

University of Wollongong Thesis Collections

University of Wollongong Thesis Collection

University of Wollongong

Year 2010

Solving semiring constraint satisfaction problems

Louise Leenen
University of Wollongong

Leenen, Louise, Solving semiring constraint satisfaction problems, Doctor of Philosophy thesis, School of Computer Science and Software Engineering - Faculty of Informatics, University of Wollongong, 2010. <http://ro.uow.edu.au/theses/3074>

This paper is posted at Research Online.

NOTE

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

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

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

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

Solving Semiring Constraint Satisfaction Problems

A thesis submitted in fulfillment of the
requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Louise Leenen

School of Computer Science and Software Engineering

January 2010

© Copyright 2010

by

Louise Leenen

All Rights Reserved

*Dedicated to
Tommie, Thomas, and André*

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Louise Leenen
January 7, 2010

Abstract

The Semiring Constraint Satisfaction Problem (SCSP) framework is a popular approach for the representation of partial constraint satisfaction problems. Considerable research has been done in solving SCSPs, but limited work has been done in building general SCSP solvers. In this thesis, we present various methods to solve SCSPs.

We first consider how a SCSP might be relaxed: we relax individual constraints until an acceptable solution can be obtained. A second semiring is used to define a measure of difference between the original problem and the relaxed problem. This research was first presented at the *International Workshop on Preferences and Soft Constraints* at CP-2005 [40], and an extended version of the paper has been published in the *Information Processing Letters* journal [41].

We then show how the two semirings of a relaxed SCSP can be combined into a single semiring structure. This combined semiring structure will make it possible to use existing tools for solving SCSPs to solve Combined SCSPs. This work appears in Leenen et al. [42].

The remainder of this thesis focusses on algorithms to solve SCSPs. A significant amount of work has been devoted to solving the well-known maximum satisfiability problem (Max-SAT) [1, 63] and the related Weighted Max-SAT problem. This prompted us to modify the methods for solving Max-SAT, into methods for solving SCSPs. We show how to translate a SCSP into a variant of the Weighted Max-SAT Problem, which we call a Weighted Semiring Max-SAT problem, and then present a local search algorithm that is a modification of the GSAT algorithm for solving Max-SAT. This work appears in Leenen et al. [38].

Finally, we extend well-known algorithms for maximal constraint satisfaction into

SCSP algorithms. We present a branch and bound algorithm, a backjumping algorithm, and a forward checking algorithm. Our branch and bound algorithm performs significantly better than CONFLEX [17], a well-known fuzzy CSP solver. The branch and bound algorithm has been presented in Leenen et al. [38]. The forward checking and backjumping algorithms perform better than the branch and bound algorithm on harder problems. This work appears Leenen et al. [39].

List of Publications resulting from the research presented in this thesis:

- L. Leenen, T. Meyer, and A. Ghose. Relaxations of semiring constraint satisfaction problems. In *Proceedings of the 7th International Workshop on Preferences and Soft Constraints (SOFT-05)*, 2005.
- L. Leenen, T. Meyer, P. Harvey, and A. Ghose. A relaxation of a semiring constraint satisfaction problem using combined semirings. In *Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence (PRICAI-2006)*, pages 907–911, 2006.
- L. Leenen, T. Meyer, and A. Ghose. Relaxations of semiring constraint satisfaction problems. *Information Processing Letters*, 103(5):177–182, 2007.
- L. Leenen, A. Anbulagan, T. Meyer, and A. Ghose. Modelling and solving semiring constraint satisfaction problems by transformation to weighted semiring max-SAT. In *Proceedings of the Twentieth Australian Joint Conference on Artificial Intelligence (AI-2007)*, pages 202–212, 2007.
- L. Leenen and A. Ghose. Branch and bound algorithms to solve semiring constraint satisfaction problems. In *Proceedings of the 10th Pacific Rim International Conference on Artificial Intelligence (PRICAI-2008)*, 2008.

Acknowledgements

There are a number of people who contributed towards the completion of my thesis through discussions, ideas, advice, friendship and support. Thank you to everyone, although I am only listing a few individuals here.

My sincere thanks to my supervisor, Aditya Ghose, for his guidance, friendship, and support.

A big thank you to my friend and husband, Tommie Meyer, for his constant interest, patience, great ideas, and reading and correcting all my writing.

Finally, to my mother and father, thank you for constantly enquiring about my progress and giving encouragement.

Financial Support

I am grateful for the financial support I received through an Australian Postgraduate Award Industry grant.

I also want to thank my current employer, the DPSS division of the Council for Scientific and Industrial Research in South Africa, for their generous support to complete this thesis.

Contents

Abstract	v
Acknowledgements	vii
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Structure	3
2 Over-constrained problems	5
2.1 Introduction	5
2.2 Alternative Approaches	9
2.2.1 Over-constrained solving with fuzzy valuations	9
2.2.2 Over-constrained solving with probabilistic valuations	9
2.2.3 Over-constrained solving with weighted valuations	10
2.3 Partial Constraint Satisfaction	10
2.3.1 Algorithms for maximal constraint satisfaction	11
2.3.2 A formal framework for partial constraint satisfaction	16
2.4 Hierarchical Approaches	17
2.5 The Semiring Constraint Satisfaction Framework	18
2.5.1 The Hotel Chain Example - Part 1	18
2.5.2 Definition of a SCSP	19
2.5.3 The Hotel Chain Example - Part 2	21
2.5.4 Finding a solution to a SCSP	22
2.5.5 The Hotel Chain Example - Part 3	24

2.5.6	Algorithms to solve SCSPs	27
2.6	Industry-scale instances of over-constrained problems	28
2.6.1	Description of the problem	28
2.6.2	Solving the problem	29
2.6.3	Results of the Industry Project	31
3	Relaxations of Semiring Constraint Satisfaction Problems	33
3.1	Motivation	33
3.2	A Relaxation of a SCSP	35
3.3	The Hotel Chain Example - Part 4	40
3.4	Related Work	43
3.5	Conclusion	44
4	A Relaxation of a Semiring Constraint Satisfaction Problem using Combined Semirings	45
4.1	A Combined Semiring	45
4.2	The Hotel Chain Example - Part 5	48
4.3	Conclusion	51
5	Weighted Semiring Max-SAT	52
5.1	SAT, Max-SAT and Weighted Max-SAT	53
5.1.1	Algorithms to solve SAT and Max-SAT	53
5.2	Weighted Semiring Max-SAT Problems	57
5.2.1	Translation of a CSP into a SAT problem	57
5.2.2	Translation of a SCSP into a WS-Max-SAT problem	58
5.2.3	Example: Transformation of a SCSP into a WS-Max-SAT	62
5.2.4	Correctness Proof	65
5.3	A GSAT-based Algorithm to solve WS-Max-SAT	66
5.3.1	Example: Solving a WS-Max-SAT problem	68
5.3.2	Results of GSAT-based Algorithm	70
5.4	Conclusion	72

6	Algorithms to solve SCSPs	73
6.1	Branch and Bound Algorithms for solving SCSPs	74
6.1.1	A Basic Branch and Bound Algorithm for SCSPs	74
6.1.2	A Basic Branch and Bound Algorithm with Variable Partitioning	81
6.1.3	A Backjumping Algorithm	82
6.1.4	A Forward Checking Algorithm	87
6.2	Experimental Setting and Results	91
6.2.1	The randomly generated problems	92
6.2.2	Branch and bound with variable partitioning vs. CON'FLEX . .	93
6.2.3	Branch and bound vs. Backjumping vs. Forward Checking . . .	94
6.3	A CSP-based Algorithm for Solving SCSPs	95
6.4	Conclusion and Future Work	98
7	Conclusion	100
7.1	Summary and Discussion	100
7.2	Future Work	101
	Bibliography	103
	Appendices	111
A	Random SCSP Generator	111
A.1	The random problem model	111
A.2	An example	112
B	List of Definitions	114

Chapter 1

Introduction

In this chapter, we give a motivation for our research and present an outline of the structure of the thesis.

1.1 Motivation

There has been considerable interest over the past decade in *over-constrained problems*, *partial constraint satisfaction problems* and *soft constraints*. This has been motivated by the observation that with most real-life problems, it is difficult to offer *a priori* guarantees that the input set of constraints to a constraint solver is solvable. In part, this is because many real-life problems are inherently over-constrained. This is also because it is difficult for human users to peruse a given set of constraints that have been obtained from a given problem to determine if it is solvable. In the general case, constraint solvers must be able to deal with problems that are potentially over-constrained.

The key challenge in dealing with an over-constrained problem is identifying appropriate *relaxations* of the original problem that are solvable. Early approaches to such relaxations largely focused on finding maximal subsets (with respect to set cardinality) of the original set of constraints that are solvable. Freuder and Wallace's [20] work on maximal constraint satisfaction is best known. Section 2.3 gives an overview of their work.

Subsequent efforts considered more fine-grained notions of relaxation, where entire

constraints did not have to be removed from consideration. Examples of such efforts include the hierarchical constraint logic programming (HCLP) framework of Wilson and Borning [61], Fuzzy CSPs [17] and Probabilistic CSPs [18]. See Sections 2.2 and 2.4 for more information on these approaches.

Bistarelli et al. [9] proposed an abstract semiring constraint satisfaction scheme that generalised most of these earlier attempts, while making it possible to define several useful new instances of the scheme. The semiring constraint satisfaction problem (SCSP) framework assumes the existence of a semiring of abstract preference values, such that the associated multiplicative operator is used for combining preference values, while the associated additive operator is used for comparison. While a classical constraint defines which combinations of value assignments to the variables in its signature are allowed, a SCSP constraint assigns a preference value to all possible value assignments to the variables in its signature. These preferences implicitly define a relaxation strategy (“try to satisfy the constraint using the most preferred tuples, else try the next most preferred tuples” and so on). Section 2.5 gives an overview of SCSPs.

In this thesis, we present various methods to solve SCSPs.

We first consider how a SCSP might be relaxed: we relax individual constraints until an acceptable solution can be obtained. A second semiring is used to define a measure of difference between the original problem and the relaxed problem. This research has been published in Leenen et al. [41]. This paper is an extended version of an initial presentation at the International Workshop on Preferences and Soft Constraints at CP-2005 [40].

We then show how the two semirings of a relaxed SCSP can be combined into a single semiring structure. This combined semiring structure will make it possible to use existing tools for solving SCSPs to solve Combined SCSPs. This work appears in Leenen et al. [42].

The remainder of this thesis focuses on algorithms to solve SCSPs. Our first approach is an attempt to adapt existing algorithms for solving satisfiability problems into SCSP

algorithms. A significant amount of work has been devoted to solving propositional satisfiability (SAT) problems, specifically the well-known maximum satisfiability problem (Max-SAT) [1, 63] and the related Weighted Max-SAT problem. There is continuing interest in translations between CSPs and SAT problems [4, 24, 59]. This prompted us to explore the application of methods for solving Max-SAT, to SCSPs.

We show how to modify the support encoding of Gent [24] to translate a CSP into a SAT, in order to translate a SCSP into a variant of the Weighted Max-SAT Problem. We call our encoding Weighted Semiring Max-SAT (WS-Max-SAT). Our encoding results in propositional clauses whose structure can be exploited. We present a local search algorithm that is a modification of the GSAT algorithm for solving Max-SAT. This work appears in Leenen et al. [38].

Our next approach is to extend the well-known algorithms of Freuder and Wallace [20] for maximal constraint satisfaction, into SCSP algorithms. We present a branch and bound algorithm, a backjumping algorithm, and a forward checking algorithm to solve SCSPs. We also show some experimental results. Our branch and bound algorithm performs significantly better than CONFLEX [17], a well-known fuzzy CSP solver. The branch and bound algorithm has been presented in Leenen et al. [38]. The forward checking and backjumping algorithms perform better than the branch and bound algorithm on some problem instances. This work appears in Leenen et al. [39].

Finally, we present a CSP-based algorithm to solve SCSPs.

1.2 Thesis Structure

Chapter 2 contains an introduction to methods and algorithms for solving over-constrained problems. This chapter also includes a description of the SCSP framework as well as the algorithms for maximal constraint satisfaction.

Chapter 3 deals with the relaxation of constraints in the case where we cannot obtain an acceptable solution for a SCSP. This results in a relaxed version of a SCSP.

In Chapter 4, we present a Combined SCSP where we show how the two semirings associated with a relaxed SCSP can be combined into a single semiring.

In Chapter 5, we show how to transform a SCSP into a variant of a Weighted Max-Sat problem, and present an incomplete local search algorithm to show that this transformation is viable.

In Chapter 6, we present a branch and bound algorithm to solve SCSPs, as well as two variants, a backjumping algorithm and a forward checking algorithm. We show results of the comparison of our algorithms, and a comparison of our branch and bound algorithm with a Fuzzy CSP solver. We also present a CSP-based algorithm to solve SCSPs.

Chapter 7 contains our conclusions and a discussion of the work we will attempt in the future.

Chapter 2

Over-constrained problems

This chapter provides the background for the rest of the thesis. We start off with an introduction to the solution of problems that are potentially over-constrained, and then describe different approaches to solving over-constrained problems.

2.1 Introduction

Constraint satisfaction problems (CSP) are problems where one must find states that satisfy a number of constraints [58]:

Definition 1 A *Constraint Satisfaction Problem* (CSP) is formulated as a triple $\langle V, D, C \rangle$ where:

- V is a finite set of variables.
- D is a set of domains for each variable in V .
- C is a set of constraints. Each constraint c , defined on some subset of V , is a relation on that subset. Each tuple of c represents one allowed combination of values that may be assigned to the variables over which c is defined.

The domain of each variable is finite. A constraint is a relation that restricts the values that the variables in its scope (the set of variables over which the relation is defined) may take. Such a relation can be unary, binary or of a higher arity. A constraint is

said to be satisfied if the values that are assigned to the variables in its scope, forms a tuple that belongs to the relation.

A solution to a CSP specifies values for all the variables such that the constraints are satisfied. CSPs are usually solved with backtracking and consistency checking techniques.

A tuple of values that violates a constraint is called an *inconsistency*. A consistency technique removes inconsistent values from the domains of variables. Most consistency techniques are incomplete and are applied in conjunction with other methods to solve CSPs completely. Two simple consistency techniques are node and arc consistency.

Mackworth [44] provides the following definitions for node and arc consistency. Node consistency is achieved when a value satisfies a unary constraint, that is, a constraint that has a single variable. In node consistency preprocessing, the domains of variables that appear in unary constraints are checked for consistency and inconsistent values are removed from their domains. Arc consistency preprocessing removes values from the domains of variables which are inconsistent with binary constraints. A variable X is arc consistent with another variable Y if, for every value a in the domain of X there exists a value b in the domain of Y such that (a, b) satisfies the binary constraint between X and Y . A value a is unsupported if there does not exist a value in the domain of Y such that the binary constraint between the variables X and Y is satisfied. A problem is arc consistent if every variable is arc consistent with each other variable in the problem. Arc consistency preprocessing consist of the removal of all unsupported values in the domains of variables (which appear in binary constraints).

A *Backtracking search* algorithm constructs a solution by assigning values to variables, one variable at a time, whilst ensuring consistency. Such a search follows a particular variable ordering, and starts by assigning a value to the first variable. It then proceeds to assign a value to the second variable such that this value is consistent with the first variable's value. The remaining variables are assigned values in turn, making sure that each assigned value is consistent with previously assigned values. If the algorithm finds a variable for which no domain value is consistent with previous assigned values, backtracking takes place. The algorithm changes the value assigned to the previous

variable, if possible, and the search continues. The search halts when all the variables have been assigned consistent values, or if no solution exists. Many improvements on the basic backtracking algorithm have been defined to reduce the size of the explored search space and to improve the search strategy. Examples of such improvements are techniques for variable and value ordering. Consult Dechter [15] for a thorough introduction to constraint processing algorithms.

Many real-life applications are inherently *over-constrained*, i.e. a solution does not exist. Constraint solvers must be able to deal with problems that are potentially over-constrained. These problems have motivated considerable interest into approaches other than the traditional constraint algorithms.

The traditional approaches to solve CSPs, backtracking and consistency checking, cannot be used to solved over-constrained problems unless we change our objective to maximising the number of satisfied constraints. In the following subsections we discuss alternative approaches to classical CSPs, partial constraint satisfaction, hierarchical approaches, and generic approaches:

Alternative Approaches

In classical constraint satisfaction, a constraint either allows a tuple of values for the variables in the set V or it does not. These constraints are called crisp constraints. To cater for over-constrained problems, some approaches have been developed where constraints are allowed to be non-crisp. Constraints are extended by attaching probabilities, weights or preference values to tuples. We describe Fuzzy CSPs [17, 19], Weighted CSPs [56], and Probabilistic CSPs [18] in Section 2.2.

Partial Constraint Satisfaction

A key approach to solving over-constrained problems is to identify appropriate *relaxations* of the original problem that are solvable. Initial approaches were aimed at relaxing the requirement that all the constraints have to be satisfied, that is, on finding maximal subsets of the original set of constraints that are solvable (i.e. the maximal

constraint satisfaction problem). Freuder and Wallace [20] show in their Partial Constraint Satisfaction approach how to relax CSPs by weakening the original CSP. Section 2.3 gives an overview.

Hierarchical Approaches

Subsequent research has focused on the notion of relaxation where entire constraints do not have to be removed from consideration. In constraint hierarchies, hierarchical preferences can be associated with constraints, i.e. constraints are regarded to be either hard or soft constraints. Hard constraints are required to hold, while soft (preferential) constraints are satisfied as much as possible depending on some defined means of measurement. This approach is followed by Borning and Wilson [61] and Menezes et al. [46]. Section 2.4 contains an overview of constraint hierarchies.

Generic Approaches

There are generic methods for solving over-constrained problems that include the approaches described above as specific instances. These frameworks include the Semiring-based Constraint Satisfaction Problem (SCSP) and Valued CSP (VSCP) theories of Bistarelli et al. [9], and Michael Jampel's compositional theory for reasoning about over-constrained systems [33]. Michael Jampel's theory is based on constraint hierarchies.

The SCSP and VSCP frameworks associate preference levels to each tuple of a constraint, as opposed to attaching a preference level to a constraint. The SCSP framework, described in Section 2.5, is based on semirings while VSCPs are based on monoids.

Bartak [2] and Rudova [52] can be consulted for more information on over-constrained problems and solution techniques.

2.2 Alternative Approaches

We describe some extensions of CSPs where constraints are relaxed by allowing them to be non-crisp. These approaches are useful for optimisation problems and over-constrained problems.

2.2.1 Over-constrained solving with fuzzy valuations

In this approach constraints are defined as fuzzy relations. Dubois et al. [17] defines a Fuzzy CSP (FCSP) such that each constraint has an associated fuzzy level for each tuple of values which indicates its preference value. The fuzzy level has a value ranging between 0 and 1. A fuzzy level of 1 indicates the most preferred value (i.e., the tuple is allowed), while a fuzzy level of 0 indicates the least preferred value (i.e., the tuple is not allowed).

To solve a fuzzy CSP, we have to find the set of tuples of values (for all variables) which have a maximal preference value. For each tuple its minimal preference level over all constraints is considered.

Fargier et al. [19] can be consulted for more information on fuzzy constraint satisfaction.

2.2.2 Over-constrained solving with probabilistic valuations

In a probabilistic constraint satisfaction problem [18], each constraint c is regarded to have a probability p of occurring in the problem under consideration (denoted by $p(c)$). For each subset of constraints there is an associated probability of that subset being satisfied by some assignment of values to variables. This probability is calculated by taking the product of the probabilities associated with all the constraints in the subset.

Another approach is to calculate the probability of a subset of constraints that violates some assignment of values. In this case, the product of $1 - p(c)$ for every constraint c in the relevant subset is calculated. A solution to a probabilistic CSP is an instantiation

of variables with a maximum probability.

2.2.3 Over-constrained solving with weighted valuations

In weighted constraint satisfaction problems (WCSPs), every tuple of values for every constraint is given an associated cost or weight [56]. A weight is a non-negative real number. A solution to a WCSPs is a complete assignment with a minimum cost where the total cost is the sum of the costs of the chosen tuple for each constraint.

One instance of this framework is the maximal constraint satisfaction problem (Max-CSP) where all the constraints have the same weight function [20]. The total cost of an assignment is the number of unsatisfied constraints.

2.3 Partial Constraint Satisfaction

Partial constraint satisfaction is an approach where we attempt to find values for a subset of variables that satisfy a subset of the constraints [20]. In partial constraint satisfaction the objective is to seek a solution to a weakened version of a complex problem.

Maximal constraint satisfaction is an approach to partial constraint satisfaction where the objective is to find a solution that satisfies as many constraints as possible. Freuder and Wallace [20] developed methods to solve partial constraint satisfaction problems (PCSPs) that are analogous to the basic backtracking and local consistency methods for classical constraint satisfaction [35, 47, 48]. They also developed more general models of partial constraint satisfaction (in the same paper) where they employ more complex metrics than counting constraint violations.

We first describe their methods for maximal constraint satisfaction (Section 2.3.1), and then consider their generalised framework for PCSPs (Section 2.3.2). We have a particular interest in the Freuder and Wallace methods for solving maximal constraint satisfaction. They are relevant in Chapter 6 where we extend some of Freuder and Wallace's algorithms into algorithms for solving SCSPs.

Interesting applications to solve classes of over-constrained problems based on the partial satisfaction notion of Freuder and Wallace have been developed. Beaumont et al. [3] developed algorithms to solve over-constrained temporal reasoning problems. Zhou et al. [64] investigated the solution of over-constrained problems in a multi-agent environment by relaxing constraints.

2.3.1 Algorithms for maximal constraint satisfaction

A Branch and Bound algorithm for MAX-CSP

The classical branch and bound algorithm:

Branch and bound (BnB) is a general search method for solving optimisation problems. It was introduced by Land and Doig [36] in 1960. The BnB algorithm searches for a maximum solution in the search space by keeping track of a lower bound which is the best solution found so far. It also calculates an upper bound value for every partial solution which is an over-estimation of the best solution that can be extended from that partial solution. If the upper bound for a partial solution is worse than the current lower bound, the (extended) solution is pruned from the search space.

We now describe the classical branch and bound method in more detail.

- *Preliminaries:* Let the root node represent the original problem with the complete feasible search space. The feasible search space is the possible set of solutions. Let $f(x)$ be the function we want to *maximise*, with x restricted to the feasible search space. A lower bound value for $f(x)$ is assigned, and an upper bound for the root node (which represents the first variable to be assigned a value) is calculated.
- Compare the lower bound value with the upper bound value of the node under consideration.

If the two bounds are equal and the current node is a leaf node, then an optimal solution has been found and the algorithm halts. An acceptable threshold between the lower bound and upper bounds can be defined, and in this case the

search can halt if this threshold is satisfied for all nodes.

Otherwise, the feasible search space is divided into two or more subspaces, where the different subspaces cover the complete feasible space. An optimal solution to any subproblem is a feasible solution, but not necessarily an optimal solution to the original problem. Any feasible solution can be used to prune the rest of the tree.

If the upper bound value for any node is smaller than the value of the lower bound (best solution found so far), no globally optimal solution can be found by extending the partial solution represented by the node. Therefore, the node and its descendants can be pruned.

- If a leaf node has an upper bound that is better (i.e. greater) than the lower bound, the lower bound value is changed to be equal to this node's upper bound value (i.e. a better solution has been found).
- The search continues until all nodes have been traversed or pruned. If the whole search space has not been traversed, visit the next node and calculate its upper bound.

Return to the step where the upper bound of this node is compared to the current lower bound.

Freuder & Wallace's branch and bound algorithm for MAX-CSP:

Branch and bound search for maximal constraint satisfaction is just as suitable as backtracking search is for constraint satisfaction. Backtrack search will not be able to find a solution to an over-constrained problem, while branch and bound search allows us to find a solution by violating a minimal number of constraints.

The number of constraint checks in a CSP algorithm is a useful measure of the amount of work that is done. In the Freuder and Wallace CSP analogues for maximal constraint algorithms, this is not a viable measure because local failure is defined differently. Local failure happens when the current search path fails and backtracking is initiated. A CSP search path fails as soon as a single constraint is not satisfied; in the maximal constraint satisfaction algorithms a search path only fails once the number of unsatisfied

constraints reaches a pre-defined limit.

Freuder and Wallace's branch and bound algorithm [20] keeps track of the best solution found so far, and prunes a search path when it becomes clear that it cannot reach a better solution. The algorithm stores the number of inconsistencies (i.e. the number of unsatisfied constraints) in the best solution up to a particular point in the search. This is called the necessary bound, N . The algorithm cuts off the search as soon as the number of inconsistencies found reaches a sufficient bound, S , or run out of search paths to try. Freuder and Wallace's algorithm is a depth-first implementation of a branch and bound algorithm.

In Chapter 6 we extend Freuder and Wallace's BnB algorithm to solve semiring constraint satisfaction problems.

Look-back Algorithms

These techniques test a new value for a variable by looking back and checking consistency with previous choices before extending the search path.

The classical backjumping algorithm:

The backjumping algorithm of Gaschnig [22] stores information about previous failures to avoid unnecessary constraint checks to find the same failures. Classical backtracking blindly backtracks to the previous level and continues its search. Backjumping recognises that all values tried for a given variable may not be inconsistent with the same previous value. When no consistent value can be found for a current node, the algorithm jumps back directly to the most recent ancestor node that is inconsistent with the current node. When backjumping assigns values to a particular variable, it remembers the depth of failure, i.e. the deepest level, l at which any of the values fails. If there are no more values to assign to this variable, the backjumping algorithm jumps back directly to that level l . Gaschnig's backjumping algorithm only backjumps if the current node is a leaf node.

