 The Phase Transition in CSPs

Unsurprisingly, some problem instances are harder than others. However, the degree of difficulty is not merely a function of the size of the problem, but depends also on its internal structure. A *phase transition* can be observed where problem instances change from being easy to solve, to being difficult to solve, back to being easy to prove to be insoluble [22]. Given a problem

Fig. 2.17. The Phase Transition

of a particular size and connectivity (the proportion of variables connected by a constraint), the phase transition can be observed as the tightness (the proportion of tuples disallowed by each constraint) is increased, as presented in Figure 2.17.

The region over which the phase transition from solubility to insolubility takes place is known as the *mushy region* [149]. The analogy is with a material passing from a solid to a liquid state. The width of this region increases with the connectivity of the constraint graph, as does the maximal average search effort [128]. It has been demonstrated empirically [128] using random problem instances that the average search effort is maximal close to the point where half the instances are soluble, and half insoluble.

When solving problem instances at the phase transition, it has been found [138] that algorithms which work harder to spot approaching dead ends (arc consistency enforcing hybrids such as MAC, for example) show a marked improvement over less informed approaches. The insight into the expected difficulty of solving a particular problem provided by the concept of the phase transition is used in Chapter 4 when constructing problems to test the new algorithms developed herein.

2.8 Graph-Based Methods

Various techniques exist which directly exploit the topology of the constraint graph to solve the corresponding CSP. Four such graph-based methods are described below. The first of these, the *Invasion Procedure* [145] successively ‘conquers’ an increasing portion of the constraint graph in order to compute the set of solutions. The other three approaches all make use of the observation that a tree-structured CSP can be solved in linear time (see Section 2.3.4): a tree-structured graph is found or created given an input CSP.

Fig. 2.18. Solution Graph for the Problem of Figure 2.1

2.8.1 The Invasion Procedure

The *Invasion Procedure* [145] is an approach designed to find all the solutions to a CSP. It is based on a sequence of subgraphs, $G_1, \dots, G_n$, called an *invasion* of an n -node constraint graph, G . The number of nodes in G_i must be i , and G_i must be a subgraph of G_{i+1} . The *front*, F_i , is the set of nodes of G_i that are adjacent to the nodes not in G_i , and the *front length*, f_i is the number of nodes in F_i . The front length of an invasion is the maximum of its individual front lengths. Consider the example presented in Figure 2.1. A typical invasion might be constructed as follows: $G_1 = \{x_1\}$, $G_2 = \{x_1, x_2\}$, $G_3 = \{x_1, x_2, x_3\}$, $G_4 = \{x_1, x_2, x_3, x_4\}$. The fronts associated with each subgraph are: $F_1 = \{x_1\}$, $F_2 = \{x_1, x_2\}$, $F_3 = \{x_1, x_2, x_3\}$, $F_4 = \{\}$. Note that F_n is always empty since G_n contains the whole constraint graph.

Given such an invasion, this algorithm constructs a *solution graph*, S , which represents the set of solutions to the original CSP. The nodes of S are grouped into sets, $S_0, \dots, S_n$ such that S_i ($i \neq 0, i \neq n$) is a set of value assignment tuples for the front F_i . S_n and S_0 contain single nodes, s_n and s_0 that represent the start and end points of a solution respectively. A directed arc from a tuple of S_i to a tuple of S_{i-1} is labelled with an assignment to x_i (where x_i is the variable in G_i which is not in G_{i-1}). This assignment must satisfy all constraints involving x_i and the variables in F_{i-1} , given the assignments represented by the particular tuple of S_{i-1} . A path from s_n to s_0 , with instantiations of x_i corresponding to each arc travelled, corresponds to a single solution.

Fig. 2.19. Effects of Instantiating a Cycle Cutset

The solution graph to the problem in Figure 2.1 is presented in Figure 2.18. S is constructed one tuple set, S_i , at a time in the order $S_1, \dots, S_n$. When constructing S_i , each tuple of the previous tuple set, S_{i-1} , is considered in turn along with all potential assignments to x_i . A tuple is added to S_i for each assignment to x_i which satisfies all constraints with a particular assignment tuple in S_{i-1} . An arc is added from each newly created tuple in S_i to the tuple of S_{i-1} it is based on, labelled with the value of x_i .

The time complexity of this algorithm is $O(em^{f+1})$, where e is the number of constraints, m is the maximum domain size and f is the front length of the invasion. Hence, the efficiency of this algorithm depends on finding a good invasion i.e. one with a small front length. Unfortunately, good invasions cannot be guaranteed to exist - a complete n -node graph (such as the example problem) only has invasions of front length $n - 1$.

2.8.2 The Cycle-Cutset Method

The *cycle-cutset* method [30] is founded on the fact that instantiating a variable fixes its value and removes it from further consideration for this branch of the search tree. This effectively changes the connectivity of the remainder of the constraint graph. The cycle cutset of a constraint graph is the set of nodes which, once removed (or instantiated), leaves a tree-structured graph. Consider the example presented in Figure 2.19a (the exact nature of the constraints and variable domains are not significant). Instantiating x_2 ‘blocks’ the path through this variable such that there is only one path between any two variables. The resulting constraint graph is given in Figure 2.19b, with the now fixed variable x_2 replicated for each adjacent variable.

If the set of instantiated variables are equivalent to a cycle cutset, the remaining graph is tree-structured and may be solved very efficiently. This method therefore makes a locally consistent assignment to the variables of a cycle cutset and solves the remaining problem quickly. If no solution can be found, another consistent assignment to the cycle cutset must be found and the remainder solved again, and so on.

Finding a minimal cycle-cutset is itself an NP-complete problem. In light of this, it is suggested that the cycle-cutset method is embedded in a tree search algorithm such as Backtrack. It would then be necessary to monitor the

Fig. 2.20. Non-Separable Components

connectivity of the constraint graph and, if the set of instantiated variables constitute a cycle-cutset, the rest of the problem could be solved quickly. In comparisons with Backtrack on random problems, Backtrack with cycle-cutset was found on average to improve performance by around 20 percent. When combined with Backjump the average improvement was 25 percent over the naive version.

2.8.3 Non-separable Components

Non-separable components [35, 55] can also aid in solving a CSP. A *separation node*, x_i , in the constraint graph is such that all paths between further nodes must pass through it. A *separable* graph contains a separation node and a *non-separable* graph does not. A non-separable component is a subgraph with no separation nodes. An example is presented in Figure 2.20, where $Comp_i$ indicates a non-separable component. Even [45] shows an $O(e)$ algorithm to find all non-separable components and separation nodes.

The next step, having discovered all non-separable components, is to find all possible solutions for each of them (using any suitable method). The entire CSP may then be solved by concatenating the solutions to each component, such that the assignments to each separation node are compatible between adjacent components. Dechter and Pearl [35] present an $O(nm^r)$ algorithm that embodies this approach, where n is the number of variables and r is the size of the largest component.

2.8.4 Tree-Clustering

Tree-clustering [36] transforms arbitrary n -ary CSP into an acyclic form in the dual graph representation by forming *clusters* of variables whose interaction is tree-structured. An n -ary CSP is acyclic if its primal graph (see Section 2.1 for primal and dual graphs) possesses the following properties:

- *Chordal*: Every cycle of at least four nodes in length contains a *chord* - an arc which joins two nodes which are not consecutive in the cycle.

Fig. 2.21. Creating a Chordal Primal Graph

- *Conformal*: Each of its *maximal cliques* (completely connected node subsets) correspond to a single constraint in the CSP.

A triangulation algorithm [158] can be employed to enforce these conditions using two steps: a) Compute the maximum-cardinality ordering (see Section 2) of the nodes, and b) Recursively add arcs between nodes that are connected by nodes higher in the ordering. Figures 2.21a and b (from [36]) respectively present the primal graph of a CSP before and after such a process. The maximal cliques of this graph are the clusters which constitute the constraints of the transformed CSP (see Figure 2.22a).

To create a tree from this representation it remains to remove *redundant* arcs from the graph, i.e. those whose removal has no effect on the set of solutions. In the dual graph representation, an arc is redundant if the set of variables it is labelled with is shared by all arcs on a separate path between its endpoints. Figure 2.22b describes one way to create a tree from the initial problem. The complexity of this procedure is $O(nrm^r)$, where r is the size of the largest maximal-clique.

2.9 Extending the CSP Framework

Classical constraint satisfaction techniques, as presented thus far, support only *hard* constraints specifying exactly the allowed variable assignment combinations. A solution to a classical CSP must satisfy every constraint. This

Fig. 2.22. Creating a Tree in the Dual Graph

Fig. 2.23. An Example of Flexible CSP

rigidity of structure has the advantage of simplicity: less information is necessary to describe the problem, which in turn means that problem-solving procedures are simpler. There are, however, some problems inherent in classical CSP. For example, when an over-constrained problem (i.e. which has no solution) is encountered, some constraints must be *relaxed* (removed or weakened) before a solution to a less-constrained (but still interesting) problem can be found. Without an indication of the relative importance of each constraint, however, it is difficult to do this consistently such that a useful solution will be found.

Real problems in general are not easily described in such definite terms as classical CSP requires. It is common for some kind of *flexibility* to be inherent in the problem. A simple example is the expression of *preferences* among the set of assignments, either to the whole problem or local to an individual constraint. *Prioritised* constraints are also useful in the case of an over-constrained problem. If constraints with a lower priority are relaxed it is more likely that the eventual solution will be useful. In addition, for real-time tasks finding flexible solutions may be the only option. Classical techniques find a perfect solution or no solution at all, presenting a serious problem if time runs out for the problem-solver. A flexible system could return the best solution so far, embodying an *anytime* algorithm, as noted in [56]. There may also be uncertainty concerning the problem structure. The set of constraints that define the problem may not be precisely known, in which case the CSP is *ill-defined* [47].

Consider the example presented in Figure 2.23, which is a prioritised version of the original example of Figure 2.1. The constraint arcs are annotated with their relative priority in the range $[0, 1]$, with the endpoints denoting totally irrelevant and imperative constraints respectively. There is an additional imperative constraint $c(x_1, x_2, x_3, x_4)$ that stipulates that the word formed by the sequential assignment to these variables must be a colour. There are two possible words that satisfy this last constraint: *blue* and *pink*, but which

Fig. 2.24. Rating Optimisation by Cost Minimisation for Flexible CSP

constitutes a valid assignment such that all other constraints are satisfied? In fact it is not possible to find a perfect solution to this problem, but it *is* possible to find a 'best' solution by selectively relaxing the least important constraints, as will be shown.

Without relative priorities it would not be clear at all which constraint to remove, disallowing a concept of 'better' or 'worse' solutions. This type of flexibility is useful for real problems, where complete constraint satisfaction is often not possible. Flexible constraints are representative of the inherent *softness* found in real problems, and are essential in properly modelling them.

There are several existing extensions to classical CSP which support some types of flexibility [42, 58]. The common ground between them is the use of an operator to aggregate the satisfaction level of individual constraints. An overall rating for a complete variable assignment may then be computed and optimised to find the 'best' solution to the problem, i.e. the assignment which offers the optimal balance of satisfaction/violation over the entire set of constraints. The interpretation of the rating varies between different approaches. In some cases it is interpreted as counting the cost of any necessary constraint violations, which should be minimised (see Figure 2.24), while in others it should be maximised to find the best solution to the problem. There are two main methods of combining rating information, as will be described below: *additive* methods [52, 58] and *priority maximisation* methods [42, 140].

2.9.1 Extending Tree Search

In order to support flexible constraints, some modification of classical CSP solution methods must be made. For example, classical Backtrack can prune a branch of the search tree whenever a constraint is violated. Since flexible CSP tolerates some constraint violation it is not possible to use simple Backtrack, but the intuition remains the same: a partial solution must be discarded

as soon as it becomes clear that all possible extensions to full solutions are unacceptable.

Branch and bound [93], as the analogue of classical Backtrack, is a very common technique for solving flexible CSPs [42, 58] (see Section A.9). The essential idea of this approach is to keep a record of the best solution found so far; this allows the satisfaction procedure to abandon partial assignments as it becomes obvious that they cannot be extended to full assignments that are better than the current best solution. As the search progresses, the problem solver rates the current partial solution using one of the rating optimisation methods. When the first complete assignment is found, its rating is recorded as α , the *necessary bound* on future solutions. From this point on, partial assignments are pruned as soon as their rating reaches α . If a complete assignment is found with a better rating than α , α is updated to this new value and the process continues. The algorithm terminates when it is clear that none of the remaining avenues of search can lead to a solution that is better than the current best.

There are two further ways to improve basic branch and bound [58]. First, α , can be preset based on some prior knowledge about the search space. Hence, more of the space can be pruned. Secondly, a *sufficient bound*, S , can be used so that the algorithm will terminate when a solution of quality equivalent to S or better is found. Again, this allows the algorithm to terminate earlier if possible, exploring less of the search space. Branch and bound is explored further in Chapter 3 as part of a dynamic flexible combination algorithm.

2.9.2 Solution via Graph Decomposition

A general flexible solution method, based on dynamic programming [10] which exploits graph decomposition is shown in [9]. The constraint graph representing the whole problem is decomposed into subgraphs for which solutions are sought. An optimal solution is sought with respect to each possible value for the nodes which connect a subgraph to the rest of the graph. Solution synthesis then combines subgraphs via the connecting nodes in order to generate a solution to the whole problem. There is a clear relationship between this approach and that of the non-separable components method for solving classical CSPs (see Section 2.8.3).

An example is given in Figure 2.25. Here, assignments to x_j, x_k, x_l are optimised with respect to each possible assignment from D_i to x_i which connects this subgraph to the rest of the graph. Depending on the assignment made to x_i as a result of solution synthesis the optimal assignments to the other three variables are immediately available.

Fig. 2.25. Subgraph. Optimise with Respect to Assignments to x_i (Adapted from [9])

2.9.3 Additive Flexible CSP

A common extension to the CSP framework is characterised by the use of a summation operator to compute a rating for a partial assignment. This rating varies according to how well the assignment satisfies individual constraints, indicating its overall quality. The rating may then be used within a branch and bound algorithm (see Section 2.9.1) to find the optimal solution. The interpretation of the summation varies between different approaches: it is sometimes interpreted as counting the cost of any necessary constraint violations, in other cases it should be maximised to find the best solution to the problem, as will be discussed.

Weights and Ratings

Possibly the simplest additive method sums the number of satisfied constraints given a partial assignment to produce a simple rating of its quality. This approach is embodied by MAX-CSP [171]. A natural extension to this initial scheme associates a *weight* with each constraint. This value expresses the relative priority or importance of an individual constraint with respect to the rest of the constraints in the problem. The set of constraints C can now be described as a set of pairs, (c_i, w_i) , such that w_i is the weight of constraint c_i . An early technique that made use of this type of extension stemmed from the need in machine vision to perform an inexact match between the structural descriptions of two objects [147]. In this case, the weight of a constraint indicates the importance of a particular object part or a relationship amongst a subset of the parts of an object. A complete assignment, a , in such a system is given a rating based on the following:

Fig. 2.26. Solution to the Problem of Figure 2.23 using Additive Methods

$$\sum_{c_i \in V(a)} w_i$$

where $V(a)$ is the set of constraints violated by the proposed variable assignment. Hence, the cost of any necessary violations is calculated, indicating the quality of the current assignment. A threshold, ϵ , is used that the rating of a complete assignment must lie below. If, during Backtrack search, this threshold is exceeded by a partial assignment, this branch of the search tree is pruned.

Figure 2.26 describes how a simple additive solution procedure would solve the problem of Figure 2.23, if the priorities attached to the constraints are taken as weights to be summed upon violation. The detail of the diagram is restricted to the areas of the search tree surrounding the colour solutions, since any other solution cannot have a rating better (i.e. lower) than 1. This is because the weight 1 quaternary constraint will be violated by any such solution. Each node is annotated with its assignment, and the rating of the current partial assignment in parentheses. The dashed, directed line follows the course of the search.

The first ‘colour’ solution that is found is *pink*. This solution violates $\text{arc}(1, 2)$, $\text{arc}(1, 3)$ and $\text{arc}(1, 4)$, and hence has a rating of $0.3 + 0.2 + 0.3 = 0.8$. This rating is now used as a bound to prune the remainder of the search tree whenever the rating of a partial solution exceeds it. The solution *blue*, however, violates only $\text{arc}(2, 4)$, earning a rating of 0.4. Hence, $\{x_1 = B, x_2 = L, x_3 = U, x_4 = E\}$ is returned as the optimal solution.

Although suitable for the simple example flexible CSP, the problem with this relatively simple scheme is its coarseness: the computed rating for a complete assignment does not take into account the *level* of satisfaction of individual constraints, i.e. there is no way to express a preference amongst different assignment tuples. A constraint is regarded as either completely satisfied, in which case it will not contribute to the cost, or violated and its weight is added to the cost of this assignment. It is therefore possible for two

Fig. 2.27. The Failure of Simple Summation to Capture Satisfaction Levels

assignments which differ only in the extent to which they satisfy a particular constraint to be rated as the same quality.

An example is presented in Figure 2.27: nodes are labelled with their respective assignments, constraint arcs are labelled with their weights and the violated constraint is represented by a dashed line. The costs of the assignments in Figures 2.27a and 2.27b are the same since $c(x_1, x_3)$ is violated. 2.27a comes much closer to satisfying this constraint than does 2.27b.

An approach to this problem is to grade the level of satisfaction of an individual constraint. When integer domains are used, as per Figure 2.27, a simple metric is the distance that an assignment is from satisfying the constraint. Other domains may require more complex procedures, perhaps attaching levels of satisfaction to particular assignment tuples. These techniques are effectively adding *preference* information.

Satisfaction levels are commonly used in several constraint satisfaction systems. ISIS-II is a constraint-based reasoning system for the job-shop scheduling problem [52]. Here, variables represent start and end times and the allocation of resources to a set of operations required to produce a particular product. The constraints, which include organisational goals such as due dates and physical limitations such as the capabilities of individual machines, are assigned weights which denote their relative importance. In addition, each constraint generates an individual rating for a particular variable assignment which indicates how well the constraint is satisfied. These two sets of values are then combined when calculating the overall cost of the assignment.

One approach to combining weights and satisfaction levels to gain a more detailed idea of the quality of a variable assignment is as follows:

$$\sum_{c_i} w_i f_i(a)$$

where f_i is the rating function associated with constraint c_i . Given that this summation is counting the cost of constraint violations, the hypothetical rating function might work on the following basis:

- Complete satisfaction of c_i : $f_i(a) = 0$
- Complete violation of c_i : $f_i(a) = 1$

- Partial satisfaction of c_i : $0 < f_i(a) < 1$

If c_i is totally satisfied by an assignment, a , its weight is not added to the cost of this assignment. If c_i is totally violated (by some criteria), the full weight w_i will be added to the cost of a . For the example of Figure 2.27a, the rating of $c(x_1, x_3)$ might be 0.1 since the constraint is close to being satisfied, giving an overall rating of 0.05. In contrast, the rating of $c(x_1, x_3)$ might be 0.9, as the constraint is not very well satisfied, giving an overall rating of 0.45.

Dechter *et al* [33] present a general method of finding the optimal solution to a CSP, given a problem with multiple solutions. Rating functions are associated with variable subsets. Subsets are not restricted to those involved in individual constraints, as above, but may span more than one constraint. The rating function indicates how well an assignment satisfies the associated variable subset. Here, a higher rating indicates a more preferred variable assignment. The optimisation process consists of maximising the criterion function:

$$\sum_{i \in T} f_i(a^i)$$

where T is a set of indices of the subsets of the problem variables that have an associated function, f_i , and a^i is a complete variable assignment, a , restricted to this particular variable subset.

Partial Constraint Satisfaction [56, 58] is a general system which seeks to partially solve an over-constrained CSP by satisfying a maximal number of constraints. A metric is used to evaluate the utility of potential solutions. The metric may vary from being as simple as a count of the number of violated constraints to being as complex as necessary to capture subtle preferences between solutions. The most general view of partial constraint satisfaction considers a space of problems which are ordered based on the associated set of solutions. It is shown [58] how Partial CSPs can be solved via branch and bound methods. This basic approach is further extended to approximate Backjump and Backmark to reduce the effects of thrashing (see Section 2.2.1).

Consistency Counts

Arc consistency is enforced by filtering domain elements that cannot take part in any solution. This depends on the fact that constraints are hard: if an element necessarily violates a constraint it may be removed since all constraints must be satisfied. When some constraint violation is tolerated, elements may not be filtered so easily. If the necessary bound, α , on the quality of the solution is known beforehand then a value which necessarily causes that bound to be reached may be removed. An example may be seen in the MAX-CSP framework: any element which must violate at least α constraints can be removed.

When such prior knowledge is not available, the method of *arc consistency counts* has been proposed to aid in search [58]. For each $d_{i_a} \in D_i$, a record

is kept of the number of domains connected to D_i by an arc that contain no support for $d_{i\alpha}$. It is then possible to calculate the minimum cost incurred by making a particular assignment, enabling the calculation of a bound, β , on the quality of a complete assignment based on the current partial assignment. If this bound reaches α during search, this search branch may be pruned.

Consistency counts may also be used to create the Forward Checking hybrid for use with additive CSP. When arcs from future variables are revised the number of current assignments with which a particular domain element is inconsistent with is recorded as an *inconsistency count*. A domain element may then be filtered out when its inconsistency count reaches the bound, α . Freuder and Wallace develop increasingly sophisticated Forward Checking algorithms on this basis for the PCSP framework [58], and it is shown in [171] how these algorithms can be further improved. A similar approach, applied to the inexact structural match system can be found in [147].

2.9.4 Priority Maximisation Flexible CSP

Priority maximisation methods attempt to maximise the rating of a complete assignment via minimising the priority of the most important violated constraint. They work based on the aggregation (generally using *min/max* operations) of the satisfaction degrees of the constraints whose weights represent their individual priorities. An advantage of priority maximisation methods is the fact that the *max* operator is idempotent (i.e. for all possible arguments, x , it is the case that $x \otimes x = x$). As several researchers have shown [42, 141], this allows many techniques developed for classical CSP to be extended to this type of flexible CSP framework in a straightforward fashion. This is generally not the case for additive methods [58, 141].

An early example of this type of approach is the GARI system [37]. Here, the goal is to produce plans for machining parts given their drawings, composed of an ordered series of cuts to be made using a specified machine. The constraints on the generation of the plans are stated in terms of weighted pieces of *advice* which embody the domain knowledge of engineers. GARI starts with a loose plan and iteratively applies constraints drawn from its knowledge base as they are seen to be relevant in the current context. When a conflict arises, at least one piece of advice is rejected such that the highest weight of the violated pieces of advice is minimised. More recently, fuzzy constraint satisfaction (see below) has been applied to the domain of job-shop scheduling [41] and to the wine industry [137].

Necessity Based Constraints

The *Necessity*-based flexible CSP system [140] makes use of possibilistic logic [92] to model the gradual satisfaction of constraints. A possibility distribution, over the set of all possible complete variable assignments, A , is a function

π from A to $[0, 1]$. This distribution supports a gradual notion of the consistency of a given assignment, encapsulating preferences among the set of assignments.

A distribution, π , induces the measures of *possibility*, Π_π , and *necessity*, N_π , over the set of constraints. The possibility, $\Pi_\pi(c_i)$ represents the possibility of constraint c_i being satisfied, given π . The necessity measure, $N_\pi(c_i)$, represents the extent to which the satisfaction of constraint c_i is entailed given π . The necessity of a constraint tends to 1 as the possibility of it being unsatisfied tends to 0. The definition of both measures follows (adapted from [140]):

$$\begin{aligned}\Pi_\pi(c_i) &= \text{Sup}(\{\pi(a), a \models c_i\} \cup \{0\}) \\ N_\pi(c_i) &= \text{Inf}(\{1 - \pi(a), a \models \neg c_i\} \cup \{1\}) \\ &= 1 - \Pi_\pi(\neg c_i)\end{aligned}$$

where $a \in A$ is a complete variable assignment; $\neg c_i$ is the complement of c_i : the subset of the Cartesian product of the domains of the constrained variables that does *not* satisfy c_i ; and $a \models c_i$ signifies that assignment a satisfies c_i .

Possibilistic CSP replaces hard constraints with necessity-valued constraints. The necessity expresses the relative preference of satisfaction of each constraint, as per Figure 2.23. The set of constraints, C , is represented by the set of pairs, (c_i, α_i) , with $\alpha_i \leq N_\pi(c_i) \leq 1$. Hard constraints are represented via $(c_i, 1)$. Since necessity depends on the possibility distribution π , a distribution must be chosen which induces the required necessity on each constraint, denoted: $\pi \models (c_i, \alpha_i)$, for $N_\pi(c_i) \geq \alpha_i$. As Schiex proves, it is possible to construct a maximal possibility distribution for a given set of necessity-valued constraints as follows:

$$\forall a \in A, \pi^*(a) = \text{Inf}_{(c_i, \alpha_i) \in C} (\{1 - \alpha_i \mid a \models \neg c_i\} \cup \{1\})$$

Possibilistic CSPs are consistent to a *degree* rather than consistent or inconsistent as per classical CSPs. The degree of consistency of a possibilistic CSP is the maximum value assigned by π^* to an assignment $a \in A$, where A is the set of all possible complete variable assignments. The set of ‘best’ assignments A^* for a particular problem contains those that receive this highest value assignment.

A branch and bound algorithm is used to find such a best assignment. The upper bound on the rating of a complete assignment, β , may be computed as shown below, by taking the lowest value assigned by π given the constraints applicable to the current partial assignment. β may only remain the same or decrease in the light of future assignments. V_i are the variables constrained by c_i , P is the set of variables currently instantiated, and a is an assignment to the variables in P .

$$\beta = \text{Inf}_{(c_i, \alpha_i) \in C} (\{1 - \alpha_i \mid V_i \subset P, a \models \neg c_i\} \cup \{1\})$$

Fig. 2.28. Solution to the Problem of Figure 2.23 using Necessity-based Constraint Satisfaction

This bound may be computed incrementally by taking the minimum of the current value of β and the new value found by considering those constraints involving the current assignment to X_i :

$$\beta' = \text{Inf}(\{\beta\} \cup \{1 - \alpha_i \mid (c_i, \alpha_i) \in C, V_i \subset P \cup \{x_i\}, V_i \not\subset P, a \models \neg c_i\})$$

In summary, this system offers a possibilistic framework to support relative constraint priorities, but does not immediately support constraint assignment preferences. The basic branch and bound solution technique has been extended to use a form of the hybrid *forward checking* algorithm [77] to increase efficiency, and a method for achieving arc consistency is also defined [140].

Figure 2.28 presents the search tree generated by a necessity-based technique in solving the problem of Figure 2.23. The difference from the additive methods is clear, instead of accumulating a cost through constraint violation, this method attempts to minimise the priority of the most important violated constraint. Hence, the algorithm attempts to maximise the rating of the overall solution. Again, *pink* is the first colour solution found, with a rating of 0.7. This algorithm regards *pink* as a better solution than *blue* (rating 0.6) since the constraints it violates have a priority of 0.3 or less.

A Generic Fuzzy Constraint Satisfaction Framework

Fuzzy Constraint Satisfaction [42, 119] supports both prioritised and preference constraints, again using possibility theory [44]. Both types of constraint are modelled by a fuzzy relation, R , which is in turn defined by μ_R , a *membership function* which associates a level of satisfaction of an assignment tuple in $D_1 \times \dots \times D_n$ with a value in a totally ordered *satisfaction scale*, $L = [0, 1]$. These flexible constraints are used to order partial assignments in order to find an overall best complete assignment.

A preference constraint, c_i , amongst a set, A , of potential assignments to its constrained variables can be modelled as follows, where a , $a \in A$, is a particular partial assignment. The membership function μ_{R_i} of a satisfying constraint c_i is defined on L :

$$\begin{aligned}\mu_{R_i}(a) &= 1 \text{ if } a \text{ totally satisfies } c_i \\ \mu_{R_i}(a) &= 0 \text{ if } a \text{ totally violates } c_i \\ 0 < \mu_{R_i}(a) < 1 &\text{ if } a \text{ partially satisfies } c_i\end{aligned}$$

Prioritised constraints are supported as follows. The scale V (effectively L in reverse) is used, representing *possibility of violation* or equivalently the priority of a constraint. A priority degree, $Pr(c_j) \in V$, is associated with each prioritised constraint, c_j , indicating the necessity of its satisfaction. A constraint with priority 1 is imperative (i.e. with no possibility of violation) and a constraint with priority 0 is totally irrelevant. The constraints of the example of Figure 2.23 may be represented as prioritised constraints in this framework.

The bijection b is used to map from V to L such that $L = b(V)$. The quantity $b(Pr(c_j)) \in L$ indicates the satisfaction level of the complement (i.e. those tuple disallowed) of c_j . A prioritised constraint c_j with priority $Pr(c_j)$ is modelled as follows: it is fully satisfied by an assignment if c_j is satisfied and to a degree of $b(Pr(c_j))$ otherwise.

$$\begin{aligned}\mu_{R_j}(a) &= 1 \text{ if } a \text{ satisfies } c_j \\ \mu_{R_j}(a) &= b(Pr(c_j)) \text{ if } a \text{ violates } c_j\end{aligned}$$

It is also possible to specify a constraint which is prioritised *and* expresses a preference amongst assignments. A prioritised-preference constraint is represented by a fuzzy relation R_k and is defined as shown below, with μ_{R_i} describing the preference component. Hence a constraint c_k , represented by R_k will be satisfied to at least the degree $b(Pr(c_k))$ since the priority degree defines a bound on the damage to the solution that is incurred by the violation of this constraint. The constraint may be satisfied above this level by satisfying the preference component to a higher degree.

$$\mu_{R_k}(a) = \max(b(Pr(c_k)), \mu_{R_i}(a))$$

The consistency level of a partial assignment is calculated from the aggregated membership values of the constraints involving the assigned variables as follows, where a is an assignment to $x_1, \dots, x_k$; $\otimes$ represents the conjunctive combination of fuzzy relations³; and $Vars(R_i)$ is the set of variables constrained by the constraint represented by R_i :

$$Cons(a) = \mu_{[\otimes\{R_i \mid Vars(R_i) \subseteq \{x_1, \dots, x_k\}\}]}(a)$$

³ The conjunctive combination of two fuzzy relations, $R_i \otimes R_j$ is usually interpreted as the minimum membership value assigned by either relation.

Fig. 2.29. A Fuzzy CSP and Associated Search Path

This quantity is the upper bound on the consistency of a complete assignment. It is computed incrementally during search by considering only constraints that involve the current variable.

Since the constraints in the example of Figure 2.23 are all prioritised, FCSP performs exactly as per the necessity-based method in this case (see Figure 2.28). To illustrate the other constraint types mentioned, Figure 2.29 presents a new example. A hard constraint (i.e. priority 1) constrains x_1 and x_2 to equality, a unary preference constraint prefers x_2 to be as close to the value 3 as possible and a prioritised preference constraint (with priority 0.8) prefers the assignment of x_2 to be at least 2 more than that of x_3 . The search tree for this problem is given in Figure 2.29b. The (intuitive) best solution found is $x_1 = 2, x_2 = 2, x_3 = 1$.

A fuzzy CSP involves a set of flexible constraints of the types described above over a set of n variables. The consistency level of a partial assignment is calculated from the aggregated membership values of the constraints involving the subset of the variables which currently have values assigned. This is used as the basis of a branch and bound search to find the best solution to the problem. The standard consistency degree calculation provides quite a coarse definition of the best solution to the problem, since only the most violated constraint is used. Approaches exist which discriminate amongst solutions with the same basic consistency degree at a greater computational cost [43, 105].

2.10 Dynamic Constraint Satisfaction

A combination of classical and flexible constraints allows many problems to be accurately modelled. However, one important aspect of many real problems has yet to be addressed. So far a *static* problem has been assumed, precluding any changes to the problem structure after its initial specification. To a cer-

Fig. 2.30. An Evolving Problem Modelled via *rrDCSP*

tain extent, flexible CSP can be seen as adding dynamicity to the problem: the problem is effectively changed when constraints are softened/removed to achieve a compromise solution (see, for example, [75]). Many problems, however, undergo changes, sometimes continually, which may occur while the solution to the original problem is being executed or, perhaps more critically, while a solution to the original problem is still being sought.

The ability to reason about a dynamic environment is crucial in many areas of Artificial Intelligence. Unfortunately, the majority of CSP research has focused on static problems, i.e. the set of constraints and problem variables are known beforehand and fixed. Nevertheless, techniques for solving *Dynamic Constraint Satisfaction Problems* (DCSPs) do exist, with three alternative formulations that make slightly different assumptions about the nature of the dynamic problems involved.

2.10.1 Restriction/Relaxation-based Dynamic Constraint Satisfaction Problems

The first formulation [32] views a dynamic environment as a sequence of CSPs linked by *restrictions* and *relaxations*, where constraints are added to and removed from the problem respectively (Figure 2.30). This type of CSP is hereafter referred to as *rrDCSP*. Note that the addition or removal of variable domain elements is effectively the addition or removal of unary constraints to the problem. Naively, each individual problem in the sequence may be solved from scratch using static CSP techniques, but this method discards all the work done in solving the previous (probably similar) problem. This wasted effort is usually unacceptable in real-time applications that can experience rapid changes in the problem. The key to efficiently solving *rrDCSPs* is to re-use as much as possible of the effort required to solve previous problems in the sequence in solving the current problem.

The existing approaches to dynamic constraint satisfaction may be broadly classed into three categories. The *oracles* approach searches through previously solved instances for a less constrained version of the current problem.

If one is found, the part of the search space before the associated solution is not explored, since no solution exists in it for the more constrained current problem. A simplified version of this method is exploited in the combination of dynamic and flexible methods in the following chapter.

A second method, known as *local repair* [114, 167], maintains all assignments from the solution to the previous problem in the sequence to use as a starting point. This initial variable assignment is then progressively modified by a sequence of modifications to individual variable assignments until an acceptable solution to the current problem is obtained. Modifications take the form of element reassignment in order to resolve as many conflicts (i.e. constraint violations) involving this variable as possible. A particular local repair algorithm is the *heuristic repair* algorithm [114] which searches through the space of possible repairs using the heuristic that there should be as few constraint violations as possible at each stage to guide the search.

The *Local Changes* (LC) local repair algorithm [167] (also utilised as part of a dynamic-flexible combination in Chapter 3) maintains a list of currently unassigned variables. When the problem changes in an *rrDCSP* this would consist of any new variables added to the problem or variables whose current assignment has been lost. This repair algorithm repeatedly selects a variable from the set that are unassigned and assigns a value to it. If the assignment causes some constraint violations, LC attempts to repair the current set of assignments to resolve the inconsistencies. Repairs consist of unassigning the subset of the assigned variables that is perceived to have caused the inconsistency, and attempting to reassign values to them such that all constraints are satisfied. This algorithm obtains an improvement over heuristic repair by avoiding the repair of variables which have no bearing on the existing inconsistencies, and is therefore related to static CSP algorithms such as Backjump.

Finally, constraint recording methods [143, 166] infer new constraints from the existing problem definition which disallow inconsistent assignment combinations not directly disallowed by the original constraints. The justifications of inferred constraints are recorded in order that they can be used in future problems where the same justifications hold to converge on a solution more quickly. In fact, the Local Changes algorithm can be improved through the use of such a scheme [167]. Again the same caveat applies concerning the extra work incurred by checking many extra constraints.

An *rrDCSP* Example

Returning to the original CSP given in Figure 2.1, consider the effect of changing this problem as presented in Figure 2.31. Constraints $c(x_2, x_3)$, $c(x_3, x_4)$ have been removed and the variables x_4, x_5 have been added with constraints connecting them to each other and the rest of the problem. Lastly, the element E has been added to D_3 . Clearly, additional work is required to solve this new problem. The naive reaction is to simply apply the original static

Fig. 2.31. An *rr*DCSP Example, Updated from Figure 2.1

CSP solution procedure (e.g. Backtrack) to the whole updated problem. This is wasteful of the work done on the original (similar) problem.

As an example of the savings that can be made using an *rr*DCSP technique, it is useful to examine how the Local Changes algorithm would go about solving this new problem on the basis of the original solution: $\{x_1 = B, x_2 = I, x_3 = V, x_4 = K\}$. Figure 2.32 describes the solution procedure, where V_1 is the set of variables whose assignments are fixed. In addition V_2 is the set of variables which have assignments but which are not fixed, and V_3 is the set of unassigned variables. The algorithm terminates with the solution: $\{x_1 = B, x_2 = I, x_3 = E, x_4 = K, x_5 = E, x_6 = J\}$, having done just a fraction of the work that would be required to solve the problem from scratch.

Through making use of the previous solution in the sequence, there is a significant efficiency saving made by avoiding re-solving a large part of the problem. This simple example serves as an indication of the saving that can be made generally by maintaining as much of the effort expended on the previous solution as possible. In addition to efficiency, this type of technique offers the benefits of *stability* to more complex problems. That is, by maintaining as much of the previous solution as possible, there is as little disruption from one solution to the next as possible. An *rr*DCSP technique is much more likely to achieve a reasonable level of stability than a static CSP technique applied to the new problem since the latter essentially throws away all the work done on the previous solution.

2.10.2 Recurrent Dynamic Constraint Satisfaction Problems

A second DCSP formulation was developed more recently, and is known as *recurrent* DCSP [172], referred to herein as *rc*DCSP. This technique is founded on the fact that many alterations to a problem may be temporary

Initial: $x_1 = B, x_2 = I, x_3 = V, x_4 = K$ (consistent)
 $V_1 = \{\}, V_2 = \{x_1, x_2, x_3, x_4\}, V_3 = \{x_5, x_6\}$
Select $x_5 \in V_3$. Consider $c(x_3, x_5)$
 $x_5 \leftarrow X$. Violates $c(x_3, x_5)$: Choose x_3
 $V_1 = \{x_5\}, V_2 = \{x_1, x_2, x_4\}, V_3 = \{x_3\}$

Select $x_3 \in V_3$. Consider $c(x_1, x_3), c(x_3, x_5)$
All assignments violate $c(x_3, x_5)$. x_5 is fixed. Fail.

 $x_5 \leftarrow N$. Violates $c(x_3, x_5)$: Choose x_3
 $V_1 = \{x_5\}, V_2 = \{x_1, x_2, x_4\}, V_3 = \{x_3\}$

Select $x_3 \in V_3$. Consider $c(x_1, x_3), c(x_3, x_5)$
 $x_3 \leftarrow V, x_3 \leftarrow U$ violate $c(x_3, x_5)$. x_5 is fixed.
 $x_3 \leftarrow N$ succeeds.
 $V_1 = \{\}, V_2 = \{x_1, x_2, x_3, x_4, x_5\}, V_3 = \{x_6\}$

Select $x_6 \in V_3$. Consider $c(x_5, x_6), c(x_4, x_6)$
 $x_6 \leftarrow M$. Violates $c(x_5, x_6), c(x_4, x_6)$: Choose x_4, x_5
 $V_1 = \{x_6\}, V_2 = \{x_1, x_2, x_3\}, V_3 = \{x_4, x_5\}$

Select $x_4 \in V_3$. Consider $c(x_1, x_4), c(x_2, x_4), c(x_4, x_6)$
All assignments violate $c(x_4, x_6)$. x_6 is fixed. Fail.

 $x_6 \leftarrow J$. Violates $c(x_5, x_6)$. Choose x_5
 $V_1 = \{x_6\}, V_2 = \{x_1, x_2, x_3, x_4\}, V_3 = \{x_5\}$

Select $x_5 \in V_3$. Consider $c(x_3, x_5), c(x_5, x_6)$
 $x_5 \leftarrow X, x_5 \leftarrow N$ violate $c(x_5, x_6)$. x_6 is fixed
 $x_5 \leftarrow E$ violates $c(x_3, x_5)$. Choose x_3
 $V_1 = \{x_5, x_6\}, V_2 = \{x_1, x_2, x_4\}, V_3 = \{x_3\}$

Select $x_3 \in V_3$. Consider $c(x_1, x_3), c(x_3, x_5)$
 $x_3 \leftarrow V, x_3 \leftarrow U, x_3 \leftarrow N$ violate $c(x_3, x_5)$. x_5 is fixed
 $x_3 \leftarrow E$ succeeds.
Solution: $x_1 = B, x_2 = I, x_3 = E, x_4 = K, x_5 = E, x_6 = J$

Fig. 2.32. Solution to the Problem Presented in Figure 2.31 via Local Changes

and that these changes may be subsequently reversed. *rcDCSP* concentrates on changes to the set of constraints that could potentially render the current solution invalid. These changes include the addition of unary constraints (domain element removal), the addition of k -ary constraints, and the loss of some acceptable tuples in existing constraints. The aim of *rcDCSP* is to find solutions that are not ‘broken’ by such changes. A good solution to an *rcDCSP* is therefore stable in a slightly different sense: it remains acceptable in light of alterations to the problem.

An important point to note is that all the changes supported by *rcDCSP* result in the temporary loss of one or more domain elements as potential members of acceptable solutions. In light of this, a probability of loss may be associated with each domain element. A stable solution may then make use of a set of assignments with as low a probability of loss as possible. Since it is unlikely in general that individual loss probabilities are known beforehand, this information is extracted dynamically by recording the frequency of loss of each value as the problem evolves.

Solution techniques for *rcDCSP* have been developed based on a local repair-type technique which uses hill-climbing, taking into account the loss probabilities to maximise stability. Although hill-climbing does not guarantee that an optimal solution will be found, this is a moot point since the loss probabilities are only estimates. The principle argument in favour of hill-climbing is that there is no restriction on the timing of changes to the problem. Hence, if the problem changes while a complete method is in the process of solving it, one of the assigned domain elements may be lost, forcing the retraction of any assignments after this point. Further, stability calculations may be rendered invalid, removing the guarantee of finding an optimally stable solution. Hill-climbing methods do not suffer from these problems. In the face of change, hill-climbing can proceed as normal, pausing only to reassign variables whose assigned elements have been removed.

This solution technique has been tested on graph-colouring and random CSPs [172]. It was found that repairing a partial solution not only increases efficiency in finding a solution to the current problem, but is also effective in finding stable solutions (i.e. those likely to remain valid in the face of change). The technique of dynamically acquiring loss probabilities for individual domain elements was particularly successful in leading to stable solutions, particularly when there were very few stable solutions to be found. A related approach is in the use of *supermodels* [71]. Here, a ‘robust’ solution is sought such that if the assignments to a subset (size a) of the problem variables are modified, another solution can be found by repairing a disjoint subset of variables (of size at most b). Hence, the amount of repair required for solutions found by this technique is bounded.

2.10.3 Activity-based Dynamic Constraint Satisfaction Problems

Activity-based DCSP [115], referred to henceforth as *aDCSP*, takes a quite different view of dynamicity to those already presented. An *aDCSP* consists of a classical CSP together with constraints describing how variables are added (or removed) from a problem based on other particular variable assignments. Hence, not all variables in an *aDCSP* must be assigned a value to find a solution, since some assignments render other variables irrelevant.

An *aDCSP* specifies a set of initial variables, $X_{initial} \in X$, that must appear in all solutions. As per a classical CSP, each $x_i \in X$ has an associated domain, D_i , describing its potential values. The set of constraints over these variables, however, is subdivided into two types. *Compatibility* constraints, $C_c \subset C$, correspond exactly to the constraints found in classical CSPs, specifying allowed combinations of variable assignments. *Activity* constraints, $C_a \subset C$ specify the conditions under which a variable and its associated domain become relevant in constructing a solution. An activity constraint may be one of two types, requiring the presence or the removal of a variable respectively. Basic activity constraints take the following general forms:

Variable	Domain
$x_{Package} \in X_i$	{luxury, deluxe, standard}
$x_{Frame} \in X_i$	{convertible, sedan, hatchback}
$x_{Engine} \in X_i$	{small, medium, large}
$x_{Battery}$	{small, medium, large}
$x_{Sunroof}$	{sunroof1, sunroof2}
$x_{AirConditioner}$	{airconditioner1, airconditioner2}
x_{Glass}	{tinted, not-tinted}
x_{Opener}	{opener, manual}

Table 2.3. Variables and Domains for the Car Configuration Example

$$c_1 \in C_a : x_i = d_{i_a}, x_j = d_{j_b}, \dots \rightarrow active(x_k)$$

$$c_2 \in C_a : x_i = d_{i_a}, x_j = d_{j_b}, \dots \rightarrow inactive(x_k)$$

Special cases are also defined to allow the easy specification of the case where the presence of one or a set of variables (irrespective of current assignments) always requires the presence/removal of another variable.

$$c_3 \in C_a : active(x_i), active(x_j), \dots \rightarrow active(x_k)$$

$$c_4 \in C_a : active(x_i), active(x_j), \dots \rightarrow inactive(x_k)$$

Consider an example concerning the configuration of a car (adapted from [115]). The variables and their associated domains are presented in Table 2.3. Compatibility constraints are given in a dual-graph format (see Section 2.2) in Figure 2.33. In addition, the activity constraints for this problem are described in Table 2.4.

Since $x_{Package}$, x_{Frame} , and x_{Engine} are specified as initial variables, all solutions to this problem must contain assignments to each of these three variables. Indeed, several solutions to this problem exist which encompass

Antecedent	Activity Constraint
$x_{Package} = \text{luxury}$	$active(x_{Sunroof})$
$x_{Package} = \text{luxury}$	$active(x_{AirConditioner})$
$x_{Package} = \text{deluxe}$	$active(x_{Sunroof})$
$x_{Sunroof} = \text{sunroof1}$	$active(x_{AirConditioner})$
$x_{Sunroof} = \text{sunroof1}$	$inactive(x_{Opener})$
$x_{Sunroof} = \text{sunroof2}$	$active(x_{Opener})$
$active(x_{Sunroof})$	$active(x_{Glass})$
$active(x_{Engine})$	$active(x_{Battery})$
$active(x_{Opener})$	$active(x_{Sunroof})$
$active(x_{Glass})$	$active(x_{Sunroof})$
$x_{Frame} = \text{convertible}$	$inactive(x_{Sunroof})$
$x_{Battery} = \text{small}, x_{Engine} = \text{small}$	$inactive(x_{AirConditioner})$

Table 2.4. Activity Constraints for the Car Configuration Example

Fig. 2.33. Compatibility Constraints for the Car Configuration Example

only the initial variables. For example: $\{x_{Package} = \text{standard}, x_{Frame} = \text{convertible}, x_{Engine} = \text{small}\}$. Other assignments to the initial variables require the consideration of other members of X , as specified by the activity constraints. For example, choosing the luxury package entails further choices as to the type of sunroof and air-conditioning required. All assignments to the currently active variables are subject to the compatibility constraints as per classical CSP.

Mittal and Falkenhainer present an ATMS-based implementation of an α DCSP solution technique [115] which is shown to be successful on both the configuration type of problem discussed here and also on the domain of compositional modelling [46, 85]. Kambhampati [82] describes the application of these techniques to the domain of AI planning in the Graphplan [14] framework. This work inspired the use of other dynamic CSP techniques for solving AI planning problems, a detailed discussion of which begins in Chapter 5.

2.11 Summary

This chapter has initially examined the *constraint satisfaction problem* (CSP) and the fundamental techniques employed to solve this type of problem. The classical (imperative, Boolean and static constraints) formulation was discussed and also dynamic and flexible extensions to the original framework.

Different graphical means of representing a CSP were investigated, which can then be used to guide a solution method. Tree search solution techniques were discussed, beginning with *Backtrack*. The *thrashing* problem was analysed, with attempts to avoid it: *Backjump* and *Conflict-Directed Backjump* reduce the number of nodes in the search tree. Conversely, *Backmark* reduces the number of constraint checks. Tree search hybrids exploit savings made by both approaches. *Dynamic Backtracking* dynamically re-arranges the search order to avoid erasing potentially useful work during a backward jump. No

single tree search algorithm uniformly maintains the best performance, however - CSP structure is crucial. A CSP of arbitrary size may be defined upon which Backtrack outperforms more sophisticated algorithms by making the 'right' choices without the overhead incurred by more informed algorithms.

Pre-processing a CSP reduces subsequent tree search by filtering domain elements and/or modifying constraints. Enforcing *arc consistency* is a popular technique due to its relatively low cost. Enforcing *path consistency* modifies constraints to rule out inconsistent *pairs* of values. *k-consistency* was discussed in the context of achieving a *backtrack-free* search. Finally, a fixed instantiation order may be exploited to avoid much consistency processing. Although easier problems are produced, the overhead of the more powerful pre-processing techniques combined with subsequent tree search may be more than the effort required by a single tree search algorithm applied to the initial CSP. Difficult CSPs, typically those found at the phase transition, benefit most.

Tree search and consistency enforcing techniques may be integrated to create *hybrids* which enforce a measure of consistency at each search node. Due to the computational overhead involved in enforcing consistency at each search node, care must be taken when combining two such approaches to ensure that the overheads do not outweigh the benefits. *Partial* arc consistency procedures were seen to successfully provide a net gain when combined with tree search. However, more sophisticated consistency enforcing techniques can result in significant increased efficiency as part of a hybrid on more difficult problems. Furthermore, *Conflict Recording* extracts information from a dead end in the search which is stored and used to prune the search tree subsequently.

The performance of the basic CSP solution techniques may be improved in various ways. For example, via *heuristics* which primarily establish an instantiation order, either statically or dynamically, which most benefits the search process. Heuristics which influence the order in which values are assigned to each variable or in which constraint checks are made were also presented. It was seen that improving either can aid a solution technique.

Graph-based algorithms were examined as an alternative or an aid to conventional solution procedures. The *Invasion Procedure* is an efficient method of finding all solutions to a CSP if a good invasion can be found. The fact that a tree-structured CSP can be solved in linear time was seen to be fundamental to the approaches of *cycle-cutset*, *non-separable components*, and *tree-clustering*. Each attempts to find or create a tree-structured graph which can then be solved efficiently.

Classical CSP was seen to be insufficient to solve many real problems. The first major deficiency is in its rigidity of structure: it is common in reality for constraints to contain some degree of flexibility. *Flexible* CSP was presented as an extension of the classical framework which supports several types of 'soft' constraint. Flexibility encompasses constraints which possess a relative

priority, or express a preference over potential solutions, as well as *ill-defined* CSPs where the set of constraints which define the problem are unknown beforehand. Generally, a flexible CSP is implemented via an operator which aggregates ratings of local sections of the whole problem in order to compute a globally optimal complete variable assignment. *Branch and bound* search was identified as the most common method of solving flexible CSPs.

Flexible techniques are becoming prevalent in attempting to deal with the flexibility inherent in real applications. One line of research is in applying the more sophisticated classical CSP techniques (e.g. Backjump, consistency enforcing) to flexible CSP. Schiex *et al* [141] compared the properties of various types of flexible CSP and found that the aggregation operator used is important in extending classical CSP techniques to a flexible framework. The use of idempotent⁴ operators (as per [42]) is seen to offer the greatest ease of extension of such classical methods, while it is shown that greater difficulty can be expected in extending the same approaches to flexible CSP frameworks with non-idempotent operators such as additive methods.

The second deficiency of classical CSP is in its assumption of a static problem. As a means of supporting a problem which changes over time by adding or removing constraints, *restriction/relaxation-based dynamic CSP* was presented. *Recurrent* dynamic CSP is also based on restriction and relaxation, but addresses problems that exhibit primarily temporary changes that may subsequently be reversed. By identifying assignments with a high probability of remaining acceptable as the problem changes, robust solutions can be found. *Activity*-based dynamic CSP takes a different view of dynamics, considering problems whose internal structure changes based on the current assignments to subsets of the problem variables.

The review contained in this chapter summarises current constraint satisfaction techniques. The next part of the book explores the further development of CSP algorithms to cope with increasingly complex problems. Specifically, those problems that are subject to change *and* exhibit flexibility in their definition.

⁴ An operator, $\otimes$, is idempotent if for all arguments, a , it is the case that $a \otimes a = a$.

3 Dynamic Flexible Constraint Satisfaction

This chapter describes the further evolution of constraint satisfaction techniques to support both flexible constraints and dynamic changes to the problem structure: dynamic flexible constraint satisfaction problems (DFCSPs). Through the combination of the different dynamic and flexible extensions to classical CSP detailed in Chapter 2, a matrix of possible DFCSP instances is defined. A third dimension to this matrix consists of the solution techniques appropriate to each DFCSP instance.

A specific DFCSP instance is considered in detail, composed from restriction/relaxation dynamic CSP and fuzzy constraint satisfaction as two powerful representative examples of dynamic and flexible CSP respectively. In addition, two basic solution methods for this type of DFCSP are developed. The first is a heuristic enhancement of a branch and bound solution technique usually applied to static fuzzy CSP, and the second is a new algorithm, Flexible Local Changes, based upon an existing *rrDCSP* approach.

Complexity issues are considered for the new algorithm, but for clarity of exposition, a formal proof of soundness and completeness is presented separately in Appendix B. Hybrid consistency-enforcing variants of both the branch and bound and FLC algorithms are also discussed. The operation of both algorithms is demonstrated via a small worked example prior to detailed experimental results in the next chapter.

3.1 Towards Dynamic Flexible Constraint Satisfaction

The previous chapter discussed at length two extensions to the classical CSP framework: flexible CSP, and dynamic CSP respectively. However, the flexible methods dealt only with a static set of constraints, whilst the dynamic CSP techniques were seen to be based exclusively on classical Boolean constraints. Little work has been done in the area of combining the two disparate extensions to classical CSP techniques to produce a framework able to model the changes in a dynamic environment, while retaining the greater expressive power afforded by flexible CSP (see Figure 3.1). Such a framework would be capable of dealing with a greater number of real problems than is currently possible. In a dynamic environment where time may be limited, the ability

Fig. 3.1. Integration of Disparate Extensions of Classical CSP

of flexible CSP to produce the best current solution *anytime* and the use of a sufficiency bound to limit search will prove even more valuable.

3.1.1 Concepts of DFCSP

Given the multiple existing methods of supporting both flexible constraints and dynamic problems (see Chapter 2), DFCSP becomes an umbrella term for a range of possible frameworks that capture different concepts of dynamicity and flexibility as presented in Figure 3.2. The x dimension denotes the different instances of dynamic CSP, and the y dimension denotes the instances of flexible CSP as described in Chapter 2. A third dimension could be added to this matrix, comprising the solution techniques for each associated DFCSP instance. This book begins to investigate the matrix of DFCSP instances and starts to populate the solution technique dimension, developing two algorithms to solve a particular representative DFCSP instance, but which may easily be modified to support other instances.

Although relatively sparsely populated currently, the x dimension could be extended by combining a DCSP and either rr DCSP or rc DCSP (see Section 2.10). This would allow problems to be modelled which both change over time *and* exhibit changes in structure based on certain assignments. Suggested names for these new DCSP techniques are a/rr DCSP and a/rc DCSP as per the figure. Of course, these can be further combined with flexible CSP techniques to create even more powerful instances of DFCSP.

The potential applications of DFCSP are wide, encompassing many dynamic problems that require more flexibility than classical CSP can provide. Some examples include planning (see Chapter 5), scheduling, machine vision, model-based reasoning (see Chapter 9), and game-playing.

