CONDITIONAL AND COMPOSITE (TEMPORAL) CSPs

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN COMPUTER SCIENCE

UNIVERSITY OF REGINA

By

Amrudee Sukpan

Regina, Saskatchewan

July, 2008

© Copyright 2008: Amrudee Sukpan

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 978-0-494-42515-2

Our file Notre référence

ISBN: 978-0-494-42515-2

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

UNIVERSITY OF REGINA
FACULTY OF GRADUATE STUDIES AND RESEARCH
SUPERVISORY AND EXAMINING COMMITTEE

Amrudee Sukpan, candidate for the degree of Doctor of Philosophy in Computer Science, has presented a thesis titled, ***Conditional and Composite (Temporal) CSPs***, in an oral examination held on July 2, 2008. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner:	Dr. Maroua Bouzid, University of Caen
Supervisor:	Dr. Malek Mouhoub, Department of Computer Science
Committee Member:	Dr. Howard J. Hamilton, Department of Computer Science
Committee Member:	Dr. Samira Sadaoui-Mouhoub, Department of Computer Science
Committee Member:	Dr. Joseph Piwowar, Department of Geography
Chair of Defense:	Dr. Dongyan Blachford, Faculty of Graduate Studies and Research

ABSTRACT

We propose the Conditional and Composite Constraint Satisfaction Problem (CCCSP) model, a unique framework to efficiently manage dynamic CSPs where dynamic sets of variables and constraints are known under predefined conditions. Two resolution techniques for solving CCCSPs: (1) complete methods based on constraint propagations and (2) incomplete methods based on Stochastic Local Search (SLS), have been developed. The CCCSP framework is augmented to encode numeric and symbolic temporal constraints. The new temporal model is called Conditional and Composite Temporal Constraint Satisfaction Problem (CCTCSP). In addition, the CCCSP resolution methods have been adapted for solving CCTCSPs. In the case of searching for the preferred solution according to a given objective function, preferences defined at different levels of CCCSPs and CCTCSPs indicate degrees of satisfaction and are treated as soft constraints. In order to handle these preferences, the CCCSP and the CCTCSP frameworks are extended with the c-semiring, and the complete solving methods are modified following the Branch and Bound technique. CCCSPs and CCTCSPs with preferences can be applied to applications such as configuration problems, planning and scheduling problems, software designs and expert systems. The proposed algorithms are compared using an extensive experimental study over randomly generated problems. The experimental results demonstrate that in the case of hard problems (i.e., middle constrained and highly dynamic structured instances), our complete solving methods with the advanced constraint propagation techniques outperform the general ones while obtaining the same results. The results also

show that, the incomplete methods using SLS techniques are faster than the complete ones in the case of hard problems. These SLS techniques are relevant in case we want to trade the quality of solutions for the time efficiency.

ACKNOWLEDGEMENTS

I acknowledge Dr. Malek Mouhoub for his excellent supervision. I have gained substantive knowledge in Constraint Processing and research experience from him. He also gave me the opportunity to develop my communication skills through participation in several international conferences. I also thank him for his financial support during my graduate studies.

I thank the external member, Dr. Maroua Bouzid from the University of Caen, France and the members of the supervisory committee, Dr. Howard Hamilton, Dr. Samira Sadaoui and Dr. Joe Piwovar for examining the thesis. Their feedback and comments are very appreciated.

I benefited from financial support from the Department of Computer Science and the Faculty of Graduate Studies and Research.

I am grateful to my colleagues, Roger Barbour, Colin Witow, Siritorn Srisodsai, Xinkai Feng and Chonghai Wang for their support during my studies.

DEDICATION

To my family in Thailand and my husband, Khang Tuan Nguyen, who always stood by me throughout my studies.

I give a special thanks to my Peace Mennonite Church Regina friends. Their encouragement and care have made me feel at home in Regina.

TABLE OF CONTENTS

ABSTRACT	i
ACKNOWLEDGEMENTS	iii
DEDICATION	iv
TABLE OF CONTENTS	v
LIST OF TABLES	ix
LIST OF FIGURES	x
LIST OF ACRONYMS	xiv
1. INTRODUCTION	1
1.1 Problems, Motivations and Objectives	1
1.2 Proposed Solutions and Contributions	11
1.3 Thesis Outline	13
2. CONSTRAINT PROCESSING	16
2.1 Constraint Satisfaction Problems (CSPs)	16
2.1.1 Binary Constraint Network	18
2.2 Complete Methods	20
2.2.1 Generate and Test (GT) and Backtrack Search (BT) Techniques . .	20

2.2.2	Backjumping and Dynamic Backtracking Techniques	21
2.2.3	Local Consistency Techniques	24
2.2.4	Constraint Propagation	29
2.2.5	Variable and Value Ordering	29
2.3	Incomplete Methods	33
2.4	Dynamic Constraint Satisfaction Problems	38
2.4.1	Conditional CSPs (CCSPs)	40
2.5	Soft Constraints	44
2.5.1	Valued Constraint Satisfaction	45
2.5.2	Semiring-based Constraint Satisfaction	46
2.5.3	CSPs with Preferences	47
2.6	Conclusions	49
3.	TEMPORAL CONSTRAINT PROCESSING	50
3.1	Temporal Constraints	51
3.1.1	Numeric Temporal Models	51
3.1.2	Symbolic Temporal Models	54
3.1.3	Hybrid Temporal Models	58
3.2	Dynamic Temporal CSPs	68
3.3	Temporal Constraints with Preferences	71
3.4	Conclusions	73
4.	MANAGING CONDITIONAL CONSTRAINTS AND COMPOSITE VARIABLES	74

4.1	CCCSP Framework	76
4.1.1	Activity Constraints	77
4.1.2	Composite Variables	78
4.1.3	CCCSP Example	79
4.2	Solving CCCSPs	81
4.2.1	Complete Methods for CCCSPs	83
4.2.2	Incomplete Methods for CCCSPs	98
4.3	Conclusions	109
5.	CCCSP FRAMEWORK FOR TEMPORAL PROBLEMS	110
5.1	Conditional and Composite Temporal Constraint Satisfaction Problems (CCTCSPs)	111
5.2	Slicing SOPOs and Composite Variables	115
5.3	Numeric to Symbolic Conversion	129
5.4	Numeric Temporal Constraints and SOPOs	130
5.5	Conclusions	132
6.	CCCSPs AND CCTCSPs WITH PREFERENCES	134
6.1	CCCSPs with Preferences	134
6.2	CCCSP Example	138
6.3	CCTCSPs with Preferences	141
6.3.1	Representing Temporal preference modifiers	146
6.4	CCTCSPP Example	146
6.5	Solving CCCSPPs and CCTCSPPs	150

6.6	Conclusions	154
7.	EXPERIMENTATION	155
7.1	CCCSP Instances	157
7.2	CCTCSP Instances	161
7.3	Experimental Environment	162
7.4	Test Results for Complete Methods	163
7.4.1	Varying the Density of Activity Constraints, <i>act</i>	164
7.4.2	Varying the Number of Initial Variables, <i>iv</i>	168
7.4.3	Varying the Number of Composite Variables, <i>cn</i>	170
7.4.4	Varying <i>r</i>	170
7.5	The Results for Incomplete Methods	172
7.6	Result of Various Branch and Bound Methods for CCCSPs and CCTCSPs with Preferences	186
7.7	Conclusions	187
8.	CONCLUSION AND FUTURE WORK	190
8.1	Summary	190
8.2	Future Work	193
8.2.1	Composite Variables Including Other Composites	193
8.2.2	Solving CCCSPs and CCTCSPs in Unknown Dynamic Environments	194
8.2.3	Incomplete Methods for CCCSPs and CCTCSPs with Preferences .	195
8.2.4	Real World Applications	196

LIST OF TABLES

2.1	Instances of SCSPs	47
3.1	Allen Relations	55
3.2	IA Composition	56
3.3	The Basic Relations Between a Point, P , and an Interval, $I = [I^-, I^+]$ [68]	60
3.4	A Full Transitivity Table [68]	61
3.5	Composition of PP and PI Relations [68]	61
3.6	Composition of PI and IP Relations [68]	61
3.7	Composition of IP and PI Relations [68]	61
3.8	Composition of PI and II Relations [68]	61
3.9	Composition of PA [113]	62
3.10	QUAN Translation [68]	63
3.11	Numeric to Symbolic Conversion	66
7.1	Constraint Density of Random Generated CCCSP Instances	161
7.2	The Summary of Complete Methods	172

LIST OF FIGURES

2.1	A Timetable	19
2.2	An Equivalent Constraint Network of Timetabling Problem in Figure 2.1 .	19
2.3	BT Algorithm	22
2.4	BJ Algorithm	23
2.5	DB Algorithm	24
2.6	NC Algorithm	25
2.7	Arc Revision Algorithm	27
2.8	AC-3 Algorithm	27
2.9	AC-3 for FC Algorithm	30
2.10	AC-3 for MAC Algorithm	30
2.11	Hill-Climbing Algorithm	35
2.12	MCRW Algorithm	36
2.13	SDRW Algorithm	37
2.14	Tabu Search Algorithm	37
3.1	A Numeric Temporal Constraint Network [31]	53
3.2	AI Constraint Network [29]	57
3.3	Path Consistency Algorithm [64]	57
3.4	A PA Constraint Network [29]	58
3.5	An QA Constraint Network [68]	64
3.6	SOPO of an Event	65

3.7	A TemPro Network	67
3.8	A Planning Example [107]	70
3.9	The Skiing Plan as a Conditional STP [107]	70
3.10	Handling Fuzziness of Numeric Constraints Using Non-Normalized Trape- zoidal Distribution [7]	73
4.1	A Dress Up Game Example	82
4.2	Compatible Constraints for the Example in Figure 4.1	83
4.3	AC3-CCCSP Algorithm	86
4.4	A CCCSP Example Before Performing AC3-CCCSP	87
4.5	Perform AC3-CCCSP to (a) C_{X_1,Y_1} and (b) C_{Y_1,X_1}	87
4.6	Perform AC3-CCCSP to (c) C_{Y_2,Y_1} and (b) C_{Y_1,Y_2}	88
4.7	BT-CCCSP Algorithm	89
4.8	Steps of BTCCCSP for Dress Up Game Example Figure 4.1	90
4.9	FC-PROPAGATE-CONSTRAINT Algorithm	92
4.10	MAC-PROPAGATE-CONSTRAINT Algorithm	92
4.11	FUTURE-ACTIVE-VARIABLES Algorithm	93
4.12	FCCCCSP for the Dress Up Game Example Figure 4.1	95
4.13	MAC+CCCSP for the Dress Up Game Example Figure 4.1	96
4.14	MCRW-CCCSP _{Static} Algorithm	101
4.15	MAIN-MCRW-CCCSP _{Static} Algorithm	102
4.16	MCRW-CCCSP _{Static} for the Dress Up Game Example (Figure 4.1)	103
4.17	MCRW-CCCSP _{Dynamic} Algorithm	104
4.18	MCRW-CCCSP _{Dynamic} for the Dress Up Game Example (Figure 4.1)	106

4.19	MCRW-CCCSPP _{Incremental} Algorithm	107
4.20	MCRW-CCCSPP _{Incremental} for the Dress Up Game Example (Figure 4.1) . .	108
5.1	CCTCSP Network of Example 5.1.1	115
5.2	Arc Consistency on the Initial CCTCSP of Example 5.1.1	116
5.3	Constraint Propagation on the CCTCSP of Example 5.1.1	117
5.4	Slicing SOPO by Start Before X	119
5.5	Slicing SOPO by Start After X	120
5.6	Slicing SOPO by Finish Before X	121
5.7	Slicing SOPO by Finish After X	122
5.8	CCTCSP of Cooking Example	124
5.9	Representing Having-Dinner, Cooking-Soup, Cooking-Maindish by Com- posite Variables	127
5.10	Steps in Solving the CCTCSP in Example 5.2.1 using BT	128
5.11	Steps in Solving the CCTCSP in Example 5.2.1 using FC ⁺	128
5.12	CCTCSP Network for Example 5.4.1	132
6.1	Qualitative Temporal Preference Functions	147
6.2	The Soft Numeric Temporal Preference Function of Event <i>JPL</i>	148
6.3	Example 5.1 with Four Types of Preferences	149
6.4	BnBCCCSPP Algorithm	151
6.5	Steps in Solving the CCCSPP example in Section 6.2 using BnB Algorithm and Heuristic Function	153
7.1	Easy Versus Hard Problems	156

7.2	The Comparison of FC, FC ⁺ , MAC and MAC ⁺ when Varying the Density of Activity Constraints (Test 7.4.1)	166
7.3	The Inverse Graphs of Figure 7.2 where the X-axis is the Density of Activity Constraints	167
7.4	The Comparison of FC, FC ⁺ , MAC and MAC ⁺ where $0.2 \leq iv \leq 0.9$ and $0.2 \leq t \leq 0.7$ (Test 7.4.2)	169
7.5	The Comparison of FC, FC ⁺ , MAC and MAC ⁺ where $cn = 5, 10, 15$ and $0.4 < t < 0.8$ (Test 7.4.3)	171
7.6	The Comparison of FC, FC ⁺ , MAC and MAC ⁺ where $0.2 \leq r \leq 1.0$ and $0.2 \leq t \leq 0.7$ (Test 7.4.4)	173
7.7	Results of Incomplete Methods (Test 7.4.1: $t=0.25$)	175
7.8	Results of Incomplete Methods (Test 7.4.1: $t=0.35$)	176
7.9	Results of Incomplete Methods (Test 7.4.1: $t=0.45$)	177
7.10	Results of Incomplete Methods (Test 7.4.1: $t=0.55$)	178
7.11	Results of Incomplete Methods (Test 7.4.1: $t=0.65$)	179
7.12	Results of Incomplete Methods (Test 7.4.1: $t=0.75$)	180
7.13	Results of Incomplete Methods (Test 7.4.1: $t=0.85$)	181
7.14	Results of Incomplete Methods (Test 7.4.2: $t=0.5$)	182
7.15	Results of Incomplete Methods (Test 7.4.2: $t=0.6$)	183
7.16	Results of Incomplete Methods (Test 7.4.2: $t=0.7$)	184
7.17	Results of Incomplete Methods (Test 7.4.2: $t=0.8$)	185
7.18	Comparison of BnBFC, BnBFC ⁺ , BnBMAC and BnBMAC ⁺	188

LIST OF ACRONYMS

AC	Arc Consistency
AI	Artificial Intelligence
BnB	Branch and Bound
BT	Backtrack Search
CSP	Constraint Satisfaction Problem
CCSP	Conditional Constraint Satisfaction Problem
CCCSPP	Conditional and Composite Constraint Satisfaction Problem
CCCSPP	Conditional and Composite Constraint Satisfaction Problem with Preferences
CCTCSP	Composite and Conditional Temporal Constraint Satisfaction Problem
CCTCSPP	Composite and Conditional Temporal Constraint Satisfaction Problem with Preferences
CLP	Constraint Logic Programming
FC	Forward Checking
FC+	Advanced Forward Checking
FLA	Full Look Ahead
GT	Generate and Test
HC	Hill Climbing
IA	Interval Algebra
MAC	Maintaining Arc Consistency
MAC+	Advanced Maintaining Arc Consistency
MC	Min Conflict
MCRW	Min Conflict Random Walk
NC	Node Consistency
NP	Nondeterministic Polynomial-time
PA	Point Algebra
OOP	Object Oriented Programming
PC	Path Consistency

QA	Qualitative Algebra
RW	Random Walk
SLS	Stochastic Local Search
SDRW	Steepest Descent Random Walk
SOPO	Set Of Possible Occurrences
STP	Simple Temporal Problem
TCSP	Temporal Constraint Satisfaction Problem
TS	Tabu Search

1. INTRODUCTION

“Research is the process of going up alleys to see if they are blind”

Marston Bates, 1967

1.1 Problems, Motivations and Objectives

Over the past three decades, the Constraint Satisfaction Problem (CSP) approach has been applied to a variety of Artificial Intelligence (AI) domains: Machine Vision [23], Natural Language Understanding [3, 35], Distributed Systems [121], Planning and Scheduling [110], Expert Systems [34] and Temporal Reasoning [4, 31, 113]. CSP captures the core of most computational problems in terms of *constraints* over *inquiring objects*. Informally, a CSP consists of a set of constraints, a set of variables and their associated domains. When a problem is formulated as a CSP, CSP resolution mechanisms can be applied to solve the problem.

The literature on CSPs is well established and has three main research tracks:

1. improving the efficiency of solving algorithms [61, 100, 60],
2. implementing constraint languages [37] and
3. enhancing expressiveness [30, 69, 93, 15, 16].

In the early years of research on constraint satisfaction [61, 100, 60], advancements were made on the first track. CSP is a NP complete problem [27]. In computational

complexity theory, the complexity class NP-complete, also known as NP-C or NPC, is a subset of NP (“non-deterministic polynomial time”). A NP-complete problem is a NP-hard problem which is the most difficult problem in NP, and any NP-problem can be translated into this NP-complete problem in a polynomial time [85].

A typical complete method to solve a CSP is the Backtrack (BT) search. To reduce redundancy checking and thrashing, Local Consistency, Constraint Propagation, Heuristics and Back Jumping are used to enhance BT search [108, 29]. A tractable subclass of CSPs using tree structures has been identified, any problem in this class can be solved in polynomial time [66, 25]. On the other hand, alternative polynomial schemes based on stochastic local search (Hill Claiming, Min Conflict, Tabu Search and Steepest-Descent-Random-Walk) are adopted as incomplete methods [28, 97, 98].

On the second track, a fashionable programming paradigm for solving CSPs and constraint problems in general is Constraint Logic Programming (CLP). Some CLPs extend the Logic Programming paradigm to include the CSP framework, to describe a constraint problem’s declaration and to provide both basic and advanced CSP resolution techniques [37]. These CLP systems include ECLⁱPS^e [6], Oz [96], Ilog SOLVER [54], CLP(FD) [33], CHR [42], SICStus [62] and B-Prolog [123]. The CSP paradigm has shown promising potential in many problem domains [23, 3, 35, 121, 110]. To give more choices to programmers, constraint solving methods have been implemented in Object Oriented Programming (OOP) C++ (such as Disolver [50], QDOOCS [117] and ILog SOLVER [54]) and Java (such as CHOCO [22], Cream [104] and JCL [55]).

On the third track, the standard CSP has been extended to be more expressive in new contents (such as for incomplete information, or when an objective function is taken into

account), and to be functional in dynamic environments [36, 92, 15, 30, 69, 10, 112, 102]. A weaker version of global satisfactions is used to relax over constrained problems and MAX-CSPs [24]. Fuzzy CSPs and Uncertainty CSPs are able to capture incomplete, vague and unknown information [92, 36]. Cost functions, weighted functions and preference levels in optimization problems are handled by *soft constraints* [16, 40]. Soft constraints differ from hard constraints that is hard constraints are to be necessarily satisfied, while soft constraints only express a preference of some solutions (those having a high or low cost) over other ones (those having lower or higher cost). Two generic soft constraint frameworks are Semiring-Based CSPs [15] and Valued CSPs [95]. The soft constraint concept brings more life to the computational world. It especially adds capacity for filtering the most preferable solution.

The CSP approach has been applied to dynamic real world problems where variables, constraints and domains may change over time. There are two main dynamic CSP paradigms. The first one is the dynamic searching algorithm for reusing and repairing a current solution when variables, domains and constraints have changed [30, 10, 112, 75]. The source of these changes is the unknown external environment. An example of an application encountering unforeseen changes is flight scheduling. Normally when scheduling a flight, an airline has a fixed and complete flight schedule for the whole season. However at times, weather conditions do not permit flights to take place. As a result, the airline needs to reschedule all future flights due to cancelled flights.

In contrast, the second dynamic CSP paradigm (mainly addressed in this thesis) relates to internal elements where dynamic sets of variables and constraints are known under predefined conditions. These dynamic variables are not initially considered in the

problems to solve. But later on, some of them will be added into the problems if their conditions are satisfied. This dynamic framework is called Conditional CSPs (CCSPs). The CSP principle is augmented into a CCSP to capture a new type of hard constraint, called an *activity constraint* [69]. Activity constraints are used to limit the conditional variables with their associated domains and compatible constraints to be part of a final solution.

Originally, activity constraints were introduced for configuration problems. Due to the nature of configuration problems, a set of components is predefined along with a design specification which will control the quality and compatibility of all parts of a product. An example of a dynamic instance of a configuration problem is designing laptops' components. If we compare the series T and X notebooks from IBM/Lenovo¹, we see that the active protection system is a common standard feature. However, the UltraNav multipointing system feature is not available in the X series but is a standard feature in the T series, and the ATI integrated graphics feature is an option in the T series but not available in the X series.

Informally, a CCSP contains a set of CSPs that share the same set of main variables, but each CSP also includes a different set of conditional variables. Conditional variables are added into a final solution if their conditions are true. Activity constraints are used here to control the extra variables in each CSP.

Complete methods for solving Conditional Constraint Satisfaction Problems (CCSPs) are classified in three different approaches [69, 44, 102, 94, 46, 45]:

¹International Business Machines Corporation (IBM) develops and manufactures information technologies, including computer systems, software, networking systems, storage devices, and microelectronics worldwide. Company's website <http://www.lenovo.com/ca>

1. Reformulating an activity constraint into compatibility constraints by adding a *null* value to the domain of conditional variables. Then the CCSP is reformulated into a CSP. General CSP techniques are directly applied to solve the reformulated CSP [94, 46, 45].
2. Generating a set of all possible standard CSPs from a CCSP by viewing the CCSP as a Meta-CSP. General CSP techniques are directly applied to solve each CSP [69].
3. Solving a CCSP directly. Problem spaces are simultaneously generated during the solving process [69, 94, 45, 46].

A CCSP is an NP-complete problem [102]. In practice, the performance in time and space of the above three techniques is in a decreasing order from (2)>(3)>(1) [46]. The weakness of reformulating CCSPs (approach 1) is the increase in domain size, constraints and variables of the reformulated problems. The generation of all possible CSPs (approach 2) from a CCSP consumes more memory space for all generated CSPs, which is wasteful since some of the generated CSPs may not be explored during the search. A small change of an activity constraint causes reformulating a whole new equivalent CSP or regenerating all possible sets of standard CSPs. The direct solving of CCSPs approach (approach 3) is a promising approach. There is neither an increase in the size of the problem nor a requirement for extra space. Integrating constraint techniques into the third approach can prune inconsistent problem configurations [94, 45, 46].

To improve the performance of the direct solving of CCSP approach, the Forward Checking (FC) and Maintaining Arc Consistency (MAC) constraint propagation techniques are applied during the search and also to the newly active variables resulting from

any satisfied activity constraint [94]. In addition, activity constraints are propagated during the search. However, using such constraint propagation still suffers from thrashing (i.e., repeated failure due to the same reason) and redundant work at some points when a new variable is added to a solution. If the newly added variable is inconsistent with the previous assignments and is always required in the current assignment, then backtracking is performed until it returns to the source of the variable in conflict.

To the best of our knowledge, no work in the existing literature has employed incomplete methods to solve CCSPs. One of the reasons could be because the CCSP approach was first motivated by configuration problems which only require complete solutions and in which time is not a critical resource. A strategy of incomplete methods is based on Stochastic Local Search (SLS). Although SLS neither explores all the search space nor guarantees finding a consistent solution, it is efficient in terms of time complexity. Incomplete methods are well suited to some real time applications where a trade off between solution quality and speed is admissible. In addition, incomplete solutions are useful and acceptable in some applications (e.g., game software and self training/learning software). Thus, one of our interests is to investigate the use of SLS for solving CCSPs.

The CCSP paradigm has motivated us to seek more common properties of activity constraints and also to seek its use in other application domains beyond configuration problems. Conditions, options and hierarchy structures are ordinary characteristics of many human endeavors. Conditional statements are if-then structures embedded in the logical thinking of human beings. An optional object is a group of items where only one of them can be selected at a time. A composite object is a hierarchical group of things. The lowest level in the hierarchy indicates atomic subjects that cannot be further decomposed.

A higher level is a generic group consisting of its lower level components. The co-existence of conditional statements, optional objects and composite objects is very common. For example, if you are renting a car and traveling with a child, you will probably also need a child seat (which is optional in rental cars). Otherwise, there is no need to deal with this option. In the renting process, a car type has to be selected first, then the size, and then the brand because cars are classified in categories. Thus, only cars in a chosen type, size and brand will be considered at a time while ignoring other cars which are in other groups.

Although previous researchers have considered the three factors independently, we perceived a CCSP's relationship with optional objects and composite objects. In fact, composite and optional objects are special types of conditional variables in the CCSP framework. Consequently, a redefinition of the CCSP paradigm is appropriate so that composite and optional objects can be applied to intensified notions of activity constraints.

When searching for the preferred solution according to a given objective function, managing preferences is the key. Here preferences are considered as soft constraints and indicate degrees of satisfaction. Preferences play a part in a favorite choice. For example, under a given car rental agreement, customers may favor full-size to mid-size, compact or mini sizes. Japanese cars may be preferred to American or Korean cars.

The common use of conditional, composite and preference features is present in temporal problems. Time is essential in our daily life. In scheduling and planning, one must recognize the ordering of tasks within the available time schedule under conditional options and users' preferences. For example, in scheduling a class, a rule could state:

If a lab size is bigger than 30 students, then the lab needs an additional lab assistant. Otherwise only a single lab instructor is assigned to the lab. It is preferable that the lab will be in the afternoon. Some instructors prefer morning classes rather than afternoon classes.

In exploring archeology and the history of mankind, a time line is used to list important events for successive years within a particular historical period. In the health care domain, for instance, the progression of a disease assists doctors to diagnose diseases and to prescribe remedies for patients according to the symptoms in each developing phase. Patients receive different kinds of treatments according to their health conditions. Another domain where conditions exist is in simulations. For example, when simulating the launching of a rocket, the ordering and the relation of missions' operations are specified according to a given time period to verify the correctness and the tolerance of the system. Finally, in understanding natural language, time is an element in every human language (e.g., in an English sentence a specific time distinction is expressed by tenses.)

Representing and reasoning about time and temporal information in a notation of CSPs (called Temporal CSPs or TCSPs) has been studied for over two decades [4, 113, 31, 109, 106, 74] and has proven its soundness for Artificial Intelligence (AI) applications [84, 18, 83, 14]. A temporal CSP is a CSP where variables (called *events*) are defined on time points in Point Algebra (PA) [109] or time intervals in Interval Algebra (IA) [4]. Temporal constraints are either numeric or symbolic relations. Numeric and symbolic temporal information are managed in separate frameworks because formulating temporal information into symbolic relations (e.g., IA [4] and PA [113]) is different from formulating

it in numeric forms (e.g., TCSP, STP [31], DTP [106]). In fact, both symbolic and numeric forms complete the contents of temporal information. Not all symbolic temporal information can be translated into numeric temporal information and vice versa. In either direction of conversion information is lost. A hybrid temporal model can express both symbolic and numeric temporal information in a unified framework [57, 58, 68, 76, 73].

Meiri's Qualitative Algebra (QA) [68], is an early well known hybrid framework based on both time points and time intervals. It is a combination of Point Algebra (PA) of Vilain and Kautz [113], Interval Algebra (IA) of Allen [4] and a metric network [31]. QA maintains PA relations and IA relations for a pair of time points and a pair of time intervals respectively. Additional relations, Point-Interval (PI) relations and Interval-Point (IP) relations are introduced. A PI relation is defined between a point and an interval whereas an IP relation is defined between an interval and a point. QA has complicated structures, especially for handling composite operations over PI and IP relations.

In Kautz and Ladkin [57], instead of augmenting a single temporal system, symbolic and numeric forms are handled in two different temporal constraint networks. The L logical language is defined for expressing both numeric and symbolic information. Language L_M represents a numeric temporal network based on Dechter et al. [31]. Language L_A represents a symbolic temporal network based on Interval Algebra (IA) of Allen [4]. L_M and L_A can be combined using a translation algorithm.

TemPro is an alternative temporal framework to express and to manage numeric and symbolic temporal information within a unique CSP-based model [73, 74]. TemPro is based on IA and a discrete model of time. It is expressive and simple to employ. Temporal windows, also called Set Of Possible Occurrences (SOPOs), are used to manage numeric

constraints over temporal objects. As in IA, symbolic temporal relations are formed by sets of disjunctive Allen relations. The structure of SOPO supports the numeric $\rightarrow$ symbolic conversion. Therefore, some inconsistent Allen relations are eliminated prior to performing path consistency. Constraint propagation and/or stochastic local search are applied to search for a solution.

There are some extended temporal CSP frameworks for handling specific dynamic temporal problems [56, 38, 107, 77, 78]. Handling preferences in temporal CSPs are proposed by a number of researchers [59, 19, 26, 5, 40, 99, 8, 7]. Unfortunately, none of the hybrid temporal frameworks are able to handle conditional events, optional events, composite events and preferences in a unique hybrid temporal framework.

In summary, we have the following appealing perspectives to redefine the CCSP framework in:

1. developing more efficient complete searching methods,
2. applying SLS as incomplete resolution techniques,
3. defining a formal form of activity constraint,
4. managing conditional variables,
5. conducting optional and composite objects as a special case of conditional variables,
6. handling preferences and searching for a preferred solution and
7. employing CCSPs in temporal problems including numeric and symbolic temporal constraints.

1.2 Proposed Solutions and Contributions

First, we propose a unified general framework that we call, Conditional and Composite CSP or CCCSP. CCCSP is an extension of CCSPs [69] with a new type of variable called a *composite* variable. The classic variable, we called a *simple* variable (*e.g.*, Y). Domains of composite variables are sets of simple variables. Here, composite variables take three crucial roles in Conditional and Composite Constraint Satisfaction Problems (CCCSPs).

1. To handle disjunctive activity constraints (i.e., one activity constraint has more than one choice of variables to activate). If $X_i = a \xrightarrow{\text{active}} Y_j \text{ or } Y_k$, the disjunctive activity constraint is handled by a composite variable $C = \{Y_j, Y_k\}$. As a result, we obtain $X_i = a \xrightarrow{\text{active}} C$.
2. To represent composite objects and top view design problems.
3. To express optional objects that can either be in or not in the problem to solve regardless of the conditions.

A solution to a CCCSP is an assignment of all simple variables and composite variables that satisfy the compatible and activity constraints. To solve CCCSPs, we have made the following contributions [79, 81]:

1. AC-3 algorithm is augmented to propagate constraints over composite variables. We have defined a weaker arc consistency technique [81].
2. Complete methods integrating constraint propagation based on FC and MAC (more pruning than CondFC and CondMAC algorithms suggested in Sabin et al. [94]) are proposed [81].

3. Values and variables ordering based on the first fail principle are applied [81].
4. Incomplete methods are first adopted in CCSPs and CCCSPs [79]. We focus on the change of configuration problem in the repairing stage. The dynamic structure leads to three different strategies:
 - (a) *static*, which does not allow changing problem configurations in the repairing stage.
 - (b) *dynamic*, which permits changing problem configurations that may be caused by satisfied activity constraints in the repairing stage.
 - (c) *incremental*, which is a combination of the static and dynamic strategies.
5. A random CCCSP generator is implemented, called CCCSP-Generator which is based on the RB model [120]. In addition, a range of phase transitions can be estimated.
6. Experimental comparative study over randomly generated problems.

Secondly, we have applied the CCCSP framework to a specific problem's domain, *temporal reasoning*. The CCCSP framework is extended to handle temporal constraints using model TemPro [74]. We named the framework Conditional and Composite Temporal CSPs or CCTCSPs. It is more expressive than TemPro and all other hybrid models.

My additional contributions in the case of temporal constraints are listed as follows [77, 78].

1. In a unified framework, CCTCSP is expressive in terms of representing numeric temporal information, symbolic temporal information, conditional events and composite

events [77].

2. Disjunctive numeric constraints can be handled by using composite variables and slicing techniques [78].
3. A random general CCTCSP generator based on model RB is implemented. It results in obtaining a more random general TCSP generator. The experimentation is performed on sets of randomly generated CCTCSP instances.

Thirdly, we extend preferences to both CCCSPs and CCTCSPs, called CCCSPPs and CCTCSPPs, respectively. The contributions for the proposed CCCSPP and CCTCSPP frameworks are described as follows [80].

1. The soft constraint based c-semiring approach [16] is redefined and applied to the CCCSP and the CCTCSP frameworks.
2. Variant branch and bound methods based on constraint propagation are implemented.
3. A new estimated function for guiding the search to find the best solution is defined.

1.3 Thesis Outline

The thesis is organized as follows:

Chapter 2 gives the background of CSPs. The formal CSP definition and binary constraint networks are recalled in Section 2.1. Two approaches for solving CSPs (complete methods and incomplete methods including searching strategies) are covered in Section 2.2 and Section 2.3, respectively. Section 2.4 refers to Dynamic CSPs, Conditional CSPs

and Composite CSPs. Soft constraints are introduced in Section 2.5. Finally, CSPs with preferences are discussed.

Chapter 3 presents temporal constraint reasoning. Numeric and symbolic constraints are reviewed in Sections 3.1.1 and 3.1.2. The TemPro framework is reviewed in Section 3.1.3. Section 3.2 explains dynamic aspects of temporal reasoning and existing frameworks. Section 3.3 discusses applying c-semiring structure to Simple Temporal Problems (STPs) and Allen relations.

Chapter 4 is dedicated to the proposed CCCSP framework. In Section 4.2.1, we discuss a weaker arc consistency for CCCSPs. Several resolution methods including constraint propagation are described in Section 4.2.1. Finally, in Section 4.2.2 incomplete methods are addressed.

In Chapter 5, we apply the CCCSP framework to temporal problems, called CCTCSPs. The formal definition of CCTCSP framework is addressed. Formulating temporal examples to CCTCSPs are demonstrated.

Chapter 6 describes extending CCCSP and CCTCSP frameworks with c-semiring to handle preferences. The c-semiring is integrated into CCCSP and CCTCSP frameworks. Four types of preference functions are introduced. Global and local preference aggregation functions are defined. A variant of the Branch and Bound technique with a new estimation function and constraint propagation techniques are discussed.

Chapter 7 describes experimental comparative tests. CCCSP and CCTCSP random generators based on model RB [120] are discussed. The experiments are divided into three parts: (1) complete methods, (2) incomplete methods and (3) Branch and Bound methods.

Finally, concluding remarks and future work are covered in Chapter 8. The future work continues on four directions: (1) composite variables including other composites, (2) solving CCCSPs and CCTCSPs in Unknown Dynamic Environments, (3) incomplete methods for CCCSPs and CCTCSPs with preferences and (4) Real World Applications.

2. CONSTRAINT PROCESSING

“If we can pose our world knowledge as a set of local constraints on the values assigned to variables in global solutions to a problem then when we satisfy those constraints we solve the problem.”

Alan K. Mackworth, 2003 [29]

In computational problems, rules, formulas and requirements can be depicted in terms of sets of constraints which play important roles in finding suitable answers. For instance, a high-school timetable provides the weekly scheduling for all the lectures of a high school. However lectures must be assigned to time slots in such a way that no teacher (or classroom) is involved in more than one lecture at a time. All assignments must be restricted to the requirements. Generally, constraints are used to check or evaluate solutions.

This chapter will refer to principle concepts of the constraint processing approach through the CSP framework: formal definition, resolution methods and enhancement techniques via constraint propagations, local consistency techniques and heuristics. Related CSP frameworks such as CCSPs, soft constraints and CSPs with preferences are discussed.

