Available online at www.sciencedirect.com

SCIENCE DIRECT®

Artificial Intelligence 170 (2006) 653–666

Artificial Intelligence

www.elsevier.com/locate/artint

On the incompatibility of faithfulness and monotone DAG faithfulness

David Maxwell Chickering [*], Christopher Meek

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA

Received 19 November 2002; received in revised form 21 December 2005; accepted 17 March 2006

Abstract

Cheng, Greiner, Kelly, Bell and Liu [Artificial Intelligence 137 (2002) 43–90] describe an algorithm for learning Bayesian networks that—in a domain consisting of n variables—identifies the optimal solution using $O(n^4)$ calls to a mutual-information oracle. This result relies on (1) the standard assumption that the generative distribution is Markov and faithful to some directed acyclic graph (DAG), and (2) a new assumption about the generative distribution that the authors call *monotone DAG faithfulness* (*MDF*). The MDF assumption rests on an intuitive connection between active paths in a Bayesian-network structure and the mutual information among variables. The assumption states that the (conditional) mutual information between a pair of variables is a monotonic function of the set of active paths between those variables; the more active paths between the variables the higher the mutual information. In this paper, we demonstrate the unfortunate result that, for any realistic learning scenario, the monotone DAG faithfulness assumption is *incompatible* with the faithfulness assumption. Furthermore, for the class of Bayesian-network structures for which the two assumptions are compatible, we can learn the optimal solution using standard approaches that require only $O(n^2)$ calls to an independence oracle.
© 2006 Published by Elsevier B.V.

Keywords: Bayesian networks; Grafical models; Learning; Structure search; Complexity

1. Introduction

Learning Bayesian networks from data has traditionally been considered a hard problem by most researchers. Numerous papers have demonstrated that, under a number of different scenarios, identifying the "best" Bayesian-network structure is NP-hard (see, e.g., Chickering, Meek and Heckerman [2]). In a paper describing information-theoretic approaches to this learning problem, however, Cheng, Greiner, Kelly, Bell and Liu [1] (hereafter CGKBL) describe an algorithm that runs in polynomial time when given a mutual-information oracle. In particular, for a domain of n variables, CGKBL claim that the algorithm identifies the generative Bayesian-network structure using $O(n^4)$ calls to the oracle, regardless of the complexity of that generative network. The seemingly incredible result relies on an assumption about the generative distribution that CGKBL call *monotone DAG faithfulness*. Intuitively, the assumption states that in a distribution that is perfect with respect to some Bayesian-network structure $\mathcal{G}$, the (conditional) mutual information between two variables is a monotonic function of the "active paths" between those variables in $\mathcal{G}$.

[*] Corresponding author.
E-mail address: max.chickering@microsoft.com (D.M. Chickering).

0004-3702/$ – see front matter © 2006 Published by Elsevier B.V.
doi:10.1016/j.artint.2006.03.001

In this paper, we show that the standard faithfulness assumption and the monotone DAG faithfulness assumption are inconsistent with each other unless we restrict the possible generative structures to an unreasonably simple class; furthermore, the optimal member of this simple class of models can be identified using a standard independence-based learning algorithm using only $O(n^2)$ calls to an independence oracle. Unfortunately, our results cast doubt once again on the existence of an efficient and correct Bayesian-network learning algorithm under reasonable assumptions.

The paper is organized as follows. In Section 2, we provide background material and we define the monotone DAG faithfulness assumption more rigorously. In Section 3, we describe a family of independence-based and information-based learning algorithms, we consider the worst-case complexity of these algorithms, and we show how the monotone DAG faithfulness assumption can lead to the incredible result of CGKBL. In Section 4, we provide simple examples that highlight problems with the monotone DAG faithfulness assumption, and we prove that the assumption is incompatible with faithfulness unless we impose severe restrictions on the generative structure. Finally, in Section 5, we conclude with a discussion.

2. Background

In this section, we describe our notation and present relevant background material. We assume that the reader has some basic familiarity with probability theory, graph theory, and Bayesian networks.

A *Bayesian network* is used to represent a joint distribution over variables in a domain and consists of (1) a directed acyclic graph (or *DAG* for short) in which there is a single vertex associated with each variable in the domain, and (2) a corresponding set of parameters that defines the joint distribution. We use the calligraphic letter $\mathcal{G}$ to denote a Bayesian-network structure. We use *variable* to denote both a random variable in the domain and the corresponding vertex (or node) in the Bayesian-network structure. Thus, for example, we might say that variable X is adjacent to variable Y in Bayesian-network structure $\mathcal{G}$. The parameters of a Bayesian network specify the conditional distribution of each variable given its parents in the graph, and the joint distribution for the variables in the domain is defined by the product of these conditional distributions. For more information see, for example, Pearl [5].

We use bold-faced Roman letters for sets of variables (e.g., $\mathbf{X}$), non-bold-faced Roman letters for singleton variables (e.g., X) and lower-case Roman letters for values of the variables (e.g., $\mathbf{X} = \mathbf{x}$, $X = x$). To simplify notation when expressing probabilities, we omit the name of the variables involved. For example, we use $p(y|\mathbf{x})$ instead of $p(Y = y|\mathbf{X} = \mathbf{x})$. For a distribution p, we use $\mathit{Ind}_p(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$ to denote the fact that in p, $\mathbf{X}$ is independent of $\mathbf{Y}$ given set $\mathbf{Z}$; we call $\mathbf{Z}$ the *conditioning set* of the independence relation. When the conditioning set is empty, we use $\mathit{Ind}_p(\mathbf{X}; \mathbf{Y})$ instead. To simplify notation, we omit the standard set notation when considering a singleton variable in any position. For example, we use $\mathit{Ind}_p(X; Y|\mathbf{Z})$ instead of $\mathit{Ind}_p(\{X\}; \{Y\}|\mathbf{Z})$.

2.1. Independence constraints of DAGs

Any joint distribution represented by a Bayesian network must satisfy certain independence constraints that are imposed by the structure of the model. Because a Bayesian network represents a joint distribution as the product of conditional distributions, the joint distribution must satisfy the *Markov conditions* of the structure: each variable must be independent of its non-descendants given its parents. The Markov conditions constitute a basis for the independence facts that are true for all distributions that can be represented by a Bayesian network with a given structure. The *d-separation* criterion is a graphical criterion that characterizes all of these *structural* independence constraints. In order to define the d-separation criterion, we first need to define an *active path*. We provide two distinct definitions for an active path, both of which are adequate for defining the d-separation criterion. Both definitions are standard, and we include both to highlight the sensitivity of the MDF assumption to the choice of definition.

Before proceeding, we provide standard definitions for a path, a simple path, and a collider. A *path* π in a graph $\mathcal{G}$ is an ordered sequence of variables $(X_{(1)}, X_{(2)}, \ldots, X_{(n)})$ such that for each $\{X_{(i)}, X_{(i+1)}\}$, either the edge $X_{(i)} \rightarrow X_{(i+1)}$ or the edge $X_{(i)} \leftarrow X_{(i+1)}$ exists in $\mathcal{G}$, where $X_{(i)}$ denotes the variable at position i on the path. A path is a *simple path* if each variable occurs at most once in the path. Three (ordered) variables (X, Y, Z) form a *collider complex* in $\mathcal{G}$ if the edges $X \rightarrow Y$ and $Y \leftarrow Z$ are both contained in $\mathcal{G}$. A variable $X_{(i)}$ is a *collider* at position i in a path $\pi = (X_{(1)}, X_{(2)}, \ldots, X_{(n)})$ in graph $\mathcal{G}$ if $1 < i < n$ and $(X_{(i-1)}, X_{(i)}, X_{(i+1)})$ is a collider complex in $\mathcal{G}$. Note that a collider is defined by not only a variable, but the *position* of that variable in a path; a particular variable may appear both as a collider and as a non-collider within a path.

To illustrate some of these definitions, consider the path (A, C, B, C, D, C) in Fig. 1. The path is not simple because variable C occurs more than once. The variable C is a collider at position two and a non-collider at position four and six.

We now provide our two formal definitions of an active path.

Definition 1 *(Compound active path).* A path $\pi = (X_{(1)}, X_{(2)}, \ldots, X_{(n)})$ is a compound active path given conditioning set $\mathbf{Y}$ in DAG $\mathcal{G}$ if each variable $X_{(i)}$ in the path has one of the two following properties: (1) $X_{(i)}$ is not a collider at position i and $X_{(i)}$ is not in $\mathbf{Y}$, or (2) $X_{(i)}$ is a collider at position i and either $X_{(i)}$ or a descendant of $X_{(i)}$ in $\mathcal{G}$ is in $\mathbf{Y}$.

Definition 2 *(Simple active path).* A path π is a *simple active path* given conditioning set $\mathbf{Y}$ in DAG $\mathcal{G}$ if $\boldsymbol{\pi}$ is a compound active path given $\mathbf{Y}$ in $\mathcal{G}$ that is simple.

Note that we use the phrase *conditioning set* to refer to a set of variables both in active paths and in independence relations.

From the definitions above, the endpoints of a path cannot be colliders. This means that under either definition of an active path, the endpoints cannot be in the conditioning set. To emphasize the distinction between the two definitions above, consider the graph in Fig. 1. Given conditioning set D, there is exactly one simple active path between A and B, namely, $A \to C \leftarrow B$. Given this same conditioning set, there are additional compound active paths including $A \to C \to D \leftarrow C \leftarrow B$. In fact, there are an infinite number of these additional paths as we can, for example, prepend $A \to C \leftarrow A$ to any compound active path and the result is a compound active path.

The following proposition, which is proved in Appendix A, establishes the fact that simple and compound active paths are interchangeable with respect to the definition of d-separation.

Proposition 1. *There is a simple active path between X and Y given conditioning set $\mathbf{Z}$ in $\mathcal{G}$ if and only if there is a compound active path between X and Y given conditioning set $\mathbf{Z}$ in $\mathcal{G}$.*

Finally, we can define the d-separation criterion. Sets of variables $\mathbf{X}$ and $\mathbf{Y}$ are *d-separated* given a set of variables $\mathbf{Z}$ in $\mathcal{G}$ if there does not exist a simple active path between a variable in $\mathbf{X}$ and a variable in $\mathbf{Y}$ given conditioning set $\mathbf{Z}$. For example, in Fig. 1, A is d-separated from B (given nothing) and A is d-separated from D given C. In the figure, A is *not* d-separated from B given D because there exists the simple active path $A \to C \leftarrow B$. From Proposition 1, we see that d-separation is equivalently defined by the absence of a compound active path. We use $Dsep_{\mathcal{G}}(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$ to denote that $\mathbf{X}$ is d-separated from $\mathbf{Y}$ given $\mathbf{Z}$ in $\mathcal{G}$.

The d-separation criterion provides a useful connection between a DAG and the corresponding set of distributions that can be represented with a Bayesian network with that structure. In particular, Pearl [5] shows that if $Dsep_{\mathcal{G}}(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$, then for any distribution p that can be represented by a Bayesian network with structure $\mathcal{G}$, it must be the case that $Ind_p(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$.[1] Given this strong connection between d-separation and representability in a Bayesian network, it is natural to define the following property for a distribution.

Definition 3 *(Markov distribution).* A distribution p is *Markov with respect to* $\mathcal{G}$ if $Dsep_{\mathcal{G}}(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$ implies $Ind_p(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$.

We use $\mathbf{Markov}(\mathcal{G})$ to denote the set of distributions that are Markov with respect to $\mathcal{G}$.

If two DAGs $\mathcal{G}$ and $\mathcal{G}'$ represent the same independence constraints, we say that they are *equivalent*. Verma and Pearl [7] show that two DAGs are equivalent if and only if (1) they have the same adjacencies and (2) for any collider complex (X, Y, Z) in one of the DAGs such that X and Z are not adjacent, this "v-structure" also exists in the other DAG.

[1] This is the *soundness* result for d-separation. Pearl [5] also shows that d-separation is *complete*; that is, if $Ind_p(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$ for every p that can be represented by a Bayesian network with structure $\mathcal{G}$, then $Dsep_{\mathcal{G}}(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$.

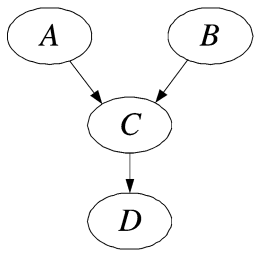


Fig. 1. A simple Bayesian-network structure.

2.2. Faithfulness

The Markov property provides a connection between the structure of a Bayesian network and independence. Namely, the absence of an edge guarantees a set of independence facts. The existence of an edge between variable X and Y in the structure $\mathcal{G}$, however, does not guarantee that a Bayesian network with structure $\mathcal{G}$ will exhibit a dependence between X and Y. Without making assumptions connecting the existence of edges in a generative structure and the joint distribution of a generative Bayesian network, it is not generally possible to recover the generative Bayesian-network structure from observed data.

Most structure-learning algorithms that have large-sample correctness guarantees assume that the distribution from which the data is generated is both Markov and *faithful* with respect to some DAG. Functionally, the faithfulness assumption implies that every edge in this DAG can be identified by a *lack* of independence in the generative distribution, for every conditioning set, between the corresponding endpoint variables. For example, if p is faithful with respect to the DAG in Fig. 1, then A cannot be independent of C in p.

Definition 4 *(Faithful distribution).* A distribution p is *faithful* to $\mathcal{G}$ if $Ind_p(\mathbf{X};\mathbf{Y}|\mathbf{Z})$ implies $Dsep_{\mathcal{G}}(\mathbf{X};\mathbf{Y}|\mathbf{Z})$.

We use **Faithful**$(\mathcal{G})$ to denote the set of distributions that are faithful to $\mathcal{G}$. As we see in the next section, the intersection **Markov**$(\mathcal{G}) \cap$ **Faithful**$(\mathcal{G})$ is an important class of distributions for proving optimality results about learning algorithms; we use **Perfect**$(\mathcal{G})$ to denote this intersection. For a distribution p, if there exists a DAG $\mathcal{G}$ such that $p \in$ **Perfect**$(\mathcal{G})$, we say that p is a *DAG-perfect* distribution, and that *p is perfect with respect to $\mathcal{G}$*.

The assumption of faithfulness might seem like an unjustifiably strong assumption, but a joint distribution represented by a Bayesian network can fail to be faithful only by a precise balancing of the parameters. This intuition is made more precise in Meek [4] and Spirtes, Glymour and Scheines [6], where it is shown that of the distributions that are Markov with respect to a structure $\mathcal{G}$, all but a measure-zero set of those distributions are also faithful to that structure. In other words, if you put a smooth measure over the distributions representable by a Bayesian network with structure $\mathcal{G}$ and choose a distribution at random, you will choose a faithful distribution with probability one.

2.3. Information and monotone DAG faithfulness

The CGKBL algorithm uses the conditional-mutual information between sets of variables to recover the structure of a Bayesian network. The correctness claims of CGKBL are based on an assumption that they call *monotone DAG faithfulness*. Similar to the assumption of faithfulness, this assumption connects properties of the generative Bayesian-network structure and the information relationships among sets of variables in the generative distribution.

The conditional mutual information between $\mathbf{X}$ and $\mathbf{Y}$ given $\mathbf{Z}$ in a probability distribution p is formally defined as:

$$Inf_p(\mathbf{X};\mathbf{Y}|\mathbf{Z}) = \sum_{\mathbf{x},\mathbf{y},\mathbf{z}} p(\mathbf{x},\mathbf{y},\mathbf{z}) \log \frac{p(\mathbf{x},\mathbf{y}|\mathbf{z})}{p(\mathbf{x}|\mathbf{z})p(\mathbf{y}|\mathbf{z})} \tag{1}$$

where 'log' denotes the base-two logarithm.

In the previous section we defined two types of active paths: simple active paths and compound active paths. Active paths as defined by CGKBL are compound active paths. We include the alternative simple definition because it is a standard definition of active path and because it highlights the sensitivity of the monotone DAG faithfulness assumption to the underlying definition of active path. When the distinction is not necessary, we use the term *active path* to refer to a path that is either a simple active path or a compound active path.