		Dynamic CSP Methods					
		<i>rr</i> DCSP	<i>rc</i> DCSP	<i>a</i> DCSP	<i>rr/a</i> DCSP	<i>rc/a</i> DCSP	...
Flexible CSP Methods	Leximin Fuzzy	☐	☐	☐	☐	☐	
	Discrimin Fuzzy	☐	☐	☐	☐	☐	
	Fuzzy	☐	☐	☐	☐	☐	
	Necessity-Based	☐	☐	☐	☐	☐	
	Subset Rated +ve	☐	☐	☐	☐	☐	
	Weighted/Pref +ve	☐	☐	☐	☐	☐	
	Weighted +ve	☐	☐	☐	☐	☐	
	MAX	☐	☐	☐	☐	☐	
		<div style="display: flex; align-items: center;"> <div style="width: 10px; height: 10px; border: 1px solid black; margin-right: 5px;"></div> <div>DFCSP Instance</div> </div>					

Fig. 3.2. The Matrix of Possible Instances of DFCSP

3.2 Examples from the Dynamic Perspective

The following sub-sections consider possible instances of DFCSP based on one of the three types of dynamic CSP covered in Section 2.10.

3.2.1 Restriction/Relaxation-based DFCSP

The first instances considered are based on restriction/relaxation based DCSP, and will therefore be referred to as *rrDCSP*. Consider the problem presented in Figure 3.3, based on that given in Figure 2.23. The parts of this problem that may be subject to change are highlighted. The first three correspond to the parts of the problem that may change in the original formulation of *rrDCSP*. The flexible information available, and hence that subject to change, will differ according to the flexible method used. For instance, a problem with weighted constraints only, such as the one given in Figure 2.23 may exhibit changes to these weights over time.

A more sophisticated system for supporting flexibility, however, such as fuzzy constraint satisfaction [42] or an additive system which associates rating functions with variable subsets [33] (see Section 2.9.3) can change in more subtle ways. For example, the balance of preferences of the individual tuples of a fuzzy preference constraint (see Section 2.9.4) may change or the rating function associated with a variable subset may be updated.

More fundamentally, when multiple constraint types are supported, a change may occur transforming one constraint into another of a different

Fig. 3.3. Potential Changes to a Flexible Problem Based on Restriction/Relaxation

type. A simple example is the addition of a priority to a preference constraint to create a prioritised-preference constraint. Equally, a prioritised constraint may become a preference constraint (newly associating a satisfaction degree with each allowed tuple).

An *rrDFCSP* may be thought of as a sequence of static flexible problems similarly to the original hard DCSP definition (see Figure 3.4). All possible changes may still be realised through restriction and relaxation. However, since flexible CSP allows more powerful constraint types, these two operations can accordingly effect the more subtle changes to the as problem described above.

3.2.2 Recurrent DFCSP

Since *rcDFCSP* is also based on restriction/relaxation, Figure 3.4 applies in this case too. As per the *rcDCSP* case, changes to the problem are regarded as temporary and likely to be reversed some time in the future. A good solution to an *rcDCSP* is able to remain acceptable despite changes to the problem.

Fig. 3.4. Restriction/Relaxation-based Dynamic Flexible CSP

Fig. 3.5. Flexible Assignment Choices for Recurrent DFCSP

The use of flexible constraints enables a more subtle view of this process. A good solution to an *rcDFCSP* will retain as high a satisfaction degree as possible in light of alterations to the problem.

As described in Section 2.10.2, *rcDCSP* associates probabilities of loss with each domain element according to the perceived likelihood of this element not being part of a solution to a future problem. This scheme can be extended to the flexible case by estimating loss probabilities for individual satisfaction levels. For example, a domain element might have a high probability of being lost from a top quality solution, but a very low probability of being lost from a relatively low quality solution.

This information can be used during the solution process. A solution should naturally be sought with an optimally high satisfaction degree for the current problem, but it should also consist of assignments with the lowest probability of this satisfaction degree being damaged in the light of changes to the problem.

Consider an example based on resource allocation, as presented in Figure 3.5. Machine A is capable of performing two tasks, task 1 and task 2. It performs task 1 very efficiently, but task 2 relatively inefficiently. Hence, it is quite naturally preferred that task 1 is assigned to machine A. However, experience has shown that machine A is fairly unreliable, part of it breaking down regularly. Although this precludes it from doing task 1, it can still do task 2. The machine breaks down completely (such that it can do neither task) much more rarely. Therefore, the probability of the assignment of task 1 to machine A ‘breaking’ in the future with respect to a high quality solution is high, but the assignment of task 2 is unlikely to break an already lower quality solution. This type of information can be weighed up by the problem solver when solving each problem in a dynamic sequence.

$c(x_{Opener}, x_{AirConditioner}, x_{Battery})$
auto, airconditioner1, small: 0.25
auto, airconditioner1, medium: 1.0
auto, airconditioner1, large: 0.75
auto, airconditioner2, medium : 0.5
auto, airconditioner2, large : 1.0

Fig. 3.6. A Flexible Compatibility Constraint for Car Configuration

3.2.3 Activity-based DFCSP

Given the characteristics of *aDCSP*, namely that portions of the problem are activated and de-activated based on particular variable assignments, *aDFCSP* presents some different possibilities when considering the addition of flexible constraints. Firstly, and as per both examples considered above, compatibility constraints (i.e. those which constrain compatible assignments between variables only) may become flexible to reflect priorities and preferences.

Consider, for example, Figure 3.6 which presents one of the constraints from the car configuration problem described in Section 2.10.3 (see Figure 2.33). The constraint tuples are now annotated with preferences. These indicate, for instance, that a type-one air conditioner will work perfectly well with a large battery, but that a medium-sized battery is all that is really required. Furthermore, a small battery will struggle to support the air conditioner.

Secondly, flexibility can be incorporated into the activity constraints that dictate how the structure of the problem changes. In this case, a different satisfaction degree might be attached to alternative extra portions of the problem. Hence, it might be preferred to consider one set of extra variables given an assignment, but failing this it might be possible to solve a less preferred set.

Consider again the car configuration example, this time from the perspective of a salesman. The original activity constraints for this example were given in Table 2.4. Table 3.1 presents an example of a more flexible approach. Given the selection of a luxury package, the salesman would obviously prefer to solve the problem of the further selection of sun-roof and air-conditioner components. However, if this is not possible because of other constraints (component availability, for example) a solution with a lesser satisfaction degree is to consider the sun-roof or air-conditioner only.

3.3 A Specific Instance of DFCSP

This book develops in detail an instance of *rrDFCSP* based on fuzzy constraint satisfaction [42] (see Chapter 2). The justification for this choice is

Antecedent	Activity Constraint	Satisfaction
$x_{Package} = \text{luxury}$	$active(x_{Sunroof}) \wedge active(x_{AirConditioner})$	0.75
$x_{Package} = \text{luxury}$	$active(x_{AirConditioner})$	0.5
$x_{Package} = \text{luxury}$	$active(x_{Sunroof})$	0.25

Table 3.1. Flexible Activity Constraints for Car Configuration

the power and generality of the individual dynamic and flexible components. *rrDCSP* was the first dynamic CSP framework to be proposed, and has been researched the most extensively. Fuzzy constraint satisfaction, although a relatively recent development, has also been the subject of much research effort. It offers a range of constraint types and, through an idempotent aggregation operator, is computationally more tractable than other methods based on summation. Hence, this instance of *rrDFCSP* should prove to be sufficiently powerful to find a range of uses and as an indicator of the utility of DFCSP in the general case.

3.3.1 The Flexible Component - a Recap

For clarity, a short recap is given of fuzzy constraint satisfaction. A more complete description was presented in Chapter 2. Constraints are represented by a fuzzy relation, R , which maps from an assignment tuple to the satisfaction scale, $L = \{l_{\perp}, l_1, l_2, \dots, l_{\top}\}$. Three types of constraint supported by this framework are used herein. Note, however, that the proposed approach is entirely compatible with the use of the more powerful constraint types supported by fuzzy CSP, such as conditional or hierarchical constraints [42].

1. *Preference* constraints. A consistency degree is attached to each tuple allowed by the constraint. The satisfaction degree of the constraint is that of the tuple specified by the assignments to the constrained variables.
2. *Prioritised* constraints. A priority degree is associated with the constraint. If the constraint is violated, the satisfaction degree of the constraint is inversely proportional to its priority.
3. *Prioritised-preference* constraints. A combination of the above two types. Below a certain satisfaction threshold, the satisfaction degree of the constraint is calculated from its priority. Otherwise, the satisfaction degree of the constraint comes from the tuple specified by the assignments to the constrained variables.

The consistency degree in L of a partial assignment is calculated from the aggregated membership values of the constraints involving the subset of the variables. Aggregation, or the conjunctive combination of satisfaction degrees, is made by the *min* operator.

3.4 Fuzzy *rr*DFCSP Solution via Branch and Bound

The first solution technique developed for the *rr*DFCSP instance described in the previous section is based on the branch and bound approach to solving static fuzzy CSP [42] (see Sections 2.9.1 and A.9). As a static solution technique, it solves each instance in the dynamic sequence from scratch. However, the solution to the previous problem in the sequence is maintained and is used to make savings in two ways:

1. Before attempting to solve the current problem in the dynamic sequence, the solution to the previous problem is checked against the current problem. If the previous solution also constitutes a solution with $l_{\top}$ satisfaction degree for the current problem, no search is required for the current problem. If it constitutes a solution for the current problem with a satisfaction degree l_i , such that $l_{\perp} < l_i < l_{\top}$, the solution is stored and used to set the necessary bound prior to search. Hence, any search branch which will necessarily lead to a satisfaction degree l_h , such that $l_h \leq l_i$, is pruned.
2. During search when attempting to solve the current problem, the solution to the previous problem is used to guide the heuristic choice domain element selection. Given several possibilities with equivalent satisfaction degrees, this heuristic prefers to make an assignment which matches that of the solution to the previous problem, if possible. This is a simplified version of the oracles method described in Section 2.10.1.

3.5 Fuzzy *rr*DFCSP Solution via Local Repair

As noted in Section 2.10.1, local repair is a successful *rr*DCSP and *rc*DCSP technique [167, 172]. To recap briefly: the solution to the previous problem in the dynamic sequence is used as a starting point. The assignments that comprise this solution are modified until an acceptable solution to the current problem in the sequence is found. Given a graded level of solution consistency, this approach could be implemented as either a hill-climbing method (which would protect against changes in the problem during the solution period) or a complete method which would guarantee finding the optimal solution to each problem in the sequence. In this case, a complete method has been chosen, to facilitate both the application of the solution technique to the domain of AI planning, as described in Chapters 5-8, and the comparison with the complete branch and bound method described above.

The second solution technique developed for fuzzy *rr*DFCSP is a novel algorithm which extends the Local Changes *rr*DCSP local repair algorithm [167] (see Section 2.10.1) to support fuzzy constraints. The following subsection provides a detailed description of Local Changes and is followed by the definition of the new Flexible Local Changes fuzzy *rr*DFCSP algorithm.

3.5.1 Local Changes

Local Changes searches for a solution by resolving conflicts that occur when examining the solution to the previous problem in the sequence in the context of the current problem. To control this search, whilst retaining maximum flexibility in allowing variables to be reassigned according to which ones are in conflict at any particular stage, LC divides the variable set X into three subsets. These are X_1 , X_2 and X_3 (such that $X_1 \cup X_2 \cup X_3 = X$). Their meanings are as follows: X_1 is the set of variables whose assignments are fixed and is used to ensure termination of the algorithm as will be shown; X_2 is the set of variables whose assignments are not fixed and X_3 is the set of variables currently without assignments.

Initially, only X_2 and X_3 are used. X_2 contains those variables that were also part of the previous problem in the sequence, and which therefore have assignments from the solution obtained to that problem. Hence, X_2 is the set of assignments that will be repaired to solve the current problem. X_3 contains variables that were not part of the previous problem, and which therefore do not yet have assignments. LC must search for consistent assignments to these variables. At the start of a problem sequence there is of course no previous problem, so X_2 is empty and X_3 contains all variables in the initial problem.

The following description of LC is made with reference to Figure 3.7. The exact nature of the search depicted is unimportant, but note the way in which the search process is controlled via the three variable sets. Starting from scratch or a partial assignment, as reflected by the state of X_2 and X_3 , LC selects and attempts to assign a value to $x_i \in X_3$ (the selected variable is shown in bold in the figure after its transfer from X_3 to X_2). If this assignment does not conflict with any assignments, LC selects another value from X_3 and so on.

If, however, the assignment violates one or more constraints, the algorithm instigates a course of repairs to resolve these inconsistencies. The repair is made with respect to the assignment to x_i , hence x_i is placed in X_1 to prevent a change to its assignment. The next step is to unassign the subset of X_2 which is causing the conflict with x_i . This subset is found by examining the set of violated constraints. The algorithm is then used recursively to solve the sub-problem of re-assigning these variables given the fixed assignment to x_i and the assignments to variables remaining in X_2 . A sub-problem is denoted in the figure by a bounded area.

During the solution of one sub-problem further repairs may be necessary, in which case another variable is added to X_1 and so on. If no repair is possible (since one or more of the variables causing the conflict are in X_1) the algorithm backtracks to solve the parent sub-problem in a different way. X_1 may be seen as a type of tabu list [101], preventing a potentially infinite cycle of repairs between mutually conflicting variables and therefore ensuring the termination of the algorithm.

Fig. 3.7. The Local Changes Algorithm: Example Search Structure

3.5.2 Flexible Local Changes: A Fuzzy *rr*DFCSP Algorithm

The Flexible Local Changes (FLC) algorithm presented here is an extension of the LC algorithm to support fuzzy constraints. The method of identifying particular conflicts and then focusing on the sub-problem of resolving those conflicts used by the LC algorithm enables an effective use of the extra information afforded by fuzzy CSP in pruning the search space.

The FLC algorithm is structurally similar to LC in its use of the variable sets X_1 , X_2 and X_3 to control the search. The flexibility afforded by the fuzzy CSP framework means that FLC searches for an optimal assignment, as opposed to the way LC searches for an assignment that does not violate any hard constraint. FLC repeatedly selects a variable, $x_i \in X_3$ and assigns to it the domain element which results in the highest possible consistency given the set of currently assigned variables in $X_1 \cup X_2$. In making this assignment repairs may be necessary to elements of X_2 , which are made in the same manner as described in Section 3.5.1. The algorithm terminates upon finding an optimally consistent complete variable assignment for the current problem. This is a more complex process than for hard constraints, but the extra information available can be used to minimise the workload of the algorithm.

A pseudo-code description of the FLC algorithm is presented below; a proof of its soundness and completeness appears in Appendix B. Initially, some useful notation is defined which is used throughout the description of the algorithm:

- *rc* is the *repair cutoff* which is used to record the current best consistency achieved thus far for the problem (or a sub-problem during repair). Any

branch of the search in which the consistency of the problem/sub-problem is necessarily less than or equal to this value may be pruned.

- $consX_1$ and $consX_{12}$ record the consistencies of the assignments to the variables in X_1 and $X_1 \cup X_2$ respectively. These two pieces of information are used to incrementally calculate current consistency degrees. Incremental consistency computation is critical to efficiency: by maintaining $consX_1$ and $consX_{12}$ it is possible, when calculating consistencies based on adding a newly assigned variable, x_i , to either set (notated as $consX_1x_i$ and $consX_{12}x_i$ respectively) to consider only those constraints that involve x_i [42].

The FLCsolve() Procedure

Figure 3.8 presents the procedure FLCsolve() which prepares for the main body of the algorithm. FLCsolve() determines the consistency of a partial assignment gained from a solution to the previous problem with respect to the current problem structure and initialises X_1 , X_2 and X_3 accordingly.

initTest() (line 5) computes the consistency degree of the initial members of X_2 , i.e. the assignment which is the solution to the previous problem, and creates a set, C_r , containing all constraints whose individual consistencies are below the highest point in the satisfaction scale, $l_\top$ (where $l_\perp$ is the corresponding lowest point in L). If X_3 is empty initially, rc is set to the consistency of the complete assignment left from the previous problem and the initial solution is stored in *initSolution* in case no better solution exists. Otherwise, rc is set to $l_\perp$ since there is not yet a complete assignment to compare future progress against.

Having established X_1 , X_2 , X_3 and C_r , it is now necessary to decide which elements of X_2 must be re-assigned to remove all inconsistencies. choose() (line 10) is responsible for returning such a variable set. For each $c_{r_i} \in C_r$, choose() unassigns at least one variable (removing it from X_2 and placing it in X_3). Since no c_{r_i} is now fully assigned, none of these constraints are considered to be violated and the consistency of the remaining assigned variables is $l_\top$.

The chosen set of elements of X_2 is then unassigned and removed from X_2 . The problem is then to re-assign these variables in a maximally consistent fashion (line 12). If the resulting assignment is not better than the initial complete assignment (assuming that the previous solution constituted a complete assignment to the current problem), then the initial assignment is restored (lines 13-16).

The FLCVariables() Procedure

The FLCVariables() procedure, as described in Figure 3.9, is responsible for selecting, via some heuristic, $x_i \in X_3$ for assignment. The FLCVariable()

```

01. Procedure FLCSolve()
02.    $X_1 \leftarrow \emptyset$ 
03.    $X_2 \leftarrow$  all currently assigned variables
04.    $X_3 \leftarrow$  all currently unassigned variables
05.    $initcons \leftarrow \text{initTest}(X_2, C_r)$  //  $C_r$  created as a side effect
06.   If  $X_3 = \emptyset$ 
07.      $rc \leftarrow initcons$ 
08.      $initSolution \leftarrow \text{assignments}(X_2)$ 
09.   Else  $rc \leftarrow l_\perp$ 
10.    $X'_3 \leftarrow \text{choose}(X_2, C_r) \cup X_3$ 
11.    $X'_2 \leftarrow X_2 - X'_3$ 
12.    $result \leftarrow \text{FLCVariables}(X_1, X'_2, X'_3, l_\perp, l_\top, rc)$ 
13.   If  $result < rc$ 
14.      $\text{assignments}(X_2) \leftarrow initSolution$ 
15.   Return  $rc$ 
16.   Else Return  $result$ 
17. End Procedure

```

Fig. 3.8. Pseudo-code for FLCSolve()

procedure described below is used to make the assignment to the x_i selected which has the highest possible consistency degree with respect to the current assignments to the variables in X_1 (line 4). This process may involve the re-assignment of elements of X_2 . $consX_{12}x_i$ is then tested against rc . If an assignment to x_i which improves upon the current best for this problem/sub-problem does not exist, then there is no point in continuing this branch of the search (line 5). If such an assignment does exist, the algorithm continues to select another variable from X_3 (line 6).

The FLCVariable() Procedure

The FLCVariable() procedure (see Figure 3.10) makes an assignment to x_i , possibly in conjunction with repairs to X_2 , which leads to an optimal consistency rating with respect to fixed set of variables, X_1 . Three pieces of information are recorded which may be thought of under the umbrella term ‘best assignment information’ (lines 2-4): $bestd_{ij} \in D_i$, when assigned to x_i

```

01. Procedure FLCVariables( $X_1, X_2, X_3, consX_1, consX_{12}, rc$ )
02.   If  $X_3 = \emptyset$  Return  $consX_{12}$ 
03.   Select And Remove  $x_i \in X_3$ 
04.    $consX_{12}x_i \leftarrow \text{FLCVariable}(X_1, X_2, x_i, consX_1, consX_{12}, rc)$ 
05.   If  $consX_{12}x_i < rc$  Return  $l_\perp$  //Backtrack point
06.   Return FLCVariables( $X_1, X_2 \cup \{x_i\}, X_3, consX_1, consX_{12}x_i, rc$ )
07. End Procedure

```

Fig. 3.9. Pseudo-code for FLCVariables()

```

01. Procedure FLCVariable( $X_1, X_2, x_i, consX_1, consX_{12}, rc$ )
02.    $bestd_{i_j} \leftarrow \text{firstElement}(D_i)$ 
03.    $bestcons \leftarrow l_{\perp}$ 
04.    $bestX_2 \leftarrow \text{assignments}(X_2)$ 
05.    $X_2Store \leftarrow \text{assignments}(X_2)$ 
06.   While  $D_i$  Not Exhausted And  $bestcons < consX_{12}$ 
07.     Select And Remove  $d_{i_j} \in D_i$ 
08.     assign( $x_i, d_{i_j}$ )
09.      $consX_1x_i \leftarrow \min(consX_1, \text{test}X_1(X_1, x_i, bestcons, rc))$ 
10.     If  $consX_1x_i > \max(bestcons, rc)$ 
11.        $consX_{12}x_i \leftarrow \min(consX_1x_i, consX_{12},$ 
                                 $\text{test}X_{12}(X_1, X_2, x_i, consX_1x_i, consX_{12}, C_r))$ 
12.       If  $consX_{12}x_i > bestcons$ 
13.          $bestcons \leftarrow consX_{12}x_i$ 
14.          $bestd_{i_j} \leftarrow d_{i_j}$ 
15.          $bestX_2 \leftarrow \text{assignments}(X_2)$ 
16.       If  $C_r \neq \emptyset$ 
17.          $repaircons \leftarrow \text{FLCRepair}(X_1, X_2, x_i, consX_1x_i, consX_{12},$ 
                                          $C_r, \max(rc, bestcons))$ 
18.         If  $repaircons > bestcons$ 
19.            $bestcons \leftarrow repaircons$ 
20.            $bestd_{i_j} \leftarrow d_{i_j}$ 
21.            $bestX_2 \leftarrow \text{assignments}(X_2)$ 
22.            $\text{assignments}(X_2) \leftarrow X_2Store$ 
23.          $\text{assignments}(X_2) \leftarrow bestX_2$ 
24.         assign( $x_i, bestd_{i_j}$ )
25.       Return  $bestcons$ 
26. End Procedure

```

Fig. 3.10. Pseudo-code for FLCVariable()

and in conjunction with the assignments to X_2 recorded in $bestX_2$, leads to the optimum consistency ($bestcons$) with respect to X_1 . In addition, X_2Store (line 5) maintains a record of the initial state of X_2 so that it may be restored following a trial course of repairs. The first step is to initialise the above variables. Note that `FirstElement()` does not change the state of D_i .

The main loop of this procedure (lines 6-22) iterates through the domain D_i of x_i . The termination condition for this loop (line 6) is that either all elements of D_i have been tried or that an element d_{i_j} has been found which leads (perhaps including some repairs) to an assignment of equal consistency to $consX_{12}$. $consX_{12}$ represents the best possible consistency to the variables in $X_1 \cup X_2$ found by the previous invocation of FLCVariable(). The conjunctive combination of constraint satisfaction levels means that this consistency level cannot be improved by making further assignments.

Following an assignment to x_i , it is first necessary to establish $consX_1x_i$ (line 9). `testX1()` evaluates the consistency of constraints involving the variables $\{\{x_i\} \cup Y \mid Y \in P(X_1) - \emptyset\}$, where $P(X_i)$ is the power set of X_i . The

final value is incrementally calculated using the previous value of $consX_1$. $consX_1x_i$ can be used for pruning areas of the search tree (line 10): if $consX_1x_i$ is not better than both rc and $bestcons$, then there is no point pursuing this assignment since elements of X_1 cannot be repaired to increase this consistency subsequently.

If $consX_1x_i$ is of sufficiently high quality, the next stage is to establish $consX_{12}x_i$ (line 11). $testX_{12}()$ evaluates the consistency of constraints constraining the variables $\{\{x_i\} \cup Y \cup Z | Y \in P(X_2) - \emptyset, Z \in P(X_1)\}$, hence avoiding work done by $testX_1()$. This procedure also creates the set C_r containing constraints whose consistency is below $consX_1x_i$ and $consX_{12}$. The minimum of these provides an upper bound on the consistency of the sub-problem of repairing the constraints in C_r , so there is no point in repairing individual constraints whose consistency is at least as high as this level.

If the assignment to x_i is found to be better than the best assignment so far (line 12), the best assignment information is updated (lines 13-15).

If C_r is non-empty following $testX_{12}()$, a course of repairs is instigated to try and improve on the consistency of $consX_{12}x_i$ by re-assigning elements of X_2 (lines 16-22). The $FLCRepair()$ procedure returns the consistency of the optimal assignment to elements of X_2 given fixed assignments to $X_1 \cup \{x_i\}$ (line 17). For the sub-problem, rc is updated to $\max(rc, bestcons)$ to reflect the consistency that the repair must achieve to be worthwhile. Following a successful repair, the best assignment information is updated accordingly (lines 19-21). Following every attempted repair, X_2 is restored to its state preceding this invocation of $FLCVariable()$ (line 22).

Once every element of D_i has been tested or an assignment of sufficient quality has been found, the best assignment information is used to set X_2 and x_i accordingly (lines 23-24). The consistency of the best assignment is then returned (line 25).

The $FLCRepair()$ Procedure

The $FLCRepair()$ procedure coordinates the solution of the sub-problem of re-assigning elements of X_2 in order to improve $consX_{12}x_i$. The $choose()$ procedure is again used to decide which elements of X_2 to re-assign (line 2), according to the set of constraints which must be repaired, C_r . The efficiency of the algorithm is dependent on $choose()$ - the smaller the set of variables to repair the better. In the case of binary constraints, simply unassigning the other variable (i.e. not x_i) for each $c_{r_i} \in C_r$ results in an optimally small set of variables to reassign [167]. In the case of k -ary constraints a tradeoff exists between searching for a small set of variables to repair versus the overhead involved [108].

For this course of repair x_i is added to X_1 , as the repair is made in the context of the assignment to this variable. Hence, $consX_1$ is updated to $consX_1x_i$ for the sub-problem (line 3). $consX_{12}$ is initially $\min(consX_{12}, consX_1x_i)$ for the sub-problem since all constraints whose consistencies fell below this level

```

01. Procedure FLCRepair( $X_1, X_2, x_i, \text{cons}X_1x_i, \text{cons}X_{12}, C_r, rc$ )
02.    $X'_3 \leftarrow \text{choose}(X_2, C_r)$ 
03.   Return FLCVariables( $X_1 \cup \{x_i\}, X_2 - X'_3, X'_3,$ 
                         $\text{cons}X_1x_i, \min(\text{cons}X_{12}, \text{cons}X_1x_i), rc$ )
04. End Procedure

```

Fig. 3.11. Pseudo-code for FLCRepair()

are being repaired. FLCVariables() is then used recursively to solve the sub-problem (line 3).

3.5.3 FLC Complexity Issues

The time complexity of FLC is dominated by the number of constraint checks that are made from within $\text{test}X_1()$ and $\text{test}X_{12}()$. The incremental calculation of the satisfaction degree (see Section 3.5.2) limits constraint checking upon the instantiation of x_i to variables adjacent to it in the constraint graph. Therefore, when $|X_1| + |X_2| = a$ the maximum number of constraint checks required when considering x_i is am , assuming a uniform domain size of m . The worst case occurs when $|X_1| = 0, |X_2| = a$ initially, allowing the maximum possible number of repairs.

In the first instance, a maximum of m possible repairs can be instigated from this assignment with respect to each of the possible assignments to x_i . At this point, x_i is placed into X_1 and (in the worst case) 1 variable is removed from X_2 . Hence, the sum $|X_1| + |X_2|$ remains as a . Each of the m sub-problems, therefore, also requires a maximum of am constraint checks. This process continues up to a times, whereupon $|X_1| = a$ and no further repairs are possible. Hence, the number of constraint checks when trying to find the optimum assignment to a subset of the variables of size a is:

$$a \sum_{b=1}^a m^b$$

As partial assignments are extended towards complete assignments, there are n such subsets, giving the time complexity of FLC:

$$\sum_{a=1}^n a \sum_{b=1}^a m^b = O(nm^n)$$

The space complexity of the FLC algorithm is dominated by the need to store the assignments to variables in X_2 for the duration of a course of repairs. The current best assignment to elements of X_2 must be stored as well as the initial assignment to these variables which can then be restored if necessary following a tentative repair process (see Figure 3.10, lines 4, 5). In the worst case, starting from a complete assignment to all n variables, repairs may be

instigated on sub-problems involving $n - 1, n - 2, \dots, 1$ variables, requiring the storage of $2n, 2(n - 1), 2(n - 2), \dots, 2$ assignments respectively. Hence, the space complexity of FLC:

$$2 \sum_{i=0}^n i = n(n + 1) = O(n^2)$$

3.6 Fuzzy Arc Consistency

As mentioned in Section 2.9.4, the idempotent nature of the *min/max* operators employed by the fuzzy constraint satisfaction framework enables the support of consistency enforcing techniques. In particular, fuzzy arc consistency [42] can significantly enhance the performance of both branch and bound and FLC. A fuzzy CSP is said to be fuzzy arc consistent if any assignment involving one variable with a satisfaction degree $l_i \in L$ can be extended to any 2 variables, maintaining a satisfaction degree of l_i . Note that the general concept of k -consistency applies in the same manner [42], but is not explored here.

Enforcing fuzzy arc consistency is essentially the same process as for classical CSP. The degree of support in L must be established for each $d_{i_j} \in D_i$ using the `FRevise()` procedure presented in Figure 3.12. Note the general similarity with the classical `Revise()` procedure given in Section A.5. The support for an individual domain element is recorded for further propagation as a unary constraint. As the support for a whole domain is enumerated, the domain is effectively annotated with the consistency degree of each constituent element.

`FRevise()` applied to $arc(i, j)$ principally establishes the level of support for each $d_{i_a} \in D_i$ with respect to D_j (lines 4-13). When considering a particular $d_{j_b} \in D_j$, this is achieved through taking into account the unary consistency of d_{i_a} and d_{j_b} as well as the consistency degree associated with the pair of assignments by $c(x_i, x_j)$. Using the conjunctive combination rule, the support given to d_{i_a} by d_{j_b} is the minimum of these three values (line 7). The maximum support for d_{i_a} with respect to $arc(i, j)$ is the maximum support offered by the elements of D_j (line 8). Similarly to classical `Revise()`, *changed* is set to be `true` if the unary consistency degree of an element of D_i is changed (line 11).

The variable *arcHeight* records the maximum consistency degree possible for $arc(i, j)$. This is simply the maximum support for the elements of D_i . This value can then be used to update β , the upper bound on the consistency of the problem as a whole. Via the conjunctive combination rule, this is simply the minimum of the heights of every constraint arc. This bound may then be used by subsequent search procedures to terminate as soon as a solution with a satisfaction degree of β is found.


```

01. Procedure FRevise( $i, j, \beta$ )
02.    $changed \leftarrow false$ 
03.    $arcHeight \leftarrow l_{\perp}$ 
04.   Foreach  $d_{i_a} \in D_i$ 
05.      $d_{i_a} support \leftarrow l_{\perp}$ 
06.     Foreach  $d_{j_b} \in D_j$ 
07.        $singleSupport \leftarrow \min(\mu_{R_i}(d_{i_a}), \mu_{R_{i_j}}(d_{i_a}, d_{j_b}), \mu_{R_j}(d_{j_b}))$ 
08.        $d_{i_a} support \leftarrow \max(d_{i_a} support, singleSupport)$ 
09.     If  $d_{i_a} support = l_{\perp}$  Delete  $d_{i_a}$ 
10.     If  $d_{i_a} support \neq \mu_{R_i}(d_{i_a})$ 
11.        $changed \leftarrow true$ 
12.        $\mu_{R_i}(d_{i_a}) \leftarrow d_{i_a} support$ 
13.        $arcHeight \leftarrow \max(arcHeight, d_{i_a} support)$ 
14.    $\beta \leftarrow \min(\beta, arcHeight)$ 
15.   Return  $changed$ 
16. End Procedure

```

Fig. 3.12. Enforcing Flexible Arc Consistency via FRevise()

The FAC3() procedure (see Figure 3.13) operates in the same way as AC3() (see Section A.7): if FRevise(i, j, β) updates any element of D_i , then all possibly affected arcs are re-revised. Note that a change in this case is not necessarily the deletion of an element, but the lowering of its unary consistency degree. FAC3() also maintains and returns the upper bound, β , on the consistency degree of the problem as a whole.

3.6.1 The Complexity of Fuzzy Arc Consistency

It has been shown that the complexity of the FAC3() procedure is of the order of $O(\ell m^3 e)$ [42], where ℓ is the number of satisfaction degrees in the scale L . This is due to the fact that each element may be revised (in the worst case) down from $l_{\top}$ to $l_{\top-1}$, etc. until finally it is deleted when its consistency degree reaches $l_{\perp}$. The increased complexity of enforcing fuzzy arc consistency is expected to be clearly visible in the empirical results presented in Chapter 4.

```

01. Procedure FAC-3()
02.    $\beta = l_{\top}$ 
03.    $Q \leftarrow \text{All arcs}(i, j)$ 
04.   While  $Q$  Not Empty
05.     Select And Remove  $arc(i, j)$  From  $Q$ 
06.     If FRevise( $i, j, \beta$ )
07.       Add To  $Q$  All arcs( $h, i$ ) ( $h \neq i, h \neq j$ )
08.   Return  $\beta$ 
09. End Procedure

```

Fig. 3.13. The FAC-3 Algorithm

3.6.2 Pre-processing with Fuzzy Arc Consistency

The simplest way to use fuzzy arc consistency (in conjunction with either branch and bound or FLC) is a pre-processing phase prior to invoking the main algorithm on each problem. Both types of algorithm benefit from the upper bound, β , on the consistency of the whole problem since it facilitates early termination when a solution of satisfaction degree of β is found. This is possible because the pre-processing phase established that no better solution exists.

FLC derives an additional benefit from β : when examining the previous solution with respect to the current problem it is only necessary to repair constraints with consistencies below β , leading to greater solution reuse than is possible with FLC. This is because FLC lacks β and so repairs all constraints whose consistency degree is less than $l_{\top}$, regardless of the fact that a solution of consistency $l_{\top}$ may not exist. FLC may also employ β when considering which constraints to repair during search.

3.6.3 Hybrids

It is also possible to enforce various levels of arc consistency during the search process to form *hybrid* algorithms [111] (see Section 2.4). The mechanisms by which hybrids are constructed based on branch and bound and FLC are different, however. In the case of branch and bound, whose search proceeds chronologically, the assignment and unassignment of variables proceeds in an ordered manner such that it can be guaranteed that a particular set of assignments is fixed for the duration of a sub-problem. Hence, a branch and bound/consistency-enforcing hybrid can be constructed in the same manner as for the non-flexible case, as described in Section 2.4.

Consistency is enforced with respect to the current variable assignments at each search node, enabling the calculation of a more accurate upper bound, β , on the satisfaction degree of the remaining sub-problem. As per basic branch and bound, if this bound does not exceed α , the satisfaction degree of the best solution found thus far, this search branch can be pruned. The more consistency enforcing is performed, the more accurate β becomes, and the more pruning is possible.

FLC's search procedure, however, does not proceed chronologically: any and all variables in the set X_2 may be unassigned in any order to effect necessary repairs. Hence, it is difficult to enforce consistency based on assignments to elements of X_2 since any changes brought about by such propagation must be undone if unassignment takes place. This is a potential weakness of this approach which will be tested in the following chapter, since it restricts the amount of constraint propagation possible.

The solution proposed for the LC algorithm for *rr*DCSPs is to enforce consistency with respect to X_1 [167], since it *is* guaranteed that assignments to variables in this set are fixed for the duration of a sub-problem. Hence,

Fig. 3.14. Sub-problem Composition Under Repair

propagated changes based on assignments to elements of X_1 cannot be undetermined by the repair process. This is also the solution adopted for FLC.

Having established that consistency must be enforced with respect to X_1 , it remains to determine which set of variables' domains to filter. Consider the situation described in Figure 3.14a. If $x_5 \in X_3$ is selected and instantiated and the current assignments to x_3 and x_4 are found to be inconsistent, the sub-problem of resolving these inconsistencies is depicted in Figure 3.14b. Since only variables $\{x_1, x_2, x_3, x_4\}$ can possibly be present in the sub-problem, it is at first sight only worth filtering the domains of these variables with respect to an assignment to x_5 .

In the general case, when considering an assignment to x_i , the domains of the current elements of X_2 are updated through consistency enforcing. This is because it is the elements of X_2 which are unassigned to be repaired in the sub-problem for which x_i is fixed.

To combat the problem of the reduced constraint propagation possible for FLC, there is the possibility of extracting some bounds information from the elements of X_3 (variables x_6, x_7, x_8 in the example). Although these variables will not feature in the repair sub-problem, they *do* have to be assigned eventually, and their assignments may interact with the variables under consideration for repair. Consider again the example given in Figure 3.14. If consistency enforcing showed, for example, that it was always the case that the current assignment to x_5 precluded any assignment to x_6 because of some constraint $c(x_5, x_6)$, then there is little point in resolving the conflicts with x_1 and x_2 . In general, constraint propagation using the elements of X_3 can contribute in this way to the calculation of β for the repair sub-problem.

```

01. Procedure FRevise( $i, j, \beta, \delta$ )
02.    $changed \leftarrow false$ 
03.    $arcHeight \leftarrow l_{\perp}$ 
04.   Foreach  $d_{i_a} \in D_i$ 
05.      $d_{i_a} support \leftarrow l_{\perp}$ 
06.     Foreach  $d_{j_b} \in D_j$ 
07.        $singleSupport \leftarrow \min(\mu_{R_i}(d_{i_a}), \mu_{R_{i_j}}(d_{i_a}, d_{j_b}), \mu_{R_j}(d_{j_b}))$ 
08.        $d_{i_a} support \leftarrow \max(d_{i_a} support, singleSupport)$ 
09.     If  $d_{i_a} support \leq \delta$ 
10.       Delete  $d_{i_a}$ 
11.        $d_{i_a} support \leftarrow l_{\perp}$ 
12.        $changed \leftarrow true$ 
13.     Else If  $d_{i_a} support \neq \mu_{R_i}(d_{i_a})$ 
14.        $changed \leftarrow true$ 
15.        $\mu_{R_i}(d_{i_a}) \leftarrow d_{i_a} support$ 
16.        $arcHeight \leftarrow \max(arcHeight, d_{i_a} support)$ 
17.    $\beta \leftarrow \min(\beta, arcHeight)$ 
18.   Return  $changed$ 
19. End Procedure

```

Fig. 3.15. Enhancing *FRevise*() via a Deletion Threshold

3.6.4 The Deletion Threshold

Once a necessary bound (i.e. the repair cutoff for FLC) for a sub-problem is established, this information can be exploited by consistency enforcing hybrids of both branch and bound and FLC. Given, for example, a necessary bound of $l_i \in L$, all domain elements that can be shown to have a unary satisfaction degree of $l_j \in L$ such that $l_j \leq l_i$ may be deleted for the duration of the sub-problem. The justification for this comes directly from the conjunction of satisfaction degrees implemented using the *min* operator: any single assignment with a satisfaction degree of l_i or less can never form part of a solution to the sub-problem with a satisfaction degree of greater than l_i .

The ability to delete domain elements early has two main benefits to the search process. Firstly, it saves further processing of the deleted elements with the *FRevise*() procedure. Without using the deletion threshold, each of these elements could potentially have been revised i more times before reaching $l_{\perp}$ and being deleted. Secondly, it is not necessarily the case that constraint propagation would reduce the satisfaction degrees of the affected elements to $l_{\perp}$. Hence, they might never be deleted. In this case, these elements would of course immediately be discarded by the search process, but their presence degrades the operation of heuristics based on selecting variables with the smallest domain first (as described in Section 2.5).

Figure 3.15 presents an updated version of *FRevise*(). Lines 9-11 support the deletion threshold, δ , removing domain elements with unary consistency degrees not above δ .

```

01. Procedure FLCVariable( $X_1, X_2, x_i, consX_1, consX_{12}, rc$ )
02.    $bestd_{ij} \leftarrow \text{firstElement}(D_i)$ 
03.    $bestcons \leftarrow 0$ 
04.    $bestX_2 \leftarrow \text{assignments}(X_2)$ 
05.    $X_2Store \leftarrow \text{assignments}(X_2)$ 
06.   While  $D_i$  Not Exhausted And  $bestcons < consX_{12}$ 
07.     Select And Remove  $d_{ij} \in D_i$ 
08.      $\text{assign}(x_i, d_{ij})$ 
09.      $testX_1result \leftarrow \text{testX}_1(X_1, x_i, bestcons, rc)$ 
10.      $consX_1x_i \leftarrow consX_1 + testX_1result$ 
11.     If  $consX_1x_i > \max(bestcons, rc)$ 
12.        $consX_{12}x_i \leftarrow consX_{12} + testX_1result +$ 
          $\text{testX}_{12}(X_1, X_2, x_i, consX_1x_i, consX_{12}, C_r)$ 
13.       If  $consX_{12}x_i > bestcons$ 
14.          $bestcons \leftarrow consX_{12}x_i$ 
15.          $bestd_{ij} \leftarrow d_{ij}$ 
16.          $bestX_2 \leftarrow \text{assignments}(X_2)$ 
17.       If  $C_r \neq \emptyset$ 
18.          $repaircons \leftarrow \text{FLCRepair}(X_1, X_2, x_i, consX_1x_i, consX_{12},$ 
            $C_r, \max(rc, bestcons))$ 
19.         If  $repaircons > bestcons$ 
20.            $bestcons \leftarrow repaircons$ 
21.            $bestd_{ij} \leftarrow d_{ij}$ 
22.            $bestX_2 \leftarrow \text{assignments}(X_2)$ 
23.          $\text{assignments}(X_2) \leftarrow X_2Store$ 
24.        $\text{assignments}(X_2) \leftarrow bestX_2$ 
25.        $\text{assign}(x_i, bestd_{ij})$ 
26.     Return  $bestcons$ 
27. End Procedure

```

Fig. 3.16. Adapting FLCVariable() to Additive Flexible CSP

3.7 Solution Techniques for other DFCSP Instances

Both the branch and bound and Flexible Local Changes algorithms could also be used to support other instances of *rrDFCSP* through modifications to the satisfaction aggregation method used. Adaption to, for example, other priority-maximisation flexible methods, such as necessity-based FCSP (see Section 2.9.4), is immediate. This is because the aggregation of satisfaction degrees uses the same *min* and *max* operators.

Branch and bound and FLC could be used with additive FCSP by modifying the algorithm to use summation operators. The conversion to an additive method is relatively straightforward. Consider, for example, Figure 3.16 which presents a modified version of FLCVariable() using additive satisfaction aggregation. The calculation of the satisfaction degrees of the variable subsets $X_1 \cup \{x_i\}$ and $X_1 \cup X_2 \cup \{x_i\}$ has been updated (see lines 9-12) as necessary. This framework can support any type of additive FCSP (see Sec-

Fig. 3.17. Example Problem

tion 2.9.3), according to the implementation of `testX1()` and `testX12()`. Hence, this one algorithm provides the foundation for the exploration of a substantial part of the matrix of DFCSP instances and solution techniques described in Figure 3.2.

3.8 An Example

Figure 3.17 presents an example concerning a course schedule (adapted from [42]) which will be used to illustrate the operation of the basic (i.e. non-pre-processing and non-hybrid) versions of the branch and bound and FLC approaches.

The variables x_1 , x_2 and x_3 denote the number of lectures, exercises and training sessions in the course respectively. The preference (priority p_3) of Professor A to give about 4 lectures is represented by c_1 , while the desire of Dr B to give 3 or 4 exercise sessions (priority p_2) is expressed by c_2 . The fact that there should be about 1 additional training session is represented by c_3 and the hard constraint c_4 stipulates that there must be 7 sessions in total.

3.8.1 Solution of Initial Problem via Branch and Bound

The branch and bound approach solves the problem as described in Figure 3.18. Since this is the first problem in the sequence, the problem is treated as a static fuzzy CSP. Note that in the figure, a triangle denotes a sub-tree in which the algorithm failed to find a solution better than the best currently known. The domain element heuristic used is to prefer the assignment with the highest unary satisfaction degree first. The algorithm quickly finds the solution $\{x_1 = 4, x_2 = 1, x_3 = 2\}$ with the satisfaction degree l_2 . This bound

Fig. 3.18. Branch and Bound Solution to the Problem Given in Figure 3.17

on further solutions allows it to effectively prune the search tree during the rest of the solution process. Hence, the optimal solution $\{x_1 = 3, x_2 = 3, x_3 = 1\}$, with satisfaction degree l_3 , is found shortly afterwards.

3.8.2 Solution of Initial Problem via FLC

FLC solves this problem as shown below, using the heuristic that the domain element giving the best consistency with respect to the current assignment is chosen first. Note that the solution procedure is only shown up to the point that an optimal solution is found. In reality, the algorithm must exhaust all other avenues to determine that no better assignment is possible.

- Select x_1 . Relevant constraint: $c_1(\mathbf{x}_1)$.
 - $x_1 \leftarrow 4$, consistency = $l_\top$.
 - $X_1 = \{ \}$, $X_2 = \{x_1\}$, $X_3 = \{x_2, x_3\}$.
- Select x_2 . Relevant constraint: $c_2(\mathbf{x}_2)$.
 - $x_2 \leftarrow 3$, consistency = $l_\top$.
 - $X_1 = \{ \}$, $X_2 = \{x_1, x_2\}$, $X_3 = \{x_3\}$.
- Select x_3 . Relevant constraints: $c_3(\mathbf{x}_3)$, $c_4(x_1, x_2, \mathbf{x}_3)$.
 - For all assignments, consistency = $l_\perp$.
 - Start repair from $x_3 \leftarrow 1$ (best local assignment).
 - Repair $c_4(x_1, x_2, x_3)$: Choose x_1 .
 - $X_1 = \{x_3\}$, $X_2 = \{x_2\}$, $X_3 = \{x_1\}$.
- Select x_1 . Relevant constraints: $c_1(\mathbf{x}_1)$, $c_4(\mathbf{x}_1, x_2, x_3)$.
 - $x_1 \leftarrow 3$, consistency = l_3 .
 - Assignment $\{x_1 = 3, x_2 = 3, x_3 = 1\}$, consistency = l_3 , is optimal.

3.8.3 The Problem Changes

The next time the course is run (see Figure 3.19), it is expanded to 8 sessions (c_4) and the preference is now for a greater number of training sessions (c_3).

Fig. 3.19. Example Problem - Part 2

Additionally, other commitments mean that Professor A prefers to give fewer lectures (c_1). The solution procedures for both algorithms on this updated problem are shown below. This simple example illustrates the savings that can be achieved through the maintenance of effort from previous problems.

3.8.4 Solution of Updated Problem via Branch and Bound

The solution to the previous problem in the sequence is first examined with respect to this new problem. Since it violates the hard constraint and therefore has a satisfaction degree of $l_{\perp}$, it does not help in setting a necessary bound on the satisfaction degree of any new solution. However, the previous solution *is* used during domain element selection: the assignments $x_1 = 3$ and $x_2 = 3$ are made because they match the assignments to those variables in the previous solution. Hence, the optimal solution $\{x_1 = 3, x_2 = 3, x_3 = 2\}$ is found with little effort.

3.8.5 Solution of Updated Problem via FLC

The solution FLC finds, based on the solution to the original problem, is shown below. An examination of the solution to the previous problem reveals that constraints c_3 and c_4 are violated. Since these constraints share a variable, x_3 , it is possible that by reassigning just this one variable, both constraints may be satisfied. This proves to be the case: by reassigning x_3 to element 2, the optimal solution is found.

- Evaluate assignment $\{x_1 = 3, x_2 = 3, x_3 = 1\}$: consistency = $l_{\perp}$.
 - Repair $c_3(x_3), c_4(x_1, x_2, x_3)$: Choose x_3 to cover both.
 - $X_1 = \{\}, X_2 = \{x_1, x_2\}, X_3 = \{x_3\}$.

Fig. 3.20. Branch and Bound Solution to the Problem Given in Figure 3.19

- Select x_3 . Relevant constraints: $c_3(\mathbf{x}_3)$, $c_4(x_1, x_2, \mathbf{x}_3)$.
 - $x_3 \leftarrow 2$, consistency = l_3 .
 - Assignment $\{x_1 = 3, x_2 = 3, x_3 = 2\}$, consistency = l_3 , is optimal.

3.9 Summary

This chapter has argued for the combination of two separately successful extensions to the classical CSP framework: dynamic CSP and flexible CSP respectively. This new framework, dynamic flexible CSP (DFCSP), is able to solve a variety of problems previously beyond the scope of current CSP techniques. Since there are several instances of both dynamic and flexible solution techniques, these may be combined to produce a matrix of DFCSP instances and associated solution techniques.

One particular DFCSP instance, founded on the combination of fuzzy flexible CSP and restriction/relaxation-based dynamic CSP (*rrDCSP*) was discussed in detail. Fuzzy *rrDFCSP* was chosen as being representative of a variety of useful situations. Two algorithms were developed to solve fuzzy *rrDFCSP*s. The first was developed from the perspective of extending a static flexible algorithm to solve dynamic problems. This branch and bound-based algorithm uses the solution to the previous problem in a dynamic sequence both as a means to set a necessary bound on the satisfaction degree of the solutions it searches for in the current problem, and also as a heuristic guide during search in an attempt to match assignments in the current problem to those in the solution to the previous problem.

The second algorithm was developed from the opposite viewpoint. Flexible Local Changes (FLC) is a new algorithm based upon the extension of the existing Local Changes *rrDCSP* algorithm to support fuzzy constraints. FLC does not start the search from scratch for each new instance in a dynamic

sequence, but attempts to directly modify the solution to the previous problem until an acceptable solution to the current problem is found. Complexity issues for this algorithm were addressed, and a proof of its soundness and completeness is presented in Appendix B

The use of fuzzy arc consistency to enhance both algorithms was discussed in detail. Used as a pre-processing phase, it establishes an upper bound, β , on the satisfaction degree of the solution to the current problem. FLC also derives the extra benefit that constraints with a satisfaction degree equal to or above β do not need to be repaired.

In addition, the construction of hybrid search/consistency enforcing variants of both branch and bound and FLC were discussed. Although they require different search mechanisms, hybrids of both algorithms work on the principal of propagating the effects of assignments already committed to in the search process. It was also shown how the extra effort of enforcing consistency on fuzzy constraints can be minimised by filtering domain elements whose unary consistency falls below a *deletion threshold*, defined in terms of the necessary bound on the solution.

The following chapter contains a detailed empirical analysis of fuzzy *rrDFCSPs*, comparing the performance of several variants of the branch and bound and FLC algorithms over a set of random problem sequences.

4 An Empirical Study of Fuzzy *rr*DFCSPs

This chapter presents an empirical study of the solution of random fuzzy *rr*DFCSPs. The algorithms used are variants of the core branch and bound and Flexible Local Changes algorithms presented in the previous chapter. Due to the size of the study (almost 40,000 individual problems solved tens of times each), full results are presented separately in Appendix C. In this chapter, therefore, a selection has been made to show graphically and in tables where appropriate to reflect the general trend of the data as a whole.

The purpose of this study is twofold. Firstly, it provides a comparison of the performance of the branch and bound and FLC algorithms over a variety of problem instances. Furthermore, the performance of heuristics developed for these algorithms is analysed. Secondly, the structure of the fuzzy *rr*DFCSPs themselves is analysed. Of special interest are the conditions under which these problems can be expected to be most difficult to solve.

4.1 The Problems

The fuzzy *rr*DFCSPs examined in this chapter are uniformly composed of sequences of ten related fuzzy CSP instances. For the purposes of this study, binary constraints only are considered, following the approach of many studies of classical CSP [66, 128]. A preliminary study of non-binary fuzzy *rr*DFCSPs is made in [108].

The first random problem instance in a dynamic sequence is generated using the following parameters:

- n . The number of variables. This parameter takes values from $\{20, 30, 40\}$ in order to gauge the effects of increasing problem size.
- m . The size of the domains. This parameter is fixed at 6 throughout to limit the size of this study. Further studies should examine the effects of varying this parameter.
- ℓ . The number of consistency degrees in L . This parameter takes values from $\{3, 4, 5\}$ in order to examine the effects of an increasing amount of flexibility on the difficulty of finding the ‘best’ solution.
- con . The connectivity of the constraint graph, which is the proportion of the variables which are related by a constraint. This parameter takes

values from $\{0.25, 0.5, 0.75\}$, allowing an examination of the effects of an increasingly connected graph.

- t . The tightness of the constraints: the proportion of possible assignment combinations that are disallowed by each constraint. This parameter takes values from $\{0.1, 0.2, \dots, 0.8, 0.9\}$. Values of 0 and 1 are not examined since they produce trivially solvable and unsolvable problems respectively.

The granularities of the connectivity and tightness parameters have been chosen in order to make this study feasible. Of course, interesting phenomena may occur between the values chosen (as is presented in the results below). Future studies might investigate such phenomena by considering an increased number of values for both con and t , if only within restricted ranges.

The set of constraints is divided into an equal number of fuzzy prioritised, preference, and prioritised-preference types. The possible priority degrees (defined by ℓ) are evenly distributed amongst the prioritised and prioritised-preference constraints. Similarly, the possible preference degrees are evenly distributed amongst the tuples of each preference and prioritised-preference constraint.

In order to produce changes in the problem structure to approximate a dynamically changing problem, restriction/relaxation operations are performed at random. A further parameter, ch , the size of the changes determines the proportion of constraints that are removed and replaced¹. To maintain a uniform connectivity of the constraint graph, the same number of constraints are added as are taken away. In addition, when a constraint of a certain type is removed, it is replaced by a constraint of the same type. The tightness of the new constraint is as specified for the original instance. If it is a prioritised or prioritised-preference constraint, the priority assigned is the same as the removed constraint to maintain the original distribution.

The justification for the effort of maintaining a uniform problem structure is to eliminate the effects of differing proportions of fuzzy constraint types or priority/preference distributions from the results. Further studies will be necessary to determine the effects of varying these proportions on the difficulty of the problem instances generated and that of solving the sequence as a whole.

For each parameter combination, five dynamic sequences containing ten individual problem instances were generated. Mean results are reported over the five sequences, hence each point on the graphs presented and each table entry represents the solution of 50 individual problem instances. In all, 3645 dynamic sequences were generated, totalling 36450 individual problem instances to be solved.

¹ A special case used in this study is when $ch = \text{'1 constraint'}$, i.e. a single constraint is replaced between problem instances.

4.2 The Algorithms Studied

The algorithms tested on the random fuzzy *rrDFCSPs* are variations of the basic branch and bound and FLC algorithms described in Chapter 3. The performance of the non-hybrid variants such as basic branch and bound or FLC was relatively so poor that it was impractical to include them in this study. The variations considered are all based on enforcing fuzzy arc consistency (see Section 3.6), either before search as a pre-processing phase, and/or during search to form a hybrid search/consistency enforcing algorithm.

The naming convention for the different variants is as follows. An ‘AC’ prefix denotes that fuzzy arc consistency is established in a pre-processing phase prior to search. A suffix denotes the use of a measure of fuzzy consistency enforcing during search to form a hybrid. If the suffix is ‘FC’, then the consistency enforcing process corresponds to the classical CSP Forward Checking algorithm [77]: The `FRevise()` procedure (see Figure 3.15) is used to process the domains of variables as yet unassigned during search. Similarly, if the suffix is ‘FMAC’, then a hybrid is formed corresponding to the classical MAC algorithm: full fuzzy arc consistency is established initially and with respect to each new variable assignment. Unless otherwise stated, the deletion-threshold equipped version (see Section 3.6.4) of `FRevise()` is used. A comparison with Dynamic Backtracking [68] is omitted since, as recognised by [167], this algorithm has strong similarities with LC. Updating Dynamic Backtracking to support dynamic CSPs and flexible constraints would produce a very similar algorithm to FLC.