2.1 Constraint Satisfaction Problems (CSPs)

Over the last decades, the CSP approach has proven to be useful for modeling and solving constraint problem applications. The CSP concept was first introduced by Huffman

and Clowes in scene labeling problems [53, 23]. More general algorithms for solving CSPs have been studied [64, 65] and increasingly applied to solve a wide range of problems, among them Interactive Graphics [39], Configuration [93], Circuit Design [88], Natural Language Understanding [3, 35], Planing and Scheduling [105, 63], Expert Systems [34] and Temporal Reasoning [4, 31, 113, 109, 73, 106, 74, 77, 78]. CSP captures the core of most computational problems which are constraints and the possible values associated with variables. Thus, the problems can be formulated as CSPs.

The initial stage of the constraint processing approach is encoding problems. A given problem is analyzed and formulated into the generic form of a CSP. We restate a standard CSP definition from Mackworth [64].

Definition 2.1.1. *A Constraint Satisfaction Problem (CSP) is a triple $P = (V, D, C)$, where*

- $V = \{v_1, \dots, v_n\}$ is the set of variables;
- $D = \{D_1, \dots, D_n\}$ is the set of domains. Each domain is a finite set containing the possible values for the corresponding variable;
- $C = \{c_1, \dots, c_n\}$ is the set of constraints. A constraint c_i is a relation defined on a subset $\{v_{i_1}, \dots, v_{i_{k_i}}\}$ for all the variables, that is $\{D_{i_1} \times \dots \times D_{i_{k_i}}\} \supseteq c_i$.

A solution to a CSP is a complete assignment of values to all of the variables that satisfy all the constraints.

The high school timetabling problem can be encoded as a CSP. Each available time slot of the weekly timetable is represented as a variable, V_i , where its domain is the name

list of teachers, for instance $D_i = \{Peter, John, Susan\}$. If two time slots V_i and V_j overlap, a binary constraint C_{ij} , V_i 's value $\neq V_j$'s value is applied.

Constraints can be expressed either in an implicit (e.g., equations or formulas) or explicit (e.g., sets of incompatible or compatible tuples) manner. Determining an appropriate manner of constraints depends on the nature of each problem domain. However in general, explicit constraints can be generated from implicit constraints and vice versa.

Constraints are categorized by the number of associated variables. A constraint involving *one* variable, *two* variables or *n* variables is called a *unary* constraint, a *binary* constraint or a *n-ary* constraint, respectively. Unary and binary constraints commonly engage in problems. Unary constraints are easy to conduct and they can be converted to binary constraints. Furthermore, CSP with n-ary constraints can be encoded to another equivalent binary CSP [91, 103]. Positively, binary constraint techniques are adequate to tackle non-binary real world problems. Thereby, binary CSPs are a margin of our proposed framework.

2.1.1 Binary Constraint Network

A binary CSP can be depicted as a constraint network in which each node represents a variable, and each arc represents a constraint between variables represented by the end points of the arc. In the case of the high school timetabling problem, if all available time slots on Monday morning are given as in Figure 2.1, our formulation of the problem is depicted in Figure 2.2.

Fundamentally, two main steps of searching a CSP solution are

1. assigning variables and
2. checking conflicting constraints

Figure 2.1: A Timetable

Figure 2.2: An Equivalent Constraint Network of Timetabling Problem in Figure 2.1

Checking conflicting constraints is performed either after assigning one variable, a certain number of variables or all variables. All possible variable assignments of CSP are a combination of the variables' domains. CSP is an NP-complete problem where the

complexity of search space is D^N , where N is the number of variables and D is the size of the domain [66]. In the timetabling problem, there are $3^9 = 19683$ possible assignments. If V_1 is assigned to *Peter*, then we cannot assign *Peter* to V_7 and V_5 because it violates the constraints, $C_{1,7}$ and $C_{1,5}$. A consistent assignment is $V_1 = \textit{Peter}, V_2 = \textit{Peter}, V_3 = \textit{Susan}, V_4 = \textit{Susan}, V_5 = \textit{John}, V_6 = \textit{Peter}, V_7 = \textit{Susan}, V_8 = \textit{John}$ and $V_9 = \textit{John}$.

In Section 2.2, we describe complete search methods that guarantee to return a solution if the problem is consistent. Then in Section 2.3, we discuss alternative methods which are incomplete and do not guarantee finding a complete solution but that are better in terms of searching time.

2.2 Complete Methods

2.2.1 Generate and Test (GT) and Backtrack Search (BT) Techniques

Complete methods such as Generate and Test (GT) and Backtrack Search (BT) search for solutions in a systematic manner (also called Systematic Search), and guarantee to find complete solutions if they exist. In GT, a complete assignment is generated and then tested, one by one until a consistent solution (or all consistent solutions) is (are) found. If the CSP is inconsistent, the search space will be completely explored without returning solutions. No source of conflicts will be determined/recorded during each generated and tested assignment.

BT searches for a complete solution by extending a partial solution (see the algorithm in Figure 2.3 [29]). The search space is explored in a depth-first search manner. At each

stage, a new variable is assigned and checked against constraint violations of all previously instantiated variables. If the new variable can be assigned, it will be added to the current partial solution. Otherwise backtracking is performed (a new value is chosen to replace the previous assignment) on the previous recently added variable. Therefore, all possible assignments (which are surely inconsistent) belonging to the failed variable assignment are skipped. The process is repeated until all variables are assigned and then BT returns a complete solution or the search space is explored without success.

Similarly to GT, the time complexity for BT grows by the size of the search space. Nevertheless BT is better than GT because some of the inconsistent assignments can be detected earlier and skipped. Even though BT is more efficient than GT, BT still suffers from late detection of conflicts, thrashing of assignment and redundant checking [43]. Repeatedly rechecking the same inconsistencies and partial successes during the search is manifested, since each one step backtracking cannot directly reach the source of conflict. More advanced techniques to improve BT are discussed in the next section. In addition, local consistency and constraint propagation are used to prune the future search space during assignment of a partial solution (Section 2.2.3).

2.2.2 Backjumping and Dynamic Backtracking Techniques

The Backjumping (BJ) technique [43] improves BT by removing the thrashing problem. Instead of a backtrack to the last instantiated variable, the source of conflict is identified and then a back jump to the farthest back variable in conflict is made (see the algorithm in figure 2.4 [29, 43]). All violated constraints associated with the current variable and past variables are considered. For each value of the current variable, BJ finds

```

FUNCTION BT
Input: A constraint network  $P = (V, D, C)$ 
Output: Either solution, or notification that  $P$  is inconsistent
   $i \leftarrow 1$  (initialize variable counter)
   $\dot{D}_i \leftarrow D_i$  (copy domain)
  while  $1 \leq i \leq n$ 
    instantiate  $V_i \leftarrow \text{SELECT-VALUE}$ 
    if  $V_i$  is null (no value was returned)
       $i \leftarrow i - 1$ 
    else
       $i \leftarrow i + 1$ 
       $\dot{D}_i \leftarrow D_i$ 
  end while
  if  $i = 0$ 
    return "inconsistent"
  else
    return instantiated values of  $\{V_1, \dots, V_n\}$ 
END BT

SUBFUNCTION SELECT-VALUE (return a value in  $\dot{D}_i$ )
  while  $\dot{D}_i$  is not empty
    select  $a \in \dot{D}_i$ 
     $\dot{D}_i \leftarrow \dot{D}_i - a$ 
    if  $V_i = a \wedge$  satisfies  $\forall C_{m,i}$ , where  $1 \leq m \leq i$ 
      return  $a$ 
  end while
  return null (no consistent value)
END SELECT-VALUE

```

Figure 2.3: BT Algorithm

the closest variable in each violated constraint. In the list of all the closest variables, BJ selects the farthest variable.

```

FUNCTION BJ
Input: A constraint network  $P = (V, D, C)$ 
Output: Either solution, or notification that  $P$  is inconsistent
   $i \leftarrow 1$                                 (initialize variable counter)
   $\dot{D}_i \leftarrow D_i$                         (copy domain)
   $farthest_i \leftarrow 0$                     (initialize farthest variable)
  while  $1 \leq i \leq n$ 
    instantiate  $V_i \leftarrow \text{BJ-SELECT-VALUE}$ 
    if  $V_i$  is null                            (no value was returned)
       $i \leftarrow farthest_i$ 
    else
       $i \leftarrow i + 1$ 
       $\dot{D}_i \leftarrow D_i$ 
    end while
  if  $i = 0$ 
    return "inconsistent"
  else
    return instantiated values of  $\{V_1, \dots, V_n\}$ 
END BJ

SUBFUNCTION BJ-SELECT-VALUE (return a value in  $\dot{D}_i$ )
   $farthest_i \leftarrow i$ 
  while  $\dot{D}_i$  is not empty
    select  $a \in \dot{D}_i$ 
     $\dot{D}_i \leftarrow \dot{D}_i - a$ 
    if  $V_i = a \wedge \text{satisfies } \forall C_{m,i}, \text{ where } 1 \leq m \leq i$ 
      return  $a$ 
    else
      if  $farthest_i > \max(m)$  of  $\forall$  inconsistent  $C_{m,i}$ 
         $farthest_i \leftarrow \max(m)$ 
    end while
  return null                                (no consistent value)
END SELECT-VALUE

```

Figure 2.4: BJ Algorithm

Unfortunately, the non-violated in-between assignment is lost when jumping back to the source of conflict since the variables are preordered and static. This problem can be resolved by Dynamic Backtracking (DB) [49]. In DB (Figure 2.5 [29, 49, 9]), variable ordering can be changed during the back jump. All non-conflict past variables which

are pre-placed in between the inconsistent current variable and the farthest variable in conflict are placed prior to the farthest variable.

```

FUNCTION DB
Input: A constraint network  $P = (V, D, C)$ 
Output: Either solution, or notification that  $P$  is inconsistent
   $Labelled \leftarrow \{\}$ 
   $Unlabelled \leftarrow V$ 
  while  $Unlabelled \neq \{\}$ 
    select  $X$  in  $Unlabelled$ 
    instantiate  $X \leftarrow SELECT - VALUE$ 
    if  $X$  is null                                     (no value was returned)
      let  $E$  be an explanation of the conflict      (set of conflicting variables)
      if  $E = \{\}$  failure
      else
        let  $Y$  be the most recent variable in  $E$ 
        unassign  $Y$  (from labelled) with eliminating explanation  $E - (Y)$ 
        remove all the explanations involving  $Y$ 
      end if
    end while
END DB

```

Figure 2.5: DB Algorithm

2.2.3 Local Consistency Techniques

Local consistency techniques reduce the size of original CSP networks without removing any possible consistent solution. Some inconsistent values can be detected early and removed in the preprocessing stage. Searching becomes more efficient for the more explicit and tight (but equivalent) constraint networks. Levels of consistency-enforcing are determined by the number of variables involved.

Node Consistency (NC)

Definition 2.2.1. Node Consistency (NC) [9]. V_i is node consistent iff every value in D_i satisfies all the unary constraints imposed on the variable V_i . A CSP is node consistent iff every V_i is node consistent .

NC is the easiest consistency technique to reduce the size of domains. To make a CSP network node consistent, all inconsistent values in D_i with unary constraint imposed on V_i are removed. If a domain becomes empty, the CSP network is inconsistent [29].

```
FUNCTION NC
Input: A constraint network  $P = (V, D, C)$ 
Output: An equivalent NC  $\hat{P}$  or  $P$  is inconsistent
  for each  $V_i \in V$ 
    for each  $a \in D_i$ 
      if unary constraint on  $V_i$  is inconsistent with  $a$ 
         $D_i \leftarrow D_i - a$ 
      end for
    if  $D_i$  is empty
      return "INCONSISTENCY"
    end for
  end NC
```

Figure 2.6: NC Algorithm

Arc Consistency (AC)

Definition 2.2.2. Arc Consistency (AC) [9]. $Arc(V_i, V_j)$ is arc consistent iff every value x in D_i there exists a value y in D_j such that the valuation $V_i = x$ and $V_j = y$ satisfies all the binary constraints on V_i, V_j . CSP is arc consistent iff every $Arc(V_i, V_j)$ is arc consistent in both directions.

A CSP is made to be an AC CSP by repeating the revision procedure until there is

no change in the domains. Figures 2.7 and 2.8 illustrate the arc consistency algorithm AC-3 [64, 9]. $Arc(V_i, V_j)$ is obtained by removing all values x from the domain D_i that are inconsistent with all the values in D_j . Function $REVISE(i, j)$ is repeatedly called for in every arc in a CSP network to ensure arc consistency. AC techniques give one of the following conclusions:

- If a domain becomes empty, the CSP has no a solution (Backtrack search is not needed).
- If every domain has only one value left, there is one possible assignment for the problem. Only one checking of a consistent solution is required.
- AC gives a new simple consistent CSP where all the constraints are arc consistent.

Although a CSP is arc consistent, it does not mean the CSP must have a solution.

Different strategies AC-1, AC-2, AC-3, AC-4, AC-5, AC-6 and AC-7 attempt to reduce unnecessary repeated revision checking [61, 64, 9, 122, 70, 111, 11, 12, 13, 29]. AC-1, AC-2, AC-3 [64, 9], AC-3.1 [122] and AC-2001 [13] are based on repeating the arc revision procedure when a domain changes, while AC-4 [70], AC-5 [111], AC-6 [11] and AC-7 [12] are based on maintaining supports. Among them AC-3 is the most popular one that is easy to implement. It gives $O(CD^3)$ for the worst time complexity and $O(V^2D^2)$ for the space complexity, where C is the number of constraints, V is the number of variables and D is the domain size of variables.

```

FUNCTION REVISE( $i, j$ )
  DELETED  $\leftarrow$  false
  for each  $x \in D_i$ 
    if there is no such  $y \in D_j$  such that  $(X, Y)$  is consistent
       $D_i \leftarrow D_i - x$ 
      DELETE  $\leftarrow$  true
    end if
  end for
  return DELETED
end REVISE

```

Figure 2.7: Arc Revision Algorithm

```

FUNCTION AC-3
Input: A constraint network  $P = (V, D, C)$ 
Output: An equivalent AC  $\dot{P}$  or  $P$  is inconsistent
   $Q \leftarrow \{(V_i, V_j) \in C, i \neq j\}$ 
  while  $Q$  is not empty
    select and delete any arc  $(V_k, V_m)$  from  $Q$ 
    if REVISE( $V_k, V_m$ ) then
      if  $D_k$  is empty
        return "INCONSISTENCY"
       $Q \leftarrow Q \cup \{(i, k) | (i, k) \in C, i \neq k, i \neq m\}$ 
    end if
  end while
end AC-3

```

Figure 2.8: AC-3 Algorithm

Path Consistency (PC)

Path Consistency (PC) is a stronger consistency technique, and it is a natural extension of arc consistency. Instead of considering inconsistent values of one arc between a pair of variables, PC considers inconsistent values among every pair of variables on a given path [72]. In other words, path length of AC is equal to one, whereas path length of PC is greater than or equal to two.

Definition 2.2.3. Path Consistency (PC) [9]. *path($V_0, \dots, V_m$) is path consistent iff for every pair of values $x \in D_0$ and $y \in D_m$ satisfying all the binary constraints on V_0, V_m , there exists an assignment of variables $V_1, \dots, V_{m-1}$ such that all the binary constraints between the neighbouring variables V_i, V_{i+1} are satisfied. CSP is path consistent iff every path is consistent.*

Montanari has proven that path consistency is equivalent to 3-consistency and paths of length two are enough to make the CSP path consistent [72]. Therefore, most PC algorithms only consider a path consistency of length two. A number of algorithms PC-1 [64, 66], PC-2 [64, 66], PC-3 [70], PC-4 [51], PC-5 [100] and PC-6 [21] have been designed for achieving path consistency in constraint networks. Like AC algorithms, PC algorithms make paths consistent by repeating revisions. PC removes inconsistent values. Furthermore PC eliminates inconsistent binary constraints. However PC algorithms of length two cannot guarantee that all the constraints among the variable on the path are satisfied.

Applying NC, AC and PC techniques in the preprocessing stage makes the original CSP tighter. In other words, the size of the equivalent CSP is smaller in terms of the

size of the domain. For highly constrained problems where the number of the consistent solutions is small, most inconsistent values or inconsistent constraints are eliminated from the problem. Unfortunately, in less and medium constrained problems, only a small number or none of inconsistent values can be deserted. In the next section, we will discuss integrating arc consistency techniques inside the searching stage.

2.2.4 Constraint Propagation

Constraint Propagation [9, 29] is a technique for checking arc consistency of unassigned variables against assigned variables (a partial consistent solution) in the searching stage. BT search is integrated with an AC technique to propagate constraint checking. Degrees of constraint propagation indicate the group of unassigned variables. Forward Checking (FC) only considers the next connected unassigned variables called *next future* variable (i.e., variables that have their associated constraints with the current assigning variable) (Figure 2.9 [9]). Full Look Ahead (FLA) propagates constraints farther than FC. FLA also considers the group of unassigned variables which are connected to the next future variables. If constraints are propagated to all unassigned variables, we also call this approach Maintaining Arc Consistency (MAC) (Figure 2.10 [9]).

In the next section, we describe heuristics to order variables and values in order to improve the efficiency of complete methods.

2.2.5 Variable and Value Ordering

Different orders of assigning variables and selecting values result in different search times in practice [32, 116, 108, 29]. A good order can avoid a lot of backtracking or be

```

FUNCTION AC-3 for FC
Input: A constraint network  $P = (V, D, C)$ 
Output: return true for consistent and false for inconsistent
 $Q \leftarrow (V_i, V_{cv}) \in \text{arcs}(C), i > cv$ 
while  $Q$  is not empty
    select and delete any arc  $(V_k, V_m)$  from  $Q$ 
    if  $REVISE(V_k, V_m)$  then
        if  $D_k$  is empty
            return false
         $Q \leftarrow Q \cup \{(i, k) | (i, k) \in C, i \neq k, i \neq m\}$ 
    end if
end while
return true (consistent)
end AC-3 for FC

```

Figure 2.9: AC-3 for FC Algorithm

```

FUNCTION AC-3 for MAC
Input: A constraint network  $P = (V, D, C)$ 
Output: return true for consistent and false for inconsistent
 $Q \leftarrow (V_i, V_{cv}) \in \text{arcs}(C), i > cv$ 
while  $Q$  is not empty
    select and delete any arc  $(V_k, V_m)$  from  $Q$ 
    if  $REVISE(V_k, V_m)$  then
         $Q \leftarrow Q \cup \{(V_i, V_k) \text{ such that } (V_i, V_k) \in \text{arcs}(G), i \neq k, i \neq m, i > cv\}$ 
        if  $D_k$  is empty
            return false
         $Q \leftarrow Q \cup \{(i, k) | (i, k) \in C, i \neq k, i \neq m\}$ 
    end if
end while
return true (consistent)
end AC-3 for MAC

```

Figure 2.10: AC-3 for MAC Algorithm

backtrack free. The ordering may be either static or dynamic [89]:

1. in a static ordering, the order of the variables/values is specified in the preprocessing stage before the search begins, and it is not changed thereafter;
2. in a dynamic ordering, the choice of the next variable/value to be considered, depends on the current stage of the search.

In the following portion of Section 2.2.5, different variable and value ordering heuristics for CSP are introduced.

Variable Ordering Heuristics [108]

Minimal Width Ordering (MWO) is a heuristic for static ordering of variables. First, the variables are arranged in sets of different orders from most constrained variables to less constrained variables. We calculate the width of each ordering. The width of node V is the number of nodes before V and adjacent to V . So the width of node V depends on its order. The width of an ordering is the maximum width of all the nodes in its ordering. Finally, we use the ordering that gives the minimal width of all possible ordering.

Minimal bandwidth ordering (MBO) is a heuristic for a dynamic ordering. The current variable and all of the future variables are reordered before backtracking is required. The bandwidth of each possible order is calculated. The bandwidth of node V in an ordered graph is the maximum distance between V and any node which is adjacent to V according to the ordering. Similar to MWO, the maximum bandwidth of all the nodes represents the bandwidth of the ordering. Then, an ordering having the minimal bandwidth is the new variable ordering.

The Fail First Principle (FFP) is a general heuristic for searching. It refers to the performing a task that is most likely to fail first. Measuring failure can be considered by constraint and size of the domain of variables. The variable that has the most constraints or the variable that has the smallest domain is assumed to be most likely to lead to failure. Therefore, these variables are considered first. FFP can be implemented in either a dynamic or a static manner.

Value Ordering [108]

Value ordering can have a substantial impact on the time it takes to find the first solution. Once the decision is made to instantiate a variable, it may have several values available. Changing the value ordering rearranges the branches emanating from each node of the search tree. Such a change is an advantage if it ensures a branch that leads to a solution is searched prior to branches that lead to failure. For example, if the CSP has a solution and if a correct value is chosen for each variable, then a solution can be found in a backtrack free manner. If all solutions are required or there are no solutions, then value ordering is not beneficial.

The **succeed first principle** is a strategy in which a value likely to succeed is selected first. One possible heuristic is to prefer those values that maximize the number of options available. Algorithm AC-4 [70] uses this heuristic as it counts the supporting values. The value that has the most supporters is selected first. Another heuristic is to prefer a value that leads to a structure for the CSP that can be solved more easily. An example of a CSP with a simple structure is one that can be converted into a tree-structured CSP by deleting a minimum number of arcs because an efficient procedure exist to find all

solutions of the resulting CSP.

For randomly generated problems, and possibly in general, the work involved in assessing each value is not worth the benefit of choosing a value which, on average will more likely lead to a solution than the default choice. On the other hand, in particular problems, there may be information available which allows the values to be ordered according to the principle of choosing first those most likely to succeed.

In summary, GT, BT, BJ and DB are complete methods that guarantee the return of a complete solution if the problem is consistent. Complete methods explore the whole search space. Some local consistency techniques (NC, AC and PC) are applied to reduce the search space in the preprocessing stage. FC and MAC are constraint propagation techniques to improve BT search. In addition, variable and value ordering may assist a complete method in finding a solution more quickly. In the next section we introduce alternative solving techniques called incomplete methods.

2.3 Incomplete Methods

Unlike complete methods, incomplete methods (i.e., greedy search) initiate and explore a search space from a randomly chosen complete assignment [29, 9, 89]. While a consistent solution has not been found, such a method the inconsistent assignment continues to repair to minimize the number of conflicts. Since not all of the search space is explored, there is no guarantee that this approach will find a consistent solution if it exists.

The Hill Climbing (HC), Min Conflict (MC), Min Conflict Random Walk (MCRW), Steepest Descent Random Walk (SDRW) and Tabu Search (TS) algorithms are based on

a common idea known as Stochastic Local Search (SLS) [97, 1]. In local search, an initial configuration (consisting of values for all variables) is generated and the algorithm moves from the current configuration to a neighboring configuration until a solution (decision problems) or a good solution (optimization problems) has been found or the resources available are exhausted. Stochastic local search algorithms use different repair heuristics and stochastic algorithms to guide searching.

The common steps of HC, MC, RW, MCRW, SDRW and TS are [61]:

1. Randomly select a complete assignment as the current state.
2. Evaluate the numbers of violated constraints of the current state.
3. Try to repair the current state by changing to a better neighbouring state (a state that differs from the current state in the value of one variable of the evaluation).
4. If the current state is not the global optimum and it cannot be improved, the state is detected as a local optimum.
5. Escape from the local optimum, if necessary.
6. Repeat step 2 to 5 until a global optimum is found or resources are exhausted.

A neighbour is a state identical to the current state except that one assigned variable has a different value. The methods of choosing a neighbouring state and escaping from local optimum are different in each of the following: HC, MC, RW, MCRW, SDRW and TS [61].

HC considers the neighbours, that differ in the value of all conflict variables. The HC neighbourhood is large because all conflict variables are explored to find the best

neighbour of the current state. HC escapes from the local optimum via restarting with a new random assignment of a variable (Figure 2.11 [9]).

```

FUNCTION Hill-Climbing(MaxMoves)
  S ← random an initial statement of variables
  Moves ← 0
  while Moves ≤ MaxMoves
    if eval(S) = 0 then return S
    if S strict local minimum then
      randomly pick a new initial statement of variables
      Moves ← 0
    else
      S ← neighbourhood with the smallest evaluation value
      Moves ← j + 1
    end if
  end while
  return S
END Hill-Climbing

```

Figure 2.11: Hill-Climbing Algorithm

MC considers the neighbours, that differ in the value of a conflict variable. The MC neighbourhood is smaller than the HC neighbourhood because only one conflict variable is explored at a time. Similar to HC, MC escapes from the local optimum via restarting with a new random assignment of a variable.

RW considers neighbours that can be any randomly selected neighbour. Finding a solution with this technique is difficult because it has no heuristic for searching.

MCRW improves MC through leaving the local optimum by adding some noise to the algorithm. Noise is added by introducing a random walk to the algorithm. MCRW combines heuristic search and random walk (Figure 2.12 [9]). The probability of using the random walk is p , and the probability of using a heuristic to guide the search is $1-p$.

SDRW combines HC with RW. The algorithm will not restart every time it is trapped

```

FUNCTION MCRW(MaxMoves, p)
  S ← a random initial statement of variables
  Moves ← 0
  while Moves ≤ MaxMoves and eval(S) > 0
    if probability p verified then
      randomly choose a variable V in conflict
      randomly choose value v for V
    else
      randomly choose a variable V in conflict
      choose a value v that minimize the number of conflicts for V
    end if
    if v ≠ current value of V then
      assign v to V
      Moves ← Moves + 1
    end if
  end while
  return S
END MCRW

```

Figure 2.12: MCRW Algorithm

in a local optimum. As with MCRW, the probability of using the random walk is p and the probability of using the heuristic to guide the search is $1-p$ (Figure 2.13 [9]).

TS memorizes a list of changed variables and values $\langle V, v \rangle$ of a few of the last visited states (called *tabu list*). The tabu list contains forbidden states that a next neighbouring state cannot have. However there is an aspiration criterion that allows the forbidden states when the move leads to a better solution than has been obtained. The tabu list strategy prevents TS from being trapped in short-term cycling and allows the search process to go beyond the local optimal (Figure 2.14 [9]).

```

FUNCTION SDRW(MaxMoves, p)
  S ← a random initial statement of variables
  Moves ← 0
  while Moves ≤ MaxMoves and eval(S) > 0
    if probability p verified then
      randomly choose a variable V in conflict
      randomly choose a value v for V
    else
      choose a move  $\langle V, v \rangle$  with the best performance
    end if
    if v ≠ current value of V then
      assign v to V
      Moves ← Moves + 1
    end if
  end while
  return S
END SDRW

```

Figure 2.13: SDRW Algorithm

```

FUNCTION Tabu-Search(MaxMoves)
  S ← a random initial statement of variables
  Moves ← 0
  randomly initialize the tabu list
  while Moves ≤ MaxMoves and eval(S) > 0
    choose a move  $\langle V, v \rangle$  with the best performance among the non-tabu moves
    and the moves satisfying the aspiration criteria

    adds  $\langle V, v \rangle$  in the tabu list, where v is the current value of V
    remove the oldest move from the tabu list
    assign v to V
    Moves ← Moves + 1
  end while
  return S
END Tabu-Search

```

Figure 2.14: Tabu Search Algorithm

2.4 Dynamic Constraint Satisfaction Problems

Dynamic Constraint Satisfaction Problems (DCSP) involve the change of variables, values and constraints. Sources of change are either from *external* or *internal* environments. The external environment means availability of resources and the change of requirements over time or uncertainty of constraints which are mostly unpredictable. Temporarily, some variables, constraints or values may not be considered. Moreover, new variables, constraints or values are involved in the current solutions. Therefore, the current solutions are no longer valid. Dynamic searching algorithms will reuse and repair a current solution when variables, domains and constraints have changed [30, 10, 112, 75].

The internal environment implies that all changeable situations are known under predefined conditions. Mittal and Falkenhainer [69] firstly proposed a dynamic CSP framework involving conditional variables for configuration problems. There are two sets of variables: initial variables and conditional variables. Initial variables are main variables that are always required in a solution. Conditional variables are variables that will be in a final solution only if their conditions are satisfied. A new type of constraint, called an *activity constraint* was introduced to limit participation of the conditional variables (e.g., $V_i = a \xrightarrow{\text{require}} V_j$ means if $V_i = 1$, V_j will be added into the solution). This framework is named Conditional CSPs (CCSPs) to distinguish it from the dynamic CSPs caused by external environments. (Our thesis is mainly addressed on this CCSP framework described in Section 2.4.1).

Additional research on CCSPs has been published. Bowen and Bahler [20] suggested

a set of sentences in free logic to handle conditional existence of variables in a mathematically well-founded fashion. Gelle and Falting [45] expanded CCSPs for mixed domains involving both numeric and discrete values.

Disjunctive and *clustering* are two complicated properties of activity constraints which are addressed in configuration domains [69]. For example:

If $V_i = a$, either V_j or V_k will be added into solutions.

$$\text{Disjunctive : } V_i = a \xrightarrow{\text{require}} V_j \vee V_k$$

If $V_i = b$, both V_m and V_n will be added into solutions.

$$\text{Clustering : } V_i = b \xrightarrow{\text{require}} V_m \wedge V_n$$

Soininen et al. [102] proposed a fix-point condition framework to handle disjunctive activities and default negations. They showed that extending the original Conditional CSP, easily results in $\sum_2 P$ -hard decision problems. Geller and Veksler [47] defined a new framework called Activity CSPs (ACSPs) to manage disjunctive and clustering activity constraints. The ACSP framework has two differences from the conditional CSP framework. First, activity variables (with boolean domains) are defined and will participate in every solution. Each non-active variable is associated with an assignment of activity variable. Second there is no distinction between compatibility and activity constraints.

Jónsson and Frank [56] proposed a general framework using procedural constraints for solving dynamic CSPs. This framework has been extended to a new paradigm called Constraint-Based Attribute and Interval Planning (CAIP) for representing and plan reasoning [38]. CAIP and its implementation, (the EUROPA system), enable the description

of planning domains with time, resources, concurrent activities, disjunctive preconditions and conditional constraints.

Composite CSP is an another internal dynamic CSP framework. Sabin and Freuder [93] have extended the standard CSP framework by including the combination of three new CSP paradigms: Meta CSPs, Hierarchical Domain CSPs, and Dynamic CSPs. The variable values can be an entire subset of CSPs. A domain can be a set of variables instead of atomic values. The domains of variable values can be hierarchically organized.

Conditional CSPs and Composite CSPs were specifically introduced for configuration problem domains. As stated in Chapter 1, we observed that these properties are also common in varying problem domains. The entire thesis entails dealing with the internal dynamic environment caused by conditional variables and composite variables. Our composite variables differ from the Composite CSP framework. Here the domain of composite variables is a set of variables instead of a set of atomic values. The composite variables represent a high level of choice selections. In addition, we also handle optional variables and disjunctive activity constraints via the composite variables. The full details of our framework are given in Chapter 4.

2.4.1 Conditional CSPs (CCSPs)

In this section we discuss more details of CCSP of Mittal [69]. To control participation of the additional/conditional components in a system, an *activity constraint* is extended to the standard CSP. Activity constraints trigger the status of variables. In conditional CSPs, a variable has either *active* or *non-active* status. All initial variables which have to be in every solution are set active whereas the additional/conditional variables are set

non-active. Each non-active variable has at least one associated activity constraint (i.e., a conditional if-then rule). While extending a partial solution if of an activity constraint is fired, a new variable is added into the problem by changing its status from non-active to active (if it is not yet active). A solution is only the set of active variables which require value assignments. In the following we define the conditional CSP.

Definition 2.4.1. A *CCSP* is a CSP $\langle V, D, C_c \rangle$ with additional tuple $\langle V_I, C_A \rangle$ where

- V_I is the set of initial variables, $V_I \subseteq V$;
- C_A is the set of activity constraints. Given two variables V_i and V_j , four types of activity constraints have the following forms:

active : $(V_i) \wedge ((V_i = a_{i1}) \vee \dots (V_i = a_{ip})) \rightarrow V_j$ (1) *require variable*

active : $(V_i) \rightarrow V_j$ (2) *always require*

active : $(V_i) \wedge ((V_i = a_{i1}) \vee \dots (V_i = a_{ip})) \xrightarrow{rn} V_j$ (3) *require not*

active : $(V_i) \xrightarrow{rn} V_j$ (4) *always require not*

A solution to a CCSP is an assignment of active variables that satisfies all compatibility and activity constraints.

In definition 2.4.1, the *require* activity constraint will activate V_j if the active variable V_i is assigned one of the values $a_{i1} \dots a_{ip}$ from its domain whereas the *always require* activity constraint will activate V_j if V_i is active. In contrast, for the *require not* and the *always require not* activity constraints, if the condition is true, V_j is not required in a solution, or is set to a *non-active* variable. The order of fired activity constraints is important. In a situation where $V_i = a \rightarrow V_j$ is fired earlier (V_j becomes active), then $(V_{i+k} = b) \xrightarrow{rn} V_j$ is fired later. In this case, V_j cannot be changed to *non-active*

as long as $V_i = a$ even though the assignment $V_{i+k} = b$ is consistent with all of the previous assignments. An additional unary constraint, $V_{i+k} \neq b$ is dynamically imposed here to ensure the consistency of all activity constraints. Saying it in another way, if $(V_{i+k} = b) \xrightarrow{rn} V_j$ is fired earlier, an unary constraint, $V_i \neq a$, is imposed.

Solving CCSPs by Complete Methods

Conditional CSPs can be solved by three different approaches [69, 44, 102, 94, 46]:

1. Reformulating an activity constraint into compatibility constraints by adding a *null* value to the domain of conditional variables. A CCSP is reformulated into a CSP. Then general CSP techniques are directly applied to solve the reformulated CSP [44, 94, 46].
2. Viewing a conditional CSP as a meta-CSP and then generating a set of all possible standard CSPs from a *Conditional CSP*. General CSP techniques are directly applied to solve each CSP [69, 46].
3. Solving a conditional CSP directly. Problem spaces are simultaneously generated during the resolution process [69, 94, 44].

A CCSP is an NP-hard problem [102]. In practice, the comparison of the performance in time and space of the three techniques is in a decreasing order from (2) $\succ$ (3) $\succ$ (1) [46]. The weakness of reformulating CCSPs (approach 1) is the increase in domain size, constraints and variables of the reformulated problems. The generation of all possible CSPs (approach 2) from a CCSP consumes more memory space for all generated CSPs, in which some of them may not have been explored during the search. A small change of an activity

constraint causes reformulating a whole new equivalent CSP or regenerating all possible sets of standard CSPs. The direct solving of CCSPs (approach 3) is promising. There is neither an increase in the size of a problem nor a requirement for extra memory space. Integrating constraint techniques into the third approach prunes inconsistent problem configurations [94, 45, 46].

To improve the performance of the direct solving of CCSPs approach, the constraint propagation techniques of Forward Checking (FC) and Maintaining Arc Consistency (MAC) are applied during the search and also to the newly active variables resulting from any satisfied activity constraint [94]. In addition, activity constraints are propagated during the search. However, we observed that using such constraint propagation still suffers from thrashing and redundant work at some points when a new variable is added to a solution. If the newly added variable is inconsistent with the previous assignments and is always required in the current assignment, then the backtracking is performed until it returns to the source of the variable in conflict.

To the best of our knowledge, no work in the existing literature has employed incomplete methods to solve CCSPs. One of the reasons could be because the CCSP approach was first motivated by configuration problems which only require complete solutions and in which time is not a critical resource. As previously mentioned, one strategy for incomplete methods is based on SLS. Although SLS neither explores all the search space nor guarantees finding a consistent solution, it is efficient in terms of time complexity. Incomplete methods are alternative resolutions for some real time applications where a trade off between solution quality and speed is admissible. In Chapter 4, we proposed applying SLS (i.e., MCRW, SDRW and TS) for solving CCSPs. MCRW, SDRW and TS

are applied with different strategies to control changing status of conditional variables in the repairing stage.