Freuder & Wallace's backjumping algorithm for Max-CSP:

Freuder and Wallace [20] present a backjumping variation of the branch and bound

algorithm for maximal constraint satisfaction. It differs from the conventional backjumping algorithm in that we cannot always jump back all the way to the deepest level l where a path was pruned. If any values below the level l failed when tested, i.e. resulted in an increase in the count of unsatisfied constraints (at the time this value was chosen), we can only jump back to the deepest such level. The reason is that we may miss better solutions resulting from other choices of values (not yet tried) at level l . The search has to be continued from this deeper level.

Apart from storing the depth of failure, the algorithm of Freuder and Wallace also keeps track of the deepest level at which a value was (last) inconsistent. If this value is greater than l , then backtracking will only jump back to this level.

In Chapter 6 we extend Freuder and Wallace's backjumping algorithm to solve semiring constraint satisfaction problems.

The classical backmarking algorithm:

The backmarking algorithm [23] is another algorithm that looks back at previous choices in attempting to avoid unnecessary successful constraint checking and re-finding inconsistencies. This algorithm records the individual level, *Mark*, in a search tree at which an inconsistency is detected for each value of a decision variable V for which it attempts to assign a value. For example, if it finds a consistent value for a variable and then finds an inconsistent value later in the same search path, *Mark* is set equal to the level of the inconsistent node in the search tree. If no inconsistency is found for that variable value, its *Mark* value is set to the level of variable's ancestor in the tree.

In case a particular search path cannot be extended to a complete solution, and have to resume the search at a level higher than V , the backmarking algorithm also keeps track of the highest level, *Backto*, to which the search has backed up since the previous time V was visited. The next time V is visited again and a value v is considered, the algorithm compares the values of *Backto* and *Mark*: if $Mark < Backto$ then v will fail again, otherwise we have to start testing values v at the level *Backto* in the tree.

Freuder & Wallace's backmarking algorithm for Max-CSP:

Freuder and Wallace [20] extended the backmarking algorithm for maximal constraint

satisfaction. As before, a value v is not pruned necessarily the first time an unsatisfied constraint is found. This algorithm keeps track of the level, *Lastmark*, at which the last unsatisfied constraint was found. *LastMark* is the level where the maximum inconsistency bound was reached and where the search path is to be pruned. However, it also keeps a record of where the first inconsistency with v was found, as well as the number of inconsistencies between v and the values in the search path down to level *Lastmark*. All this information enables the algorithm to determine whether v should be tested or whether it should be pruned.

Look-ahead Algorithms

These techniques look at paths yet to be extended in the search tree to check for some form of local consistency before continuing the search. If the node (i.e. current value for a variable) does not meet consistency requirements, it is pruned. Forward checking [30] combines a look-ahead technique with backtracking.

The classical forward checking algorithm:

In forward checking, each assignment of a value v to a variable V is followed by some arc consistency checking: the domains of variables that appear in the same constraints as the variable V are checked for consistency with v . In a partial CSP context, it is not possible to discard values by doing arc consistency checking, unless there is an initial value for the necessary bound. Freuder and Wallace [20] extended the notion of arc consistency checking for maximal constraint satisfaction as follows: for each value, the number of domains with no supporting values can be counted (it is called the inconsistency count for that value). The inconsistency count is a lower bound on the increase in the expected number of unsatisfied constraints that will be found if this value is assigned.

In a particular search path, the inconsistency count for a proposed value for the current variable can be added to the number of unsatisfied constraints up to date in that search path, and compared to the current bound, N . Recall that N gives the least number of unsatisfied constraints found in any complete search path so far. If the sum is not less than N , then we know that the current search path will not result in a better solution.

It is important to note again that definition of failure in maximal constraint satisfaction context is different to that in the classical CSP context: if there is an unsatisfied constraint, a value is not rejected unless the total number of currently chosen values with which it is inconsistent, is equal to the bound N .

Freuder & Wallace's forward checking algorithm for Max-CSP:

In Freuder and Wallace's extension [20], a proposed value for a variable is known to be consistent (in the PCSP context) with previously assigned values (otherwise it would already have been pruned), and it is tested against the domains of the remaining uninstantiated variables. To be more specific, when a value v is assigned to a variable V , the domains of all the uninstantiated variables that share a constraint with V are checked for values that are inconsistent with v : these values are deleted from the domains. If the variable V is assigned another value at a later stage, the pruned value has to be restored.

In Chapter 6 we extend Freuder and Wallace's forward checking algorithm to solve semiring constraint satisfaction problems.

2.3.2 A formal framework for partial constraint satisfaction

Freuder and Wallace [20] also developed a more general model of partial constraint satisfaction in which they compare alternative problems rather than alternative solutions. They view partial satisfaction of a problem P as a search through a space of alternative problems for a solvable problem close enough to P . In their maximal constraint satisfaction algorithms, they allow a certain number of constraints to be violated. In this more general model, they consider the fact that the weakening of a constraint or constraints, creates a different problem with a different solution set. They define a space of problems with an ordering on solutions sets where the ordering gives a measurement of how close an altered problem (together with its solution set) is to the original problem (with its solution set).

They define a partial constraint satisfaction problem (PCSP) to be a CSP with a metric on the problem space as well as with a sufficient, S , and a necessary, N , solution for the

problem. An alternative problem from the defined problem space, P' , together with a solution are regarded as a sufficient solution to the original problem if the metric distance between the original problem and P' is smaller or equal to S . The alternative problem and a solution are regarded to be a necessary solution for the original problem if the metric distance between the two problems is less than N .

The metric on the problem space evaluates a solution by comparing problems. This differs from the maximal constraint satisfaction algorithms where violated constraints are counted. Freuder and Wallace note that the metric can ideally be defined in terms of the partial order that defines the distance between P' and the original problem, P , to be the number of solutions not shared by P and P' . When $P' \leq P$, this metric determines the increased number of solutions that has been added by weakening P .

2.4 Hierarchical Approaches

In a constraint hierarchy, a strength or preference is associated with every constraint. The constraint hierarchies approach [11, 12] is the first that tried to handle preferences in constraint systems. Many problems have both hard and weaker or preferential constraints. The hard constraints must hold. The aim is to satisfy the preferred constraints, but a solution is found even if they cannot all be satisfied. Relaxing the strength of constraints helps to find a solution of a previously over-constrained system of constraints. A *comparator* is used to compare two assignments of the hierarchy. A comparator is an irreflexive, transitive relation. Note that comparators may give partial orders of assignments, and that there exist many different comparators. We also have to be able to give some measure of how well a particular assignment satisfies a given constraint. An error function gives a measurement of how well an assignment satisfies a constraint. Rudova [52] gives an overview of constraint hierarchies.

The most popular algorithms to solve constraint hierarchies follow one of two approaches, *Refining Methods* or *Local Propagation Methods*. Bartak [2] gives an overview of these algorithms.

Refining algorithms start by satisfying the hard constraints, followed by the (non-hard)

constraints with the strongest preference levels, down to the least preferred constraints. A disadvantage of these methods is that solutions have to be recalculated from the start if a constraint is added to the problem. An example of a refining algorithm is the DeltaStar algorithm of Wilson and Borning [61].

The local propagation algorithms repeatedly select satisfiable constraints in order to find a solution. The idea is to start with a single constraint which leads to a choice of a value for a variable, and then use another constraint to choose a value for another variable, and so on. An example of a local propagation algorithm is the DeltaBlue algorithm of Sannella et al. [53].

2.5 The Semiring Constraint Satisfaction Framework

The semiring constraint satisfaction framework of Bistarelli et al. [9] is a general framework for satisfaction and optimisation where classical CSPs, FCSPs, WCSP, PCSPs, and other CSPs (over finite domains) can be formulated as special instances. This framework assumes a semiring of abstract preference values together with operators to combine and compare preference values.

In the rest of this thesis we present various methods to solve Semiring Constraint Satisfaction Problems (SCSPs). We give an introduction to the semiring constraint satisfaction framework by means of an illustrative example which is used again later in this thesis.

2.5.1 The Hotel Chain Example - Part 1

A hotel chain acquires a star rating that is an accumulation of the different branches. Currently it has a four star rating and it aims for a five star rating. Various renovations can be done at the branches to increase the rating of the hotel chain:

1. Lay new carpets.

2. Upgrade a swimming pool.
3. Paint the building.

The manager of the hotel chain has to choose which (minimal) renovations to do at which branches under certain restrictions (such as the budget, renovations needed at each branch, and the constraints under which the renovating teams operate, etc.). The hotel chain consist of three branches which are denoted by X, Y, and Z. To avoid unnecessary disruptions, the manager wants at most one renovation job to be done at a particular branch, and as few renovation jobs in total as possible.

We can express this problem as a SCSP where the semiring structure allows the manager to express his preferences for particular tuples of domain values of the constraints. The definition of a SCSP [9] follows in the next subsection.

2.5.2 Definition of a SCSP

When we deal with constraints, the type of semirings that are used are called c-semirings. Bistarelli et al. [9] define a c-semiring, a constraint system, a constraint and a SCSP with respect to c-semirings.

Definition 2 [9] A *c-semiring* is a tuple $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that

- A is a set with $\mathbf{0}, \mathbf{1} \in A$;
- $+$ is defined over (possibly infinite) sets of elements of A as follows ¹:
 - for all $a \in A$, $\sum(\{a\}) = a$;
 - $\sum(\emptyset) = \mathbf{0}$ and $\sum(A) = \mathbf{1}$;
 - $\sum(\bigcup A_i, i \in I) = \sum(\{\sum(A_i), i \in I\})$ for all sets of indices I (flattening property);
- \times is a commutative, associative, and binary operation with operands from the set A , such that $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element;

¹When $+$ is applied to sets of elements, we will use the symbol \sum in prefix notation.

- \times distributes over $+$ (i.e. for any $a \in A$ and $B \subseteq A$, $a \times \sum(B) = \sum(\{a \times b, b \in B\})$).

For the remainder of the thesis, we refer to c-semirings as semirings.

The elements of the set A are the preference values to be assigned to tuples of values of the domains of constraints. The operator \times is used to combine constraints in order to find a solution (i.e. a single constraint) to a SCSP, and the operator $+$ is used to define the semiring value of the projection of a tuple of values over a set of variables onto a subset of the variables.

We derive a partial ordering \leq_S over the set A : $\alpha, \beta \in A$, $\alpha \leq_S \beta$ iff $\{\alpha\} + \{\beta\} = \{\beta\}$. The minimum element in the ordering is $\mathbf{0}$, while $\mathbf{1}$ is the maximum element. This partial order will be used to distinguish the maximal solution (see Definition 10) of a SCSP.²

Definition 3 [9] A *constraint system* is a 3-tuple $CS = \langle S_p, D, V \rangle$, where $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$ is a semiring, V is an ordered finite set of variables, and D is a finite set containing the allowed values for the variables in V .

For each tuple of values (of D) for the involved variables of a constraint, a corresponding element of A_p is assigned.

A constraint (see Definition 4) specifies variables and the values that they can be assigned. For each tuple of values for the involved variables, an element of the semiring's set A_p is associated. This element can be interpreted as the tuple's preference value.

Definition 4 [9] Given a constraint system $CS = \langle S_p, D, V \rangle$, where $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$, a *constraint* over CS is a pair $c = \langle def_c^p, con_c \rangle$ where $con_c \subseteq V$ is called the type of the constraint, and $def_c^p : D^k \rightarrow A_p$ (where k is the cardinality of con_c) is called the value of the constraint.

In the rest of this thesis, we refer to the value of a constraint as its preference value or its associated semiring value.

²In this thesis we write singleton subsets of the set A without braces.

We now have the building blocks required to define a SCSP.

Definition 5 [9] Given a constraint system $CS = \langle S_p, D, V \rangle$, a *Semiring Constraint Satisfaction Problem* (SCSP) over CS is a pair $P = \langle C, con \rangle$ where C is a finite set of constraints over CS and $con = \bigcup_{c \in C} con_c$. We also assume that $\langle def_{c_1}^p, con_c \rangle \in C$ and $\langle def_{c_2}^p, con_c \rangle \in C$ implies $def_{c_1}^p = def_{c_2}^p$.

We continue with the hotel chain example introduced in Section 2.5.1 to illustrate how the problem can be expressed as a SCSP.

2.5.3 The Hotel Chain Example - Part 2

Let us express the problem as a SCSP with $CS = \langle S_p, D, V \rangle$ and $P = \langle C, con \rangle$, where $V = con = \{X, Y, Z\}$, $D = \{0, 1, 2, 3\}$, $C = \{c_1, c_2, c_3\}$, and $S_p = \langle \{0, 0.25, 0.5, 0.75, 1\}, max, min, 0, 1 \rangle$.

The value of a decision variable indicates which renovation job is to be done at a particular branch: let the value 0 represent no renovation to be done, the value 1 represent re-carpeting, the value 2 represent pool renovation, and the value 3 represent painting a building. A renovation job with a higher value will contribute more towards a higher star rating.

Assume three binary constraints, $c_1 = \langle def_{c_1}^p, \{X, Y\} \rangle$, $c_2 = \langle def_{c_2}^p, \{Y, Z\} \rangle$, and $c_3 = \langle def_{c_3}^p, \{X, Z\} \rangle$. The tuples of these constraints together with their preference values (i.e. associated semiring values) are given in Table 2.1.

The manager's choice of semiring value to associate with a tuple, represents the desirability of that particular tuple. Consider $def_{c_1}^p(\langle 0, 2 \rangle) = 0.75$: tuple $\langle 0, 2 \rangle$ of constraint c_1 represents the case where nothing is to be done at branch X, while the swimming pool at branch Y is to be upgraded. Its high preference value indicates that it is preferred, for instance, to the tuple $\langle 1, 1 \rangle$ with a value of 0.25. The tuple $\langle 1, 1 \rangle$ represents the case where both branches X and Y will receive new carpets.

Also consider the preference values for constraint c_3 (the fourth column of Table 2.1): the manager prefers either one of the tuples $\langle 0, 3 \rangle$ or $\langle 3, 0 \rangle$ over any other tuples. These

Table 2.1: Constraint Definitions

t	$def_{c_1}^p(t)$	$def_{c_2}^p(t)$	$def_{c_3}^p(t)$
$\langle 0, 0 \rangle$	0	0	0
$\langle 0, 1 \rangle$	0.25	0.25	0.25
$\langle 0, 2 \rangle$	0.75	0.75	0.75
$\langle 0, 3 \rangle$	1	1	1
$\langle 1, 0 \rangle$	0.25	0.25	0.25
$\langle 1, 1 \rangle$	0.25	0.25	0.25
$\langle 1, 2 \rangle$	0.5	0.5	0.5
$\langle 1, 3 \rangle$	0.5	0.5	0.5
$\langle 2, 0 \rangle$	0.5	0.5	0.5
$\langle 2, 1 \rangle$	0.5	0.5	0.5
$\langle 2, 2 \rangle$	0.5	0.5	0.5
$\langle 2, 3 \rangle$	0.25	0.25	0.25
$\langle 3, 0 \rangle$	1	1	1
$\langle 3, 1 \rangle$	0.5	0.5	0.5
$\langle 3, 2 \rangle$	0.5	0.5	0.5
$\langle 3, 3 \rangle$	0	0	0

tuples represent the cases where the buildings at either one of branch X or branch Z is to be painted. A tuple with an associated value of 0 is highly undesirable.

In the next section we describe how a solution to a SCSP is found.

2.5.4 Finding a solution to a SCSP

The definitions of combination and projection operations [9] which are used to calculate a solution to a SCSP are given in this section.

The two semiring operators, multiplicative and additive, are used to compute preference values for the tuples of the variables in the set *con* from the preference values specified for the tuples of each constraint. The multiplicative operation combines the semiring (preference) values of the tuples of each constraint to get the semiring value of a tuple for all the variables, and the additive operation is used to obtain the semiring value of the tuples of the variables in the type of the problem.

The following three definitions from Bistarelli et al. [9] define the projection of a tuple

from one set of variables to another set, the combination of two constraints, and the projection of a constraint from its type to another set of variables, respectively.

Definition 6 [9] Given a constraint system $CS = \langle S_p, D, V \rangle$ with V totally ordered via \preceq , consider any k -tuple $t = \langle t_1, t_2, \dots, t_k \rangle$ of values of D and two sets $W = \{w_1, \dots, w_k\}$ and $W' = \{w'_1, \dots, w'_m\}$ such that $W' \subseteq W \subseteq V$ and $w_i \preceq w_j$ if $i \leq j$ and $w'_i \preceq w'_j$ if $i \leq j$. The *projection of the tuple t from W to W'* , written $t \downarrow_{W'}^W$, is defined as the tuple $t' = \langle t'_1, \dots, t'_m \rangle$ with $t'_i = t_j$ if $w'_i = w_j$.

The next definition defines the operation of combining two constraints to form a single constraint. We will use this operation repeatedly to combine all the constraints of a problem, into a single constraint.

Definition 7 [9] Given a constraint system $CS = \langle S_p, D, V \rangle$ where $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$. Let c_1 and c_2 be two constraints with $c_1 = \langle def_{c_1}^p, con_{c_1} \rangle$ and $c_2 = \langle def_{c_2}^p, con_{c_2} \rangle$ over CS. Their *combination*, written $c_1 \otimes c_2$, is the constraint $c = \langle def_c^p, con_c \rangle$ with $con_c = con_{c_1} \cup con_{c_2}$ and $def_c^p(t) = def_{c_1}^p(t \downarrow_{con_{c_1}}^{con_c}) \times_p def_{c_2}^p(t \downarrow_{con_{c_2}}^{con_c})$.

The operation \otimes is commutative and associative because \times_p is. We can extend the operation \otimes to more than two arguments, say $C = \{c_1, \dots, c_n\}$, by performing $c_1 \otimes c_2 \otimes \dots \otimes c_n$, which we denote by $(\otimes C)$.

Definition 8 [9] Given a constraint system $CS = \langle S_p, D, V \rangle$, where $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$, a constraint $c = \langle def_c^p, con_c \rangle$ over CS, and a set I of variables ($I \subseteq V$). The *projection of c over I* , written $c \Downarrow I$, is the constraint $c' = \langle def_{c'}^p, con_{c'} \rangle$ over CS with $con_{c'} = I \cap con_c$ and $def_{c'}^p(t') = \sum_{\{t \mid t \downarrow_{I \cap con_c}^{con_c} = t'\}} def_c^p(t)$.

A solution to a SCSP can now be defined.

Definition 9 [9] Given a SCSP $P = \langle C, con \rangle$ over a constraint system CS, the *solution of the SCSP P* is a constraint defined as the set $Sol(P) = (\otimes C)$.

A solution to a SCSP is formed by the combination of all the original constraints of the problem into a single constraint. Such a constraint has a semiring value associated with each tuple of values of D for the variables in con . A maximal solution consists of the set of k -tuples of D whose associated semiring values are maximal with respect to \leq_{S_p} .

Definition 10 Given a SCSP problem $P = \langle C, con \rangle$ over a constraint system $\langle S_p, D, V \rangle$ with $Sol(P) = \langle def_c^p, con \rangle$, the *maximal solution* of P is the set

$$MSol(P) = \{ \langle t, v \rangle \mid def_c^p(t) = v \text{ and there is no } t' \text{ such that } v <_{S_p} def_c^p(t') \}.$$

Let $MSolV(P) = \{ v \mid \langle t, v \rangle \in MSol(P) \}$.

Note that the set $MSol(P)$ contains all the maximal tuples of the single constraint, $(\otimes C)$, that depicts the solution to P *together* with each maximal tuple's associated preference value. The set $MSolV(P)$ contains only the associated preference values of theses tuples, i.e. this set only contains incomparable preference values.

In the next section we illustrate how a maximal solution for the hotel chain problem described in Sections 2.5.1 and 2.5.3 is calculated.

2.5.5 The Hotel Chain Example - Part 3

A solution to the problem consists of the combination of the three constraints in the set C .

The first step is to combine the first two constraints, c_1 and c_2 . Table 2.2 shows the semiring values associated with each tuple of the constraint $c'_1 = c_1 \otimes c_2$. For example, $def_{c'_1}^p(\langle 0, 1 \rangle) \times_p def_{c_2}^p(\langle 1, 3 \rangle) = \min(0.25, 0.5) = 0.25$, so tuple $\langle 0, 1, 3 \rangle$ of constraint c'_1 has a preference value of 0.25.

Then we combine the constraint c'_1 and the constraint c_3 to get the solution, constraint $c'_2 = c'_1 \otimes c_3$. Table 2.3 shows the semiring values associated with constraint c'_2 .

The following tuples of constraint c'_2 all have a preference value of 0.5 and form the maximal solution with $MSolV(P) = \{0.5\}$:

$\langle 0, 2, 2 \rangle; \langle 0, 3, 2 \rangle; \langle 1, 2, 2 \rangle; \langle 1, 3, 2 \rangle; \langle 2, 0, 2 \rangle; \langle 2, 1, 2 \rangle; \langle 2, 2, 0 \rangle; \langle 2, 2, 1 \rangle; \langle 2, 2, 2 \rangle; \langle 3, 0, 2 \rangle;$

Table 2.2: Definition of Constraint c'_1

t	$def_{c'_1}^p(t)$	t	$def_{c'_1}^p(t)$
$\langle 0, 1, 0 \rangle$	0.25	$\langle 2, 0, 1 \rangle$	0.25
$\langle 0, 1, 1 \rangle$	0.25	$\langle 2, 0, 2 \rangle$	0.5
$\langle 0, 1, 2 \rangle$	0.25	$\langle 2, 0, 3 \rangle$	0.5
$\langle 0, 1, 3 \rangle$	0.25	$\langle 2, 1, 0 \rangle$	0.25
$\langle 0, 2, 0 \rangle$	0.5	$\langle 2, 1, 1 \rangle$	0.25
$\langle 0, 2, 1 \rangle$	0.5	$\langle 2, 1, 2 \rangle$	0.5
$\langle 0, 2, 2 \rangle$	0.5	$\langle 2, 1, 3 \rangle$	0.5
$\langle 0, 2, 3 \rangle$	0.25	$\langle 2, 2, 0 \rangle$	0.5
$\langle 0, 3, 0 \rangle$	0.1	$\langle 2, 2, 1 \rangle$	0.5
$\langle 0, 3, 1 \rangle$	0.5	$\langle 2, 2, 2 \rangle$	0.5
$\langle 0, 3, 2 \rangle$	0.5	$\langle 2, 2, 3 \rangle$	0.25
$\langle 1, 0, 1 \rangle$	0.25	$\langle 2, 3, 0 \rangle$	0.25
$\langle 1, 0, 2 \rangle$	0.25	$\langle 2, 3, 1 \rangle$	0.25
$\langle 1, 0, 3 \rangle$	0.25	$\langle 2, 3, 2 \rangle$	0.25
$\langle 1, 1, 0 \rangle$	0.25	$\langle 3, 0, 1 \rangle$	0.25
$\langle 1, 1, 1 \rangle$	0.25	$\langle 3, 0, 2 \rangle$	0.75
$\langle 1, 1, 2 \rangle$	0.25	$\langle 3, 0, 3 \rangle$	1
$\langle 1, 1, 3 \rangle$	0.25	$\langle 3, 1, 0 \rangle$	0.25
$\langle 1, 2, 0 \rangle$	0.5	$\langle 3, 1, 1 \rangle$	0.25
$\langle 1, 2, 1 \rangle$	0.5	$\langle 3, 1, 2 \rangle$	0.5
$\langle 1, 2, 2 \rangle$	0.5	$\langle 3, 1, 3 \rangle$	0.5
$\langle 1, 2, 3 \rangle$	0.25	$\langle 3, 2, 0 \rangle$	0.5
$\langle 1, 3, 0 \rangle$	0.5	$\langle 3, 2, 1 \rangle$	0.5
$\langle 1, 3, 1 \rangle$	0.5	$\langle 3, 2, 2 \rangle$	0.5
$\langle 1, 3, 2 \rangle$	0.5	$\langle 3, 2, 3 \rangle$	0.25
all other tuples	0		

Table 2.3: Definition of Constraint c'_2

t	$def_{c'_2}^p(t)$	t	$def_{c'_2}^p(t)$
$\langle 0, 1, 1 \rangle$	0.25	$\langle 2, 0, 1 \rangle$	0.25
$\langle 0, 1, 2 \rangle$	0.25	$\langle 2, 0, 2 \rangle$	0.5
$\langle 0, 1, 3 \rangle$	0.25	$\langle 2, 0, 3 \rangle$	0.25
$\langle 0, 2, 1 \rangle$	0.25	$\langle 2, 1, 0 \rangle$	0.25
$\langle 0, 2, 2 \rangle$	0.5	$\langle 2, 1, 1 \rangle$	0.25
$\langle 0, 2, 3 \rangle$	0.25	$\langle 2, 1, 2 \rangle$	0.5
$\langle 0, 3, 1 \rangle$	0.25	$\langle 2, 1, 3 \rangle$	0.25
$\langle 0, 3, 2 \rangle$	0.5	$\langle 2, 2, 0 \rangle$	0.5
$\langle 1, 0, 1 \rangle$	0.25	$\langle 2, 2, 1 \rangle$	0.5
$\langle 1, 0, 2 \rangle$	0.25	$\langle 2, 2, 2 \rangle$	0.5
$\langle 1, 0, 3 \rangle$	0.25	$\langle 2, 2, 3 \rangle$	0.25
$\langle 1, 1, 0 \rangle$	0.25	$\langle 2, 3, 0 \rangle$	0.25
$\langle 1, 1, 1 \rangle$	0.25	$\langle 2, 3, 1 \rangle$	0.25
$\langle 1, 1, 2 \rangle$	0.25	$\langle 2, 3, 2 \rangle$	0.25
$\langle 1, 1, 3 \rangle$	0.25	$\langle 3, 0, 1 \rangle$	0.25
$\langle 1, 2, 0 \rangle$	0.25	$\langle 3, 0, 2 \rangle$	0.5
$\langle 1, 2, 1 \rangle$	0.25	$\langle 3, 1, 0 \rangle$	0.25
$\langle 1, 2, 2 \rangle$	0.5	$\langle 3, 1, 1 \rangle$	0.25
$\langle 1, 2, 3 \rangle$	0.25	$\langle 3, 1, 2 \rangle$	0.5
$\langle 1, 3, 0 \rangle$	0.25	$\langle 3, 2, 0 \rangle$	0.5
$\langle 1, 3, 1 \rangle$	0.25	$\langle 3, 2, 1 \rangle$	0.5
$\langle 1, 3, 2 \rangle$	0.5	$\langle 3, 2, 2 \rangle$	0.5
all other tuples	0		

$\langle 3, 1, 2 \rangle$; $\langle 3, 2, 0 \rangle$; $\langle 3, 2, 1 \rangle$ and $\langle 3, 2, 2 \rangle$.