The variants of branch and bound tested are as follows:

- BBFC
- ACBBFC
- BBFMAC

A similar number of basic variants of FLC are also tested, as detailed below. In addition, three extra hybrids are tested which also take into account the domains of variables outwith those immediately in line for repair, as detailed in Section 3.6.3. These are denoted by the extra suffix ‘ft’ (for ‘full test’).

- FLCFC, FLCFCft
- ACFLCFC, ACFLCFCft
- FLCFMAC, FLCFMACft

4.3 Evaluation Criteria

There are three criteria by which the algorithms’ performance will be judged. The first two are the standard measures of performance of classical CSP solution techniques. The last is a measure specific to dynamic problems.

- *Constraint checks.* Every time a constraint is queried for the consistency degree of a pair of assignments, this is counted as a constraint check.
- *Search nodes expanded.* Every assignment of $d_{ij} \in D_i$ to x_i corresponds to a node in the search tree.
- *Solution stability.* Since, in this study, the set of variables does not change within a dynamic sequence, the stability of the solution to each problem as compared with the last can easily be measured as the proportion of assignments that are the same. It is appropriate to use this measure, since there may be a number of solutions with optimal satisfaction degrees. Solution stability is often a desirable property in reality, since the changes needed to move from the existing solution to the new solution are minimised.

Although stability is not an explicit optimisation criterion, it is expected that FLC's method of repairing the previous solution and the domain element selection heuristics of both branch and bound and FLC (see below) will produce stable solutions. In future, preservation of stability could be added as a means of breaking ties between solutions with optimal consistency degrees, although at a greater computational cost.

Run-times are not reported, although they have been observed to track consistency checks. Due to the size of the study, the tests were performed over a large network of machines of varying capabilities and under differing loads. It would be difficult, therefore, to place any great faith in run-time results. Other possible criteria are the number of backtracks made by the algorithms and the number of branches in the search tree. A more detailed future study should also consider these measurements.

4.4 Heuristics Investigated

As described in Chapter 2, the three operations which are ordered heuristically are the selection of a variable to instantiate next, the choice of domain element to instantiate, and the order in which constraints are checked. The heuristics used for each operation for both branch and bound and FLC are described below.

4.4.1 Variable Selection

As the analogue of classical Backtrack, it is simple to use the Brelaz heuristic [19] with branch and bound. That is, to select the variable with smallest remaining domain first, breaking ties by preferring the variable with the greatest connectivity to the currently unassigned variables.

FLC may also use a Brelaz-type heuristic, but the search structure of this algorithm, controlled by the sets X_1 , X_2 and X_3 provides the possibility of creating novel variations on the variable selection heuristic. Four variable

heuristics are tested for the FLC variants, and they are enumerated here for easier reference in the following sections.

1. **SmallestD**: The smallest domain heuristic is used as a baseline and simply selects the variable with the smallest remaining domain.
2. **SmallestD- X_2** : This heuristic takes advantage of the structure of a sub-problem identified in Section 3.6.3. That is, only elements of the set of currently assigned variables, X_2 , can be modified in a repair sub-problem. Hence, this heuristic starts with the smallest domain approach and breaks ties using connectivity restricted to the elements of X_2 .
3. **SmallestD- X_3** : This heuristic also starts with the smallest domain approach, breaking ties using connectivity restricted to the elements of X_3 . In some sense it is the opposite of the **SmallestD- X_2** heuristic.
4. **SmallestD- X_{23}** : Finally, this heuristic starts by selecting the variable with the smallest remaining domain and breaks ties using the connectivity of the variables in $X_2 \cup X_3$, that is, all those variables whose assignments are free to change.

Note that even though elements of X_3 do not figure in the sub-problem of repairing assignments, they do have to be assigned eventually and hence interact with the current variable. Unless otherwise stated, the **SmallestD- X_{23}** heuristic is employed, since it typically produces the most competitive results.

4.4.2 Domain Element Selection

The domain element selection heuristic is the main method by which the branch and bound algorithms make use of dynamic information. In order to attempt to find the best solutions first, the element with the highest unary consistency degree is chosen first. However, if there are several such elements, ties are broken by examining the solution to the previous problem in the sequence, and if one of the tied elements matches the assignment in the previous solution, it is chosen.

For FLC, value selection is split into two cases, where repairs are and are not necessary respectively. Taking the latter, simpler case first, it is natural to select the element with the highest unary consistency degree first in order to find the solution with the highest consistency degree earlier. As per branch and bound, however, ties may be broken by selecting, if possible, the same domain element for a particular variable as appeared in the solution to the previous problem.

When repairs are necessary, it is intuitive to try to limit the amount of repairs needed/select a repair that is most likely to succeed. The justification for this is that in the irreparable case, all alternatives must be searched anyway to ensure completeness. The domain element selection heuristics for repair are enumerated below:

1. **ConsDeg**: The simplest heuristic is to select the element with the highest unary consistency degree. In the hybrid case as studied here, this is the element that results in the highest upper bound, β , on the consistency degree of the repair sub-problem.
2. **ConsDeg- C_r** : A more sophisticated heuristic breaks ties on the above by selecting the element that causes the smallest number of variables to be unassigned for repair, i.e. that which minimises $|C_r|$.
3. **ConsDeg-*con***: An alternative approach to heuristic 2 is to break ties on the connectivities of the variables to be repaired, minimising this value.
4. **ConsDeg-PrevSoln-*con***: The final, most complex heuristic tested follows the approach of the branch and bound algorithm to an extent in first breaking ties by matching against the previous solution. If this fails, ties are broken by minimising the connectivities as per the **ConsDeg-*con*** heuristic.

Unless otherwise stated, the **ConsDeg-PrevSoln-*con*** heuristic is used throughout, since it too generally provides the most competitive results.

4.4.3 Constraint Check Selection

In order to focus on the most constrained part of the problem (see Section 2.5), the size of the domains D_i, D_j at the end of each $\text{arc}(x_i, x_j)$ are checked. The heuristic prefers those arcs with the smallest summed domain size. Ties are broken by comparing each D_i , since this is the domain being filtered. Unless otherwise stated, this heuristic is used by all algorithms tested in this chapter.

4.5 Results: 3-point Satisfaction Scale

The first set of experiments concerns the case where $\ell = 3$, i.e where just one extra consistency degree has been added to the usual Boolean pair of $l_\perp$ and $l_\top$. The three hybrids of branch and bound are tested against three hybrids of FLC. The ‘ft’ variants hold a small but consistent advantage over those variants of FLC that do not perform constraint propagation on elements of X_3 (see Appendix C). For clarity, only those results concerning the ‘ft’ variants are presented in this chapter.

Figure 4.1 presents results for: $ch = 1$ constraint, $n = 20$ and $con = 0.25$. The first thing that is apparent is that the forward-checking algorithms require many fewer constraint checks than the algorithms maintaining fuzzy arc consistency. In terms of search nodes, however, the roles are mostly reversed, with the arc consistency-maintaining algorithms performing significantly better. This is unsurprising: by performing more constraint propagation, these algorithms can detect dead ends in the search tree more quickly.

Fig. 4.1. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

A second feature evident in the figure is the presence of two phase transitions. This is unsurprising given the two degrees of consistency available above $l_{\perp}$: a transition exists from when the generated problems are solvable

to degree $l_{\top}$ to being unsolvable to degree $l_{\top}$ yet still solvable to degree l_1 . A further transition occurs from when the problems are solvable to degree l_1 to being completely unsolvable.

Within the constraint check results, these transitions are clear for those algorithms that do not perform an initial fuzzy arc consistency step, but not for those that do. The cost, in constraint checks, of enforcing fuzzy arc consistency initially (known to be more expensive than enforcing Boolean arc consistency, see Section 3.6.1) dominates the cost of the rest of the search. Compare, for example, the results for ACBBFC and BBFC, ACFLCFCft and FLCFCft. Both pairs of algorithms differ only in the enforcement or not of fuzzy arc consistency initially. The peak observed for the pre-processing algorithms is therefore indicative of a phase transition in the cost of enforcing fuzzy arc consistency also observed in Boolean problems [65].

A single peak might be explained by the fact that enforcing arc consistency, as opposed to searching for a solution, is centred around constraint propagation. When the constraints are less tight, independent of the number of consistency degrees, less propagation is possible, hence enforcing fuzzy arc consistency costs less. Conversely, when the constraints are tighter, strong propagation is possible, revising the unary consistency degree of many domain elements quickly. Between these two cases, revisions are possible, but in small amounts, resulting in the need to check many more constraints before fuzzy arc consistency is enforced.

Two phase transitions are present for all the algorithms in the search node results. The cost of the pre-processing step is factored out in these results, which reflect only the search process itself. Enforcing fuzzy arc consistency initially does, however, reduce the size of the subsequent search tree. Again, this may be seen by comparing the results for ACBBFC and BBFC, ACFLCFCft and FLCFCft.

The connectivity is increased to 0.75 while the other parameters' values are maintained in the results presented in Figure 4.2. Since these problems are significantly more difficult to solve, enforcing fuzzy arc consistency is no longer the dominant cost in terms of constraint checks. This explains the more uniform behaviour observed for each algorithm. The superior propagation ability of the branch and bound based algorithms noted in Section 3.6.3 is clearly evident in these results. The fact that a single transition peak is visible is due to the resolution of the tightness axis: the high connectivity means that the phase transitions occur more quickly as the tightness parameter is increased [128].

Figure 4.3 presents the results for: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$. The superiority of the branch and bound algorithms on more difficult problems is underlined. The larger number of variables has a similar effect to raising the connectivity, accounting for the single visible peak.

Table 4.1 presents stability results for $ch = 1$ constraint. Note that the last row of entries per $\langle n, con \rangle$ tuple are typically based on stability results

Fig. 4.2. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$.

for fewer problems since this row coincides with the final phase transition to unsolvable problems. In this case, stability is measured between the previous solvable problem and the current solvable problem, rather than simply be-

Fig. 4.3. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$.

tween adjacent problems in the fuzzy *rrDFCSP* sequence. This is true of the stability results presented throughout this chapter and Appendix C.

Given the low level of change between problem instances, the performance of all six algorithms is quite similar. However, the FLC variants do have a small but consistent advantage over their branch and bound counterparts. Furthermore, ACFLCFCft and FLCFMACft are also consistently better by this measure than FLCFCft. This is as predicted in Section 3.6.2: fuzzy arc consistency pre-processing enables a smaller initial repair from the assignment constituting the solution to the previous problem in the sequence.

A preliminary examination of the relationship between the difficulty of solving a particular fuzzy *rr*DFCSP and the ability of the algorithms tested to maintain stable solutions reveals little at this stage. It might be expected that poorer stability results would be obtained around the phase transitions observed in Figure 4.1 where there are fewer solvable instances of the associated consistency degree. If anything, the reverse is true, with slight peaks in the stability results being visible close to the phase transitions.

The reduced number of solutions at the hardness peaks may therefore actually be helping the algorithms maintain stability: given fewer solutions to pick from and small changes between problem instances, the likelihood is increased that a dynamic algorithm will find a solution similar to that chosen for the previous problem.

Results for settings of $ch = 0.1$ and 0.25 are presented in Figures 4.4 and 4.5, with the other parameters set to: $\ell = 3$, $n = 20$, $con = 0.25$. In terms of constraint checks, the BBFMAC algorithm performs most poorly in the face of an increasing amount of change from problem instance to problem instance. The number of constraint checks required by the other algorithms is not significantly different from those presented in Figure 4.1.

The number of search nodes required by each algorithm does steadily increase with ch , but the increase is evenly distributed across the set of algorithms. This trend is indicative of the extra effort required to maintain a stable solution in the face of an increasingly dynamic problem structure.

Stability results for $ch = 0.1$ may be examined in Section C.1, Table C.3. Table 4.2 presents the equivalent results for $ch = 0.25$. In the presence of higher levels of dynamicity, the results are, as expected, lower than those given in Table 4.1, for each algorithm. In addition, a clearer picture emerges concerning the relationship between problem structure and the stability results achievable by the algorithms tested.

Initial observations that the most stable solutions are found for problems around the phase transitions are confirmed, becoming more evident as ch is increased. This effect is easiest to observe at the phase transition between l_2 and l_1 solutions. Since solutions of both consistency degrees are actually classed as solutions (as opposed to a complete assignment with an $l_\perp$ consistency degree) it is easy to measure their relative stability.

Again, the FLC variants produce slightly more stable solutions overall. The gap in between the performance FLCFCft and that of ACFLCFCft and FLCFMACft is more pronounced than in the $ch = 1$ constraint case, high-

n	con	t	BBFC	ACBBFC	BBFMAC	FLCFCft	ACFLCFCft	FLCFMACft
20	0.25	0.1	91.66	91.55	90.55	95.37	95.74	96.48
20	0.25	0.2	95.44	96.44	97.0	98.33	99.07	99.07
20	0.25	0.3	95.22	97.66	97.88	94.07	99.62	99.62
20	0.25	0.4	91.11	93.33	95.33	92.03	97.96	97.96
20	0.25	0.5	92.77	94.55	96.44	90.37	94.25	96.85
20	0.25	0.6	92.5	100.0	100.0	100.0	100.0	100.0
20	0.5	0.1	98.55	99.11	99.11	98.7	99.62	99.62
20	0.5	0.2	95.88	97.33	97.33	97.22	99.07	99.07
20	0.5	0.3	92.66	95.66	95.77	98.88	99.62	99.62
20	0.5	0.4	62.5	62.5	62.5	62.5	62.5	65.0
20	0.75	0.1	96.77	98.55	98.55	100.0	99.62	99.62
20	0.75	0.2	94.44	98.22	97.88	97.77	99.07	99.07
20	0.75	0.3	100.0	100.0	100.0	100.0	100.0	100.0
30	0.25	0.1	99.48	99.48	99.77	95.92	100.0	100.0
30	0.25	0.2	97.99	98.22	99.03	91.35	99.5	99.5
30	0.25	0.3	96.51	98.22	98.29	92.09	99.25	99.01
30	0.25	0.4	91.03	96.88	97.03	91.48	96.79	97.53
30	0.5	0.1	99.55	99.85	99.85	100.0	100.0	100.0
30	0.5	0.2	98.14	98.07	98.29	98.39	99.62	99.62
30	0.75	0.1	99.03	99.03	99.92	100.0	99.75	99.75
30	0.75	0.2	100.0	100.0	100.0	100.0	100.0	100.0
40	0.25	0.1	99.72	99.72	99.72	100.0	99.9	99.9
40	0.25	0.2	97.83	97.88	98.05	98.14	99.9	99.9
40	0.25	0.3	97.88	97.44	97.61	97.87	98.98	98.61
40	0.5	0.1	99.27	98.33	98.33	100.0	99.9	99.9
40	0.5	0.2	99.53	99.53	99.53	100.0	100.0	99.68
40	0.75	0.1	96.05	98.83	98.27	99.53	100.0	100.0

Table 4.1. Solution Stability. $\ell = 3$, $ch = 1$ constraint.

lighting the advantage in terms of stability results of performing fuzzy arc consistency pre-processing. The results for the BBFMAC algorithm are comparable to those for BBFC and ACBBFC, suggesting that the solution-reuse heuristic employed by the branch and bound algorithms, while successfully maintaining stable solutions, are causing this algorithm in particular to perform many more constraint checks, as presented in Figures 4.1 and 4.5.

The variants of branch and bound and FLC that enforce fuzzy arc consistency, whether as a pre-processing step or during search, consistently produce more stable solutions than forward checking versions. Enforcing fuzzy arc consistency provides a bound, β , on the consistency of the remaining problem (see Section 3.6). In the case of FLC, constraints already satisfied to consistency degree at least β need not be repaired, aiding stability by not unnecessarily disturbing assignments. Similarly for branch and bound, a stable assignment of at least consistency degree β can be preferred over an-

Fig. 4.4. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.1$, $n = 20$, $con = 0.25$.

other with higher consistency degree: in all circumstances the overall problem cannot have a consistency degree exceeding β .

Fig. 4.5. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$.

Given the utility of a more informed bound on the remaining problem, it is unsurprising that BBFMAC and FLCFMACft produce, almost uniformly, the most stable solutions of the branch and bound and FLC variants respec-

n	con	t	BBFC	ACBBFC	BBFMAC	FLCFCft	ACFLCFCft	FLCFMACft
20	0.25	0.1	43.88	43.33	43.55	51.29	49.07	47.96
20	0.25	0.2	77.44	79.0	81.33	66.11	75.74	84.07
20	0.25	0.3	77.44	82.0	82.22	62.59	90.92	91.48
20	0.25	0.4	65.44	68.88	70.44	57.4	81.29	81.66
20	0.25	0.5	48.77	60.77	62.44	59.62	69.81	72.77
20	0.25	0.6	41.75	36.08	36.83	42.77	48.05	48.05
20	0.5	0.1	78.66	86.66	87.33	57.59	92.4	92.4
20	0.5	0.2	62.33	71.88	69.99	47.22	84.81	84.81
20	0.5	0.3	50.44	56.55	53.33	54.44	65.55	61.29
20	0.5	0.4	35.0	35.0	35.0	37.5	39.99	37.5
20	0.75	0.1	76.55	79.99	80.11	42.77	91.48	91.48
20	0.75	0.2	53.44	55.11	55.66	51.48	66.29	62.4
20	0.75	0.3	19.16	22.5	20.83	27.5	27.5	22.5
30	0.25	0.1	86.22	88.88	91.7	57.4	90.37	92.22
30	0.25	0.2	73.62	81.85	81.85	51.85	89.01	89.01
30	0.25	0.3	59.4	69.92	68.88	54.19	79.62	77.16
30	0.25	0.4	41.7	46.29	46.29	48.51	50.61	47.28
30	0.25	0.5	26.66	26.66	26.66	20.0	16.66	16.66
30	0.5	0.1	74.59	79.11	79.11	45.67	90.0	90.0
30	0.5	0.2	46.96	54.88	49.62	50.24	60.86	56.54
30	0.5	0.3	20.0	20.0	20.0	23.33	23.33	26.66
30	0.75	0.1	62.22	66.07	66.74	43.95	78.27	78.27
40	0.25	0.1	82.5	85.05	85.61	45.37	93.24	93.24
40	0.25	0.2	62.11	67.33	65.0	50.37	83.7	83.51
40	0.25	0.3	47.5	51.44	53.94	49.81	52.4	59.53
40	0.5	0.1	66.88	71.22	70.22	40.55	83.98	83.98
40	0.5	0.2	31.93	33.6	31.31	29.58	31.72	31.24
40	0.75	0.1	45.99	52.5	53.0	52.68	59.44	59.81

Table 4.2. Solution Stability. $\ell = 3$, $ch = 0.25$.

tively. Enforcing fuzzy arc consistency appears to be particularly important to the stability of FLC. Without this, FCLFCft produces less stable solutions than BBFC, whereas ACFLCFCft and FLCFMACft maintain a consistent advantage over their branch and bound counterparts. The advantage of the FLC algorithms is also a result of the type of search they employ. Not only are stable assignments preferred at all choice points, but FLC's repair-based search actively seeks to leave in place assignments that match the previous solution.

4.6 Results: 4-point Satisfaction Scale

The second set of experiments concerns the case where $\ell = 4$. The increased flexibility afforded by the greater number of consistency degrees is expected to entail a higher degree of difficulty in solving the problems generally. Tests are performed on the same set of algorithms for ease of comparison.

Figure 4.6 presents results for the case $ch = 1$ constraint, $n = 20$ and $con = 0.25$. The performance of the algorithms in terms of constraint checks is divided into three groups. BBFMAC clearly performs less well than the others. Since this algorithm performs the most constraint propagation of all those considered, the increased size of ℓ seems to have affected it to the greatest extent. Conversely, the BBFC and FLCFCft algorithms again produce the best results by a significant margin. The performance of the remainder falls in between these two extremes.

Compared with the equivalent results for $\ell = 3$ (see Figure 4.1), there is clear evidence of a greater number of constraint checks being required as ℓ increases. This follows expectations, with a higher number of consistency degrees producing a more complex problem.

The cost of fuzzy arc consistency pre-processing continues to dominate (for those algorithms that perform this step) on these relatively easy problems, obscuring the phase transition behaviour. The search node results more clearly exhibit the three transition peaks that are expected for $\ell = 4$. Again, those hybrids that perform the most work to spot dead ends in the search perform best by this measure.

The connectivity is increased to 0.75 while the other parameters' values are maintained in the results presented in Figure 4.7. The superior propagation ability of the branch and bound algorithms is much less evident (in terms of constraint checks) in these results than it was in Figure 4.2. This is likely to be due to a combination of factors. Firstly, as noted above, an increase in ℓ leads also to an increase in the effort required to enforce fuzzy arc consistency. Given that the other parameters have remained constant, therefore, an increase in ℓ from 3 to 4 may lead to the cost of enforcing fuzzy arc consistency beginning to dominate again.

The second factor contributing to these constraint checks results is that the true hardness peaks lie between the marks on the t axis. This fact can clearly be seen in the search nodes results. Recall that the advantage of the branch and bound algorithms only manifests in Figure 4.2 on the problems that are significantly more difficult.

Figure 4.8 presents the results for: $ch = 1$ constraint, $n = 40$, $con = 0.25$. Significantly more difficult problems are created by the combination of a higher setting for ℓ and the use of a larger number of variables, both as compared with simply raising the connectivity and with their equivalents for $\ell = 3$ (see Figure 4.3). As a result, the advantage of the branch and bound algorithms on the most difficult problems is clear once again.

Fig. 4.6. Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

Table 4.3 presents stability results for $ch = 1$ constraint. The higher number of consistency degrees has had very little detrimental effect on the quality of these results as compared to Table 4.1: the high proportion of similarity

Fig. 4.7. Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 1$ constraint, $n = 20$, $con = 0.75$.

between problem instances still allows each algorithm to maintain solution stability effectively. The advantage held by the FLC variants over the branch

Fig. 4.8. Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 1$ constraint, $n = 40$, $con = 0.25$.

and bound algorithms evident in the $\ell = 3$ data remains, as does that held by ACFLCFCft and FLCFMACft over FLCFCft.

n	con	t	BBFC	ACBBFC	BBFMAC	FLCFCft	ACFLCFCft	FLCFMACft
20	0.25	0.1	94.55	95.66	95.66	93.33	99.07	99.07
20	0.25	0.2	91.66	95.66	96.0	94.81	97.59	97.4
20	0.25	0.3	88.77	92.88	94.33	95.0	93.14	95.92
20	0.25	0.4	89.77	92.11	90.22	91.85	96.29	96.85
20	0.25	0.5	95.22	97.44	97.11	92.59	99.07	99.07
20	0.25	0.6	90.99	93.33	94.11	92.77	97.59	97.59
20	0.5	0.1	94.77	88.33	90.66	97.77	96.85	95.74
20	0.5	0.2	96.22	97.33	98.33	97.59	98.33	100.0
20	0.5	0.3	90.77	92.99	98.11	97.03	96.66	97.96
20	0.5	0.4	94.77	94.44	93.0	94.81	96.85	96.85
20	0.75	0.1	98.88	99.44	99.55	100.0	100.0	100.0
20	0.75	0.2	99.55	96.33	97.55	99.07	100.0	99.62
20	0.75	0.3	94.22	94.55	94.55	98.33	99.25	97.4
30	0.25	0.1	94.74	96.44	96.44	93.08	98.88	98.88
30	0.25	0.2	93.33	92.66	91.85	94.81	96.41	95.18
30	0.25	0.3	98.59	98.44	97.7	96.04	97.65	99.25
30	0.25	0.4	95.55	97.03	96.81	94.19	99.01	99.13
30	0.25	0.5	95.18	94.74	92.37	96.54	94.19	95.8
30	0.5	0.1	99.92	99.11	99.92	100.0	100.0	99.87
30	0.5	0.2	97.85	98.0	99.4	97.28	100.0	100.0
30	0.5	0.3	97.4	96.37	94.81	96.66	99.75	100.0
30	0.75	0.1	98.74	98.29	99.55	100.0	100.0	99.25
30	0.75	0.2	97.33	97.48	97.48	98.27	98.39	100.0
40	0.25	0.1	93.72	95.22	95.16	94.53	94.62	97.49
40	0.25	0.2	98.72	98.22	98.38	91.94	98.24	99.72
40	0.25	0.3	97.66	99.38	98.55	100.0	98.98	99.81
40	0.25	0.4	96.77	98.18	97.24	99.35	98.51	99.81
40	0.5	0.1	99.83	99.83	99.5	100.0	100.0	100.0
40	0.5	0.2	98.55	95.5	96.66	99.44	97.96	99.07
40	0.75	0.1	99.77	99.66	99.72	100.0	100.0	100.0

Table 4.3. Solution Stability. $\ell = 4$, $ch = 1$ constraint.

Results for settings of $ch = 0.1$ and 0.25 are presented in Figures 4.9 and 4.10, with the other parameters set to $n = 20$ and $con = 0.25$ respectively. In terms of constraint checks, the BBFMAC algorithm responds most poorly to an increasing amount of change from problem instance to problem instance. This increased dynamicity appears to exacerbate the decline in behaviour by this algorithm observed in Figure 4.6. The number of constraint checks required by the other algorithms is not significantly different from those given in Figure 4.6. The number of search nodes required by each algorithm does steadily increase with ch , but the increase is evenly distributed across the set of algorithms.

Stability results for $ch = 0.1$ may be examined in Section C.1, Table C.15. Table 4.4 presents stability results for $ch = 0.25$. Following the trend

Fig. 4.9. Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 0.1$, $n = 20$, $con = 0.25$.

observed in the $\ell = 3$ results, in the presence of higher levels of dynamicity the results are lower than those presented in Table 4.3. When compared with the $\ell = 3$ results for equivalent settings of ch , presented in Tables C.3 and

Fig. 4.10. Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 0.25$, $n = 20$, $con = 0.25$.

4.2 respectively, a further decline in solution stability can be observed in the $\ell = 4$ results. The increased structural complexity introduced with the extra

n	con	t	BBFC	ACBBFC	BBFMAC	FLCFCft	ACFLCFCft	FLCFMACft
20	0.25	0.1	65.66	79.55	79.0	59.25	85.74	85.74
20	0.25	0.2	59.77	67.66	70.11	49.44	79.62	78.51
20	0.25	0.3	50.0	59.22	60.44	47.77	60.92	69.62
20	0.25	0.4	59.33	61.66	62.22	50.74	64.62	71.11
20	0.25	0.5	66.11	69.77	72.66	52.96	78.33	77.96
20	0.25	0.6	52.94	64.43	65.0	49.97	65.27	60.02
20	0.5	0.1	40.88	42.55	42.77	37.03	33.51	41.29
20	0.5	0.2	61.55	71.88	75.66	41.29	51.11	69.44
20	0.5	0.3	54.33	58.66	68.99	51.85	58.7	68.88
20	0.5	0.4	36.52	38.65	39.83	47.22	42.59	46.85
20	0.75	0.1	76.55	79.88	83.33	43.33	52.96	57.59
20	0.75	0.2	53.0	54.55	63.33	48.33	51.48	66.85
20	0.75	0.3	36.22	36.99	34.33	46.85	43.88	47.59
30	0.25	0.1	55.62	62.22	63.77	48.64	68.27	57.77
30	0.25	0.2	44.74	46.22	46.22	39.87	43.58	44.81
30	0.25	0.3	61.55	61.48	74.44	38.27	42.96	71.11
30	0.25	0.4	51.62	57.48	50.66	52.83	60.37	64.07
30	0.25	0.5	38.85	41.08	43.83	37.72	43.01	41.46
30	0.5	0.1	71.85	75.7	76.81	41.23	42.46	54.69
30	0.5	0.2	47.7	52.81	61.85	51.35	50.86	61.23
30	0.5	0.3	37.73	40.89	38.91	41.11	38.02	34.1
30	0.75	0.1	62.88	63.11	71.92	35.67	37.65	59.5
30	0.75	0.2	30.96	32.29	32.66	35.3	37.9	38.02
40	0.25	0.1	38.16	43.55	42.88	32.77	33.14	37.87
40	0.25	0.2	63.33	67.44	72.66	36.11	38.79	66.94
40	0.25	0.3	51.77	58.27	57.94	47.31	70.18	69.35
40	0.25	0.4	38.52	41.42	42.33	41.38	39.62	38.48
40	0.5	0.1	62.77	68.22	74.44	31.66	35.55	64.25
40	0.5	0.2	34.27	40.05	41.11	43.33	43.79	45.83
40	0.75	0.1	47.38	50.55	59.77	41.2	48.7	60.92

Table 4.4. Solution Stability. $\ell = 4$, $ch = 0.25$.

consistency degree appears to have a similar complicating effect on the task of maintaining stability as it has on searching for a solution.

The other patterns observed in the stability results previously continue to hold. For instance, peaks in stability continue to occur close to hardness peaks: compare the $n = 20$, $con = 0.25$ entries in Table 4.4 with Figure 4.10. In addition, the FLC variants, particularly ACFLCFCft and FLCFMACft still dominate the stability results.

4.7 Results: 5-point Satisfaction Scale

The largest number of consistency degrees considered is $\ell = 5$. This enables an observation of whether the increase in solution difficulty experienced when moving from $\ell = 3$ to $\ell = 4$ is matched when moving from $\ell = 4$ to $\ell = 5$. The same set of algorithms as in the previous two sections are investigated.

Figure 4.11 gives results for $ch = 1$ constraint, $n = 20$, $con = 0.25$. Although, in terms of constraint checks, the most difficult problems are no harder than those found for $\ell = 4$ (see Figure 4.6), the difficulty of the problems as t increases is maintained at a high level. The reason for this seems to be the increased number of phase transition peaks caused by the higher setting of ℓ .

The subdivision of algorithm performance with respect to this measure is also maintained, with BBFMAC clearly the poorest and the non-pre-processing forward checking variants clearly the best. The search nodes results exhibit the four transition peaks as clearly as is possible with the resolution used for t . An increase in effort from $\ell = 4$ to $\ell = 5$ is visible by this measure, both in terms of most difficult problem and on average as t increases.

The connectivity of the graph is increased to 0.75 in Figure 4.12. A significant increase in effort is evident when compared with the same results for $\ell = 4$ (see Figure 4.7). The large difference is, however, mostly in terms of the worst performing algorithm. BBFMAC, for example, performs almost twice as many constraint checks as it does on the equivalent $\ell = 4$ problems. The increase for the other algorithms is much more modest. Similarly, FLCFCft and ACFLCFCft require a significantly larger search tree, whereas the other algorithms are largely unaffected with regard to this measure by the increased number of consistency degrees.

When the number of variables is increased to 40 (see Figure 4.13), the problems generated are easier to solve in terms of both constraint checks and search nodes than their equivalents for the $\ell = 4$ case (see Figure 4.8). As noted previously, this is likely to be because the true peaks in hardness occur between the marks on the tightness axis.

Table 4.5 presents the stability results for $ch = 1$ constraint. In the face of such small changes to the problem structure, the ability of each algorithm to maintain solution stability remains largely unaffected (compare the results given in Table 4.1), despite the further increase in the number of consistency degrees.

An increase in ch to 0.1 and 0.25 is presented in Figures 4.14 and 4.15 respectively. In accordance with the other results in this section, although the peak in hardness is not greatly increased as compared with similar parameter settings for $\ell = 4$ (see Figure 4.14) the difficulty of solving the problems is on average higher as t increases.

Stability results for $ch = 0.1$ may be examined in Section C.1, Table C.24. Stability results for $ch = 0.25$ are presented in Table 4.6. A further decline in stability levels as ℓ is increased is evident. This may be seen clearly

Fig. 4.11. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

in comparison with the equivalent $\ell = 4$ results given in Tables C.15 and 4.4 respectively. Not unexpectedly, the general trend therefore appears to be that the ability of these algorithms to maintain stable solutions is affected

Fig. 4.12. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 1$ constraint, $n = 20$, $con = 0.75$.

principally by the amount of change present in the problem structure and the difficulty of identifying a solution to a problem instance.

Fig. 4.13. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 1$ constraint, $n = 40$, $con = 0.25$.

n	con	t	BBFC	ACBBFC	BBFMAC	FLCFCft	ACFLCFCft	FLCFMACft
20	0.25	0.1	89.11	91.11	91.22	88.33	98.51	97.03
20	0.25	0.2	89.44	90.99	93.44	91.29	96.11	97.59
20	0.25	0.3	91.66	94.44	95.33	90.18	92.77	95.18
20	0.25	0.4	91.11	91.99	91.11	87.77	91.85	92.59
20	0.25	0.5	90.0	91.66	90.55	92.03	94.99	94.99
20	0.25	0.6	89.22	92.44	92.11	94.25	96.66	96.66
20	0.25	0.7	90.83	90.83	90.83	98.33	99.16	99.16
20	0.5	0.1	91.22	94.55	96.55	98.33	95.92	97.59
20	0.5	0.2	87.88	89.66	88.77	89.62	91.29	95.92
20	0.5	0.3	95.77	94.44	95.77	88.14	97.77	95.37
20	0.5	0.4	90.88	92.22	92.66	94.44	94.81	97.03
20	0.5	0.5	100.0	100.0	100.0	100.0	100.0	100.0
20	0.75	0.1	90.11	88.44	88.33	91.66	92.96	95.55
20	0.75	0.2	97.44	98.44	97.22	100.0	100.0	99.81
20	0.75	0.3	95.22	92.33	93.99	100.0	100.0	100.0
30	0.25	0.1	94.37	97.18	95.85	97.03	96.04	96.79
30	0.25	0.2	92.51	94.81	95.92	91.23	97.65	98.39
30	0.25	0.3	85.92	87.7	87.11	90.74	97.28	95.06
30	0.25	0.4	97.4	99.11	98.81	97.4	98.88	99.5
30	0.25	0.5	88.56	95.68	93.98	95.78	95.87	95.89
30	0.5	0.1	96.29	96.59	97.11	98.51	99.01	99.38
30	0.5	0.2	97.18	96.59	98.14	96.54	96.17	100.0
30	0.5	0.3	97.85	96.44	96.14	96.41	98.76	100.0
30	0.75	0.1	98.66	98.66	99.62	100.0	100.0	100.0
30	0.75	0.2	96.81	95.92	95.85	100.0	98.88	99.62
40	0.25	0.1	94.44	93.72	96.44	95.37	97.5	99.81
40	0.25	0.2	90.38	97.61	96.27	93.14	99.81	95.92
40	0.25	0.3	98.11	96.22	97.88	99.35	96.85	97.12
40	0.25	0.4	91.94	96.72	95.22	96.38	98.05	96.2
40	0.5	0.1	99.72	99.94	99.77	100.0	100.0	100.0
40	0.5	0.2	95.55	96.66	95.66	98.79	97.96	98.24
40	0.75	0.1	99.11	98.83	99.22	100.0	100.0	99.9

Table 4.5. Solution Stability. $\ell = 5$, $ch = 1$ constraint.

Fig. 4.14. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 20$, $con = 0.25$.

Fig. 4.15. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.25$, $n = 20$, $con = 0.25$.

n	con	t	BBFC	ACBBFC	BBFMAC	FLCFCft	ACFLCFCft	FLCFMACft
20	0.25	0.1	56.55	64.66	63.33	49.81	70.0	67.03
20	0.25	0.2	56.99	63.66	68.11	46.66	71.29	74.62
20	0.25	0.3	56.66	62.77	63.0	45.55	69.07	66.11
20	0.25	0.4	50.44	62.11	64.77	49.81	66.85	67.59
20	0.25	0.5	53.11	58.77	61.44	45.74	57.03	64.44
20	0.25	0.6	54.65	63.06	63.12	46.89	71.48	72.36
20	0.25	0.7	50.0	48.33	50.0	36.66	51.66	51.66
20	0.5	0.1	40.0	57.55	66.77	33.7	68.33	70.0
20	0.5	0.2	44.44	38.33	41.66	37.4	47.59	44.25
20	0.5	0.3	52.44	63.33	62.22	39.62	47.03	61.11
20	0.5	0.4	42.22	47.66	47.22	43.88	52.59	54.44
20	0.5	0.5	35.99	45.99	45.0	45.83	56.66	56.66
20	0.75	0.1	38.66	40.11	42.66	33.88	38.51	35.0
20	0.75	0.2	51.66	59.44	56.11	39.44	42.96	64.44
20	0.75	0.3	33.55	43.22	49.33	47.4	47.59	54.81
30	0.25	0.1	57.48	62.74	68.81	33.58	52.09	71.23
30	0.25	0.2	47.11	57.7	54.88	40.24	61.48	64.69
30	0.25	0.3	46.74	51.48	49.4	38.64	46.17	45.18
30	0.25	0.4	52.96	57.18	59.85	43.45	53.08	67.28
30	0.25	0.5	42.69	49.01	52.28	47.03	54.42	58.1
30	0.5	0.1	48.29	53.55	55.7	33.2	34.56	30.86
30	0.5	0.2	55.18	59.77	57.55	39.87	40.86	62.96
30	0.5	0.3	40.37	44.96	44.22	44.93	42.59	52.71
30	0.75	0.1	62.14	67.33	71.11	35.67	32.46	44.19
30	0.75	0.2	34.96	37.11	44.96	41.85	45.92	45.67
40	0.25	0.1	45.77	56.49	57.55	38.7	57.4	60.09
40	0.25	0.2	50.55	56.05	54.72	33.14	31.38	39.35
40	0.25	0.3	55.16	54.05	61.33	41.38	44.62	59.16
40	0.25	0.4	40.0	41.94	48.0	44.44	44.44	51.94
40	0.5	0.1	63.11	67.55	75.22	30.18	31.94	35.18
40	0.5	0.2	39.38	41.38	47.55	48.24	48.05	54.72
40	0.75	0.1	55.38	54.88	57.44	38.98	41.75	50.0

Table 4.6. Solution Stability. $\ell = 5$, $ch = 0.25$.

4.8 The Utility of Dynamic Information

The two basic algorithms, branch and bound and FLC, both make extensive use of information stored in the solution to the previous problem in a dynamic sequence in an attempt to improve the efficiency of searching for a solution to the current problem. The question is how much is this information helping in the solution of a fuzzy *rrDFCSP* sequence? This section compares ‘crippled’ variants of branch and bound and FLC that do not have access to dynamic information against their fully dynamic counterparts. In the case of branch and bound, this means that it no longer has a heuristic ‘guide’ when making variable assignments, and FLC must solve each new problem instance from scratch.

The particular algorithms chosen for this part of the study are BBFC, BBFMAC, FLCFCft and FLCFMACft. This choice allows pre-processing and non-pre-processing versions of both algorithms to be tested. The suffix ‘noDyn’ indicates those algorithms whose dynamic capability has been removed. An asterisk (employed because of space constraints) denotes the same thing in respect of the tables given in this section.

Figure 4.16 presents results in terms of constraint checks and search nodes for $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$. With respect to constraint checks, it is immediately clear that BBFMAC suffers the most from losing the dynamic capability. It appears that this algorithm is using dynamic information to target the greater amount of propagation it does more effectively. For the other algorithms the benefit of making use of dynamic information is less pronounced in terms of constraint checks and at these parameter settings. The number of search nodes by each algorithm, however, *is* significantly affected. The use of dynamic information allows each algorithm to make a more informed exploration of the search tree, hence reducing its overall size.

As constraint tightness increases to the point that the problems are unsolvable, dynamic information is no longer helpful in reducing search effort. Nor does it hinder search, however, which might have been the case if the domain element selection heuristics had made poor choices with respect to the effort to prove unsolvability.

Table 4.7 gives the stability results for $\ell = 3$ and $ch = 1$ constraint. A clear and almost uniform improvement is maintained by all four algorithms over their non-dynamic counterparts, with the dynamic versions being up to one third better in some cases. The effects of increasing ch from a single constraint to 0.1 and 0.25 are presented in Figures 4.17 and 4.18. Unsurprisingly, the advantage of trying to re-use information from the solution to the previous problem in the sequence is eroded as the proportion of the problem which changes from instance to instance is raised. A clear advantage may still be seen however, even when fully one quarter of the problem changes between instances. Similarly, Tables 4.8 and 4.9 present stability results as ch is increased. Again, the advantage held by the dynamic algorithms is re-

Fig. 4.16. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

duced as more and more of the problem changes, but is still clearly visible at $ch = 0.25$.

Fig. 4.17. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.1$, $n = 20$, $con = 0.25$.

A higher number of consistency degrees are considered in Figures 4.19 and 4.20. The former gives results for $\ell = 4$, $ch = 0.25$, $n = 20$, $con = 0.25$ and the latter for $\ell = 5$, $ch = 0.1$, $n = 20$, $con = 0.25$. As the number of consistency

Fig. 4.18. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$.

degrees increases, the utility of making use of dynamic information remains. Stability results for the $\ell = 4$ and $\ell = 5$ cases are presented in Tables C.43

Fig. 4.19. Mean Constraint Checks/Search Nodes: $\ell = 4$, $ch = 0.25$, $n = 20$, $con = 0.25$.

and 4.10. As has been seen throughout this section, a strong advantage is maintained by the dynamic algorithms.

Fig. 4.20. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 20$, $con = 0.25$.

n	con	t	BBFC	BBFC*	BBFMAC	BBFMAC*	FLCFCft	FLCFCft*	FLCFMACft	FLCFMACft*
20	0.25	0.1	91.66	69.11	90.55	71.66	95.37	68.11	96.48	63.33
20	0.25	0.2	95.44	80.22	97.0	85.0	98.33	73.11	99.07	79.44
20	0.25	0.3	95.22	74.22	97.88	84.11	94.07	65.0	99.62	84.55
20	0.25	0.4	91.11	84.88	95.33	83.44	92.03	68.44	97.96	83.0
20	0.25	0.5	92.77	74.66	96.44	75.22	90.37	50.77	96.85	81.66
20	0.25	0.6	92.5	92.5	100.0	93.75	100.0	80.0	100.0	72.5
20	0.5	0.1	98.55	88.77	99.11	93.66	98.7	69.99	99.62	93.33
20	0.5	0.2	95.88	75.44	97.33	81.0	97.22	61.11	99.07	82.44
20	0.5	0.3	92.66	80.22	95.77	81.88	98.88	42.99	99.62	63.66
20	0.5	0.4	62.5	57.5	62.5	57.5	62.5	65.0	65.0	65.0
20	0.75	0.1	96.77	72.55	98.55	79.22	100.0	58.77	99.62	84.55
20	0.75	0.2	94.44	76.88	97.88	76.22	97.77	38.88	99.07	53.44
20	0.75	0.3	100.0	100.0	100.0	100.0	100.0	91.66	100.0	95.0
30	0.25	0.1	99.48	92.22	99.77	96.0	95.92	73.62	100.0	94.0
30	0.25	0.2	97.99	78.74	99.03	80.81	91.35	77.11	99.5	89.77
30	0.25	0.3	96.51	85.48	98.29	91.03	92.09	69.03	99.01	84.22
30	0.25	0.4	91.03	84.0	97.03	84.14	91.48	57.18	97.53	61.7
30	0.5	0.1	99.55	90.81	99.85	91.92	100.0	73.92	100.0	94.29
30	0.5	0.2	98.14	78.96	98.29	86.07	98.39	35.92	99.62	67.03
30	0.75	0.1	99.03	80.07	99.92	81.7	100.0	73.4	99.75	83.55
30	0.75	0.2	100.0	100.0	100.0	100.0	100.0	98.66	100.0	100.0
40	0.25	0.1	99.72	95.27	99.72	96.16	100.0	84.44	99.9	95.94
40	0.25	0.2	97.83	89.11	98.05	87.22	98.14	79.55	99.9	89.16
40	0.25	0.3	97.88	77.83	97.61	82.22	97.87	59.72	98.61	53.72
40	0.5	0.1	99.27	87.38	98.33	87.66	100.0	80.16	99.9	91.88
40	0.5	0.2	99.53	91.29	99.53	92.26	100.0	75.13	99.68	78.55
40	0.75	0.1	96.05	79.44	98.27	79.55	99.53	48.83	100.0	62.44

Table 4.7. Solution Stability. $\ell = 3$, $ch = 1$ constraint.

n	con	t	BBFC	BBFC*	BBFMAC	BBFMAC*	FLCFCft	FLCFCft*	FLCFMACft	FLCFMACft*
20	0.25	0.1	75.55	45.99	77.11	49.0	80.0	39.44	83.7	43.11
20	0.25	0.2	92.0	65.33	94.66	79.77	76.11	43.77	93.51	69.66
20	0.25	0.3	88.44	62.66	90.11	66.33	72.03	39.55	93.14	62.0
20	0.25	0.4	85.33	53.11	87.88	59.0	84.81	38.77	93.33	66.33
20	0.25	0.5	75.77	50.77	76.55	52.33	76.11	41.33	90.92	52.22
20	0.25	0.6	64.42	58.07	71.77	59.16	77.5	56.45	82.5	55.05
20	0.5	0.1	88.0	59.0	90.66	72.55	77.4	40.88	97.22	75.88
20	0.5	0.2	80.33	49.66	85.55	49.22	70.74	36.66	91.48	54.66
20	0.5	0.3	71.55	45.0	70.11	44.88	69.07	23.88	80.18	36.11
20	0.5	0.4	30.83	32.5	30.83	27.49	50.0	31.66	58.33	31.66
20	0.75	0.1	85.99	55.33	88.55	56.22	66.66	33.77	95.18	59.22
20	0.75	0.2	63.33	32.22	75.0	37.77	70.18	25.0	77.03	29.0
20	0.75	0.3	67.5	45.0	67.5	47.5	50.0	50.0	60.0	47.5
30	0.25	0.1	91.7	65.03	95.25	75.4	75.18	40.22	97.16	67.18
30	0.25	0.2	86.59	57.4	90.96	58.0	71.85	37.11	96.29	67.33
30	0.25	0.3	79.33	45.62	79.48	57.25	71.35	35.18	90.74	53.03
30	0.25	0.4	70.66	40.66	63.55	42.29	66.66	32.29	71.97	34.88
30	0.5	0.1	83.7	54.37	87.85	55.48	62.83	37.18	95.92	63.33
30	0.5	0.2	58.88	28.0	65.62	32.0	65.55	22.66	77.77	25.03
30	0.75	0.1	74.37	33.55	81.48	34.74	51.72	25.7	91.6	39.62
40	0.25	0.1	89.44	65.11	94.33	70.88	68.33	47.44	96.66	80.44
40	0.25	0.2	75.5	42.61	80.99	43.38	59.25	28.55	91.94	50.77
40	0.25	0.3	63.88	30.72	63.38	35.38	60.37	27.16	74.53	28.33
40	0.5	0.1	80.61	42.38	83.05	45.05	48.33	29.83	91.94	52.05
40	0.5	0.2	37.08	34.84	38.06	34.0	44.17	32.93	36.96	34.93
40	0.75	0.1	62.72	27.27	65.83	29.61	64.9	24.0	73.98	27.38

Table 4.8. Solution Stability. $\ell = 3$, $ch = 0.1$.

n	con	t	BBFC	BBFC*	BBFMAC	BBFMAC*	FLCFCft	FLCFCft*	FLCFMACft	FLCFMACft*
20	0.25	0.1	43.88	31.11	43.55	32.55	51.29	32.11	47.96	33.22
20	0.25	0.2	77.44	51.44	81.33	58.55	66.11	35.22	84.07	46.11
20	0.25	0.3	77.44	52.44	82.22	55.11	62.59	34.0	91.48	64.22
20	0.25	0.4	65.44	45.22	70.44	43.66	57.4	27.44	81.66	45.22
20	0.25	0.5	48.77	33.0	62.44	40.44	59.62	31.33	72.77	35.66
20	0.25	0.6	41.75	32.83	36.83	25.91	42.77	31.75	48.05	29.16
20	0.5	0.1	78.66	51.0	87.33	64.0	57.59	32.66	92.4	70.33
20	0.5	0.2	62.33	39.55	69.99	42.99	47.22	26.77	84.81	46.22
20	0.5	0.3	50.44	25.44	53.33	35.33	54.44	22.33	61.29	28.22
20	0.5	0.4	35.0	33.75	35.0	33.74	37.5	35.0	37.5	32.5
20	0.75	0.1	76.55	48.55	80.11	52.66	42.77	28.11	91.48	60.33
20	0.75	0.2	53.44	25.88	55.66	31.77	51.48	23.22	62.4	25.88
20	0.75	0.3	19.16	19.16	20.83	17.5	27.5	21.66	22.5	20.0
30	0.25	0.1	86.22	66.74	91.7	73.48	57.4	40.29	92.22	65.7
30	0.25	0.2	73.62	45.99	81.85	49.77	51.85	32.59	89.01	56.0
30	0.25	0.3	59.4	38.37	68.88	36.29	54.19	31.03	77.16	42.51
30	0.25	0.4	41.7	29.11	46.29	27.48	48.51	21.48	47.28	29.62
30	0.25	0.5	26.66	20.0	26.66	20.0	20.0	23.33	16.66	16.66
30	0.5	0.1	74.59	47.11	79.11	54.37	45.67	35.11	90.0	57.85
30	0.5	0.2	46.96	23.62	49.62	25.55	50.24	21.7	56.54	26.22
30	0.5	0.3	20.0	20.0	20.0	20.0	23.33	20.0	26.66	23.33
30	0.75	0.1	62.22	35.25	66.74	38.81	43.95	24.59	78.27	40.88
40	0.25	0.1	82.5	57.33	85.61	62.83	45.37	39.38	93.24	66.61
40	0.25	0.2	62.11	35.88	65.0	36.11	50.37	28.16	83.51	44.66
40	0.25	0.3	47.5	31.05	53.94	27.77	49.81	25.38	59.53	24.27
40	0.5	0.1	66.88	37.0	70.22	40.94	40.55	31.0	83.98	49.16
40	0.5	0.2	31.93	30.63	31.31	26.63	29.58	28.0	31.24	25.68
40	0.75	0.1	45.99	25.38	53.0	24.66	52.68	22.88	59.81	23.83

Table 4.9. Solution Stability. $\ell = 3$, $ch = 0.25$.

n	con	t	BBFC	BBFC*	BBFMAC	BBFMAC*	FLCFCft	FLCFCft*	FLCFMACft	FLCFMACft*
20	0.25	0.1	75.66	52.77	82.33	52.44	76.85	37.44	80.74	46.11
20	0.25	0.2	77.44	55.66	86.66	58.88	70.0	38.55	92.96	53.33
20	0.25	0.3	74.77	51.44	84.0	57.77	65.74	46.88	89.25	52.22
20	0.25	0.4	71.55	51.66	80.0	60.44	61.11	41.66	84.25	46.11
20	0.25	0.5	65.77	43.55	71.22	50.33	69.07	40.77	80.55	48.44
20	0.25	0.6	76.44	56.27	79.11	63.5	74.9	45.61	86.48	53.66
20	0.25	0.7	88.33	88.33	90.0	90.0	93.33	83.33	90.0	90.0
20	0.5	0.1	67.88	43.11	80.44	51.66	47.59	28.11	83.14	39.22
20	0.5	0.2	63.11	41.33	65.88	38.11	56.11	35.33	61.29	35.0
20	0.5	0.3	69.0	46.77	79.88	46.66	65.55	33.88	87.03	43.77
20	0.5	0.4	55.88	42.88	62.11	40.77	64.25	30.33	65.55	37.0
20	0.5	0.5	42.62	42.62	56.0	45.37	72.5	39.12	90.0	36.62
20	0.75	0.1	60.0	41.44	60.44	42.88	60.55	33.55	60.92	32.0
20	0.75	0.2	65.33	34.33	74.66	40.0	56.11	26.66	73.7	32.55
20	0.75	0.3	48.66	29.44	63.66	36.77	66.29	26.66	74.62	26.55
30	0.25	0.1	69.11	46.07	86.37	48.74	59.75	36.29	85.55	45.7
30	0.25	0.2	54.81	32.96	76.88	47.7	50.61	32.81	78.02	33.33
30	0.25	0.3	48.51	35.03	60.14	37.85	50.98	33.85	59.87	34.96
30	0.25	0.4	64.96	43.18	77.77	50.51	57.16	31.92	85.55	34.88
30	0.25	0.5	70.24	52.24	76.42	52.69	67.35	44.6	83.56	47.06
30	0.5	0.1	52.22	34.51	59.62	37.77	52.22	34.66	56.41	36.07
30	0.5	0.2	64.96	41.62	80.81	42.44	51.23	27.62	73.95	34.44
30	0.5	0.3	56.51	32.07	57.4	38.14	65.18	28.88	63.7	24.96
30	0.75	0.1	68.81	38.88	78.0	44.66	51.48	26.74	63.82	28.0
30	0.75	0.2	39.55	29.92	59.33	28.59	63.95	24.37	62.71	27.85
40	0.25	0.1	62.33	36.77	73.22	44.66	48.24	27.94	76.94	31.72
40	0.25	0.2	49.83	36.38	55.88	36.0	49.72	36.66	51.66	35.94
40	0.25	0.3	67.22	41.0	74.16	43.38	55.27	31.0	71.85	38.27
40	0.25	0.4	51.33	37.16	65.44	37.83	55.74	27.38	57.59	32.0
40	0.5	0.1	78.38	46.77	84.88	56.05	52.49	30.38	50.92	30.83
40	0.5	0.2	47.05	24.61	62.77	29.77	55.92	26.38	65.27	23.72
40	0.75	0.1	65.0	32.5	78.05	35.16	46.48	24.77	66.01	25.55

Table 4.10. Solution Stability. $\ell = 5$, $ch = 0.1$.

4.9 The Utility of the Deletion Threshold

Section 3.6.4 described the deletion threshold, δ , which is used throughout the experiments in this chapter to enable consistency-enforcing hybrids to remove domain elements from consideration as early as possible. This section attempts to measure the utility of this technique by comparing the performance of branch and bound and FLC hybrids with and without δ .

The particular algorithms chosen for this part of the study are again BBFC, BBFMAC, FLCFCft and FLCFMACft to allow pre-processing and non-pre-processing versions of both algorithms to be compared. The suffix ‘noDel’ indicates those algorithms whose deletion threshold has been set to $l_{\perp}$. Since the deletion threshold primarily affects the number of constraint checks and search nodes, results in terms of these measurements are reported in this section. Full results are available in Appendix C.

Figure 4.21 presents the case: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$. Most benefit is gained in terms of constraint checks by the algorithms maintaining fuzzy arc consistency. This is as expected, since those algorithms perform relatively more constraint propagation. In terms of search nodes, however, the forward checking algorithms are clearly improved by the use of this technique. As noted in Section 3.6.4, this is because filtering domain elements as early as possible can improve the performance of the variable ordering heuristic used.

If con is increased to 0.75, maintaining the other parameter settings (see Figure 4.22) the effect of using the deletion threshold can be seen to be extremely limited. The higher connectivity naturally leads to more constraint propagation. Hence, the likelihood is increased that the unary consistency degree of each domain element is revised to $l_{\perp}$, rendering the deletion threshold technique redundant.

A greater number of variables is considered in Figure 4.23, with parameter settings: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$. Despite the lower connectivity, the deletion threshold technique also has a very limited effect in this case. Figure 4.24 presents results for: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$. The improvement gained by using the deletion threshold at lower numbers of variables and connectivities is maintained. This is as expected: there is no reason why a greater degree of change should significantly affect the amount of propagation within individual problem instances.

