Proceedings of the 10th International Workshop
on Preferences and Soft Constraints
(SofT-10)

Held in conjunction with the 16th International
Conference on Principles and Practice of Constraint
Programming (CP-2010)

September 6, 2010
University of Saint Andrews, Scotland

Organized by

Simon de Givry
Felip Manyà
Joao Marques-Silva
Preface

This volume contains the papers presented at SofT-10: 10th Workshop on Preferences and Soft Constraints held on 6th September 2010 and co-located with CP 2010 in Saint Andrews, Scotland.

Preferences are ubiquitous in real life: most problems are over-constrained and would not be solvable if we insist that all their requirements are strictly met. Moreover, many problems are more naturally described via preferences rather than hard statements. Soft constraints are the way the constraint community has extended its classical framework to deal with the concept of preferences.

The SofT-10 workshop brings together researchers interested in all aspects of soft constraints and cost function processing, such as:

- theoretical frameworks
- problem modeling
- solving algorithms
- languages
- preference aggregation and elicitation
- multi-objective or qualitative optimization
- combining/integrating different frameworks and algorithms
- comparative studies
- real-life applications

The workshop is an opportunity to share knowledge between researchers working on algorithms and solvers for different formalisms, including Weighted Max-SAT, Soft CSP, Bayesian Networks, Random Markov Field, Factor Graphs, Pseudo Boolean Optimization, SAT Modulo Theories, and related formalisms.

The workshop received 10 submissions. Each submission was reviewed by at least 3 programme committee members. The committee decided to accept all papers. The programme gives a broad overview of the various researches done in the field of Preferences and Soft Constraints. There are papers presenting original algorithms to tackle new challenging optimization tasks such minimizing decision changes, finding the next optimal solution, or solving distributed constraint optimization problems. Other papers propose new complete or incomplete or parallel methods to solve fuzzy or weighted CSPs. Last but not least a few papers deal with related frameworks such as qualitative preferences in SAT, qualitative temporal CSP, and argumentation or voting theory. The programme also includes an invited talk on 2010 MaxSAT Evaluation Report by Felip Manyà.

These Proceedings have been generated by EasyChair.

August 2010

Simon de Givry
Felip Manyà
Joao Marques-Silva
Soft-10 Workshop Organization

Programme Chairs

Simon de Givry, INRA, France
Felip Manyà, IIIA-CSIC, Spain
Joao Marques-Silva, University College Dublin, Ireland

Programme Committee

Fahiem Bacchus, University of Toronto, Canada
Hachemi Bennaceur, CRIL, France
Stefano Bistarelli, University of Perugia, Italy
Jimmy Lee, Chinese University of Hong Kong, China
Chu-Min Li, Université de Picardie, France
Ines Lynce, Technical University of Lisbon, Portugal
Pedro Meseguer, IIIA-CSIC, Spain
Thierry Petit, LINA-CNRS, France
Jordi Planes, Universitat de Lleida, Spain
Emma Rollon, Technical University of Catalonia, Spain
Francesca Rossi, University of Padova, Italy
Thomas Schiex, INRA, France
Kristen Brent Venable, University of Padova, Italy
Gérard Verfaillie, ONERA, France
Tomas Werner, Czech Technical University, Czech Republic
Nic Wilson, 4C, Ireland

Additional Reviewers

Paola Campli
Ka Lun Leung
Francesco Santini
Zhu Zhu
# Table of Contents

Argumentation-based Interactions among Agents: Solving Weighted and Bipolar Frameworks with Soft CSPs .......................... 5  
*Stefano Bistarelli, Daniele Pirolandi, Francesco Santini*

Finding the next solution in constraint- and preference-based knowledge representation formalisms ........................................ 21  
*Ronen Brafman, Francesca Rossi, Domenico Salvagnin, K. Brent Venable, Toby Walsh*

Multi-agent soft constraint aggregation: a sequential approach ........ 36  
*Giorgio Dalla Pozza, Francesca Rossi, Kristen Brent Venable*

Load Balancing for Parallel Branch and Bound .......................... 51  
*Rina Dechter, Lars Otten*

Combining approaches for solving Satisfiability problems with Preferences and their Evaluation ................................. 66  
*Emanuele Di Rosa, Enrico Giunchiglia, Barry O’Sullivan*

A local search approach to solve incomplete fuzzy and weighted CSPs ... 82  
*Mirco Gelain, Maria Silvia Pini, Francesca Rossi, Kristen Brent Venable, Toby Walsh*

Enforcing Soft Local Consistency on Multiple Representations for DCOP Solving .................................................. 98  
*Patricia Gutierrez, Pedro Meseguer*

Model Tracking for Dynamic SAT with Decision Change Costs ....... 113  
*Katsutoshi Hirayama, Daisuke Hatano, Yuta Sugimoto*

Ordering Intervals: From Qualitative Temporal Constraint Satisfaction Problems to Preference Representation ..................... 128  
*Souhila Kaci, Cédric Piette*

A decomposition method for valued CSPs .................................. 143  
*Wady Naanaa, Maher Helaoui, Béchir Ayeb*
Argumentation-based Interactions among Agents: Solving Weighted and Bipolar Frameworks with Soft CSPs

Stefano Bistarelli\textsuperscript{1,2}, Daniele Pirolandi\textsuperscript{1} and Francesco Santini\textsuperscript{1}

\textsuperscript{1} Dipartimento di Matematica e Informatica, Universit\`a di Perugia, Italy 
\{bista,francesco.santini\}@dmi.unipg.it, daniele.pirolandi@gmail.com 
\textsuperscript{2} Istituto di Informatica e Telematica (CNR), Pisa, Italy

stefano.bistarelli@iit.cnr.it

Abstract. Some interactions among agents can be well modeled with argumentation theory. We suggest soft constraints as a mean to parametrically represent and solve “weighted” and bipolar argumentation problems: different kinds of (bipolar) preference levels related to arguments, e.g. a score representing a “fuzziness”, a “cost” or a probability level of each argument, can be represented by choosing different semiring algebraic structures. The novel idea is to provide a common computational and quantitative framework where the computation of the classical Dung’s semantics has an associated score representing “how much good” the computed set is. Moreover, we propose a mapping from weighted and bipolar Argumentation Frameworks to Soft Constraint Satisfaction Problems (SCSPs); with this mapping we can compute Dung semantics (e.g. admissible and stable) by solving the related SCSP. To implement this mapping we use JaCoP, a Java constraint solver.

1 Introduction

Argumentation [15] is based on the exchange and the evaluation of interacting arguments which may represent information of various kinds, especially beliefs or goals. Argumentation can be used for modeling some aspects of reasoning, decision making, and dialogue. For instance, when an agent has conflicting beliefs (viewed as arguments), a (nontrivial) set of plausible consequences can be derived through argumentation from the most acceptable arguments for the agent. Argumentation can be seen as the process emerging from exchanges of among agents to persuade each other and bring about a change in intentions [24,22]. Argumentation has become an important subject of research in Artificial Intelligence and it is also of interest in several disciplines, such as Logic, Philosophy and Communication Theory [26].

\textsuperscript{*} This work has been partially supported by MIUR PRIN 20089M932N project: “Innovative and multi-disciplinary approaches for constraint and preference reasoning”.

\textsuperscript{\star} This work has been partially supported by MIUR PRIN 20089M932N project: “Innovative and multi-disciplinary approaches for constraint and preference reasoning”.
Many theoretical and practical developments build on Dung’s seminal theory of argumentation. A Dung Argumentation Framework (AF) is a directed graph consisting of a set of arguments and a binary conflict-based attack relation among them. The sets of arguments to be considered are then defined under different semantics, where the choice of semantics equates with varying degrees of scepticism or credulousness.

The other ingredient in our research is Constraint Programming [27]. The idea behind our semiring-based formalism [7, 5] was to further extend the classical constraint notion by adding the concept of a structure representing the levels of satisfiability of the constraints. Such a structure (see Sec. 3 for further details) is a set with two operations: one + is used to generate an ordering over the preference levels, while × is used to combine these levels. Because of the properties required on such operations, this structure is similar to a semiring (see Sec. 3). From here the terminology of “semiring-based soft constraint” [7, 5] (and Sec. 3), that is, constraints with several levels of satisfiability, and whose levels are (totally or partially) ordered according to the semiring structure. In general, Problems defined according to this semiring-based framework are called Soft Constraint Satisfaction Problems (SCSPs).

In this paper we show that different weighted AFs based on fuzziness, probability or a preference in general (and already studied in literature, e.g. in [26, 3]), can be modeled and solved with the same soft constraint framework by only changing the related semiring in order to optimize the different criteria. Also classical AFs can be represented inside the soft framework by adopting the Boolean semiring. We provide a mapping from AFs to (S)CSPs in a way that the solution of the SCSP consists in the “best” desired extension, where “best” is computed by aggregating (with ×) the preference scores of all the chosen arguments, and comparing the final values (with +). The classical extensions of Dung can be found with our mapping, i.e. admissible, preferred, complete, stable and grounded ones. At last, we show an implementation of a CSP with JaCoP [25], a Java Constraint Programming solver.

Having a quantitative framework permits us to quantify the aggregation of chosen arguments and to prefer a set of arguments over another. Examples in the real world are represented by scores given to comments in Youtube or news in Slashdot, or topics in Discussion Fora in general [21]. As the set of arguments gets wider, the search of the best solutions becomes a demanding task, and constraint-based frameworks come with many and powerful solving techniques: notice that deciding if a set is a preferred extension is a \textit{CO-NP}-complete problem [4]. Within the same framework we can also describe and solve two or more weighted AFs at the same time (with the cartesian product of semirings [7]) and bipolar AFs (i.e. both attacks and supports) with bipolar preference structure based on semirings [8], leading to a representation of a richer argumentation structure.

The remainder of this paper is organized as follows. In Sec. 2 we report the theory behind Dung Argumentation, while in Sec. 3 we summarize the background about soft constraints. Section 4 shows the basic idea of weighted AF based on semirings; in Sec. 5 we propose the mapping from AFs to SCSPs, the
proofs of their solution equivalence and we show a practical encoding in JaCoP. Section 6 presents how to deal with bipolar extensions. A comparison with related work is given in Sec. 7. Finally, Sec. 8 presents our conclusions.

2 Dung Argumentation

In [15], the author has proposed an abstract framework for argumentation in which he focuses on the definition of the status of arguments. For that purpose, it can be assumed that a set of arguments is given, as well as the different conflicts among them. An argument is an abstract entity whose role is solely determined by its relations to other arguments.

Definition 1. An Argumentation Framework (AF) is a pair \( \langle \mathcal{A}_{rgs}, R \rangle \) of a set \( \mathcal{A}_{rgs} \) of arguments and a binary relation \( R \) on \( \mathcal{A}_{rgs} \) called the attack relation. \( \forall a_i, a_j \in \mathcal{A}, a_i R a_j \) means that \( a_i \) attacks \( a_j \). An AF may be represented by a directed graph (the interaction graph) whose nodes are arguments and edges represent the attack relation. A set of arguments \( \mathcal{B} \) attacks an argument \( a \) if \( a \) is attacked by an argument of \( \mathcal{B} \). A set of arguments \( \mathcal{B} \) attacks a set of arguments \( \mathcal{C} \) if there is an argument \( b \in \mathcal{B} \) which attacks an argument \( c \in \mathcal{C} \).

![Fig. 1. An example of Dung Argumentation Framework; e.g. c attacks d.](image)

In Fig. 1 we show an example of AF represented as an interaction graph: the nodes represent the arguments and the directed arrow from \( c \) to \( d \) represents the attack of \( c \) towards \( d \), that is \( c R d \). Dung [15] gave several semantics to “acceptability”. These various semantics produce none, one or several acceptable sets of arguments, called extensions. In Def. 2 we define the concepts of conflict-free and stable extensions:

Definition 2. A set \( \mathcal{B} \subseteq \mathcal{A}_{rgs} \) is conflict-free iff it does not exist two arguments \( a \) and \( b \) in \( \mathcal{B} \) such that \( a \) attacks \( b \). A conflict-free set \( \mathcal{B} \subseteq \mathcal{A}_{rgs} \) is a stable extension iff for each argument which is not in \( \mathcal{B} \), there exists an argument in \( \mathcal{B} \) that attacks it.

The other semantics for “acceptability” rely upon the concept of defense:

Definition 3. An argument \( b \) is defended by a set \( \mathcal{B} \subseteq \mathcal{A}_{rgs} \) (or \( \mathcal{B} \) defends \( b \)) iff for any argument \( a \in \mathcal{A}_{rgs} \), if \( a \) attacks \( b \) then \( \mathcal{B} \) attacks \( a \).
An admissible set of arguments according to Dung must be a conflict-free set which defends all its elements. Formally:

**Definition 4.** A conflict-free set $B \subseteq \mathcal{A}_{rgs}$ is admissible iff each argument in $B$ is defended by $B$.

Besides the stable semantics, three semantics refining admissibility have been introduced by Dung [15]:

**Definition 5.** A preferred extension is a maximal (w.r.t. cardinality) admissible subset of $\mathcal{A}_{rgs}$. An admissible $B \subseteq \mathcal{A}_{rgs}$ is a complete extension iff each argument which is defended by $B$ is in $B$. The least (w.r.t. cardinality) complete extension is the grounded extension.

A stable extension is also a preferred extension and a preferred extension is also a complete extension. Stable, preferred and complete semantics admit multiple extensions whereas the grounded semantics ascribes a single extension to a given argument system.

Notice that deciding if a set is a stable extension or an admissible set can be computed in polynomial time, but deciding if a set is a preferred extension is a $CO$-$NP$-complete problem [4].

### 3 Soft Constraints

A $c$-semiring [7, 5] $S$ (or simply semiring in the following) is a tuple $(A, +, \times, 0, 1)$ where $A$ is a set with two special elements $(0, 1 \in A)$ and with two operations $+$ and $\times$ that satisfy certain properties: $+$ is defined over (possibly infinite) sets of elements of $A$ and thus is commutative, associative, idempotent, it is closed and $0$ is its unit element and $1$ is its absorbing element; $\times$ is closed, associative, commutative, distributes over $+$, $1$ is its unit element, and $0$ is its absorbing element (for the exhaustive definition, please refer to [7]). The $+$ operation defines a partial order $\leq_S$ over $A$ such that $a \leq_S b$ iff $a + b = b$; we say that $a \leq_S b$ if $b$ represents a value better than $a$. Other properties related to the two operations are that $+$ and $\times$ are monotone on $\leq_S$, $0$ is its minimum and $1$ is its maximum, $(A, \leq_S)$ is a complete lattice and $+$ is its least upper bound. Finally, if $\times$ is idempotent, then $+$ distributes over $\times$, $(A, \leq_S)$ is a complete distributive lattice and $\times$ its greatest lower bound.

A soft constraint [7, 5] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = (A, +, \times, 0, 1)$ and an ordered set of variables $V$ over a finite domain $D$, a soft constraint is a function which, given an assignment $\eta : V \rightarrow D$ of the variables, returns a value of the semiring. Using this notation $C = \eta \rightarrow A$ is the set of all possible constraints that can be built starting from $S$, $D$ and $V$. Any function in $C$ involves all the variables in $V$, but we impose that it depends on the assignment of only a finite subset of them. So, for instance, a binary constraint $c_{x,y}$ over variables $x$ and $y$, is a function $c_{x,y} : V \rightarrow D \rightarrow A$, but it depends only on the assignment of variables...
{x, y} ⊆ V (the support of the constraint, or scope). Note that \(c\eta[v := d_1]\) means \(c\eta'\) where \(\eta'\) is \(\eta\) modified with the assignment \(v := d_1\). Note also that \(c\eta\) is the application of a constraint function \(c : V \rightarrow D \rightarrow A\) to a function \(\eta : V \rightarrow D\); what we obtain, is a semiring value \(c\eta = a\). \(\bar{0}\) and \(\bar{1}\) respectively represent the constraint functions associating \(0\) and \(1\) to all assignments of domain values (i.e. the \(\bar{a}\) function returns the semiring value \(a\)).

Given the set \(C\), the combination function \(\otimes : C \times C \rightarrow C\) is defined as 
\[
(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta
\] (see also [7, 5]). Informally, performing the \(\otimes\) or between two constraints means building a new constraint whose support involves all the variables of the original ones, and which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate sub-tuples.

Given a constraint \(c \in C\) and a variable \(v \in V\), the projection [7, 5, 6] of \(c\) over \(V - \{v\}\), written \(c \downarrow_{(V\setminus\{v\})}\) is the constraint \(c'\) such that \(c'\eta = \sum_{d \in D} c\eta[v := d]\). Informally, projecting means eliminating some variables from the support.

A SCSP [5] defined as \(P = \langle C, con \rangle\) (\(C\) is the set of constraints and \(con \subseteq V\), i.e. a subset the problem variables). The best level of consistency notion is defined as \(blevel(P) = Sol(P) \downarrow_\emptyset\), where \(Sol(P) = (\otimes C) \downarrow_{con}\) [5]. A problem \(P\) is \(\alpha\)-consistent if \(blevel(P) = \alpha\) [5]; \(P\) is instead simply “consistent” if there exists \(\alpha > S_0\) such that \(P\) is \(\alpha\)-consistent [5]. \(P\) is inconsistent if it is not consistent.

### 4 Weighted Argumentation

Weighted argumentation systems [11, 16] extend Dung-style abstract argumentation systems by adding numeric weights to every node (or attack) in the attack graph, intuitively corresponding to the strength of the attack, or equivalently, how reluctant we would be to disregard it. To illustrate the need to extend the classical AF with preferences, we consider two individuals \(P\) and \(Q\) exchanging arguments \(A\) and \(B\) about the weather forecast (the example is taken from [26]):

\[
\text{P:} \quad \text{Today will be dry in London since BBC forecast sunshine} = A \\
\text{Q:} \quad \text{Today will be wet in London since CNN forecast rain} = B
\]

\(A\) and \(B\) claim contradictory conclusions and so attack each other. Under Dung’s preferred semantics, there are two different admissible extensions represented by the sets \(\{A\}\) and \(\{B\}\), but neither argument is sceptically justified. One solution is to provide some means for preferring one argument to another in order to find a more informative answer, for example, the most trustworthy extension. For example, one might reason that \(A\) is preferred to \(B\) because the \(BBC\) are deemed more trustworthy than \(CNN\). Suppose to have a fuzzy trust score associated with each argument, as shown in Fig. 2. This score, (between 0 and 1 that is between low and high trustworthiness) can be then used to prefer \(\{A\}\) with a score of 0.9 over \(\{B\}\) with a score of 0.7, i.e. forecast from \(BBC\) than from \(CCN\).

In some works [20] the preference score is associated with the attack relationship instead of with the argument itself and, thus, it models the “strength”
of the attack, e.g. a fuzzy attack. This model can be cast in ours by composing these strengths in a value representing the preference of the argument, as in Fig. 3, where the trustworthiness of argument CNN-rain can be computed as the mathematical mean (or in general a function \( \odot \), as defined also in [10] for computing the trust of a group of individuals) of the values associated with the attack towards it, i.e. \((0.9 + 0.5) \div 2 = 0.7\). Computing a trust evaluation of a node by considering a function of the links ending in it is a well-known solution, e.g. the PageRank of Google [21]. By composing attack and support values, it is also possible to quantitatively study bipolar argumentation frameworks [1].

\[
\begin{array}{ccc}
\text{BBC} & \text{sunshine} & \text{CNN} \\
0.9 & \rightarrow & 0.7 \\
\text{rain} & \rightarrow & \text{sunshine} \\
\end{array}
\]

**Fig. 2.** The CNN/BBC example with trust scores.

\[
\begin{array}{ccc}
\text{BBC} & \text{sunshine} & \text{CNN} \\
0.9 & \rightarrow & 0.7 \\
\text{rain} & \rightarrow & \text{sunshine} \\
\end{array}
\]

**Fig. 3.** A fuzzy Argumentation Framework with fuzzy scores modeling the attack strength.

Notice that in [26, 3, 2] the preference among arguments is given in a qualitative way, that is argument \( a \) is better than argument \( b \), which is better than argument \( c \); in this section we study the problem from a quantitative point of view, with scores associated with arguments. We suggest the algebraic semiring structure (see Sec. 3) as a mean to parametrically represent and solve all the “weighted” AFs presented in literature (see Sec. 7), i.e. to represent the scores; in the following we provide some examples on how semirings fulfil these different tasks:

- An argument can be seen as a chain of possible events that makes the hypothesis true. The credibility of a hypothesis can then be measured by the total probability that it is supported by arguments. The proper semiring to solve this problem consists in the Probabilistic semiring [5]: \( \langle [0..1], \text{max}, \times, 0, 1 \rangle \), where the arithmetic multiplication (i.e. \( \times \)) is used to compose the probability values together.
- The Fuzzy Argumentation [29] approach enriches the expressive power of the classical argumentation model by allowing to represent the relative strength of the attack relationships between arguments, as well as the degree to which
arguments are accepted. In this case, the Fuzzy semiring \(([0..1], \text{min}, \text{max}, 0, 1)\) can be used.

– In addition, the Weighted semiring \((\mathbb{R}^+, \text{min}, \hat{+}, 0, 1)\), where \(\hat{+}\) is the arithmetic plus, can model the (e.g. money) cost of the attack: for example, during an electoral campaign, a candidate could be interested in how many efforts or resources he should spend to counteract an argument of the opposing party.

– At last, with the Boolean semiring \(\{\text{true}, \text{false}\}, \lor, \land, \text{false}, \text{true}\) we can cast the classic AFs originally defined by Dung [15] in the same semiring-based framework.

Moreover, notice that the cartesian product of two semirings is still a semiring [7, 5], and this can be fruitfully used to describe multi-criteria constraint satisfaction and optimization problems. For example, we can have both a probability and a fuzzy score given by a couple \(t, f\); we can optimize both costs at the same time.

We can extend the definitions provided in Sec. 3 in order to express all these weights of the attack relations with a semiring based environment. The following definition models the semiring-based problem.

**Definition 6.** A semiring-based Argumentation Framework (AFS) is a quadruple \(\langle \text{Args}, R, W, S \rangle\) of a semiring \(S = \langle A, +, \times, 0, 1 \rangle\), a set \(\text{Args}\) of arguments, the attack binary relation \(R\) on \(\text{Args}\), and a unary function \(W : \text{Args} \rightarrow A\) called the weight function. \(\forall a \in \text{Args}, W(a) = s\) means that \(a\) has a preference level \(s \in A\).

Therefore, the weight function \(W\) associates each argument with a semiring value \((s \in A)\) that represents the preference expressed for that argument in terms of cost, fuzziness and so on. For example, using the Fuzzy semiring \(([0..1], \text{min}, \text{max}, 0, 1)\) semiring for the problem represented in Fig. 2 allows us to state that the admissible extension \(\{A\}\) (with a score of 0.9) is better than the other admissible extension \(\{B\}\) (with a s.core of 0.7) since 0.9 > 0.7. Therefore, with an AFS our goal is to find the extensions proposed by Dung (e.g. the admissible extensions), but with an associated preference value. Therefore, soft constraints can be used to solve these problems while considering also the best solution(s) (according to the notion of blevel, and to cut the solutions with a preference below a threshold \(\alpha\).

**Example 1.** Concerning the interaction graph in Fig. 4, it represents the Weighted AFS \(W = (\text{Args}, R)\) with \(S = (\mathbb{R}^+, \text{min}, \hat{+}, \infty, 0)\) and \(\text{Args} = \{a, b, c, d, e\}\), \(R(a, b) = 0.7\), \(R(c, b) = 0.8\), \(R(c, d) = 0.9\), \(R(d, c) = 0.8\), \(R(d, e) = 0.5\), \(R(e, e) = 0.6\) and \(W(a) = 7\), \(W(b) = 20\), \(W(c) = 6\), \(W(d) = 10\), \(W(e) = 12\). Notice that \(e\) attacks itself, that is in contrast with itself, e.g. “We have sunshine and it’s raining” (it may be possible).
In this section we present a mapping from AF (and AFs) to (S)CSPs. Given an AF $S = (\mathcal{A}_{rgs}, R, W, S)$, we define a variable for each argument $a_i \in \mathcal{A}_{rgs}$, i.e. $V = \{a_1, a_2, \ldots, a_n\}$ and each of these argument can be taken or not, i.e. the domain of each variable is $D = \{1, 0\}$, and if it is taken, a cost in the semiring can be assigned, mapping the level of preference of this argument.

To represent the quantitative preference over arguments, in this mapping we need only unary soft constraints on each variable, while the other constraints modeling, for example, the conflict-free relationship (see Sec. 2) are crisp even if represented in the soft framework. We plan to extend also these constraints to properly-said soft ones as suggested in Sec. 8. In the following explanation, notice that $b$ attacks $a$ means that $b$ is a parent of $a$ in the interaction graph, and $c$ attacks $b$ attacks $a$ means that $c$ is a grandparent of $a$. To compute the (weighted) extensions of Dung we need to define specific sets of constraints:

1. **Preference constraints.** The weight function $W(a_i) = s (s \in A)$ of an AF $S$ can be modeled with the unary constraints $c_{a_i}(a_i = 1) = s$, otherwise, when $a_i$ is assigned to 0, the argument is not taken in the considered extension and so its cost must not be computed.

2. **Conflict-free constraints.** Since we want to find the conflict-free sets, if $R(a_i, a_j)$ is in the graph we need to prevent the solution to include both $a_i$ and $a_j$ in the considered extension: $c_{a_i, a_j}(a_i = 1, a_j = 1) = 0$. For the other possible assignment of the variables ($(a = 0, b = 1)(a = 1, b = 0)$ and $(a = 0, b = 0)$), $c_{a_i, a_j} = 1$, since these assignments are permitted: in these cases we are choosing only one argument between the two (or none of the two) and thus, we have no conflict.

3. **Admissible constraints.** For the admissibility, we need that, if child argument $a_i$ has a parent node $a_f$ but $a_i$ has no grandparent node $a_g$ (parent of $a_f$), then we must avoid to take $a_i$ in the extension because it is attacked and cannot be defended by any ancestor: expressed with a unary constraint, $c_{a_i}(a_i = 1) = 0$. Moreover, if $a_i$ has several grandparents $a_{g1}, a_{g2}, \ldots, a_{gk}$ w.r.t. parent $a_f$ (child of $a_{g1}, a_{g2}, \ldots, a_{gk}$), we need to add a $k + 1$-ary constraint $c_{a_i, a_{g1}, \ldots, a_{gk}}(a_i = 1, a_{g1} = 0, \ldots, a_{gk} = 0) = 0$. The explanation is

![Fig. 4. An example of a weighted interaction graph.](image_url)
that at least a grandparent must be taken in the admissible set, in order to
defend \( a_i \) from one of his parents \( a_f \). Notice that, if a node is not attacked (i.e. he has no parents), he can be taken or not in the admissible set.

4. **Complete constraints.** To compute a complete extension \( \mathcal{B} \), we need that each argument \( a_i \) which is defended by \( \mathcal{B} \) is in \( \mathcal{B} \) (see Sec. 2). This can be enforced by imposing that for each \( a_i \) taken in the extension, also all its \( a_s_1, a_s_2, \ldots, a_s_k \) grandchildren must be taken in the extension, i.e. 

\[ c_{a_i,a_s_1,\ldots,a_s_k}(a_i = 1, a_s_1 = 1, \ldots, a_s_k = 1) = 1 \]

and also if \( a_i = 0 \) this constraint is satisfied; \( 0 \) otherwise.

5. **Stable constraints.** If we have a child node \( a_i \) with multiple parents \( a_{f_1}, a_{f_2}, \ldots, a_{f_k} \), we need to add the constraint 

\[ c_{a_i,a_{f_1},\ldots,a_{f_k}}(a_i = 0, a_{f_1} = 0, \ldots, a_{f_k} = 0) = 0 \]

In words, if a node is not taken in the extension (i.e. \( a_i = 0 \)), then it must be attacked by at least one of the taken nodes, that is at least a parent of \( a_i \) needs to be taken in the stable extension (that is, \( a_{f_j} = 1 \)). Moreover, if a node \( a_i \) has no parent in the graph, it has to be included in the stable extension (notice \( a_i \) cannot be attacked by nodes inside the extension, since he has no parent). The corresponding unary constraint is 

\[ c_{a_i}(a_i = 0) = 0 \]

Notice that by using the Boolean semiring, also the class of preference constraints becomes crisp and we can consequently model classical Dung AFs, that is not weighted frameworks. The following proposition states the equivalence between solving an \( \mathcal{AF}_S \) and its related SCSP.

**Proposition 1 (Solution equivalence).** Given an \( \mathcal{AF}_S = \langle \mathcal{A}_{rgs}, R, W, S \rangle \) and 
\( S = \langle A, +, \times, 0, 1 \rangle \), the solutions of the related SCSP obtained with the mapping corresponds to find over \( \mathcal{AF}_S \) the best (according to +)

- conflict-free extensions by using preference and conflict-free constraint classes.
- admissible extensions by using preference, conflict-free and admissible constraint classes.
- complete extensions by using preference, conflict-free and admissible constraint classes.
- stable extensions by using preference, conflict and stable constraint classes.

By using the Boolean semiring the solutions of the \((S)CSP \) respectively correspond to all the classical admissible, complete and stable extensions of Dung [15]. Moreover, to find the preferred extension (see Sec. 2) we simply need to find all the maximal (w.r.t. set inclusion) admissible extensions of \( \mathcal{A}_{rgs} \), that is to find all the admissible sets and then to return only those subsets with the maximal set of variables assigned to 1. Similar considerations hold for the grounded extension (see Sec. 2), that is we need to find all the complete extensions (the first four classes of constraints) and then to return only those subsets with the minimal set of variables assigned to 1.

\[ ^3 \text{Different interpretations of grounded/preferred extensions can be given by considering their cost instead of their the cardinality.} \]
As suggested in Sec. 4, an AF can be represented as a weighted interaction graph as in Fig. 4, where we instead suppose to use a Weighted semiring, i.e. $(\mathbb{R}^+, \min, \hat{+}, \infty, 0)$, e.g. the argument $a$ has received 7 negative comments. The goal in this case is to choose the extensions of Dung and to minimize the sum of the negative comments at the same time.

Notice that the presented soft constraint framework can be easily used to solve argumentation problems with additional constraints, as proposed in [14] only for boolean constraints. We can find further requirements on the sets of arguments which are expected as extensions, like “extensions must contain argument $a$ when they contain $b$” or “extensions must not contain one of $c$ or $d$ when they contain $a$ but do not contain $b$”.

Solving with JaCoP The Java Constraint Programming solver [25], JaCoP in short, is a Java library, which provides Java user with Finite Domain Constraint Programming paradigm. It provides different type of constraints: most commonly used primitive constraints, such as arithmetical constraints, equalities and inequalities, logical, reified and conditional constraints, combinatorial (global) constraints. The last version of JaCoP proposes many features, such as pruning events, multiple constraint queues, special data structures to handle efficiently backtracking, iterative constraint processing, and many more [25]. With JaCoP we have implemented all the constraints of the proposed mapping. The full description of the code can be found in [9].

In Fig. 5 we show the definition in JaCoP of all the conflict-free and stable constraints used to solve the AF example in Fig. 4. The full description of the code can be found in [9]. Considering for example the first conflict-free constraint $v[0], v[1]$, means that the constraint is between $a$ and $b$ and $(1, 1)$ that the the constraint is not satisfied if both variables are taken in the set.

Considering the example in Fig. 4 the admissible sets are: $\{a\}, \{c\}, \{d\}, \{a,c\}, \{a,d\}$. Dung’s semantics induce the following acceptable sets: one stable extension $\{a,d\}$, two preferred extensions $PE_1 = \{a,c\}, PE_2 = \{a,d\}$, three complete extensions $CE_1 = \{a,c\}, CE_2 = \{a,d\}, CE_3 = \{a\}$ and the grounded extension $\equiv \{a\}$. With our quantitative interpretation of AFs with preferences and considering the Weighted semiring $(\mathbb{R}^+, \min, \hat{+}, \infty, 0)$, we can prefer $PE_1$ over $PE_2$ ($W(a)\hat{+}W(c) = 13, W(a)\hat{+}W(d) = 17$ and $CE_3$ over $CE_1$ and $CE_2$, since $W(a) = 7$. All these best solutions are obtained by using JaCoP.

6 Bipolar Constraints and Bipolar AFs

In Sec. 2 we show that the description of a AF includes an attack relation $R$ defined on the $A$ set of arguments. However, an argument can also “support” another argument with a distinct $S$ relationship. It is the case for instance if an agent gives an argument which confirms a premise used by an argument provided by another agent. Each argument is associated with two weights respectively describing the positive and negative (i.e. the negative weight) preferences. An interaction graph is shown in Fig. 6.
// Defining the Variables of the SCSP
v[0] = new BooleanVariable(store, "a");
v[1] = new BooleanVariable(store, "b");
v[2] = new BooleanVariable(store, "c");
v[3] = new BooleanVariable(store, "d");
v[4] = new BooleanVariable(store, "e");

// conflict-free constraints
public static void imposeConstraintConflictFree(Store store, BooleanVariable[] v) {
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[1]}, new int[][]{{1, 1}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[1]}, new int[][]{{1, 1}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[3]}, new int[][]{{1, 1}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[2]}, new int[][]{{1, 1}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[4]}, new int[][]{{1, 1}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[4], v[4]}, new int[][]{{1, 1}}));
}

// admissible constraints
public static void imposeConstraintAdmissibleSet(Store store, BooleanVariable[] v) {
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[1]}, new int[][]{{0, 1}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[1]}, new int[][]{{0, 1}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[4]}, new int[][]{{0, 1}}));
}

// stable constraints
public static void imposeConstraintStableExtensions(Store store, BooleanVariable[] v) {
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0]}, new int[][]{{0}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[2], v[1]}, new int[][]{{0, 0, 0}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[3]}, new int[][]{{0, 0}}));
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[4]}, new int[][]{{0, 0}}));
}

// complete constraints
public static void imposeConstraintCompleteExtensions(Store store, BooleanVariable[] v) {
  store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0]}, new int[][]{{0}}));
}

Fig. 5. The constraint in JaCoP for the mapping of Fig. 4.

Fig. 6. A bipolar argumentation problem: negative values represent negative preferences.
This distinction between positive and negative preferences is supported by studies in cognitive psychology which have shown that these two types of preferences are independent and processed separately in the mind [13]. Bipolar argumentation frameworks have been already studied in [1, 12]. The support relation is assumed to be totally independent of the attack relation.

To represent also these bipolar preference problems we can adopt the soft constraint framework as well. The structure we can use to model negative preferences is exactly a semiring as presented in Sec. 3, while when dealing with positive preferences, we want two main properties to hold: combination should bring to better preferences, and indifference should be lower than all the other positive preferences. To model bipolar problems, a negative and a positive preference structure are linked together [8]. A combination operator between positive and negative preferences can be defined to model preference compensation, that is to compare positive against negative aspects and compensating them w.r.t. their strength. What we obtain is a bipolar preference structure [8].

**Definition 7.** A positive preference structure is a tuple \((P, +_p, \times_p, \bot_p, \top_p)\) s.t.

- \(+_p\), the additive operator, is commutative, associative, idempotent, with \(\bot_p\) as its unit element and \(\top_p\) as its absorbing element.
- \(\times_p\), the multiplicative operator, is associative, commutative and distributes over \(+_p\), \((a \times_p (b +_p c)) = (a \times_p b) +_p (a \times_p c)\), with \(\bot_p\) as its unit element and \(\top_p\) as its absorbing element.

The additive operator of this structure has the same properties as the corresponding one in semirings [8]. On the other hand, the multiplicative operator has different properties w.r.t. semirings. More precisely, the best element in the ordering (\(\top_p\)) is now its absorbing element, while the worst element (\(\bot_p\)) is its unit element. This reflects the desired behavior of the combination of positive preferences. An example of a positive preference structure is \(P = (\mathbb{R}^+, \max, +, 0, +\infty)\), where preferences are positive reals.

**Definition 8.** A bipolar preference structure is a tuple \((N, P, +, \times, \bot, \top, \Box)\), where:

- \((N, +_n, \times_n, \bot, \top)\) is a positive preference structure;
- \((N, +_n, \times_n, \bot, \top)\) is a semiring;
- \(+ : (N \cup P)^2 \rightarrow (N \cup P)\) is s.t. \(a_n +_n a_p = a_p\) for any \(a_n \in N\) and \(a_p \in P\); this operator induces as as partial ordering on \(N \cup P\) : \(\forall a, b \in N \cup P, a \leq b\) iff \(a + b = b\);
- \(\times : (N \cup P)^2 \rightarrow (N \cup P)\) is an operator (called the compensation operator) that, for all \(a, b, c \in N \cup P\), satisfies the following properties:
  - commutativity: \(a \times b = b \times a\);
monotonicity: if $a \leq b$, then $a \times c \leq a \times b$.

An example is $S_{bip} = \langle [-1..0], [0..1], \text{max}, \hat{+}, -1, 0, 1 \rangle$, with positive preferences between 0 and 1 and negative preferences between $-1$ and 0 (as represented in Fig. 6). Aggregation of positive preferences is $\text{max}$ and of negative preferences is $\text{min}$, while compensation between positive and negative preferences is $\text{sum}$, and the order is given by $\text{max}$. In the $\langle [0..1], [1..+\infty], \text{max}, \times, 0, 1, +\infty \rangle$ structure we use positive preferences between 1 and $+\infty$ and negative preferences between 0 and 1. Compensation is obtained by multiplying the preferences and ordering is again via $\text{max}$.

A bipolar constraint [8] is a constraint where each assignment of values to its variables is associated to one of the elements in a bipolar preference structure. A bipolar CSP $(V, C)$ [8] is then just a set of variables $V$ and a set of bipolar constraints $C$ over $V$. It can be then used to solve bipolar argumentation problems [1, 12] as the one in Fig. 6, for example, with the $S_{bip}$ semiring. Positive preferences represent what the agent really wants, and negative preferences what the agent rejects. This distinction between positive and negative preferences is supported by studies in cognitive psychology which have shown that these two types of preferences are independent and processed separately in the mind and, from the quantitative point of view, in our semiring-based bipolar framework.

7 Related Work

In [29], the authors have developed the notion of fuzzy unification and incorporated it into a novel fuzzy argumentation framework for extended logic programming: the attacks are associated to a fuzzy strength value, i.e. a $V$-attack. As well, a $V$-argument $A$ is $V$-acceptable w.r.t. the set $\text{Args}$ of $V$-arguments if each argument $V$-attacked $A$ is $V$-attacked by an argument in $\text{Args}$.

In [3], AFs have been also extended to Value Based Argumentation Frameworks (VAF) where $V$ is a generic nonempty set of values and $\text{Val}$ is a function which maps from elements of $\text{Args}$ to elements of $V$.

The work in [2] concerns the “acceptability” of arguments in preference-based argumentation frameworks. Preferences are represented with a preordering relationships (partial or total) that resembles the ordering defined by the $+$ operator of semirings (see Sec. 3).

Probabilistic Argumentation [19, 23]. This theory is an alternative approach for non-monotonic reasoning under uncertainty. It allows to judge open questions (hypotheses) about the unknown or future world in the light of the given knowledge. From a qualitative point of view, the problem is to derive arguments in favor and against the hypothesis of interest.

In [26] the author has extended Dung’s theory of argumentation to integrate metalevel argumentation about preferences. Dung’s level of abstraction is preserved, so that arguments expressing preferences are distinguished by being the source of a second attack relation that abstractly characterizes application of preferences by attacking attacks between the arguments that are the subject of the preference claims.
A close work is represented by [17]: there the authors introduce and investigate a natural extension of Dung's well known model of argument systems in which attacks are associated with a weight, indicating the relative strength of the attack. A key concept in that framework is the notion of an inconsistency budget, which characterizes how much inconsistency we are prepared to tolerate: given an inconsistency budget $\beta$, we would be prepared to disregard attacks up to a total cost of $\beta$.

**Comparison.** The framework proposed in this paper is able to solve all the above reported AFs (including the classical Dung framework [15]), both from the qualitative and (main novelty) quantitative point of view. Since in this paper we mainly propose a solving framework, we compare it with other related works.

In [17] weights are associated with attacks instead of arguments, as in our proposal. Moreover, no solving mechanism is proposed to solve the problems presented in the paper, even if their solution is proved to be difficult in the paper (e.g. NP-Complete). Moreover, in [17] the combination of the weights and the preference of the solution correspond to our Weighted semiring, while other possibilities are not considered.

In [22] crisp constraint have been used to model argumentation as constraint propagation in Distributed Constraint Satisfaction Problem (DSCP). Different agents represent the distributed points in the problem. The paper shows the appropriateness of constraints in solving large-scale argumentation systems. However, it seems to only solve classical problems, (i.e. no qualitative or quantitative extensions).

In literature there exist some frameworks based on Logic Programming-like languages. For example, the system ASPARTIX [18] is a tool for computing acceptable extensions for a broad range of formalizations of Dung’s argumentation framework and generalizations thereof, e.g. value-based AFs [3] or preference-based [2]. ASPARTIX relies on a fixed disjunctive datalog program which takes an instance of an argumentation framework as input, and uses the answer-set solver DLV for computing the type of extension specified by the user. However, ASPARTIX does not solve any quantitative argumentation case, as well as other Answer Set Programming systems [28].

In [11] the authors solve over-constrained weighted AF problems, where weights are associated with arcs and represent the cost of the attack between two arguments. The authors relax the notion of conflict-free, admissible, complete and stable extensions to $\alpha$-conflict-free ones (and also for the other extensions of Dung), in order to include in the same set also attacking arguments, whose attack costs are not worse than a threshold $\alpha$. Therefore, [11] solves an extended problem w.r.t. to the work presented in this paper, which instead solves the classical problems proposed by Dung.

8 Conclusions and Future Work

In the paper we have revised the notions provided by Dung [15] in order to associate the bipolar argument preference with a weight (taken from a semiring
structure) that represents the “goodness” of the argument in terms of cost, fuzziness, probability or else. Further on, we have suggested the Dung’s semantics in their soft version. Moreover, we have presented a mapping from SCSPs to AFs and solved the obtained SCSP with JaCoP, a Java Constraint Programming solver, thus finding the solution of the related AF. We have proposed an unifying computational framework with strong mathematical foundations and solving techniques, where by only parametrically changing the semiring we can deal with different bipolar weighted (or not) AFs. By having a uniform framework, it may be possible to see more clearly the relationships between different proposals.

The user only needs to state the problem, while the underlying machinery is able to efficiently satisfy the constraints. Constraint solving techniques prove to be able to deal with large scale problems [22], even if the treated problems are difficult: for example, deciding if a set is a preferred extension is a \( CO-NP \)-complete problem [4]. Practical applications may consist, for example, in automatically study Discussion Fora where arguments are rated by users.

In the future, we would like to cluster arguments according to their (for example) coherence, still using soft constraints as the framework to obtain the solution. This can be useful to check the discrepancies/likeness during a negotiation process, inside different interviews to the same political candidate or during discussions in general. As an example, “We do not want immigrants with the right to vote” is clearly closer to “Immigration must be stopped”, than to “We need a multicultural and open society in order to enrich the life of everyone and boost our economy”, and should belong to the same cluster.

References


Finding the next solution in constraint- and preference-based knowledge representation formalisms

Ronen Brafman¹, Francesca Rossi², Domenico Salvagnin², K. Brent Venable², and Toby Walsh³

¹ Department of Computer Science Ben Gurion University Beer-Sheva, Israel
² Dipartimento di Matematica Pura ed Applicata, University of Padova, Italy,
³ NICTA and UNSW Sydney, Australia

Abstract. In constraint or preference reasoning, a typical task is to compute a solution, or an optimal solution. However, when one has already a solution, it may be important to produce the next solution following the given one in a linearization of the solution ordering, where more preferred solutions are ordered first. In this paper, we study the computational complexity of finding the next solution in some common preference-based representation formalisms. We show that this problem is hard in general CSPs, but it can be easy in tree-shaped CSPs and tree-shaped fuzzy CSPs. However, it is difficult in weighted CSPs, even if we restrict the shape of the constraint graph. We also consider CP-nets, showing that the problem is easy in acyclic CP-nets, as well as in constrained acyclic CP-nets where the (soft) constraints are tree-shaped and topologically compatible with the CP-net.

1 Introduction and motivation

In combinatorial satisfaction and optimization problems, the main task is finding a satisfying or optimal solution. There have been many efforts to develop efficient algorithms to perform such tasks, to study the computational complexity of this problem in general, and to find islands of tractability [9]. Another important task is to be able to compare two solutions and to say if one dominates another [3]. In this paper, we address another task that is crucial in many scenarios. When one has already a solution, it can be useful to be able to produce the next solution following the given one in the solution ordering where more preferred solutions are ordered first. If the solution ordering has ties or incompatibility, the next solution could be any solution which is tied or incomparable to the given one. In general, however, the next solution is the solution following the given one in a linearization of the solution ordering. The problem of finding the next solution is related to the problem of enumerating all the solutions of a model [6, 10] and to the ranking problem [12, 1], although it is quite different from the latter because we assume only a reference solution in input, and not all previous ones.

In this paper we study the computational complexity of the problem of returning the next solution in some constraint and preference-based formalisms. We show that this is a hard problem in constraint satisfaction problems (CSPs), but it can be easy
in tree-shaped CSPs [8] and tree-shaped fuzzy CSPs [14]. However, it is difficult in weighted CSPs, even if we restrict the shape of the constraint graph. Nevertheless, for the special case of unary constraints, it is only weakly NP-hard, since we give a pseudo-polynomial algorithm for it. Moreover, we also show that this problem is easy in acyclic CP-nets [3], as well as in constrained acyclic CP-nets [4] where the (soft) constraints are tree-shaped and topologically compatible with the CP-net graph.

We came across the problem of computing the next solution when studying the stable marriage problem [11]. This problem has many practical applications (from matching resident doctors to hospitals, to matching students to schools, to matching applicants to job offers, to any two-sided market). Stable marriage problems are usually solved with the Gale-Shapley algorithm. One of the main operations in this algorithm is computing the next solution in a preference ordering. However, the ability to compute the next solution is useful in many other scenarios. For instance, when we ask for the top $k$ solutions in a web search, we want to find the optimal solution and the next $k - 1$ solutions in the ordering. As a second example, suppose we are configuring a product, and the user doesn’t like the first configuration we compute as we only know their preferences partially. We might choose to compute the next most preferred solution according to the preferences that we do know. Finally, an efficient next operation can be used as a tool for producing diversity in solutions, a feature which is increasingly interesting in practical applications.

We intend to look for other tractable cases, and to investigate scenarios where the sufficient conditions for tractability, considered in this paper, naturally hold. We also plan to test experimentally how difficult it is in practice to find the next solution.

Parts of this paper appeared already in [5]. Due to lack of space, some formal proofs are omitted and others are only sketched.

2 Background

**Hard and soft constraints.** A soft constraint [14, 2] is a constraint [9] where each instantiation of its variables has an associated value from a (totally or partially ordered) set coming from a c-semiring. A c-semiring is defined by $\langle A, +, \times, 0, 1 \rangle$ where $A$ is this set of values, $+$ is a commutative, associative, and idempotent operator, $\times$ is used to combine preference values and is associative, commutative, and distributes over $+$, $0$ is the worst element, and $1$ is the best element. The c-semiring induces a partial or total order $\leq$ over preference values where $a \leq b$ iff $a + b = b$.

A classical CSP [9] is just a soft CSP where the chosen c-semiring is $S_{CSP} = \langle \{false, true\}, \lor, \land, \{false, true\} \rangle$. Fuzzy CSPs [14] are instead modeled with $S_{FCSP} = \langle [0, 1], \max, \min, 0, 1 \rangle$. That is, we maximize the minimum preference. For weighted CSPs, the c-semiring is $S_{WCSP} = \langle \mathbb{R}^+, min, +, +\infty, 0 \rangle$: preferences are interpreted as costs from 0 to $+\infty$, and we minimize the sum of costs.