All other tuples have preference values of either 0.25 or 0. The manager can choose any of the tuples in the set $MSolV(P)$.

We mentioned before that the SCSP framework is a general framework for solving constraint satisfaction problems and over-constrained problems. Interesting instances of semirings are:

- A classical CSP: $\langle \{0, 1\}, \vee, \wedge, 0, 1 \rangle$;
- A fuzzy CSP (see section 2.2.1): $\langle [0, 1], \supremum, \infimum, 0, 1 \rangle$;
- A weighted CSP (see section 2.2.3): $\langle \mathbb{Z}^+, \text{minimum}, +, \infty, 0 \rangle$. (Note that the

operator, $+$, presents addition on the set of nonnegative integers.)

2.5.6 Algorithms to solve SCSPs

Considerable interest has been shown in developing algorithms for solving SCSPs, but there does not exist an efficient general SCSP solver as yet. In this section, we summarise a few of these methods.

Georget and Codognet [26] developed a backtracking-based logic programming system, $clp(FD, S)$, for solving SCSPs. Their system integrates their semiring kernel (SFD) and the constraint logic programming system $clp(FD, S)$.

Bistarelli et al. [5] proposed a dynamic programming approach which solves a subset of the original set of constraints repeatedly, and then replaces this subset by a new generated constraint.

Rossi and Pihan [10] developed an abstraction based algorithm to solve fuzzy CSPs. Their algorithm passes an abstract version (a simpler version) of a given soft problem to be solved, and then returns some of the information gathered during the solving process back to the original problem in order to transform it into an equivalent problem that is easier to solve.

Bistarelli et al. [7] presented a prototype SCSP solver based on an incomplete local search method which employs problem transformation and soft consistency techniques.

Wilson [62] showed how to use decision diagrams to solve SCSPs but provides no implementation.

Delgado et al. [16] solved SCSPs by using the logic constraint solver, Mozart.

We present and implement an incomplete local search algorithm, three branch-and-bound algorithms, and a CSP-based algorithm to solve SCSPs in Chapters 5 and 6 of this thesis.

2.6 Industry-scale instances of over-constrained problems

The research described in this thesis was conducted in the context of an industry project with a major steel producer. The problem that we addressed was highly over-constrained and was, in part, the driver for our research into over-constrained problems.

2.6.1 Description of the problem

During the steel making process, steel slabs are cast by the castor, and then need to be heated to a required temperature before they can be processed in the Hot Strip Mill (HSM) where the slabs are rolled into required products. There are numerous restrictions placed on the order in which the slabs enter the HSM. The goal of these requirements are to reduce the wear and tear on the equipment, especially the rollers of the HSM.

Our task was to set up a processing schedule for slabs entering the HSM such that the specified requirements are met, and in addition, that some of the slabs are *hot charged*, i.e. reach the HSM while their temperature is still high enough to be processed without additional heating.

The steel producer currently uses a human expert assisted by an expert system to do the scheduling. We had access to existing schedules and the human expert.

Each slab has approximately 20 characteristics, e.g its cast date and time, width, thickness, steel and chemical grade, temperature, length, weight, etc. Some of the restrictions on the scheduling are based on the differences in width, thickness, grade, and chemistry between a slab and the adjacent slabs in the processing order. There are also requirements for the total length of a schedule and the first few slabs in a schedule (called the *start-up*). One of the most important constraints requires the width of the sequence of slabs in a schedule to form a coffin shape, i.e. the width of the slabs must initially increase (referred to as the *runout phase*), and after the widest slab has been processed, the width of the remaining slabs must decrease (*rundown phase*). There are

constraints that determine the lengths of the two phases.

The slabs that are to be hot charged, have to reach the HSM in succession and within a certain time after having been cast.

Most of the requirements are supplied in the form of tabular information, but some characteristics have to be calculated (e.g. the hardness of a slab).

2.6.2 Solving the problem

We decided to define the problem as a constraint satisfaction problem and use a commercial constraint solver, ILOG Solver³, to implement the solution.

Stage 1: Solving a simplified version of the problem as a CSP

It was clear that we were dealing with a very over-constrained problem. We started by considering a subset of the requirements. We defined this simplified version of the problem as a CSP and implemented a solution in ILOG Solver.

Although we were able to solve small instances of this version of the problem, we realised that the problem is highly over-constrained and that we were unlikely to find a solution.

Stage 2: Relaxing the constraints in the simplified version

We identified possible relaxations of the included constraints in our simplified version with the assistance of a human expert in steel making (e.g. by allowing bigger jumps, etc.).

This modification resulted in unpredictable behaviour from our ILOG Solver. In some small instances, our solver was unable to find solutions while managing to solve some larger instances.

³<http://www.ilog.com>

Stage 3: Including a pre-sorting phase

In this stage we included pre-processing in the form of sorting the input slabs according to various characteristics such as width, thickness or chemical grade.

This modification of the solver also resulting in inconsistent results.

Stage 4: Analysing actual schedules used in the plant

During this stage, we analysed existing schedules plus the actual sets of input slabs to determine how the steel plant relaxed the requirements. We analysed 13 existing schedules and focused on the width, thickness and chemistry (grade) characteristics of the scheduled slabs.

The conclusions that followed from this analysis enabled us to set guidelines for the relaxation of the constraints and viable forms of pre-processing.

The existing schedules are drawn up by a human expert assisted by an expert system. We could determine which constraints were more often unsatisfied. This information enabled us to define a measurement of the extent to which a schedule does not satisfy all the requirements.

Although these modifications improved the performance of our solver to some extent, it still did not produce consistent results. We realised that we needed a new approach.

Stage 5: Adding more constraints and more input slabs

At this stage we cut down the size of the search tree by incorporating more of the requirements, i.e. by adding more constraints. We also identified some constraints that have to be satisfied, i.e. hard constraints.

The version of ILOG solver that was available at this stage, did not offer a lot of partial constraint satisfaction facilities. We managed to build some partial solving by setting goals. We also incorporated partial constraint satisfaction by allowing an input set that is greater than the required length of the schedule, i.e. if a schedule of size n is required we made an input size of $n + k$ slabs available, with k and n positive integers.

This version of the solver produced schedules double the size of any previous version. Although this was a big improvement, the solver needed to produce longer schedules in an acceptable processing time (less than 10 minutes).

Stage 6: Dividing the problem in two stages

A HSM schedule consists of a runout and a rundown phase as described previously.

We decided to regard the scheduling of the slabs in the runout and rundown phases as two separate CSPs. During the scheduling of the runout slabs, we still pre-selected and hard coded the start-up slabs, and then solved the runout problem.

The last coil of the runout was then hard coded as the first slab in the rundown problem, and all the slabs scheduled in the runout schedule were deleted from the input set for the rundown problem. Then we solved the rundown problem.

Apart from this division of the problem into two separate problems and the hard coded slabs, we also pre-sorted the set of input slabs in order of increasing width for the runout scheduling, and in order of decreasing width for the rundown problem.

These modifications proved to be very successful and we managed to create long enough schedules in a few seconds.

2.6.3 Results of the Industry Project

At the conclusion of this project our solver produced a single schedule very fast, but required more coils as input than the size of the schedule that is produced, and did not yet include hot charge constraints.

The unscheduled input slabs were not deemed to be a problem as they can either be left for a next schedule or manually inserted by the human operator.

The next planned stage was to modify the solver to produce 3 to 4 schedules and include the hot charge constraints. The steel producer aims to develop a solver that requires an expert scheduler to adjust some input values to the program until the resulting schedules are satisfactory.

We believe that the repertoire of techniques that we describe in this dissertation will eventually find application in industry settings such as described in this section.

Chapter 3

Relaxations of Semiring Constraint Satisfaction Problems

The main approach to solving an over-constrained problem is to identify an appropriate relaxation of the original problem that is solvable. Our aim in this chapter is to define how a SCSP might be relaxed. In Chapter 2 we described the SCSP framework of Bistarelli et al. [9], and gave the definition for maximal solution to a SCSP. A maximal solution to a SCSP is defined as the best set of solution tuples for the variables in the problem. Sometimes this maximal solution may not be good enough, and in this case we want to change the constraints so that we solve a problem that is slightly different from the original problem but has an acceptable solution.

In this chapter, we propose a relaxation of a SCSP, and use a second semiring to give a measure of the difference between the original SCSP and the relaxed SCSP. We introduce a relaxation scheme but do not address the computational aspects. Note that this work was first presented at the *International Workshop on Preferences and Soft Constraints* at CP-2005 [40], and an extended version of the paper has appeared in the *Information Processing Letters* journal [41].

3.1 Motivation

The SCSP framework generalises different approaches to identify relaxations to over-constrained problems. A SCSP constraint assigns a preference value to all possible value assignments to the variables in its type. These preferences implicitly define a

relaxation strategy (“try to satisfy the constraint using the most preferred tuples, else try the next most preferred tuples” and so on).

At first blush, it may appear counter-intuitive to attempt to relax a SCSP. We will explain our motivations by describing it in terms of a generic optimisation problem (C, O) , defined by a set of constraints C and an objective function O . Assume that we have been given a lower bound on the value of the optimal solution (e.g., a minimal threshold on profit by a business unit set by management). Consider a situation where the optimal solution obtained fails to meet this threshold (e.g., the optimal profit figure falls short of the profit target). We are interested in seeking a new (and potentially relaxed) set of constraints C' that is minimally different from the original set C (under some notion of minimal difference that we will leave undefined for the time being), such that the revised optimisation problem (C', O) admits an optimal solution that satisfies the threshold. The revised (or relaxed) set of constraints C' is potentially very useful, because it can point to minimal changes in the physical reality being modelled by the constraints, which, if effected, would permit us to meet the threshold on the value of the objective function.

In this chapter, we attempt such an exercise in the context of SCSPs. A SCSP does not have an explicit objective function. Objectives are implicitly articulated (in a distributed fashion) via the preferences over tuples in each SCSP constraint. Instead of an optimal solution, we are able to articulate the preference values of the (potentially many) “best” solutions to a SCSP. Consider a SCSP P and a threshold β on the preference value of the “best” solution(s) to P . Assume that the “best” solutions to P fall short of this threshold. We define a mechanism by which we may “minimally” alter (i.e. relax) P to obtain a P' such that it admits a “best” solution that meets this threshold.

We propose a relaxation scheme for SCSPs in Section 3.2, but further research is required to develop efficient algorithms to compute these relaxations. We describe our proposal by defining what a good enough solution is, and how to find a suitable relaxation for a SCSP.

Consider the example of the hotel chain we introduced in Chapter 2. Assume the hotel

in question is unable to attain a five-star rating and wishes to determine the minimal changes required to its infrastructure in order to achieve such a rating. Recall that the star rating of the hotel is modelled via semiring preference values. In Section 3.3 we extend this example by relaxing the problem such that the required five-star rating may be reached.

In Section 3.4 we discuss frameworks by Bistarelli et al. [6] and Ghose and Harvey [27] that are related to our proposal.

3.2 A Relaxation of a SCSP

We are interested in a SCSP for which the maximal solution is not considered to be good enough. For example, the manager of the hotel chain example (Sections 2.5.1 and 2.5.5) may require a better solution: suppose he needs a solution tuple with a given minimum preference value of 0.75. The constraints of a problem model requirements that may be relaxed. We attempt to find a satisfactory solution to a relaxed version of the original problem.

We define a *mathematical filter* below. The concept of a filter was first introduced by Cartan [13] in 1937. The definition below is taken from [60].

Definition 11 A non-empty subset F of a partially ordered set (A, \leq_A) is called a *filter*, if the following conditions hold:

- For every $x, y \in F$, there is some element $z \in F$, such that $z \leq_A x$ and $z \leq_A y$.
- For every $x \in F$ and $y \in A$, $x \leq_A y$ implies that $y \in F$.

The smallest filter that contains a given element α is a principal filter, and α is called its principal element. The *principal filter* for α is given by the set $\uparrow \alpha = \{x \in A \mid \alpha \leq_A x\}$.

Recall that the preference values associated with a maximal solution of a SCSP P , i.e. the set $MSolV(P)$, is defined in Definition 10. In Definition 12 below, we identify

a partially ordered set of “lower bound” preference values that are regarded as being good enough.

Definition 12 Let P be a SCSP over a constraint system $\langle S_p, D, V \rangle$ with $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$. Suppose B is the set of all sufficient preference values, that is $B = \{\beta \mid \beta \in A_p \text{ \& } \beta \text{ is sufficient}\}$ and let $LB = \{\bigcup(\uparrow \beta) \mid \beta \in B\}$. A *good enough solution* for P is such that some element in $MSolV(P)$ is a member of the set LB .

The set LB contains the union of principal filters for preference values $\beta \in A_p$ such that all values of LB are regarded as being good enough. If $MSolV(P) \cap LB \neq \emptyset$, we have found a good enough solution for a problem P , i.e. there exist a solution with an associated preference value that is regarded as being good enough. If this is not the case, we want to find a relaxation P' of P , such that $MSolV(P') \cap LB \neq \emptyset$. P' should be as close to P as possible, that is, P' should be such that there does not exist any other relaxation of P closer to P than P' .

We first define a relaxation of a single constraint.

Definition 13 A constraint $c_j = \langle def_j^p, con_j \rangle$ is called a *c_i -weakened constraint* of the constraint $c_i = \langle def_i^p, con_i \rangle$ iff the following hold:

- $con_i = con_j$, and
- for all tuples t , $def_i^p(t) \leq_s def_j^p(t)$.

In Definition 13 the constraints c_i and c_j share the same set of tuples, but a tuple t may have a different associated semiring value in constraint c_j than in constraint c_i .

Note that a constraint c is itself a c -weakened constraint.

We now introduce a second semiring structure to represent the closeness of a c -weakened constraint to the constraint c . We refer to these semiring values of this second semiring as *difference values*. Every c -weakened constraint of a constraint c (including the constraint c itself) will be assigned such a difference value.

Definition 14 Given a constraint system $CS = \langle S_p, V, D \rangle$ and a SCSP $P = \langle C, con \rangle$, for each $c \in C$, let W_c be the set containing all c -weakened constraints, i.e.

$$W_c = \{c_j \mid c_j \text{ is a } c\text{-weakened constraint}\}.$$

Let $S_d = \langle A_d, +_d, \times_d, \mathbf{0}, \mathbf{1} \rangle$ be a semiring and $wdef_c^d : W_c \rightarrow A_d$ be any function such that

- A_d is a well-founded set (it contains no infinite descending chains);
- $wdef_c^d(c_j) = \mathbf{0}$ iff $c_j = c$;
- $\forall c_i, c_j \in W_c$, if for all tuples t $def_i^p(t) \leq_{S_p} def_j^p(t)$ then $wdef_c^d(c_i) \leq_{S_d} wdef_c^d(c_j)$;
- if there exists one tuple t such that $def_i^p(t) <_{S_p} def_j^p(t)$ and for all tuples s we have $def_i^p(s) \leq_{S_p} def_j^p(s)$, then $wdef_c^d(c_i) <_{S_d} wdef_c^d(c_j)$.

The function $wdef_c^d : W_c \rightarrow A_d$ is called a *difference function*.

We use $+_d$ for comparing, and \times_d for combining semiring values (see Definition 18). Observe that the set A_d is restricted to sets that do not contain infinite chains of weaker values such as the set of reals.

Definition 14 describes a function $wdef_c^d$ that assigns semiring (or difference) values from the set of the semiring S_d to each c -weakened constraint. This function is restricted by the preference values associated with the tuples of the c -weakened constraints. If the preference values of all the tuples of a c -weakened constraint c_j are at least as good as their preference values in another c -weakened constraint c_i , $wdef_c^d$ assigns a difference value for c_j that is at least as good as the difference value it assigns to c_i . If there is at least one tuple that has a better preference value in c_j than in c_i (and all other tuples have preference values in c_j that are at least as good as those in c_i), then $wdef_c^d$ will assign a better difference value to c_j than to c_i . (We compare semiring values in terms of the partial order defined on them.) This framework is deliberately broad so as to accommodate any reasonable application.

Now we define the concept of *closeness* with respect to a constraint c and a c -weakened constraint.

Definition 15 Let $c \in C$ be a constraint in a SCSP $P = \langle C, \text{con} \rangle$, and let wdef_c^d be a difference function with the semiring $S_d = \langle A_d, +_d, \times_d, \mathbf{0}, \mathbf{1} \rangle$.

- The c -weakened constraint c_i is closer to c than the c -weakened constraint c_j , iff $\text{wdef}_c^d(c_i) <_{S_d} \text{wdef}_c^d(c_j)$.
- The c -weakened constraint c_i is no closer to c than the c -weakened constraint c_j , iff $\text{wdef}_c^d(c_j) \leq_{S_d} \text{wdef}_c^d(c_i)$.
- The c -weakened constraints c_i and c_j are incomparable with respect to closeness to c iff $\text{wdef}_c^d(c_i) \not\leq_{S_d} \text{wdef}_c^d(c_j)$ and $\text{wdef}_c^d(c_j) \not\leq_{S_d} \text{wdef}_c^d(c_i)$.

Below we define a relaxation of a SCSP, and then we describe a way to formalise “closeness” of a relaxation to the original problem.

Definition 16 A SCSP $P' = \langle C', \text{con} \rangle$ is a d -relaxation of the SCSP $P = \langle C, \text{con} \rangle$ where $S_d = \langle A_d, +_d, \times_d, \mathbf{0}, \mathbf{1} \rangle$, iff there is a bijection $f : C \rightarrow C'$ and $\forall c \in C, f(c)$ is a c -weakened constraint with a difference function wdef_c^d .

For a $f(c) \in C'$ and $c \in C$, $\text{wdef}_c^d(f(c))$ is an indication of the closeness of $f(c)$ to c . For every $c \in C$, C' contains one c -weakened constraint, i.e. every c can be regarded as being replaced by a c -weakened constraint $f(c)$. We want to find a d -relaxation $P' = \langle C', \text{con} \rangle$ of $P = \langle C, \text{con} \rangle$ such that every c -weakened constraint $c' \in C'$ is the closest possible to the constraint $c \in C$ while the maximal solution of P' is still good enough (with respect to the set LB defined in Definition 12).

It is necessary to place some restrictions on the multiplicative operator \times_d so that the difference value of a d -relaxation will indeed reflect the closeness of the relaxed problem to the original problem.

Theorem 1 Given a SCSP $P' = \langle C', \text{con} \rangle$ which is a d -relaxation of the SCSP $P = \langle C, \text{con} \rangle$ where $S_d = \langle A_d, +_d, \times_d, \mathbf{0}, \mathbf{1} \rangle$, let c_{ik} be a c_i -weakened constraint, and c_{jm} and c_{jn} be c_j -weakened constraints with $c_i, c_j \in C$.

If $\text{wdef}_{c_j}^d(c_{jm}) <_{S_d} \text{wdef}_{c_j}^d(c_{jn})$, then

$$\text{wdef}_{c_i}^d(c_{ik}) \times_d \text{wdef}_{c_j}^d(c_{jm}) <_{S_d} \text{wdef}_{c_i}^d(c_{ik}) \times_d \text{wdef}_{c_j}^d(c_{jn}).$$

Proof 1 The multiplicative operator of a semiring is monotone on a partial order on the set of a semiring (see [9]). Thus \times_d is monotone on $<_{S_d}$.

We now define the set of all possible d -relaxations of a SCSP P , and go on to define a maximal solution of a d -relaxation.

Definition 17 Let $R(P) = \{P' \mid P' \text{ is a } d\text{-relaxation of } P\}$,
 $R_{LB}(P) = \{P' \in R(P) \mid MSolV(P') \cap LB \neq \emptyset\}$, and
 $MSolR_{LB}(P) = \{\langle t, v \rangle \mid \langle t, v \rangle \in MSol(P') \ \& \ P' \in R_{LB}(P)\}$.

The set $R_{LB}(P)$ contains all those SCSPs that are weakened versions of P whose best tuples intersect with the set LB , i.e. whose maximal solutions are good enough. Note that every tuple in $MSol(P')$ is a tuple of P' with a maximal semiring (preference) value, and that $MSolV(P')$ contains those maximal preference values. The set $MSolR_{LB}(P)$ contains the tuples and associated preference values of the maximal solution for a particular d -relaxation P' .

We now define a measure of difference between a problem P and a d -relaxation P' .

Definition 18 Given a d -relaxation $P' = \langle C', con \rangle$ of a SCSP $P = \langle C, con \rangle$ where $S_d = \langle A_d, +_d, \times_d, \mathbf{0}, \mathbf{1} \rangle$, such that $P' \in R_{LB}(P)$, let $d(P') = \times_d_{c \in C} (wdef_c^d(f(c)))$ be the *difference* between P and P' .¹

We have to find every $P' \in R_{LB}(P)$ with a minimal difference between P' and a problem P .

Definition 19 Let $MR_{LB}(P) = \{P' \in R_{LB}(P) \mid \nexists P'' \in R_{LB}(P) \text{ such that } d(P') <_{S_d} d(P'')\}$.

The set $MR_{LB}(P)$ contains the maximal weakened versions of the SCSP P .

In the example below we show how the hotel chain manager (see Sections 2.5.1 and 2.5.5) can find an improved solution to his problem by solving a relaxed version of the original problem.

¹We use \times_d in prefix notation when it is applied to more than two arguments.

3.3 The Hotel Chain Example - Part 4

The manager of the hotel chain problem wants to raise the hotel's four-star rating to a five-star rating. He has calculated that he needs a solution that gives a preference value of at least 0.75. The maximal solution to the original problem has a preference value of 0.5 which is not good enough.

We now attempt to find a d -relaxation to this problem with a sufficient solution. Note that we illustrate this process by only considering some relaxations of the second constraint as shown in Tables 3.1 and 3.2. The preference values that have been adjusted are written in bold face in both the tables.

In each of the c_2 -weakened constraints c_{2_1} up to c_{2_7} (Table 3.1), we only changed a single tuple's preference value from 0.5 to 0.75. These constraints represent c_2 -weakened constraints with the smallest possible change from the original constraint, c_2 . In each of the constraints c_{2_8} up to $c_{2_{11}}$ (Table 3.2), we only changed a single tuple's preference value from 0.25 to 0.75.

Although there exist many other relaxations of the second constraint, they will not enable us to obtain the required maximal solution.

We will use the semiring $S_d = \langle \{0, 1, 2, 3, 4, 5\}, \min, \max, 5, 0 \rangle$ to present a measure of difference between the c_2 -weakened constraints and the original constraint c_2 . Note that the partial order \leq_{S_d} is the order \geq over the integers. Table 3.3 shows the semiring (difference) values we associate with each of the relaxed constraints. The function $wdef$ should reflect the manager's preferred way to relax constraints: $wdef$ assigns a smaller difference value for relaxed constraints in which the least measure of change to the preferences values of tuples (i.e. the smallest adjustments) has been made, i.e. an adjustment from 0.5 to 0.75 is regarded as being less of a change than an adjustment from 0.25 to 0.75.