Finally, a setting of $\ell = 5$ is presented in Figure 4.25 in combination with $ch = 0.1$, $n = 20$ and $con = 0.25$. These results confirm the expectation that the deletion threshold should be more worthwhile with greater numbers of consistency degrees (since an ever greater number of revisions are otherwise required to revise the unary consistency degree of a domain element to $l_{\perp}$). A positive effect is visible for all four algorithms in terms of constraint checks and a substantial reduction is made in the number of search nodes explored by the two forward checking algorithms.

Fig. 4.21. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

Fig. 4.22. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$.

Fig. 4.23. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$.

Fig. 4.24. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$.

Fig. 4.25. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 20$, $con = 0.25$.

Fig. 4.26. Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

4.10 The Utility of the Constraint Check Ordering Heuristic

This section attempts to measure the utility of the constraint check selection heuristic described in Section 4.4.3. Hybrid variants of branch and bound and FLC with naive constraint check selection procedures are compared with those with the heuristic. Since the forward checking algorithms perform a single revision step per domain element with no further propagation, benefit will only be gained by these algorithms in terms of detecting a dead end in the search earlier. It is therefore expected that the effects of this heuristic will be seen most clearly in the comparison of the FMAC-based algorithms.

Unsurprisingly, this heuristic primarily affects the number of constraint checks taken by each algorithm. Hence, although full results are reported in Appendix C, only graphs of constraint check results are presented in this chapter. The same combination of pre-processing and non-pre-processing algorithms is tested as in the previous two sections. The suffix 'noFACFC' denotes an algorithm whose constraint check ordering capability has been disabled, for either maintaining fuzzy arc consistency or forward checking as appropriate.

Figure 4.26 presents the case where $\ell = 3$, $ch = 1$ constraint, $n = 20$ and $con = 0.25$. In this case, the algorithms with the heuristic are never worse than those without it. As expected, the effect of the heuristic is most pronounced

Fig. 4.27. Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$.

on the algorithms maintaining fuzzy arc consistency. A minimal effect is seen on the forward checking algorithms, but these are already performing strongly in terms of constraint checks.

The same trend can be observed in Figure 4.27, which presents the case where $\ell = 3$, $ch = 1$ constraint, $n = 20$, and $con = 0.75$. Given the high connectivity of these problems (and hence the higher number of constraints), most benefit seems to be gained by employing the heuristic at the phase transition at around $t = 0.3$. Furthermore, FLCFMACft appears to gain more by using the heuristic at these parameter settings than any other algorithm, but its performance was already relatively weak in terms of constraint checks.

At either side of the phase transition, however, the effect of the heuristic is minimal on all the algorithms tested. This is perhaps because, in the solvable cases, little propagation is possible, so constraint selection order is immaterial. In the unsolvable cases, with such a high number of constraints present, there may be many alternative methods to find inconsistency, also rendering constraint selection order unimportant.

Figure 4.28 presents the case where $\ell = 3$, $ch = 1$ constraint, $n = 40$ and $con = 0.25$. Similarly, the heuristic gives most benefit at the phase transition, and FLCFMACft gains more than BBFMAC. Given the weaker performance of FLCFCft at the phase transition for this higher number of variables, it is noticeable that it too derives more benefit from the heuristic at around $t = 0.4$.

Fig. 4.28. Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$.

Settings of $\ell = 4$ and $\ell = 5$ are considered in Figures 4.29 and 4.30 respectively, with the other parameters set to: $ch = 1$ constraint, $n = 30$, and $con = 0.25$. Given a higher number of consistency degrees, the effect of the heuristic is more widespread. The benefit gained by FLCFCft is also more obvious, essentially because these problems are more difficult to solve and the search therefore contains a larger number of dead ends. The improvement is still significantly smaller than that gained by the algorithms maintaining fuzzy arc consistency. The benefit gained by BBFC remains small but this again is likely to be because of its strong performance on these sets of problems in terms of constraint checks.

It does not appear that the effect of the heuristic simply scales according to the number of consistency degrees: the amount of benefit gained is roughly comparable in both cases. Part of the reason for this is likely to be that this ordering heuristic does not allow any extra constraint propagation, but simply increases propagation efficiency. Hence, a bound must exist on the benefit gained. As ℓ increases, given the increased effort required to enforce fuzzy arc consistency, more benefit from this heuristic might be expected, however. A more detailed study of the effects of this heuristic over a wider range of consistency degrees is necessary to determine if this is the case.

Fig. 4.29. Mean Constraint Checks: $\ell = 4$, $ch = 1$ constraint, $n = 30$, $con = 0.25$.

Fig. 4.30. Mean Constraint Checks: $\ell = 5$, $ch = 1$ constraint, $n = 30$, $con = 0.25$.

4.11 The Utility of FLC Variable Selection Heuristics

A number of novel variable selection heuristics, enabled by the particular search structure of the FLC algorithm, were described in Section 4.4.1. This section compares these heuristics to discover which is most effective in general and also whether certain heuristics are more effective in some situations than others. The numbering scheme adopted in the figures reflects that given in Section 4.4.1. The **ConsDeg-PrevSoln-con** domain element selection heuristic (see Section 4.4.2) is used throughout for ease of comparison. Testing all sixteen combinations of variable and domain element ordering heuristics, as well as developing new heuristics, should be the subject of a future study.

Figure 4.31 presents the case where $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$. Clearly, there is almost nothing to choose between the heuristics in terms of either constraint checks or search nodes. A connectivity of 0.75, maintaining the settings of the other parameters is considered in Figure 4.32. There is a clear difference between the algorithms at around $t = 0.3$, but surprisingly, it is the simple **SmallestD** heuristic that outperforms the others. In this case, taking into account the connectivity of the current variable with respect to the ‘non-fixed’ variables seems to actively damage performance.

Table 4.11 presents the stability results for $\ell = 3$, $ch = 1$ constraint. By this measure, the **SmallestD- X_{23}** heuristic has a slight edge over the others, but the difference is not great. When a larger number of variables at a lower connectivity are considered, however, the **SmallestD** heuristic is clearly worse by both measures (see Figure 4.33). As the number of variables increases, the more sophisticated heuristics appear to be more successful at targetting the sections of a loosely constrained graph to repair.

Figure 4.34 gives results for the case: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$. As compared with Figure 4.31, there remains very little to choose between the heuristics even given this higher degree of change between problem instances. Table 4.12 presents stability results for $\ell = 3$, $ch = 0.25$. Again, the **SmallestD- X_{23}** heuristic emerges as being slightly better than the others, as was the case for lower values of ch .

Finally, Figure 4.35 considers the effects of raising the number of consistency degrees. Again, however, the performance of all four heuristics is very evenly matched. Stability results for $\ell = 5$, $ch = 0.1$ may be examined in Table C.51. It is noticeable that the **SmallestD- X_{23}** heuristic outperforms the others by this measure for a greater proportion of the cases than it does for lower values of ℓ . Overall, it appears that examining the full set of ‘non-fixed’ variables, as opposed to simply the elements of X_2 or X_3 , does pay dividends. This seems to be the case particularly in terms of maintaining solution stability.

Fig. 4.31. Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

Fig. 4.32. Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$.

n	con	t	FLCFMACft1	FLCFMACft2	FLCFMACft3	FLCFMACft4
20	0.25	0.1	91.0	91.77	91.55	96.48
20	0.25	0.2	96.55	96.88	96.44	99.07
20	0.25	0.3	98.99	99.33	98.77	99.62
20	0.25	0.4	96.66	97.77	96.44	97.96
20	0.25	0.5	93.88	91.66	95.33	96.85
20	0.25	0.6	75.0	75.0	75.0	100.0
20	0.5	0.1	99.55	99.44	99.33	99.62
20	0.5	0.2	98.33	98.88	98.66	99.07
20	0.5	0.3	95.11	97.55	96.99	99.62
20	0.5	0.4	65.0	65.0	65.0	65.0
20	0.75	0.1	99.66	99.66	99.44	99.62
20	0.75	0.2	92.66	94.55	96.22	99.07
20	0.75	0.3	100.0	100.0	100.0	100.0
30	0.25	0.1	99.4	99.25	99.4	100.0
30	0.25	0.2	99.25	99.4	99.48	99.5
30	0.25	0.3	98.66	99.4	98.59	99.01
30	0.25	0.4	92.22	91.25	92.07	97.53
30	0.5	0.1	99.77	99.77	99.77	100.0
30	0.5	0.2	97.4	94.66	97.85	99.62
30	0.75	0.1	99.48	99.7	99.85	99.75
30	0.75	0.2	100.0	100.0	100.0	100.0
40	0.25	0.1	99.72	99.72	99.66	99.9
40	0.25	0.2	99.44	99.77	99.77	99.9
40	0.25	0.3	96.72	93.77	98.94	98.61
40	0.5	0.1	99.77	99.72	99.16	99.9
40	0.5	0.2	96.2	97.56	97.63	99.68
40	0.75	0.1	99.61	99.38	97.83	100.0

Table 4.11. Solution Stability. $\ell = 3$, $ch = 1$ constraint.

Fig. 4.33. Mean Constraint Checks: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$.

Fig. 4.34. Mean Constraint Checks: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$.

n	con	t	FLCFMACft1	FLCFMACft2	FLCFMACft3	FLCFMACft4
20	0.25	0.1	47.33	46.0	51.55	47.96
20	0.25	0.2	79.88	79.55	81.0	84.07
20	0.25	0.3	89.33	90.22	88.66	91.48
20	0.25	0.4	80.77	80.66	83.33	81.66
20	0.25	0.5	70.11	68.0	69.88	72.77
20	0.25	0.6	43.91	46.33	40.33	48.05
20	0.5	0.1	92.99	91.44	91.22	92.4
20	0.5	0.2	83.22	82.88	85.88	84.81
20	0.5	0.3	61.88	57.33	65.99	61.29
20	0.5	0.4	33.75	35.0	35.0	37.5
20	0.75	0.1	89.66	87.88	90.0	91.48
20	0.75	0.2	62.22	57.55	58.77	62.4
20	0.75	0.3	22.5	22.5	25.83	22.5
30	0.25	0.1	88.07	89.62	87.25	92.22
30	0.25	0.2	88.37	88.14	87.11	89.01
30	0.25	0.3	80.96	78.66	77.25	77.16
30	0.25	0.4	49.62	48.0	48.96	47.28
30	0.25	0.5	20.0	16.66	20.0	16.66
30	0.5	0.1	89.25	86.66	90.07	90.0
30	0.5	0.2	58.74	54.07	57.03	56.54
30	0.5	0.3	23.33	20.0	23.33	26.66
30	0.75	0.1	77.85	82.07	80.37	78.27
40	0.25	0.1	91.33	92.33	91.27	93.24
40	0.25	0.2	83.44	80.55	82.38	83.51
40	0.25	0.3	51.38	51.22	52.72	59.53
40	0.5	0.1	84.61	83.77	82.11	83.98
40	0.5	0.2	29.21	27.26	29.71	31.24
40	0.75	0.1	57.94	56.11	59.66	59.81

Table 4.12. Solution Stability. $\ell = 3$, $ch = 0.25$.

Fig. 4.35. Mean Constraint Checks: $\ell = 5$, $ch = 0.1$, $n = 30$, $con = 0.25$.

4.12 The Utility of FLC Domain Element Selection Heuristics

The search structure of the FLC algorithm also allows the development of novel domain element selection heuristics, as described in Section 4.4.2. This section performs a comparison of these heuristics with respect to a particular variable selection heuristic in order to discover if one emerges as the clear winner. The variable ordering heuristic used throughout is **SmallestD- X_{23}** , as described in Section 4.4.1. Although a comparison based on the other variable ordering heuristics is equally valid (and should be the subject of a future study), this heuristic was chosen as being clearly better than the others in at least one criterion: that of solution stability, as described in the previous section. The numbering scheme adopted in the figures reflects that given in Section 4.4.2.

Figure 4.36 presents the case where $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$. Mirroring the results gained when comparing variable ordering heuristics, the performance of all four heuristics is very similar. If the connectivity is increased to 0.75, however, maintaining the settings of the other parameters (see Figure 4.32), the naive **ConsDeg** heuristic emerges as clearly inferior to the others. This is further evidence to suggest that re-using information from the previous solution in the dynamic sequence can have a positive effect on the amount of search effort required.

Figure 4.38 considers $n = 40$, with $\ell = 3$, $ch = 1$ constraint and $con = 0.25$. Again, the naive heuristic is clearly worse, although there is very little to choose between the other heuristics. Table 4.13 gives the stability results for $\ell = 3$, $ch = 1$ constraint. By this measure, the **ConsDeg-PrevSoln-con** heuristic has a slight edge over the others, being the best in almost all cases. In addition, the naive **ConsDeg** heuristic is almost always the worst.

Figure 4.39 presents results for the case: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$. As compared with Figure 4.36, there remains very little to choose between the heuristics even given this higher degree of change between problem instances. Table 4.14, presents stability results for $\ell = 3$, $ch = 0.25$. Again, the **ConsDeg-PrevSoln-con** heuristic emerges as being consistently better than the others, and the **ConsDeg** heuristic consistently performs the worst.

Finally, Figure 4.40 considers the effects of raising the number of consistency degrees. In this case, the more sophisticated heuristics **ConsDeg-con** and **ConsDeg-PrevSoln-con** hold a clear advantage, suggesting that they cope better with the increased complexity of a higher setting for ℓ . Stability results for $\ell = 5$, $ch = 0.1$ are given in Table 4.15. The **ConsDeg-PrevSoln-con** and **ConsDeg** heuristics retain their positions as the best and worst by this measure respectively.

A clear trend evident from these results is that it is possible to do better than the naive **ConsDeg** heuristic, especially on more difficult problems. In the case of the other three heuristics studied herein, this is achieved by trying

Fig. 4.36. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.25$.

to limit the repairs needed to resolve an inconsistency. In terms of search effort, there is little to choose between these three. The most sophisticated heuristic tested, *ConsDeg-PrevSoln-con*, does however appear to produce

Fig. 4.37. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 20$, $con = 0.75$.

slightly better stability results by also considering the solution to the previous problem when choosing an assignment with respect to which repairs are made.

Fig. 4.38. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 1$ constraint, $n = 40$, $con = 0.25$.

n	con	t	FLCFMACft1	FLCFMACft2	FLCFMACft3	FLCFMACft4
20	0.25	0.1	90.88	93.66	91.44	96.48
20	0.25	0.2	96.55	96.55	96.55	99.07
20	0.25	0.3	98.66	98.77	98.77	99.62
20	0.25	0.4	95.66	94.55	96.11	97.96
20	0.25	0.5	88.66	93.77	92.88	96.85
20	0.25	0.6	77.5	77.5	75.0	100.0
20	0.5	0.1	99.33	99.33	99.33	99.62
20	0.5	0.2	98.11	98.44	98.66	99.07
20	0.5	0.3	95.66	95.99	98.0	99.62
20	0.5	0.4	65.0	65.0	65.0	65.0
20	0.75	0.1	99.55	99.55	99.55	99.62
20	0.75	0.2	96.44	97.22	95.55	99.07
20	0.75	0.3	100.0	100.0	100.0	100.0
30	0.25	0.1	99.48	99.48	99.48	100.0
30	0.25	0.2	98.51	99.33	99.4	99.5
30	0.25	0.3	96.22	99.33	98.37	99.01
30	0.25	0.4	89.18	86.22	96.14	97.53
30	0.5	0.1	99.85	99.85	99.85	100.0
30	0.5	0.2	93.62	96.0	98.81	99.62
30	0.75	0.1	99.48	99.18	99.62	99.75
30	0.75	0.2	100.0	100.0	100.0	100.0
40	0.25	0.1	99.83	99.83	99.83	99.9
40	0.25	0.2	98.88	99.55	99.33	99.9
40	0.25	0.3	93.38	98.77	95.27	98.61
40	0.5	0.1	99.94	99.5	99.44	99.9
40	0.5	0.2	94.67	97.89	94.34	99.68
40	0.75	0.1	94.5	99.88	99.27	100.0

Table 4.13. Solution Stability. $\ell = 3$, $ch = 1$ constraint.

Fig. 4.39. Mean Constraint Checks/Search Nodes: $\ell = 3$, $ch = 0.25$, $n = 20$, $con = 0.25$.

n	con	t	FLCFMACft1	FLCFMACft2	FLCFMACft3	FLCFMACft4
20	0.25	0.1	40.0	44.55	45.44	47.96
20	0.25	0.2	80.0	79.77	80.0	84.07
20	0.25	0.3	84.44	85.11	88.33	91.48
20	0.25	0.4	71.77	79.66	78.88	81.66
20	0.25	0.5	59.44	62.88	71.66	72.77
20	0.25	0.6	36.91	45.99	44.0	48.05
20	0.5	0.1	90.66	90.55	90.33	92.4
20	0.5	0.2	73.66	81.88	83.66	84.81
20	0.5	0.3	37.0	57.55	57.33	61.29
20	0.5	0.4	35.0	35.0	32.5	37.5
20	0.75	0.1	87.66	88.77	89.22	91.48
20	0.75	0.2	36.77	58.88	53.66	62.4
20	0.75	0.3	22.5	20.83	20.83	22.5
30	0.25	0.1	86.59	86.22	87.62	92.22
30	0.25	0.2	84.74	89.55	85.48	89.01
30	0.25	0.3	70.81	76.66	76.96	77.16
30	0.25	0.4	29.77	40.44	44.59	47.28
30	0.25	0.5	20.0	20.0	20.0	16.66
30	0.5	0.1	85.33	88.29	87.55	90.0
30	0.5	0.2	29.85	48.96	50.29	56.54
30	0.5	0.3	26.66	26.66	20.0	26.66
30	0.75	0.1	69.25	77.77	79.11	78.27
40	0.25	0.1	90.83	92.27	92.11	93.24
40	0.25	0.2	70.5	80.16	81.33	83.51
40	0.25	0.3	29.38	49.0	52.38	59.53
40	0.5	0.1	74.11	80.38	82.33	83.98
40	0.5	0.2	26.8	29.46	29.05	31.24
40	0.75	0.1	26.38	54.77	59.27	59.81

Table 4.14. Solution Stability. $\ell = 3$, $ch = 0.25$.

Fig. 4.40. Mean Constraint Checks/Search Nodes: $\ell = 5$, $ch = 0.1$, $n = 30$, $con = 0.25$.

n	con	t	FLCFMACft1	FLCFMACft2	FLCFMACft3	FLCFMACft4
20	0.25	0.1	75.22	80.33	79.0	80.74
20	0.25	0.2	85.77	86.77	86.66	92.96
20	0.25	0.3	80.22	81.44	82.0	89.25
20	0.25	0.4	75.55	77.22	79.0	84.25
20	0.25	0.5	71.88	74.22	75.33	80.55
20	0.25	0.6	74.94	84.0	82.33	86.48
20	0.25	0.7	90.0	90.0	90.0	90.0
20	0.5	0.1	64.22	75.22	75.22	83.14
20	0.5	0.2	61.11	62.88	55.88	61.29
20	0.5	0.3	68.88	76.11	78.88	87.03
20	0.5	0.4	53.66	66.88	60.22	65.55
20	0.5	0.5	50.0	53.62	54.25	90.0
20	0.75	0.1	53.77	52.11	54.88	60.92
20	0.75	0.2	63.11	65.11	60.33	73.7
20	0.75	0.3	37.77	54.44	55.22	74.62
30	0.25	0.1	79.48	82.37	79.03	85.55
30	0.25	0.2	60.51	70.81	71.18	78.02
30	0.25	0.3	46.74	53.92	54.88	59.87
30	0.25	0.4	64.59	73.62	75.62	85.55
30	0.25	0.5	60.17	75.11	79.58	83.56
30	0.5	0.1	42.51	44.44	48.66	56.41
30	0.5	0.2	54.44	57.77	57.25	73.95
30	0.5	0.3	34.0	39.33	50.96	63.7
30	0.75	0.1	50.59	51.7	46.74	63.82
30	0.75	0.2	27.55	44.22	37.62	62.71
40	0.25	0.1	54.27	70.94	70.11	76.94
40	0.25	0.2	38.5	37.22	40.33	51.66
40	0.25	0.3	58.22	61.33	66.11	71.85
40	0.25	0.4	35.61	46.88	50.83	57.59
40	0.5	0.1	36.16	42.27	48.33	50.92
40	0.5	0.2	27.05	41.49	43.5	65.27
40	0.75	0.1	45.0	52.27	50.44	66.01

Table 4.15. Solution Stability. $\ell = 5$, $ch = 0.1$.

4.13 Summary

This chapter and its companion, Appendix C, have presented an extensive initial study of fuzzy *rrDFCSPs*, and measured the performance of variants of the branch and bound and FLC algorithms developed in the previous chapter by three criteria: constraint checks, search nodes and stability of solutions produced. The effects of varying the number of consistency degrees, the number of variables, the connectivity of the constraint graph and the tightness of the constraints in each problem instance and the size of the changes between instances were all investigated. The difficulty of finding a solution for a fuzzy *rrDFCSP* as these parameters are varied was of special interest, since this type of problem has not been investigated before. The phase transitions widely reported for Boolean problems were clearly present. Indeed, multiple phase transitions were recorded, dependent on the number of consistency degrees in the problem.

One of the most important conclusions that can be drawn from these results is that the use of dynamic solution techniques is justified: the effort required to maintain and re-use information from the previous problem in a dynamic sequence is compensated for by an increase in the quality of results as measured by all three criteria listed above. In addition, it was found that the deletion threshold technique described in Section 3.6.4 was also worthwhile, especially on more loosely connected problems or as the number of consistency degrees increases.

Due to their increased use of constraint propagation, a heuristic ordering on the selection of constraints during constraint propagation was seen to be most effective for those algorithms maintaining fuzzy arc consistency. Benefit is also gained by forward checking algorithms on more difficult problems. The use of variable ordering heuristics with the FLC variant algorithms was less successful. A small increase in stability was seen using a more sophisticated heuristic, but in terms of constraint checks and search nodes, there was no clear winner amongst the alternatives tested. Domain element selection heuristics for the FLC variants were, however, more successful. A clear improvement is seen over a naive selection order, although there is little to choose between the more informed heuristics tested.

It is difficult to choose a clear ‘winner’; a best algorithm to apply to any given fuzzy *rrDFCSP*. Although the forward checking algorithms consistently gave better results in terms of constraint checks, they were often poorer when the search nodes results were considered. Conversely, the algorithms that maintain fuzzy arc consistency consistently explored fewer search nodes, but made many more constraint checks. One general rule is that the branch and bound heuristics are more efficient, as predicted, on the harder problems, since their more rigid search structure allows more effective constraint propagation. Another is that the FLC variants produce more stable solutions since they actively work to maintain the assignments in the solution

to the previous problem in an instance rather than simply re-assigning the same domain element as a tie-breaker.

Following this initial study, a more focused examination of the location and difficulty of the phase transition peaks in solution difficulty would be useful. It should also help to determine which peak, if any, is the hardest among those that occur for a given number of consistency degrees. In addition, the effects of different distributions of the types of constraints (e.g. prioritised and prioritised-preference) should be measured, as should the effects of varying domain sizes and the utility of different combinations of the variable and domain element selection heuristics defined herein.

5 Dynamic CSP in Domain-independent AI Planning

This chapter introduces domain-independent AI Planning as an application domain for both dynamic and dynamic flexible CSP solution techniques, the latter development being discussed in Chapters 7 and 8. Recent advances in AI planning have established that the reduction of planning to a constraint satisfaction problem enables a gain in efficiency via the use of the dedicated search algorithms available in this field [82]. This chapter continues in this vein, presenting a novel technique which exploits (restriction/relaxation-based) dynamic CSP in order to further improve planner performance.

Using the standard Graphplan [14] framework as a foundation, it is shown how further significant efficiency gains may be obtained by viewing plan extraction as the solution of a hierarchy of *rrDCSPs* to create a new Boolean planner, GP-*rrDCSP*. Within this hierarchical framework, issues of efficient conflict recording (see Section 2.6) and heuristic choices (Section 2.5) to maximise search efficiency are identified and discussed in detail, before being tested empirically in the following chapter.

5.1 AI Planning

A plan represents a course of action designed to achieve a set of pre-specified goals. The act of *planning*, i.e. the construction of a plan prior to its execution, is a very common everyday activity. For instance, the simple task of making a pot of tea requires the identification of a correctly ordered sequence of actions if it is to be successfully completed. It is no good, for example, trying to add boiling water before the teapot has been located.

Artificial Intelligence planning [2] is an active and long-established research area. It has a wide applicability to such tasks as automating data-processing procedures [23], game-playing [152], and large-scale logistics problems [177]. Solving such problems demands a highly efficient approach due to their size and complexity.

An abstract view of a planning agent is presented in Figure 5.1. The agent wants to manipulate the environment in order to achieve its goals. It is able to perceive the current state of the environment and to modify the environment through the execution of the actions available to it. A plan,

Fig. 5.1. Components of A Planning Agent

therefore, constitutes a sequence of actions which, upon execution, changes the environment to achieve the specified goals.

Many techniques exist for automating the solution of a wide variety of planning problems [175]. The types of planning problem are differentiated by the expressivity of the language in which the problem is described. Unsurprisingly, as this expressivity increases, the effort required for synthesising plans to satisfy the associated problem also sharply increases because of the commensurate increase in the space of possibilities that the planning algorithm must explore.

Classical AI planning, upon which this chapter concentrates, makes the following simplifying assumptions [174]:

- Atomic time: Plan execution is modelled as an atomic transformation from one world state to another. Hence, the planner does not need to consider the state of the environment during the execution of an action.
- Deterministic effects: The effect of executing any action is a deterministic function of the action and of the state of the environment when the action is executed.
- Omniscience: The agent has complete knowledge of the initial state of the environment and the nature of its own actions.
- Sole cause of change: The environment is static and changes only by the agent's own actions.

Early planners cast the planning problem as search through the space of world states. A plan from this perspective is therefore a path through the state space. Progressive and regressive planning algorithms [122, 174] search from the initial conditions to the goal and vice versa respectively. The difficulty with this technique is that it constrains the planning algorithm to

search adjacent states only, whereas it is often useful to add new actions to the plan at arbitrary points.

An alternative view casts plan synthesis as search through a space of plans [139, 159]. Nodes in this space represent partially specified plans, with arcs between nodes representing plan refinement operations such as the addition of an action. This is a more powerful approach, allowing the equivalent of making large jumps around the state space.

5.1.1 Constraint Satisfaction in Planning

Reducing the classical AI planning problem to the solution of a constraint satisfaction problem allows the use of the efficient solution techniques developed in that field [82]. Graphplan [14] is a typical example that has proven to be successful by employing CSP methods to extract a valid plan. The process of plan extraction in Graphplan is usually formulated as the solution of an *activity*-based dynamic constraint satisfaction problem [82, 115] (see Section 2.10.3). Activity constraints specify that new parts of a problem must be considered (or that existing parts can now be ignored), based on the choices made when searching for a solution. This type of constraint naturally corresponds to the relationship between a plan action and its preconditions, i.e. if an action is to be considered as part of the plan, search must be performed to support its preconditions also.

An activity-based formulation of plan extraction allows the problem to be considered in one large piece. This can be unwieldy because of its size, and inefficient because a large portion of this problem may be irrelevant to the final solution. An alternative formulation of dynamic constraint satisfaction, based on restriction (addition of constraints) and relaxation (removal of constraints) [32] (see Section 2.10.1) is considered herein as a means of significantly improving the efficiency of the plan extraction process by tightly focusing the search and re-using as much search effort as possible. The application of this technique is made possible through a novel hierarchical structuring of the search process such that each level of the hierarchy constitutes a restriction/relaxation-based dynamic CSP.

5.2 An Overview of Graphplan

The Graphplan [14] framework is a natural choice of synthesis technique upon which to base both the improved Boolean and flexible planning systems. It possesses some important properties, including: optimality in the number of actions and length of a synthesised plan; soundness and completeness; and polynomial time and space complexity of the creation of the planning graph structure (see Section 5.2.1) that is central to the CSP methods employed in this book. In addition, Graphplan is a planning-specific approach, with the planning graph providing some structure that may lead to more efficient

Fig. 5.2. An Example Problem.

solution methods than those based solely on a more basic representation. See [4] for recent work in this regard.

Graphplan continues the long tradition in planning research of concentrating on a subset of issues known as the STRIPS domain [50]. Maintaining this tradition allows the development and evaluation of a flexible capability in a conceptually clean framework. This domain type consists of plan goals, represented as a propositional conjunction, and a set of *actions* available to the planner. Each action is described by a conjunctive *precondition* and a conjunctive *effect*, defining a transition function from one universe of discourse to another. The action can be executed in any universe of discourse satisfying the preconditions.

An example problem from the STRIPS domain, presented in Figure 5.2, is used throughout this chapter first to illustrate the operation of Graphplan and then (in an extended form) the operation of FGP. In the figure, the c_i represent three cities, r_1 and r_2 represent major roads and r_3 an unsafe track through the hills. The goal of this problem is to transport both packages (pkg_1 , pkg_2) to city c_3 . Figure 5.3 presents one action from the example problem concerning the loading of a package onto a truck. This operator requires that the truck and package are at the same location and also makes the implicit assumption that the package is valuable - the presence of a guard on the truck is required. Further operators are defined for this problem to allow unloading of packages, driving the truck between cities (using main roads only) and to allow the guard to board the truck.

```
(operator LOAD-TRUCK
  (params (t truck) (p package) (l location)
    (g guard))
  (preconds (at t l) (at p l) (on g t))
  (effects (on p t) (not(at p l))))
```

Fig. 5.3. The Load-truck Operator.

5.2.1 The Planning Graph

Graphplan synthesises plans through constructing and analysing a structure called a Planning Graph in two interleaved phases: a forward phase where the planning graph is extended until the plan goals are found, and a backward phase where the planning graph is searched to extract a valid plan. A planning graph consists of two types of nodes, propositions and actions, divided into *levels*. $Level_i$ contains a set of actions ($actions_i$) and a set of propositions ($propositions_i$). $Level_0$ is special in that it contains only proposition nodes which capture the initial state of the problem.

Extending the planning graph to $level_i$ from $level_{i-1}$ consists of introducing every action whose preconditions exist (and are mutually consistent) in $propositions_{i-1}$ to form $actions_i$. The effect propositions of these actions are placed in $propositions_i$. A special action, *Noop*, records the persistence of a proposition from level to level. Figure 5.4 presents levels 0 to 2 of the planning graph constructed for the example problem.

A second component of a planning graph is the use of mutual exclusion constraints. These binary constraints record the fact that the two specified actions (or propositions) cannot occur together at one level of a valid plan. A pair of nodes in $actions_i$ are inconsistent if:

Inconsistent Effects An effect of one action in $propositions_i$ is the negation of an effect of the other action in $propositions_i$. For example, driving the truck from c_1 to c_2 asserts that the truck is at c_2 , while driving the truck from c_2 to c_3 asserts that it is *not* at c_2 , but at c_3 .

Interference An effect of one action in $propositions_i$ is the negation of a necessary precondition of the other action in $propositions_{i-1}$. For example, driving the truck from c_2 to c_1 asserts that the truck is not at c_2 , a necessary precondition for driving the truck from c_2 to c_3 .

Competing Needs The actions have mutually exclusive preconditions in $propositions_{i-1}$. For example, driving the truck from c_1 to c_2 requires that the truck is currently at c_1 , whereas the guard boarding the truck at c_2 requires that the truck is at c_2 .

A pair of nodes in $propositions_i$ are inconsistent if:

Negation One proposition is the negation of the other.

Inconsistent Support All nodes in $actions_i$ which produce one proposition are exclusive of all actions that produce the other. For example, all methods of asserting that the truck is at c_1 are exclusive of all methods of asserting that it is at c_2 .

Fig. 5.4. Planning Graph $Levels_{s_0-2}$ for the Problem Given in Figure 5.2. For clarity, only mutual exclusion relations between non-Noop actions are given in $actions_2$ and mutual exclusion relations are not given in $propositions_2$.

5.2.2 Basic Plan Extraction

Extracting a plan from a planning graph consists of identifying a subgraph connecting the goal propositions at some $level_g$ back to the initial conditions. The subgoals and action nodes at each level of this subgraph must be mutually consistent. A basic method of finding a valid plan is as follows:

- Expand the graph until all goal propositions exist in the current level ($level_g$) and are mutually consistent.
- For each set of consistent nodes in $actions_g$ which *support* (i.e. have as effects) the goal propositions, the preconditions of this set of actions in $propositions_{g-1}$ become subgoals which must also be supported, and so on. If, for example, ‘at $truck_1$ c_2 ’ (meaning that $truck_1$ is at c_2) becomes a sub-goal at $level_2$, it can be supported by either a *Noop* or *Drive-truck* action.
- For each set of subgoals at $level_i$:
 - If no consistent set of nodes in $actions_i$ can be found to support the current set of subgoals in $propositions_i$, backtrack to $level_{i+1}$ to find an alternative set of consistent nodes in $actions_{i+1}$.
 - If the search reaches $level_0$ and the subgoals match the initial conditions, a valid plan has been found.
- If no valid plan can be found, extend the graph to $level_{g+1}$ and repeat.

Using this method, an eight-step plan can be synthesised for the example problem. It involves driving to c_2 to fetch the guard and load pkg_2 before returning to load pkg_1 . The packages are then transported via the main roads to c_3 and unloaded.

5.2.3 Memoisation

The basic scheme described above has no ability to prune the search tree so that search paths which will necessarily lead to failure are abandoned as soon as possible. A mutually-consistent set of nodes in $propositions_i$ are *conjunctively unsupportable* if all combinations of nodes in $actions_i$ that assert these propositions are either directly inconsistent or their own preconditions are unsupportable. In the example problem, the goal propositions ‘at pkg_1 c_3 ’ and ‘at pkg_2 c_3 ’ first occur together at $level_7$. Although they are not directly inconsistent, it may quickly be established that these two propositions are unsupportable at this level: at $level_5$ no mutually-consistent combinations of supporting actions exist for the propositions ‘at $truck_1$ c_2 ’, ‘on pkg_1 $truck_1$ ’, and ‘on pkg_2 $truck_1$ ’, which are necessary to achieve the goals in seven steps.

Memoisation is the process of recording such unsupportable sets, known as *memosets*[106]. If a memoset (or a superset thereof) is encountered again at the level at which it was recorded, the current search branch can be pruned immediately. Memoisation in its simplest form records the entire set of nodes

currently considered at *propositions_i* once the fact that they are unsupportable has been established. However, it is likely in general that some of the propositions are irrelevant to the failure. More effective methods of memoisation can significantly improve the efficiency of plan extraction, as is discussed in Section 5.4.2.

5.3 Viewing the Planning Graph as a CSP

In order to view the extraction of a valid plan from the planning graph as a constraint satisfaction problem, it is necessary to map the elements of the planning graph onto the elements of a CSP. A node in *propositions_i* is regarded as a CSP variable, and the set of nodes in *actions_i* which assert this proposition as an effect constitute the variable's domain.

As an example, if 'at *truck₁ c₂*' in *propositions₂* (see Figure 5.4) is regarded as a CSP variable, its domain comprises the **Noop** and **Drive-truck** actions that specify it as an effect. Binary constraints come directly from the mutual exclusion relations between the action nodes in the domains of the variables. If 'on *guard₁ truck₁*' is a second variable, with the **Guard-boards-truck** action being its single domain element, a constraint is constructed over the two variables allowing the **Noop** but not **Drive-truck** to be combined with **Guard-boards-truck**, since a mutual exclusion relation disallows the latter combination.

5.4 Plan Extraction via Dynamic Constraint Satisfaction

This section describes an efficient method of plan extraction in the Graphplan framework founded on restriction/relaxation-based dynamic CSP techniques. An implementation of this planning technique, called GP-*rrDCSP* is presented in Section 5.5. A detailed discussion is presented of how repair-based solution techniques for such dynamic CSPs can reduce overall search effort and facilitate the exploitation of memoset information.

5.4.1 A Hierarchical Approach

Previously, *rrDCSP* has not been applied to the problem of plan extraction due to its lack of ability to cope with activity constraints. However, by considering not just a single *rrDCSP* but a hierarchy of such problems, each corresponding to a level in the planning graph, activity constraints can be decoupled from the solution procedure within a single level.

Consider the problem formed at *level_g*, where the plan goals exist. Depending on the structure of the mutual exclusion constraints between the action nodes which support the plan goals, there may be several solutions (in

Fig. 5.5. Plan Extraction as a Hierarchy of *rrDCSPs*

terms of mutually consistent sets of supporting actions) to this initial problem. Each such solution, through the preconditions of the associated actions, specifies a different sub-problem at $level_{g-1}$. Instead of viewing this new set of sub-problems individually, they may instead be considered as an *rrDCSP*, as presented in Figure 5.5.

Each sub-problem in a dynamic sequence can then be solved without explicit reference to activity constraints. GP-*rrDCSP* makes use of the Local Changes *rrDCSP* algorithm described in Section 3.5.1 to solve these sequences. The justification for the use of this *rrDCSP* algorithm, as opposed to, say, a heuristically guided static CSP algorithm as explored in Chapter 3 is primarily because of this algorithm's ability to maintain solution stability. This ability was demonstrated for the equivalent *rrDFCSP* algorithm, FLC, in Chapter 4. A stable solution to a sequence of sub-problems enables GP-*rrDCSP* to focus more tightly on particular conflicts that prevent the subgoal propositions under consideration from being supportable. It is also essential to the construction of propagated memosets, as discussed in Section 5.4.2 below.

Fig. 5.6. Supporting Several Phases of Plan Extraction

This hierarchical method of plan extraction has two immediate advantages:

1. Assuming that the solutions to the problem at $level_g$ have some intersection (a likely occurrence in all but the most tightly constrained cases), there will be similarities between the sub-problems at $level_{g-1}$. An *rrDCSP* algorithm can exploit these similarities by re-using effort applied in solving the previous problem in the sequence when solving the current sub-problem.
2. Effort used in levels visited in a failed attempt at plan extraction can be re-used when the graph is expanded and a further attempt at plan extraction is made. Figure 5.6 presents this process: each sub-problem is labelled with the number of the extraction phase that explored it.

5.4.2 Memoisation in the Hierarchical Context

In order to increase efficiency in searching for a valid plan, it is necessary to prune branches of the search space which will necessarily lead to failure. The simple memoisation strategy described in Section 5.2.3 is of limited usefulness because the size of the recorded memoset is likely to be large. Such large sets are unlikely to be encountered many times in the search process and hence will not contribute greatly to the pruning of the search tree. Since pruning the search tree can significantly reduce the number of sub-problems that must be solved, it is worthwhile adopting a more intelligent strategy to discover much smaller memosets which are likely to be encountered many times.

Local Memoset Discovery via Sub-problem Pre-processing

The memoset discovery strategy adopted herein is based on the CSP technique of enforcing *arc consistency* (see Section 2.3.1), which is commonly used to simplify a CSP prior to search, reducing the effort required by the subsequent search algorithm. Variable domains are filtered to remove elements that cannot take part in any solution. This local update is the basis for further filtering, hence the result of any change is propagated through the problem.

If the mutual exclusion constraints are such that a supporting action in the domain of a subgoal is incompatible with all elements in the domain of another subgoal, that element may be filtered from consideration since it cannot take part in any valid solution to the sub-problem. The corresponding action node is *not* removed from the planning graph since its removal can only be inferred in the context of this sub-problem. The process continues until no further elements may be filtered. The problem is then said to be arc consistent. However, if at any stage all elements in a domain are removed the problem is inconsistent (i.e. insoluble). Efficient algorithms exist to enforce arc consistency [12, 98, 116].

Fig. 5.7. Enforcing Arc Consistency on a Sub-problem

Figure 5.7 presents a sub-problem at $level_5$ taken from the example problem. As noted in Section 5.2.3, this sub-problem is encountered when trying to establish support for the plan goals ('at $pkg_1 c_3$ ' and 'at $pkg_2 c_3$ ') at $level_7$. First, the Noop in the domain of 'on $pkg_1 truck_1$ ' can be filtered, since it is inconsistent with all elements of the domain of 'on $pkg_2 truck_1$ '. The Load-truck action in the same domain can also be filtered, since it is inconsistent with all elements in the domain of 'at $truck_1 c_2$ '. Hence, the sub-problem is inconsistent and its constituent propositions are conjunctively unsupportable.

Having established arc consistency, assuming it is not inconsistent a sub-problem is easier to solve due to its reduced size. However, this is not the chief goal of enforcing arc consistency in this context. By keeping track of the reason for the removal of each element, explanations for failure to find a solution can be constructed. The *eliminating explanation* of a domain element d_i , a term borrowed from [68], is used herein to specify the variable, x_e , responsible for the removal of d_i , i.e. that there are no elements in D_e which are compatible with d_i . In Figure 5.7, for example, 'on $pkg_2 truck_1$ ' is an eliminating explanation for Noop in the domain of 'on $pkg_1 truck_1$ '.

In the case of an inconsistent problem, the eliminating explanations for each element of the totally filtered domain can be traced to provide an overall explanation for the inconsistency of the problem in terms of a conjunctively unsupportable subset of propositions. The basic tracing procedure, using eliminating explanations gained while enforcing arc consistency, is given in Figure 5.8. The domains of all eliminating explanations must also be examined since they may contain elements filtered by variables external to the current memoset. If these external variables are not also added to the memoset, the filtered domain elements would consequently be reinstated, potentially allowing combinations of assignments to the memoset variables that are in fact supportable.

```

01. Procedure constructMemoSet( $D_i$ )
02.    $trace \leftarrow \{D_i\}$ 
03.    $memoSet \leftarrow \{x_i\}$ 
04.   While  $trace \neq \emptyset$ 
05.     Select and Remove  $D_t \in trace$ 
06.     Foreach  $d_{t-f} \in D_t$  where  $d_{t-f}$  has been filtered
07.        $x_e$  is the eliminating explanation for  $d_{t-f}$ 
08.       If  $x_e \notin memoSet$  And  $D_e \notin trace$ 
09.         Add  $x_e$  To  $memoSet$ 
10.         Add  $D_e$  To  $trace$ 
11. End Procedure

```

Fig. 5.8. Basic Memoset Construction Following Total Filtering of D_i

The resulting subset of propositions can then be recorded as a memoset. The benefit of this process comes from the size of the memosets recorded. In the example problem, ‘connects r_1 c_1 c_2 ’ has no bearing on the inconsistency of the problem and is therefore excluded from the memoset. In this example the saving is small, but in more complex examples with larger sub-problems the saving is generally greater, leading to a significant pruning of the search tree.

Figure 5.7 describes only one of several ways in which to enforce arc consistency on the given sub-problem. It is equally possible, for example, to start with ‘on pkg_2 $truck_1$ ’, filtering the Load-truck action since it is incompatible with all elements of the domain of ‘on pkg_1 $truck_1$ ’. In this simple example the starting point makes no difference to the memoset identified, but this is not uniformly the case. A poor variable/constraint selection order can lead to sub-optimal memosets being constructed.

Section 5.8 outlines a heuristic method of identifying small memosets, but there is a simple method of improving the memoset construction procedure itself to remove variables that are not part of the inconsistency. Consider Figure 5.9, which presents an inconsistent sub-problem with all elements of D_1 filtered. Tracing the eliminating explanations for this initial domain immediately derives the partial memoset $\{x_1, x_2, x_3\}$. Ordinarily, x_4 must also be added to the memoset. In this case it is not necessary since a mutual exclusion constraint exists between d_{1-2} and d_{3-1} . Hence, even if d_{3-1} had not been eliminated by x_4 , x_3 would eliminate d_{1-2} . This *irrelevance test* may be performed on all filtered domain elements to test whether it is actually necessary to add their eliminating explanations to the current partial memoset. The test is whether or not every domain element (in this case d_{1-2}) whose eliminating explanation causes the current domain (D_3) to be considered is incompatible with the filtered domain element currently under consideration (d_{3-1}).

Irrelevance testing works to limit the effects of a poor order of constraint selection when enforcing arc consistency. In the example presented in Figure

Fig. 5.9. Avoiding Irrelevant Variables during Memoset Construction

5.9, if the constraints involving x_1 , x_2 and x_3 are processed first, all domain elements are still filtered from D_1 , leading to the memoset $\{x_1, x_2, x_3\}$ without ever considering x_4 .

Local Memoset Discovery via Search

Enforcing arc consistency is not guaranteed to detect an inconsistent problem [109]. It is possible that a more complex inconsistency will be uncovered only during subsequent search. For this reason, there must also be a mechanism for constructing a memoset while solving a sub-problem. When a dead end is discovered in the search, all elements of the current variable's domain are incompatible with the current partial assignment, forming a *conflict set* with it. This inconsistency can be recorded as a partial memoset for this sub-problem.

For the purpose of memoset discovery, only the variables involved in the conflict are recorded. *Shallow Learning* [30] is then used to remove irrelevant elements from this conflict set, where an irrelevant variable has an assignment which is compatible with all possible assignments to the current variable. If the sub-problem is inconsistent, the memoset is constructed from the union of all discovered conflict sets. The number and size of these conflict sets may be limited heuristically, as described in Section 5.8 below.

Propagated Repair

Memoset information gathered at child sub-problems is useful in guiding further solution extraction at the parent sub-problem. The memoset gained from the current sub-problem at $level_{i-1}$ can be mapped back onto the action nodes in the solution assignment at $level_i$ which generated the sub-problem at $level_{i-1}$ using the precondition relations in the planning graph.

Fig. 5.10. Reverse Mapping from a Memoset to a Sub-assignment via Activity Constraints

Figure 5.10 presents the propagation of the memoset constructed from the sub-problem given in Figure 5.7. In general, a memoset propagated from $level_{i-1}$ can specify more than one combination of assignments at $level_i$ since a number of actions can share the same precondition. Furthermore, the variables of a sub-problem at $level_{i-1}$ might be a superset of more than one memoset previously recorded at $level_{i-1}$. This situation occurs if there are multiple reasons for a lack of support for a particular set of actions. Hence several memosets can be propagated at once.

The sub-assignments thus specified represent new constraints (which may be binary, or a larger arity) that can be added to the current instance of a sub-problem at $level_i$, creating a new instance in the dynamic sequence. Note that this process is *not* the same as conflict recording during the solution of a single CSP, since the new constraints here come from an external source.

Through repairing the previous solution, the Local Changes algorithm can then solve this new problem which disallows the combination of assignments (i.e. ‘at truck₁ c₃’ $\leftarrow$ Drive-truck, ‘on pkg₁ truck₁’ $\leftarrow$ Noop, ‘on pkg₂ truck₁’ $\leftarrow$ Noop) that led to the memoset or memosets.

The propagation process means that LC only has to discover one solution for each of a limited sequence of increasingly constrained sub-problems, rather than attempting to enumerate all solutions to a single sub-problem. This is more efficient since each propagated repair can potentially rule out many of the solutions to the original sub-problem. The propagated repair process emphasises the usefulness of *rrDCSP* techniques in this hierarchical context. It is described in Figure 5.11.

Propagated repair creates closely related *sub-sequences* of sub-problems based on an initial sub-problem and extended through the addition of propagated constraints. All sub-problems in a sub-sequence contain the same set of propositions represented as CSP variables, with an increasing number of

Fig. 5.11. The Sub-problem Hierarchy under Propagated Repair

constraints. The whole $rrDCSP$ for $level_i$ now constitutes a set of such sub-sequences. The transition between sub-sequences occurs when the current sub-problem of one sub-sequence is found to be over-constrained due to the number of extra constraints added. A memoset for this sub-sequence is then returned to $level_{i+1}$. The process for deriving a memoset for a sub-sequence is detailed below. The current sub-sequence at $level_{i+1}$ is therefore extended, and the solution to the current sub-problem at $level_{i+1}$ defines the start of a new sub-sequence at $level_i$.

Propagated Memoset Discovery for Sub-sequences

The propagation of memoset information up the level hierarchy is also essential for the discovery of memosets composed of subsets of the propositions associated with a sub-sequence. The discovery of such memosets at $level_i$ is necessary to identify the constraints required for propagated repair at the current sub-problem at $level_{i+1}$.

A sub-sequence memoset at $level_i$ is constructed incrementally as memosets are propagated upwards from $level_{i-1}$. Each sub-assignment tuple of the solution to the current sub-problem at $level_i$ is associated with a subset of the propositions in the sub-problem. This proposition subset comprises a partial memoset for the sub-sequence, covering just the related sub-assignment tuple. In the example given in Figure 5.10, there is a simple one-to-one mapping, yielding the partial memoset $\{\text{'at truck}_1 c_3\}$, $\{\text{'on pkg}_1 truck_1\}$, $\{\text{'on pkg}_1 truck_1\}\}$. In general, however, each action can map to multiple proposition nodes, corresponding to an action's multiple effects. To ensure completeness, each such proposition must be added to the partial memoset.

Given the potential multiplicity of constraints and partial memosets that can be recorded as a result of a single propagation step, there is the opportunity for heuristic choice as to which of the potential constraints are added to

the current sub-problem. If a partial memoset for this sub-sequence already exists from a previous propagation step, it is best to record and propagate only those constraints which add minimally to the size of this memoset. Note that this does *not* lead to incompleteness, because all sub-assignments not recorded as new constraints are potentially free to re-appear within solutions to future instances in the sub-sequence. Therefore, a constraint crucial to the explanation of the unsupportability of a sub-sequence must be dealt with eventually.

When limiting the number of new constraints added, it is hoped that the constraints that are added will also directly or indirectly disallow the other potential sub-assignments found through propagation. A simple example of this is when one sub-assignment is the subset of another. Clearly, adding a constraint to disallow the former will also rule out the latter and in so doing, the need to add the extra variable required by the larger subset to the partial memoset. The heuristic propagation of memosets can potentially lead to a much more compact memoset for the current sub-sequence which will, in turn, lead to smaller memosets in the levels above when it is propagated.

5.5 The GP-*rr*DCSP Algorithm

The GP-*rr*DCSP algorithm presented in this section is an enhanced version of the Graphplan algorithm [14] for synthesising plans in STRIPS-style [50] domains. The hierarchical decomposition of the problem represented by the planning graph generated by Graphplan allows the effective application of *rr*DCSP techniques to minimise search effort. A proof of the soundness and completeness of this algorithm is given in Appendix B.

5.5.1 The Top-level Procedure

Figure 5.12 presents the procedure GP-*rr*DCSP(), which is responsible for interleaving the graph expansion and solution extraction phases. The initial level of the planning graph is first constructed (line 3) from the planning problem description. The planning graph is then extended until the problem goals exist, and are mutually consistent in the current level (lines 4,5,7). At this point, the algorithm attempts to extract a valid plan (line 6). If it succeeds, the algorithm terminates, otherwise the planning graph is extended to the next level and search continues.

5.5.2 The extract() Procedure

The `extract()` procedure, as presented in Figure 5.13, coordinates the search for a valid subgraph connecting the goal propositions to the initial conditions. The base case for this recursive procedure is when *level₀* is reached (lines 2-4).

```

01. Procedure GP-rrDCSP(planning-problem)
02.   plan  $\leftarrow \emptyset$ , goals  $\leftarrow$  getGoals(planning-problem)
03.   levelg  $\leftarrow$  graph-initialise(planning-problem)
04.   While plan  $= \emptyset$ 
05.     If goals  $\subseteq$  levelg And consistent(goals, levelg)
06.       If extract(levelg, goals)  $= \emptyset$  Return
07.       levelg  $\leftarrow$  extend(levelg, planning-problem)
08. End Procedure

```

Fig. 5.12. Pseudo-code for GP-rrDCSP()

At this point, if the current sub-goals match the initial conditions, a valid plan has been found and the search can terminate.

The first task when trying to identify support for a set of sub-goals is to check whether this sub-goal set (or a subset of it) has already been shown to be unsupportable (line 5). If this is the case, then all such subsets are returned (each representing an alternative reason for failure) for propagation at the level above (line 6). Otherwise, the CSP sub-problem corresponding to the first sub-problem of the sub-sequence is constructed (line 7).

Local Changes is used to solve this first instance (line 9), repairing the solution to the previous sub-problem solved at this level. If it is unsolvable, this is established via arc consistency pre-processing or search and an appropriate memoset is constructed and returned. If a solution can be found to the initial sub-problem, the preconditions of the action nodes which constitute

```

01. Procedure extract(leveli, subGoalsi)
02.   If leveli  $= 0$ 
03.     If matchInitConds(subGoalsi) Return  $\emptyset$ 
04.     Else Return {subGoalsi}
05.   memoSetsi  $\leftarrow$  checkMemoSets(subGoalsi, leveli)
06.   If memoSetsi  $\neq \emptyset$  Return memoSetsi
07.   probInstance  $\leftarrow$  constructProblemInstance(leveli, subGoalsi)
08.   memoSeti  $\leftarrow \emptyset$ 
09.   solution  $\leftarrow$  ACLCFC(probInstance, memoSeti)
10.   While solution  $\neq \emptyset$ 
11.     subGoalsi-1  $\leftarrow$  getPreconditions(solution)
12.     memoSetsi-1  $\leftarrow$  extract(leveli-1, subGoalsi-1)
13.     If memoSetsi-1  $= \emptyset$  Return  $\emptyset$ 
14.     propagateMS(memoSetsi-1, memoSeti, solution, probInstance)
15.     solution  $\leftarrow$  ACLCFC(probInstance)
16.   record(memoSeti)
17.   Return {memoSeti}
18. End Procedure

```

Fig. 5.13. Pseudo-code for extract()

```

01. Procedure propagateMS(memoSetsi-1, memoSeti, solutioni, probInstancei)
02.   subAssTuples  $\leftarrow \emptyset$ 
03.   Foreach memoSeti-1  $\in$  memoSetsi-1
04.     subAssTuples  $\leftarrow$  subAssTuples  $\cup$  getSubAssTuples(memoSeti-1, solutioni)
05.   Select subAssTuple  $\in$  subAssTuples
06.     memoSeti  $\leftarrow$  memoSeti  $\cup$  getEffects(subAssTuple, probInstancei)
07.   addConstraint(subAssTuple, probInstancei)
08. End Procedure

```

Fig. 5.14. Pseudo-code for *propagateMS*()

the solution are identified (line 11) and a recursive call is made to try and support them in turn (line 12).

If this recursive call is successful (i.e. a consistent subgraph has been identified, connecting the subgoals to the initial conditions), a plan has been found and the algorithm can terminate (line 13). Otherwise, the memoset or memosets responsible for the failure at the level below must be propagated back into the current level in order to determine which part of the current solution was responsible (line 14). The propagation procedure, as described below, adds a constraint to the current problem instance, defining the next sub-problem in the sub-sequence. The current solution is then repaired by LC to attempt to find another solution.

This process continues until either valid support can be found for the input subgoals, or it is shown that the subgoals are conjunctively unsupportable. In the latter case, the memoset generated by propagation is recorded and returned (lines 16, 17).