2.5 Soft Constraints

In the standard CSP framework, constraints are limited to represent obvious and explicit knowledge. All defined constraints must be satisfied. These constraints are called *hard constraints*. In fact in many real life problems vague/incomplete information, cost functions as well as preference choices are also presented. Partial solutions are acceptable. This kind of information and the relaxation of rules are not hard constraints but can be viewed as weaker types of constraints called *soft constraints*. Specific CSP extended models using soft constraints are Fuzzy, Probabilistic and Weighted CSPs. In addition, soft constraints are useful for indicating preferred solutions.

A Fuzzy CSP (FCSP) is a CSP in which all eligible tuples, t , of a constraint (C_i) are associated with preference levels between 0 and 1 [92, 16]. If $C = \{C_1, \dots, C_n\}$, then there is a function $f(C_i) : t \rightarrow a$ where $a \in [0, 1]$. The global preference degree of a solution, S , is computed from $\min \{f(C_i) : t \rightarrow a\}$. The best solution is a solution that holds the maximum global preference degree among all possible solutions. For over-constrained problems, there are no complete solutions. The FCSP framework is applied to a search for a partial solution [114].

In uncertain environments, a given problem cannot be completely predefined. Some constraints that will affect the outcomes of solutions are ambiguous at the time of the problem definition or these constraints may change over time. For example, say that

there is a 50% chance of snow today. To deal with such a partially known situation, a probability of this situation occurring is correlated to constraints.

A Probabilistic CSP (Prob-CSP) framework associates each constraint c with a certain probability $p(c)$ or with a probability $1 - p(c)$ in that c is not in the problem or a violated constraint [36]. A solution probability is $Sum \{Product \{p(c)\}\}$ or $Product \{(1 - p(c))\}$. The maximum among all possible assignments of values is the most possible solution.

A Weighted CSP (WCSP) framework is usually applied to optimization problems. In contrast with FCSPs, each compatible tuple, t , in each constraint, is associated with a cost, C , (e.g., $cost(C, t)$). The total cost of a solution is the sum of all assignment tuple costs in the solution. The minimum cost solution is the optimal solution.

In the next section, two generic soft CSP models are covered: *Valued CSPs* and *Semiring-Based CSPs*.

2.5.1 Valued Constraint Satisfaction

A *Valued CSP (VCSP)* framework is a generic soft constraint model based on an ordered monoid structure, called *valuation structure*. The definition of a VCSP is described as follows [95].

Definition 2.5.1. A *valuation structure* is defined as a tuple $\langle E, \odot, \succ \rangle$ such that:

- E is a set, whose elements are called *valuations*, which is totally ordered by $\succ$, which is a maximum element noted $\top$ and a minimum element noted $\perp$;
- $\odot$ is a commutative, associative closed binary operation on E that satisfies:
 - Identity: $\forall a \in E, a \odot \perp = a$;

- *Monotonicity:* $\forall a, b, c \in E, (a \succeq b) \Rightarrow ((a \odot c) \succeq (b \odot c));$
- *Absorbing element:* $\forall a \in E, (a \odot \top) = \top$

Definition 2.5.2. A *valued CSP* is defined by a classical $CSP\langle V, D, C \rangle$, a valuation structure $S = (E, \odot, \succ)$, and an application φ from C to E . It is denoted by $\langle V, D, C, S, \varphi \rangle$ and is called the valuation of c .

2.5.2 Semiring-based Constraint Satisfaction

A semiring-based constraint satisfaction is another generic soft constraint model. The standard CSP framework is extended with a semiring definition in order to express soft constraints. We recall a semiring definition from Bistarelli et al. [16].

Definition 2.5.3. A *semiring* is a tuple $\langle A, +, \times, 0, 1 \rangle$ such that:

- A is a set and $0, 1 \in A$;
- $+$, called the additive operation, is a close (i.e., $a, b \in A$ implies $a + b \in A$), commutative (i.e., $a + b = b + a$) and associative (i.e., $a + (b + c) = (a + b) + c$) operation such that $a + 0 = 0 + a$ (i.e., 0 is its unit element);
- $\times$, called the multiplicative operation, is a closed and associative operation such that 1 is its unit element and $a \times 0 = 0 = 0 \times a$ (i.e., 0 is its absorbing element);
- $\times$ distributes over $+$ (i.e., $a \times (b + c) = (a \times b) + (a \times c)$).

A *c-semiring* is a semiring such that $+$ is idempotent (i.e., $a \in A$ implies $a + a = a$), $\times$ is commutative, and 1 is the absorbing element of $+$.

Definition 2.5.4. A *Semiring based CSP (SCSP)* is a tuple $SCSP = \langle S, D, V \rangle$ where $S = (A, +, \times, 0, 1)$ is a c-semiring, D is a finite set, and V is an ordered set of variables. A constraint over SCSP is a pair $\langle def, con \rangle$, where $con \subseteq V$ is called the type of the constraint, and $def : D^k \rightarrow A$ (where k is the size of con , that is, the number of variable in it), is called the value of the constraint. Moreover, a constraint problem P over SCSP is a pair $P = \langle C, con \rangle$, where C is a set (or a multiset if $\times$ is not idempotent) of constraints over SCSP and $con \subseteq V$

In other words, each constraint is associated with a value ($a \in A$). A partial order relation $\leq$ is defined over A to compare tuples of values and constraints. This value can be interpreted as weight, or cost, or degree of preference, etc. The two operators define combination and projection over the set of constraints. Different properties of semiring define different types of CSP frameworks [16].

Table 2.1: Instances of SCSPs

Framework	Semiring
Standard CSPs	$S = \langle \{0, 1\}, \vee, \wedge, 0, 1 \rangle$
Fuzzy CSPs	$S = \langle \{x x \in [0, 1]\}, max, min, 0, 1 \rangle$
Probabilistic CSPs	$S = \langle \{x x \in [0, 1]\}, max, \times, 0, 1 \rangle$
Weighted CSPs	$S = \langle \mathbb{R}^-, max, +, -\infty, 0 \rangle$

2.5.3 CSPs with Preferences

This section is intended to discuss handling preferences. Many kinds of preferences are classified and studied in Rossi [90] and Bistarelli et al. [17]. Preferences are mainly classified into either qualitative categories (“I prefer ice tea to ice coffee and juice”) or quantitative categories (“I rank my favorite beverages for ice tea, ice coffee and juice at

10, 8 and 4 respectively”). Qualitative preferences are usually used in natural language to express preferences over simple and small sets of information, while quantitative terms indicate degrees of preferences. In the qualitative preferences, we are only able to see which item is preferable over another one. However, as seen in the preceding example, we do not know how much stronger or weaker the desire for ice tea or ice coffee is to juice in a quantitative manner. Rather, we must use a quantifier adjective (e.g., a lot, very much or a little).

In addition, several properties of preferences are observed. Preferences can depend on conditions (e.g., “If it is a late night, I prefer juice to coffee. Otherwise I prefer coffee to juice.”) Playing with words in qualitative preferences result in another two new properties: positive and negative. For example a positive qualitative preference “I like coffee, and I like tea even more than coffee” or a negative qualitative preference “I do not like coffee, and I really do not like tea.” Handling preferences with all these properties and exploring any new preference property draws a lot of attention in our CSP community.

Bistarelli et al. [17] extends a semiring-based soft constraint framework to handle quantitative preferences in both positive and negative dimension. Negative preferences are modeled on c-semiring $\langle N, +_n, \times_n, \perp_n, \top_n \rangle$. A new positive preference structure in a tuple $\langle P, +_p, \times_p, \perp_p, \top_p \rangle$ is defined to handle positive preferences. P is a set and $\top_p, \perp_p \in P$. $+_p$ is the additive operation which has the same properties as the corresponding one in c-semiring. $\times_p$ is the multiplicative operation. It is different from $\times$ in c-semiring. The best element in the ordering ($\top_p$) is the absorbing element, while the worst element ($\perp_p$) is the unit element. The two semiring structures are combined and integrated to CSPs as bipolar constraints.

The CP-net framework is a model for qualitative and conditional preferences under ceteris paribus [19, 26, 5, 87]. Preferences are represented separately from hard constraints. CP-nets are a graphical model for expressing a set of conditional ceteris paribus preference statements.

Lexicographically ordered CSP is another alternative framework for preferred variables and values [40]. In this latter model, variable selection is the primary factor while value assignment is secondary. Recently, this framework has been extended to Conditional lexicographic CSP [115] for conditional preferences.

2.6 Conclusions

We discussed CSP literature which covers formal definitions, complete methods, consistency and constraint propagation techniques, incomplete methods, conditional CSPs as well as soft constraints. Our proposed CCCSP framework is based on the solid CSP principles. We redefined the conditional CSP of Mittal and Falkenhainer [69] with a new type of variable and a new form of activity constraint. The FC and MAC propagation techniques are applied in our complete resolution techniques. The MCRW, SDRW and TABU techniques are used in our alternative incomplete resolutions. Preferences are handled by the c-semiring structure of Bistarelli et al. [16]. The full detail of our proposed framework including examples is described in Chapter 4.

3. TEMPORAL CONSTRAINT PROCESSING

“Time is not an empirical concept. For neither co-existence nor succession would be perceived by us, if the representation of time did not exist as a foundation a priori.”

Immanuel Kant, 1781

Time is crucial in every type of human activity. Many problems in this world can be described in terms of time and events/actions. For example, in scheduling and planning, it is obvious that one must order tasks within the available time schedule subject to any given conditions. In exploring archeology and the history of mankind, a time line is used to list important events for successive years within a particular historical period. In the health care domain, for instance, the progression of a disease assists doctors in diagnosing diseases and allows them to prescribe remedies for patients according to the symptoms in each developing phase. Patients receive different kinds of treatments according to their health conditions. Another domain where time-dependent conditions exist is simulation. For example, when simulating launching a rocket, the ordering of the relationships among the missions' operations are specified according to a given time period. Finally, time is an element in every human language (e.g., in an English sentence, a specific time distinction is expressed by tenses) and must be considered in natural language understanding systems.

A *temporal CSP* is a particular class of CSP where variables (called *events*) are defined on time points or time intervals and constraints are sets of allowed temporal relations between the events. A temporal constraint can be described in terms of either symbolic or numeric form. In general, both symbolic and numeric forms are used together to express contents of temporal information. In this chapter, we explain how to represent symbolic and numeric temporal constraints. Integrating preferences into temporal constraint networks is addressed. In addition, some dynamic temporal constraint frameworks are discussed.

3.1 Temporal Constraints

3.1.1 Numeric Temporal Models

A notable numeric temporal model is the Temporal Constraint Satisfaction Problem (TCSP) model developed by Dechter et al. [31]. A TCSP is specified as a set of time points (variables) having continuous domains and a set of temporal constraints (binary constraints). With this model, an event is represented by a beginning time point and an ending time point (i.e., $Event = [begin, end]$). Each time point is represented by a variable. A temporal constraint, T_{ij} , between two time points X_i and X_j is a set of intervals $\{I_1, \dots, I_k\} = \{[a_1, b_1], \dots, [a_k, b_k]\}$ where a_i and b_i are the minimum and the maximum time differences between time points X_i and X_j . The set of intervals represents the disjunction

$$(a_1 \leq X_j - X_i \leq b_1) \vee, \dots, \vee (a_k \leq X_j - X_i \leq b_k)$$

If a unary constraint involves in a TCSP, the unary constraint is converted to an equivalent binary temporal constraint by using an initial time point referring to the beginning of the scenario. Consider the following example (taken from [31]):

Example 3.1.1.

John goes to work either by car (30-40 minutes), or by bus (at least 60 minutes). Today John left home between 7:10 and 7:20, and Fred arrived at work between 8:00 and 8:10. John arrived at work about 10-20 minutes after Fred left home. We wish to know if the story is consistent. Is it possible that John took a bus and Fred used the carpool? What are possible times when Fred left home, and so on.

If J is an event of “John going to work” then J_{begin} and J_{end} are the starting and ending times of J , respectively. If F is an event of “Fred going to work” then F_{begin} and F_{end} are the starting and ending times of F , respectively. E_{start} is a time at least as early as the beginning of any of the events; for this example E_{start} is 7:00a.m. The numeric temporal information can be formulated as a set of inequalities described as follows (Figure 3.1 represents the same information in pictorial form):

- John goes to work either by car (30-40 minutes), or by bus (at least 60 minutes)
 $\equiv 30 \leq J_{end} - J_{begin} \leq 40$ or $J_{end} - J_{begin} \geq 60$
- Fred goes to work either by car (20-30 minutes), or in a carpool (40-50 minutes)
 $\equiv 20 \leq F_{end} - F_{begin} \leq 30$ or $40 \leq F_{end} - F_{begin} \leq 50$

- Today John left home between 7:10 and 7:20

$$\equiv 10 \leq J_{begin} - E_{start} \leq 20$$

- Fred arrived at work between 8:00 and 8:10

$$\equiv 60 \leq F_{end} - E_{start} \leq 70$$

- John arrived at work about 10-20 minutes after Fred left home.

$$\equiv 10 \leq J_{end} - F_{begin} \leq 20$$

Figure 3.1: A Numeric Temporal Constraint Network [31]

Deciding consistency and finding a solution for a TCSP is NP-complete. However, a TCSP in which all constraints specify a single interval is called a *Simple Temporal Problem (STP)*. The STPs form a tractable subclass of TCSP. An STP can be solved in polynomial time by using shortest path algorithms. Steps in solving an STP are checking consistency and finding a minimal graph. Consistency of an STP can be checked by using a distance graph (*d-graph*) $G_d = (V, E_d)$, which is a graph where the vertices represent the same set of TCSP variables and each edge $i \rightarrow j \in E_d$ is labeled by a distance a_{ij} .

An STP is *consistent* if its distance graph has no negative cycle where simply by examining the sign of the diagonal elements. Finding a minimal graph of a consistent STP can result in a solution of the STP.

A *Disjunctive Temporal Problem (DTP)* is an extended STP that allows disjunction of constraints. Every constraint $C_i \in C$ is a set of disjunctive constraints, $C_{i1} \vee \dots \vee C_{in}$, where each C_{ij} is defined as a duration constraint in STP. Moreover C_i can be an arbitrary constraint. For example (taken from [106]):

If you can, stop by the post office for 10-15 minutes, then take route A for 10-15 minutes, or else take route B for 10-15 minutes.

A constraint representing this scenario is

$$(10 \leq PO_{begin} - PO_{end} \leq 15 \wedge 10 \leq A_{begin} - B_{end} \leq 15 \wedge PO_{begin} = A_{end}) \vee B_{end} - A_{begin} \leq 15$$

where PO , A and B represent events of stopping at the post office, taking route A and taking route B , respectively. The subscript *begin* and *end* represent the starting and ending time points for each event, respectively.

As with STPs, a solution to a DTP is an assignment of all variables in which every constraint C_i is satisfied. C_i is satisfied when at least one C_{ij} is satisfied. In fact, a DTP is a set of STPs, also called *component STPs*. To decompose a DTP to STPs, each variable represents a constraint C_i . The domain of C_i is $\{C_{i1} \vee \dots \vee C_{in}\}$. This process is named as a *meta-CSP*. A solution to a DTP consists of a solution to any one of its component STPs in which all assigned C_i s are consistent. If a component STP is consistent, then the DTP is consistent.

3.1.2 Symbolic Temporal Models

Allen's Interval Algebra (IA) [4] and Vilain and Kautz's Point Algebra (PA) [113] are models for representing symbolic temporal problems where variables represent time points

and time intervals, respectively. In an IA network, thirteen primitive relations (Table 3.1) can hold between two time intervals. These relations, which are called the *Allen relations*, are commonly used in temporal constraint frameworks to represent symbolic temporal constraints [68, 109, 74]. In a PA network, three basic relations ($<$, $=$ and $>$) can hold between two time points. A subclass of IA networks that can be converted to PA networks is described by van Beek [109].

Interval Algebra (IA)

IA is a symbolic temporal constraint-based model for temporal reasoning [4]. Temporal objects are represented in term of time intervals. As mentioned, thirteen primitive relations can hold between intervals (Table 3.1).

Table 3.1: Allen Relations

Relation	Symbol (R)	Inverse (R^{-1})	Meaning $I R J$ or $J R^{-1} I$
equal	eq	eq	$a_i \xrightarrow{I} b_i$ $a_j \xrightarrow{J} b_j$
before	b	B	$a_i \xrightarrow{I} b_i$ $a_j \xrightarrow{J} b_j$
start	s	S	$a_i \xrightarrow{I} b_i$ $a_j \xrightarrow{J} \quad b_j$
finish	f	F	$a_i \xrightarrow{I} b_i$ $a_j \xrightarrow{J} \quad b_j$
overlap	o	O	$a_i \xrightarrow{I} b_i$ $a_j \xrightarrow{J} \quad b_j$
during	d	D	$a_i \xrightarrow{I} b_i$ $a_j \xrightarrow{J} \quad b_j$
meet	m	M	$a_i \xrightarrow{I} b_i$ $a_j \xrightarrow{J} \quad b_j$

Let $I = \{eq, b, B, m, M, o, O, d, D, s, S, f, F\}$ be the set of Allen relations. IA is an

algebra underlying set 2^I with three operators: *unary inverse*, *binary intersection* ($\oplus$) and *composition* ($\otimes$). The relation I means there is no constraint between two intervals.

Table 3.2: IA Composition

$\otimes$	b	B	d	di	o	O	m	M	s	S	f	F
b	b	I	Q d s	b	b	Q d s	b	Q d s	b	B	Q d s	b
B	I	B	N d f	B	N d f	B	N d f	B	N d f	B	B	B
d	b	B	d	I	Q d s	N d f	b	B	d	D o F	d	Q d s
D	Q D F	N D S	L	D	D o F	D O S	D o F	D O S	D o F	O	D O S	D
o	b	N D S	d o s	Q D F	Q	L	b	D O S	o	D o F	d o s	Q
O	Q D F	B	d O f	N D S	L	N	D O F	B	d O f	N	O	D O S
m	b	N D S	d o s	b	b	d o s	b	eq f F	m	m	d o s	b
M	Q D F	B	d O f	B	d O f	B	eq s S	B	d O f	B	M	M
s	b	B	d	Q D F	Q	d O f	b	M	s	eq s S	d	Q
S	M D F	B	d O f	D	D o F	O	D o F	M	eq s S	S	O	D
f	b	B	d	N D S	d o s	N	m	B	S N	N	f	eq f F
F	b	N D S	d o s	D	o	D O S	m	D O S	D	D	eq f F	F

$I = \{eq, b, B, m, M, o, O, s, S, f, F\}$
 $L = \{eq, d, D, o, O, s, S, f, F\}$
 $Q = \{b, o, m\}$
 $N = \{B, O, M\}$

An *IA constraint network* is a network of binary relations where variables are temporal events. The associated domain of each variable is the set of ordered pairs of real numbers $\{< a, b > | a, b \in R, a < b\}$, with a and b representing the start and end points of the interval, respectively. A binary relation, R_{xy} , between temporal events X and Y is a disjunction of Allen relations. For instance, $R_{xy} = X \{eq, b, B, o, f\} Y = X eq Y \text{ or } X b Y \text{ or } X B Y \text{ or } X o Y \text{ or } X f Y$.

Consider the following temporal scenario (taken from [29]):

Example 3.1.2.

Fred was reading the paper while eating his breakfast. He put the paper down and drank the last of his coffee. After breakfast he went for a walk.

The story can be formulated as an IA constraint network (Figure 3.2(a)). Path consistency (Figure 3.3) is enforced on an IA network by repeatedly applying the intersection and composition operations (Table 3.2) to every R_{ij} where $R'_{ij} \leftarrow R_{ij} \oplus ((R_{ik} \otimes R_{kj}))$.

Figure 3.2: AI Constraint Network [29]

FUNCTION Path-Consistency

Input: An IA network represented as a matrix C where entry C_{ij} is the label on edge (i, j)

Output: A path consistency approximation to the minimal labels for $C_{ij}, i, j = 1, \dots, n$

$Q \leftarrow \{(i, k, j) | 1 \leq i \leq n, 1 \leq k \leq n, k \neq i, k \neq j, i \neq j\}$

while Q is not empty

 select and delete a path (i, k, j) from Q

$t \leftarrow C_{ij} \oplus (C_{ik} \otimes C_{kj})$

 if $(t \neq C_{ij})$

$C_{ij} \leftarrow t$

$C_{ji} \leftarrow INVERSE(t)$

$Q \leftarrow Q \cup RELATED-PATHS(i, j)$

 end if

end while

end PC

FUNCTION RELATED-PATHS(i, j)

 return $\{(i, j, k), (k, i, j) | 1 \leq k \leq n, k \neq i, k \neq j\}$

end RELATED-PATHS

Figure 3.3: Path Consistency Algorithm [64]

Point Algebra (PA)

Vilain and Kautz [113] presented three basic relations ($<$, $=$, $>$) that can hold between two time points to represent symbolic temporal information. The relations between two points is allowed to be a disjunction of the basic relations. For brevity, disjunctions are shown with sets of relational operators. For example, point A is less than or equal to point B , can be represented by $A \{<, =\} B$. Let $\mathcal{R}$ be the set of all basic relations, $\{<, =, >\}$. Therefore the set of possible relations between two points is $\{0, <, \leq, =, >, \geq, \neq, ?\}$.

A *PA network* is a network of binary constraints where the variables represent time points, the domains of the variables are the set of real numbers, and the binary relations between variables are disjunctions of the basic point relations. In Figure 3.4, the PA network corresponding to the IA network shown in Figure 3.2.

Figure 3.4: A PA Constraint Network [29]

3.1.3 Hybrid Temporal Models

Formulating symbolic temporal information is different from formulating numeric temporal information. As a result, two separate frameworks are commonly used [4, 113, 68]. Not every symbolic temporal piece of information can be translated into numeric temporal information and vice versa. When converting in either direction, some temporal information is lost or difficult to

represent. For instance, in Example 3.1.1, if we add a piece of temporal information that “John and Fred met at a traffic light on their way to work”, the TCSP model cannot represent this information. In another direction, if we add that “It takes Fred 5-10 minutes to read the paper” to Example 3.1.2, the IA network cannot represent the temporal information.

As a result, hybrid temporal models have been defined to represent both types of temporal information. In Kautz and Ladkin [57], symbolic and numeric forms are handled in two different temporal constraint networks. The L logical language was designed to express both numeric and symbolic information. Language L_M represents a numeric temporal network based on the TCSP framework of Dechter et al. [31]. Language L_A represents a symbolic temporal network based on Interval Algebra (IA) of Allen [4]. L_M and L_A can be combined using translation algorithms.

Qualitative Algebra [68] and TemPro [74] are hybrid temporal frameworks that both numeric and symbolic temporal information are handled in unique frameworks. These frameworks are described in the remainder of this section.

Qualitative Algebra

Meiri [68] proposed Qualitative Algebra (QA), which is capable of handling both qualitative and quantitative information. IA [4], PA [113] and TCSP [31] are combined in QA. In QA, a temporal event (X_i) can be a time point or a time interval. Intervals correspond to time periods during which events occur or propositions are held, and points represent the beginning and ending points of some events, as well as reference points of time. The QA framework as described in Meiri [68] is recalled.

A qualitative constraint between two temporal events X_i and X_j is a disjunction of the form

$$(X_i r_1 X_j) \vee \dots \vee (X_i r_k X_j) \equiv X_i \{r_1, \dots, r_k\} X_j$$

where each of the r_i is a basic relation that may exist between two events. There are three types

of basic relations:

- *Basic Interval-Interval* (II) relations that can hold between a pair of intervals (as in IA, these are the thirteen Allens relations),
- *Basic Point-Point* (PP) relations that can hold between a pair of points, denoted by the set $\{<, =, >\}$ (as in PA) and
- *Basic Point-Interval* (PI) relations that can hold between a point and an interval, and *basic Interval-Point* (IP) relations that can hold between an interval and a point (shown in Table 3.3).

Table 3.3: The Basic Relations Between a Point, P , and an Interval, $I = [I^-, I^+]$ [68]

PI Relation	Graphical	Relation on Endpoint	PI Symbol	PI Symbol Inverse
P before I	$P \bullet$ $I^- \text{---} \bullet I^+$	$P < I^-$	b	B
P starts I	$P \bullet$ $I^- \text{---} \bullet I^+$	$P = I^-$	s	S
P during I	$P \bullet$ $I^- \text{---} \bullet I^+$	$I^- < P < I^+$	d	D
P finishes I	$P \bullet$ $I^- \text{---} \bullet I^+$	$P = I^+$	f	F
P after I	$P \bullet$ $I^- \text{---} \bullet I^+$	$P > I^+$	a	A

The possible qualitative constraints in QA are all legal constraints among the 2^{13} II relations, the 2^3 PP relations, the 2^5 PI relations and the 2^5 IP relations. Composition and intersection operations of two qualitative constraints, R_{ik} and R_{jl} are defined as follows:

- The composition of two qualitative constraints, $R_{ik} \otimes R_{jl}$ is defined by the transitivity tables shown in Tables 3.4 through 3.8.
- The intersection of two qualitative constraints, $R_{ij} \oplus R_{jl}$ is the set-theoretic intersection.

Table 3.4: A Full Transitivity Table [68]

	PP	PI	IP	II
PP	$[T_{PA}]$	$[T_1]$	$[\emptyset]$	$[\emptyset]$
PI	$[\emptyset]$	$[\emptyset]$	$[T_2]$	$[T_4]$
PP	$[T_1]$	$[T_3]^t$	$[\emptyset]$	$[\emptyset]$
PI	$[\emptyset]$	$[\emptyset]$	$[T_4]^t$	$[T_{IA}]$

Table 3.5: Composition of PP and PI Relations [68]

T_1	b	s	d	f	a
$<$	b	b	b s d	b s d	?
$=$	b	s	d	f	a
$>$	?	d f a	d f a	a	a

Table 3.6: Composition of PI and IP Relations [68]

T_2	A	F	D	S	B
b	$<$	$<$	$<$	$<$	?
s	$<$	$<$	$<$	$=$	$>$
d	$<$	$<$	?	$>$	$>$
f	$<$	$=$	$>$	$>$	$>$
a	?	$>$	$>$	$>$	$>$

Table 3.7: Composition of IP and PI Relations [68]

T_3	b	s	d	f	a
A	b	b	b m o d s	b m o d s	?
F	b	m	o s d	f F eq	a M O S D
D	b m o D F	o D F	o O eq s d f S D F	O D S	a M O S S
S	b m o D F	s S eq	O d f	M	a
B	?	a M O d f	a M O d f	a	a

Table 3.8: Composition of PI and II Relations [68]

T_4	b	a	d	D	o	O	m	M	s	S	f	F	eq
b	b	?	b s d	b	b	b s d	b	b s d	b	b	b s d	b	b
s	b	a	d	b	b	d	b	f	s	s	d	b	s
d	b	a	d	?	b s d	d f a	b	a	d	d f a	d	b s d	d
f	b	a	d	a	d	a	s	a	d	a	f	f	f
a	?	a	d f a	a	d f a	a	d f a	a	d f a	a	a	a	a

Table 3.9: Composition of PA [113]

$\otimes$	$=$	$<$	$\leq$	$>$	$\geq$	$\neq$
$=$	$=$	$<$	$\leq$	$>$	$\geq$	$\neq$
$<$	$<$	$<$	$<$	$?$	$?$	$?$
$\leq$	$\leq$	$<$	$\leq$	$?$	$?$	$?$
$>$	$>$	$?$	$?$	$>$	$>$	$?$
$\geq$	$\geq$	$?$	$?$	$>$	$\geq$	$?$
$\neq$	$\neq$	$?$	$?$	$?$	$?$	$?$

The T_{IA} and the T_{PA} transitive tables are defined in Tables 3.2 and 3.9, respectively. Symbol “?” refers to subsets that contain all basic relations for each qualitative constraint type.

A quantitative constraint in QA is either an absolute location or a distance between points.

There are two types of quantitative constraints:

- A unary constraint on a point P_i , restricts the location of P_i to the specified set of intervals.
- A binary constraint, C_{ij} between points P_i and P_j , constrains the permissible values for the distance $P_j - P_i$.

As in TCSP, P_o is introduced as a reference time point and it is used to assist in converting a unary constraint into a binary constraint. For two quantitative constraints $C_{ik} = \{I_{ik_1}, \dots, I_{ik_k}\} = \{[a_1, b_1], \dots, [a_k, b_k]\}$ and $C_{lj} = \{I_{lj_1}, \dots, I_{lj_l}\} = \{[c_1, d_1], \dots, [c_j, d_j]\}$, intersection and composition are defined as follows:

- The composition of two quantitative constraint $C_{ik} \otimes C_{lj} = z | \exists x \in I_{ik}, \exists y \in I_{lj}, x + y = z$.
- The intersection of two quantitative constraints $C_{ik} \oplus C_{lj} = x | x \in I_{ik}, x \in I_{lj}$.

The existence of a constraint of one type sometimes implies the existence of an implicit constraint of the other type.

- If a quantitative constraint, $C_{ij} = \{I_1, \dots, I_k\}$, between P_i and P_j exists, then the implied qualitative constraint, $QUAL(C)$ is defined as follows:

- If $0 \in \{I_1, \dots, I_k\}$, then “=” $\in QUAL(C)$.
- If a value $v > 0$ exists, where $v \in \{I_1, \dots, I_k\}$, then “<” $\in QUAL(C)$.
- If a value $v < 0$ exists, where $v \in \{I_1, \dots, I_k\}$, then “>” $\in QUAL(C)$.
- If a qualitative constraint, R_{ij} , between P_i and P_j exists, then the implied quantitative constraint, $QUAL(C)$ is defined as follows:
 - If “<” $\in R_{ij}$, then $\{0, \alpha\} \in QUAN(C)$.
 - If “=” $\in R_{ij}$, then $[0] \in QUAN(C)$.
 - If “>” $\in R_{ij}$, then $\{-\alpha, 0\} \in QUAN(C)$.

Table 3.10: QUAN Translation [68]

<	$(0, \alpha)$
$\leq$	$[0, \alpha)$
=	$[0]$
>	$(-\alpha, 0)$
$\geq$	$(-\alpha, 0]$
$\neq$	$(-\alpha, 0), (0, \alpha)$
?	$(-\alpha, \alpha)$

The intersection and composition operations are extended in cases where the constraints are of different types.

1. If C' is a quantitative constraint and C'' is a qualitative constraint, then an intersection is defined as a quantitative intersection: $C' \oplus C'' = C' \oplus QUAN(C'')$. Composition depends on the type of C'' .
2. If C'' is a *PP* relation, then composition is quantitative: $C' \otimes C'' = C' \otimes QUAN(C'')$ If C'' is a *PI* relation, then composition is qualitative. $C' \otimes C'' = QUAN(C') \otimes C''$.

Figure 3.5 shows a QA constraint network representing the following scenario (taken from [68]):

John and Fred are going to work. It takes John less than 20 minutes and Fred 15-20 minutes to get to work at a local office. If John has to go to the main office, it takes him at least 60 minutes. Today John left home between 7:05-7:10 a.m. and Fred arrived at work between 7:50-7:55 a.m. We also know that they met at a traffic light on their way to work.

Figure 3.5: An QA Constraint Network [68]

QA is a hybrid temporal framework, but it is very complicated and difficult to implement. In our view, extending QA to include conditional events, activity constraints and preferences requires a different treatment for each type of variables and constraints in the QA model.

TemPro

TemPro is an alternative hybrid temporal framework that can be used to express numeric and symbolic temporal information [76, 74]. It is based on IA and is a discrete model of time. TemPro is expressive and simple to employ. Temporal windows are used to manage numeric constraints over temporal objects. As in IA, symbolic temporal relations are formed by sets of disjunctives of Allen relations. Hence, the symbolic constraint between two temporal events ev_1 and ev_2 can take the following forms: $I_1 r_1 r_2 \dots r_n I_2$ where $ev_1 = OCCUR(p, I_1)$, $ev_2 = OCCUR(p, I_2)$ and r_i is one of the thirteen Allen primitives. In Mouhoub [74], the following definition and terms

are defined.

Definition 3.1.1. A *TemPro* based temporal constraint satisfaction consists of a set of variables $\{ev_1, \dots, ev_n\}$, each defined on a discrete domain D_i standing for the set of possible occurrences (SOPO) in which the corresponding event can take place, and a set of binary constraints, each representing a qualitative disjunctive relation between a pair of events and thus restricting the values that the events can simultaneously take.

- **Timeline:** A temporal reference T_r is the maximal set of discrete and adjacent temporal units U_i . Each unit represents the smallest discrete portion of time that is in the temporal reference.
- **Interval:** An interval I is a finite set of adjacent units U_k represented by a pair of units (U_i, U_j) where U_i and U_j are the beginning and the ending times of I , respectively.
- **Events:** An event $Evt = OCCUR(p, I)$ represents the fact that proposition p occurred in interval I .
- **SOPO:** A Set Of Possible Occurrences (SOPO) represents the set of time interval where a given event can occur (Figure 3.6).

Figure 3.6: SOPO of an Event

A numeric constraint can be represented by a SOPO:

$$SOPO = [beginntime, endtime, duration, step]$$

where *begin*time is the earliest possible begin time of the event and *end*time is the latest possible end time, *duration* is the duration of the event and *step* is defined as the distance between the starting time of two adjacent intervals within the SOPO.

Thus, if ev_i is an event numerically constrained by $SOPO [inf_i, sup_i, d_i, s_i]$, then the set of possible occurrences of e_i is defined as follows:

$$\begin{aligned}
I &= \{ occ_j | begin(occ_j) = inf_i + k \times s_i, \\
&\quad end(occ_j) = begin(occ_j) + d_i, \\
&\quad end(occ_j) \leq sup_i, \\
&\quad k \in [0, sup_i - inf_i - d_i] \cap N \}
\end{aligned}$$

The *begin* and *end* functions on interval yield the begin and the end points of a given interval, respectively. The structure of *SOPO* supports numeric $\rightarrow$ symbolic conversion (Table 3.11) in which the corresponding symbolic relations are used to reduce the size of the original problem. In this case, some inconsistent Allen relations are eliminated before performing path consistency.

Table 3.11: Numeric to Symbolic Conversion

#	Rule	Reduction
1	$SOPO_i.inf > SOPO_j.sup$	only <i>B</i> can hold
2	$SOPO_i.sup > SOPO_j.inf$	only <i>b</i> can hold
3	$SOPO_i.d \neq SOPO_j.d$	remove <i>eq</i>
4	$SOPO_i.d > SOPO_j.d$	remove <i>s, f, d</i>
5	$SOPO_i.d < SOPO_j.d$	remove <i>S, F, D</i>
6	$SOPO_i.d = SOPO_j.d$	remove <i>DdSsFf</i>
7	$SOPO_i.inf + SOPO_i.d > SOPO_j.sup - SOPO_j.d$	remove <i>m, b</i>
8	$SOPO_i.sup - SOPO_i.d < SOPO_j.inf + SOPO_j.d$	remove <i>M, B</i>
9	$SOPO_i.inf > SOPO_j.sup - SOPO_j.d$	remove <i>S, s, o, D</i>
10	$SOPO_i.inf + SOPO_i.d > SOPO_j.sup$	remove <i>F, f, d</i>
11	$SOPO_i.sup - SOPO_i.d < SOPO_j.inf$	remove <i>s</i>
12	$SOPO_i.sup < SOPO_j.inf + SOPO_j.d$	remove <i>F, f</i>

Figure 3.7 represents a TemPro network for Example 3.1.3 (taken from [74]).

Figure 3.7: A TemPro Network

Example 3.1.3.

John, Mike and Lisa work for a company in Calgary. It takes John 20 minutes, Mike 25 minutes and Lisa 30 minutes to get to work. Every day, John leaves home between 7:20 and 7:26. Mike arrives at work between 7:55 and 8 and Lisa arrives at work between 7:50 and 8. We also know that John and Mike meet on their way to work, that Mike arrives to work before Lisa and that Lisa and John go to work at the same time.