We associate a difference value of 0 (the best difference value) with the constraint c_2 , a difference value of 1 with each of the constraints c_{2_1} up to c_{2_7} , and a difference value of 2 with the constraints c_{2_8} up to $c_{2_{11}}$. There are a number of other possible relaxations of c_2 (not shown in Table 3.2) of which the most relaxed one will be a constraint where

Table 3.1: Definitions of some c_2 -weakened constraints

t	c_2	c_{2_1}	c_{2_2}	c_{2_3}	c_{2_4}	c_{2_5}	c_{2_6}	c_{2_7}
$\langle 0, 1 \rangle$	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
$\langle 0, 2 \rangle$	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75
$\langle 0, 3 \rangle$	1	1	1	1	1	1	1	1
$\langle 1, 0 \rangle$	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
$\langle 1, 1 \rangle$	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
$\langle 1, 2 \rangle$	0.5	0.75	0.5	0.5	0.5	0.5	0.5	0.5
$\langle 1, 3 \rangle$	0.5	0.5	0.75	0.5	0.5	0.5	0.5	0.5
$\langle 2, 0 \rangle$	0.5	0.5	0.5	0.75	0.5	0.5	0.5	0.5
$\langle 2, 1 \rangle$	0.5	0.5	0.5	0.5	0.75	0.5	0.5	0.5
$\langle 2, 2 \rangle$	0.5	0.5	0.5	0.5	0.5	0.75	0.5	0.5
$\langle 2, 3 \rangle$	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
$\langle 3, 0 \rangle$	1	1	1	1	1	1	1	1
$\langle 3, 1 \rangle$	0.5	0.5	0.5	0.5	0.5	0.5	0.75	0.5
$\langle 3, 2 \rangle$	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.75
$\langle -, - \rangle$	0	0	0	0	0	0	0	0

Table 3.2: Definitions of additional c_2 -weakened constraints

t	c_2	c_{2_8}	c_{2_9}	$c_{2_{10}}$	$c_{2_{11}}$...
$\langle 0, 1 \rangle$	0.25	0.75	0.25	0.25	0.25	...
$\langle 0, 2 \rangle$	0.75	0.75	0.75	0.75	0.75	...
$\langle 0, 3 \rangle$	1	1	1	1	1	...
$\langle 1, 0 \rangle$	0.25	0.25	0.75	0.25	0.25	...
$\langle 1, 1 \rangle$	0.25	0.25	0.25	0.75	0.25	...
$\langle 1, 2 \rangle$	0.5	0.5	0.5	0.5	0.5	...
$\langle 1, 3 \rangle$	0.5	0.5	0.5	0.5	0.5	...
$\langle 2, 0 \rangle$	0.5	0.5	0.5	0.5	0.5	...
$\langle 2, 1 \rangle$	0.5	0.5	0.5	0.5	0.5	...
$\langle 2, 2 \rangle$	0.5	0.5	0.5	0.5	0.5	...
$\langle 2, 3 \rangle$	0.25	0.25	0.25	0.25	0.75	...
$\langle 3, 0 \rangle$	1	1	1	1	1	...
$\langle 3, 1 \rangle$	0.5	0.5	0.5	0.5	0.5	...
$\langle 3, 2 \rangle$	0.5	0.5	0.5	0.5	0.5	...
$\langle -, - \rangle$	0	0	0	0	0	...

Table 3.3: Difference (semiring) values for some of the c_2 -weakened constraints

$wdef_{c_2}^d(c_2)$	0
$wdef_{c_2}^d(c_{21})$	1
$wdef_{c_2}^d(c_{22})$	1
$wdef_{c_2}^d(c_{23})$	1
$wdef_{c_2}^d(c_{24})$	1
$wdef_{c_2}^d(c_{25})$	1
$wdef_{c_2}^d(c_{26})$	1
$wdef_{c_2}^d(c_{27})$	1
$wdef_{c_2}^d(c_{28})$	2
$wdef_{c_2}^d(c_{29})$	2
$wdef_{c_2}^d(c_{210})$	2
$wdef_{c_2}^d(c_{211})$	2

all tuples have a preference value of 1. All these other possible c_2 -weakened constraints can be allocated some difference value higher than 2. Note that a higher difference value represent a more relaxed constraint. The worst value that can be assigned to a constraint is 5.

Our d -relaxation must be as close as possible to the original problem. We initially consider any one of the c_2 -weakened constraints with an associated difference value of 1.

If we select the constraint c_{25} in the place of constraint c_2 , we will be able to raise the maximal solution. In this d -relaxation of the original problem, we simply give a higher preference value to a solution where the swimming pools are to be upgraded at both branches Y and Z. Recall that the tuple $\langle 2, 2 \rangle$ represents the case where the swimming pools at both these branches are to be upgraded, and initially a preference value of 0.5 has been assigned to this tuple. In the relaxed constraint c_2 we now associate a preference value of 0.75 with this tuple.

The resulting d -relaxation of the problem P is $P'_1 = \langle C'_1, con \rangle$ with $C'_1 = \{c_1, c_{25}, c_3\}$. Table 3.4 shows the solution to the combination of the constraints, $p'_1 c = c_1 \otimes c_{25} \otimes c_3$. Note that the tuple with the best associated semiring value is: $def_{pc_1}^p(\langle 0, 2, 2 \rangle) = 0.75$, with $MSolV(P'_1) = \{0.75\}$. Thus $d(P'_1) = 0 \times_d 1 \times_d 0 = \max(0, 1, 0) = 1$.

Table 3.4: Definition of Constraint p'_1c .

t	$def_{p'_1c}^p(t)$
$\langle 0, 2, 2 \rangle$	0.75
$\langle 1, 2, 2 \rangle$	0.5
$\langle 2, 2, 2 \rangle$	0.5
$\langle 3, 2, 2 \rangle$	0.5
all other tuples	0

This means that our abstract solution is good enough, and the manager can raise the star rating of the hotel chain.

Note that if we select the constraint c_{2_7} to replace the constraint c_2 , we will also be able to find a good enough solution with a preference value of 0.75 and with a difference value of 1.

3.4 Related Work

Bistarelli et al. [6] use the semiring-based framework to model partial CSPs: they show how to use a semiring to represent a notion of distance between a solution and a problem.

It has also been shown that tradeoffs between user preferences (if all requirements cannot be met) can be modelled as additional constraints. Bistarelli et al. [8] present a framework where “tradeoffs” between preferences are modelled in the semiring framework. Our work can be seen as a form of tradeoff where the added and removed constraints involve the same variables.

Ghose & Harvey [27] extended the SCSP framework by specifying a metric for each constraint in addition to the preference values associated with the tuples of values. The metric provides real valued differences between the preference values which are used to measure a real-valued distance between a solution to a SCSP and some desired solution that is regarded to be good enough. Metric SCSPs are similar to our proposal in the sense that both frameworks allow us to establish whether a solution is regarded

as being good enough. Both approaches obtain a measure of the deviation required from a problem P to a relaxation of p that has a good enough solution. In our proposal the measure of deviation is given in terms of a semiring value.

3.5 Conclusion

We have proposed an extension to the SCSP framework for solving SCSPs, where a relaxation of a SCSP is constructed if the solution for the original SCSP is not good enough. We define a suitable relaxation of the SCSP by adjusting the preferences associated with the tuples of some of the constraints of the original SCSP. An additional semiring structure associates difference values with each relaxed constraint, such that alternative relaxations of a problem can be compared in terms of their difference from the original problem.

Our future work will focus on computational aspects of this process. When a solution to a SCSP P is not good enough, we use a set of cut-off values, LB , to define a threshold that should be reached. We need only consider relaxations to constraints of P that have the potential to form a good enough solution. Such relaxed constraints can be found by looking for at least one tuple in the original constraint with a preference value that is not in the set LB and then raise it so that it has a preference value in LB . We plan to develop efficient algorithms to find suitable subsets of relaxations, and to develop techniques to calculate the best relaxation for a SCSP efficiently.

Chapter 4

A Relaxation of a Semiring Constraint Satisfaction Problem using Combined Semirings

Sometimes the maximal solution to a SCSP is not good enough, and then we want to change the constraints such that we solve a problem that is slightly different from the original problem but has an acceptable solution. In Chapter 3, we proposed a relaxation of a SCSP where we define a measure of difference (a semiring value of a second semiring) between the original SCSP and a relaxed SCSP.

In this chapter, we show how the two semiring structures can be combined into a single semiring structure. Recall that the semiring values are associated with tuples of the constraints to present preferences, while the second semiring's values are associated with relaxed constraints to present differences between original constraints and their relaxed versions.

This combined semiring structure may allow us to use existing SCSP tools to solve Combined Semiring Relaxations of SCSPs. This work appeared in Leenen et al. [42].

4.1 A Combined Semiring

In this section, we show how to combine the preference value semiring structure and the difference value semiring structure into a single semiring structure. A constraint c together with a c -weakened constraint c_r (see Definition 13), can be mapped to a member of the combined semiring. We thus define a Combined Semiring Relaxation

of a SCSP using a single semiring.

In a d -relaxation $P' = \langle C', \text{con} \rangle$ of an original SCSP $P = \langle C, \text{con} \rangle$, every constraint $c' \in C'$ is a c -weakened version of a constraint $c \in C$ (see Definition 16). Suppose every constraint $c' \in C'$ has k value tuples. Then c' has k preference values associated with its tuples, as well as one difference value. Thus, a constraint c' is described by an ordered pair of semiring values: k preference values and one difference value.

In the following definition, we define the construction of the combination of two of these ordered pairs of semiring values.

Definition 20 A semiring $S_U \langle U, \oplus_U, \otimes_U, \mathbf{0}_U, \mathbf{1}_U \rangle$ with

$$U = \{ \langle \langle a_1, \dots, a_k \rangle, b \rangle \mid a_i \in A, \text{ and } b \in B \}$$

for some fixed non-negative integer k , is a *Combined Semiring* of semirings $S_A = \langle A, \oplus_A, \otimes_A, \mathbf{0}_A, \mathbf{1}_A \rangle$ and $S_B = \langle B, \oplus_B, \otimes_B, \mathbf{0}_B, \mathbf{1}_B \rangle$ if the statements below hold. Let $u_1, u_2 \in U$, with $u_1 = \langle \langle a_{1_1}, \dots, a_{1_k} \rangle, b_1 \rangle$ and $u_2 = \langle \langle a_{2_1}, \dots, a_{2_k} \rangle, b_2 \rangle$, then

- The multiplicative operator \otimes_U is defined as follows.

$$u_1 \otimes_U u_2 : \langle \langle a_{1_1} \otimes_A a_{2_1}, \dots, a_{1_k} \otimes_A a_{2_k} \rangle, b_1 \otimes_B b_2 \rangle.$$

- The additive operator \oplus_U is defined as follows.

$$u_1 \oplus_U u_2 : \langle \langle a_{1_1} \oplus_A a_{2_1}, \dots, a_{1_k} \oplus_A a_{2_k} \rangle, b_1 \oplus_B b_2 \rangle.$$

- The worst and best elements are:

$$\mathbf{0}_U = \langle \langle a_1, \dots, a_k \rangle, b \rangle \text{ such that } b = \mathbf{0}_B \text{ and every } a_i = \mathbf{0}_A \text{ for } i = \{1, \dots, k\}, \text{ and}$$

$$\mathbf{1}_U = \langle \langle a_1, \dots, a_k \rangle, b \rangle \text{ such that } b = \mathbf{1}_B \text{ and every } a_i = \mathbf{1}_A \text{ for } i = \{1, \dots, k\}.$$

A pre-order \leq_U over the set U is defined as $u_1 \leq_U u_2$ iff $b_1 \leq_B b_2$.

In the next definition, we describe how to map a relaxed constraint and relaxed constraint pair to a combined semiring value: the first coordinate of the combined semiring value contains a preference value for every tuple over all the variables in the type of the problem, and the second coordinate of the combined semiring value is the difference value between the original constraint and the relaxed constraint.

Definition 21 Let $P = \langle C, con \rangle$ be a SCSP over a constraint system $CS = \langle S_p, D, V \rangle$ with $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$, and $P' = \langle C', con \rangle$ be a d -relaxation of P with $S_d = \langle A_d, +_d, \times_d, \mathbf{0}, \mathbf{1} \rangle$. Suppose $S_U = \langle U, +_U, \otimes_U, \mathbf{0}_U, \mathbf{1}_U \rangle$ is a Combined Semiring of S_p and S_d . A *Combined Semiring Relaxation* of P is a tuple $\langle P', g \rangle$ where $g : C \times C' \rightarrow U$, i.e. for every $c = \langle def_c^p, con_c \rangle \in C$ and every c -weakened constraint $c_r \in C'$, we have $g(\langle c, c_r \rangle) = u_{cr}$ with $u_{cr} \in U$.

Assume all tuples of values from D are strictly ordered. Let $u_{cr} = \langle Pref_{cr}, b_{cr} \rangle$, where $b_{cr} \in A_d$ is the difference value associated with the constraint c_r , and $Pref_{cr} = \langle a_{cr_1}, \dots, a_{cr_k} \rangle$ where $a_{cr_i} \in A_p$, for $i = \{1, \dots, k\}$, are the preference values associated with the constraint c_r projected over the set of variables con .

Note that the coordinates in the set $Pref_{cr}$ are the preferences values associated with the k tuples in the relaxed constraint (over the variables in the type of the problem) and the value b_{cr} represents the difference between the relaxed constraint c_r and the original constraint c .

A solution and a maximal solution to a Combined Semiring Relaxation can now be defined.

Definition 22 Given a Combined Semiring Relaxation $RP = \langle P', g \rangle$ of a SCSP $P = \langle C, con \rangle$ and a d -relaxation $P' = \langle C', con \rangle$ with $S_U = \langle U, +_U, \otimes_U, \mathbf{0}_U, \mathbf{1}_U \rangle$, where $g : C \times C' \rightarrow U$, the *solution of the Combined Semiring Relaxation* RP is a constraint defined as the set $ComRSol(RP) = (\bigotimes C')$ where $g(\bigotimes C, \bigotimes C') = u_{cr}$.

Recall from Definition 9 that a solution to a SCSP is a single constraint formed by the combination of all the constraints in the SCSP. Similarly, in Definition 22 above, the solution to a Combined Semiring Relaxation is a single constraint, and the combining function g is also defined.

Definition 23 Given a Combined Semiring Relaxation $RP = \langle P', g \rangle$ of a SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$ with $P' = \langle C', con \rangle$, a d -relaxation of P , and $S_U = \langle U, +_U, \otimes_U, \mathbf{0}_U, \mathbf{1}_U \rangle$, $g : C \times C' \rightarrow U$, $g(\bigotimes C, \bigotimes C') = u_{cr} = \langle Pref_{cr}, b_{cr} \rangle$

with $Pref_{cr} = \langle a_{cr_1}, \dots, a_{cr_k} \rangle$. S_U is the combined semiring of $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$ and $S_d = \langle A_d, +_d, \times_d, \mathbf{0}, \mathbf{1} \rangle$. The *maximal solution of the Combined Semiring Relaxation* RP is the set

$$ComRMSol(RP) = \{ \langle t, a \rangle \mid a \in Pref_{cr} \text{ and there is no tuple } \langle t', a' \rangle \text{ such that } a <_{S_p} a' \}.$$

$$\text{Let } ComRMSolV(RP) = \{ a \mid \langle t, a \rangle \in ComRMSol(RP) \}.$$

The set $ComRMSol(RP)$ contains maximal solutions of the Combined Relaxation RP and their associated preference values, while the set $ComRMSolV(RP)$ contains the preference values of the maximal solutions to RP .

We now define a set that contains Combined Semiring Relaxations with solutions that are good enough and that are minimally different from the original problem P .

Definition 24 Given a SCSP $P = \langle C, con \rangle$, let

$$ComR(P) = \{ RP \mid RP \text{ is a Combined Semiring Relaxation of } P \}.$$

Suppose for $RP = \langle P', g \rangle \in ComR(P)$ with $P' = \langle C', con \rangle$, we have $g(\bigotimes C, \bigotimes C') = u_{cr} = \langle Pref_{cr}, b_{cr} \rangle$, and $Pref_{cr} = \langle a_{cr_1}, \dots, a_{cr_k} \rangle$. Let

$$ComR_{LB}(P) = \{ RP \in ComR(P) \mid ComRMSolV(RP) \cap LB \neq \emptyset, \text{ and there is no } PR' \in ComR(P) \text{ such that } b_{cr} <_{S_d} b_{cr'} \},$$

and let

$$ComRMSol_{LP}(P) = \{ \langle t, a \rangle \mid \langle t, a \rangle \in ComRMSol(PR) \text{ and } PR \in ComR_{LB}(P) \}$$

The set $ComR_{LB}(P)$ contains all the maximal Combined Relaxations of P that have maximal solutions that are good enough. The set $ComRMSol_{LP}(P)$ contains the actual solutions of the members of $ComR_{LB}(P)$ with their associated preference values.

4.2 The Hotel Chain Example - Part 5

Consider the hotel example introduced in Sections 2.5.1 and 2.5.5. Recall that the maximal solution was not regarded as being good enough, and we presented a relaxation in Section 3.3. We chose $S_d = \langle \{0, 1, 2, 3, 4, 5\}, min, max, 5, 0 \rangle$ and considered the d -relaxation, $P' = \langle C', \{X, Y, Z\} \rangle$ with $C' = \{c_1, c_{2s}, c_3\}$.

Table 4.1: Definitions of the constraints in the set C'

t	$def_{c_1}^p(t)$	$def_{c_{25}}^p(t)$	$def_{c_3}^p(t)$
$\langle 0, 0 \rangle$	0	0	0
$\langle 0, 1 \rangle$	0.25	0.25	0.25
$\langle 0, 2 \rangle$	0.75	0.75	0.75
$\langle 0, 3 \rangle$	1	1	1
$\langle 1, 0 \rangle$	0.25	0.25	0.25
$\langle 1, 1 \rangle$	0.25	0.25	0.25
$\langle 1, 2 \rangle$	0.5	0.5	0.5
$\langle 1, 3 \rangle$	0.5	0.5	0.5
$\langle 2, 0 \rangle$	0.5	0.5	0.5
$\langle 2, 1 \rangle$	0.5	0.5	0.5
$\langle 2, 2 \rangle$	0.5	0.75	0.5
$\langle 2, 3 \rangle$	0.25	0.25	0.25
$\langle 3, 0 \rangle$	1	1	1
$\langle 3, 1 \rangle$	0.5	0.5	0.5
$\langle 3, 2 \rangle$	0.5	0.5	0.5
$\langle 3, 3 \rangle$	0	0	0

Table 4.1 shows the definition for the constraints in the set C' . The difference values for the three constraints in C' are 0, 1, and 0, respectively.

We can express the problem as a Combined Semiring Relaxation, $\langle P', g \rangle$. Every constraint and its relaxation can be mapped to a semiring value in the combined semiring, S_U . In order to map a constraint and a relaxed constraint to a semiring value, we have to project the constraint tuples of every constraint over all the variables in the type of the problem. This is done by calculating the projection of each constraint in the set C' over the set $\{X, Y, Z\}$ (see Definition 8). There are 64 tuples for each constraint in our problem, and they are shown in Table 4.2.

In the case of constraint c_1 , we will have the tuples

$$\langle 0, 0, 0 \rangle = \langle 0, 0, 1 \rangle = \langle 0, 0, 2 \rangle = \langle 0, 0, 3 \rangle = 0, \text{ and}$$

$$\langle 0, 1, 0 \rangle = \langle 0, 1, 1 \rangle = \langle 0, 1, 2 \rangle = \langle 0, 1, 3 \rangle = 0.25,$$

and so on. Assuming a lexicographical ordering of the tuples of each constraint, we have

$$g(c_1, c_1) = \langle \langle 0, 0, 0, 0, 0.25, 0.25, 0.25, 0.25, 0.75, \dots, 0.5, 0, 0, 0, 0 \rangle, 0 \rangle,$$

Table 4.2: Definition of the constraints in C' projected over the set $\{X, Y, Z\}$

t	$def_{c_1}^p(t)$	$def_{c_{25}}^p(t)$	$def_{c_3}^p(t)$	t	$def_{c_1}^p(t)$	$def_{c_{25}}^p(t)$	$def_{c_3}^p(t)$
$\langle 0, 0, 0 \rangle$	0	0	0	$\langle 2, 0, 0 \rangle$	0.5	0	0
$\langle 0, 0, 1 \rangle$	0	0.25	0.25	$\langle 2, 0, 1 \rangle$	0.5	0.25	0.5
$\langle 0, 0, 2 \rangle$	0	0.75	0.75	$\langle 2, 0, 2 \rangle$	0.5	0.75	0.5
$\langle 0, 0, 3 \rangle$	0	1	1	$\langle 2, 0, 3 \rangle$	0.5	1	0.25
$\langle 0, 1, 0 \rangle$	0.25	0.25	0	$\langle 2, 1, 0 \rangle$	0.5	0.25	0.5
$\langle 0, 1, 1 \rangle$	0.25	0.25	0.25	$\langle 2, 1, 1 \rangle$	0.5	0.25	0.5
$\langle 0, 1, 2 \rangle$	0.25	0.5	0.75	$\langle 2, 1, 2 \rangle$	0.5	0.5	0.5
$\langle 0, 1, 3 \rangle$	0.25	0.5	1	$\langle 2, 1, 3 \rangle$	0.5	0.5	0.25
$\langle 0, 2, 0 \rangle$	0.75	0.5	0	$\langle 2, 2, 0 \rangle$	0.5	0.5	0.5
$\langle 0, 2, 1 \rangle$	0.75	0.5	0.25	$\langle 2, 2, 1 \rangle$	0.5	0.5	0.5
$\langle 0, 2, 2 \rangle$	0.75	0.75	0.75	$\langle 2, 2, 2 \rangle$	0.5	0.75	0.5
$\langle 0, 2, 3 \rangle$	0.75	0.25	1	$\langle 2, 2, 3 \rangle$	0.5	0.25	0.25
$\langle 0, 3, 0 \rangle$	1	1	0	$\langle 2, 3, 0 \rangle$	0.25	1	0.5
$\langle 0, 3, 1 \rangle$	1	0.5	0.25	$\langle 2, 3, 1 \rangle$	0.25	0.5	0.5
$\langle 0, 3, 2 \rangle$	1	0.5	0.75	$\langle 2, 3, 2 \rangle$	0.25	0.5	0.5
$\langle 0, 3, 3 \rangle$	1	0	1	$\langle 2, 3, 3 \rangle$	0.25	0	0.25
$\langle 1, 0, 0 \rangle$	0.25	0	0.25	$\langle 3, 0, 0 \rangle$	1	0	1
$\langle 1, 0, 1 \rangle$	0.25	0.25	0.25	$\langle 3, 0, 1 \rangle$	1	0.25	0.25
$\langle 1, 0, 2 \rangle$	0.25	0.75	0.5	$\langle 3, 0, 2 \rangle$	1	0.75	0.5
$\langle 1, 0, 3 \rangle$	0.25	1	0.5	$\langle 3, 0, 3 \rangle$	1	1	0
$\langle 1, 1, 0 \rangle$	0.25	0.25	0.25	$\langle 3, 1, 0 \rangle$	0.5	0.25	1
$\langle 1, 1, 1 \rangle$	0.25	0.25	0.25	$\langle 3, 1, 1 \rangle$	0.5	0.25	0.5
$\langle 1, 1, 2 \rangle$	0.25	0.5	0.5	$\langle 3, 1, 2 \rangle$	0.5	0.5	0.5
$\langle 1, 1, 3 \rangle$	0.25	0.5	0.5	$\langle 3, 1, 3 \rangle$	0.5	0.5	0
$\langle 1, 2, 0 \rangle$	0.5	0.5	0.25	$\langle 3, 2, 0 \rangle$	0.5	0.5	1
$\langle 1, 2, 1 \rangle$	0.5	0.5	0.25	$\langle 3, 2, 1 \rangle$	0.5	0.5	0.5
$\langle 1, 2, 2 \rangle$	0.5	0.75	0.5	$\langle 3, 2, 2 \rangle$	0.5	0.75	0.5
$\langle 1, 2, 3 \rangle$	0.5	0.25	0.5	$\langle 3, 2, 3 \rangle$	0.5	0.25	0
$\langle 1, 3, 0 \rangle$	0.5	1	0.25	$\langle 3, 3, 0 \rangle$	0	1	1
$\langle 1, 3, 1 \rangle$	0.5	0.5	0.25	$\langle 3, 3, 1 \rangle$	0	0.5	0.5
$\langle 1, 3, 2 \rangle$	0.5	0.5	0.5	$\langle 3, 3, 2 \rangle$	0	0.5	0.5
$\langle 1, 3, 3 \rangle$	0.5	0	0.5	$\langle 3, 3, 3 \rangle$	0	0	0

$$g(c_2, c_{2_5}) = \langle \langle 0, 0.25, 0.75, 1, 0.25, 0.25, 0.5, 0.5, \dots, 0.25, 1, 0.5, 0.5, 0 \rangle, 1 \rangle, \text{ and}$$

$$g(c_3, c_3) = \langle \langle 0, 0.25, 0.75, 1, 0, 0.25, 0.75, 1, 0, \dots, 0, 1, 0.5, 0.5, 0 \rangle, 0 \rangle.$$

We calculate

$$g(\bigotimes C') = g(c_1, c_1) \otimes_u g(c_2, c_{2_5}) \otimes_u g(c_3, c_3)$$

$$= \langle \langle 0, 0, 0, 0, 0, 0.25, 0.25, 0.25, 0, 0.25, 0.75, \dots, 0.5, 0.5, 0, 0, 0, 0, 0 \rangle, 1 \rangle$$

to find the maximal solution, $\{\langle \langle 0, 2, 2 \rangle, 0.75 \rangle\}$.

This means that our maximal solution is good enough, and the manager can raise the star rating of the hotel chain to a five-star rating by selecting the only tuple in the maximal solution as the solution.

4.3 Conclusion

A relaxation to an SCSP is found by adjusting the preferences associated with the tuples of some of the constraints (i.e. semiring values of the first semiring) of the original SCSP. In other words, the constraints of the original problem are relaxed until the resulting problem has a satisfactory solution. Difference values (i.e. semiring values from a second semiring) are associated with each relaxed constraint so that different relaxations of a problem can be compared in terms of the difference between a relaxation and the original problem.

In this chapter, we showed how to combine the two semirings into a single semiring in the context of Combined Semiring Relaxations. If the preference value associated with the solution of a SCSP is not regarded as good enough, a suitable relaxation of the SCSP that has a good enough solution is found.

The combined semiring allows us to rely on existing techniques for solving SCSPs.