5.5.3 The *propagateMS*() Procedure

The *propagateMS*() procedure described in Figure 5.14 is responsible for propagating one or more memosets found at *level*_{*i*-1} back to *level*_{*i*}. It corresponds to the process described for propagated repair and sub-sequence memoset construction in Section 5.4.2. The result of this propagation is the creation of a new problem instance in the dynamic sequence through the addition of an extra constraint and the update of the partial memoset for the current sub-sequence at *level*_{*i*}. Note that only a single constraint, which is considered heuristically to be the best, is added on each invocation of this procedure.

propagateMS() iterates through the set of memosets returned to *level*_{*i*} (lines 3-4). Precondition relations are used to construct an associated set of sub-assignment tuples. One such tuple is selected heuristically in an attempt to limit the size of the memoset eventually recorded for the current sub-sequence. This tuple is posted as a new constraint, creating the next sub-problem in the sub-sequence and the associated propositions are recorded as a partial memoset.

5.6 Complexity Issues

In Section 3.5.3, the time complexity of the Flexible Local Changes algorithm (of which local changes is a special case) was established as $O(nm^n)$. If the number of plan goals in a g -level planning graph is G , m is again the domain size (i.e. the maximum number of action nodes that assert a particular proposition) and p is the maximum number of preconditions for any of the operators in the problem, the worst-case time complexity of synthesising a plan may be calculated as follows.

At $level_g$, there are a maximum of m^g sub-problems in the sub-sequence. Hence, Local Changes takes time $O(Gm^{2g})$ to solve them. In the worst case, every action node appearing in a solution has p preconditions, hence the number of variables in the sub-problems at $level_{g-1}$ is $O(Gp)$, taking time $O(Gpm^g m^{2gp})$ to solve. Over the whole planning graph, therefore, the worst-case time complexity can be expressed:

$$\sum_{i=1}^g Gp^{i-1} m^{2Gp^{i-1}} \prod_{j=2}^i m^{Gp^{j-2}}$$

This is approximately equal to:

$$g[Gp^{g-1} m^{2Gp^{g-1}} m^{Gp^{g-1}}]$$

Hence the time complexity of GP-*rr*DCSP is:

$$O(gGp^{g-1} m^{3Gp^{g-1}})$$

The space required by the Graphplan planning graph has been shown to be polynomial in the number of plan objects, the number of propositions in the initial conditions, the number of operators, the size of longest effects list of any of the operators, and finally the number of levels in the graph [14]. However, the worst-case space complexity of GP-*rr*DCSP as a whole is exponential when memoisation is considered. This follows from the fact that an exponential number of sub-problems are solved in the worse case, each leading to the recording of a memoset.

Given these worst-case results, increasing the efficiency of both search and memoset construction on average is fundamental to GP-*rr*DCSPs overall performance. Methods for achieving these goals are discussed in the following sections.

5.7 Avoiding Irrelevant Variables in Memosets Created by Propagation

When constructing a memoset at $level_i$ through propagation, the *core memoset* is defined as those elements of $propositions_i$ that are directly specified

Fig. 5.15. Propagated Irrelevance Testing and Propagation of Constrainedness Information

by the propagation of memosets from $level_{i-1}$. It is also necessary to consider eliminating explanations for elements of this core memoset. This is because, as per the local case described in Section 5.4.2, filtered domain elements may allow different assignment tuples to the core memoset variables that are supportable. An irrelevance test in this case is not as simple as when considering local constraints only. It is, however, worthwhile as it minimises the size of the memosets, hence pruning overall search.

The solution adopted herein is based on the creation of a *micro-problem* with variables consisting of the propositions in the core memoset. Through solving this initial micro-problem and likewise the micro-problems that arise at lower levels based on each solution found, support is sought for the core memoset. If a consistent subgraph exists connecting the core memoset to the initial conditions then, by definition, its constituent variables are supportable at this level. Therefore, the eliminating explanations for the core memoset variables must be relevant, and a new memoset containing both the core memoset and its associated eliminating explanations is recorded. If, on the other hand, no such subgraph exists, then the core memoset is established as unsupportable and recorded as a new memoset in its own right.

Micro-problems constructed for propagated memoset irrelevance testing are fitted seamlessly into the dynamic sequence of problems, as presented in Figure 5.15. This is possible because in both ‘normal’ and ‘relevance-testing’ modes, GP-*rr*DCSP is performing the same task: the identification of a subgraph (if one exists) connecting a subset of propositions at some level to the initial conditions. A useful side-effect of propagated relevance testing is that memosets may also be discovered and recorded during this process. Since the micro-problems are typically small, memosets generated in this way

are likely to be small also. Furthermore, the small size of the micro-problems means that they are easier to construct and solve on an individual basis.

Propagated irrelevance testing is used recursively, (as also described in Figure 5.15), such that if a core memoset discovered at a micro-problem also contains variables whose domains have elements filtered by variables external to it, a further irrelevance test can be initiated. This technique leads to a highly focused exploration of the search space, extracting the smallest memosets possible.

5.8 Focusing the Search

Focusing on the most constrained area of the problem [150] is an established means of reducing the effort required in solving a CSP, as noted in Section 2.5. Minimising search effort in the hierarchical context means pruning as much of the space of sub-problems as possible. This is achieved by identifying and recording minimal memosets, which are typically composed of the most constrained variables in a sub-problem. Memosets recorded at higher levels have greater pruning power than those recorded lower down the hierarchy. Therefore, the solution of a particular sub-problem cannot be considered in isolation, but rather in terms of how the memoset recorded for it will contribute (via propagation) to the memoset at the level above.

As stated in Chapter 2, the three operations of a constraint solver which may be ordered heuristically to improve efficiency are: the selection of the variable to instantiate next, the domain element to assign to the selected variable, and the order in which constraint checks are made. The following sections describe how GP-rrDCSP orders these three operations to identify the smallest memosets, hence promoting overall efficiency.

5.8.1 Variable Selection

Variable selection during search is well established as having a marked effect on efficiency when solving an individual CSP (see Section 2.5). This process is equally important when trying to identify minimal memosets. A good selection order maximises the chance of detecting an inconsistent sub-problem earlier, and reduces the risk of adding an irrelevant variable to a memoset. Hence, the overall likelihood of a minimal memoset being found can be increased.

Given a partial memoset discovered at $level_i$, the goal when solving the current sub-problem at $level_{i-1}$ is to minimise the necessary additions to this memoset under propagation. The heuristic used is to select those variables first which map directly back to the elements of the current partial memoset at $level_i$, as presented in Figure 5.16. The likelihood is therefore increased that when the memoset recorded at $level_{i-1}$ is propagated back to $level_i$ it

Fig. 5.16. Variable Selection Based on Parent-node Membership of Partially-Constructed Memosets

will not cause an increase (or, at worst, will cause minimal increases) in the size of the partial memoset at $level_i$.

Partial memoset membership information can be propagated down the level hierarchy so that the need to minimise the size of the memosets recorded at higher levels is taken into consideration even when making variable selections at a much lower level in the level hierarchy. This process is described in Figure 5.17. The likelihood is accordingly increased that, when the memosets at the lower levels are propagated back up the hierarchy, minimal memosets can be established at the important higher levels.

The smallest domain size heuristic is well established as a means of improving efficiency in single CSPs (Section 2.5). It is used here to break ties if two or more variables have the same number of parents already involved in partial memosets. Variables with smaller domains are more likely to form smaller memosets since fewer conflicts are needed to explain the inconsistency of each domain element. Furthermore, fewer sub-problems will have to be explored to generate these conflicts.

Domain size information is propagated down the level hierarchy in the same way as partial memoset membership, as presented in Figure 5.18. This information is particularly useful when little or no partial memoset information exists (when solving a sub-problem generated from the first solution of the current sub-problem at the level above, for example).

5.8.2 Value Selection

The selection of a particular action to support a proposition directly specifies the composition of the sub-problem to be solved at $level_{i-1}$. Hence, it is important to select assignments which are most likely to lead to smaller

Fig. 5.17. Propagating Partial Memoset Information down the Level Hierarchy

memosets being recorded at $level_{i-1}$, as well as memosets which, when propagated back to $level_i$, lead to the smallest number of additions to the current partial memoset.

The heuristic used here selects the domain element in $actions_i$ whose preconditions in $propositions_i$ are mutually exclusive with the *smallest* number of other proposition nodes in $level_i$. The rationale behind this heuristic is as follows: such a scheme will tend to lead to the construction of a sub-problem at $level_{i-1}$ with fewer constraints and hence a higher likelihood of containing smaller memosets. At the least, this scheme will tend to alleviate the need for propagated irrelevance testing.

5.8.3 Constraint Check Selection

The inconsistencies which are used to construct the memosets are identified as constraints are checked during either pre-processing or search itself, as noted in Section 5.4.2. A sub-problem may, however, contain many inconsistencies, some of which will contribute to more compact memosets than others. Consider again Figure 5.9, describing the possibility of mistakenly recording a non-minimal memoset. If, however, the ‘right’ constraints are checked first this situation can be avoided. This has the added benefit of alleviating the need for propagated irrelevance testing, which may entail the solution of several extra sub-problems.

Fig. 5.18. Propagating Domain Size Information down the Level Hierarchy

Since it is potentially expensive to try the required constraint checks in all possible orders, heuristics are employed to try and find this ‘right’ order, or at least a relatively good order, leading to the immediate identification of small memosets. The primary basis upon which constraints are selected is herein the same as for variable selection: the number of parents already in partial memosets, with ties broken using propagated domain sizes. However, in the case of further ties, some local information can be used. If, for instance, an $arc(i, j)$ has been responsible for a deletion already, it is worth checking first since several deletions caused by a single variable are likely to lead to smaller memosets. Note that constraint selection heuristics are employed in both the pre-processing and search phases, with the heuristic rank being based on both variables involved in the constraint.

5.9 Summary

This chapter has considered the AI planning problem [2], and techniques to improve the efficiency of plan synthesis as an application of the CSP approaches considered in earlier chapters. A novel hierarchical structuring of

the plan extraction process in the Graphplan framework [14] was presented as the foundation for these efficiency gains. As the next chapter will show, this structuring also facilitates the incorporation of flexible information into the planning graph and hence the planning problem itself. It was first shown how, by viewing a sequence of sub-problems at a particular level of the planning graph as a restriction/relaxation-based dynamic constraint satisfaction problem, dedicated *rrDCSP* algorithms could for the first time be applied to the plan synthesis problem.

The solution mechanism based on the hierarchical search of the planning graph was discussed in detail. The discovery and propagation of small *memosets*, containing conjunctively unsupportable proposition nodes with respect to a particular level of the planning graph, was shown to further increase the efficiency of this process. It works both by pruning the search in the space of sub-problems and by facilitating the repair of existing complete solutions to sub-problems rather than the simple enumeration of every such solution.

Pseudo-code was given for an implementation based on the techniques described above: *GP-rrDCSP*. The soundness and completeness of this algorithm was established, and issues of time and space complexity were addressed. In order to improve the average-case performance of this planner, means of constructing smaller and therefore more powerful (in terms of pruning power) *memosets* were discussed. Furthermore, focus heuristics were described in order to aid the plan extraction mechanism in finding the most useful *memosets* to learn. The following chapter evaluates *GP-rrDCSP* on benchmark problems against leading Boolean planners.

6 GP-*rr*DCSP: Experimental Results

This chapter investigates the efficacy of the hierarchical decomposition approach to Boolean planning embodied by GP-*rr*DCSP. The motivation for this study is not to show that GP-*rr*DCSP is a vast improvement over the current state of the art, although in some of the tests excellent results are obtained, but to establish this planner as a solid platform upon which to build a fully flexible planner in the subsequent chapters.

Standard benchmark problems are employed to compare the performance of GP-*rr*DCSP against leading Boolean solvers. The solvers chosen include Graphplan itself as a baseline, and three others chosen to represent the current state of the art, including: STAN 3 [96], IPP 4 [86] and BLACKBOX 3.6b (SATZ-RAND) [83]. Five domains were tested. The first two of these, the Logistics and Blocks-world domains can be obtained as part of the Blackbox distribution¹. The remainder; the Gripper domain, the Movie domain and the Grid domain, were used in the AIPS-98 planning competition [95]². The performance of GP-*rr*DCSP on a number of these benchmark problems is examined in detail to highlight the operation of various parts of the algorithm and to suggest how it could be improved further.

Also tested is the utility of the propagated irrelevance testing technique for limiting the size of memosets recorded, as described in Section 5.7. Throughout, the version of the planner that does not employ propagated irrelevance testing is identified as GP-*rr*DCSP*. Experiments are not performed on the GP-*rr*DCSP algorithm without the use of focus heuristics (see Section 5.8), since these are an integral part of the planner without which its efficiency is severely impaired. As an example, the LogisticsA problem (see Section 6.1.3) that can be solved so efficiently with the full version of the planner takes hundreds of times the amount of effort to solve without the focus heuristics.

Given that the results are reported in terms of run-times, which is common in AI Planning, it should be noted that GP-*rr*DCSP is implemented in Java and the other solvers in C/C++. The hardware used is a Sun Ultra 5-360

¹ <http://www.cs.washington.edu/homes/kautz/blackbox/index.html>

² <ftp://ftp.cs.yale.edu/pub/mcdermott/papers/aipscomp.tar.gz>. Note that the AIPS-2000 planning competition has since taken place (<http://www.cs.toronto.edu/aips2000/>), but also contains logistics and blocks world problems.

Fig. 6.1. The RocketA Problem

with 128 Megabytes of RAM. All planners are run on this hardware in order that a meaningful comparison is obtained.

6.1 The Logistics Domain

The goal in the Logistics domain is the transportation of a number of objects, via truck and aeroplane, from one city to a destination at another. The trucks are limited to staying within a single city, with the aeroplanes being used to carry items between different airports. Both the trucks and aeroplanes can carry an unlimited number of items.

6.1.1 The RocketA Problem

The RocketA problem is presented in Figure 6.1. This is a relatively simple logistics problems that does not use trucks. The destination points for each person/item are always airports. The solution is also relatively short: an optimal parallel plan takes seven time-steps.

As Table 6.1 describes, GP-*rr*DCSP performs very well on this problem, being second only to Blackbox. The reason for this success is the identification of a small set of propositions that are unsupportable during the first extraction phase, followed by focusing the search on the propositions and actions related to this small set subsequently, as will be shown.

The goal propositions first occur at *level*₄. However, via arc consistency pre-processing GP-*rr*DCSP immediately establishes that the propositions that April, Michelle and Lisa are at their respective destinations of Boston, JFK and Paris are unsupportable at this level. The corresponding memoset contains less than one third of the variables in the sub-problem (which represents the ten goal propositions of the people and packages) and therefore enables strong pruning and efficient propagated repair of the search tree later on in the search process.

Note that there are several, equally good alternatives to this initial memoset. The important point is not which one is chosen, but, having made the choice, focusing the search on related propositions and actions in order to preserve optimally small memosets at the higher levels of the search tree.

Following the failed initial extraction phase, the planning graph is extended to *level*₅. The first solution to the sub-problem at this level produces a sub-problem at *level*₄ whose variables are a superset of the 3-element memoset as recorded above. Propagating this memoset back to *level*₅ produces a partial memoset {'at April Boston', 'at Michelle JFK', 'at Lisa Paris'}. GP-*rr*DCSP is now in a position to a) repair just the assignments to the variables representing these propositions and b) use the partial memoset to focus the search at *level*₄.

The next steps consist of investigating each combination of assignments to the variables representing April, Michelle and Lisa's goal propositions and finding that each is unsupportable at *level*₄. Because the search is focused on the support for these three propositions, GP-*rr*DCSP is able to record a 3-element memoset at *level*₄ in each case, until all combinations have been exhausted and it can record {'at April Boston', 'at Michelle JFK', 'at Lisa Paris'} as a memoset at *level*₅.

The attempted extraction at *level*₆ is lengthier. The problem is almost solvable in 6 steps, with eight of the 10 goals attainable (from Figure 6.1, aeroplane 1 can deliver Jason, Betty and Michelle, and aeroplane 2 can deliver Lisa, Paper, April, Pencil and MXF in 6 steps). Hence, it is more difficult to prove the problem unsolvable in 6 steps. However, by a similar process to that described above, GP-*rr*DCSP records a minimal 9-element memoset, which is then used to guide the successful search for a solution from *level*₇.

Table 6.1 also demonstrates that the version of GP-*rr*DCSP that does not use propagated irrelevance testing actually performs fractionally better than the full version of the algorithm. This is unsurprising given that this technique is little used in this problem even when it is available. Hence, the effort expended on the solution of several micro-problems is unlikely to be rewarded with significant performance gains.

6.1.2 The RocketB Problem

Figure 6.2 presents the RocketB problem, a variation on the RocketA problem of Figure 6.1. An optimal plan for this problem also contains 7 steps, although

Problem	Length	GP- <i>rr</i> DCSP	GP- <i>rr</i> DCSP*	GP	STAN	IPP	BLACKBOX (SATZ-RAND)
Rocket-a	7	13850	13575	75410	33428	47110	5450
Rocket-b	7	20979	37804	154330	2021	76060	8312

Table 6.1. Comparative Run-times (in milliseconds) for the Logistics Rockets Problems.

it seems to be slightly more difficult to solve. This is possibly because of the more even distribution of people/objects: with respect to GP-*rr*DCSP at least, this makes it less obvious which items are the most tightly constrained and therefore likely to form part of the smallest memosets.

As the run-time results in Table 6.1 demonstrate, GP-*rr*DCSP is also successful on this problem. Again, the goal propositions first occur at *level*₄ and a 3-element memoset is established through arc-consistency pre-processing: {'at Michelle JFK', 'at Betty JFK', 'at Paper Paris'}. This memoset is used in exactly the same manner to prune and focus the search during the extraction phases from *level*₅ and *level*₆ before the solution is found in 7 steps.

In contrast with the RocketA problem, propagated irrelevance testing is used much more in this problem, and can be seen to be effective in reducing search effort. In this problem, GP-*rr*DCSP uses this technique on few occasions, but makes an extensive effort (in terms of number of micro-problems solved) to prove that variables external to the core memoset are irrelevant. The reward for this effort is twofold, with both the core memoset and the

Fig. 6.2. The RocketB Problem

Fig. 6.3. The LogisticsA Problem

small memosets recorded during propagated irrelevance testing leading to increased pruning of the search tree.

6.1.3 The LogisticsA Problem

The LogisticsA problem (see Figure 6.3) is a more complex problem than those presented thus far, with each of three cities split into an airport and post-office area. A truck local to each city is used to transport packages between the two areas. As the results given in Table 6.2 demonstrate, GP-*rr*DCSP copes very well with this extra complexity, actually finding this problem easier than the rockets problems. This is also the case for STAN, but Graphplan and IPP find this problem significantly more difficult.

GP-*rr*DCSP may be finding this problem easier because there are only three cities, but it is also likely that the arrangement of packages is a contributing factor. Some of the packages, such as package 4 and package 6 are clearly less constrained than others, having only to be delivered to an airport. The focus heuristics employed by GP-*rr*DCSP lead it away from constructing memosets based on propositions concerning these packages and towards those concerning, for example, packages 3 and 7 which must both be taken to a post-office, requiring the use of a second truck when they arrive at their destination city.

The goal propositions first appear at *level*₉. The sub-problem at this level representing the goal propositions is soluble, as are subsequent sub-problems at *level*₈ and *level*₇. The sub-problem at *level*₆, however, is inconsistent and

the memoset {'at package3 la-airport', 'at package7 pgh-airport', 'at package8 pgh-airport'} is recorded.

Although there are several other combinations of propositions that are mutually inconsistent, these three were focused on via the propagation of constrainedness information (since no partial memoset information is initially available). The utility of this process is underlined by the following section of the search. Firstly, when propagated upwards to *level*₇ the propagated repair process discovers that the solution to the relevant part of the sub-problem cannot be repaired, hence another minimal memoset, {'in package3 la-truck', 'in package7 pgh-truck', 'in package8 pgh-truck'}, is recorded at this level.

Furthermore, when this second memoset is propagated upwards, propagated repair again can find no alternative consistent assignment to the relevant part of the sub-problem at *level*₈. Since domain elements of variables in the core memoset {'in package3 la-truck', 'in package7 pgh-truck', 'in package8 pgh-truck'}, were filtered by the initial arc consistency enforcing step with respect to two variables external to the core memoset, 'at pgh-truck pgh-po' and 'at la-truck la-po', propagated irrelevance testing may be employed at this point.

As can be seen from the results, this technique has little effect on the search effort required to solve the LogisticsA problem, largely because GP-*rr*DCSP can already solve this problem efficiently. However, propagated irrelevance testing *does* have a significant effect on the efficiency of GP-*rr*DCSP on the more difficult problems in the Logistics domain, and this problem provides a useful illustration of the operation of this technique. It is therefore examined below.

In order to determine whether the external variables should actually be added to the memoset, GP-*rr*DCSP examines the micro-problem concerning just the core memoset variables, with all their domain elements restored. This process entails solving a further 20 micro-problems, which is not an insignificant number, but the size of these problems is, by definition, relatively small. In addition, further 3-element memosets are recorded during propagated irrelevance testing which, due to their small size, may be of use later in the search. This process ends with the result that the core memoset alone is established as unsupportable and it is therefore recorded. This is clearly correct: by inspection of Figure 6.3, no matter what the position of the trucks involved, the propositions in the core memoset described above cannot be achieved in eight steps.

The final part of this extraction phase is to propagate the memoset recorded at *level*₈ upwards. Again, there is no alternative consistent assignment to the relevant part of the sub-problem at *level*₉. Hence, the minimal memoset {'at package3 la-po', 'at package8 pgh-po', 'at package7 pgh-po'} is recorded. This memoset is very powerful in terms of pruning power, leading

Problem	Length	GP- <i>rr</i> DCSP	GP- <i>rr</i> DCSP*	GP	STAN	IPP	BLACKBOX (SATZ-RAND)
LogisticsA	11	7995	7307	1955780	1038	1513450	7012
LogisticsB	13	99138	230892	862100	2627	633290	24389
LogisticsC	13	123064	193825	-	704317	-	46126
LogisticsD	14	171556	50460293	-	7096003	-	108261

Table 6.2. Comparative Run-times (in milliseconds) Logistics Problems a-d. A dash indicates no solution found in 24 hours.

to a minimal amount of search during the extraction phase at $level_{10}$ before a solution is found in 11 steps.

The success of GP-*rr*DCSP on this problem is based on focusing on the most tightly constrained propositions as the search progresses down the hierarchy: it is no accident that propagated repair cannot find alternative consistent assignments as the memosets were propagated upwards.

6.1.4 The LogisticsB and LogisticsC Problems

Figure 6.4 presents the LogisticsB problem. This problem is more difficult to solve than LogisticsA for GP-*rr*DCSP, as the results given in Table 6.2 demonstrate. Part of the reason for this is that the solution contains 13 steps, increasing both graph expansion and search time. This stems from the fact

Fig. 6.4. The LogisticsB Problem

Fig. 6.5. The LogisticsC Problem

that there are four rather than three cities, and also that each package must be delivered to a post-office.

GP-*rr*DCSP focuses on packages 3, 5 and 7, based on constrainedness propagation. This is a good choice since all three have different starting cities, therefore the most resources are required to achieve subgoals involving these three packages concurrently. Hence, memosets involving these three packages are likely to be small. This is indeed the case, with 3-element memosets involving packages 3, 5 and 7 recorded at $level_9$ and $level_{10}$ following the first two failed extraction phases.

The extraction phases at $level_{11}$ and $level_{12}$ are more difficult, however. From Figure 6.4, it may be seen that the goals of any four of the packages can be satisfied simultaneously in 11 steps. Hence, 5-element memosets are recorded following these two extraction phases. Since these memosets constitute the entire set of goals, they lead to little pruning themselves, providing a further explanation for the increased difficulty of this problem. Minimal contributory memosets recorded at the lower levels of the hierarchy may still perform useful pruning, however.

As can be seen in Figure 6.5, the LogisticsC problem is simply a superset of LogisticsB. The extra packages, 4 and 6, need only to be delivered to airports and are therefore less constrained than those packages also in the LogisticsB problem. GP-*rr*DCSP is therefore still able to focus on the most constrained packages, solving this problem with little extra effort. Some of the other planners tested do have considerably more difficulty in solving this

Fig. 6.6. The LogisticsD Problem

problem, as presented in Table 6.2, with Graphplan and IPP unable to find a solution within 24 hours.

The positive effect of propagated irrelevance testing on the LogisticsB and LogisticsC problem is quite marked, as presented in Table 6.2. This is unsurprising, given the similarity in structure between these problems and LogisticsA: GP-*rr*DCSP is able to filter out irrelevant elements from memosets in the same manner as described in the previous section. Since the planner finds these two problems significantly more difficult, however, there is considerably more scope for propagated irrelevance testing to reduce search effort.

6.1.5 The LogisticsD Problem

The LogisticsD problem presented in Figure 6.6 is the most difficult problem from the Logistics domain considered in this study. Not only is the number of cities increased, but each city contains three connected areas. Nevertheless, some goals are easier to achieve than others. Consider, for example, that package 9 has a delivery destination within the same city. Hence, memosets that

Fig. 6.7. The Blocks-world 12Step Problem

contain this package are unlikely to lead to much pruning and GP-*rr*DCSP avoids constructing them. Conversely, packages such as 2 and 6 require two trucks and an aeroplane to be delivered and are therefore focused on by the GP-*rr*DCSP search procedure, leading to the short run-time presented in Table 6.2.

As the run-times also demonstrate, employing propagated irrelevance testing on the LogisticsD problem results in a large improvement. As the number of plan objects and options (routes and cities in this case) increase, the importance of recording small memosets in order to keep the size of the search space manageable also increases.

6.2 The Blocks-world Domain

The Blocks-world domain involves the stacking of several blocks, rearranging them from their initial state into the goal state. The resource available is a single robot arm which can carry just one block at a time. Hence, one action is possible per time-step, presenting a different challenge to the Logistics domain.

6.2.1 The 12-step Problem

The first problem considered from the Blocks-world domain is presented in Figure 6.7. The 12Step problem is relatively simple, largely because many of the objects in the initial definition are not specified in the goals. It can be solved efficiently by all the planners considered, as presented in Table 6.3. As has been the case with all the problems that GP-*rr*DCSP has found the easiest to solve, propagated irrelevance testing is little used, even when it is available to the algorithm. This suggests that the need to resort to this technique for a particular problem is a good indicator that the problem will provide a greater challenge for GP-*rr*DCSP.

As its name suggests, the solution to this problem contains 12 single steps. The goal propositions first appear at *level*₉. GP-*rr*DCSP quickly establishes

Problem	Length	GP- <i>rr</i> DCSP	GP- <i>rr</i> DCSP*	GP	STAN	IPP	BLACKBOX (SATZ-RAND)
bw-12step	12	928	923	980	212	250	6152
bw-Large-a	12	4784	5166	3590	579	880	48961
bw-Large-b	18	120682	112596	198340	32486	15300	-

Table 6.3. Comparative Run-times (in milliseconds) for the Blocks-world Problems. A dash indicates no solution found in 24 hours.

that the goals are unsupportable, however, as a result of finding the memoset {'on C D', 'clear F', 'arm-empty'} at $level_7$. Upon propagation, memosets containing just two elements, {'holding F', 'on C D'} and {'on F A', 'on C D'} are recorded at $level_8$ and $level_9$ respectively.

The discovery of such two-element memosets not detected by the graph expansion process suggests a future addition to GP-*rr*DCSP. Since a two-element memoset corresponds exactly to a binary mutual-exclusion relation between its constituent propositions, such a relation can be added to the planning graph. As the planning graph is extended from $level_i$, this extra relation can then be directly used when determining the actions and their mutual exclusion relations at $level_{i+1}$. The result of this process is likely to be a reduction in subsequent search, due to there being either fewer actions or more mutual exclusion relations at $level_{i+1}$. In the latter case, a subproblem is more likely to be inconsistent because of the increased number of constraints.

Given the strong pruning power of the two small memosets recorded, the extraction phases at $level_{10}$ and $level_{11}$ produce a very limited amount of search before failing. GP-*rr*DCSP then quickly finds the solution at $level_{12}$.

6.2.2 The Blocks-world LargeA Problem

The Blocks-world problem LargeA presented in Figure 6.8 contains two more blocks than the 12Step problem. However, it too can be solved in 12 steps since some blocks (e.g. blocks 4, 6 and 7) are already in their goal positions. Because of this, the LargeA problem is very little more difficult for most of the algorithms considered, as can be seen in Table 6.3.

The fact that three of the blocks are in their goal positions in the initial state means that the whole set of goal propositions first appear and are consistent at $level_8$. However, GP-*rr*DCSP determines that this initial subproblem is inconsistent and immediately extends the planning graph to $level_9$. The algorithm is similarly successful here, discovering two small 3-element memosets at $level_8$ and then employing propagated irrelevance testing to limit the size of the memoset recorded at $level_9$. On this occasion the fact that the core memoset is unsupportable is established by solving just four micro-problems (each resulting in a further 3-element memoset being recorded).

Fig. 6.8. The Blocks-world LargeA Problem

Due to the ease with which this problem is solved, however, propagated irrelevance testing can again be seen to provide no gain in performance. This technique is used more on the LargeA problem than on the 12Step problem, but requires too much effort when proving external variables irrelevant in each case to provide an overall gain in performance. As is discussed below, this appears to be a feature of the Blocks-world domain.

6.2.3 The Blocks-world LargeB Problem

Figure 6.9 describes the Blocks-world LargeB problem, containing 11 blocks. Despite the fact that three of the blocks are again in their goal positions initially (and do not have to be moved), this problem is significantly harder than the two others from the Blocks-world domain discussed so far. This fact can clearly be seen from the results given in Table 6.3.

With respect to GP-*rr*DCSP in particular, problems with longer solutions present more of a challenge because of the opportunity for the propagation of errors. Consider the case where a non-minimal memoset is recorded at $level_i$, perhaps as a result of a failure of the focus heuristics. When this memoset is propagated to $level_{i+1}$, the likelihood is that a non-minimal memoset will be constructed at the higher level also. The greater the number of levels in the planning graph, the further this error can propagate, eventually leading to a large memoset with very little utility for pruning. This is the main reason for the comparative difficulty GP-*rr*DCSP has in solving the Blocks-world LargeB problem. A significant challenge for future versions of this planner is therefore to eliminate, as far as possible, such non-minimal memosets.

The run-time results also demonstrate, however, that propagated irrelevance testing actually has a small detrimental effect on this problem. This can also be ascribed to the size of the problem in terms of the number of steps in the solution. Given an increased number of levels in the planning graph, it is not surprising that the irrelevance testing process can sometimes be very expensive. Consider the case where there are several satisfying assignment tuples for a core memoset. If the core memoset is unsupportable, each of these tuples must be shown to be unsupportable. If, on the other hand, the

Fig. 6.9. The Blocks-world LargeB Problem

core memoset is supportable, then at least one of the satisfying tuples forms part of a consistent subgraph including the initial conditions.

As the size of the planning graph grows, the difficulty of establishing such a consistent subgraph increases. Furthermore, if the single supportable tuple among many unsupportable tuples is examined last by the propagated irrelevance testing process, much effort can be wasted. Two potential improvements could be made which should help to overcome these problems. The first is placing a bound on the propagated irrelevance testing process: if an assignment tuple cannot be proven unsupportable within a certain number of planning graph levels, it is assumed to be supportable. Note that this does not lead to incompleteness, but to non-minimal memosets.

The second improvement is to place a heuristic ordering on propagated irrelevance testing. Those assignment tuples that are deemed most likely to be supportable would then be examined first, reducing wasted effort.

6.3 The Gripper Domain

The Gripper domain is deceptively simple. Each problem involves the carrying of several balls from one room to another. The robot that performs this action has two grippers, hence it can carry just two balls at a time. The difficulty lies in the symmetry of the problem: the balls and their associated goals are essentially indistinguishable. Therefore, as noted in [95], there are many ways in which the problem is *almost* solvable. The difficult nature of this domain is demonstrated by the fact that, of the 20 problems in the Gripper domain, only the first 4 are solvable by the planners in this study, as given in Table 6.4.

The first problem in the Gripper domain, comprising just four balls, is presented in Figure 6.10. As the problem number increases, so does the number of balls, hence problem 2 has six balls, problem 3 has eight balls, and so on.

GP-*rr*DCSP performs as well as the other planners on problem 1, recording barely any memosets with more than three elements during its short search. It is similarly successful on problem 2, where it is noticeable that

Fig. 6.10. The Gripper Problem, Number 1

propagated irrelevance testing begins to have a visible benefit on the search process. As the number of balls increases, however, GP-*rr*DCSP becomes less competitive with the most successful planners on this domain, although it still solves the problems.

Given the lengths of the solutions to problems 3 and 4, two effects also visible in the Blocks-world LargeB problem can be seen clearly. Firstly, the propagation of non-minimal memosets lower in the level hierarchy inevitably leads to non-minimal (and quite large) memosets higher up the hierarchy as the search progresses. This severely hampers GP-*rr*DCSPs efforts to prune the search tree, underlining the need for an improved memoset-identification process.

The increased cost of propagated irrelevance testing on larger planning graphs identified in the previous section also manifests on the longer Gripper problem. In the case of the Gripper domain, however, the effort invested in this technique does pay dividends. It is reasonable to assume, therefore, that the improvements to the propagated irrelevance testing process suggested in Section 6.2.3 could significantly improve the performance of GP-*rr*DCSP on this domain.

6.4 The Movie Domain

The Movie domain involves making the preparations for watching a movie. That is, making sure that snack foods are present, rewinding the videotape and resetting the counter on the video. Each successive problem of the thirty that make up the Movie domain has more of each type of foodstuff, potentially swamping a planner with choices.

Problem	Length	GP- <i>rr</i> DCSP	GP- <i>rr</i> DCSP*	GP	STAN	IPP	BLACKBOX (SATZ-RAND)
Gripper1	7	474	404	80	53	40	383
Gripper2	11	4098	10036	1030	746	510	106396
Gripper3	15	17932	30457	25380	10623	6930	-
Gripper4	19	280342	637649	349050	157927	68620	-

Table 6.4. Comparative Run-times (in milliseconds) for the Gripper Problems. A dash indicates no solution found in 24 hours.

However, each problem in this domain can be solved in just two parallel time-steps. In the first, the foodstuffs are obtained and the videotape is rewound. In the second, the counter is reset. None of the planners in the AIPS-98 planning competition had any difficulty in solving any of the problems in this domain [95]. GP-*rr*DCSP follows this trend, solving each in under one second. This is due to the goal-directed nature of its search. Instead of searching all the possible combinations of snack foods that could be selected, the planner simply selects the first that satisfy the subgoals at the associated level.

6.5 The Grid Domain

The Grid domain is composed of a set of linked nodes laid out in a grid formation. Some of these nodes may be locked, and the locks have a particular shape in the set {diamond, square, circle, triangle} that can only be opened by a similar-shaped key. A single robot exists in this domain, with a single arm (hence it can carry just one object at a time). The available operators allow the robot to move from one node to another adjacent and unlocked node, to pick up and put down objects, and to unlock an adjacent locked node if it possesses the correctly shaped key. The goals in the Grid domain are to move one of the shaped keys to a particular destination.

Figure 6.11 presents the first problem in this domain, and Table 6.5 gives the run-times for each of the planners. This problem is comparatively easy: the goal propositions first occur at *level*₁₄ which is the number of steps required by an minimal solution. GP-*rr*DCSP solves the problem without needing to record any memosets.

Of the five problems in the Grid domain, only one is solvable by any of the planners tested. The reason for this, as noted in [95], stems from the fact that the plans are relatively long with no parallel steps. This leads to the graph construction phase becoming prohibitively expensive. Indeed, lack of memory was uniformly the cause of failure on problem 2 for those planners that could solve the first problem, rather than insufficient time.

Problem	Length	GP- <i>rr</i> DCSP	GP- <i>rr</i> DCSP*	GP	STAN	IPP	BLACKBOX (SATZ-RAND)
Grid1	14	9435	9178	n/a	3110	3140	-

Table 6.5. Comparative Run-times (in milliseconds) for the Problem, Number 1. Graphplan was unable to solve this problem due to the size of the planning graph. A dash indicates no solution found in 24 hours.

6.6 Summary

This chapter has concerned an empirical evaluation of the GP-*rr*DCSP planner against leading Boolean solvers on benchmark problems. The results are very positive, demonstrating that GP-*rr*DCSP performs competitively across the range of benchmarks. Particularly encouraging results were obtained for the Logistics domain, with GP-*rr*DCSP able to focus on the most constrained goals and in so doing reduce search considerably.

Fig. 6.11. The Grid Problem, Number 1

The results from this study have also brought current weaknesses of GP-*rr*DCSP to the fore. The first of these is the problem of propagating non-minimal memosets, which leads to the creation of many more non-minimal memosets. The higher up the level hierarchy a memoset is, the more chance it has of performing significant pruning of the search tree. Hence, generating non-minimal memosets at the top of the level hierarchy through propagation has a marked detrimental effect on search.

The second weakness identified is the cost of propagated irrelevance testing on some problems. As for the propagation of non-minimal memosets, this weakness is exacerbated by the size of the planning graph. Methods were suggested to improve the efficiency of propagated irrelevance testing, including heuristically ordering the solution of micro-problems and setting a bound on the effort expended on this process. Improved propagated irrelevance testing will lead to more minimal memosets being recorded, and hence provide at least a partial solution to the first weakness identified above.

The Gripper domain, upon which GP-*rr*DCSP performed the least well, also proved to be hard for the other planners considered. The inherent symmetry of this problem was seen to make focusing the search difficult, suggesting symmetry-breaking techniques as another important future consideration for this planner.

Having established GP-*rr*DCSP as a competitive Boolean planner, the following chapters show how it can be straightforwardly extended to synthesise *flexible* plans able to contain compromises, making use of the novel hierarchical structuring of the search.

7 Flexible Planning Problems & Flexible Graphplan

This chapter argues that, just as in the case of classical CSP, the classical AI Planning framework is too rigid to capture the full subtlety of many real problems. This is due to the fact that it too is founded on hard constraints, and hence has no capability of making compromises. In order to address this issue, a new *flexible planning problem* is defined which is designed to allow a tradeoff between the length of a plan and the number of compromise decisions it contains.

Fuzzy constraint satisfaction is the formal foundation underlying the definition of a flexible planning problem. Subjective truth degrees are associated with propositions and satisfaction degrees are associated with plan operators and goals. The latter group, represented as fuzzy constraints, express how well their preconditions are satisfied by a particular set of flexible propositions. Overall plan satisfaction degrees are calculated from the satisfaction degrees of their constituent instantiated operators and goals, enabling a direct comparison amongst a number of plans containing different compromises.

A flexible plan synthesis strategy named Flexible Graphplan (FGP) based, as per GP-*rr*DCSP (see Chapter 5), on the Graphplan framework is also developed. FGP is a natural extension of the GP-*rr*DCSP classical AI Planning technique, operating in two interleaved phases: planning graph construction, and plan synthesis. However, FGP constructs a flexible planning graph, recording the truth and satisfaction degrees attached to propositions and operators/goals respectively. FGP makes use of hierarchical decomposition of the flexible planning graph during plan synthesis, based on the success of the same approach employed in GP-*rr*DCSP.

Using the decomposition approach, it is shown how flexible plan synthesis can be cast as the solution of a sequence of linked fuzzy *rr*DFCSPs. The Flexible Local Changes algorithm defined in Chapter 3 is exploited to solve each sequence.

7.1 Background

The assumption that an intelligent agent possesses complete and correct information has received much attention in the form of contingent and conformant AI planning research. In contingent planning a planner is equipped

with sensing actions that can be used to determine the state of the world during plan execution [3, 72, 125, 131]. In conformant planning a planner is provided with knowledge about the possible states the world can be in at the start of execution and the possible outcomes of executing each action [15, 91, 151]. These approaches are complementary and their integration has been explored [40, 73].

Flexible planning is distinct from but complementary to contingent and conformant techniques. With respect to contingent approaches, FGP is not furnished with information gathering actions. However, the flexible propositions used complement the natural imprecision in real-world sensing actions as has successfully been exploited in the area of fuzzy control [124].

Conformant planners model uncertainty as a distribution over the effects of a given operator yet the propositions themselves are exclusively true or false. In a flexible planning problem, propositions are assigned subjective truth degrees and operators map deterministically from the space of flexible preconditions onto a set of flexible effects and a satisfaction degree. Therefore, subjective knowledge can be utilised to entail inferences over preferred combinations of actions.

Recently, probability-based planners have attracted significant attention in the literature. It might be conjectured that a probabilistic formalism could be overloaded so that probabilities attached to effects would signify operator satisfaction degrees. Invoking a probabilistic planner on such an overloaded domain description highlights the different semantics of satisfaction degrees and probabilities. For example, PGraphplan [15] produces a contingent plan that recommends repeated application of an operator in the event of it not producing the desired effects on its first application. This is not correct in the flexible context as repeating an action will not affect its satisfaction degree. While a probabilistic planner will not correctly solve flexible planning problems, the approaches are complementary and could be combined. An investigation into this remains as an important piece of future work.

Also recently, applied planning research explored a mixed-initiative paradigm where the user is supported in exploring the range of plans that can be used to achieve a task. In TRIPS [49] and O-Plan [160], the user is expected to interact extensively with the planner to explore the solution space. However, a user may not understand the solution space being explored and vital solution areas may be missed. Flexible Graphplan is in line with the developments of the the SIPE planner [176], reported in [120], where the system automatically produces a number of courses of action. The important difference is that FGP can automatically find a range of plans given a flexible planning problem using a single run, as opposed to the multiple runs required by SIPE.

There has also been work [84, 168] on incorporating numerically weighted constraints into planning to give a quantitative means of differentiating between different potential plans, as opposed to the qualitative methods in this chapter. However, should there be knowledge available regarding the

truth and/or satisfaction degrees in actual numerical terms, a flexible planning problem can be readily adapted to make use of such knowledge without changing the underlying representation mechanism.

7.2 Flexible Planning Problems

Many real-world planning problems present the need for soft constraints. Reconsider the example problem given in Figure 5.2. The preconditions of the `Load-truck` action state that (i) the truck and (implicitly) valuable package must be co-located, and (ii) that a guard must be present. While precondition (i) is imperative, precondition (ii) is a preference or soft constraint and can be relaxed with an associated damage to the resultant plan. The *flexible planning problem*, described below, adds the ability to reason about the relaxation of soft constraints and the resultant damage to the plan synthesised to the Graphplan framework. The following section then describes Flexible Graphplan, a solution technique capable of synthesising flexible plans given an input flexible planning problem.

7.2.1 Representation

A flexible planning problem, Ψ , consists of a 4-tuple, $\langle \Phi, O, I, \Gamma \rangle$, as follows:

- Φ : A set of plan objects.
- O : A set of flexible operators.
- I : A set of initial conditions consisting of flexible propositions.
- Γ : A set of goal conditions.

Boolean propositions are herein replaced by flexible propositions, ρ , of the form $(\rho \phi_1, \phi_2, \dots, \phi_j k_i)$, where $\phi_i \in \Phi$ and k_i is an element of a totally ordered set, K , which denotes the subjective degree of truth of the proposition. K is composed of a finite number of membership degrees, $k_\perp, k_1, \dots, k_\top$. The original Boolean proposition type is captured at the end points of K , with $k_\perp \in K$ and $k_\top \in K$ indicating total falsehood and total truth respectively. For brevity, when dealing with propositions which only ever take a truth value of $k_\perp$ or $k_\top$, the Boolean style of $\neg(\rho \phi_1, \phi_2, \dots, \phi_j)$ and $(\rho \phi_1, \phi_2, \dots, \phi_j)$ is adopted respectively.

A flexible proposition can easily be described by a *fuzzy relation* [124], R , which is defined by a membership function $\mu_R(.) : \Phi_1 \times \Phi_2 \times \dots \times \Phi_j \rightarrow K$, where $\Phi_1 \times \Phi_2 \times \dots \times \Phi_j$ is the Cartesian product of the subsets of Φ allowable at this place in the proposition. For instance, flexible proposition (*valuable* $?pkg_i k_j$) presented in Figure 7.1, where $?pkg_i$ is instantiated to a package in a logistics domain, is represented by a discrete unary fuzzy relation which maps each package to an associated truth degree in K .

Fig. 7.1. Discrete Unary Fuzzy Relation Underlying the Flexible Proposition (*valuable ?pkg_i k_j*)

A flexible operator, $o \in O$, must recognise *how well* its preconditions are satisfied. Flexible operators are described by fuzzy relations which map from the precondition space onto a totally ordered *satisfaction scale*, L (the same scale used by fuzzy constraint satisfaction, see Section 3.3.1) and a set of flexible effects propositions. L is distinct from K in that it represents the degree of compromise required to apply an operator in the current situation as opposed to the truth of a particular proposition.

It is worth re-iterating that, as per K , L is composed of a finite number of membership degrees, $l_{\perp}, l_1, \dots, l_{\top}$. The endpoints, $l_{\perp} \in L$ and $l_{\top} \in L$ respectively denote a complete lack of satisfaction (in which case the operator is not added to the flexible planning graph based on these preconditions) and complete satisfaction. The Noop action is a special case which has a satisfaction of $l_{\top}$.

A flexible operator (see Figure 7.2) consists of a set of disjoint conditional clauses, Σ (similar to conditional effects [123]). Each $\sigma \in \Sigma$ is a triple $\langle \Theta, E, k_i \rangle$ respectively denoting a conjunction of flexible preconditions, a conjunction of flexible effect propositions and the satisfaction degree of this operator given these preconditions. Each $\theta \in \Theta$ has the form $(\rho \phi_1, \phi_2, \dots, \phi_j \tau \kappa)$, where τ is a precondition operator with argument set κ , as defined in Table 7.1.

```
(operator o
  (params param1, param2, ...)
   $\sigma_i$ : {when (preconds  $\theta_{i_1} \theta_{i_2} \dots$ )
          (effects  $\rho_{i_1} \rho_{i_2} \dots$ )
          (satisfaction  $l_i$ )}
   $\sigma_j$ : {when (preconds  $\theta_{j_1} \theta_{j_2} \dots$ )
          (effects  $\rho_{j_1} \rho_{j_2} \dots$ )
          (satisfaction  $l_j$ )}
  etc)
```

Fig. 7.2. General Format of a Flexible Operator

Precondition Operator, τ	Arguments, κ
$\equiv$	k_i
$<$	k_i
$>$	k_i
<i>range</i>	$k_i \ k_j$
<i>discrete</i>	$k_a \ k_b \ \dots$

Table 7.1. Operators Used to Define Flexible Preconditions and Goals

Each σ_i maps a subset of the space of preconditions to a particular set of effects and a satisfaction degree in L . Again, when dealing with preconditions which only ever take a truth value of $k_\perp$ or $k_\top$, the Boolean style of $\neg(\rho \ \phi_1, \phi_2, \dots, \phi_j)$ and $(\rho \ \phi_1, \phi_2, \dots, \phi_j)$ is adopted for θ respectively.

A flexible plan goal $\gamma \in \Gamma$ maps from the space of flexible propositions to L . Each goal is defined using a number of clauses, as presented in Figure 7.3. The preconditions are defined exactly as those used in the flexible operators. More than one set of mutually-consistent propositions may exist which satisfy the plan goals to some extent. Hence, the satisfaction degree of the plan as a whole must take into account the goal satisfaction degrees as well as those of the flexible operators.

The satisfaction degree of a flexible plan can be determined by the conjunctive combination of the satisfaction degrees of each operator and each goal used in the plan. The conjunctive combination of two fuzzy relations, $R_i \otimes R_j$, is usually interpreted as the minimum membership value assigned by either relation. The *min* operator has the desirable property of idempotency ($\forall u, u \oplus u = u$) which enables classical CSP k -consistency enforcing techniques to be straightforwardly extended to support flexible CSP [42] and is central to the computational efficiency of the FGP solver.

The disadvantage of using an idempotent operator is the so-called *drowning effect* [141], where a low satisfaction degree resulting from one assignment ‘drowns’ several others whose satisfaction degrees, whether optimal or sub-optimal, are not reflected in the overall satisfaction degree. This problem can, however, be counteracted to an extent by exploring the path of highest satisfaction first, as demonstrated in Chapter 8.

```

(goal  $\gamma$ 
  {when  $\theta_i$  (satisfaction  $l_i$ )}
  {when  $\theta_j$  (satisfaction  $l_j$ )}
  etc)

```

Fig. 7.3. General Format of a Flexible Goal

Fig. 7.4. An Example Problem (Flexible Version)

The *quality* of a plan is its satisfaction degree combined with its length, where the shorter of two plans with equivalent satisfaction degrees is better. Hence, plans with a satisfaction degree below $l_{\top}$ should not be viewed simply as degraded versions of a ‘perfect’ plan, but as compromises which trade the satisfaction of flexible operators and goals against a (potentially significant) reduction in plan length. Indeed, by introducing flexibility it may be possible to find a compromise plan where no plan at all exists in the Boolean case because the problem is over-constrained.

7.2.2 The Valuable Package Problem (Continued)

A flexible version of the example used throughout Chapter 5 is now presented to illustrate the operation of FGP (see Figure 7.4). The basic structure of the problem remains the same in terms of the connections between the three cities and the initial placement of plan objects, but flexibility is added to the plan operators and goals to allow FGP to consider compromise solutions. For simplicity, the following definitions are used for K and L : $K = \{k_{\perp}, k_1, k_2, k_{\top}\}$, $L = \{l_{\perp}, l_1, l_2, l_{\top}\}$. In general, K and L do not have to have the same cardinality.

Figure 7.5 describes the flexible operator **Load-truck** which considers the value of the package (in this problem case the propositions ‘valuable pkg_1 k_2 ’ and ‘valuable pkg_2 k_1 ’ are asserted) and the presence of a guard. If the package is not very valuable as is the case for pkg_2 , the guard’s presence is immaterial - a satisfaction degree of $l_{\top}$ is assigned. If the package is quite valuable, as is the case for pkg_1 , loading it onto the truck without a guard damages the plan, resulting in a lower satisfaction degree, l_2 .

Figure 7.6 presents the flexible **Drive-truck** operator that expresses the damage done to a plan by sending a truck across the dangerous track, while indicating that using the major roads has no damaging effect on the plan. The other operators used are the same as for the original problem, using inflexible preconditions and assigning a satisfaction degree of $l_{\top}$.

The plan goals, given in Figure 7.7, remain to transport both packages to c_3 . However, since pkg_2 is not very valuable it has a lower priority and one

```

(operator LOAD-TRUCK
  (params (?t truck) (?p package) (?l location)
    (?g guard))

  {when (preconds (at ?t ?l) (at ?p ?l) (valuable ?p <= k1))
    (effects (on ?p ?t) (not (at ?p ?l)))
    (satisfaction l⊥)}}

  {when (preconds (at ?t ?l) (at ?p ?l) (on ?g ?t)
    (valuable ?p >= k2))
    (effects (on ?p ?t) (not (at ?p ?l)))
    (satisfaction l⊥)}}

  {when (preconds (at ?t ?l) (at ?p ?l) (not (on ?g ?t))
    (valuable ?p >= k2))
    (effects (on ?p ?t) (not (at ?p ?l)))
    (satisfaction l2)}}

```

Fig. 7.5. The Flexible Operator Load-Truck

```

(operator DRIVE
  (params (?v vehicle) (?o location) (?d location)
    (?r1 major-road) (?r2 track))

  {when (preconds (at ?v ?o) (connects ?r1 ?o ?d))
    (effects (not (at ?v ?o)) (at ?v ?d))
    (satisfaction l⊥)}}

  {when (preconds (at ?v ?o) (connects ?r2 ?o ?d))
    (effects (not (at ?v ?o)) (at ?v ?d))
    (satisfaction l1)}}

```

Fig. 7.6. The Flexible Operator Drive

possible compromise is to deliver only pkg_1 , although this does considerable damage to the plan satisfaction degree.

7.3 Flexible Graph Expansion

Having established the definition of a flexible planning problem, this section begins the definition of the Flexible Graphplan flexible plan synthesis technique. As it is based on Graphplan, FGP operates in the two interleaved

```

(goal pkg1-destination
  (when (at pkg1 c3) l⊥))

(goal pkg2-destination
  (when (at pkg2 c2) l1)
  (when (at pkg2 c3) l⊥))

```

Fig. 7.7. Flexible Goals for the Valuable Package Problem

phases of graph expansion and plan synthesis. The former phase is described initially below.

7.3.1 Mutual Exclusivity

In order that mutual exclusion relations can be added to the flexible planning graph, mutual exclusivity must first be defined in the context of a flexible planning problem. Two flexible propositions are labelled as exclusive if either they express a different truth degree for the same proposition or (as per Graphplan) all ways of creating one are exclusive of all ways of creating the other. Mutual exclusion constraints between action nodes express that no valid plan could contain both actions. Two flexible actions in $actions_i$ are mutually exclusive for three reasons, as in the Graphplan framework:

Inconsistent Effects An effect of one action in $propositions_i$ expresses a different truth degree concerning the same proposition from the effect of the other action in $propositions_i$.

Interference An effect of one action in $propositions_i$ expresses a different truth degree concerning the same proposition from a necessary precondition of the other action in $propositions_{i-1}$.

Competing Needs The actions have mutually exclusive preconditions.

7.3.2 Basic Flexible Graph Expansion

Initial conditions are placed in the first proposition level of the graph ($level_0$). A generic action level is generated as follows. Each clause of each flexible operator is instantiated in all possible ways to propositions of the previous level. Provided that all preconditions are mutually consistent, an action instance with the associated satisfaction is added to the planning graph for each such instantiation. Noop actions are added in the same manner as for Graphplan. Mutual exclusion relations are added to the planning graph according to the conditions defined above.

Fig. 7.8. Flexible Planning Graph $Levels_{1,2}$ for the Flexible Version of the Problem given in Figure 5.2. For clarity, mutual exclusion relations are given only in $actions_1$. Note that all pairs of non-Noop nodes in $actions_2$ except $\{G-T\ l_{\top}, L-T\ l_{\top}\}$ are mutually exclusive.

For the example problem, $level_1$ and $level_2$ of the flexible planning graph are presented in Figure 7.8. This graph is relatively complex compared to the Boolean planning graph given in Figure 5.4 since it includes the compromise options afforded by the flexible operators (see Figures 7.5 and 7.6). The extra complexity is, however, offset by the fact that a first ‘compromise’ plan can be extracted as early as $level_3$, as detailed in Section 7.4.4. By contrast, recall that an optimal plan synthesised for the original Boolean version of the problem contains eight steps.

7.3.3 Limited Graph Expansion

Clearly, flexible graph expansion has an increased cost compared with Boolean graph expansion. However, the cost of graph expansion can be reduced by imposing limits over the expansion process. Consider the case where a plan of satisfaction degree l_α has been found. If $l_\alpha < l_\top$ FGP searches onwards in order to look for a plan with a higher satisfaction degree. Yet, it can be seen that there is no point in instantiating flexible operator clauses with a satisfaction degree less or equal to l_α : a plan with this satisfaction degree has been found already - a longer plan with the same satisfaction degree is deemed to be of a lower quality.

The conjunctive combination rule implemented via the *min* operator ensures that no plan of satisfaction degree l_b can contain an action of satisfaction l_a , where $l_a < l_b$. Hence, the completeness of the search is not affected by omitting such actions in future flexible planning graph levels.