We now provide a formal definition of monotone DAG faithfulness (MDF). Let $Active^s_{\mathcal{G}}(X;Y|\mathbf{Z})$ denote the set of simple active paths between X and Y given conditioning set $\mathbf{Z}$ in $\mathcal{G}$. Similarly, let $Active^c_{\mathcal{G}}(X;Y|\mathbf{Z})$ denote the set of compound active paths between X and Y given conditioning set $\mathbf{Z}$ in $\mathcal{G}$. We use $Active_{\mathcal{G}}(X;Y|\mathbf{Z})$ to denote the set of active paths under one of the two definitions of active path when we want to avoid specifying which definition of active path to use.

Definition 5 *(Simple monotone DAG faithfulness).* A distribution p is *simple monotone DAG faithful* with respect to a DAG $\mathcal{G}$ if

$$Active^s_{\mathcal{G}}(X;Y|\mathbf{Z}) \subseteq Active^s_{\mathcal{G}}(X;Y|\mathbf{Z}') \Rightarrow Inf_p(X;Y|\mathbf{Z}) \leqslant Inf_p(X;Y|\mathbf{Z}')$$

Definition 6 *(Compound monotone DAG faithfulness).* A distribution p is *compound monotone DAG faithful* with respect to a DAG $\mathcal{G}$ if

$$Active^c_{\mathcal{G}}(X;Y|\mathbf{Z}) \subseteq Active^c_{\mathcal{G}}(X;Y|\mathbf{Z}') \Rightarrow Inf_p(X;Y|\mathbf{Z}) \leqslant Inf_p(X;Y|\mathbf{Z}')$$

The property is called "monotone" because it states that information in p is a monotonic function of simple (compound) active paths in $\mathcal{G}$. More specifically, simple (compound) monotone DAG faithfulness states that if we do not remove (or "block") any simple (compound) active paths between two variables in $\mathcal{G}$ by changing the conditioning set, then the information does not decrease. We will see that, depending on the definition of an active path, the property can have different consequences. We use $\mathbf{MDF}^s(\mathcal{G})$ and $\mathbf{MDF}^c(\mathcal{G})$ to denote the set of distributions that are monotone DAG faithful with respect to $\mathcal{G}$ using simple and compound active paths, respectively. When we want to avoid specifying the definition of active path, we use $\mathbf{MDF}(\mathcal{G})$ instead.

CGKBL define "monotone DAG faithfulness" only for DAG-perfect distributions, which makes it unclear whether non-DAG-perfect distributions can satisfy this property. In contrast, we define $\mathbf{MDF}^s(\mathcal{G})$ and $\mathbf{MDF}^c(\mathcal{G})$ without reference to other properties of distributions (e.g., perfectness) in order to analyze the relationship between faithfulness and monotone DAG faithfulness (simple or compound). As previously described, CGKBL use the compound definition of active paths, and thus their definition of monotone DAG faithfulness is precisely our definition of compound monotone DAG faithfulness restricted to distributions that are faithful.

3. Independence-based and information-based learning algorithms

In this section, we discuss independence-based and information-based algorithms for learning Bayesian-network structures and discuss the corresponding worst-case running times. Instead of providing formal complexity analyses, which would require us to provide a detailed description of specific instances of these algorithms, we present simple arguments to provide the reader with an intuitive understanding of how each type of algorithm handles the most difficult learning scenarios.

In practice, these learning algorithms take an observed set of data and perform statistical tests to evaluate independence and/or mutual information. Thus we can expect the running times of these algorithms to grow with the number of samples in the data. For simplicity, our analyses avoid statistical-sampling issues by effectively assuming that the algorithms have infinite data; each algorithm will have access to an "oracle" that can evaluate independence and/or information as if it had access to the generative distribution. The complexity for an algorithm is then evaluated by the number of times the oracle is called. In practice, an independence oracle and an information oracle can be approximated with increasing accuracy as the number training cases increases and the number of variables in the query decreases.

3.1. Independence-based learning algorithms

Structure-learning algorithms typically assume that training data is a set of independent and identically distributed samples from some generative distribution p^* that is perfect with respect to some DAG $\mathcal{G}^*$. The goal of the learning algorithm is then to identify $\mathcal{G}^*$ or any DAG that is equivalent to $\mathcal{G}^*$.

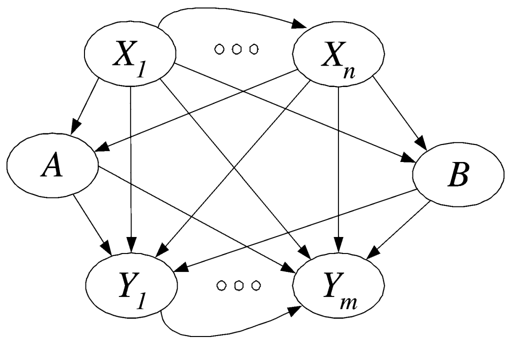


Fig. 2. Worst-case scenario for independence-based learning algorithms.

A large class of structure-learning algorithms, which we call *independence-based algorithms*, use independence tests to identify and direct edges. If p^* is DAG-perfect and an *independence oracle*—that is, an oracle that provides yes/no answers to queries about conditional independencies in p^*—is available, these algorithms can identify a DAG that is equivalent to $\mathcal{G}^*$ (see, for example, Spirtes, Glymour and Scheines [6] or Verma and Pearl [7]).

Although many different algorithms have been developed, the basic idea behind independence-based algorithms is as follows. In a first phase, the algorithms identify pairs of variables that must be adjacent in the generative structure. Under the assumption that p^* is DAG-perfect, variables that are adjacent in the generative structure have the property that they are not independent given any conditioning set. The independence oracle is used to check whether this property holds for each pair of variables. Various algorithms provide improvements over an exhaustive search over all subsets of variables. In the second phase, the identified edges are directed.

3.2. Why independence-based learning is hard

A worst-case scenario for the independence-based algorithms is when the generative structure is as shown in Fig. 2 in which all variables are adjacent *except* for A and B. More specifically, (1) the variables in $\mathbf{X} = \{X_1, \ldots, X_n\}$ are parents of A, B and all variables in $\mathbf{Y} = \{Y_1, \ldots, Y_m\}$, (2) both A and B are parents of all the variables in $\mathbf{Y}$, (3) X_i is a parent of X_j for all $i < j$, and (4) Y_i is a parent of Y_j for all $i < j$. For this structure, the independence oracle will return "not independent" for *any* test other than "is A independent of B given $\mathbf{X}$?". This extreme example demonstrates that—when using an independence oracle—the only way to determine whether A and B are adjacent is to enumerate and test all possible conditioning sets; using an adversarial argument, we could have the oracle return "not independent" on all but the *last* conditioning set. Because there are $2^{|\mathbf{X}|+|\mathbf{Y}|}$ possible conditioning sets, identifying whether or not the generative network contains an edge between A and B is intractable.

3.3. Information-based learning algorithms

CGKBL take a slightly different approach to learning Bayesian networks. Instead of using conditional independence directly, they use conditional-mutual information both to test for independence and to help guide the learning algorithm. Information can be used to measure the degree of conditional dependence among sets of variables; the following well-known fact about information (e.g., Cover and Thomas [3]) helps provide insight into this relationship.

Fact 1. $\mathit{Inf}_p(\mathbf{X}; \mathbf{Y}|\mathbf{Z}) = 0$ *if and only if* $\mathit{Ind}_p(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$.

Fact 1 demonstrates that any algorithm that utilizes an independence oracle can be modified to use an information oracle. The potential for improvement lies in the fact that we receive additional information when using an information oracle. With this additional information and the MDF assumption, CGKBL claim that their algorithm identifies the generative structure using a polynomial number of queries to the information oracle in the worst case. It turns out that the worst-case scenario considered in the previous section is also the key scenario for information-based algorithms. In particular, it is reasonably easy to show that if we can identify the set $\mathbf{X}$ (i.e., the parents of A and B) in Fig. 2 efficiently, then we can identify the entire generative structure efficiently.

Consider again the graph in Fig. 2. We can use MDF to identify the set $\mathbf{X}$ using the following *greedy* algorithm: start with the conditioning set $\mathbf{S} = \mathbf{X} \cup \mathbf{Y}$, and then repeatedly remove from $\mathbf{S}$ the variable that results in the largest *decrease* in information between A and B, until no removal decreases the information. If the resulting information is zero, we know that there is no edge between A and B; otherwise, we conclude that there is an edge.

A non-rigorous argument for why the greedy algorithm is correct for the example is as follows. First, the algorithm never removes any element of $\mathbf{X}$ from $\mathbf{S}$ because the removal of any such element—when the remaining elements of $\mathbf{X}$ are in $\mathbf{S}$ —cannot "block" any active paths (under either definition of an active path) between A and B. Thus, because the number of active paths has necessarily increased, we conclude by MDF that the information in the generative distribution p^* cannot decrease from such a removal. Second, it is possible to show that the deepest variable $Y_i \in \mathbf{Y} \cap \mathbf{S}$ (the variable with the largest index) has the property that if it is removed from $\mathbf{S}$, no active paths are created; thus, because removing Y_i from $\mathbf{S}$ will "block" the previously active path $A \to Y_i \leftarrow B$, we conclude from MDF that the information cannot increase in p^* from the removal. For simplicity, we ignore the boundary cases where removing a member from either $\mathbf{X}$ or $\mathbf{Y}$ does not change the information; under this scenario (1) the information increases as a result of removing any variable from $\mathbf{X}$, and (2) there is always a variable Y_i from $\mathbf{Y} \cap \mathbf{S}$ such that the information decreases by removing Y_i from $\mathbf{S}$. We conclude that the greedy algorithm will terminate with the correct conditioning set $\mathbf{S} = \mathbf{X}$. Furthermore, each iteration of the algorithm requires at most $|\mathbf{S}| = |\mathbf{X}| + |\mathbf{Y}|$ calls to the information oracle, and there will be $|\mathbf{Y}|$ such iterations. Thus, the greedy algorithm will terminate after $O(|\mathbf{Y}|^2 + |\mathbf{X}| \cdot |\mathbf{Y}|)$ calls to the information oracle.

CGKBL define a specific information-based learning algorithm that overcomes the worst-case exponential behavior described in the previous section by using a greedy search as above to determine whether or not an edge should be present. Furthermore, they provide a similar argument as above to claim that given $p^* \in \mathbf{Perfect}(\mathcal{G}^*) \cap \mathbf{MDF}(\mathcal{G}^*)$, the algorithm will recover the generative structure (up to equivalence).

4. The monotone DAG faithfulness assumption

Without studying the details of MDF, the assumption may seem intuitively appealing at first: suppose that removing a variable from the conditioning set "deactivates" some paths between A and B in the generative structure without simultaneously "activating" any other paths. Then we might be tempted to believe that the mutual information between A and B should decrease, or at least not increase. CGKBL state:

> In real world situations most faithful models are also monotone DAG-faithful. We conjecture that the violations of monotone DAG-faithfulness only happen when the probability distributions are 'near' the violations of DAG-faithfulness.

If the CGKBL conjecture were true, it would have significant consequences for learning. First, most structure-learning algorithms assume faithfulness to prove correctness and thus, by assuming a little bit more, we could obtain an algorithm that requires only a polynomial number of calls to an information oracle. Second, for a given structure $\mathcal{G}$, almost all distributions in $\mathbf{Markov}(\mathcal{G})$ are faithful, and thus we could be confident that our assumptions are not too limiting.

Our main result is that MDF is incompatible with faithfulness unless we are in an unrealistic learning scenario for which the optimal structure can be identified using standard approaches with $O(n^2)$ calls to an independence oracle. Before proving our main result, we find it useful to explore some examples that demonstrate some specific problems with MDF. In Section 4.1, we provide a simple example of a distribution that violates MDF and is not simultaneously "close" to being non-faithful. In Section 4.2, we show a simple example where MDF leads to a counterintuitive consequence. In Section 4.3, we prove our main result: unless the generative structure comes from a severely restricted class of models, MDF and faithfulness are incompatible.

4.1. A simple violation of MDF

In this section, we provide a simple example of a faithful distribution that does not satisfy the MDF assumption. As described above, we will show in Section 4.3 that for most graphs it is *impossible* to simultaneously satisfy both

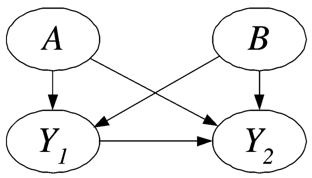


Fig. 3. Structure of a Bayesian network that violates the MDF assumption.

Table 1
Parameters of a Bayesian network that violates the MDF assumption

A	B	Y_1	$p(Y_1 \mid A, B)$
0	0	0	0.38
0	0	1	0.62
0	1	0	0.01
0	1	1	0.99
1	0	0	0.20
1	0	1	0.80
1	1	0	0.99
1	1	1	0.01

A	B	Y_1	Y_2	$p(Y_2 \mid A, B, Y_1)$
0	0	0	0	0.96
0	0	0	1	0.04
0	0	1	0	0.22
0	0	1	1	0.78
0	1	0	0	0.35
0	1	0	1	0.65
0	1	1	0	0.91
0	1	1	1	0.09
1	0	0	0	0.89
1	0	0	1	0.11
1	0	1	0	0.99
1	0	1	1	0.01
1	1	0	0	0.05
1	1	0	1	0.95
1	1	1	0	0.50
1	1	1	1	0.50

A	$p(A)$
0	0.5
1	0.5

B	$p(B)$
0	0.5
1	0.5

faithfulness and MDF; the structure in our simple example happens to be a member of the restricted class of graphs for which it is possible to satisfy both conditions.

Consider the Bayesian-network structure shown in Fig. 3 and the corresponding set of parameters shown in Table 1. Note that the structure of this example is a particular instance of the worst-case-scenario model from Section 3.2.

Under either definition of MDF, this Bayesian network provides an example of a violation of MDF. In particular, under either definition of an active path, the set of active paths between A and B given both Y_1 and Y_2 is a superset of the set of active paths when only Y_1 is in the conditioning set. Thus, for any distribution p contained in either $\mathbf{MDF}^s(\mathcal{G})$ or $\mathbf{MDF}^c(\mathcal{G})$ we have

$$Inf_p(A; B \mid Y_1, Y_2) \geqslant Inf_p(A; B \mid Y_1)$$

For the joint distribution q obtained from the conditional distributions in the table, however, we have $Inf_q(A; B \mid Y_1, Y_2) = 0.33$ and $Inf_q(A; B \mid Y_1) = 0.35$. If we consider the equivalent structure in which the edge between Y_1 and Y_2 is reversed, we obtain the inequality $Inf_p(A; B \mid Y_1, Y_2) \geqslant Inf_p(A; B \mid Y_2)$. Using the same distribution q (which is Markov with respect to the modified structure) we have $Inf_q(A; B \mid Y_1, Y_2) = 0.33$ and $Inf_q(A; B \mid Y_2) = 0.40$. Thus in both cases, the distribution q is not contained in either $\mathbf{MDF}^s(\mathcal{G})$ or $\mathbf{MDF}^c(\mathcal{G})$.

To demonstrate that our distribution is faithful, we enumerated all 23 dependence facts between singleton variables[2] and measured the corresponding information. We then compared these information values to the thresholds that CGKBL use for detecting dependence: CGKBL deem two variables conditionally independent only if the corresponding mutual information is less than either 0.01 or 0.0025 (depending on the experiment).[3] Out of the 23

[2] There are four-choose-two pairs of singletons to consider; for each pair, we consider (1) no conditioning set, (2) each of the two singleton-element conditioning sets, and (3) the single two-element conditioning set. Because the Markov conditions guarantee exactly one independence fact ($Ind_p(A; B)$), we are left with 23 dependence facts to check.

[3] CGKBL do not define explicitly the base of the logarithm that they use, but they present two values of particular information calculations from experiments using the ALARM network; by calculating these values from the known generative structure, it is clear that they are using base two, which is standard when calculating information.

information values, the *smallest* value was $Ind_p(A; Y_2) = 0.028$, which is nearly three times bigger than the largest threshold used by CGKBL.