Chapter 5

Weighted Semiring Max-SAT

A significant amount of work has been devoted to solving propositional satisfiability (SAT) problems, and specifically the well-known maximum satisfiability problem (Max-SAT) [1, 63]. There is also continuing interest in translations between CSPs and SAT problems [4, 24, 59], with the objective of using the existing satisfiability algorithms to solve CSPs. This prompted us to explore the modification of algorithms for solving Max-SAT problems, into algorithms for solving SCSPs.

In this chapter, we modify the support encoding of a CSP into SAT by Gent [24], in order to translate a SCSP into a variant of the Weighted Max-SAT Problem, which we call a Weighted Semiring Max-SAT problem (WS-Max-SAT). We present a local search algorithm that is a modification of the GSAT algorithm for solving Max-SAT [55]. Our encoding results in propositional clauses whose structure is exploited in our algorithm to solve WS-Max-SAT.

The next section contains an introduction to satisfiability problems and the algorithms to solve them, including the well-known GSAT algorithm to solve Max-SAT (Section 5.1.1). We describe the transformation of a SCSP into a WS-Max-SAT problem in Section 5.2, and extend the GSAT algorithm into a GSAT-based algorithm to solve WS-Max-SAT in Section 5.3. Section 5.3.2 contains experimental results, and we conclude in the final section.

Note that this work appeared in Leenen et al [38].

5.1 SAT, Max-SAT and Weighted Max-SAT

Propositional satisfiability is the problem of deciding if there is an assignment of values for the variables in a propositional formula that makes the formula true, i.e. to determine if the given propositional formula has a model (is satisfiable). If no such an assignment exists, the formula is regarded as being unsatisfiable. The abbreviation SAT is commonly used to refer to Boolean or propositional satisfiability.

In the *maximum satisfiability* problem (Max-SAT) the aim is to find a variable assignment that maximises the number of satisfied clauses of a SAT problem.

In *Weighted Max-SAT* [57], a weight is assigned to each clause and the goal becomes to maximise the total weight of the satisfied clauses. Let $C = \{C_1, C_2, \dots, C_m\}$ be a set of clauses involving n Boolean variables, x_1, x_2, \dots, x_n , that can take the value *true* or *false*, and w_i with $i = 1, \dots, m$, is the weight assigned to each clause C_i . The objective function is defined as

$$f(x) = \sum_{i=1}^m w_i \cdot I(C_i) ,$$

where $I(C_i)$ is equal to the value *one*, if and only if the clause C_i is satisfied, and otherwise it is equal to the value *zero*. A solution to a Weighted Max-SAT problem is an assignment of values to the variables that maximises the objective function.

When we refer to Max-SAT in the remainder of this chapter, we assume the weighted version.

5.1.1 Algorithms to solve SAT and Max-SAT

We give a brief summary of SAT algorithms below, but Gu et al. [28] can be consulted for a general overview.

An exact algorithm guarantees an optimal solution in finite time. The best-known exact solvers follow either a satisfiability-based approach or a branch and bound approach. In the latter approach, most of the solvers are simple depth-first branch and bound type extensions of back-tracking algorithms, for which good quality lower bounds

exist. Some examples of these type of solvers are the Lazy solver [1], MaxSatz [43], MiniMaxSat [31], the Weighted CSP-based solver of De Grivey et al. [14], and MaxSolver [63]. (Section 2.3.1 contains a description of a basic branch-and-bound algorithm.)

In the satisfiability approach, each Max-SAT problem is converted into a number of SAT problems, and a SAT solver is used to solve these SAT problems. Examples of this type of solver are ChaffBS and ChaffLS [21], and SAT4J-MaxSAT [37].

Sometimes problems have to be solved under time constraints and then approximation algorithms such as *local search algorithms* or solution construction algorithms may be useful. These algorithms can usually not guarantee to find an optimal solution in finite time, i.e. they are incomplete.

The main approach used in local search algorithms for SAT and Max-SAT are based on the GSAT and WalkSAT algorithms. Although these algorithms were designed to solve SAT problems, they can be used to solve Max-SAT problems as well.

Local search algorithms start from some given solution and try to find a better solution in the neighbourhood of the current solution. Consult Stützle et al. [57] for a review of local search algorithms for Max-SAT. Some of the best performing local search SAT solvers are presented in Pham et al. [49] and Ishtaiwi et al. [32].

In local search algorithms for Max-SAT, the search space usually contains the set of all possible truth value assignments for all the variables in the given problem. Most of these algorithms are based on a 1-flip neighbourhood relation [57]: two truth value assignments are neighbours if they differ in the truth value of exactly one variable. A *variable flip* is the action of changing the truth value assigned to the propositional variable under consideration.

Local search algorithms for Max-SAT mostly have an objective function which is defined as the sum of the weights of the clauses that are satisfied under the given variable assignment [57]. These algorithms traverse the search space randomly but with the aim of maximising the objective function. (Maximising the weight of the satisfied clauses is equivalent to minimising the weight of the unsatisfied clauses). Local search algorithms for SAT and Max-SAT mainly differ in their implementation of the step that chooses

the next variable to be flipped.

Most local search algorithms are incomplete, and this can be attributed to the randomness of the exploration of the search space. A common disadvantage of local search algorithms is that they can get trapped in local minima and plateau regions of the search space. One way to resolve early entrapments in the search is a random restart: the search is reinitialised after a fixed number of steps if no solution has been found.

We now describe the well known GSAT and WalkSAT algorithms in more detail below. Later on in this chapter, we modify the GSAT algorithm for Max-SAT to solve our Weighted Semiring Max-SAT problem.

The GSAT algorithm

The GSAT algorithm was developed in 1992 by Selman, Levesque, and Mitchell [55]. GSAT is greedy algorithm in the sense that it tries to maximise the number of satisfied clauses by selecting different variable assignments based on the *score* of a variable x under the current assignment α . The score of a variable is defined as the difference between the weight of the clauses *unsatisfied* by α and the assignment obtained by flipping x in α [57].

The GSAT algorithm generates a random truth assignment. It then flips a variable with a maximal score. Note that if there is more than one variable that will give a maximal increase in the number of satisfied clauses if it is to be flipped, then one of these variables is chosen randomly. This random choice of variable makes it unlikely that the same sequence of changes is repeated by the algorithm.

In GSAT we have to initialise two parameters, *MaxFlips* and *MaxSteps*, which respectively controls the number of flips the algorithm will perform before giving up and restarting (randomly), and the number of times the search can be restarted before halting.

The GSAT algorithm is reproduced as Algorithm 1.

There are a number of variations on GSAT, for example, GSAT with Random Walk (GWSAT) [54], GSAT with tabu search [29], and HSAT [25].

Algorithm 1 The GSAT Algorithm

Require:

A set of clauses C ;
 $MaxFlips$;
 $MaxSteps$.

Ensure:

A satisfying truth assignment for C .
1: **for** $i = 1$ **to** $MaxSteps$ **do**
2: Let α = a randomly generated truth assignment for C ;
3: **for** $j = 1$ **to** $MaxFlips$ **do**
4: **if** α satisfies C **then**
5: **return** α
6: **end if**
7: p = a propositional variable such that a change in its truth assignment gives the largest increase in the total number of clauses of C that are satisfied by α .
8: $\alpha = \alpha$ with the truth assignment of p reversed;
9: **end for**
10: **end for**
11: **return** no satisfying assignment found;

The WalkSAT Algorithm

WalkSAT was originally introduced in 1994 by Selman, Kautz, and Cohen [54], and later formally defined as an algorithm by McAllester, Selman, and Kautz [45].

In the WalkSAT approach, a variable is chosen during a 2-stage process: a currently unsatisfied clause c' is chosen randomly in the first stage, and then one of the variables in c' is chosen to be flipped in the second phase. The *score* of a variable (in the second phase) only counts the weights of the clauses that are currently satisfied, but will become unsatisfied by flipping a given variable. Stützle [57] describes the process of selecting a variable to be flipped:

- If there is a variable with a score of 0 in clause c' (selected in phase 1), this variable will be flipped. In this case clause c' can be satisfied without breaking any other clauses.
- If no such a variable exists:
 - With a certain probability p , a variable (in clause c') with a maximal score is chosen.

- In the remaining cases, a variable is chosen randomly from c' to be flipped.

WalkSAT generally performs better than GSAT for solving SAT [54]. There are a number of variants on WalkSAT, for example, WalkSAT with tabu search, Novelty, and R-Novelty [45].

5.2 Weighted Semiring Max-SAT Problems

In this section, we show how to encode a SCSIP into a variant of a Weighted Max-Sat problem which we call a Weighted Semiring Max-SAT problem (WS-Max-SAT). Our encoding of a SCSIP into a WS-Max-SAT is based on Gent's support encoding of a CSP into a Boolean SAT [24], which we describe below.

5.2.1 Translation of a CSP into a SAT problem

Gent introduced a method to translate a CSP into a SAT problem which he called the *support encoding* [24]. He only considers the encoding of binary constraints (i.e. constraints that have at most two variables), but sees no theoretical problem extending his work to the non-binary case. For the non-binary case, translated problems may become too large for practical purposes.

In support coding, Gent's main idea is the encoding of the *support* for a value into clauses. This idea was first introduced by Kasif [34], and is different to the idea of encoding *conflicts* into clauses. The latter approach is described as *direct encoding* by Walsh [59].

The following paragraphs contain a summary of the description Gent provides in [24]. Assume a CSP has m constraints and n variables, each with a domain size d . The purpose of these assumptions is simply to reduce the number of subscripts. There is a SAT variable for each value v of each CSP variable Y . The SAT variable $X_{Y,v}$ means "the SCSIP variable Y takes the value v " so that $X_{Y,v}$ is *false* if $Y \neq v$, and *true* if $Y = v$.

In the support encoding according to Kasif [34], the support of a value v for a variable Y , that is for $Y = v$, across a constraint, is the set of values of the other variable which allow $Y = v$. Suppose a constraint contains variables Y and Z , and the values w_1, w_2, \dots, w_k for the variable Z are the supporting values for variable Y to take the value v , that is $Y = v$, then the support clause is:

$$X_{Z,w_1} \vee X_{Z,w_2} \cdots \vee X_{Z,w_k} \vee \neg X_{Y,v}$$

One support clause exist for each pair of variables (Y, Z) involved in a constraint, and for each value in the domain of Y . A similar clause is required in each “direction”: one for the pair (Y, Z) and one for the pair (Z, Y) . This results in a total of $2md$ constraints of size d .

Apart from the support clauses, the support encoding requires two additional sets of clauses:

- *at-least-one* clauses to encode that each CSP variable takes at least one value; and
- *at-most-one* clauses to encode that each CSP variable takes at most one variable.

5.2.2 Translation of a SCSP into a WS-Max-SAT problem

We now extend the support encoding for CSPs to SAT, into a encoding to translate a SCSP into a variant of a Weighted Max-SAT problem. Every propositional clause in our encoding will receive a semiring value from the semiring associated with the SCSP as a weight.

Suppose we have a SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$, such that every variable $Y_i \in V$ has $D = \{w_1, \dots, w_d\}$ as its domain, and $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$. Assume the cardinality of V is n , the cardinality of C is m , and that there are s semiring values in the set A_p . Every constraint $c \in C$, is such that $c = \langle def_c^p, con_c \rangle$ with r the cardinality of con_c , and $con_c \subseteq V$.

For every SCSP variable $Z \in V$, we have a total of d WS-Max-SAT variables: the WS-Max-SAT variable X_{Z,w_j} is *true* if the SCSP variable Z has the value $w_j \in D$ with

$j \in \{1, \dots, d\}$, and *false* otherwise.

Our *WS-Max-SAT encoding* (encoding of the SCSP P) has three kinds of clauses:

1. *at-least-one* clauses to ensure that each SCSP variable is assigned at least one value from its domain;
2. *at-most-one* clauses to ensure that each SCSP variable is assigned at most one value from its domain;
3. *generalised support* clauses to represent all the possible value pairs (tuples) with their associated semiring values for every constraint. For every constraint we have at most s such clauses, and every clause receives the associated semiring value as its weight.

In our case, it is not necessary to restrict the description of our encoding to binary constraints.

We encode the support values for each constraint by explicitly representing all the allowed tuples for a constraint in different generalised support clauses, according to the semiring value associated with each tuple: all the tuples of a constraint with the same associated semiring value are represented in the same generalised support clause.

We now describe each of the three types of clauses in more detail.

- For every SCSP variable $Z \in V$ we have one *at-least-one* clause:

$$X_{Z,w_1} \vee X_{Z,w_2} \vee \dots \vee X_{Z,w_d}.$$

- For every SCSP variable $Z \in V$ we have a total of $0.5[d(d-1)]$ *at-most-one* clauses: a clause, $\neg X_{Z,w_i} \vee \neg X_{Z,w_j}$, for every pair (w_i, w_j) , where $1 \leq i < j \leq d$.
- For every constraint, we have at most s generalised support clauses: one generalised clause to represent all the tuples that have the same associated semiring value.

Suppose $A_p = \{p_1, p_2, \dots, p_s\}$, then every constraint $c \in C$ has at most s generalised support clauses, $\text{sup}_c^{p_k}$ with $k = 1, \dots, s$.

A generalised support clause, $sup_c^{p_k}$, for a constraint $c = \langle def_c^p, con_c \rangle$, is a disjunction of the representation of every tuple t (with arity r) of the constraint c that has the associated semiring value p_k .

Let such a tuple $t = \langle X_{Z_1, v_1}, \dots, X_{Z_r, v_r} \rangle$ with $Z_i \in con_c$ and $v_i \in D$ for $i = 1 \dots r$, be represented by the conjunction of all these literals:

$$X_{Z_1, v_1} \wedge X_{Z_2, v_2} \wedge \dots \wedge X_{Z_r, v_r}$$

Keep in mind that con_c is the type of the variable, i.e. the set of variables that occur in the constraint c , and that its arity is r . Also note that our support clauses are disjunction of conjuncts and thus, strictly speaking, not clauses. This is why we call our support clauses *generalised* support clauses.

This means that every constraint c has at most s generalised support clauses, $sup_c^{p_1}, sup_c^{p_2}, \dots, sup_c^{p_s}$, where each generalised support clause, $sup_c^{p_k}$, is a disjunction of a number of conjuncts. The number of conjuncts in a particular generalised support clause, $sup_c^{p_k}$, depends on the number of tuples the constraint c has with an associated semiring value of p_k , for $k = 1, 2, \dots s$. (Every conjunction represents one value tuple of the constraint c .)

Now that we know which clauses we need, we are ready to formulate our variant of a Weighted Max-SAT problem.

We want all the clauses in our encoding, with the exception of the generalised support clauses, to be satisfied: we call these clauses the *hard* clauses. Another way to express this requirement, is to say that the negation of each hard clause must remain unsatisfied. To accomplish this, we include the negation of the hard clauses in our encoding, and assign the minimum semiring value, $\mathbf{0}$, as the weight of the negation of each hard clause.

If a hard clause is satisfied by a truth assignment, its negation is unsatisfied and this unsatisfied clause contributes the minimum semiring value to the total weight. Keep in mind that we combine semiring values by using the multiplicative operator of a semiring, and that the multiplicative operator's absorbing element is the minimum semiring value.

Note that at most one generalised support clause for each constraint can be satisfied by a truth assignment if we assume that the *at-most-one* and *at-least-one* clauses are all satisfied. We regard the generalised support clauses as soft clauses: the weight assigned to each generalised support clause is the semiring value associated with each tuple represented in that generalised support clause.

In the next definition, we describe a solution tuple for an encoding of a SCSP P .

Definition 25 Given a SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$ with $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$. Let Cl be the set of clauses that represents a WS-Max-SAT encoding of P , and let the set Sup contain all the generalised support clauses in the encoding, with $Sup \subset Cl$. α is some truth assignment for the clauses in Cl .

Then $t_{sol}^\alpha = \langle X_{Z_1, val_1}, \dots, X_{Z_n, val_n} \rangle$ with $V = \{Z_1, Z_2, \dots, Z_n\}$ and $val_i \in D$ for $i = 1, \dots, n$, is a solution tuple for the encoding iff the following holds:

- For every constraint $c \in C$ exactly one of its generalised support clauses, $sup_c^{p_k}$ with $1 \leq k \leq s$ and $p_k \in A_p$, is satisfied by α . This generalised support clause represents a disjunction of tuples of this constraint. Each tuple is a conjunction of WS-Max-SAT variables, with exactly one tuple, $t_c = \langle X_{C_1, v_1}, \dots, X_{C_r, v_r} \rangle$ with $C_i \in con_c$, $v_i \in D$, $i = 1 \dots r$, such that all $X_{C_i, v_i} = true$.
- If C_i is equal to Z_j where $j = 1, \dots, n$, then $v_i = val_j$.

The semiring value associated with t_{sol}^α is $sr_{sol}^\alpha = sp_1 \times_p \dots \times_p sp_m$ where $sp_c \in A_p$ for $c = 1, \dots, m$, and $sup_c^{sp_c}$ is the generalised support clause satisfied by the truth assignment α for every constraint $c \in C$.

The last bullet point in Definition 25 refers to the case where a WS-Max-SAT variable X_{C_i, v_i} represents the SCSP variable Z_j , and Z_j has been assigned the value $val_j \in D$. Note that semiring value that is associated with a solution tuple for an encoding, is calculated by combining the semiring values associated with each “chosen” (or satisfied) generalised support clause.

We can now define a Weighted Semiring Max-SAT problem.

Definition 26 Given a SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$ with $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$. Suppose Cl is the set of clauses that represents the WS-Max-SAT encoding of P . Let $Cl_h = \{cl_1, \dots, cl_h\} \subseteq Cl$ be the set of clauses that are satisfied by some truth assignment α .

The *Weighted Semiring Maximum Satisfiability problem* is to find

$$\max f(\alpha) = \bigotimes_{i=1}^h w_i$$

where

- $w_i \in A_p$ is the weight assigned to clause $i \in Cl_h$,
- $f(\alpha)$ is maximal with respect to the partial order $<_{S_p}$, and
- $t_{sol}^\alpha = \langle X_{Z_1, val_1}, \dots, X_{Z_n, val_n} \rangle$ with $Z_j \in V$ and val_j for $j = 1 \dots n$, is a solution tuple for the encoding of P .

The ordered pair $\langle \langle Z_1, \dots, Z_n \rangle, f(\alpha) \rangle \in \text{MSol}(P)$.

Let us now consider an example where we start with a simple SCSP, and transform it into a WS-Max-SAT problem.

5.2.3 Example: Transformation of a SCSP into a WS-Max-SAT

Suppose we have a SCSP $P = \langle C, con \rangle$ over the constraint system $\langle S_p, D, V \rangle$, with $C = \{C_1, C_2, C_3\}$, $V = \{A, B, C\}$ and $D = \{1, 2, 3\}$. The three constraints are explicitly defined by $C_1 : A < B$, $C_2 : B < C$, and $C_3 : C < A$.

The semiring is $S_p = \langle \{0, 5, 10\}, \max, \min, 0, 10 \rangle$, and the constraint definitions are given in Table 5.1.

The first column shows the tuples of the constraints: the first coordinate reflects the value of the first variable of a particular constraint, and the second coordinate reflects the value of the second variable of that constraint. The second, third and fourth columns contain the associated semiring values for each of the tuples of the three

Table 5.1: Constraint definitions

t	$def_{C_1}^p(t)$	$def_{C_2}^p(t)$	$def_{C_3}^p(t)$
$\langle 1, 1 \rangle$	5	5	5
$\langle 1, 2 \rangle$	10	10	10
$\langle 1, 3 \rangle$	10	10	10
$\langle 2, 1 \rangle$	0	0	0
$\langle 2, 2 \rangle$	5	5	5
$\langle 2, 3 \rangle$	10	10	10
$\langle 3, 1 \rangle$	0	0	0
$\langle 3, 2 \rangle$	0	0	0
$\langle 3, 3 \rangle$	5	5	5

constraints, respectively. For example, the value tuple $\langle 1, 3 \rangle$ for constraint C_1 represents the case where the variable A has the value 1 and variable B has the value 3, and this tuple's associated preference value, 10, can be found in column 2, row 4.

Value tuples that satisfy a constraint get the highest preference value, 10. Among the tuples that do not satisfy the constraints, we prefer those where the first coordinate equals the second coordinate, for instance $\langle 2, 2 \rangle$ is preferred over $\langle 2, 1 \rangle$. These tuples get a preference value of 5, and all the remaining tuples get the worst value, 0.

The WS-Max-SAT has nine variables: $X_{A,1}, X_{A,2}, X_{A,3}, X_{B,1}, X_{B,2}, X_{B,3}, X_{C,1}, X_{C,2}$, and $X_{C,3}$.

The clauses in the encoding are given below.

At-least-one clauses:

- $X_{A,1} \vee X_{A,2} \vee X_{A,3}$
- $X_{B,1} \vee X_{B,2} \vee X_{B,3}$
- $X_{C,1} \vee X_{C,2} \vee X_{C,3}$

At-most-one clauses:

- $\neg X_{A,1} \vee \neg X_{A,2}, \neg X_{A,2} \vee \neg X_{A,3}, \text{ and } \neg X_{A,1} \vee \neg X_{A,3}$

- $\neg X_{B,1} \vee \neg X_{B,2}, \neg X_{B,2} \vee \neg X_{B,3}, \text{ and } \neg X_{B,1} \vee \neg X_{B,3}$
- $\neg X_{C,1} \vee \neg X_{C,2}, \neg X_{C,2} \vee \neg X_{C,3}, \text{ and } \neg X_{C,1} \vee \neg X_{C,3}$

Generalised support clauses:

- $C_1^{10} : [(X_{A,1} \wedge X_{B,2}) \vee (X_{A,1} \wedge X_{B,3}) \vee (X_{A,2} \wedge X_{B,3})]$
- $C_1^5 : [(X_{A,1} \wedge X_{B,1}) \vee (X_{A,2} \wedge X_{B,2}) \vee (X_{A,3} \wedge X_{B,3})]$
- $C_1^0 : [(X_{A,2} \wedge X_{B,1}) \vee (X_{A,3} \wedge X_{B,1}) \vee (X_{A,3} \wedge X_{B,2})]$
- $C_2^{10} : [(X_{B,1} \wedge X_{C,2}) \vee (X_{B,1} \wedge X_{C,3}) \vee (X_{B,2} \wedge X_{C,3})]$
- $C_2^5 : [(X_{B,1} \wedge X_{C,1}) \vee (X_{B,2} \wedge X_{C,2}) \vee (X_{B,3} \wedge X_{C,3})]$
- $C_2^0 : [(X_{B,2} \wedge X_{C,1}) \vee (X_{B,3} \wedge X_{C,1}) \vee (X_{B,3} \wedge X_{C,2})]$
- $C_3^{10} : [(X_{C,1} \wedge X_{A,2}) \vee (X_{C,1} \wedge X_{A,3}) \vee (X_{C,2} \wedge X_{A,3})]$
- $C_3^5 : [(X_{C,1} \wedge X_{A,1}) \vee (X_{C,2} \wedge X_{A,2}) \vee (X_{C,3} \wedge X_{A,3})]$
- $C_3^0 : [(X_{C,2} \wedge X_{A,1}) \vee (X_{C,3} \wedge X_{A,1}) \vee (X_{C,3} \wedge X_{A,2})]$

In our example, each generalised support clause has its associated semiring value as its weight. The negation of all the remaining (hard) clauses are each given a weight of 0:

1. $\neg X_{A,1} \wedge \neg X_{A,2} \wedge \neg X_{A,3}$
2. $\neg X_{B,1} \wedge \neg X_{B,2} \wedge \neg X_{B,3}$
3. $\neg X_{C,1} \wedge \neg X_{C,2} \wedge \neg X_{C,3}$
4. $X_{A,1} \wedge X_{A,2}$
5. $X_{A,2} \wedge X_{A,3}$
6. $X_{A,1} \wedge X_{A,3}$
7. $X_{B,1} \wedge X_{B,2}$

8. $X_{B,2} \wedge X_{B,3}$

9. $X_{B,1} \wedge X_{B,3}$

10. $X_{C,1} \wedge X_{C,2}$

11. $X_{C,2} \wedge X_{C,3}$

12. $X_{C,1} \wedge X_{C,3}$

We want to find an assignment that satisfies all the hard clauses, or equivalently, an assignment that does not satisfy the negations of the hard clauses, i.e. the clauses that are numbered 1 to 12 above. A truth assignment that satisfies one or more of the generalised support clauses with an associated maximum semiring value, is preferred over one that satisfies generalised support clauses with lower associated semiring values.

We solve this WS-Max-SAT problem later on in this chapter.

5.2.4 Correctness Proof

In this section, we prove that our encoding is correct.

Theorem 2 Let the set of clauses, Cl , be a WS-Max-SAT encoding for the SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$ with $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$. Let $P_{WS-Max-SAT}$ be the WS-Max-SAT problem associated the clauses in Cl , and let α be a truth assignment for $P_{WS-Max-SAT}$. Suppose the constraint $e = Sol(P) = \langle def_e^p, con \rangle$ is a solution for P .

If $t_{sol}^\alpha = \langle X_{Z_1, val_1}, \dots, X_{Z_n, val_n} \rangle$ with $Z_i \in V$ and $val_i \in D$ for $i = 1 \dots n$, is a solution tuple for $P_{WS-Max-SAT}$, and sr_{sol}^α is the semiring value associated with t_{sol}^α , then $t = \langle val_1, \dots, val_n \rangle$ with $Z_i = val_i$ for $i = 1, \dots, n$, is such that $def_e^p(t) = sr_{sol}^\alpha$ (that is, t together with its associated semiring value belongs to the solution of P).