7.3.4 Satisfaction Propagation During Graph Expansion

A further step which can considerably reduce the search effort required during plan extraction is to propagate satisfaction degrees forwards as the graph is expanded, again exploiting the fact that conjunctive combination is defined using the *min* operator. While expanding the graph to $level_i$, keep track of the maximum satisfaction degree of all the actions that produce each node in $propositions_i$. Then, when expanding the graph to $level_{i+1}$ the satisfaction degree of a node in $actions_{i+1}$ is the minimum of the satisfaction degree generated by the instantiated operator clause and the minimum of maximum satisfaction degrees attached to each of its preconditions.

This is possible because an activity constraint specifies that an action must be supported by all of its preconditions and the conjunctive combination rule guarantees that the overall satisfaction degree of an action plus its supported preconditions is the minimum satisfaction degree over all the actions involved.

Search during plan extraction is thus reduced because it will be obvious at an earlier stage (since extraction starts at the highest current level) that, say, a certain action can never be part of a plan with satisfaction degree $l_\top$.

This process may be seen in Figure 7.8: the `Unload-truck` action requires ‘on pkg_1 $truck_1$ ’ as a precondition. The sole way of asserting this proposition at $level_1$ is via a `Load-truck` action which has a satisfaction degree of l_2 . The other precondition of `Unload-truck`, ‘at $truck_1$ c_1 ’, is an initial condition, so the minimum of maximum satisfaction degrees attached to the preconditions of this action is l_2 . Hence, the satisfaction degree of `Unload-truck` at $level_2$ is the minimum of l_2 and the satisfaction degree of the instantiated operator clause ($l_{\top}$), resulting in a satisfaction degree of l_2 as seen in the figure. Similarly, the satisfaction degrees attached to several of the `Noop` nodes in $actions_2$ are revised and a satisfaction degree of l_1 is assigned to the `Drive-truck` action which moves the truck from c_3 to c_2 .

7.4 Flexible Plan Extraction via *rr*DFCSP

This section describes the component of FGP responsible for extracting a range of plans from a flexible planning graph as it is extended. As compared with the Boolean case described in Chapter 5, this process is complicated by the satisfaction degrees attached to flexible operators and goals. When making use of a hierarchical synthesis technique similar to that of GP-*rr*DCSP, the former requires the solution of dynamic flexible sub-problems and the latter requires one extraction phase per subset of propositions which satisfy the goals to a certain extent. Following the high level description below, the full FGP algorithm is described in detail in Section 7.5.

7.4.1 The FCSP Viewpoint

When examining the flexible planning graph with a view to plan synthesis via constraint satisfaction, it can be seen that many of the elements remain the same as in the Boolean case described in Chapter 5. A node in $propositions_i$ is still regarded as a CSP variable, and the set of nodes in $actions_i$ which assert this proposition as an effect still constitute the corresponding domain. However, the additional satisfaction degree information present in the flexible planning graph means that it is not possible to view flexible plan synthesis as a classical constraint satisfaction problem.

The satisfaction degrees associated with the domain elements of a particular variable are used to construct a unary *preference* constraint (see Section 2.9) which specifies a preference degree for each of the assignments that satisfy it. The preference degree assigned to each domain element corresponds exactly to the satisfaction degree of the associated node in the planning graph. As an example, consider ‘at $truck_1$ c_1 ’ in $propositions_2$ (see Figure 7.8) with a domain comprising two `Drive-truck` actions and a `Noop` action. A unary preference constraint is constructed specifying that the assignment of one of the `Drive-truck` actions and the `Noop` action have satisfaction degree

$l_{\top}$, while the assignment of the other Drive-truck action has a satisfaction degree of l_1 .

Boolean binary constraints are generated from the mutual-exclusion relations in the planning graph in exactly the same manner as for Graphplan and GP-*rr*DCSP. A restricted type of fuzzy CSP is therefore formed containing Boolean binary constraints and unary preference constraints. Dynamic sequences of these problems, as generated by the hierarchical decomposition process described below can be solved by either the branch and bound or FLC algorithms defined in Chapter 3. The FLC algorithm is used in the implemented version for the same reason that LC was used within GP-*rr*DCSP (see Section 5.4.1): its superior ability to maintain solution stability as demonstrated in Chapter 4.

7.4.2 The Hierarchical Approach Revisited

The decomposition of the flexible planning graph is essentially the same as that used by GP-*rr*DCSP. A hierarchy of *rr*DFCSPs is used to support the synthesis process, with each *rr*DFCSP corresponding to a level in the planning graph.

Consider the current level, $level_g$, from which plan synthesis is about to start. In the Boolean case, the required goal propositions either exist and are mutually consistent at this level (extraction proceeds) or they do not (the graph is expanded a further level). In the flexible case, however, there may be several sets of mutually-consistent propositions that satisfy the goals to some extent. An attempt is made to identify a sub-graph connecting each set that conjunctively satisfies the goal propositions with a satisfaction degree above l_{α} back to the initial conditions. In order to discover the best plan of length g first, this search is performed in descending order of satisfaction degree (see Figure 7.9).

The *rr*DFCSP representing each level of the planning graph is further subdivided into sub-sequences, as per GP-*rr*DCSP. The mechanism for this subdivision is also the same: a solution to a parent sub-problem at level i defines the initial sub-problem in a sub-sequence at level $i-1$. This new sub-problem is also solved, and the memoset information returned as a result is used to *a*) add new constraints to it, forming the next sub-problem instance in the sub-sequence, and *b*) contribute to the construction of memoset information for the sub-sequence. The difference is that memoset information is constructed and propagated for each satisfaction degree above the current value of l_{α} . The construction of local memosets in the flexible context is discussed in detail in the next section.

Figure 7.9 describes the plan synthesis process at the sub-sequence level of description, and Figure 7.10 shows a generic sub-sequence in detail. The FLC algorithm always returns the best possible solution (in terms of satisfaction degree) to a sub-problem, leading to a satisfaction degree-‘banded’ exploration of a sub-sequence. This process maintains completeness and helps

Fig. 7.9. FGP Hierarchical Plan Synthesis: Flexible Sub-problem Sub-sequence Level

to identify plans with the highest satisfaction degree first. Only when a sub-problem instance is constructed which admits no solution above l_α does the sub-sequence terminate and will the memoset information be returned to the level above for propagation.

7.4.3 Memoisation for Flexible Plan Synthesis

The increased complexity of flexible plan synthesis, as compared with the original Boolean type, makes the discovery and propagation of memoset information even more important. A memoset was defined as a set of nodes which are conjunctively unsupportable. In the flexible case, this definition must be expanded to include a satisfaction degree. An l_α -memoset for level _{i} contains proposition nodes which cannot form part of a plan with a satisfaction degree of l_α or above at level _{i} .

Fig. 7.10. A Flexible Sub-problem Sub-sequence in Detail

Memosets are recorded for each satisfaction degree (excluding $l_\perp$) at each level of the planning graph. During plan synthesis, the recorded memosets for the current level are examined at the start of each new sub-sequence to determine whether the current search branch can be pruned. This is the case if an l_a -memosets exist for each satisfaction degree greater than l_a .

The lowest satisfaction degree at which memosets do exist for the variables in the initial problem in a sub-sequence is denoted by l_m . As the solution of each sub-problem in a sub-sequence progresses, further memosets are constructed for the satisfaction degrees l_a , where $l_\alpha < l_a < l_m$. This is achieved through either local memoset discovery, as described below, or through propagated memoset discovery. The latter is essentially the same process as in the Boolean case, with an l_a -memoset propagated from $level_{i-1}$ contributing to the construction of an l_a -memoset at $level_i$. Hence, when the current sub-sequence is terminated, there will be a memoset for each satisfaction degree above the current value of l_α . These memosets are propagated to the level above.

Local l_a -Memoset Discovery via Sub-problem Pre-processing

Local memosets, as per GP-rrDCSP, are discovered during either pre-processing or search of the initial problem in a sub-sequence. Fuzzy arc consistency, as described in Section 3.6 is first established. If the result of this process is an upper bound on the satisfaction degree of the sub-problem that is below l_m , information recording during the arc consistency enforcing process can again be used to construct memoset explanations.

An *eliminating explanation* was used in Chapter 5 to trace the reason why a particular domain element was removed from consideration. In order to construct memosets for different satisfaction degrees, this must be replaced by a *reduction explanation* that records the reason why the unary satisfaction degree of a domain element was revised below a particular satisfaction degree. As per the Boolean eliminating explanations, reduction explanations are specified in terms of responsible variables. Hence, a reduction explanation for l_a of x_r for domain element d_i records the fact that x_r is responsible for the revision of the unary consistency of d_i below l_a . This situation occurs when there are no elements in D_r that are compatible with d_i and have a satisfaction degree of at least l_a .

An example is given in Figure 7.11. Figure 7.11a describes the initial state of a sub-problem, before any constraint propagation is applied. Figure 7.11b presents the state of the problem following the revision of $arc(2, 1)$ and $arc(2, 3)$. Consider domain elements d_{2-1} and d_{2-2} : both are compatible only with elements in D_1 with unary satisfaction degrees of a maximum of l_2 . The unary satisfaction degree of d_{2-1} and d_{2-2} can therefore be reduced to l_2 . The reduction explanation for each element records that x_1 is responsible for this reduction. Similarly, the unary consistency of d_{2-3} is reduced to l_1 due to the constraint with x_3 .

Fig. 7.11. Constructing Memosets by Tracing Reduction Explanations

At this point, the maximum satisfaction degree of any element in D_2 is l_2 . Hence, the satisfaction degree of the problem is also reduced to l_2 , since some element in D_2 *must* be assigned to x_2 in any solution. Making use of the information recorded in the reduction explanations, a memoset can be constructed containing the variables which conjunctively do not admit a solution with a satisfaction degree of $l_{\perp}$. As per the Boolean case described in Chapter 5, this is done by recursively tracing the reduction explanation for each element of D_2 as described in Figure 7.12.

```

01. Procedure constructMemoSetla( $x_i, l_a$ )
02.    $trace \leftarrow \{D_i\}$ 
03.    $memoSet_{l_a} \leftarrow \{x_i\}$ 
04.   While  $trace \neq \emptyset$ 
05.     Select and Remove  $D_t \in trace$ 
06.     Foreach  $d_{t-f} \in D_t$  where  $\mathbf{satDegree}(d_{t-f}) < l_a$ 
07.        $x_r$  is the reduction explanation for  $d_{t-f}$ 
08.       If  $x_r \notin memoSet_{l_a}$  And  $D_r \notin trace$ 
09.         Add  $x_r$  To  $memoSet_{l_a}$ 
10.         Add  $D_r$  To  $trace$ 
11.   End Procedure

```

Fig. 7.12. Memoset Construction Following Reduction of Unary Consistency Degree of all Elements of D_i below l_a

Local l_a -Memoset Discovery via Search

As noted in Section 5.4.2, enforcing arc consistency is not guaranteed to detect an inconsistent problem. Equivalently, enforcing fuzzy arc consistency is not guaranteed to detect that a problem does not contain a solution of satisfaction degree l_a . Hence, FGP also requires a method for constructing a memoset while solving the initial sub-problem in a sub-sequence.

The method adopted is a generalisation of that used for GP-rrDCSP. The current search branch cannot lead to a solution of satisfaction degree l_a when all elements of D_i that are compatible with assignments to existing variables have a unary satisfaction degree below l_a . In this case, the currently assigned variables form a conflict set with x_i with respect to l_a . As per the Boolean case, this inconsistency can be recorded as a partial memoset for the satisfaction degree l_a . Irrelevant elements, i.e. those demonstrably not responsible for the conflict are removed from the conflict set. Given the fact that flexibility is present only in the unary preference constraints associated with each variable, irrelevant variables are simply those with assignments with satisfaction degrees of l_a or above.

7.4.4 The Valuable Package Problem - Synthesised Plans

This section describes the three possible plans for the example given in Figure 5.2. The first short plan is as follows, with the major compromises being the route across the unsafe track and the fact that only pkg_1 is delivered.

1. Load-truck pkg_1 truck₁ (l_2)
2. Drive-truck truck₁ c_1 to c_3 via r_3 (l_1)
3. Unload-truck pkg_1 truck₁ ($l_\top$)

Using the main roads to avoid the unsafe track, both packages may be carried to c_3 in 5 steps as follows, with a satisfaction degree of l_2 since the guard is not present when pkg_1 is loaded.

1. Load-truck pkg_1 truck₁ (l_2)
2. Drive-truck truck₁ c_1 to c_2 via r_1 ($l_\top$)
3. Load-truck pkg_2 truck₁ ($l_\top$)
4. Drive-truck truck₁ c_2 to c_3 via r_2 ($l_\top$)
5. – Unload-truck pkg_1 truck₁ ($l_\top$)
– Unload-truck pkg_2 truck₁ ($l_\top$)

The above plan is still not regarded as having the highest satisfaction degree because the guard was not present when the first package was loaded. The ‘no compromise’ plan contains seven steps and is described below.

1. Drive-truck truck₁ c_1 to c_2 via r_1 ($l_\top$)
2. – Guard-gets-on-truck truck₁ guard₁ ($l_\top$)
– Load-truck pkg_2 truck₁ ($l_\top$)
3. Drive-truck truck₁ c_2 to c_1 via r_1 ($l_\top$)

4. **Load-truck** $truck_1$ $package_1$ $guard_1$ ($l_{\top}$)
5. **Drive-truck** $truck_1$ c_1 to c_2 via r_1 ($l_{\top}$)
6. **Drive-truck** $truck_1$ c_2 to c_3 via r_2 ($l_{\top}$)
7. – **Unload-truck** pkg_1 $truck_1$ ($l_{\top}$)
- **Unload-truck** pkg_2 $truck_1$ ($l_{\top}$)

This simple example demonstrates how a range of plans of different lengths and containing alternative compromises can be synthesised from a given flexible planning problem. The user can then select the one which is deemed to offer the best compromise between length and satisfaction degree.

7.5 The FGP Algorithm

The FGP algorithm presented here is based upon the GP-*rr*DCSP algorithm given in Chapter 5, which may be seen as a special case of FGP. Hence, FGP can also solve Boolean problems by restricting the truth and satisfaction scales to $\{k_{\perp}, k_{\top}\}$ and $\{l_{\perp}, l_{\top}\}$ respectively. A proof of the soundness and completeness of this algorithm is in Appendix B.

7.5.1 The Top-level Procedure

Figure 7.13 presents the top-level procedure $\text{FGP}()$, which is responsible for interleaving the graph expansion and solution extraction phases. The satisfaction degree of the best plan found so far ($plan_{\alpha}$) is stored in l_{α} (a global variable), which is initialised to $l_{\perp}$ (line 2). As per GP-*rr*DCSP, the initial level of the planning graph is first constructed (line 3) from the flexible planning problem description. The main loop (lines 4-9) then repeatedly attempts to extract plans with satisfaction degrees above l_{α} , extending the graph until a plan with satisfaction degree $l_{\top}$ is found.

Following the construction of each new level, the procedure performs a satisfaction degree-banded extraction process (lines 5-8). At each band, l_g ,

```

01. Procedure  $\text{FGP}(\Psi)$ 
02.    $l_{\alpha} \leftarrow l_{\perp}$ 
03.    $level_g \leftarrow \text{graph-initialise}(\Psi)$ 
04.   While Not  $\text{compromise-free}(plan_{\alpha})$ 
05.     Foreach  $l_g \in L$ ,  $l_g > l_{\alpha}$  (descending order)
06.       Foreach  $goalSet \subseteq level_g$ ,  $\text{satDegree}(goalSet) = l_g$ 
07.          $\text{FExtract}(level_g, goalSet, l_g)$ 
08.         If  $l_{\alpha} = l_g$  Break
09.        $level_g \leftarrow \text{extend}(level_g, \Psi)$ 
10.   Return  $plan_{\alpha}$ 
11. End Procedure

```

Fig. 7.13. Pseudo-code for $\text{FGP}()$

```

01. Procedure FExtract( $level_i, subGoals_i, l_g$ )
02.   If  $level_i = 0$ 
03.     If matchInitConds( $subGoals_i$ )
04.        $plan_\alpha \leftarrow plan_\beta, l_\alpha \leftarrow getSatDegree(plan_\alpha)$ 
05.       Return  $\emptyset$ 
06.     Else Return  $\{\{subGoals_i\}\}$ 
07.    $memoSets_{i,L} \leftarrow \emptyset$ 
08.    $l_m \leftarrow checkMemoSets(subGoals_i, level_i, memoSets_{i,L})$ 
09.   If  $l_m \leq l_\alpha$  Return  $memoSets_{i,L}$ 
10.    $subAssTuples_{i,L} \leftarrow \emptyset$ 
11.    $probInstance \leftarrow constructProblemInstance(level_i, subGoals_i)$ 
12.    $solution \leftarrow ACFLCFC(probInstance, memoSets_{i,L}, l_u)$ 
13.   While  $satDegree(solution) > l_\alpha$  And  $l_g > l_\alpha$ 
14.      $subGoals_{i-1} \leftarrow getPreconditions(solution)$ 
15.      $memoSets_{i-1,L} \leftarrow FExtract(level_{i-1}, subGoals_{i-1}, l_g)$ 
16.      $getSubAssTuples(memoSets_{i-1,L}, solution, subAssTuples_{i,L}, \min(l_u, l_m))$ 
17.      $propagateLowest(subAssTuples_{i,L}, solution, probInstance)$ 
18.      $solution \leftarrow ACFLCFC(probInstance, memoSets_{i,L})$ 
19.    $memoSets_{i,L} \leftarrow genMemoSets(subAssTuples_{i,L}, \min(l_u, l_m), probInstance)$ 
20.    $record(memoSets_{i,L}, l_m)$ 
21.   Return  $memoSets_{i,L}$ 
22. End Procedure

```

Fig. 7.14. Pseudo-code for FExtract()

an extraction phase is instigated for every mutually-consistent combination of propositions with a satisfaction degree of l_g with respect to the flexible goals (lines 6-7). Note that, due to the use of the *min* operator, a plan with an overall lower satisfaction degree can be found even if the goals themselves are satisfied to a high satisfaction degree.

If a plan was found during an extraction phase, l_α will have been updated (by the FExtract() procedure below), and the extraction phase will continue, or halt so that the graph can be extended, as appropriate (lines 7-8).

7.5.2 The FExtract() Procedure

Figure 7.14 describes the Fextract() procedure for FGP. As per GP-rrDCSP, this procedure is responsible for coordinating the search for a valid plan. The difference here is that it must search for plans of all non- $l_\perp$ satisfaction degrees. The base case occurs when $level_0$ is reached (lines 2-6). At this point, if the current sub-goals match the initial conditions, a valid plan has been found and $plan_\alpha$ and l_α can be updated (lines 4). Note that $plan_\beta$ is a reference to a global structure where the current partial plan is stored.

The output of this procedure is the set, $memoSets_{i,L}$ of memosets for each satisfaction degree above l_α for which no subgraph exists connecting the sub-goals and initial conditions. The first task when trying to identify support

for a set of sub-goals is to check whether this sub-goal set (or a superset of it) has already been shown to be unsupportable down to a particular satisfaction degree, l_m . Each recorded unsupportable sub-goal set is added to the appropriate satisfaction degree band of $memoSets_{i,L}$ (line 8).

If this sub-goal set has been shown to be unsupportable down to a satisfaction degree lower or equal to that of the current best plan, this branch of the search can be pruned (line 9). Otherwise, the initial sub-problem in the sub-sequence associated with these sub-goals is constructed (line 11).

The ACFLCFC algorithm is used to solve this first instance, establishing l_u , the satisfaction degree down to which this instance is unsolvable (line 12). In addition, local memosets are identified for satisfaction degrees l_a , where $l_u \leq l_a < l_m$. The remaining sub-problems in the sub-sequence are then solved (lines 13-18). The sub-sequence is terminated when the satisfaction degree of either the solution to the current sub-problem or the current flexible goal combination is no better than l_α (line 13).

For all sub-problems in the sub-sequence, an optimal solution is found via the ACFLCFC algorithm (line 18). The preconditions of the action nodes which constitute the solution are identified (line 14) and a recursive call is made to try and support them (line 15). Following this recursive call, memosets returned from $level_{i-1}$ are propagated to $actions_i$ in order to identify and store the corresponding sub-assignment tuples (line 16). This is done for each satisfaction degree, l_a , such that $l_\alpha < l_a < \min(l_u, l_m)$.

In order to maintain completeness, a constraint corresponding to the lowest value of l_a only is posted to create the next sub-problem in the sub-sequence (line 17). The propagation process is essentially the same as for the Boolean case described in Section 5.5.3, concentrating on a particular satisfaction degree.

Once the sub-sequence has terminated, the l_a -memosets, where $l_\alpha < l_a < l_m$ are generated from the stored sub-assignment tuples and recorded (lines 19, 20). All memosets for the satisfaction degrees above l_α are then returned (line 21).

7.5.3 Complexity Issues

The time complexity of the GP-*rr*DCSP algorithm, a special case of FGP, was shown to be $O(g\mathcal{G}p^{g-1}m^3\mathcal{G}p^{g-1})$, where g is the number of levels in the graph between the goals and the initial conditions, $\mathcal{G}$ is the number of plan goals, m is the maximum number of actions that support any single proposition, and p is the maximum number of preconditions specified by any operator. Two factors contribute to the increased complexity of FGP. The first is the flexibility in each of the sub-problems that must be solved by the FLC algorithm, and the second is the use of flexible goal conditions.

As ℓ , the number of satisfaction degrees in the scale L increases, the length of the sub-sequences presented in Figure 7.10 will also tend to increase. This is because solutions of all satisfaction degrees above $l_\perp$ must be explored. Hence,

there is an increase in the number of sub-problems that FLC must solve. This increase is, however, in the average rather than the worst case, since with a constant number of sub-problem variables and domain size (neither of which are affected by ℓ) the maximum number of potential solutions to explore remains the same.

If there are $\mathcal{G}$ flexible goals, the worst case occurs when each can be satisfied in all $\ell - 1$ ways at the current level, producing $\mathcal{G}^{\ell-1}$ goal combinations. Hence, the worst case time complexity of plan synthesis from $level_g$ of a flexible planning graph is:

$$O(g\mathcal{G}^\ell p^{g-1} m^{3\mathcal{G}p^{g-1}})$$

The worst-case space required by a flexible planning graph is no greater than that required by a Boolean planning graph. This may be seen by re-considering the space complexity analysis made in [14]. Consider a Boolean planning problem with n objects, i propositions in the initial conditions, s STRIPS operators each having k formal parameters and an effects list of size e . Since operators cannot create new objects, the number of different propositions that can be created by instantiating an operator is $O(en^k)$. So, the maximum number of nodes in any proposition level is $O(i + sen^k)$. Since any operator can be instantiated in at most $O(n^k)$ different ways, the maximum number of nodes in any action level of the graph is $O(en^k)$. The difference as regards the flexible planning graph is that the operators have several clauses. However, since these clauses must be disjoint (see Section 7.2.1) the number of ways any operator can be instantiated remains as $O(n^k)$.

The worst-case space complexity of FGP as a whole, taking memoset recording into consideration, is exponential. This is due to the fact that an exponential number of sub-problems must be examined in the worst case, each of which can lead to the recording of a memoset. This result reinforces the need for an informed memoset recording mechanism.

7.6 Summary

This chapter introduced the flexible planning problem, a representation designed to capture the inherent ‘softness’ found in many real problems. This is facilitated through the assignment of subjective truth degrees to propositions which are exploited by flexible operators that express one of a range of satisfaction degrees depending on how well their preconditions are satisfied.

This general representation is independent of the technique used to synthesise flexible plans. A solution strategy, called Flexible Graphplan (FGP), which extends the Graphplan framework was established, utilising fuzzy *rr*DFCSP as a foundation for plan synthesis. FGP relies on the construction of a flexible planning graph annotated with truth and satisfaction degrees. Plan synthesis via the hierarchical decomposition of the flexible planning graph was adopted, generalising the GP-*rr*DCSP approach described in

Chapter 5. The Flexible Local Changes algorithm was employed to solve the dynamic flexible sub-problem sequences thus created.

Methods for reducing the added computational complexity of supporting flexible planning were discussed. These comprise the limitation of the addition of flexible action instances to those that could form part of a plan with a higher satisfaction degree than is currently known and the forwards-propagation of satisfaction degrees to aid the early pruning of the search space.

Development of FGP should include a flexible version of propagated irrelevance testing, as described in Section 5.7. Given the added complexity of flexible planning, obtaining the smallest memosets possible is likely to be of even greater importance.

The disadvantage of using the idempotent *min* operator that underpins satisfaction aggregation for the flexible planning problem is the so-called *drowning effect*, where a low satisfaction degree resulting from one assignment ‘drowns’ several others whose satisfaction degrees, whether optimal or sub-optimal, are not reflected in the overall satisfaction degree. Although the drowning effect can be counteracted to an extent by always following the path of highest satisfaction degree first, as demonstrated in the following chapter, it would be preferable to have a finer resolution when comparing different plans (short of using a large number of satisfaction degrees).

A possible means of addressing this problem is Leximin CSP [118]. This approach combines additive and possibilistic techniques, although at a significantly higher computational cost. Making use of such an approach in the context of the flexible planning problem is an important item of future work.

8 FGP: Experimental Results

This chapter presents an evaluation of Flexible Graphplan made in two ways. Firstly, a test suite of twelve problems is examined, engineered to contain plans of three different satisfaction degrees at different distributions of plan length. The results for this test allow an examination of the relationship between solution distribution and difficulty of plan synthesis. Boolean versions of the twelve problems are also solved in order to investigate the overhead incurred by searching for a range of plans. In addition, the utility of the limited graph expansion and satisfaction propagation techniques described in Sections 7.3.3 and 7.3.4 respectively are examined with respect to the test suite.

The second part of the chapter describes a new flexible planning problem benchmark: the Rescue problem, which entails the evacuation of a small island. The Rescue problem is designed to exhibit complex interactions between a range of flexible operators and goals, testing the ability of FGP to synthesise a range of plans, from short ‘compromise’ solutions to substantially longer plans that avoid violating any constraints.

8.1 The Test Suite

The first set of experiments is based on a test suite of twelve problems, which can be examined in Appendix D. These are logistics-style problems similar in type to the worked example presented in Chapters 5 and 7. As an example, Figure 8.1 gives an overview of the second problem in the test suite. The target is to transport both packages to their respective destinations. The other problems in the test suite are similar, but with more/less complex internal connections in the three cities.

In order to compare the effort of searching for a range of solutions, as compared with synthesising a single plan from a Boolean version of the same problem, two sets of operators are used when solving these problems. The first set are flexible operators (see Section D.1.1) and allow compromises to be made in order to synthesise one or more shorter plans as well as a plan containing no compromises.

The truth and satisfaction degree scales used are $K = \{k_{\perp}, k_1, k_2, k_{\top}\}$ and $L = \{l_{\perp}, l_1, l_2, l_{\top}\}$ respectively. Hence, a maximum of three different plans

Fig. 8.1. Flexible Problem 2 (of 12) for Comparing Boolean/Flexible Extraction

can be synthesised for each problem, one per satisfaction degree. Indeed, the internal structure of the 12 problems has been constructed such that it is always possible to synthesise three different plans with lengths as presented in Figure 8.2.

The flexibility in this case comes from two areas. First, it is preferred that the guard is present when a package is loaded onto a truck (if not, the satisfaction degree of the Load-truck action is l_2). Second, it is preferred not to transport the valuable packages through the dangerous part of town (if this preference is ignored, the satisfaction degree of the Drive-truck action drops to l_1).

Fig. 8.2. Test Suite of Flexible Problems: Composition

Fig. 8.3. Test Suite of Problems: Cumulative Run-times Utilising Both Flexible and Boolean Operators

The second set of operators (see Section D.1.1 also) are essentially Boolean in nature. They make imperative the two preferences described for the *Load-truck* and *Drive-truck* actions above, using just the endpoints of K and L . This effectively creates an equivalent Boolean version of each problem, allowing the synthesis of the shortest plan with satisfaction degree $l_{\top}$ only.

8.2 The Test Suite: Plan Synthesis Results

FGP was first used to solve the twelve problems in the test suite using both the flexible and Boolean operators. The comparative run-times are presented in Figure 8.3, with cumulative results presented for each satisfaction degree. As expected, it always takes longer to find a compromise-free plan when also searching for shorter ‘compromise’ plans than it does to simply solve the Boolean problem. However, the time taken to produce shorter compromise plans is significantly lower than to solve the Boolean problem, providing an attractive ‘anytime’ behaviour.

The difficulty of plan synthesis using the Boolean operators closely mirrors that of using the flexible operators to extract the final $l_{\top}$ plan. Although problems 3 to 12 all have the same length of compromise-free plan, the increasing effort required to solve them is reflected in the increasing difficulty of making compromises to find shorter plans using the flexible operators.

One main contributing factor to the overall difficulty of plan synthesis may be seen in Figure 8.3. That is the amount of time spent by FGP in searching for a plan of each satisfaction degree. As noted in Section 7.3.3,

Fig. 8.4. Test Suite of Problems: Cumulative Run-times Utilising Flexible Operators without Limited Graph Expansion or Satisfaction Propagation

as soon as a plan with satisfaction degree l_i has been found, FGP no longer needs to search for plans nor consider actions of satisfaction degree less or equal to l_i when performing banded solution extraction. This explains why the earlier problems in the test suite, where the l_1 and l_2 plans have short lengths, are easiest to solve. It also explains why problems 10, 11 and 12 are the hardest: FGP spends a lot of time initially looking for plans of all three non- $l_{\perp}$ satisfaction degrees before one of l_1 is discovered.

The difficulty of synthesising just the l_1 plan closely corresponds to the length of the l_1 plan for each problem. Since l_1 is the lowest non- $l_{\perp}$ satisfaction degree, all propositions are asserted by an action with at least satisfaction degree l_1 and all actions in the flexible planning graph may be considered for inclusion in an l_1 plan, resulting in a less complex search process.

The difference in length between the l_1 and l_2 plans seems to be the most influential on the amount of overall search effort (see problem 10, for example). If this difference in length is decreased, there is very little increase in overall effort even if the length of the l_1 plan is increased (compare the difficulty of problems 5 and 6, and 10 and 11).

The region of the search where only an l_1 plan has been found is likely to be the most intensive. Prior to the discovery of the l_1 plan, a significant number of levels of the graph may be constructed without any search being required since a mutually-consistent set of propositions matching the plan goals do not yet exist. After the l_2 plan has also been found, the search becomes easier, with FGP able to discard all actions with a satisfaction degree below $l_{\top}$.

Fig. 8.5. Test Suite of Problems: Cumulative Run-times Utilising Flexible Operators and Limited Graph Expansion Only

8.2.1 The Utility of Limited Graph Expansion and Satisfaction Propagation

It is also useful to gauge the efficacy of the limited graph expansion and satisfaction propagation techniques described in Sections 7.3.3 and 7.3.4 respectively. In order to achieve this, results were first obtained (presented in Figure 8.4) on the same test suite problems using a version of FGP which performs neither limited graph expansion nor satisfaction propagation.

The results demonstrate a clear deterioration as compared with the run-times given in Figure 8.3, with the most difficult problem taking almost one third longer than previously. The times to synthesise the l_1 plan are, however, unaffected. Limited graph expansion has no effect until at least the l_1 plan has been found. Furthermore, since l_1 is the lowest non- $l_\perp$ satisfaction degree, satisfaction propagation cannot revise the satisfaction degree of any action below this degree.

The effort required to synthesise the l_2 plans is increased, but not to the same extent as that required to synthesise the l_3 plans. This follows firstly from the fact that the limited graph expansion has more effect, the more satisfaction degrees it can rule out. Hence, following the synthesis of the l_2 plan, only actions with satisfaction degree $l_\top$ are added to the planning graph, increasing the efficiency of graph expansion and sub-problem construction. With limited graph expansion turned off, actions of all satisfaction degrees are continually added to the graph, significantly adding to its size and the cost of processing it into sub-problems.

Fig. 8.6. Test Suite of Problems: Cumulative Run-times Utilising Flexible Operators and Satisfaction Propagation Only

The second reason for the disproportionate increase in time to find the l_T plan is that satisfaction propagation also provides most benefit to search for plans with the highest satisfaction degrees. The revision of action satisfaction degrees is always down the satisfaction scale, excluding them from taking any part in plans with higher satisfaction degrees. When satisfaction propagation is not present, many more actions will have higher satisfaction degrees associated with them, requiring more search to exclude them from the current plan.

In order to determine which technique has the most beneficial effect on the plan synthesis process, the test suite problems were solved twice more, using only limited graph expansion or satisfaction propagation in each case. These results are presented in Figures 8.5 and 8.6 respectively. They demonstrate that, on this test suite, satisfaction propagation is more effective than limited graph expansion in reducing search effort. It is expected that this is generally true: limited graph expansion decreases the size of the flexible planning graph and the time taken both to expand it and construct sub-problems, whereas satisfaction propagation improves the efficiency of the plan extraction process itself, the dominant cost of plan synthesis.

8.3 The Rescue Problem

To demonstrate the utility of flexible planning on a more substantial problem, the example given in Figure 8.7 is used. The scenario considered is a rescue operation: a volatile volcano on Volcano Island has started to erupt.

be employed). The exact details of the problem and domain specification are presented in Appendix D. This is not a trivial problem: there are 432 possible flexible goal combinations alone. FGP is able to synthesise 5 plans for the rescue problem in under 30 seconds, one corresponding to each satisfaction degree, discounting $l_{\perp}$. This is not the case for all problems since, as noted in the previous chapter, given two plans of the same length the one with the higher satisfaction degree is returned. There are, of course, alternative plans with the same satisfaction degree and length as the ones presented, but this is no different from the situation found in Boolean planning.

8.3.1 The Shortest Plan

The shortest plan of four steps is given below. Although the other actions in the plan are effectively ‘drowned’ by the fact that the helicopter must fly to Volcano Village, by exploring the highest quality goals first the effects of this have been counteracted: the helicopter is flown to Far Village when strictly speaking a plan with an equivalent satisfaction degree would be to take the civilians to the Safe Point.

Plan with Satisfaction Degree l_1

1. LOAD-TRUCK lab-truck scientific-equipment ($l_{\top}$)
LOAD-TRUCK lab-truck scientist ($l_{\top}$)
FLY-HELICOPTER Far Village, Volcano Village (l_1)
2. LOAD-HELICOPTER civilian1, civilian2 (l_4)
DRIVE-TRUCK lab-truck Science-lab, Safe Point ($l_{\top}$)
3. UNLOAD-TRUCK lab-truck scientific equipment ($l_{\top}$)
UNLOAD-TRUCK lab-truck scientist ($l_{\top}$)
FLY-HELICOPTER Volcano Village, Far Village ($l_{\top}$)
4. UNLOAD-HELICOPTER civilian1, civilian2 ($l_{\top}$)

8.3.2 A Five Step Plan

A five-step plan with satisfaction degree l_2 can be synthesised as follows. Here, two trucks are used to transport the people and possessions to Safe Point. All the actions used have the highest satisfaction degree, with the major compromise being that the people remain on Volcano island.

Plan with Satisfaction Degree l_2

1. DRIVE-TRUCK v-truck Volcano Beach2, Volcano Village ($l_{\top}$)
LOAD-TRUCK lab-truck scientific equipment ($l_{\top}$)
LOAD-TRUCK lab-truck scientist ($l_{\top}$)
2. LOAD-TRUCK v-truck personal-posessions ($l_{\top}$)
LOAD-TRUCK v-truck civilian1 ($l_{\top}$)
LOAD-TRUCK v-truck civilian2 ($l_{\top}$)
DRIVE-TRUCK lab-truck Science Lab, Safe Point ($l_{\top}$)

3. UNLOAD-TRUCK lab-truck scientific equipment (l_T)
UNLOAD-TRUCK lab-truck scientist (l_T)
DRIVE-TRUCK v-truck Volcano Village, Science lab (l_T)
4. DRIVE-TRUCK v-truck Science Lab, Safe Point (l_T)
5. UNLOAD-TRUCK v-truck civilian1 (l_T)
UNLOAD-TRUCK v-truck civilian2 (l_T)
UNLOAD-TRUCK v-truck personal possessions (l_T)

8.3.3 A Nine Step Plan: The People are Evacuated

In order to evacuate the people from Volcano Island in the shortest number of steps, a nine-step plan with satisfaction degree l_3 may be used as given below. The boat is used to take the scientist and scientific equipment to Neighbour Island, while a helicopter rescues both civilians from Safe Point.

Again, note the avoidance of the drowning effect: the civilians are flown to Far Village, when flying them to Neighbour Village would have produced a plan with an equivalent satisfaction degree. Furthermore, the scientific equipment could have been taken to Safe Point with the civilians' possessions, but is instead transported to Neighbour Village with the scientist.

Plan with Satisfaction Degree l_3

1. LOAD-TRUCK lab-truck scientist1 (l_T)
LOAD-TRUCK lab-truck scientific equipment (l_T)
FLY-HELICOPTER Far Village, Safe Point (l_3)
DRIVE-TRUCK v-truck Volcano Beach2, Volcano Village (l_T)
2. LOAD-TRUCK v-truck personal possessions (l_T)
LOAD-TRUCK v-truck civilian1 (l_T)
LOAD-TRUCK v-truck civilian2 (l_T)
DRIVE-TRUCK lab-truck Science Lab, Volcano Beach 2 (l_T)
3. SAIL-BOAT Neighbour Beach, Volcano Beach2 (l_3)
DRIVE-TRUCK v-truck Volcano-Village, Science Lab (l_T)
UNLOAD-TRUCK lab-truck scientist1 (l_T)
UNLOAD-TRUCK lab-truck scientific equipment (l_T)
4. LOAD-BOAT scientist1 (l_T)
LOAD-BOAT scientific equipment (l_T)
DRIVE-TRUCK v-truck Science Lab, Safe Point (l_T)
5. UNLOAD-TRUCK v-truck civilian1 (l_T)
UNLOAD-TRUCK v-truck civilian2 (l_T)
UNLOAD-TRUCK v-truck personal possessions (l_T)
SAIL-BOAT Volcano Beach2, Neighbour Beach (l_3)
6. LOAD HELICOPTER civilian1, civilian2 (l_3)
UNLOAD-BOAT scientist1 (l_T)
UNLOAD-BOAT scientific equipment (l_T)
DRIVE-TRUCK n-truck Neighbour Village, Neighbour Beach (l_T)
7. LOAD-TRUCK n-truck scientist1 (l_T)
LOAD-TRUCK n-truck scientific equipment (l_T)
FLY-HELICOPTER Safe Point, Far Village (l_T)

8. UNLOAD-HELICOPTER civilian1, civilian2 ($l_{\top}$)
DRIVE-TRUCK n-truck Neighbour Beach, Neighbour Village($l_{\top}$)
9. UNLOAD-TRUCK n-truck scientist1 ($l_{\top}$)
UNLOAD-TRUCK n-truck scientific equipment ($l_{\top}$)

8.3.4 An Eleven Step Plan: All People and Possessions Evacuated

An eleven-step plan may be synthesised that compromises only in the destination of the three people: the boat is used to evacuate the people, equipment and possessions (avoiding the dangerous waters) to Neighbour Village.

Plan with Satisfaction Degree l_4

1. LOAD-TRUCK lab-truck scientist1 ($l_{\top}$)
LOAD-TRUCK lab-truck scientific equipment ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Beach2, Volcano Village ($l_{\top}$)
2. LOAD-TRUCK v-truck personal possessions ($l_{\top}$)
LOAD-TRUCK v-truck civilian1 ($l_{\top}$)
LOAD-TRUCK v-truck civilian2 ($l_{\top}$)
DRIVE-TRUCK lab-truck Science Lab, Volcano Beach 2 ($l_{\top}$)
3. SAIL-BOAT Neighbour Beach, Volcano Beach1 ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Village, Volcano Beach2 ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Beach2, Volcano Beach1 ($l_{\top}$)
4. UNLOAD-TRUCK lab-truck scientist ($l_{\top}$)
UNLOAD-TRUCK lab-truck scientific equipment ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Beach2, Volcano Beach1 ($l_{\top}$)
5. LOAD-BOAT scientist ($l_{\top}$)
LOAD-BOAT scientific equipment ($l_{\top}$)
UNLOAD-TRUCK v-truck civilian1 ($l_{\top}$)
UNLOAD-TRUCK v-truck civilian2 ($l_{\top}$)
UNLOAD-TRUCK v-truck personal possessions ($l_{\top}$)
6. LOAD-BOAT civilian1 ($l_{\top}$)
LOAD-BOAT civilian2 ($l_{\top}$)
LOAD-BOAT personal possessions ($l_{\top}$)
7. SAIL-BOAT Volcano Beach1, Neighbour Beach ($l_{\top}$)
DRIVE-TRUCK n-truck Neighbour Village, Neighbour Beach ($l_{\top}$)
8. UNLOAD-BOAT scientist ($l_{\top}$)
UNLOAD-BOAT civilian1 ($l_{\top}$)
UNLOAD-BOAT civilian2 ($l_{\top}$)
UNLOAD-BOAT scientific equipment ($l_{\top}$)
UNLOAD-BOAT personal possessions ($l_{\top}$)
9. LOAD-TRUCK n-truck scientist ($l_{\top}$)
LOAD-TRUCK n-truck civilian1 ($l_{\top}$)
LOAD-TRUCK n-truck civilian2 ($l_{\top}$)
LOAD-TRUCK n-truck scientific equipment ($l_{\top}$)
LOAD-TRUCK n-truck personal possessions ($l_{\top}$)
10. DRIVE-TRUCK n-truck Neighbour Beach, Neighbour Village ($l_{\top}$)

11. UNLOAD-TRUCK n-truck scientist ($l_{\top}$)
UNLOAD-TRUCK n-truck civilian1 ($l_{\top}$)
UNLOAD-TRUCK n-truck civilian2 ($l_{\top}$)
UNLOAD-TRUCK n-truck scientific equipment ($l_{\top}$)
UNLOAD-TRUCK n-truck personal possessions ($l_{\top}$)

8.3.5 A Plan with No Compromises

Finally, a ‘no compromise’ plan, presented below, takes twelve steps to perform. The preferences of the helicopter pilot to avoid Volcano Island and the boat’s captain to avoid the dangerous route between Volcano Beach2 and Neighbour Beach are both satisfied. The flexible goals are also satisfied to the degree $l_{\top}$, with all the people, equipment and possessions safely at Far Village.

Plan with Satisfaction Degree $l_{\top}$

1. LOAD-TRUCK lab-truck scientist1 ($l_{\top}$)
LOAD-TRUCK lab-truck scientific equipment ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Beach2, Volcano Village ($l_{\top}$)
2. LOAD-TRUCK v-truck personal possessions ($l_{\top}$)
LOAD-TRUCK v-truck civilian1 ($l_{\top}$)
LOAD-TRUCK v-truck civilian2 ($l_{\top}$)
DRIVE-TRUCK lab-truck Science Lab, Volcano Beach 2 ($l_{\top}$)
3. SAIL-BOAT Neighbour Beach, Volcano Beach1 ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Village, Volcano Beach2 ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Beach2, Volcano Beach1 ($l_{\top}$)
4. UNLOAD-TRUCK lab-truck scientist ($l_{\top}$)
UNLOAD-TRUCK lab-truck scientific equipment ($l_{\top}$)
DRIVE-TRUCK v-truck Volcano Beach2, Volcano Beach1 ($l_{\top}$)
5. LOAD-BOAT scientist ($l_{\top}$)
LOAD-BOAT scientific equipment ($l_{\top}$)
UNLOAD-TRUCK v-truck civilian1 ($l_{\top}$)
UNLOAD-TRUCK v-truck civilian2 ($l_{\top}$)
UNLOAD-TRUCK v-truck personal possessions ($l_{\top}$)
6. LOAD-BOAT civilian1 ($l_{\top}$)
LOAD-BOAT civilian2 ($l_{\top}$)
LOAD-BOAT personal possessions ($l_{\top}$)
7. SAIL-BOAT Volcano Beach1, Neighbour Beach ($l_{\top}$)
DRIVE-TRUCK f-truck Far Village, Far Beach ($l_{\top}$)
8. SAIL-BOAT Neighbour Beach, Far Beach ($l_{\top}$)
9. UNLOAD-BOAT scientist ($l_{\top}$)
UNLOAD-BOAT civilian1 ($l_{\top}$)
UNLOAD-BOAT civilian2 ($l_{\top}$)
UNLOAD-BOAT scientific equipment ($l_{\top}$)
UNLOAD-BOAT personal possessions ($l_{\top}$)

10. LOAD-TRUCK f-truck scientist (l_T)
LOAD-TRUCK f-truck civilian1 (l_T)
LOAD-TRUCK f-truck civilian2 (l_T)
LOAD-TRUCK f-truck scientific equipment (l_T)
LOAD-TRUCK f-truck personal possessions (l_T)
11. DRIVE-TRUCK f-truck Far Beach, Far Village (l_T)
12. UNLOAD-TRUCK f-truck scientist (l_T)
UNLOAD-TRUCK f-truck civilian1 (l_T)
UNLOAD-TRUCK f-truck civilian2 (l_T)
UNLOAD-TRUCK f-truck scientific equipment (l_T)
UNLOAD-TRUCK f-truck personal possessions (l_T)

Each plan is returned as soon as it is found, providing an anytime solution to the Rescue problem. Having extracted all five solutions, however, each offers a different balance between compromises made and the length of the plan. A user may then select and execute the one which is considered to make the optimal tradeoff.

8.4 Summary

This chapter was firstly concerned with the evaluation of FGP on a test suite of problems engineered to contain a range of plans at different distributions of plan length. This enabled an investigation as to the amount of extra effort required compared with solving a Boolean version of the same problem. Although the Boolean problem was, unsurprisingly, always easier to solve than to synthesise plans of all satisfaction degrees from the flexible version, the time taken to produce short, low-satisfaction plans was lower still, providing an attractive ‘anytime’ behaviour. Furthermore, the overall quality of a plan containing compromises may even be higher than such a strict Boolean ‘perfect’ plan because of the differences in plan length.

The distribution of the range of plans in a flexible planning problem was established as a major contributor to the difficulty of solving the problem. In particular, the easiest problems contained the shortest compromise plans, upon finding which FGP can concentrate on searching only for plans with the higher satisfaction degrees. Conversely, the hardest problems contain no short compromise plans and exhibit larger differences in length between the plans of different satisfaction degrees, forcing FGP to search for the whole range of plans for longer.

An examination of the utility of the limited graph expansion and satisfaction propagation techniques detailed in the previous chapter was also made. It was established that both were beneficial to search, and cumulative in effect. Satisfaction propagation was seen to be most beneficial since it reduces the dominant cost of plan synthesis: plan extraction search.

A followup study might consider a larger test suite with problems containing a greater number of satisfaction degrees. This would allow a more

detailed exploration of the relationship between flexible planning problem structure and solution difficulty.

Finally, a new benchmark flexible planning problem was presented: the Rescue problem. This problem is a simplified example of the real problem of planning non-combatant military evacuations. When deciding upon evacuation points and evacuation routes, the examples used of avoiding dangerous helicopter landing sites and sea routes mirror the real-world tradeoffs. As such, the Rescue problem, and the success of FGP in solving it, provided evidence of the utility of flexible planning when faced with the time limits and soft constraints which often pervade real problems.

9 Conclusion

This chapter brings the book to a conclusion. It begins by summarising the work completed and reported herein. Future developments of the techniques developed which have been suggested throughout are then enumerated and preliminary suggestions as to their execution are made. Finally, a number of potential application areas for dynamic flexible CSP are discussed as well as how the novel search structure employed by GP-*rr*DCSP and FGP might be utilised in other ways.

9.1 A Summary

There follows a brief summary of the contents and major results of this book.

9.1.1 Dynamic Flexible Constraint Satisfaction

This book has been concerned primarily with dynamic flexible constraint satisfaction problems (DFCSPs). These problems represent the integration of two previously separate extensions to classical constraint satisfaction which address two of its main limitations:

- The inability to cope gracefully with changes to the problem structure.
- The inability to support compromises when an over-constrained problem (i.e. one that admits no ‘perfect’ solution) is encountered.

DFCSPs are therefore able to model complex situations where there is both an inherent softness to the problem description and the possibility that the problem structure may change.

Through a thorough review of the existing body of literature, a number of approaches were identified which separately address the issues of dynamics and flexibility. In light of this, a matrix of instances of DFCSP was defined, where each instance combines different dynamic and flexible methods. A third dimension to this matrix was also defined, comprising solution methods tailored to each DFCSP instance.

The foundations have been laid herein for the exploration and extension of the matrix of DFCSPs. A particular instance of DFCSP was investigated in

detail, combining restriction/relaxation-based dynamic CSP (*rrDCSP*) and fuzzy CSP. The dynamic component, *rrDCSP*, was the earliest proposed method of supporting dynamicity [32] and as such is the most thoroughly researched dynamic extension to classical CSP. The flexible component, fuzzy CSP, was defined more recently [42] and is a powerful approach which efficiently supports a range of flexible constraint types. The resulting instance of DFCSP is general enough to be practically useful to other fields of research, as demonstrated by the Flexible Graphplan application (see Chapter 7).

New algorithms were developed to solve these *rrDFCSP*s from two different perspectives, with one extending a static flexible CSP approach to cope with dynamic changes, and the other extending a non-flexible dynamic CSP approach to support flexible constraints. The first algorithm is based on the branch and bound approach to solving static fuzzy CSPs suggested in [42]. Each problem instance in a dynamic sequence is solved from scratch, but when attempting to solve it, the solution to the previous problem is used as a guide. Hence, the algorithm maintains as much of the solution to the previous problem as possible.

The second algorithm established adds support for fuzzy constraints to the Local Changes [167] *rrDCSP* algorithm. The Flexible Local Changes (FLC) algorithm thus created starts from the solution to the previous problem in a dynamic sequence, and repairs it with respect to the flexible constraints imposed by the current problem. Repairs are targeted at the sub-assignments that violate constraints, hence maintaining as much as possible of the previous solution.

Hybrid consistency enforcing variants of both algorithms were created, combining their basic search procedures with consistency enforcing techniques that enable the early detection of dead ends in the search tree. These hybrids exploit the fact that the idempotent min/max aggregation operator employed by fuzzy CSP allows the straightforward extension of classical CSP consistency enforcing techniques [141].

The structure and properties of this instance of *rrDFCSP* and the performance of branch and bound and FLC were investigated using a range of random problem sequences. Five parameters were varied when generating these sequences: the number of variables (problem size), the number of satisfaction degrees, the connectivity of the constraint graph, the tightness of the constraints and the size of the random changes made to a problem instance when generating the next instance in the sequence.

Results from solving these sequences with variants of both the branch and bound and FLC algorithms showed a number of peaks in solution difficulty corresponding to a phase transition when the problems change from being solvable to a particular satisfaction degree to being unsolvable to that satisfaction degree but still solvable to the adjacent satisfaction degree below. This was seen to be a generalisation of the same phenomenon found in em-

pirical studies of classical CSP, where one phase transition is found in the region where problems move from being solvable to unsolvable.

Around the hardness peaks, the enhanced branch and bound algorithm found solutions most efficiently, due to its relatively more effective constraint propagation. However, the variants of FLC consistently produced more stable solutions, which is important practically if a significant effort has been invested in the execution of the current solution.

9.1.2 The Application of Dynamic and Dynamic Flexible CSP to AI Planning

The second half of this book concerned the practical application of dynamic and dynamic flexible constraint satisfaction to the field of AI planning. Activity-based dynamic CSP (*aDCSP*) has already been successfully applied to this field [82]. An approach was developed herein to explore the alternative of using *rrDCSP* (and indeed *rrDFCSP*). A new Boolean planner, *GP-rrDCSP* was created which does not consider plan synthesis as one large problem. Instead, the planning graph structure created by the standard Graphplan [14] algorithm is hierarchically decomposed into a number of connected levels. Each level is viewed as a dynamic sub-problem which can then be solved with restriction/relaxation-based algorithms. Efficiency is firstly enhanced in this hierarchical search via the solution re-use capabilities of restriction/relaxation-based techniques. Further efficiency gains are obtained since the parts of an attempted solution which are identified as the cause of a conflict can easily be repaired.

GP-rrDCSP was tested on a number of Boolean benchmark problems against state of the art classical planning systems. The results obtained demonstrate that the new planner is competitive across the set of benchmarks, consistently outperforming the original Graphplan on which it is based and sometimes outperforming the state of the art planners. This established *GP-rrDCSP* as the firm basis upon which a more sophisticated planning algorithm was later developed.

Having developed the new synthesis technique for classical planning, a novel type of planning problem definition was developed to capture the inherent softness of many real problems. This *flexible planning problem* is founded on the use of subjective truth and satisfaction degree scales. Truth degrees associated with propositions support uncertainty concerning the exact state of the environment. Satisfaction degrees associated with operators and goals enable the planner to make compromises, assigning an appropriate degree of satisfaction to each instantiated operator and goal according to how well their preconditions are satisfied. Hence, a range of plans can be synthesised, trading plan length (and synthesis time) against the compromises made. This supports an attractive ‘anytime’ behaviour, where, given limited time, the plan can return the best plan found so far.

In order to synthesise plans from an input flexible planning problem, the Graphplan framework was extended to create the Flexible Graphplan (FGP) planner. FGP is a natural extension of the GP-*rr*DCSP planner, making use of the same hierarchical decomposition approach. In this case, sequences of fuzzy CSP instances are generated from the flexible planning graph to form an *rr*DFCSP level. These sequences are then solved with the FLC algorithm.

Flexible Graphplan was tested on a suite of problems engineered to contain plans of certain satisfaction degrees of particular lengths, and also on a more complex benchmark problem. The results from the test suite demonstrate that the performance of FGP is sensitive to this internal structure. For example, if a short ‘compromise’ plan can be found, this allows the planner to concentrate on finding plans with fewer compromises for the remainder of the search, reducing the effort required. Also evident was the unsurprising result that it is always more expensive to search for a range of plans than to search for one compromise-free plan. However, it is often possible to find short, low satisfaction degree plans very quickly, supporting the anytime behaviour.

The benchmark ‘Rescue’ problem results were also very encouraging. FGP was able to synthesise five plans of different lengths for this problem in a very short time, each containing different compromise decisions. The user is therefore provided with a range of options trading plan length versus the number and severity of the compromises made.

Substantial parts of this book have been published in peer-refereed literature. The introduction to each chapter gives a clear indication of where the associated material was published.

9.2 Future Work

As noted throughout the book, a number of pieces of future work remain and are described below. The further pursuit of each item is desirable in order to build upon the foundations laid herein.

9.2.1 Enrichment of the DFCSP Matrix

A principle element of future work is the further exploration (and indeed extension) of the matrix of dynamic flexible problem instances and corresponding solution techniques established in Chapter 3. Of particular interest is the combination of activity based and restriction/relaxation based dynamic CSP. Apart from the extension of the dynamic techniques dimension of the DFCSP matrix, this would produce two new dynamic CSP techniques which could be used independently of the need for flexibility. In combination with a flexible CSP method, this would produce a powerful DFCSP instance capable of modelling a complex evolving problem.