Given an assignment $s$ to all the variables of an SCSP $P$, its preference, written $\text{pref}(P, s)$, is obtained by combining the preferences associated by each constraint to the subtuples of $s$ referring to the variables of the constraint. For example, in fuzzy CSPs, the preference of a complete assignment is the minimum preference given by the constraints. In weighted constraints, it is instead the sum of the costs given by the
constraints. An optimal solution of an SCSP $P$ is then a complete assignment $s$ such that there is no other complete assignment $s'$ with $\text{pref}(P, s) <_S \text{pref}(P, s')$.

Constraint propagation in classical CSPs reduces variable domains, and thus improves search performance. For some classes of constraints, constraint propagation is enough to solve the problem [8]. This is the case for tree-shaped CSPs, where directional arc-consistency, applied bottom-up on the tree shape of the problem, is enough to make the search for a solution backtrack-free. Given a variable ordering $o$, a CSP is directional arc-consistent (DAC) if, for any two variables $x$ and $y$ linked by a constraint $c_{xy}$, such that $x$ precedes $y$ in the ordering $o$, we have that, for every value $a$ in the domain of $x$ there is a value $b$ in the domain of $y$ such that $(a, b)$ satisfies $c_{xy}$.

Constraint propagation can be applied also to soft CSPs, and it maintains the usual properties, as in classical CSPs, if the soft constraint class is based on an idempotent semiring (that is, one where the combination operator is idempotent). This is the case for fuzzy CSPs, for example. As for classical CSPs, DAC is enough to find the optimal solution to a fuzzy CSP when the problem has a tree shape [14]. The same result holds for non-idempotent totally ordered fair soft CSPs [7].

Fuzzy CSPs can also be solved via a cut-based approach. Given a fuzzy CSP $P$, an $\alpha$-cut of $P$, where $\alpha$ is between 0 and 1, is a classical CSP with the same variables, domains, and constraint topology as the given fuzzy CSP, and where each constraint allows only the tuples that have preference above $\alpha$ in the fuzzy CSP. We will denote such a problem by $\text{cut}(P, \alpha)$. The set of solutions of $P$ with preference greater than or equal to $\alpha$ coincides with the set of solutions of $\text{cut}(P, \alpha)$.

CP-nets. CP-nets [3] are a graphical model for compactly representing conditional and qualitative preference relations. CP-nets are sets of ceteris paribus (cp) preference statements. For instance, the statement “I prefer red wine to white wine if meat is served.” asserts that, given two meals that differ only in the kind of wine served and both containing meat, the meal with red wine is preferable to the meal with white wine. A CP-net has a set of features $F = \{x_1, \ldots, x_n\}$ with finite domains $D(x_1), \ldots, D(x_n)$. For each feature $x_i$, we are given a set of parent features $Pa(x_i)$ that can affect the preferences over the values of $x_i$. This defines a dependency graph in which each node $x_i$ has $Pa(x_i)$ as its immediate predecessors. Given this structural information, the agent explicitly specifies her preference over the values of $x_i$, for each complete assignment on $Pa(x_i)$. This preference is assumed to take the form of total or partial order over $D(x_i)$. An acyclic CP-net is one in which the dependency graph is acyclic.

Consider a CP-net whose features are $A$, $B$, $C$, and $D$, with binary domains containing $f$ and $\overline{f}$ if $F$ is the name of the feature, and with the preference statements as follows: $a \succ \pi$, $b \succ \overline{b}$, $(a \land b) \lor (\pi \land \overline{b}) : c \succ \tau$, $(a \land \overline{b}) \lor (\pi \land b) : \tau \succ c$, $c : d \succ \overline{d}$, $\tau : D = d$. Here, statement $a \succ \pi$ represents the unconditional preference for $A=a$ over $A=\overline{a}$, while statement $c : d \succ \overline{d}$ states that $D=d$ is preferred to $D=\overline{d}$, given that $C=c$.

The semantics of CP-nets depends on the notion of a worsening flip. A worsening flip is a change in the value of a variable to a less preferred value according to the cp statement for that variable. For example, in the CP-net above, passing from $abcd$ to $abc\overline{d}$ is a worsening flip since $c$ is better than $\tau$ given $a$ and $b$. One outcome $\alpha$ is better than another outcome $\beta$ (written $\alpha \succ \beta$) iff there is a chain of worsening flips from $\alpha$
to β. This definition induces a preorder over the outcomes, which is a partial order if the CP-net is acyclic.

In general, finding the optimal outcome of a CP-net is NP-hard [3]. However, in acyclic CP-nets, there is only one optimal outcome and this can be found in linear time by sweeping through the CP-net, assigning the most preferred values in the preference tables. For instance, in the CP-net above, we would choose A=a and B=b, then C=c, and then D=d.

3 Solution orderings and linearizations

Each of the constraint or preference-based formalisms recalled in the previous section generates a solution ordering over the variable assignments, where solutions dominate non-solutions, and more preferred solutions dominate less preferred ones. This solution ordering can be a total order, a total order with ties, or even a partial order with ties. However, the problem of finding the next solution needs a strict linear order over the variable assignments, thus we will need to consider a linearization of the solution ordering.

CSPs generate a solution ordering which is total order with ties: all the solutions are in a tie (that is, they are equally preferred), and dominate in the ordering all the non-solutions, which again are in a tie. In soft constraints, the solution ordering is in general a partial order with ties: some assignments are equally preferred, others are incomparable, and others dominate each other. If we consider fuzzy or weighted CSPs, there can be no incomparability (since the set of preference values is totally ordered), so again we have a total order with ties, and a solution dominates another one if its preference value is higher. In this context, linearizing the solution ordering just means giving an order over the elements in each tie.

In acyclic CP-nets, the solution ordering is a partial order. In this scenario, any linearization of the solution ordering has to order every pair of incomparable assignments.

In the following, given a problem P and a linearization l of its solution ordering, we will denote with Next(P,s,l) the problem of finding the solution just after s in the linearization l. Note that, while there is only one solution ordering for a problem P, there may be several linearizations of such a solution ordering.

It is not tractable to compute l explicitly, since it has an exponential length and it would mean knowing all the solutions and their relative order. For these reasons, we will assume the linearization is implicitly given to the Next procedure. For example, a lexicographic order on the variable assignments induces a linearization of the solution ordering of a problem, yet it is polynomially describable.

4 Finding the next solution in CSPs

Let P be a CSP with n variables, and let us consider any variable ordering o = (x_1, ..., x_n) and any value orderings o_1, ..., o_n, where o_i is an ordering over the values in the domain of variable x_i. We will denote with O the set of orderings \{o, o_1, ..., o_n\}. These orderings naturally induce a lexicographical linearization of the solution ordering, that we call \text{l}ex(O), where, given two variable assignments, say s and s', we write
$s \preceq_{lex(O)} s'$ (that is, $s$ precedes $s'$) if either $s$ is a solution and $s'$ is not, or $s$ precedes $s'$ in the lexicographic order induced by $O$ (that is, $s = (s_1, \ldots, s_n)$, $s' = (s'_1, \ldots, s'_n)$, and there exists $i \in [1, n]$ such that $s_i \preceq_{o_i} s'_i$ and $s_j = s'_j$ for all $j < i$). We will now show that, if we take the linearization given by $lex(O)$, the problem of finding the next solution is NP-hard.

**Theorem 1** Computing $Next(P, s, lex(O))$, where $P$ is a CSP, $s$ is one of its solutions and $O$ is a set of orderings, is NP-hard.

The proof is based on a reduction from SAT.

This result can be extended to a wider class of orderings, as the following theorem states.

**Theorem 2** For each polynomially describable total order $\omega$ over complete variable assignments such that its top element does not depend on the constraints of the CSP and is polynomially computable, let us consider the linearization of the solution ordering induced by $\omega$, say $l(\omega)$. Then there exists a solution $s$ such that computing $Next(p, s, l(\omega))$, where $p$ is a CSP, is NP-hard.

### 5 Next on tree-shaped CSPs

We know that finding an optimal solution becomes easy if we restrict the constraint graph of the problem to have the shape of a tree. It is therefore natural to consider this class to see whether also the Next problem becomes easy under this condition. We will see that this is indeed so: if the CSP is tree-shaped, it can be easy to find the next solution.

In this section we focus on tree-shaped CSPs. However, the same results hold for bounded tree-width. For a tree-shaped CSP with variable set $X = \{x_1, \ldots, x_n\}$, let us consider the linearization $tlex(O)$, which is the same as $lex(O)$ defined in the previous section, with the restriction that the variable ordering $o$ respects the tree shape: each nodes comes before its children. For example, let us consider the tree-shaped CSP shown in Figure 1, and assume that $o = (x_1, x_2, x_3, x_4, x_5)$ and that in all domains $a \preceq_o b \preceq_o c$. The solutions of the CSP are then ordered by $tlex(O)$ as follows: $(a, b, a, b, b) \prec (a, b, a, c, b) \prec (b, a, b, a, b) \prec (b, a, b, c, a) \prec (b, a, b, c, b) \prec (b, b, b, b, b)$.

We will now describe an algorithm that, given as input a directionally arc consistent tree-shaped CSP $P$ and a solution $s$ for $P$, it either returns the consistent assignment following $s$ according to $tlex(O)$, or it detects that $s$ is the last consistent assignment in such an ordering. The algorithm works bottom-up in the tree, looking for new variable values that are consistent with the value assigned to their father (denoted by $f(i)$ in Algorithm 1) and successive to the ones assigned in $s$ in the domain orderings. As soon as it finds a variable for which such a value exists, it resets all the following variables (according to the variable ordering $o$) to their smallest compatible values w.r.t. the domain orderings (via procedure $Reset-succ$).

For example, if we run CSP-Next giving in input the CSP of Figure 1 and solution $s=(b,a,b,a,b)$, the algorithm first tries to find a value for $x_5$ consistent with $x_2 = a$ and
Algorithm 1: CSP-Next

Input: tree-shaped and DAC CSP P, orderings o, o₁, ..., oₙ, assignment s
Output: an assignment s′, or “no more solutions”

for i = n to 1 do
    Search D(xᵢ) for the next value w.r.t. oᵢ which is consistent with sₓᵢ, say v′;
    if v′ exists then
        sᵢ ← v′
        Reset-succ(s, i)
    return s
return “no more solutions”

Theorem 3 Consider a tree-shaped and DAC CSP P and the ordering tlex(O) defined above. If s is not the last solution in ordering tlex(O), the output of CSP-next(P, s) is the successor of s according to tlex(O); otherwise, the output of CSP-next(P, s) is “no more solutions”.

If |D| is the cardinality of the largest domain, it is easy to see that the worst case complexity of CSP-next is O(n|D|), since both looking for consistent assignments and resetting to the earliest consistent assignment takes O(|D|) times, and such operations are done O(n) times.

From Theorem 3 we can thus conclude that Next(P, s, tlex(O)) is polynomial, since it can be computed by applying DAC to P and then CSP-Next to P and s, both of which are polynomial-time algorithms.

Note also that the choice of the linearization is crucial for the complexity of the algorithm. Indeed, a different choice for l may turn Next(P, s, l) into an NP-hard problem, even on tree-shaped CSP, as proved in the following theorem.

Theorem 4 Computing Next(P, s, l), where P is a tree-shaped CSP, s is one of its solutions, and l is an arbitrary linearization, is NP-hard.
The proof is based on a reduction from the subset sum problem.

6 Next on weighted CSPs

In the context of a weighted CSP, finding the next solution means that, given a solution, we want to return the next assignment in lexicographical order with the same cost or, if there is no such assignment, the first assignment in lexicographical order with the next smallest cost. The following holds:

**Theorem 5** Computing Next(P, s, l), where P is a weighted CSP and s is one of its solutions, is NP-hard, for any linearization l.

*Proof.* We give a reduction from the subset sum problem, akin to the proof of Theorem 4. Given a set C of integers \{t_1, \ldots, t_n\} and an integer t, the subset sum problem consists in finding a subset C′ of elements of C whose sum equals t, or showing that no such subset exists. Given a subset sum problem \{t, t_1, \ldots, t_n\}, consider a weighted CSP P, with n + 1 binary variables \{x_0, x_1, \ldots, x_n\} and unary constraints c_i on the variable domains as follows: c_0(1) = t - 1/2, c_i(1) = t_i \forall i \in \{1, \ldots, n\}, c_i(0) = 0 \forall i \in \{0, \ldots, n\}. Consider the solution s = \{1, 0, \ldots, 0\} with cost \(t - 1/2\). For any linearization l, if \(s' = \text{Next}(P, s, l)\) has cost \(t\), we have found a feasible solution for our subset sum problem, otherwise we have proven that there is none.

Note that theorems 4 and 5, while very similar in proof, have quite a different implication. Indeed, for tree-shaped CSPs computing Next is NP-hard only for some choices of the linearization l, while for weighted CSPs computing Next is always NP-hard, irrespective of the linearization, because the “native” solution ordering is already sufficient for NP-hardness.

Being subset sum NP-hard only in the weak sense, given the above proof it is reasonable to consider whether the same holds for computing the next solution in a weighted CSP. The answer is negative in general. Indeed, since Max-SAT is a special case of weighted CSP, and at the same time it is the optimization version of the classical satisfiability problem, then we could use a pseudo-polynomial procedure that computes the next solution on a weighted CSP to compute the next solution on a satisfiability problem in polynomial time (because the maximum cost in Max-SAT is polynomial in the size of the original satisfiability problem), which is in contradiction with Theorem 1.

However, if we consider the lexicographical ordering, and we restrict to weighted CSP with unary constraints only, then the NP-hardness is indeed weak, since we can provide a pseudo-polynomial algorithm for it.

**Theorem 6** Given a weighted CSP P with unary constraints only, a solution a, and a linearization l induced by a lexicographic ordering, computing Next(P, a, l) is weakly NP-hard.

To show that finding the next solution is only weakly NP-hard, we give a pseudo-polynomial algorithm which is a simple generalization of the dynamic programming algorithm for deciding subset sum [13, 15]. Given a weighted CSP with unary constraints
only, and an assignment \( a = (a_1, \ldots, a_n) \) to its variables \( x_i \) with \( 1 \leq i \leq n \), with cost \( c(a) = \sum c_j(a_j) = s \), we define \( Q_a(j,t) \) as the lexicographically smallest assignment \( b = (b_1, \ldots, b_j) \) with cost \( t \), that involves only the first \( j \) variables, and such that \( b \succ_{lex} a \). If no such assignment exists, then \( Q_a(j,t) = \text{nil} \). A simple recursion can be used to compute \( Q_a(j,t) \), for any \( j \) from 1 to \( n \), and any cost level \( t \) from 0 to \( s \). In particular, given \( v_j = c_j(b_j) \), we can initialize \( Q_a(1,t) \) as \( Q_a(1,t) = \text{lexmin}_{a_i \in D_i} \{ (b_1) : v_1 = t \land b_1 \succ_{lex} a_1 \} \) and then compute recursively \( Q_a(j,t) = \text{lexmin}_{a_i \in D_i} \{ \{ Q_a(j-1,t-v_j), b_j \} : (b_1, \ldots, b_j) \succ_{lex} (a_1, \ldots, a_j) \} \cup \{ (a_1, \ldots, a_{j-1}, b_j) : v_j = c_j(a_j) \land b_j \succ_{lex} a_j \} \).

To compute the next solution given an assignment \( a \) with cost \( s \), we just need to check whether \( Q_a(n,s) \) is not \( \text{nil} \). If it is, \( Q_a(n,s) \) contains the lexicographically next assignment with cost \( s \). Otherwise, we use a very similar dynamic program to compute the lexicographically smallest assignment with the next smallest cost. The running time of both dynamic programs is \( O(n \cdot s \cdot |D|) \), where \( |D| \) is the size of largest domain.

Thus we can compute the next solution in pseudo-polynomial time.

**Example:** Consider a weighted CSP with five Boolean variables \( x_1 \) to \( x_5 \) and unary costs \( c_i(0) = 0 \) and \( c_i(1) = i \). Suppose we are given the solution \( a = (0,1,1,0,0) \) of cost \( s = 5 \) and we want to compute the lexicographically next solution. The dynamic programming algorithm will build from the bottom-left corner the following table:

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b_1 )</th>
<th>( b_2 )</th>
<th>( b_3 )</th>
<th>( b_4 )</th>
<th>( b_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 3 )</td>
<td>( nil )</td>
<td>( nil )</td>
<td>(1,0,0,1)</td>
<td>(1,0,0,1)</td>
<td>(1,0,0,0)</td>
</tr>
<tr>
<td>( 4 )</td>
<td>( nil )</td>
<td>(1,0,1)</td>
<td>(1,0,1,0)</td>
<td>(1,0,1,0)</td>
<td>(1,0,0,0)</td>
</tr>
<tr>
<td>( 3 )</td>
<td>(1,1)</td>
<td>(1,1,0)</td>
<td>(1,1,0,0)</td>
<td>(1,1,0,0)</td>
<td>(1,0,0,0)</td>
</tr>
<tr>
<td>( 2 )</td>
<td>( nil )</td>
<td>( nil )</td>
<td>( nil )</td>
<td>( nil )</td>
<td>( nil )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>(1,0)</td>
<td>(1,0,0)</td>
<td>(1,0,0,0)</td>
<td>(1,0,0,0)</td>
<td>(1,0,0,0)</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( nil )</td>
<td>( nil )</td>
<td>( nil )</td>
<td>( nil )</td>
<td>( nil )</td>
</tr>
</tbody>
</table>

For example, \( Q_a(j = 3, t = 4) = (1,0,1) \). This means that \( (1,0,1) \) is the lexicographically smaller assignment with cost 4, that involves only the first 3 variables, and such that it follows \( (0,1,1) \) (that is, the restriction of \( a \) to the first 3 variables) lexicographically. Since \( Q_a(j = 5, t = 5) \) is not \( \text{nil} \), then the lexicographically next assignment with cost 5 is \( (1,0,0,1,0) \).

7 Next on tree-shaped fuzzy CSPs

Turning our attention to fuzzy CSPs, we will show that Next on tree-like fuzzy CSPs can be easy. Let \( P \) be a fuzzy tree-shaped CSP with variable set \( X = \{ x_1, \ldots, x_n \} \) and set of constraints \( C \), and let us consider a variable ordering \( o = \{ x_1, \ldots, x_n \} \) which respects the tree shape. Moreover, let \( o_i \) be a total order over the values in the domain of \( x_i \), for \( i = 1, \ldots, n \).

We will consider set \( T = \{ t = (x_i = v_i, x_j = v_j) \mid i < j, \exists c \in C, t \in c, \text{pref}_c(t) > 0 \} \), where \( \text{pref}_c(t) \) denotes the preference assigned to \( t \) by constraint \( c \) that is, the set of all pairs of variables assignments appearing in \( P \) with preference greater than 0. The
preferences assigned to tuples by the constraints in $C$ and the orderings $o, o_1, \ldots, o_n$ induce the following ordering $o_T$ over $T$: $(x_i = v, x_j = w) \prec_{o_T} (x_h = z, x_k = u)$ if

- the preference associated to tuple $(x_i = v, x_j = w)$ by its constraint is higher than the preference associated to tuple $(x_h = z, x_k = u)$ by its constraint, or
- they have the same preference, and the variable pair $(x_i, x_j)$ lexicographically precedes the variable pair $(x_h, x_k)$ according to $o$, or
- they have the same preference, $i = h, j = k$ and the value pair $(v, w)$ lexicographically precedes the value pair $(z, u)$ according to domain orderings $o_i$ and $o_j$.

We will now use this strict total order over the set of tuples of $P$ to define a strict total order over the set of solutions of $P$. Given two complete assignments to $X$, say $s$ and $s'$, let $t_s = \min_{o_T} \{ t \text{ tuple of } s \text{ with preference } \text{pref}(s) \}$ and $t'_s = \min_{o_T} \{ t \text{ tuple of } s' \text{ with preference } \text{pref}(s') \}$. Let $\text{opt}(P)$ denote the optimal preference of a fuzzy CSP $P$. We write $s \prec_f s'$ (that is, $s$ precedes $s'$ in ordering $\prec_f$), if

- $\text{pref}(s) > \text{pref}(s')$, or
- $\text{pref}(s) = \text{pref}(s') = \text{opt}(P)$ and $s$ precedes $s'$ in the lexicographic order induced by $o$ and the domain orderings $o_1, \ldots, o_n$, or
- $\text{pref}(s) = \text{pref}(s') < \text{opt}(P)$ and $t_s \prec_{o_T} t'_s$,
- $\text{pref}(s) = \text{pref}(s') < \text{opt}$, $t_s = t'_s$ and $s$ precedes $s'$ in the lexicographic order induced by $o$ and the domain orderings $o_1, \ldots, o_n$.

It is possible to show that $\prec_f$ is a linearization of the solution ordering.

![Fig. 2. A tree-shaped DAC fuzzy CSP.](image)

For example, let us consider the tree-shaped DAC fuzzy CSP shown in Figure 2. Assume that $o = (x_1, x_2, x_3, x_4)$ and that $a \prec_o b$ for $i = 1, \ldots, 4$. Then, if we consider solutions $s = (b, a, a, b)$ and $s' = (a, b, b, b)$, we have that $s \prec_f s'$ since $\text{pref}(s) = \text{pref}(s') = 0.5 < 1 = \text{opt}(P)$, $t_s = (x_1 = b, x_2 = a, x_3 = b)$, and thus $t_s \prec_{o_T} t'_s$. If instead we consider solutions $s = (b, b, a, a)$ and $s' = (b, b, a, b)$, we have again that $s \prec_f s'$, since $\text{pref}(s) = \text{pref}(s') = 0.2 < 1 = \text{opt}(P)$, $t_s = (x_1 = b, x_2 = b) = t'_s$, and $s$ precedes $s'$ lexicographically.
As with CSPs, we provide a polynomial time algorithm that solves the Next problem for tree-shaped fuzzy CSPs. The main idea that we exploit is that, in a fuzzy CSP, a solution can have preference $p$ only if it includes a tuple that has preference $p$. In

```
Algorithm 2: FuzzyCSP-Next

Input: tree-shaped and DAC Fuzzy CSP $P$, orderings $o, o_1, \ldots, o_n, o_T$, assignment $s$ with preference $p$
Output: an assignment $s'$, or “no more solutions” if $p = \text{opt}(P)$

if $p = \text{opt}(P)$ then
    $P' \leftarrow \text{cut}(P, p)$
    if CSP-next($P', s$) $\neq$ “no more solutions” then
        return CSP-next($P', s$)
if $p \neq \text{opt}(P)$ then
    compute tuple $t_s$
    else let $t^*$ be the first tuple s.t. $\text{pref}(t^*) = \text{next}(p)$
    $p^* = \text{pref}(t^*)$
    $P' \leftarrow \text{cut}((\text{fix}(P, t^*)), p^*)$
    if CSP-next($P', s$) $\neq$ “no more solutions” then
        return CSP-next($P', s$)
    $\text{pref}(t) \leftarrow 0$, $\forall t \in T$ such that $\text{pref}(t) = p^*$ and $t \preceq_{o_T} t^*$
    $c\text{pref} \leftarrow p^*$
    for each tuple $t >_{o_T} t^*$ following order $o_T$ until $\text{pref}(t) > 0$ do
        if $\text{pref}(t) < c\text{pref}$ then
            reset all preferences, previously set to 0, to their original values
        if $\text{pref}(\text{Solve}(\text{cut}(\text{fix}(P, t^*)), \text{pref}(t))) = \text{pref}(t)$ then
            $c\text{pref} \leftarrow \text{pref}(t)$
            $\text{pref}(t) = 0$
        return “no more solutions”
```

Algorithm 2:

- procedure fix($P$, $t$) takes in input a fuzzy CSP $P$ and one of its tuples, $t = (x_i = v, x_j = w)$ and returns the fuzzy CSP obtained from $P$ by removing from the domains of variables $x_i$ and $x_j$ all values except $v$ and $w$;
- procedure cut($P$, $p$) takes in input a fuzzy CSP $P$ and a preference $p$ and returns the CSP corresponding to the $p$-cut of $P$ as defined in Section 2;
- procedure Solve($P$) takes in input a CSP $P$ and returns the first solution in a lexicographic order given the variable and the domain orderings.

Intuitively, when solution $s$ with preference $p$ is given in input, if $s$ is optimal, we look for the next solution in the CSP obtained from $P$ by performing a cut at level $p$ and running CSP-next. If no solution is returned, then $s$ must have been the last solution with optimal preference in the ordering and its successor must be sought for at lower preference levels.
If $s$ is not optimal, we consider its tuples and we identify the smallest tuple of $s$, say $t_s$, according to ordering $o_T$, that has preference $p$ in the corresponding constraint. We fix such a tuple, via $fix(P, t_s)$, and we cut the obtained fuzzy CSP at level $p$. We then look for the solution lexicographically following $s$ in such a CSP by calling CSP-next. If no such solution exists, $s$ must be the last solution with preference $p$ among those that get their preference from $t_s$.

The next solution may have preference $p$ or lower. However, if it does have preference $p$, such a preference must come from a tuple with preference $p$ which follows $t_s$ in the ordering $o_T$. In order to avoid finding solutions with preference equal to $p$ that come from tuples with preference $p$ preceding $t_s$ according to $o_T$, we set the preference of all such tuples to 0. If none of the tuples with preference $p$ following $t_s$ generate solutions with preference $p$, we move down one preference level, restoring all modified preference values to their original values. This search continues until a solution is found or all tuples with preference greater than 0 have been considered.

**Theorem 7** Given a tree-shaped DAC fuzzy CSP $P$ and a solution $s$, algorithm FuzzyCSP-Next computes the successor of $s$ according to $\prec_f$ if $s$ is not the last solution with preference greater than 0, and outputs “no more solutions” otherwise. The worst case time complexity of algorithm FuzzyCSP-Next is $O(|T||D|n)$, where $|T|$ is the number of tuples of $P$ and $|D|$ the cardinality of the largest domain.

The correctness of FuzzyCSP-Next follows directly from the description of the algorithm. For the complexity, we notice that the complexity of FuzzyCSP-Next is bounded by that of running $|T|$ times the CSP-next algorithm.

Given a tree-shaped fuzzy CSP $P$, one of its solutions $s$, and the solution ordering $\prec_f$, Next$(P, s, \prec_f)$ can be therefore computed in polynomial time: we just need to achieve DAC (which is polynomial) and then run algorithm FuzzyCSP-Next.

Again, it is not difficult to prove that the choice of the order is crucial for the complexity of the algorithm, and that Next$(P, s, l)$ is in general NP-hard even on tree-shaped fuzzy CSPs. Indeed, since tree-shaped fuzzy CSPs admit tree-shaped CSPs as a special case, the result is a direct consequence of Theorem 4.

### 8 Next on acyclic CP-nets

We now consider the complexity of the Next operation in acyclic CP-nets. It turns out that Next is easy on such CP-nets, if we consider a certain linearization of the solution ordering. We first define the concept of contextual lexicographical linearization of the solution ordering. Let us consider any ordering of the variables where, for any variable, its parents are preceding it in the ordering. Let us also consider an arbitrary total ordering of the elements in the variable domains. For sake of simplicity, let us consider Boolean domains. Given an acyclic CP-net with $n$ variables, we can associate a Boolean vector of length $n$ to each complete assignment, where element in position $i$ corresponds to variable $i$ (in the variable ordering), and it is a 0 if this variable has its most preferred value, given the values of the parents, and 1 otherwise. Therefore, for example, the optimal solution will correspond to a vector of $n$ zeros.
To compute such a vector from a complete assignment, we just need to read the variable values in the variable ordering, and for each variable we need to check if its value is the most preferred or not, considering the assignment of its parents. This is polynomial if the number of parents of all variables is bounded. Given a vector, it is also easy to compute the corresponding assignment.

Let us now consider a linearization of the ordering of the solutions where incomparability is linearized by a lexicographical ordering over the vectors associated to the assignments. We will call such a linearization a contextual lexicographical linearization. Note that there is at least one of such linearizations for every acyclic CP-net.

**Theorem 8** Computing Next\((N,s,l)\), where \(N\) is an acyclic CP-net, \(s\) is one of its solutions, and \(l\) is any contextual lexicographical linearization of its solution ordering, is in \(P\).

Given any solution \(s\) and its associated vector, as defined above, the vector of the next solution in \(l\) can be easily obtained by a standard Boolean vector increment operation. Therefore, given any solution \(s\), it is possible to obtain the next solution by 1) computing the vector associated to \(s\), 2) incrementing it, and 3) computing the solution associated to the new vector. Since each of these steps is polynomial, the overall process is polynomial. The same proof can be easily extended to non-Boolean domains. ☐

Figure 3 shows an acyclic CP-net, with features A, B, and C, and its solution ordering. It is assumed that the variables have each two values: for A we have \(a\) and \(\bar{a}\), and similarly for B and C. Also, the variable ordering is \(A \prec B \prec C\). Given solution \(abc\) (that is, \(A=a\), \(B=b\), \(C=c\)), the associated Boolean vector (as described above) is 000, since \(a\) is the most preferred value for \(A\), \(b\) is the most preferred value for \(B\) given \(A=a\), and \(c\) is the most preferred value for \(C\). Instead, the vector associated to solution \(\bar{a}\bar{b}c\) is 100, and the vector associated to \(\bar{a}\bar{b}\bar{c}\) is 111. Given vector 101, the associated solution is \(\bar{a}\bar{b}\bar{c}\). In Figure 3 it is possible to see the CP-net, the solution ordering, and the vector for each solution. Also, if we order the solutions according to a standard lexicographical order over their vectors, we get a linearization of the partial solution ordering.

![Fig. 3. An acyclic CP-net and its solution ordering.](image)
9 Next on constrained CP-nets

Some statements are better expressed via constraints, other via preferences. Moreover, some preferences are better modeled via soft CSPs, other via CP-nets. However, usually in a real-life problem we may have statements of all these kinds, thus requiring to use all the above considered formalisms in the same problem. It is therefore useful to consider problems where CP-nets and CSPs, or soft CSPs, coexist [4].

We thus consider here the notion of a constrained CP-net, which is just a CP-net plus some (soft) constraints [4]. Given a CP-net $N$ and a constraint problem $P$, we will write $(N,P)$ to denote the constrained CP-net given by $N$ and $P$. For sake of simplicity, in the following we will assume that the CP-net and the CSP involve the same variables. Nevertheless, our results hold also for the more general setting.

Given a constrained CP-net $(N,P)$, its solution ordering, written $\prec_{np}$, is that given by the (soft) constraints, where ties can be broken by the CP-net preferences. More precisely, solution $s$ dominates solution $s'$ (that is, $s \prec_{np} s'$) if $s$ dominates $s'$ according to the constraints in $P$, or $s$ and $s'$ are equally preferred according to the constraints in $P$, but $s$ dominates $s'$ according to the CP-net $N$.

We now consider the complexity of computing the next solution in a linearization of this ordering. The first results says that the problem is difficult if we take the lexicographical linearization (given $o$, which is an ordering over the variables) of $\prec_{np}$, denoted with $\text{lex}(o, \prec_{np})$.

**Theorem 9** Computing $\text{Next}((N,P),s,\text{lex}(o, \prec_{np}))$, where $(N,P)$ is a constrained CP-net and $s$ is one of its solutions, is NP-hard.

The statement can be proven by reducing Next on a CSP to Next on a constrained CP-net. The same proof applies also to constrained CP-nets where the CP-net is acyclic.

Next becomes easy if we consider acyclic CP-nets, tree-shaped CSPs, and we add a compatibility condition between the acyclic CP-net and the constraints. This compatibility condition is related to the topology of the CP-net dependency graph and of the constraint graph.

Consider two variables in an acyclic CP-net, say $x$ and $y$. We say that $x$ depends on $y$ if there is a dependency path from $y$ to $x$ in the acyclic DAG of the CP-net. Given an acyclic CP-net $N$ and a tree-shaped CSP $P$, we say that $N$ and $P$ are compatible if there exists a variable of the CSP, say $r$, such that: for any two variables $x$ and $y$ such that $x$ is the father of $y$ in the $r$-rooted tree, we have that $x$ does not depend on $y$ in the CP-net. Informally, this means that it is possible to take a tree of the constraints where the top-down father-child links, together with the CP-net dependency structure, do not create cycles. If the compatibility holds for any root taken from a set $S$, then we will write that $N$ and $P$ are $S$-compatible.

Figure 4 shows an example of a CP-net DAG and two trees, of which the one in Fig. 4 (b) is compatible with the CP-net: if we choose A as the root, the father-child relationship is not contradicted by the CP-net dependencies. Instead, the tree in Fig. 4 (c) is incompatible with the CP-net: whatever root is chosen, some tree links are contradicted by the CP-net dependencies.
Theorem 10 Consider an acyclic CP-net $N$ and a tree-shaped CSP $P$, and assume that $N$ and $P$ are $S$-compatible, where $S$ is a subset of the variables of $P$. Taken a solution $s$ for $(N, P)$, and a variable ordering $o$ which respects the tree shape of $P$ with root an element of $S$, we have that $\text{Next}((N, P), s, \text{lex}(o, \prec_{np}))$ is in $P$.

To compute $\text{Next}((N, P), s, \text{lex}(o, \prec_{np}))$, we use algorithm CSP-Next, except that we dynamically order each variable domains according the CP-net: for any variable, we order its domain according to the row of its CP table associated to the fixed assignment to the parent variables.

Under these same conditions, Next remains easy even if we consider CP-nets constrained by fuzzy CSPs rather than hard CSPs. We just need to adapt in a similar way algorithm FuzzyCSP-Next.

Acknowledgments

Research partially supported by the Italian MIUR PRIN project 20089M932N: “Innovative and multi-disciplinary approaches for constraint and preference reasoning”. We thank and anonymous referee for his precious comments that greatly enhanced the quality (and correctness) of Section 6.

References

Multi-agent soft constraint aggregation: a sequential approach

Giorgio Dalla Pozza, Francesca Rossi, K. Brent Venable

University of Padova, Italy.
Email: gdallapo@studenti.math.unipd.it, frossi@math.unipd.it, kvenable@math.unipd.it

Abstract. We consider a scenario where several agents express their preferences over a common set of variable assignments, by means of a soft constraint problem for each agent, and we propose a procedure to compute a variable assignment which satisfies the agents’ preferences best. Such a procedure considers one variable at a time and, at each step, asks all agents to express its preferences over the domain of that variable. Based on such preferences, a voting rule is used to decide on which value is the best for that variable. At the end, the values chosen constitute the returned variable assignment.

We study several properties of this procedure and we show that the use of soft constraints allows for a great flexibility on the preferences of the agents, compared to similar work in this setting where agents model their preferences via CP-nets, where several restrictions on the agents’ preferences need to be imposed to obtain similar properties.

1 Introduction

In many real-life situations, we need to express our preferences over a set of objects, in order to select the most preferred one. This may be useful, for example, when activating a search over the web, or when looking for a product in a catalogue, or when trying to decide on the best resource allocation or timetabling.

Expressing our preferences over a set of objects means giving an ordering to this set, where more preferred objects appear earlier in the ordering. This ordering need not be total, that is, we may have incomparable objects. Indifference may be present as well, since some objects may be equally preferable to us.

When the set of objects is small, we may just present them ordered to express our preferences over them. However, when the set of objects is very large, this is unfeasible. Unfortunately, this is usually the case in real-life situation, since often objects have a combinatorial structure, that is, each object can be seen as the combination of certain features, where each feature has a set of possible instances. Consider for example a PC: usually it is not seen as a single item, but as a combination of a features, such as its processor, its hard disk, its RAM, and its monitor. Each of these features has some possible instances, and a PC is the combination of such feature instances.

Usually the instances for each feature are not many, but because of the combination the set of objects can still be very large. Consider an example, where we need to choose a menu for a meal, composed of a first course, a main course, a dessert, and a wine. Assuming to have 5 choices for each of these, we should choose among $5^4 = 625$
different meals. Fortunately, when the set of objects has a combinatorial structure, we may describe our preference in a compact way, using one of the several formalisms available in the literature, such as soft constraints [7] and CP-nets [2]. The aim of such structures is to allow one to express in time and space polynomial in the number of features an ordering over the set of all objects, which may be exponential in such a number.

A structure such as a CP-net or a soft constraint problem automatically induces a preference ordering over the set of objects. For such formalisms, the literature tells us how expensive it is to find the most preferred object, or to test if an object is the most preferred, or also to test whether one object is more preferred to another one. For example, for both types of structures, it is NP-hard to find the most preferred object, although it becomes linear in the number of features when some topological restrictions hold.

Often, we cannot take decisions by ourselves, but we need to compromise our preferences with those of other people. To come back to the dinner example above, we may think of a set of friends who want to organize a dinner and need to agree on a common menu. This is the kind of scenarios we consider in this paper. More precisely, we consider a set of agents, each expressing its preferences over a common set of objects with a combinatorial structure. The goal is now to combine such preferences and to find the object which is the most preferred according to this combination. Of course, the resulting object will depend on the chosen rule to combine the preferences of the agents.

Voting theory [1] is a very wide research area, between economic theory and operations research, that considers exactly these scenarios: in an election, voters (that we would call agents) vote by expressing their preferences over a set of candidates (that we would call objects), and a voting rule decides who the winner candidate is. Thus it is natural to look in this area for possible approaches to multi-agent preference aggregation.

Voting theory provides many rules to aggregate preferences. If there are only two objects, the best rule, according to many criteria, is majority voting. When there are more than two objects, there are many voting rules one could use (Plurality, Borda, STV, approval, etc.), each with its pros and cons. Each one of such rules takes in input (a part of) the preference orderings of the agents and gives as output the "winner" object, that is, the object that is considered to be the best according to the rule.

The most naive way to use voting rules in our context consists of choosing a voting rule and giving to it what it needs to know about the preference orderings of the agents, then running the voting rule and see what result comes out. This is however not feasible in general. In fact, if the chosen voting rule needs to know a large part of the preference ordering from the agents, it may take exponential time only to give the input to the rule.

A valid alternative is to use the voting rule several times, on each feature of the object set. That is, the rule asks the agents to provide their preferences on each feature at a time, and at each step a winner instance for a certain feature will be returned. At the end, all the winner instances will constitute the winning object.

This approach is certainly more attractive computationally, since usually the number of instances of each feature is small. Also, in special cases, it may return useful results.
For example, if the preferences of the agents on the various features are independent of each other, then the winning object will be optimal for all agents. Unfortunately, this is a very rare case, since usually there are dependencies among the various features. What can be done in this situation is to study when, even in presence of dependencies among features, voting can be performed on each single feature at a time rather than on complete objects. This study has been done already for CP-nets [6], showing that a sequential single-feature voting protocol can find a winner object in polynomial time, and have several other desirable properties, when the CP-nets satisfy certain conditions on their dependencies. In particular, the CP-nets must be acyclic, and their dependency graphs must all be compatible with a given graph ordered according to the feature ordering in the voting procedure. We are now interested to undergo the same study for soft constraints.

More precisely, in our setting there is a common set of variables and domains, and each agent expresses its preferences over all variable assignments via a soft constraint problem. Given a variable ordering, say \( v_1, \ldots, v_n \), and a sequence of \( n \) voting rules \( r_1, \ldots, r_n \), the sequential voting procedure is simply a sequence of as many steps as the number of variables, where at each step \( i \) agents are required to report their preferences over the domain of variable \( v_i \). To do this, each agent needs to compute the influence of the whole soft constraint problem over \( v_i \). Voting rule \( r_i \) is then applied to such preferences over the domain of \( v_i \), and the result is a value \( d_i \) for \( v_i \). Step \( i \) ends by adding constraint \( v_i = d_i \) to the soft constraint problems of all agents. After all \( n \) steps have been executed, the collection of all computed values is the variable assignment chosen by the sequential procedure.

To make each step (and the whole procedure) polynomial in the number of agents and variables, we have to assume that each \( r_i \) is polynomial to compute, and also that each agent can easily compute its preferences over a single variable. For example, we may assume that agents model their preferences via tree-shaped soft constraints.

Our results are similar to those obtained in [6], where agents express their preferences with CP-nets. However, the use of soft constraints allows us to avoid imposing many restrictions on the preferences of the agents. In fact, contrarily to CP-nets, soft constraints are not directional, and thus information can flow from one variable of a constraint to another one without a predefined ordering between them. This allows us to not tie the variable ordering used by the sequential procedure to the topology of the constraint graph of each agent. This makes the approach much more generally applicable. In fact, the tractability assumption over the constraint graphs is similar to the assumptions that CP-nets are acyclic. However, we do not need to impose that the constraint graphs are mutually compatible and with a graph structure based on the variable ordering.

A viable alternative to the sequential voting approach, that we don’t consider in this paper, would be to define voting rules whose input is not a preference ordering, but a compact structure representing it. In this way, the drawbacks outlined above of the one-step approach would disappear, since the rule would directly work on a small structure rather than on a large ordered set.

Another alternative aggregation strategy is that of a direct aggregation of all the soft constraint networks at once. This path has been in followed in [8] and in [5] among
others. With respect to this approach, the one presented in this paper has the advantage of having less restrictive assumptions that guarantee tractability and that it allows for the use of several voting rules instead of a single preference combination criterion.

2 Background

2.1 Soft Constraints

A soft constraint [7] is a constraint [4] where each instantiation of its variables has an associated value from a (totally or partially ordered) set which has the structure of a c-semiring. A c-semiring is defined by \( \langle A, +, \times, 0, 1 \rangle \) where \( A \) is this set of values, + is a commutative, associative, and idempotent operator, \( \times \) is used to combine preference values and is associative, commutative, and distributes over +, 0 is the worst element, and 1 is the best element. The c-semiring \( S \) induces a partial or total order \( \leq_s \) over preference values, where \( a \leq_s b \) iff \( a + b = b \). A Soft Constraint Satisfaction Problem (SCSP) is a tuple \( \langle V, D, C, A \rangle \) where \( V \) is a set of variables, \( D \) is the domain of the variables and \( C \) is a set of constraints over \( V \) associating values from c-semiring \( A \).

A classical CSP [4, 9] is just a soft CSP where the chosen c-semiring is \( S_{CSP} = \langle \{false, true\}, \lor, \land, false, true \rangle \). Fuzzy CSPs [7] are instead modeled with \( S_{FCSP} = \langle [0, 1], max, min, 0, 1 \rangle \). That is, we want to maximize the minimum preference. For weighted CSPs, the c-semiring is \( S_{WCSP} = \langle R^+, min, +, +\infty, 0 \rangle \): preferences are interpreted as costs from 0 to +\( \infty \), and we want to minimize the sum of costs.

As an example, consider the following fuzzy CSP where \( V = \{X, Y\} \), \( D = \{a, b\} \) and \( C = \{c_Y, c_{XY}\} \). Soft constraint \( c_Y \) is defined over the values of \( Y \) and associates preference 0.4 to \( a \) and to 0.7 to \( b \). Constraint \( c_{xy} \), instead, is defined over \( X \) and \( Y \) and associates 0.9, 0.8, 0.7, 0.6 to, respectively, tuples \((X = a, Y = a)\), \((X = a, Y = b)\), \((X = b, Y = a)\) and \((X = b, Y = b)\).

Two main operations have been defined on soft constraints: combination, denoted with \( \otimes \), and projection, denoted with \( \downarrow \). In words, combining two constraints means building a new constraint involving all the variables of the original ones, and which associates to each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate subtuples. Considering the example of fuzzy CSP above, we have that \( c_Y \otimes c_{XY} \) is a constraint on \( X \) and \( Y \) associating 0.4, 0.7, 0.4 and 0.6 to, respectively, tuples \((X = a, Y = a)\), \((X = a, Y = b)\), \((X = b, Y = a)\) and \((X = b, Y = b)\).

Projecting a constraint on a subset of variables means eliminating the others by associating to each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. For example, constraint \( c_Y \otimes c_{XY} \downarrow_X \) is a constraint defined only over \( X \), i.e. the variable on which we are projecting, and associates 0.7 to \( a \) and 0.6 to \( b \), since 0.7 is the highest preference of any tuple of \( c_Y \otimes c_{XY} \) where \( X = a \) and 0.6 is the highest preference of any tuple of \( c_Y \otimes c_{XY} \) where \( X = b \).

To obtain the solution of an SCSP, we just combine all constraints. Thus, solving an SCSP means inducing an ordering over the set of complete assignments. In the case of fuzzy and weighted CSPs, such and ordering is total with ties. In the example above,
the induced ordering has \((X = a, Y = b)\) at the top with a preference of 0.7, \((X = b, Y = b)\) just below with 0.6 and \((X = b, Y = a)\) and \((X = b, Y = b)\) tied at the bottom with 0.4. An optimal solution of an SCSP is then a complete assignment with an undominated preference. The optimal solution of the example is \((X = a, Y = b)\). Finding an optimal solution in a set of soft constraints is an NP-hard problem.

Constraint propagation in SCSPs may be very helpful in improving the behavior of the search. For some classes of constraints, constraint propagation is enough to solve the problem [3]. This is the case for tree-shaped fuzzy CSPs, where directional arc-consistency (DAC), applied bottom-up on the tree shape of the problem, is enough to make the search for an optimal solution backtrack-free. DAC is also enough to compute the preferences over the values of the root variable, in dependence of the rest of the problem. That is, DAC is equivalent to combining all constraints and projecting over the root variable.

For the purpose of this paper we note that, if we project the solution of a SCSP over a single variable, we obtain a total order with ties over the values of that variable, where each value is associated to the preference of the best solution of the SCSP having such variable instantiated to such a value. Given an SCSP \(P\) and one of its variables \(v\), we will denote as \(\text{top}(v, P)\) the set of values of \(v\) that are assigned the highest preference in such an ordering. In our running example, if we consider \(c_Y \otimes c_{XY} \dl_X\), the induced ordering over the values of the domain of \(X\) is \(a > b\).

### 2.2 Voting Rules

A voting rule allows a set of voters to choose one among a set of candidates. Voters need to submit their vote, that is, their preference ordering over the set of candidates, and the voting rule aggregates such votes to yield a final result, usually called the winner.

In the classical setting [1], given a set of candidates \(C\), a profile is a sequence of orderings over the set of candidates, one for each voter. Usually, such orderings are total orders, however several extensions have been studied such as when the orderings are partial orders or total orders with ties. Given a profile, a voting rule, also known as a social choice function, maps it onto a single winning candidate.

In this paper, we will often use a terminology which is more familiar to multi-agent settings, and we will therefore sometimes call "agents" the voters, "solutions" the candidates, and "decision" or "best solution" the winning candidate.

Some examples of widely used voting rules are:

- Plurality: each voter states who the preferred candidate is, and the candidate who is preferred by the largest number of voters wins;
- Majority: like plurality, but with only two candidates;
- Borda: given \(m\) candidates, each voter gives a ranking of all candidates and the \(i^{th}\) ranked candidate scores \(m - i\); the candidate with the greatest sum of scores wins;
- Approval: each voter approves between 1 and \(m - 1\) candidates on \(m\) total candidates; the candidate with most votes of approval wins;
- Cup (or sequential majority voting): a sequence of pairwise majority elections is performed, according to a tree which sets the agenda for the pairwise elections; at each step, a candidate faces another candidate; the loser is eliminated and the
winner goes on to the next step; the process continues until only one candidate remains.

- Copeland: the winner is the candidate that wins the most pairwise competitions against all the other candidates.

The research on voting theory has mainly been concerned with the definition of properties of voting systems that are desirable. These properties are desirable also in automated contexts, so we will recall a few of them, since we will later be interested in studying their presence (or absence) in the preference aggregation system we propose:

- Condorcet-consistency: A voting rule is Condorcet-consistent if, when a candidate is preferred by a majority of the agents to every other candidate (namely, a Condorcet winner) exists, that candidate is always elected; Condorcet winners are unique and may not exist.