Our example is particularly interesting because it illustrates that violations can occur during crucial phases of the CGKBL learning algorithm. Namely, in order for the algorithm to learn that there is no edge between A to B, it must successfully identify the marginal independence. To get to the point where this independence test is made, the algorithm must first find that either $Inf_p(A; B|Y_1, Y_2) > Inf_p(A; B|Y_1)$ or $Inf_p(A; B|Y_1, Y_2) > Inf_p(A; B|Y_2)$, neither of which is true in this example. This failure would lead the algorithm to learn incorrectly that there is an edge between A and B.

4.2. Counterintuitive consequence of MDF

In this section, we explore a counterintuitive consequence of the MDF assumption by considering the DAG shown in Fig. 1. We provide an example in which the consequence is satisfied and we show that it is satisfied in a small but non-negligible fraction of randomly sampled distributions.

For the DAG shown in Fig. 1, the two definitions of MDF (simple and compound) correspond to two different sets of distributions for this example. In particular, for the simple definition, we have $Active^s_{\mathcal{G}}(A; B|C) = Active^s_{\mathcal{G}}(A; B|D)$ and thus $Inf_p(A; B|C) = Inf_p(A; B|D)$ for any distribution p in $\mathbf{MDF}^s(\mathcal{G})$. For the compound definition, we have $Active^c_{\mathcal{G}}(A; B|C) \subseteq Active^c_{\mathcal{G}}(A; B|D)$ and thus $Inf_p(A; B|C) \leqslant Inf_p(A; B|D)$ for any distribution p in $\mathbf{MDF}^c(\mathcal{G})$. The equality of the information for distributions in $\mathbf{MDF}^s(\mathcal{G})$ is *a priori* unreasonable whenever there is a stochastic (i.e., non-deterministic) relationship between C and D. The inequality for distributions in $\mathbf{MDF}^c(\mathcal{G})$, on the other hand, seems counterintuitive. That is, it seems plausible that there should be *more* dependence between A and B when given C than when given D, and thus we might expect an information inequality in the opposite direction than what holds in $\mathbf{MDF}^c(\mathcal{G})$. Rather surprising, this inequality can be satisfied using the conditional distributions in Table 2. For this distribution, the difference $Inf_p(A; B|C) - Inf_p(A; B|D) = -0.006$.

To help understand how often the information inequality implied by the compound version of MDF occurs, we performed a simple simulation study in which we randomly sampled distributions that are Markov with respect to the structure in Fig. 1—where each variable was binary—and computed $Inf_p(A; B|C) - Inf_p(A; B|D)$ for each sampled distribution p. We defined "zero" to be (a conservative) 0 ± 10^{-8} to make sure we did not miss any equalities due to numerical imprecision. Our experiment using 100,000 sampled distributions yielded the following results for the information differences: (a) positive in 99,969 samples, (b) negative in 31 samples, and (c) "zero" in 0 samples. We were surprised by both the existence and the frequency of sampled distributions in which the difference $Inf_p(A; B|C) - Inf_p(A; B|D)$ was negative.

4.3. Incompatibility of MDF and faithfulness

In this section, we prove the main result of this paper: MDF is incompatible with faithfulness unless we are in an unrealistic learning scenario for which the optimal structure can be identified using standard approaches with $O(n^2)$ calls to an independence oracle. Before proceeding, we present the following "axiom" that follows from MDF for DAG-perfect distributions, the proof of which is given in Appendix A.

Table 2
Parameters of a Bayesian network for the structure given in Fig. 1. The resulting joint distribution satisfies the counterintuitive consequence of MDF

A	$p(A)$
0	0.5
1	0.5

B	$p(B)$
0	0.6
1	0.4

A	B	C	$p(C\|AB)$
0	0	0	0.99
0	0	1	0.01
0	1	0	0.5
0	1	1	0.5
1	0	0	0.99
1	0	1	0.01
1	1	0	0.99
1	1	1	0.01

C	D	$p(D\|C)$
0	0	0.9
0	1	0.1
1	0	0.01
1	1	0.99

Theorem 1. *Let $\mathcal{G}$ be any DAG and let p be any distribution in* $\mathbf{MDF}(\mathcal{G}) \cap \mathbf{Perfect}(\mathcal{G})$*, where* $\mathbf{MDF}(\mathcal{G})$ *is defined using either of the two definitions of an active path. Then for any set* $\mathbf{V}$,

$$Dsep_{\mathcal{G}}(X; Y|\mathbf{V}) \Rightarrow Ind_p(X; Y)$$

In other words, if two variables are d-separated given *any* conditioning set in $\mathcal{G}$, then for all distributions in $\mathbf{MDF}(\mathcal{G}) \cap \mathbf{Perfect}(\mathcal{G})$, those variables are *marginally* independent. To understand the implication of this result, we define what it means for a DAG *to have a chain*:

Definition 7 *(DAG $\mathcal{G}$ has a chain).* A DAG $\mathcal{G}$ *has a chain* if, for any pair of variables X and Y that are not adjacent in $\mathcal{G}$, one of the following three sub-graphs occurs in $\mathcal{G}$:

- $X \to Z \to Y$,
- $X \leftarrow Z \to Y$,
- $X \leftarrow Z \leftarrow Y$.

In other words, a graph has a chain if there is a length-two path between non-adjacent variables that is not a "v-structure" $X \to Z \leftarrow Y$. The following result, proved by Verma and Pearl [7], will be useful for proving our main result.

Lemma 1. (Verma and Pearl [7]) *Let X and Y be non-adjacent variables in DAG $\mathcal{G}$, and let $\mathbf{Z}$ denote the union of their parents. Then $Dsep_{\mathcal{G}}(X; Y|\mathbf{Z})$.*

For the convenience of the interested reader, we provide a proof of Lemma 1 in Appendix A. We now prove the main result of this paper:

Theorem 2. *The following statements are jointly inconsistent*:

- *$\mathcal{G}$ has a chain,*
- $p \in \mathbf{Perfect}(\mathcal{G})$,
- $p \in \mathbf{MDF}(\mathcal{G})$,

where $\mathbf{MDF}(\mathcal{G})$ *is defined using either of the two definitions of an active path.*

Proof. Suppose $\mathcal{G}$ has a chain, and let p be any distribution in $\mathbf{MDF}(\mathcal{G}) \cap \mathbf{Perfect}(\mathcal{G})$. By definition of a chain, there exists a non-adjacent pair of variables X and Y in $\mathcal{G}$ that are connected by a length-two path through Z, where Z is a parent of either X or Y (or both). From Lemma 1, we know $Dsep_{\mathcal{G}}(X; Y|\mathbf{Z})$ where $\mathbf{Z}$ is the union of the parents of X and Y in $\mathcal{G}$. From Theorem 1, this implies that $Ind_p(X; Y)$. But the length-two path between X and Y through Z constitutes a simple (and compound) active path in $\mathcal{G}$ given the empty conditioning set, and we conclude that p is not faithful to $\mathcal{G}$, contradicting the supposition that $p \in \mathbf{Perfect}(\mathcal{G})$. □

The optimality result of CGKBL requires the generative distribution to be both perfect and monotone DAG faithful. Thus, as a consequence of Theorem 2, the optimality result of CGKBL does not apply to any generative structure that has a chain. The simple structure in Fig. 1 is one such example.

One possible "fix" in light of this negative result would be to weaken the requirement that the generative distribution be faithful. As described in Section 2.2, however, almost all distributions in $\mathbf{Markov}(\mathcal{G})$ are also in $\mathbf{Faithful}(\mathcal{G})$, so we can conclude that for generative structures that have chains, the MDF assumption is not reasonable. We might hope that the MDF assumption is useful in learning scenarios where it is reasonable to assume that the generative distribution is perfect with respect to some DAG with no chain. As we saw in Section 4.1, the assumption can be violated for such a distribution, but given the $O(n^4)$ result of CGKBL it might be worth restricting the possible generative distributions. In this scenario, however, we can apply an independence-based learning algorithm that (1) does not need to assume MDF and (2) identifies the optimal structure in just $O(n^2)$ calls to an independence oracle. In particular, because we know ahead of time that only marginal independence facts hold in the generative distribution p, we can identify all of them by testing, for each pair of variables A and B, whether $Ind_p(A; B)$. After all the independence facts have been identified, we direct all the edges using standard approaches (see, e.g., Spirtes et al., [6]).

5. Discussion

In this paper, we demonstrated that the monotone DAG faithfulness assumption is incompatible with the faithfulness assumption unless we are in an unrealistic learning scenario where the optimal structure can be identified using standard approaches with $O(n^2)$ calls to an independence oracle. Unfortunately, this means that the optimality guarantees of the CGKBL algorithm apply only in unrealistic situations where a faster learning algorithm is also optimal. Furthermore, because an independence oracle can be implemented with an information oracle, the faster algorithm requires a less powerful oracle.

Given the unreasonable consequences of MDF, it is intriguing that the assumption is so intuitively appealing. We believe that the source of the misguided intuition stems from the fact that—assuming faithfulness—information is zero if and only if there are no active paths. In particular, this fact implies that for any faithful distribution, the "information flow" between two variables necessarily increases when the set of active paths changes from the empty set to something other than the empty set. The mistake is to extrapolate from this base case and conclude that a non-zero "information flow" does not decrease when we add (zero or more elements) to the set of active paths.

Our study of MDF has led to a surprising result about distributions Markov with respect to the structure in Fig. 1. Namely, the conditional mutual information between A and B can be *larger* when given D than when given C. Although we found that such distributions were not common given our sampling scheme, they occurred regularly enough that they cannot be discarded as anomalous.

Finally, we believe that CGKBL have brought up an interesting question: can we make some connection between active paths and information that might lead to more efficient learning algorithms? Perhaps replacing MDF with an alternative assumption would yield more realistic constraints on distributions and yet still lead to an efficient algorithm.

Appendix A. Proofs

In this appendix, we prove Proposition 1, Lemma 1, and Theorem 1. We begin by proving three propositions about active paths, the first of which is Proposition 1.

Proposition 1. *There is a simple active path between X and Y given conditioning set $\mathbf{Z}$ in $\mathcal{G}$ if and only if there is a compound active path between X and Y given conditioning set $\mathbf{Z}$ in $\mathcal{G}$.*

Proof. Because a simple active path is also a compound active path, we need only show the existence of a simple active path between X and Y given a compound active path π between X and Y. We establish this result by showing that any sub-path of π that begins and ends with the same variable W may be "skipped" by replacing the entire sub-path with the single variable W, such that the resulting path π' remains active; after repeatedly removing all such sub-paths, the resulting (active) path will necessarily be simple. It is easy to see that after removing the sub-path from W to itself from π, the two properties of an active path in Definition 1 continue to hold for all variable/positions *other* than the variable W at the position where the sub-path was removed.

To complete the proof, we consider the following three cases:

(a) If $W \in \mathbf{Z}$, then W must be a collider at every position along π, and therefore in π' both edges incident to W at the position where the sub-path was removed are directed into W; thus, because W is a collider at this position in π, it satisfies condition (2) of an active path in Definition 1.

(b) If $W \notin \mathbf{Z}$, and W is not a collider in π' at the position where the sub-path was removed, then W at this point satisfies condition (1) of Definition 1.

(c) The final case to consider is if $W \notin \mathbf{Z}$ and W is a collider in π' at the position the sub-path was removed. Consider a traversal of the sub-path from W to itself that was removed from π to produce π'. If either the first or the last edge in this path is directed *toward* W, then W is a collider at that point in π, from which we conclude that W has a descendant in $\mathbf{Z}$ and thus W satisfies condition (2) of Definition 1. Otherwise, the first and last edge of this W-to-W

path are both directed *away* from W. This means that at some point along the traversal, we must hit some collider on π that satisfies condition (2) of Definition 1. Because the *first* such collider is a descendant of W, condition (2) of Definition 1 is satisfied for W at this position in π', and thus the proposition follows. □

Proposition 2. *For either definition of an active path, if $\pi \in Active_{\mathcal{G}}(Z; Y|\mathbf{W})$ but $\pi \notin Active_{\mathcal{G}}(Z; Y|\mathbf{W}, X)$ then X occurs as a non-collider at some position in π.*

Proof. We know that π is active when the conditioning set is $\mathbf{W}$, but if we add X to the conditioning set, π is no longer active. Therefore, from Definition 1, after adding X to the conditioning set either (1) there is a non-collider on the path that is now in the conditioning set, or (2) there is a collider on the path that—after the addition—is not in the conditioning set and has no descendants in the conditioning set. Because no variables were removed from the conditioning set, we know that only (1) is possible and that X is a non-collider at some position on π. □

Proposition 3. *Let $\pi = \{A_{(1)}, \ldots, A_{(n)}\}$ be any path in $Active^c_{\mathcal{G}}(Z; Y|\mathbf{W})$. Then for any $A_{(i)}$ and $A_{(j)}$ such that $i < j$, $A_{(i)} \notin \mathbf{W}$ and $A_{(j)} \notin \mathbf{W}$, the sub-path $\pi' = \{A_{(i)}, \ldots, A_{(j)}\}$ is in $Active^c_{\mathcal{G}}(A_{(i)}; A_{(j)}|\mathbf{W})$.*

Proof. From Definition 1, all variables in π' satisfy one of the two necessary conditions, with the possible exception of the endpoints; these variables can be colliders in the original path, but are necessarily non-colliders (see the definition of a *collider at a position* in Section 2.1) in π'. Because neither endpoint is in $\mathbf{W}$, condition (1) in Definition 1 is satisfied for the endpoints and the proposition follows. □

Proposition 3 simply asserts that any sub-path of an active path between two variables that are not in the conditioning set is itself active. Because a simple path is a compound path, the proposition also holds for simple active paths.

Lemma 1. (Verma and Pearl [7]) *Let X and Y be non-adjacent variables in DAG $\mathcal{G}$, and let $\mathbf{Z}$ denote the union of their parents. Then $Dsep_{\mathcal{G}}(X; Y|\mathbf{Z})$.*

Proof. Suppose that X and Y are not adjacent in DAG $\mathcal{G}$ but that there is a simple active path π between X and Y given $\mathbf{Z}$. Because $\mathbf{Z}$ contains the parents of both X and Y, the variable immediately following X (preceding Y) must be a descendant of X (Y). It follows that there must be a collider at some position on path π. Furthermore, the collider at the position nearest to X on π is a descendant of X and, similarly, the collider at the position nearest to Y on π is a descendant or Y. For the path π to be active, however, these colliders must be in $\mathbf{Z}$ or have descendants in $\mathbf{Z}$, which would imply the existence of a directed cycle and thus a contradiction. □

Note that the next lemma is relevant to *simple* active paths.

Lemma 2. $Dsep_{\mathcal{G}}(X; Y|\mathbf{W}, Z) \Rightarrow Active^s_{\mathcal{G}}(Z; Y|\mathbf{W}) \subseteq Active^s_{\mathcal{G}}(Z; Y|\mathbf{W}, X)$.

Proof. If either X or Y is an element of $\mathbf{W}$, the lemma follows easily; for the remainder of the proof we assume that neither variable is contained in the conditioning set. Suppose $Dsep_{\mathcal{G}}(X; Y|\mathbf{W}, Z)$ and there exists a path π in $Active^s_{\mathcal{G}}(Z; Y|\mathbf{W})$ that is not in $Active^s_{\mathcal{G}}(Z; Y|\mathbf{W}, X)$. From Proposition 2, we conclude that X must be a non-collider at some position i along π. We now consider the sub-path π' of π that starts at variable X in position i, and continues to variable Y. Because neither X nor Y is in $\mathbf{W}$, we know from Proposition 3 that $\pi' \in Active_{\mathcal{G}}(X; Y|\mathbf{W})$. Furthermore, because π is a *simple* path that starts at Z, we know that π' does not contain Z and consequently must be contained in $Active_{\mathcal{G}}(X; Y|\mathbf{W}, Z)$. But this contradicts the supposition $Dsep_{\mathcal{G}}(X; Y|\mathbf{W}, Z)$. □

We find it convenient to use $\mathbf{Pa}^{\mathcal{G}}_X$ to denote the set of parents of variable X in DAG $\mathcal{G}$.