A further important item in this regard is in the development of a greater number of DFCSP solution techniques. Although the two solution methods

developed in this book have been applied to a single representative DFCSP instance, both could be relatively cheaply adapted to support other flexible methods. In addition, further solution techniques could be created through the extension of alternative existing dynamic or flexible algorithms. For example, a local search based algorithm could be used, performing hill climbing on the quality of the current solution as dictated by the flexible constraints. As has been shown for purely dynamic CSP (see Section 2.10.1 and [172]), hill-climbing can be especially useful in rapidly evolving problems, since it is not reliant on the stability of previously assigned variables. This point is equally valid for the dynamic flexible case.

The examination of different DFCSP instances and associated solution techniques should be accompanied by an empirical analysis of the type performed for the instance studied in this book. This analysis will help to establish the properties of each DFCSP instance, especially the relationship between the problem structure and the expected difficulty of producing a solution. As regards the empirical study of fuzzy *rr*DFCSP made herein, it too may be extended. Important avenues to explore include focusing problem generation on those parameter settings now known to produce the most difficult problems, and investigating how the distribution of the different fuzzy constraint types affects the solution algorithms.

As the DFCSP matrix is explored and the solution techniques for each instance well developed, the methodological issue becomes significant. When modelling a particular problem, an important step is to choose an instance of DFCSP with appropriate dynamic and flexible components. The choice of solution technique is also important: is completeness an issue? How often and how predictable are the changes to the problem structure? Hence, a taxonomy of the available DFCSP instances and solution techniques is required based on their features and requirements. A taxonomy of application tasks is also required. Having developed the taxonomies, a mapping is finally needed from the application taxonomy to the representation/solution taxonomy.

9.2.2 Improvement of GP-*rr*DCSP

The evaluation of GP-*rr*DCSP made in Chapter 6 revealed a number of areas in which this planner could be further improved. The first of these was in the updating of the planning graph itself via a particular kind of conflict identified during plan synthesis. Since the graph expansion process detects only a limited number of conflicts, which are added to the graph as mutual exclusion relations, many more are generally uncovered during the search process - hence the importance of memoisation.

If conflicts discovered at the highest current planning graph level are added to the graph, the graph expansion process can take them into account. This should lead to fewer action nodes being added to the next level of the graph and those that are added being more tightly constrained, reducing subsequent search effort. It is suggested that binary conflicts only are added

in this way, since they may be translated directly and cheaply to the mutual exclusion relations that the graph expansion process currently operates with.

A particularly important future direction for improving GP-*rr*DCSP is the development of more efficient memoisation strategies. Propagated irrelevance testing was seen generally to provide a substantial improvement to the performance of the planner because of its ability to prune irrelevant elements from memosets. However, the propagated irrelevance testing process itself can become inefficient, especially when faced with large planning graphs. Placing a bound on the depth to which propagated irrelevance testing searches when trying to prove that certain elements are irrelevant would help to alleviate the situation where a disproportionate amount of effort is spent generating a memoset relative to the amount of pruning it is likely to do.

A method of improving the efficiency of propagated irrelevance testing without potentially sacrificing the quality of the memosets generated is to take a more informed approach to this process. If the avenue that is most likely to lead to support (and hence establish that the memoset elements in question *are* relevant) is examined first, significant wasted effort can be saved. This may be seen by considering that propagated irrelevance testing immediately terminates upon the above condition.

9.2.3 Further Development of Flexible Graphplan

The flexible planning problem, as defined in this book, is founded on the use of a min/max aggregation operator for the calculation of satisfaction degrees. From a computational standpoint, this idempotent operator is very attractive, rendering plan synthesis tractable and allowing the straightforward application of efficient classical CSP techniques, as evidenced by the number of hybrid approaches tested in Chapter 4. The disadvantage, however, is the so-called *drowning effect* [141]. This describes the situation where a low satisfaction degree resulting from a particular assignment (the use of a particular operator in this case) ‘drowns’ several others whose satisfaction degrees, whether optimal or sub-optimal, are not reflected in the overall satisfaction degree.

One possible solution to this problem is to make use of Leximin FCSP [118]. This approach combines additive and min/max operations in order to consider the satisfaction degrees of all constraints (flexible operators and goals in the case of the flexible planning problem), not just the one with the lowest satisfaction degree. This refinement, however, comes at a significantly higher computational cost.

Alternatively, a fully additive aggregation operator could be employed. This would support a finer-grained form of flexibility, and would avoid the problem of the drowning effect. Unfortunately, the non-idempotent summation operator used would be expected to add significantly to the cost of plan synthesis. In addition, as has been noted by [141], the non-idempotent sum-

mation operator does not allow the straightforward extension of powerful classical CSP consistency-enforcing techniques.

As well as improving the mechanism for handling flexibility, alternative methods of plan synthesis could be explored. For example, one of the other variants of the Flexible Local Changes algorithm, as described in Chapters 3 and 4, could be employed to solve the hierarchically decomposed sub-problems. Another possibility is the use of one of the branch and bound techniques also explored in this book. A comparative study could be used to discover if the different methods are suited to different types of flexible planning problem.

A departure from the hierarchical decomposition synthesis method advocated in this book might also be attempted. An activity-based instance of DFCSP could be used to support a method of plan synthesis which considers the flexible planning graph as a single problem, and which could benefit from the extension of the *a*DCSP techniques currently employed by state of the art classical planners.

Another synthesis technique could follow the BlackBox [83] approach to classical planning, encoding the flexible planning graph as a propositional satisfiability (SAT) problem. Existing efficient solution techniques for SAT problems are then used to solve this problem, hence providing a valid plan. In order to cope with the non-Boolean nature of the flexible planning graph, a ‘banded’ plan synthesis technique would be necessary. Consider the case with a satisfaction scale of $L = \{l_{\perp}, l_1, l_2, l_3, l_{\top}\}$. Plan extraction would work by first considering only the sub-graph containing flexible goals and actions with the satisfaction degree $l_{\top}$. Given a uniform satisfaction degree, each action and goal encoded into the SAT problem can be given equal weight, regaining a Boolean structure.

If a solution to the resultant SAT problem is found, then a plan with satisfaction degree $l_{\top}$ has been found. If, however, a solution cannot be found, the sub-graph containing both satisfaction degrees $l_{\top}$ and l_3 is considered. When encoding a SAT problem from this sub-graph, all actions and goals are again weighted equally to gain a Boolean problem. This is possible because of the min operator: since a plan with satisfaction degree $l_{\top}$ is known (or assumed if an incomplete SAT solver is used) not to exist, a plan will not be found that contains only actions and goals with the satisfaction degree $l_{\top}$. Further, the min aggregation operator guarantees that a plan containing actions and goals with a combination of l_3 and $l_{\top}$ satisfaction degrees has an overall satisfaction degree of l_3 . Hence, if a solution is found by the SAT solver, a plan of satisfaction degree l_3 has been found.

This process can continue down to l_1 , where, if no solution is found, the flexible planning graph is extended to the next level. Preliminary work in this regard has been carried out. Some encouraging results are presented in [79].

Fig. 9.1. A Simple Behaviour Tree

9.2.4 Further Applications

The creation of the flexible planning problem, as described in Chapter 7, is just an application of one instance of DFCSP. Many other potential applications exist for this instance and others. Three new potential applications for these techniques are proposed below.

Flexible Qualitative Simulation

The field of qualitative systems simulation [89, 148], as briefly described in Chapter 1, requires the construction of a tree of possible system behaviours given an initial system state and some constraints. Transition rules specify, given a particular variable state in terms of a magnitude and rate of change, the possible next states for that variable. Constraints between variables restrict the possible combinations of the next states of all the system variables. A system state consists of a state assignment to each variable such that all constraints are satisfied. Any given system state may give rise to a number of child system states, hence a behaviour tree develops, as illustrated in Figure 9.1, with each branch describing a possible system behaviour.

Classical constraint satisfaction is traditionally used in the process of generating system states [107] while satisfying all specified constraints. However, both dynamicity and flexibility could be used to improve this process.

The constraints governing the behaviour of the system under consideration are not subject to change for this application. However, the domains of each variable, specified by transition rules from the previous system state *do* change. Hence, since not all variable states will change from one system state to the next, a dynamic CSP algorithm could be used to repair previous system states and find possible next states more quickly.

In the case of fuzzy qualitative simulation, embodied by the FuSim algorithm [148], some assignments of next states to variables are preferred over others. This is due to the fact that the fuzzy set corresponding to a preferred assignment satisfies the system constraints more closely than those corresponding to the less preferred assignments. Hence, flexible CSP could be used to construct the most preferred system state first, leading to a best-first search of the behaviour tree. It is envisaged that an appropriate DFCSP algorithm would be especially beneficial in the case of fuzzy qualitative simulation, simultaneously supporting preferred assignments and dynamic repair of previous system states.

Flexible Compositional Modelling

Many applications require the use of system models. The modelling tasks involved require techniques which augment human skills by rapidly and automatically generating a reasonable-sized set of alternative solutions. A successful modelling tool should be able to: develop complex, large-scale models; offer high maintainability should a design be subject to change; support the re-use of model components, enabling a ‘plug and play’ type framework to facilitate model development across many distinct scenarios; allow modifications to cope with the predicted effects of the current model; and make sensible tradeoffs when faced with conflicting goals.

Compositional Modelling [46, 85] (CM) is a knowledge-based approach to model construction which attempts to address these requirements. The CM methodology constructs domain system models by defining, selecting and then composing *model fragments*. Knowledge is decomposed into semi-independent fragments using explicit modelling assumptions. Each fragment contains a model for part of a domain system, where a part may constitute a sub-system or component of the system. Model fragments may represent different perspectives on the same component, hence reflecting alternative modelling decisions.

Currently, the domain-theory languages [16, 59] used to define model fragments and the procedures used to select from a set of candidate models [46] use, at best, ad-hoc methods of supporting tradeoffs in the model formulation process. This is one of the main obstacles preventing the application of CM in complex industrial settings. The CM process would benefit greatly from the extra information afforded by a rigorous means of supporting flexibility, by being able to make informed decisions as to what compromises must be made if a particular problem does not admit of a ‘perfect’ solution (i.e. the domain system model required).

Flexible constraints may be generated directly from the model-fragment library, for example there may be a preference to combine fragments of the same precision and/or resolution [94] if at all possible, and there may be a high priority attached to establishing the model scope. Further flexible constraints may stem from the task specification to maximise the utility of

the composed model for a particular purpose. For instance, a tradeoff may exist between constructing an accurate model of adequate detail which will include all relevant aspects of the system required for the given task and minimising extraneous detail. Given a flexible modelling process, questions of accuracy, adequacy and relevance immediately cease to be Boolean, since each may be satisfied to an extent, thereby allowing informed compromises to be made to construct a model which optimally satisfies the flexible goals.

Dynamic flexible constraint satisfaction may be used to provide a formal framework for model construction. Indeed, the process of static, non-flexible model composition has been recognised as equivalent to solving an activity-based CSP [115]. DFCSP represents a formal approach to flexibility, as opposed to the ad-hoc methods of constructing and choosing a candidate model currently available. Furthermore, by incorporating flexible constraints directly, a DFCSP solution technique would be able to automatically construct the optimal model without the need for a further model selection process from a set of possible candidates, subject to the given set of priorities and preferences.

The activity-based nature of the model composition process, i.e. each tentative modelling choice can lead to further choices becoming relevant, means that it bears strong similarities to the process of plan extraction explained in Chapters 5 and 7. Hence, the same method of hierarchical decomposition of the initial large problem into tractable sub-problem instances could be employed. However, the compositional modelling process does not necessarily enjoy the same strictly levelled structure as a planning graph. It would therefore be necessary to generalise the decomposition process accordingly.

A further benefit of using a restriction/relaxation-based instance of DFCSP is the immediate support for changes to the global compositional modelling task. Instead of starting the modelling process from scratch, it is possible to repair the current model with respect to a new task specification (assuming reasonable similarity between the two), thus re-using much of the effort. The overall repair process will be supported at each sub-problem level simply by extending the dynamic sub-problem sequence and adapting the previous sub-problem solution to the current requirements.

Alternatively for this problem, a new instance of activity-based DFCSP (*a*DFCSP) could be created. Direct support for activity constraints enables the consideration of the whole problem at once, as per Graphplan. However, an *a*DFCSP approach would not be able to directly support changes to the specification of the global compositional modelling task. This task appears therefore to be a target application for the *a/rr*DFCSP techniques proposed in Section 9.2.1.

Meta-CSP

This is a theoretical rather than a practical application, based on the solution techniques employed in developing the GP-*rr*DCSP and Flexible Graphplan

Fig. 9.2. Constraint Graph Decomposition into Non-separable Components to Form Meta-Variables

planners. Chapters 5-8 described how a particular type of problem, i.e. that of solution extraction in a Graphplan planning graph, could be efficiently solved via the decomposition of the original large problem into smaller sub-problems which were then solved in sequences. The search for a solution thus becomes the search through a sub-problem space, identifying the combination of sub-problems that can be solved conjunctively.

A potential use for this technique is to solve large generalised CSPs. In order to do this, a suitable decomposition process would need to be identified. The non-separable components method described in Section 2.6.3 appears to be a promising method to do this. However, instead of simply enumerating every solution to each sub-problem created by the decomposition process, conflict propagation could be employed as per the extended Graphplan synthesis techniques described in this book. Hence, if a sub-problem instance is insoluble due to the current solutions to connected sub-problems, a constraint could be recorded locally to explain this failure, and also propagated back to the offending sub-problems. The propagated constraint could then be used to add constraint(s) to these sub-problems, whose solutions would be repaired using an *rrDCSP* (or indeed *rrDFCSP*) technique.

This process may be viewed as a type of meta-CSP, if the sub-problems are considered as variables with domains consisting of solutions. With respect to the problem sub-sequences described in Chapter 5, the initial problem corresponds to a variable whose domain has not yet been filtered, whilst the other members of a sub-sequence represent the same variable with domain elements removed by propagation. Adopting this viewpoint, it may be possible to use the more advanced techniques to search the space of sub-problem meta-variables. For instance, Backjump might be used if it can be established that the current sub-problem is unsolvable (i.e. the domain of the meta-variable is empty), directly as a result of a previous sub-problem solution/meta-variable assignment.

Figure 9.2 presents an example. Here, the non-separable components correspond to meta-variables. Taking the naive instantiation order μ_1, μ_2, μ_3 , μ_1 would first be instantiated via the solution of the associated sub-problem, thus fixing part of the sub-problem corresponding to μ_2 . The second meta variable, μ_2 would then be assigned and so on. Given a simple structure like this, it is a simple matter to apportion blame in the event of an insoluble sub-problem. In the general case, however, the conflict recording techniques developed for plan synthesis in this book may be exploited to propagate discovered conflicts to adjacent meta-variables.

9.3 And Finally

To conclude, it is hoped that the techniques and results presented in this book are found useful by others working in both the constraint satisfaction and planning communities, as well as other related disciplines.

References

- [1] J. Allen. Toward a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.
- [2] J. Allen, J. Hendler, and A. Tate. *Readings in Planning*. Morgan Kaufmann, 1990.
- [3] J.A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 83–88, 1988.
- [4] M. Baiocchi, S. Marcugini, and A. Milani. Dpplan: an algorithm for fast solutions extraction from a planning graph. *Proceedings of the 5th International Conference on Artificial Intelligence Planning*, 2000.
- [5] A.B. Baker. The hazards of fancy backtracking. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 288–293, 1994.
- [6] P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 600–606, 1995.
- [7] P. Baptiste and C. Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 375–389, 1997.
- [8] P. Baptiste, C. Le Pape, and L. Peridy. Global constraints for partial cps: A case-study of resource and due date constraints. *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming*, pages 87–101, 1998.
- [9] S. Beale. Using branch-and-bound with constraint satisfaction in optimization problems. *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 209–214, 1997.
- [10] U. Bertele and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.
- [11] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

- [12] C. Bessière, E.C. Freuder, and J. Régin. Using inference to reduce arc consistency computation. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 592–598, 1995.
- [13] R. Bitner and M. Reingold. Backtrack programming techniques. *Communications of the Association of Computing Machinery*, 18(11):651–656, 1975.
- [14] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.
- [15] A. Blum and J. Langford. Probabilistic planning in the graphplan framework. *Proceedings of the Fifth European conference on Planning Systems, Durham, UK*, pages 320–332, 1999.
- [16] D. Bobrow, B. Falkenhainer, A. Farquhar, R. Fikes, K. Forbus, T. Gruber, Y. Iwasaki, and B. Kuipers. A compositional modelling language. *Proceedings of the Tenth International Workshop on Qualitative Reasoning*, pages 12–21, 1996.
- [17] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, 1989.
- [18] R.D. Boyle and R.C. Thomas. *Computer Vision: A First Course*. Blackwell Scientific Publications, 1988.
- [19] D. Brelaz. New methods to colour the vertices of a graph. *Communications of the Association of Computing Machinery*, 22(4):251–256, 1979.
- [20] M. Bruynooghe. Solving combinatorial search problems by intelligent backtracking. *Information Processing Letters*, 12(1):36–39, 1981.
- [21] M. Bruynooghe. Graph coloring and constraint satisfaction. Technical Report CW-44, Department Computerwetenschappen, Katholieke Universiteit Leuven, 1989.
- [22] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the *really* hard problems are. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 331–337, 1991.
- [23] S. Chien. Static and completion analysis for planning knowledge base development and verification. *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, pages 53–61, 1996.
- [24] C.K. Chiu, C.M. Chou, J.H.M. Lee, H.F. Leung, and Y.W. Leung. A constraint-based interactive train rescheduling tool. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 104–118, 1996.
- [25] A. Chmeiss and P. Jegou. Path-consistency: When space misses time. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 196–201, 1996.
- [26] M.C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41:89–95, 1989.

- [27] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association of Computing Machinery*, 7:201–215, 1960.
- [28] J. de Kleer. A comparison of ATMS and CSP techniques. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 290–296, 1989.
- [29] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 412–417, 1997.
- [30] R. Dechter. Enhancement schemes for constraint processing: Back-jumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
- [31] R. Dechter. Constraint networks. In *Encyclopedia of Artificial Intelligence*, pages 276–285. John Wiley and Sons, 1992.
- [32] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 37–42, 1988.
- [33] R. Dechter, A. Dechter, and J. Pearl. Optimization in constraint networks. In R.M. Oliver and J.Q. Smith, editors, *Influence Diagrams, Belief Nets and Decision Analysis*, pages 411–425. John Wiley and Sons, 1990.
- [34] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 271–277, 1989.
- [35] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
- [36] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.
- [37] Y. Descotte and J. Latombe. Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, 27:183–217, 1985.
- [38] Y. Deville and P. van Hentenryck. An efficient arc consistency algorithm for a class of CSP problems. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, page 325 330, 1991.
- [39] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
- [40] D. Draper, S. Hanks, and D. Weld. Probabilistic planning with information gathering and contingent execution. *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 31–36, 1994.
- [41] D. Dubois, H. Fargier, and H. Prade. Fuzzy constraints in job-shop scheduling. *Journal of Intelligent Manufacturing*, 6:215–234, 1995.

- [42] D. Dubois, H. Fargier, and H. Prade. Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty. *Applied Intelligence*, 6:287–309, 1996.
- [43] D. Dubois, H. Fargier, and H. Prade. Refinements of the maximin approach to decision making in a fuzzy environment. *Fuzzy Sets and Systems*, 81:103–122, 1996.
- [44] H. Dubois, D. Prade. *Possibility Theory: An Approach to Computerised Processing of Uncertainty*. Plenum Press, New York, 1988.
- [45] S. Even. *Graph Algorithms*. Computer Science Press, 1979.
- [46] B. Falkenhainer and K.D. Forbus. Compositional modelling: Finding the right model for the job. *Artificial Intelligence*, 51:95–143, 1991.
- [47] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems. *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (Lecture Notes in Computer Science 747)*, pages 97–104, 1993.
- [48] R. Feldman and M.C. Golumbic. Constraint satisfiability algorithms for interactive student scheduling. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1010–1016, 1989.
- [49] G. Ferguson and J. Allen. TRIPS: An intelligent integrated problem-solving assistant. *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 567–573, 1998.
- [50] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208, 1971.
- [51] G. Fowler, R. Haralick, and F.G. Gray. Efficient graph automorphism by vertex partitioning. *Artificial Intelligence*, 21:245–269, 1983.
- [52] M.S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. Pitman, Morgan Kaufmann, 1987.
- [53] E. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the Association for Computing Machinery*, 32(4):755–761, 1985.
- [54] E.C. Freuder. Synthesizing constraint expressions. *Communications of the Association of Computing Machinery*, 21(11):958–966, 1978.
- [55] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the Association of Computing Machinery*, 29(1):24–32, 1982.
- [56] E.C. Freuder. Partial constraint satisfaction. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 278–283, 1989.
- [57] E.C. Freuder and C.D. Elfe. Neighbourhood inverse consistency processing. *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 202–208, 1996.
- [58] E.C. Freuder and R.J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

- [59] M.P.J. Fromberz, V.A. Saraswat, and D. Bobrow. Model-based computing: Developing flexible machine control software. *Artificial Intelligence*, 114:157–202, 1999.
- [60] D. Frost and R. Dechter. Dead-end driven learning. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 294–300, 1994.
- [61] D. Frost and R. Dechter. In search of the best constraint satisfaction search. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, 1994.
- [62] D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 572–578, 1995.
- [63] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Department of Computer Science, Carnegie-Mellon University, 1979.
- [64] P.A. Geelen. Dual viewpoint heuristics for the binary constraint satisfaction problem. *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 31–35, 1992.
- [65] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 327–340, 1997.
- [66] I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 179–193, 1996.
- [67] I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 246–252, 1996.
- [68] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
- [69] M.L. Ginsberg, M. Frank, M. Halpin, M.P. Halpin, and M.C. Torrance. Search lessons learned from crossword puzzles. *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 210–215, 1990.
- [70] M.L. Ginsberg and D.A. McAllester. GSAT and dynamic backtracking. *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pages 226–237, 1994.
- [71] M.L. Ginsberg, A.J. Parkes, and A. Roy. Supermodels and robustness. *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 334–339, 1998.
- [72] R. Goldman and M. Boddy. Conditional linear planning. *Proceedings of the Second International conference on Artificial Intelligence Planning Systems*, pages 80–85, 1994.

- [73] R. Goldman and M. Boddy. Epsilon-safe planning. *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 253–261, 1994.
- [74] Q. Guan and G. Friedrich. Extending constraint satisfaction problem solving in structural design. *Proceedings of the Fifth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (Lecture Notes in Computer Science 604)*, Paderborn, Germany, 604:341–350, 1992.
- [75] H.W. Guesgen and J. Hertzberg. *A Perspective of Constraint Based Reasoning*, volume 597 of *LNAI*. Springer, Berlin, 1992.
- [76] C. Han and C. Lee. Comments on Mohr and Henderson’s path consistency algorithm. *Artificial Intelligence*, 36:125–130, 1988.
- [77] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
- [78] J. Jaffar, S. Michayov, P. Stuckey, and R.H.C Yap. The CLP(R) language and system. *Association of Computing Machinery Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
- [79] P. Jarvis, I. Miguel, and Q. Shen. Flexible blackbox: Preliminary results. *Proceedings of the AAAI-2000 Workshop on Representational Issues for Real-World Planning Systems*, pages 43–49, 2000.
- [80] A.K. Jonsson and M.L. Ginsberg. Experimenting with new systematic and non-systematic search techniques. *Proceedings of the AAAI Symposium on AI and NP-hard Problems*, 1993.
- [81] D. Joslin and M.E. Pollack. Passive and active decision postponement in plan generation. *Proceedings of the 3rd European Workshop on Planning*, pages 37–48, 1995.
- [82] S. Kambhampati. Planning graph as (dynamic) CSP: Exploiting EBL, DDB and other CSP techniques in graphplan. *Journal of AI Research*, 12:1–34, 2000.
- [83] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 318–325, 1999.
- [84] H. Kautz and J. Walser. State-space planning by integer optimization. *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 526–533, 1999.
- [85] J. Keppens and Q. Shen. On compositional modelling. *Knowledge Engineering Review (to appear)*, 16, 2001.
- [86] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. *Proceedings of the Fourth European Conference on Planning*, pages 273–285, 1997.
- [87] T.H. Kolbe. Constraints for object recognition in aerial images - handling of unobserved features. *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science 1520*, pages 295–309, 1998.

- [88] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 541–547, 1995.
- [89] B. Kuipers. *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. MIT Press, 1994.
- [90] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, pages 32–44, 1992.
- [91] N. Kushmerick, S. Hanks, and D. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1-2):239–286, 1995.
- [92] J. Lang. Possibilistic logic as a logical framework for min-max discrete optimisation problems and prioritised constraints. *Proceedings of the International Conference on the Fundamentals of Artificial Intelligence Research*, pages 112–126, 1991.
- [93] E.L. Lawler and D.E. Wood. Branch-and-bound methods: a survey. *Operations Research*, 14:699–719, 1966.
- [94] R. Leitch, Q. Shen, G. Coghill, and M. Chantler. Choosing the right model. *IEEE Proceedings - Control Theory and Applications*, 146(5):435–449, 1999.
- [95] D. Long. The aips-98 planning competition. *AI Magazine*, 21(2):13–33, 2000.
- [96] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *Journal of AI Research*, 10:87–115, 1998.
- [97] A. Mackworth and E. Freuder. The complexity of some polynomial network-consistency algorithms for constraint-satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
- [98] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
- [99] A.K. Mackworth. On reading sketch maps. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 598–606, 1977.
- [100] A.K. Mackworth. Constraint satisfaction problems. In *Encyclopedia of Artificial Intelligence*, pages 285–293. John Wiley and Sons, 1992.
- [101] B. Mazure, L. Saïs, and E. Grégoire. Tabu search for SAT. *Proceedings of the 15th National Conference on Artificial Intelligence*, pages 281–285, 1997.
- [102] D. McDermott. A general framework for reason maintenance. *Artificial Intelligence*, 50:289–329, 1991.
- [103] J. McGregor. Relational consistency algorithms and their applications in finding subgraph and graph isomorphism. *Information Sciences*, 19:229–250, 1979.
- [104] P. Meseguer. Constraint satisfaction problems: An overview. *AI Communications*, 2(1):3–17, 1989.
- [105] P. Meseguer and J. Larrosa. Solving fuzzy constraint satisfaction problems. *Proceedings of FUZZ-IEEE*, pages 1233–1238, 1997.

- [106] I. Miguel, P. Jarvis, and Q. Shen. Flexible graphplan. *Proceedings of the Fourteenth European Conference on Artificial Intelligence*, pages 506–510, 2000.
- [107] I. Miguel and Q. Shen. Extending qualitative modelling for simulation of time-delayed behaviour. *Proceedings of the Twelfth International Workshop on Qualitative Reasoning*, pages 161–166, 1998.
- [108] I. Miguel and Q. Shen. Extending FCSP to support dynamically changing problems. *Proceedings of the Eighth International Conference on Fuzzy Systems*, pages 1615–1620, 1999.
- [109] I. Miguel and Q. Shen. Hard, flexible and dynamic constraint satisfaction. *Knowledge Engineering Review*, 14(3):199–220, 1999.
- [110] I. Miguel and Q. Shen. Dynamic flexible constraint satisfaction. *Applied Intelligence*, 13(3):231–245, 2000.
- [111] I. Miguel and Q. Shen. Solution techniques for constraint satisfaction problems: Advanced approaches. *Artificial Intelligence Review*, 15(4):269–293, 2001.
- [112] I. Miguel and Q. Shen. Solution techniques for constraint satisfaction problems: Foundations. *Artificial Intelligence Review*, 15(4):243–267, 2001.
- [113] I. Miguel, Q. Shen, and P. Jarvis. Efficient flexible planning via dynamic flexible constraint satisfaction. *Engineering Applications of Artificial Intelligence*, 14(3):301–327, 2001.
- [114] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
- [115] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 25–32, 1990.
- [116] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
- [117] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.
- [118] J. Moura Pires, F. Moura Pires, and R. Almeida Ribeiro. Structure and properties of leximin fcsp and its influence on optimisation problems. *Proceedings of the Seventh Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pages 188–194, 1998.
- [119] J. Moura Pires and H. Prade. Logical analysis of fuzzy constraint satisfaction problems. *Proceedings of FUZZ-IEEE-98*, pages 957–962, 1998.
- [120] K. Myers and T. Lee. Generating qualitatively different plans through metatheoretic biases. *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 570–576, 1999.

- [121] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
- [122] N. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.
- [123] E. Pednault. ADL: Exploring the middle ground between strips and the situation calculus. *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.
- [124] W. Pedrycz and F. Gomide. *An Introduction to Fuzzy Sets: Analysis and Design*. MIT Press, 1999.
- [125] M. Peot and D. Smith. Conditional non-linear planning. *Proceedings of the First International Conference on Artificial Intelligence Planning Systems*, pages 189–197, 1992.
- [126] P. Prosser. A reactive scheduling agent. *Proceedings of The Eleventh International Joint Conference on Artificial Intelligence*, pages 1004–1009, 1989.
- [127] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(1):268–299, 1993.
- [128] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
- [129] P. Prosser. The dynamics of dynamic variable ordering heuristics. *Proceedings of the Fourth International Conference on the Principles and Practice of Constraint Programming*, pages 17–23, 1998.
- [130] P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. *Proceedings of the Sixth International Conference on the Principles and Practice of Constraint Programming*, pages 353–368, 2000.
- [131] L. Pryor and G. Collins. Planning for contingencies: A decision based approach. *Journal of AI Research*, 4:287–339, 1996.
- [132] P.W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.
- [133] J.F. Rit. Propagating temporal constraints for scheduling. *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 383–386, 1986.
- [134] R. Rodosek and M. Wallace. A generic model and hybrid algorithm for hoist scheduling problems. *Proceedings of The Fourth International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science 1520*, pages 385–399, 1998.
- [135] A. Rosenfeld, R. Hummel, and S. Zucker. Scene labeling by relaxation operations. *IEEE Transactions on Systems, Man, and Cybernetics*, 6:420–433, 1976.
- [136] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 1995.

- [137] R. Sabbadin, D. Dubois, and H. Prade. A fuzzy constraint satisfaction problem in the wine industry. *Journal of Intelligent and Fuzzy Systems*, 6:361–374, 1998.
- [138] D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 125–129, 1994.
- [139] E. Sacerdoti. The nonlinear nature of plans. *Proceedings of the fourth International Joint Conference on Artificial Intelligence*, pages 206–214, 1975.
- [140] T. Schiex. Possibilistic constraint satisfaction problems, or how to handle soft constraints. *Proceedings of the Eighth Conference on Uncertainty in Artificial Intelligence*, pages 268–275, 1992.
- [141] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 631–637, 1995.
- [142] T. Schiex, J. Régis, C. Gaspin, and G. Verfaillie. Lazy arc consistency. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 216–221, 1996.
- [143] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
- [144] T. Schiex and G. Verfaillie. Stubbornness: a possible enhancement for backjumping and nogood recording. *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 165–169, 1994.
- [145] R. Seidel. A new method for solving constraint satisfaction problems. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 338–342, 1981.
- [146] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.
- [147] L. Shapiro and R. Haralick. Structural descriptions and inexact matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(5):504–518, 1981.
- [148] Q. Shen and R. Leitch. Fuzzy qualitative simulation. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(4):1038–1061, 1993.
- [149] B.M. Smith. Phase transition and the mushy region in constraint satisfaction problems. *Proceedings of the Eleventh European Conference on Artificial Intelligence*, pages 100–104, 1994.
- [150] B.M. Smith and S.A. Grant. Trying harder to fail first. *Proceedings of the Thirteenth European Conference on Artificial Intelligence*, pages 249–253, 1998.

- [151] D. Smith and D. Weld. Conformant graphplan. *Proceeding of the Fifteenth International Conference on Artificial Intelligence*, pages 889–896, 1998.
- [152] S. Smith, D. Nau, and T. Throop. Total-order multi-agent task-network planning for contract bridge. *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 108–113, 1996.
- [153] R.M. Stallman and G.J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.
- [154] M. Stefik. Planning with constraints (molgen: Part 1). *Artificial Intelligence*, 16:111–140, 1981.
- [155] K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.
- [156] K. Stergiou and T. Walsh. Encodings of non-binary constraint satisfaction problems. *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 163–168, 1999.
- [157] H.S. Stone and J. Stone. Efficient search techniques: An empirical study of the n-queens problem. Technical report rc 12057, IBM T.J. Watson Research Center, Yorktown Heights, New York, 1986.
- [158] R.E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 13(3):566–579, 1984.
- [159] A. Tate. Generating project networks. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.
- [160] A. Tate, J. Dalton, and J. Levine. Generation of multiple qualitatively different plans. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 27–34, 1998.
- [161] E.P.K. Tsang. The consistent labeling problem in temporal reasoning. *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 251–255, 1987.
- [162] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.
- [163] E.H. Turner and R.M. Turner. A constraint-based approach to assigning system components to tasks. *Applied Intelligence*, 10(2):155–172, 1999.
- [164] J.R. Ullman. An algorithm for subgraph isomorphism. *Journal of the Association of Computing Machinery*, 23:31–42, 1976.
- [165] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
- [166] P. van Hentenryck and T.L. Provost. Incremental search in constraint logic programming. *New Generation Computing*, 9:257–275, 1991.

- [167] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 307–312, 1994.
- [168] T. Vossen, M. Ball, A. Lotem, and D. Nau. Applying integer programming to AI planning. *Knowledge Engineering Review*, 15(1):85–100, 2000.
- [169] M. Wallace. Survey: Practical applications of constraint programming. *Constraints Journal*, 1(1):139–168, 1995.
- [170] R.J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in csp. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 239–245, 1993.
- [171] R.J. Wallace. Enhancements of branch and bound methods for the maximal constraint satisfaction problem. *The Thirteenth National Conference on Artificial Intelligence*, pages 188–195, 1996.
- [172] R.J. Wallace and E.C. Freuder. Stable solutions for dynamic constraint satisfaction problems. *The Fourth International Conference on Principles and Practice of Constraint Programming*, pages 447–461, 1998.
- [173] D. Waltz. Understanding line drawings of scenes with shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.
- [174] D.S. Weld. An introduction to least-commitment planning. *AI Magazine*, 15(4):27–61, 1994.
- [175] D.S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.
- [176] D. Wilkins. *Practical Planning*. Morgan Kaufmann, 1988.
- [177] D. Wilkins and K. Myers. A multiagent planning architecture. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 154–162, 1998.
- [178] R. Zabih and D. McAllester. A rearrangement strategy for determining propositional satisfiability. *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 155–160, 1988.

A Pseudo-code

For completeness, this appendix contains pseudo-code descriptions of some of the basic algorithms discussed in Chapter 2.

A.1 Backtrack

Figure A.1 describes the Backtrack algorithm. For simplicity, the pseudo-code presented herein makes no distinction between finding no solution and finding no *more* solutions; **False** is returned on both occasions. Binary constraints are also assumed for clarity, but this is not a weakness of the algorithms presented. This and all algorithms described in this appendix have access to **test**(i, j), which returns *true* if $c(x_i, x_j)$ is satisfied by the current assignments to x_i and x_j . If $c(x_i, x_j)$ does not exist, **test**() returns **true** (this is *not* counted as a constraint check).

A.2 Backjump

Figure A.2 describes the Backjump algorithm. If a constraint check with a past variable x_h fails, the *maxCheckLevel* (the deepest variable that a check

```
01. Procedure BT( $i$ )
02.   Foreach  $d_{i_j} \in D_i$ 
03.     assign( $x_i, d_{i_j}$ )
04.      $consistent \leftarrow true$ 
05.     For  $h \leftarrow 1$  To  $i - 1$  While  $consistent$ 
06.        $consistent \leftarrow test(i, h)$ 
07.     If  $consistent$ 
08.       If  $i = n$  showSolution()
09.       Else BT( $i + 1$ )
10.   Return  $false$ 
11. End Procedure
```

Fig. A.1. The Backtrack Algorithm

```

01. Procedure BJ( $i$ )
02.    $maxCheckLevel \leftarrow 0$ 
03.    $returnDepth \leftarrow 0$ 
04.   Foreach  $d_{i_j} \in D_i$ 
05.     assign( $x_i, d_{i_j}$ )
06.      $consistent \leftarrow true$ 
07.     For  $h \leftarrow 1$  To  $i - 1$  While  $consistent$ 
08.        $consistent \leftarrow test(i, h)$ 
09.        $maxCheckLevel \leftarrow h - 1$  ; obtain last loop value
10.     If  $consistent$ 
11.       If  $i = n$  showSolution()
12.     Else
13.        $maxCheckLevel \leftarrow BJ(i + 1)$ 
14.       If  $maxCheckLevel < i$ 
15.         Return  $maxCheckLevel$ 
16.      $returnDepth \leftarrow \max(returnDepth, maxCheckLevel)$ 
17.   Return  $returnDepth$ 
18. End Procedure

```

Fig. A.2. The Backjump Algorithm

is made against) is set to h^1 (lines 7-9). After a successful assignment the $maxCheckLevel$ is $i - 1$ (checks against all past variables succeeded). The $returnDepth$ is the deepest level checked against by any element of D_i (line 17). If D_i is exhausted, Backjump ‘jumps’ back to the $returnDepth$ level. A new consistent value is assigned at this level and search continues.

Given a consistent assignment, Backjump proceeds as per Backtrack. If no future variables remain a solution has been found, otherwise Backjump is called recursively (line 13). The result of this invocation is the $returnDepth$ at level $i + 1$. If this value, say h , is less than i , Backjump is in the process of ‘jumping’ back to level h , effectively bypassing level i (lines 14-15). The invocations are unwound until level h is reached. Otherwise, the algorithm has jumped (or indeed stepped) back to this level, i , so the main loop proceeds to select a new value to assign to x_i .

A.3 Conflict-directed Backjump

Conflict-directed Backjump (CBJ) [127] improves Backjump further (see Figure A.3). Multiple backward jumps are enabled by recording the *conflict set* ($cSet_i$ at level i) of past variables that *failed* constraint checks with the current variable. Jumps are made to the deepest value in this set (line 18). When jumping from i to h , $cSet_h$ is set to the union of the conflict sets at levels

¹ It is assumed that, due to its nature, the FOR loop will terminate with h having a value 1 greater than the current level, which is compensated for by the algorithm.

```

01. Procedure CBJ( $i$ )
02.    $cSet_i \leftarrow \{0\}$ 
03.   Foreach  $d_{i_j} \in D_i$ 
04.     assign( $x_i, d_{i_j}$ )
05.      $consistent \leftarrow true$ 
06.     For  $h \leftarrow 1$  To  $i - 1$  While  $consistent$ 
07.        $consistent \leftarrow test(i, h)$ 
08.     If Not  $consistent$ 
09.        $cSet_i \leftarrow cSet_i \cup \{h - 1\}$  ; obtain last loop value
10.     If  $consistent$ 
11.       If  $i = n$ 
12.         showSolution()
13.          $cSet_i \leftarrow cSet_i \cup \{n - 1\}$ 
14.       Else
15.          $returnedDepth \leftarrow CBJ(i + 1)$ 
16.         If  $returnedDepth < i$ 
17.           Return  $returnedDepth$ 
18.        $returnDepth \leftarrow \max(cSet_i)$ 
19.        $cSet_{returnDepth} \leftarrow cSet_{returnDepth} \cup cSet_i - \{returnDepth\}$ 
20.     Return  $returnDepth$ 
21. End Procedure

```

Fig. A.3. The Conflict-directed Backjump Algorithm

i and h (line 19). A further jump, if necessary, is made to the deepest variable in conflict with either x_i or x_h , avoiding the problems encountered by max-fail Backjump.

At each invocation of CBJ(i), $cSet_i$ is reset to clear the conflict set for this level of any information from previous invocations (line 2). If a solution is found, $\{n - 1\}$ is added to the conflict set to ensure that CBJ backtracks to level $n - 1$ when D_n is exhausted (line 13). It is only possible to backtrack one level in this case because there is no inconsistency with this level (a solution was found here). If a successful assignment is found but future variables remain, CBJ is called recursively, and the result stored in $returnedDepth$ (lines 14-15). This value is used, as by Backjump, to discern whether the algorithm has returned to this level to select a new value, or is in the process of jumping back over this level (lines 16-17).

A.4 Backmark

Backmark (see Figure A.4) counts through the elements of D_i , using this index to access $mark[]$ (line 2). For each element of D_i , it checks whether $mark[i, DIndex]$ is less than $minBackupLevel[i]$ (line 3). If so, the assignment to x_i must fail, since the past variable that caused it to fail previously is unchanged. It is possible to skip to a new assignment without redun-

```

01. Procedure BM( $i$ )
02.   For  $DIndex \leftarrow 1$  To  $\text{length}(D_i)$ 
03.     If  $\text{mark}[i, DIndex] \geq \text{minBackupLevel}[i]$ 
04.       assign( $x_i, d_{i, DIndex}$ )
05.        $\text{consistent} \leftarrow \text{true}$ 
06.       For  $h \leftarrow \text{minBackupLevel}[i]$  To  $i - 1$  While  $\text{consistent}$ 
07.          $\text{consistent} \leftarrow \text{test}(i, h)$ 
08.          $\text{mark}[i, DIndex] \leftarrow h - 1$  ; obtain last loop value
09.       If  $\text{consistent}$ 
10.         If  $i = n$  showSolution()
11.         Else BM( $i + 1$ )
12.        $\text{minBackupLevel}[i] \leftarrow i - 1$ 
13.   For  $\text{index} \leftarrow i + 1$  To  $n$ 
14.      $\text{minBackupLevel}[\text{index}] \leftarrow \min(\text{minBackupLevel}[\text{index}], i - 1)$ 
15. End Procedure

```

Fig. A.4. The Backmark Algorithm

dantly checking this value. If $\text{mark}[i, DIndex]$ is greater than or equal to $\text{minBackupLevel}[i]$, some saving can still be made. Past assignments before $\text{minBackupLevel}[i]$ are unchanged and the shallowest level of inconsistency with the current assignment was deeper than this level, so consistency checks with assignments before $\text{minBackupLevel}[i]$ must succeed. Only checks with the levels between $\text{minBackupLevel}[i]$ and i exclusive are necessary (lines 6-7). If a constraint check fails, its level is recorded in $\text{mark}[i, DIndex]$.

If D_i is exhausted, chronological backtracking occurs. Since the algorithm is about to backtrack to level $i - 1$, it is known that the assignment at that level will change. Hence, $\text{minBackupLevel}[i]$ is updated to $i - 1$ (line 12). In addition, the minBackupLevel of the levels deeper than i may be updated if $i - 1$ is less than their current value (lines 13-14).

A.5 Revise()

The **Revise()** procedure, shown in Figure A.5, embodies the process of enforcing local arc consistency. **Revise()** applied to $\text{arc}(i, j)$ traverses D_i (line 3), testing each d_{i_a} against all elements of D_j (line 4). If no support for D_{i_a} exists, d_{i_a} is filtered from D_i (line 5). **Revise()** returns **True** if D_i is changed and **False** if not (lines 6, 7).

A.6 AC-1()

Figure A.6 (adapted from [98]) describes the simple **AC-1()** algorithm for enforcing arc consistency. The procedure **NC()** enforces node consistency (line 3) and Q is a queue of arcs to be processed (line 4). Every arc in the queue is

```

01. Procedure Revise( $i, j$ )
02.    $delete \leftarrow false$ 
03.   ForEach  $d_{i_a} \in D_i$ 
04.     If no  $d_{j_b} \in D_j$  Satisfies  $c(x_i, x_j)$ 
05.       Remove  $d_{i_a}$  From  $D_i$ 
06.        $delete \leftarrow true$ 
07.   Return  $delete$ 
08. End Procedure

```

Fig. A.5. Enforcing Arc Consistency via *Revise*()

then revised until none of the revisions produces any domain filtering (lines 5-9).

A.7 AC-3()

The popular AC-3() arc consistency enforcing algorithm is presented in Figure A.7. Q is initialised to contain the full set of arcs (line 4). Subsequently, arcs are selected and removed from Q (line 6) and *Revise*() is applied to them, enforcing local arc consistency (line 7). AC-3 gains in efficiency since arcs are not added to the queue for further revision unless they are possibly affected by the revision just performed. Note the special case where enforcing arc consistency on $arc(i, j)$ cannot violate arc consistency on $arc(j, i)$ (line 8): all elements removed from D_i were removed exactly because there were no corresponding values in D_j .

A.8 AC-1/4()

The AC-1/4() algorithm is presented in Figure A.8. The procedure *Empty*() returns *True* if the input domain is empty (line 5). AC-1/4() revises all constraint arcs from future variables to the current variable (lines 3, 4), as shown

```

01. Procedure AC-1()
02.   For  $i \leftarrow 1$  To  $n$ 
03.     NC( $i$ )
04.      $Q \leftarrow$  All  $arcs(i, j)$ 
05.     Repeat
06.        $change \leftarrow false$ 
07.       ForEach  $arc(i, j) \in Q$ 
08.          $change \leftarrow$  Revise( $i, j$ )  $\vee$   $change$ 
09.     Until Not  $change$ 
10. End Procedure

```

Fig. A.6. The AC-1() Algorithm

```

01. Procedure AC-3()
02.   For  $i \leftarrow 1$  To  $n$ 
03.     NC( $i$ )
04.    $Q \leftarrow$  All  $\text{arcs}(i, j)$ 
05.   While  $Q \neq \emptyset$ 
06.     Remove Some  $\text{arc}(i, j) \in Q$ 
07.     If  $\text{Revise}(i, j)$ 
08.       Add To  $Q$  All  $\text{arcs}(h, i)$  ( $h \neq i, h \neq j$ )
09. End Procedure

```

Fig. A.7. The AC-3() Algorithm

```

01. Procedure AC-1/4( $i$ )
02.    $wipe \leftarrow false$ 
03.   For  $j \leftarrow i + 1$  To  $n$  While Not  $wipe$ 
04.      $\text{Revise}(j, i)$ 
05.     If  $\text{empty}(D_j)$   $wipe \leftarrow true$ 
06.   Return  $wipe$ 
07. End Procedure

```

Fig. A.8. The AC-1/4() Procedure

in Figure 2.15a. This renders all checks with past variables redundant: any remaining domain element at the current level, i , must be consistent with a previous assignment at level h because of the revision done on $\text{arc}(i, h)$ when considering level h .

A.9 Branch and Bound

The branch and bound algorithm is presented in Figure A.9. The global variable, α keeps track of the quality of the best solution found thus far. This version of the algorithm considers an additive mechanism for calculating consistency, with the goal to minimise this calculation (lines 4-6). If a solution is discovered that is better than the best currently known, it is output and α is updated (lines 8-10).


```

01. Procedure BB( $i$ )
02.   Foreach  $d_{i_j} \in D_i$ 
03.     assign( $x_i, d_{i_j}$ )
04.      $consistency \leftarrow 0$ 
05.     For  $h \leftarrow 1$  To  $i - 1$  While  $consistency < \alpha$ 
06.        $consistency \leftarrow consistency + \mathbf{test}(i, h)$ 
07.       If  $consistency < \alpha$ 
08.         If  $i = n$ 
09.           showSolution()
10.            $\alpha \leftarrow consistency$ 
11.         Else BB( $i + 1$ )
12.   Return false
13. End Procedure

```

Fig. A.9. The Branch and Bound Algorithm

B Proofs

This appendix contains soundness and completeness proofs for the three novel algorithms developed in this thesis: Flexible Local Changes, GP-rrDCSP and Flexible Graphplan.

B.1 Soundness and Completeness of FLC

Theorems 1-3 below express the soundness and completeness properties of the three main FLC procedures. Theorem 1 is a direct consequence of Theorem 2, which is in turn a direct consequence of Theorem 3.

Theorem 1 *Given the set, A_1 , of complete assignments to $X = X_1 \cup X_2 \cup X_3$, if the current problem in the DFCSP is consistent, $\text{FLCSolve}()$ returns a complete assignment, $a_i \in A$, such that $\neg \exists a_j \in A_1, \text{cons}(a_j) > \text{cons}(a_i)$. If the current problem is inconsistent $l_\perp$ is returned.*

Theorem 2 *Given a set of assignments, A_2 to $X_1 \cup X_2 \cup X_3$ such that for each $a \in A_2$, $a \downarrow_{X_1} = \text{assignments}(X_1)$ ¹, $\text{FLCVariables}()$ returns an assignment, $a_i \in A_2$ such that $\neg \exists a_j \in A_2, \text{cons}(a_j) > \text{cons}(a_i)$. If $A_2 = \emptyset$, $l_\perp$ is returned.*

Theorem 3 *Given a set of assignments A_3 to $X_1 \cup X_2 \cup \{x_i\}$ such that for each $a \in A_3$, $a \downarrow_{X_1} = \text{assignments}(X_1)$, $\text{FLCVariable}()$ returns an assignment, $a_i \in A_3$ such that $\neg \exists a_j \in A_3, \text{cons}(a_j) > \text{cons}(a_i)$. If $A_3 = \emptyset$, $l_\perp$ is returned.*

A proof of Theorems 1-3 follows in terms of $|X_{23}| = |X_2 \cup X_3|$, the cardinality of the set of variables without fixed assignments.

Consider first the case $|X_{23}| = 1$. Part a) of this case is where $X_2 = \emptyset$ and $X_3 = \{x_i\}$. Given the assignment of d_{i_j} to x_i , $\text{cons}X_1x_i$ is incrementally calculated by considering the value of $\text{cons}X_1$ and that returned by $\text{test}X_1()$. All assignments to x_i which mean that $\text{cons}X_1x_i < \max(\text{bestcons}, rc)$ are discarded immediately, since the assignments to X_1 are fixed so the consistency cannot be improved. The use of bestcons restricts the selection of an assignment to one which maximises the consistency of the assigned variables, and the comparison with rc restricts the set of assignments considered to the set A_3 (as defined in Theorem 3). Given $X_2 = \emptyset$, $\text{cons}X_1x_i = \text{cons}X_{12}x_i$ and

¹ $a \downarrow_{X_i}$ restricts an assignment to the variables in X_i

no repair is possible (since there are no variables which may be re-assigned). In the case where $A_3 = \emptyset$, all possible assignments have a consistency below rc and are rejected, hence $l_\perp$ is returned.

Part b) of this case is where $X_2 = \{x_i\}$ and $X_3 = \emptyset$. This is only possible given a single-variable problem with an assignment to x_i received from the solution to the previous problem in the sequence. If this assignment is optimal, *initSolution* is correctly returned. Otherwise, *initTest()* detects the sub-optimal assignment and x_i is unassigned. The situation is now identical to part a). Therefore Theorem 3 and consequently Theorems 2 and 1 are proven in this case.

Consider now the general case where $|X_{23}| = k$, assuming that Theorems 1-3 hold for $|X_{23}| < k$. Having calculated $consX_1x_i$ as above, $consX_{12}x_i$ is also obtained incrementally through the use of the procedure *testX₁₂*(x_i). This procedure also creates the set C_r containing constraints whose consistency is below $\min(consX_{12}, consX_1x_i)$ for subsequent repair; $consX_{12}$ is the consistency prior to the assignment of x_i , $consX_1x_i$ represents $consX_1$ for the sub-problem of repairing the constraints in C_r with respect to x_i . Neither can be improved by further assignments (guaranteed by the conjunctive combination of constraint satisfaction degrees). Hence, all constraints whose consistency is at least equal to this level need not be repaired. In addition, the testing of elements of D_i may be halted as soon as a d_{ij} is found to have achieved $consX_{12}$. Again, the use of *bestcons* ensures the selection of an assignment to one which maximises the consistency of the assigned variables.

If $C_r \neq \emptyset$, *FLCRepair()* initiates repairs on X_2 . *choose()* is used to decide which variables to unassign, given the contents of C_r , and *FLCVariables()* is invoked on the sub-problem of re-assigning these variables. *FLCVariable()* is then called with $|X_{23}| = k - 1$, allowing the use of the induction assumption. In the case where $A_3 = \emptyset$, again, all possible assignments have a consistency below rc and are rejected and $l_\perp$ is returned. Hence Theorem 3 and consequently Theorems 2 and 1 are proven in the general case. $\square$

B.2 Soundness and Completeness of GP-rrDCSP

Theorem 4 *Given a locally inconsistent sub-problem, the set of propositions (represented by CSP variables) contained in a memoset generated by the constructMemoSet() procedure have no set of mutually-consistent supporting actions (represented by CSP domain elements) at the same level, i.e. they are conjunctively unsupportable locally.*

proof If a variable, x_i , is inconsistent, i.e. conjunctively unsupportable locally, with respect to a subset of variables, X_s , and this inconsistency is provable by attempting to enforce arc consistency, one of two situations are possible. Situation 1: constraints on $\{x_i\} \cup X_s$ disallow one or more assignment tuples to the variables in X_s because one or more domain elements of the variables in X_s are incompatible with every element in D_i . x_i forms

an eliminating explanation for each such domain element, which can be filtered from consideration. Situation 2: constraints on $\{x_i\} \cup X_s$ are such that each element in D_i is incompatible with the entire domain of one or more variables in X_s , leaving D_i empty. The `constructMemoSet()` procedure is invoked upon the discovery of the first such empty domain, and the associated variable always forms the basis for the output memoset. Hence, for all other variables which are part of the locally inconsistent set, situation 1 must be the case. `ConstructMemoSet()` adds all variables to a partial memoset which form eliminating explanations for elements of that partial memoset. $\square$

Theorem 5 *Given a locally inconsistent sub-problem, the set of propositions (represented by CSP variables) contained in a memoset generated by shallow learning during search have no set of mutually-consistent supporting actions (represented by CSP domain elements) at the same level, i.e. they are conjunctively unsupportable locally.*

proof At each dead end in the search, the conflicting variables are determined via shallow learning and recorded as a partial memoset. Since the Local Changes algorithm is complete, all dead ends necessary to establish the inconsistency of the problem are examined. The output memoset is the union of all recorded partial memosets. It must therefore contain a set of variables, all assignments or sub-assignments to which lead to a dead end. This set of variables represents a set of propositions which are conjunctively unsupportable locally. $\square$

Theorem 6 *Given a set of sub-goal propositions and a subgraph, G , of a planning graph, GP-rrDCSP either identifies a subgraph of G connecting the sub-goals to the initial conditions such that the actions at each level are mutually-consistent, or returns one or more memosets containing subsets of the sub-goal propositions that are conjunctively unsupportable.*

proof If G contains just the initial conditions, the theorem is trivially true: if the sub-goals match the initial conditions, success is returned, otherwise the set of sub-goals are recorded as a memoset and returned.

Assume now that the theorem is true when G contains i levels, and consider the case where G has $i + 1$ -levels. The sub-goals and their supporting actions form the initial problem in an rrDCSP sub-sequence. If the sub-goals are a superset of one or more memosets previously recorded at this level, those memosets are returned.

If the initial sub-problem is unsolvable, this fact is established by arc consistency pre-processing or during the search process. In either case, a valid memoset is returned by Theorems 4 and 5. If the sub-problem is solvable, the mutually-consistent action nodes constituting the solution can be straightforwardly mapped to their precondition propositions using the precondition relations in the planning graph. Using the induction assumption, either success is returned if a consistent subgraph of G connecting these actions to the initial conditions has been found, or one or more valid memosets are returned, each containing a conjunctively unsupportable set of propositions at $level_i$.