- Anonymity: A voting rule is anonymous when the results of an election are the same even under a permutation on the voters’ set. This means that the result is independent of who are the voters, or in which order they are considered, but it depends only on their set.

- Neutrality: A voting rule is neutral when, for any permutation on the set of candidates, the permuted results of an election are identical to the results of the same election where the permutation on the candidates’ set occurs before applying the voting rule. This is the analogous of anonymity for the candidates: the result does not depend on the names of the candidates.

- Monotonicity: A voting rule is monotonic if, when a candidate wins, and a voter improves his vote in favor of this candidate, then the same candidate still wins.

- Strong Monotonicity: A voting rule is strongly monotonic if, given a subset of the candidates’ set and a modified restricted profile over this subset (where we increase the preferences over the candidates of the subset), the winner of the election over the subset belongs to the subset’s candidates or to the winner of the election over the full set of candidates.

- Consistency: A voting rule is consistent if, when considering preferences of 2 disjoint sets of voters - who decide over the same issues and elect the same winner - the result obtained by a vote of the joint set of voters is the same as the ones obtained by the disjoint set of voters.

- Participation: A voting rule is participative if, given any profile, and given a new vote over a set of issues by a new voter, the result obtained from the new profile is equally or more preferred by the new voter. This means that a voter has an incentive to participate in the voting process, since by doing so he can get a better result.

- Efficiency (or unanimity): A voting rule is efficient if, given a winner over an election, there’s no candidate who is preferred to the winner by all voters. This means that, when all voters agree that a certain candidate should win, he will be declared the winner.

These properties are present in some of the voting rules defined above. More precisely, all such rules are anonymous and neutral. Moreover, all but Cup are efficient, only Cup and Copeland are Condorcet consistent, and all but Cup and Copeland are consistent and participative.
3 Soft profiles

As noted above, a soft constraint problem is a way to express our preferences over a set of objects with a combinatorial structure. The set of objects over which preferences are expressed is the set of all assignments of all variables of the problem. If we have \( n \) variables with \( m \) values in each variable domain, this set contains \( m^n \) objects. In the fuzzy example, each variable assignment is given a preference between 0 and 1, and assignments with a higher preference value are more preferred.

Assume we have a set of agents, each one expressing his preferences over a common set of objects via a soft constraint problem. Since the objects are common to all agents, this means that all the soft constraint problems have the same set of variables and the same variable domains. However, they may have different constraints, as well as different preferences over the variable domains. We will call a soft profile the preference of a set of \( m \) agents, identified by a triple \((V, D, P)\): a set of variables \( V \), a sequence \( D \) of \( |V| \) domains, and a sequence \( P \) of \( m \) soft constraint problems over variables in \( V \) with domains in \( D \). A fuzzy profile is the same as a soft profile, except that the preferences of each agent are modelled via a fuzzy constraint problem.

4 Sequential preference aggregation

The idea is to sequentially vote on each variable via a voting rule. We do not restrict ourselves to use always the same voting rule for all variables, so we will have a sequence of as many voting rules as the variables in \( V \).

Given a profile \((V, D, P)\), assume \( |V| = n \), and consider an ordering of such variables \( O = \langle v_1, \ldots, v_n \rangle \) and a sequence of voting rules \( R = \langle r_1, \ldots, r_n \rangle \). The sequential voting procedure we propose is a sequence of \( n \) steps, where at each step \( i \):

1. We ask all agents to report their preference ordering over the domain of variable \( v_i \).
   
   If we have \( m \) agents, let us call such preference orderings \( \langle po_{i1}, \ldots, po_{im} \rangle \).

2. We apply voting rule \( r_i \) to this profile, returning a winning assignment for variable \( v_i \), say \( d_i \). If there are ties in the result, the first one following a lexicographical order will be taken assuming an order over the values has been fixed.

3. We add the constraint \( v_i = d_i \) to the preferences of each agent.

After all \( n \) steps have been executed, we collect the winning assignments in the tuple \( \langle d_1, \ldots, d_n \rangle \), which will be reported as the chosen assignment for the variables in \( V \).

We write \( Seq_{O,R}(V, D, P) = (d_1, \ldots, d_n) \).

This short description of the sequential voting procedure does not say what it means for an agent to report their preference ordering over the domain of variable \( v_i \). In general, since we do not make any assumption on the voting rules \( r_i \), the agent needs to provide the rule with a preference ordering over the whole domain of \( v_i \). Since this variable can be connected to other parts of the agent’s soft constraint problem, in order to report the correct preferences over the domain of \( v_i \), the agent needs to consider the influence of the rest of the problem over \( v_i \). This means that, in general, the agent needs to compute the projection over \( v_i \) of its whole soft constraint problem. This is a soft unary constraint over \( v_i \), which is the desired preference ordering, to be given to the
voting rule. Once this is provided by each agent, the rule is applied to the profile and the winner is computed. We note that all the rules we consider in this paper are polynomial to compute.

On the other hand the task of solving the SCSP is in general NP-hard, so it may require exponential time to accomplish it, unless the class of constraint problems used by the agent is tractable. This is for example the case of tree-like shaped soft constraint problems, which are polynomial to solve. Thus, if agents decide to express their preferences via tree-shaped soft constraint problems, at each step of the sequential voting procedure it is polynomial to provide the input to the voting rule relevant for that step. While being tractable, tree-shaped SCSPs can represent a wide variety of interconnections between preferences over issues.

5 Properties of the sequential voting procedure

We will now prove several properties of this sequential preference aggregation procedure.

5.1 Order independence

The first thing we may wonder is how much the choice of the ordering $O$ influences the result of our procedure. Of course, in general, different orderings will lead to different results. However, some ordering will produce the same result. Intuitively, these are orderings that differ just for the relative position of variables that are independent of each other, according to all agents.

More precisely, given an ordering $O$ over $V$, this can be used to direct all edges in the constraint graph of each agent, by setting the direction from $v_1$ to $v_2$ if $v_1$ is earlier than $v_2$ in $O$. Let us call $G_i(O)$ this directed graph relative to agent $i$. If two variables $v$ and $v'$ are not connected in the transitive closure of this graph, then they are said to be independent.

**Theorem 1.** Consider any two orderings $O_1$ and $O_2$ over $V$, that differ for the relative position of two variables $v_1$ and $v_2$, and assume these two variables are independent in $G_i(O)$, for all $i$. Then $Seq_{O_1,R}(V, D, P) = Seq_{O_2,R}(V, D, P)$.

To prove this statement, we just need to prove that, if two variables $v_i$ and $v_j$ are independent for an agent, its preference ordering over $v_i$, given that $v_j$ has already been instantiated, is the same as the one when $v_i$ has still its whole domain. This is certainly so, since the two variables are independent in the agents’ constraint graph.

Given a soft profile, we may wonder if there exist a *best* variable ordering to use for the sequential voting procedure, where by *best* we mean that the resulting variable assignment reflects the preferences of the agents as much as possible. If $v_j$ depends on $v_i$ for all agents, then we should choose an ordering where $v_i$ comes before $v_j$. If we order such two variables in the opposite way, agents could regret what they reported as their preferences over $v_j$ when $v_i$ will be instantiated.
5.2 Condorcet consistency

It is now natural to ask ourselves if the result returned by the sequential voting procedure has some relation with what is considered to be most preferred by the agents. In this respect, it is natural to consider the notion of Condorcet winner, which is classical in voting theory.

As defined above, a Condorcet winner (CW) is a candidate which is strictly or equally preferred to any other candidate by a majority of agents. Given a totally ordered profile, as in classical voting theory, there can be zero or exactly one Condorcet winner. In our context, however, since we may have ties in the preference orderings of the agents, there could be more than one Condorcet winner, since several variable instantiations could be considered optimal for a majority of agents. For any voting rule, it is very desirable that it is Condorcet-consistent, that is, that it returns a Condorcet winner if there is one.

First, we define the notion of sequential Condorcet winner (SCW). Given an SCSP $Q$, we will denote as $Q_{v_1=d_1,\ldots,v_h=d_h}$ the problem obtained from $Q$ by fixing variables $v_1,\ldots,v_h$ to the corresponding values. Let $P_i$ denote the fuzzy constraint problem of agent $i$. Given a soft profile $(V,D,P)$ with $m$ agents and $n$ variables, and an ordering $O$ over $V$, $(d_1,\ldots,d_m)$ is a SCW iff, for all $j = 1,\ldots,n$, $\{i|d_j \in top(v_j, P_i|v_1=d_1,\ldots,v_{j-1}=d_{j-1})\} > m/2$. In words, a sequential Condorcet winner is the combination of local Condorcet winners.

If all the local rules are Condorcet consistent, the sequential voting procedure returns a SCW by definition. However, to conclude that Seq is Condorcet consistent, we need to prove that $SCW = CW$. The following results shows that a CW is always an SCW, but unfortunately the opposite does not hold.

**Theorem 2.** Given a soft profile $(V,D,P)$ and an ordering $O$ over $V$, if $d$ is a CW for $(V,D,P)$, then it is a SCW for $(V,D,P)$. Thus, if Seq is Condorcet consistent, all local voting rules are so.

In fact, if $d = (d_1,\ldots,d_m)$ is a CW, then a majority of voters prefers it to all other candidates. Thus, at each step $i$ of the sequential voting procedure, the same majority prefers $d_i$ to all other values in the domain of $v_i$ given the values already chosen for the previous variables.

The opposite does not hold in general, even if all voting rules are Condorcet consistent.

**Theorem 3.** If all local voting rules are Condorcet consistent, the sequential voting procedure may be not Condorcet consistent.

In fact, consider a fuzzy profile $(V,D,P)$ where: $V = \{X,Y\}$, $D_X = D_Y = \{a,b\}$ and there are 5 agents. The fuzzy SCSPs of all agents have a single constraint over $\{X,Y\}$. For two agents we have: $def(X = a,Y = b) = 0.9$, $def(X = b,Y = b) = 0.8$, $def(X = a,Y = a) = 0.7$, $def(X = b,Y = a) = 0.6$; for one agent: $def(X = a,Y = a) = 0.9$, $def(X = a,Y = b) = 0.5$, $def(X = b,Y = a) = 0.4$, $def(X = b,Y = b) = 0.6$; for the other two agents: $def(X = b,Y = a) = 0.9$, $def(X = b,Y = b) = 0.8$, $def(X = a,Y = a) = 0.7$, $def(X = a,Y = b)}$
b) = 0.6. When each agent solves the problem and projects on variable X, for the
first two agents we have $\text{pref}_X(a) = 0.9$ and $\text{pref}_X(b) = 0.8$; for the third agent $\text{pref}_X(a) = 0.9$ and $\text{pref}_X(b) = 0.7$; and for the last two agents $\text{pref}_X(a) = 0.7$ and $\text{pref}_X(b) = 0.9$. Thus, 3 over 5 agents agree that $X = a$ is optimal. Since the voting rule $r_X$ is Condorcet-consistent, this value will be chosen for $X$. Given $X = a$, the preferences of the agents for $Y$ are: for the first two agents $\text{pref}_Y(a) = 0.7$ and $\text{pref}_Y(b) = 0.9$; for the third agent $\text{pref}_Y(a) = 0.9$ and $\text{pref}_Y(b) = 0.7$; and for the last two agents $\text{pref}_Y(a) = 0.7$ and $\text{pref}_Y(b) = 0.6$. Thus $Y = a$ will be chosen, since $r_Y$ is Condorcet consistent, and $(X = a, Y = a)$ will be the SCW. However, $(X = a, Y = a)$ is not a CW, while $(X = b, Y = b)$ is. This means that there could be results of the sequential voting procedure that are not CWs. To make sure that the procedure is Condorcet consistent, we need to impose some restrictions on the profile, similarly to what is done in [6].

5.3 Anonymity

It is also important to make sure that a preference aggregation system does not depend on the names or the order of the agents. This corresponds to saying that the rule is anonymous.

In our setting, a permutation of voter set corresponds, basically, to a permutation of the soft constraint problems. It is easy to see that if the sequential voting rule respects anonymity, then also all the local voting rules do so, and, vice versa, if all the local voting rules are anonymous, so is the resulting sequential rule.

5.4 Neutrality

Neutrality is a property that requires for a rule to be insensitive to permutations of the candidates. This means that the result does not depend on the names of the candidates, but only on their position in the preference orderings. It is thus very important that a voting system is neutral.

We note that the candidates of the local voting rules are the values in the variable domains, while the candidates of the sequential voting rule are the complete assignments to all variables.

While a permutation of the values in the domains always corresponds to a permutation of the variable assignments, not all of the permutations of variable assignments can be obtained via permutations of domain values. For example, if we have assignments: $s_1 = (X = a, Y = b)$, $s_2 = (X = a, Y = b)$, $s_3 = (X = b, Y = a)$, and $s_4 = (X = b, Y = b)$, the permutation that swaps $s_1$ and $s_2$, leaving $s_3$ and $s_4$ fixed, cannot be modelled by a permutation of domain values.

From this observation it derives that neutrality of the local voting rules does not imply neutrality of the sequential voting rule, while the neutrality of the sequential voting rule implies the neutrality of each local voting rule.
5.5 Consistency

As defined above, a voting rule $r$ is consistent if, when considering two profiles $P_1$ and $P_2$ with disjoint sets of voters, who vote over the same candidates, such that $r(P_1) = r(P_2)$, we have $r(P_1 \cup P_2) = r(P_1)$.

**Theorem 4.** All the local voting rules in $R$ are consistent if and only if $Seq_{O,R}$ is consistent.

In fact, if all the local rules are consistent, at every step $i$ of the sequential procedure, applied to profile $P_1 \cup P_2$, the result for variable $v_i$ is the same as the result in profile $P_1$ (and also in profile $P_2$), so the overall result $(d_1, \ldots, d_n)$ will be the same as the result obtained by the sequential procedure in profile $P_1$ and in profile $P_2$. On the other hand, if one of the local rules, say $r_i$, is not consistent, then the result of the sequential procedure cannot be the same.

5.6 Participation

**Theorem 5.** If the sequential voting procedure is participative, then each local voting rule is so.

Let us assume that there is a local rule, say $r_i$ that is not participative. This means that there is a profile over the values of variable $v_i$, $p_i$ and an agent $h$ with preference $p_{i,h}$ over the values of $v_i$, such that agent $h$ strictly prefers $r_i(p_i)$ to $r_i(p_i \cup p_{i,h})$.

Let us now consider the soft profile $(V, D, P)$, defined as follows:

- for each agent $j = 1, \ldots, m$, the preferences over the values of $v_i$ are as in $p_i$;
- all other unary constraints are the same for all agents and associate preference 1 to exactly one value per variable and 0 to all other variables. There are no other constraints.

Let us assume that the SCSP $P_h$ of agent $h$ is defined as follows:

- his preference over variable $v_i$ is $p_{i,h}$,
- all other unary constraints associate preference 1 to exactly one value per variable and 0 to all other variables. There are no other constraints.

Note that for each agent $j$, including $h$, his preference for any assignment corresponds with his preference on the value assigned to the first variable. Since we have that $Seq_{O,R}(V, D, P) \downarrow v_i = r_i(p_i)$ and $Seq_{O,R}(V, D, (P \cup P_h)) \downarrow v_i = r_i(p_i \cup p_{i,h})$, agent $h$ strictly prefers $Seq_{O,R}(V, D, P)$ to $Seq_{O,R}(V, D, (P \cup P_h))$. Therefore the sequential voting procedure is not participative.

On the other hand, it is possible that all local voting rules are participative, but the sequential voting procedure is not so.

**Theorem 6.** If all the local voting rules are participative, the sequential voting procedure may not be participative.
To see this, consider the profile where $V = \{x, y\}$, $D = \{(a, b, c), \{a, b\}\}$, and $P$ is a sequence of two fuzzy SCSPs which coincide and contain a unary constraint on $x$ (associating preference 1 to $a$, 0.8 to $b$, and 0.6 to $c$), a binary constraint on $x$ and $y$ (associating preference 1 to $(a, b)$, 0.9 to $(a, a)$, 0.8 to $(b, a)$, 0.7 to $(b, b)$, 0.6 to $(c, a)$, and 0.5 to $(c, b)$), and a unary constraint over $y$ (associating preference 1 to both $a$ and $b$). It is easy to see that this SCSPs are DAC. Assume also that variables are ordered $x \sim_O y$ and that $r_1$ is the scoring rule with score vector $\langle 3, 2, 0 \rangle$ and $r_2$ is the majority rule. In this profile, $Seq_{Q,R}(V, D, P) = (x = a, y = b)$. We now consider a third voter, with a fuzzy SCSP with a unary constraint on $x$ (associating preference 0.8 to $a$, 1 to $b$, and 0.9 to $c$), a binary constraint on $x$ and $y$ (associating preference 0.8 to $(a, b)$, 0.5 to $(a, a)$, 0.7 to $(b, a)$, 1 to $(b, b)$, 0.6 to $(c, a)$, and 0.5 to $(c, b)$), and a unary constraint over $y$ (associating preference 1 to both $a$ and $b$). In this new profile $P'$, $Seq_{Q,R}(V, D, P') = (x = b, y = a)$. However, the third voter prefers $(x = a, y = b)$ to $(x = b, y = a)$. Thus the third voter would be better off not participating to the sequential voting process.

5.7 Efficiency

**Theorem 7.** If the sequential voting procedure is efficient, then each local voting rule is so.

Let us assume that there is a local rule, say $r_i$ that is not efficient. This means that there is a profile over the values of variable $v_i$, $p_i$, such that $r_i(p_i) = d$ but all agents prefer another value $d'$.

Let us now consider the soft profile $(V, D, P)$, defined as follows:

- for each agent $j$ the preferences over the values of $v_i$ are as in $p_i$;
- all other unary constraints are the same for all agents and associate preference 1 to exactly one value per variable and 0 to all other variables. There are no other constraints.

We will have that $Seq_{Q,R}(V, D, P) = (d_1, d_2, \ldots, d_i, \ldots, d_n)$. For each agent $j$ his preference for $Seq_{Q,R}(V, D, P)$ corresponds with his local preference for $d$. Thus each agent will prefer $(d_1, d_2, \ldots, d_i, \ldots, d_n)$ to $Seq_{Q,R}(V, D, P)$.

On the other hand, it is possible that all local voting rules are efficient, but the sequential voting procedure is not so.

In fact, let us consider the case where we have two variables $X$ and $Y$ with domain $D(X) = \{a, b\}$ and $D(Y) = \{c, d, e\}$ and three voters. The SCSP of each voter has a single binary constraint on $X$ and $Y$ where tuples $(X = a, Y = c)$, $(X = a, Y = d)$, $(X = a, Y = e)$, $(X = b, Y = c)$, $(X = b, Y = d)$ and $(X = b, Y = e)$ are associated respectively to preferences:

- $1, 0.2, 0.2, 1, 0.2, 0.2$ by the first voter;
- $0.2, 1, 0.2, 1, 0.2, 0.2$ by the second voter;
- $0.2, 0.2, 1, 1, 0.2, 0.2$ by the third voter.
Assume we have that $X < Y$ in the variable ordering, that we are using plurality for all the issues, and that ties are broken lexicographically. Then, when $X$ is considered all voters will vote for $a$. However, there is no complete assignment with $X = a$ that is optimal for all voters, while $(X = b, Y = c)$ is optimal for all voters.

However, if we add the condition that there is a single candidate which is optimal for all agents, then the sequential voting procedure is efficient as well.

**Theorem 8.** If all the local voting rules are efficient, and there is a single candidate which is strictly preferred to all other candidates for all voters, then the sequential voting procedure is efficient.

In fact, if there is a single candidate, say $d = (d_1, \ldots, d_n)$ that is optimal for all agents, then we have that, after DAC, for each agent $i$ and for each variable $j$, $\text{top}(v_j, P_i|v_1=d_1, \ldots, v_{j-1}=d_{j-1}) = d_j$. Thus since each rule is efficient it will elect the value of $d$ assigned to its variable. Thus $\text{Seq}_{O,R}(V, D, P) = d$.

We note that for profiles where there is a unique candidate that is optimal for all agents, efficiency coincides with Condorcet consistency. Thus, given such profiles, the Condorcet consistency of the local rules implies the Condorcet consistency of the sequential rule.

### 5.8 Monotonicity

As defined above, a voting rule is monotonic if, when a candidate wins, and one or more voters improve their vote in favor of this candidate, then the same candidate still wins.

**Theorem 9.** If the sequential voting procedure is monotonic, then each local voting rule is so.

Let us assume that there is a local rule, say $r_i$ that is not monotonic. This means that there are two profiles over the values of variable $v_i$, say $p_i$ and $p'_i$, such that $r_i(p_i) = d_i$, $p'_i$ is as $p_i$ except that some agents have moved $d_i$ up in their orderings, and $r'_i(p_i) = d'_i \neq d_i$.

Let us now consider the fuzzy profile $(V, D, P)$ where:

- for each agent, the preferences over the values of $v_i$ are as in $p_i$;
- all other unary constraints are the same for all agents and associate preference 1 to exactly one value per variable and 0 to all other variables. There are no other constraints.

Assume the result of applying the sequential rule is $\text{Seq}_{O,R}(V, D, P) = (d_1, d_2, \ldots, d_i, \ldots, d_n)$.

Now let us consider the fuzzy profile $(V, D, P')$ where $P'$ differs from $P$ only on the preferences on variable $v_i$ that, for each agent, are as in $p'_i$. We note that in each agent’s SCSP, both in $P$ and $P'$, there are as many solutions as the values in the domain of $v_i$ and that their preference coincides with the one of the corresponding value of $v_i$. Thus $P'$ differs from $P$ only on the preference of the only solution involving value $d_i$, i.e. $\text{Seq}_{O,R}(V, D, P)$, that has been moved up in the ordering of some of the agents. However, the result $\text{Seq}_{O,R}(V, D, P')$ will be $(d_1, d_2, \ldots, d'_i, \ldots, d_n)$. Thus $\text{Seq}$ is not monotonic.
Theorem 10. If each local rule is monotonic, so is the sequential rule.

Let us assume that the sequential rule is not monotonic. This means that there is at least one soft profile \((V, D, P)\) such that \(Seq_{O,R}(V, D, P) = d\) and another soft profile \((V, D, P')\), where \(d\) has been moved up in the preference orderings of some agents, but \(Seq_{O,R}(V, D, P') = d' \neq d\). Let us denote with \(d_i\), resp. \(d'_i\), the value assigned to variable \(v_i\) in \(d\), resp. \(d'\). We note that there must be at least one value on which they differ. For each SCSP in \(P'\) and for each variable \(v_i\), \(d_i\) has either improved w.r.t. \(d'_i\) or remained as in \(P\). Let \(v_j\) be any of the variables such that \(d_j \neq d'_j\). Then \(r_j\) is not monotonic.

The same results can be proven for strong monotonicity, that is, all local voting rules are strongly monotonic iff the sequential rule is so.

6 Conclusions and Future Work

We have proposed a sequential preference aggregation procedure for settings where several agents express their preferences over a common set of variable assignments via soft constraints. We have then studied some properties of this procedure. These properties are similar to those obtained in a setting where agents use CP-nets. However, soft constraints allow us to impose no restrictions on the profiles to obtain such properties, while CP-net profiles need to be severely restricted.

We just considered a few properties of the sequential voting procedure. We plan to study many others in order to better characterize the result of the procedure in terms of the preferences of the agents. We also plan to develop heuristics for an efficient computing of such a result even when the soft constraint problems are not from a tractable class.

Finally we will conduct an experimental study for comparing the performance of sequential vs. classical preference aggregation when a single rule is used for all the variables.

7 Acknowledgments

Research partially supported by the Italian MIUR PRIN project 20089M932N: "Innovative and multi-disciplinary approaches for constraint and preference reasoning”.

References

Load Balancing for Parallel Branch and Bound

Lars Otten and Rina Dechter

Bren School of Information and Computer Sciences
University of California, Irvine
{lotten,dechter}@ics.uci.edu

Abstract. A strategy for parallelization of a state-of-the-art Branch and Bound algorithm for weighted CSPs and other graphical model optimization tasks is introduced: independent worker nodes concurrently solve subproblems, managed by a Branch and Bound master node; the problem cost functions are used to predict subproblem complexity, enabling efficient load balancing, which is crucial for the performance of the parallelization process. Experimental evaluation on up to 20 nodes yields very promising results and suggests the effectiveness of the scheme. The system runs on loosely coupled commodity hardware, simplifying deployment on a larger scale in the future.

1 Introduction

This paper explores parallelization of combinatorial optimization tasks over graphical models like weighted or soft CSP problems and Bayesian networks. Specifically, we consider a state-of-the-art exact optimization algorithm, AND/OR Branch and Bound (AOBB). AOBB, which exploits independencies and unifiable subproblems, has demonstrated superior performance for these tasks compared with other state-of-the-art exact solvers [1] (e.g., it was ranked first or second in the UAI’061 and ’082 evaluations).

To parallelize AOBB we use the established concept of parallel tree search [2] where the tree is explored centrally up to a certain depth and the remaining subtrees are solved in parallel. For graphical models this can be implemented straightforwardly by exploring the search space of partial instantiations up to a certain depth and solving the remaining conditioned subproblems in parallel. This approach has already proven successful for likelihood computation in Superlink-Online, which parallelizes cutset conditioning for linkage analysis tasks [3]. Our work differs in focusing on optimization and in exploiting the AND/OR paradigm, leveraging additional subproblem independence for parallelism. Moreover, we use the power of Branch and Bound in a central search space that manages (and prunes) the set of conditioned subproblems.

The main difference however is that, compared to likelihood computation, optimization presents far greater challenges with respect to load balancing. Hence the primary challenge in search tree parallelization is to determine the “cutoff”, the parallelization frontier. Namely, we need a mechanism to decide when to terminate a branch in the central search space and send the corresponding subproblem to a machine on the network.

1 http://ssli.ee.washington.edu/bilmes/uai06InferenceEvaluation/
2 http://graphmod.ics.uci.edu/uai08/
There are two primary issues: (1) Avoid redundancies: caching of unifiable subproblems is lost across the independently solved subproblems, hence some work might be duplicated; (2) Maintain load balancing among the grid resources, dividing the total work equally and without major idling periods. While introducing redundancy into the search space can be counterproductive for both tasks, load balancing is a far greater challenge for optimization, since the cost function is exploited in pruning the search space. Capturing this aspect is essential in predicting the size of a subproblem and thus the focus of this paper.

The contribution of this work is thus as follows: We suggest a parallel B&B scheme in a graphical model context and analyze some of its design trade-offs. We devise an estimation scheme that predicts the size of future subproblems based on cost functions and learns from previous subproblems to predict the extent of B&B pruning within future subproblems. We show that these complexity estimates enable effective load distribution (which was not possible via redundancy analysis only), and yield very good performance on several very hard practical problem instances, some of which were never solved before. Our approach assumes the most general master-worker scenario with minimal communication and can hence be deployed on a multitude of parallel setups spanning hundreds, if not thousands of computers worldwide. Our current empirical results were obtained on 20 networked desktop computers, but we believe the potential for scaling up is very promising.

Related work: The idea of parallelized Branch and Bound in general is not new, but existing work often assumes a shared-memory architecture or extensive inter-process communication [2, 4–7], or specific grid hierarchies [8]. Early results on estimating the performance of search go back to [9] and more recently [10], which predict the size of general backtrack trees through random probing. Similar schemes were devised for Branch and Bound algorithms [11]: B&B is run for a limited time and the partially explored tree is extrapolated. Our method, on the other hand, is not based on sampling or probing but only uses parameters available a priori and information learned from past subproblems which is facilitated through the use of depth-first branch and bound to explore the master search space.

The paper is organized as follows: Section 2 provides necessary definitions and concepts, while in Section 3 we outline our parallelized AOBB scheme and analyze its parameters through a set of initial experiments. Section 4 derives the complexity estimates required for load balancing, with which we obtain the experimental results in Section 5. Section 6 concludes.

2 Background

We assume the usual definitions of a graphical model as a set of functions over discrete variables, its induced graph, and induced width. In a weighted constraint problem (WCSP), for instance, we aim to find a complete assignment that minimizes the sum of all costs. Figure 1(a) depicts the primal graph of an example problem with six variables. The induced graph for the example problem along ordering $d = A, B, C, D, E, F$ is depicted in Figure 1(b), with two new induced edges, $(B, C)$ and $(B, E)$. Its induced width is 2. Note that different orderings will vary in their implied induced width; find-
Fig. 1: (a) Example primal graph with six variables, (b) its induced graph along ordering \( d = A, B, C, D, E, F \), (c) a corresponding pseudo tree, and (d) the resulting context-minimal AND/OR search graph.

An ordering of minimal induced width is known to be NP-hard, in practice heuristics like minfill are used to obtain approximations [12].

2.1 AND/OR Search Spaces

The concept of AND/OR search spaces has recently been introduced as a unifying framework for advanced algorithmic schemes for graphical models to better capture the structure of the underlying graph [13]. Its main virtue consists in exploiting conditional independencies between variables, which can lead to exponential speedups. The search space is defined using a pseudo tree, which captures problem decomposition:

**Definition 1 (pseudo tree).** Given an undirected graph \( G = (X, E) \), a pseudo tree of \( G \) is a directed, rooted tree \( T = (X, E') \) with the same set of nodes \( X \), such that every arc of \( G \) that is not included in \( E' \) is a back-arc in \( T \), namely it connects a node in \( T \) to an ancestor in \( T \). The arcs in \( E' \) may not all be included in \( E \).

**AND/OR Search Trees:** Given a graphical model instance with variables \( X \) and functions \( F \), its primal graph \( (X, E) \), and a pseudo tree \( T \), the associated AND/OR search tree consists of alternating levels of OR and AND nodes. OR nodes are labeled \( X_i \) and correspond to the variables in \( X \). AND nodes are labeled \( \langle X_i, x_i \rangle \), or just \( x_i \) and correspond to the values of the OR parent’s variable. The structure of the AND/OR search tree is based on the underlying pseudo tree \( T \): the root of the AND/OR search tree is an OR node labeled with the root of \( T \). The children of an OR node \( X_i \) are AND nodes labeled with assignments \( \langle X_i, x_i \rangle \) that are consistent with the assignments along the path from the root; the children of an AND node \( \langle X_i, x_i \rangle \) are OR nodes labeled with the children of \( X_i \) in \( T \), representing conditionally independent subproblems. It was shown that, given a pseudo tree \( T \) of height \( h \), the size of the AND/OR search tree based on \( T \) is \( O(n \cdot k^h) \), where \( k \) bounds the domain size of variables [13].

**AND/OR Search Graphs:** In an AND/OR search tree, different nodes may root identical subproblems. These nodes can be merged, yielding an AND/OR search graph of smaller size, at the expense of using additional memory during search. Some mergeable nodes can be identified by their contexts:
**Definition 2 (context).** Given the pseudo tree $T$ of an AND/OR search space, the context of an OR node $X_i$ is the set of ancestors of $X_i$ in $T$, that are connected in the primal graph to $X_i$ or its descendants (in $T$). The context of $X_i$ separates the subproblem below $X_i$ from the rest of the network. Merging all context-mergeable nodes in the AND/OR search tree yields the context minimal AND/OR search graph [13].

**Proposition 1.** [13] Given a graphical model, its primal graph $G$, and a pseudo tree $T$, the size of the context-minimal AND/OR search graph is $O(n \cdot k^{w^*})$, where $w^*$ is the induced width of $G$ over a depth-first traversal of $T$ and $k$ bounds the domain size.

**Example 1.** Figure 1(c) depicts a pseudo-tree extracted from the induced graph in Figure 1(b) and Figure 1(d) shows the corresponding context-minimal AND/OR search graph. Note that the AND nodes for $B$ have two children each, representing independent subproblems and thus demonstrating problem decomposition. Furthermore, the OR nodes for $D$ (with context $\{B, C\}$) and $F$ (context $\{B, E\}$) have two edges converging from the AND level above them, signifying caching.

**Weighted AND/OR Search Graphs:** Given an AND/OR search graph, each edge from an OR node $X_i$ to an AND node $x_i$ can be annotated by weights derived from the set of cost functions $F$ in the graphical model: the weight $l(X_i, x_i)$ is the sum of all cost functions whose scope includes $X_i$ and is fully assigned along the path from the root to $x_i$, evaluated at the values along this path. Furthermore, each node in the AND/OR search graph can be associated with a value: the value $v(n)$ of a node $n$ is the minimal solution cost to the subproblem rooted at $n$, subject to the current variable instantiation along the path from the root to $n$. $v(n)$ can be computed recursively using the values of $n$’s successors [13].

### 2.2 AND/OR Branch and Bound

AND/OR Branch and Bound is a state-of-the-art algorithm for solving optimization problems over graphical models. Assuming a minimization task, it traverses the context-minimal AND/OR graph in a depth-first manner while keeping track of a current upper bound on the optimal solution cost. It interleaves forward node expansion with a backward cost revision or propagation step that updates node values (capturing the current best solution to the subproblem rooted at each node), until search terminates and the optimal solution has been found [13].

### 3 Parallel Setup and Scheme

We assume a very general parallel framework in which autonomous hosts are loosely connected over some network – in our case we use ten dual-core desktop computers, with CPU speeds between 2.33 and 3.0 GHz, on a local Ethernet, thus allowing experiments with up to 20 parallel nodes. We impose a master-worker hierarchy on the computers in the network, where a special master node runs a central process to coordinate the workers, which cannot communicate with each other. This general model is
chosen to accommodate a wide range of parallel resources, where direct node communication is often either prohibitively slow or entirely impossible; it also facilitates flexible deployment on geographically dispersed, heterogeneous resources in the future.

The setup is similar to Superlink-Online [3], which has been very successful in using large-scale parallelism in likelihood algorithms for genetic linkage analysis, or SETI@home [14], which uses Internet-connected PCs around the world to search through enormous amounts of radio data. Like Superlink-Online, our system is implemented on top of the Condor grid middleware [15].

3.1 Parallel AND/OR Branch and Bound

Given a reasoning problem over a graphical model instance and a pseudo tree $T$, a straightforward approach to parallelize the search process is to have the master process explore a start pseudo tree:

**Definition 3 (start pseudo tree, parallelization frontier).** Given an undirected graph $G = (X, E)$, a directed rooted tree $T_c = (X_c, E_c)$, where $X_c \subset X$, is a start pseudo tree if it has the same root as, and is a subgraph of some pseudo tree of $G$. Given a start pseudo tree $T_c$, we refer to the set of variables corresponding to the leaf nodes of $T_c$ as the parallelization frontier. Each variable in the parallelization frontier roots a collection of subproblems characterized by value assignments along the path from the root to the variable.

**Example 2.** Consider again the AND/OR search graph in Figure 1(d). Given a start pseudo tree having $A$ and $B$, we can illustrate the parallelization scheme through Figure 2: the search space of the master process is marked in gray, and each of the eight independent subproblems rooted at $C$ or $E$ can be solved in parallel. (Notice, however, that some redundancy is introduced.)

3.2 Master Process Details

As a Branch and Bound algorithm, the master implements node expansion (or exploration) and propagation as outlined in the following (see [16] for details):
Table 1: Results on hard pedigree instances with $p=15$ workers (timeout 24 hours). $N$ is the number of problem variables, $k$ the max. domain size, $w$ the induced width along the chosen minfill ordering, and $h$ the height of the corresponding pseudo tree. $T_s$ is the solution time (in seconds) of sequential AOBB graph search on a single 3.0 GHz processor. The parallel solution time $T_p$ is given for varying cutoff depth $d$.

<table>
<thead>
<tr>
<th></th>
<th>$N$</th>
<th>$k$</th>
<th>$w$</th>
<th>$h$</th>
<th>$T_s$</th>
<th>$d=4$</th>
<th>$d=5$</th>
<th>$d=6$</th>
<th>$d=7$</th>
<th>$d=8$</th>
<th>$d=9$</th>
<th>$d=10$</th>
<th>$d=11$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ped7</td>
<td>1068</td>
<td>432</td>
<td>90</td>
<td>19,114</td>
<td>21,334</td>
<td>3,552</td>
<td>3,610</td>
<td>4,560</td>
<td>4,675</td>
<td>7,075</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ped13</td>
<td>1077</td>
<td>332</td>
<td>102</td>
<td>2,752</td>
<td>379</td>
<td>519</td>
<td>504</td>
<td>662</td>
<td>1,184</td>
<td>1,192</td>
<td>4,342</td>
<td>4,758</td>
<td>6,414</td>
</tr>
<tr>
<td>ped19</td>
<td>793</td>
<td>325</td>
<td>98</td>
<td>time</td>
<td>47,600</td>
<td>30,781</td>
<td>27,797</td>
<td>27,327</td>
<td>39,282</td>
<td>47,568</td>
<td>64,148</td>
<td>81,103</td>
<td></td>
</tr>
<tr>
<td>ped31</td>
<td>1183</td>
<td>530</td>
<td>85</td>
<td>77,580</td>
<td>68,472</td>
<td>47,089</td>
<td>46,837</td>
<td>43,097</td>
<td>41,286</td>
<td>22,582</td>
<td>15,200</td>
<td>15,313</td>
<td></td>
</tr>
<tr>
<td>ped41</td>
<td>1062</td>
<td>533</td>
<td>100</td>
<td>14,643</td>
<td>4,256</td>
<td>3,069</td>
<td>2,173</td>
<td>2,251</td>
<td>2,881</td>
<td>4,476</td>
<td>7,662</td>
<td>11,159</td>
<td></td>
</tr>
</tbody>
</table>

Exploration. The master process explores the AND/OR graph in a depth-first manner guided by the start pseudo tree $T_c$. Upon expansion of a node $n$ it consults a heuristic lower bound $lb(n)$ to make pruning decisions, where the computation of the upper bound $ub(n)$ can take into account previous subproblem solutions. If $lb(n) \geq ub(n)$, the current subtree can be pruned. Exploration is halted when the parallelization frontier is reached. The master then sends the respective subproblem, given by the subproblem root variable and its context instantiation, to a worker node.

Propagation. The master process also collects and processes subproblem solutions from the worker nodes. Upon receipt of a solved subproblem, its solution is assigned as the value of the respective node in the master search space and recursively propagated upwards towards the root, updating node values identical to sequential AOBB.

Assuming a fixed number of workers $p$, the master initially generates only the first $p$ subproblems; whenever a worker finishes, its solution is propagated and the central exploration is resumed to generate the next subproblem.

3.3 Initial Experiments

The central decision is obviously where to place the parallelization frontier, i.e., at which point to cut off the master search space, which will determine subproblems sent to worker nodes. And while in the end the parallel scheme should make this decision automatically, we investigate the performance of the general parallelization approach through initial experiments with the cutoff set manually.

We consider two sets of hard problems, pedigree networks and mastermind game instances, both part of the UAI’08 evaluation\(^3\). Based on a pseudo tree computed from a minfill variable ordering, we enforce the parallelization frontier at constant depth $d$ in the master search space. The mini bucket scheme is used to generate the heuristic function [17].

Haplotyping Problems: The first set of problems consists of pedigree networks from the area of human genetic analysis, specifically haplotyping problems. These can

\(^3\)http://graphmod.ics.uci.edu/uai08/
Table 2: Results on Mastermind instances with $p = 10$ workers (columns as in Table 1).

<table>
<thead>
<tr>
<th>instance</th>
<th>N</th>
<th>k</th>
<th>w</th>
<th>h</th>
<th>$d=5$</th>
<th>$d=6$</th>
<th>$d=7$</th>
<th>$d=8$</th>
<th>$d=9$</th>
<th>$d=10$</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm_03_08_03-0011</td>
<td>1612</td>
<td>2</td>
<td>37</td>
<td>99</td>
<td>1.698</td>
<td>2.866</td>
<td>1.433</td>
<td>1.546</td>
<td>1.518</td>
<td>1.260</td>
</tr>
<tr>
<td>mm_03_08_03-0012</td>
<td>1612</td>
<td>2</td>
<td>37</td>
<td>81</td>
<td>7.568</td>
<td>2.386</td>
<td>2.146</td>
<td>1.430</td>
<td>1.498</td>
<td>1.649</td>
</tr>
<tr>
<td>mm_04_08_04-0009</td>
<td>2616</td>
<td>2</td>
<td>37</td>
<td>79</td>
<td>10.620</td>
<td>1.594</td>
<td>1.322</td>
<td>1.306</td>
<td>1.358</td>
<td>1.539</td>
</tr>
<tr>
<td>mm_06_08_03-0009</td>
<td>814</td>
<td>2</td>
<td>31</td>
<td>72</td>
<td>12.595</td>
<td>1.798</td>
<td>1.797</td>
<td>1.798</td>
<td>1.820</td>
<td>1.905</td>
</tr>
<tr>
<td>mm_10_08_03-0009</td>
<td>2600</td>
<td>2</td>
<td>47</td>
<td>99</td>
<td>26.102</td>
<td>4.593</td>
<td>4.417</td>
<td>4.481</td>
<td>4.866</td>
<td>4.259</td>
</tr>
<tr>
<td>mm_10_08_03-0011</td>
<td>2558</td>
<td>2</td>
<td>47</td>
<td>101</td>
<td>84.029</td>
<td>13.413</td>
<td>11.052</td>
<td>11.279</td>
<td>11.044</td>
<td>10.887</td>
</tr>
<tr>
<td>mm_10_08_03-0012</td>
<td>2558</td>
<td>2</td>
<td>47</td>
<td>82</td>
<td>5.638</td>
<td>2.420</td>
<td>1.357</td>
<td>1.362</td>
<td>1.381</td>
<td>1.473</td>
</tr>
<tr>
<td>mm_10_08_03-0013</td>
<td>2558</td>
<td>2</td>
<td>46</td>
<td>99</td>
<td>10.385</td>
<td>4.229</td>
<td>4.339</td>
<td>4.248</td>
<td>4.466</td>
<td>2.413</td>
</tr>
</tbody>
</table>

be translated into a MPE task over a Bayesian network [18] or, by moving to the log domain, a weighted CSP. Looking at the results in Table 1, the parallel scheme seems effective in almost all cases that we tested: for instance, ped13 can be solved in less than 10 minutes for several values of $d$, whereas the sequential algorithm takes 45 minutes. ped31 takes close to 22 hours on a single computer, while the parallel scheme with $d = 10$ solves the problem in 4 hours and 15 minutes. Moreover, the parallel scheme was able to solve two instances, ped19 and ped51, on which the sequential algorithm timed out after 24 hours.

Mastermind Problems: Table 2 documents experiments on some hard Mastermind instances, with $T_s$ from a few hours to almost a day. Similar to the pedigree instances, parallelization enabled significant improvements in overall running time. For example, with $d = 7$, mm_04_08_04-0000 went from 3 hours to little over 22 minutes, while mm_10_08_03-0011 improved from more than 23 hours to about 3 hours for $d = 9$.

As evidenced by the results above, our parallel scheme carries high potential, yet its performance depends highly on the chosen parallelization frontier. In making this decision we can identify the following three objectives: (1) Minimize the redundancy and the associated blowup in the search space, that is inherent to the conditioning scheme (cf. Example 2). (2) Balance the workload over all processing units, each of which should be utilized equally to optimize efficiency and improve overall running time. (3) Minimize overhead resulting from grid communication and resource management. We address the issue of redundancy next.

3.4 Practical Impact of Redundancy

Recall that enforcing the parallelization frontier can introduce redundancies into the search space, since caching is not possible across subproblem boundaries. We have therefore developed fine-grained expressions which use the underlying structure of the graphical model to analytically capture the size of conditioned subproblems and the overall parallel search space as a function of the cutoff depth $d$ – for space reasons we have to refer to [16] for details.

We point out, however, that this reasoning only yields an upper bound on the search space size since it accounts neither for the pruning in AOBB nor for determinism, which causes early backtracking. The latter can be partially incorporated as described in [19],
yielding tighter bounds. As we show next, however, the main shortcoming of these bounds in the context of Branch and Bound actually lies in their disregard of the cost function, the guiding heuristic, and the associated pruning effect. Note that when we modified our scheme to compute likelihood (e.g., probability of evidence), in a very preliminary evaluation we observed a far better connection between our redundancy-based estimation and the true size of the search space. This suggests that structure-based analysis can play an important role for parallelizing likelihood queries, an aspect we plan to investigate in the future.

**Redundancy**: To investigate the practical impact of redundancy, we recorded the number of nodes generated in the master search space and across all subproblems for each parallel run; we also computed the upper bound from [16], extended to account for determinism [19]. Figure 3(a) contrasts these two measures as a function of \( d \) on two pedigrees: for both instances we observe that the exponential blowup of the upper bound with increasing \( d \) (due to redundancy) is not at all reflected in the actual number of nodes generated – evidently the pruning of AOBB is powerful enough to compensate for this and only a very small portion of the total search space is actually explored.

**Subproblem Bounds**: In order to evaluate the obtained subproblem bounds with respect to the size of the explored search space we recorded the number of nodes actually generated by the worker and contrast it with the respective precomputed subproblem bound. Exemplary results for two pedigree instances are shown in Figure 3(b), where the first 75 subproblems are plotted in order of their generation. We note that the upper bound doesn’t change across subproblems; yet the actual size exhibits significant variance, going anywhere from a few thousand nodes (solvable in seconds) to many millions. Similar findings hold for other haplotyping and Mastermind instances.

In the following, we will thus focus on predicting the impact of pruning in AOBB, based on the problem’s cost function and the resulting upper and lower bounds.
4 Predicting Subproblem Size Using the Cost Function

In this section we derive a scheme for estimating the size of the explored search space of a conditioned subproblem using parameters associated with the problem’s cost function. Our goal is to discriminate between “easy” and “hard” subproblems to allow efficient load-balancing within our parallel scheme. In particular, we wish to enforce an upper bound on the complexity of subproblems (measured in the number of nodes expanded).

When considering a particular subproblem rooted at node \( n \), we propose to estimate its complexity \( N(n) \) (i.e., the number of nodes AOBB explores to solve it) as a function of the heuristic lower bound \( L(n) \) as well as the upper bound \( U(n) \), which can be computed based on earlier parts of the search space or through an approximation algorithm like local search; we will also use the height \( h(n) \) of the subproblem pseudo tree. The general expression we propose has the form:

\[
N(n) = b^{U(n) - L(n) - h(n)^\alpha}
\]  

(1)

where \( b, inc, \) and \( \alpha \) are constants. In the following we provide the rationale for this functional form and demonstrate how the free parameters can be learned as the search progresses.

4.1 Main Assumptions

We consider a node \( n \) that roots the subproblem \( P(n) \). If the search space below \( n \) was a perfectly balanced tree of height \( D \), with every node having exactly \( b \) successors, clearly the total number of nodes is \( N = (b^{D+1} - 1)/(b - 1) \approx b^D \).

However, even if the underlying search space is balanced, the portion expanded by B&B, guided by some heuristic evaluation function, is not: the more accurate the heuristic, the more focused around the optimal solution paths the search space will be. In state-based search spaces it is therefore common to measure effectiveness in post-solution analysis via the effective branching factor defined as \( b = \sqrt[\bar{D}]{N} \) where \( D \) is the length of the optimal solution path and \( N \) is the actual number of nodes generated [20].

Inspired by this approach, for a subproblem rooted at \( n \) we adopt the idea of approximating the explored search space by a balanced tree and express its size through \( N(n) = b(n)^D(n) \). However, in place of the optimal solution path length (which corresponds to the pseudo tree height in our case), we propose to interpret \( D(n) \) as the average leaf node depth \( \bar{D}(n) \) defined as follows:

**Definition 4 (Average leaf node depth).** Let \( l_1, \ldots, l_j \) denote the leaf nodes generated when solving subproblem \( P(n) \). We define the average leaf node depth of \( P(n) \) to be \( D(n) := \frac{1}{j} \sum_{k=1}^{j} d_{n}(l_k) \), where \( d_{n}(l_i) \) denotes the depth of leaf node \( i \) relative to the subproblem root \( n \).