Lemma 3. *Let X and Y be any pair of variables that are not adjacent in $\mathcal{G}$, and for which X is not an ancestor of Y, and let $\mathbf{D}$ be any non-empty subset of $\mathbf{Pa}_X^{\mathcal{G}} \cup \mathbf{Par}_Y^{\mathcal{G}}$ such that $Dsep_{\mathcal{G}}(X; Y|\mathbf{D})$. Let $\mathbf{W} = \mathbf{D} \setminus Z$, for any variable $Z \in \mathbf{D}$. Then under either of the two definitions of an active path*

$$Active_{\mathcal{G}}(Z; Y|\mathbf{W}) \subseteq Active_{\mathcal{G}}(Z; Y|\mathbf{W}, X)$$

Proof. Because $Dsep_{\mathcal{G}}(X; Y|\mathbf{W}, Z)$, the lemma follows immediately from Lemma 2 for the simple definition of an active path. For the remainder of the proof, we consider only the compound definition of an active path.

We prove the lemma by contradiction. In particular, we show that if there exists an active path in $Active^c_{\mathcal{G}}(Z; Y|\mathbf{W})$ that is not in $Active^c_{\mathcal{G}}(Z; Y|\mathbf{W}, X)$, then there exists some $W \in \mathbf{W}$ that is a descendant of X in $\mathcal{G}$. Identifying such a W yields a contradiction by the following argument: if W is a parent of X, then we have identified a directed cycle in $\mathcal{G}$, and if W is a parent of Y, then X is an ancestor of Y.

The remainder of the proof demonstrates the existence of $W \in \mathbf{W}$ that is a descendant of X in $\mathcal{G}$. Let $\pi = \{A_{(1)}, \ldots, A_{(n)}\}$—where $A_{(1)} = Z$ and $A_{(n)} = Y$—be any path in $Active^c_{\mathcal{G}}(Z; Y|\mathbf{W})$ such that $\pi \notin Active^c_{\mathcal{G}}(Z; Y|\mathbf{W}, X)$. From Proposition 2, X must appear as a non-collider at some position i along π; that is, $A_i = X$ and π must contain one of the following three sub-paths:

1. $A_{(i-1)} \to X \to A_{(i+1)}$,
2. $A_{(i-1)} \leftarrow X \leftarrow A_{(i+1)}$,
3. $A_{(i-1)} \leftarrow X \to A_{(i+1)}$.

We now consider any path π' that starts at X and then follows the edges in π (toward either $A_{(1)}$ or $A_{(n)}$) such that the first edge is directed *away* from X. That is, if π contains sub-path (1) above we have

$$\pi' = X \to A_{(i+1)} - \cdots - A_{(n)}$$

where '$-$' denotes an edge in the path without specifying its direction. Similarly, if π contains sub-path (2) above we have

$$\pi' = X \to A_{(i-1)} - \cdots - A_{(1)}$$

Finally, if π contains sub-path (3) above, π' can be defined as either of the previous two paths.

To simplify our arguments, we rename the elements of π' as follows:

$$\pi' = X \to B_{(1)} - \cdots - B_{(m)}$$

Consider a traversal of π', starting at the first element X and continuing through each element $B_{(i)}$ for increasing i. If the traversal ever encounters variable Y, it must *first* encounter variable Z; if not, the sub-path from X to Y would constitute an active path that remains active when Z is in the conditioning set, which contradicts the fact that $Dsep_{\mathcal{G}}(X; Y|\mathbf{D})$. Because the last element of the path ($B_{(m)}$) is by definition either Z or Y, we conclude there exists a sub-path π'' of π'

$$\pi'' = X \to B_{(1)} - \cdots - B_{(r)} - Z$$

that does not pass through variable Y. We know that there must be some edge in π'' that is directed as $B_{(j)} \leftarrow B_{(j+1)}$. Otherwise, there would be a directed path from X to Z in $\mathcal{G}$; if Z is a parent of X this would mean $\mathcal{G}$ contains a cycle, and if Z is a parent of Y this would mean X is an ancestor of Y. Without loss of generality, let $B_{(j)} \leftarrow B_{(j+1)}$ be the *first* edge so directed:

$$\pi'' = X \to B_{(1)} \to \cdots \to B_{(j-1)} \to B_{(j)} \leftarrow B_{(j+1)} - B_{(j+2)} - \cdots - B_{(r)} - Z$$

Because π'' is a sub-path of π—and because neither endpoint X nor endpoint Z is an element in $\mathbf{W}$ —we know from Proposition 3 that it is active given conditioning set $\mathbf{W}$, and thus because it contains the collider $B_{(j-1)} \to B_{(j)} \leftarrow B_{(j+1)}$, we know from Definition 1 that there is a $W \in \mathbf{W}$ such that either $B_{(j)} = W$ or $B_{(j)}$ is an ancestor of W. Because $B_{(j)}$ is a descendant of X, it follows that W is also a descendant of X, and the proof is complete. □

The following three facts about mutual information are well-known. See, for example, Cover and Thomas [3].

Fact 1. $\mathit{Inf}_p(\mathbf{X};\mathbf{Y}|\mathbf{Z}) = 0$ *if and only if* $\mathit{Ind}_p(\mathbf{X};\mathbf{Y}|\mathbf{Z})$.

Fact 2. *For any p,* $\mathit{Inf}_p(\mathbf{X};\mathbf{Y};\mathbf{Z}) \geqslant 0$ *for all* $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$.

The last fact is known as the *chain rule* for mutual information.

Fact 3. $\mathit{Inf}_p(Y; X_1,\ldots,X_n|\mathbf{W}) = \sum_{i=1}^{n} \mathit{Inf}_p(Y; X_i|X_1,\ldots,X_{i-1},\mathbf{W})$.

Lemma 4. *Let p be any distribution. Then*

$$\mathit{Ind}_p(X;Y|\mathbf{W},Z) \Rightarrow \mathit{Inf}_p(Z;Y|\mathbf{W}) - \mathit{Inf}_p(Z;Y|\mathbf{W},X) = \mathit{Inf}_p(X;Y|\mathbf{W})$$

Proof. We expand the quantity $\mathit{Inf}_p Y X, Z\mathbf{W}$ using the chain rule with two different orders for the variables to obtain

$$\mathit{Inf}_p(Z;Y\mathbf{W}) + \mathit{Inf}_p(X;Y|\mathbf{W},Z) = \mathit{Inf}_p(X;Y|\mathbf{W} + \mathit{Inf}_p(Z;Y|\mathbf{W},X)$$

From the independence assumption and Fact 1 we have $\mathit{Inf}_p(X;Y|\mathbf{W},Z) = 0$ and the lemma is established. □

Finally, we can prove the theorem.

Theorem 5. *Let $\mathcal{G}$ be any DAG and let p be any distribution in* $\mathbf{MDF}(\mathcal{G}) \cap \mathbf{Perfect}(\mathcal{G})$*, where* $\mathbf{MDF}(\mathcal{G})$ *is defined using either of the two definitions of an active path.*

$$\mathit{Dsep}_{\mathcal{G}}(X;Y|\mathbf{V}) \Rightarrow \mathit{Ind}_p(X;Y)$$

Proof. Suppose this is not the case, and that $\mathit{Dsep}_{\mathcal{G}}(X;Y|\mathbf{V})$ but there exists some $p \in \mathbf{MDF}(\mathcal{G}) \cap \mathbf{Perfect}(\mathcal{G})$ in which X and Y are not marginally independent. Because X and Y are d-separated given $\mathbf{V}$, we know that X and Y are not adjacent in $\mathcal{G}$ and thus by Lemma 1 they are d-separated given $\mathbf{Pa}_X^{\mathcal{G}} \cup \mathbf{Par}_Y^{\mathcal{G}}$. Let $\mathbf{D}$ be any minimal subset of $\mathbf{Pa}_X^{\mathcal{G}} \cup \mathbf{Par}_Y^{\mathcal{G}}$ for which $\mathit{Dsep}_{\mathcal{G}}(X;Y|\mathbf{D})$; by *minimal* we mean no proper subset of $\mathbf{D}$ also satisfies this property. We know that $\mathbf{D}$ has at least one element because otherwise, by virtue of the fact that $p \in \mathbf{Perfect}(\mathcal{G}) \subseteq \mathbf{Markov}(\mathcal{G})$, X and Y would be marginally independent.

Because $\mathcal{G}$ is a DAG, we know that X and Y cannot be ancestors of each other and thus, without loss of generality, we assume that X is not an ancestor of Y. Let Z be any element of $\mathbf{D}$. From Lemma 3, we know that for $\mathbf{W} = \mathbf{D} \setminus Z$, we have $\mathit{Active}_{\mathcal{G}}(Z;Y|\mathbf{W}) \subseteq \mathit{Active}_{\mathcal{G}}(Z;Y|\mathbf{W},X)$ and thus because $p \in \mathbf{MDF}(\mathcal{G})$, it follows that $\mathit{Inf}_p(Z;Y|\mathbf{W}) \leqslant \mathit{Inf}_p(Z;Y|\mathbf{W},X)$, or equivalently, $\mathit{Inf}_p(Z;Y|\mathbf{W}) - \mathit{Inf}_p(Z;Y|\mathbf{W},X) \leqslant 0$. From Lemma 4, however, it follows that this difference is equal to $\mathit{Inf}_p(X;Y|\mathbf{W})$; because information is non-negative (Fact 2), it follows that $\mathit{Inf}_p(X;Y|\mathbf{W}) = 0$ and we conclude from Fact 1 that $\mathit{Ind}_p(X;Y|\mathbf{W})$. Because $\mathbf{W}$ is a proper subset of $\mathbf{D}$, we know from the minimality of $\mathbf{D}$ that p cannot be perfect, yielding a contradiction. □

References

[1] J. Cheng, R. Greiner, J. Kelly, D. Bell, W. Liu, Learning Bayesian networks from data: An information-theory based approach, Artificial Intelligence 137 (2002) 43–90.

[2] D.M. Chickering, C. Meek, D. Heckerman, Large-sample learning of Bayesian networks is NP-hard, Journal of Machine Learning Research 5 (2004) 1287–1330.

[3] T.M. Cover, J.A. Thomas, Elements of Information Theory, John Wiley and Sons, Inc., New York, 1991.

[4] C. Meek, Strong-completeness and faithfulness in belief networks, in: S. Hanks, P. Besnard (Eds.), Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence, Montreal, QU, Morgan Kaufmann, San Mateo, CA, 1995, pp. 411–418.

[5] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann, San Mateo, CA, 1988.

[6] P. Spirtes, C. Glymour, R. Scheines, Causation, Prediction, and Search, second ed., MIT Press, Cambridge, MA, 2000.

[7] T. Verma, J. Pearl, Equivalence and synthesis of causal models, in: M. Henrion, R. Shachter, L. Kanal, J. Lemmer (Eds.), Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence, 1991, pp. 220–227.

Available online at www.sciencedirect.com
SCIENCE @ DIRECT®
Artificial Intelligence 170 (2006) 667–685
Artificial Intelligence
www.elsevier.com/locate/artint

Rough intervals—enhancing intervals for qualitative modeling of technical systems

M. Rebolledo

Universität Stuttgart, Institute of Industrial Automation and Software Engineering, Pfaffenwaldring 47, 70569 Stuttgart, Germany

Received 8 October 2004; received in revised form 29 December 2005; accepted 15 February 2006

Abstract

The success of model-based industrial applications generally depends on how exactly models reproduce the behavior of the real system that they represent. However, the complexity of industrial systems makes the construction of accurate models difficult. An alternative is the qualitative description of process states, for example by means of the discretization of continuous variable spaces in intervals. In order to reach the required precision in the modeling of complex dynamic systems, interval-based representations usually produce qualitative models, which are sometimes too large for practical use. The approach introduced in this paper incorporates vague and uncertain information based on principles of the Rough Set Theory as a way of enhancing the information contents in interval-based qualitative models. The resulting models are more compact and precise than ordinary qualitative models.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Rough set; Qualitative modeling; Qualitative reasoning; Interval-based knowledge representation; Vagueness and uncertainty management

1. Introduction

Many industrial systems rely on models to support applications such as process simulation, advanced control, fault diagnosis and online monitoring. Working on a correct and precise system model is always cheaper, faster, easier and safer than working with the real system. But modeling real systems is not always straightforward. Models can only be employed if they mimic the original system precisely. However, this model precision, i.e. how well this model reproduces the system behavior or characteristics, is not determined by the model itself, but by the intended application, and is therefore often more a requirement than a feature. A further problem is that industrial systems are usually too complex to be described, meaningfully and wholly, with precise deterministic models. To make matters worse, system complexity and precision demand in model-based applications increase with each new technological development.

Humans are able of abstracting complex processes and describing them qualitatively using expert knowledge and common sense. Therefore, a way of dealing with the described problems is focusing only upon the system aspects to be analyzed, enhanced or supervised, and representing these aspects in qualitative models. Such qualitative models usually describe continuous variables by dividing the spaces where they are defined in intervals [3,15]. Yet, using in-

E-mail address: mrrw2002@yahoo.com (M. Rebolledo).

0004-3702/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2006.02.004

Table 1
Representing qualities as binary messages

(a)	A	AB	B	BC	C
	0	1	0	0	0

(b)	A	B	C
	1	1	0

tervals to represent continuous physical dimensions implies losing information about the system behavior during state transitions, because an interval-based partitioned variable can change its qualitative value instantaneously. Defining additional transitional intervals reduces this information loss. Important transitions can be well described, for instance, using one-point intervals, but it is impossible to describe the behavior of the system as it nears or leaves these points. Besides, defining one-point intervals may result in models that are too large to describe entire complex dynamic systems.

Interval-based qualitative models are analogous to using just one bit at a time in a N-bits message (e.g. 10000, 01000, 00100). This is underutilizing the 2^N representation capability of these N-bits. However, because of continuity, it is not always the case that 2^N different messages are physically possible. Using a message such as 01001 to describe a particular system state may be nonsense. For instance, a given continuous variable may be represented using three intervals (representing qualities A, B and C respectively) and two one-point transitional intervals. The equivalent binary representation is a 5-bits word, which is only allowed to get one of five possible values: 10000, 01000, 00100, 00010 or 00001. Table 1(a) shows the equivalent binary representation of a system in the transition point between quality A and quality B.

This paper introduces a concept to enhance intervals as information representation framework, by allowing the overlapping of qualities in intermediate states. This eliminates, for example, the need of defining one-point transitional intervals and makes easier modeling the vagueness and uncertainty in transitions. Following the previous example, this would correspond to using only three bits to code the five possible states of the variable (Table 1(b)). Of course, binary values 000 and 101 are still not allowed.

The proposed concept takes advantage of principles of the Rough Set Theory. The resulting representation, the Rough Interval, is more suited for dealing with partially unknown or ill-defined parameters and variables. Additionally, Rough Intervals are capable of coding more information than ordinary intervals, as it is suggested by the example of the analog binary representation. The resulting qualitative models are, therefore, more compact for a required precision, or the same, more precise for a given model size. This improvement in the relation precision/size in qualitative models expands the applicability of qualitative modeling methods in complex systems. The new concept was integrated in the qualitative modeling method SQMA (Situation-based Qualitative Modeling and Analysis) [33], which allowed the verification of the described enhancements. This concept is, however, independent from SQMA and can therefore be integrated into other qualitative reasoning techniques as well.

2. Theoretic framework

The following sections describe briefly the techniques and methods that served as basis for the development and further application of the proposed concept. In the first place, the most cited qualitative reasoning techniques are introduced, followed for a detailed analysis of interval-based methods. This gradual approach, from general to particular, closes with a brief description of SQMA, technique used as case study. The Rough Set Theory, the theory upon with the interval-based representation of qualitative information is enhanced, considering vagueness and uncertainty, is introduced in this section as well.

2.1. Qualitative reasoning

A way of dealing with system complexity is describing system behavior using rules or tables, based on experience and common sense. One approach that implements this idea is *qualitative reasoning*, which approximates human thinking and reasoning by capturing fundamental system behavior in a computer model, while suppressing much of the detail. Qualitative reasoning is an area of Artificial Intelligence that supports reasoning with very little information, providing promising tools for research and engineering activities.