TemPro is adequately expressive for our needs. Durations are restricted to constants, which promotes efficiency. Therefore TemPro is adopted in our proposed CCCSP framework for conditional temporal constraint problems.

3.2 Dynamic Temporal CSPs

In this section, we focus on literature related to conditional temporal CSPs. Jónsson and Frank [56] first proposed a general framework using procedural constraints for solving dynamic CSPs. This framework has been extended to a new paradigm, called Constraint-Based Attribute and Interval Planning (CAIP), for representing and reasoning about plans [38]. CAIP and its implementation, the EUROPA system, enable the description of planning domains with time, resources, concurrent activities, disjunctive preconditions and conditional constraints. The set of constraints, variables and their possible values do not need to be enumerated beforehand, which gives a general definition of dynamic CSPs. The definition of dynamic CSPs in Jónsson and Frank [56] is also more general than the one in Dechter and Dechter [30] because in this latter work, domains of variables are predetermined.

Tsamardinos et al. [107] proposed the Conditional Temporal Problem (CTP) formalism for conditional planning under numeric temporal constraints. This model extends the well known qualitative temporal networks, STP [31] and DTP [106] by adding instantaneous events (called *observation nodes*) representing conditional constraints and attaching labels to all nodes to indicate the situation. We recall the definitions of a CTP and a CSTP [107].

Definition 3.2.1. *A Conditional Temporal Problem (CTP) is a tuple $\langle V, E, L, OV, O, P \rangle$, where*

- *P is a finite set of proposition $A, B, C, \dots$*
- *V is a set of nodes (interchangeably called variables, time-points, events $\{x, y, z, \dots\}$)*
- *E is a set of constraints between nodes in V*
- *$L : V \rightarrow P^*$ is a function attaching a label to each node*

- $OV \subseteq V$ is the set of observation nodes
- $O : P \rightarrow OV$ is a bijection associating a proposition with an observation node. $O(A)$ is the node that provides the truth-value for the proposition.

Definition 3.2.2. A *Conditional Simple Temporal Problem (CSTP)* is a CTP where the constraints in E are STP-like constraints.

Figure 3.8 (taken from Tsamardinos et al. [107]) is an example of a dynamic planning problem of the CTP model. In the problem, there are two choices of skiing areas: Snowbird and Park City. A skier wants to find a trip plan from Home to either Snowbird or Park City. To go to Snowbird, the skier must pass town B. To go to Park City, the skier must go through town C. Two temporal constraints are given:

- If the skier skis at Snowbird, he/she should arrive after 1 p.m., because Snowbird offers great discounts in the afternoons.
- If the skier skis at Park City instead, he/she should arrive at point C no later than 11 a.m., because traffic is very heavy afterward.

If the skier leave home at 12:00am, what will be a consistent plan? The part of the plan that involves traveling from *Home* to *Snowbird* of the skiing problem is formulated as a Conditional STP in Figure 3.9. The $go(x, y)_s$ action represents the starting point of the action of going from point x to point y . The $go(x, y)_e$ action represents the end point of the action of going from point x to point y . The $obs(road\ b\ s)$ observation describes the condition of the road from point x to point y when the skier arrives at point x . The *Start* point represent the starting time of the story at 12 a.m.

Figure 3.8: A Planning Example [107]

Figure 3.9: The Skiing Plan as a Conditional STP [107]

3.3 Temporal Constraints with Preferences

Integrating preference knowledge into a temporal constraint framework is a current research. In temporal constraint reasoning, quantitative preferences have been integrated into some existing temporal frameworks [4, 113, 31, 109]. Khatib et al. [59] introduced the Simple Temporal Problem with Preferences (STPP). In this latter model, the preference of an interval I is a function with co-domain A (the c-semiring). Preferences are integrated into the STP framework by extending STP with the standard semiring-based soft constraints. Fuzzy CSP is added to the STP framework to express the preferences. For each constraint, C_{ij} , there is a preference function $f : I \rightarrow A$, where $I = [l_{ij}, u_{ij}]$ and A is a set of preference values. The additive operation is defined as the *min* function and the multiplicative operation is defined as the *max* function. Solving STPPs is NP-Complete. However the literature shows that if the preference functions of an STPP is presented in terms of *semi-convex* functions, the STPP is traceable.

In a similar way, a Disjunctive Temporal Problem (DTP) can be extended with preferences using SAT techniques [99]. The TCSP of Dechter et al. [31] is integrated with the addition of a mechanism for specifying preferences, based on the soft constraint formalism. The optimal solutions to a TCSP with preferences are those solutions which have the best global preference values in the ordering of the values in the c-semiring.

In Badaloni and Giacomini [8], IA is extended by the fuzzy CSP framework, which allows expression of preference among solutions. The thirteen Allen relations are assigned with a preference degree, belonging to the interval $[0, 1]$ called IA^{fuz} . IA^{fuz} is closed under *Inverse*, *Conjunctive Combination* and *Composition*. IA^{fuz} is defined in the following set :

$$I = (r_{1[\alpha_1]}, r_{2[\alpha_2]} \dots, r_{13[\alpha_{13}]}) \text{ where } \alpha_i \in [0, 1], r_i \in R, i = 1, \dots, 13.$$

If α_i is 0, then r_i is an inconsistent relation.

Given: $R' = (r'_{1[\alpha'_1]}, r'_{2[\alpha'_2]}, \dots, r'_{13[\alpha'_{13}]})$, and $R'' = (r''_{1[\alpha''_1]}, r''_{2[\alpha''_2]}, \dots, r''_{13[\alpha''_{13}]})$

inverse $R^{-1} = (r'^{-1}_{1[\alpha_1]}, r'^{-1}_{2[\alpha_2]}, \dots, r'^{-1}_{13[\alpha_{13}]})$

conjunctive $R' \otimes R'' = R = (r_{1[\alpha_1]}, r_{2[\alpha_2]}, \dots, r_{13[\alpha_{13}]})$

where $\alpha_i = \min \{\alpha'_i, \alpha''_i\} \ i \in \{1, \dots, 13\}$

disjunctive $R' \oplus R'' = R = (r_{1[\alpha_1]}, r_{2[\alpha_2]}, \dots, r_{13[\alpha_{13}]})$

where $\alpha_i = \max \{\alpha'_i, \alpha''_i\} \ i \in \{1, \dots, 13\}$

composition $R' \circ R'' = R = (r_{1[\alpha_1]}, r_{2[\alpha_2]}, \dots, r_{13[\alpha_{13}]})$

where $\alpha_i = \min \{\alpha'_i, \alpha''_i\}, r_i = r'_i \circ r''_i$

Later, IA^{fuz} is integrated into the QA framework [7]. This framework is mainly for expressing fuzziness. Instead of applying the c-seriming as used by others, a non-normalized trapezoidal distribution is defined to handle fuzziness of numeric constraints (distance between two time points). If A and B are two time points, a non-normalized trapezoidal constraint of the distance between two time points is $A \{(a, b, c, d) [\alpha]\}$, where a, b, c and d are four constraining points of the trapezoid and α is a degree of consistency. Consider the following example (taken from [7]):

John leaves home between 7:05-7:10 a.m. and he usually arrives at work around 8:00 a.m., but sometimes he arrives later; before 9:00 a.m.

It can be represented by $J^- \{(5, 10, 60, 65) [1]\} J^+$ and $J^- \{(5, 10, 120, 120) [0.5]\} J^+$ (Figure 3.10).

Additional operations (i.e., inversion, conjunctive, composition and disjunctive) for non-normalized trapezoidal algebra are defined in IA^{fuz} [7].

Figure 3.10: Handling Fuzziness of Numeric Constraints Using Non-Normalized Trapezoidal Distribution [7]

3.4 Conclusions

In this chapter, we reviewed literature on temporal constraints which includes numeric temporal models (TCN, STP and DTP), symbolic temporal models (PA and IA), hybrid temporal models (QA and TemPro), conditional temporal planning models (CAIP and CTP) and temporal constraints with preferences (STPP, IA^{fuz}). In this thesis, we propose augmenting TemPro with our new CCCSP framework for conditional temporal constraints. The full details of the augmented TemPro framework are provided in Chapter 5.

4. MANAGING CONDITIONAL CONSTRAINTS AND COMPOSITE VARIABLES

In this chapter, we propose an alternative approach, CCCSP to handle conditional CSPs in which composite variables, disjunctive activity constraints, clustering and conjunctive conditional activity constraints are allowed. In Chapter 2, we have recalled the definition of activity constraint from Mittal and Falkenhainer [69]. The disjunctive activity constraints involve a set of non-active disjunctive variables. An activity constraint $Act_i \rightarrow \{V_t, \dots, V_z\}$ can activate a non-active variable from $\{V_t, \dots, V_z\}$. Clustering and conjunctive conditional activity constraints impose complex conditions in which the number of conditional variables is greater than one. We introduce a new type of variable, *Composite Variable* to handle disjunctive activity constraints and composite objects. We also define a simplified type of activity constraint which is adequate to capture all four types of activity constraints stated in Mittal and Falkenhainer [69] for configuration problems. Inference rules are applied to the four types to derive tighter activity constraints in one uniform framework. A preliminary version of this chapter has been published [79, 82]

Originally, composite variables represent a conceptual view of structured variables. Activity constraints are used to control a set of conditional variables. These two concepts can be represented in two separated frameworks when a problem clearly states in terms of conceptual structure objects and conditional objects. Combining composite variables and conditional variables in our CCCSP framework, clearly shows the expressiveness of representing complex

problems. Moreover, we found a complement relationship between composite, optional and conditional variables as follows:

1. Composite variables are a special case of conditional variables and can be controlled by activity constraints. Only one variable in a domain of composite variables can be activated if the composite variable is active. For example a composite variable $Y = \{X_1, X_2, X_3\}$, where X_1, X_2 and X_3 are simple variables. X_1, X_2 and X_3 are counted as conditional variables that depend on variable Y . Three activity constraints are implicitly derived from the composite variable Y , $(Y = X_1) \rightarrow X_1$, $(Y = X_2) \rightarrow X_2$ and $(Y = X_3) \rightarrow X_3$.
2. Simple variables in a domain of composite variables support expressing disjunctive activity constraints. To express disjunctive activity constraints such as $X_1 = a \xrightarrow{\text{require}} X_2 \text{ or } X_3$, we can apply a composite variable to express this condition, $X_1 = a \rightarrow Y$, where $Y = \{X_2, X_3\}$. If we look at this problem clearly, Y and Z are not grouped in a conceptual structure object. No composite variables are clearly defined in the activity constraint. Y and Z are just two simple variables (choices) with the condition. Moreover, Y and Z have no relation to each other. However, Y and Z are considered to be at the same level and only one of them can be presented in a solution. In this case, a composite variable can express the disjunctive activity constraint.
3. There is a special type of composite variable that can be used to apply for an optional object. An optional object here is a variable where there is no explicit constraint to control the object to be or not to be in a solution. The concept of an optional object is usually found in a configuration problem. For example, a web cam is an optional component when building a laptop computer. There is no direct conditional constraint to control a web cam in order to be in a set of building a laptop unless users make a request. Laptop

configurations are acceptable in both cases with and without web cam. In this case we can apply a composite variable to represent the web cam as it will be selected or not selected with the laptop. For example composite variable $Y = \{Webcam, Null\}$. *Null* is a special value to indicate that no variable is selected under composite variable Y .

We propose two resolution approaches: using complete methods and using incomplete methods. For complete methods, solving CCCSPs is based on BT search and constraint propagation. A variant of the Arc Consistency (AC) algorithm (AC-3 [64]) is implemented when dealing with the composite variables. Constraints are propagated to non-active variables further from active variables using advanced FC^+ and MAC^+ .

Applying incomplete methods to solve CCCSPs is a new approach. As we have seen in configuration problems, complete solutions are required. However in other domains (e.g., scheduling and planning) the processing time is a sensitive issue and incomplete solutions can be a good alternative. We focus our study on three major incomplete methods, MCRW, SDRW and TS using static, dynamic and incremental strategies.

4.1 CCCSP Framework

Definition 4.1.1. *A Conditional and Composite Constraint Satisfaction Problem (CCCSP) is a tuple $\langle X, D_X, Y, D_Y, IV, C, Act \rangle$, where*

- $X = \{x_1, \dots, x_n\}$ is a finite set of simple variables. Each x_i has either active status or non-active status.
- $D_X = \{D_{x_1}, \dots, D_{x_n}\}$ is the set of domains of the simple variables. Each domain D_{x_i} is the atomic values that x_i can take.

- $Y = \{y_1, \dots, y_m\}$ is the finite set of composite variables. For each y_i has either active status or non-active status.
- $D_Y = \{D_{y_1}, \dots, D_{y_m}\}$ is the set of domains of the composite variables. Each domain $D_{y_i} = \{x_r, \dots, x_t\} \subseteq X$ is the set of simple variables that the composite variable y_i can take. The domains of the composite variables may not overlap. $x_r, \dots, x_t$ have non-active status and only one $x \in D_{y_i}$ will change its status to active iff y_i is active and x is assigned to y_i .
- $IV = \{iv_1, \dots, iv_k\}$ is the set of initial variables. Initial variables can be simple variables or composite variables initially set to active status. $IV \subseteq X \cup Y$.
- $C = \{C_{x_i, x_j}, \dots, C_{x_y, x_z}\}$ is the set of binary compatibility constraints. A binary compatibility constraints of C_{x_m, x_n} is a set of pairs of allowed values from D_{x_m} and D_{x_n} .
- $Act = \{Act_1, \dots, Act_q\}$ is the set of activity constraints. Act_i is defined as:

$$\{Cond_1 \wedge \dots \wedge Cond_p\} \xrightarrow{\text{active}} \{V_v \wedge \dots \wedge V_u\},$$

where $Cond_i$ is $Variable = value$ or $Variable \neq value$. If all conditions $Cond_1, \dots, Cond_p$ are true, the activity constraint is fired and activates the variables, $V_v, \dots, V_u$.

A solution to a CCCSP is an assignment of all initial variables and newly active variables that satisfy all of the binary compatibility constraints on active variables.

4.1.1 Activity Constraints

Adding a new variable to a problem during the resolution process is controlled by a set of activity constraints. In Mittal and Falkenhainer [69], four types of activity constraints (Require, Always Require, Require Not, Always Require Not) for configuration problems can be handled through one form of our activity constraint.

For example the set of activity constraints

$$\left\{ V_1 = a \xrightarrow{Require} V_9 \text{ and } V_9 \xrightarrow{AlwaysRequire} V_{10} \right\}$$

can be rewritten into

$$V_1 = a \xrightarrow{Active} V_9 \wedge V_{10}$$

In the beginning of all target variables are already set to non-active, there is no need for “Require Not” activity type. For instance, the set of activity constraints

$$\left\{ V_1 = a_i \xrightarrow{RequireNot} V_9, V_8 = a_j \xrightarrow{Require} V_9 \right\},$$

can be rewritten as

$$\left\{ V_1 \neq a_i \wedge V_8 = a_j \xrightarrow{Active} V_9 \right\}$$

An activity constraint is satisfied either (1) when its conditions is true and its target variables activated in the solution; (2) when its conditions is false. Furthermore activity constraints can be applied to control composite variables.

4.1.2 Composite Variables

Composite variables are introduced and defined in CCCSP framework to basically handle a group of disjunctive variables where only one variable can be selected from the group. Composite variables are a special type of conditional variable. Here activity constraints are used to implicitly control participation of the selected variables. When a composite variable $Y_i = \{X_i, \dots, X_k\}$ is defined, the following activity constraints are generated into the CCCSP:

$$\begin{array}{ccc} Y_i = X_i & \xrightarrow{active} & X_i \\ \vdots & & \vdots \\ Y_i = X_k & \xrightarrow{active} & X_k \end{array}$$

If X_i is assigned to Y_i , the corresponding activity constraint is fired and then the simple variable X_i is activated and needs to be assigned a value from its domain. Explicitly we can also apply composite variables to represent disjunctive activity constraints.

Disjunctive Activity Constraints and Optional Variables

Handling disjunctive activity constraints is performed via composite variables and activity constraints. When a disjunctive activity constraint

$$X_i = a \xrightarrow{\text{active}} X_j \text{ or } \dots \text{ or } X_k,$$

$X_j, \dots, X_k$ are originally independent simple variables. then we will treat $X_j \text{ or } \dots \text{ or } X_k$ under a composite variable, Y_m in which

$$D_{y_m} = \{X_j, \dots, X_k\} \text{ and } X_j, \dots, X_k \text{ depend on } Y_m$$

Thus, the disjunctive activity constraint is replaced with

$$X_i = a \xrightarrow{\text{active}} Y_m$$

In a CCCSP, an optional variable is a special conditional variable where can be active or non-active in a solution regardless of conditions. An optional variable, X_i is encoded by a composite variable, Y_i and a special type of variables, *Null*. Thus the domain of composite variable $Y_i = X_i, \text{Null}$, and Y_i is initially active in the problem.

4.1.3 CCCSP Example

We show the use of the CCCSP framework through the following dress up game (Figure 4.1). In general, dress up games entertain kids by providing sets of clothes, accessories and shoes. Kids can use their free style *Mix'n Match*¹ imagination to create a complete outfit. To

¹Mix'n Match clothing is a dress up way to combine different parts of wardrobes such as tops, bottoms and accessories that do not come in a set.

assist kids to have an appropriate outfit and be in fashion, we can enhance the dress up games by adding tips and advices from fashion designers, for example:

- “*running shoes do not match with pants*”,
- “*pantsuit with business shirt make you look elegant*”,
- “*jacket with jeans is for a weekend look*”,
- “*set of skirt, jacket and shirt makes you look feminine*” and
- “*two piece matched suit without jewelry is an appropriate dress for a job interview*”.

Fashion designers are experts and their knowledge can be viewed as a set of compatible constraints, activity constraints, conditional variables and composite variables. Figure 4.1 shows a simple dress up game that can assist kids to enjoy certain rules to dress up. A possible mix-match dress up is to wear a T-shirt, shorts and running shoes with a baseball cap. Another possible clothes set is a dress, necklace (*J1*) and handbag (*HB3*) with pumps. In the example, we show the use of a CCCSP in formulating the interactive dress up game as follows:

- *APPAREL* is a composite variable; *SHOES*, *SET*, *TOP*, *BOTTOM*, *HAT*, *BELT*, *JEWELRY* and *HANDBAG* are simple variables.
- *APPAREL* and *SHOES* are initial active variables; the rest (*SET*, *TOP*, *BOTTOM*, *HAT*, *BELT*, *JEWELRY* and *HANDBAG*) are nonactive variables.
- A set of activity constraints:

$$\begin{aligned}
APPAREL &= TOP \xrightarrow{active} BOTTOM \\
APPAREL &= BOTTOM \xrightarrow{active} TOP \\
SET &= Dress \xrightarrow{active} JEWELRY, HANDBAG \\
TOP &= T-Shirt \xrightarrow{active} HAT \\
BOTTOM &= Pants \xrightarrow{active} BELT \\
BOTTOM &= Jeans \xrightarrow{active} BELT \\
BOTTOM &= Skirt \xrightarrow{active} HANDBAG
\end{aligned}$$

- Binary Constraints (see compatible tuples in Figure 4.2):
 - *TOP-SHOES*
 - *SET-SHOES*
 - *BOTTOM-SHOES*
 - *TOP-BELT*
 - *HAT-BOTTOM*
 - *JEWELRY-HANDBAG*
 - *SHOES-HANDBAG*

4.2 Solving CCCSPs

In our research, we proposed two CCCSP solving approaches: using complete methods and using incomplete methods. In this section we discuss and demonstrate the resolution processes.

Figure 4.1: A Dress Up Game Example

JEWELRY	HAND BAG	SET	SHOES	BOTTOM	SHOES	HANDBAG	SET
J1	HB2	Skirt Suit	Pump	Pants	Casual	HB2	Dress
J2	HB2	Skirt Suit	Boot	Pants	Pump	HB1	Skirt Suit
J3	HB3	Dress	Pump	Jeans	Running	HB4	Pant Suit
J3	HB4	Pant Suit	Pump	Skirt	Boot	HB3	Dress
J3	HB1	Pant Suit	Casual	Skirt	Pump	HANDBAG	SHOES
J4	HB1	HAT	BOTTOM	Capris	Sandal	HB1	Sandal
J4	HB2	Bucket	Skirt	Shorts	Sandal	HB2	Pump
J4	Hb3	Visor	Capris	Shorts	Running	HB3	Pump
TOP	BOTTOM	Visor	Shorts	TOP	BELT	HB4	Casual
Blouse	Pants	Baseball	Shorts	Jacket	B4	HB3	Boot
Blouse	Skirt	Baseball	Jeans	Tank	B3	HB4	Sandal
Shirt	Pants	Toyo	Skirt	Tank	B4		
Shirt	Skirt	TOP	SHOES	T-Shirt	B3		
T-Shirt	Jeans	Blouse	Pump	Blouse	B2		
T-Shirt	Capris	Shirt	Casual	Blouse	B3		
T-Shirt	Shorts	T-shirt	Sandal	Shirt	B2		
Tank	Shorts	T-shirt	Running	Shirt	B4		
Tank	Capris	Tank	sandal				
Jacket	Jeans						

Figure 4.2: Compatible Constraints for the Example in Figure 4.1

4.2.1 Complete Methods for CCCSPs

We will tackle the original CCCSPs directly rather than reformulating CCCSP to an equivalent CSP or first generating all possible CSPs and then later solving the CSPs. BT is a fundamental systematic technique in the complete methods. In CCCSPs enforcing arc consistency is a variant of the classical CSP because our model includes composite variables and non-active variables.

A Weaker Arc Consistency for CCCSPs

Arc Consistency (AC) is a constraint propagation technique which reduces search space by removing inconsistent values from domains [9]. The existence of an active composite variable in CCCSPs entails a weaker *AC*. The *AC* is distributed to every variable within a domain of a composite variable. The following rules are applied:

- **Case 1:** the constraint is (X_1, X_2) where X_1 and X_2 are simple variables

- The traditional arc consistency [64] is applied here. Each value a of X_1 should have a support in the domain of X_2 .
- **Case 2:** the constraint is (Y, X) where Y is a composite variable and X is a simple variable
 - Apply case 1 by viewing Y as a variable. Each value a , from the domain of a given variable $Y_i \in Y$, should have a support in the domain of X .
- **Case 3:** the constraint is (X, Y)
 - Apply case 1 by viewing Y as a variable. Each value a , from the domain of a given variable, should have a support in at least one domain of the variable within Y . Note that enforcing AC between X and Y can be skipped if at least one C_{X, Y_i} does not exist. In this case X 's domain remains the same. Every domain value of X has a support in Y_i .
- **Case 4:** the constraint is (Y_i, Y_j) where Y_i and Y_j are two composite variables
 - Apply case 2 between Y_i and each variable X_i within Y_j .

Using the above rules, we have implemented a new arc consistency algorithm for CCCSPs as shown in Figure 4.3. This algorithm is an extension of the well known AC-3 procedure [64]. Figure 4.4, 4.5 and 4.6 show applied AC3-CCCSP. X_i is a simple variable and $D_{X_i} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Y_1 and Y_2 are composite variables where $D_{Y_1} = \{X_2, X_3, X_4\}$ and $D_{Y_2} = \{X_5, X_6, X_7\}$. Lists of compatible tuples are labeled on directional arcs. After the $\text{REVISE_COMP}(X_1, Y_1)$ function is called, D_{X_1} is reduced to $\{1, 2, 3, 4, 5\}$ (Figure 4.5). Then $D_{X_2} = \{2, 3\}$, $D_{X_3} = \{6, 7\}$ and $D_{X_4} = \{1, 8\}$ after calling the $\text{REVISE_COMP}(Y_1, X_1)$ function

(Figure 4.6). Finally the $\text{REVISE_COMP}(Y_2, Y_1)$ and the $\text{REVISE_COMP}(Y_1, Y_2)$ functions are performed, consequently, D_{X_6} is reduced to $\{1, 5, 7\}$ (Figure 4.6).

Backtrack Search and Constraint Propagation

To solve CCCSPs in a systematic approach, we propose five new algorithms based on backtrack search and constraint propagation: (1)BTCCCSP, (2)FCCCCSP, (3)MACCCCCSP, (4)FC⁺CCCSP and (5)MAC⁺CCCSP

BTCCCSP is a dynamic variant of BT algorithm that can handle changeable numbers of assigned variables (active variables) during the resolution process. Figure 4.7 shows the BTCCCSP algorithm. Lists of assigned variables and active non-assigned variables are handled by two stacks, AV_{stack} and IV_{stack} respectively. At the initial state, all initial variables are stored in IV_{stack} , and AV_{stack} is empty. While IV_{stack} and AV_{stack} are not empty (note after the first variable is assigned), a non-assigned variable, $V_{current}$ is picked from the top of IV_{stack} . Function **SELECT-VALUE** returns a consistent value from domain $D_{v_{current}}$ if a_{x_j} is consistent to all the previous assignment otherwise return NULL. If $V_{current}$ finds a consistent value, $V_{current}$ is removed from IV_{stack} to AV_{stack} . Next all associated activity constraints with $V_{current}$ are checked. If they are satisfied, IV_{stack} is updated by adding the list of non-active variables resulted from the satisfied activity constraints. $V_{current}$ is assigned as the parent of the recently active variables. For all initial active variables, parent values are set to -1 . If backtracking occurs, all assigned and non-assigned active variables that share the same parent with the current fail variable are deactivated (removed from IV_{stack} and AV_{stack}).

Figure 4.8 shows the BTCCCSP search on the dress up game example (Figure 4.1). The algorithm starts by assigning Sandal to *SHOES* and then *SET* to *APPAREL*. Because *APPAREL* is a composite variables, then it activates variable *SET* in to the set of solution. *SET*

```

REVISE( $D_i, D_j$ )
  REVISE  $\leftarrow$  false
  For each value  $a \in D_i$  do
    if not compatible( $a, b$ ) for any value  $b \in D_j$  then
      remove  $a$  from  $D_i$ 
      REVISE  $\leftarrow$  true
    end if
  end for

REVISE_COMP ( $D_i, D_j$ )
  REVISE_COMP  $\leftarrow$  false
  if  $i$  is a simple variable and  $j$  is a composite variable
    if  $\exists (C_{i,k})$  does not exist in the problem, where event  $k \in D_j$ 
      no further revision necessary,  $X$ 's domain remains the same
    else
      ( $D_{tmp}$  saves all value in  $D_i$  that already found a support)
       $D_{tmp} \leftarrow \emptyset$ 
      For each event  $k \in D_j$  do
        ( $D$  contains all value in  $D_i$  that have not found a support yet)
         $D \leftarrow D_i - D_{tmp}$ 
        REVISE_COMP  $\leftarrow$  REVISE_COMP OR REVISE( $D, D_k$ )
         $D_{tmp} \leftarrow D_{tmp} \cup D$  (update  $D_{tmp}$ )
      end for
       $D_i \leftarrow D_{tmp}$ 
    end if
  end if
  if  $i$  is a composite variable and  $j$  is a simple variable
    For each event  $k \in D_i$  do
      REVISE_COMP  $\leftarrow$  REVISE_COMP OR REVISE( $D_k, D_j$ )
    end for
  end if
  if  $i$  and  $j$  are composite variables
    For each event  $k \in D_i$  do
      REVISE_COMP( $D_k, D_j$ )
    end for
  end if
end if

AC3-CCCSP
Given a graph  $G = (X, U)$ 
 $Q \leftarrow \{(i, j) | i, j \in U\}$ 
while  $Q \neq Nil$  do
   $Q \leftarrow Q - \{(i, j)\}$ 
  if  $i$  and  $j$  are variables
    if REVISE( $D_i, D_j$ ) then
       $Q \leftarrow \cup \{(k, i) | k, i \in U \text{ and } k \neq j\}$ 
    end if
  else
    if  $i$  or  $j$  is composite variable
      if REVISE_COMP( $D_i, D_j$ ) then
         $Q \leftarrow \cup \{(k, i) | k, i \in U \text{ and } k \neq j\}$ 
      end if
    end if
  end if
end while

```

Figure 4.3: AC3-CCCSP Algorithm

Figure 4.4: A CCCSP Example Before Performing AC3-CCCSP

Figure 4.5: Perform AC3-CCCSP to (a) C_{X_1, Y_1} and (b) C_{Y_1, X_1}

Figure 4.6: Perform AC3-CCCSP to (C) C_{Y_2, Y_1} and (b) C_{Y_1, Y_2}

cannot be assigned by any value from its domain. Backtracking is required; therefore, variable *SET* is deactivated. *2-Pieces* is assigned to *APPAREL* then *2-Pieces*, *TOP* and *BOTTOM* are active. In this example, the *2-Pieces* variable has only one value, *top-bottom* in its domain. Value *top-bottom* is assigned to *2-Pieces* (this step is omitted from Figures 4.8, 4.12 and 4.13).

FCCCCSP and **MACCCCCSP** take advantage of constraint propagation to dynamically reduce the search space. Here, AC3-CCCSP is applied to all active variables in different degrees including in the preprocessing stage. All methods start with a CCCSP containing a list of initially active variables and composite variables. The AC3-CCCSP algorithm is applied to the initial variables and composite variables in order to reduce some inconsistent values which will reduce the size of the search space. If the variables are not consistent (in the case of an empty domain) then the method will stop. The CCCSP is inconsistent in this case. While FCCCCSP

```

FUNCTION BTCCCSP
Input: A CCCSP  $(X, D_X, Y, D_Y, IV, C, Act)$ 
Output: Either solution, or notification that CCCSP is inconsistent
   $AV_{Stack} \leftarrow \emptyset$  /*initialize assigned variable stack*/
   $IV_{Stack} \leftarrow IV$  /*initialize initial active variable stack*/
  do
     $V_{current} \leftarrow Top(IV_{Stack})$  /*pick a variable*/
     $Assign_{V_{current}} \leftarrow SELECT - VALUE$  /*assign a consistent value to the current variable*/
    if  $Assign_{V_{current}}$  is null /*no value is returned*/
      if  $Parent(V_{current}) == -1$  /*initial active variables*/
         $pop(AV_{Stack}, V_{current})$  /*back track to the previous variable*/
         $push(IV_{Stack}, V_{current})$ 
      else /*activated by another previous assigned variable*/
        while  $(PARRENT(Top(IV_{Stack})) == PARRENT(V_{current}))$ 
          /*deactivate all future variables that share the same parent*/
           $pop(IV_{Stack})$ 
        end while
        do
          /*deactive all previous variables that share the same parent*/
           $pop(AV_{Stack}, V_{temp})$ 
        until  $(V_{temp} \neq PARRENT(V_{current}))$ 
         $push(IV_{Stack}, PARRENT(V_{current}))$ 
      end if
    else
       $pop(IV_{Stack})$ 
       $push(AV_{Stack}, V_{current})$ 
      check if  $A_i^{active} V_m, \dots, V_p$  is true
      /*check activity constraints that relate to  $V_{current}$ */
       $push(IV_{Stack}, non - active \{V_m, \dots, V_p\})$ 
      set  $PARRENT(non - active \{V_m, \dots, V_p\}) = V_{current}$ 
    end if
  while  $(IV_{Stack} \neq \emptyset) \wedge (AV_{Stack} \neq \emptyset)$ 

  if  $AV_{Stack} == \emptyset$ 
    return "inconsistent"
  else
    return instantiated values of all variables in  $AV_{Stack}$ 
  end if
END BTCCCSP

FUNCTION SELECT-VALUE
Input: A Variable  $X_j$ 
Output: A consistent value with all the past assigned variables
  or -1 if cannot find a consistent value
  select next available value  $a_j$  from  $D_{x_j}$ 
  if no next available value left in  $D_{x_j}$ 
    reset pointer of  $D_{x_j}$ 
    return -1
  else
    check  $a_j$  to  $C_{x_i, x_j}$ , where  $x_j$  is all the previous assigned variables
    if  $a_j$  consistent to all  $x_j$ 
      return  $a_j$ 
    end if
    goto select next available value
  end if
END FUNCTION SELECT-VALUE

FUNCTION PARENT
Input: A Variable
Output: A parent variable which activates the input variable
  or -1 if the input variable is initially active
END FUNCTION PARENT

```

Figure 4.7: BT-CCCSP Algorithm

Figure 4.8: Steps of BTCCCSP for Dress Up Game Example Figure 4.1

only propagates constraints of the current variable with the next future variables, MACCCCSP does this for all future variables. Furthermore in FC^+CCCSP and MAC^+CCCSP , we perform more constraint propagation on the set of non-active variables which may become active in the future.

Figure 4.9 and Figure 4.10 show additional constraint propagation functions in FCCCCSP and MACCCCSP algorithms. Further checking in each selection of a consistent value is required. In addition, the value must have a support in non-assigned future variables (active). After checking a value that satisfies the past variables (performed by function `SELECT-VALUE`), function `FC-PROPAGATE-CONSTRAINS` (or function `MAC-PROPAGATE-CONSTRAINT`) each iteration is called on to accomplish two tasks:

1. Produce a list of all related constraints between the current assigned variable and non-assigned future variables (in the case of MACCCCSP, it also includes all constraints between those all non-assigned variables).
2. Propagate the constraints onto the domains of the non-assigned future variables. If a domain becomes empty, select the next available value. If there is no value left, then backtrack to the previous assigned variable.

Another part of BTCCCSP that has been modified is when a backtrack occurs. The domains of all future variables are restored back to the previous configuration before the last propagation was performed.

More Advanced FC^+ and MAC^+

Propagating constraints to the set of active variables prunes inconsistent branches from all currently active variables. However trashing and redundant checking could occur when activating a new variable. For instance $IV = \{V_1, \dots, V_{20}\}$ is a set of active variables. V_1 to V_{12} are assigned

```

FUNCTION FC-PROPAGATE-CONSTRAINT
  repeat
    choose next value  $a$  in  $D_{V_{current}}$ 
     $C_{list} \leftarrow C_{(V_{next}, V_{current})}$  where  $V_{next} \in IV_{Stack}$ 
    if  $AC3 - CCCSP(CCCSP, C_{list})$  is consistent
      return  $a$ 
    else
      restore domains of all next future variables
    end if
  until all values in  $D_{V_{current}}$  are selected
  return null
END FUNCTION FC-PROPAGATE-CONSTRAINT

```

Figure 4.9: FC-PROPAGATE-CONSTRAINT Algorithm

```

FUNCTION MAC-PROPAGATE-CONSTRAINT
  repeat
    choose next value  $a$  in  $D_{V_{current}}$ 
     $C_{list} \leftarrow C_{(V_{next}, V_{current})}$  where  $V_{next} \in IV_{Stack}$ 
     $C_{list} \leftarrow C_{list} \cup C_{(V_i, V_j)}$  where  $V_i, V_j \in IV_{Stack}$ 
    if  $AC3 - CCCSP(CCCSP, C_{list})$  is consistent
      return  $a$ 
    else
      restore domains of all next future variables
    end if
  until all values in  $D_{V_{current}}$  are selected
  return null
END FUNCTION MAC-PROPAGATE-CONSTRAINT

```

Figure 4.10: MAC-PROPAGATE-CONSTRAINT Algorithm

variables. When assigning V_{13} , it always activates a new variable V_{21} which is inconsistent with the assignment of V_5 . Until V_{13} is assigned, this inconsistency cannot be detected because V_{21} is non-active.