Figure 4 plots the number of nodes generated within a subproblem as a function of the average leaf node depth for \( d = 8 \) and \( d = 10 \), respectively. We can see a log-linear behavior (note the logarithmic vertical axis), thus supporting the general exponential form of \( N(n) = b(n)^{D(n)} \).

We next aim to express \( b(n) \) and \( D(n) \) as functions of the subproblem parameters \( L(n), U(N), \) and \( h(n) \) (using other parameters is subject to future research).
4.2 Estimating the Effective Branching Factor

For the sake of simplicity we assume an underlying, “true” effective branching factor $b$ that is constant for all possible subproblems. We feel this is a reasonable assumption since all subproblems are conditioned within the same graphical model. Figure 4 exhibits some variance in complexity for fixed average leaf depth. This suggests modeling $b(n)$ as a random variable and assuming a normal distribution we can take the mean as the constant $b$. An obvious way to learn this parameter is then to average over the effective branching factors of previous subproblems, which is known to be the right statistic for estimating the true average of a population.

**Estimating $b$ for new Subproblem $P(n)$:**

Given a set of already solved subproblems $P(n_1), \ldots, P(n_r)$, we can compute $D(n_i)$ and derive effective branching degrees $b(n_i) = \frac{d(n_i)}{\sqrt{N(n_i)}}$ for all $i$. We then estimate $b$ through $b = \frac{1}{r} \sum_{i=1}^{r} b(n_i)$.

4.3 Deriving and Predicting Average Leaf Depth

With each subproblem $P(n)$ rooted at a node $n$ we associate a lower bound $L(n)$ based on the heuristic estimate and an upper bound $U(n)$ derived from the best solution from previous subproblems\(^4\). Both $L(n)$ and $U(n)$ are known before we start solving $P(n)$. We can assume $L(n) < U(n)$, since otherwise $n$ itself could be pruned and $P(n)$ was trivially solved. We denote with $lb(n')$ and $ub(n')$ the lower and upper bounds of nodes $n'$ within the subproblem $P(n)$ at the time of their expansion and similarly assert that $lb(n') \leq ub(n')$ for any expanded node $n'$.

Since the upper bound is derived from the best solution found so far it can only improve throughout the search process. Furthermore, assuming a monotonic heuristic function (that provides for any node $n'$ a lower bound on the cost of the best solution path going through $n'$), the lower bounds along any path in the search space are non-decreasing and we can state that any node $n'$ expanded within $P(n)$ satisfies:

$$L(n) \leq lb(n') < ub(n') \leq U(n)$$

Consider now a single path within $P(n)$, from $n$ down to leaf node $l_k$, and denote it by $\pi_k = (n_0', \ldots, n_{d_n(l_k)}')$, where $n_0' = n$ and $d_n(l_k)$ is again the depth of $l_k$ with respect to $n$ (and hence $n_{d_n(l_k)}' = l_k$). We will write $lb_i$ for $lb(n_i')$ and $ub_i$ for $ub(n_i')$, respectively, and can state that $lb_i \geq lb_{i-1}$ and $ub_i \leq ub_{i-1}$ for all $1 \leq i \leq d_n(l_k)$ (note \(^4\)In the following we assume a graphical model with addition as the combination operator (a weighted CSP, for instance). Adaption to multiplication is straightforward.)
that \( l_0 = L(n) \) and \( u_0 = U(n) \). An internal node \( n' \) is pruned iff \( lb(n') \geq ub(n') \) or equivalently \( ub(n') - lb(n') \leq 0 \), hence we consider the (non-increasing) sequence of values \((ub_i - lb_i)\) along the path \( \pi_k \); in particular we are interested in the average change in value from one node to the next, which we capture as follows:

**Definition 5 (Average path increment).** The average path increment of \( \pi_k \) within \( P(n) \) is defined by the expression:

\[
inc(\pi_k) = \frac{1}{d_n(l_k)} \sum_{i=1}^{d_n(l_k)} ((ub_i - lb_i) - (ub_{i-1} - lb_{i-1}))
\]  

(2)

We note that \( l_k \) is a leaf node and assume \((ub_{d_n(l_k)} - lb_{d_n(l_k)}) = 0\), so the sum reduces to \((U(n) - L(n))\). Thus rewriting Expression 2 for \( d_n(l_k) \) and averaging to get \( D(n) \) as in Definition 4 yields:

\[
D(n) = (U(n) - L(n)) \cdot \frac{1}{j} \sum_{k=1}^{j} \frac{1}{inc(\pi_k)}
\]  

(3)

We now define \( inc(n) \) of \( P(n) \) through \( inc(n)^{-1} = \frac{1}{j} \sum_{k=1}^{j} \frac{1}{inc(\pi_k)} \), with which Expression 3 becomes \( D(n) = (U(n) - L(n)) \cdot inc(n)^{-1} \), namely an expression for \( D(n) \) as a ratio of the distance between the initial upper and lower bounds and \( inc(n) \).

Note that in post-solution analysis \( D(n) \) is known and \( inc(n) \) can be computed directly, without considering each \( \pi_j \).

One more aspect that has been ignored in the analysis so far, but which is likely to have an impact, is the actual height \( h(n) \) of the subproblem pseudo tree. We therefore propose to scale \( D(n) \) by a factor of the form \( h(n)^\alpha \); in our experiments we found \( \alpha = 0.5 \) to yield good results\(^5\). The general expression we obtain is thus:

\[
\frac{D(n)}{h(n)^\alpha} = \frac{(U(n) - L(n))}{inc(n)}
\]  

(4)

**Predicting \( D(n) \) for New Subproblem \( P(n) \):** Given previously solved subproblems \( P(n_1), \ldots , P(n_r) \), we need to estimate \( inc(n) \) in order to predict \( D(n) \). Namely, we compute \( inc(n_i) = (U(n_i) - L(n_i)) \cdot h(n_i)^\alpha \cdot D(n_i)^{-1} \) for \( 1 \leq i \leq r \). Assuming again that \( inc(n) \) is a random variable distributed normally we take the sample average to estimate \( inc^* = \frac{1}{r} \sum_{i=1}^{r} inc(n_i) \). Using Equation 4, our prediction for \( D(n) \) is:

\[
\hat{D}(n) = \frac{(U(n) - L(n)) \cdot h(n)^\alpha}{inc}
\]  

(5)

**Predicting \( N(n) \) for a New Subproblem \( P(n) \):** Given the estimates \( \hat{b} \) and \( \hat{inc} \) as derived above, we will predict the number of nodes \( N(n) \) generated within \( P(n) \) as:

\[
\hat{N}(n) = \hat{b} \cdot \hat{D}(n)
\]  

(6)

The assumption that \( inc \) and \( \hat{b} \) are constant across subproblems is clearly too strict, more complex dependencies will be investigated in the future. For now, however, even this basic approach has proven to yield good results, as we demonstrate in Section 5.

\(^5\) Eventually \( \alpha \) could be subject to learning as well.
Table 3: Results of the automated parallel scheme (ped: \( p = 15 \), mm: \( p = 10 \)). \( T_s \) is the parallel time from Tables 1/2, \( T_{SLS} \) the time of sequential AOBB with one iteration of SLS to find an initial bound, \( T^*_p \) the best-performing fixed-depth cutoff from Tables 1/2, and \( T_{auto} \) the overall solution time of the automated parallel scheme.

<table>
<thead>
<tr>
<th>instance</th>
<th>( T_s )</th>
<th>( T_{SLS} )</th>
<th>( T^*_p )</th>
<th>( T_{auto} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ped7</td>
<td>19,114</td>
<td>19,309</td>
<td>2,783</td>
<td>2,783</td>
</tr>
<tr>
<td>ped13</td>
<td>77,580</td>
<td>37,844</td>
<td>15,230</td>
<td>3,910</td>
</tr>
<tr>
<td>ped19</td>
<td>2,752</td>
<td>2,796</td>
<td>379</td>
<td>359</td>
</tr>
<tr>
<td>ped41</td>
<td>14,643</td>
<td>13,999</td>
<td>2,173</td>
<td>2,251</td>
</tr>
<tr>
<td>mm_03_08_05-0011</td>
<td>9,715</td>
<td>2,943</td>
<td>1,430</td>
<td>1,357</td>
</tr>
<tr>
<td>mm_03_08_05-0012</td>
<td>7,568</td>
<td>2,030</td>
<td>1,430</td>
<td>1,357</td>
</tr>
<tr>
<td>mm_04_08_04-0000</td>
<td>10,620</td>
<td>7,807</td>
<td>1,306</td>
<td>1,357</td>
</tr>
<tr>
<td>mm_06_08_03-0000</td>
<td>12,985</td>
<td>259</td>
<td>1,797</td>
<td>1,238</td>
</tr>
</tbody>
</table>

4.4 Parameter Initialization

To find an initial estimate of both the effective branching factor as well as the average increment, the master process performs 15 seconds of sequential search. It keeps track of the largest subproblem \( P(n_0) \) solved within that time limit and extracts \( b(n_0) \) as well as \( inc(n_0) \), which will then be used as initial estimates for the first set of cutoff decisions. As an additional preprocessing step, we perform a brief run of stochastic local search [21], which returns a solution that is not necessarily optimal, but in practice usually close to it. This is given as an additional input to the master process to provide initial lower bounds for the subproblem estimation.

5 Experiments

We ran our parallel AOBB scheme on the same set of problem instances as in Section 3.3, using the above prediction scheme to make the cutoff decision. The master process, however, now makes the cutoff decisions fully automatically: for each node \( n \) in the master search space, the complexity of \( P(n) \) solved within that time limit and extracts \( b(n_0) \) as well as \( inc(n_0) \), which will then be used as initial estimates for the first set of cutoff decisions. As an additional preprocessing step, we perform a brief run of stochastic local search [21], which returns a solution that is not necessarily optimal, but in practice usually close to it. This is given as an additional input to the master process to provide initial lower bounds for the subproblem estimation.

Pedigree Networks: Results on the set of complex pedigree instances are given in Table 3. We can see that in all cases the automatic scheme does at least as good as the best fixed cutoff, in some cases even better. Again it is important to realize that \( T^*_p \) in Table 3 is the result of trying various fixed cutoff values \( d \) and selecting the best one, whereas \( T_{auto} \) requires no such “trial and error”. In case of ped31 the SLS initialization is quite effective for the sequential algorithm, cutting computation from 21 to approx. 10 hours – yet the automated scheme improved upon this by a factor of almost 10, to just above one hour. Furthermore, for ped51 and in particular ped19, both of which could not be solved sequentially, \( T_{auto} \) marks a good improvement over \( T^*_p \).

Mastermind Networks: Table 3 also includes results of the automated scheme for the set of Mastermind instances. Here the SLS preprocessing has a larger impact in
general, improving the sequential solution times considerably. And again, in most cases the automated scheme performs at least as well as the best fixed cutoff (determined after trying various depths). There are, however, some notable exceptions like mm_04_08_04, where \( T_{auto} \) is about two times \( T_p^* \) – our analysis here showed that the initial parameters for the subproblem prediction were too far off. We are therefore confident that a future, improved initialization scheme would alleviate these issues.

5.1 Subproblem Statistics

Figures 5(a) and (b) contain detailed subproblem statistics for the first 75 subproblem generated by the automated parallelization scheme on ped31 and ped51, respectively. Each plot shows actual and predicted number of nodes as well as the (constant) threshold that was used in the parallelization decision. The cutoff depth of the subproblem root is depicted against a separate scale to the right.

We see that the prediction scheme does not give perfect predictions (which was expected), but it reliably captures the trend. Furthermore, the actual subproblem complexities are all contained within an interval of roughly one order of magnitude, which is significantly more balanced than the results for fixed cutoff depths (cf. Figure 3(b)).

We also note that “perfect” load balancing is impossible to obtain in practice, because subproblem complexity can vary greatly from one depth level to the next along a single path. In particular, if a subproblem at depth \( d \) is deemed too complex, most of this complexity might stem from only one of its child subproblems at depth \( d+1 \), with the remaining ones relatively simple – yet solved separately. In light of this, we consider the above results very promising.

5.2 Performance Scaling

At this time we only have a limited set of computational resources at our disposal, yet we wanted to perform a preliminary evaluation of how the system scales with \( p \), the number of workers. We hence ran the automated parallel scheme with \( p \in \{5, 10, 15, 20\} \) workers and recorded the overall solution time in each case.
Figure 6 plots the relative speedup of the overall solution in relation to \( p = 5 \) workers. For nearly all instances the behavior is as expected, at times improving linearly with the number of workers, although not always at a 1:1 ratio. It is evident that relatively complex problem instances profit more from more resources; in particular ped51 sees a two-, three-, and fourfold improvement going to twice, thrice, and four times the number of workers, respectively. For simpler instances, we think the subproblem threshold of approx. 20 minutes is too close to the overall problem complexity, thereby inhibiting better scaling.

6 Conclusion & Future Work

This paper presents a new framework for parallelization of AND/OR Branch and Bound (AOBB), a state-of-the-art optimization algorithm over graphical models. In extending the known idea of parallel tree search to AOBB, we show that generating independent subproblems can itself be done through an AOBB procedure, where previous subproblem solutions are dynamically used as bounds for pruning new subproblems. The underlying parallel framework is very general and makes minimal assumptions about the available parallel infrastructure, making this approach viable on many different parallel and distributed resources pools (e.g. just a set of networked commodity hardware).

We addressed the two central objectives of any parallelization scheme – minimizing redundancy and optimizing load balancing – in the context of our AOBB algorithm. In particular we analyzed the relation of the above aspects to the cutoff frontier, the main parallelization parameter of our scheme. The very restricted communication in the assumed parallel architecture makes this the central decision, in contrast to more forgiving, “work-stealing” approaches that can still compensate later on [6, 7].

Experiments have shown that analytic expressions quantifying redundancy based only on the structure of the underlying search space are not effective in practice, since performance is dominated by the pruning power of AOBB. Our focus in this paper is therefore on deriving predictions that better capture the performance of AOBB using the problem’s cost function, which underlies the algorithm’s pruning decisions. We proposed to estimate the size of the explored search space using an exponential functional form using certain subproblem parameters. We then proposed a scheme for learning this function’s free parameters from previously solved subproblems. We have demonstrated empirically the effectiveness of the estimates, leading to far better workload balancing and improved solution times on hard problems.

We acknowledge that this initial estimation scheme, while justified and effective, still includes some ad hoc aspects. We aim to advance the scheme by taking into account additional parameters and by providing firm theoretical grounds for our approach.

Besides extending the scheme itself, future work will also more thoroughly investigate the issue of scaling, using larger grid setups than what we had access to so far. We
also need to conduct more experiments on hard problems from various domains. Suitable ones must not be too easy to yield meaningful results with our advanced scheme, yet not too complex to run experiments in a reasonable time frame, which has proven somewhat elusive.

References

Combining approaches for solving Satisfiability problems with Preferences and their Evaluation

Emanuele Di Rosa, Enrico Giunchiglia, and Barry O’Sullivan

1 DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
{emanuele, enrico}@dist.unige.it

2 Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{b.osullivan@4c.ucc.ie}

Abstract. The ability to effectively reason in the presence of qualitative preferences on literals or formulas is a central issue in Artificial Intelligence. In the last few years, two procedures have been presented in order to reason with propositional satisfiability (SAT) problems in the presence of additional, partially ordered qualitative preferences on literals or formulas: the first requires a modification of the branching heuristic of the SAT solver in order to guarantee that the first solution is optimal, while the second computes a sequence of solutions, each guaranteed to be better than the previous one. The two approaches have been compared on specific classes of instances —each having an empty partial order— showing that the second has superior performance.

In this paper we show that these two approaches can be combined yielding a new effective procedure. In particular, we modify the branching heuristic —as in the first approach— by possibly changing the polarity of the returned literal, and then we continue the search —as in the second approach— looking for better solutions. We extended the experimental analysis conducted in previous papers by considering a wide variety of problems, having both an empty and a non-empty partial order: the results show that the new procedure performs better than the two previous approaches on average, and especially on the “hard” problems.

1 Introduction

The ability to effectively reason in the presence of qualitative preferences on literals or formulas is a central issue in Artificial Intelligence. For example, in [2] qualitative preferences are represented as CP-nets and various examples of applications are discussed, while in [1, 8] qualitative preferences are used to impose additional “soft” constraints to be possibly satisfied while planning. In two previous papers [7, 11], it has been shown how it is possible to extend current state-of-the-art procedures for propositional satisfiability (SAT) in order to compute an optimal model w.r.t. a given set of qualitative preferences, expressed as a partially ordered set \( \langle S, \prec \rangle \) of literals or formulas: intuitively, \( S \) represents the preferences that we would like to have satisfied, \( \prec \) models their relative importance, and a model \( \mu \) of a formula \( \varphi \) is optimal if it is a minimal element of the partial order on the models of \( \varphi \) induced by \( \langle S, \prec \rangle \).

Given a formula \( \varphi \) and a (qualitative) preference \( \langle S, \prec \rangle \), each of the two procedures presented in [7, 11] has its own pros and cons. The first, called OPTSAT-HS, requires
a simple modification of the branching heuristic of the SAT solver in order to explore
the search space according to \( \langle S, \prec \rangle \): in this way, the first model computed by the procedure is guaranteed to be optimal; however, it is well known that imposing a given predetermined (partial) order on the branching heuristic can have a dramatic impact on the performance of the solver (see, e.g. [4, 9]). The second, called \textsc{Optsat-BF}, whenever a model \( \mu \) of \( \varphi \) is found, extends \( \varphi \) with a formula \( \varphi_\mu \) ruling out the models which are not preferred to \( \mu \): in this way, no modification of the heuristic of the SAT solver is needed; however, finding the optimal model \( \mu \) may require the generation of (exponentially) many non-optimal intermediate models. \textsc{Optsat-HS} and \textsc{Optsat-BF} have been compared in [11] on the qualitative version of various (partial) min-one and maxsat problems, showing that \textsc{Optsat-BF} has better performance than \textsc{Optsat-HS} on average. However, these benchmarks are somehow very specific because they correspond to problems in which preferences are not partially ordered.

In this paper we propose a new procedure, called \textsc{Sat&Pref}, which can be seen as a combination of \textsc{Optsat-HS} and \textsc{Optsat-BF}. In fact, in \textsc{Sat&Pref} the branching heuristic is modified in order to have, when possible, the branching literal in the set \( S \) of preferences, and, when a model is found, the search is continued looking for better models. In more details, assuming we are given a preference \( \langle S, \prec \rangle \) on literals and that the branching heuristic of the SAT solver returns that the variable \( x \) has to be assigned to true:

1. if \( x \in S \) or \( x \not\in S \), \textsc{Sat&Pref} assigns \( x \) to true without affecting the branching heuristic,
2. otherwise, \( x \not\in S \) and \( x \in S \), and in this case \textsc{Sat&Pref} assigns \( x \) to false, thus changing the polarity assigned by the SAT solver.

Thus, in \textsc{Sat&Pref},

1. No ordering on the branching heuristic is imposed, and thus the negative results in [4, 9] affecting \textsc{Optsat-HS}, do not apply to \textsc{Sat&Pref}; and
2. By fixing, when possible, the polarity of the assigned variable in accordance with the preferences, we expect \textsc{Sat&Pref} to generate less intermediate models than \textsc{Optsat-BF}, and thus to gain in performance, especially on problems with many, totally ordered preferences.

In order to test our hypotheses, we implemented \textsc{Optsat-HS}, \textsc{Optsat-BF} and \textsc{Sat&Pref} on a common platform based on the SAT solver \textsc{Minisat} [5]. We then consider the benchmarks in [7, 11], and show that changing the polarity of the branching variable in accordance with the preferences can have a dramatic positive impact on the number of intermediate models generated and performance. We further extend the experimental analysis by considering problems with varying \( |S| \) and \( |\prec| \). In particular, we randomly generated 24 families of instances with 200 variables and 800 clauses, and with 500 variables and 1750 clauses: each family is characterized by a given \( |S| \) and \( |\prec| \), and consists of 100 problems. The analysis confirms our hypotheses, i.e., that \textsc{Sat&Pref} performs consistently better than \textsc{Optsat-HS} and \textsc{Optsat-BF} on difficult problems, characterized by a high number of \( |S| \) or \( |\prec| \). Overall, of the 2400 randomly generated instances with 500 variables, \textsc{Sat&Pref}, \textsc{Optsat-BF} and \textsc{Optsat-HS} solve 2342, 2167 and 1751 instances respectively.
The paper is structured as follows. In Section 2 we review the formalism for expressing preferences. Section 3 is dedicated to the presentation of the three algorithms OPTSAT-HS, OPTSAT-BF and SAT&PREF discussing the pros and cons of each one. Section 4 presents the experimental analysis we conducted. Section 5 ends the paper with the conclusions.

2 Satisfiability and Qualitative Preferences

Consider a finite set \( P \) of Boolean variables. A literal \( l \) is a variable \( x \) or its negation \( \overline{x} \). We assume \( \overline{\overline{x}} = x \); thus, \( \overline{\overline{x}} \) returns the literal with the same variable of \( l \) but opposite polarity. A clause is a finite disjunction of literals and a formula is a finite conjunction of clauses. As customary in SAT, we also represent clauses as sets of literals and formulas as sets of clauses, and we use \( \top \) and \( \bot \) to denote the empty set of clauses and the empty clause respectively. For example, assuming \( P = \{ \text{Pizza}, \text{Pasta}, \text{Soup} \} \), the formula

\[
\{\{\text{Pizza}, \text{Pasta}, \text{Soup}\}\}
\]

imposes that we cannot have Pizza, Pasta and Soup at the same time.

An assignment is a consistent set of literals, i.e. a set \( \mu \) of literals such that for each variable \( x \), \( x \notin \mu \) or \( \overline{x} \notin \mu \). If \( l \in \mu \), we say that both \( l \) and \( \overline{\overline{l}} \) are assigned by \( \mu \). An assignment \( \mu \) is total if each literal \( l \) is assigned by \( \mu \). A total assignment \( \mu \) satisfies a formula (or is a model of) \( \varphi \) if for each clause \( C \in \varphi \), \( C \cap \mu \neq \emptyset \).

A (qualitative) preference (on literals) is a (strictly) partially ordered set of literals, i.e., a pair \( \langle S, \prec \rangle \) where \( S \) is a set of literals (the set of preferences), and \( \prec \) is an irreflexive, transitive binary relation over \( S \). For example, the preference

\[
\langle \{\text{Pizza}, \text{Pasta}, \text{Soup}\}, \{\text{Pizza} \prec \text{Pasta} \prec \text{Soup}\}\rangle
\]

models the fact that we would like to have Pizza, Pasta and Soup, and that having Pizza is more important than having Pasta, and having Pasta is more important than having Soup. A qualitative preference \( \langle S, \prec \rangle \) on literals can be extended to the set of total assignments as follows: given two total assignments \( \mu \) and \( \mu' \), \( \mu \) is preferred to \( \mu' \) (\( \mu \prec \mu' \)) if and only if

1. there exists a literal \( l \in S \) with \( l \in \mu \) and \( \overline{l} \in \mu' \); and
2. for each literal \( l' \in S \cap (\mu' \setminus \mu) \), there exists a literal \( l \in S \cap (\mu \setminus \mu') \) such that \( l \prec l' \).

We recall that a binary relation \( \prec \) over a set \( S \) is

1. **irreflexive** if for each \( x \in S \), \( x \not\prec x \);  
2. **transitive** if for each \( x, y, z \in S \), if \( x \prec y \) and \( y \prec z \) then also \( x \prec z \);  
3. **total** if for each \( x, y \in S \), either \( x \prec y \) or \( y \prec x \).

Since \( \prec \) is transitive, we write \( a \prec b \prec c \) for \( a \prec b \) and \( b \prec c \).
\langle S, \prec \rangle := \text{a qualitative preference on literals;}
\psi := \top; \muopt := \emptyset

function \text{PREF-DLL}(\varphi) \quad \text{return} \quad \text{PREF-DLL-R}(\varphi, \emptyset)

function \text{PREF-DLL-R}(\varphi, \mu)
1 \quad \text{if} \quad (\bot \in (\varphi \cup \psi) \mu) \quad \text{return} \quad \text{FALSE};
2 \quad \text{if} \quad (\mu \text{ is total}) \quad \muopt := \mu; \psi := \text{Reason}(\mu, S, \prec); \quad \text{return} \quad \text{FALSE};
3 \quad \text{if} \quad (\{l\} \in (\varphi \cup \psi) \mu) \quad \text{return} \quad \text{PREF-DLL-R}(\varphi, \mu \cup \{l\});
4 \quad l := \text{ChooseLiteral}(\varphi \cup \psi, \mu);
5 \quad \text{return} \quad \text{PREF-DLL-R}(\varphi, \mu \cup \{l\}) \quad \text{or} \quad \text{PREF-DLL-R}(\varphi, \mu \cup \{\emptyset\}).

Fig. 1. The algorithm of \text{PREF-DLL}.

Considering the preference (2), we have that the set of total assignments is totally ordered, i.e.,

\{\text{Pizza, Pasta, Soup}\} \prec \{\text{Pizza, Pasta, Soup}\} \prec \{\text{Pizza, Pasta, Soup}\}
\prec \{\text{Pizza, Pasta, Soup}\} \prec \{\text{Pizza, Pasta, Soup}\} \prec \{\text{Pizza, Pasta, Soup}\}.

A model \mu of a formula \varphi is \textit{optimal} if it is a minimal element of the partially ordered set of models of \varphi, i.e., if it does not exist a model \mu' of \varphi with \mu' \prec \mu. The optimal model of (1) with respect to the preference (2) is \{\text{Pizza, Pasta, Soup}\}.

We recall that qualitative preferences on formulas can be reduced to qualitative preferences on literals (by introducing definitions for the formulas in the preferences, see [8]); this allows, for instance, to model both \textit{conditional} and \textit{unconditional qualitative} preferences. Moreover, since we can put both positive and negative literals in the set of preferences, it is also possible to model \textit{bipolar} preferences [3], i.e. expressing, at the same time, what we like and what we dislike, respectively. Finally, by encoding the objective function to maximize/minimize, it is also possible to model quantitative preferences, as shown in [7].

3 \quad \text{OPTSAT-HS, OPTSAT-BF and SAT&PREF}

3.1 \quad \text{A high level description}

Two procedures for computing an optimal model of a formula \varphi w.r.t. a preference \langle S, \prec \rangle have been presented in [7] and [11], called \text{OPTSAT-HS} and \text{OPTSAT-BF} respectively in [11]; both are based on the Davis-Logemann-Loveland procedure (DLL) and both can be seen as a specialization of the procedure in Figure 1. In the figure,

- \langle S, \prec \rangle \text{ is a qualitative preference on literals, } \psi \text{ and } \muopt \text{ are two global variables initialized to the empty set of clauses and the empty assignment, respectively;}
- (\varphi \cup \psi) \mu \text{ is the set of clauses obtained from } \varphi \cup \psi \text{ by (i) deleting the clauses } C \in \varphi \cup \psi \text{ with } \mu \cap C \neq \emptyset, \text{ and (ii) substituting the other clauses } C \in \varphi \cup \psi \text{ with } C \setminus \{T: l \in \mu\};
- **Reason**($\mu, S, \prec$) returns a set of clauses $\psi$ (corresponding to the preference formula reported in [11]) such that an assignment $\mu'$ satisfies $\psi$ if and only if $\mu' \prec \mu$ w.r.t. $(S, \prec)$. We recall that

$$\psi = (\forall l \in S, l \notin \mu) \land (\forall l', \mu'(\forall l \in S, l \notin \mu, l \prec l' \lor l'))$$

(3)

and imposes that the next model $\mu'$ of $\varphi$ (if it exists) has to be better than the last computed $\mu$ w.r.t. $(S, \prec)$: it thus follows that if $\mu$ is optimal, $\varphi \cup \psi$ does not have any model.

- **ChooseLiteral**($\varphi \cup \psi, \mu$) returns a literal in $\varphi \cup \psi$ which is unassigned by $\mu$.

- It is assumed the operator “or” is left associative, i.e., that at line (5), first

  $\text{PREF-DLL-R}(\varphi, \mu \cup \{l\})$ is executed and then $\text{PREF-DLL-R}(\varphi, \mu \cup \{l'\})$.

It is easy to see that $\text{PREF-DLL}$ is exactly the same as DLL, except that once a model $\mu$ is determined (see line 2),

1. $\mu$ is stored in $\mu_{\text{opt}}$;
2. $\text{Reason}(\mu, S, \prec)$ is stored in $\psi$; and
3. FALSE is returned.

Notice that once a model $\mu$ is found, search is not started from scratch, but it continues exploring the search tree looking for models of $\varphi$ that are better than $\mu$: if $\mu$ is an optimal model of $\varphi$, $\varphi \cup \text{Reason}(\mu, S, \prec)$ is unsatisfiable and no other model of $\varphi$ is generated. The procedure terminates, storing an optimal model of $\varphi$ in $\mu_{\text{opt}}$, assuming $\varphi$ is satisfiable, as sanctioned by the following theorem.

**Theorem 1** ([11]). For any formula $\varphi$, $\text{PREF-DLL}(\varphi)$ terminates, and then $\mu_{\text{opt}}$ is empty if $\varphi$ is unsatisfiable, and is an optimal model of $\varphi$ w.r.t. the qualitative preference $(S, \prec)$ otherwise.

Notice that the above Theorem holds for any particular implementation of $\text{ChooseLiteral}(\varphi \cup \psi, \mu)$, as long as such function returns a literal in $\varphi \cup \psi$ which is unassigned by $\mu$. Thus, the Theorem holds for $\text{OPTSAT-HS}$, $\text{OPTSAT-BF}$ and also (as we will see) for $\text{SAT\&PREF}$. Indeed, $\text{OPTSAT-HS}$ and $\text{OPTSAT-BF}$ differ for the implementation of $\text{ChooseLiteral}$:

1. In $\text{OPTSAT-HS}$, the returned literal $l$ has to satisfy the additional requirement that either $l \in S$ and each literal $l' \prec l$ has been already assigned or that all the literals in $S$ have been already assigned. With this restriction, the search space is explored according to the preference $(S, \prec)$: the first model $\mu$ of $\varphi$ that $\text{PREF-DLL}(\varphi)$ computes is guaranteed to be optimal and of course $\varphi \cup \text{Reason}(\mu, S, \prec)$ is unsatisfiable. However, it is well known that imposing an ordering on $\text{ChooseLiteral}$ may lead to exponential degradations in the performance [4, 9].

2. In $\text{OPTSAT-BF}$, $\text{ChooseLiteral}$ does not have any additional requirement: as a consequence, the first model $\mu$ of $\varphi$ that $\text{PREF-DLL}(\varphi)$ computes, is not guaranteed to be optimal, and several models of $\varphi$ can be computed before determining an optimal one. In general, we expect that the number of models possibly generated by $\text{OPTSAT-BF}$ increases with $|S|$ and $|\prec|$; when $S$ is totally ordered by $\prec$, $\text{OPTSAT-BF}$ can generate up to $2^{|S|}$ models.
⟨S, ≺⟩ := a qualitative preference on literals; ϕ is a set of static clauses;
ϕ′ := {}; µopt := ∅; µ := ∅; modelsCounter := 0; decisionLevel := 0;

function MAINLOOP()
1 while (TRUE)
2   while (propagateGivesConflict())
3      if (decisionLevel == 0 and modelsCounter == 0) return UNSAT;
4      else if (decisionLevel == 0 and modelsCounter > 0) µopt := µ; return OPT;
5      else analyzeConflict();
6      restartIfApplicable();
7      removeLemmasIfApplicable();
8      if (!decide(algo)) // ALL VARS ASSIGNED, I.E. MODEL µ FOUND.
9         modelsCounter := modelsCounter + 1;
10     if (algo == OPTSAT-HS) µopt := µ; return OPT;
11     if (algo == OPTSAT-BF or algo == SAT&PREF)
12        addPreferenceFormula(µ, S, ≺);
13     // GO TO PROPAGATE.

Fig. 2. The algorithm of the SAT&PREF system.

Given the drawbacks of OPTSAT-HS and OPTSAT-BF, in SAT&PREF we consider yet another variant of ChooseLiteral, obtained by switching the polarity of the returned literal l, in case l ∈ S and l̸∈ S (notice that the set S of preferences does not have to be consistent). For SAT&PREF, Theorem 1 still holds, and

1. Given that no ordering to ChooseLiteral is imposed, the negative results in [4, 9] do not apply to SAT&PREF, and
2. By switching the polarity of the returned literal l when l ∈ S and l̸∈ S, we ask the solver to not look for models that we know somehow deviate from the optimal ones.

Of course, it is still possible for SAT&PREF to go through all the possible models of ϕ before finding the optimal one as OPTSAT-BF, but we expect this not to be the case “in practice”. For instance, in the case ϕ is (1) and ⟨S, ≺⟩ is (2):

1. OPTSAT-BF may generate all the 7 models of (1) before determining that {Pizza, Pasta, Soup} is optimal, while
2. SAT&PREF can generate at most two intermediate models (i.e., {Pizza, Pasta, Soup} and {Pizza, Pasta, Soup}, obtained assuming ChooseLiteral first returns Soup and then Pasta) before the optimal one.

3.2 A low level description

The SAT&PREF system is implemented on top of the Conflict Driven Clause Learning (CDCL) solver MINISAT, with the preprocessor SatELite turned off. In Figure 2, we show the main loop of the SAT&PREF system, obtained by modifying MINISAT main loop:

- ⟨S, ≺⟩ is a qualitative preference on literals, ϕ is a set of static clauses (the input formula); both are global variables;
– \( \phi' \) is the set of learned clauses; \( \mu_{\text{opt}} \) stores the best model computed so far; \( \mu \) stores the current partial assignment; \( \text{modelsCounter} \) and \( \text{decisionLevel} \) are the number of models generated by OPTSAT-BF and SAT&PREF, and current decision level, respectively; all these variables are global and properly initialized in the preamble;
– \( \text{propagateGivesConflict()} \) performs the Boolean constraint propagation on the formula (\( \phi \) and \( \phi' \)), and returns TRUE whether a conflict is found, FALSE otherwise;
– \( \text{analyzeConflict()} \) backtracks up to the first Unique Implication Point (UIP) as MINISAT (see also [6]) adding the corresponding asserting clause to \( \phi' \);
– \( \text{restartIfApplicable()} \), if some conditions are satisfied, makes a restart, i.e. backtracks until decision level 0 and goes to line 2 (to propagate);
– \( \text{removeLemmasIfApplicable()} \), if some conditions are satisfied, removes some clauses from \( \phi' \), i.e. from the set of learned clauses;
– \( \text{decide(algo)} \) returns FALSE if all variables are assigned, i.e. a model of \( \phi \) is found; otherwise, adds the most preferred variable that has not been assigned yet to the partial assignment \( \mu \), in the case that \( \text{algo} \) is OPTSAT-HS\(^4\); it adds the variable with the highest score (obtained by MINISAT heuristic) to \( \mu \), in the case that \( \text{algo} \) is OPTSAT-BF or SAT&PREF. Finally, it returns TRUE;
– \( \text{addPreferenceFormula}(\mu, S, \prec) \) adds a set of clauses, called preference formula in [11], to the input formula \( \phi \), imposing that the next model has to be better than \( \mu \) with respect to \( (S, \prec) \). Notice that, since the preference formula is falsified under the current assignment, a backtracks is forced after the subsequent propagation.

MINISAT main loop can be obtained by removing line 4, lines from 9 to 13, and returning SAT in line 9. At the implementation level, apart from the \( \text{addPreferenceFormula} \) procedure (in line 12), the major modifications to MINISAT have been done in the heuristic (\( \text{decide(algo)} \) procedure in line 8), in order to implement the ideas underlying OPTSAT-HS and SAT&PREF. In MINISAT, during the search, the variables are maintained totally ordered according to some internal score: for branching, the variable in the highest position and not yet assigned is selected and then assigned with a predetermined polarity. Given the above, the heuristic modification for the SAT&PREF approach has been trivial: it fixes the polarity of a unassigned variable \( x \) (selected by the MINISAT heuristic) to true if \( x \in S \), to false if \( \neg x \in S \), and to the default polarity of MINISAT otherwise. Instead, it has not been trivial for OPTSAT-HS. Indeed, for OPTSAT-HS, if in the input preference \( (S, \prec) \), the partial order \( \prec \) is either empty or total, it is relatively easy to modify MINISAT heuristic in order to

1. first split on the literals in \( S \), choosing which one according to MINISAT score on the variables, in the case \( \prec \) is empty, or
2. force the split according to the preference despite MINISAT scoring, if \( \prec \) is total.

If \( \prec \) is neither empty nor total, the key issue is how to modify MINISAT internal ordering \( \prec_m \) (induced by the score)\(^5\) in order to be compatible with the ordering \( \prec \) of the

\(^4\) Score calculation of the most preferred variable will be shown later.
\(^5\) MINISAT orders the variables according to the score assigned to the variable, introducing also some randomization in the process. For simplicity, in our description we assume there is no randomization and that MINISAT ordering is entirely based on the score of the variables. In OPTSAT-HS, the heuristic does not have any random component.
preference. Intuitively, we want to obtain an order $\prec_{\text{m}}$ on the variables such that we split as soon as possible on the variables with highest score. To obtain such a $\prec_{\text{m}}$, we modified MINISAT scoring mechanism. In particular, if $sc(x)$ returns the original score assigned by MINISAT to variable $x$, the new score $nsc(x)$ associated to a variable $x$ such that either $x$ or $\overline{x}$ is in the preference, is

$$nsc(x) = \max\{sc(x), nsc(\overline{l_1}), \ldots, nsc(\overline{l_n})\} + \epsilon$$

where $\epsilon$ is an arbitrary small value, and $\{l_1, \ldots, l_n\}$ is the set $S_x$ of literals $l$ such that $x < l$ or $\overline{x} < l$. The new score can be effectively computed bottom up starting from the variables $x$ for which $S_x$ is empty whenever MINISAT updates the scores. For example, if we have 6 variables $x_1, \ldots, x_6$, the preference

$$\langle \{\overline{x_1}, x_2, x_3, x_4\}, \{x_1 < \overline{x_2}, \overline{x_1} < x_3\} \rangle,$$

and

$$sc(x_6) < sc(x_3) < sc(x_5) < sc(x_1) < sc(x_4) < sc(x_2)$$

we obtain

$$nsc(x_3) < nsc(x_4) < nsc(x_2) < nsc(x_1).$$

The ordering $\prec_{\text{m}}$ is then obtained by listing first the variables involved in the preferences sorted according to $nsc$, and then the others sorted according to $sc$: both sortings are done in decreasing order. In our example, we obtain

$$x_1 \prec_{\text{m}} x_2 \prec_{\text{m}} x_4 \prec_{\text{m}} x_3 \prec_{\text{m}} x_5 \prec_{\text{m}} x_6$$

Thus, OPTSAT-HS heuristic selects the unassigned variable $x$ occurring earlier in $\prec_{\text{m}}$, and fixes the polarity to true if $x \in S$, to false if $\overline{x} \in S$, and to the default polarity of MINISAT otherwise. The system is available on line for research purposes at the web site

http://www.star.dist.unige.it/~emanuele/sat&pref.

The web site also contains detailed instructions about how to use it. To the best of our knowledge there is no other implemented system dealing with satisfiability problems in the presence of partially ordered qualitative preferences.

4 Experimental analysis

Given a preference $\langle S, \prec \rangle$, from here on, we assume that the set of preferences $S$ is consistent: this is not a limitation because given a variable $x$ with both $x$ and $\overline{x}$ in $S$, we can uniformly replace, e.g., $x$ with a new variable $y$ in both $S$ and $\prec$, and then add the two clauses $\{\overline{x}, y\}$ and $\{x, \overline{y}\}$ to the input formula.
4.1 Evaluation on \textit{MIN-ONE}_\subseteq and \textit{MAX-SAT}_\subseteq problems

In order to evaluate the three systems, we started taking into account the min-one and maxsat benchmarks used to evaluate \textit{OPTSAT-HS} and \textit{OPTSAT-BF} in [7, 11], and then we extended the benchmark set with problems coming from the last maxsat evaluation.\footnote{See http://www.maxsat.udl.cat/09/}

Of course, our focus is on problems with qualitative preferences, and thus we consider only the qualitative version of such benchmarks, called (partial) \textit{MIN-ONE}_\subseteq and \textit{MAX-SAT}_\subseteq problems. In a \textit{MIN-ONE}_\subseteq (resp. \textit{MAX-SAT}_\subseteq) problem, a model \(\mu\) of a formula \(\varphi\) in the set \(P\) of variables is optimal if and only if there is no other model \(\mu'\) of \(\varphi\) with \(\mu' \cap P \subseteq \mu \cap P\) (resp. \(\{C : C \in \varphi, C \cap \mu \neq \emptyset\} \subset \{C : C \in \varphi, C \cap \mu' \neq \emptyset\}\)), and its partial version is defined analogously but w.r.t. a given subset of \(P\) (resp. \(\varphi\)).

Consider a formula \(\varphi\), \textit{MIN-ONE}_\subseteq problems can be directly encoded as preferences on literals: the set \(S\) of preferences is \(\{x \in P\}\), while \(\prec\) is empty. \textit{MAX-SAT}_\subseteq problems can also be encoded as preferences on literals: each clause \(C_i\in\varphi\) is substituted with \(\{x_i\}\cup C_i\), where \(x_i\) is a newly introduced variable called clause selector, and the set of preferences is fixed to consist of the newly introduced variables, while \(\prec\) is empty as for \textit{MIN-ONE}_\subseteq. The partial versions of \textit{MIN-ONE}_\subseteq and \textit{MAX-SAT}_\subseteq are mapped to qualitative preference on literals in an analogous way.

![Table 1](image)

Table 1. Results on \textit{MIN-ONE}_\subseteq and \textit{MAX-SAT}_\subseteq problems.

1. Each row represents a family of problems, the first two rows being families of \textit{MIN-ONE}_\subseteq problems from [11], while the other are \textit{MAX-SAT}_\subseteq problems from the
last maxsat evaluations (in particular, the problems in rows 1-11 are those used in [11]).
2. Column “#I” shows the number of problems in the family.
3. Columns “#SI” show the number of instances which are solved in less than 1800s (i.e., the timeout has been set to 1800s).
4. Columns “AT” show the average time in seconds taken by each system to solve the instances in the family.
5. Columns “AM” show, for OPTSAT-BF and SAT&PREF, the average number of models computed before terminating. For OPTSAT-HS such number is not reported because always equal to 1.

A bold number in a column “AT”, means that the other systems in the table either solved less instances, or took a longer time on average. Notice that—as in [11] and in the reports of the maxsat evaluations— columns “AT” (and also “AM”) does not consider the data of the instances on which the solver timed out.

Comparing SAT&PREF, OPTSAT-HS and OPTSAT-BF, the results show that the former performs better than the latter two: SAT&PREF solves 106 and 6 more instances than OPTSAT-HS and OPTSAT-BF, respectively. However, a closer look at OPTSAT-BF and SAT&PREF results shows that the two procedures have the same performance on MIN-One\(_C\) problems, while this is not true in MAX-Sat\(_C\) cases: in particular, on MAX-Sat\(_C\) families, OPTSAT-BF always generates significantly more intermediate models than SAT&PREF. This is explained by the fact that the systems are based on MINISAT, whose branching heuristic assigns variables to false: given this,

1. There are no differences between OPTSAT-BF and SAT&PREF on MIN-One\(_C\) problems given that the polarity of the literal returned by the heuristic of OPTSAT-BF and SAT&PREF is the same, and in agreement with the preferences, while
2. There can be significant differences between OPTSAT-BF and SAT&PREF on MAX-Sat\(_C\) problems given that if \(l = \pi\) is the literal returned by MINISAT heuristic, \(l\) is the literal assigned by OPTSAT-BF, while SAT&PREF would switch \(l\) polarity (thus assigning \(x\) to true) if \(x\) is a clause selector.

These results show the importance of assigning literals according to the preferences. To further evidentiate this fact, we set MINISAT (and thus OPTSAT-BF) heuristic in order to always return literals with positive polarity, and we call OPTSAT-BF\_POS the resulting procedure. In this way, on MAX-Sat\(_C\) problems, the polarity of the literal \(l\) returned by the heuristic of

1. OPTSAT-BF\_POS is always positive, while for
2. SAT&PREF is positive if \(l\) is a clause selector, and negative otherwise.

The reduction (resp. increase) in the number of iterations for OPTSAT-BF\_POS with respect to OPTSAT-BF on MAX-Sat\(_C\) (resp. MIN-One) problems shows the importance of switching the polarity in accordance with the preferences. Such a reduction corresponds to a reduction in the timings of OPTSAT-BF\_POS whose performance are now comparable to those of SAT&PREF on MAX-Sat\(_C\) problems: for MIN-One problems, we do not see an increase in OPTSAT-BF\_POS timings because of a single instance (named “psr31”) which is solved by OPTSAT-BF\_POS in 2.7s, but in 238.3s by OPTSAT-BF and
SAT&PREF. There are significant differences in the number of intermediate models generated by OPTSAT-BF, OPTSAT-BF_POS, and SAT&PREF for bcp-FIR instances, as shown in Figure 3. In this figure, the y-axis is in logarithmic scale and reports the number of models computed before finding an optimal solution, considering only instances where the optimal solution has been found within the timeout. In the case of the SAT&PREF approach, we can see that the first model computed is always optimal. This is surprising since using the SAT&PREF approach for solving Partial MAX-SAT⊆ problems does not guarantee that the first solution computed is optimal. Instead, the OPTSAT-BF approach needs the generation of a number of intermediate models that is four order of magnitude bigger than SAT&PREF, in the worst case; the OPTSAT-BF_POS version of OPTSAT-BF needs to generate two order of magnitude more models than SAT&PREF, in the worst case. Moreover, in the case of OPTSAT-BF, we can see that there are three points of the plot under the value 1: these points correspond to instances on which the system failed to find an optimal solution.

### 4.2 Evaluation on random instances

All the problems in Table 1 have an empty partial order. In order to test the procedures with \( \langle S, \prec \rangle \) of increasing size, and in order to get results which are not biased by the specific structure of the problems considered, we randomly generated 3CNF formulas, with randomly generated preferences. In details, we considered problems with 200 and 500 variables, and preferences \( \langle S, \prec \rangle \) in which
– $S$ is a consistent set of literals obtained by first randomly selecting 25%, 50%, 75%, 100% of the variables, and then negating each of them with probability 0.5.
– $\prec$ is the transitive closure of the DAG whose nodes are the literals in $S$ and with an arc between two nodes with probability $\rho = 0, T/2, T, 2T, 4T, 1$, where $T = \ln(|S|)/|S|$: fixing $\rho = 0$ will generate problems with an empty partial order, while increasing $\rho$ corresponds (on average) to increasing $|\prec|$ up to the point in which $\rho = 1$ and $|\prec| = 1/2|S|(|S| - 1)$, corresponding to a totally ordered set of preferences. As reported in [10], when $\rho > T$, it is likely that there exists a vertex connected by a path to every other vertex.

We generated 10 3-SAT instances with 200 variables/800 clauses and 500 variables/1750 clauses; then for each instance we randomly generated 10 preferences per each possible value of $S$ and $\prec$ according to the above methodology. As result we obtain 2400 instances for both 200 variables/800 clauses and 500 variables/1750 clauses; the numbers of variables/clauses have been selected in order to get different ratios clauses to variables. The results for OPTSAT-HS, OPTSAT-BF and SAT&PREF are shown in Figures 4 (200 variables) and 5 (500 variables), where on the $x$-axis we have 24 points, each corresponding to a pair $(|S|, \rho)$: the pairs are first ordered according to $|S|$ and then to $\rho$. Thus, the first point has $|S| = 25\%$ of the variables, and $\rho = 0$, the second point has again $|S| = 25\%$ of the variables but $\rho = T/2$, and analogously for the others. Thus, points 6, 12, 18, 24 have $\rho = 1$, i.e., they are the points corresponding to a totally ordered set of preferences. The $y$-axis is the average time each system takes on the 100 instances having the same $(|S|, \rho)$: when a system times out, we consider its time equal to the timeout (which for these experiments has been set to 600s). As it can be seen, SAT&PREF overall performs better than the others:

1. OPTSAT-HS has significantly worse performance than SAT&PREF when $0 < \rho < 1$, and is better of a factor when $\rho = 1$ and $n = 500$ (left plot): for $\rho = 4T$, OPTSAT-HS is not able to solve any single instance with either 500 variables or 200 variables and $|S| = 150, 200$.
2. OPTSAT-BF is slightly better than SAT&PREF if $\rho = 0, T/2, T$, but these problems are easily solved by SAT&PREF as well.
3. SAT&PREF has a clear edge with respect to OPTSAT-HS on average, and also with respect to OPTSAT-BF especially when $\rho = 1$.