The first of two processes normally included in qualitative reasoning is Qualitative modeling. It abstracts continuous aspects of real processes, such as space, time and other physical quantities to model the fundamental behavior of a system. The second process is *qualitative simulation*, which uses previously determined qualitative models to predict possible behaviors of the modeled system. In order to achieve this, a qualitative simulator generates an expected output from the real system's inputs based on the qualitative model and compares simulated and actual behavior. Important representatives of qualitative reasoning are the methods introduced by de Kleer and Brown [15], Kuipers [3], Forbus [18,19], Lunze [16] and Laufenberg and Fröhlich [33]:

- De Kleer's approach employs a special *Physics based on Confluences* for qualitative system modeling. Confluences are the qualitative depiction of differential equations, where only the symbols $[-]$, $[0]$, $[+]$ and $[?]$ (for negative, zero, positive and undetermined) are accepted as arguments. Confluences are managed with sign arithmetic represented in a tabular form. This arithmetic includes the derivative function $\delta[x] = \mathrm{Sign}(\mathrm{d}x/\mathrm{d}t)$ to represent system dynamics.
- Kuipers' *Qualitative Simulation* (QSim) is one of the most cited qualitative reasoning methods. QSim's qualitative model uses an interval-based representation of physical variables and qualitative differential equations. Intervals and qualitative differential equations are operated based on the sign $(+/-)$ of the variables and threshold values associated with qualitative system changes.
- The *Qualitative Process Theory* introduced by Forbus regards the modeled system as a collection of *Objects*, *Quantities* and *Processes*. *Objects* are entities in the universe of the problem that are capable of interacting with each other. *Quantities* describe the continuous properties of *Objects* in terms of ordinal relations, while *Processes* modify the *Objects*, their relationships and their properties. The Qualitative Process Theory also models the possibility of entities being created and destroyed, as part of a dynamic universe.
- Lunze introduced an approach based on the qualitative representation of the process dynamics with nondeterministic automata. This approach considers process states and events as sets of qualities, which results in a *Qualitative Automaton*. If the probability of each particular transition is represented in the model, the resulting state machine is known as a *Qualitative Stochastic Automaton*.
- Laufenberg and Fröhlich's *Situation-based Qualitative Modeling and Analysis* approach (SQMA) models a system using interval-based qualitative descriptions of situations that can take place. These situations, gathered in a table of valid situations, correspond in technical systems to a qualitative description of the finite set of states where the process can reside. SQMA serves as case study for the evaluation of the proposed representation concept and is, for this reason, described in detail in Section 2.3.

The above described approaches have been employed, for example, to monitor and analyze the correct functioning of industrial processes. Many successful process monitoring and diagnosis applications, such as MIMIC [5,6], SQUID [13] and SQMD [30], have been developed based on these methods. However, qualitative modeling has been also applied to the treatment of problems in other areas, such as biological [11] and socio-economical [10] systems, and to support academic activities as in the case of CyclePad [20]. Each technique has strengths and weaknesses, according to its application. In common, they have their capability of handling incomplete information by system modeling.

2.2. Interval based qualitative representation

In mathematics, a container that includes or excludes any given element of a completely defined universe of discourse is called a *set*. According to the Classic Set Theory, an object (element) can either be part of a given set or be excluded; i.e. each element of the universe of discourse is either in the set or out of it. A classic set can represent any idea or concept, as shown in the example in Fig. 1, where the set C should contain all the big objects of the depicted universe of six elements.

Even though concept borders in most real problems are not crisp, classic sets are usually employed to approximate them, as it is shown in Fig. 1. But objects 3 and 4 have almost the same size, with object 3 just a little bit bigger. It can be considered, for instance, that object 4 is also big. On the other hand, it is perhaps not big enough. Similar

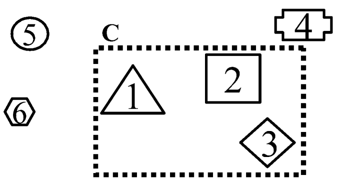


Fig. 1. Classic set C of the big objects.

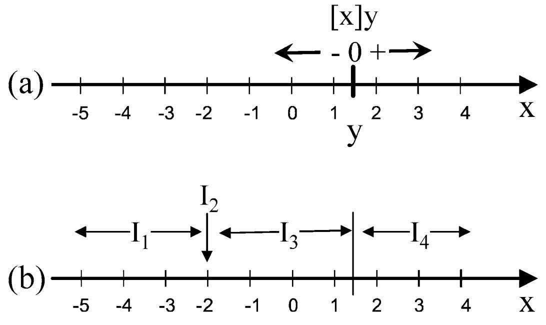


Fig. 2. Prefix and interval representation.

questions can be formulated about object 3. A classic set resolves only approximately this problem by adopting a trade-off position, such as that of the set C.

This classic set notion can be adapted to continuous variables (e.g. the object's cross area) instead of objects, just by delimiting value ranges in the dimension where these variables are defined. Value ranges are frequently used for the depiction of vague concepts and imprecise values, such as in the case of "big" in the previous example. They are usually associated with specific system characteristics or behaviors enabling the qualitative modeling of a system. Value ranges can be used to represent discrete and logic values as well, which offers a common framework for the qualitative modeling of real process variables and parameters.

A form of value range notation is the prefix sign representation relative to a reference point [15]. In this notation system, a qualitative value is derived from the prefix of the quantitative value in the reference system determined by the referential point. Fig. 2(a) shows this kind of qualitative representation. $[x]y$ is a qualitative variable with the values "+", "0" or "−"; where x is a variable and y is a parameter that serves as reference point for the qualitative variable $[x]y$. The qualitative variable $[x]y$ obtains respectively, for $x < y$, $x = y$ and $x > y$, the qualitative values "−", "0" and "+". It is valid therefore that $[x]0 = \mathrm{Sign}(x)$. De Kleer and Brown's Physics based on Confluences [15] relies on prefix sign representation. Additionally, this notational principle is generally used to represent process dynamics in qualitative models.

Based on the principle above-described, many reference points may be defined at will, in order to generalize the interval bordering. The result of the superposition of these qualitative variables is an interval representation. The interval $[a\ b]$, with $a < b$, describes the value range between the reference points a and b. For example, $I_3 = [-2\ 1.4]$ in Fig. 2(b) includes the values 0, 1, and 0.5 among others. Following this notation, intervals of the form $[a\ a]$ describe the reference point exactly as the corresponding real number a (e.g. $I_2 = [-2\ -2] = -2$ in Fig. 2(b)). Closed and open interval limits (square brackets and parentheses respectively) define whether a given reference point must be included in the interval or not. In any case, the borders of these intervals are crisp (from here the name "crisp interval"); there is no continuous transition or common point between adjacent intervals. Finally, there are also unbounded intervals in the form $(-\infty\ a]$ and $[b\ +\infty)$. They describe the area from $-\infty$ to point a ($I_1 = (-\infty\ -2]$ in Fig. 2(b)) and the region between the reference point b and $+\infty$ ($I_4 = [1.4\ \infty)$ in Fig. 2(b)). It is not necessary to portray the entire dimension; it often suffices representing excerpts of it.

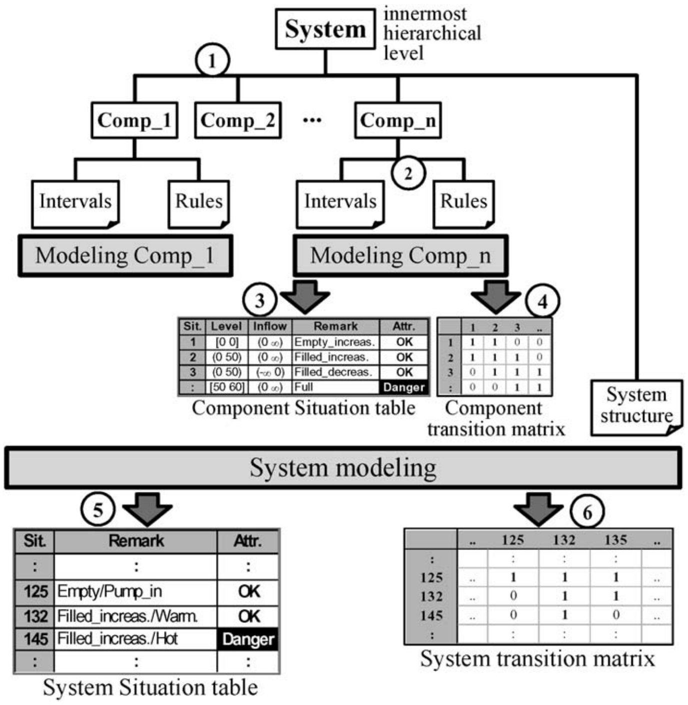
For working with intervals, Moore [29] developed the interval arithmetic as an extension of the basic arithmetic operations with real numbers (addition, subtraction, etc.). This arithmetic also redefines the relational operators. A relationship, such as "greater than", "lower than" or "equal to" is satisfied, as soon as individual points of the compared intervals satisfy it. Equality, for example, is determined based on the existence of intersection (i.e. common points) between intervals. Relational operators in interval arithmetic are therefore nondeterministic, i.e. for any two overlapping intervals A and B it can be possible to satisfy the three relationships $A = B$, $A > B$ and $A < B$ at the same time. This is the case, for example, of the intervals (1 5) and [3 6). This nondeterminism reproduces the natural nondeterminism in human qualitative appraisals.

Interval arithmetic enabled the realization of complex calculations with intervals, and thus, the development of the interval-based qualitative methods. Situation-based Qualitative Modeling and Qualitative Automata are two methods based on the representation of physical variables using previously defined intervals. One of the most cited qualitative modeling techniques, Kuipers's Qualitative Simulation, defines intervals whose reference values are dynamically determined during the simulation process.

2.3. Situation-based qualitative modeling and analysis, SQMA

Situation-based Qualitative Modeling and Analysis (SQMA) is a modeling method developed at the Institute of Industrial Automation and Software Engineering of the Universität Stuttgart. SQMA was conceived for safety-related applications, such as hazard analysis [31], fault detection [25,30], diagnosis [2] and reliability assurance [12], which require the comprehensive modeling of complex systems. SQMA models a system using qualitative descriptions of the situations that can take place. These situations, gathered in a table of valid situations, correspond in technical systems to a qualitative description of the finite set of states where the process can reside. Each situation considers for each system variable a particular qualitative value, which is represented by an interval. SQMA modeling works as shown in Fig. 3.

The system modeler structures the whole system hierarchically and decomposes the innermost level (#1 in Fig. 3) into components. After that, component variables are modeled using intervals and characteristic values represented as one-value-intervals (#2 in Fig. 3). Physical rules that will be used for the situation verification are formulated using interval arithmetic to complete the description of each component. The computer-aided modeling of a component checks every possible interval combination (component situation). A given component situation is valid if it complies with all the rules declared for the corresponding component. This makes it possible to detect and to remove impossible



Sit.	Level	Inflow	Remark	Attr.
1	[0 0]	(0 ∞)	Empty_increas.	OK
2	(0 50)	(0 ∞)	Filled_increas.	OK
3	(0 50)	(-∞ 0)	Filled_decreas.	OK
:	[50 60]	(0 ∞)	Full	Danger

	1	2	3	..
1	1	1	0	0
2	1	1	1	0
3	0	1	1	1
:	0	0	1	1

Sit.	Remark	Attr.
:	:	:
125	Empty/Pump_in	OK
132	Filled_increas./Warm.	OK
145	Filled_increas./Hot	Danger
:	:	:

	..	125	132	135	..
:		:	:	:	
125	..	1	1	1	..
132	..	0	1	1	..
145	..	0	1	0	..
:		:	:	:	

Fig. 3. Situation based qualitative modeling, SQMA.

interval combinations. An SQMA component model is defined upon these valid interval combinations collected in a component situation table (#3 in Fig. 3), which describes the normal behavior of the component.

Envisioning in SQMA components relies on a transition rule testing procedure. This procedure is similar to the procedure implemented for situation validation. It uses the relationships determining the system dynamics, which are described with interval arithmetic as well, to validate the transitions between component situations. The modeling tool checks each possible situation transition and marks ("1") the valid ones, while invalid transitions remain unmarked ("0"). A quadratic matrix with situations in rows and columns, the component transition matrix (#4 in Fig. 3), collects the result of this validation and completes the component model. This matrix, representing all states the system can take, with all of the transitions between them, is formally known as its envisionment in qualitative reasoning literature. SQMA component models can be reused (in case of similar components in the system) or further combined at system level. Moreover, complete component models can be stored in a component library for future use.

To model the current hierarchical system level, the situation tables of all the components at this level are combined. An automated procedure assembles the rules that describe the relationships between these components and between components and system terminals following the way they are connected with each other. These system rules express material and energy balances, similar to Kirchhof's laws in electric circuits, and are used to validate coincidences of component situations collected in an initial system situation table. The system rule validation follows the same rule verification primitives employed for situation and transition validation in the components. That results in a situation table, which lists all the possible behaviors of the entire system (#5 in Fig. 3).

The corresponding system transition matrix (#6 in Fig. 3) is built by verifying the component transition matrices after the following rule: *A transition between two system situations* (in the system transition matrix) *is only possible* (and therefore marked), *if the corresponding transition* (in the component transition matrix) *is also possible for each single component*. The envisioning at system level completes the situation based qualitative model of the current hierarchical level. The structure of this model is identical to the structure of component models, and can by similarly reused, further combined or stored in an SQMA model library.

In very complex systems, it can be necessary to resolve several hierarchical levels to model the whole system. For example, pumps, valves, heaters and vessels can be modeled as single components and combined to model evaporators, reactors or distillation columns. These process units can be combined, following different configurations, to form a number of bigger and more complex process units. These process units can then be combined to model first process plants and then entire petrochemical or refinery complexes with a group of plants, tens of processing units and hundreds of single-component types. In these cases, the described procedure is repeated for each hierarchical level until the completion of the system model.

A common problem of qualitative modeling techniques is the size of envisionments that result from modeling (for instance) entire process plants. In the same way, SQMA transition matrices grow very fast with the complexity and size of the modeled system. Even if building up the envisionment hierarchically is faster and easier, the resulting table can be too large for practical use. An advantage of SQMA, however, is the possibility of generating and managing models at multiple levels of abstraction. This allows a convenient definition of the different hierarchical levels to avoid or at least to mitigate this problem.

2.4. Rough Set Theory

Pawlak proposed the Rough Set Theory in 1982 [34] as a method for the joint management of vagueness and uncertainty. It is based on the Classic Set Theory, but is inspired by Zadeh's fuzzy sets [23]. Reasoning in the Rough Set Theory is a matter of classification, not a matter of degrees of truth. A basic assumption in Rough Set Theory is that any concept can be defined through the collection of all the objects that exhibit the properties associated with it. Therefore, a vague concept, expressed as a vague classification criterion, is always susceptible of being decomposed in two well-delimited concepts (here, two classification criteria) that can be further handled independently. For a vague concept R, it can be formulated:

- a *lower approximation* R_* containing all the elements that *are surely included* in R, and
- an *upper approximation* R_* containing those elements that *cannot be excluded*, beyond doubt, from R.

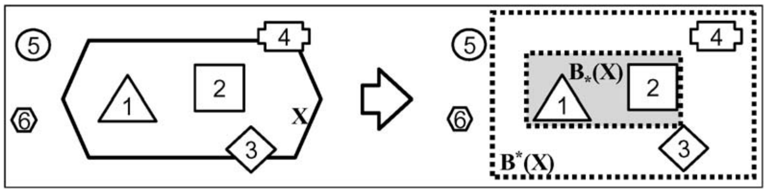


Fig. 4. Vague information with rough sets.