To detect this failure earlier, we extend the propagation to the non-active variables but not to all of them. Only the possible future active variables are considered. A possible future active variable is evaluated from a non-assigned active variable and its associated activity constraints with its remaining domain. For example when assigning V_4 , V_{13} has only one value, a , left in the domain and this activity constraint $V_{13} = a \rightarrow V_{21}$ is applied. Thus, V_{21} is a possible future active variable. The advanced constraint propagation on V_{21} is performed. In the future, when V_{21} is set to active, it is surely consistent with the assigned variables and non-assigned active variables. Similarly, this feature is added to the FC^+CCCSP and the MAC^+CCCSP algorithms. In contrast, FC^+CCCSP only checks the consistency of non-active variables related to the current variables but not all of them as in the MAC^+CCCSP .

```

FUNCTION FUTURE-ACTIVE-VARIABLES
  INPUT List of activity constraint  $Act$ 
  OUTPUT List of non-active variables,  $NAV$ , that possibly will be active in the future
   $NAV \leftarrow \emptyset$ 
  for each  $Act_i : Cond_1 \wedge \dots \wedge Cond_m \xrightarrow{active} X_v \wedge \dots \wedge X_y \in Act$ 
     $Flag \leftarrow true$ 
    for each  $Cond_j : X_j = a_j$ 
      if  $X_j \in IV_{stack}$  and  $a_j \notin D_{x_j}$ 
         $Flag \leftarrow false$ 
        exit for
      end if
    end for
    if  $Flag$  is true
       $NAV = NAV \cup \{X_v \wedge \dots \wedge X_y\}$ 
    end if
  end for
END FUNCTION FUTURE-ACTIVE-VARIABLES

```

Figure 4.11: FUTURE-ACTIVE-VARIABLES Algorithm

Figure 4.11 shows a function for generating the list of future active variables (non-active).

This function is added to the FC-PROPAGATE-CONSTRAINTS (Figure 4.9) and the MAC-PROPAGATE-CONSTRAINTS (Figure 4.10) functions in order to alter the list of constraints C_{list} at the initial state before calling the AC3-CCCSP function. For FC⁺CCCSP, if there is a constraint between the current variable and a variable from NAV , $C_{v_{current}, NAV}$ is added to C_{list} . For MAC⁺CCCSP, additional constraints between a pair of variables from the IV_{stack} and NAV are added to C_{list} .

Figure 4.12 shows the propagate constraints of the FCCCCSP algorithm on the dress up game example. *APPAREL*'s domain remains only *2-Pieces* value when, *Sandal* is assigned to *SHOES*. After assigning *Capris* to *BOTTOM*, only *T-Shirt* and *Tank* are left in the domain of *TOP*. *HAT*'s status changes to active after assigning *T-Shirt* to *TOP*.

Figure 4.13 shows propagate constraints of the MAC⁺CCCSP algorithm on the dress up game example. The MAC⁺CCCSP algorithm does more constraint propagation than the FCC-CCSP, the FC⁺CCCSP and the MACSCCCSP algorithms. After assigning *Sandal* to *SHOES*, *APPAREL*'s domain remains only *2-Pieces* value. *BOTTOM*'s domain has only *Capris* and *Shorts*. *TOP* has only *T-Shirt* in its domain. *HAT* is the possible active variable. Therefore, *HAT*'s domain remains only value *Visor*.

Variable and Value Ordering

In the preprocessing stage, we order the variables and values of the domains. Following the “First Fail Principle,” the most constrained variables are the first to be processed. Each variable is evaluated by the degree of constraints as follows:

1. If it is a simple variable, the degree of the variable is the summation of:
 - the number of constraints involving variables
 - the degree of future active variables that the variables possibly will be activated

Figure 4.12: FCCCCSP for the Dress Up Game Example Figure 4.1

Figure 4.13: MAC+CCCSP for the Dress Up Game Example Figure 4.1

(becomes a parent) in the future. If there is more than one variable, we pick the maximum degree of the future active variables

2. If it is a composite variable, the degree of the composite variable is the minimum degree of its variable domains.

In Figure 4.6, the degrees of $X_1 = 3, X_2 = 3, X_3 = 4, X_4 = 3, X_5 = 2, X_6 = 3, X_7 = 2$. So the degree of composite variables $Y_1 = 3$ and $Y_2 = 2$. The variable ordering is Y_2, Y_1 and X_1 (or Y_2, X_1 and Y_1). We order domains of composite variables by the increasing degree of ordering. Therefore, the domain of Y_1 is in the X_2, X_4 and X_3 ordering and the domain of Y_2 is in the X_5, X_7 and X_6 ordering. If we add an activity constraint $Y_1 = X_2 \xrightarrow{\text{Active}} X_8$ where X_8 has constraints with X_2 and X_7 , then the degree of X_2 is changed to 4. We obtain the domain of Y_1 in X_4, X_2 and X_3 ordering. The degree of X_7 is changed to 3. We obtain the domain of Y_2 in X_5, X_6 and X_7 ordering.

In Example 4.2, the degrees of variables *JEWELRY*, *HANDBAG*, *BELT* and *HAT* are 1. The degree of *TOP* is 4. It is the summation of three (from its three associated binary constraints) and one (from activate *HAT* in the future). The degree of *BOTTOM* is 4. It is calculated from its three associated binary constraints and the minimum degree between *JEWELRY* and *HANDBAG* variables that will be active in the future. The degree of *SHOES* is 4. The degree of *SET* is 3. The degree of *2-PIECES* is 8 since it will active both *BOTTOM* and *TOP* variables. Therefore, the degree of *APPAREL* is 3 because the degree of *SET* holds the minimum degree. Only *SHOES* and *APPAREL* are active in the beginning so the order of variables at the initial search is to assign *SHOES* first and then *APPAREL*. The domain of *APPAREL* is reordered to *SET* and *2-PIECES*.

4.2.2 Incomplete Methods for CCCSPs

Incomplete methods are alternative resolutions for some real time applications where a trade off between solution quality and speed is admissible. In addition, incomplete solutions are useful and acceptable in some applications (e.g., game softwares and self training/learning softwares). There is another interesting way to play the dress up game. Instead of using a systematic way to dress up by choosing one item after another item, it would be more challenging for kids to figure out “*what is wrong with the given outfit?*”, and try to rearrange the outfit to make it look better. A better matched outfit can be rewarded by a higher score. This will help kids learn how to fix a problem when an incomplete solution is provided. In the same direction, incomplete methods can be useful in self learning software designs.

To handle the dynamic aspect of CCCSPs, we propose three strategies: (1) *static*, (2) *dynamic* and (3) *incremental* and applied to SLS (i.e., MCRW, SDRW and TS). Basically, MCRW, SDRW and TS start from a complete assignment of variables and iterate by improving the quality of the assignment (number of solved constraints) at each step (repairing stage) until a complete solution is found or the maximum number of iterations is reached. In the repairing stage, the number of variables involving in a solution may change because some activity constraints are satisfied or failed. The three strategies handle changing of variables (CSP configuration) in different ways. No adding and removing variables are allowed in the static strategy while the changes are permitted in the dynamic strategy. The incremental strategy is a combination of the static and dynamic strategy.

Static Strategy

In the static strategy, we avoid changing the CSP configuration during the repairing step by randomly picking a CSP configuration. The selected CSP configuration is generated by assigning

all initial composite variables and activating some nonactive variables. The set of non-active variables is divided into two groups:

1. must be active
2. never be active

Activating *must be active* nonactive variables is performed by satisfying their associated activity constraints. For example in an activity constraint $V_1 = 3 \xrightarrow{\text{active}} V_5$, where $D_{V_1} = \{1, \dots, 10\}$, we make V_5 active by reducing $D_{V_1} = \{3\}$.

To ensure that the set of *never be active* of nonactive variables will not be added into the CSP during the repairing loop, we remove all values that satisfy their associate activity constraints. For instance, V_6 is a non-active variable and is not selected to be in the CSP configuration. Its associated activity constraint is $V_7 = 2 \xrightarrow{\text{active}} V_6$. So 2 will be removed from D_{V_7} . Figure 4.14 shows how we apply the static strategy to MCRW. The static strategy can also be applied to SDRW and TS. The neighborhood configurations of SDRW and TS involve all violent variables. SDRW is similar to MCRW in that it uses noise as probability p to escape local optimum. In contrast TS saves a list of current moves, to protect the circle assignment path.

Figure 4.14 shows the MCRW-CCCSP_{Static} algorithm. All initial active variables are assigned to the set of variable solution, S . R non-active variables are enabled to be active by enforcing their associated activity constraints to be fired. The domains of conditional variables in the associated activity constraints remain only the values that will always enable the activity constraints. The rest of the non-active variables, l where are not in R , will be set to never be active. Their associated domains are removed the values that will fire the activity constraints. The set of active composite variables in S is first randomly assigned a value (which is a simple variable). The selected simple variable is activated into S . Then a full random assignment of all simple variables in S is performed. The repairing state starts from the initial assignment of

a set of simple variables in S . Variable Max_{moves} sets the maximum time that the algorithm will repair the solution, S . In the repairing loop, if the $Eval$ function indicates an inconsistent solution of S (number of violated constraints > 0) and the number of iterations does not reach the maximum setting, a conflict variable, V is randomly selected and reassigned by the best value which minimizes the number of conflicts for V . Probability p is used here as a noise to avoid trapping in a local optimal. If p is verified, the algorithm will randomly pick a conflict variable and also randomly reassign a new value to the variable. The algorithm returns the final assignment of S when a consistent solution is found or the number of moves has reached the maximum setting.

In addition, to varying the numbers of randomly generated CSP configurations from a CCCSP, we call the $MCRW-CCCSP_{Static}$ function in a loop of main function $MAIN-MCRW-CCCSP_{Static}$ (Figure 4.15).

Figure 4.16 shows applying $MCRW-CCCSP_{Static}$ to solve the dress up game (Figure 4.1). Variable HAT is randomly selected to be active in the final solution. Therefore TOP 's domain contains only *T-Shirt* value. $APPAREL$'s domain remains only *2-Piece* value. $BOTTOM$'s domain remains only values *Capris* and *Shorts*. Then HAT becomes active in a solution. In the first randomly initial assignment there are 2 violated constraints, $C_{BOTTOM-SHOES}$ and $C_{HAT-BOTTOM}$. $SHOES$ is selected as a conflict variable in the second repairing, and it is repaired with a new value *Sandal*. The number of violated constraints is reduced to 1. In the final repairing loop, HAT is selected and changed its value to *Visor* and the algorithm returns the consistent solution.

```

FUNCTION MCRW-CCCSPStatic(Maxmoves, p)
  S ← IV /* all initial variable assigned to solution S */
  randomly choose R nonactive variables
  for each Vr ∈ R
    randomly choose its associated activity constraints,
      Vi = ai  $\xrightarrow{\text{active}}$  Vr
      DVi ← ai /*enforce the activity constraint to be true*/
      S ← S ∪ Vr /*add the new variable to the current solution*/
  end for

  /* disable the rest of non-active variable */
  for each nonactive variable l ∉ R
    for each associated activity constraints Vj = ak  $\xrightarrow{\text{active}}$  Vl
      DVj = DVj - ak
    end for
  end for

  for each composite variable Vc ∈ S
    randomly select a value xa ∈ DVc and assign to Vc
    DVc ← xa
    S ← S ∪ xa /*add the new variable to the current solution*/
  end for

  randomly initial assign all variable in S
  numbermoves ← 0
  while Eval(S) > 0 and numbermoves < Maxmoves do
    if probability p verified then
      randomly choose a variable V in conflict
      randomly choose a value v̂ for V
      end repeat
    else
      choose randomly a variable V in conflict
      choose a value v̂ that minimizes the number of conflicts for V
      end repeat
    end if
    if v̂ ≠ current value of V then
      assign v̂ to V
      numbermoves ← numbermoves + 1
    end if
  end while (Maxmoves)
  return s
end MCRW-CCCSPStatic

FUNCTION Eval
  INPUT: a list of assigned variables S
  OUTPUT: the number of violated constraints
  violate ← 0
  for each Ci,j
    if Ci,j is not satisfied
      violate ← violate + 1
    end if
  end for
  return violate
end Eval

```

Figure 4.14: MCRW-CCCSP_{Static} Algorithm

```

FUNCTION MAIN-MCRW-CCCSPStatic(Maxconfiguration)
  numberconfiguration ← 0
  while(numberconfiguration < Maxconfiguration)
    Si ← MCRW - CCCSPStatic
    if Si is a consistent solution
      return Si
    end if
    if Eval(Si) < Eval(Si1)
      Sbest ← Si
    else
      Sbest ← Si-1
    end if
    numberconfiguration ← numberconfiguration + 1
  end while
  return Sbest
END FUNCTION MAIN-MCRW-CCCSPStatic

```

Figure 4.15: MAIN-MCRW-CCCSP_{Static} Algorithm

Dynamic Strategy

As opposed to the static strategy, the dynamic strategy allows for changing the CSP configuration during the repairing stage. Because of the dynamic aspect of CCCSPs, the configuration of CCCSPs can be changed in each repairing loop. If an activity constraint is satisfied, a new variable is added into the solution. During the repairing stage of the MCRW and the SDRW algorithms, we analyze the changes in problem configuration. At the initial a new complete assignment is randomly generated. The random configuration may be modified later in the repairing iteration loop. Figure 4.17 shows the MCRW-CCCSP_{Dynamic} algorithm. At the initial stage, all active variables are randomly assigned values. Activity constraints are then checked and new active variables may be added with randomly assigned values. Some active variables may be removed from the solution (non-active). Checking activity constraints is repeated until no new variables are added to the assignment. Following this, the complete assignment is passed on to the repairing loop to improve the quality with probability $1 - p$, where p is a probability for random walk (noise). Every time a selected variable changes a value, checking activity constraints is performed.

ONLY 5 Variables will be active in the solution: APPAREL, TOP, BOTTOM, SHOES, HAT
Repairing Loops:

- [1] APPAREL= 2-Pieces, TOP=T-Shirt, BOTTOM= Short, SHOES= Pump, HAT=Toyo
#Violated Constraints=2
- [2] APPAREL= 2-Pieces, TOP=T-Shirt, BOTTOM= Short, SHOES= Sandal, HAT=Toyo
#Violated Constraints=1
- [3] APPAREL= 2-Pieces, TOP=T-Shirt, BOTTOM= Short, SHOES= Sandal, HAT=Visor
#Violated Constraints=0

Figure 4.16: MCRW-CCCSP_{Static} for the Dress Up Game Example (Figure 4.1)

```

FUNCTION MCRW-CCCSPDynamic(Maxmoves,  $p$ )
   $S \leftarrow$  random valuation of initial active variables
  repeat until no new variables added
    check activity constraints
    add new variables to  $S$ 
    randomly assign the newly added variables
  end repeat
   $number_{moves} \leftarrow 0$ 
  while  $Eval(S) > 0$  and  $number_{moves} < Maxmoves$  do
    if probability  $p$  verified then
      randomly choose a variable  $V$  in conflict
      randomly choose a value  $\hat{v}$  for  $V$ 
      repeat until no new variables added
        check activity constraints
        add/remove new variables to/from  $S$ 
        randomly assign the newly added variables
      end repeat
    else
      randomly choose a variable  $V$  in conflict
      choose a value  $\hat{v}$  that minimizes the number of conflicts for  $V$ 
      repeat until no new variables added
        check activity constraints
        add/remove new variables to/from  $S$ 
        choose the best values for the new active variables
      end repeat
    end if
    if  $\hat{v} \neq$  current value of  $V$  then
      assign  $\hat{v}$  to  $V$ 
       $number_{moves} \leftarrow number_{moves} + 1$ 
    end if
  end while (Maxmoves)
  return  $s$ 
end MCRW-CCCSPDynamic

```

Figure 4.17: MCRW-CCCSP_{Dynamic} Algorithm

Figure 4.18 shows the searching solution of the MCRW-CCCSP_{Dynamic} algorithm. The numbers of variables involved in the solution of each repairing loop, can be dynamically changed. In the first repairing loop *JEWELRY* and *HANDBAG* are added into the solution because *SET=Dress*. Later in the third repairing loop they are removed because the associated activity condition no longer holds when *APPAREL=2-Piece*. *HAT* is added into the solution at the fifth repairing loop. Finally a consistent solution is found at the eighth repairing loop.

Incremental Strategy

To avoid unnecessary configuration changes during the repairing loop as in the dynamic strategy, the repairing loop is divided into two repairing steps. In the first repairing loop, the number of variables in the set of initial active variables is controlled. A consistent assignment for the initial active variables should be found in the first repairing loop. Later activity constraints are considered. All new active variables are repaired and improved in the second repairing loop. In the first repairing loop of MCRW-CCCSP_{Incremental}, the whole set of initial variables (both simple and composite) in the loop is considered (Figure 4.19).

Figure 4.20 shows applying MCRW-CCCSP_{Incremental} to the dress up game. In the first repairing loop only three variables were involved *APPERAL*, *SHOES* and *SET*. After the algorithm found a consistent assignment for the three variables (*APPAREL=SET*, *SET=Dress*, *SHOES=Pump*), associated activity constraints were checked. Two new variables, *JEWELRY* and *HANDBAG* were set as a new set of variables to the new repairing call. The second repairing call found a consistent assignment (*JEWELRY=J4*, *HANDBAG=HB2*). The final solution returned the union of the two consistent assignments set, (*APPAREL=SET*, *SET=Dress*, *SHOES=Pump*, *JEWELRY=J4*, *HANDBAG=HB2*).

For the static strategy we apply the MCRW and SDRW and TS search. For the dynamic

```

1) Randomly assign 2 initial active variables: APPAREL and SHOES
   APPAREL=SET, SHOES=Running
2) Check Activity Constraints:
   Active SET
   Randomly assign a value to SET
   SET=Plant Suit
3) Repairing Loops: #Violated Constraints=1
   [1] Randomly select SET as a conflict variable and assign a new value Dress to SET
       APPAREL= SET, SHOES=Running, SET=Dress
       #Violated Constraints=1
       SET=Dress activates JEWELRY and HANDBAG
       JEWELRY=J3
       HANDBAG=HB4
       #Violated Constraints = 1+1
   [2] Randomly select HANDBAG as a conflict variable
       HANDBAG= HB1
       #Violated Constraints = 2
   [3] Probability p verified (Noise)
       APPAREL is randomly selected and assigned a new value
       APPAREL = TOP
       Deactivate: SET, JEWELRY and HANDBAG
       Activate: BOTTOM and TOP and randomly assigned new value
       BOTTOM=Skirt
       TOP=Tank
       #Violated Constraints = 3
   [4] Randomly select SHOES as a conflict variable
       SHOES=Sandal
       #Violated constraints = 2
   [5] Randomly select TOP as a conflict variable
       TOP=T-shirt activate HAT
       HAT=Toyo
       #Violated constraints = 2
   [6] Randomly select BOTTOM as a conflict variable
       BOTTOM=Jean
       #Violated constraints = 2
   [7] Randomly select HAT as a conflict variable
       HAT=Baseball
       #Violated constraints = 1
   [8] Randomly select SHOES as a conflict variable
       SHOES=Running
       #Violated constraints = 0

Solution: APPAREL=2-Pieces, TOP=T-shirt, BOTTOM=Jean, SHOES= Running, HAT=Baseball

```

Figure 4.18: MCRW-CCCSP_{Dynamic} for the Dress Up Game Example (Figure 4.1)

```

FUNCTION MCRW-CCCSPIncremental( $S, \hat{S}, Max_{moves}, p$ )
   $\hat{S}$  is empty for the first call
  Randomly assign all variables in  $S$ 
  Add only new variables caused by initial active composite variables to  $S$ 
   $number_{moves} \leftarrow 0$ 
  while  $Eval(\hat{S} + S) > 0$  and  $number_{moves} < Max_{moves}$  do
    if probability  $p$  verified then
      randomly choose a variable  $V$  in conflict in  $S$ 
      randomly choose a value  $\hat{v}$  for  $V$ 
    else
      randomly choose a variable  $V$  in conflict in  $S$ 
      choose a value  $\hat{v}$  that minimizes the number of conflicts for  $V$ 
    end if
    if  $\hat{v} \neq$  current value of  $V$  then
      assign  $\hat{v}$  to  $V$ 
       $number_{moves} \leftarrow number_{moves} + 1$ 
    end if
  end while ( $Max_{moves}$ )
  Check activity constraints
   $NV \leftarrow$  all new active variables
   $S \leftarrow S \cup \hat{S}$ 
  if  $NV$  is not empty
    MCRW-CCCSPIncremental( $S, NV, Max_{moves}, p$ )
  else return  $S$ 
end MCRW-CCCSPIncremental

```

Figure 4.19: MCRW-CCCSP_{Incremental} Algorithm

```

1) Randomly assign 2 initial active variables: APPAREL and SHOES
   APPAREL=SET, SHOES=Running
2) Check Activity Constraints:
   Active SET
   Randomly assign a value to SET
   SET=Plant Suit
   S={APPAREL,SHOES,SET}
3) Repairing Loops: S={APPAREL,SHOES,SET}, #Violated Constraints=1
   [3.1] Randomly select SET as a conflict variable and assign a new value Dress to SET
   APPAREL= SET, SHOES=Running, SET=Dress
   #Violated Constraints=1
   [3.2] Randomly select SET as a conflict variable
   SHOES= Pump
   #Violated Constraints = 0
   a consistent assignment found for S = {APPAREL=SET,SET=Dress,SHOES=Pump}
   Exit the repairing loop
4) Check Activity Constraints
   SET=Dress activates JEWELRY and HANDBAG
   JEWELRY = J4
   HANDBAG = HB4

   Repairing Loops: S'={JEWELRY, HANDBAG}, #Violated Constraints=2
   [4.1] Randomly select HANDBAG as a conflict variable
   HANDBAG = HB2
   #Violated constraints = 0
   Exit the repairing loop
5) Check Activity Constraints
   No new active variable

Solution: APPAREL=SET, SET=Dress, SHOES=Pump, JEWELRY=J4, HANDBAG=HB2

```

Figure 4.20: MCRW-CCCSP_{Incremental} for the Dress Up Game Example (Figure 4.1)

and the incremental strategies we only apply the MCRW and the SDRW algorithms because the TS algorithm only works with a static configuration.

4.3 Conclusions

We proposed a general CCCSP framework to handle conditional variables, compatible constraints and composite variables. Four variant complete methods for solving CCCSPs: FC-CCCSP, FC⁺CCCSP, MACCCCSP, and MAC⁺CCCSP algorithms are proposed. Different degrees of constraint propagation techniques are applied in these methods. In addition, we introduced incomplete methods to solve dynamic CSPs in the CCCSP framework. It is a novel approach to invest incomplete methods in conditional CSPs. The static, the dynamic and the incremental strategies are used in the repairing stages of MCRW, SDRW and TS. The experiments comparing these algorithms are described in Chapter 7.

As we mentioned in the introduction, in a case of searching for the most preferred solution (e.g., if we are going to a party, we prefer to wear a dress with pump), handling preferences is the key. Later on, the CCCSP framework is extended to manage preferences as soft constraints. The full details of this extension is described in Chapter 6

5. CCCSP FRAMEWORK FOR TEMPORAL PROBLEMS

A well known approach to managing the numeric and symbolic aspects of time is to view them as Constraint Satisfaction Problems (CSPs). Our aim is to extend the temporal CSP formalism in order to express dynamic aspects of temporal problems. Indeed, in many real life applications, the set of variables involved in solving the temporal constraint problem is not known in advance. More precisely, while some temporal variables (called events) are present in the initial problem, others can be dynamically added to the problem during the resolution process via activity constraints and composite variables. Activity constraints allow some temporal events to be activated (i.e., added to the problem) when activity conditions are satisfied. Composite variables representing disjunctive temporal events describe finite domains of events. We call these problems *Conditional and Composite Temporal Constraint Satisfaction Problems* (CCTCSPs). The CCTCSP framework is more expressive than the hybrid temporal framework of Meiri [68]. Indeed, conditional and disjunctive temporal events can be represented and managed in our model and not in Meiri [68].

In this chapter, we define the CCCSP framework for temporal constraints. Representing temporal problems involving conditional constraints and composite events in the CCTCSP framework is discussed through temporal examples. A preliminary version of this chapter has been published [77, 78].

5.1 Conditional and Composite Temporal Constraint Satisfaction Problems (CCTCSPs)

The CCTCSP framework is based on TemPro [74] and our CCCSP framework as described in Chapter 4 [82]. The CCTCSP model has the ability to represent constraint problems involving numeric information, symbolic information, conditional constraints and composite variables. Numeric and symbolic temporal constraints, as well as conditional constraints and composite variables, are managed within the same constraint graph. We define a CCTCSP based on the definition of CCCSP and TemPro given in Definitions 4.1.1 and 3.1.1.

Definition 5.1.1. *A Composite and Conditional Temporal Constraint Satisfaction Problem (CCTCSP) is a tuple $\langle E, D_E, X, D_X, IV, C, Act \rangle$, where*

- $E = \{e_1, \dots, e_n\}$ is a finite set of temporal variables called events. Events have a uniform representation made up of a proposition and its temporal qualification: $Evt = OCCUR(p, I)$ denotes the proposition p occurring over the interval I . For the sake of notation simplicity, an event is used in this thesis to denote its temporal qualification.
- $D_E = \{D_{e_1}, \dots, D_{e_n}\}$ is the set of domains of the events. Each domain D_{e_i} is the finite and discrete set of numeric intervals the event e_i can take. D_{e_i} is expressed by SOPO, the quadruple $[begttime_{e_i}, endtime_{e_i}, duration_{e_i}, step_{e_i}]$, where $begttime_{e_i}$ and $endtime_{e_i}$ are respectively the earliest start time and the latest end time of the corresponding event. $duration_{e_i}$ is the duration of the event and $step_{e_i}$ defines the distance between the starting time of two adjacent intervals within the event domain. The $step_{e_i}$ is defined as the distance between the starting time of two adjacent intervals within the SOPO.
- $X = \{x_1, \dots, x_m\}$ is the finite set of composite variables.

- $D_X = \{D_{x_1}, \dots, D_{x_m}\}$ is the set of domains of the composite variables. Each domain D_{x_i} is the set of possible events the composite variable x_i can take.
- IV is the set of initial variables. An initial variable can be a composite variable or an event. $IV \subseteq E \cup X$.
- $C = \{C_1, \dots, C_p\}$ is the set of compatibility constraints. Each compatibility constraint is a qualitative temporal relation between two variables. A qualitative temporal relation is a disjunction of Allen relations [4] (see Table 3.1 for the definition of the Allen relations).
- Act is the set of activity constraints. Act has the same format as defined in the CCCSP framework.

The differences between a CCTCSP and a CCCSP are that a CCTCSP variable particularly represents a temporal event, the domain of each variable is a set of time intervals generated from its associated SOPO and a binary constraint represents a compatible disjunctive of Allen relations.

Throughout the rest of this chapter, we demonstrate representing conditional temporal constraint problems using the CCTCSP framework. We introduce using slicing SOPO techniques with composite variables and disjunctive SOPOs. Also an advanced form of inconsistent checking is performed by the numeric and symbolic conversion.

Example 5.1.1.

John, Mike and Lisa are going to see a movie on Friday. John will pick Lisa up and Mike will meet them at the theater. If John arrives at Lisa's before 7:30, then they will stop at a convenience store to get some snacks and pop. It will take them 35 minutes to reach the theater if they stop at the store and 15 minutes if they do not

stop. There are three different shows playing: $movie_1$, $movie_2$ and $movie_3$. If they finish the movie by 9:15, they will stop at a pizza place 10 minutes after the end of the movie and will stay there for 30 minutes. John leaves home between 7:00 and 7:20. Lisa lives far from John (15 minutes driving). Mike leaves home between 7:15 and 7:20 and it takes him 20 minutes to go to the theater. $movie_1$, $movie_2$ and $movie_3$ start at 7:30, 7:45 and 7:55 and finish at 9:00, 9:10 and 9:20 respectively.

The goal here is to check if this story is consistent (i.e., is a feasible scenario). The story can be represented by the CCTCSP of Figure 5.1. There are six events JPL , JLM , JLS , M , $P1$ and $P2$ and one composite variable WM , which represent the following information:

- JPL : John will pick Lisa up.
- JLM : John and Lisa are going to see a movie.
- JLS : John and Lisa will stop at a convenience store before going to the theater.
- M : Mike is going to see a movie.
- $P1$: John, Mike and Lisa will stop at a pizza place after watching $movie_1$.
- $P2$: John, Mike and Lisa will stop at a pizza place after watching $movie_2$.
- WM : John, Mike and Lisa are watching a movie. WM can take one of the following three values from its domain: $M1$, $M2$ and $M3$ corresponding to $movie_1$, $movie_2$ and $movie_3$ respectively.

Each event domain is represented by the quadruple $[begin\ time, end\ time, duration, step]$. In the case of JPL , the domain is $[0, 35, 15, 1]$ where 0 (the reference time corresponding to 7:00) is the earliest start time, 35 is the latest end time, 15 is the duration, and 1 (corresponding

to 1 min) is the step. By the SOPO structure, the step parameter can be applied for any n number in different time granular units (e.g., hours, days and months). For the sake of simplicity, all the events in this story are assumed to have the same step. Arcs represent either a compatibility constraint or an activity constraint (indicated by an arc with middle arrow) between variables. The compatibility constraint is denoted by one or more qualitative relations. The activity constraint shows the condition to be satisfied and the qualitative relation between the two variables in case where the condition is true. Each qualitative relation is a disjunction of some Allen relations [4]. For example, the relation bm between JPL and JLM denotes the disjunction $before \vee meet$.

Figure 5.2 shows the result of example 5.1.1 after performing arc consistency in the preprocessing stage. Variable $M1$ is removed from the domain of composite variable WM because variable $M1$ has no support in variable M . Figure 5.3 illustrates the steps of searching for a solution with the constraint propagation technique (FC). The JPL event is assigned by time interval $(0, 15)$ (top-left). It satisfies activity constraint $JPL = (0, 15) \xrightarrow{active} JLS$. Then the FCCCTCSP algorithm dynamically propagate constraint to the future variable JLS . After assigning the value to JLS and propagating the constraint to the next future variable WM , an inconsistency is detected. None of variables in the domain of composite variable WM supports any interval pair in the domain of JLS . The algorithm backtracks to the previous variable JPL and assigns the next available value. Backtracking occurs repeatedly until interval $(15, 30)$ is assigned to variable JPL . Then variable JLM is added to the solution. Variable $M3$ is removed from composite variable WM because it has no support for the partial current assignment (top-right). Variable M and JLM are assigned values by $(15, 35)$ and $(30, 45)$. Composite variable WM has only one temporal event, $M2$, left in its domain. $M2$ is activated. The activity constraint $WM = M2 \xrightarrow{active} P2$ is satisfied, and $P2$ is added into the problem. A consistent

assignment is “John picks Lisa up at 7:15 p.m. They do not stop at a convenience store but directly go to the theater and they arrive at 7:45 p.m. Mike leaves home at 7:15 and he arrives at the theater at 7:35. They watch movie *M2*. After the movie they stop at a pizza place from 9:20-9:50.”

Figure 5.1: CCTCSP Network of Example 5.1.1

5.2 Slicing SOPOs and Composite Variables

Slicing SOPO techniques can be used to divide temporal events' domains when there are numeric temporal conditions on the events (e.g., if we start the dinner before 6:00 p.m. and if

Figure 5.2: Arc Consistency on the Initial CCTCSP of Example 5.1.1

Figure 5.3: Constraint Propagation on the CCTCSP of Example 5.1.1

cooking finishes after 6:00 p.m.)

Given $event_i$ and its corresponding $SOPO_i = [inf_i, sup_i, d_i, s_i]$, an associated numeric temporal condition has one of the following forms (where X is a time point):

- If $event_i$ starts before X
- If $event_i$ starts after X
- If $event_i$ finishes before X
- If $event_i$ finishes after X

The inf_i and sup_i values can be sliced in order to generate two sets of time intervals where the first set satisfies the numeric temporal condition and the second set fails to satisfy the conditional statement. Using composite variables with slicing SOPO techniques makes temporal problems more explicit. A formula for slicing $SOPO_i$, if $event_i$ starts before X , is defined in Figure 5.4. $SOPO_i$ is divided to $SOPO_{i_{s-before-x}}$ and its negation, $\neg SOPO_{i_{s-before-x}}$. Figures 5.5, 5.6 and 5.7 are slicing SOPO formulas for start after, finish before and finish after conditions, respectively.

In the next temporal example (dinner preparation), we demonstrate the use of slicing SOPO techniques with composite variables.

Example 5.2.1.

A maid is required to prepare an appetizer, a soup and a main dish for a supper. She is allowed to use the kitchen from 4:00-8:00 p.m. The food must be ready by 6:00 p.m. The dinner takes 45 minutes and it can be started any time between 6:00-7:00 p.m. The soup and the main dish must be reheated if the dinner starts after 6:15 p.m. and the food is finished before 5:45 p.m. If the dinner starts before

$$\text{SOPO}_{s\text{-after } x} = [\text{inf}_{s\text{-after } x}, \text{sup}_{s\text{-after } x}, d_i, S_i]$$

$$\sim\text{SOPO}_{s\text{-after } x} = \begin{cases} [\text{inf}_i, X + d_i, d_i, S_i], & \text{if } Y = X \\ [\text{inf}_i, S_c - S_i + d_i, d_i, S_i], & \text{if } Y > X \end{cases}$$

where

$$Y = (\text{inf}_i + \lceil (X - \text{inf}_i) / S_i \rceil * S_i)$$

$$\text{inf}_{s\text{-after } x} = \begin{cases} Y, & \text{if } Y > X \\ Y + S_i, & \text{if } Y = X \end{cases}$$

$$\text{sup}_{s\text{-after } x} = \text{sup}_i$$

Figure 5.5: Slicing SOPO by Start After X

$$\text{SOPO}_{f\text{-before } x} = [\text{inf}_{f\text{-before } x}, \text{sup}_{f\text{-before } x}, d_i, S_i]$$

$$\sim \text{SOPO}_{f\text{-before } x} = \begin{cases} [Y - d_i, \text{sup}_i, d_i, S_i], & \text{if } Y = X \\ [Y + S_i, \text{sup}_i, d_i, S_i], & \text{if } Y < X \end{cases}$$

where

$$Y = (\text{inf}_i + \lfloor (X - \text{inf}_i) / S_i \rfloor * S_i + d_i)$$

$$\text{inf}_{f\text{-before } x} = \text{inf}_i$$

$$\text{sup}_{f\text{-before } x} = \begin{cases} Y, & \text{if } Y < X \\ Y - S_i, & \text{if } Y = X \\ X, & \text{if } Y > X \end{cases}$$

Figure 5.6: Slicing SOPO by Finish Before X

$$\text{SOPO}_{f\text{-after } X} = [\text{inf}_{f\text{-after } X}, \text{sup}_{f\text{-after } X}, d_i, S_i]$$

$$\sim\text{SOPO}_{f\text{-after } X} = [\text{inf}_i, X, d_i, S_i]$$

where

$$Y = (\text{inf}_i + \lfloor (X - \text{inf}_i) / S_i \rfloor * S_i)$$

$$Y' = Y + d_i$$

$$\text{inf}_{f\text{-after } X} = \begin{cases} Y + S_i, & \text{if } Y' \leq X \\ Y, & \text{if } Y' > X \text{ and } Y' - X < S_i \\ Y - S_i * \lceil (Y' - X) / S_i \rceil, & \text{if } Y' > X \text{ and } Y' - X \geq S_i \end{cases}$$

$$\text{sup}_{f\text{-after } X} = \text{sup}_i$$

Figure 5.7: Slicing SOPO by Finish After X

6:30p.m, the maid will have to make and serve a desert. The desert can be served from 7:15 p.m. to 7:30 p.m. The order of serving the dinner is the appetizer, the soup, the main dish and then the desert if served. The maid is also responsible for setting up the dining table, serving each dish and cleaning up. Today the maid has her to-do list as follows:

1. Set up table between 5:00-6:00 p.m. (10 minutes)
2. Clean up between 7:00-8:00 p.m. (30 minutes)
3. Main dish - Chicken green curry with rice (40 minutes)
4. Appetizer - Spring rolls (25 minutes)
5. Soup - Tom Yum (30 minutes)
6. Desert - Ice cream and apple pie (10 minutes) / Coffee (7 minutes)

If tonight the boss wants to have the desert and no reheating food before serving, what will be a consistent cooking schedule for the maid?