A detailed comparison of OPTSAT-BF and SAT&PREF is given in Table 2. As it can be seen, SAT&PREF, on average, never generates more intermediate models and always solves more instances than OPTSAT-BF, for a fixed $|S|$ and $\rho$. Columns “AMwTO” report the average of the number of models generated by the system, considering also the instances not solved by the system, while Columns “ATwTO” show the average time to handle the instances, considering also the timeouts. Columns “$k \times <<$” (resp. “$> >$”) give the number of instances solved by SAT&PREF (resp. OPTSAT-BF) in at least 2s and on which OPTSAT-BF (resp. SAT&PREF) is $k$ times faster. Column “<<<” (resp. “>>”) gives the number of instances solved by OPTSAT-BF (resp. SAT&PREF) and not by SAT&PREF (resp. OPTSAT-BF). Column “<<<” gives the number of instances which are not solved by both systems.

The results indicate that for $\rho = 0, T/2$, OPTSAT-BF is slightly better than SAT&PREF, but these are problems which are solved in less than 1s on average, as it can be seen from
When $\rho > T$, **SAT&PREF** is faster and/or solves more problems than **OPTSAT-BF**. Even more,

1. If we look at the rows having $\rho = 2T, 4T$, we see that the gap between **SAT&PREF** and **OPTSAT-BF** increases with $|S|$, despite the fact that —for $\rho = 2T, 4T$— the actual value of $\rho$ decreases with $|S|$, going, e.g., from 7.7 for $|S| = 125$ down to 2.5 for $|S| = 500$.

2. If we consider the results of **SAT&PREF** and **OPTSAT-BF** for a fixed $|S|$, we see that **SAT&PREF** performs better and better as $\rho$ increases.

Overall, **SAT&PREF** is better for solving “hard” problems with an high number of preferences, especially in the presence of a significant order between them: for $|S| = 500$ and $\rho = 1$ the gap between **SAT&PREF** and **OPTSAT-BF** is maximal: **SAT&PREF** and **OPTSAT-BF** solve 79 and 6 out of the 100 instances, respectively.

Moreover, we analyzed the average number of intermediate models generated by **SAT&PREF** and **OPTSAT-BF** on random benchmarks, for both 200 and 500 variables, and the results are showed in Figure 6. We considered the logarithmic scale for the

### Table 2. **OPTSAT-BF** and **SAT&PREF** on random problems with 500 variables and 1750 clauses and random preferences.

| $|S|$ | $\rho \times 100$ | **OPTSAT-BF** | **SAT&PREF** | **OPTSAT-BF vs SAT&PREF** |
|------|------------------|----------------|--------------|---------------------------|
|      |                  | $\# S$ | AMwTO | TwTO | $\# S$ | AMwTO | TwTO | 2×<10×<100×<1<10×<100×2×>10×>>> |
| 125  | 0                | 100   | 17.0  | 0.1  | 100   | 7.7  | 0.3  | 3  1  0  0  0  0  0                 |
| 125T/2 = 1.9 | 100   | 23.7  | 0.4  | 1     | 100   | 9.0  | 0.9  | 6  4  0  2  1  0                 |
| 125  | $T = 3.9$        | 100   | 47.8  | 1.6  | 100   | 15.3 | 1.3  | 12 3 0 14 6 0                 |
| 125  | $T = 7.7$        | 100   | 139.8 | 7.2  | 100   | 35.5 | 4.5  | 21 7 0 27 7 0                 |
| 125  | $T = 15.8$       | 98    | 407.9 | 46.4 | 100   | 76.9 | 26.9 | 18 4 0 39 15 2                 |
| 125  | 100              | 76    | 1479.8 | 242.3 | 93    | 215.4 | 136.2 | 12 3 2 37 11 19 5             |
| 250  | 0                | 100   | 21.2  | 0.1  | 100   | 5.3  | 0.4  | 4  4 0 0 0 0                 |
| 250T/2 = 1.1 | 100   | 24.3  | 0.1  | 100   | 5.9  | 0.3  | 4  2 0 0 0 0                 |
| 250  | $T = 2.2$        | 100   | 58.1  | 0.5  | 100   | 10.0 | 0.7  | 8  3 0 2 2 0                 |
| 250  | $T = 4.4$        | 100   | 219.8 | 4.1  | 100   | 26.5 | 3.2  | 15 0 0 29 6 0                 |
| 250  | $T = 8.8$        | 99    | 767.2 | 36.7 | 100   | 69.2 | 7.8  | 9  0 0 63 18 1                 |
| 250  | 100              | 63    | 3222.0 | 375.4 | 84    | 399.0 | 177.2 | 5  1 4 46 19 12              |
| 375  | 0                | 100   | 25.2  | 0.1  | 100   | 2.9  | 0.1  | 0  0 0 0 0 0                 |
| 375T/2 = 0.8 | 100   | 30.3  | 0.1  | 100   | 3.1  | 0.3  | 4  4 0 0 0 0                 |
| 375  | $T = 1.6$        | 100   | 72.7  | 0.6  | 100   | 6.2  | 0.9  | 2  2 0 1 1 0                 |
| 375  | $T = 3.2$        | 100   | 282.8 | 8.3  | 100   | 17.4 | 3.4  | 3  0 0 29 4 0                 |
| 375  | $T = 6.3$        | 100   | 1314.3 | 57.0 | 100   | 57.4 | 11.7 | 0  0 0 67 16 0                 |
| 375  | 100              | 27    | 7061.9 | 520.5 | 86    | 544.6 | 190.8 | 0  0 1 20 11 60 13            |
| 500  | 0                | 100   | 29.1  | 0.1  | 100   | 1.0  | 0.1  | 1  1 0 0 0 0                 |
| 500T/2 = 0.6 | 100   | 35.1  | 0.1  | 100   | 1.2  | 0.2  | 0  0 0 0 0 0                 |
| 500  | $T = 1.2$        | 100   | 74.6  | 0.9  | 100   | 2.5  | 0.8  | 1  0 0 2 0 0                 |
| 500  | $T = 2.5$        | 100   | 387.8 | 12.8 | 100   | 10.4 | 8.9  | 0  0 0 10 1 0                 |
| 500  | $T = 5.0$        | 98    | 1492.3 | 85.0 | 100   | 36.0 | 27.2 | 0  0 0 43 2 2                 |
| 500  | 100              | 6     | 5824.3 | 580.1 | 79    | 534.5 | 261.6 | 1 0 1 3 0 74 20              |
| Total |                  | 2167  | 1044.1 | 82.5 | 2342  | 87.2 | 36.1 | 129 39 8 434 120 883 50       |

In summary, if $\rho > T$, **SAT&PREF** is faster and/or solves more problems than **OPTSAT-BF**. Even more:

1. If we look at the rows having $\rho = 2T, 4T$, we see that the gap between **SAT&PREF** and **OPTSAT-BF** increases with $|S|$, despite the fact that —for $\rho = 2T, 4T$— the actual value of $\rho$ decreases with $|S|$, going, e.g., from 7.7 for $|S| = 125$ down to 2.5 for $|S| = 500$.

2. If we consider the results of **SAT&PREF** and **OPTSAT-BF** for a fixed $|S|$, we see that **SAT&PREF** performs better and better as $\rho$ increases.

Overall, **SAT&PREF** is better for solving “hard” problems with an high number of preferences, especially in the presence of a significant order between them: for $|S| = 500$ and $\rho = 1$ the gap between **SAT&PREF** and **OPTSAT-BF** is maximal: **SAT&PREF** and **OPTSAT-BF** solve 79 and 6 out of the 100 instances, respectively.

Moreover, we analyzed the average number of intermediate models generated by **SAT&PREF** and **OPTSAT-BF** on random benchmarks, for both 200 and 500 variables, and the results are showed in Figure 6. We considered the logarithmic scale for the
Fig. 4. OPTSAT-HS, OPTSAT-BF and SAT&PREF performance on randomly generated problems with 200 variables and 800 clauses. The $y$-axis of the plot has been cut to 10s in order to evidence the differences between OPTSAT-BF and SAT&PREF.

$y$-axis and 24 points for the $x$-axis, each corresponding to a pair $(|S|, \rho)$ as in Figures 4 and 5. In the calculation of the average number of intermediate models plotted in Figure 6, we considered instances where the optimal model has been found within the timeout; this choice has been made to obtain different values respect to those that are reported in table 2. In fact, column AMwTO in table 2 takes even into account the number of models generated when an optimal solution has not been found within the timeout, by introducing noise in the calculation of intermediate models necessary to obtain an optimal model. We can see that, also in this kind of calculation, the SAT&PREF approach computes much less intermediate models than OPTSAT-BF in both plots. It is also interesting to highlight that in the right plot in Figure 6, i.e. in the case of 500 variables, there is a decreasing behaviour for instances corresponding to points 1, 7, 13, and 19, i.e. where there is an empty partial order; instead, there is an increasing behaviour for points 6, 12, 18, and 24, i.e. where preferences are totally ordered.

5 Conclusions

We presented a new approach for solving satisfiability problems with preferences, called SAT&PREF, which combines the ideas underlying OPTSAT-HS and OPTSAT-BF, but limiting their drawbacks: OPTSAT-HS imposes a predetermined (partial) order on the branching heuristic, determining a potential dramatic impact on the performance; OPTSAT-BF may require the generation of (exponentially) many non-optimal intermediate solutions. Instead, SAT&PREF is a combination of the previous approaches, where the
branching heuristic is modified —as in OPTSAT-HS— by changing the polarity of the returned literal according to the preference, and then the search continues —as in OPTSAT-BF— looking for better solutions.

We conducted an extensive experimental evaluation of the three approaches, significantly extending the previous ones, which only considered problems with empty partial order. The results show that SAT&PREF, on average performs better than OPTSAT-HS and OPTSAT-BF, especially on the “hard” problems, having many, possibly partially ordered, preferences.

References

Fig. 6. Comparing the average number of intermediate models generated by OPTSAT-BF and SAT&PREF to find an optimal model on randomly generated problems with both 200 variables (left) and 500 variables (right). The y-axis is in logarithmic scale.

A local search approach to solve incomplete fuzzy and weighted CSPs

Mirco Gelain, Maria Silvia Pini, Francesca Rossi, K. Brent Venable, and Toby Walsh

1 Dipartimento di Matematica Pura ed Applicata, Università di Padova, Italy
E-mail: {mgelain,mpini,frossi,kvenable}@math.unipd.it

2 NICTA and UNSW Sydney, Australia,
Email: Toby.Walsh@nicta.com.au

Abstract. We consider soft constraint problems where some of the preferences may be unspecified. This models, for example, settings where agents are distributed and have privacy issues, or where there is an ongoing preference elicitation process. In this context, we study how to find an optimal solution without having to wait for all the preferences. In particular, we define local search algorithms that interleave search and preference elicitation, with the goal to find a solution which is "necessarily optimal", that is, optimal no matter what the missing data are, while asking the user to reveal as few preferences as possible. While in the past this problem has been tackled with a branch & bound approach, which was guaranteed to find a solution with this property, we now want to see whether a local search approach can solve such problems optimally, or obtain a good quality solution, with fewer resources. At each step, our local search algorithm moves from the current solution to a new one, which differs in the value of a variable. The variable to reassign and its new value are chosen so to maximize the quality of the next solution. To compute this, we elicit some of the preferences missing in the neighbor solutions. Experimental results on randomly generated fuzzy and weighted CSPs with missing preferences show that our local search approach is promising, both in terms of percentage of elicited preferences and its scaling properties.

1 Introduction

Constraint programming [3] is a powerful paradigm for solving scheduling, planning, and resource allocation problems. A problem is represented by a set of variables, each with a domain of values, and a set of constraints. A solution is an assignment of values to the variables which satisfies all constraints and which optionally maximizes/minimizes an objective function. Soft constraints [9] are a way to model optimization problems by allowing for several levels of satisfiability, modeled by the use of preference or cost values that represent how much we like a certain way to instantiate the variables of a constraint. For instance, if we prefer a desktop PC with 8GB of RAM rather than a laptop with 4GB, we will assign a higher preference value to the former than the latter. Incomplete soft
constraint problems (ISCSPs) [6, 4, 5] can model situations in which data are not completely known before solving starts by allowing some of the preferences to be missing. Thus ISCSPs can model a form of uncertainty which is captured by the absence of some preferences in the constraints. In this scenario, the notion of optimal solution is replaced by other notions, among which the most attractive is that of a "necessarily optimal solution". They are those solutions which are optimal no matter what the missing preference values will turn out to be. These are solutions which are completely robust with respect to the uncertainty of the problem.

If an ISCSP has some variable assignments with this property, it is ideal to return one of them. However, the set of necessarily optimal solutions of an ISCSP may be empty. In this case, we may reduce the uncertainty of the problem by asking the user to reveal some of the missing preferences, and then see whether the new constraint problem has some necessarily optimal solutions. In fact, if some preferences are revealed, the new set of necessarily optimal solutions may be larger than the one we had before.

In [6] a complete basic branch & bound approach (without sophisticated propagations techniques) is used to find a necessarily optimal solution of an ISCSP, by interleaving search and elicitation. The approach was tested over both fuzzy and weighted ISCSPs, and several different elicitation strategies were considered. The experimental results showed that some of the concrete algorithms based on that approach could find a necessarily optimal solution while eliciting a small percentage of the missing preferences.

In this paper we intend to see whether ISCSPs can be better solved by a local search approach. As for the branch & bound approach, we interleave local search steps and elicitation. In this case, elicitation is used to guide the future steps. More precisely, our algorithm starts from a randomly chosen assignment to all the variables and elicits enough of the missing preferences related to this assignment to compute its preference. We then move to a new assignment which differs for the value of one variable. The variable to re-instantiate, and the new value to assign to it, are chosen as in [2], except that we compute preference values as if all the missing preferences were uninfluential. During the search, the algorithm maintains the best solution found so far. Random moves and Tabu Search are also used to avoid stagnation in local minima. The algorithm stops when a given limit on the number of steps is reached.

This basic local search approach, although not being very sophisticated, shows a very good behavior under several aspects. In our experimental tests, performed over randomly generated fuzzy and weighted ISCSPs, and considering several elicitation strategies, we measured the runtime and the percentage of elicited preferences, and we compared both of them to those of the best complete algorithm based on branch & bound.

When the number of variables is small the complete algorithm is better in terms of elicited preferences. However, the amount of elicitation needed by the

\footnote{The fuzzy and weighted frameworks are the most used but other frameworks can be easily used by changing the underlying c-semiring.}
local search approach is not much higher, and always less than 25% of the total number of missing preferences for fuzzy ISCSPs. In terms of runtime, the local search approach is always much better, even with small problems. In fact, the local search approach can handle problems of up to 100 variables by using the same time the complete algorithm needs to solve problems with only 25 variables.

We also measured the quality of the solutions returned by the local search approach, as the distance from the quality of the preference of the solution returned by the complete algorithm based on branch & bound. In this respect, the local search approach is very good, since such a distance appears lower than 2% in all considered cases.

By looking at the runtime behaviour of the local search algorithm, we also noticed that all the measured properties (that is, number of elicited preference values and solution quality) are achieved in the first steps of the search. Thus, we could safely stop the algorithm much earlier than the step limit, while obtaining solutions with the same quality and eliciting the same amount of missing preferences. This means that our local search algorithm can actually handle much larger instances than those considered in the experiments.

2 Background

In this section we give the basic notions about incomplete soft constraint problems and local search.

2.1 Incomplete soft constraint problems

Incomplete Soft Constraints Problems (ISCSPs) [6] extend Soft Constraint Problems (SCSPs) [1] to deal with partial information. We will focus on two specific instances of this framework, in which the soft constraints are fuzzy and weighted. In the fuzzy case, preference values are between 0 and 1, the preference of a solution is the minimum preference value contributed by the constraints, and the optimal solutions are those with the highest value. On the other hand, in weighted CSPs, preference values are interpreted as costs and range is between 0 and $+\infty$, the preference of a solution is the sum of all the costs contributed by the constraints, and the optimal solutions are those with lowest cost.

More formally, given a set of variables $V$ with finite domain $D$, an incomplete soft constraint is a pair $(idef, con)$ where $con \subseteq V$ is the scope of the constraint and $idef : D^{\text{con}} \rightarrow (A \cup \{?\})$ is the preference function of the constraint associating to each tuple of assignments to the variables in $con$ either a preference value that belongs to the set $A$, or ?. $A$ is the set of preference values, which is $[0, 1]$ for fuzzy constraints, and $[0, +\infty]$ for weighted constraints. All tuples mapped into ? by $idef$ are called incomplete tuples, meaning that their preference is unspecified.

An Incomplete Fuzzy (resp., Weighted) Constraint Problem (IFCSP, resp, IWCSP) is a triple $(C, V, D)$ where $C$ is a set of incomplete fuzzy (resp., weighted) constraints over variables $V$ with domain $D$. 
Given an assignment $s$ to all the variables of an IFCSP or an IWCSP $P$, $\text{pref}(P, s)$ is the preference of $s$ in $P$. More precisely, it is defined as $\text{pref}(P, s) = \ominus \min_{\langle \text{idef}, \text{con} \rangle \in C | \text{idef}(s_{\downarrow \text{con}}) \neq \text{idef}(s_{\downarrow \text{con}}')}$, where $\ominus$ is the minimum for fuzzy constraints and the sum for weighted constraints. It is obtained by taking the minimum or the sum of the known preferences associated to the projections of the assignment, that is, of the appropriated sub-tuples in the constraints.

**Example 1.** Figure 1 shows an example of an IFCSP with three variables $A$, $B$, and $C$, with domains $D(A) = \{a, b, c\}$, $D(B) = \{d, e\}$, and $D(C) = \{r, s\}$. The presence of the question marks identifies the missing preference values. In this example, the preference of the variable assignment $\langle A = a, B = e, C = r \rangle$ is $\min(0.3, 0.4, 1, 0.9) = 0.3$. Since this variable assignment involves some missing preferences, such as the one for the tuple $\langle A = a, B = e \rangle$, its preference should be interpreted as an upper bound of the actual preference for this assignment.

![Fig. 1. An example of ISCSP](image)

When all preferences are specified, a complete assignment of values to all the variables is an optimal solution if its preference is the best one (that is, maximal for fuzzy constraints and minimal for weighted constraints). This optimality notion is generalized to ISCSPs through the notion of necessarily optimal solutions, that is, complete assignments which are optimal no matter the value of the unknown preferences. Of course, these are the solutions which are most appealing, since they are completely robust w.r.t. the missing data. However, the set of necessarily optimal solutions of an ISCSP may be empty.

In Example 1, the assignment $\langle A = a, B = e, C = r \rangle$ is not necessarily optimal. In fact, if all missing preferences are 1, this assignment has preference 0.3, while the assignment $\langle A = b, B = d, C = r \rangle$ has preference 0.6. In this example, there are no necessarily optimal solutions. Consider now the same example where the preferences of both $A = b$ and $A = c$ are set to 0.2. In this new IFCSP, the assignment $\langle A = a, B = d, C = s \rangle$ has preference 0.3 and is
necessarily optimal. In fact, whatever values are given to the missing preferences, all other assignments have preference at most 0.2 (if \( A = b \) or \( c \)) or 0.3 (if \( A = a \)).

In [6] several algorithms are proposed to find a necessarily optimal solution of an IFCSP or an IWCSP. All these algorithms follow a branch and bound schema where search is interleaved with elicitation. Elicitation is needed since the given problem may have an empty set of necessarily optimal solutions. By eliciting more preferences, this set eventually becomes non-empty. Several elicitation strategies are considered in [6] in the attempt to elicit as little as possible before finding a necessarily optimal solution.

2.2 Local search

Local search [8] is one of the fundamental paradigms for solving computationally hard combinatorial problems. Local search can also naturally be used to solve optimization problems. Given a problem instance, the basic idea underlying local search is to start from an initial search position in the space of all possible assignments (typically a randomly or heuristically generated assignment, which may be infeasible, sub-optimal or incomplete), and to improve iteratively this assignment by means of minor modifications. At each search step we move to a new assignment selected from a local neighborhood, chosen via a heuristic evaluation function. This process is iterated until a termination criterion is satisfied.

The termination criterion is usually the fact that a solution is found or that a predetermined number of steps is reached. To ensure that the search process does not stagnate, most local search methods make use of random moves: at every step, with a certain probability a random move is performed rather than the usual move to the best neighbor. Another way to prevent local search from locking in a local minima is Tabu search [7] that uses a short term memory to prevent the search from returning to recently visited assignments for a specified amount of steps.

A local search approach has been defined in [2] to find an optimal solution in a soft constraint problem. This approach starts from a randomly chosen assignment to all the variables, say \( s \), and at each step it moves to a new assignment which is obtained by changing the value of one variable. Such a variable is one of those whose local preference is minimal in \( s \). The local preference of a variable in an assignment \( s \) is the combination of the preferences identified by \( s \) in all constraints involving the variable.

Consider for example the problem in Figure 1 where all missing preferences are set to 1. This is a classical fuzzy constraint problem. Consider now the assignment \( (A = a, B = d, C = r) \). In this assignment, the local preference of variable A is \( \min(0.3, 0.4, 0.9) = 0.3 \), while it is 0.4 for B and 0.9 for C. Thus, variable A would be chosen by the local search algorithm.

Once the variable, say \( x \), is chosen, its new value is identified by computing, for each new value \( v \), the preference of the new assignment (that is, \( s \) with the new value \( v \) for \( x \)). The value which gives the best preference for the new assignment is chosen.
Consider again the example in Figure 1 where all missing preferences are set to 1, and assignment \( \langle A = a, B = d, C = r \rangle \). This assignment has preference 0.3. Variable \( A \) can be changed to values \( b \) or \( c \). With \( A = b \), the new assignment has preference 0.6, while with \( A = c \), the new assignment has preference 0.5. Thus the algorithm would move to the assignment \( \langle A = b, B = d, C = r \rangle \) what has preference 0.6.

In the following we adapt this algorithm to deal with incompleteness.

3 Local search on ISCSPs

We will now present our local search algorithm for ISCSPs that interleaves elicitation with search. We basically follow the same algorithm as in [2], except for the following.

To start, we randomly generate an assignment of all the variables. To assess the quality of such an assignment, we compute its preference. However, since some missing preferences may be involved in the chosen assignment, we ask the user to reveal them.

In each step, when a variable is chosen, its local preference is computed by setting all the missing preferences to the best preference value (which is 1 for fuzzy constraints and 0 for weighted constraints). In other words, if there are missing preferences, they are not considered in computing the local preference of a variable in a given assignment (since the best value is the neutral element for the combination).

Consider again the example in Figure 1 and the assignment \( \langle A = a, B = e, C = r \rangle \). In this assignment, the local preference of variable \( A \) is \( \min(0.3, 0.4, 0.9) = 0.3 \), while for \( B \) is 0.4, and for \( C \) is \( \min(0.9, 1) = 0.9 \). Thus our algorithm would choose variable \( A \).

To choose the new value for the selected variable, we compute the preferences of the assignments obtained by choosing the other values for this variable. Since some preference values may be missing, in computing the preference of a new assignment we just consider the preferences which are known at the current point. We then choose the value which is associated to the best new assignment. If two values are associated to assignments with the same preference, we choose the one associated to the assignment with the smaller number of incomplete tuples. In this way, we aim at moving to a new assignment which is better than the current one and has the fewest missing preferences.

In the running example above, from assignment \( \langle A = a, B = e, C = r \rangle \), once we know that variable \( A \) will be changed, we compute \( \text{pref}(\langle A = b, B = e, C = r \rangle) = 0.4 \) and \( \text{pref}(\langle A = c, B = e, C = r \rangle) = 0.3 \). Thus we would select the value \( b \) for \( A \).

Since the new assignment, say \( s' \), could have incomplete tuples, we ask the user to reveal enough of this data to compute the actual preference of \( s' \). Of course, asking for all the missing preferences is always a correct strategy; we call ALL the elicitation strategy that elicits all the missing preferences associated to the tuples obtained projecting \( s' \) on the constraints. However, depending on
the class of soft constraints considered, asking for less than all the preferences could be sufficient. For fuzzy constraints, we also consider an elicitation strategy, called WORST, that asks the user to reveal only the worst preference among the missing ones, if it is less than the worst known preference. This is enough to compute the actual preference of \( s' \) since the preference of an assignment coincides with the worst preference in its constraints.

For weighted constraints, we consider the following three strategies (besides ALL):

- WW: we elicit the worst missing cost (that is, the highest) until either all the costs are elicited or the current global cost of the assignment is higher than the preference of the best assignment found so far;
- BB: we elicit the best (i.e., the minimum) cost with the same stopping condition as for WW;
- BW: we elicit the best and the worst cost in turn, with the same stopping condition as for WW.

As in many classical local search algorithms, to avoid stagnation in local minima, we employ tabu search and random moves. Our algorithm has two parameters: \( p \), which is the probability of a random move, and \( t \), which is the tabu tenure. When we have to choose a variable to re-assign, the variable is either randomly chosen with probability \( p \) or, with probability \( (1-p) \), we perform the procedure described above. Also, if no improving move is possible, i.e., all new assignments in the neighborhood are worse than or equal to the current one, then the chosen variable is marked as tabu and not used for \( t \) steps.

While in classical local search scenarios the underlying problem is always the same, and we just move from one of its solutions to another one, in our scenario we also change the problem via the elicitation strategies. Since the change involves only the preference values, the solution set remains the same, although the preferences of the solutions may decrease over time.

During search, the algorithm maintains the best solution found so far, which is returned when the maximum number of allowed steps is exceeded. In the ideal case, the returned solution is a necessarily optimal solution of the initial problem with the preferences added by the elicitation. However, there is no guarantee that this is so: via elicitation we can reach a problem with necessarily optimal solutions, but the algorithm may fail to find one of those. However, we will show later that, even in this case, the quality of the returned solutions is not very far from that of the necessarily optimal solutions.

4 Experimental settings and results

4.1 Problem generator

The test sets for fuzzy and weighted incomplete CSPs are created using a generator that has the following parameters:

- \( n \): number of variables;
- \( m \): cardinality of the variable domains;
- \( d \): density, that is, the percentage of binary constraints present in the problem w.r.t. the total number of possible binary constraints that can be defined on \( n \) variables (\( n(n - 1)/2 \));
- \( t \): tightness, that is, the percentage of tuples with the worst preference (that is, 0 for the fuzzy constraints and \( +\infty \) for the weighted constraints) in each constraint and in each domain w.r.t. the total number of tuples (\( m^2 \) for the constraints, since we have only binary constraints, and \( m \) in the domains);
- \( i \): incompleteness, that is, the percentage of incomplete tuples (that is, tuples with preference ?) in each constraint and in each domain.

Given values for these parameters, we generate ISCSPs as follows. We first generate \( n \) variables and \( d\% \) of the \( n(n - 1)/2 \) possible constraints. Then, for every domain and for every fuzzy (resp., weighted) constraint, we generate a non-zero (resp., non-\( +\infty \)) random preference value for each of the tuples (that are \( m \) for the domains, and \( m^2 \) for the constraints); we randomly set \( t\% \) of these preferences to 0 (resp., \( +\infty \)); we randomly set \( i\% \) of the preferences in each constraint and domain as incomplete. Preference values range in \([0, 1]\) (with 101 possible different values) for IFCSPs and in \([0, 10] \cup +\infty \) (with 12 possible different values) for IWCSPs.

Our experiments measure the percentage of elicited preferences (over all the missing preferences), the solution quality (as the normalized distance from the quality of necessarily optimal solutions), and the execution time, as the generation parameters vary.

We executed our algorithm using a step limit of 100000, a random walk probability of 0.2 and tabu tenure of 1000. All results are an average over 100 problem instances.

### 4.2 Incomplete fuzzy CSPs

We tested the performance of our algorithm using both the ALL and the WORST elicitation strategy. We also compared the result with one of the best algorithms in [6], called here FBB (which stands for fuzzy branch and bound). In [6], this algorithm corresponds to the one called DPI.WORST.BRANCH.

We first considered the quality of the returned solution. To do this, we computed the distance between the preference of the returned solution and that of the necessarily optimal solution returned by algorithm FBB. Such a distance is measured as the percentage over the whole range of preference values. For example, if the preference of the solution returned is 0.4 and the one of the solution given by FBB is 0.5, the preference error reported is 10%. A higher error denotes a lower solution quality.

Figure 2 shows the preference error when density, incompleteness, tightness, and the number of variables vary (please notice that the y-axis ranges from 0% to 10%). We can see that the error is always very small and its maximum value is 3.5% when we consider problems with 20 variables. In most of the other cases,
it is below 1.5%. We also can notice that the solution quality is practically the same for both elicitation strategies.

If we look at the percentage of elicited tuples (Figure 3), we can see that the WORST strategy elicits always less tuples than ALL, eliciting only 20% of incomplete tuples in most of the cases. When tightness is above 40%, WORST elicits very few tuples since the algorithm discovers soon that there are no solutions. The FBB algorithm elicits about half as many preferences as WORST. Thus, with 10 variables, FBB is better than our local search approach, since it guarantees to find a necessarily optimal solution while eliciting a smaller number of preferences.

We also tested the WORST strategies varying the number of variables from 10 to 100. In Figure 4(a) we show how the elicitation varies up to 100 variables. It is easy to notice that with more than 70 variables the percentage of elicited tuples decreases. This is because the probability of a complete assignment with a 0 preference arises (since density remains the same). Moreover, we can see how the local search algorithms can scale better than the branch and bound approach. In Figure 4(b) the FBB reaches a time limit of 10 minutes with just 25 variables, while the WORST algorithm needs the same time to solve instances of size 100.

We also investigated the runtime behavior of our local search algorithm. Surprisingly, the algorithm elicits almost all the tuples it needs within the first 10000 steps. Moreover the distance from the necessarily optimal preference decreases
Fig. 3. Percentage of elicited tuples. When a parameter is fixed, its value is: $n=10$, $m=5$, $d=50\%$, $i=30\%$, $t=10\%$.

Fig. 4. Values of the fixed parameters: $m=10$, $d=35\%$, $i=30\%$, $t=5\%$. 
significantly during the first 20000 steps and then it decreases slightly until the end of the search. This behavior is the same no matter which parameter is varying. Hence we can stop our algorithm after 20-30000 steps whilst still ensuring a good solution quality.

4.3 Incomplete weighted CSPs

We tested the performance of our algorithm using the ALL, BB, WW and BW elicitation strategies. We also compared the result with one of the best algorithms in [6] for solving incomplete weighted CSPs, which we call WBB here (in [6] it was called DPI.BW.TREE). Figure 6 shows the solution quality, in terms of the preference error, which is measured similarly to the fuzzy context but with a specific treatment to deal with $+\infty$. More precisely, the error is the percentage difference between the solution preference found by the local search algorithm and the one found by WBB over the preference range. Notice that, with costs in $[0, 10] \cup +\infty$, solutions have preferences that may be between 0 and $((n \times (n - 1))/2) + n \times 10$, or $+\infty$. When the returned preference is $+\infty$ and the correct preference is different, we report an error of 100%. For example, with 10 variables and preferences in $[1, 10] \cup +\infty$, if the local search algorithm returns a solution with preference 120 and the necessarily optimal preference is 100, then the error is $(120-100)*100/550 = 3.64\%$.

In most cases, all the local search algorithms find a solution with a preference very close to the necessarily optimal one. The peaks at 17 variables in Figure 6(a), at $d = 80\%$ in Figure 6(b), and at $t = 40\%$ in Figure 6(d) show a phase transition where the number of solutions with infinite cost increase significantly. This affects the quality of the solutions found by local search, while the branch and bound approach is able to find an optimal solution with a finite cost.

As in the fuzzy case, we measured the percentage of elicited tuples. As before, the local search approach elicits more preferences than the branch and bound based algorithms in [6]. However, the difference is fairly small and independent of the amount of incompleteness (see Figure 7(c)). Moreover, it is small also with density is below 60\%, or when tightness is less than 30\% or greater than 65\% (Figure 7(b)). Among the local search algorithms, as expected, the algorithms BB, WW, and BW elicit less preferences than ALL.

To study the runtime behavior of our local search approach, we focused on algorithm BB, which is one of the best algorithms. Figure 8 shows how the percentage of elicited tuples and the distance from the necessarily optimal preference vary as the execution proceeds. From Figures 8(a), 8(c), 8(e) and 8(g) we can see that the elicitation process takes place mainly in the first 10000 steps. This behavior does not depend on the parameter that is varying.

Furthermore, if we consider the distance from the necessarily optimal preference, we can see that the algorithm finds the best solution in the first 20000 steps (see Figures 8(b), 8(d), 8(f) and 8(h)). Exceptions occur around the peaks of Figure 6. For example, in Figure 8(b) with $n = 17$ or $n = 20$ the solution preference is improved during the whole execution and not only in the first steps. Other examples are for $d = 80\%$ in Figure 8(d) and for $t = 40\%$ in Figure 8(h).
Fig. 5. Runtime behavior of WORST. When a parameter is fixed, its value is: \( n=10, \ m=5, \ d=50\%, \ i=30\%, \ t=10\% \).
Fig. 6. Solution quality. When a parameter is fixed, its value is: $n=10$, $m=5$, $d=50\%$, $i=30\%$, $t=10\%$.

Fig. 7. Percentage of elicited tuples. When a parameter is fixed, its value is: $n=10$, $m=5$, $d=50\%$, $i=30\%$, $t=10\%$. 
As in the fuzzy case, we tested our local search algorithms (using the WB strategy in this case) on instances up to 100 variables. From Figure 8(a) we can
see that the algorithm elicits around 30% of incomplete preferences from 10 to 90 variables. From 90-100 variables, the instances start to have solutions equal to $+\infty$ and the algorithm elicits more preferences. In Figure 9(b) we measured the execution time of WB compared with WBB. The branch and bound algorithm reaches the time limit of 10 minutes per instance with just 15 variables so we stopped the execution at 30 variables. On the other hand, the local search algorithm, can solve instances up to 90 variables taking less time.

Summarizing, we can see that our local search approach finds a solution with a quality which is very close (with an error of at most 2%) to the quality of the necessarily optimal ones. Moreover, such a quality can be obtained also if execution is stopped after only 10000 steps.

5 Conclusions and future work

We developed and tested a local search algorithm to solve incomplete fuzzy and weighted CSPs. We tested different elicitation strategies and, in both cases, our best strategies have shown good results compared with the branch and bound solver described in [6]. More precisely, our local search approach shows a very good solution quality when compared with complete algorithms. In addition it shows better scaling properties than such complete methods. Moreover we can notice that, due to the different nature of IFCSPs and IWCSPs, and in particular due to the multiplicative idempotent operator in the fuzzy case, both local search and branch and bound approaches need less elicitation when solving IFCSPs than with IWCSPs.

We plan to test our algorithm also using real world case study, such as time tabling assignment problems where employees have partially expressed their preferences. Doing this, we will be able to evaluate how the performance of our search approach is influenced by the hidden structure usually present in real world problems.
6 Acknowledgements

Research partially supported by the Italian MIUR PRIN project 20089M932N: "Innovative and multi-disciplinary approaches for constraint and preference reasoning".

References


Enforcing Soft Local Consistency on Multiple Representations for DCOP Solving

Patricia Gutierrez and Pedro Meseguer

IIIA, Institut d’Investigació en Intel·ligència Artificial
CSIC, Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain.
{patricia|pedro}@iiia.csic.es

Abstract. Connecting soft arc consistency with distributed search in DCOP solving has been very beneficial for performance. However, including higher levels of soft arc consistency breaks usual privacy requirements. To avoid this issue, we propose to keep different representations of the same problem on each agent, on which soft arc consistencies are enforced respecting privacy. Deletions caused in one representation can be legally propagated to others. Experimentally, this causes significant benefits.

1 Introduction

Recently, a number of methods for solving Distributed Constraint Optimization Problems (DCOP) have appeared [7–10] possibly as consequence of the raise of multiagent technology. In this problems, variables are distributed into several agents and the task of interest is to find a global optimal assignment in a distributed way.

Distributed search solves DCOP by exploring the search space exchanging messages among agents. Lately, the BnB-ADOPT+ algorithm has been enhanced with some forms of soft arc consistency maintenance (specifically, AC* and FDAC*) [4], which have been shown very beneficial for performance, saving an important number of exploratory messages. Moving into the next soft arc consistency level, EDAC*, we have found that its enforcement breaks the privacy requirements usually assumed in the distributed setting. With the double aim of improving as much as possible distributed search performance while respecting agent privacy, we present the following approach. To enforce FDAC* agents must be ordered. At each agent, we propose to keep several representations of the other agents (each representation originally corresponds to a different order), from which some values can be deleted. Interestingly, these deletions can be legally propagated among representations. In some sense, this remembers channeling constraints [1], where different modelings of the same problem coexist in the solving process and pruning in one model is propagated to other models by channeling constraints. Experimentally, our approach causes significant savings on two benchmarks.

This paper is organized as follows. After recalling basic concepts, we present the idea of connecting BnB-ADOPT+ with soft arc consistency, showing that EDAC* connection breaks privacy requirements. To avoid this issue, we propose maintaining multiple representations. Experimentally, we obtain significant benefits on two benchmarks.
2 Preliminaries

COP. A binary Constraint Optimization Problem (COP) is defined by \( \langle X, D, C \rangle \), where \( X = \{ x_1, \ldots, x_n \} \) is a set of variables, \( D = \{ D_1, \ldots, D_n \} \) is a set of finite domains \((x_i \text{ takes values in } D_i)\), and \( C \) is a set of unary and binary cost functions; \( C \in C \) specifies the cost of every combination of values of \( \text{var}(C) \). The cost of a complete tuple is the addition of all individual cost functions evaluated on that particular tuple. An optimal solution is a complete tuple with minimum cost. This definition assumes the weighted model of soft constraints [6].

Soft Arc Consistency. Two COP instances defined over the same set of variables are equivalent if they define the same cost distribution on complete assignments. Considering a single COP: \((i, a)\) means the value \(a\) of variable \(x_i\), \(\top\) is the lowest unacceptable cost, \(C_{ij}\) is the binary cost function between \(x_i, x_j\), \(C_i\) is the unary cost function on \(x_i\) values, \(C\) is a zero-ary cost function that represents a necessary global cost of any complete assignment. As [5, 2], we consider the following local consistencies for the weighted model (variables connected by a cost function are ordered):

- **Node Consistency**: \((i, a)\) is node consistent if \(C_{\top} + C_i(a) < \top\); \(x_i\) is NC if all its values are NC and there is \(a \in D_i\) such that \(C_i(a) = 0\); a COP is NC if every variable is NC.
- **Arc consistency**: \((i, a)\) is arc consistency (AC) with respect to cost function \(C_{ij}\) if there is \(b \in D_j\) s.t. \(C_{ij}(a, b) = 0\); \(b\) is a simple support of \(a\); \(x_i\) is AC if all its values are AC with respect to every binary cost function involving \(x_i\); a COP is AC if every variable is AC and NC.
- **Directional arc consistency**: \((i, a)\) is directional arc consistent (DAC) with respect to cost function \(C_{ij}\), \(j > i\), if there is \(b \in D_j\) such that \(C_{ij}(a, b) + C_j(b) = 0\); \(b\) is a full support of \(a\); \(x_i\) is DAC if all its values are DAC with respect to every \(C_{ij}\), \(j > i\); a COP is DAC if every variable is DAC and NC.
- **Full DAC**: a COP is FDAC if it is DAC and AC.
- **Existential arc consistency**: Variable \(x_i\) is existential arc consistent (EAC) if there is at least one value \(a \in D_i\) such that \(C_i(a) = 0\) and it has a full support in every cost function \(C_{ij}\); a COP is EAC if every variable is NC and EAC.
- **EDAC**: a COP is EDAC if it is FDAC and EAC.

AC/DAC can be reached forcing simple/full supports to NC values and pruning values not NC. Simple supports can be forced on every value by projecting the minimum cost from its binary cost functions to its unary costs, and then projecting the minimum unary cost into \(C\). Full supports can be forced in the same way, but first it is needed to extend from the unary costs of neighbors to the binary cost functions the minimum cost required to perform in the next step the projection over the value. The systematic application of these operations (projection and extension) produces equivalent instances, so they do not change the minimum cost nor the optimal solutions of the original instance [5]. When we prune a value from \(x_i\) to ensure AC/DAC, we need to recheck AC/DAC on every variable that \(x_i\) is constrained with, since the deleted value could be the simple/full support.
DCOP. A Distributed Constraint Optimization Problem (DCOP) is defined by a tuple \( \langle X, D, C, A, \alpha \rangle \), where \( \langle X, D, C \rangle \) define a COP, \( A = \{1, \ldots, p\} \) is a set of \( p \) agents, and \( \alpha : X \rightarrow A \) maps each variable to one agent. We assume that each agent holds exactly one variable (so the terms variables and agents can be used interchangeably) and cost functions are unary and binary only. Agents communicate through messages, which could be delayed but never lost. Between two agents, messages are delivered in the order they were sent.

**BnB-ADOPT / BnB-ADOPT**. BnB-ADOPT [10] is a reference algorithm for DCOP. It is a depth-first version of ADOPT [7], showing a better performance. As ADOPT, it arranges agents in a DFS tree. Each agent holds a context, which is a set of assignments involving the agent’s ancestors, and will be updated with message exchange. BnB-ADOPT uses three message types: VALUE\((i, j, val, th)\) – \(i\) informs descendant \(j\) that \(i\) takes value \(val\) with threshold \(th\)– COST\((k, j, context, lb, ub)\) – \(k\) informs parent \(j\) that its bound are \(lb/ub\) in context – and TERMINATE\((i, j)\) – \(i\) informs child \(j\) that \(i\) terminates--. A BnB-ADOPT agent executes the following loop: it reads and processes all incoming messages and takes value. Then, it sends a VALUE to each child or pseudochild and a COST to its parent. Here, we assume that the reader has some familiarity with BnB-ADOPT code.

**BnB-ADOPT** \(^+\) [3] is a version of BnB-ADOPT that saves most of redundant messages, causing substantial reductions in communication costs with respect to the original algorithm. BnB-ADOPT \(^+\) keeps the optimality and termination of BnB-ADOPT. Basically, it stores the last VALUE and COST messages sent to each destination, and it checks if some VALUE or COST messages to be sent at the current iteration might be redundant. If messages are found redundant, they are not sent in most of the cases.

### 3 Connecting Distributed Search with Soft Arc Consistency

Combining search with soft arc consistency brings substantial benefits to search performance. Taking BnB-ADOPT \(^+\) as the distributed search algorithm, its combination with AC\(^*\) and FDAC\(^*\) soft arc consistency levels [4] has provided very good results. The resulting algorithms maintain BnB-ADOPT \(^+\) optimality and termination, improving its performance: soft arc inconsistent values are removed from their domains, making smaller the search space, which causes substantial reductions in the search effort.

Initially, the proposed level of soft arc consistency is assured in a preprocess step, and during execution it is enforced every time soft arc consistency is broken. The AC\(^*\)/FDAC\(^*\) condition could be violated during execution if one of the following cases occur: a value is deleted (a support could be lost), a new \(\top\) is found, or \(C_\phi\) increments. AC\(^*\)/FDAC\(^*\) is enforced implementing the projection and extension operators for the distributed case. In the next two sections we summarize existing approaches to connect BnB-ADOPT \(^+\) with AC\(^*\) and FDAC\(^*\) soft arc consistency levels [4]. In addition, we discuss its combination with EDAC\(^*\), the next soft arc consistency level.
3.1 Connecting BnB-ADOPT+ with AC*

AC* requires that every value of every variable has a simple support on any other variable it is constrained with. AC* maintenance involves projecting binary costs into unary ones, and unary costs on $C_\phi$.

As a consequence of distributed search and AC* maintenance, some deletions may occur. In distributed search, NC* property is relaxed to allow equality with $\top$, $C_\phi + C_f(a) \leq \top$ because otherwise it could be pruned an optimal solution. In the centralized case this is not a problem, because each time a new solution is found it is stored. In the distributed case, however, this is not possible since there is no special agent holding the whole solution. So a value is pruned when its cost exceeds $\top$.

To propagate value deletions, and inform of $\top$ and $C_\phi$ values some information has to be exchanged between agents: the subtree contribution of each agent to the $C_\phi$ ($\text{subtreeContribution}$), the global $C_\phi$, and the global $\top$. This information is included in the original BnB-ADOPT+ messages: VALUE and COST. A new message type is added: DEL to notify value deletions. The message structure appears in Figure 1.

It is assumed that cost functions are initially AC*. If not, they are made AC* by the preprocess that appears in Figure 2. The execution logic of the original BnB-ADOPT+ algorithm remains mostly the same, with the following minor changes:

1. When $\text{self}$ receives a VALUE message, updates its local copy of $\top$ and $C_\phi$. These values are later used for NC* pruning. When $\text{self}$ sends VALUE messages, if $\text{self} = \text{root}$, it calculates the global $C_\phi$ as the sum of contributions from all its children plus its own local contribution, and it is sent downwards.
2. When $\text{self}$ receives a COST message from a child $c$, it records the contribution of $c$ to the global $C_\phi$ ($\text{subtreeContribution}$). Then, when $\text{self}$ sends a COST message, it calculates its own $\text{subtreeContribution}$ as the contributions of its children plus its own local contribution, and this is sent upwards.
3. After an agents reads and process all incoming messages, it checks if any of its values can be unconditionally deleted. If so, the value is deleted and AC* is reinforced. A value can be deleted either because it is found not NC*, or because its lower bound exceeds the $\top$ of the problem (to assure unconditional deletions the lower bound should aggregate only costs not involved with any higher agent).