Table 2
Properties of the rough membership

$\rho_X^B(z) = 1$	if $z \in B_*(X)$
$\rho_X^B(z) = 0$	if $z \notin B^*(X)$
$0 < \rho_X^B(z) < 1$	if $z \in BN_B(X)$
$\rho_{U-X}^B(z) = 1 - \rho_X^B(z)$	for any $z \in U$
$\rho_{X \cup Y}^B(z) \geqslant \max(\rho_X^B(z), \rho_Y^B(z))$	for any $z \in U$
$\rho_{X \cap Y}^B(z) \leqslant \min(\rho_X^B(z), \rho_Y^B(z))$	for any $z \in U$

Another important premise is that some information about the elements of the universe of discourse is available. If the same information can be associated with several elements of this universe, these elements are indiscernible; i.e. they cannot be separated from each other. Hence, the lower and the upper approximation of a set X, representing the original vague concept, can be defined based on this indiscernibility regarding the available information B:

$$\text{Lower approximation:}\quad B_*(X) = \{z \in U\colon\ B(z) \subseteq X\}, \tag{1}$$

$$\text{Upper approximation:}\quad B_*(X) = \{z \in U\colon\ B(z) \cap X \neq 0\}, \tag{2}$$

$B(z)$ represents in these definitions the set of all the elements of U, which are indiscernible from z based on the information in B.

Fig. 4 illustrates the classification-based notion of rough set. The left side shows the elements $z \in U = \{1, 2, 3, 4, 5, 6\}$ to be classified according to their membership to a vague concept X (e.g. representing the feature "big") and the available information B. The right side of the figure represents the vague concept X following rough set's principles. The inner set B_* (lower approximation) comprises elements $\{1, 2\}$ that are unequivocally included in X. The outer set consists of elements $\{5, 6\}$ that definitively are not part of X(e.g. they are small objects). The classification of $\{3, 4\}$ is uncertain, e.g. it is not possible to decide whether they are big or small. Elements $\{1, 2, 3, 4\}$ define together the upper approximation of X.

The vagueness problem is so transformed in an uncertainty problem, which is concentrated in the set $B^*(X) - B_*(X)$, known as the Boundary region ($BN_B(X)$) of the concept X based on the information B. This boundary set represents the uncertainty in the concept, where no solid conclusion can be reached about whether objects are big or small. A probability-based rough membership function is then necessary to address the uncertainty problem. This rough membership can be defined using the indiscernibility relationship:

$$\rho_X^B(z) = \frac{|X \cap B(z)|}{|B(z)|}. \tag{3}$$

The symbol ρ will be used to represent rough membership instead of the generally used μ, with the intention of explicitly marking the difference with fuzzy membership, where the symbol μ is also employed. The function $|\ldots|$ represents the size of the set, i.e. the number of elements in it. Table 2 shows the most important properties of the rough membership function (see [35] for a complete list). There is no further restriction about the form of a rough membership function beyond these properties.

3. Enhancing intervals considering vagueness and uncertainty

Most variables in industrial processes are defined in continuous numerical spaces, and most of these continuous variables can only change their values gradually, because their behavior is bound with the energy flow in the system.

Under these conditions, representing dynamic systems based on crisp intervals is very imprecise. Crisp intervals provide static views of the continuous dynamic variables; therefore, interval-based models can only provide static views of reality.

This section describes a new approach for partitioning continuous variables, which is based on Rough Intervals. The approach takes advantage of the representation and management of vagueness and uncertainty in interval-based qualitative models, applying principles of the Rough Set Theory to improve the model precision and compactness.

3.1. Interval-based management of uncertainty and vagueness

The term "incomplete information" is usually cited in association with interval-based qualitative modeling. However, this term does not necessarily mean that the system information is incomplete in the sense of its availability (partial or total ignorance) or in any way insufficient. For instance, even though intervals can be arbitrarily chosen, a deep knowledge about the system may support a better partition of the variable spaces. Incomplete information in qualitative modeling can be associated with uncertainty or ignorance about the process, but also with the deliberate simplification (abstraction) of system information before using it to build a model. Crisp intervals are successfully employed to approximate this kind of "incomplete" information. They allow representing the most probable parameter values or scenarios in qualitative relationships. Another problem, however, regards the uncertainty arising from the nondeterministic nature of qualitative models [24], in the form of the typically huge envisionments delivered by qualitative reasoning methods. Interval-based qualitative models do not offer indexes or measures such as probabilities or possibilities [4], which could support the analysis of a solution space containing multiple alternative behaviors.

A different aspect is the modeling based on approximate or inaccurate descriptions of a system. Modeling vague level concepts such as "too high" or "almost empty" would require defining new intervals for each one of these cases, but the definition of these additional intervals would mean enlarging the model drastically. The space of a qualitative model grows in a geometrical progression with the number of intervals defined on its variables. Thus, it is necessary to achieve a fair qualitative representation of vague information without increasing the model size.

An alternative way of representing vagueness is using weighting values to express the confidence on the available information. Qualitative modeling methods, such as FuSim [28], QuaSi [1] and FRenSi [32], follow this approach. They introduce the management of vague information using fuzzy sets and fuzzy numbers. FRenSi and QuaSi, for instance, preprocess vagueness using fuzzy numbers and then transform the intermediate results into equivalent intervals to continue operating with interval arithmetic, while FuSim adapts QSim and Moore's interval arithmetic to work with fuzzy sets, which results in a complex hybrid fuzzy/interval notation and an extended fuzzy arithmetic. However, none of these methods take uncertainty into account.

A new method is required, which offers a consistent solution for the compact modeling and further handling of vagueness and uncertainty in interval-based qualitative models. Hence, a method based on rough sets is proposed. Rough Set Theory unifies vagueness and uncertainty as different aspects of the same problem. Additionally, information representation with rough sets is based on crisp sets, which is less demanding than representing and processing continuous fuzzy membership functions.

3.2. Rough intervals

Rough sets are defined over sets and elements and therefore cannot be used for the representation of continuous variables [9,22] as it is the case with crisp intervals and fuzzy sets. Rough Intervals are introduced to adapt the rough set principles to the modeling of continuous variables. A Rough Interval (RI) is a particular case of a rough set. They fulfill all the rough set's properties and core concepts, including the upper and lower approximation definitions. A Rough Interval comprises two parts: an Upper Approximation Interval (UAI) and a Lower Approximation Interval (LAI). The variable could assume inside of the UAI the represented qualitative value (a vague concept in rough sets), or what is the same, it is clear that the variable cannot get this qualitative value outside of this interval. The second element of a RI, the LAI, can be also redefined on this basis; In the LAI it is sure that the variable takes the represented qualitative value.

The Lower and Upper Approximation Interval concepts satisfy the mathematical definition of rough set's upper and lower approximation in (1) and (2), adapted to intervals. If a particular qualitative value C must be represented over a variable, the two enveloping intervals, A^* and A_*, can be defined. The implications (4) and (5) represent the

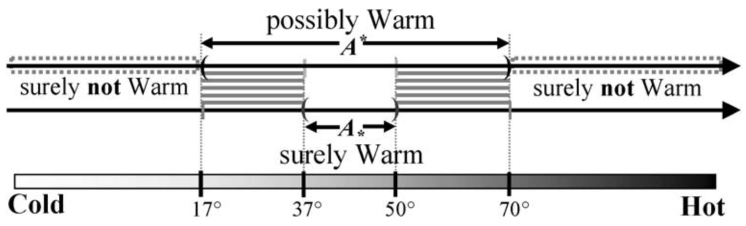


Fig. 5. "Warm" represented as a Rough Interval.

relationship between the variable $x \in U$ (where the universe U can be any complete set of continuous or discrete values), the qualitative value A defined on x, and the intervals A^* and A^*.

$$x \in A^* \Rightarrow x \in A, \tag{4}$$

$$x \notin A^* \Rightarrow x \notin A. \tag{5}$$

No special knowledge is required to use these simple definitions as design rules. They do not complement each other, in fact, the second one is a subset of the first one; the Lower Approximation Interval can be and must be defined inside of the Upper Approximation Interval. For instance, the definition of "Warm" in a temperature variable may be considered. Common sense and general knowledge can help defining its limits, for example:

– The temperature of the human body is about 37 °C. Nothing warmer will be considered "Cold".
– In average, a human hand cannot hold an object, whose temperature is over 70°C, because it is "Hot".
– If something is colder than the environment (let us say 17 °C), it cannot be considered "Warm" anymore.

A few statistical considerations, together with some sense of symmetry may assist an expert completing the definition of this Rough Interval and its neighbors. A set of two crisp intervals can represent the resulting RIs. The Rough Interval shown in Fig. 5 represents the qualitative value "Warm". The key fact in this example was the use of precise concepts to define an imprecise one. They may be supported by verifiable knowledge, statistics or physical laws, which are in general measurable, trustworthy and easier to model than the original vague concept.

The Rough Set Theory reduces the vagueness of a concept to uncertainty areas at their borders. Within these uncertainty areas, in Rough Intervals as in rough sets, no definitive conclusion about the problem is possible. Hence, rough membership values (ρ) must be defined for Rough Intervals as well. These rough memberships membership values, expressed in rough membership functions, must satisfy all the conditions listed in Table 2 and [35], but also the following properties:

Complementarity. RI overlaps are transient areas between two clearly typified regions (LAIs, where the membership of any value to the RI is 1). In an overlap, the variable changes gradually from one quality to another, like from "warm" to "cold" or "short" to "long". Thus, the rough membership of a given real value inside of this transition area to the one RI decreases, as the rough membership of the same real value to the adjacent one grows. Consequently, totalizing the rough membership of any value to its corresponding RIs will always result in 1.

Monotonicity. In a numeric space, there is a relation of precedence and consequence, an order among its elements, which determines the necessity of using monotonic growing and decreasing functions in the boundary regions of a Rough Interval to represent these rough membership values.

Border Conditions. Rough membership is an expression of the probability of being part of a clearly defined concept, which is in RI delimited by the LAI. Thus, and considering the properties above, a given number in the boundary region is part of the represented concept with a probability that increases with the proximity of this number to the LAI, where this probability, and consequently the rough membership function, reaches its maximum (1). For the same reason, a rough membership function has its minimum (0) at the borders of the UAI.

Vague intermediate qualitative values such as "between warm and cold", "almost cold" or "not warm enough" can describe the uncertainty in the boundary regions, or what is the same, vague qualitative values can be represented as the

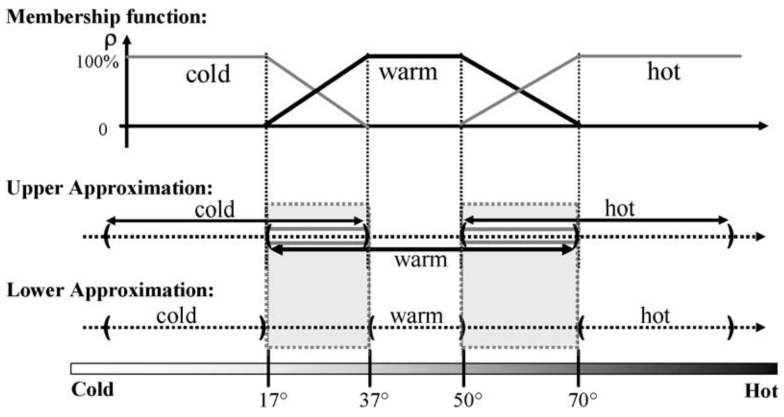


Fig. 6. The Rough Interval notation.

superposition of two clearly defined intervals. By analyzing a particular value inside of these overlaps, the qualitative values corresponding to both intervals must be considered together with their complementary rough memberships. This enhances the precision and descriptive power of the model in the transition areas, where a higher degree of detail is usually required.

For practical reasons a straight linear function is adopted to describe rough interval membership (Fig. 6, top). It not only satisfies all the conditions described above offering a trade-off among the infinite possibilities, but also implies the lowest calculation and representation effort, which at the end results in a more efficient model processing. This linear representation of the probabilistic rough membership function corresponds to the assumption of a uniform probability distribution, which is adopted, as a rule, as worse case scenario for the representation of systems under uncertainty, i.e. when no information about the probabilistic distribution of the modeled events is available.

The separate representation of UAIs and LAIs makes evident the redundancy in this notation (shaded area in Fig. 6). First, the empty spaces in the Lower Approximation Intervals coincide exactly with the overlapping of Upper Approximation Intervals. Second, the Rough Interval membership function can be totally reproduced from any of both segments, without further parameters. Hence, just the overlapping representation of the UAIs will be taken as notation for the Rough Intervals, which is not different from the interval-based notation employed for qualitative modeling.

The described representation and interpretation framework, based on the UAIs, could have been developed using the LAIs instead. However, the superposition of qualitative features (overlaps) is a notion that is closer to the human understanding (e.g. "almost cold" can be understood as cold and warm at the same time) than representing featureless empty areas with qualitative descriptors assigned ad hoc ("almost cold" $\neq$ "neither cold, nor warm"). The chosen approach, representing UAIs, allows handling Rough Intervals independently from their overlaps during the modeling. Overlaps emerge from the evaluation of a real situation based on the qualitative model.

Indiscernibility [34], a property defined inside of the Rough Set Theory, allows the extraction of precise information from vague concepts, by comparing these vague concepts with the available information about the problem. Based on these properties Rough Intervals associate qualitative values to the different value ranges, as crisp intervals do. Furthermore, RI can represent the indiscernibility in the region between contiguous qualitative concepts, providing a new framework for the interpretation of interval-based qualitative models based on indiscernibility.

The resulting crisp-interval-like notation is crucial for the integration of Rough Intervals in interval-based qualitative methods. So represented, there is no difference between Rough Intervals and overlapping crisp intervals. This representation assures a minimal impact with the introduction of the Rough Interval concept in interval-based modeling procedures. Qualitative models with crisp intervals and RIs are identical. Only the interval overlaps, which are not permitted in the conventional interval-based qualitative methods, indicate the use of RIs.

4. Qualitative computing with Rough Intervals

The question: "how do a qualitative model profit from using RIs instead of common crisp intervals?" must be answered for each particular qualitative modeling and reasoning approach separately. Section 5 makes a detailed analysis of the use of RI in SQMA as case study. Similar analysis can be performed for other qualitative and hybrid techniques such as QSim, SQMD or Lunze's Qualitative Automata. On the other hand, the utility of using RIs in qualitative models and reasoning based on these models must take into account the complexity of computing with RI and how difficult is their determination for a real problem. These two aspects (computing with RI and RI modeling) are addressed in the following sections in detail, taking the modeling and the operations with crisp intervals and fuzzy sets as reference.

4.1. Rough Interval arithmetic

As it was explained in Section 3.2, RIs can be completely represented in a qualitative model using only the UAIs because of the probabilistic complementarity of contiguous RIs. Nevertheless, computing with RI requires using every single RI isolated from other RIs in the same process variable. For this reason, each RI in a single-interval notation (as it is found in the model) must be changed to its equivalent double-interval notation before being used in arithmetical or logical operations. This double-interval notation declares UAI and LAI of the Rough Interval explicitly, separated by a colon (UAI : LAI).

This extended RI notation made possible the definition of arithmetic RI operations based on Moore's Interval Arithmetic [29]. The resulting Rough Interval Arithmetic (RIA) separates an arithmetic operation, such as addition or subtraction, into two identical interval operations. The first one operating over the UAIs and the second using the corresponding LAI as operands. No attention must be paid to membership functions during these operations. For example, with the Rough Interval $A = [4\ 9] : [5\ 8]$ and $B = [2\ 8] : [4\ 6]$, the following operations can be performed:

Addition: $A + B = [4\ 9] : [5\ 8] + [2\ 8] : [4\ 6] = [4 + 2\ 9 + 8] : [5 + 4\ 8 + 6] = [6\ 17] : [9\ 14]$;

Subtraction: $B - A = [2\ 8] : [4\ 6] - [4\ 9] : [5\ 8] = [2\ 8] : [4\ 6] + [-9\ -4] : [-8\ -5] = [-7\ 4] : [-4\ 1]$;

Negation: $-A = -[4\ 9] : [5\ 8] = [-9\ -4] : [-8\ -5]$.

Set and logic operations exploit the "set" nature of the Rough Intervals. The set operations *unification*, *intersection* and *complementation* are defined for the RIA. The same analogies between set and logic operations while working with crisp intervals and fuzzy sets (AND ⇔ intersection; OR ⇔ unification; NOT ⇔ complement) can be translated and adapted for RIs. For the unification of Rough Intervals no changes were introduced, since this operation coincides with that defined for rough sets (see shaded area in Fig. 7(a) for $A \cup B$).