Proof 2 Assume a set of clauses, Cl , that represents a WS-Max-SAT encoding, called $P_{WS-Max-SAT}$, for a SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$

with $S_p = \langle A_p, +_p, \times_p, \mathbf{0}, \mathbf{1} \rangle$. Suppose α is a truth assignment for $P_{WS-Max-SAT}$, and let the tuple $t_{sol}^\alpha = \langle X_{Z_1, val_1}, \dots, X_{Z_n, val_n} \rangle$ with its associated semiring value sr_{sol}^α be a solution for $P_{WS-Max-SAT}$. Suppose $e = \langle def_e^p, con \rangle$ is a solution for P

- For every constraint $c_j \in C$, $j = 1, \dots, m$, there is exactly one generalised support clause satisfied by α , say sup_c^k with $k \in A_p$. Suppose $c_j = c = \langle def_c^p, con_c \rangle$ with $con_c = \{Y_{c_1}, \dots, Y_{c_r}\}$, all $Y_{c_i} \in V$ for $i = 1, \dots, r$. The generalised support clause sup_c^k represents a tuple of the constraint c such that all $X_{Y_{c_i}, val_{c_i}} = true$. Every $Y_{c_i} = Z_j$ for some $j \in \{1, \dots, n\}$, and this means that $Z_j = val_{c_i}$ with $val_{c_i} \in \{val_1, \dots, val_n\}$.

Since $V = \bigcup_{c=1}^m con_c = \{Z_1, \dots, Z_n\}$, the tuple $\langle val_1, \dots, val_n \rangle$ with its associated semiring value is a member of the solution of P .

- In Definition 25, we saw that the tuple t_{sol}^α is calculated by combining a total of m semiring values, each value contributed by one constraint: the generalised support clause of each constraint that is satisfied by the truth assignment α , contributes its associated semiring value. (Recall that there are m constraints in c .)

Since we have established that t_{sol}^α is a solution for $P_{WS-Max-SAT}$, it must be the case that the tuple t has the same associated semiring value than t_{sol}^α .

5.3 A GSAT-based Algorithm to solve WS-Max-SAT

In this section, we extend the GSAT algorithm described in Section 5.1.1 into a GSAT-based algorithm for WS-Max-SAT.

Most SAT and Max-SAT solvers require the propositional clauses to be in conjunctive normal form (CNF). In our encoding, the propositional clauses are highly structured and we do not have to convert them into CNF.

Our adjusted GSAT algorithm is presented in Algorithm 2.

Algorithm 2 GSAT for WS-Max-SAT

Require:

A WS-Max-SAT encoding with a set of clauses, Cl , for a SCSP P ;
 A semiring, S_p ;
 A set of weights, W (from the set of semiring values);
 $MaxFlips$;
 $MaxSteps$.

Ensure:

A satisfying truth assignment for the WS-Max-SAT problem formulated from Cl .

```

1: Initialise  $f_{\alpha_{Best}} = \mathbf{0}$ ;
2: for  $i = 1$  to  $MaxSteps$  do
3:   Let  $\alpha$  = a randomly generated truth assignment for  $Cl$ ;
4:   for  $j = 1$  to  $MaxFlips$  do
5:     if  $f_{\alpha_{BEST}} <_{S_p} f(\alpha)$  then
6:        $\alpha_{BEST} = \alpha$ ;
7:        $f_{\alpha_{BEST}} = f(\alpha)$ ;
8:     end if
9:     if  $f_{\alpha_{BEST}} = \mathbf{1}$  then
10:      return  $\alpha_{BEST}$ 
11:    else
12:       $x = chooseVariable(Cl, \alpha)$ ;
13:       $\alpha = \alpha$  with truth value of  $x$  flipped;
14:    end if
15:  end for
16: end for
17: return  $\alpha_{BEST}$ ;
```

The best truth assignment found up to date is stored in the variable α_{BEST} , and its objective function value, $f(\alpha_{BEST})$, is stored in the variable $f_{\alpha_{Best}}$. Its value is initialised to the worst semiring value.

The first step is to generate a random variable assignment α for the WS-Max-SAT problem. We want to generate truth assignments where the *at-least-one* and the *at-most-one* clauses are satisfied, i.e. we want to make sure that for each $i = 1 \dots d$, exactly one variable $X_{Z,i} = true$. (In other words, the negations of these clauses will be unsatisfied.) This leaves only the generalised support clauses that have to be checked for satisfiability: exactly one generalised support clause can be satisfied for every constraint. We simply search for the generalised support clause with the best associated semiring value for each constraint that is satisfied under the current truth assignment.

The next step is to calculate the objective function value, $f(\alpha)$, compare it to the value of $f_{\alpha_{Best}}$, and store the best value. (Keep in mind that we calculate the value of $f(\alpha)$ by combining the semiring values associated with all the satisfied clauses under the truth assignment α .) If $f_{\alpha_{Best}}$ is equal to the best semiring value the algorithm has found an optimal solution, otherwise it has to flip a variable.

The procedure *chooseVariable* selects the next variable to be flipped by considering the *score* of each variable X in WS-Max-SAT. The score of a variable X in an assignment α is the value of the objective function, $f(\alpha_X)$, if α_X is identical to α except for the truth value assigned to X . There may be more than one variable with a maximum score. In this case, one of the variables with a maximal score is chosen at random to be flipped.

When a WS-Max-SAT variable $X_{Z,w}$ is chosen to be flipped, we want to ensure that we get a new truth assignment such that the *at-least-one* and the *at-most-one* clauses are all satisfied, or rather, such that the negations of these clauses are unsatisfied. Thus we consider two cases:

- The current value of $X_{Z,w} = \text{true}$ and has to be flipped to become *false*. Some variable $X_{Z,v}$ with $v = 1, \dots, d$ and $v \neq w$ is chosen at random to become *true*.
- The current value of $X_{Z,w} = \text{false}$ and has to be flipped to become *true*. Some variable $X_{Z,v}$ with $v = 1, \dots, d$ and $v \neq w$ is currently *true* and is given the value *false*.

After a variable has been flipped, the truth assignment α is updated. If we have made less than *MaxFlips* number of flips, we continue with another iteration, otherwise we randomly generate a new truth assignment. This process continues until an optimal solution is found, or the maximum number of steps has been reached.

5.3.1 Example: Solving a WS-Max-SAT problem

Consider the SCSP that was translated to a WS-Max-SAT problem in Example 5.2.3. We now show how our GSAT-based algorithm (Algorithm 2) can be used to find a

solution.

Suppose we generate the following random truth assignment α_1 :

$$\begin{aligned} X_{A,1} = X_{A,2} = X_{B,1} = X_{B,2} = X_{C,1} = X_{C,3} &= false, \quad \text{and} \\ X_{A,3} = X_{B,3} = X_{C,2} &= true. \end{aligned}$$

This truth assignment satisfies the *at-most-one* and the *at-least-one* clauses, as well as the following three generalised support clauses: c_1^5 , c_2^0 , and c_3^{10} .

The weights (associated semiring values) of the satisfied generalised support clauses are 5, 0, and 10, respectively. The negations of the *at-most-one* and the *at-least-one* clauses are included in the encoding's set of clauses, and they all have the minimum semiring value, 0, as their weight. None of these negated clauses are satisfied by α_1 . We thus apply the semiring's multiplicative operator, *min*, to combine the weights of the three satisfied clauses in the encoding:

$$f(\alpha_1) = \min(5, 0, 10) = 0.$$

Keep in mind that our procedure *chooseVariable* is implemented to flip variables in such a way that it produces a modified assignment where the *at-most-one* and the *at-least-one* clauses are satisfied. In the current assignment α , the variable $X_{C,3}$ is the only variable with a score that is not equal to the minimum semiring variable. The score of $X_{C,3}$ is 5 and it is chosen to be flipped.

This gives a modified assignment, α_2 , which is similar to α_1 except for the values of the variables $X_{C,2}$ and $X_{C,3}$:

$$\begin{aligned} X_{A,1} = X_{A,2} = X_{B,1} = X_{B,2} = X_{C,1} = X_{C,2} &= false, \quad \text{and} \\ X_{A,3} = X_{B,3} = X_{C,3} &= true. \end{aligned}$$

The generalised support clauses c_1^5 , c_2^5 , and c_3^5 are satisfied, and they all contribute a weight of 5. The value of the objective function is:

$$f(\alpha) = \min(5, 5, 5) = 5.$$

The algorithm will continue iterating depending on the values of *MaxSteps* and *MaxFlips*, but the current solution is in fact maximal for the problem.

Table 5.2: The random Fuzzy CSP problems

Set	1	2	3
# domain values	10	10	10
# variables	80	100	120
#constraints	10	10	20

5.3.2 Results of GSAT-based Algorithm

We implemented our algorithm in the C++ programming language, and used an Intel Pentium 4 processor at 2.53GHz with 512MB RAM for our experiments. Refer to Appendix A for information on the random SCSP generator we used to generate random SCSPs.

We solved three sets of randomly generated binary SCSPs. Table 5.2 summarises the characteristics of the problems. Each set of problems contains 100 instances.

In each set of 100 problems, 50 instances have a tightness of 70% and the other 50 instances have a tightness of 90%. A tightness (T) of x% means that (100-x)% of all the tuples have been assigned the best semiring value. All the problems have the following semiring structure, $S_p = \langle \{0, 0.3, 0.5, 0.8, 1\}, max, min, 0, 1 \rangle$.

Note that these problems are instances of fuzzy CSPs because the semiring values form a total order. The algorithm will perform equally well on SCSPs with semirings that form partial orders: the multiplicative operator of a semiring's unit element is the best semiring value, and its absorbing element is the worst semiring value. The algorithm may find a maximal solution that is incomparable with other existing maximal solutions.

All our problems have optimal solutions with the best semiring value, 1, with the exception of three instances in set 3 with a tightness of 90%. These three instances all have optimal solutions with associated semiring values of 0.8.

The results of our experiments are given in Table 5.3. The entries in row two (S=1) show the number of instances (out of a total of 50) for which our algorithm found

Table 5.3: Results for the GSAT-based algorithm

Set	1	1	2	2	3	3
T	70%	90%	70%	90%	70%	90%
S = 1	50	2	50	0	1	0
S = 0.8	0	48	0	49	33	0
S = 0.5	0	0	0	1	16	38
S = 0.3	0	0	0	0	0	12

Table 5.4: Results for instances where a maximal solution was found

Set	1	1	2	2	3	3
T	70%	90%	70%	90%	70%	90%
S = 1	50	2	50	0	1	0
steps	1.34	2.00	1.28	n/a	1	n/a
flips	452.4	836.5	438.5	n/a	329.0	n/a

the optimal solution. Rows three to five show the number of instances for which our algorithm found solutions with an associated semiring value of 0.8, 0.5, and 0.3 respectively. There were no instances where a solution with the minimum semiring value, 0, was found.

Table 5.4 contains information for those instances where the optimal solution (i.e. a solution with an associated semiring value of 1) were found: it shows on average when the solution was found. We show during which step, and after how many flips, the algorithm halted. Each step consists of a maximum of 1000 flips.

Note that our algorithm failed to find any solutions with the maximum associated semiring value for the problems in set 2, with a tightness of 90%, and in set 3, with a tightness of 90%. This is the reason why these cases do not have entries in the last two rows of Table 5.4.

Our GSAT algorithm performs reasonably well for a relatively small number of steps and flips, and this shows that our approach to the solution of SCSPs is viable.

5.4 Conclusion

In this chapter, we showed how to translate a SCSP into a variant of a Weighted Max-SAT problem, which we call a Weighted Semiring Max-SAT problem. The objective is to make use of the efficient algorithms that are available to solve Max-SAT, and modify them to solve WS-Max-SATs. We developed a GSAT-based algorithm to solve WS-Max-SAT, and tested the algorithm on binary SCSPs. The results of our experiments show that this is a viable approach to the solution of moderately sized binary SCSPs.

The next step is to modify some of best available algorithms for Max-SAT. There are several local search algorithms that should be easy to modify for WS-Max-SAT. Recent advances in local search algorithms are interesting to consider: Pham et al. [50] show that local SAT search techniques can be improved by building structural information into the search, and Pham et al. [51] developed a SAT solver that automatically recognise CSP structure hidden in the SAT encodings.

The existing exact algorithms will require more effort to modify, but are likely to produce good results.

Chapter 6

Algorithms to solve SCSPs

Considerable interest has been shown in solving SCSPs, but limited work has been done in building general SCSP solvers. Existing work on SCSP methods include Georget and Codognet's [26] logic programming system, a dynamic programming approach by Bistarelli et al. [5], an abstraction-based algorithm by Bistarelli et al. [10], a prototype solver based on an incomplete local search method by Bistarelli et al. [7], and Wilson's [62] application of decision diagrams. We gave an overview of solution techniques for SCSPs in Section 2.5.6.

In this chapter, we present three branch and bound algorithms we developed to solve SCSPs: a basic branch and bound algorithm, a backjumping algorithm and a forward checking algorithm. These algorithms are described in Section 6.1. In Section 6.2 we present experimental results and a comparison of the performance of our branch and bound algorithm with CON'FLEX [17], a fuzzy CSP solver. Our branch and bound algorithm performs significantly better than CON'FLEX, while the backjumping and forward checking algorithms perform better than the branch and bound algorithm on some problem instances. A part of this work has been published in [38] and the rest appears in [39].

Finally, we present an algorithm for SCSPs that solves a finite sequence of CSPs. This algorithm is given in Section 6.3.

6.1 Branch and Bound Algorithms for solving SCSPs

Freuder and Wallace [20] proposed algorithms for solving maximal constraint satisfaction problems that are analogous to the basic backtracking and local consistency methods for solving CSPs. These algorithms are described in Section 2.3.1. Our algorithms for solving SCSPs are analogous to the Freuder and Wallace algorithms.

In the following section, we present our basic branch and bound algorithm, and in Section 6.1.2 we extend it by adding variable partitioning. In Section 6.1.3 we describe our backjumping algorithm, and our forward checking algorithm follows in Section 6.1.4.

6.1.1 A Basic Branch and Bound Algorithm for SCSPs

Branch and bound (BnB) is a general search method introduced by Land and Doig [36] in 1960. There is a description of a classical branch and bound algorithm in Section 2.3.1.

Freuder and Wallace [20] developed a branch and bound algorithm for solving maximal satisfaction problems. We extend their algorithm into a branch and bound algorithm for solving SCSPs. Our algorithm (Algorithm 3) is based on an algorithm we presented in a previous paper [38].

In the maximal constraint satisfaction BnB algorithm, a search path does not fail until enough inconsistencies have been counted such that a cutoff bound is reached. The goal is to satisfy a maximal number of constraints. The Freuder and Wallace algorithm keeps track of the best solution found so far, and cuts off any branch that will not lead to an improved solution. In our algorithm for SCSPs, failure of a search path occurs if the estimated semiring value associated with the current node in the search path (its upper bound value) cannot improve the best solution found so far, that is, the maximal semiring associated with any complete assignment so far. We explain the test for local failure in more detail later on.

For the remainder of this chapter, assume a SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$, with the cardinality of V equal to n , and S_p has a best semiring value value of $\mathbf{1}$, and a worst semiring value of $\mathbf{0}$. Assume that the n decision variables in V are numbered from 0 to $n - 1$.

Our goal is to find a tuple that belongs to the maximal solution of P . The lower bound, LB , contains the semiring value of the best solution found so far and is initialised with the worst semiring value in S_p . For each node in the search tree, we calculate an upper bound value, UB , which represents a feasible solution. The upper bound of the root node is initialised with the best semiring value in S_p .

Our BnB algorithm (Algorithm 3) proceeds by instantiating a decision variable, and comparing its upper bound value with the lower bound value. If $UB \leq_{S_p} LB$, the subtree below the current node is pruned, and the algorithm backtracks to a node at a higher level in the tree. Otherwise, if $LB <_{S_p} UB$, the algorithm tries to find a better solution by instantiating one additional variable. The solution is the assignment that produced the final value of LB after the whole tree has been explored.

The decision variables are ordered according to their membership in the types of the constraints. We place the decision variables in the type of the first constraint on a queue, and then check whether these variables appear in the types of other constraints. If they do, the unlisted variables in these other constraints' types are also placed on the queue. Repeat this step until all constraints' types have been checked. Every decision variable has a *main constraint* which is the constraint where the variable was first identified to be placed on the queue. The algorithm has five parameters:

- *NoInstantiated*, the number of decision variables already instantiated;
- *Queue*, the decision variables in instantiation order;
- LB , the lower bound value;
- UB , the upper bound value associated with a search path up to a particular node;
- *BestSolution*, the best solution found so far.

A few local variables are used in the algorithm:

- variables to store the current assignment;
- *NewUB* and *FullUB* to store upper bound values;
- *var*, an object to store the current decision variable;
 - *var.value* is used to store the value assigned to *var*; and
 - *var.domain* contains the set of remaining domain values for *var*.

The initial call to the algorithm is $BnB(0, Queue, \mathbf{1}, \mathbf{0}, BestSolution)$. *Queue* is initialised in advance as explained earlier, and *BestSolution* has no variable assignment initially. When the algorithm halts, the parameter *LB* contains the semiring value of the variable assignment stored in *BestSolution*, i.e. of the maximal solution. The return value of the procedure *BnB* is used to halt the procedure immediately if a solution with the best semiring value is found. The return value of a call to the procedure can be either 0 or 1: a return value of 1 means that a solution has been found with the best semiring value, and in this case the procedure is halted. A return value of 0 means that the procedure should continue to search for a better solution.

In lines 1 to 5, a decision variable is instantiated. We search among the tuples of a decision variable's main constraint for a tuple with a maximal associated semiring value under the current partial assignment (line 4). From this chosen tuple we get a value to instantiate the current decision variable.

The next step is to calculate a upper bound value for the corresponding node (i.e. the current decision variable) in the search tree. This upper bound value is the combination of the semiring value associated with the nodes in the search path from the root node to the current node, combined with an estimation of the semiring value associated with the search path below the current node up to a leaf in the search tree. The calculation of this upper bound value consists of three steps:

1. In line 6 we calculate the semiring value associated with the current node by considering each constraint which has *var* (represented by this node in the tree)

Algorithm 3 BnB(*NoInstantiated*, *Queue*, *UB*, *LB*, *BestSolution*)

Require:Input: V ; C ; D ; S_p ; n .Variable parameters: LB ; $Best_Solution$ Value parameters: $NoInstantiated$; $Queue$; UB .

```

1: if ( $NoInstantiated < n$ ) then
2:    $var = \text{pop } Queue$ ;
3:   while ( $var.domain$  not empty) do
4:      $var.value = \text{select best value from } var.domain$ ;
5:      $var.domain = var.domain - var.value$ ;
6:      $NewUB = \text{upper bound value for current node}$ ;
7:      $NewUB = \times_p(UB, NewUB)$ ; [upper bound value for search path including current node]
8:     if  $NoInstantiated \neq n-1$  then
9:        $FullUB = FindFullUB(NewUB)$ ; [estimated upper bound]
10:    else
11:       $FullUB = NewUB$ ; [complete assignment]
12:    end if
13:    if ( $LB <_{S_p} FullUB$ ) then
14:       $UB = NewUB$ ;
15:      if ( $NoInstantiated = n-1$ ) then
16:         $LB = UB$ ;
17:         $BestSolution = \text{current assignment of values for decision variables}$ ;
18:        if ( $LB = \mathbf{1}$ ) then
19:          return 1;
20:        end if
21:      end if
22:      if ( $BnB(NoInstantiated+1, Queue, UB, LB, BestSolution) = 1$ ) then
23:        return 1;
24:      end if
25:    end if
26:  end while
27: end if
28: return 0;

```

as a member of its type, and which is such that all the decision variables in its type have been instantiated (i.e. we do a back check). We simply combine the semiring values of all the variable-value tuples of all these constraints.

2. In line 7, we combine the semiring value associated with the current node ($NewUB$) with the upper bound value of the search path up to current node's ancestor, i.e. UB .
3. In lines 8 to 12, we calculate an estimated associated (upper bound) semiring value for the search path below the current node up to a leaf, i.e. the semiring value if the assignment should be completed. Note that we want an over-estimation of the actual (unknown) semiring value. This estimated upper bound is a semiring value a such that $NewUB \otimes a$ is maximal. This value a is combined with the value $NewUB$ of step 2. The *if* part in line 8 is true if the current assignment is not complete; in the *else* part the current node is a leaf and $NewUB$ contains the actual upper bound value for the complete assignment.

In line 13 we test for local failure. If the upper bound value for the current assignment is better than LB , we update the upper bound value for this extended search path (current node is added) in line 14. Line 15 checks if we have a complete assignment, and if it is the case, it updates LB and the best solution found so far (lines 16 and 17). If we have found a maximal solution (in line 18), we halt in line 19. Line 22 extends the search by calling the *BnB* procedure to instantiate the next decision variable.

In the example below, we solve a small SCSP with our branch and bound algorithm.

Example: Solving a SCSP with the basic branch and bound algorithm

Consider a SCSP with $CS = \langle S_p, Dom, V \rangle$ and $P = \langle C, con \rangle$, where $S_p = \langle \{0, \dots, 5\}, max, min, 0, 5 \rangle$, $V = con = \{A, B, C, D, E, F, G, H\}$, $Dom = \{0, 1, 2, 3, 4\}$, and $C = \{c_1, c_2, c_3, c_4, c_5\}$.

Table 6.1 contains the constraint definitions with $c_1 = \langle def_{c_1}^p, \{A, B\} \rangle$, $c_2 = \langle def_{c_2}^p, \{C, D\} \rangle$, $c_3 = \langle def_{c_3}^p, \{E, A\} \rangle$, $c_4 = \langle def_{c_4}^p, \{F, G\} \rangle$, and $c_5 = \langle def_{c_5}^p, \{F, H\} \rangle$.

Table 6.1: Constraint Definitions

t	$def_{c_1}^p(t)$	$def_{c_2}^p(t)$	$def_{c_3}^p(t)$	$def_{c_4}^p(t)$	$def_{c_5}^p(t)$
$\langle 0, 0 \rangle$	3	3	3	4	2
$\langle 0, 1 \rangle$	2	2	0	0	0
$\langle 0, 2 \rangle$	3	3	4	1	2
$\langle 0, 3 \rangle$	1	0	3	2	2
$\langle 0, 4 \rangle$	0	2	0	2	0
$\langle 1, 0 \rangle$	0	1	2	2	3
$\langle 1, 1 \rangle$	2	1	2	2	3
$\langle 1, 2 \rangle$	3	1	3	1	2
$\langle 1, 3 \rangle$	4	3	3	2	2
$\langle 1, 4 \rangle$	2	3	3	2	0
$\langle 2, 0 \rangle$	3	1	3	2	0
$\langle 2, 1 \rangle$	2	4	2	0	4
$\langle 2, 2 \rangle$	4	0	3	2	3
$\langle 2, 3 \rangle$	1	1	0	3	0
$\langle 2, 4 \rangle$	2	0	2	2	0
$\langle 3, 0 \rangle$	2	4	3	3	0
$\langle 3, 1 \rangle$	2	3	3	2	3
$\langle 3, 2 \rangle$	0	0	0	1	0
$\langle 3, 3 \rangle$	0	2	2	0	2
$\langle 3, 4 \rangle$	0	3	0	3	2
$\langle 4, 0 \rangle$	0	1	0	4	1
$\langle 4, 1 \rangle$	2	3	3	3	3
$\langle 4, 2 \rangle$	2	1	1	3	3
$\langle 4, 3 \rangle$	1	2	3	2	4
$\langle 4, 4 \rangle$	2	2	4	1	0

The order of instantiation (*Queue*) is : A, B, E, C, D, F, G, H. The algorithm is initially called with $NoInstantiated=0$, $LB=0$, and $UB=5$. It selects the tuple $\langle 1, 3 \rangle$ in column 2 to assign the value 1 to the variable A , and the $NewUB$ value of this root node is 4. The procedure *FindFullUB* will return $FullUB = 4$ because it over-estimates the semiring value for the remainder of the (unexplored) search path to be 5, and $\min(5, 4) = 4$. Note that *FindFullUB* will always return the value of $NewUB$ in this example because our multiplicative operator is *min*.

Now it assigns the value 3 to the variable B , and $NewUB$ remains 4. This is because c_1 is the main constraint for both the variables A and B . In the rest of this example we will refer only to UB because $NewUB = FullUB = UB$ for any node in this problem.

In column 4, the tuple $\langle 3, 1 \rangle$ is selected to assign the value 3 to the variable E . UB becomes 3.

The algorithm proceeds and make the following assignments: $C = 2$, $D = 1$, $F = 0$, and $G = 0$. UB remains 3. Then we assign the value 0 to the variable H , and now UB changes to 2.

The assignment is complete and we find a new lower bound value, $LB = 2$.

During the second phase, backtracking occurs until we find an improved lower bound:

$H=2$; $H=3$; backtrack;
 $G=3$; $G=4$; $G=2$; backtrack;
 $F=4$ ($UB=3$); extend search path;
 $G=0$;
 $H=3$.

We have found a better lower bound: $LB= 3$.