BnB-ADOPT+:

```
VALUE(sender, destination, value, threshold)
COST(sender, destination, context[], lb, ub)
STOP(sender, destination)
```

BnB-ADOPT+ with AC* maintenance:

```
VALUE(sender, destination, value, threshold, $\top$, $C_\phi$)
COST(sender, destination, context[], lb, ub, subtreeContribution)
STOP(sender, destination, emptydomain)
DEL(sender, destination, value)
```

BnB-ADOPT+ with FDAC* maintenance: BnB-ADOPT+ with AC* maintenance plus

```
UCO(sender, destination, vectorOfExtensions)
```

Fig. 1. Messages of original BnB-ADOPT+ and BnB-ADOPT+ maintaining AC* and FDAC*.
procedure \(AC^*\)-preprocess(\(\top\))
initialise;
\(AC^*\)-pre();
while \(\neg\) end \& \(\neg\) quiescence do
while input queue is not empty do
.msg \(\leftarrow\) getMsg();
switch(msg.type)
\(DEL\) \(\leftarrow\) ProcessDelete(msg); \(STOP\) \(\leftarrow\) ProcessStop(msg);
PruneDomainSelf();
end

procedure \(AC^*\)-pre()
for each \(i \in\) neighbors(self) do
if \(i <\) self then
FromUnaryToBinary(self, i);
FromBinaryToUnary(i, self);
else
FromBinaryToUnary(i, self);
FromUnaryToUnary(self, i);
FromUnarySelfToC();
end

procedure FromBinaryToUnary(i, j)
for each \(a \in D_i\) do
\(v \leftarrow\) argminn \(\in\) \(\partial\) \(C_{ij}(a, b)\); \(\alpha \leftarrow\) \(C_{ij}(a, v)\);
for each \(b \in D_j\) do \(C_{ij}(b, a) \leftarrow\) \(C_{ij}(a, b) - \alpha\);
if \(i =\) self then \(C_i(a) \leftarrow\) \(C_i(a) + \alpha\);
end

procedure PruneDomainSelf()
for each \(a \in D_{self}\) do if \(C_{self}(a) + C_o >\) \(\top\) then DeleteValue(a);
end

procedure DeleteValue(a)
\(D_{self} \leftarrow D_{self} -\) \{a\};
if \(D_{self} =\) \(\emptyset\) then
for each \(j \in\) neighbors(self) do sendMsg(\(\top\), self, j, true);
end \(\leftarrow\) true;
else
for each \(j \in\) neighbors(self) do
sendMsg(\(DEL\), self, j, a);
FromBinaryToUnary(j, self);
FromUnarySelfToC();
if \(a =\) myValue then \(\text{myValue} \leftarrow\) argminn \(\in\) \(\partial\) \(LB(v)\);
end

procedure ProcessDelete(msg)
\(D_{sender} \leftarrow D_{sender} -\) \{msg.value\};
FromBinaryToUnary(self, msg.sender);
FromUnarySelfToC();
end

procedure ProcessStop(msg)
if (msg.emptyDomain = true) then
for each \(i \in\) neighbors(self), \(j \neq\) sender do sendMsg(\(\top\), self, j, true);
end \(\leftarrow\) true;
end

Fig. 2. AC* preprocess.

3.2 Connecting BnB-ADOPT+ with FDAC*

In the same way as section 3.1, BnB-ADOPT+ is connected with FDAC*. Given a variable ordering, FDAC* implies that every value of every variable has a simple support with higher variables in the ordering (AC*) and a full support with lower variables in the
procedure FDAC*-preprocess(T)
initialize;
AC*-pre();
while ¬end ∧ ¬quiescence do
while input queue is not empty do
msg ← getMsg();
switch(msg.type)
DEL: ProcessDelete(msg);
UCO: ProcessUnaryCosts(msg);
STOP: ProcessStop(msg);
PruneDomainSelf();
DAC*-pre();
procedure DAC*-pre()
for each i ∈ neighbors(self) do
for each a ∈ Di do P[a] ← minb∈Dself{Ci,ucf(a, b) + Cself(b)};
for each b ∈ Dj do E[b] ← maxa∈Di{P[a] − Ci,ucf(a, b)};
if E ̸= [0,...,0] then
sendMsg(UCO, self, i, E);
FromUnarySelfToBinary(i, E);
FromBinaryToUnary(i, self);
procedure FromUnarySelfToBinary(i, vector)
for each b ∈ Dself do
for each a ∈ Di do Ci,ucf(a, b) ← Ci,ucf(a, b) + vector[b];
Cself(b) ← Cself(b) − vector[b];
procedure ProcessUnaryCosts(msg)
for each b ∈ Dself do
for each a ∈ Di do Cself,send(a, b) ← Cself,send(a, b) + msg.vector(b);/* extension */
FromBinaryToUnary(self, sender); FromUnarySelfToCφ();

Fig. 3. FDAC* preprocess. Missing procedures appear in Figure 2.

ordering (DAC*). In addition to the projection operator, maintaining FDAC* requires
the extension operator, which extends unary costs into binary ones.

The full supports requirement causes a new message type, the UCO message, that
carries the unary costs that a lower variable is willing to offer to a higher variable. The
structure of this new message appears in Figure 1. We assume that initial cost functions
are FDAC* with respect to the ordering used. If not, they can be made FDAC* with
the preprocess that appears in Figure 3, where first all cost functions are made AC*
(both ways), and second they are made DAC* (one way: from the higher to the lower
agent, the ordering is considered the same as the DFS tree). The order in which soft
arc consistencies is done is relevant, because DAC* enforcing is prepared to respect
previous AC* enforcing [5].

3.3 Maintaining EDAC*

In the distributed case, it is usually assumed that each agents knows its variable and
the cost functions that it has with other agents. This second assumption implies that it
knows the domain of the variables it is constrained with. To enforce soft arc consistencies
higher than NC*, it is required that if agent i is constrained with agent j by Cij, i
has to represent locally Dj. For privacy reasons, we assume that the unary costs of the
values of an agent are held by itself, who knows them and updates them accordingly. An agent neither can know nor update unary costs of other agents.

Maintaining AC*/FDAC* during distributed search requires each agent to know the binary cost functions in which it is involved and the unary costs of its values. These requirements are in agreement with the privacy requirements not permitting an agent to know the unary costs of values of other agents. However when moving to EDAC* (the next soft arc consistency level) this privacy requirement is broken. EDAC* maintenance requires that at each variable there is a value with unary cost 0 which is fully supported in both directions (cost functions linking higher agents with self, cost functions linking self with lower agents).

Let us consider two agents \( i, j, i < j \) that share a cost function \( C_{ij} \). To assure that \( j \) has a value fully supported by \( i \), \( i \) has to extend some of its unary costs into the binary ones, which will be projected on the unary costs of \( j \) values. However, \( i \) will only extend its unary costs to \( j \) if it is sure that from this operation \( C_{\phi} \) will increase (otherwise termination is not guaranteed). But this condition can only be assured if \( i \) knows the unary costs of \( j \). Therefore, aiming at EDAC* maintenance breaks the natural privacy requirements explained above, which represents a serious drawback in the distributed environment.

A possible way to partially avoid this issue, while enforcing some soft arc consistency that prunes more than FDAC*, comes from the following fact. Observe that the first variable in a FDAC* ordering satisfies the EDAC* property: for FDAC* each value has a simple/full support and there is a value with cost 0 (for NC*); since it is the first variable in the ordering, these supports have to be full supports. This suggests us an alternative way for the distributed setting: instead of having a single ordering of agents, we may have several orderings. On each ordering we enforce FDAC*, and the first variable of every ordering satisfies EDAC*. Next we show that having different representations and propagating deletions among them is legal and does not compromise the correctness of this idea.

4 Multiple Representations

It is known that with different variable orderings FDAC* maintenance may cause pruning different values (see the example at the end of the section). This fact motivates the present approach. It is unclear how to determine the best ordering, in the sense of the ordering that prunes most. Instead of looking for that ordering, we consider as alternative to keep multiple orderings \( O_1, \ldots, O_r \) at each agent, on which FDAC* is separately enforced. Maintaining FDAC* in \( O_p \) may cause the deletion of value \( a \) of variable \( i \): this deletion is propagated to all other orderings \( O_1, \ldots, O_{p-1}, O_{p+1}, O_r \). Propagating value deletions among different orderings is legal as proved next.

**Proposition 1.** Let us assume that enforcing FDAC* on the ordering \( O_1 \) causes to delete value \( (i, a) \), while enforcing FDAC* on the ordering \( O_2 \) causes to delete value \( (j, b) \). Then, both values can be deleted without losing any optimal solution.

---

1 This can be clearly seen in line 1 of function FindExistentialSupport of [2]. The expression of \( \alpha \) involves \( C_i(a) \) and \( C_j(b) \), unary costs of values of \( x_i \) and \( x_j \). While this causes no difficulties in a centralized approach, it becomes a real issue in a distributed setting.
Proof. If enforcing FDAC∗ using ordering O1 we delete value (i, a), this means that value a for variable i will not appear in any optimal solution of the problem. This fact derives directly from soft arc consistency, and it is independent of the ordering used. The same situation happens with ordering O2 and value (j, b). Therefore, it is legal to remove both values independently of the ordering used.

Since cost functions evolve depending on the ordering used, we prefer to talk about different representations instead of different orderings (clearly, each ordering defines a representation). Propagating value deletions between equivalent representations (representations that define the same cost distribution on complete assignments) produce new equivalent representations, as proved next.

**Proposition 2.** Let us assume that we have two equivalent representations R1 and R2 of the same problem, and enforcing FDAC∗ on R1 causes to remove (i, a) producing a new representation R′1. Removing (i, a) from R2 we obtain a new representation R′2. Then, R′1 and R′2 are equivalent.

**Proof.** Let us call A the set of complete tuples with value a for variable i, and CT(R) the set of complete tuples generated by representation R. Then, we have

\[
CT(R'_1) = CT(R_1) \setminus A
\]

\[
CT(R'_2) = CT(R_2) \setminus A
\]

Since R1 and R2 are equivalent representations, they have the same cost distributions on complete tuples. It is direct to see that R′1 and R′2 are equivalent representations. □

As explained in the previous section, by maintaining FDAC∗ on multiple orderings we assure EDAC∗ on the first variable of each order. However, even if we have k variables, and we assure EDAC∗ on every one of them maintaining k orderings (one EDAC∗ variable per ordering, the first one), this is not necessarily as strong as maintaining all variables EDAC∗ in one single order, as shown in Figure 4. Observe that maintaining EDAC∗ with order \{x1, x2, x3\} accumulates \(C_\phi = 2\). However maintaining EDAC∗ separately on variable x1 for order \{x1, x2, x3\}, x2 for order \{x2, x3, x1\}, and x3 for order \{x3, x2, x1\} only accumulates \(C_\phi = 1\) on every case. The quality of the representations used may have an impact on performance. The detailed study of this point is left as future work.

### 4.1 Implementation

The current implementation of BnB-ADOPT+ -FDAC∗ includes

- BnB-ADOPT+ works with the original cost functions, while AC∗ is enforced on a copy of them. When a value is deleted, this is included in the original cost functions.
- There is a pre-preprocess to compute an initial \(\top\) to pass to AC∗-preprocess(\(\top\)).
The idea of multiple representations can be combined with distributed search, producing the new BnB-ADOPT$^+\text{-FDAC}^\ast\text{-MR}$ algorithm. For single order FDAC$^\ast$ enforcing, we maintain a single copy of the cost functions in which we enforce FDAC$^\ast$, following the order in which agents appear in the DFS tree branches. Implementing $r$ representations requires each agent holding a set of cost functions $\{C_1, C_2, \ldots, C_r\}$. On all $r$ cost functions agents enforce FDAC$^\ast$. The direction of the arc consistency enforcement will be defined by the set of partial orders $\{O_1, O_2, \ldots, O_r\}$.

Orders are generated in the following way. Initially $r$ different agents are selected, each of them will be the first agent in one of the $r$ orders. Each agent chooses randomly a neighbor and sends a message containing the order (at this moment the order only contains the sender agent as the first agent). When this message arrives, if the receiver is not already in the order and the order is not complete, the receiver selects if it wants to be the next agent in the order. After this, the receiver chooses another neighbor randomly and sends the order. When the order is complete, it is informed to all agents in the DFS tree.

Having different orders produces different flows of costs and as result, some values may be found node-inconsistent in some representation. Then these values are deleted from all the representations. Every time there is a deletion, the agent will need to reinforce FDAC$^\ast$ over the $r$ representations. To do this, agents need to store:

1. One partial order for every representation $r$.
2. One copy of the binary and unary cost functions for every representation $r$.
3. One $C_\phi$ value for every representation $r$. Since different projections and extensions are performed on every representation, different $C_\phi$ values are obtained.
4. All children subtreeContribution to $C_{\phi}$ for every representation $r$. Since different projections and extensions are performed on every representation, agents will contribute to the $C_{\phi}$ differently on every one of them.

The following changes in messages are needed to maintain the previous structures:

- **VALUE**: a vector $C_{\phi}$ is sent containing the $C_{\phi}$ values for every representation.
- **COST**: a vector subtreeContribution is sent containing the subtree contribution to the $C_{\phi}$ for every representation.
- **UCO**: a vector vectorOfExtensions is sent containing the extensions for every representation.

4.2 Example

Consider the problem in Figure 5 with $\top = 4$. If FDAC* is enforced with the order $\{x_0, x_1, x_2\}$, we get cost function $C_{01}$ as displayed in Figure 6 (b). On the other hand, if FDAC* is enforced following the order $\{x_1, x_0, x_2\}$ value $(1, a)$ is found node inconsistent, it is deleted and we get cost function $C_{01}$ as displayed in Figure 6 (c) and (d). Initially we do not know which is the best order to maintain FDAC*, but if we work with both orders maintaining two representations we will be able to prune inconsistent values in any of them.

For this problem, BnB-ADOPT+ maintaining FDAC* with order $\{x_0, x_1, x_2\}$ requires 23 messages, while maintaining FDAC* with both orders $\{x_0, x_1, x_2\}$ and $\{x_1, x_0, x_2\}$ requires 21 messages. We present a reduced execution trace of BnB-ADOPT+ -FDAC* (Table 1, left) with order $\{x_0, x_1, x_2\}$ and BnB-ADOPT+ -FDAC*-MR (Table 1, right) with both orders $\{x_0, x_1, x_2\}$ and $\{x_1, x_0, x_2\}$.

A short description of the execution follows. From lines 1 to 5 of Table 1 both algorithms behave in the same way: $x_0$ assigns value $a$ and sends a VALUE message to $x_1$. Knowing that $x_0 = a$, $x_1$ assigns value $b$ (best value with current information) and sends a VALUE message to $x_2$. Then, $x_2$ informs the cost of the current assignment to $x_1$, and finally $x_1$ sends a COST message to $x_0$. When this COST arrives to $x_0$, a complete solution has been found and $\top = 4$. $x_0$ changes value to $b$ and sends a VALUE message to $x_1$ that contains $\top$. Some other concurrent messages are sent, but we do not comment them because they are not relevant for the example.

From lines 6 to 9 algorithms behave differently. In BnB-ADOPT+ -FDAC*-MR (right), when $x_0$ VALUE message arrives, $x_1$ is able to delete value $a$ with $\top = 4$, since this value is not node consistent in the order $\{x_1, x_0, x_2\}$. In lines 6-8 this value is
deleted and neighbors are informed (notice that this deletion is not performed on BnB-ADOPT\textsuperscript{+}-FDAC\textsuperscript{*} until lines 16-17). Now, \(x_1\) can only take value \(b\) since value \(a\) has been deleted. In BnB-ADOPT\textsuperscript{+}-FDAC\textsuperscript{*} (left), in line 9, agent \(x_1\) takes value \(a\) since it is the one that minimize the LB for the current context (\(\text{LB}(a) = 5\) and \(\text{LB}(b) = 6\)). This VALUE message will generate the corresponding COST message of line 12.

In lines 10-11 of both algorithms, \(x_0\) receives a COST message that inform the cost of the assignment \(x_0 = b\) and decides to delete its value \(b\). The only possible value for \(x_0\) is now \(a\), so \(x_0\) assigns it and terminates. From line 13 to 15, \(x_1\) receives DEL, VALUE

<table>
<thead>
<tr>
<th>BnB-ADOPT\textsuperscript{+}-FDAC\textsuperscript{*}</th>
<th>BnB-ADOPT\textsuperscript{+}-FDAC\textsuperscript{*}-MR</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) (x_1) received VALUE: (x_0 = a)</td>
<td>(x_1) received VALUE: (x_0 = a)</td>
</tr>
<tr>
<td>(2) (x_2) received VALUE: (x_1 = b)</td>
<td>(x_2) received VALUE: (x_1 = b)</td>
</tr>
<tr>
<td>(3) (x_1) received COST: sender=(x_2), UB=3, LB=3</td>
<td>(x_1) received COST: sender=(x_2), UB=3, LB=3</td>
</tr>
<tr>
<td>(4) (x_0) received COST: sender=(x_1), UB=4, LB=4</td>
<td>(x_0) received COST: sender=(x_1), UB=4, LB=4</td>
</tr>
<tr>
<td>(5) (x_1) received VALUE: (x_0 = b)</td>
<td>(x_1) received VALUE: (x_0 = b)</td>
</tr>
<tr>
<td>(6) (x_1) delete value (a)</td>
<td>(x_1) delete value (a)</td>
</tr>
<tr>
<td>(7) (x_2) received DEL: (x_1 = a)</td>
<td>(x_2) received DEL: (x_1 = a)</td>
</tr>
<tr>
<td>(8) (x_0) received DEL: (x_1 = a)</td>
<td>(x_0) received DEL: (x_1 = a)</td>
</tr>
<tr>
<td>(9) (x_2) received VALUE: (x_1 = a)</td>
<td>(x_2) received VALUE: (x_1 = a)</td>
</tr>
<tr>
<td>(10) (x_0) received COST: sender=(x_1), UB=6, LB=5</td>
<td>(x_0) received COST: sender=(x_1), UB=6, LB=6</td>
</tr>
<tr>
<td>(11) (x_0) delete value (b)</td>
<td>(x_0) delete value (b)</td>
</tr>
<tr>
<td>(12) (x_1) received COST: sender=(x_2), UB=1, LB=1</td>
<td>(x_1) received COST: sender=(x_2), UB=6, LB=6</td>
</tr>
<tr>
<td>(13) (x_1) received DEL: (x_0 = b)</td>
<td>(x_1) received DEL: (x_0 = b)</td>
</tr>
<tr>
<td>(14) (x_1) received VALUE: (x_0 = a)</td>
<td>(x_1) received VALUE: (x_0 = a)</td>
</tr>
<tr>
<td>(15) (x_1) received STOP</td>
<td>(x_1) received STOP</td>
</tr>
<tr>
<td>(16) (x_1) delete value (a)</td>
<td>(x_1) delete value (a)</td>
</tr>
<tr>
<td>(17) (x_2) received DEL: (x_1 = a)</td>
<td>(x_2) received DEL: (x_1 = a)</td>
</tr>
<tr>
<td>(18) (x_2) received VALUE: (x_1 = b)</td>
<td>(x_2) received VALUE: (x_1 = b)</td>
</tr>
<tr>
<td>(19) (x_2) received STOP</td>
<td>(x_2) received STOP</td>
</tr>
</tbody>
</table>

Table 1. Summary of BnB-ADOPT\textsuperscript{+}-FDAC\textsuperscript{*} and BnB-ADOPT\textsuperscript{+}-FDAC\textsuperscript{*}-MR execution on the example of Figure 5.
and STOP messages from $x_0$. From line 16 to 18 of BnB-ADOPT$^+$-FDAC* (left) $x_1$ deletes value $a$ and assigns value $b$. Finally on line 19 of both algorithms, $x_2$ receives a STOP message and execution is concluded.

Notice that, by detecting a deletion sooner (lines 6-8 for BnB-ADOPT$^+$-FDAC*-MR, and lines 16-18 in BnB-ADOPT$^+$-FDAC*) it is possible to end the execution with 2 messages less and 1 cycle less. This 2 messages saved are the ones represented in lines 9 and 12 for BnB-ADOPT$^+$-FDAC* (left).

5 Experimental Results

We evaluate the efficiency of BnB-ADOPT$^+$-FDAC*-MR (multiple representations) with respect to BnB-ADOPT$^+$-FDAC* (single representation) on unstructured instances with binary random DCOPs, and on structured distributed meeting scheduling.

Binary random DCOP are characterized by $\langle n, d, p_1 \rangle$, where $n$ is the number of variables, $d$ is the domain size and $p_1$ is the network connectivity. We have generated random DCOP instances: $\langle n = 10, d = 10, p_1 = 0.3, 0.4, 0.5, 0.6 \rangle$. Costs are selected from an uniform cost distribution. Two types of binary cost functions are used, small and large. Small cost functions extract costs from the set $\{0, \ldots, 10 \}$ while large ones extract costs from the set $\{0, \ldots, 1000 \}$. The proportion of large cost functions is 1/4 of the total cost functions (this is done to introduce some variability among tuple costs).

On the meeting scheduling formulation, variables represent meetings, domain represent time slot assigned for each meeting, and there are constraints between meetings that share participants. We present 4 cases obtained from the DCOP repository [11] with different hierarchical scenarios and domain 10: case A (8 variables), case B (10 variables), case C (12 variables) and case D (12 variables).

Figure 7 shows experimental results for meeting scheduling and random problems averaged over 30 and 50 instances respectively, with a number of representations from 2 to 8. For an easy comparison, BnB-ADOPT$^+$-FDAC* results are drawn as an horizontal line. On random DCOPs, BnB-ADOPT$^+$-FDAC*-MR showed clear benefits on communication costs with respect to BnB-ADOPT$^+$-FDAC*. Maintaining from 4 to 6 representations, the number of exchanged messages is divided by a factor of at least 2. Also, the number of cycles required to reach the solution is divided by a factor from 2 to 3. For meeting scheduling instances we also observe a decrement in the number of cycles and messages exchanged, although to a smaller extent. In Figure 7 we observe that benefits in communication are unevenly distributed: the #saved messages/#representations ratio is higher in the left-half of the plots. This suggests that $n/2$ could be a good number of representations. More work is needed to substantiate this conjecture. Table 2 shows the details of the experiments maintaining 6 representations.

Assuming that processing each message type requires approximately the same time, a decrement in cycles combined with a decrement in the number of messages per cycle is an improvement indicator. Since agents need to process less information coming from their neighbors on each iteration, and they perform less iterations to reach the optimum, this combined reduction is beneficial for agent performance.

Notice that maintaining FDAC* on multiple representations has produced only a few extra DEL and UCO messages. In the case of DEL messages, this slight increment
is because more deletions have been produced. In the case of UCO messages, the increment is because UCO messages are only sent if their costOfExtension vector is different from zero. As we are maintaining several representations with different or-
## Random DCOPs

<table>
<thead>
<tr>
<th>p1</th>
<th>#Msgs</th>
<th>VALUE</th>
<th>COST</th>
<th>DEL</th>
<th>UCO</th>
<th>Cycles</th>
<th>NCCC</th>
<th>Deletions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>6,128</td>
<td>2,795</td>
<td>3,047</td>
<td>230</td>
<td>28</td>
<td>1,039</td>
<td>519,112</td>
<td>80</td>
</tr>
<tr>
<td>0.4</td>
<td>41,147</td>
<td>19,309</td>
<td>21,357</td>
<td>245</td>
<td>69</td>
<td>480</td>
<td>1,778,525</td>
<td>86</td>
</tr>
<tr>
<td>0.6</td>
<td>110,696</td>
<td>48,281</td>
<td>62,056</td>
<td>288</td>
<td>53</td>
<td>17,937</td>
<td>9,910,897</td>
<td>78</td>
</tr>
</tbody>
</table>

## Distributed Meeting Scheduling

<table>
<thead>
<tr>
<th>p1</th>
<th>#Msgs</th>
<th>VALUE</th>
<th>COST</th>
<th>DEL</th>
<th>UCO</th>
<th>Cycles</th>
<th>NCCC</th>
<th>Deletions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td>2,001</td>
<td>1,056</td>
<td>1,210</td>
<td>200</td>
<td>9</td>
<td>462</td>
<td>382,676</td>
<td>49</td>
</tr>
<tr>
<td>0.4</td>
<td>3,056</td>
<td>1,513</td>
<td>1,792</td>
<td>219</td>
<td>23</td>
<td>650</td>
<td>2,487,359</td>
<td>61</td>
</tr>
<tr>
<td>0.6</td>
<td>1,467</td>
<td>697</td>
<td>505</td>
<td>225</td>
<td>6</td>
<td>125</td>
<td>71,439</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 2. Experimental results of BnB-ADOPT+ -FDAC*- (first row) compared to BnB-ADOPT+ -FDAC*-MR (second row) maintaining 6 representations.

ders, is more probable that the extensions will be different from zero in any of the 6 representations.

The number of NCCCs increases since more projection and extensions are needed to maintain FDAC* on all representations. However, observe that this increment is not linear with respect to the number of representations maintained, it is smoothed by the fact that less messages are generated and more deletions are performed. So there are messages on the BnB-ADOPT+ -FDAC* algorithm that will not be needed to process with multiple representations, and also there are values that will not be needed to assign or to check for node consistency.

In general, by including few DEL and UCO messages and performing extra local computation to enforce FDAC* on multiple representations, BnB-ADOPT+ -FDAC*-MR is able to obtain important savings in communication. We assume the usual case where communication time is higher than computation time, then the total elapsed time is dominated by communication time. The flows of costs from one agent to another, implemented by UCO messages, allows an agent to pass some of their unitary costs to higher agents following different orders. This accumulations of costs on higher agents following different orders brings more pruning opportunities. In general, it is expected that the combination of multiple enforcements will be able to extend the inference benefits.

### 6 Conclusion

Maintaining soft arc consistency during distributed search has proved to be beneficial for performance with AC* and FDAC* levels. To assure FDAC* and EDAC*, a partial order among variable is needed. In this paper we discuss connecting BnB-ADOPT+ -FDAC*.
with the next soft arc consistency level EDAC∗. It turns out that to maintain EDAC∗ an
agent needs to know the unary cost of neighboring agents, which violates usual privacy
requirements.

To avoid this issue, we propose to maintain multiple orders among variables, and the
same number of cost functions on which we enforce FDAC∗ following the corresponding
order. It is known that with different orderings FDAC∗ maintenance may detect different
node inconsistent values, however if we maintain several orders we are able to detect
the node inconsistent values on every order and delete them, propagating this deletions
to other orders. Propagating deletions among different orders is legal and furthermore,
maintaining FDAC∗ on multiple representations assures EDAC∗ on the first variable of
the order. Experimental results have shown benefits in terms of number of cycles and
messages exchanged.

References

2. S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer
5. J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP.
9. M. Silaghi and M. Yokoo. Nogood-based asynchronous distributed optimization (ADOPT-
11. Z. Yin. USC dcop repository. Meeting scheduling and sensor net datasets,
Model Tracking for Dynamic SAT with Decision Change Costs

Katsutoshi Hirayama, Daisuke Hatano, and Yuta Sugimoto
Kobe University,
5-1-1 Fukaeminami-machi, Higashinada-ku, Kobe 658-0022, Japan
hirayama@maritime.kobe-u.ac.jp
083w312w@stu.kobe-u.ac.jp
099w110w@stu.kobe-u.ac.jp
http://www.edu.kobe-u.ac.jp/fmsc-hrymlab/

Abstract. We will address a dynamic decision problem, in which decision makers must pay some costs when they change their decisions along the way. To deal with this issue formally, we introduce a general framework of dynamic SAT (DynSAT) with decision change costs, whose goal is to find a sequence of models that would minimize the aggregation of the costs for changing variables. Then, we provide two solutions for solving the specific case of this problem. The first solution is to use a Weighted Partial MaxSAT solver after we encode the entire problem as the Weighted Partial MaxSAT problem. The second solution, which we believe is novel, is to use the Lagrangian decomposition technique. This technique divides the entire problem into sub-problems, each of which can be solved by an exact Weighted Partial MaxSAT solver separately, and produces both lower and upper bounds on the optimal in an anytime manner. The experimental results show that a solver based on Lagrangian decomposition can produce tighter quality upper bounds even when the size of a sequence of SAT instances becomes larger.

Keywords: dynamic SAT, Weighted Partial MaxSAT, Lagrangian decomposition

1 Introduction

SAT (satisfiability testing) is undoubtedly one of the most important decision problems in both theory and practice. It is well known that all of the NP problems can be translated into SAT in theory. Furthermore, by using a state-of-the-art SAT solver like MiniSAT [6], very large instances that consist of millions of Boolean variables can be solved within a few minutes in practice. With recent success in SAT solving, we have observed that there is growing need for extending SAT in order to deal with more sophisticated problems in the real world. Dynamic SAT (DynSAT) [11] is one of such extensions that aims at modeling the dynamic nature of real problems. DynSAT can be considered a special case of Dynamic CSP (DynCSP), which was originally proposed in [5]. In DynSAT
Katsutoshi Hirayama, Daisuke Hatano, and Yuta Sugimoto

and DynCSP, we are given a sequence of problem instances and required to solve this sequence in some way.

The solutions for DynCSP (including DynSAT) are largely divided into two categories [21].

The first encompasses the reactive approaches, which use no knowledge about the possible directions of future changes. Their goal is to find a solution for a new problem instance, which has been produced by some changes in the current problem instance. The key idea of reactive approaches is to reuse something. Both solution reuse [22] and reasoning reuse [4, 18] are typical techniques in the literature. The advantage is that they can react to any kind of change in a problem instance. However, the drawback is that the produced solutions are generally not so robust because they do not exploit any knowledge about the possible future changes.

The second category encompasses the proactive approaches, which exploit all the knowledge they may have about the possible future changes. By using such knowledge, they have a chance to produce solutions that will resist those possible changes. The robust solution [24, 7, 25] and the flexible solution [8, 9] are examples of this. However, the drawback of these solutions is the cost. It may be more expensive to find such solutions than to find a normal solution to the plain CSP.

In this work, we address a dynamic decision problem, in which decision makers must pay some costs when they change their decisions along the way. We can observe such costs in real-life problems, such as the setup cost in planning and scheduling. Suppose, for example, that you have to make this month’s schedule for the hospital staff. You may have some information on possible future events, such as someone will be on holiday until this weekend, a delicate surgery will be planned next week, etc. In face of such future events, you may want an efficient schedule, by which day-to-day operations would go smoothly and, more specifically, the cost of arrangement would be minimized. Even though the costs of decision changes are widely observed in various dynamic decision problems, we have not seen much work dealing with this issue in the context of CSP or SAT.

Notable exceptions are the studies aimed at finding a minimal-change solution for DynCSP [17, 16], where they considered the cost of decision changes to be the number of variables that are assigned new values (Hamming distance between a current solution and a new one), and presented algorithms that can produce a new solution whose Hamming distance from the current one is minimal. Similar approaches are also proposed, such as diverse and similar solutions for CSP [10] and distance-SAT whose goal is to find a model that disagrees with a given partial assignment on at most a specified number of variables [1]. Clearly, the main concern of these works is for a short-term reactive solution. On the other hand, to our knowledge, a proactive solution has not been fully investigated for a general decision problem with the cost of decision changes.

We introduce a general framework of dynamic SAT with decision change costs. We restricted our attention to SAT, not CSP, as a decision problem because there are many efficient solvers and effective encoding methods for SAT. The
input of this problem is a sequence of SAT instances and the decision change costs for variables. The output is a sequence of models (solutions) that would minimize the aggregation of the costs for changing variables. We may be able to say that this notion of solutions is one of proactive approaches for DynSAT.

We also provide two solutions for the specific case of DynSAT with decision change costs. The first solution is to use a Weighted Partial MaxSAT solver after we encode the entire problem as the Weighted Partial MaxSAT problem. The second solution, which we believe is novel, is to use the Lagrangian decomposition technique [3]. This technique divides the entire problem into sub-problems, each of which can be solved by an exact Weighted Partial MaxSAT solver separately, and produces both lower and upper bounds on the optimal in an anytime manner. To our knowledge, there has been very few work on applying Lagrangian decomposition to the constraint problem. It may be, however, worth to mention that, [14] addresses discrete MRF optimization, which is actually weighted constraint satisfaction, and [23] addresses distributed constraint optimization, by using Lagrangian decomposition.

The remainder of this paper is organized as follows. We first provide the definitions of SAT and DynSAT, respectively. Then, we present a general framework of DynSAT with decision change costs. Next, we describe two solutions for the specific case of DynSAT with decision change costs. Finally, we evaluate the performance of various solvers, each of which is based on one of the two solutions, through experiments and conclude this work.

2 SAT and Dynamic SAT

SAT is a decision problem whose goal is to decide whether a given CNF formula has a model or not. A CNF formula is a conjunction of clauses, where each clause is a disjunction of literals and a literal is a Boolean variable or its negation. A truth assignment is a mapping from Boolean variables to truth values, where we mean true by 1 and false by 0, and a model for a CNF formula is a truth assignment that makes the formula true.

Dynamic SAT (DynSAT) is an extension of SAT that aims at modeling the dynamic nature of real problems. Here, we have a definition of DynSAT [11].

Definition 1 (DynSAT). An instance of the dynamic SAT is given by \((X, \phi)\), where \(X = \{x_1, \ldots, x_n\}\) is a set of Boolean variables, and \(\phi\) is a function \(\phi : T \rightarrow \text{CNF}(X)\), where \(T\) is a set of non-negative integers and \(\text{CNF}(X)\) is the set of all possible CNF formulas that use only the Boolean variables in \(X\).

Namely, an instance of DynSAT forms an (infinite) sequence of CNF formulas on the Boolean variables in \(X\), where a CNF formula at time \(t\) will be given by function \(\phi\).

The \(k\)-stage DynSAT is the DynSAT that does not change after a fixed number \(k\) of time steps.

Definition 2 (\(k\)-stage DynSAT). An instance of \(k\)-stage dynamic SAT is given by \((k, X, \phi)\), where \(\forall t \geq k : \phi(t) = \phi(k - 1)\).
We have two variants in solving DynSAT. The decision variant of DynSAT is, given a DynSAT instance, to determine whether $\phi(t)$ has a model $M(t)$ at any time $t$. We say $\phi$ is satisfiable if this is the case, and otherwise, we say $\phi$ is unsatisfiable. On the other hand, the model tracking variant of DynSAT is, given a DynSAT instance, to find a sequence of models. In this paper, we focus on the model tracking variant on the assumption that decision makers make some decisions based on such a sequence of models.

3 Dynamic SAT with Decision Change Costs

It would be natural to consider that when a model has been changed over time, decision makers change their decisions accordingly. We assume that if they change their decisions in the real world, they must pay some cost for that (such as the setup cost in planning and scheduling). We generally call this cost the decision change cost.

The decision change cost should be defined carefully depending on application domains. However, in this work, we formulate it as the aggregation of the costs for changing variables. We first define the cost for changing a variable at a certain time.

Definition 3 (cost for changing variable at a certain time). The cost for changing variable $x_i$ at time $t$ is given by a function $f : T \setminus \{0\} \times X \times \{1, 0\} \times \{1, 0\} \rightarrow \mathbb{R}^+$, where $T$ is a set of non-negative integers, $X$ is a set of Boolean variables, and $\mathbb{R}^+$ is a set of positive real numbers.

For example, $f(t, x_i, 1, 0)$ returns a cost $c_{0,t}^{i}$ when we change variable $x_i$ from 1 to 0 at time $t$. Similarly, $f(t, x_i, 0, 1)$ returns a cost $c_{1,t}^{i}$ when we change variable $x_i$ from 0 to 1 at time $t$. Obviously, both $f(t, x_i, 1, 1)$ and $f(t, x_i, 0, 0)$ must return 0 for any $x_i$ and $t$, since there should be no cost when we keep the same value for a variable.

Given two consecutive models $M(t-1)$ and $M(t)$, we can identify the cost for changing variable $x_i$ between these two models, denoted by $\text{cost}(x_i, M(t-1), M(t))$, by referring to the above cost function of $f$. For example, given that $x_i$ is 1 in $M(t-1)$ but 0 in $M(t)$, the value of $\text{cost}(x_i, M(t-1), M(t))$ must be $f(t, x_i, 1, 0)$. Then, by aggregating all of the costs for variables with some local aggregation operator $\oplus$, we can compute the cost for changing a model from $M(t-1)$ to $M(t)$, which we will denote by $\text{cost}(M(t-1), M(t))$. Namely, we have

$$\text{cost}(M(t-1), M(t)) \equiv \bigoplus_{x_i \in X} \text{cost}(x_i, M(t-1), M(t)).$$

For example, if $\oplus$ is ‘+’, we have

$$\text{cost}(M(t-1), M(t)) \equiv \sum_{x_i \in X} \text{cost}(x_i, M(t-1), M(t)).$$

Furthermore, let $M$ be a sequence of models over a set of non-negative integers $T$, i.e., $M = \{M(t) \mid t \in T\}$. We can define the cost of this sequence of
models $M$ by:
\[
\text{cost}(M) \equiv \bigodot_{t \in T \setminus \{0\}} \text{cost}(M(t-1), M(t)),
\] (1)

where $\bigodot$ is a global aggregation operator over the costs for changing models. For example, if $\bigodot$ is \'+\', we have
\[
\text{cost}(M) \equiv \sum_{t \in T \setminus \{0\}} \text{cost}(M(t-1), M(t)).
\]

We can now define DynSAT with decision change costs.

**Definition 4 (DynSAT with decision change costs).** An instance of dynamic SAT with decision change costs is given by 5-tuple $(X, \phi, f, \oplus, \bigodot)$, where $X$ and $\phi$ are in Definition 1, $f$ is in Definition 3, $\oplus$ and $\bigodot$ are local and global aggregation operators, respectively.

As with the plain DynSAT, we can define $k$-stage DynSAT with decision change costs as follows.

**Definition 5 ($k$-stage DynSAT with decision change costs).** An instance of $k$-stage dynamic SAT with decision change costs is given by 6-tuple $(k, X, \phi, f, \oplus, \bigodot)$, where $\forall t \geq k : \phi(t) = \phi(k-1)$.

Given an instance of ($k$-stage) DynSAT with decision change costs, our goal is to find a sequence of models whose cost, generally defined by (1), is minimized. We call this minimal cost the optimal value for ($k$-stage) DynSAT with decision change costs. Furthermore, if the optimal value is finite, we refer to the sequence of models that achieves the minimal cost as an optimal solution.

**Example 1 (3-stage DynSAT with decision change costs).** As illustrated in Figure 1, let us assume we have $(3, \{x_1, x_2\}, \phi, f, +, +)$, where
- $\phi(0) = (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2)$,
- $\phi(1) = (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2)$,
- $\phi(2) = (x_1 \lor x_2) \land (\neg x_1 \lor \neg x_2)$,
and

\[- \ f(1, x_1, 1, 0) = 1, \ f(1, x_1, 0, 1) = 3, \ f(1, x_2, 1, 0) = 4, \ f(1, x_2, 0, 1) = 2, \]
\[- \ f(2, x_1, 1, 0) = 5, \ f(2, x_1, 0, 1) = 7, \ f(2, x_2, 1, 0) = 4, \ f(2, x_2, 0, 1) = 6. \]

In the above, we have two possible models for each CNF formula \( \phi(t) \):

\[- \ M(0) \in \{(x_1 = 0, x_2 = 1), (x_1 = 1, x_2 = 0)\}, \]
\[- \ M(1) \in \{(x_1 = 0, x_2 = 0), (x_1 = 1, x_2 = 1)\}, \]
\[- \ M(2) \in \{(x_1 = 0, x_2 = 1), (x_1 = 1, x_2 = 0)\}. \]

An optimal solution \( M^* \) for this instance is a sequence of models, \( M^*(0) = (x_1 = 1, x_2 = 0), M^*(1) = (x_1 = 1, x_2 = 1), M^*(2) = (x_1 = 1, x_2 = 0) \), whose cost is \( f(1, x_2, 0, 1) + f(2, x_2, 1, 0) \) resulting in 6.

4 Solutions

Among the possible DynSAT with decision change costs, we focus on the \( k \)-stage problem specified by \( (k, X, \phi, f_+, +) \), which we believe is one of the natural settings for this framework. In this section, we provide two solutions for it.

4.1 Weighted Partial MaxSAT Solving

In our first solution, we first translate a given instance of \( (k, X, \phi, f_+, +) \) into a Weighted Partial MaxSAT (WPMaxSAT) problem instance, and then solve it using an exact WPMaxSAT solver, such as Sat4j [2] or WMaxSatz, a weighted version of MaxSatz [15], or an incomplete WPMaxSAT solver, such as the one in UbcSat [20]. A WPMaxSAT problem instance comprises hard clauses that must be satisfied and soft clauses that can be violated by paying some designated costs (called weights). The goal of WPMaxSAT solving is to find a truth assignment that satisfies all of hard clauses and minimizes a weighted sum of violated soft clauses.

The translation is as follows. For every clause of each CNF formula \( \phi(t) \), we introduce a hard clause with its Boolean variables labeled by \( t \). For example, clause \( x_1 \lor x_2 \) of \( \phi(2) \) results in the hard clause of \( x_1^2 \lor x_2^2 \). Furthermore, for each mapping defined by \( f \), we introduce a soft clause that bridges the same Boolean variables belonging to different times. For example, the mapping of \( f(2, x_1, 0, 1) = 7 \) indicating that we must pay the cost of 7 when changing variable \( x_1 \) from 0 to 1 at time 2, results in the soft clause of \( x_1^{t-1} \lor \neg x_1^t \) with weight 7. Generally, \( f(t, x_i, 1, 0) = c_{0,t}^i \) results in \( \neg x_i^{t-1} \lor x_i^t \) with weight \( c_{0,t}^i \), while \( f(t, x_i, 0, 1) = c_{1,t}^i \) results in \( x_i^{t-1} \lor \neg x_i^t \) with weight \( c_{1,t}^i \).

4.2 Lagrangian Decomposition

Our second solution is to use the Lagrangian decomposition technique [3], which can provide both lower and upper bounds on the optimal value of the problem.
Decomposition First, we translate the cost function of $f$ into the 0-1 integer programming (IP) problem. Suppose we have $f(t, x_i, 1, 0) = c_{i,0}^t$ and $f(t, x_i, 0, 1) = c_{i,1}^t$, meaning that, at time $t$, we must pay $c_{i,0}^t$ when changing variable $x_i$ from 1 to 0 and $c_{i,1}^t$ when changing it from 0 to 1, respectively. On top of that, we also have $f(t, x_i, 1, 1) = 0$ and $f(t, x_i, 0, 0) = 0$, since there should be no cost when we keep the same value for variable $x_i$. These cost mappings can be achieved by solving the following 0-1 IP problem.

$$\begin{align*}
\min & \quad c_{i,0}^t y_{i,0}^t + c_{i,1}^t y_{i,1}^t \\
\text{s.t.} & \quad x_{i,t-1}^t - x_{i,t}^t - y_{i,0}^t + y_{i,1}^t = 0, \\
& \quad x_{i,t-1}^t, x_{i,t}^t, y_{i,0}^t, y_{i,1}^t \in \{0, 1\},
\end{align*}$$

where $x_{i,t-1}^t$ and $x_{i,t}^t$ are variable $x_i$ at time $t - 1$ and $t$, respectively, and both $y_{i,0}^t$ and $y_{i,1}^t$ are auxiliary 0-1 variables.

Since solving $P$ has all of the CNF formulas (3) as constraints, a feasible solution for $P$ is a sequence of models. Furthermore, the objective value of such feasible solution is a total sum of costs for changing variables. Therefore, we can get an optimal solution for the entire problem by solving $P$, or more specifically, by projecting an optimal solution for $P$ onto the variables of $x_i$.

Since we are dealing with the case where local and global aggregation operators are additive, the entire problem of $(k, X, \phi, f, +, +)$ can be represented as what we call the CNF-included 0-1 integer programming problem, which is formalized as follows.

$$\begin{align*}
P &: \quad \min & \quad \sum_{t=1}^{k-1} \sum_{i=1}^{n} (c_{i,0}^t y_{i,0}^t + c_{i,1}^t y_{i,1}^t) \\
\text{s.t.} & \quad x_{i,t-1}^t - x_{i,t}^t - y_{i,0}^t + y_{i,1}^t = 0, & \quad i = 1, \ldots, n, \quad t = 1, \ldots, k - 1, & \quad (2) \\
& \quad \phi(t), & \quad t = 0, \ldots, k - 1, & \quad (3) \\
& \quad x_{i,t}^t \in \{0, 1\}, & \quad i = 1, \ldots, n, \quad t = 0, \ldots, k - 1, \\
& \quad y_{i,0}^t, y_{i,1}^t \in \{0, 1\}, & \quad i = 1, \ldots, n, \quad t = 1, \ldots, k - 1.
\end{align*}$$

Since problem $P$ has all of the CNF formulas (3) as constraints, a feasible solution for $P$ is a sequence of models. Furthermore, the objective value of such feasible solution is a total sum of costs for changing variables. Therefore, we can get an optimal solution for the entire problem by solving $P$, or more specifically, by projecting an optimal solution for $P$ onto the variables of $x_i$.
where $\mu$ is called Lagrange multiplier vector, each element of which can take any real number. Note that we omit the 0-1 constraints on decision variables hereafter.

A simple calculation reveals that the objective function of $L$ can be re-written to produce

$$ L : L(\mu) = \min \sum_{t=1}^{k-1} \sum_{i=1}^{n} (c_{0}^{i,t} - \mu^{i,t}) y_{0}^{i,t} + \sum_{t=1}^{k-1} \sum_{i=1}^{n} (c_{1}^{i,t} + \mu^{i,t}) y_{1}^{i,t} $$

$$ + \sum_{i=1}^{n} \mu^{i,1} x_{i}^{0} + \sum_{t=1}^{k-2} \sum_{i=1}^{n} (\mu^{i,t+1} - \mu^{i,t}) x_{i}^{t} + \sum_{i=1}^{n} (-\mu^{i,k-1}) x_{i}^{k-1}, $$

s. t. $\phi(t), \ t = 0, \ldots, k - 1$.

Then, this problem can be decomposed into the following $k + 1$ sub-problems:

$$ L^{aux}(\mu) = \min \sum_{t=1}^{k-1} \sum_{i=1}^{n} (c_{0}^{i,t} - \mu^{i,t}) y_{0}^{i,t} + \sum_{t=1}^{k-1} \sum_{i=1}^{n} (c_{1}^{i,t} + \mu^{i,t}) y_{1}^{i,t} \quad (4) $$

$$ L^{0}(\mu) = \min \sum_{i=1}^{n} \mu^{i,1} x_{i}^{0}, \ \text{s. t.} \ \phi(0), \quad (5) $$

and, for each time $t$ from 1 to $k - 2$,

$$ L^{t}(\mu) = \min \sum_{i=1}^{n} (\mu^{i,t+1} - \mu^{i,t}) x_{i}^{t}, \ \text{s. t.} \ \phi(t), \quad (6) $$

and

$$ L^{k-1}(\mu) = \min \sum_{i=1}^{n} (-\mu^{i,k-1}) x_{i}^{k-1}, \ \text{s. t.} \ \phi(k-1). \quad (7) $$

It must be pointed out that each of these sub-problems is actually the WP-MaxSAT problem. Furthermore, we can see that (4) is trivial because it consists of only soft unit clauses on auxiliary variables, whereas each of the other sub-problems consists of hard clauses in $\phi(t)$ and soft unit clauses on the variables of time $t$.

On the other hand, the Lagrangian dual problem is formally defined by

$$ D : \max \ L(\mu) \ \text{s. t.} \ \mu \in \mathbb{R}, $$

where $L(\mu)$ is the optimal value for $L$, which should vary on $\mu$. This is obviously an unconstrained maximization problem over Lagrange multipliers. A value of the objective function of this Lagrangian dual problem is a lower bound on the optimal value of $P$. We should notice that the decomposition of $L$ results in the decomposition of $D$, which produces

$$ D : \max \ L^{aux}(\mu) + \sum_{t=0}^{k-1} L^{t}(\mu) \ \text{s. t.} \ \mu \in \mathbb{R}. \quad (8) $$
Our procedure solves this (decomposed) Lagrangian dual problem to search for the values of Lagrange multipliers that gives the highest objective, i.e., the highest lower bound on the optimal value of $P$. This lower bound is useful because it can be exploited in a search algorithm for $P$. Note that, for specific values to $\mu$, we can compute a lower bound on the optimal value of $P$ by simply taking a total sum of the optimal values of the sub-problems from (4) to (7).

**Outline of Procedure** Here we describe the outline of our procedure that can provide both lower and upper bounds on the optimal value of $P$ along with a feasible solution for $P$.

**step 1:** Set every element in $\mu$ to 0;

**step 2:** Solve all of the sub-problems from (4) to (7) using an exact WPMaxSAT solver;

**step 3:** Compute the highest lower bound $LB$, the lowest upper bound $UB$, and a feasible solution $M$ with the lowest upper bound;

**step 4:** If CanTerminate? then return $LB$, $UB$, and $M$; otherwise update $\mu$ and go to step 2.

Starting from step 1, this procedure repeats steps 2 through 4 until the termination condition is met. We will refer to this one iteration from steps 2 to 4 as a round. Next, we will mainly describe steps 3 and 4.