Figure 5.8 shows a CCTCSP corresponding to Example 5.2.1. In the CCTCSP, there are nine initial active variables and six non-active variables, including one composite variable. An ordering of the initial active variables by decreasing degree (indicated in parentheses) of the most constrained variable is *Having-Dinner* (7), *Serving-Soup* (5), *Serving-Appetizer* (4), *Serving-Maindish* (4), *Cooking-Maindish* (1), *Cooking-Soup* (1), *Cooking-Appetizer* (1), *Setup-Table* (1) and *Clean-up* (1).

In Example 5.2.1, there are the following numeric temporal conditions:

- “if the dinner starts after 6:15 p.m.”,
- “if the food is finished before 5:45 p.m.” and

Figure 5.8: CCTCSP of Cooking Example

- “If the dinner starts before 6:30 p.m.”

The SOPO of event *Having-Dinner*, $[120, 225, 45, 5]$, generates a set of time intervals that can be *sliced* (i.e., divided) into two subsets. The first set, $[140, 225, 45, 5] = \{(140, 185), (145, 190), \dots, (180, 225)\}$, corresponds to cases where the “if the dinner starts after 6:15” condition is satisfied. The second set, $[120, 180, 45, 5] = \{(120, 165), \dots, (135, 180)\}$ corresponds to cases where this condition is not satisfied. If we only consider the case that “the dinner starts after 6:15 p.m.”, the SOPO of event *Having-Dinner* should be restricted to the first subset, $[140, 225, 45, 5]$.

An alternative way to represent Example 5.2.1 is to use the above slicing SOPO technique and composite variables. The original SOPO $[120, 225, 45, 5]$ variable *Having-Dinner* can be sliced into three sub-SOPOs according to the if-conditions:

- *If the dinner starts after 6:15 p.m.:*
 - Slice $SOPO_{Having-Dinner} = [120, 225, 45, 5]$ after $X = 135$
 - $S_c = 120 + \left\lceil \frac{(135-120)}{5} \right\rceil \times 5$
 - $S_c = X = 135$, $inf = S_c + S_i = 135 + 5 = 140$
 - $SOPO_{Having-Dinner_{s-after-6:15}} = [140, 225, 45, 5]$
 - $\neg SOPO_{Having-Dinner_{s-after-6:15}} = [120, 180, 45, 5]$
- *If the dinner starts after 6:15 and before 6:30 p.m.:*
 - Slice $SOPO_{Having-Dinner_{s-after-6:15}} = [140, 225, 45, 5]$ by $X = 150$
 - $S_c = 140 + \left\lceil \frac{(150-140)}{5} \right\rceil \times 5$
 - $S_c = X = 150$, $sup = S_c - S_i + d_i = 150 - 5 + 45 = 190$
 - $SOPO_{Having-Dinner_{s-after-6:15-before-6:30}} = [140, 190, 45, 5]$

$$\circ \neg SOPO_{Having-Dinner_{s-after-6:15-before-6:30}} = [150, 225, 45, 5]$$

Variable *Having-Dinner* is represented with a composite variable containing three events in its domains:

1. The dinner starts before (up to) 6:15, $SOPO_{Up-to-6:15} = [120, 180, 45, 5]$
2. The dinner starts between 6:15-6:30, $SOPO_{S-Between-6:15-6:30} = [140, 190, 45, 5]$
3. The dinner starts from 6:30, $SOPO_{S-From-6:30} = [150, 225, 45, 5]$

Figure 5.9 shows an alternative way to represent the *Having-Dinner*, the *Cooking-Soup* and the *Cooking-Maindish* events by slicing SOPOs and using the slicing techniques and composite variables.

To find a consistent schedule, we first determine the desired configuration, where *Making-Desert* must be active in the initial problem and *Reheating-Maindish* and *Reheating-Soup* must not be in the solution. In this configuration, the domain of composite variable *Having-Dinner* has only variable *S-Between-6:15-6:30* left. The domain of variable *Cooking-Soup* has variable *F-After-5:45* left. The domain of variable *Cooking-Maindish* has variable *F-After-5:45* left.

Figure 5.10 shows the steps used when solving the problem by the BTCCTCSP algorithm. First the *Having-Dinner* variable is assigned *S-Between-6:15-6:30*. Then variable *S-Between-6:15-6:30* is activated and assigned (140, 185). It activates the *Making-Desert* composite variable which has only one variable in its domain. The *Pie* variable is activated and a time interval, (140,150), is assigned to it and the *Serving-Pie* variable is activated. However, there is no time interval in the *Serving-Pie* domain that satisfies the current assignment to the *S-Between-6:15-6:30* variable. The algorithm backtracks to the previous variable *Pie*. Thrashing and redundant checking occur among the variables until there is no value left in the *Having-Dinner*'s domain to assign and the algorithm stops with no solution for this configuration.

Figure 5.9: Representing Having-Dinner, Cooking-Soup, Cooking-Maindish by Composite Variables

Figure 5.10: Steps in Solving the CCTCSP in Example 5.2.1 using BT

Figure 5.11: Steps in Solving the CCTCSP in Example 5.2.1 using FC⁺

Thrashing and redundant checking can also occur if either the FCCCTCSP or the MAC-CCTCSP algorithms are applied to solve the problem because the *Making-Desert*, *Pie* and *Serving-Pie* variables are non-active variables in the initial state. Variables *Pie* and *Serving-Pie* will not be active until variable *Making-Desert* is assigned value *Pie*.

The advanced constraint propagation techniques that we proposed in Chapter 4 can detect the some inconsistencies earlier. In Figure 5.11, the steps followed when searching for a solution show that the FC⁺CCTCSP algorithm can detect inconsistencies earlier than FCCCTCSP and BTCCTCSP by propagating constraints to the future active variables *Pie* and *Serving-Pie*. An inconsistency is detected at *Serving-Pie* because no support value can be found for variable *S-Between-6:15-6:30*.

5.3 Numeric to Symbolic Conversion

Conversion of numeric constraints to symbolic ones is used in TemPro [74] in the preprocessing state to eliminate inconsistent symbolic constraints. All conversion rules are defined in Table 3.11. Numeric to symbolic conversion can be applied to a CCTCSP either as preprocessing before the search, in which case conversion is applied to all binary constraints of the initial active variables, or during the search, in which case conversion is applied to the current variable, any newly possible future active variables before assigning a value to the current variable. Possible future active variables are considered when conversion is used with the FC⁺CCTCSP and the MAC⁺CCTCSP algorithms.

Consider again the cooking problem defined in Example 5.2.1. For this problem, directly applying numeric to symbolic conversion in the preprocessing state for all the initial active variables does not remove any symbolic constraints. However, if this conversion is applied during the search, using either the FCCCTCSP or the MAC⁺CCTCSP algorithms, some inconsistencies

may be detected earlier than if no conversion is used. The search starts from considering *Having-Dinner* as the current variable and *Making-Desert*, *Pie* and *Serving-Pie* as future active variables. The inconsistency is detected after checking all the values in variable *S-Between-6:15-6:30*'s domain. The advantage of the numeric and symbolic conversion here is detecting inconsistencies prior to constraint propagation to *Serving-Pie*. In this case, the numeric to symbolic conversion is applied over constraint:

$$Serving-Pie_{[195,210,3,5]} \xrightarrow{df} S-Between-6:15-6:30_{[140,190,45,5]}$$

As a result, the d and f relations are removed from the constraint by rule number 10 from Table 3.11, which is as follows:

$$SOPO_i.inf + SOPO_i.d > SOPO_j.sup \text{ remove } F, f, d$$

No temporal relation is left in the constraint that can be satisfied by the SOPO and the inconsistency is detected.

The slicing techniques for SOPOs support numeric to symbolic conversion. When a SOPO is sliced, and only one sub SOPO is considering during the search, checking based on numeric to symbolic conversion is tighter than it would be if performed on the original SOPO.

5.4 Numeric Temporal Constraints and SOPOs

Frequently, in a temporal problem, an event has a range of permitted intervals (e.g., going to school takes 10-20 minutes). Composite variables and SOPOs are practical to represent such numeric temporal constraints. In this final example, we show the flexibility of the CCTCSP framework to represent numeric temporal constraints on events' intervals.

Example 5.4.1.

Tom has a class at 8:00 am. He can either make breakfast for himself, which takes 10-15 minutes, or get something to eat from a local store in less than 5 minutes. After breakfast he goes to school either by car which takes 20-30 minutes or by bus which takes at least 45 minutes. Today Tom gets up between 7:30 and 7:40.

There are two composite temporal events occurring in the temporal problem:

1. Tom having breakfast is denoted by the composite variable *Breakfast*, which has two optional events:
 - (a) making his own breakfast, denoted as *Make*
 - (b) buying something to eat, denoted as *Buy*
2. Tom going to school is denoted by *GotoSchool*, which has two optional events:
 - (a) taking a bus is denoted by the event *Bus*
 - (b) taking a car is denoted by the event *Car*

Unlike in the previous examples, there are numeric constraints over the events' intervals (e.g., buying breakfast takes 1-5 minutes). Because the basic structure of SOPO only allows an absolute time interval in an event; therefore a set of SOPOs is required to encode all enumerated absolute time intervals of the event. The temporal problem states that John gets up between 7:30 and 7:40 and it takes him 1-5 minutes to buy breakfast. The following set of SOPOs represent the domain of event *Buy* and the numeric temporal constraints: $\{ [30,41,1,1], [30,42,2,1], [30,43,3,1], [30,44,4,1], [30,45,5,1] \}$.

Figure 5.12(a) shows a complete CCTCSP network with all enumerated SOPOs for the problem. The binary relation between composite variables *Breakfast* and *GotoSchool* is *bm*. Figure 5.12(b) shows how the sets of SOPOs can be simplified to formulas. In this case, the

earliest time for John to go to school is 7:41 if he gets up at 7:30 and buys breakfast in 1 minute. If he gets up at 7:40 and makes his own breakfast for 15 minutes, the latest time for John to go to school is 7:55.

Figure 5.12: CCTCSP Network for Example 5.4.1

5.5 Conclusions

We applied the CCCSP framework to temporal constraint problems. CCTCSP is based on TemPro [74] and is sufficiently expressive to describe complex temporal problems where numeric and symbolic temporal constraints, as well as conditional and composite events are presented. Some techniques for encoding and checking for inconsistencies are described. First, using composite variables with the slicing SOPO techniques makes temporal problems more explicit. Secondly, the numeric to symbolic conversion can detect some inconsistent partial assignments prior to constraint propagation. Finally, numeric constraints on events' intervals can be encoded by a set of SOPOs. Next in Chapter 6, the CCTCSP framework is extended

with preferences.

6. CCCSPs AND CCTCSPs WITH PREFERENCES

In a case of searching for the most preferable solution according to a given objective function, managing preferences is the key. We apply a c-semiring structure defined in Bistarelli et al. [16] to represent quantitative preferences in CCCSPs and CCTCSPs. We call the extended frameworks: CCCSPs and CCTCSPs with Preferences (CCCSPPs and CCTCSPPs respectively). Here, preferences are functions mapping a selected value to a preference degree (i.e., a real number from 0.0 to 1.0). The most preferred value is indicated by the highest preference degree whereas the lowest preference degree indicates the least preferred choice. In this chapter, we define different preference levels and aggregate functions for evaluating global preference degrees and configuration preference degrees of complete solutions. The Branch and Bound technique [86] is integrated to the complete methods to find preferable solutions. Encoding some temporal preference modifiers are discussed.

6.1 CCCSPs with Preferences

Definition 6.1.1. *A CCCSP with Preferences is a CCCSP $\langle X, D_X, Y, D_Y, IV, C, Act \rangle$ augmented with a c-semiring, S , where S is a tuple $\langle P_f, +, \times, 0, 1 \rangle$ such that:*

- *P_f is a set of preference functions, $\{C_1 : P_{f_1}, \dots, C_n : P_{f_n}\}$, and $[0.0, 1.0] \in P_{f_i}$. C_i is a condition of P_{f_i} (i.e., if $X = a : P_{f_i}$). If C_i is true then P_{f_i} is considered,*
- *$+$, called the additive operation, is*
 - *close (i.e., $a, b \in P_{f_i}$ implies $a + b \in P_{f_i}$),*

- commutative (i.e., $a + b = b + a$),
- associative (i.e., $a + (b + c) = (a + b) + c$),
- idempotent (i.e., $a \in P_{f_i}$ implies $a + a = a$),
- 1 is the absorbing element of $+$ (i.e., $a \in P_{f_i}$ implies $1 + a = a + 1 = 1$),
- $\times$, called the multiplicative operation is
 - close,
 - commutative,
 - associative,
 - distributes over $+$ (i.e., $a \times (b + c) = (a \times b) + (a \times c)$),
 - 0 is the absorbing element of $\times$ (i.e., $a \in P_{f_i}$ implies $a \times 0 = 0 \times a = 0$).

The structure of the CCCSP framework, implies different preference levels. Preferences can be over the domains of simple variables, the compatible constraints, the domains of composite variables, and the activity constraints. Preferences over the domains of simple variables are for individual favorite values. Preferences over the compatible pairs on a binary constraint indicate favorite combination values. Preferences over the domains of composite variables suggest desire variables among the disjunctive choices. Preferences over the activity constraints indicate favors on adding conditional variables. Consequently, we classify preference functions over different elements into the following four levels.

Definition 6.1.2. *A simple variable preference function, $P_{f_s}(X, a)$, is a one to one function mapping each value, a , in a domain of sample variables, X , to a degree of preferences. The preference function indicates like and not-like atomic values when it assigned to a solution.*

Definition 6.1.3. A *Composite variable preference function*, $P_{f_{cp}}(Y, X)$, is a one to one function mapping each variable, X , in a domain of composite variables, Y , to a degree of preferences. The preference function indicates a preferred configuration solution on selecting a disjunctive variable.

Definition 6.1.4. A *Constraint Preference function*, $P_{f_c}((X_i, X_j), (a, b))$, is a one to one function mapping each compatible pair, (a, b) , on a binary constraint, C_{X_i, X_j} , to a preference degree. The preference function indicates a preference over combination values.

Definition 6.1.5. A *Activity Preference function*, $P_{f_a}(X_i = a \xrightarrow{\text{active}} X_j)$, is a one to one function mapping each activity constraint to a preference degree of conditional variables, X_j . The preference function indicates a preferred configuration solution on having a new active/conditional variable through the activity constraints.

If more than one preference function are imposed on the same element (i.e., domains of simple variables, domain of composite variables, compatible constraints and activity constraints), each preference function is restricted under a condition. Only the preference function that its condition satisfied is calculated.

In a CCCSPP, the four levels of preference functions have to be aggregated in order to evaluate a global preference degree and a configuration preference degree of a complete solution. Then, all possible complete solutions are compared to find the best preferred solution.

Definition 6.1.6. A *Global Preference Degree (GP)* of a set of assigned variables indicates a global preference level of the assigned variables and compatible constraints. If $Sol = \{a_1, \dots, a_n\}$, where a_i is an assignment of an active variable V_i , is a complete solution and the preference degree for each assigned simple variable is $P_{f_s}(V_i, a_i)$ and the preference degree of each compatible pairs is $P_{f_c}((V_i, V_j), (a_i, a_j))$, the global preference degree for Sol , $GP(Sol) =$

$\min(\min(P_{f_s}(V_i, a_i)), \min(P_{f_c}((V_i, V_j), (a_i, a_j))))$ where $1 \leq i, j \leq n$. The $\min$ function returns the minimum value of the simple variable and constraint preference functions.

Definition 6.1.7. The **Best Assignment (BA)** is a complete solution having the highest global preference degree. If $Sol_{all} = \{Sol_1 \dots Sol_m\}$ is a set of all possible complete solutions, the best solution $BA(Sol_{all}) = \max(GP(Sol_i))$ where $1 \leq i \leq m$. The $\max$ function returns the solution having the maximum value of the global preference degree.

A CCCSPP contains many different CSP configurations which are caused by choosing different variables for composite variables and by adding new variables through activity constraints. By default, the activity preference degrees of all initial active variables are set to the highest level (i.e., 1.0) because they must be in every solution regardless of conditions. The preference functions over domains of composite variables and activity constraints guide the search to find the most favorite generated CSP configuration. Choosing a preferred CSP configuration is a higher level of preferences. Therefore, the configuration preference degree (defined below) and the global preference degree indicate the satisfaction degree of the CCCSPP solution.

Definition 6.1.8. A **Configuration Preference Degree (CP)** of a set of active variables represents a preference level of the generated CSP configuration. If $CSP = \{V_1, \dots, V_n\} \cup \{NV_1, \dots, NV_m\} \cup \{CV_1, X_{cv_1}, \dots, CV_p, X_{cv_p}\}$ where V_i is an initial active variable, NV_j is a new active variable via a satisfied activity constraint, CV_k is a composite variables and X_{cv_k} is the chosen variable of CV_k , then $CP(CSP) = \min(\min(P_{f_a}(NV_i)), \min(P_{f_{cp}}(CV_j, X_{cv_j})))$ where $1 \leq i \leq m$ and $1 \leq j \leq p$. The $\min$ function returns the minimum value of the activity and composite preference functions.

Definition 6.1.9. The **Best Configuration (BC)** is a complete CSP configuration having the highest configuration preference degree. If $CCCSPP_{all} = \{CSP_1 \dots CSP_m\}$ is a set of all

possible generated CSPs, the best configuration is $BC(CCCSPP_{all}) = \max(CP(CSP_i))$ where $1 \leq i \leq m$. The $\max$ function returns the CSP having the highest configuration preference degree.

Definition 6.1.10. *The **Best Solution (BS)** of a CCCSPP is the best assignment that belongs the best configuration. If $CCCSPP_{all} = \{CSP_1 \dots CSP_m\}$ is a set of all possible generated CSPs and for each $CSP_i = \{Sol_{i_1}, \dots, Sol_{i_k}\}$ is a set of all possible assignments, then $BS(CCCSPP) = BA(BC(CCCSPP_{all}))$.*

In the next section, we demonstrate encoding the four different preference functions and applying the configuration preference degree and the global preference degree to the dress up game example (Figure 4.1).

6.2 CCCSPP Example

As mentioned in Chapter 4, in the dress up game example or our daily life dress up style, it is common that we all have preferences on what kind/style of clothes or shoes we really like and feel more comfortable to wear. Moreover, in different occasions such as a job interview, a party, a sport game or camping, we still want to look good in an appropriate dress or outfit. Therefore, preferences can be over a set of items (domains), a set of matched items (constraints), choices on group items (composites), as well as upon occasions (conditions). In order to describe the CCCSPP framework, the dress up game example is extended by the following extra information.

- There are four occasions: job interview, party, sport and camping to be considered.
 - If the occasion is an job interview or a party, we prefer to wear set to mix-and-match pieces (e.g., top and bottom).

- If the occasion is a camping or a sport event, we prefer to wear top and bottom to set.
- We always prefer to wear casual and boot to sandal, running and pump.
- We like handbag HB3 and HB4 the most.
- For matches clothes (the top-bottom constraint): we like blouse with skirt, T-shirt with capri and jacket with jean the most.
- For the set-shoes constraint: we prefer skirt-suit with boot and pant-suit with casual to the rests.

We can formulate the preference information on the dressing game example as follows.

- We add a new simple variable called occasion where

$$D_{occasion} = \{interview, party, sport, camping\}.$$

- There is no new binary constraint added.
- Two simple variable preference functions:

$$P_{f_{s_{Shoes}}} = \{Casual_{1.0}, Boot_{0.9}, Sandal_{0.8}, Running_{0.7}, Pump_{0.6}\},$$

$$P_{f_{s_{Handbag}}} = \{HB3_{1.0}, HB4_{1.0}, HB1_{0.8}, HB2_{0.8}\}.$$

(notes: for the other simple variables, the simple variable preference degrees are all set to 1.0.)

- Two composite variable preference functions with their conditions:

$$\begin{aligned}
P_{f_{cp}Apparel_1} &= \{Set_{1.0}, 2-pieces_{0.8}\} \\
P_{f_{cp}Apparel_2} &= \{Set_{0.8}, 2-pieces_{1.0}\} \\
V_{Occasion} &= interview : P_{f_{cp}Apparel_1} \\
V_{Occasion} &= party : P_{f_{cp}Apparel_1} \\
V_{Occasion} &= sport : P_{f_{cp}Apparel_2} \\
V_{Occasion} &= camping : P_{f_{cp}Apparel_2}
\end{aligned}$$

- Two constraint preference functions: ¹

$$\begin{aligned}
P_{f_{cTop,Bottom}} &= \{(Bl, Pt)_{0.8}, (Bl, Sk)_{1.0}, (Sh, Pt)_{0.8}, (Sh, Sk)_{0.8}, (TS, Cp)_{1.0}, (TS, St)_{0.8}, \\
&\quad (Tk, St)_{0.8}, (Tk, Cp)_{0.8}, (Jk, Jn)_{1.0}\} \\
P_{f_{cSet,Shoes}} &= \{(SS, PP)_{0.8}, (SS, BT)_{1.0}, (DS, PP)_{0.8}, (PS, PP)_{0.8}, (PS, CL)_{1.0}\}.
\end{aligned}$$

If the occasion is a job interview, dressing up in a set of pant-suit with casual shoes is the best choice. The configuration and the global preference degree of the solution are 1.0, where $P_{f_{cp}Apparel_1}$ is active because $V_{Occasion} = interview$, $P_{f_{cp}}(Apparel_1, Set) = 1.0$, $P_{f_s}(Set, PS) = 1.0$, $P_{f_c}((Set, Shoes), (PS, CL)) = 1.0$ and $P_{f_s}(Shoes, CL) = 1.0$. Another possible dress up is to wear a skirt-suit with boot, however the global preference degree of the second choice decreases to 0.9 because $P_{f_s}(Shoes, BT) = 0.9$.

For all initial variables in the dressing game, the degree of each variable is set to 1, If we say, “we do not really like to carry a handbag or to wear a jewelry” then we set the activity preference degree on the activity constraints: $SET = Dress \xrightarrow{active} JEWELRY_{0.9}, HANDBAG_{0.8}$ and $BOTTOM = Skirt \xrightarrow{active} HANDBAG_{0.8}$. Any solution that has either jewelry or handbag participating, the configuration preference of the solution is lower than not having them because when jewelry, handbag are not active in the solution the configuration preference degree is 1.0. Otherwise, wearing a dress with some jewelries and also carrying a handbag produces the

¹notes: $Bl=Blouse$, $Pt=Pant$, $Sk=Skirt$, $Sh=Shirt$, $TS=T-Shirt$, $Cp=Capri$, $Tk=Tank$, $St=Short$, $Jk=Jacket$, $Jn=Jean$, $SS=Skirt-Suit$, $DS=Dress$, $PS=Pant-Suit$, $PP=Pump$, $CL=Casual$, $BT=Boot$

configuration preference degree at 0.8, whereas wearing a skirt and carrying handbag obtains the configuration preference degree at 0.9.

In the next section, we demonstrate integrating and applying the CCCSPP framework into the CCTCSP framework including encoding some temporal preference modifiers.

6.3 CCTCSPs with Preferences

We acknowledge that preferences play an important role in many real temporal problems to express non-crisp desire of start/end times, time intervals and temporal relations of feasible scenarios. For instance, *“We have to leave home before 8:30 am. The earlier we leave is the better because the road is not busy and not too hot.”* Temporal preferences are often implicit. In the scenario, *“earlier”* indicates different favors on the *“leaving”* time points. To deal with the implicit temporal preferences (such as *early, late, about k pm, etc.*), we need to transform each of them into a formal explicit corresponding preference function. Similar to CCCSPPs, in the CCTCSPP framework, there are four types of preferences related to temporal problems:

1. numeric temporal preferences,
2. symbolic temporal preferences,
3. composite temporal preferences and
4. activity temporal preferences.

Numeric and symbolic temporal preferences associate time intervals and temporal relations with degrees of preferences in order to favor some temporal decisions, respectively. The composite temporal preference is a higher level of preferences among the temporal choices of a composite variable. The activity temporal preference favors new active conditional temporal variables.

TemPro supports hybrid temporal problems, so preference values can be imposed on both numeric and symbolic temporal constraints. There are two types of soft temporal constraints: soft numeric temporal constraints and soft symbolic temporal constraints, both of which are defined over the c-semiring.

Definition 6.3.1. *A Soft Numeric Temporal Constraint (SNTC) is defined on a one to one function $f_n : B \rightarrow P_f$, where $b_i \in B$ is a possible beginning time point of a SOPO. P_f is a set of preference values. (denote $f_{n:e_i}$ is a SNTC of a temporal variable, e_i)*

Definition 6.3.2. *A Soft Symbolic Temporal Constraint (SSTC) is defined on a one to one function $f_s : R \rightarrow P_f$. $R = \{r_1, \dots, r_{13}\}$, where r_i is an Allen's relation. P_f is a set of preference values. (denote $f_{s:c_{ij}}$ is a SSTC of a symbolic temporal relation, c_{ij} , between e_i and e_j).*

At this point, combining the local numeric degrees and the local symbolic degrees is required to calculate the global preference degrees for solutions. In fact, a pair of two intervals associates with an Allen's relation. A local preference degree of an associated relation can be obtained from combining a pair of the local numeric preference degrees by the min operation.

Definition 6.3.3. *An Associated Local Symbolic Preference (ALSP) is a value of a local symbolic preference degree which the relation is derived from two local numeric preference degrees to its associated Allen's relation.*

Given: two intervals $I = (b_i, e_i)$ and $J = (b_j, e_j)$,

$A_{R_{ij}} \in R$, is an associated Allen's relation between I and J

Then:

$$ALSP(A_{R_{ij}}) = \min(f_n(b_i), f_n(b_j)),$$

where the min function returns the minimum value of the soft numeric temporal

constraints.

A CCTCSPP solution is composed by a list of intervals. Each assignment consists of two intervals and an Allen's relation. We define a new level of local preferences for a consistent binary assignment to represent satisfaction level of the local assignment. The global preference of a solution is computed from combining these local preferences of the consistent binary assignments by the min operation, whereas the max operation is used to compare the global preferences to get the optimal solution.

Definition 6.3.4. *A Consistent Binary Assignment Preference (CBAP) is a value of a local binary assignment representing a satisfaction level of an assignment which is derived from combining local numeric and local symbolic preference degrees.*

Given: a consistent binary assignment $E_i r E_j$, where

$E_i = (b_i, e_i)$ and $E_j = (b_j, e_j)$, and

$\alpha_i = f_{n:E_i}(b_i), \alpha_j = f_{n:E_j}(b_j), \alpha_r = f_{s:C_{ij}}(r),$

Then: $ALSP(A_{R_{ij}}) = \min(\alpha_i, \alpha_j)$

$CBAP(E_i r E_j) = \min(ALSP(A_{R_{ij}}), \alpha_r)$

Definition 6.3.5. *A Global Temporal Preference (GTP) is a value to represent a satisfaction level of a solution which is combining from local preference temporal degrees of all consistent binary assignments.*

Given: a solution t ,

a set of consistent binary assignments,

$A = \{A_{ij}, \dots, A_{mn}\}$

Then:

$$GP(t) = \min \{CBAP(A_{ij}), \dots, CBAP(A_{mn})\},$$

where the *min* function returns the minimum value of CBAPs.

Definition 6.3.6. An *Optimal Solution (Opt)* is a solution having the highest global preference degree.

Given: a CCTCSPP, P , and a set of solutions

$$T = \{t_1, \dots, t_n\}$$

Then:

$$Opt(P) = \max \{GP(t_1), \dots, GP(t_n)\},$$

where the *max* function returns a solution, t , having the maximum value of GPs.

A CCTCSPP's configuration is changed by composite temporal events and conditional temporal events. As in the CCSPP framework, we can evaluate the configuration preference degree separately from the global preference degree. However, the configuration preference degree and the global preference degree can be aggregated as an alternative objective function. We define two temporal preference functions for composite temporal events and conditional temporal events. In addition, a multiplication operation is defined to adjust a new preference value.

Definition 6.3.7. A *Soft Composite Temporal Constraints (SCTC)* is defined on a one to one function $f_c : E \rightarrow P_f$. $E = \{e_1, \dots, e_p\}$, where e_i is a temporal event in the domain of a composite temporal variable, X_i . P_f is a set of preference values.

Definition 6.3.8. A *Soft Activity Temporal Constraints (SATC)* is defined on a one to one function $f_a : Act \rightarrow P_f$, where $E_i = (b_i, e_i) \rightarrow E_j \in Act$. P_f is a set of preference values over the conditional events, E_j .

A multiplication operation, $*$, takes place to combine the preference degree on the chosen temporal event and on the conditional temporal event to its associated numeric preference

functions. The new levels of numeric and symbolic preference degrees are adjust, if the temporal variable takes part in the problem.

Given: a composite temporal event, X , and its domain

$$D_X = \{e_1, \dots, e_u\},$$

a preference function $f_{c:X}$,

e_i is the selected event.

Then: new levels of numeric and symbolic preference degrees

$$f_{n:X_{e_i}}(b_j) = f_{c:X}(e_i) * f_{n:e_i}(b_j),$$

where b_j is a possible beginning time of e_i .

$$f_{s:C_{e_i e_l}}(r_j) = f_{c:X}(e_i) * f_{s:C_{e_i e_l}}(r_j)$$

where r_j is a symbolic relation between e_i and another event e_l .

Conditional preferences allow a set of preference functions under certain conditions associates with a temporal event or a composite event. Only corresponding preference functions are considered, if its condition is true. In other word, preference functions are dynamic and depend on an assignment of another variable.

To express the conditional preferences, a temporal event or a composite event has two preference states to indicate its preference status: instantiated-preference or non-instantiated-preference. A temporal event without conditional preferences is always set to the instantiated-preference state. Because there is only one associated preference function for the temporal event, no need to be changed in the future. However a temporal event under conditional preferences is set its preference status as non-instantiated at the initial problem. During the search for a solution, the preference status of an non-instantiated-preference event is changed dynamically to preference-instantiated when one of its conditional preference functions is satisfied.

Definition 6.3.9. A *Conditional Temporal Preferences (CTP)* is defined in term of relation between a set of preferences functions under conditions and a temporal event. A temporal event which cannot define a preference function at the initial state is set as a non-instantiated preference state.

Given: a temporal event, e , with non-instantiated preference

a set of preference functions and associated conditions of e ,

$F_e = \{cp_1 \rightarrow f_1, \dots, cp_n \rightarrow f_n\}$ *where*

cp_i is a condition assignment of $e_j = v_l$

Then: if cp_i is fired, f_i is hold and instantiated with event e .

The preference status of e is changed from non-instantiated preference to instantiated preference.

6.3.1 Representing Temporal preference modifiers

If “Lisa prefers John to pick her up *early*” instead of by 7:20 pm, we need to define a formal preference function for *early*. Consider common temporal preference modifiers such as *early*, *late*, *about k* , there is a certain property of each temporal modifiers. Figure 6.1 shows graphical preference functions of early, late and about k where $X = \{x_1, \dots, x_n\}$ and x_i is a set of possible starting/ending time points of event X and $x_i < x_{i+1}$. These general preference functions can use to convert many temporal preference modifiers.

6.4 CCTCSPP Example

Consider the following additional preference information to our temporal example (Figure 5.1).

(a) Early Preference, $P_f : X \rightarrow A$ and $P_f(x_i) > P_f(x_j), i < j$

(b) Late Preference, $P_f : X \rightarrow A$ and $P_f(x_i) < P_f(x_j), i < j$

(c) About(k) Preference, $P_f : X \rightarrow A$ where $P_f(x_k) = P_f(x_{k-1}) = P_f(x_{k+1}) = 1$, if $i+1 < k-1$ then $P_f(x_i) < P_f(x_{i+1})$ and if $i+1 > k+1$ then $P_f(x_i) > P_f(x_{i+1})$

Figure 6.1: Qualitative Temporal Preference Functions

Lisa prefers John to pick her up by 7:20 (numeric preference). They prefer to arrive at the theater before the movie starts a little bit to get good seats (symbolic preference). Lisa prefers to watch *movie*₁ to *movie*₂ and *movie*₃ (composite temporal preference). Whereas, Mike prefers *movie*₃ to *movie*₂ and *movie*₁(composite preference). Whoever gets there first, will pick the movie that he/she likes (conditional preference on a composite variable).

The SNTC corresponding to “*Lisa prefers John to pick her up by 7:20*” is $f_{n:JPL}$ shown in figure 6.2.

Figure 6.2: The Soft Numeric Temporal Preference Function of Event *JPL*

The soft symbolic temporal preference “*They prefer to arrive at the theater before the movie starts a little bit to get good seats*” puts favor on *before* relation. If $C_{M,WM}$ is a binary constraint between event M and event MW , then the degree of $f_{s:C_{M,WM}}(B)$ is higher than the degree of $f_{s:C_{M,WM}}(M)$. Here, we set $f_{s:C_{M,WM}}(B) = 1.0$ and $f_{s:C_{M,WM}}(M) = 0.8$. These preference degrees are also set to other SSTCs, $f_{s:C_{JL,WM}}$ and $f_{s:C_{JLS,WM}}$. When a new variable is added in to the current partial solution, corresponding ALSPs are calculated. If $JPL = (10, 25)$ and $JL = (30, 45)$, then $A_{R_{JPL,JL}} = B$ and $ALSP(B) = 0.8$.

Figure 6.3 shows integrating the four types of preferences to the temporal problem.

Figure 6.3: Example 5.1 with Four Types of Preferences

6.5 Solving CCCSPPs and CCTCSPPs

In general, searching for a preferred solution of CCCSPPs and CCTCSPPs is an optimization problem where an objective function is defined to evaluate the best feasible solution. An objective function of CCCSPPs and CCTCSPPs is the maximum values of the global and the configuration preference degrees. The feasible solution that holds the highest global and configuration preference degrees is the best solution. Searching for a feasible solution and evaluating its preference satisfaction can be in parallel. In the BT algorithm, when a partial solution is extended with a new variable, the objective function is applied to evaluate the partial solution. In addition, an objective estimation function is applied forward to future variables in order to evaluate the maximum bound of the future extended partial solution. The estimation function will guide the search to find the best solution by skipping checking some search space.

Branch and Bound (BnB) is a well know method for solving optimization problems [86]. Our complete methods are extended by the BnB technique to discard examination on some sub search spaces where bounds of the subspaces are lower than a min bound. The lower bound of searching a CCCSPP is set by the global and the configuration preference degrees of the first found solution. Then the algorithm will backtrack to look for a better solution. The additional steps of integrating our complete solving methods with the BnB technique are described in Figure 6.4.