During the third phase, backtracking occurs until we find an improved lower bound:

$H=1$; $H=2$; $H=0$; backtrack;
 $G=1$; $G=2$; $G=3$; $G=4$; backtrack;
 $F=2$; $F=3$; $F=1$; backtrack;
 $D=0$; $D=3$; backtrack;
 $C=3$; $C=0$; $C=1$; $C=4$; backtrack;
 $E=4$; $E=1$; $E=2$; backtrack;
 $B=2$; $B=1$; $B=4$; backtrack;
 $A=2$ ($UB=4$); extend search path;
 $B=2$; extend search path;
 $E=0$; extend search path;
 $C=2$; extend search path;
 $D=1$; extend search path;
 $F=0$; extend search path;

Take note of the following extension and backtracking sequence - called phase 4 - for future reference:

$G=0$; extend search path;
 $H=0$ ($UB=2$); backtrack;
 $H=2$; $H=3$; backtrack;
 $G=3$; $G=4$; $G=2$; backtrack.
 $F=4$ ($UB=4$); extend search path;
 $G=0$; extend search path;
 $H=3$.

A maximal solution with an associated semiring value of 4 has been found: $A=2$; $B=2$;
 $C=2$; $D=1$; $E=0$; $F=4$; $G=0$; $H=3$.

6.1.2 A Basic Branch and Bound Algorithm with Variable Partitioning

The decision variables in any SCSP can be partitioned in disjoint sets such that every variable in a particular partition will occur in the same constraint as at least one other variable in that partition. (The only exception is a variable that occurs in the type of a unary constraint. Such a variable can form a singleton partition if it does not appear in the type of any other constraint.) Each of these partitions can be solved as a smaller instance of the original problem. If the type con_c of a constraint c is a subset of a partition Pa , and at least one of the variables in con_c also belongs to the type of another constraint, e , then $con_e \subset Pa$. Algorithm 4 shows how to find the partitions of a SCSP, $P = \langle C, con \rangle$, over a constraint system $CS = \langle S_p, D, V \rangle$.

If the variables in a SCSP can be partitioned into at least two partitions, the performance of the basic branch and bound algorithm of the previous section is generally improved considerably by finding optimal solutions for every partition of the variables in a SCSP individually. The reason for this improvement is that we solve much smaller instances of the original problem and this results in smaller search trees.

We modified the procedure *BnB* to halt as soon as it finds its first complete assignment and the first lower bound value. This first call to the procedure finds an initial solution to the complete problem so that we have a global lower bound when we solve the

Algorithm 4 Partition

Require:Input: V (set of decision variables); C (set of constraints)

Output: A set of partitions

Local variables: Pa (a set of decision variables)

```

1:  $Pa = \emptyset$ ;
2: while ( $C$  is not empty) do
3:   select any  $k \in C$ ;
4:    $C = C - \{k\}$ ;
5:    $Pa = Pa \cup con_k$ ;
6:   for every decision variable  $i \in con_k$  do
7:     insert  $i$  on Queue;
8:   end for
9:   while Queue not empty do
10:     $var = \text{pop } Queue$ ;
11:    for every constraint  $l \in C$  such that  $var \in con_l$  do
12:      insert every decision variable  $j \in con_l, i \neq j$ , on Queue;
13:       $C = C - \{l\}$ ;
14:       $Pa = Pa \cup con_l$ ;
15:    end for
16:  end while
17:  store  $Pa$  as partition of  $V$ ;
18:   $Pa = \emptyset$ ;
19: end while

```

first partitioned problem. Then we place the variables in the first partition on the queue, *Queue*, and call the original procedure *BnB*, which finds an optimal solution for the reduced problem containing only the decision variables that belong to the first partition. Call *BnB* for every partition of the SCSP.

Example: Partitioning a set of variables

The set of decision variables of the example in the previous section is partitioned as follows: $P_1 = \{A, B, E\}$, $P_2 = \{C, D\}$, and $P_3 = \{F, G, H\}$.

6.1.3 A Backjumping Algorithm

Gaschnig presented a *backjumping* algorithm [22] that stores information about previous inconsistencies to reduce re-checking of constraints. Classical backtracking always

continues its search from the previous level. Backjumping takes note of the fact that all values tried for a given variable may not be inconsistent with the same previous value. The algorithm remembers the depth of failure, i.e. the deepest level, l at which any of the values it assigns to a variable fails. When there are no more values to be tried for this variable, backjumping continues its search at the level l .

A backjumping algorithm for maximal constraint satisfaction is described in Section 2.3.1. In Algorithm 5 we present a backjumping algorithm to solve SCSPs based on the backjumping algorithm for maximal constraint satisfaction. This work is to appear in [39].

The Freuder and Wallace backjumping algorithm for maximal constraint satisfaction [20] differs from conventional backjumping in not always jumping back all the way to the deepest level of failure. If any values below the level l failed when chosen, i.e. produced more inconsistencies, the algorithm can only jump back to the last deepest level l : other choices of values at level l may result in fewer failures. The search has to continue from this deeper level because we may be able to improve the solution. This algorithm also keeps track of the deepest level at which a value was (last) inconsistent. If this value is greater than the deepest level of failure, then backtracking will only jump back to this level.

In our backjumping algorithm for SCSPs, Algorithm 5, we have to keep track of various levels in the search tree:

- The first one is the level of the node associated with the current decision variable that is instantiated: we store this value in *NoInstantiated*.
- *FailDepth1* contains, for the current decision variable and its instantiated value, the level where this value may fail first (i.e. closer to the root). If this value does not fail, we update *FailDepth1* to the level of the current decision variable.
- *LD* (inconsistency depth) keeps track of the deepest level of failure for any value: If a value fails (by giving the current node a upper bound value that is worse than upper bound of its predecessor), then *LD* gets the value of the current node's level in the tree.

Algorithm 5 BJ(NoInstantiated, Queue, Domains, UB, LB, BestSolution, R_D, I_D)**Require:**Input: V ; C ; S_p ; n .Variable parameters: LB ; $Best_Solution$.Value parameters: $NoInstantiated$; $Queue$; UB ; $Domains$; R_D ; I_D .

```

1: if ( $NoInstantiated < n$ ) then
2:    $var = \text{pop } Queue$ ;
3:   if ( $var.domain$  not empty) then
4:      $var.value = \text{select best value from } var.domain$ ;
5:      $var.domain = var.domain - var.value$ ;
6:      $NewUB = \text{upper bound for current node}$ ;
7:      $NewUB = \times_p(UB, NewUB)$ ;
8:      $FailDepth1 = \text{first node from root where } var.value \text{ fails}$ 
9:     if  $NoInstantiated \neq n-1$  then
10:       $FullUB = FindFullUB(NewUB)$ ; [estimated upper bound]
11:     else
12:       $FullUB = NewUB$ ; [complete assignment]
13:     end if
14:   else
15:     return  $R\_D$ ;
16:   end if
17:   if ( $LB <_{S_p} FullUB$ ) then
18:      $FailDepth1 = NoInstantiated$ ;
19:     if ( $NoInstantiated = n-1$ ) then
20:        $LB = FullUB$ ;
21:        $BestSolution = \text{current assignment of values for decision variables}$ ;
22:       if ( $LB = 1$ ) then
23:         return  $n$ ; /* finished */
24:       end if
25:     end if
26:     if ( $FullUB <_{S_p} UB$ ) then
27:        $I\_D = NoInstantiated$ ;
28:     end if
29:      $FailDepth2 = BJ(NoInstantiated+1, Queue, Domains, FullUB, LB, BestSolution, 0, I\_D)$ ;
30:     if ( $FailDepth2 < NoInstantiated$ ) or ( $FailDepth2 = n$ ) then
31:       return  $FailDepth2$ ;
32:     else
33:        $R\_D = \max(FailDepth1, R\_D, I\_D)$ ;
34:       return  $BJ(NoInstantiated, Queue, Domains, UB, LB, BestSolution, R\_D, I\_D)$ ;
35:     end if
36:   end if
37:    $R\_D = \max(FailDepth1, R\_D, I\_D)$ ;
38:   return  $BJ(NoInstantiated, Queue, Domains, UB, LB, BestSolution, R\_D, I\_D)$ ;
39: end if
40: return  $NoInstantiated - 1$ ;

```


- R_D (return depth) is adjusted if a value fails: it is assigned the maximum value at which a failure has been detected.
- $FailDepth2$ keeps track of the level at which the search should continue after the search path has been extended. The value of $FailDepth2$ can either be smaller than the current node's level (a backjump), or it can be at the leaf level (search has been completed), or it can be equal to the current level (a new value must be assigned to the current variable).

Note that this algorithm has an additional parameter, $Domains$ (not in the procedure BnB), that maintains the domains of the decision variables.

The initial call is $BJ(0, Queue, D, 1, 0, BestSolution, 0, 0)$. When the algorithm halts, the parameter UB contains the semiring value associated with the variable assignment stored in $BestSolution$, i.e. the semiring value associated with the maximal solution. The return value of the procedure BJ is used to halt the procedure immediately if a solution with the best semiring value is found. The return value of a call to the procedure can be equal to any level of the search tree. If the return value is equal to the level of a leaf node, the algorithm halts; if the value is smaller than the level of the current node in the tree, the algorithm backjumps; if the value is equal to that of the current node in the tree, another value is selected for the current variable.

The backjumping algorithm first instantiates a decision variable and calculates the value of the upper bound (in lines 1 to 7), and then calculates the value of $Fail_Depth1$ in line 8. Lines 9 to 13 are similar to lines 8 to 12 of the procedure BnB where we calculate a upper bound value and test for failure.

Note that if the current decision variable does not have any domain values left (line 14), we return the value of R_D . R_D is initialised with the value 0 (the level of the root node), and its value is updated in lines 33 or 37. It controls the level to which recursion is rolled back.

If the current search path can improve the lower bound value (line 17), we adjust the value of $FailDepth1$ (line 18) to indicate that this value has not failed.

The *if* statement in line 26 is satisfied if the current node decreases the upper bound

value of the search path. In this case, this level becomes the new value of L_D level (line 27).

In line 29 we extend the search path. If the *if* statement in line 30 is satisfied, it means we can either backjump ($FailDepth2 < NoInstantiated$) or are finished ($FailDepth2 = n$). This returned value ($FailDepth2$) controls the extent to which recursion is rolled back.

If we reach line 33, it means that the current value has failed and we try another value in line 34. Lines 37 and 38 are executed in case the *if* statement in line 17 is not satisfied: R_D receives the deepest return level in the tree and we try another value.

In the example below, we solve the same SCSP introduced in Section 6.1.1, using our backjumping algorithm.

Example: Solving a SCSP with the Backjumping Algorithm

This algorithm proceeds in exactly the same way as the basic branch and bound algorithm until the fourth phase is reached. After finding an lower bound of 3, both the algorithms backtrack until variable A is assigned the value 2, extend the search path up to the last decision variable, H , try various values for H , and then backtrack.

At this point the branch and bound algorithm backtracks and tries various values for G , and backtracks again before assigning the value 4 to F . However, the **backjumping algorithm avoids backtracking from H to G : it backjumps from H to F .**

Note that as long as values do not fail, $FailDepth1$ is the level in which the decision variable occurs in the search tree.

Below, we show the execution of the algorithm from the point where A gets the value 2 and the search path is extended: (Note that we only show the value of $FailDepth1$ at a node where a change in its value occurs.)

$B = 2$; extend search path;

$E = 0$ ($FailDepth1 = 0$); extend search path;

$C = 2$; extend search path;

$D = 1$ ($FailDepth1 = 3$); extend search path;

$F = 0$ ($FailDepth1 = 5$); extend search path.

Phase 4:

$G=0$; extend search path;

$H=0$; $H=2$; $H=3$; **Backjump to level 5!**

$F=4$; extend search path;

$G=0$;

$H=3$.

The same maximal solution has been found before, but backjumping has managed to decrease the search effort.

6.1.4 A Forward Checking Algorithm

Classic forward checking (FC) [30] is a hybrid algorithm: it consists of backtracking with a check for local consistency ahead in the search tree. Section 2.3.1 contains a description of forward checking. Freuder and Wallace [20] developed a forward checking algorithm to solve maximal constraint satisfaction problems. We extend their algorithm to develop a forward checking algorithm for solving SCSP. Our algorithm is presented as Algorithm 6. This work appears in [39].

In a partial CSP context, it is not possible to eliminate values by doing forward arc consistency checking without having an initial value for the bound on the number of allowed inconsistencies. Freuder and Wallace extended the notion of arc consistency checking in maximal constraint satisfaction context: for each value, the number of domains with no supporting values can be counted (inconsistency count). The inconsistency count is a lower bound on the increase in the expected number of unsatisfied constraints that will be found if this value is to be added to the solution. In a particular search path, the arc consistency count for a proposed value assignment to the current decision variable can be added to the number of unsatisfied constraints up to date in that search path, and compared to the current bound, NB (least number of unsatisfied constraints found in any search path so far). If the sum is not less than NB , then we know that the current search path will not result in a better solution.

Note that when a value v is assigned to a decision variable X , the domains of all the uninstantiated variables that appear in a constraint with X are checked for values that are inconsistent with v : these values are deleted from the domains. If the decision variable X is assigned another value at a later stage, the pruned value has to be replaced.

We present a forward checking algorithm for SCSPs in Algorithm 6, that is based on the Freuder and Wallace algorithm. Our algorithm is similar to the basic BnB algorithm except for the procedure we call in line 7 to perform forward checking.

The initial call is $FC(0, Queue, \mathbf{1}, \mathbf{0}, BestSolution)$. When the algorithm halts, the parameter UB contains the semiring value of the variable assignment stored in $BestSolution$, i.e. of the maximal solution. The return value of the procedure BnB is used to halt the procedure immediately if a solution with the best semiring value is found. The return value of a call to the procedure can be either 0 or 1: a return value of 1 means that a solution has been found with the best semiring value and in this case the procedure is halted. A return value of 0 means that the procedure should continue to search for a better solution.

The forward checking procedure ensures that the newly instantiated decision variable has a consistent value in the domains of all (uninstantiated) decision variables that appear in the same constraints. It finds every constraint which has the newly instantiated decision variable (var) in its type and has at least one variable that has not yet been instantiated. For each of these constraints, it ensures that every one of the uninstantiated decision variables of the constraint has at least one domain value such that the resulting value tuple's associated semiring value does not fail.

The test for failure is similar to the estimation of an upper bound value for the completion of a partial assignment. This means that the forward check for the current node is satisfied if there exist at least one domain value for every uninstantiated decision variable that appears with the current instantiated decision variable in the same constraint.

Note that we have to restore domain values (line 28) that have been removed from domains during a forward check for a decision variable var with the value var_value , if

this value fails. Previous domain values were stored in line 6.

In the example below, we solve the SCSP introduced in section 6.1.1, with our forward checking algorithm.

Example: Solving a SCSP with the Forward Checking algorithm

This algorithm proceeds in exactly the same way as the basic branch and bound algorithm to find the lower bound value of 3, and then to backtrack and try the values of 1 and 2 for H .

The branch and bound algorithm then assigns the value 0 to H , but **forward checking eliminates this value from the domain of H as part of its forward checking** of constraints at the point where it assigned the value of 4 to F . It used constraint c_4 to find this value for F , but the value of 0 for H fails in constraint c_5 during forward checking.

Both algorithms then backtrack to variable G and try the values 1 and 2. The branch and bound algorithm also tries the values 3 and 4 for G , while **forward checking has eliminated both these values from the domain of G** because they fail in constraint c_4 against the current lower bound.

We show this part of the execution (after an improved lower bound, $LB=3$, has been found):

Third phase:

$H=1$; $H=2$; **Next value for H has been removed!**; backtrack;

$G=1$; $G=2$; **Next two values for G have been removed!**; backtrack;

$F=2$; ...

Now both algorithms proceed exactly in the same way until phase 4 is reached. After A is assigned the value 2, the branch and bound algorithm extends the search path down to the leaf node, where H is assigned the value 0. (See the second line of phase 4 in the branch and bound example.) The forward checking algorithm only extends the search path up to the parent of a leaf node, that is, up to the assignment of the value 0 to G (first line of phase 4). **Forward checking removed all values from the domain**

Algorithm 6 *FC(NoInstantiated, Queue, UB, LB, BestSolution)*

Require:Input: $V; C; D; S_p; n$.Variable parameters: $LB; Best_Solution$.Value parameters: $NoInstantiated; Queue; UB$.

```

1: if ( $NoInstantiated < n$ ) then
2:    $var = \text{pop } Queue$ ;
3:   while ( $var.domain$  not empty) do
4:      $var.value = \text{select best value from } var.domain$ ;
5:      $var.domain = var.domain - var.value$ ;
6:      $TempDomain = var.domain$ ;
7:      $ForwardCheck(var)$ ;
8:      $NewUB = \text{upper bound value for current node}$ ;
9:      $NewUB = \times_p(UB, NewUB)$ ;
10:    if  $NoInstantiated \neq n-1$  then
11:       $FullUB = FindFullUB(NewUB)$ ;
12:    else
13:       $FullUB = NewUB$ ;
14:    end if
15:    if ( $LB <_{S_p} FullUB$ ) then
16:       $UB = NewUB$ ;
17:      if ( $NoInstantiated = n-1$ ) then
18:         $LB = UB$ ;
19:         $BestSolution = \text{current assignment of values for decision variables}$ ;
20:        if ( $LB = 1$ ) then
21:          return 1;
22:        end if
23:      end if
24:      if ( $FC(Noinstantiated+1, Queue, UB, LB, BestSolution) = 1$ ) then
25:        return 1;
26:      end if
27:    end if
28:     $var.domain = TempDomain$ ;
29:  end while
30: end if
31: return 0;

```

of H when F was instantiated. The forward checking algorithm then backtracks one level, but **all values for G have also been removed**, and it backtracks again and assigns the value 4 to F .

Summary of Phase 4:

$G=0$; extend search path;

All values of H have been removed!; backtrack;

All values of G has been removed!; backtrack;

$F=4$; extend search path;

$G=0$; extend search path;

$H=3$.

The same maximal solution has been found, but the forward checking algorithm has managed to reduce the search effort.

6.2 Experimental Setting and Results

The only implemented general SCSP solver that we are aware of is Georget and Codognet's constraint logic programming system [26], but this system is no longer maintained. In Section 6.2.2 below, we show that our branch and bound algorithm with variable partitioning performs significantly better than the well-known fuzzy CSP solver, CON'FLEX [17]. The runtimes of the branch and bound algorithm (with partitioning) is a minimum of 25 times faster than those of CON'FLEX. Note that we experienced difficulty in running the CON'FLEX software on larger problems.

In Section 6.2.3, we compare the backjumping and forward checking algorithms (without variable partitioning) to the branch and bound algorithm, both with and without variable partitioning. We show that the backjumping and forward checking algorithms perform better than the branch and bound algorithm (without variable partitioning) on harder problems. The branch and bound algorithm with partitioning is on average three times faster than the branch and bound algorithm without variable partitioning. This means that all three our algorithms perform much better than CON'FLEX.

In the future we plan to implement the backjumping and forward checking algorithms with variable partitioning, and it is reasonable to expect that variable partitioning will improve both these algorithms to the same extent as the improvement in the branch and bound algorithm when variable partitioning was added.

6.2.1 The randomly generated problems

Please refer to Appendix A for details on our random SCSP generator. We solve randomly generated binary SCSPs that are instances of fuzzy CSPs. All the problems have the semiring structure, $S_p = \langle \{0, 0.3, 0.5, 0.8, 1\}, \max, \min, 0, 1 \rangle$, and each decision variable has 10 domain values.

Note that our algorithms will also handle SCSPs with semirings that form partial orders: the multiplicative operator of a semiring's unit element is the best semiring value, and its absorbing element is the worst semiring value. They may find a maximal solution that is incomparable with other existing maximal solutions.

For the comparison of the performance of our algorithms (in Section 6.2.3), we use the same three sets of problems we used in Section 5.3.2 to test the GSAT-based algorithm, but we added 50 additional problems to set 3. Table 5.2 describes these problems, but for convenience we summarise the characteristics of these problems below.

The problems in sets 1, 2 and 3 have 80, 100 and 120 decision variables, and 10, 10, and 20 constraints respectively. Sets 1 and 2 each contains 100 problems; 50 of the instances have a tightness of 70% and the other 50 instances have a tightness of 90%. Set 3 has 150 instances: 50 of the instances have a tightness of 70%, 50 of the instances have a tightness of 85%, and 50 of the instances have a tightness of 90%. A tightness (T) of x% means that (100-x)% of all the tuples have been assigned the best semiring value.

All the problems have optimal solutions with the best semiring value, 1, with the exception of three instances in set 3 with tightness 90% (last column). These three instances have optimal solutions with associated semiring values of 0.8.

In Section 6.2.2 we had to generate smaller sized problems because we experienced

Table 6.2: Average runtimes for Sets 4 and 5 (in seconds)

Set	4	4	5	5
Tightness	60%	80%	60%	80%
CONFLEX	12.34	19.89	154.42	165.14
BnB	0.44	0.29	0.05	0.26

problems when we attempted to run the CON'FLEX software on larger problems. Here we generated two sets of problems. The 40 problem instances in Set 4 have 10 variables and 10 constraints each: 20 of the instances have a tightness of 60% and 20 of the instances have a tightness of 80%. The 20 problem instances in Set 5 have 15 variables and 10 constraints each: 10 of the instances have a tightness of 60% and 10 of the instances have a tightness of 80%.

6.2.2 Branch and bound with variable partitioning vs. CON'FLEX

We used the CON'FLEX executable for Windows, downloaded from the CON'FLEX website.¹ Our branch and bound algorithm was implemented in C++ and compiled using the gcc compiler, version 3.3.3 (under cygwin). We used an Intel Pentium 4 processor at 2.53GHz with 512MB RAM, running Microsoft Windows XP Home Edition version 2002.

Table 6.2 shows the results for each of the two algorithms and each set. The second row indicates the tightness of the problems in the particular set, and the two remaining rows show the average runtimes for the two algorithms. We experienced difficulty in running larger problems on the Windows version of the CONFLEX software. However, it is clear from these experiments that our branch and bound algorithm is considerably faster than CONFLEX, in fact, at least 25 times faster.

Note that these results appear in [38].

¹<http://www.inra.fr/internet/Departements/MIA/T//conflex/adressesConflex.html>

Table 6.3: Average runtimes of the three algorithms (in seconds)

Set	1	1	2	2	3	3	3
T	70%	90%	70%	90%	70%	85%	90%
BnB	0.0124	0.0552	0.0206	0.0544	0.0688	0.1518	0.8078
BJ	0.0102	0.0774	0.0210	0.0772	0.0778	0.2142	0.6596
FC	0.0194	0.0696	0.0216	0.0698	0.0876	0.2056	0.4758
BPart	0.0164	0.0346	0.0204	0.0382	0.061	0.0916	0.3234

6.2.3 Branch and bound vs. Backjumping vs. Forward Checking

All the algorithms are implemented in C++ and compiled with the gcc compiler, version 4.0.1. We used an Intel Core 2 Duo processor at 2GHz with 2GB RAM, running Mac OS X, Version 10.4.11.

Table 6.3 shows the average runtimes for each set and each algorithm. The second row indicates the tightness of the problems in the particular set, and the third, fourth, fifth and sixth rows show the average runtimes of the basic branch and bound algorithm (BnB), the backjumping algorithm (BJ), the forward checking algorithm (FC), and the branch and bound algorithm with partitioning (BPart), respectively.

There is not a marked difference in the performance of the different algorithms for smaller problems, but the results of the last set (Set 3 with 90% tightness) show an improvement for the backjumping and forward checking algorithms on the branch and bound algorithm (without variable partitioning) of 18% and 41% respectively. Note that branch and bound algorithm with variable partitioning's runtimes are at most 3 times faster than those of the basic branch and bound algorithm, but on average at least 25 times faster than CON'FLEX. In future, we plan to implement the forward checking and backjumping algorithms with variable partitioning.

6.3 A CSP-based Algorithm for Solving SCSPs

In this section we show that solutions to some SCSPs are guaranteed by solving a sequence of at most n complete CSPs, where n is bounded from above by the number of distinct semiring values in the associated semiring structure. This is a theoretical alternative to the algorithms introduced previously.

Assume a SCSP $P = \langle C, con \rangle$ over a constraint system $CS = \langle S_p, D, V \rangle$, with $S_p = \langle \{p_1, p_2, \dots, p_n\}, +_p, \times_p, p_1, p_n \rangle$. The set of constraints is $C = \{c_1, c_2, \dots, c_m\}$ where $c_i = \langle def_{c_i}^p, con_{c_i} \rangle$, k_{c_i} is the cardinality of con_{c_i} , and $con_{c_i} \subseteq con$ for $i = 1, 2, \dots, m$. The following statement has to hold for the multiplicative operator, \times_p : if $\alpha, \beta \leq_p \gamma$ then $\times_p(\alpha, \beta) \leq_p \gamma$ for $\alpha, \beta, \gamma \in \{p_1, p_2, \dots, p_n\}$.