**Lower and Upper bounding** As mentioned, we can compute a lower bound on the optimal value of $P$ as a total sum of the optimal values of sub-problems. On top of that, we can provide a feasible solution for $P$ by forcing some auxiliary variables to flip in their optimal solutions so that they can satisfy each of the constraints (2) that have been relaxed to produce the Lagrangian relaxation problem. Such a feasible solution for $P$ clearly provides an upper bound on the optimal value of $P$. Therefore, at each round, we can compute both lower and upper bounds on the optimal value of $P$ as well as a feasible solution for $P$. At step 3 in the procedure, we keep the best one among those found in the previous rounds.

**Termination** We have two ways to detect the fact that the procedure has found an optimal solution for $P$. The first one relies on the following theorem on the relation between the optimal solutions for the sub-problems and the optimal solution for $P$.

**Theorem 1.** If all of the optimal solutions for the sub-problems from (4) to (7) satisfy all of the constraints (2) that have been relaxed, then these optimal solutions constitute an optimal solution for $P$.

This is obvious because such optimal solutions provide not only a lower bound but also an upper bound on the optimal value of $P$. Accordingly, we can terminate the procedure when the optimal solutions for the sub-problems satisfy all of the constraints (2).
The second one is straightforward. When a "forced" feasible solution for $P$ has the value of objective function that is equal to $LB$, this feasible solution is optimal because both $LB$ and $UB$ now reach the same value. This fact can also be used for terminating the procedure.

On the other hand, the procedure can also be terminated at anytime after performing at least one round. In that case, we can get the best bounds and the best feasible solution found so far.

**Update Lagrange Multipliers** When an optimal solution for $P$ is not found, we will update $\mu$ so that we can find a tighter lower bound on the optimal value of $P$. This involves a search algorithm for the Lagrangian dual problem. To solve this problem, we will use the sub-gradient ascent method [3].

Starting from initial values to $\mu$, this method systematically produces a sequence of values to Lagrange multiplier $\mu^{i,t}$ as follows:

1. compute sub-gradient

   \[
   G^{i,t} = x_t^{i-1} - x_t^i - y_0^i + y_1^i,
   \]

   for each $i$ and $t$, which is the LHS of (2) or the coefficient for $\mu^{i,t}$ in the objective function of $L$, by using the current optimal solutions for the sub-problems;

2. update $\mu^{i,t}$ for each $i$ and $t$ as

   \[
   \mu^{i,t} \leftarrow \mu^{i,t} + D \cdot G^{i,t},
   \]

   where $D$, called step length, is typically computed by

   \[
   D = \frac{\pi (UB - LB)}{\sum_{i,t} (G^{i,t})^2},
   \]

   in which $\pi$ is a scalar parameter gradually reduced from its initial value of 2.

This rule implies that we increase (decrease) $\mu^{i,t}$ if its coefficient $G^{i,t}$ in the objective function of $L$ is positive (negative) hoping that $L(\mu)$, a lower bound on the optimal value of $P$, will increase in the next round.

Although the sub-gradient ascent method is quite simple, we should point out that it is not monotonic. Namely, the lower bound may increase or decrease in each round. Furthermore, this method does not necessarily converge to an optimal solution for $P$. If the termination condition is not met after convergence, it means that we have only a strict lower bound on the true optimal value of $P$.

5 Experiments

We compared the performance of the following solvers, each of which is based on one of the two solutions, using randomly generated problem instances.
Solvers based on Weighted Partial MaxSAT solving
• **Sat4j**: an exact WPMaxSAT solver that uses a SAT encoding and a state-of-the-art SAT solver, which works better empirically for structured instances [2].
• **WMaxSatz**: an exact WPMaxSAT solver that is based on the branch and bound method, which works better empirically for random instances [15].
• **WalkSAT**: an incomplete and stochastic WPMaxSAT solver that is originally developed for SAT [19].
• **Irots**: an incomplete and stochastic WPMaxSAT solver that combines iterative local search and tabu search [13].

Solvers based on Lagrangian decomposition
• **Ld**: a Lagrangian decomposition method, where each sub-problem is solved by Sat4j. A feasible solution is identified at every round by forcing some auxiliary variables to flip in the optimal solutions of sub-problems so that they can satisfy each of the relaxed constraints.
• **LdWalk**: a Lagrangian decomposition method, where each sub-problem is solved also by Sat4j. A feasible solution, on the other hand, is further improved by, starting from the one obtained by Ld, running WalkSAT for a constant number of flips. This further improvement is performed only when the best lower bound is updated.
• **LdIrots**: a Lagrangian decomposition method that is similar to Ld-Walk. The only difference is to use Irots instead of WalkSAT in performing further improvement of a feasible solution.

Our goal is to see, given a certain time bound, how tight the obtained bounds are as we change the size $k$ of a sequence of SAT instances. When dealing with time-critical dynamic applications, we believe that this property must be crucial for any solvers. Note that, when a run is cut off at a certain time bound, WPMaxSAT can provide only an upper bound, but Lagrangian decomposition can provide not only an upper bound but also a lower bound.

We generated the instances of $(k, X, \phi, f, +, +)$ such that

- $k$ ranges over \{10, 15, ..., 35\};
- $X$ is a set of Boolean variables of size 100, $\{x_1, ..., x_{100}\}$;
- $\phi$ returns, for each $t$, a CNF formula randomly selected from the satisfiable instances of $uf100-430$ in SATLIB [12];
- $f$ returns an integer, the cost for changing a variable at a certain time, randomly selected from \{1, 2, ..., 10^6\}. Note that the minimum cost for changing a variable is one, not zero, for any variable and time. It means that no variable is freely changeable at all times.

As the performance measure, we used the quality upper bound $UB/LB$ of a feasible solution found by each solver within a specified time bound. Needless to say, this quality upper bound should be the lowest. Each run was made on an Intel Xeon X5460@3.16GHz, 4 cores, 32GB memory. The code was basically written in Java and compiled with JDK 1.6.0_11 on RedHat Enterprise Linux 5.
Sat4j and WmaxSATZ were downloaded from the authors’ web pages and run their latest versions with the default settings. WalkSAT and Irots were from the UBCSAT version 1.1 [20] and run with the ‘-w’ option. It must be pointed out that Lagrangian decomposition fits very well to parallel processing because, once $\mu$ is fixed to some specific values, the decomposed sub-problems are virtually independent. Therefore, in this experiment, we exploited the multi-core in our LD-based solvers to allow sub-problems to be solved in parallel. Furthermore, to make a fair comparison, we also exploited the multi-core in other methods by performing portfolio type parallelization.

Table 1 shows the ratio of runs in which each solver can find a feasible solution within 5 minutes. For each value to $k$, we created 30 different sequences of that size and made one run for each sequence. As you can see, Sat4j, LD, LDWalk, and LDIrots never fail to find a feasible solution. WmaxSATZ, on the other hand, sometimes fails and never finds feasible solutions for the sequences of size 35. Although WmaxSATZ was the best in the random Weighted Partial MaxSAT category in the 2009 MaxSAT Evaluation, it did not work well for our instances because they are highly structured and include a larger number of hard clauses. For stochastic solvers, WalkSAT did not work especially for large-sized instances and Irots never found a feasible solution for any $k$.

<table>
<thead>
<tr>
<th>Size $k$</th>
<th>Sat4j</th>
<th>WmaxSATZ</th>
<th>WalkSAT</th>
<th>Irots</th>
<th>LD</th>
<th>LDWalk</th>
<th>LDIrots</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.00</td>
<td>0.63</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>15</td>
<td>1.00</td>
<td>0.27</td>
<td>1.00</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>20</td>
<td>1.00</td>
<td>0.13</td>
<td>0.87</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>25</td>
<td>1.00</td>
<td>0.07</td>
<td>0.70</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>30</td>
<td>1.00</td>
<td>0.03</td>
<td>0.50</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>35</td>
<td>1.00</td>
<td>0.00</td>
<td>0.47</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Only for the solvers that can successfully find feasible solutions within 5 minutes, we plotted the average quality upper bounds in Figure 2. In this figure, $x$-axis represents the size $k$ of a sequence and $y$-axis represents the average quality upper bounds. As you can see, SAT4J worked better than LD especially when the size of a sequence becomes larger. However, LDWalk and LDIrots, both of which are Lagrangian decomposition solvers having additional upper bounding, worked very fine in these experiments. This additional upper bounding is done by, starting from a feasible solution found by LD, running local search (WalkSAT or Irots) whenever LD updates the current best lower bound. According to this experiment, this technique is clearly effective in finding tighter upper bounds.

In Figure 3, we also plotted, for 30 sequences of size 35, the average quality upper bounds when setting different time bounds over 1, 5, 15, and 30 minutes. This figure clearly shows that, even with increased time bounds, LDWalk and LDIrots still outperformed SAT4J in terms of the quality upper bounds. It
should be pointed out that other WPMAXSAT solvers, WMAXSAT, WALKSAT, and IROTS, still fail to find a feasible solution even with a 30 minute time bound.

6 Conclusions and Future work

In this work, we provided a general framework of DynSAT with decision change costs, and two solutions, Weighted Partial MaxSAT solving and Lagrangian decomposition, for solving the specific case of this problem. Among these solutions, only the Lagrangian decomposition can provide lower bounds for the problem. Furthermore, solvers based on Lagrangian decomposition, LDWALK or LDIROTS, seem to be very promising since they worked very well empirically to find a feasible solution of good quality even when a time bound is tight. Considering the dynamic nature of problem, we believe this property is crucial.

Our future work would include making our ongoing experiments complete, considering another case of this framework, and extending this framework and solutions so that they can handle DynCSP.

References

Fig. 3. Quality upper bounds vs. time bounds (sequences of size 35).

Ordering Intervals: From Qualitative Temporal Constraint Problems to Preference Representation

Souhila Kaci and Cédric Piette
Université Lille-Nord de France, Artois
CRIL, CNRS UMR 8188 - IUT de Lens
F-62307, France
{kaci,piette}@cril.fr

Abstract. In this paper, we highlight the closeness between two frameworks namely qualitative temporal constraint problems and preference representation. More specifically, we focus on Allen’s algebra and show that it can be enriched in order to handle unordered intervals. For this purpose we provide an encoding of relations between intervals in preference representation.

1 Introduction

Representing and reasoning about time and space are important issues in many domains such as natural language processing, geographic information systems, computer vision, robot navigation. Several qualitative approaches have been proposed to represent spatial or temporal entities and their relations. For example Allen [1] represents temporal entities as intervals over a qualitative scale time. Their relations correspond to the different relative positions that may occur between two intervals. In this algebra, an interval is a non-decomposable and abstract entity which is only determined by its start and end points. Given a set of intervals and a set of constraints representing how they should be related, a constraint satisfaction problem consists in finding one or different solutions which define possible positions of the intervals which satisfy the constraints. However in some situations, an interval is not a non-decomposable entity. In fact it is a set of objects which need to be rank-ordered over a qualitative scale time. For example, an interval may be the time allocated to a set of courses which need to be scheduled. So a relation between two intervals is a constraint between two sets of courses to be scheduled each in its associated interval. The scheduling of all courses should respect the constraint between the two intervals. Unfortunately, Allen’s algebra is not expressive enough for representing intervals whose start and end points are not known and which are composed of objects to be rank-ordered. Nevertheless it is a powerful algebra for expressing all possible relations between intervals.

Besides, representing and reasoning about preferences are also important skills in many real-life applications. Several formalisms have been developed for representing preferences ranging from quantitative to qualitative representations. Specifically, (conditional) comparative preference statements for instance “I prefer going to restaurants than going to movies”, “If I am casually dressed, then I prefer going to movies than to restaurants”, etc. have been extensively developed. They offer a simple and natural
way to express users’ preferences. Such statements can be interpreted following several more or less strong semantics, each corresponding to a particular reasoning behavior [9, 5, 4, 2, 13]. A preference for \( p \) over \( q \) is interpreted as a preference for outcomes where \( p \) is true and \( q \) is false over outcomes where \( q \) is true and \( p \) is false. Given a set of comparative preference statements and a given semantics, algorithms have been developed in order to rank-order the set of outcomes in a way these preference statements are satisfied [9, 5, 4, 2, 13], of course when they are not contradictory.

While the above two frameworks have been developed for different purposes, they manifest a close interesting behavior. In this paper, we show that the problem of Allen’s algebra advocated above can be solved by establishing a correspondence between this algebra and preference representation. In fact, an interval composed of unordered objects can be seen as a preference formula in preference representation. Moreover relations between intervals can be encoded by means of one or several comparative preference formulas obeying different semantics.

The remainder of this paper is organized as follows. In the next section, we give a motivating example. We recall in Section 3 Allen’s qualitative algebra. In Section 4 we recall comparative preference statements. Section 5 provides an encoding of relations between intervals in Allen’s algebra using comparative preference statements. Lastly, we conclude with some perspectives.

2 A Motivating Example

A university department proposes a certificate which can be obtained by accomplishing a series of nine courses, identified by \( C_i, i = 1, \ldots, 9 \). All these courses have to be completed by the candidate. They cannot be completed in any order, however. Specifically their scheduling must respect the following constraints:

- There are four primary courses \( \{C_1, C_2, C_3, C_4\} \) and four secondary courses \( \{C_5, C_6, C_7, C_8\} \). All these courses must be pursued but the department is flexible on the order they are scheduled. Nevertheless the department requires that at least one primary course must be scheduled before all secondary courses, all primary courses must finish before secondary courses finish and that primary and secondary courses must overlap.
- Courses \( C_4 \) and \( C_8 \) both introduce basic knowledge which is needed for courses \( C_1 \) and \( C_5 \). \( C_4 \) and \( C_8 \) are too close so pursuing one of them is sufficient to pursue \( C_1 \) and \( C_5 \). Note however that they are not exclusive, i.e. a student is allowed to pursue both courses. Accordingly, at least one of the two former courses has to be achieved before the two latter start. For the same reason, \( C_1 \) or \( C_2 \) must be achieved before \( C_3 \) and \( C_4 \).
- A mandatory internship \( (C_9) \) has to be pursued by each student. This internship can only be achieved after all other courses \( C_1 \ldots C_8 \) are achieved.

The problem now is how to schedule (or rank-order over a qualitative time scale) the nine courses according to the above constraints. For this purpose, a constraint can be seen as a constraint between two intervals, each refers to a set of courses. Let us consider the first constraint. Let \( I(C_1, C_2, C_3, C_4) \) (resp. \( I(C_5, C_6, C_7, C_8) \)) be the time interval...
Fig. 1. Two rank-orderings fulfilling the first constraint in which the courses $C_1$, $C_2$, $C_3$ and $C_4$ (resp. $C_5$, $C_6$, $C_7$ and $C_8$) are achieved. Notice that courses are not strictly ordered over the scale time, i.e. two or more courses may be scheduled at the same time. No restriction is made. Different rank-orderings over $C_1, \ldots, C_9$ are possible to fulfill this constraint. Figure 1 visualizes two of them. Let us note that the rank-ordering in Figure 1(b) involves two courses ($C_5$ and $C_8$) occurring at the same period. It is important to notice that we do not represent the duration of a course in time scale. Consequently, when we say that a course $C_i$ starts/finishes before $C_j$ then this simply means that $C_i$ is scheduled before $C_j$. Note however that these rank-orderings have a common point; the time interval $I(C_5, C_6, C_7, C_8)$ overlaps the time interval $I(C_1, C_2, C_3, C_4)$. Regarding the second constraint, the time interval $I(C_4, C_8)$ (resp. $I(C_1, C_2)$) must start before the time interval $I(C_1, C_5)$ (resp. $I(C_3, C_4)$). Lastly the third constraint states that $I(C_9)$ must start after $I(C_1, \ldots, C_8)$ finishes.

Before we investigate the way the courses can be rank-ordered over a qualitative time scale given these constraints, we need to represent these constraints in a formal framework. Handling constraints over time intervals naturally advocates qualitative interval algebra developed by Allen [1]. In the next section, we recall this algebra and show how our example is represented within it.

3 Allen’s Interval Algebra

Qualitative reasoning deals with constraint-based formalisms for representing, and reasoning with temporal or spatial information over infinite domains. The qualitative aspect is based on relational schemas that abstract from concrete metrical data of entities (e.g.
coordinate positions, time points, distances) by substituting similar topological configurations of entities into one qualitative representation.

For this purpose, various qualitative approaches have been proposed to represent temporal entities and their relations and reason about them (see e.g. [1, 10]). Mainly, a time interval can have thirteen possible positions with respect to another interval. These positions can be combined in a disjunctive way, expressing imprecise knowledge about the way both time intervals are located. Seven of those positions are presented in the first column of Table 1. Reversing the positions of X and Y defines inverse relations, except for “equals” which stays identical. Note that the thirteen possible positions are jointly exhaustive and pairwise disjoint.

Most qualitative resolution methods (e.g. the solver QAT [6]) are mainly based on composition tables. Composition tables encode possible configurations between two entities if sufficient information is available about how these entities are related to a third one. This leads to a consistency algorithm that is fundamental to the efficiency of any search procedure, aiming at finding some coherent instantiation of possible relations, or proving no such instantiation exists.

Therefore the three constraints given in Section 2 can be encoded in Allen’s algebra by means of the following relations between intervals:

- $I_{C_1,C_2,C_3,C_4}$ overlaps $I_{C_5,C_6,C_7,C_8}$
- $(I_{C_1,C_2}$ overlaps $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ precedes $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ meets $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ during $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ finishes $I_{C_3,C_4})$
- $(I_{C_1,C_2}$ overlaps $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ precedes $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ meets $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ during $I_{C_3,C_4})$ or $(I_{C_1,C_2}$ finishes $I_{C_3,C_4})$
- $I_{C_1}$ after $I_{C_1...C_8}$

Table 1. Qualitative Interval Algebra
Unfortunately we cannot go beyond a representation of our example in Allen’s algebra. In fact this algebra is built on abstract and non-decomposable intervals which are identified only by their start and end points. Our example is more complicated than a simple representation by relations between intervals. In fact:

- an interval in our example is a set of discrete and finite objects which need to be rank-ordered,
- an interval may be a “point” (e.g. $I_C$),
- the different intervals may share some objects.

Consequently our motivating example cannot be solved within qualitative interval algebra framework. Nevertheless a connection with preference representation can be established which enables to reason about the constraints.

4 Preference Representation

4.1 Notations & Definitions

Let $V = \{X_1, \ldots, X_h\}$ be a set of $h$ variables. Each variable $X_i$ takes its values in a domain $\text{Dom}(X_i)$. A possible outcome, denoted $\omega$, is the result of assigning a value in $\text{Dom}(X_i)$ to each variable $X_i$ in $V$. $\Omega$ is the set of all possible outcomes. We write $\omega \models \varphi$ when $\omega$ makes the formula $\varphi$ true. We say that $\omega$ satisfies $\varphi$.

An ordering relation $\succeq$ on $X = \{x, y, z, \ldots\}$ is a reflexive binary relation such that $x \succeq y$ stands for $x$ is at least as preferred as $y$. $x$ and $y$ are equally preferred when both $x \succeq y$ and $y \succeq x$ hold. They are incomparable when neither $x \succeq y$ nor $y \succeq x$ holds. A strict ordering relation on $X$ is an irreflexive binary relation such that $x \succ y$ means that $x$ is strictly preferred to $y$. We also say that $x$ dominates $y$. A strict ordering relation $\succ$ can be defined from an ordering relation $\succeq$ as $x \succ y$ if $x \succeq y$ holds but $y \succeq x$ does not. When neither $x \succ y$ nor $y \succ x$ holds, we also write $x \sim y$. $\succeq$ (resp. $\succ$) is a preorder (resp. order) on $X$ if and only if $\succeq$ (resp. $\succ$) is transitive, i.e. if $x \succeq y$ and $y \succeq z$ then $x \succeq z$ (resp. if $x \succ y$ and $y \succ z$ then $x \succ z$). $\succeq$ (resp. $\succ$) is a complete preorder (resp. order) if and only if $\forall x, y \in X$, we have either $x \succeq y$ or $y \succeq x$ (resp. either $x \succ y$ or $y \succ x$).

The set of the best (or undominated) elements of $A \subseteq X$ w.r.t. $\succ$, denoted $\max(A, \succ)$, is defined by $\max(A, \succ) = \{x \in A, \exists y \in A, y \succ x\}$. The set of the worst elements of $A \subseteq X$ w.r.t. $\succ$, denoted $\min(A, \succ)$, is defined by $\min(A, \succ) = \{x \in A, \exists y \in A, x \succ y\}$. The best (resp. worst) elements of $A$ w.r.t. $\succeq$ is $\max(A, \succeq)$ (resp. $\min(A, \succeq)$) where $\succ$ is the strict ordering relation associated to $\succeq$.

A complete preorder $\succeq$ can also be represented by a well ordered partition of $\Omega$. This is an equivalent representation, in the sense that each preorder corresponds to one ordered partition and vice versa. A sequence of sets of outcomes of the form $(E_1, \ldots, E_n)$ is a partition of $\Omega$ if and only if (i) $\forall i, E_i \neq \emptyset$, (ii) $E_1 \cup \cdots \cup E_n = \Omega$, and (iii) $\forall i, j, E_i \cap E_j = \emptyset$ for $i \neq j$.

A partition of $\Omega$ is ordered if and only if it is associated with a preorder $\succeq$ on $\Omega$ such that $(\forall \omega, \omega' \in \Omega$ with $\omega \in E_i, \omega' \in E_j$ we have $i \leq j$ if and only if $\omega \succeq \omega'$).

Complete preorders can be compared following specificity principle [14]:
Definition 1 (Minimal/Maximal specificity principle). Let $\succeq$ and $\succeq'$ be two complete preorders on a set of outcomes $\Omega$ represented by ordered partitions $(E_1, \cdots, E_n)$ and $(E'_1, \cdots, E'_n)$ respectively. We say that $\succeq$ is less specific than $\succeq'$, written as $\succeq \sqsubseteq \succeq'$, iff $\forall \omega \in \Omega$, if $\omega \in E_i$ and $\omega \in E'_j$ then $i \leq j$. $\succeq$ belongs to the set of the least (resp. most) specific preorders among a set of preorders $\mathcal{O}$ if there is no $\succeq'$ in $\mathcal{O}$ such that $\succeq' \sqsubseteq \succeq$ (resp. $\succeq \sqsubseteq \succeq'$), i.e., $\succeq' \sqsubseteq \succeq$ (resp. $\succeq \sqsubseteq \succeq'$) holds but $\succeq \sqsubseteq \succeq'$ (resp. $\succeq' \sqsubseteq \succeq$) does not.

4.2 Comparative preference statements

The elicitation of a preference relation over the set of outcomes is a hard task. Generally, users are not able to compare all possible pairs of outcomes due to their cognitive limitation. Instead users are more willing to express their preferences over partial descriptions of outcomes. Different forms of preference representations have been studied in the literature. For the purpose of this paper, we focus on comparative preference statements [13, 8].

Comparative preference statements have the form “prefer $p$ to $q$”, denoted by $p > q$. These statements may be conditional (or contextual) where the statement $p > q$ only holds if some condition, let us say $r$, holds. Note however that a conditional statement is equivalent to $r \land p > r \land q$. Indeed we will simply focus on statements of the form $p > q$. Although comparative statements offer a simple and natural way to express preferences, they come however with difficulties regarding their interpretation. How should we interpret such statements? For example, given the preference statement “I prefer going to restaurants than to movies”, how do we rank-order outcomes involving going to restaurants and those going to movies? Four semantics have been proposed in the literature:

- **strong preferences**: [5, 13]
  
  any restaurant-related outcome is preferred to any movie-related outcome.

- **optimistic preferences**: [3, 5, 9]
  
  at least one restaurant-related outcome is preferred to all movie-related outcomes.

- **pessimistic preferences**: [2]
  
  at least one movie-related outcome is less preferred to all restaurant-related outcomes.

- **opportunistic preferences**: [11]
  
  at least one restaurant-related outcome is preferred to at least one movie-related outcome.

Comparative preference statements are generally expressed in a strict form (i.e. $>$. For the purpose of this paper, we use both strict and non-strict comparative statements, denoted $>$ and $\geq$ respectively, as already defined in [8].

We define preference of the formula $p$ over the formula $q$ as preference of $p \land \neg q$ over $q \land \neg p$. This is standard and known as von Wright’s expansion principle [12]. Additional clauses may be added for the cases in which sets of outcomes are non-empty, to prevent the satisfiability of preferences like $p > \top$ and $p > \bot$. We do not consider this borderline condition to keep the formal machinery as simple as possible.

We denote the preference of $p$ over $q$ following strong semantics (resp. optimistic, pessimistic, opportunistic) by $p >_{st} q$ (resp. $p >_{opt} q$, $p >_{pes} q$, $p >_{opp} q$). Non-strict preferences are denoted $p \geq_{st} q$, $p \geq_{opt} q$, $p \geq_{pes} q$ and $p \geq_{opp} q$. 

Definition 2. Let $p$ and $q$ be two propositional formulas and $\succeq$ be a preorder on $\Omega$.

- $\geq$ satisfies $p \succ_{st} q$ (resp. $p \geq_{st} q$), denoted $\models p \succ_{st} q$ (resp. $\models p \geq_{st} q$), iff 
  $\forall \omega \models p \land \neg q, \forall \omega' \models q \land \neg p$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).
- $\geq$ satisfies $p \succ_{opt} q$ (resp. $p \geq_{opt} q$), denoted $\models p \succ_{opt} q$ (resp. $\models p \geq_{opt} q$), iff 
  $\exists \omega \models p \land \neg q, \exists \omega' \models q \land \neg p$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).
- $\geq$ satisfies $p \succ_{pes} q$ (resp. $p \geq_{pes} q$), denoted $\models p \succ_{pes} q$ (resp. $\models p \geq_{pes} q$), iff 
  $\exists \omega' \models q \land \neg p, \forall \omega \models p \land \neg q$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).
- $\geq$ satisfies $p \succ_{opp} q$ (resp. $p \geq_{opp} q$), denoted $\models p \succ_{opp} q$ (resp. $\models p \geq_{opp} q$), iff 
  $\exists \omega \models p \land \neg q, \exists \omega' \models q \land \neg p$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).

The following lemma gives the mathematical description of Definition 2 [8]:

Lemma 1. Let $p$ and $q$ be two propositional formulas and $\succeq$ be a preorder on $\Omega$.

- $\geq$ satisfies $p \succ_{st} q$ (resp. $p \geq_{st} q$) iff $\forall \omega \in \min(p \land \neg q, \geq), \forall \omega' \in \max(q \land \neg p, \geq)$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).
- $\geq$ satisfies $p \succ_{opt} q$ (resp. $p \geq_{opt} q$) iff $\forall \omega \in \min(p \land \neg q, \geq), \forall \omega' \in \max(q \land \neg p, \geq)$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).
- $\geq$ satisfies $p \succ_{pes} q$ (resp. $p \geq_{pes} q$) iff $\forall \omega \in \min(p \land \neg q, \geq), \forall \omega' \in \max(q \land \neg p, \geq)$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).
- $\geq$ satisfies $p \succ_{opp} q$ (resp. $p \geq_{opp} q$) iff $\forall \omega \in \min(p \land \neg q, \geq), \forall \omega' \in \max(q \land \neg p, \geq)$ we have $\omega \succ \omega'$ (resp. $\omega \geq \omega'$).

Definition 3 (Preference set). A preference set of type $\succ$, denoted $\mathcal{P}_\succ$, is a set of preferences of the form $\{p_i \succ q_i | i = 1, \ldots, n\}$, where $\succ \in \{ \succ_{st}, \succ_{opt}, \succ_{pes}, \succ_{opp}, \geq_{st}, \geq_{opt}, \geq_{pes}, \geq_{opp} \}$. A complete preorder $\geq$ is a model of $\mathcal{P}_\succ$ if and only if $\geq$ satisfies each preference $p_i \succ q_i$ in $\mathcal{P}_\succ$.

A set $\mathcal{P}_\succ$ is consistent if and only if it admits a model, namely if there exists a preorder which satisfies each preference in $\mathcal{P}_\succ$.

4.3 From comparative preference statements to complete preorders

Two kinds of queries can be drawn when we deal with a set of comparative preference statements: (1) which are the preferred outcomes? (2) is an outcome preferred to another outcome? In many applications (for e.g. database queries), users are more concerned with the preferred outcomes. However preferred outcomes may no longer be feasible so we have look for the immediately less preferred outcomes w.r.t. user’s preferences. In such a case a complete preorder is needed to answer user’s preferences.

Therefore we focus on procedures that derive complete preorders on outcomes. In the next subsections, we recall algorithms which derive a unique complete preorder given a set of preferences of the same type following minimal/maximal specificity principle.

Proposition 1. [8]

- The least specific model of $\mathcal{P} \succ_{opt} \cup \mathcal{P} \geq_{opt}$ (resp. $\mathcal{P} \succ_{st} \cup \mathcal{P} \geq_{st}$) is unique.
- The most specific model of $\mathcal{P} \succ_{pes} \cup \mathcal{P} \geq_{pes}$ (resp. $\mathcal{P} \succ_{st} \cup \mathcal{P} \geq_{st}$) is unique.
The most specific model of $P_{> \text{opt}} \cup P_{\geq \text{opt}}$ (resp. $P_{> \text{opp}} \cup P_{\geq \text{opp}}$) is not unique.
The least specific model of $P_{> \text{pes}} \cup P_{\geq \text{pes}}$ (resp. $P_{> \text{opt}} \cup P_{\geq \text{opt}}$) is not unique.

Note that the notion of minimal/maximal specificity principle for preference sets consisting of non-strict preferences only is not interesting. Indeed the model is trivial in which all outcomes are equivalent.

A unique complete preorder for opportunistic preferences does not exist w.r.t. these principles. We do not consider such preferences in the remainder of this section.

Let $P_\geq = \{s_i : p_i \triangleright q_i | i = 1, \ldots, n\}$ be a preference set with $\triangleright \in \{>_{x} \geq_{x} | x \in \{st, opt, pes\}\}$. Given $P_\geq$, we define a set of pairs on $\Omega$ as follows:

$$C(P_\geq) = \{(T_i = (L(s_i), R(s_i)) | s_i \in P_\geq, i = 1, \ldots, n),\}
$$

where $L(s_i) = \{w | w \in \Omega, w \models p_i \land \neg q_i\}$ and $R(s_i) = \{w | w \in \Omega, w \models \neg p_i \land \neg\neg q_i\}$.

Example 1. Let tonight_plan, how_dressed and way_to_go be three variables such that $\text{Dom}(\text{tonight_plan}) = \{\text{restaurant, movies}\}$, $\text{Dom}(\text{way_to_go}) = \{\text{rent_car, public_transport}\}$ and $\text{Dom}(\text{how_dressed}) = \{\text{casually_dressed, well_dressed}\}$.

We have:

$$\Omega = \{\omega_0 = \text{restaurant-casually_dressed Pública_transporte}, \omega_1 = \text{restaurant-casually_dressed_rent_car}, \omega_2 = \text{casually_dressed_public_transport}, \omega_3 = \text{casually_dressed_rent_car}, \omega_4 = \text{well_dressed_public_transport}, \omega_5 = \text{well_dressed_rent_car}\}$$

We have: $C(P_\geq) = \{T_1 = (\{\omega_0, \omega_1, \omega_2, \omega_3\}, \{\omega_4, \omega_5, \omega_6, \omega_7\}), T_2 = (\{\omega_3, \omega_7\}, \{\omega_0, \omega_1, \omega_2, \omega_3\})\}$.

4.4 Optimistic preferences

As stated in Proposition 1, a unique model of optimistic preferences can be characterized following minimal specificity principle, used in System Z [9]. Following this principle, each outcome is put in the highest possible rank in the model. Therefore the principle is built under the assumption that an outcome is preferred unless there is a reason to state the contrary. Algorithm 1.1 gives the way this preorder is computed. At each step $i$ of the algorithm, we put in $E_i$ outcomes that are not dominated by any other outcome. These outcomes are those which do not appear in the right-hand side of any pair $(L(s_i), R(s_i))$ of $C(P_{> \text{opt}}) \cup C(P_{\geq \text{opt}})$.

Example 2. (Example 1 con’d)

Let $P_{> \text{opt}} = \{s_1 : \text{restaurant} > \text{opt movies}, s_2 : \text{well_dressed} \land \text{rent_car} > \text{opt casually_dressed} \land \text{public_transport}, s_3 : \text{restaurant} \land \text{casually_dressed} > \text{opt restaraunt} \land \text{well_dressed}\}$. We have $E_1 = \{\omega_1\}$. We remove $T_1$ and $T_3$ since $s_1 = \text{restaurant} > \text{opt movies}$ and $s_3 : \text{restaurant} \land \text{casually_dressed} > \text{opt restaraunt} \land \text{well_dressed}$ are satisfied. We get $C(P_{> \text{opt}}) = \{T_2 = (\{\omega_3, \omega_7\}, \{\omega_0, \omega_4\})\}$. Now $E_2 = \{\omega_2, \omega_3, \omega_5, \omega_6, \omega_7\}$. We remove $T_2$ since $s_2 : \text{well_dressed} \land \text{rent_car} > \text{opt casually_dressed} \land \text{public_transport}$ is satisfied. So $C(P_{> \text{opt}}) = \emptyset$. Lastly, $E_3 = \{\omega_0, \omega_4\}$. Indeed
Data: A preference set $\mathcal{P}_{>\text{opt}} \cup \mathcal{P}_{\geq \text{opt}}$.

Result: A complete preorder $\succeq$ on $\Omega$.

begin
\[ l = 0; \]
while $\Omega \neq \emptyset$ do
\[ l = l + 1, k = 1; \]
\[ E_l = \{\omega \in \Omega, \exists (L(s_i), R(s_i)) \in C(\mathcal{P}_{>\text{opt}}), \omega \in R(s_i)\}; \]
/** Handling non-strict optimistic preferences **/
while $k = 1$ do
\[ k = 0; \]
foreach $(L(s_i), R(s_i))$ in $C(\mathcal{P}_{\geq \text{opt}})$ do
\[ \text{if } L(s_i) \cap E_l = \emptyset \text{ and } R(s_i) \cap E_l \neq \emptyset \text{ then} \]
\[ E_l = E_l \setminus R(s_i), k = 1 \]
/** if preferences are inconsistent, the algorithm stops and the current preorder is returned **/
if $E_l = \emptyset$ then return $\succeq = (E_1, \cdots, E_{l-1})$;
$\Omega = \Omega \setminus E_l$;
/** remove satisfied preferences **/
remove $(L(s_i), R(s_i))$ in $C(\mathcal{P}_{>\text{opt}}) \cup C(\mathcal{P}_{\geq \text{opt}})$ where $L(s_i) \cap E_l \neq \emptyset$;
return $\succeq = (E_1, \cdots, E_l)$;
end

Algorithm 1.1: A complete preorder associated to $\mathcal{P}_{>\text{opt}} \cup \mathcal{P}_{\geq \text{opt}}$.

$\succeq = (\{\omega_1\}, \omega_2, \omega_3, \omega_5, \omega_6, \omega_7, \{\omega_0, \omega_4\})$. We can check that each outcome has been put in the highest possible rank in $\succeq$. Therefore, if we push any outcome to a higher rank then the preorder would not satisfy the preference set. For instance, $\succeq' = (\{\omega_1, \omega_5\}, \{\omega_2, \omega_3, \omega_6, \omega_7\}, \{\omega_0, \omega_4\})$ does not satisfy $s_1 : \text{restaurant} >_{\text{opt}} \text{movies}$.

4.5 Pessimistic preferences

The unique model of pessimistic preferences obeys maximal specificity principle. Accordingly, each outcome is put in the lowest possible rank in the preorder. The principle is built under the assumption that an outcome is not preferred unless there is a reason to state the contrary. Due to space limitation, we do not recall the algorithm which computes this preorder, but we illustrate its construction in the next example.

Example 3. (Example 1 con’d)

Let $\mathcal{P}_{>\text{pes}} = \{s_1 : \text{restaurant} >_{\text{pes}} \text{movies}, s_2 : \text{well_dressed} \land \text{rent_car} >_{\text{pes}} \text{casually_dressed} \land \text{public_transport}, s_3 : \text{restaurant} \land \text{casually_dressed} >_{\text{pes}} \text{restaurant} \land \text{well_dressed}\}$. The least preferred outcomes are those which are not preferred to any other outcome. They are outcomes which do not appear in any $L(s_i)$ in $C(\mathcal{P}_{>\text{pes}})$. There is no reason to prefer such outcomes.

We have $E_1 = \{\omega_4, \omega_7, \omega_9\}$. We remove $T_1$ and $T_2$ since $s_1 : \text{restaurant} >_{\text{pes}} \text{movies}$ and $s_2 : \text{well_dressed} \land \text{rent_car} >_{\text{pes}} \text{casually_dressed} \land \text{public_transport}$ are satisfied. We repeat the same reasoning and get $E_2 = \{\omega_2, \omega_3, \omega_7\}$ and $E_3 = \{\omega_0, \omega_1\}$.
So $\succeq = (\{\omega_0, \omega_1\}, \{\omega_2, \omega_3, \omega_7\}, \{\omega_4, \omega_5, \omega_6\})$. We can check that each outcome has been put in the lowest possible rank in the preorder.

4.6 Strong preferences

Strong preferences induce a unique partial order on outcomes. We can use both construction principles used in optimistic and pessimistic preferences to linearize the partial order and compute a unique complete preorder. Again, due to space limitation, we only illustrate the computation of the unique complete preorder following minimal specificity principle.

Example 4. (Example 1 cont’d) Let $P_{st} > st = \{s_1 : \text{restaurant} > st \text{ movies}, s_2 : \text{well dressed} \land \text{rent car} > st \text{ casually dressed} \land \text{public transport}, s_3 : \text{restaurant} \land \text{casually dressed} > st \text{ restaurant} \land \text{well dressed}\}$. There is no complete preorder which satisfies $P_{st}$ so $P_{st} > st$ is inconsistent. This is due to $s_1$ and $s_2$. Following $s_1$, $\omega_0$ is preferred to $\omega_7$ whereas $\omega_7$ is preferred to $\omega_0$ following $s_2$.

Example 5. (Consistent strong preferences)

Let $P_{st} > st = \{s'_1 : \text{restaurant} \land \text{casually dressed} > st \text{ restaurant} \land \text{well dressed}, s'_2 : \text{well dressed} \land \text{rent car} > st \text{ well dressed} \land \text{public transport}, s'_3 : \text{movies} \land \text{well dressed} > st \text{ movies} \land \text{casually dressed}\}$. We have $C(P_{st} > st) = \{T'_1 = (\{\omega_0, \omega_1\}, \{\omega_2, \omega_3\}), T'_2 = (\{\omega_3, \omega_7\}, \{\omega_2, \omega_4\}), T'_3 = (\{\omega_0, \omega_7\}, \{\omega_4, \omega_5\})\}$. As for $P_{opt}$, the most preferred outcomes are those which are not dominated. So $E_1 = (\omega_0, \omega_1, \omega_7)$. Now unlike optimistic preferences, the constraints in $C(P_{opt})$ should be removed or updated. We remove $T'_1$ since $s'_1$ is satisfied. We update $T'_2$ and $T'_3$ by removing $\omega_7$ from $L(s'_{2})$ and $L(s'_{3})$. We get $C(P_{st} > st) = \{T'_1 = (\{\omega_0, \omega_1\}, \{\omega_2, \omega_3\}), T'_2 = (\{\omega_0, \omega_7\}, \{\omega_4, \omega_5\})\}$. We repeat this process and get $E_2 = (\omega_3), E_3 = (\omega_6), E_4 = (\omega_4, \omega_5)$. So $\succeq = (\{\omega_0, \omega_1, \omega_7\}, \{\omega_3\}, \{\omega_2, \omega_6\}, \{\omega_4, \omega_5\})$.

5 From Qualitative Temporal Constraint Problems to Preference Representation

There is a close relationship between Allen’s qualitative algebra and preference representation. As we have seen in Section 3, our motivating example can be represented in Allen’s algebra by means of intervals which are sets of discrete and finite elements. Relations between intervals given in this algebra (see Table 1) can be written in terms of constraints over the start and end of intervals. More precisely, let $\text{min}(I)$ and $\text{max}(I)$ respectively denote the start and the end of an interval $I$. All relations between two intervals are expressed with those notations in the second column of Table 2. For instance, “an interval $X$ precedes an interval $Y$” can be expressed under the form $\text{max}(X) < \text{min}(Y)$. Most relations are encoded under several conditions. For example, “$X$ during $Y$” needs both $\text{max}(Y) > \text{max}(X)$ and $\text{min}(X) > \text{min}(Y)$ to hold. It is important to notice that intervals in our framework are sets of objects which are not ordered. So $\text{min}(I)$ and $\text{max}(I)$ are not known.
Considering intervals as preference formulas, the mathematical encoding of the relations between intervals enables us to build an equivalent encoding in preference representation. The latter is presented in the third column of Table 2. Let $X^*$ be the set of objects composing the interval $X$. Each relation between intervals is translated into different (sets of) preference semantics. For instance, "$X$ during $Y$" is encoded using two preference semantics. Indeed, $Y^*$ must be preferred to $X^*$ in an optimistic way to ensure that at least one outcome in $Y^*$ is preferred to all outcomes in $X^*$. Moreover, $X^*$ must be pessimistically preferred to $Y^*$ to translate that there exists at least one outcome in $Y^*$ that is less preferred to all outcomes in $X^*$. The preference relation over objects of $X$ and $Y$ should be understood here as a scheduling of these objects over a qualitative time scale. The least preferred is an object, the earliest it is.

Our example described in Section 2 is translated into the following preference statements:

1. $\{C_5, C_6, C_7, C_8\} >_{\text{opt}} \{C_1, C_2, C_3, C_4\}$, $\{C_5, C_6, C_7, C_8\} >_{\text{pcs}} \{C_1, C_2, C_3, C_4\}$ and $\{C_1, C_2, C_3, C_4\} >_{\text{opp}} \{C_5, C_6, C_7, C_8\}$

This statement ensures the department ordering constraints (finishing all primary courses before finishing all secondary ones, etc.) are respected.
2. $\{C_1, C_5\} \succ_{pes} \{C_4, C_8\}, \{C_3, C_4\} \succ_{pes} \{C_1, C_2\}$

These two statements ensure that at least one course of a given set (e.g. $\{C_4, C_8\}$) must end before some other (resp. $\{C_1, C_5\}$) starts. As shown in Section 3, this constraint can be respected with several interval positions, which are considered in a disjunctive way. In the preference framework, this corresponds to the pessimistic semantics. In fact we have ($I_{C_1, C_5}$ overlaps $I_{C_1, C_5}$) or ($I_{C_1, C_5}$ precedes $I_{C_1, C_5}$) or ($I_{C_1, C_5}$ meets $I_{C_1, C_5}$) during $I_{C_1, C_5}$ or ($I_{C_1, C_5}$ finishes $I_{C_1, C_5}$) that is encoded in preference representation by ($\{C_4, C_8\} \succ_{opt} \{C_1, C_5\}$ and $\{C_1, C_5\} \succ_{pes} \{C_4, C_8\}$ and $\{C_1, C_5\} \succ_{opt} \{C_4, C_8\}$) or ($\{C_4, C_8\} \geq_{app} \{C_1, C_5\}$ and $\{C_1, C_5\} \geq_{st} \{C_4, C_8\}$) or ($\{C_4, C_8\} \succ_{opt} \{C_1, C_5\}$) or ($\{C_4, C_8\} \geq_{app} \{C_1, C_5\}$ and $\{C_1, C_5\} \geq_{st} \{C_4, C_8\}$) or ($\{C_4, C_8\} \succ_{opt} \{C_4, C_8\}$) which is equivalent to $\{C_1, C_5\} \succ_{pes} \{C_4, C_8\}$.

3. $\{C_5\} \succ_{st} \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$

The internship represented as $\{C_5\}$ must be the last pursued course, which is exactly encoded by the strict semantics.

It is important to notice that our encoding is based on the hypothesis that intervals are “compact” in the sense that they are delimited (their start and end points) by the objects they are composed of. More importantly, we first need to describe relations between intervals in Allen’s algebra before we encode them in preference representation. For example the direct assessment of overlap relation in preference representation framework is not intuitive.

Note however that the above encoding needs to manipulate different semantics simultaneously. Algorithms described in Section 4 treats a set of preferences obeying the same semantics. The authors of [8] have shown that the different semantics can be divided into two categories following specificity principle. Specifically, we have the following result:

**Proposition 2. [8]**

- The least specific model of $P \succ_{st} \cup P \geq_{st} \cup P \succ_{opt} \cup P \geq_{opt}$ is unique.
- The most specific model of $P \succ_{st} \cup P \geq_{st} \cup P \succ_{pes} \cup P \geq_{pes}$ is unique.

Nevertheless the above proposition is not sufficient since the encoding of relations between intervals in preference representation may involve semantics from both subsets identified in the proposition. Moreover in [7] we have shown that the separate use of minimal and maximal specificity principles may be debatable. Algorithm 1.2 gives a way to compute a complete preorder given a set of preference statements obeying any semantics.

The basic idea of our algorithm is that outcomes are not ordered following the semantics of comparative preference statements at hand. Instead, we look for (1) outcomes which dominate other outcomes w.r.t. at least one statement, but are not dominated w.r.t. any statement, and (2) outcomes which do not dominate any other outcomes. The first set of outcomes is the set of the best (i.e. preferred) outcomes and the second is the set of the least preferred outcomes. Once these two sets are computed, the semantics of a given preference statement is used to check whether the statement is satisfied or not.
Fig. 2. A possible scheduling

yet. The algorithm starts each step, or iteration, \( l \) by computing two sets of outcomes \( E_l \) and \( E'_l \) that contain the outcomes that dominate and the ones that do not dominate other outcomes, respectively. Then, the special case of non strict preferences (under the form \( \geq_x \) with \( x \in \{ st, \text{opt, pes, opp} \} \)) is handled by possibly removing outcomes from \( E_l \) and \( E'_l \). If both \( E_l \) and \( E'_l \) sets are empty, the algorithm is stopped and returns the current preorder, since preferences are inconsistent. Otherwise, satisfied preferences are removed. If the current set of remaining outcomes is not empty, the algorithm starts an \( l + 1 \) iteration. Otherwise, the preorder is complete and is returned.

Algorithm 1.2 does not provide a unique model following minimal or maximal specificity principle since it combines both principles. However it is unique following the basic idea of its construction which consists in computing the sets of outcomes which dominate other outcomes and those which do not dominate any other outcomes. Therefore at each step of the algorithm we ensure that \( E_l \) (resp. \( E'_l \)) contains all outcomes that dominate (resp. do not dominate) other outcomes. It is worth noticing that our algorithm also handles opportunistic preferences for which a unique model does not exist following minimal and maximal specificity principles when considered separately. Let us also emphasize that although our algorithm can deal with the four semantics of preferences, all of them are not necessary and our approach can also be used with any subset of those semantics.

Let us consider again our motivating example. We have \( \mathcal{P} = \mathcal{P}_{\geq_{\text{opt}}} \cup \mathcal{P}_{\geq_{\text{pes}}} \cup \mathcal{P}_{\geq_{\text{st}}} \cup \mathcal{P}_{\geq_{\text{opp}}} \) with \( \mathcal{P}_{\geq_{\text{opt}}} = \{ s_1 : \{ C_5, C_6, C_7, C_8 \} \geq_{\text{opt}} \{ C_1, C_2, C_3, C_4 \} \} \), \( \mathcal{P}_{\geq_{\text{pes}}} = \{ s_2 : \{ C_5, C_6, C_7, C_8 \} \geq_{\text{pes}} \{ C_1, C_2, C_3, C_4 \} \), \( s_3 : \{ C_1, C_8 \} \geq_{\text{pes}} \{ C_4, C_8 \} \), \( s_4 : \{ C_3, C_4 \} \geq_{\text{pes}} \{ C_1, C_2 \} \), \( \mathcal{P}_{\geq_{\text{st}}} = \{ s_5 : \{ C_9 \} \geq_{\text{st}} \{ C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8 \} \) and \( \mathcal{P}_{\geq_{\text{opp}}} = \{ s_6 : \{ C_1, C_2, C_3, C_4 \} \geq_{\text{opp}} \{ C_5, C_6, C_7, C_8 \} \} \).