We integrate the BnB technique with a heuristic function to the complete methods (called BnBFC, BnBFC⁺, BnBMAC and BnBMAC⁺.) Every time a new variable is considered into the current partial solution, the algorithm will apply the heuristic function to estimate (if a value assigned to the variable):

1. the possible future maximum global preference via the simple variable preference function,


```

(1) Search for the first solution:
  1.1 do
    for each current variable,  $V_i$ 
      1.2 If the domain of  $V_i$  is unsorted
        1.2.1 For each value,  $a_i$ , in the domain of the current variable
          checks the consistent of the value
          (constraint propagation is applied here i.e., FC, FC*, MAC and MAC*)
          if  $a_i$  is inconsistent then
            1.2.1.1 discard  $a_i$  from the current domain
          else
            1.2.1.2 estimates the bounds of the future subspace if  $a_i$  is selected
          end if
        End for
        1.2.2 sort the remaining values in the domain of the current variable
        by an decreasing order of the estimated bounds
      End if
    1.3 If the domain of the current variable is not empty
      1.3.1 assigns the next value in the sorted domain to the current variable
      1.3.2 checks activity constraints and add new variables to the list of active variables
    else
      1.3.3 restores the domain of  $V_{i-1}$ , ( $V_i$  is considered as an unsorted variable)
      1.3.4 remove all conditional variables activated by this variables
      1.3.5 backtrack to the previous variable,  $V_{i-1}$ 
    end if
  until all active variables are assigned or cannot find a solution

  1.4 If the first solution,  $S$ , is found
    1.4.1 set lower bounds for the global and the configuration preference degrees
    1.4.2 go to step (2) to search for a better solution
  else
    1.4.3 the problem is inconsistent
  End if

```

Note: The backtracking steps in (2) are very similar to searching for the first solution. In contrast, the BJ technique is applied here to jump back to the source variable where its assignment causes the lowest preference degree. In addition, the lower bound is used to compare which a new partial assignment in order to skip unnecessarily search. The extra steps are described in (2).

```

(2) Search for a better solution by backtrack search:
  2.1 finds the farrest variable,  $V_i$  having the lowest preference degree on its assignment
  2.2 backtracks to the previous variable before  $V_i$ 
    (all the variables in the between the current variable and the farrest variable
    are restored to the previous state as unsorted variables)
  2.3 if the estimated bounds of the next value in  $V_{i-1}$  is greater than the current lower bounds
    2.3.1 assigns the next value to  $V_{i-1}$ 
    2.3.2 extends the partial solution to the next variable,  $V_{i+1}$ 
    (similar to steps 1.2 and 1.3, in addition, discards  $a_i$  where the estimation bounds
    are less than the lower bounds)
  else
    2.3.3 discards this branch
    2.3.4 backtracks to the next farrest variable
  end if
  2.4 If an improve solution,  $S'$ , is found
    2.4.1 set new lower bounds for the global and the configuration preference degrees
    2.4.2  $S = S'$ 
    2.4.3 go to step (2) to search for another improve solution
  else
    2.4.3  $S$  is the best solution
  end if

```

Figure 6.4: BnBCCCSP Algorithm

2. the possible future maximum configuration preference via the composite variable preference function and the activity preference function,
3. the new adjusted local preference degrees via the constraint preference function and satisfied/unsatisfied preference functions.

Only a consistent value leading to a better solution is assigned to the current variable. The domain of the current variable is then sorted in decreasing order by the heuristic function and compared to the lower bound. The heuristic function (step 1.2.1.2 in Figure 6.4) for the current variable, $h(V_i, a_i)$, is calculated from sub three functions:

- $h_a(a_i)$ is a function to estimate the possible maximum global preference level from all unassigned future variables. For each domain of the future simple variables including the simple variables in the domains of composite variables, the function finds the maximum preference degree then the function returns the minimum of the maximum preference degrees of all the future variables.
- $h_b(a_i)$ is a function to estimate the configuration preference degree. For each future composite variable, the function finds the maximum composite preference degree. In addition, if the assignment satisfies an activity constraint, the activity preference degree are involved. The function then returns the minimum degree of the maximum degrees of the composite preference functions and the involved activity preference functions.
- $h_c(a_i)$ is a function to recalculate the global preference degree of the extending partial assignment. The function finds the new associated compatible constraint preference degrees and satisfied/unsatisfied preference functions, then the function returns the minimum degree of the degrees.

The domain of V_i is sorted decreasingly by $h_b(v_i)$ and $\min(h_a(a_i), h_c(a_i))$ where the current

Heuristic Calculation					Assignment		
[1] Occasion	Interview	Party	Sport	Camping	Interview	1.0	1.0
h_a	1.0	1.0	1.0	1.0			
h_b	1.0	1.0	1.0	1.0			
h_c	1.0	1.0	1.0	1.0			
[2] Apparel	2-Pieces	Set			Set	1.0	1.0
h_a	1.0	1.0					
h_b	0.8	1.0					
h_c	1.0	1.0					
[3] Set	Skirt-Suit	Pant-Suit	Dress		Skirt-Suit	1.0	1.0
h_a	1.0	1.0	1.0				
h_b	1.0	1.0	0.8				
h_c	1.0	1.0	1.0				
[4] Shoes	Casual	Boot	Sandal	Running	Pump	Boot	1.0 0.9 (1 st solution Occasion=Interview, Apparel=Set, Set=Skirt-Suit and Shoes=Boot, then backtrack to Set)
h_a	1.0	0.9	0.8	0.7	0.6		
h_b	1.0	1.0	1.0	1.0	1.0		
h_c	0.0	1.0	0.0	0.0	0.8		
[5] Set	Skirt-Suit	Pant-Suit	Dress		Pant-Suit	1.0	1.0
h_a	1.0	1.0	1.0				
h_b	1.0	0.8	1.0				
h_c	1.0	1.0	1.0				
[6] Shoes	Casual	Boot	Sandal	Running	Pump	Casual	1.0 1.0 (2 nd solution, the best solution Occasion=Interview, Apparel=Set, Set=Pant-Suit and Shoes=Casual)
h_a	1.0	0.9	0.8	0.7	0.6		
h_b	1.0	1.0	1.0	1.0	1.0		
h_c	1.0	0.0	0.0	0.0	0.8		

Figure 6.5: Steps in Solving the CCCSPP example in Section 6.2 using BnB Algorithm and Heuristic Function

variable is V_i and assigned by a_i . Figure 6.5 shows steps of searching for the best solution of the dress up game example with preference in Section 6.2.

The best of the best solutions

The best of the best solutions problem arises when there are more than one optimal solutions. The question here is which one is actually the real best. Two complete solutions may give the same global preference degree. For example, S_1 and S_2 are the optimal solutions for a CCTCSPP, P . The assignment of S_1 is $\{e_1 = (1, 5)_{0.5}, e_2 = (5, 10)_{0.9}, e_3 = (2, 8)_1\}$ (the subscripts indicate local preference degrees of the assigned value). The assignment of S_2 is $\{e_1 = (2, 6)_{0.5}, e_2 = (6, 11)_{0.6}, e_3 = (1, 7)_{0.5}\}$. The min operation grounds all the local preference degrees to the lowest local preference to represent its global solution. In this sense, comparing the quality of optimal solutions on the top of $\min$ operation is required. This comparison can be

done by using an additional objective function, mul , to compute the best value of multiplication of local preference degrees. $mul(S_1) = 0.5 \times 0.9 \times 1 = 0.45$ and $mul(S_2) = 0.5 \times 0.6 \times 0.5 = 0.150$. So S_1 is better than S_2 .

6.6 Conclusions

We have extended the CCCSP and the CCTCSP frameworks with the c-semiring to handle preferences. The different preference levels are defined over the domains of simple variables, the domain of composite variables, the compatible constraints, and the activity constraints. The min operation is used to aggregate the global preference and the configuration degrees of a complete solution. The max operation is applied to compare the all complete solution in order to obtain the optimal solution. The Branch and Bound technique is applied to the complete solving methods to skip some space examinations by using the lower bound strategy. The domain of the current variable is dynamically sorted in decreasing order by the heuristic function where the global and the configuration preference degrees are estimated. When the first solution is found, the lower bound is then obtained. The backjumping technique is applied to jump back to find a better solution.

7. EXPERIMENTATION

In this chapter, we report and analyze the experimental results of the tests conducted to compare the efficiency of the complete and the incomplete algorithms. We have implemented a random generator to create CCCSP/CCTCSP instances (including those with preferences) for our testing. An advantage of the random generator is producing restricted instances under different parameter settings. In general, the difficulty of CSP instances are classified into two groups (easy and hard) according to the density of constraints, the tightness and the number of consistent solutions (see Figure 7.1). The density of constraints is the proportion of constraint that actually exist between any pair of variables. The tightness T_{ij} of an arc ij is given as the ratio of the number of all constraints on the arc ij to the number of all compound-labels on the arc ij . The tightness of a problem is the average value of T_{ij} over the arcs.

The easy group represents all problems of the following types:

- Under constrained problems that have few constraints where the number of consistent solutions is high (as shown on the left side of the graph in Figure 7.1). For such problems, performing AC in the preprocessing stage does not result in a tighter problem.
- Over constrained problems that have many constraints where there is no solution (as shown on the right side of the graph in Figure). For such problems, performing AC in the preprocessing stage results in a tight problem where inconsistency can easily be detected.

In contrast, the hard group represents all problems of the following types:

- Medium constrained problems that have medium number constraints where the number of solutions is small. Performing AC in the preprocessing stage does not result in a tighter problem.

- Close to the phase transition point (in the region from where almost all problems are satisfiable to a region where almost all problems are unsatisfiable) and very hard to find a solution or to detect inconsistency. The phase transition point of a set of random generated CSPs is indicated by the critical tightness value where the random generated CSPs change from satisfiable to unsatisfiable.

CCCSP instances are classified, in the same manner. Moreover, we consider dynamic factors caused by the density of activity constraints (*act*), the percentage of initial variables (*iv*) and the number of composite variables (*cn*). CCCSPs have more dynamic structures when *act* and *cn* are high and *iv* is low. This means the CCCSP instance contains a large set of possible CSPs. The set of generated CSPs has various structures in the newly activated variables. In other words, CCCSPs have less dynamic structures when *act* and *cn* are low and *iv* is high. The set of generated CSPs from the less dynamic structure is small and most of them have the same set of variables in the solutions. Only a few variables are different.

Figure 7.1: Easy Versus Hard Problems

7.1 CCCSP Instances

In order to test different strategies for the complete methods and incomplete methods for solving CCCSPs, a random CCCSP generator was implemented. Basically, CSP randomly generated instances can be generated by four standard models: A , B , C and D [101, 48]. However, these models can easily generate flawed random CSP instances [2]. Model E [2] and a generalized model [71] have been proposed to avoid trivial insoluble problems. Nevertheless, they are not practical [119]. Model RB (revision of standard model B) as proposed by Xu and Li [120], stands out from other models as the best. It has the following features: it is easy to implement, it has a predictable phase transition, and it generates asymptotically hard CSPs. Moreover, model RB has been used to generate CSP benchmarks in the CSP Solver Competition 2005 and 2006 [118].

Definition 7.1.1. *Model RB randomly selects with repetition $m = rn \ln n$ random constraints. Each random constraint is formed by selecting without repetition 2 of n variables. For each constraint we uniformly select without repetition, $q = td^2$ incompatible tuples of values. A random CSP instance is generated by the parameters: $CSP\langle n, r, \alpha, t \rangle$, where n is the number of variables, $r > 0$ is constant, α and $t \in [0, 1]$ are constant. A phase transition point of model RB computes from equation the $p = 1 - e^{-\alpha/r}$.*

In a similar way, to avoid the problem of generating flawed random CCCSP instances, our CCCSP generator is based on the randomly generated CSP instances of model RB. Hence, a random CCCSP is generated by the parameters: $CCCSP\langle n, \alpha, r, t, cn, cd, iv, act \rangle$ where:

- n = number of variables,
- α = a positive real constant number, where $D = n^\alpha$ is the domain size of each variable,
- r = a positive real constant number, where $m = rn \ln n$ is the number of constraints,
- t = tightness, a constant and $t \in R$, and $q = td^2$ is the number of
incompatible pairs per each constraint,
- cn = number of composite variables,
- cd = domain size of composite variables,
- iv = fraction of initial variables from the set of $N = n + cn$. $|IV| = iv \times N$,
where IV is the set of initial variables, and
- act = density of activity constraints, activating a non active variable
from the set of $|IV| \times d$.

Steps to Generate a CCCSP $\langle n, \alpha, r, t, cn, cd, iv, act \rangle$

1. Randomly generate a CSP $\langle n, \alpha, r, t \rangle$ based on the model RB:
 - (a) Randomly select $m = rn \ln n$ binary constraints.
 - (b) For each constraint (X, Y) randomly generate $q = td^2$ incompatible pairs of X and Y .
2. Randomly generate cn composite variables and $cn \times cd$ simple variables, group them into cn groups and then assigns each group to each composite variable domain.
3. Randomly generate and adds $m' = r \times N \times \ln N - m$ binary constraints between each domain (variables) of composite variable and the remaining variables, where $N = n + cn * cd$ is the total number of simple variables. By adding the m' binary constraints, the binary constraint density of generated CCCSP respects model RB, if all composite variables are

active.

4. Randomly select a set of $iv \times (n + cn)$ initial variables, IV .
5. For each non-selected variable, NIV_j , from (4), randomly generate $act \times (|IV| \times D)$ activity constraints, $IV_i = a_i \rightarrow NIV_j$, where $IV_i \in IV$. Then for next $NIV_k, IV = IV \cup \{NIV_k\}$.

Phase Transition

A phase transition point of model RB can be calculated from the equation $p_c = 1 - e^{-\alpha/\tau}$. The formula is applied here only to estimate a range of phase transitions of the generated CCCSP instances since a CCCSP instance consists of many CSP instances. We compute a range of possible generated CSPs under these two assumptions:

1. From the set of initial variables IV , there is no activated variables are added into the solution. This CSP represents the smallest possible solution generated from the CCCSP instance. A phase transition calculated from this CSP represents the lower bound of a phase transition of the CCCSP instance.
2. All non-active variables are added into the solution. This CSP represents the largest possible solution generated from the CCCSP instance. A phase transition calculated from this CSP represents the upper bound of a phase transition of the CCCSP instance.

Densities of constraints under the above two assumptions are calculated immediately after each CCCSP instance is generated. When n and m are known, the actual values of α and τ can be calculated as follows.

$$\begin{aligned}\alpha &= \frac{\ln d}{\ln n} \\ \tau &= \frac{p(n-1)}{2 \ln n}, \text{ where } p = \frac{m}{N \times (N-1)} \text{ is the density of constraints, } N = n + (cn \times cd)\end{aligned}$$

For example, in an experiment we generated a set of CCCSP instances where $n=20$, $\alpha=0.7686$, $r=1.5856$, $cn=5$, $cd=3$, $iv=0.8$ and $0.1 \leq t \leq 0.9$ (which corresponds to $d=10$ and $m=95$). The generator gave a CCCSP instance that contained the smallest possible CSP $\langle n=20, d=10, p=0.6 \rangle$ and the largest possible CSP $\langle n=25, d=10, p=0.37 \rangle$. The phase transition of CSP $\langle n=20, d=10, p=0.6 \rangle$ is at 0.33 ($\alpha = \frac{\ln 10}{\ln 20} = 0.768$, $r = \frac{0.6(20-1)}{2 \ln 20} = 1.902$, $p_c = 1 - e^{-0.768/1.902} = 0.33$) and the phase transition of CSP $\langle n=25, d=10, p=0.37 \rangle$ is at 0.4 ($\alpha = \frac{\ln 10}{\ln 25} = 0.715$, $r = \frac{0.37(25-1)}{2 \ln 25} = 1.379$, $p_c = 1 - e^{-0.715/1.379} = 0.4$). By the calculation, we can estimate that the set of generated CCCSPs is hard to solve when $0.33 \leq t \leq 0.4$.

Because the generator randomly chooses a set of initial variables after randomly generating constraints, it may affect the density of the smallest possible CSP if the set of constraints is not well distributed. We audited the results from the generator. Table 7.1 demonstrates the density of eight sets of randomly generated initial CCCSP instances for samples generated by the following parameters: $n=50$, $\alpha=0.6$, $r=0.6$, $cn=5$, $cd=3$ and $p=0.2$, while varying iv from 0.2 to 0.9 by 0.1 increments. In each set, ten sample CCCSP instances were generated and the average number of constraints in them were calculated. In our testing, the randomly generated CCCSP instances appropriately maintain the constraint density property between *min* and *max* when $0.6 \leq iv \leq 0.9$. When iv is less than 0.6, the average density of the constraints of the generated CCCSP instances is below the *min* and when iv is close to 0.2 the average density of constraints is significantly offset. Since the initial generated CSP instances have $D=22$, $m=117$ and $IV=44$. If we assume that none of the 5 composite variables are involved in the initial set of variables, the initial instance will have 100 constraints. If one composite variable exists, the number of constraints is 105, and so on until with 5 composite variables, the number of constraints is 129. Thus, for $iv=0.8$, the minimum number of constraints (*min*) is 100 and the maximum number of constraints (*max*) is 129.

Table 7.1: Constraint Density of Random Generated CCCSP Instances

iv	CCCSP instance sample										Number of Constraints	
	1	2	3	4	5	6	7	8	9	10	Sample AVG	min-max
0.9	125	132	137	122	128	127	125	130	130	122	127.8	115-145
0.8	123	110	123	122	125	114	129	122	131	123	122.2	100-129
0.7	93	98	89	107	85	87	96	104	101	83	94.3	84-113
0.6	64	54	60	102	65	78	73	79	70	78	72.3	69-97
0.5	48	47	49	42	43	39	37	35	57	34	43.1	54-82
0.4	40	30	29	27	25	30	31	29	42	23	30.6	45-66
0.3	16	16	24	15	13	14	23	12	20	20	17.3	32-52
0.2	9	3	7	9	5	5	9	6	9	20	8.2	20-38

7.2 CCTCSP Instances

Unlike CCCSPs, CCTCSPs describe constraints in terms of disjunctives of Allen's relations. Generally, the tightness of a TCSP is defined by the number of Allen's relations. In order to adopt model RB, we need to define the tightness of a CCTCSP.

Definition 7.2.1. *The tightness of a CCTCSP is the average ratio of the number of compatible interval pairs satisfying a disjunctive Allen's relation, R_{xy} , to $|SOPO_x| \times |SOPO_y|$, where $|SOPO_x|$ and $|SOPO_y|$ are the numbers of intervals generated from $SOPO_x$ and $SOPO_y$, and $SOPO_x$ and $SOPO_y$ are any pair of SOPOs in CCTCSP. R_{xy} is a disjunctive Allen's relation between two events X and Y .*

All generating steps of the CCTCSP generator are the same as in the CCCSP generator shown in Section 7.1, except the step that generates SOPOs and the Allen's relations between a pair of SOPOs. The CCTCSP generator will randomly generate 2 SOPOs for variables X and Y where:

- Each $SOPO$ contains d intervals and there is a set of disjunctive Allen's relations, $RELS_{xy}$, in which the number of compatible interval pairs of X and Y respects the tightness q .
- To increase the freedom in choosing $SOPO_x$, $SOPO_y$ and $RELS_{xy}$, the tightness q is

allowed to adjust by $\pm e$, where e is a real constant that is less than $0.05 \times D^2$, where D is the domain size of the problem.

- After $SOPO_x$ and $SOPO_y$ are randomly generated, we count the number of compatible pairs for each Allen's Relation. The dynamic bin packing technique [67] is applied here to find a set of disjunctive Allen's relation, $RELS_{xy}$, such that the tightness is in the range $q \pm e$.
- If there is such $RELS$, then it is assigned to constraint R_{xy} .
- If not, then $SOPO_x, SOPO_y$ and $RELS_{xy}$ will be regenerated until it reaches tightness q .

Controlling a randomly generated CCTCSP at a certain degree of tightness q requires more checking than conducting Allen's relation tightness. Consequently at an inappropriate parameter setting, the generator may fail to return an output, such as when the setting horizon¹ is too small, but the given d is too large or the setting tightness is low, but D is large. In a successful application, the CCTCSP generator guarantees the production of hard CCTCSP instances because it is framed under model RB. Moreover, an estimate of the range of the phase transition is computable. A phase transition point of model RB can be computed from equation $p = 1 - e^{-\alpha/r}$. On the other hand, in CCTCSP, the phase transition formula is revised to $e^{-\alpha/r}$.

7.3 Experimental Environment

The experiments were conducted using the following environment.

- System configuration: Sun Microsystems sun4u Sun Fire V240

¹A parameter of the astar generator [74] need to generate a TCSP instance. The horizon value indicates the maximum length of the time line for the generated temporal scenario.

- System clock frequency: 167 MHz
- RAM size: 8GB

All algorithms and generators were coded in C⁺⁺. Some parts of the generators are modified versions of random uniform CSP generators of Frost et al. [41] and Mouhoub’s astar TCSP generator [74]. Because CCCSPs and CCCTCSPs have more unique features than CSPs and TCSPs, reusing the built-in CSP/TCSP libraries was unsuitable due to their inflexibility. Therefore we implemented the FC, FC⁺, MAC and MAX⁺ algorithms using C⁺⁺ in the CCCSP solver. In our preliminary tests, we let the CCCSP solver ran to observe the overall efficiency. We set the time limit for each algorithm to search for a solution at 2 hours. We generated different sizes (number of variables) of CCCSP instances to study an appropriate size of instances that our solver can handle. $n=50$ is the maximum size that our solver can handle (for some algorithms). In the CSP solver competition 2005 [120], the results also demonstrated that among the 16 competition solvers, there were only four solvers able to solve CSPs where $n=50$. None of the solvers could reliably solve CSP instances where $n > 50$ within 10 minutes. Since solving a CCCSP is solving a set of CSPs, in our experimentation we set the time limit up to 2 hours that at least one of the complete algorithms can find a solution in a group of hard generated problems.

7.4 Test Results for Complete Methods

In this section, we report the comparative results of applying the complete methods (FCC-CCSP, FC⁺CCCSP, MACCCCS and MAC⁺CCCSP, abbreviated to FC, FC⁺, MAC and MAC⁺, respectively) to randomly generated CCTCSP instances.

7.4.1 Varying the Density of Activity Constraints, act

In Test 7.4.1 described in this section, CCCSPs/CCCTCSPs have highly dynamic structures when the density of activity constraints increases. Highly dynamic instances involve greater numbers of checking activity constraints, adding and removing variables. We generated the test instances set using $n=50$, $\alpha=0.8$, $r=0.6$, $cn=5$, $cd=3$ and $iv=0.8$. The tightness and the density of activity constraints were varied as follows: $t=0.25, 0.35, 0.45, 0.55, 0.65$ and 0.75 and $act=0.02, 0.04, 0.06, 0.08, 0.1$ and 0.3 . From the generated instances, we obtained the following observations:

- The generator generates the set of CCTCSP instances where the range of the phase transition is between $t=0.35$ and $t=0.45$. The corresponding results obtained from model RB calculation are as follows:
 - The smallest generated CSP, with $n = 44$, $d = 22$, $\alpha = \frac{\ln 22}{\ln 44} = 0.8$, $p = 0.13$, $r = 0.13 \times \frac{43}{2 \times \ln 44} = 0.73$, phase transition $= e^{\frac{-\alpha}{r}} = 2.71^{\frac{-0.8}{0.73}} = 0.335$.
 - The largest generated CSP, with $n = 65$, $d = 22$, $\alpha = \frac{\ln 22}{\ln 65} = 0.74$, $p = 0.12$, $r = 0.12 \times \frac{64}{2 \times \ln 65} = 0.91$, phase transition $= e^{\frac{-\alpha}{r}} = 2.71^{\frac{-0.74}{0.91}} = 0.44$.

The generated CCTCSP instances are hard to solve when $0.35 \leq t \leq 0.45$, which is in the range of the estimate phase transition when $0.335 \leq t \leq 0.44$.

- The CCTCSP instances generated at $t \leq 0.25$ are mostly inconsistent and easily detected by the AC3-CCCSP function in the preprocessing stage. The average actual time needed to detect the inconsistent problems is less than 1 ms. (The results for $t=0.25$ are omitted from Figure 7.2.)
- The CCTCSP instances generated at $t=0.35$ are mostly inconsistent and hard to solve.

FC and FC⁺ cannot solve problems in this this group. MAC and MAC⁺ can only solve some problems.

- The CCTCSP instances generated at $t=0.45$ are mostly inconsistent and hard to solve. FC and FC⁺ cannot solve the instances in this group, except when $act=0.06$. However, MAC is able to solve some of the problems. MAC⁺ can solve all the problems and it is faster than MAC.

Figures 7.2 and 7.3 illustrate comparison graphs of the results. For the purpose of clarifying the comparison, the X symbol in the table is used to represent any unsolvable problem for which the algorithm could not return a result within two hours.

The comparison results demonstrate the efficiency of each algorithm as follows.

- For the hard and highly dynamic instances (where $0.35 \leq t \leq 0.45$ which are around the phase transition, Figures 7.2(b) and 7.2(a)), MAC⁺ outperforms the other algorithms. In most cases, FC and FC⁺ cannot return any solution. When $t=0.45$, MAC returns solutions except when $act=0.08$, but it is much less efficient than MAC⁺ when $act=0.06$ and 0.1.
- For the easy and highly dynamic instances (Figures 7.2(d) and 7.2(e), where $act=0.3$), MAC⁺ is the fastest algorithm.
- For the easy and lowly dynamic instances (Figures 7.2(d) and 7.2(e), where $0.04 \leq act \leq 0.06$), FC and FC⁺ are the most efficient algorithms.
- For the easy instances, all algorithms can easily find a solution when $t=0.65$ and $t=0.75$. However FC and FC⁺ outperform MAC and MAC⁺.

(a) $t=0.35$

(b) $t=0.45$

(c) $t=0.55$

(d) $t=0.65$

(e) $t=0.75$

Figure 7.2: The Comparison of FC, FC⁺, MAC and MAC⁺ when Varying the Density of Activity Constraints (Test 7.4.1)

- MAC^+ can solve all the generated instances when $0.45 < t < 0.75$ which is not in the case for FC, FC^+ and MAC.

(a) $act=0.06$

(b) $act=0.08$

(c) $act=0.1$

(d) $act=0.3$

Figure 7.3: The Inverse Graphs of Figure 7.2 where the X-axis is the Density of Activity Constraints

7.4.2 Varying the Number of Initial Variables, iv

In Test 7.4.2, described in this section, the percentage of initial variables, iv , is varied. A CCCSP problem is more dynamic when iv is small and act is large. According to the results from Test 7.4.1 (Figure 7.3), most algorithms are able to return a solution at $act=0.06$. Therefore, this parameter setting is used in this test. We generated instances with $n=50$, $\alpha=0.8$, $r=0.6$, $cn=5$, $cd=3$ and $act=0.06$. We varied t from 0.2 to 0.7 and iv from 0.2 to 0.9. The formula adopted from model RB cannot be applied to estimate the range of the phase transition because the test varies the number of initial variables. In other words, the smallest size of the possible CPS in the generated instance changes as iv changes. We obtained the following results (Figure 7.4):

- When $t < 0.7$, a hard problem is generated. Especially at $t=0.5$, most generated problems are very hard to solve (Figure 7.4(d)).
- When the value of iv is small (Figure 7.4(a), where iv less than 0.6), it makes problems at $t=0.2$ (easy to solve) hard, exception when $iv \geq 0.6$.
- FC, FC⁺, MAC and MAC⁺ cannot find a solution when $t=0.5$ and $iv=0.3, 0.5, 0.6$ or 0.8 (Figure 7.4(d)). The CCCSPs are close to the phase transition, and any iv in this range makes the problems more dynamic.
- MAC⁺ outperforms the other algorithms when the problem is highly dynamic and falls in the range of the phase transition.

(a) $t=0.2$

(b) $t=0.3$

(c) $t=0.4$

(d) $t=0.5$

(e) $t=0.6$

(f) $t=0.7$

Figure 7.4: The Comparison of FC, FC⁺, MAC and MAC⁺ where $0.2 \leq iv \leq 0.9$ and $0.2 \leq t \leq 0.7$ (Test 7.4.2)

7.4.3 Varying the Number of Composite Variables, cn

In this test, we vary the number of composite variables. The instances have highly dynamic structures when the number of composite variables is high. The randomly instances are generated with $n=50$, $\alpha=0.8$, $r=0.6$, $iv=0.8$, $cd=3$, $act=0.06$, $0.4 \leq t \leq 0.8$ and $cn=5, 10$ and 15 . The results are shown in Figure 7.5. MAC^+ still retains its outstanding performance for hard and highly dynamic instances. Moreover, MAC and MAC^+ are reliable algorithms because they can solve all instances. For the easy and less dynamic instances (Figures 7.5(c) and 7.5(d) where $cn=5$), FC and FC^+ are more efficient than MAC and MAC^+ .

7.4.4 Varying r

In generating a CCCSP/CCTCSP instance, the density of binary constraints is restricted by parameters r and n . Here, r is a constant and $r > 0$, where $m=rn \ln n$ is the number of binary constraints. In this test, the dynamic structure of the instances is fixed (i.e., $act=0.06$), but we vary the density of binary constraints by adjusting r . The instances are generated with $n=50$, $\alpha=0.8$, $iv=0.8$, $cn=5$, $cd=3$, $0.4 \leq t \leq 0.8$ and $0.2 \leq r \leq 1.0$. Figure 7.6 shows the comparison result. We obtained the following outcomes.

- MAC^+ is the most reliable algorithm because it can solve all the less, medium and highly constrained instances.
- For the easy and highly constrained instances (Figure 7.6(f)), MAC outperforms the other algorithms.
- For the hard and medium constrained instances (Figure 7.6(c)), MAC^+ outperforms the other algorithms.

(a) $t=0.4$

(b) $t=0.5$

(c) $t=0.6$

(d) $t=0.7$

(e) $t=0.8$

Figure 7.5: The Comparison of FC, FC⁺, MAC and MAC⁺ where $cn = 5, 10, 15$ and $0.4 < t < 0.8$ (Test 7.4.3)

- FC and FC⁺ are inefficient for medium constrained instances but outperform the other algorithms for easy and less constrained instances.

In summary, MAC⁺ is the most efficient complete algorithm for the hard and highly dynamic CCCSPs. MAC is also intended for the hard CCCSPs, but it outperforms MAC⁺ when the CCCSPs have low dynamic structures. For easy and less dynamic problems, FC and FC⁺ are the most efficient algorithms. Table 7.2 illustrates the best performing algorithm by problem characteristics.

Table 7.2: The Summary of Complete Methods

Problem	Dynamic Structure	
	Low	High
Hard	MAC	MAC+
Easy	FC	FC+

7.5 The Results for Incomplete Methods

In section 4.2.2, we proposed the static, dynamic and incremental strategies in incomplete methods for solving CCCSPs and CCTCSPs. We have implemented these strategies for MCRW, SDRW and TS as follows:

1. the static strategy for MCRW, SDRW and TS
2. the dynamic strategy for MCRW and SDRW, and
3. the incremental strategy for MCRW and SDRW.

In preliminary tests, several runs were performed and the results were observed in order to find appropriate parameter settings for the algorithms. The resulting parameter settings are as follows:

(a) $t=0.2$

(b) $t=0.3$

(c) $t=0.4$

(d) $t=0.5$

(e) $t=0.6$

(f) $t=0.7$

Figure 7.6: The Comparison of FC, FC⁺, MAC and MAC⁺ where $0.2 \leq r \leq 1.0$ and $0.2 \leq t \leq 0.7$ (Test 7.4.4)

- For the static strategy, MAX-CSP is 20 and MAX-MOVES is 50.
- For the dynamic strategy, MAX-MOVES is 1000.
- The probability of noise is 0.03 for the MCRW and the SDRW algorithms.
- The size of tabu list is 20 for the TS algorithm.

In the first part, the set of CCTCSP instances in Test 7.4.1 and 7.4.2 are used for the incomplete algorithms to show the speed and the quality performance comparisons (Figure 7.7 to 7.13). The quality of solutions is measured by the number of non violated constraints over the total number of constraints. We also include the results of solving inconsistent instances where $t=0.25$, $t=0.35$ and very easy instances where $t=0.85$. We obtained the following outcomes:

- The static strategy ($MCRW_{Static}$, $SDRW_{Static}$ and TS (note recorded as TABU in test results)) may fail to solve the instances, due to the inability to find a consistent CSP/TCSP from a given CCCSP/CCTCSP. However, if the problem contains many consistent CSPs (easy problems), the static strategy has the best performance in terms of the solution quality and time execution.
- Overall, the dynamic strategy ($MCRW_{Dynamic}$ and $SDRW_{Dynamic}$) returns better quality solutions than the static strategy.
- The quality of solutions returned by the incremental strategy ($MCRW_{Incremental}$ and $SDRW_{Incremental}$) is better than those from the static strategy but not as good as those from the dynamic strategy.
- The MCRW algorithm runs faster than the SDRW and TS algorithms.
- The $SDRW_{Dynamic}$ algorithm returns higher quality solutions than other algorithms for the hard and highly dynamic instances.

(a) quality, $t=0.25$

(b) speed, $t=0.25$

Figure 7.7: Results of Incomplete Methods (Test 7.4.1: $t=0.25$)

(a) quality, $t=0.35$

(b) speed, $t=0.35$

Figure 7.8: Results of Incomplete Methods (Test 7.4.1: $t=0.35$)

(a) quality, $t=0.45$

(b) speed, $t=0.45$

Figure 7.9: Results of Incomplete Methods (Test 7.4.1: $t=0.45$)

(a) quality, $t=0.55$

(b) speed, $t=0.55$

Figure 7.10: Results of Incomplete Methods (Test 7.4.1: $t=0.55$)

(a) quality, $t=0.65$

(b) speed, $t=0.65$

Figure 7.11: Results of Incomplete Methods (Test 7.4.1: $t=0.65$)

(a) quality, $t=0.75$

(b) speed, $t=0.75$

Figure 7.12: Results of Incomplete Methods (Test 7.4.1: $t=0.75$)

(a) quality, $t=0.85$

(b) speed, $t=0.85$

Figure 7.13: Results of Incomplete Methods (Test 7.4.1: $t=0.85$)

In the second part, we generated a new set of CCCSP instances that are unsolvable by the complete methods. Figure 7.14 to Figure 7.17 show the speed and the quality of solutions returned from $MCRW_{Static}$, $MCRW_{Dynamic}$, $SDRW_{Static}$ and $SDRW_{Dynamic}$ the incomplete methods. The quality of solutions indicates that about 95% of constraints are satisfied.

(a) quality, $t=0.5$

(b) speed, $t=0.5$

Figure 7.14: Results of Incomplete Methods (Test 7.4.2: $t=0.5$)

(a) quality, $t=0.6$

(b) speed, $t=0.6$

Figure 7.15: Results of Incomplete Methods (Test 7.4.2: $t=0.6$)

(a) quality, $t=0.7$

(b) speed, $t=0.7$

Figure 7.16: Results of Incomplete Methods (Test 7.4.2: $t=0.7$)

(a) quality, $t=0.8$

(b) speed, $t=0.8$

Figure 7.17: Results of Incomplete Methods (Test 7.4.2: $t=0.8$)

7.6 Result of Various Branch and Bound Methods for CCCSPs and CCTCSPs with Preferences

In order to generate CCCSPP and CCTCSPP instances, we initially generated CCCSP instances. Then preference degrees were randomly added to the instances over the domains of variables, composite variables, constraints and activity constraints. The generator also randomly put some conditions onto the preferences functions. The Branch and Bound (BnB) algorithm (Figure 6.4) was integrated into the four complete constraint propagation algorithms, FC, FC⁺, MAC and MAC⁺. The algorithms stop when the search space is completely explored and return optimal solutions. Figure 7.18 shows the comparison results of BnBFC, BnBFC⁺, BnBMAC and BnBMAC⁺ where the instances are generated with $n=20$, $\alpha=0.6$, $r=0.6$, $0.2 \leq t \leq 0.8$, $cn=5$, $cd=3$, $act=0.06$ and $iv=0.7$. As with the CCCSP experimentation results (Section 7.4 and Table 7.2), the BnBFC and BnBFC⁺ algorithms perform well on easy problems that have many consistent solutions, where $t \leq 0.45$ and $t \geq 0.75$. The BnBFC algorithm outperforms the other algorithm when $t=0.35$ and $t=0.60$. BnBFC⁺ demonstrates outstanding results when $t=0.25$, 0.45 , 0.75 and 0.8 . For problems that are hard to solve and have relatively few consistent solutions, BnBMAC and BnBMAC⁺ are the best algorithms. BnBMAC outperforms the other algorithms when $t=0.50$, 0.55 and 0.70 and BnBMAC⁺ outperforms the other algorithms when $t=0.65$.