Algorithm 7 CSP-based Algorithm

Require:

- Input: $V; C; D; S_p$.
Output: A maximal solution for the original SCSP.
- 1: Let $A_{all} = \{v \mid \exists c \in C \text{ such that } def_c^p(t) = v \text{ and } t \text{ is a tuple of } c\}$;
 - 2: Let $A_{current} = \emptyset$;
 - 3: **while** ($A_{all} \neq \emptyset$) **do**
 - 4: Non-deterministically select any $v \in A_{all}$ such that $\nexists w \in A_{all}$ where $v <_{S_p} w$;
 - 5: $A_{current} = A_{current} \cup \{v\}$;
 - 6: $A_{all} = A_{all} - \{v\}$;
 - 7: Let $C' = \emptyset$;
 - 8: **for** each $c \in C$ **do**
 - 9: Let $c' = \{t \mid def_c^p(t) = v \text{ with } v \in A_{current} \text{ and } t \text{ is a tuple of } c\}$;
 - 10: $C' = C' \cup \{c'\}$;
 - 11: **end for**
 - 12: Let $P' = \langle C', con \rangle$;
 - 13: Solve P' as a classical CSP;
 - 14: If a solution exists, exit and return the solution;
 - 15: **end while**
-

In Algorithm 7 we start by constructing a CSP where we only allow tuples from the original SCSP that have the maximal semiring value associated. Let the first constructed CSP be $P'_1 = \langle V, D, C'_1 \rangle$ with $C'_1 = \bigcup_{i=1}^m c'_i$ and $c'_i = \{t \mid def_{c_i}^p(t) = p_n\}$ with $c_i \in C$, $t = \langle t_1, t_2, \dots, t_{k_{c_i}} \rangle$ and $t_j \in D$ for $j = \{1, \dots, k_{c_i}\}$. If we find a solution for P'_1 , this solution is also a solution for the SCSP P .

Otherwise, we continue by selecting any semiring value $v \in S_p$ such that v is not dominated by any other semiring value in $S_p - \{p_n\}$. The new CSP is $P'_2 = \langle V, D, C'_2 \rangle$ where $C'_2 = \bigcup_{i=1}^m c'_i$ where $c'_i = \{t \mid \text{def}_{c_i}^p(t) \in \{p_n, v\}\}$ with $t = \langle t_1, t_2, \dots, t_{k_{c_i}} \rangle$ and $t_j \in D$ for $j = \{1, \dots, k_{c_i}\}$. If we find a solution for P'_2 , this solution is also a solution for the SCSP P .

We continue this process by constructing CSPs with an increasing number of allowed tuples until we find a solution.

Consider Algorithm 7. The set A_{all} in line 1 contains all the distinct semiring values that are associated with tuples of constraints in C . In line 4 we select any non-dominated semiring value v from the set A_{all} . In the next line, the value v is added to the set $A_{current}$, which contains all the semiring values that are to be taken into consideration when building the next CSP to be solved. In line 6 we remove the value v from the set A_{all} . The *for* loop in lines 8 to 11 constructs the constraints of the next CSP to be solved by only allowing tuples from the constraints in the set C that have associated semiring values that are members of the set $A_{current}$. In line 12 the new CSP is constructed and it is solved in line 13.

It is trivial to show that if a solution is found to a constructed CSP, this solution is a maximal solution to the SCSP (with the restriction on the multiplicative operator of the semiring). Our algorithm finds a maximal solution; it does not find all maximal solutions.

We illustrate the algorithm with a simple example.

Example: Solving a SCSP with the CSP-based Algorithm

Consider the SCSP with $CS = \langle S_p, D, V \rangle$ and $P = \langle C, \text{con} \rangle$, where $V = \text{con} = \{A, B, C\}$, $D = \{1, 2, 3\}$, $C = \{c_1, c_2, c_3\}$, and $S_p = \langle \{s_1, s_2, \dots, s_7\}, +_p, \times_p, s_1, s_7 \rangle$.

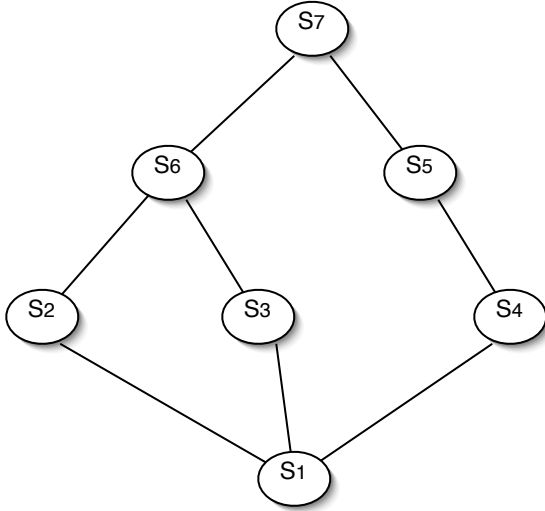
Assume three binary constraints, $c_1 = \langle \text{def}_{c_1}^p, \{A, B\} \rangle$, $c_2 = \langle \text{def}_{c_2}^p, \{B, C\} \rangle$, and $c_3 = \langle \text{def}_{c_3}^p, \{C, A\} \rangle$. The tuples of these constraints together with their associated semiring values are given in Table 6.4.

The partial ordering, \leq_{S_p} , over the set of semiring values, $\{s_1, s_2, \dots, s_7\}$, is shown

Table 6.4: Constraint definitions

t	$def_{c_1}^p(t)$	$def_{c_2}^p(t)$	$def_{c_3}^p(t)$
$\langle 1, 1 \rangle$	s_6	s_7	s_4
$\langle 1, 2 \rangle$	s_7	s_4	s_7
$\langle 1, 3 \rangle$	s_1	s_6	s_3
$\langle 2, 1 \rangle$	s_3	s_5	s_2
$\langle 2, 2 \rangle$	s_2	s_6	s_7
$\langle 2, 3 \rangle$	s_2	s_3	s_2
$\langle 3, 1 \rangle$	s_5	s_1	s_6
$\langle 3, 2 \rangle$	s_6	s_6	s_5
$\langle 3, 3 \rangle$	s_7	s_2	s_3

below:



$A_{all} = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$ and $A_{current} = \emptyset$.

First iteration of the *while* loop:

Select $v = s_7$.

$A_{current} = \{s_7\}$ and $A_{all} = \{s_1, s_2, s_3, s_4, s_5, s_6\}$.

$c_1 = \{\langle 1, 2 \rangle, \langle 3, 3 \rangle\}$, $c_2 = \{\langle 1, 1 \rangle\}$, and $c_3 = \{\langle 1, 2 \rangle, \langle 2, 2 \rangle\}$.

$C' = \{c_1, c_2, c_3\}$.

$P' = \langle C', con \rangle$ does not have a solution.

Second iteration of the *while* loop:

Select $v = s_5$.

$A_{current} = \{s_5, s_7\}$ and $A_{all} = \{s_1, s_2, s_3, s_4, s_6\}$.

$c_1 = \{\langle 1, 2 \rangle, \langle 3, 1 \rangle, \langle 3, 3 \rangle\}$, $c_2 = \{\langle 1, 1 \rangle, \langle 2, 1 \rangle\}$, and $c_3 = \{\langle 1, 2 \rangle, \langle 2, 2 \rangle, \langle 3, 2 \rangle\}$.

$C' = \{c_1, c_2, c_3\}$.

$P' = \langle C', con \rangle$ does not have a solution.

Third iteration of the *while* loop:

Select $v = s_6$.

$A_{current} = \{s_5, s_6, s_7\}$ and $A_{all} = \{s_1, s_2, s_3, s_4\}$.

$c_1 = \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle\}$, $c_2 = \{\langle 1, 1 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 3, 2 \rangle\}$, and

$c_3 = \{\langle 1, 2 \rangle, \langle 2, 2 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle\}$.

$C' = \{c_1, c_2, c_3\}$.

$P' = \langle C', con \rangle$ has a solution.

A solution is $A = 1, B = 1, C = 3$. This maximal solution for the SCSP has an associated semiring value of s_6 .

6.4 Conclusion and Future Work

We presented three branch and bound algorithms to solve SCSPs that are based on the well-known algorithms for maximal constraint satisfaction by Freuder and Wallace: a BnB algorithm (with and without variable partitioning), a backjumping algorithm and a forward checking algorithm.

We also presented an algorithm to solve SCSPs by solving a finite sequence of CSPs.

Our branch and bound algorithm with variable partitioning is significantly faster than the fuzzy CSP solver, CON'FLEX. The branch and bound algorithm with variable partitioning performs better than the basic version. The backjumping and forward checking algorithms perform better than the basic branch and bound on some problem instances.

In the future, we plan to implement our forward checking and backjumping algorithms with variable partitioning, to develop a backmarking algorithm, and to investigate different versions of the forward checking algorithm. We also want to test our algorithms

on larger-scale SCSPs.

Chapter 7

Conclusion

7.1 Summary and Discussion

In this thesis, we presented methods for solving Semiring Constraint Satisfaction Problems.

Our first approach was to present a way to relax a SCSP by relaxing individual constraints until an acceptable solution could be found. A second semiring was used to provide a measure of difference between the original problem and its relaxation. We then showed how to combine the two semiring structures: the one associated with the original SCSP and the second one that measured the difference between the SCSP and its relaxation.

Our second approach focused on developing algorithms to solve SCSPs. We showed how to transform an SCSP into a variant of a Weighted Maximum Satisfiability problem, which we called a Weighted Semiring Maximum Satisfiability Problem (WS-Max-SAT). We presented an algorithm to solve a WS-Max-SAT that is based on the well-known GSAT algorithm for maximum satisfiability problems.

Our next three algorithms to solve SCSPs were a branch and bound algorithm, a backjumping algorithm, and a forward checking algorithm. These algorithms are based on the algorithms of Freuder and Wallace [20] for maximal constraint satisfaction. Our branch and bound algorithm performed significantly better than CON'FLEX, a fuzzy CSP solver [17], and the forward checking and backjumping algorithms performed

better on some problem instances than the branch and bound algorithm.

Finally, we presented a CSP-based algorithm to solve some types of SCSPs. This algorithm solved a finite number of CSPs in order to find a solution for a SCSP.

The research in this thesis was done in the context of an industry project for a major steel producer. The problem that was addressed in this project was highly over-constrained, and a motivation for our research into over-constrained problems.

The publications that resulted from work in this thesis are [38, 39, 40, 41, 42].

7.2 Future Work

Relaxation of a SCSP: In Chapter 3 we presented a way to relax a SCSP so that an acceptable solution could be found. In future, we will focus on the computational aspects of this process.

When a solution to a SCSP P is not good enough, we used a set of cut-off values, LB , to define a threshold that should be reached. We needed only to consider relaxations to constraints of P that have the potential to form a good enough solution. Such relaxed constraints can be found by looking for at least one tuple in the original constraint with a preference value that is not in the set LB and then raise it so that it has a preference value in LB . We plan to develop efficient algorithms to find suitable subsets of relaxations, and to develop techniques to calculate the best relaxation for a SCSP efficiently.

Combined Semirings: In Chapter 4 we presented a Combined Relaxation of a SCSP. Our future work will focus on computational aspects of this process. We aim to develop techniques to calculate solutions to a maximal Combined Relaxation of a SCSP efficiently.

Transformation of a SCSP to a Weighted Semiring Max-SAT Problem: In Chapter 5, we showed how to translate a SCSP into a variant of a Weighted Max-SAT problem, which we called a Weighted Semiring Max-SAT problem. The objective was to make use of the efficient algorithms that are available to solve Max-SAT, and modify them

to solve WS-Max-SATs. We developed a simple GSAT-based algorithm to show that this is a viable approach to the solution of SCSPs.

There are several local search algorithms that should be easy to modify for WS-Max-SAT. Recent advances in local search algorithms are interesting to consider: Pham et al. [50] show that local SAT search techniques can be improved by building structural information into the search, and Pham et al. [51] developed a SAT solver that automatically recognise CSP structure hidden in the SAT encodings.

The existing exact algorithms will require more effort to modify, but are likely to produce good results.

Algorithms to solve SCSPs: We developed four algorithms to solve SCSPs in Chapter 6: a branch and bound algorithm, a forward checking algorithm, a backjumping algorithm, and a CSP-based algorithm. The first algorithm favourably compared one of them against a fuzzy CSP solver.

In the future, we plan to test our algorithms against SCSPs that model real life problems. We also want to develop other variations of branch and bound algorithms, such as a backmarking algorithm for SCSPs, as well as variants of the forward checking algorithm. A next step is to implement our backjumping and forward checking algorithms with variable partitioning.

Bibliography

- [1] T. Alsinet, F. Manyà, and J. Planes. Improved exact solver for weighted Max-SAT. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-2005)*, pages 371–377, 2005.
- [2] R. Bartak. *On-line guide to constraint programming*. <http://kti.mff.cuni.cz/bartak/constraints>.
- [3] M. Beaumont, J. Thornton, A. Sattar, and M. Maher. Solving over-constrained temporal reasoning problems using local search. In *Proceedings of the 8th Pacific Rim International Conference on Artificial Intelligence (PRICAI-2004)*, pages 134–143, 2004.
- [4] H. Bennaceur. The satisfiability problem regarded as a constraint satisfaction problem. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 1996)*, pages 155–159, 1996.
- [5] S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Basic properties and comparison. *Constraints*, 4:199–240, 1999.
- [6] S. Bistarelli, E. Freuder, and B. O’Sullivan. Encoding partial constraint satisfaction in the semiring-based framework for soft constraints. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 240–245, 2004.

-
- [7] S. Bistarelli, S. Fung, J. Lee, and H. Leung. A local search framework for semiring-based constraint satisfaction problems. In *Proceedings of the 5th International Workshop on Soft Constraints (Soft-2003)*, 2003.
 - [8] S. Bistarelli, J. Kelleher, and B. O’Sullivan. Tradeoff generation using soft constraints. In *Proceedings of Constraint Solving and Constraint Logic Programming (CSCLP 2003)*, pages 124–139, 2003.
 - [9] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
 - [10] S. Bistarelli, F. Rossi, and I. Pilan. Abstracting soft constraints: Some experimental results on fuzzy CSPs. In *Proceedings of the Joint Annual Workshop of ERCIM Working Group on Constraints and the CologNET area on Constraint and Logic Programming*, 2003.
 - [11] A. Borning, B. Freeman-Benson, and W. Molly. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992.
 - [12] A. Borning, M. Maher, A. Martindale, and W. Molly. Constraint hierarchies and logic programming. In *Proceedings of the Sixth International Conference on Logic Programming (ICLP-1989)*, pages 149–164, 1989.
 - [13] H. Cartan. *Théorie des filtres*. CR Academic Press, Paris, 1937.
 - [14] S. de Givry, J. Larrosa, Meseguer, and T. Schiex. Solving Max-SAT as weighted CSP. In *Proceedings of the Ninth International Conference On Principles and Practice of Constraint Programming (CP-2003)*, 2003.
 - [15] R. Dechter. *Constraint Processing*. Morgan Kaufman Publishers, San Francisco, 2003.
 - [16] A. Delgado, C. Olarte, J. Perez, and C. Ruede. Implementing semiring-based constraints using Mozart. In *Proceedings of the second international Mozart/Oz Conference (MOZ 2004)*, 2004.

-
- [17] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of the IEEE Conference on Fuzzy Systems*, pages 1131–1136, 1993.
 - [18] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proceedings of the Symbolic and Quantitative Approaches to Reasoning and Uncertainty (ECSQARU 2003)*, pages 97–104, 1993.
 - [19] H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in fuzzy constraint satisfaction problems. In *Proceedings of the First European Congress on Fuzzy and Intelligent Technologies*, volume 3, pages 1128–1134, 1993.
 - [20] E. C. Freuder and J. W. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
 - [21] Z. Fu and S. Malik. On solving the partial Max-SAT problem. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT 2006)*, 2006.
 - [22] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second National Conference for the Canadian Society for Computational Studies of Intelligence (CCSCSI-78)*, 1978.
 - [23] J. Gaschnig. A general backtrack algorithm that eliminates most redundant checks. In *Proceedings of the Second National Conference for the Canadian Society for Computational Studies of Intelligence (CCSCSI-78)*, 1978.
 - [24] I. P. Gent. Arc consistency in SAT. In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-2002)*, 2002.
 - [25] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-1993)*, pages 28–33, 1993.

-
- [26] Y. Georget and P. Codognet. Compiling semiring-based constraint with `clp(fd,s)`. In *Proceedings of the 4th International Conference on the Principles and Practice of Constraint Programming (CP-1998)*, 1998.
- [27] A. Ghose and P. Harvey. Partial constraint satisfaction via semiring CSPs augmented with metrics. In *Proceedings of the 15th Australian Joint Conference on AI (AUS-AI 2002)*, volume 2557 of *LNCS*, pages 443–454. Springer, 2002.
- [28] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability problem: a survey. In D. Du, J. Gu, and P. Pardalos, editors, *Satisfiability problem: Theory and Applications*, volume 35 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.
- [29] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.
- [30] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
- [31] F. Heras, J. Larrosa, and A. Oliveras. MiniMaxSat: A new weighted Max-SAT solver. In *Proceedings of Theory and Applications of Satisfiability Testing (SAT-2007)*, 2007.
- [32] A. Ishtaiwi, J. Thornton, A. Sattar, and D. Pham. Neighbourhood clause weight redistribution in local search for SAT. In *Principles and Practice of Constraint Programming (CP 2005)*, pages 772–776, 2005.
- [33] M. Jampel. A compositional theory of constraint hierarchies. In *Over-Constrained Systems*, volume 1106 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 1995.
- [34] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.
- [35] V. Kumar. Algorithms for constraint satisfaction problems: a survey. *AI Magazine*, 13:32–44, 1992.

-
- [36] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.
 - [37] D. Le Berre. *Sat4j project homepage*. <http://www.sat4j.org>.
 - [38] L. Leenen, A. Anbulagan, T. Meyer, and A. Ghose. Modelling and solving semiring constraint satisfaction problems by transformation to weighted semiring max-SAT. In *Proceedings of the Twentieth Australian Joint Conference on Artificial Intelligence (AI-2007)*, pages 202–212, 2007.
 - [39] L. Leenen and A. Ghose. Branch and bound algorithms to solve semiring constraint satisfaction problems. In *Proceedings of the 10th Pacific Rim International Conference on Artificial Intelligence (PRICAI-2008)*, 2008.
 - [40] L. Leenen, T. Meyer, and A. Ghose. Relaxations of semiring constraint satisfaction problems. In *Proceedings of the 7th International Workshop on Preferences and Soft Constraints (SOFT-05)*, 2005.
 - [41] L. Leenen, T. Meyer, and A. Ghose. Relaxations of semiring constraint satisfaction problems. *Information Processing Letters*, 103(5):177–182, 2007.
 - [42] L. Leenen, T. Meyer, P. Harvey, and A. Ghose. A relaxation of a semiring constraint satisfaction problem using combined semirings. In *Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence (PRICAI-2006)*, pages 907–911, 2006.
 - [43] C. M. Li, F. Manyà, and J. Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.
 - [44] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1), 1977.
 - [45] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence (ECAI 1996)*, pages 321–326, 1997.

-
- [46] F. Menezes, P. Barahoma, and P. Codognet. An incremental hierarchical constraint solver. In *Proceedings of the First Workshop on Principles and Practice of Constraint Programming (PPCP93)*, pages 190–193, 1993.
- [47] P. Meseguer. Constraint satisfaction problems: an overview. *AI Communications*, 2:3–27, 1989.
- [48] B. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
- [49] D. Pham, J. Thornton, C. Greton, and A. Sattar. Advances in local search for satisfiability. In *Proceedings of the Twentieth Australian Joint Conference on Artificial Intelligence (AI-2007)*, pages 213–222, 2007.
- [50] D. Pham, J. Thornton, and A. Sattar. Building structure into local search for SAT. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 2359–2364, 2007.
- [51] D. Pham, J. Thornton, A. Sattar, and A. Ishtaiwi. SAT-based versus CSP-based constraint weighting for satisfiability. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005)*, pages 455–460, 2005.
- [52] H. Rudova. *Constraint Satisfaction with Preferences*. PhD thesis, Faculty of Informatics, Masaryk University, 2001.
- [53] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. *Software: Practice and Experience*, 23(5):529–566, 1993.
- [54] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-1994)*, pages 337–343. AAAI/The MIT Press, 1994.
- [55] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-1992)*, pages 440–446. AAAI/The MIT Press, 1992.

-
- [56] L. G. Shapiro and R. M. Haralick. Structural descriptions and inexact matching. *IEEE Transaction of Pattern Analysis and Machine Intelligence*, 3:504–519, 1981.
- [57] T. Stützle, H. Hoos, and A. Roli. A review of the literature on local search algorithms for MAX-SAT. *Technical report AIDA-01-02, FG Intellektik FB, Informatik Darmstadt University of Technology*, 2001.
- [58] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.
- [59] T. Walsh. SAT v CSP. In *Proceedings of the Sixth International Conference on the Principles and Practice of Constraint Programming (CP-2000)*, Lecture Notes in Computer Science, pages 441–456. Springer, 2000.
- [60] Wikipedia. *Filter(Mathematics)*. [http://en.wikipedia.org/wiki/Filter_\(mathematics\)](http://en.wikipedia.org/wiki/Filter_(mathematics)).
- [61] M. Wilson and A. Borning. Hierarchical constraint logic programming. *Journal of Logic Programming*, 16:277–318, 1993.
- [62] N. Wilson. Decision diagrams for the computation of semiring valuations. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.
- [63] Z. Xing and W. Zhang. MaxSolver: An efficient exact algorithm for (weighted) maximum satisfiability. In *Artificial Intelligence Journal*, volume 164(1-2), 2005.
- [64] L. Zhou, A. Sattar, and S. Goodwin. Handling over-constrained problems in distributed multi-agent systems. In *Proceedings of the 18th Conference of the Canadian Society for Computational Studies on Intelligence (AI 2005)*, 2005.

Appendices

Appendix A

Random SCSP Generator

In this appendix, we describe our random generator for the problems we used in our experiments in Chapter 5 and Chapter 6.

Our random SCSP generator is based on a random CSP generator supplied at the url: <http://www.lirmm.fr/~bessiere/generator.html>. Please consult this website for a description of the random CSP generator.

A.1 The random problem model

Our SCSP generator requires seven parameters:

- The number of decision variables in the problem.
- The number of values in the domain of each decision variable. Each decision variable has a domain of the same size.
- The number of constraints. All constraints are binary (between exactly two variables). The constraints are chosen at random from a uniform distribution.
- The tightness of each constraint. All constraints have the same tightness. Tightness refers to the number of value pairs that do not have an associated semiring value that is equal to the maximum semiring value. These specific pairs are chosen at random from a uniform distribution. Tightness is specified as an integer. For instance, if a problem has variables with domain size of 5, then the maximum

number of value pairs that do not have the maximal semiring value associated, is $5 * 5 = 25$.

- A bound on the maximum semiring value, called P . The set of semiring values consists of the integers between and including 0 and $P - 1$. The integer 0 is the minimum semiring value and $P - 1$ is the maximum semiring value.
- A seed value for the generator to insure that the same set of problems can be reproduced.
- The number of instances of problems that are to be generated.

A.2 An example

Given the set of parameters 25 5 5 23 5 1 1, our SCSP generated the following problem:

```
25 5 5 23 5
5 15 : 3 2 3 1 0 0 2 3 4 2 3 2 4 1 2 2 2 0 0 0 0 2 2 1 2
21 23 : 3 2 3 0 2 1 1 1 3 3 1 4 0 1 0 4 3 0 2 3 1 3 1 2 2
4 5 : 3 0 4 3 0 2 2 3 3 3 3 2 3 0 2 3 3 0 2 0 0 3 1 3 4
8 16 : 4 0 1 2 2 2 2 1 2 2 2 0 2 3 2 3 2 1 0 3 4 3 3 2 1
8 12 : 2 0 2 2 0 3 3 2 2 0 0 4 3 0 0 0 3 0 2 2 1 3 3 4 0
```

The first line depicts the generated SCSP: This problem has 25 variables, each variable with a domain size 5, 5 constraints, a tightness of 23, a maximum semiring value of 4 (and a minimum semiring value of 0). Note that there are only 7 variables that occur in the types of the five constraints.

Each of the remaining five lines depicts one constraint. The two values before the colon on a line, are the two decision variables in the type of that constraint. The remaining values show the semiring values associated with each value pair where the value pairs are assumed to be in lexicographical order. Note that there are only two value pairs in

each line that has the maximum semiring value, 4.

This particular problem has been used in the running example of Chapter 6. We renamed the seven decision variables that are involved in constraints in the following way: variable 5 is named variable A , variable 15 is named variable B , variable 21 is renamed variable C , variable 23 is renamed variable D , variable 4 is renamed variable E , variable 8 is renamed variable F , variable 16 is renamed variable F , and variable 12 is renamed variable G .

Please consult Table 6.1 to see the semiring values associated with each value tuple of each of the constraints. The constraints are numbered from 0 to 4.

Appendix B

List of Definitions

Def. no	Description	Page no
1	Constraint Satisfaction Problem (CSP)	5
2	c-semiring	19
3	Constraint system	20
4	Constraint	20
5	Semiring Constraint Satisfaction Problem (SCSP)	21
6	The projection of a tuple from a set W to a set W'	23
7	The combination of two constraints	23
8	The projection of a constraint over a set of variables	23
9	Solution of a SCSP	23
10	Maximal Solution of a SCSP	24
11	Filter and Principal Filter	35
12	A Good Enough Solution for a SCSP	36
13	Weakened Constraint	36
14	W_c , the set of all c -weakened constraints	37
15	$wdef_c^d$, a difference function for a weakened constraint	38
16	d -relaxation of a SCSP	38
17	Maximal Solution of a d -relaxation of a SCSP	39
18	Difference between a d -relaxation and a SCSP	39
19	Maximal Weakened d -relaxations of a SCSP	39
20	Combined Semiring	46
21	Combined Semiring Relaxation	46
22	Solution of a Combined Semiring Relaxation	47
23	Maximal Solution of a Combined Semiring Relaxation	47
24	Good enough Maximal Combined Semiring Relaxations	48
25	Solution tuple for the WS-Max-SAT Encoding	61
26	Weighted Semiring Max-SAT Problem (WS-Max-SAT)	62