The complement of a Rough Interval must be clearly disjoined from the sign inversion operation (Negation), which regards the interval (not the set) nature of the Rough Intervals. To avoid notational ambiguity, as symbol for RI complement, its logical equivalent "¬ " (NOT) is used instead of the traditional "−", which remains consequent with Moore's interval arithmetic. The complement operation results in the set of all the elements in the universe of discourse that are not contained in the RI introduced as argument for this operation. The shaded area in Fig. 7(c) shows the set $\neg A$.

By the application of the intersection (compare with shaded area in Fig. 7(b) for $A \cap B$), however, particular conditions may arise that require further definition. An arithmetic operation over one or two RIs must always result in a RI. However, the result of a RI intersection may not fulfill the defined notation conventions, as in the intersection of A with the RI $C = [3\ 5] : [4\ 4]$ in Fig. 8. $A \cap B$ produces a new RI $[4\ 5] : [5\ 4]$ (see Fig. 8) with a maximum membership of 0.5, and where the end of the LAI precedes its starting point. In this case, a maximum rough membership lower than one would denote that there is no certainty region about this concept. This kind of RI is called "Minor". Minor

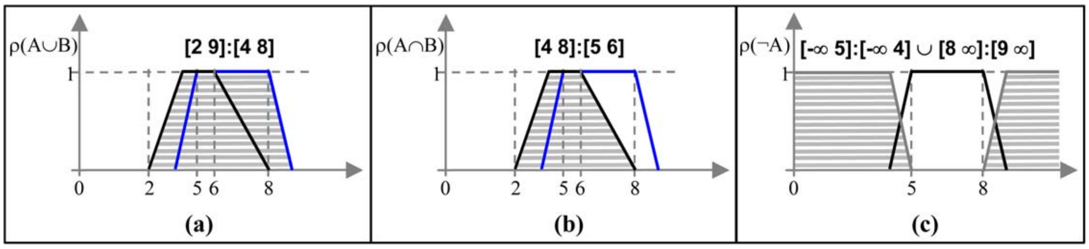


Fig. 7. Set operations in RAI.

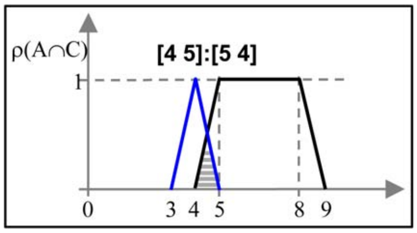


Fig. 8. RI intersection.

RIs represent qualitative values or logic statements without 100% of certainty area, but that cannot be completely discarded.

In general, a logic statement is TRUE (logic 1), if a non-empty RI can be found as result of the logic operations that describe this statement. An empty or non-existing RI would indicate that the logic expression is FALSE (logic 0). The effect of a minor RI resulting from an AND operation can be interpreted as a confidence between 0 and 1 for a TRUE result instead of a clear TRUE or FALSE. "Rough Interval confidence" is introduced here as the maximum membership value that can be reached within a minor RI. According to this definition, RI confidence is always 1 for normal RIs. This value represents the "degree of truth" of the evaluated logic statement.

Even though each arithmetic operation with RI must be performed twice (once with the UAIs and then with the LAIs), the single operations are not more complex than computing with crisp intervals. The same applies for logic and set operations. Only the final interpretation of the RI intersection was enhanced to accommodate the management of vagueness and uncertainty through the definition of a confidence factor. However, as comparison, set and logic operations with RI are simpler than the same operations with fuzzy sets. Fuzzy sets must handle continuous membership functions and cannot take advantage of Moore's interval calculus; in the way RIs do to reduce computational complexity.

4.2. Modeling of Rough Intervals

Determining an adequate partitioning of the process variables is one of the most challenging tasks in building interval-based qualitative models. This is also the case when Rough Intervals are to be used. Nonetheless, techniques adapted from other areas such as Artificial Intelligence, probabilities and curve analysis can support the determination of a basic set of Rough Intervals that can be then adapted to fit the problem requirements. Some approaches to RI modeling [26] are:

(a) *Heuristic definition of Rough Intervals*. Determination of RI based on the interpretation of the variable according to the definitions of Upper and Lower Approximation Intervals. First, the different qualitative descriptors are separately defined based on the available knowledge about the variable, as in the example of "warm" in Section 3.2. The resulting RIs, put together, define a sub optimal partitioning of the space of the variable (compare Fig. 6).

(b) *Experimental definition of Rough Intervals*. In many applications, the knowledge about the process and process variables is insufficient to allow the complete definition of RIs based on the previously described heuristic definition. Assuming that some structural and behavioral information is available, the still-required knowledge about the process parameters may be gained through experimentation. This approach looks for "interesting points (points associated with alarms, events or important changes in the process) using an approximate model of the system. An alternative modeling procedure is based on the concurrent simulation of the system behavior and ad-hoc defined monitoring application. Initial values for RIs can be arbitrarily chosen and gradually tuned by trial and error [8, 24].

(c) *Episodic interval identification*. The Interval Identification method [7] can be adapted to support the identification of Rough Intervals as well. It decomposes an experimental run of the system, where the input variable (the one to be modeled) monotonically increases or decreases, into qualitative "episodes". Episodes are defined through the combination of the sign of the input variable, the sign of the system response and the first and second derivatives of this response. Those value areas of the variable where there is no change of episode would correspond to the

Table 3
Comparison of qualitative representation principles

	Fuzzy subset	Rough Interval	Crisp interval
Representation of	Vagueness	Vagueness and uncertainty	Computational/ measurement imprecision
Membership functions	Trapezoid, Gaussian, sigmoid, ...	Crisp In/Out discrimination with straight membership in boundary reg.	Crisp In/Out discriminations
Overlaps	No restrictions or conditions	Mandatory total overlapping of boundary regions, not allowed in LAI	Not allowed
Cross-dependency	Each set is defined independently	Probabilistic complementarity	No overlaps and no empty spaces allowed
Total membership	$\sum \mu > 1$ allowed (Fuzziness)	$\sum \rho = 1$ at any point (Probability)	$\sum m = 1$ at any point ($m \in \{1, 0\}$)
Interpretation	Membership or degrees of truth, possibility	Probability of being member of the interval	Decision: inside or outside of the interval

same interval. The undefined region between two stable sequences of episodes would correspond to the overlap of contiguous RIs.

In general, Rough Intervals may be represented as two superposed intervals with straight lines in the boundary regions. So results a trapezoid, with the first interval, i.e. the UAI, as base. The second interval delimits the "lower approximation", the region where the concept certainty is total. The terms "upper" and "lower" in Rough Intervals (and rough sets) are not used regarding their graphic representation. This trapezoid representation of RIs, however, is very similar to the one frequently used for fuzzy membership functions. An UAI in a Rough Interval would correspond to the "base" of the Fuzzy Interval, whereas the LAI corresponds to its "top" or "core". A fourth approach for the determination of Rough Intervals takes advantage of these similarities between fuzzy membership functions and Rough Intervals.

(d) *Rough Intervals definition based on fuzzy set identification*. Tools and methods, such as clustering and neuro-fuzzy approaches, have been developed for the identification of sub-optimal variable space partitions in fuzzy set applications, using sampled process data. Most of these concepts can be adapted for the determination of RIs as well. Adaptation is required because the identification process must fulfill the particular features of RIs: First, fuzzy sets and membership functions are identified. Second, the membership functions of the fuzzy sets resulting from this first step are converted into trapezoid functions. Third, these trapezoid membership functions are transformed into Rough Intervals, which assures that the resulting trapezoid covers the entire variable space and complies with characteristics of the RI overlapping [17] such as complementarity.

Nonetheless, the described similarities of fuzzy sets and RI must not lead to confusion between both techniques. Table 3 summarizes some important characteristics and properties of Rough Intervals and compares them with that of crisp intervals and the equivalent application of the membership of continuous variables to fuzzy subsets for their qualitative representation.

5. Case study: Situation-based qualitative modeling with Rough Intervals

The previous sections made a general description of Rough Intervals and their application to the qualitative representation of continuous variables, using vague concepts such as "warm", "far" or "low". Now it is described the actual integration of RIs in the interval-based qualitative modeling approach SQMA. This will support the evaluation of the effect of using RIs, instead of crisp intervals, for the qualitative modeling of dynamic systems.

5.1. Enhancing SQMA with Rough Intervals

The Rough Interval concept is compatible with the interval-based information modeling and representation scheme used in SQMA. By modeling a component variable using RIs, i.e. declaring the RIs' Upper Approximation Intervals, characterizes the complete variable space. SQMA processes UAIs, exactly as if they were crisp intervals, in order to produce component and system situation tables, as well as component and system transition matrices. The resulting model differs physically from ordinary SQMA models only in the overlaps between their declared intervals.

Interval overlapping and membership functions enhance the information contents in SQMA models. A variable defined over n Rough Intervals (which defines the size of the model) contains information distributed in n LAIs plus $(n-1)$ overlapping areas, i.e. in $(2*n-1)$ sections. Because each one of these $(2*n-1)$ sections can be independently perceived and interpreted, the resulting model delivers almost twice as much information as an SQMA model using crisp intervals, which only can identify n different qualitative values. For instance, "*Hot*" (with $\rho = 1$), "*Cold*" (with $\rho = 1$) or both qualitative values simultaneously, with different but complementary ρ values, correspond to three different states. The third case could be described as "*moderately Hot*", "*between Hot and Warm*" or "*almost Hot*", depending on the relationship between the membership values and given linguistic conventions. Vague linguistic expressions can be used and generated from the model, which are closer to the human way of description.

Another effect is the superposition of system situations in the model. As a result of the interval overlapping, multiple situations with different, but complementary, membership values (ρ) appear, which describe transient system conditions that otherwise would remain invisible in the qualitative model [25]. Eq. (6) shows the calculation of the global rough membership of a particular system situation (ρ_{SIT}) for a given transient system condition. ρ_{SIT} is computed as the multiplication of the rough membership of each process value (x_i) in the described system condition to its corresponding RI ($\rho(x_{\text{i}})$, compare Fig. 6, top). Based on the probabilistic interpretation of the rough membership, ρ_{SIT} would correspond to the compound probability of all the qualitative values that define the situation, taking place simultaneously.

$$\rho_{\text{SIT}} = \prod_i \rho(x_i). \tag{6}$$

The following example illustrates the improvement in the descriptive power of SQMA models through the identification of multiple situations for each process state. Fig. 9 shows two possible SQMA descriptions of a given process state. A single situation (top) results of using an SQMA model, where only crisp intervals are defined. At most, a plant operator would interpret this information as: "*The system is in a normal situation. The tank is filled with warm liquid and has nominal inflow and outflow*".

The second table (Fig. 9, bottom) corresponds to an SQMA model with Rough Intervals. The membership values for each situation are calculated using (6) for the following process condition:

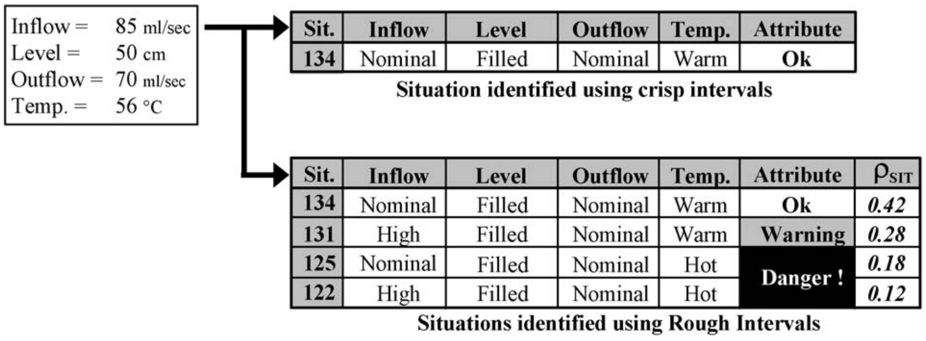
Inflow = 85 ml/sec → Nominal ($\rho = 0.6$); High ($\rho = 0.4$).

Level = 50 cm → Filled ($\rho = 1$).

Outflow = 70 ml/sec → Nominal ($\rho = 1$).

Temperature = 56 °C → Warm ($\rho = 0.7$); Hot ($\rho = 0.3$).

So results for Sit 134, for example, a membership value $\rho_{\text{SIT}} = 0.6 * 1.0 * 1.0 * 0.7 = 0.42$. Now the same process state could be described, based on the several identified situations, as: "*The liquid filling the tank is moderately hot* (between "Warm" and "Hot"). *The tank inflow is a little high* (between "High" and "Nominal") *while the outflow is nominal; the inflow is in any case greater than the outflow, therefore, the level is increasing. With that, the system*



Inflow =	85 ml/sec
Level =	50 cm
Outflow =	70 ml/sec
Temp. =	56 °C

Sit.	Inflow	Level	Outflow	Temp.	Attribute
134	Nominal	Filled	Nominal	Warm	**Ok**

Situation identified using crisp intervals

Sit.	Inflow	Level	Outflow	Temp.	Attribute	ρ_{SIT}
134	Nominal	Filled	Nominal	Warm	**Ok**	***0.42***
131	High	Filled	Nominal	Warm	**Warning**	***0.28***
125	Nominal	Filled	Nominal	Hot	**Danger !**	***0.18***
122	High	Filled	Nominal	Hot		***0.12***

Situations identified using Rough Intervals

Fig. 9. SQMA description of a process state with and without Rough Intervals.

transits towards a dangerous state". The descriptive power of the SQMA model using RIs is visibly superior to that using crisp intervals, even though the number of situations in both models (and consequently their sizes) is the same.

The above-described property of qualitative models with RI also implies that it is possible to reduce the size of the model by keeping the original resolution. Each overlapping area can be precisely identified and handled in the SQMA model with RIs, which allows their interpretation as independent intervals. In consequence, the number of intervals defined in a variable may be reduced, following the considerations presented at the beginning of this section, in a relation $0.5 * (n + 1) : n$ without losing information in the model. That is, using RIs to describe a given process variable can reduce the initial number of situations in a SQMA model to e.g. 2/3, 3/5 or 5/9 of the amount of situations it would have had, if the same variable would have been described with 3, 5 or 9 crisp intervals instead.

This is an important size reduction for SQMA models, whose situation tables grow geometrically with the amount of intervals defined on the system's variables. For the same case described in the previous paragraph, the initial SQMA model space (before validating against the situation rules) can be compressed to 4/9, 9/25 or 25/81 of its original size. The final ratio of model size reduction is a geometric function of how many variables are described with R and how many intervals are defined on them. In general, assuming that the n crisp intervals in m process variables (for simplification, assume also that these process variables are equally partitioned) are replace with $(n + 1)/2$ RIs, it can be verified that:

$$\lim_{n\to\infty} \{\text{new size}\} = \frac{1}{2^m}\{\text{original size}\}. \tag{7}$$

An analogous analysis can be made for the resolution increase by keeping the number of intervals, i.e. when a RI substitutes each crisp interval in a process variable. The model resolution can growth up to 2^m (when $n \to \infty$) the original one, if m variables are expressed using RI. These two competing tendencies (size reduction and resolution enhancement) originate two opposing ways of considering the employment of Rough Intervals. At the end, as in most engineering solutions, the problem is reduced to a trade-off between both criteria.

The situation membership (ρ_{SIT}) provides an important criterion to order, rank, classify or filter the delivered information. However, ρ_{SIT} must not be understood only as a post-processing criterion. Using multiple situations with their respective ρ-values to represent a process state (as in Fig. 9, bottom) allow for a better simulation, monitoring and analysis of dynamic systems. The different situation membership values would continuously change in a time-varying system, in a direction, sense and magnitude, which corresponds to the system dynamics. Qualitative models based on crisp intervals (e.g. Fig. 9, top) cannot deliver this kind of information.