The following reasons explain the experimentation results:

- When the numbers of consistent solutions in a CCCSPP is high, combining the objective function with FC is quite accurate. Most future variables contain consistent values. As shown in the CCCSP experimentation results (Section 7.4 and Table 7.2), FC is the most effective algorithm when the problems have low tightness. The BnBFC algorithm is

strengthened by constraint propagation and the appropriate objective function.

- BnBFC⁺ enables more constraint propagation than BnBFC. When the problems have more dynamic behavior, BnBFC⁺ drives the algorithm to find the first solution in the shorter path and to move to the next consistent solution faster than BnBFC.
- In the hard class of problems, the number of consistent solutions in a problem is low and hard to find. Combining the objective function with a MAC technique results in a better performance because it prunes more inconsistent values from future variables that may mislead the objective function. Surprisingly, the performance of BnBMAC⁺ is poorer than BnBMAC in some cases. The explanation of this phenomena is that MAC⁺ does more checking of the future non-activated variables than MAC, but the objective function only estimates the set of active variables. BnBMAC⁺ may find the first solution faster than BnBMAC but it may later suffer more when it tries to find an optimal solution. On highly dynamic problems, BnBMAC⁺ shows outstanding efficiency over BnBMAC.

In conclusion, BnBFC and BnBFC⁺ are good choices for the easy class of CCCSPPs/CCTCSPs. BnBFC is preferred for less dynamic problems with a low density of activity constraints whereas BnBFC⁺ handles average to highly dynamic problems better. For the hard problems in which the consistent solutions are few and hard to find, BnBMAC and BnBMAC⁺ yield outstanding results. BnBMAC⁺ does a better search in the highly dynamic problem environment.

7.7 Conclusions

The characteristics of problems affect on the efficiency of the proposed algorithms. The main factors for classifying CCCSPs, CCTCSPs, CCCSPPs, CCTCSPPs are the tightness, the density of constraints and the dynamism of their structures. The dynamic behavior of the

Figure 7.18: Comparison of BnBFC, BnBFC⁺, BnBMAC and BnBMAC⁺

problems is influenced by the number of initial variables, the percent of activity constraints and the number of composite variables. The higher number of activity constraints and composite variables indicate the higher dynamic behavior of the problems. In addition, if the number of initial variables is low and the density of activity constraints is high the problems have highly dynamic structures.

We implemented a random CCCSP generator based on model RB [120]. The generator generates sets of CCCSP instances where the phase transitions can be estimated. We also augmented the CCCSP generator to generate CCTCSP instances where the binary constraints are the sets of disjunctive Allen's relations and the numeric constraints are SOPOs.

The results of tests performed on four complete methods demonstrate that MAC⁺ is the most efficient algorithm for the hard problems with highly dynamic structure, whereas MAC is also good for the hard problems with less dynamic behavior. In contrast, FC and FC⁺ outperform MAC⁺ and MAC for easy problems. By performing more pruning, FC⁺ outperforms FC when the problem is involved in a highly dynamic environment.

For incomplete methods, the static strategy is the efficient strategy for easy and less dynamic problems. Overall, for medium and highly dynamic problems, the dynamic strategy obtains the best results. SDRW always returns higher quality solutions than MCRW and TS.

The Branch and Bound technique is applied to solve CCCSPPs and CCTCSPPs. The BnBFC and BnBFC⁺ algorithms show the best performance for the easy class problem. BnBMAC and BnBMAC⁺ are only efficient for the group of hard problems.

8. CONCLUSION AND FUTURE WORK

8.1 Summary

In this thesis, we have proposed a framework for handling CSPs in a dynamic environment in the case where the possible change is known a priori. In addition, we have applied the framework to address temporal problems. Our framework provides a new approach to solving dynamic constraint problems involving conditional contexts, composite objects, temporal constraints and preferences. The model and the resolution techniques using complete and incomplete methods can be applied to many different applications such as games, configuration problems and temporal problems.

The CCCSP framework is based on the conditional CSP framework of Mittal and Falkenhainer [69]. It consists of sets of simple variables, composite variables, binary constraints and activity constraints. All required variables are set to active status at the initial stage whereas the rest (conditional variables) are set to non-active status. The set of activity constraints is used to control activation of the conditional variables. The generic form of our activity constraint supports the four basic types of activity constraints proposed in Mittal and Falkenhainer [69]. Moreover, activity constraints are employed to select a simple variable from the domain of a composite variable. The idea of composite variables is not only practical for representing groups or hierarchies of objects in the conceptual view, but also useful for formulating optional variables (non active variables regardless of conditions), disjunctive activity constraints and unary temporal conditions over temporal events.

Complete methods and incomplete methods for solving CCCSPs and CCTCSPs are addressed and implemented. The complete resolution techniques are enhanced by ordering

heuristics and constraint propagation techniques. The First Fail principle heuristic for variable ordering is used to improve the complete search. Each simple variable is calculated for the degree of its associated constraints (involving both active and non-active variables). The degree of composite variables is the maximum degree of the simple variables in its domain. The order of variables is sorted by decreasing degree. The advantage of variable ordering is to guide the search to avoid deep levels of backtracking which are usually caused by most constrained variables. Therefore, the most-constrained variables are considered early.

In the searching stage, FC and MAC propagate constraint checking for the current assigned variable to the next future variables and to all future variables respectively, as long as the future variables are active. FC^+ and MAC^+ look ahead to new future non-active variables by extending the scope of the future variables to non-active variables. The experimentation results on the sets of randomly generated instances, demonstrate that considering non-active variables early in the constraint propagation process increases the efficiency in the time performance. MAC^+ returns solutions and takes less time than the other algorithms when the problems fall in the hard category (close to the phase transition), whereas the FCCCCSP algorithm performs better than MACCCCSP, MAC^+ and FC^+ when the problems are easy (under constrained or over constrained).

For incomplete methods, stochastic local search (i.e., MCRW, SDRW and TS) is applied with variant strategies. The static strategy first randomly generates a consistent CSP from the given CCCSP. The activity constraints are considered and are randomly controlled to trigger some non-active variables to become active. The rest of the non-active variables are discarded. Then, MCRW, SDRW and TS are directly applied to find a solution from the chosen CSP. Applying the static strategy to MCRW, SDRW and TS may fail to find a consistent CSP if the problems are hard or over constrained. However the static technique is suitable for less

constrained problems because the probability of consistent CSPs for these problems is highly.

The dynamic strategy allows the number of variables to change in the repairing stage. Activity constraints are checked in each iterative repairing stage to add or remove conditional variables. The experimental results show that overall, the quality of solutions produced from $MCRW_{Dynamic}$ and $SDRW_{Dynamic}$ are better than the results from $MCRW_{Static}$, $SDRW_{Static}$ and TS.

Finally the incremental strategy is the combination of the static and the dynamic approaches. This strategy starts from a random assignment of a set of active variables. The random assignment is repaired and improved until a consistent assignment is found or the algorithm reaches the maximum number of repairs. Then, activity constraints are checked to activate a new set of active variables. The new set of active variables is passed through the repairing loop to find a consistent assignment with the previous assignment. The algorithm stops when there are no new active variables. The experimental results show that the performance of $MCRW_{Incremental}$ and $SDRW_{Incremental}$ falls between the groups of static algorithms and dynamic algorithms.

We have extended the CCCSP framework to handle temporal constraints (CCTCSPs). Handling numeric and symbolic temporal constraints in the CCTCSP framework is based on TemPro [74]. Some encoding and checking inconsistency techniques are addressed. First, using composite variables with the slicing SOPO techniques makes temporal problems more explicit. Secondly, the numeric to symbolic conversion can detect some inconsistent partial assignments prior to the constraint propagation. Thirdly, numeric constraints on events' intervals can be encoded by a set of SOPOs.

The preferences are integrated into the CCCSP and the CCTCSP frameworks via the c-semiring structure. The preference functions can be over the domains of simple variables, the domains of composite variables, the constraints and the activity constraints. Also CCCSPPs

and CCTCSPPs are able to handle conditional preferences. The most preferred solution of a CCCSPP is evaluated by the degrees of global configuration, and global assignment. The Branch and Bound technique is integrated using FC, FC⁺, MAC and MAC⁺ strategies in order to estimate the upper bound of each future possible branch solution. The experimental results show that the BnBFC algorithm is most efficient for the group of easy problems where there are less activity constraints. BnBFC⁺ is also for the group of easy problems but it is able to handle a higher dynamic level than BnBFC. For the group of hard problems, BnBMAC and BnBMAC⁺ are the best algorithms. In addition, for highly dynamic problems, BnBMAC⁺ outperforms BnBMAC.

In the experimentation, the CCCSP random generator based on model RB [119] is implemented and also extended for generating preferences and temporal constraints. Good instances are assure to be produced from the generator and the phase transition can be estimated.

8.2 Future Work

8.2.1 Composite Variables Including Other Composites

The definition of our composite variables assumes one decomposed level in which only a set of simple variables is allowed. For some complex applications that require more than one level of decomposition variables, extending the definition of the original composite variables to include both composite variables and simple variables is necessary. Redefining the composite variable definition shows a potential of extensibility of CCCSP and CCTCSP frameworks. To support the recursive composite variable structure, the definition of AC3-CCCSPP needs to be revised. Consequently, all corresponding algorithms are forced to be remodified to support the change. We consider this extension structure in our future work.

8.2.2 Solving CCCSPs and CCTCSPs in Unknown Dynamic Environments

Beyond the scope of variable and constraint alteration defined in the CCCSP and the CCTCSP frameworks, uncertainties in unpredictable dynamic environments involve changing problem definitions from time to time. This class of problems is called dynamic CSPs [30]. Dynamic CSP algorithms dynamically search by reusing and repairing the current solution, when variables, domains and constraints have changed [30, 10, 112, 75]. One possible future line of work is on reusing and repairing current solutions of CCCSPs. In the case of the dynamic CCCSP framework, the uncertainty of a problem involves adding/removing variables, compatible constraints and activity constraints. One alternative approach is to classify the status of the associated variables/constraints/activity constraints into four groups:

1. The change only involves the set of initial active variables.
2. The change only involves the set of newly active variables.
3. The change only involves the set of non active variables.
4. The change involves combining variables from the above groups.

The resolution method for dynamic CCCSP will repair a current solution by considering the group of the associated variables. The advantage of detecting the primitive status of associated variables is to reduce the unnecessary consistency checking/repairing. The associated variables in group (1) may affect the entire current solution. In group (2), disabling the newly active variables may void the changes. In group (3), the current solution is consistent without repairing, if the associated variables are all non active in the current solution. An intensive analysis is a mandatory key to achieving this perspective.

8.2.3 Incomplete Methods for CCCSPs and CCTCSPs with Preferences

We have proposed different branch and bound methods (based on different constraint propagation techniques) for solving CCCSPs with Preferences. Exploring the incomplete mechanisms over CCCSPs is a worthwhile next step. Incomplete methods are a potential direction when preferences take place in a problem. Indeed, in this case, time is a sensitive issue and an incomplete solution is acceptable. Solving the dynamic CCCSPs using approximation methods is an open issue to research. In this case, we try to find a solution that is close to the optimal solution instead of maximizing the objective function. Possibly, the three proposed strategies (static, dynamic and incremental), will be a starting point for the challenge.

The first approach combines the optimization functions. At least two optimal functions should be defined. One is for the constraint satisfaction evaluation and another one is for evaluating the degree of preference satisfaction. In general, a local search accepts an incomplete solution; therefore, a new defined optimal function for the global preference degree is required because preference degrees of all inconsistent variables are set to zero. In each repair step, the two optimal functions are combined and evaluated to move to a better optimal solution which is in the next neighborhood configuration. Improving the quality of constraint satisfaction should have a higher priority than the preference degree factor.

The second alternative approach is dividing the algorithm into two main parts: (1) repairing stage for constraints satisfaction and (2) improving preference degree satisfaction. When an acceptable solution is found from the first routine, the current assignment is passed through the second routine to improve the degree of preferences. The algorithm recursively runs through step (1) and step (2) until it reaches the time limit. Then it returns the optimal solution, or

runs until a complete solution is found and the preference degree is at 1.

8.2.4 Real World Applications

Dress Up Games

We are interested to implement a real dress up game application based on this research framework. Developing a dress up game software involves designing a graphic user interface as a front end for users to play the game and to input user preferences. Our resolution techniques are embedded in the application to return the users one or all complete preferred solution(s). Our complete solving techniques can be adopted to find all possible solutions or all preferred solutions when a particular global preference degree is given and will be set as a lower bound for an acceptable solution. In addition, our incomplete methods can support playing the game in solving and learning approach. An inappropriate outfit can be first randomly generated. The number of violated constraints or the global preference degree can be used to indicate the score of the outfit. Then the players try to rearrange the outfit items in order to get to a higher score. Various occasions or scenarios can be added to challenge the players to learn and enjoy the game.

Robotic Applications

We will continue the above research on temporal reasoning. Moreover, we will also explore the extended framework onto a real world application domain, namely robotics. Robots are built for specific tasks or missions. There are different types of robots such as personal robots, service robots, security and defense robots. Usually a robot is placed in a dynamic and/or risky environment to perform a task for humans or to entertain people.

For instance, ASIMO (Advanced Step in Innovative Mobility) is a humanoid robot created

by Honda Motor Company [52]. ASIMO is an intelligent robot that has human-like interaction abilities. It has five recognition technology features: recognition of moving objects, recognition of postures and gestures, environment recognition, distinguishing sounds and facial recognition. It also has network integration to retrieve online information from the Internet and to support a user's network system.

If we want to program a robot maid that can prepare and serve a dinner as in Example 5.2.1, the steps of robot's movement must be considered and integrated to the problem.

Programming a robot involves various disciplines in Artificial Intelligent including Automated Agents, Neural Networks, Real Time Processing, Temporal Constraints and Constraint Processing. Temporal information relates to autonomous motion planning of robots in dynamic environments. A dynamic ordering plan to reach a goal and to respect the current temporal constraints needs to be computed at a time. Our research on the CCCSPP and the CCTCSPP frameworks shows a potential of managing complex temporal information including conditional constraints, composite variables and preferences. A comprehensive research in temporal constraints combining robotics will affirm our alternative technique.

Bibliography

- [1] E. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., New York, USA, 1997. ISBN 0471948225.
- [2] D. Achlioptas, L. M. Kirousis, E. Kranakis, D. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random Constraint Satisfaction: A More Accurate Picture. In *Principles and Practice of Constraint Programming*, pages 107–120, 1997. URL <http://citeseer.ist.psu.edu/article/achlioptas97random.html>.
- [3] J. F. Allen. *Natural Language Understanding — 2nd Edition*. The Benjamin/Cummings Publishing Company, 1995.
- [4] J.F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the Association for Computing Machinery (CACM)*, 26(11):832–843, 1983.
- [5] K. R. Apt, F. Rossi, and K. B. Venable. CP-nets and Nash Equilibria. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, 2005. URL <http://www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:cs/0509071>.
- [6] K.R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, USA, 2007. ISBN 0521866286.
- [7] S. Badaloni, M. Falda, and M. Giacomini. Integrating Quantitative and Qualitative Fuzzy Temporal Constraints. *Artificial Intelligent Communications*, 17(4):187–200, 2004. ISSN 0921-7126.

- [8] S. Badaloni and M. Giacomini. A Fuzzy Extension of Allen's Interval Algebra. In E. Lamma and P. Mello, editors, *In Proceedings of the 6th Congress of the Italian Association for Artificial Intelligence (AI*IA99)*, pages 228–237, 1999.
- [9] R. Barták. Theory and Practice of Constraint Propagation. In *Proceedings of CPDC2001 Workshop, Gliwice*, pages 7–14, 2004.
- [10] C. Bessière. Arc-Consistency in Dynamic Constraint Satisfaction Problems. In *In Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, pages 221–226, Anaheim, CA, 1991.
- [11] C. Bessière. Arc-Consistency and Arc-Consistency again. *Artificial Intelligence*, 65:179–190, 1994.
- [12] C. Bessière, E. Freuder, and J.C. Regin. Using Inference to Reduce Arc Consistency Computation. In *In Proceedings of the 1995 International Joint Conference on AI (IJCAI'95)*, pages 592–598, Montréal, Canada, 1995.
- [13] C. Bessière and J. C. Regin. Refining the Basic Constraint Propagation Algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 309–315, Seattle, WA, 2001.
- [14] C. Bettini, X. S. Wang, and S. Jajodia. Temporal Reasoning in Workflow Systems. *Distributed Parallel Databases*, 11(3):269–306, 2002. ISSN 0926-8782.
- [15] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-Based Constraint Satisfaction and Optimization. *J. Association for Computing Machinery (ACM)*, 44(2):201–236, 1997. ISSN 0004-5411.

- [16] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-Based CSPs and Valued CSPs: Frameworks, Properties, and Comparison. *Constraints*, 4(3):199–240, 1999. URL <http://citeseer.ist.psu.edu/bistarelli99semiringbased.html>.
- [17] S. Bistarelli, M. S. Pini, F. Rossi, and K. Brent. Positive and Negative Preferences. In *Proceedings of the 17th International Workshop on Preferences and Soft Constraints (Soft'05)*, pages 19–31, 2005.
- [18] M. Boddy. Practical Temporal Reasoning for Real Scheduling Applications. In *Proceedings of the 4th International Workshop on Temporal Representation and Reasoning (TIME'97)*, page 71, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7937-9.
- [19] C. Boutilier, R. Brafman, H. Hoos, and D. Poole. Reasoning with Conditional Ceteris Paribus Statements. In *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pages 71–80, 1999.
- [20] J. Bowen and D. Bahler. Conditional Existence of Variables in Generalised Constraint Networks. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, pages 215–220, 1991. URL <http://citeseer.ist.psu.edu/bowen91conditional.html>.
- [21] A. Chmeiss and P. Jégou. Path-Consistency: When Space Misses Time. *AAAI*, pages 196–201, 1996.
- [22] CHOCO. CHOCO, 2007. URL <http://choco.sourceforge.net/>.
- [23] M. B. Clowes. On Seeing Things. *Artificial Intelligence*, 2(1):79–116, 1971.
- [24] D. Cohen, M. Cooper, P. Jeavons, and A. Krokhin. Supermodular Functions and the Complexity of MAX CSP. *Discrete Appl. Math.*, 149(1-3):53–72, 2005. ISSN 0166-218X.

- [25] M. C. Cooper, D. A. Cohen, and P. Jeavons. Characterising Tractable Constraints. *Artificial Intelligence*, 65(2):347–361, 1994. URL <http://citeseer.ist.psu.edu/cooper94characterising.html>.
- [26] B. Craig, B. Ronen, D. Carmel, H. Holger, and P. David. Preference-Based Constrained Optimization with CP-Nets. *Computational Intelligence*, 20(2):137–157, 2004. URL <http://www.blackwell-synergy.com/doi/abs/10.1111/j.0824-7935.2004.00234.x>.
- [27] N. Creignou, M. Hermann, and R. Pichler. Complexity of Constraint Satisfaction Problems. URL <http://citeseer.ist.psu.edu/600503.html>.
- [28] A. Davenport, E. Tsang, C-J. Wang, and K. Zhu. GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 325–330. AAAI, 1994.
- [29] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1558608907.
- [30] R. Dechter and A. Dechter. Belief Maintenance in Dynamic Constraint Networks. In *Proceedings of the 7th National Conference on Artificial Intelligence, (AAAI'88)*, pages 37–42, St. Paul, MN, 1988.
- [31] R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.
- [32] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1988.

- [33] D. Diaz and C. FD. CLP(FD) 2.21, User's Manual, 1994. URL <http://citeseer.ist.psu.edu/diaz94clpfd.html>.
- [34] M. Dincbas. Constraint Programming: An Alternative to Expert Systems. In F. Belli and F. J. Radermacher, editors, *Proceedings of the 5th Industrial and Engineering Applications of Artificial Intelligence and Expert Systems International Conference (IEA/AIE'92)*, pages 46–48. Springer, Berlin, Heidelberg, 1992.
- [35] D. Duchier. Constraint Programming For Natural Language Processing. Lecture Notes, ESSLLI 2000, August 2000.
- [36] H. Fargier and J. Lang. Uncertainty in Constraint Satisfaction Problems: a Probabilistic Approach. In *Proceedings of the European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU'93)*, pages 97–104, London, UK, 1993. Springer-Verlag. ISBN 3-540-57395-X.
- [37] A. J. Fernández and P. M. Hill. A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5(3):275–301, 2000. URL <http://citeseer.ist.psu.edu/fernandez00comparative.html>.
- [38] J. Frank and A. J'onsson. Constraint-Based Attribute and Interval Planning. *Constraints*, 8(4):339–364, 2003. ISSN 1383-7133.
- [39] B. N. Freeman-Benson. A Module Mechanism for Constraints in Smalltalk. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 389–396, New York, USA, 1989. ACM Press. ISBN 0-89791-333-7.
- [40] E. C. Freuder, R. J. Wallace, and R. Heffernan. Ordinal Constraint Satisfaction. In *The 5th International Workshop on Soft Constraints (SOFT'02)*, 2003.

- [41] D. Frost, C. Bessière, R. Dechter, and J.C. Régin. Random Uniform CSP Generators, 1996. URL <http://www.lirmm.fr/~bessiere/generator.html>.
- [42] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998. URL <http://www.informatik.uni-muenchen.de/~fruehwir/drafts/jlp-chr1.ps.Z>.
- [43] J. Gaschnig. A General Backtrack Algorithm that Eliminates Most Redundant Tests. In *Proceedings of the 5th International Joint Conferences on Artificial Intelligence (IJCAI'77)*, page 457, Cambridge, MA, 1977.
- [44] E. Gelle. On the Generation of Locally Consistent Solution Spaces in Mixed Dynamic Constraint Problems. *Ph.D.thesis*, 1826:101–140, 1998.
- [45] E. Gelle and B. Falting. Solving Mixed and Conditional Constraint Satisfaction Problems. *Constraints*, 8:107–141, 2003.
- [46] E. Gelle and M. Sabin. Solving Methods for Conditional Constraint Satisfaction. In *In Proceedings of the Configuration Workshop jointly held with the 8th International Joint Conference on Artificial Intelligence (IJCAI'03)*, Acapulco, Mexico, 2003.
- [47] F. Geller and M. Veksler. Assumption-Based Pruning in Conditional CSP. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, Sitges, Barcelona, Spain, 2005.
- [48] I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. Random Constraint Satisfaction: Flaws and Structure. *Constraints*, 6(4):345–372, 2001.
- [49] M. L. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25, 1993.

- [50] Y. Hamadi. Disolver : A Distributed Constraint Solver. Technical Report MSR-TR-2003-91, Microsoft Research, Dec 2003.
- [51] C.C. Han and C.H. Lee. Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence*, 36:125–130, 1988.
- [52] Ltd Honda Motor Co. Honda Worldwide Site, 2008. URL <http://world.honda.com/ASIMO/>.
- [53] D.A. Huffman. Impossible Objects as Nonsense Sentences. In Meltzer and Michie, editors, *Machine Intelligence 6*. Edinburgh University Press, Edinburgh, 1970.
- [54] ILOG. ILOG CP, 1987-2003. URL <http://www.lkn.ei.tum.de/arbeiten/faq/man/ILOG/CONCERT/concert20/>.
- [55] JCL. JCL, 2007. URL <http://liawww.epfl.ch/~torrens/JCL/>.
- [56] A. Jónsson and J. Frank. A Framework for Dynamic Constraint Reasoning using Procedural Constraints, 2000. URL <http://citeseer.ist.psu.edu/jonsson00framework.html>.
- [57] H.A. Kautz and P.B. Ladkin. Integrating Metric and Qualitative Temporal Reasoning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, pages 241–246, Anaheim, CA, 1991.
- [58] S. Keretho and R. Loganantharaj. Qualitative and Quantitative Time Interval Constraint Networks. In *Proceedings of the 19th Annual Conference on Computer Science (CSC'91)*, pages 239–246, New York, USA, 1991. ACM Press. ISBN 0-89791-382-5.
- [59] L. Khatib, P. Morris, R. A. Morris, and F. Rossi. Temporal Constraint Reasoning With

- Preferences. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 322–327, 2001. URL <http://citeseer.csail.mit.edu/khatib01temporal.html>.
- [60] G. Kondrak and P. van Beek. A Theoretical Evaluation of Selected Backtracking Algorithms. In Chris Mellish, editor, *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*, Montreal, 1995. URL <http://citeseer.ist.psu.edu/kondrak95theoretical.html>.
- [61] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survey. *Artificial Intelligence Magazine*, 13(1):32–44, 1992. URL <http://citeseer.ist.psu.edu/article/kumar92algorithms.html>.
- [62] I. Laboratory, S. PO, B. S, and K. Sweden. SICStus Prolog User's Manual, 1995. URL <http://citeseer.ist.psu.edu/laboratory01sicstus.html>.
- [63] L.Zhang and S. Lau. Constructing University Timetable Using Constraint Satisfaction Programming Approach. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA '05)*, pages 55–60, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2504-0-02.
- [64] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.
- [65] A. K. Mackworth. The Logic of Constraint Satisfaction. *Artificial Intelligence*, 58(1-3): 3–20, 1992. ISSN 0004-3702.

- [66] A. K. Mackworth and E. Freuder. The Complexity of Some Polynomial Network-Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25: 65–74, 1985.
- [67] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, USA, 1990. ISBN 0-471-92420-2.
- [68] I. Meiri. Combining Qualitative and Quantitative Constraints in Temporal Reasoning. *AAAI*, pages 260–267, 1991.
- [69] S. Mittal and B. Falkenhainer. Dynamic Constraint Satisfaction Problems. In *In Proceedings of the 8th National Conference on Artificial Intelligence (AAAI’90)*, pages 25–32, Boston, MA, 1990. AAAI Press.
- [70] R. Mohr and T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.
- [71] M. Molloy. Models for Random Constraint Satisfaction Problems. *SIAM J. Comput.*, 32(4):935–949, 2003.
- [72] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences*, 7:95–132, 1974.
- [73] M. Mouhoub. Reasoning about Numeric and Symbolic Time Information. In *In Proceedings of the 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI’00)*, pages 164–172, Vancouver, 2000. IEEE Computer Society.
- [74] M. Mouhoub. Reasoning with Numeric and Symbolic Time Information. *Artificial Intelligence Rev.*, 21(1):25–56, 2004. ISSN 0269-2821.

- [75] M. Mouhoub. Solving Dynamic CSPs. In *Canadian Conference on AI (CanAI'04)*, pages 504–509, 2004.
- [76] M. Mouhoub, F. Charpillet, and J.P. Haton. Experimental Analysis of Numeric and Symbolic Constraint Satisfaction Techniques for Temporal Reasoning. *Constraints: An International Journal*, 2:151–164, Kluwer Academic Publishers, 1998.
- [77] M. Mouhoub and A. Sukpan. Solving Conditional and Composite Temporal Constraints. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 734–741, 2004.
- [78] M. Mouhoub and A. Sukpan. A New Temporal CSP Framework Handling Composite Variables and Activity Constraints. In *Proceedings of The 17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05)*, Hong Kong, 2005.
- [79] M. Mouhoub and A. Sukpan. Constraint Propagation versus Local Search for Conditional and Composite Temporal Constraints. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, Sitges, Barcelona, Spain, 2005.
- [80] M. Mouhoub and A. Sukpan. Conditional and Composite Temporal Constraints with Preferences. In *Proceedings of the 13th International Symposium on Temporal Representation and Reasoning (TIME'06)*, 2006.
- [81] M. Mouhoub and A. Sukpan. Solving Conditional and Composite Constraint Satisfaction Problems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07)*, pages 336–337, New York, USA, 2007. ACM. ISBN 1-59593-480-4.
- [82] Malek Mouhoub and Amrudee Sukpan. Managing Conditional and Composite CSPs. In

- Ziad Kobti and Dan Wu, editors, *Canadian Conference on AI (CanAI'07)*, volume 4509 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 2007. ISBN 978-3-540-72664-7.
- [83] N. Muscettola, P. Morris, B. Pell, and B. Smith. Issues in Temporal Reasoning for Autonomous Control Systems. In *Proceedings of the 2nd International Conference on Autonomous agents (AGENTS'98)*, pages 362–368, New York, USA, 1998. ACM Press. ISBN 0-89791-983-1.
 - [84] A. Nakhimovsky. Temporal Reasoning in Natural Language Understanding: the Temporal Structure of the Narrative. In *Proceedings of the 3rd Conference on European chapter of the Association for Computational Linguistics*, pages 262–269, Morristown, NJ, USA, 1987. Association for Computational Linguistics.
 - [85] C. M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994. ISBN 0201530821.
 - [86] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Prentice-Hall, 1982.
 - [87] S. Prestwich, F. Rossi, K. B. Venable, and T. Walsh. Constrained CP Nets. 2004.
 - [88] J.F. Puget and P.V. Hentenryck. A Constraint Satisfaction Approach to a Circuit Design Problem. *J. of Global Optimization*, 13(1):75–93, 1998. ISSN 0925-5001.
 - [89] R. Barták. Constraint Programming: In Pursuit of the Holy Grail. In *Proceedings of the Week of Doctoral Students (WDS99), Part IV*, pages 555–564, Prague, 1999. MatFyzPress.
 - [90] F. Rossi. Preference Reasoning. *Constraint Programming*, pages 9–12, 2005.

- [91] F. Rossi, C. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems. In Luigia Carlucci Aiello, editor, *In Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, pages 550–556, Stockholm, 1990. Pitman. URL <http://citeseer.ist.psu.edu/rossi90equivalence.html>.
- [92] Z. Ruttkay. Fuzzy Constraint Satisfaction. In *Proceedings of the 1st IEEE Conference on Evolutionary Computing*, pages 542–547, Orlando, 1994. URL <http://citeseer.ist.psu.edu/ruttkay94fuzzy.html>.
- [93] D. Sabin and E.C. Freuder. Configuration as Composite Constraint Satisfaction. In George F. Luger, editor, *Proceedings of the 1st Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996. ISBN 1-57735-003-0. URL <ftp://ftp.cs.unh.edu/pub/csp/Papers/aiman96-ds1.ps.gz>.
- [94] M. Sabin, E. C. Freuder, and R. J. Wallace. Greater Efficiency for Conditional Constraint Satisfaction. In *Proceedings of the 9th International Conference on Principles and Practice of, Constraint Programming (CP'03)*, 2833:649–663, 2003.
- [95] T. Schiex, H. Fargier, and G. Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In Chris Mellish, editor, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'95)*, Montreal, 1995.
- [96] C. Schulte and G. Smolka. Finite Domain Constraint Programming in Oz, A Tutorial. Technical report, 1999. URL <http://www.mozart-oz.org/documentation/fdt/>.
- [97] B. Selman and H. A. Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. In *In Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'93)*, pages 46–51, 1993.

- [98] B. Selman, H. A. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994. URL <http://citeseer.ist.psu.edu/selman94noise.html>.
- [99] H. M. Sheini, B. Peintner, K. A. Sakallah, and M. E. Pollack. On Solving Soft Temporal Constraints using SAT Techniques. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, 2005.
- [100] M. Singh. Efficient Path Consistency Algorithms for Constraint Satisfaction Problems. URL <http://citeseer.ist.psu.edu/article/singh95efficient.html>.
- [101] B. M. Smith and M. E. Dyer. Locating the Phase Transition in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, 81(1-2):155–181, 1996. URL <http://citeseer.ist.psu.edu/smith94locating.html>.
- [102] T. Soininen, E. Gelle, and I. Niemela. A Fixpoint Definition of Dynamic Constraint Satisfaction. In *Proceedings of the 5th Principles and Practice of Constraint Programming (CP'99)*, pages 419–433, 1999. URL <http://citeseer.ist.psu.edu/article/soininen99fixpoint.html>.
- [103] K. Stergiou and T. Walsh. Encodings of Non-Binary Constraint Satisfaction Problems. In *Proceedings of the 16th national conference on Artificial intelligence and the 11th Innovative applications of artificial intelligence conference innovative applications of artificial intelligence (AAAI '99/IAAI '99)*, pages 163–168, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence. ISBN 0-262-51106-1.
- [104] N. Tamura. Cream: Class Library for Constraint Programming in Java, 2007. URL <http://bach.istc.kobe-u.ac.jp/cream/>.

- [105] F. Torquati, M. Paltrinieri, and A. Momigliano. A Constraint Satisfaction Approach to Operative Management of Aircraft Routing. In *Proceedings of the 3rd International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE'90)*, pages 1140–1146, New York, USA, 1990. ACM Press. ISBN 0-89791-372-8.
- [106] I. Tsamardinos and M. Pollack. Efficient Solution Techniques for Disjunctive Temporal Problems, 2002. URL <http://citeseer.ist.psu.edu/tsamardinos02efficient.html>.
- [107] I. Tsamardinos, T. Vidal, and M. Pollack. CTP: A New Constraint-Based Formalism for Conditional, 2003. URL <http://citeseer.ist.psu.edu/tsamardinos03ctp.html>.
- [108] E. Tsang. *Foundation of Constraint Satisfaction*. Academic Press, 1994.
- [109] P. van Beek. Reasoning about Qualitative Temporal Information. *Artificial Intelligence*, 58:297–326, 1992.
- [110] P. van Beek and X. Chen. CPlan: A Constraint Programming Approach to Planning. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI'99)*, pages 585–590, 1999. URL <http://citeseer.ist.psu.edu/vanbeek99cplan.html>.
- [111] P. van Hentenryck, Y. Deville, and C. M. Teng. A Generic Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57:291–321, 1992.
- [112] G. Verfaillie and T. Schiex. Solution Reuse in Dynamic Constraint Satisfaction Problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, pages 307–312, 1994. URL <http://citeseer.ist.psu.edu/verfaillie94solution.html>.
- [113] M. Vilain and H. Kautz. Constraint Propagation Algorithms for Temporal Reasoning. In

- In Proceedings of the 5th National Conference of the American Association for Artificial Intelligence (AAAI'86)*, pages 377–382, Philadelphia, PA, 1986.
- [114] R. J. Wallace. Partial Constraint Satisfaction. *Lecture Notes in Computer Science*, 923: 121–138, 1995.
 - [115] R. J. Wallace. Conditional Lexicographic Orders in Constraint Satisfaction Problems. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, Sitges, Barcelona, Spain, 2005.
 - [116] R. J. Wallace and E.C. Freuder. Ordering Heuristics for Arc Consistency Algorithms. In *In Proceedings of the 9th Canadian Conference on Artificial Intelligence*, pages 163–169, Vancouver, Canada, 1992.
 - [117] K. Wiese, S. Nagarajan, and S.D. Goodwin. A C++ Class Library for Solving Binary Constraint Satisfaction Problems. Technical Report CS-96-08, Department of Computer Science, University of Regina, Sept 1996. URL <http://citeseer.ist.psu.edu/wiese96class.html>.
 - [118] K. Xu. Forced Satisfiable CSP and SAT Benchmarks of Model RB, 2003. URL <http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm>.
 - [119] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A Simple Model to Generate Hard Satisfiable Instances. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 337–342, Edinburgh, Scotland, August 2005.
 - [120] K. Xu and W. Li. Exact Phase Transitions in Random Constraint Satisfaction Problems. *Journal of Artificial Intelligence Research*, 12:93–103, 2000. URL <http://citeseer.ist.psu.edu/xu00exact.html>.

- [121] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *Knowledge and Data Engineering*, 10(5):673–685, 1998. URL <http://citeseer.ist.psu.edu/yokoo98distributed.html>.
- [122] Z. Yuanlin and R.H.C. Yap. Making AC-3 an Optimal Algorithm. In *IJCAI*, pages 316–321, 2001. URL <http://citeseer.ist.psu.edu/zhang01making.html>.
- [123] N. Zhou. B-Prolog User’s Manual (Version 7.0) Prolog, Agent, and Constraint Programming, 1994-2007. URL <http://www.probp.com/manual/index.html>.