In the latter case, the precondition relations in the planning graph can again be used straightforwardly to map the memoset onto one or more subsets of the solution assignment at $level_{i+1}$. A single such subset is selected and used to create the next problem in the sub-sequence via constraint restriction. The same subset is mapped to the proposition nodes at $level_{i+1}$ to form a partial memoset. Completeness is not compromised by these choices, since all combinations of assignments not posted as constraints may be selected in subsequent sub-problems.

This process continues until either a solution to a sub-problem in the sub-sequence can be extended to a consistent sub-graph of G including the initial conditions, or a sub-problem is created that is unsolvable. At this point, since Local Changes is complete and the constraints posted are valid by the induction assumption, all avenues for searching for support for the sub-goal propositions at $level_{i+1}$ have been explored. The output memoset is the union of all partial memosets associated with each posted constraint. It therefore contains a set of propositions all combinations or sub-combinations of whose supporting actions cannot be extended to form a consistent subgraph connecting the propositions at $level_{i+1}$ to the initial conditions, i.e. that are conjunctively unsupportable. $\square$

If G encompasses the entire planning graph, then Theorem 6 can be seen to express the soundness and completeness of GP-rrDCSP.

B.3 Soundness and Completeness of Flexible Graphplan

Theorem 7 *For a given sub-problem, an l_b -memoset is also an l_c -memoset, where $l_b \leq l_c$.*

proof All support for the propositions in an l_b -memoset has been shown to form part of a subgraph connecting the propositions to the initial conditions with satisfaction degree of at most l_a , where $l_a < l_b$. The theorem is therefore trivially true by transitivity. $\square$

Theorem 8 *Fuzzy node consistency: if there exists a variable, x_i , all of whose domain elements have unary satisfaction degree of at most l_a , the proposition represented by x_i is a unary l_b -memoset, where l_b is the adjacent satisfaction degree above l_a in L .*

proof A solution to a sub-problem consists of an assignment to every variable. Hence, an assignment must be made to x_i that has a unary satisfaction degree of at most l_a . Given that the conjunctive combination of satisfaction degrees is implemented via the *min* operator, this assignment guarantees a solution with satisfaction degree at most l_a . Since the domain has not been revised with respect to any other variables, the proposition represented by x_i is therefore a unary l_b -memoset. $\square$

Theorem 9 *Assuming all unary l_b -memosets have been established via fuzzy node consistency, where $l_a < l_b \leq l_{\top}$: given a sub-problem with*

no solution of at least satisfaction degree l_a , the set of propositions (represented by CSP variables) contained in an l_a -memoset generated by the `constructMemoSet $l_a$ ()` procedure have no set of supporting actions (represented by CSP domain elements) with conjunctive satisfaction degree greater than or equal to l_a .

proof Given a variable subset X_s with a conjunctive satisfaction degree l_s such that $l_s \geq l_a$, if a variable x_i is considered such that $\{x_i\} \cup X_s$ has a conjunctive satisfaction degree of less than l_a , one of two situations is possible. Situation 1: the conjunctive combination of one or more domain elements of the variables in X_s and any element in D_i has a satisfaction degree of less than l_a . x_i forms a reduction explanation for each such domain element. Situation 2: the conjunctive combination of all domain elements in D_i and any element of the domain of at least one variable in X_s has a satisfaction degree of less than l_a , leaving all elements of D_i with unary consistency degree of less than l_a by propagation. The `constructMemoSet $l_a$ ()` procedure is invoked upon the discovery of the first such domain, and x_i always forms the basis for the output l_a -memoset in this case. Hence, for all other variables that are part of the set whose conjunctive satisfaction degree is less than l_a , situation 1 must be the case. `constructMemoSet $l_a$ ()` adds all variables to a partial l_a -memoset that form l_a -reduction explanations for elements of that partial memoset $\square$.

Theorem 10 *Given a sub-problem with no solution of at least satisfaction degree l_a , the set of propositions (represented by CSP variables) contained in a l_a -memoset generated by FGP during search has no set of supporting actions (represented by CSP domain elements) with conjunctive satisfaction degree, l_b such that $l_b \geq l_a$.*

proof If all potential assignments to the variable currently under consideration, x_i lead to a partial assignment with satisfaction degree less than l_a , x_i forms a l_a -conflict set with the currently assigned variables which is recorded as a partial l_a -memoset. Since Flexible Local Changes is complete, all conflicts necessary to establish that the sub-problem does not have a solution of at least satisfaction degree l_a are explored. The output l_a -memoset is the union of all recorded partial l_a -memosets. It must therefore contain a set of variables, all assignments or sub-assignments to which lead to an l_a -conflict. Hence, the corresponding set of propositions have no set of supporting actions with conjunctive satisfaction degree greater or equal to l_a . $\square$

Theorem 11 *Given a set of sub-goal propositions, a subgraph, G_f , of a flexible planning graph, and the satisfaction degree, l_α , of the best known plan found so far, FGP identifies the subgraph of G_f (if it exists) connecting the sub-goals to the initial conditions whose constituent actions have a satisfaction degree l_a , where $l_a > l_\alpha$ such that there does not exist an alternative subgraph of G_f that connects the sub-goals and the initial conditions with satisfaction degree l_b , where $l_a < l_b$. FGP also returns one or more memo-*

sets per satisfaction degree l_c , where $l_a < l_c \leq l_\top$, each containing a set of propositions that are conjunctively unsupportable to l_c .

proof If the flexible planning graph contains just the initial conditions, Theorem 11 is trivially true: if the initial conditions match the flexible goal combination, a subgraph has been successfully identified. Otherwise, the sub-goals are returned as an l_a -memoset, where $l_a > l_\alpha$. From Theorem 7, an l_a -memoset is also an l_b -memoset for the same sub-problem, where $l_a \leq l_b \leq l_\top$.

Assume that Theorem 11 is true when G_f contains i levels, and consider the case where G_f contains $i + 1$ levels. The sub-goals and their supporting actions form the initial problem in an *rrDFCSP* sub-sequence. If the sub-goals are a superset of one or more l_m -memosets, where $l_\alpha < l_m \leq l_\top$, then it is already known that these sub-goals cannot be supported by a subgraph with satisfaction degree l_m . It is not necessary, therefore, to produce memosets for satisfaction degrees greater or equal to the lowest value of l_m .

If the initial problem has no solution of at least satisfaction degree l_u , where $l_u < l_m$, this fact is established by a combination of enforcing fuzzy arc consistency pre-processing and search. From Theorems 9 and 10, valid memosets are produced and recorded for each such value of l_u . If $l_u \leq l_\alpha$ it has been established that the sub-goals are not a constituent of a plan better than the best currently known and the memosets identified/recorded for the satisfaction degrees greater than l_α are returned.

By Theorems 1-3, FLC is guaranteed to find the solution with the greatest satisfaction degree l_s , where $l_\alpha < l_s < l_u$ if one exists. The mutually-consistent (to degree l_s) action nodes constituting the solution can be straightforwardly mapped to their precondition propositions using the precondition relations in the flexible planning graph. By the induction assumption, one or more memosets are returned for each satisfaction degree above the current value of l_α at which a consistent subgraph of G_f cannot be found based on this set of actions.

Precondition relations in the flexible planning graph are again used to map each returned memoset back to combinations of action nodes matching a sub-assignment of the solution to the sub-problem at *level* _{$i+1$} . To maintain completeness, a single constraint is posted for the lowest value of l_s . For all values of l_s , the sub-assignment tuples are stored. This process continues until no solution exists with a satisfaction degree greater than the current value of l_α . Since FLC is complete, and the constraints posted are valid by the induction assumption, all avenues for searching for support for the sub-goals have been explored.

At this point, stored tuples for each remaining value of l_s are mapped to nodes in *propositions* _{$i+1$} . An l_s -memoset is formed, for each value of l_s by taking the union of each such propositional combination. Memosets are returned for satisfaction degrees l_m (those that were already known at *level* _{$i+1$}), l_u (those derived locally) and l_s (those derived by propagation). In the last case, the memosets are the union of all partial memosets associated with

each memoset of the same satisfaction degree propagated from $level_i$. They therefore contain a set of propositions, all combinations or sub-combinations of whose supporting actions cannot form part of a subgraph of G_f with satisfaction degree l_s connecting the goal propositions to the initial conditions.

□

Theorem 12 *Given a flexible planning graph and a set of flexible goal combinations, FGP identifies a plan consisting of the subgraph containing one such goal combination such that the satisfaction degree of the conjunction of the plan goals and the subgraph is optimised.*

proof From Theorem 11, the algorithm identifies the subgraph with the optimal satisfaction degree based on the propositions contained in a goal combination. Hence, l_α is a true reflection of the optimal plan satisfaction degree given the goal combinations examined thus far. FGP considers each goal combination in descending order, terminating when the satisfaction degree, l_g , of the current goal combination is such that $l_g \leq l_\alpha$. Since the remaining goal combinations have a satisfaction degree of at most l_g , no plan with a better overall satisfaction degree exists, and the algorithm terminates correctly. □

C An Empirical Study of Fuzzy *rr*DFCSPs: Complete Results

Due to the size of this appendix, it is held on the following web page:
<http://springer.co.uk/DFCSP/>

D Planning Problems

This appendix contains the flexible planning problems as used to evaluate FGP in Chapter 8.

D.1 The Test Suite

This section gives the details of the test suite of 12 problems engineered to contain plans with different satisfaction degrees at a range of lengths. Note that flexible versions only of the problem descriptions themselves are shown. In the Boolean case, the less valuable package is set to being not valuable.

D.1.1 Domain Operators

Scales

```
(SatScale lbot l1 l2 ltop)
(TruthScale kbot k1 k2 ktop)
```

Common Operators

```
(operator LOAD-AIRPLANE
  (params (?o PACKAGE) (?a AIRPLANE) (?l LOC))
  {when (preconds (at ?a ?l) (at ?o ?l))
    (effects (in ?o ?a) (not (at ?o ?l)))
    ltop}
)

(operator UNLOAD-TRUCK
  (params (?v TRUCK) (?p PACKAGE) (?l LOC))
  {when (preconds (at ?v ?l) (on ?p ?v))
    (effects (not (on ?p ?v)) (at ?p ?l))
    ltop}
)

(operator UNLOAD-AIRPLANE
  (params (?a AIRPLANE) (?o PACKAGE) (?l LOC))
```

```

{when (preconds (at ?a ?l ) (in ?o ?a ))
  (effects (not (in ?o ?a)) (at ?o ?l))
  ltop}
)

(operator FLY-AIRPLANE
  (params (?a AIRPLANE) (?loc-from AIRPORT) (?loc-to AIRPORT))
  {when (preconds (at ?a ?loc-from ))
    (effects (not (at ?a ?loc-from)) (at ?a ?loc-to))
    ltop}
  )

(operator GUARD-GETS-ON-TRUCK
  (params (?v TRUCK) (?g GUARD) (?l LOC))
  {when (preconds (at ?v ?l ) (at ?g ?l ))
    (effects (on ?g ?v) (not (at ?g ?l)))
    ltop}
  )

(operator GUARD-GETS-OFF-TRUCK
  (params (?v TRUCK) (?g GUARD) (?l LOC))
  {when (preconds (at ?v ?l ) (on ?g ?v ))
    (effects (not (on ?g ?v)) (at ?g ?l))
    ltop}
  )

(operator GUARD-GETS-ON-AIRPLANE
  (params (?v AIRPLANE) (?g GUARD) (?l LOC))
  {when (preconds (at ?v ?l ) (at ?g ?l ))
    (effects (on ?g ?v) (not (at ?g ?l)))
    ltop}
  )

(operator GUARD-GETS-OFF-AIRPLANE
  (params (?v AIRPLANE) (?g GUARD) (?l LOC))
  {when (preconds (at ?v ?l ) (on ?g ?v ))
    (effects (not (on ?g ?v)) (at ?g ?l))
    ltop}
  )

```

Boolean Operator Versions

```

(operator LOAD-TRUCK
  (params (?t TRUCK) (?p PACKAGE) (?l LOC) (?g GUARD))
  {when (preconds (at ?t ?l ) (at ?p ?l )
    (not (valuable ?p)))
    (effects (on ?p ?t) (not (at ?p ?l)))
    ltop}
  )

```

```

{when (preconds (at ?t ?l ) (at ?p ?l ) (on ?g ?t )
                (valuable ?p))
  (effects (on ?p ?t) (not (at ?p ?l)))
  ltop}
)

(operator DRIVE-TRUCK
  (params (?v TRUCK) (?l1 LOC) (?l2 LOC) (?r1 MAJOR-ROAD)
           (?r2 MINOR-ROAD) (?ct CITY))
  {when (preconds (at ?v ?l1 ) (connects ?r1 ?l1 ?l2 ))
    (effects (not (at ?v ?l1)) (at ?v ?l2))
    ltop}
  {when (preconds (at ?v ?l2 ) (connects ?r1 ?l1 ?l2 ))
    (effects (not (at ?v ?l2)) (at ?v ?l1))
    ltop}
  )
)

```

Flexible Operator Versions

```

(operator LOAD-TRUCK
  (params (?t TRUCK) (?p PACKAGE) (?l LOC) (?g GUARD))
  {when (preconds (at ?t ?l ) (at ?p ?l )
                (valuable ?p <= k1))
    (effects (on ?p ?t) (not (at ?p ?l)))
    ltop}
  {when (preconds (at ?t ?l ) (at ?p ?l ) (on ?g ?t )
                (valuable ?p >= k2))
    (effects (on ?p ?t) (not (at ?p ?l)))
    ltop}
  {when (preconds (at ?t ?l ) (at ?p ?l )
                (valuable ?p >= k2) (not (on ?g ?t)))
    (effects (on ?p ?t) (not (at ?p ?l)))
    l2}
  )
)

(operator DRIVE-TRUCK
  (params (?v TRUCK) (?l1 LOC) (?l2 LOC) (?r1 MAJOR-ROAD)
           (?r2 MINOR-ROAD) (?ct CITY))
  {when (preconds (at ?v ?l1 ) (connects ?r1 ?l1 ?l2 ))
    (effects (not (at ?v ?l1)) (at ?v ?l2))
    ltop}
  {when (preconds (at ?v ?l1 ) (connects ?r2 ?l1 ?l2 ))
    (effects (not (at ?v ?l1)) (at ?v ?l2))
    l1}
  {when (preconds (at ?v ?l2 ) (connects ?r1 ?l1 ?l2 ))
    (effects (not (at ?v ?l2)) (at ?v ?l1))
    ltop}
  {when (preconds (at ?v ?l2 ) (connects ?r2 ?l1 ?l2 ))

```

```

    (effects (not (at ?v ?l2)) (at ?v ?l1))
  l1}
)

```

D.1.2 Problem 1

```

(edinburgh CITY)
(london CITY)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(edinburgh-po LOC)
(london-po LOC)
(edinburgh-ie LOC)
(london-ie LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(london-airport AIRPORT)
(london-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MINOR-ROAD)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(edinburgh-truck TRUCK)
(london-truck TRUCK)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)
(package2 PACKAGE)

(preconds
  (connects edinburgh-road1 edinburgh-airport edinburgh-ie)
  (connects edinburgh-road2 edinburgh-ie edinburgh-po)
  (connects edinburgh-road3 edinburgh-airport edinburgh-po)
  (connects london-road1 london-airport london-ie)
  (connects london-road2 london-ie london-po)
  (connects london-road3 london-airport london-po)
  (not (on guard1 edinburgh-truck))
  (not (on guard1 london-truck))
  (not (on guard1 airplane1))
)

```

```

(connects la-road1 la-airport la-ie)
(connects la-road2 la-ie la-po)
(connects la-road3 la-airport la-po)
(at la-truck la-airport)
(valuable package1 ktop)
(valuable package2 k1)
(at airplane1 london-airport)
(at london-truck london-airport)
(at edinburgh-truck edinburgh-po)
(at guard1 edinburgh-airport)
(at package1 london-airport)
(at package2 la-airport)
)
(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.3 Problem 2

```

(edinburgh CITY)
(london CITY)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(package2 PACKAGE)
(edinburgh-po LOC)
(london-po LOC)
(edinburgh-ie LOC)
(london-ie LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(london-airport AIRPORT)
(london-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MINOR-ROAD)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(edinburgh-truck TRUCK)
(london-truck TRUCK)

```

```

(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
  (connects edinburgh-road1 edinburgh-airport edinburgh-ie)
  (connects edinburgh-road2 edinburgh-ie edinburgh-po)
  (connects edinburgh-road3 edinburgh-airport edinburgh-po)
  (connects london-road1 london-airport london-ie)
  (connects london-road2 london-ie london-po)
  (connects london-road3 london-airport london-po)
  (not (on guard1 edinburgh-truck))
  (not (on guard1 london-truck))
  (not (on guard1 airplane1))
  (connects la-road1 la-airport la-ie)
  (connects la-road2 la-ie la-po)
  (connects la-road3 la-airport la-po)
  (at la-truck la-airport)
  (valuable package2 k1)
  (at package2 la-airport)
  (valuable package1 ktop)
  (at airplane1 london-airport)
  (at london-truck london-airport)
  (at edinburgh-truck edinburgh-po)
  (at guard1 edinburgh-po)
  (at package1 london-airport)
)

(effects
  (goal p1 (when (at package1 london-po ) ltop))
  (goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.4 Problem 3

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)

```


(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)

(london CITY)
(london-po LOC)
(london-ie LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-truck TRUCK)

(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)

(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
 (connects edinburgh-road1 edinburgh-airport edinburgh-1)
 (connects edinburgh-road2 edinburgh-1 edinburgh-2)
 (connects edinburgh-road3 edinburgh-2 edinburgh-3)
 (connects edinburgh-road4 edinburgh-3 edinburgh-4)
 (connects edinburgh-road5 edinburgh-4 edinburgh-5)
 (connects edinburgh-road6 edinburgh-5 edinburgh-6)
 (connects edinburgh-road7 edinburgh-6 edinburgh-7)
 (connects edinburgh-road8 edinburgh-7 edinburgh-8)

```

(connects edinburgh-road9 edinburgh-8 edinburgh-9)
(connects edinburgh-road10 edinburgh-9 edinburgh-po)
(connects london-road1 london-airport london-ie)
(connects london-road2 london-ie london-po)
(connects london-road3 london-airport london-po)
(connects la-road1 la-airport la-ie)
(connects la-road2 la-ie la-po)
(connects la-road3 la-airport la-po)
(not (on guard1 edinburgh-truck))
(not (on guard1 london-truck))
(not (on guard1 la-truck))
(not (on guard1 airplane1))
(at guard1 edinburgh-po)
(at la-truck la-airport)
(at london-truck london-airport)
(at edinburgh-truck edinburgh-po)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-airport)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.5 Problem 4

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)

```

```

(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
(connects edinburgh-road1 edinburgh-airport edinburgh-1)
(connects edinburgh-road2 edinburgh-1 edinburgh-2)
(connects edinburgh-road3 edinburgh-2 edinburgh-3)
(connects edinburgh-road4 edinburgh-3 edinburgh-4)
(connects edinburgh-road5 edinburgh-4 edinburgh-5)
(connects edinburgh-road6 edinburgh-5 edinburgh-6)
(connects edinburgh-road7 edinburgh-6 edinburgh-7)
(connects edinburgh-road8 edinburgh-7 edinburgh-8)
(connects edinburgh-road9 edinburgh-8 edinburgh-9)
(connects edinburgh-road10 edinburgh-9 edinburgh-po)
(connects london-road1 london-airport london-ie)
(connects london-road2 london-ie london-1)

```

```

(connects london-road4 london-1 london-2)
(connects london-road5 london-2 london-po)
(connects london-road3 london-airport london-po)
(connects la-road1 la-airport la-ie)
(connects la-road2 la-ie la-po)
(connects la-road3 la-airport la-po)
(not (on guard1 edinburgh-truck))
(not (on guard1 london-truck))
(not (on guard1 la-truck))
(not (on guard1 airplane1))
(at guard1 edinburgh-8)
(at la-truck la-airport)
(at london-truck london-airport)
(at edinburgh-truck edinburgh-8)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-airport)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.6 Problem 5

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)

```

```

(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
(connects edinburgh-road1 edinburgh-airport edinburgh-1)
(connects edinburgh-road2 edinburgh-1 edinburgh-2)
(connects edinburgh-road3 edinburgh-2 edinburgh-3)
(connects edinburgh-road4 edinburgh-3 edinburgh-4)
(connects edinburgh-road5 edinburgh-4 edinburgh-5)
(connects edinburgh-road6 edinburgh-5 edinburgh-6)
(connects edinburgh-road7 edinburgh-6 edinburgh-7)
(connects edinburgh-road8 edinburgh-7 edinburgh-8)
(connects edinburgh-road9 edinburgh-8 edinburgh-9)
(connects edinburgh-road10 edinburgh-9 edinburgh-po)

```

```

(connects london-road1 london-airport london-ie)
(connects london-road2 london-ie london-1)
(connects london-road4 london-1 london-2)
(connects london-road5 london-2 london-3)
(connects london-road6 london-3 london-4)
(connects london-road7 london-4 london-po)
(connects london-road3 london-airport london-po)
(connects la-road1 la-airport la-ie)
(connects la-road2 la-ie la-po)
(connects la-road3 la-airport la-po)
(not (on guard1 edinburgh-truck))
(not (on guard1 london-truck))
(not (on guard1 la-truck))
(not (on guard1 airplane1))
(at guard1 edinburgh-6)
(at la-truck la-airport)
(at london-truck london-airport)
(at edinburgh-truck edinburgh-6)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-airport)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.7 Problem 6

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)

```

(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)
(preconds
 (connects edinburgh-road1 edinburgh-airport edinburgh-1)
 (connects edinburgh-road2 edinburgh-1 edinburgh-2)
 (connects edinburgh-road3 edinburgh-2 edinburgh-3)
 (connects edinburgh-road4 edinburgh-3 edinburgh-4)
 (connects edinburgh-road5 edinburgh-4 edinburgh-5)
 (connects edinburgh-road6 edinburgh-5 edinburgh-6)
 (connects edinburgh-road7 edinburgh-6 edinburgh-7)

```

(connects edinburgh-road8 edinburgh-7 edinburgh-8)
(connects edinburgh-road9 edinburgh-8 edinburgh-9)
(connects edinburgh-road10 edinburgh-9 edinburgh-po)
(connects london-road1 london-airport london-ie)
(connects london-road2 london-ie london-1)
(connects london-road4 london-1 london-2)
(connects london-road5 london-2 london-3)
(connects london-road6 london-3 london-4)
(connects london-road7 london-4 london-po)
(connects london-road3 london-airport london-3)
(connects la-road1 la-airport la-ie)
(connects la-road2 la-ie la-po)
(connects la-road3 la-airport la-po)
(not (on guard1 edinburgh-truck))
(not (on guard1 london-truck))
(not (on guard1 la-truck))
(not (on guard1 airplane1))
(at guard1 edinburgh-6)
(at la-truck la-airport)
(at london-truck london-airport)
(at edinburgh-truck edinburgh-6)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-airport)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.8 Problem 7

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)

```


(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-5 LOC)
(london-6 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-road8 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)

(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

```

(preconds
  (connects edinburgh-road1 edinburgh-airport edinburgh-1)
  (connects edinburgh-road2 edinburgh-1 edinburgh-2)
  (connects edinburgh-road3 edinburgh-2 edinburgh-3)
  (connects edinburgh-road4 edinburgh-3 edinburgh-4)
  (connects edinburgh-road5 edinburgh-4 edinburgh-5)
  (connects edinburgh-road6 edinburgh-5 edinburgh-6)
  (connects edinburgh-road7 edinburgh-6 edinburgh-7)
  (connects edinburgh-road8 edinburgh-7 edinburgh-8)
  (connects edinburgh-road9 edinburgh-8 edinburgh-9)
  (connects edinburgh-road10 edinburgh-9 edinburgh-po)
  (connects london-road1 london-airport london-ie)
  (connects london-road2 london-ie london-1)
  (connects london-road4 london-1 london-2)
  (connects london-road5 london-2 london-3)
  (connects london-road6 london-3 london-4)
  (connects london-road7 london-4 london-5)
  (connects london-road8 london-5 london-6)
  (connects london-road9 london-6 london-po)
  (connects london-road3 london-airport london-5)
  (connects la-road1 la-airport la-ie)
  (connects la-road2 la-ie la-po)
  (connects la-road3 la-airport la-po)
  (not (on guard1 edinburgh-truck))
  (not (on guard1 london-truck))
  (not (on guard1 la-truck))
  (not (on guard1 airplane1))
  (at guard1 edinburgh-4)
  (at la-truck la-airport)
  (at london-truck london-airport)
  (at edinburgh-truck edinburgh-4)
  (valuable package2 k1)
  (at package2 la-airport)
  (valuable package1 ktop)
  (at package1 london-airport)
  (at airplane1 london-airport)
)

(effects
  (goal p1 (when (at package1 london-po ) ltop))
  (goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.9 Problem 8

```

(edinburgh CITY)
(edinburgh-po LOC)

```

(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-5 LOC)
(london-6 LOC)
(london-7 LOC)
(london-8 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-road8 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-road10 MAJOR-ROAD)
(london-truck TRUCK)

```

(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)

(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
  (connects edinburgh-road1 edinburgh-airport edinburgh-1)
  (connects edinburgh-road2 edinburgh-1 edinburgh-2)
  (connects edinburgh-road3 edinburgh-2 edinburgh-3)
  (connects edinburgh-road4 edinburgh-3 edinburgh-4)
  (connects edinburgh-road5 edinburgh-4 edinburgh-5)
  (connects edinburgh-road6 edinburgh-5 edinburgh-6)
  (connects edinburgh-road7 edinburgh-6 edinburgh-7)
  (connects edinburgh-road8 edinburgh-7 edinburgh-8)
  (connects edinburgh-road9 edinburgh-8 edinburgh-9)
  (connects edinburgh-road10 edinburgh-9 edinburgh-po)
  (connects london-road1 london-airport london-ie)
  (connects london-road2 london-ie london-1)
  (connects london-road4 london-1 london-2)
  (connects london-road5 london-2 london-3)
  (connects london-road6 london-3 london-4)
  (connects london-road7 london-4 london-5)
  (connects london-road8 london-5 london-6)
  (connects london-road9 london-6 london-7)
  (connects london-road8 london-7 london-8)
  (connects london-road9 london-8 london-po)
  (connects london-road3 london-airport london-7)
  (connects la-road1 la-airport la-ie)
  (connects la-road2 la-ie la-po)
  (connects la-road3 la-airport la-po)
  (not (on guard1 edinburgh-truck))
  (not (on guard1 london-truck))
  (not (on guard1 la-truck))
  (not (on guard1 airplane1))
  (at guard1 edinburgh-2)
  (at la-truck la-airport)
  (at london-truck london-airport))

```

```

(at edinburgh-truck edinburgh-2)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-airport)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.10 Problem 9

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-5 LOC)

```

```

(london-6 LOC)
(london-7 LOC)
(london-8 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-road8 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-road10 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)

(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
  (connects edinburgh-road1 edinburgh-airport edinburgh-1)
  (connects edinburgh-road2 edinburgh-1 edinburgh-2)
  (connects edinburgh-road3 edinburgh-2 edinburgh-3)
  (connects edinburgh-road4 edinburgh-3 edinburgh-4)
  (connects edinburgh-road5 edinburgh-4 edinburgh-5)
  (connects edinburgh-road6 edinburgh-5 edinburgh-6)
  (connects edinburgh-road7 edinburgh-6 edinburgh-7)
  (connects edinburgh-road8 edinburgh-7 edinburgh-8)
  (connects edinburgh-road9 edinburgh-8 edinburgh-9)
  (connects edinburgh-road10 edinburgh-9 edinburgh-po)
  (connects london-road1 london-airport london-ie)
  (connects london-road2 london-ie london-1)
  (connects london-road4 london-1 london-2)
  (connects london-road5 london-2 london-3)

```

```

(connects london-road6 london-3 london-4)
(connects london-road7 london-4 london-5)
(connects london-road8 london-5 london-6)
(connects london-road9 london-6 london-7)
(connects london-road8 london-7 london-8)
(connects london-road9 london-8 london-po)
(connects london-road3 london-airport london-5)
(connects la-road1 la-airport la-ie)
(connects la-road2 la-ie la-po)
(connects la-road3 la-airport la-po)
(not (on guard1 edinburgh-truck))
(not (on guard1 london-truck))
(not (on guard1 la-truck))
(not (on guard1 airplane1))
(at guard1 edinburgh-2)
(at la-truck la-airport)
(at london-truck london-airport)
(at edinburgh-truck edinburgh-2)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-airport)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.11 Problem 10

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)

```

(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-5 LOC)
(london-6 LOC)
(london-7 LOC)
(london-8 LOC)
(london-9 LOC)
(london-10 LOC)
(london-11 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-road8 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-road10 MAJOR-ROAD)
(london-road11 MAJOR-ROAD)
(london-road12 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(package2 PACKAGE)


```

(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
  (connects edinburgh-road1 edinburgh-airport edinburgh-1)
  (connects edinburgh-road2 edinburgh-1 edinburgh-2)
  (connects edinburgh-road3 edinburgh-2 edinburgh-3)
  (connects edinburgh-road4 edinburgh-3 edinburgh-4)
  (connects edinburgh-road5 edinburgh-4 edinburgh-5)
  (connects edinburgh-road6 edinburgh-5 edinburgh-6)
  (connects edinburgh-road7 edinburgh-6 edinburgh-7)
  (connects edinburgh-road8 edinburgh-7 edinburgh-8)
  (connects edinburgh-road9 edinburgh-8 edinburgh-9)
  (connects edinburgh-road10 edinburgh-9 edinburgh-po)
  (connects london-road1 london-airport london-ie)
  (connects london-road2 london-ie london-1)
  (connects london-road4 london-1 london-2)
  (connects london-road5 london-2 london-3)
  (connects london-road6 london-3 london-4)
  (connects london-road7 london-4 london-5)
  (connects london-road8 london-5 london-6)
  (connects london-road9 london-6 london-7)
  (connects london-road8 london-7 london-8)
  (connects london-road9 london-8 london-9)
  (connects london-road10 london-9 london-10)
  (connects london-road11 london-10 london-11)
  (connects london-road12 london-11 london-po)
  (connects london-road3 london-airport london-8)
  (connects la-road1 la-airport la-ie)
  (connects la-road2 la-ie la-po)
  (connects la-road3 la-airport la-po)
  (not (on guard1 edinburgh-truck))
  (not (on guard1 london-truck))
  (not (on guard1 la-truck))
  (not (on guard1 airplane1))
  (at guard1 edinburgh-airport)
  (at la-truck la-airport)
  (at london-truck london-airport)
  (at edinburgh-truck edinburgh-2)
  (valuable package2 k1)
  (at package2 la-airport)
  (valuable package1 ktop)
  (at package1 london-airport)
  (at airplane1 london-airport)
)

```

```

(effects
  (goal p1 (when (at package1 london-po ) ltop))
  (goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.12 Problem 11

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)
(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-5 LOC)
(london-6 LOC)
(london-7 LOC)
(london-8 LOC)
(london-9 LOC)
(london-10 LOC)
(london-11 LOC)
(london-airport AIRPORT)
(london-airport LOC)

```

(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-road8 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-road10 MAJOR-ROAD)
(london-road11 MAJOR-ROAD)
(london-road12 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)
(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds

(connects edinburgh-road1 edinburgh-airport edinburgh-1)
(connects edinburgh-road2 edinburgh-1 edinburgh-2)
(connects edinburgh-road3 edinburgh-2 edinburgh-3)
(connects edinburgh-road4 edinburgh-3 edinburgh-4)
(connects edinburgh-road5 edinburgh-4 edinburgh-5)
(connects edinburgh-road6 edinburgh-5 edinburgh-6)
(connects edinburgh-road7 edinburgh-6 edinburgh-7)
(connects edinburgh-road8 edinburgh-7 edinburgh-8)
(connects edinburgh-road9 edinburgh-8 edinburgh-9)
(connects edinburgh-road10 edinburgh-9 edinburgh-po)
(connects london-road1 london-airport london-ie)
(connects london-road2 london-ie london-1)
(connects london-road4 london-1 london-2)
(connects london-road5 london-2 london-3)
(connects london-road6 london-3 london-4)
(connects london-road7 london-4 london-5)
(connects london-road8 london-5 london-6)
(connects london-road9 london-6 london-7)
(connects london-road8 london-7 london-8)

```

(connects london-road9 london-8 london-9)
(connects london-road10 london-9 london-10)
(connects london-road11 london-10 london-11)
(connects london-road12 london-11 london-po)
(connects london-road3 london-airport london-6)
(connects la-road1 la-airport la-ie)
(connects la-road2 la-ie la-po)
(connects la-road3 la-airport la-po)
(not (on guard1 edinburgh-truck))
(not (on guard1 london-truck))
(not (on guard1 la-truck))
(not (on guard1 airplane1))
(at guard1 edinburgh-airport)
(at la-truck la-airport)
(at london-truck london-airport)
(at edinburgh-truck edinburgh-2)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-airport)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po ) ltop))
(goal p2 (when (at package2 la-po ) ltop))
)

```

D.1.13 Problem 12

```

(edinburgh CITY)
(edinburgh-po LOC)
(edinburgh-ie LOC)
(edinburgh-1 LOC)
(edinburgh-2 LOC)
(edinburgh-3 LOC)
(edinburgh-4 LOC)
(edinburgh-5 LOC)
(edinburgh-6 LOC)
(edinburgh-7 LOC)
(edinburgh-8 LOC)
(edinburgh-9 LOC)
(edinburgh-airport AIRPORT)
(edinburgh-airport LOC)
(edinburgh-road1 MAJOR-ROAD)
(edinburgh-road2 MAJOR-ROAD)
(edinburgh-road3 MAJOR-ROAD)
(edinburgh-road4 MAJOR-ROAD)

```

(edinburgh-road5 MAJOR-ROAD)
(edinburgh-road6 MAJOR-ROAD)
(edinburgh-road7 MAJOR-ROAD)
(edinburgh-road8 MAJOR-ROAD)
(edinburgh-road9 MAJOR-ROAD)
(edinburgh-road10 MAJOR-ROAD)
(edinburgh-truck TRUCK)
(london CITY)
(london-po LOC)
(london-ie LOC)
(london-1 LOC)
(london-2 LOC)
(london-3 LOC)
(london-4 LOC)
(london-5 LOC)
(london-6 LOC)
(london-7 LOC)
(london-8 LOC)
(london-9 LOC)
(london-10 LOC)
(london-11 LOC)
(london-12 LOC)
(london-13 LOC)
(london-14 LOC)
(london-airport AIRPORT)
(london-airport LOC)
(london-road1 MAJOR-ROAD)
(london-road2 MAJOR-ROAD)
(london-road3 MINOR-ROAD)
(london-road4 MAJOR-ROAD)
(london-road5 MAJOR-ROAD)
(london-road6 MAJOR-ROAD)
(london-road7 MAJOR-ROAD)
(london-road8 MAJOR-ROAD)
(london-road9 MAJOR-ROAD)
(london-road10 MAJOR-ROAD)
(london-road11 MAJOR-ROAD)
(london-road12 MAJOR-ROAD)
(london-road13 MAJOR-ROAD)
(london-road14 MAJOR-ROAD)
(london-road15 MAJOR-ROAD)
(london-road16 MAJOR-ROAD)
(london-truck TRUCK)
(la CITY)
(la-po LOC)
(la-ie LOC)
(la-airport AIRPORT)
(la-airport LOC)

```

(la-road1 MAJOR-ROAD)
(la-road2 MAJOR-ROAD)
(la-road3 MINOR-ROAD)
(la-truck TRUCK)
(package2 PACKAGE)
(package1 PACKAGE)
(airplane1 AIRPLANE)
(guard1 GUARD)
(package1 PACKAGE)

(preconds
  (connects edinburgh-road1 edinburgh-airport edinburgh-1)
  (connects edinburgh-road2 edinburgh-1 edinburgh-2)
  (connects edinburgh-road3 edinburgh-2 edinburgh-3)
  (connects edinburgh-road4 edinburgh-3 edinburgh-4)
  (connects edinburgh-road5 edinburgh-4 edinburgh-5)
  (connects edinburgh-road6 edinburgh-5 edinburgh-6)
  (connects edinburgh-road7 edinburgh-6 edinburgh-7)
  (connects edinburgh-road8 edinburgh-7 edinburgh-8)
  (connects edinburgh-road9 edinburgh-8 edinburgh-9)
  (connects edinburgh-road10 edinburgh-9 edinburgh-po)
  (connects london-road1 london-airport london-ie)
  (connects london-road2 london-ie london-1)
  (connects london-road4 london-1 london-2)
  (connects london-road5 london-2 london-3)
  (connects london-road6 london-3 london-4)
  (connects london-road7 london-4 london-5)
  (connects london-road8 london-5 london-6)
  (connects london-road9 london-6 london-7)
  (connects london-road8 london-7 london-8)
  (connects london-road9 london-8 london-9)
  (connects london-road10 london-9 london-10)
  (connects london-road11 london-10 london-11)
  (connects london-road12 london-11 london-12)
  (connects london-road13 london-12 london-13)
  (connects london-road14 london-13 london-14)
  (connects london-road15 london-14 london-po)
  (connects london-road3 london-2 london-8)
  (connects la-road1 la-airport la-ie)
  (connects la-road2 la-ie la-po)
  (connects la-road3 la-airport la-po)
  (not (on guard1 edinburgh-truck))
  (not (on guard1 london-truck))
  (not (on guard1 la-truck))
  (not (on guard1 airplane1))
  (at guard1 london-airport)
  (at la-truck la-airport)
  (at london-truck london-ie)

```

```

(at edinburgh-truck edinburgh-2)
(valuable package2 k1)
(at package2 la-airport)
(valuable package1 ktop)
(at package1 london-2)
(at airplane1 london-airport)
)

(effects
(goal p1 (when (at package1 london-po) ltop))
(goal p2 (when (at package2 la-po) ltop))
)

```

D.2 The Rescue Problem

D.2.1 Domain Operators

Scales

```

(SatScale lbot l1 l2 l3 l4 ltop)
(TruthScale kbot k1 k2 k3 k4 ktop)

```

Helicopter Operators

```

// overloading is possible, but not preferred
// Helicopter is small.
// Only has room for people, not belongings.
(operator LOAD-HELICOPTER
  (params (?h HELICOPTER) (?loc LOCATION) (?p1 PERSON) (?p2 PERSON))
  {when (preconds (at ?h ?loc) (at ?p1 ?loc) (empty ?h == ktop))
    (effects (not (at ?p1 ?loc)) (on ?p1 ?h) (empty ?h k2))
    ltop}
  {when (preconds (at ?h ?loc) (at ?p1 ?loc) (empty ?h == k2))
    (effects (not (at ?p1 ?loc)) (on ?p1 ?h) (empty ?h kbot))
    l4}
  {when (preconds (at ?h ?loc) (at ?p1 ?loc) (at ?p2 ?loc)
    (empty ?h == ltop))
    (effects (not (at ?p1 ?loc)) (not (at ?p2 ?loc)) (on ?p1 ?h)
      (on ?p2 ?h) (empty ?h kbot))
    l4}
  )

(operator UNLOAD-HELICOPTER
  (params (?h HELICOPTER) (?loc LOCATION) (?p1 PERSON) (?p2 PERSON))
  {when (preconds (at ?h ?loc) (on ?p1 ?h) (empty ?h == kbot))
    (effects (at ?p1 ?loc) (not (on ?p1 ?h)) (empty ?h k2))
    ltop}
  )

```

```

{when (preconds (at ?h ?loc) (on ?p1 ?h) (empty ?h == k2))
  (effects (at ?p1 ?loc) (not (on ?p1 ?h)) (empty ?h ktop))
  ltop}
{when (preconds (at ?h ?loc) (on ?p1 ?h) (on ?p2 ?h))
  (effects (at ?p1 ?loc) (not (on ?p1 ?h)) (at ?p2 ?loc)
    (not (on ?p2 ?h)) (empty ?h ktop))
  ltop}
)

// Will risk less suitable landing sites because of emergency.
(operator FLY-HELICOPTER
  (params (?h HELICOPTER) (?o LOCATION) (?d LOCATION))
  {when (preconds (at ?h ?o) (suitable-landing-site ?d >= k4))
    (effects (not (at ?h ?o)) (at ?h ?d))
    ltop}
  {when (preconds (at ?h ?o) (suitable-landing-site ?d range k2 k3))
    (effects (not (at ?h ?o)) (at ?h ?d))
    l3}
  {when (preconds (at ?h ?o) (suitable-landing-site ?d == k2))
    (effects (not (at ?h ?o)) (at ?h ?d))
    l1}
  l1}
)

```

Truck Operators

```

(operator LOAD-TRUCK
  (params (?t TRUCK) (?loc LOCATION) (?i TRANSPORTABLE-ITEM))
  {when (preconds (at ?t ?loc) (at ?i ?loc))
    (effects (not (at ?i ?loc)) (on ?i ?t))
    ltop}
  )

(operator UNLOAD-TRUCK
  (params (?t TRUCK) (?loc LOCATION) (?i TRANSPORTABLE-ITEM))
  {when (preconds (at ?t ?loc) (on ?i ?t))
    (effects (at ?i ?loc) (not (on ?i ?t))
    ltop}
  )

(operator DRIVE-TRUCK
  (params (?t TRUCK) (?loc1 LOCATION) (?loc2 LOCATION) (?r1 ROAD))
  {when (preconds (at ?t ?loc1) (connects ?r1 ?loc1 ?loc2))
    (effects (not (at ?t ?loc1)) (at ?t ?loc2))
    ltop}
  {when (preconds (at ?t ?loc2) (connects ?r1 ?loc1 ?loc2))
    (effects (not (at ?t ?loc2)) (at ?t ?loc1))
    ltop}
  )

```


Boat Operators

```
(operator LOAD-BOAT
  (params (?b BOAT) (?loc LOCATION) (?i TRANSPORTABLE-ITEM))
  {when (preconds (at ?b ?loc) (at ?i ?loc))
    (effects (not (at ?i ?loc)) (on ?i ?b))
    ltop}
)

(operator UNLOAD-BOAT
  (params (?b BOAT) (?loc LOCATION) (?i TRANSPORTABLE-ITEM))
  {when (preconds (at ?b ?loc) (on ?i ?b))
    (effects (at ?i ?loc) (not (on ?i ?b)))
    ltop}
)

// Will take a risk because of emergency situation.
(operator SAIL-BOAT
  (params (?b BOAT) (?loc1 LOCATION) (?loc2 LOCATION)
    (?wr WATER-ROUTE))
  {when (preconds (at ?b ?loc1) (connects ?wr ?loc1 ?loc2)
    (safe ?wr >= k3))
    (effects (not (at ?b ?loc1)) (at ?b ?loc2))
    ltop}
  {when (preconds (at ?b ?loc2) (connects ?wr ?loc1 ?loc2)
    (safe ?wr >= k3))
    (effects (not (at ?b ?loc2)) (at ?b ?loc1))
    ltop}
  {when (preconds (at ?b ?loc1) (connects ?wr ?loc1 ?loc2)
    (safe ?wr range k1 k2))
    (effects (not (at ?b ?loc1)) (at ?b ?loc2))
    l3}
  {when (preconds (at ?b ?loc2) (connects ?wr ?loc1 ?loc2)
    (safe ?wr range k1 k2))
    (effects (not (at ?b ?loc2)) (at ?b ?loc1))
    l3}
)
```

D.2.2 Problem Specification

Plan Objects

```
// Volcano Island:
(lab-truck TRUCK)
(v-truck TRUCK)
(science-lab LOCATION)
(volcano-village LOCATION)
(volcano-beach1 LOCATION)
```

```

(volcano-beach2 LOCATION)
(volcano-safe-point LOCATION)
(scientist PERSON)
(scientist TRANSPORTABLE-ITEM)
(scientific-equipment TRANSPORTABLE-ITEM)
(civilian1 PERSON)
(civilian1 TRANSPORTABLE-ITEM)
(civilian2 PERSON)
(civilian2 TRANSPORTABLE-ITEM)
(personal-posessions TRANSPORTABLE-ITEM)
(vroad1 ROAD)
(vroad2 ROAD)
(vroad3 ROAD)
(vroad4 ROAD)
(vroad5 ROAD)

// Neighbour Island:
(n-truck TRUCK)
(neighbour-village LOCATION)
(neighbour-beach LOCATION)
(nroad ROAD)

// Far Island:
(f-truck TRUCK)
(far-village LOCATION)
(far-beach LOCATION)
(froad ROAD)

// Water connections & boat:
(boat1 BOAT)
(wr1 WATER-ROUTE)
(wr2 WATER-ROUTE)
(wr3 WATER-ROUTE)

// Helicopter:
(helicopter1 HELICOPTER)

```

Initial Conditions

```

(preconds
// Volcano Island
(connects vroad1 science-lab volcano-safe-point)
(connects vroad2 science-lab volcano-village)
(connects vroad3 science-lab volcano-beach2)
(connects vroad4 volcano-village volcano-beach2)
(connects vroad5 volcano-beach1 volcano-beach2)
(suitable-landing-site volcano-safe-point k3)
(suitable-landing-site volcano-village k1)

```

```

// Neighbour Island
(connects nroad neighbour-beach neighbour-village)
(suitable-landing-site neighbour-village)
// Far Island
(connects froad far-beach far-village)
(suitable-landing-site far-village ktop)
// Water Connections
(connects wr1 volcano-beach1 neighbour-beach)
(safe wr1 ktop)
(connects wr2 volcano-beach2 neighbour-beach)
(safe wr2 k2)
(connects wr3 neighbour-beach far-beach)
(safe wr3 ktop)
// Machines
(at boat1 neighbour-beach)
(at lab-truck science-lab)
(at v-truck volcano-beach2)
(at n-truck neighbour-village)
(at f-truck far-village)
(at helicopter1 far-village)
(empty helicopter1 ktop)
// People/Items
(at scientist science-lab)
(at civilian1 volcano-village)
(at civilian2 volcano-village)
(at scientific-equipment science-lab)
(at personal-posessions volcano-village)
)

```

Goals

```

(effects
(goal scientist-destination
(when (at scientist volcano-safe-point) 12)
(when (at scientist neighbour-village) 14)
(when (at scientist far-village) ltop))
(goal civilian1-destination
(when (at civilian1 volcano-safe-point) 12)
(when (at civilian1 neighbour-village) 14)
(when (at civilian1 far-village) ltop))
(goal civilian2-destination
(when (at civilian2 volcano-safe-point) 12)
(when (at civilian2 neighbour-village) 14)
(when (at civilian2 far-village) ltop))
(goal scientific-eq
(when (at scientific-equipment science-lab) 11)
(when (at scientific-equipment volcano-safe-point) 13)
(when (at scientific-equipment neighbour-village) ltop)

```

```
(when (at scientific-equipment far-village) ltop))
(goal possessions
  (when (at personal-posessions volcano-village) l1)
  (when (at personal-posessions volcano-safe-point) l3)
  (when (at personal-posessions neighbour-village) ltop)
  (when (at personal-posessions far-village) ltop))
)
```

Index

- C (constraint set) 13
- D (domains) 13
- I (initial conditions) 205
- K (truth degree scale) 205
- L (satisfaction scale) 53
- O (flexible operators) 205
- X (variable set) 13
- Γ (goal conditions) 205
- Φ (plan objects) 205
- Ψ (flexible planning problem) 205
- ℓ (size of L) 91
- $a/rcDCSP$ 66
- $a/rrDCSP$ 66
- m (size of domains) 91
- n (number of variables) 13, 91
- t (tightness) 92
- con (connectivity of constraint graph) 92
- AC-1 25, 266
- AC-1/2 33
- AC-1/3 33
- AC-1/4 33, 267
- AC-2 25
- AC-3 25, 267
- AC-4 26
- AC-5 27
- AC-6 27
- AC-7 28
- ACBBFC 93
- ACFLCFC 93
- ACFLCFCft 93
- action 160, 162
- activity constraints 60, 161
- activity-based dynamic CSP ($aDCSP$) 60
- activity-based dynamic flexible CSP ($aDFCSP$) 70
- adaptive consistency 31
- additive flexible CSP 47
- AI planning 4, 8, 159, 241
- anytime 44
- applications of CSP 3, 5
- arc consistency 24, 168
- Backjump 18, 263
- Backmark 20, 265
- Backmark hybrids 21
- Backmark-conflict-directed Backjump (BM-CBJ) 21
- Backmarkjump (BMJ) 21
- Backtrack 17, 263
- backtrack-free search 29
- BBFC 93
- BBFMAC 93
- Blocks world 194
- BM-CBJ 34
- BMJ2, BM-CBJ2 22
- branch and bound 46, 72, 240, 268
- Brelaz heuristic 36, 94
- car configuration 61
- chordal graph 42
- competing needs 163, 210
- complete assignment 14
- complexity of FLC 79
- complexity of Flexible Graphplan 221
- complexity of fuzzy arc consistency 81
- complexity of GP- $rrDCSP$ 177
- conflict recording 37
- conflict-directed Backjump 19, 264
- conformal graph 43
- conformant planning 204
- connectivity 39
- ConsDeg 95

- ConsDeg- C_r 96
- ConsDeg-PrevSoln-*con* 96
- ConsDeg-*con* 96
- consistency 2
- consistency counts 50
- consistency degree 54, 71
- constraint check 17
- constraint check ordering heuristic 96, 136, 181
- constraint checks 94
- constraint graph 14
- constraint logic programming 5
- constraint network 14
- constraint recording 57
- constraint satisfaction problem (CSP) 2, 13
- constraints 1
- contingent planning 203
- course scheduling 86
- cycle cutset 41

- dead end 17, 37
- deep learning 37
- deletion threshold 84, 130
- dependency-directed backtracking 37
- directional arc consistency 31
- directional path consistency 31
- directional pre-processing 31
- domain element selection heuristic 95, 148
- domain filtering 23
- domains 13
- drowning effect 207
- dual graph 14
- Dynamic Backtracking 22
- dynamic CSP (DCSP) 7, 55
- dynamic flexible CSP (DFCSP) 7, 65, 66, 239
- dynamic variable ordering 35

- effects 162
- eliminating explanation 22, 169, 216
- equivalent problems 23
- extending tree search 45

- fail-first 35
- FC-BJ 34
- FC-CBJ 34
- first-order learning 38

- FLCFC 93
- FLCFCft 93
- FLCFMAC 93
- FLCFMACft 93
- flexible compositional modelling 247
- flexible CSP 6
- flexible goal 205, 207
- flexible graph expansion 209, 210
- Flexible Graphplan (FGP) 9, 219, 242, 244, 274
- Flexible Local Changes (FLC) 74, 240, 271
- flexible operator 205, 206
- flexible plan extraction 213
- flexible planning 8, 204
- flexible planning problem 205
- flexible problems 43
- flexible proposition 205
- flexible qualitative simulation 246
- focus heuristics 179
- FRevise() 80
- front 40
- future variables 16
- fuzzy *rr*DFCSP 70
- fuzzy arc consistency 80, 100, 106
- fuzzy CSP 53, 71
- fuzzy relation 71

- GARI 6, 51
- Generate and Test 16, 17
- goals 159, 166
- GP-*rr*DCSP 174, 243, 272
- graph decomposition 46
- graph-based methods 39
- graph-based shallow learning 37
- Graphplan 4, 8, 161
- Grid Problem 199
- Gripper Problem 197

- hard constraint 6, 14
- heuristics 3, 35, 94
- hidden variable encoding 15
- hierarchical decomposition 8
- hierarchical plan extraction 166, 214
- hill-climbing 72
- hybrids 3, 32, 82, 240

- ill-defined CSP 44
- imperative constraints 14

- inconsistent effects 163, 210
- inconsistent support 163
- inflexible constraints 14
- instantiation order 16, 35
- interference 163, 210
- invasion procedure 40
- inverse consistency 30
- irrelevance test 170
- ISIS 4, 49

- job-shop scheduling 4
- jump-back learning 38

- k-consistency 29
- Kappa heuristic 36

- learning 37
- level of a planning graph 163
- limitations of classical CSP 6
- limited graph expansion 212, 229
- Local Changes (LC) 57, 72, 73, 172
- local repair 57, 72
- logistics problem 186, 205
- LogisticsA 189
- LogisticsB 191
- LogisticsC 191
- LogisticsD 193
- lookahead value ordering 36

- Maintaining Arc Consistency (MAC) 34
- matrix of instances of DFCSP 66
- MAX CSP 47
- max-fail Backjump 19
- maximum cardinality heuristic 35
- maximum degree heuristic 35
- memoisation 165, 168, 171, 215
- memoset 165, 168, 171, 173, 215
- memoset propagation 173
- meta-csp 248
- micro-problem 178
- minimal conflict sets 37
- minimum-width heuristic 35
- Movie Problem 198
- mushy region 39
- mutual exclusion 163, 210

- necessity-based constraints 51
- neighbourhood inverse consistency 30

- node consistency 23
- nogoods 22, 37
- non-separable components 42
- Noop 163
- NP-completeness of CSP 2
- NR algorithm 38

- operator 162
- oracles 56
- ordered constraint graph 29
- over-constrained 6

- partial arc consistency 33
- partial assignment 14
- partial constraint satisfaction 50
- partial lookahead 33
- past variables 16
- path consistency 28
- PC-1 28
- PC-2 28
- PC-3 28
- PC-4 28
- phase transition 38, 96, 241
- plan 159
- plan extraction 165
- plan objects 205
- planning graph 163, 166
- pre-processing 2, 23, 82
- preconditions 162
- preference 44, 49, 53, 71
- primal graph 14
- prioritised-preference 54, 71
- priority 6, 44, 54, 71
- priority maximisation 51
- propagated irrelevance testing 177
- propagated repair 171
- propositional satisfiability (SAT) 4

- quality of a plan 207

- recurrent dynamic CSP (*rc*DCSP) 58
- recurrent dynamic flexible CSP (*rc*DFCSP) 68
- reduction explanation 216
- redundant arcs 43
- relaxation 56
- Rescue Problem 230, 242
- restriction 56
- restriction/relaxation dynamic flexible CSP (*rr*DFCSP) 67

- Revise() 24, 266
- Rho heuristic 36
- RocketA 186
- RocketB 187
- root of search tree 16
- satisfaction 1
- satisfaction degree 206, 207
- satisfaction propagation 212, 229
- satisfaction scale 71
- scheduling 4
- search branch 16
- search depth 16
- search for a solution 2
- search nodes 94
- second-order learning 38
- shallow learning 37
- singleton consistency 30
- smallest domain heuristic 35
- SmallestD 95
- SmallestD- X_2 95
- SmallestD- X_3 95
- SmallestD- X_{23} 95
- soft constraints 6
- solution 14
- solution re-use 7
- stability 94
- static problem 14
- static variable ordering 35
- STRIPS 162
- stubbornness 38
- sub-problem (solution of) 73
- sufficient bound 46
- supermodels 60
- systems simulation 5, 246
- thrashing 18
- tightness 39
- tree clustering 42
- tree search 16
- truth degree 205
- truth-maintenance systems 37
- types of dynamic change 7
- value ordering heuristics 36
- value selection heuristics 180
- value-based learning 38
- variable ordering heuristic 94, 140, 179
- variables 13
- vision 3
- Waltz algorithm 3, 25
- weakness of Backjump 19
- weaknesses (of classical CSP) 1
- width 29
- wipeout 17
- ZEBRA problem 34