We have \( \mathcal{C}(\mathcal{P}_{\geq_{\text{opt}}}) = \{ (L(s_1), R(s_1)) = \{ \{ C_5, C_6, C_7, C_8 \} \geq_{\text{opt}} \{ C_1, C_2, C_3, C_4 \} \} \} \), \( \mathcal{C}(\mathcal{P}_{\geq_{\text{pes}}}) = \{ (L(s_2), R(s_2)) = \{ \{ C_5, C_6, C_7, C_8 \} \geq_{\text{pes}} \{ C_1, C_2, C_3, C_4 \} \} \), \( L(s_3) = \{ \{ C_1, C_8 \} \}, \{ C_4, C_8 \} \}, \{ L(s_4), R(s_4) = \{ \{ C_3, C_4 \} \}, \{ C_1, C_2 \} \} \} \), \( \mathcal{C}(\mathcal{P}_{\geq_{\text{st}}}) = \{ (L(s_5), R(s_5)) = \{ \{ C_9 \} \} \} \), \( \mathcal{C}(\mathcal{P}_{\geq_{\text{opp}}}) = \{ (L(s_6), R(s_6)) = \{ \{ C_1, C_2, C_3, C_4 \} \}, \{ C_5, C_6, C_7, C_8 \} \} \}. \) Then \( E'_1 = \{ C_2 \} \) and \( E_1 = \{ C_9 \} \). We remove \( (L(s_2), R(s_2)), (L(s_4), R(s_4)) \) and \( (L(s_5), R(s_5)) \) since associated preferences are satisfied. We repeat this process and get \( \succeq = \{ \{ C_9 \}, \{ C_5, C_6, C_7 \}, \{ C_1 \}, \{ C_8 \}, \{ C_3, C_4 \}, \{ C_2 \} \} \). The schedule of the nine courses is visualized in Figure 2. So \( C_2 \) is the first course to be scheduled followed by \( C_3 \) and \( C_4 \) which can take place simultaneously. We can check that all relations between intervals are satisfied.
6 Conclusion

This paper is a first step to highlight the closeness of two frameworks dealing with intervals in different ways. While Allen’s algebra handles intervals only identified by their start and end points, preference representation enables to consider intervals composed of unordered objects which need to be rank-ordered. On the other hand, Allen’s algebra is a suitable framework to represent all possible relations between intervals. Based on these observations, we provided an encoding which permits to consider general intervals whose relations (i.e. relative positions) are expressed in Allen’s algebra. This encoding translates Allen’s relations into preference statements obeying different semantics. Using algorithms developed in preference representation framework, objects at hand are rank-ordered in a way relations between intervals are satisfied. This result leads for a deeper comparison of the two frameworks. We are working in this direction.

Acknowledgements

The authors would like to acknowledge valuable suggestions and advice from anonymous reviewers. This work has been supported by a grant from Université d’Artois (BQR “Utilisation de la programmation par contraintes multi-coeurs pour représenter et manipuler des préférences”).

References

Data: A preference set $P$.
Result: A complete preorder $\succeq$ on $\Omega$.
begin
\[
\begin{aligned}
l = 0; \\
\text{while } \Omega \neq \emptyset \text{ do} \\
\quad l = l + 1; \\
\quad \text{if } C(P_{>st}) \neq \emptyset \text{ or } C(P_{\geq opt}) \neq \emptyset \text{ or } C(P_{>opt}) \neq \emptyset \text{ then} \\
\quad \quad E'_l = \{ \omega | \omega \in \Omega, \exists L(s_i), R(s_i) \in (C(P_{>st}) \cup C(P_{>opt})) \cup C(P_{>opt}), s_i \in L(s_i) \}; \\
\quad \quad E_l = \{ \omega | \omega \in \Omega, \exists L(s_i), R(s_i) \in (C(P_{>st}) \cup C(P_{>opt})) \cup C(P_{>opt}), s_i \in R(s_i) \} \setminus E'_l; \\
\quad \text{else} \\
\quad \quad \text{if } C(P_{>opt}) \neq \emptyset \text{ then} \\
\quad \quad \quad E'_l = \bigcup \{ R(s_i)(L(s_i), R(s_i)) \in (C(P_{>opt}) \cup C(P_{>opt})) \}; \\
\quad \quad \quad E_l = \bigcup \{ L(s_i)(L(s_i), R(s_i)) \in (C(P_{>opt}) \cup C(P_{>opt})) \} \setminus E'_l \\
\quad \quad \text{else } E_l = \Omega \\
\end{aligned}
\]

/*** non-strict strong preferences ***/
\[
k = 1; \\
\text{while } k = 1 \text{ do} \\
\quad k = 0; \\
\quad \text{foreach } (L(s_i), R(s_i)) \in C(P_{>st}) \text{ do} \\
\quad \quad \text{if } L(s_i) \not\subseteq E_l \text{ and } R(s_i) \cap E_l \neq \emptyset \text{ then } E_l = E_l \setminus R(s_i), k = 1; \\
\]

/*** non-strict optimistic and non-strict opportunistic preferences ***/
\[
k = 1; \\
\text{while } k = 1 \text{ do} \\
\quad k = 0; \\
\quad \text{foreach } (s_i, R(s_i)) \in C(P_{\geq opt}) \cup C(P_{>opt}) \text{ do} \\
\quad \quad \text{if } L(s_i) \cap E_l = \emptyset \text{ and } R(s_i) \cap E_l \neq \emptyset \text{ then } E_l = E_l \setminus R(s_i), k = 1; \\
\]

/*** non-strict pessimistic preferences ***/
\[
k = 1; \\
\text{while } k = 1 \text{ do} \\
\quad k = 0; \\
\quad \text{foreach } (L(s_i), R(s_i)) \in C(P_{\geq opt}) \text{ do} \\
\quad \quad \text{if } L(s_i) \cap E'_l \neq \emptyset \text{ and } R(s_i) \cap E'_l = \emptyset \text{ then } E'_l = E'_l \setminus L(s_i), k = 1 \\
\]

/*** inconsistent preferences, the algorithm stops and the current preorder is returned ***/
\[
\text{if } E_l = \emptyset \text{ and } E'_l = \emptyset \text{ then } E_l = \Omega, \text{return } \succeq = (E_1, \ldots, E_l, E'_l, \ldots, E'_l); \\
\Omega = \Omega \setminus (E_l \cup E'_l); \\
\text{substitute } (L(s_i), R(s_i)) \in C(P_{>st}) \cup C(P_{>opt}) \text{ by } (L(s_i) \setminus E_l, R(s_i)); \\
\text{substitute } (L(s_i), R(s_i)) \in C(P_{\geq opt}) \cup C(P_{\geq opt}) \text{ by } (L(s_i) \setminus E'_l, R(s_i) \setminus E'_l); \\
\text{remove satisfied preferences ***/}
\]
\[
\text{remove } (L(s_i), R(s_i)) \in C(P_{>st}) \cup C(P_{>opt}) \text{ where } R(s_i) = \emptyset; \\
\text{remove } (L(s_i), R(s_i)) \in C(P_{\geq opt}) \cup C(P_{\geq opt}) \text{ where } L(s_i) \cap E_l \neq \emptyset; \\
\text{remove } (L(s_i), R(s_i)) \in C(P_{\geq opt}) \cup C(P_{\geq opt}) \text{ where } R(s_i) \cap E'_l \neq \emptyset; \\
\text{remove } (L(s_i), R(s_i)) \in C(P_{\geq opt}) \cup C(P_{\geq opt}) \text{ where } L(s_i) \cap E_l \neq \emptyset \text{ or } R(s_i) \cap E'_l \neq \emptyset; \\
\end{aligned}
\]

return $\succeq = (E_1, \ldots, E_l, E'_l, \ldots, E'_l)$ after removing empty $E_l$ and $E'_l$.
end

Algorithm 1.2: Computation of a complete preorder associated to $P$. 
A decomposition method for valued CSPs

Wady Naanaa¹, Maher Helaoui², and Béchir Ayeb²

¹ Faculty of sciences, University of Monastir, Tunisia
wnaanas@planet.tn

² Pôle de Recherche en Informatique du Centre (PRINCE), Institut Supérieur
d’Informatique et des Techniques de Communication, 4011 Hammam Sousse
maher.helaoui@gmail.com, ayeb.b@yahoo.com

Abstract. Several combinatorial problems can be formulated as valued constraint satisfaction problems (VCSP) where constraints are defined through the use of valuation functions to reflect degrees of coherence. The goal is to find an assignment of values to variables with an overall optimal valuation, a computationally fastidious task especially for large problems.

This article presents a domain decomposition method for solving binary VCSPs based on the class of modular functions. The decomposition process yields subproblems whose valuation functions are exclusively modular. For such VCSPs, we propose a $O(ed^2)$ identification algorithm and a $O(ed)$ solution algorithm.

1 Introduction

Constraint Satisfaction Problems (CSPs) provide a general and convenient framework to model and solve numerous combinatorial problems including planning and scheduling. In the standard CSP framework, the constraints are defined by crisp relations which specify the consistent combinations of values. However, in real-world problems, one may need to express various degrees of consistency in order to reflect the specificity of the problem at hand. The valued constraint satisfaction problems (VCSPs) approach is intended to model such situations [13]. Basically, a VCSP consists of a set of variables taking values in discrete sets called domains. A valued constraint is defined through the use of a valuation function which associates a degree of desirability to each combination of values. The problem is to find an assignment of values to variables from their respective domains with a finite optimal global cost. Finding such an assignment or proving that none exist is known to be an NP-hard task [3].

The computational complexity of finding the optimal solution to a VCSP has been studied in many works and several classes of tractable VCSPs have been identified and solved [4, 7, 3]. VCSPs involving submodular binary valuation functions is one of these classes. By expressing such VCSPs as the problem of finding a minimum weighted cut of a weighted directed graph, it is possible to solve them in $O(n^3d^3)$ steps where $n$ is the number of variables and $d$ is the size of the largest value domain [2].
Nonetheless, VCSP resulting from real situations are not limited to submodular functions. In such cases, can we proceed in a more efficient manner than an exhaustive search while keeping solution optimality? Is there any mean to exploit submodularity in a more restricted context? This paper is extended to contribute to providing a positive answer to these questions.

Motivated by the results obtained on crisp CSPs [11], we present, in this paper, a domain decomposition algorithm for solving binary VCSPs. The decomposition process yields, at each leaf of the search tree, a subproblem whose valuation functions are all modular. Modularity is a stronger condition than submodularity but in compensation, VCSPs involving modular valuation functions only can be identified and solved more efficiently. More precisely, we propose an algorithm which identifies modular VCSPs in $O(ed^2)$ and another algorithm which solves them in $O(ed)$ where $e$ is the number of constraints.

The paper is organized as follows: the next section introduces some definitions and notations. Section 3 is devoted to modular binary functions and their decomposition. In Section 4, we present the decomposition and solution algorithm for binary VCSP. Some experimental results are reported in Section 5. Finally, Section 6 is a brief conclusion.

2 Definitions and notations

In the valued CSP framework (VCSP) [13], the set of possible valuations $E$ is assumed to be a totally ordered set with a minimum ($\bot$) and a maximum ($\top$) element, equipped with a single monotonic binary operation $\oplus$ known as aggregation. These assumptions can be gathered in a valuation structure that can be specified as follows:

Definition 1. A valuation structure is defined as a tuple $S = (E, \oplus, \preceq)$ such that:

- $E$ is a set of valuations;
- $\preceq$ is a total order on $E$;
- $\oplus$ is a binary commutative, associative and monotonic operator.

As an additional requirement, we assume that the valuation structure is fair [5]. That is, for any pair of valuations $\alpha, \beta$ in $E$ such that $\alpha \preceq \beta$, there exist a unique maximum difference between $\beta$ and $\alpha$, denoted $\beta \ominus \alpha$, such that $\alpha \oplus (\beta \ominus \alpha) = \beta$.

In concrete terms, fairness guaranties that the aggregation operator $\oplus$ has a partial inverse denoted $\ominus$ which allows the substraction of any valuation from any larger valuation. Moreover, we assume that $\ominus$ is strictly monotonic, that is for all $\alpha, \beta, \gamma$ in $E$ such that $\alpha \prec \beta$ and $\gamma \neq \top$, we have $\alpha \ominus \gamma \prec \beta \ominus \gamma$.

Once the valuation structure is specified, we define the valued constraint satisfaction problem (VCSP) we are considered with as follows:
Definition 2. A valued constraint satisfaction problem (VCSP) is defined by a tuple \((X, D, C, S)\) such that:

- \(X\) is a finite set of variables.
- \(D\) is a set of value domains;
- \(C\) is a set of soft constraints. Every soft constraint \(c\) is a pair \(\langle \sigma, \phi \rangle\) where \(\sigma \subseteq X\) is the scope of \(c\) and \(\phi\) is a function of arity \(|\sigma|\) with valuations in \(E\);
- \(S \equiv (E, \oplus, \preceq)\) is a fair valuation structure such that \(\oplus\) is strictly monotonic.

The \textit{arity} of a soft constraint is the size of its scope. The \textit{arity} of a problem is the maximum arity over its constraints. In this paper, we focus on \textit{binary} VCSPs, that is, VCSPs involving constraints whose arities do not exceed two. Hence, the set of constraints \(C\) can be expressed as the union of two subsets: \(C^1\) the subset of unary soft constraints and \(C^2\) the subset of binary soft constraints.

We also need to distinguish the following subset of constraints which orders the scopes of the binary constraints

\[ \bar{C} = C^1 \cup \{c_{i,j} \in C^2 \mid x_i \rightarrow x_j \} \]

where \(\rightarrow\) denotes any total order on the variables.

A unary soft constraint over variable \(x_i\) is denoted \(c_i\) and its valuation function \(\phi_i\). We assume that every variable is in the scope of a unary soft constraint. Similarly, a binary soft constraint over variables \(x_i\) and \(x_j\) is denoted \(c_{i,j}\) and its valuation function \(\phi_{i,j}\).

As in [2], we shall assume throughout this paper that the set of valuations \(E\) is the set of nonnegative integers (or reals) together with infinity (\(\infty\)). The latter element will be used to designate a total incoherence whereas 0 will indicate a total coherence. Hence, the total order is \(\preceq\), the aggregation operator is the sum (+) and its partial inverse is subtraction (−). We obtain, therefore, the valuation structure \((E, +, \preceq)\) which possesses all the properties outlined above.

The valuation of an assignment \(t\) to a subset of variables \(V \subseteq X\) is obtained by

\[ \phi_P(t) = \sum_{c \in \bar{C}, \sigma_c \subseteq V} \phi_c(t[\sigma_c]) \]

where \(t[U]\) denotes the projection of \(t\) on the variables of \(U \subseteq V\) and \(\bar{C}\) is defined with respect to any total order on the variables. Hence, an overall optimal solution for a VCSP instance \(P = (X, D, C, S)\) is an \(n\)-tuple \(t\) such that \(n = |X|\) and \(\phi_P(t)\) is minimal over all possible \(n\)-tuples.

3 Modular binary functions

In this section, we define modular binary functions, highlight some of their properties and show that they can be expressed as the sum of two unary functions.
3.1 Definitions and properties

Denote $B$ the set of all binary functions $f : D^2 \rightarrow E$, where $D$ is a finite set and $E$ is the universe of a fair and strictly monotonic structure $(E, +, \leq)$.

**Definition 3.** A function $f \in B$ is modular if, for all $u, x, v, y \in D$, we have

$$f(u, v) + f(x, y) = f(u, y) + f(x, v)$$

(2)

Let us denote $M$ the subset of all modular functions of $B$ and let $f$ and $f^T$ be two binary functions of $B$ such that for all $u, v \in D$, we have $f(u, v) = f^T(v, u)$. Then, we can easily infer from (2) that $f$ is in $M$ if and only if $f^T$ is in $M$.

From (2), we can also infer that, for any $f$ in $M$ and any $v, y \in D$, the following statement is true

$$\exists u \in D, f(u, v) < f(u, y) \implies \forall u \in D, f(u, v) \leq f(u, y)$$

(3)

This suggests that the comparison between $f(u, v)$ and $f(u, y)$ does not depend of $u$. Moreover, since $\leq$ is a total order over $E$, it induces a total preorder over the elements of $D$ denoted by $\leq_f$ and defined as follows

$$v \leq_f y \iff \forall u \in D, f(u, v) \leq f(u, y), \text{ for all } u \in D$$

(4)

For $u, v, y \in D$ such that $f(u, y) \leq f(u, v)$ denote by $\delta_{v,y}$ the difference $f(u, v) - f(u, y)$. From (2), it is easy to deduce that $\delta_{v,y}$ also does not depend on $u$, that is

$$\delta_{v,y} = f(u, v) - f(u, y), \text{ for all } u \in D$$

(5)

3.2 Decomposing modular binary functions

In the followings, we show that a modular binary function can be efficiently reduced to the sum of two unary functions. Hence, throughout this paragraph, $f$ will designate a modular function. We also assume that

$$\forall u \in D, \exists v \in D, f(u, v) < \infty$$

(6)

**Property 1.** Let $m$ be one of the minimal elements of $D$ according to $\leq_f$ then $m$ can be computed in $|D|$ steps.

**Proof.** $m$ is computed in $|D|$ steps by setting the first argument of $f$ to any element of $D$, that is

$$m = \arg \min_{v \in D} f(u, v), \text{ for any } u \in D$$

(7)

which means that $m$ does not depend on $u$. Indeed, suppose that there exists $u' \in D$ such that $\arg \min_{v \in D} f(u', v) = m' \neq m$ and that $f(u', m') < f(u', m)$. By (7), we have $f(u, m) \leq f(u, m')$. Moreover, by (6), we have $f(u, m) < \infty$. We get, therefore, $f(u, m) + f(u', m') < f(u, m') + f(u', m)$ since $+$ is strictly monotonic. This contradicts the fact that $f$ is modular.
Consider the unary functions \( f_\uparrow \) and \( f_\downarrow \) defined over \( D \) with valuations in \( E \) as

\[
\begin{align*}
f_\uparrow(v) &= f(v, m) \\ f_\downarrow(v) &= f(x, v) - f(x, m)
\end{align*}
\]  

(8) 

where \( x \) is any element of \( D \). Note that \( f_\downarrow(v) \) exists for all \( v \in D \) since by (7), we have \( f(x, m) \leq f(x, v) \). Moreover, according to (5), \( f_\downarrow(v) \) does not depend on the choice of \( x \).

**Property 2.** \( f \) is modular if and only if \( f(u, v) = f_\uparrow(u) + f_\downarrow(v) \), for all \( u, v \in D \).

**Proof.** By (2), for any \( u, v, x \) in \( D \) and \( m \) defined as in (7), we have

\[
\begin{align*}
f(u, v) + f(x, m) &= f(u, m) + f(x, v) \\ f(u, v) &= f_\uparrow(u) + f(x, v) - f(x, m) \\ f(u, v) &= f_\uparrow(u) + f_\downarrow(v)
\end{align*}
\]

(10)

Conversely, if \( f(u, v) = f_\uparrow(u) + f_\downarrow(v) \), for all \( u, v \in D \) then, we have

\[
\begin{align*}
f(u, v) + f(x, y) &= f_\uparrow(u) + f_\downarrow(v) + f_\uparrow(x) + f_\downarrow(y) \\ &= f(u, y) + f(x, v)
\end{align*}
\]

which means that \( f \) is modular.

**Decompose** is an algorithm which takes as inputs a modular binary function \( f \) over \( D \) and an element \( x \) in \( D \) and computes the unary functions \( f_\uparrow \) and \( f_\downarrow \) according to (8) and (9).

\[
\text{Function DECOMPOSE}(f, D, x) : f_\uparrow, f_\downarrow
\]

\[
m \leftarrow \arg \min_{u \in D} f(x, u)
\]

\[
\text{for } u \in D \text{ do}
\]

\[
\text{\quad } f_\uparrow(u) \leftarrow f(u, m)
\]

\[
\text{\textbf{for } u \in D \text{ do}}
\]

\[
\text{\quad } f_\downarrow(u) \leftarrow f(x, u) - f(x, m)
\]

**Property 3.** Let \( f \) be in \( \mathcal{M} \) then algorithm **DECOMPOSE** computes two unary functions verifying Property 2 in \( O(|D|) \) steps.

**Proof.** This can be easily deduced from the pseudo-code of **DECOMPOSE**.

**Property 4.** A function in \( \mathcal{B} \) can be identified as modular in \( O(|D|^2) \) steps.

**Proof.** According to Property 2, it suffices to apply **DECOMPOSE** which computes \( f_\uparrow \) and \( f_\downarrow \) in \( O(|D|) \) steps and then check the \( |D|^2 \) equations given by (10).
3.3 Decomposition into modular binary functions

In this paragraph, we show that, given a binary function $f$ in $B$ which is not necessarily modular, it is possible to partition $D$ into subsets so that by restricting the first argument of $f$ to each of these subsets, we obtain a family of modular binary functions.

**Definition 4.** Let $f$ be in $B$ and let $u, x$ be two distinct elements of $D$. We say that $\{u, x\}$ is a modular pair with regard to $f$ (notation $u \equiv_f x$) iff the restriction of $f$ to $\{u, x\} \times D$ is modular. More precisely, we have

$$u \equiv_f x \iff \forall v, y \in D, f(u, v) + f(x, y) = f(u, y) + f(x, v)$$

Note that $f \in M \iff u \equiv_f x, \forall u, x \in D \quad (11)$

It is easy to see that $\equiv_f$ is an equivalence relation.

Algorithm **EquivClass** given below takes a binary function in $B$ and an element $x$ in $D$ as arguments and computes the equivalence class of $x$, (denoted $\bar{x}$ in the pseudo-code), in accordance with Definition 4.

**Function** EQUCLASS($f, D, x$) : $\bar{x}$

$f_1, f_1$ ← DECOMPOSE($f, D, x$)

$\bar{x} \leftarrow \emptyset$

for $u \in D$

equiv ← true

for $v \in D$

if $f(u, v) \neq f_1(u) + f_1(v)$ then

equiv ← false

break

if equiv then

$\bar{x} \leftarrow \bar{x} \cup \{u\}$

**Property 5.** For $f$ in $B$, algorithm **EquivClass** computes the equivalence class of its argument $x$ according to Definition 4 in $O(|D|^2)$ steps.

Proof. we can see from the pseudo-code of **EquivClass** that an element $u$ is inserted in $\bar{x}$ iff $f(u, v) = f_1(u) + f_1(v)$ for all $v \in D$. First, we verify that $x$ itself is in $\bar{x}$. This is true since, for all $v \in D$, we have

$$f_1(x) + f_1(v) = f(x, m) + f(x, v) - f(x, m) = f(x, v)$$
Now, we prove that if \( u \equiv_f x \) then \( u \in \bar{x} \). Since \( u \equiv_f x \), then, for all \( v \in D \), we have
\[
f(u, v) + f(x, m) = f(u, m) + f(x, v)
\]
for all \( u, x \in [i] \) and \( v, y \in I \), we have
\[
f(u, v) + f(x, y) = f(u, y) + f(x, v)
\]
Hence, each \( f_i, i: 0, \ldots, k - 1 \) is modular.

The latter equation is equivalent to \( u \in \bar{x} \).

Next, we prove the converse. Since \( u, x \in \bar{x} \), then, for all \( v \in D \), we have
\[
f(u, v) = f_1(u) + f_1(v)
\]
and \( f(x, v) = f_1(x) + f_1(v) \). By (8) and (9) and for all \( v, y \in D \), we obtain the following
\[
f(u, v) + f(x, y) = f(u, y) + f(x, v)
\]
The latter equation is equivalent to \( u \equiv_f x \).

Finally, we can easily see from the pseudo-code of algorithm \textsc{EquivClass} that it runs in \( O(|D|^2) \) steps.

\textbf{Property 6.} Let \( f \) be a modular function and let \( D_1, \ldots, D_k \) denote the partition of \( D \) induced by \( \equiv_f \). Then the restriction of \( f \) to \( D_s \times D \) is modular for \( s = 1, \ldots, k \).

\textbf{Proof.} The property results immediately from Definition 4 and the fact that each \( D_s \) is an equivalence class of \( \equiv_f \).

\textbf{Example 1.} For any finite subset \( I \) of \( \mathbb{N} \) and any positive integer \( k \), define the binary function \( f: I^2 \rightarrow \{0, \ldots, k - 1\} \) by \( f(u, v) = (u - v) \mod k \). \( f \) is not modular for \( k > 1 \), since we have
\[
0 = f(v, v) + f(v + 1, v + 1) \neq f(v, v + 1) + f(v + 1, v) = 2
\]
Now, consider the functions \( f_i: [i] \times I \rightarrow \{0, \ldots, k - 1\}, \ i: 0, \ldots, k - 1 \) defined as the restrictions of \( f \) to \([i] \times I \) where \([i]\) denotes the congruence classes of \( i \) modulo \( k \). Arithmetic congruence theory says that if \( u, x \) and \( v, y \) are two congruent modulo \( k \) pairs then so is \( u - v \) and \( x - y \), that is
\[
(u - v) \mod k = (x - y) \mod k
\]
By taking \( y = v \), we obtain \( f_i(u, v) = f_i(x, v) \) for all congruent pair \( u, x \in [i], v \in I \). It follows that, for all \( u, x \in [i] \) and \( v, y \in I \), we have \( f(u, v) + f(x, y) = f(u, y) + f(x, v) \). Hence, each \( f_i, i: 0, \ldots, k - 1 \) is modular.
In what follows, we shall write \( \text{vCSP}(L) \) to denote the class of binary VCSPs involving valuation functions in \( L \).

**Property 7.** Denote \( U \) the set of all \( D \rightarrow E \) unary functions. The time complexity of \( \text{vCSP}(U) \) is \( O(nd) \).

**Proof.** An algorithm that solves instances in \( \text{vCSP}(U) \) returns the \( n \)-tuple \( t \) that verifies

\[
t_i = \arg \min_{v \in D_i} \phi_i(v), \quad i: 1, \ldots, n
\]

This can be computed in \( O(nd) \) steps. Moreover, we have \( \phi_P(t) = \sum_{x_i \in X} \phi_i(t_i) \) is minimum since the aggregation operator + is monotone.

Below, we describe how to decompose binary VCSP instances into a set of instances in \( \text{vCSP}(U \cup M) \).

### 4.1 Modular VCSP

Let \( c_{i,j} \) be a binary constraint whose valuation function \( \phi_{i,j} \) is in \( M \). According to Section 3, \( \phi_{i,j} \) can be expressed as the conjunction of two unary constraints, denoted \( c_{i \downarrow j} \) and \( c_{i \uparrow j} \), whose scopes are respectively \( x_i \) and \( x_j \) and whose valuation functions \( \phi_{i \uparrow j} \) and \( \phi_{i \downarrow j} \) are defined in accordance with (8) and (9) and verify

\[
\phi_{i,j} = \phi_{i \uparrow j} + \phi_{i \downarrow j}
\]

**Definition 5.** Let \( \mathcal{P} = (X, D, C, S) \) be a VCSP instance of \( \text{vCSP}(U \cup M) \). The unary reduction of \( \mathcal{P} \) with respect to a total order \( \rightarrow \) on \( X \) is the VCSP \( \mathcal{P}' = (X, D, C', S) \) such that

\[
C' = \{ c'_i \mid \phi'_i = \phi_i + \sum_{c_{i,j} \in C} \phi_{i \uparrow j} + \sum_{c_{j,i} \in C} \phi_{j \downarrow i} \}
\]

Note that every unary reduction is necessarily in \( \text{vCSP}(U) \). Unary reductions depend on the total order with regard to which they are defined. Indeed, different orders imply different \( C' \) sets, which may lead to different unary reductions as it can be seen from the following example.

**Example 2.** Consider a VCSP defined on two variables \( x_1 \) and \( x_2 \) with bi-valued domains. The VCSP involves two unary soft constraints \( c_1 \) and \( c_2 \) and a binary constraint \( \phi_{1,2} \) or its transpose \( \phi_{2,1} \). The various valuation functions are

\[
\phi_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \quad \phi_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad \phi_{1,2} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}, \quad \phi_{2,1} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}
\]
If we consider the order $x_1 \rightarrow x_2$, then $\vec{C} = \{c_1, c_2, c_1, 2\}$ and we obtain

$\phi_{1|2} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \phi_{1|2} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \phi'_1 = \phi_1 + \phi_{1|2} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, \phi'_2 = \phi_2 + \phi_{1|2} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$

Now, if we consider the reverse order, then $\vec{C} = \{c_1, c_2, c_2, 1\}$ and we obtain

$\phi_{2|1} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \phi_{2|1} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \phi'_1 = \phi_1 + \phi_{2|1} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}, \phi'_2 = \phi_2 + \phi_{2|1} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$

Thus, the resulting reductions differ despite the symmetry of the original binary constraint.

**Property 8.** Let $P = (X, D, C, S)$ be an instance of $\nu$CSP($U \cup M$) then the unary reduction of $P$ can be computed in $O(ed)$, where $e = |C|$ and $d$ is the size of the largest value domain of $P$.

**Proof.** It suffices to substitute two unary constraints to each of the binary constraints of $P$ as suggested by Property 2. This task can be done in $O(d)$ for one constraint. As there are $e$ constraints in $P$, $O(ed)$ steps are needed.

**Property 9.** Any $P$ in $\nu$CSP($U \cup M$) is equivalent to its unary reduction.

**Proof.** We prove that the valuation of an assignment $t$ of a subset of variables $V \subseteq X$ is the same by $P$ and by its unary reduction $P'$, that is, $\phi_P(t) = \phi_{P'}(t)$. By (1),(13) and Definition 5, we obtain the following

\[
\phi_P(t) = \sum_{x_i \in V} \phi_i(t[x_i]) + \sum_{c_i,j \in \vec{C},x_i,x_j \in V} \phi_{i,j}(t[x_i],x_j) \\
= \sum_{x_i \in V} \phi_i(t[x_i]) + \sum_{c_i,j \in \vec{C},x_i,x_j \in V} \phi_{i,j}(t[x_i]) + \phi_{i,j}(t[x_j]) \\
= \sum_{x_i \in V} \phi_i(t[x_i]) + \sum_{c_i,j \in \vec{C},x_i,x_j \in V} \phi_{i,j}(t[x_i]) + \sum_{c_i,j \in \vec{C},x_i,x_j \in V} \phi_{i,j}(t[x_j]) \\
= \sum_{x_i \in V} \left[ \phi_i(t[x_i]) + \sum_{c_i,j \in \vec{C},x_i,x_j \in V} \phi_{i,j}(t[x_i]) + \sum_{c_i,j \in \vec{C},x_i,x_j \in V} \phi_{j,i}(t[x_i]) \right] \\
= \sum_{x_i \in V} \phi'_i(t[x_i]) \\
\phi_{P'}(t)
\]
4.2 Domain decomposition

Our decomposition strategy is essentially based on Definition 4 which is translated in the framework of VCSP as follows:

**Definition 6.** Let \( P = (X, D, C, S) \) be a binary VCSP with a total order \( \rightarrow \) over \( X \). \{u, v\} \( \subseteq D_i \) is a modular pair of \( P \) with respect to \( \rightarrow \), (notation \( u \equiv_{\beta} v \)), iff for all \( c_{i,j} \in \bar{C} \), we have \( u \equiv_{\phi_{i,j}} v \).

**Property 10.** \( \frac{i}{\beta} \equiv_{\beta} \) is an equivalence relation over \( D_i \), for \( i : 1, \ldots, n \).

**Proof.** First, notice that \( \equiv_{\beta} \) is the intersection of the \( \equiv_{\phi_{i,j}} \)'s such that \( c_{i,j} \in \bar{C} \). These are all equivalence relations. Since the intersection of a set of equivalence relations is an equivalence relation, the result follows.

**Property 11.** Let \( P \) be a binary VCSP and \( \rightarrow \) a total order over its variables. If each \( \frac{i}{\beta} \equiv_{\beta} \) yields a single equivalence class over \( D_i \) then \( P \) is in \( vCSP(U \cup M) \).

**Proof.** Having a single equivalence class induced by \( \frac{i}{\beta} \equiv_{\beta} \), \( i : 1, \ldots, n \) implies that the values of \( D_i \), \( i: 1, \ldots, n \) are all pairwise modular. By \( (11) \), we obtain that all the \( \phi_{i,j} \)'s such that \( x_i \rightarrow x_j \) are modular. Hence all the valuation functions of \( P \) are modular and then \( P \) is in \( vCSP(U \cup M) \).

However, the conditions required by Property 11 are too strong to be satisfied in real-world problems. For this reason, we propose to take advantage of modularity by decomposing the VCSPs at hand. Our decomposition schema relies on the following property:

**Property 12.** Let \( P \) be a binary VCSP, \( \rightarrow \) a total order over its variables and \( P' \) a binary VCSP obtained by reducing the domain of every variable \( x_i \) in \( P \) to an equivalence class of \( \frac{i}{\beta} \). Then \( P' \) is in \( vCSP(U \cup M) \).

**Proof.** This is obtained by observing that in every equivalence class of \( \frac{i}{\beta} \equiv_{\beta} \), \( i : 1, \ldots, n \), the values are all pairwise modular with respect to \( \rightarrow \). The result follows by applying Property 11.

4.3 Local consistency

It is well known that local consistency plays an important role in efficient constraint solving. In [8], the author defined star arc consistency (AC*) and proposed a \( O(ed^2) \) algorithm for enforcing it in weighted CSP (WCSP). The WCSP framework uses a valuation structure \( S(k) = ([0, 1, \ldots, k], \oplus, \leq) \), where \( k \) is a predefined constant. The aggregation operator \( \oplus \) is defined as \( \alpha \oplus \beta = \min\{k, a + b\} \) and its partial inverse \( \ominus \) is defined for \( \alpha \leq \beta \) as \( \beta \ominus \alpha = k \) if \( \beta = k \) and \( \beta - \alpha \) otherwise.
Contrary to the valuation structure used in the decomposition schema, which correspond to the case $k = \infty$, $S(k)$ is not strictly monotonic for $k < \infty$. This is necessary to enable value deletion even with finite costs. In practice, i.e., when solving WCSP instances, $k$ is set to the cost of the optimal solution found so far. Hence, when integrated into a solution algorithm, AC* may use several valuation structures, differing by the value of $k$, as search progresses.

The basic operations performed during establishing star arc consistency in binary WCSP are node pruning and cost projection from binary to unary constraints. Neither node pruning nor cost projection can add binary constraints to the problem on which AC* is enforced. This point is essential for our decomposition algorithm. Another crucial issue is that AC* computes a lower bound of the optimal solution cost and when the VCSP is unary, this lower bound corresponds exactly to the cost of the optimal solution.

4.4 The solution algorithm

The ModMAC* algorithm described below is a variation of the MAC* algorithm [8]. It progressively decomposes the initial VCSP into subproblems in the vCSP($\mathcal{U} \cup \mathcal{M}$) class based on modular value pairs.

The variable order according to which modular value pairs are determined is dynamic, that is, it may vary during search. This order is implicitly determined by the list of the remaining variables $Y$ whose elements are assumed to come after those of $X - Y$ in the order.

The main function in the decomposition process is ModEquivClass which is called at each node of the search tree. It outputs the equivalence class $(D_{i,v})$ of the selected value $(x_i, v)$ (see line 2 in the pseudo-code). It is based on EquivClass (see Section 3) which is called for every valuation function $\phi_{i,j}$ of $P$ such that $x_i$ is the current variable and $x_j$ is a past variable, i.e., a variable in $X - Y$. The domain of $x_i$ is therefore reduced to $D_{i,v}$ (line 3).

Next, the resulting subproblem is further transformed in order to progressively obtain a subproblem in the vCSP($\mathcal{U}$) class. This step is achieved by the ModDecompose function (line 4). The latter is also executed at each node of the search tree. It calls Decompose for every binary valuation function $\phi_{i,j}$ such that $x_i$ is the current variable and $x_j$ is a past variable. The role of this procedure is to replace $\phi_{i,j}$ by two unary valuation functions.

Then, star arc-consistency is restored. For the algorithm to proceed deeper in search, the star arc-consistent VCSP must not contain an empty domain. A recursive call is therefore performed to consider the remaining variables. This call returns the cost ($c^*$) of the best solution found by reducing the domain of $x_i$ to $D_{i,v}$. Next, the algorithm cancels the effect of reducing $D_i$ to $D_{i,v}$ and discards the elements of $D_{i,v}$ from $D_i$. It restores again the star arc-consistency of the resulting subproblem and checks if there is no empty domain. If it is the case, the algorithm performs a second recursive call which returns the cost of the best solution of the problem received as input.
Every time the algorithm succeeds to instantiate all the variables, we obtain a VCSP whose soft constraints are all unary, i.e., in the vCSP(U) class. The last call to AC∗ along such a successful path provides, therefore, the cost of the current optimal solution in polynomial time. The latter cost is saved in c∗ during the subsequent recursive call (line 1).

Algorithm ModMac∗(P = (X, D, C, S), Y, lb) : c∗
1 if Y = ∅ then c∗ ← lb
else
   xi ← SELECT(Y)
   v ← SELECT(Di)
   2 Di,v ← ModEquivClass(P, Y, Di, v)
   3 Di ← Di,v
   4 ModDecompose(P, xi)
   lb ← AC∗(P)
   if ∅ /∈ D then
      5 c∗ ← ModMac∗(P, Y − {xi}, lb)
      Restore(D)
      Di ← Di − Di,v
      lb ← AC∗(P)
   if ∅ /∈ D then
      6 c∗ ← ModMac∗(P, Y, lb)
      Restore(D)

5 Experimental Results

The problems investigated in our experiments are random binary Max-CSP, radio link frequency assignment problems (RLFAP) and a planning problems. We compared the ModMac∗ with Mac∗. The soft arc-consistency algorithm underlying both algorithms is AC∗ [8]. The initial upper bounds used by both solvers are those given in the various benchmark files. The variable ordering heuristic used by both algorithms is min-domain/deg. For value ordering, both algorithms choose the values having the lower projected unary cost. The latter cost is automatically computed by applying AC∗. The evaluation criteria are the number of expanded nodes, the CPU time in seconds and the cost of the best solution found. All algorithms were implemented in C++. They were run in a 1.8 GHZ PC having 3 Gb of RAM.

Random binary Max-CSPs: for random binary Max-CSPs, we experimented on the instances described in [9] and generated according to the well-known
four-parameters model [12]. The goal is to find a complete assignment with
a minimum number of unsatisfied constraints in an overconstrained CSP. The
problem is easily expressed as VCSP using, as binary costs, 0 for compatible
pairs of values and 1 for the incompatible ones. We experimented on some of
the instances available at [10]. These are instances obtained by considering two
levels of constraint graph density: sparse (S) and dense (D) and two levels of
constraint tightness: loose (L) and tight (T). The combination of these classes
gives 4 different VCSP classes: SL, ST, DL, DT. We precisely considered ten
instances coming from the four classes. Most of instances are rather hard to
solve (see Figure 1). They have in common a uniform value domain of size ten.
The number of variables, which is indicated immediatly after the initials of the
class in the instance name, varies from 25 to 50.

Figure 1 provides the results obtained by ModMac∗ and Mac∗. Two hours
of execution time where devoted to each instance. We reported the number of
expanded nodes, the termination time and the cost of the best solution found.
For these unstructured instances, none of the two algorithms systematically out-
performs the other. Nonetheless, we notice that ModMac∗ is more competitive
over sparsely constrained VCSPs involving loose relations.

Radio link frequency assignment problem (RLFAP): these are instances
derived from a real-world problem, namely the celar RLFAP problem [1] which
has the characteristic of being sparsely constrained. We precisely experimented
on ten instances from those available at [10], five of which are extracted from
the celar6 problem and denoted celar6-sub0 to 4. The other five instances are
extracted from the celar7 problem and are denoted celar7-sub0 to 4 (see Figure
2). Each of these instances involves 16 variables and a maximum domain size of
44 values.

By examining Figure 2, we notice that when both algorithms terminate,
ModMac∗ is always faster than Mac∗. Moreover, on the other instances, which
are harder to solve, ModMac∗ has systematically found a solution which is, at
least, as good as the one found by Mac∗. Globally, we can say that, contrary
to the random instances, ModMac∗ outperforms Mac∗ on all these structured
instances.

Planning problems: we precisely considered the driverlog transportation prob-
lem. This problem involves a set of drivers driving trucks and moving packages
from a starting location to a goal destination in an efficient way. More details on
this problem can be found in [6]. The ten instances on which we experimented
are modeled as moderately constrained WCSP since about half of the possible
constraints are present. Each of the instances involves hundreds of variables and
a maximum value domain size of eleven.

Figure 3 shows that ModMac∗ performed better than Mac∗ on all but one
of the instances (driverlog04bc).
<table>
<thead>
<tr>
<th>instance</th>
<th>expanded nodes</th>
<th>time (in sec.)</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mac</td>
<td>ModMac</td>
<td>Mac</td>
</tr>
<tr>
<td>SL50-10-10-60-1</td>
<td>6097</td>
<td>6369</td>
<td>–</td>
</tr>
<tr>
<td>SL50-10-10-60-2</td>
<td>6099</td>
<td>6726</td>
<td>–</td>
</tr>
<tr>
<td>SL40-10-13-60-1</td>
<td>1059</td>
<td>762</td>
<td>1063</td>
</tr>
<tr>
<td>SL40-10-13-60-2</td>
<td>2261</td>
<td>1837</td>
<td>2298</td>
</tr>
<tr>
<td>DL30-10-25-48-1</td>
<td>67</td>
<td>215</td>
<td>66</td>
</tr>
<tr>
<td>DL30-10-25-48-2</td>
<td>915</td>
<td>181</td>
<td>864</td>
</tr>
<tr>
<td>ST25-10-21-85-1</td>
<td>527</td>
<td>1206</td>
<td>317</td>
</tr>
<tr>
<td>ST25-10-21-85-2</td>
<td>604</td>
<td>3852</td>
<td>356</td>
</tr>
<tr>
<td>DT25-10-25-87-1</td>
<td>1871</td>
<td>12387</td>
<td>1222</td>
</tr>
<tr>
<td>DT25-10-25-87-2</td>
<td>4332</td>
<td>9833</td>
<td>2714</td>
</tr>
</tbody>
</table>

Fig. 1. Experimental results obtained by Mac* and ModMac* on random binary Max-CSP instances. We reported the number of nodes expanded (in thousands) within 2 hours, the CPU time and the cost of the best solution found. A dash indicates that the program has not terminated after 2 hours of execution time.

<table>
<thead>
<tr>
<th>instance</th>
<th>expanded nodes</th>
<th>time (in sec.)</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mac</td>
<td>ModMac</td>
<td>Mac</td>
</tr>
<tr>
<td>celer6-sub0</td>
<td>1184</td>
<td>195</td>
<td>1117</td>
</tr>
<tr>
<td>celer6-sub1</td>
<td>9285</td>
<td>8085</td>
<td>–</td>
</tr>
<tr>
<td>celer6-sub2</td>
<td>5290</td>
<td>400</td>
<td>–</td>
</tr>
<tr>
<td>celer6-sub3</td>
<td>6394</td>
<td>1653</td>
<td>–</td>
</tr>
<tr>
<td>celer6-sub4</td>
<td>3434</td>
<td>1082</td>
<td>–</td>
</tr>
<tr>
<td>celer7-sub0</td>
<td>11</td>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>celer7-sub1</td>
<td>1700</td>
<td>718</td>
<td>3210</td>
</tr>
<tr>
<td>celer7-sub2</td>
<td>4008</td>
<td>2691</td>
<td>–</td>
</tr>
<tr>
<td>celer7-sub3</td>
<td>4070</td>
<td>2597</td>
<td>–</td>
</tr>
<tr>
<td>celer7-sub4</td>
<td>2580</td>
<td>1622</td>
<td>–</td>
</tr>
</tbody>
</table>

Fig. 2. Experimental results obtained by Mac* and ModMac* on instances extracted from the celer6 and celer7 real-world problems. We reported the number of nodes expanded (in thousands) within 2 hours, the CPU time and the cost of the best solution found. A dash indicates that the program has not terminated after 2 hours of execution time.
<table>
<thead>
<tr>
<th>instance</th>
<th>expanded nodes</th>
<th>time (in sec.)</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Mac^*)</td>
<td>(ModMac^*)</td>
<td>(Mac^*)</td>
</tr>
<tr>
<td>driverlog04ac</td>
<td>7</td>
<td>1</td>
<td>117</td>
</tr>
<tr>
<td>driverlog04bc</td>
<td>116</td>
<td>444</td>
<td>1648</td>
</tr>
<tr>
<td>driverlog04cc</td>
<td>208</td>
<td>128</td>
<td>2868</td>
</tr>
<tr>
<td>driverlog05ac</td>
<td>8</td>
<td>2</td>
<td>239</td>
</tr>
<tr>
<td>driverlog05bc</td>
<td>97</td>
<td>40</td>
<td>2165</td>
</tr>
<tr>
<td>driverlog05cc</td>
<td>170</td>
<td>29</td>
<td>3645</td>
</tr>
<tr>
<td>driverlog06</td>
<td>669</td>
<td>534</td>
<td></td>
</tr>
<tr>
<td>driverlog08ac</td>
<td>52</td>
<td>37</td>
<td>2232</td>
</tr>
<tr>
<td>driverlog08bc</td>
<td>71</td>
<td>44</td>
<td>2908</td>
</tr>
<tr>
<td>driverlog08cc</td>
<td>84</td>
<td>52</td>
<td>3262</td>
</tr>
</tbody>
</table>

Fig. 3. Experimental results obtained by \(Mac^*\) and \(ModMac^*\) on instances extracted from the driverlog planning problem. We reported the number of nodes expanded (in thousands) within 2 hours, the CPU time and the cost of the best solution found. A dash indicates that the program has not terminated after 2 hours of execution time.

6 Conclusion

In this paper, we have presented a domain decomposition algorithm for solving binary valued constraint satisfactions problems (VCSP). The proposed decomposition process instantiates variables by reducing their respective domains to value subsets instead of singletons. It yields, at each leaf of the search tree, a subproblem whose valuation functions are exclusively modular. Such subproblems are solved in polynomial time.

A natural extention of this work is to derive a similar decomposition schema for \(n\)-ary VCSP since Property 2 would be easily generalized to the \(n\)-ary case. A decomposition schema based on submodular functions can also be considered.

References


Author Index

Ayeb, Béchir .................. 143
Bistarelli, Stefano .............. 5
Brafman, Ronen ................. 21
Dalla Pozza, Giorgio .......... 36
Dechter, Rina .................. 51
Di Rosa, Emanuele .............. 66
Gelain, Mirco .................. 82
Giunchiglia, Enrico ............ 66
Gutierrez, Patricia ............. 98
Hatano, Daisuke ............... 113
Helou, Maher .................. 143
Hirayama, Katsutoshi .......... 113
Kaci, Souhila .................. 128
Meseguer, Pedro ............... 98
Naanaa, Wady .................. 143
O’Sullivan, Barry ............... 66
Otten, Lars ..................... 51
Piette, Cédric .................. 128
Pini, Maria Silvia .............. 82
Pirolandi, Daniele ............. 5
Rossi, Francesca .............. 21, 36, 82
Salvagnin, Domenico .......... 21
Santini, Francesco ............. 5
Sugimoto, Yuta ................ 113
Venable, K. Brent .............. 21
Venable, Kristen Brent ......... 36, 82
Walsh, Toby ................... 21, 82