5.2. *Spurious situations in the SQMA model*

A consequence of using RI in SQMA is that some impossible situations may remain in the situation table after completing the model. These spurious situations are component or system situations that are reachable according to the model, but actually cannot take place in real systems. The adjective spurious denotes, in this case, situations that accredit false descriptions to the real system state. Because of the RI notation, a situation is accepted if the Upper Approximation Intervals defined in this situation satisfy all the component or system rules. Since UAIs are bigger than the equivalent crisp intervals, i.e. crisp intervals with limits in the midpoints of the overlaps, using RI results in the acceptance of situations that the original SQMA method would reject.

Analyzing the example of the system rule {T1.h = Valve.dp + T2.h} corresponding to the system Tank-Valve-Tank shown in Fig. 10, may help to the comprehension of this problem. T1.h and T2.h are the pressure heads in the output of tanks T1 and T2 respectively, while Valve.dp corresponds to the pressure loss across Valve. The combinations of the intervals defined in these variables (Fig. 10, right) are validated against this rule, which results in the situations shown in Table 4 (the last column, "Confidence" is described in the next section). No invalid interval combination resulted from the rule evaluation (compare the sequence in the situation numbers).

A variable partitioning using the equivalent crisp intervals results in the elimination of situations 4, 6, 7 and 9 (italic in Table 4). If their respective descriptions are considered, these four situations are impossible, for example situation 6 claims the existence of water flowing from a tank with low level to a filled one, while situations 4 and 9 indicate that there is no flow between two tanks with different levels. Yet these four situations are not eliminated from the system situation table, because they satisfy the rule {T1.h = Valve.dp + T2.h} when evaluated with the UAIs of the defined Rough Intervals.

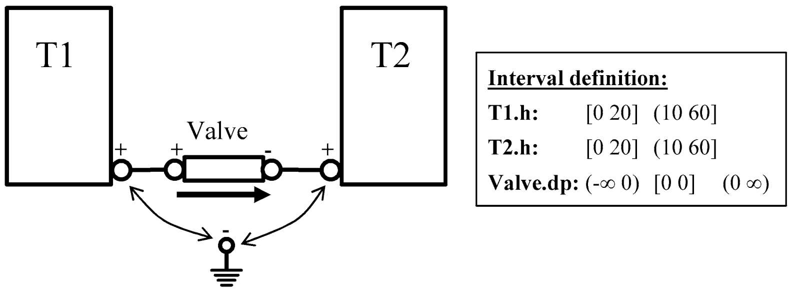


Fig. 10. The Tank-Valve-Tank system.

Table 4
Situation table for the system Tank-Valve-Tank

Sit	T1.h	Valve.dp	T2.h	Description	Confidence
1	[0 20]	$(-\infty\ 0)$	[0 20]	low ← low	1
2	[0 20]	$(-\infty\ 0)$	(10 60]	low ← filled	1
3	[0 20]	[0 0]	[0 20]	low–low	1
4	*[0 20]*	*[0 0]*	*(10 60]*	*low–filled*	*0.5*
5	[0 20]	$(0\ \infty)$	[0 20]	low → low	1
6	*[0 20]*	*$(0\ \infty)$*	*(10 60]*	*low → filled*	*0.5*
7	*(10 60]*	*$(-\infty\ 0)$*	*[0 20]*	*filled ← low*	*0.5*
8	(10 60]	$(-\infty\ 0)$	(10 60]	filled ← filled	1
9	*(10 60]*	*[0 0]*	*[0 20]*	*filled–low*	*0.5*
10	(10 60]	[0 0]	(10 60]	filled–filled	1
11	(10 60]	$(0\ \infty)$	[0 20]	filled → low	1
12	(10 60]	$(0\ \infty)$	(10 60]	filled → filled	1

Spurious situations appear because of the uncertainty inside of RI overlaps and the no determinism in the interval equality operator (see end of Section 2.2) that is also present in Rough Interval Arithmetic. The combination of these two factors causes the SQMA modeling procedure to accept impossible situations during the rule verification. Setting intervals and reference values conveniently to avoid evaluating component rules in RI overlaps can resolve this problem at component level. However, this is impossible with the system rules, because they are determined automatically during the modeling process. This demands enhancing the SQMA situation analysis at system level for the reduction of spurious situations.

5.3. Confident-based reduction of spurious situations

Using Upper Approximation Intervals to represent RIs causes spurious situations to appear in the system situation table. On the other hand, using Lower Approximation Intervals instead may cause the rejection of situations that are actually possible. A rule evaluation against LAIs should result in less than or, at most, the same number of situations resulting from evaluating the equivalent crisp intervals. A trade-off between these two extreme cases is developed based on the RI intersection (see Section 4.1), which is always in SQMA the operation closing the evaluation of a rule. Fig. 11 shows the three possible outcomes for this operation:

– The RI intersection does not exist (top-left).
– The intersection exists and delivers a conventional RI (top-right).
– The intersection exits, but delivers a minor RI (bottom).

The maximum value reachable by the corresponding membership functions is therefore: exactly zero, exactly one or a value in between (one and zero excluded) respectively. This value can express the confidence in the satisfaction of the rule evaluated with the RI intersection:

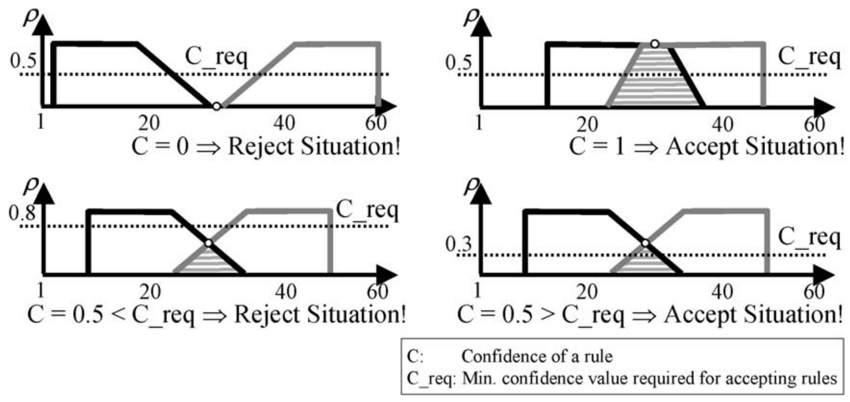


Fig. 11. Confidence-based rule evaluation with RI.

– Maximum membership value is zero ⇒ the rule is not fulfilled.
– Maximum membership value is one ⇒ the rule is fulfilled.
– Maximum membership value in between ⇒ no direct decision is possible.

The third case results in a number between zero and one, which is proportional to the confidence in the fulfillment of the rule by the corresponding situation. This value is called rule confidence (C), and is defined as *the maximum membership value in the Rough Interval intersection that determines the verification of this rule for the current situation*. Rule confidence is a value between zero and one, where $C = 0$ denotes an empty RI (no intersection), $C = 1$ a trapezoidal RI and $C \in (01)$ a minor RI.

By using rule confidence, it is possible to define thresholds to decide, during the situation validation process, whether a given rule is fulfilled or not. This enables the reduction of spurious situations in the SQMA model. Resuming the example of the Tank-Valve-Tank system, a confidence $C = 0.5$ results from the rule evaluation with situation 9 (s. Table 4, last column). This is a triangular RI with a maximum value of 0.5 (compare the intersections in Fig. 11, bottom). Establishing $C > 0.3$ as acceptance criterion for the rule validation would determine its acceptance. This minimum confidence value required for accepting a rule is called C_req. The example in Fig. 11, bottom-right corresponds in consequence to defining $C\_\text{req} = 0.3$. On the contrary, this situation must be eliminated from the model, if a confidence higher than 0.5, e.g. $C > 0.8$, is demanded (compare Fig. 11, bottom-left, $C\_\text{req} = 0.8$). The same analysis is also applicable to situations 4, 6 and 7. Establishing the rejection of any situation with a confidence under 0.5 ($C\_\text{req} = 0.5$) would result in a system situation table similar to the one resulting from using equivalent crisp intervals for the tank levels, i.e. without situations 4, 6, 7 and 9.

Eliminating situations with a small confidence value reduces the spurious situations in the situation table. This reduces the size of the SQMA model and supports a better interpretation of process states by avoiding conceptual inconsistencies in situation remarks such as "liquid flowing from a tank with low level to a filled one" in situations 6 and 7 of the Tank-Valve-Tank system. But how can an adequate value be determined for C_req? The answer to this question can be supported on the fact that the total membership of any particular value of a variable to the RIs defined for this variable must be equal to one. This rule arises from the total probability condition for complementary events and must be fulfilled even inside of the overlaps between two RIs. Hence, there exists a complementarity condition (see Section 3.2) between the membership values of a given point to the two RIs in the overlap: the probability of being part of one RI or the other must be one. So, if situations resulting with a confidence lower than 0.5 (or any C_req under 0.5) are eliminated during the situation validation, there must always exist at least one complementary situation that represents the current process state better than the one eliminated.

Another approach to explain the effect of C_req on the validation process is based on the α-cut operation. Nguyen ([14] cited by [1]) proved that by computing the value of a fuzzy mapping, a generic h-level α-cut of the resulting set only depends on the h-level α-cuts of the arguments:

$$y_h = F(x_1, x_2, \ldots, x_n)_h = F(x_{1h}, x_{2h}, \ldots, x_{nh}) \tag{8}$$

where a h-level α-cut (with $0 \leqslant h \leqslant 1$) of a fuzzy set A is the crisp set (the interval) A_h of all $x \in A$ with a fuzzy membership value greater than, or equal to, h. Consequently, any algebraic computation may be decomposed into several interval computations, each one having as arguments the α-cut intervals of the fuzzy arguments at the same level. The result is the α-cut of the original fuzzy result. The fuzzy calculus problem is then simplified to interval calculus.

Since RIs and trapezoidal fuzzy sets share the same shape and basic operations, this principle can be applied to RIs operations. Consequently, eliminating situations with a confidence $C < C\_\text{req}$ during the situation validation is the same as eliminating those situations where the C_req-level α-cut of the RI resulting from the last intersection operation is an empty interval. Considering the SQMA situation validation as function F in Eq. (8), and the RIs composing the evaluated situation as its arguments x_1 to x_n, the same spurious situations can be filtered based on the C_req-level α-cut of the RIs and interval arithmetic. For this reason, the equivalent crisp intervals introduced in last section deliver the same result as eliminating spurious situations with $C\_\text{req} = 0.5$.

Choosing $C\_\text{req} > 0.5$ (e.g. $C\_\text{req} = 0.8$) would deliver non-intersecting intervals after applying the α-cut. There are in fact gaps between these intervals. These gaps cause possible situations not being considered at all. Thus, choosing a C_req value greater than 0.5 violates the completeness of SQMA models. On the contrary, a conservative C_req between 0.5 and zero produces overlapping crisp intervals. The solution provided by these overlapping intervals is not optimal, but preserves the completeness of the SQMA model. In general, a reasonable threshold value for the rule confidence (C_req) would vary between 0.3 and 0.5.

6. Conclusions

This article introduced a new concept for the qualitative modeling of complex systems, which is based on the representation and handling of vague and uncertain information in interval-based methods. The management of vague and uncertain information enhances the relation size/precision and the descriptive power of the resulting qualitative models, improving their applicability. Principles taken from Pawlak's Rough Set Theory are employed for the representation of vague and uncertain knowledge in the original interval-based notation. The resulting Rough Interval concept enhances the resolution of the representation of dynamic systems without increasing the size of the models, or what is equivalent, allows compressing qualitative models without losing resolution.

A further pay-off of Rough Intervals is that thanks to the introduction of a rough membership function, it is now possible to identify and characterize process transitions precisely. This enables at least rough probability judgments. The uniformity assumption by the selection of the straight membership function may or may not be realistic for particular process plants, but it is still a better starting point than the probability-free estimates provided in interval-based qualitative models, particularly when no probabilistic information about the system behavior is available.

The new concept was successfully integrated in the Situation-based Qualitative Modeling and Analysis method. It allowed the integration of vague and uncertain process knowledge into SQMA models, taking advantage of the rich information hidden in the human way of perceiving and describing nature. This improved the precision and compactness of SQMA models, compared to the conventional technique based on crisp intervals. However, the introduction of vague and uncertain information causes the appearance of spurious situations in the model, which must be conveniently filtered out to assure the targeted model compactness. SQMA models using Rough Intervals have been successfully employed in online process monitoring and fault detection applications [21,27].

References

[1] A. Bonarini, G. Bontempi, A qualitative simulation approach for fuzzy dynamical models, ACM Transactions on Modeling and Computer Simulation 4 (4) (1994) 285–313.
[2] A. Grzesiak, Analysis of the applicability of SQMA to fault analysis, Diplomarbeit, IAS, Universität Stuttgart, Germany, 2003.
[3] B. Kuipers, Qualitative simulation, Artificial Intelligence 29 (1986) 289–338.
[4] D. Dubois, H. Prade, J. Lang, Possibilistic logic, in: D. Gabbay, et al. (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3, Oxford University Press, USA, 1994, pp. 439–514.
[5] D. Dvorak, B. Kuipers, Process monitoring and diagnosis: A model-based approach, IEEE Expert 6 (3) (1991) 67–74.
[6] D. Dvorak, Monitoring and diagnosis of continuous dynamic systems using semiquantitative simulation, PhD dissertation, University of Texas, Austin, TX, USA, 1992.
[7] D. Schaich, R. King, U. Keller, M. Chantler, Interval identification—a modelling and design technique for dynamic systems, in: 13th International Workshop of Qualitative Reasoning, Loch Awe Scotland, June, 1999.

[8] F. Berlin, Integration von Fuzzy-Intervallen in die qualitative Modellierungssprache SQMA, Diplomarbeit, IAS, Universität Stuttgart, Germany, 2002.
[9] F. Tay, L. Shen, Fault diagnosis based on rough set theory, Engineering Applications of Artificial Intelligence 16 (1) (2003) 39–43.
[10] G. Brajnik, M. Lines, Qualitative modeling and simulation of socio-economic phenomena, Journal of Artificial Societies and Social Simulation 1 (1) (1998).
[11] G. Coghill, S. Garrett, R. King, Learning qualitative metabolic models, in: Proceedings of the European Conference on Artificial Intelligence—ECAI'04, Valencia, Spain, 2004.
[12] G. Mayer, Development of a method concept for increasing the reliability of software-intensive systems in early development phases, Diplomarbeit, IAS, Universität Stuttgart, Stuttgart, Germany, 2004.
[13] H. Kay, B. Rinner, B. Kuipers, Semi-quantitative system identification, Artificial Intelligence 119 (1–2) (2000) 103–140.
[14] H. Nguyen, A note on the extension principle for fuzzy sets, Journal of Mathematical Analysis Applications 64 (1978) 369–380.
[15] J. de Kleer, J. Brown, A qualitative physics based on confluences, Artificial Intelligence 24 (1–3) (1984) 7–83.
[16] J. Lunze, Qualitative modelling of dynamical systems: Motivation, methods, and prospective applications, Mathematics and Computers in Simulation 46 (5–6) (1998) 465–483.
[17] J. Wille, Development of a concept for the experimental determination of Rough Intervals, Diplomarbeit, IAS, Universität Stuttgart, Germany, 2003.
[18] K. Forbus, Qualitative process theory, Artificial Intelligence 24 (1–3) (1984) 85–168.
[19] K. Forbus, Qualitative process theory: twelve years after, Artificial Intelligence 59 (1–2) (1993) 115–123.
[20] K. Forbus, P. Whalley, J. Everett, L. Ureel, M. Brokowski, J. Baher, S. Kuehne, CyclePad: An articulate virtual laboratory for engineering thermodynamics, Artificial Intelligence 114 (1–2) (1999) 297–347.
[21] L. Cachero, Online probabilistic forecasting in a three tanks system, Master Thesis, IAS, Universität Stuttgart, Germany, 2003.
[22] L. Shen, F. Tay, L. Qu, Y. Shen, Fault diagnosis using rough sets theory, Computers in Industry 43 (2000) 61–72.
[23] L. Zadeh, Fuzzy sets, Information and Control 8 (1965) 338–353.
[24] M. Rebolledo, Development of a concept for the handling of vagueness in the SQMA modeling approach, Special Project, IAS, Universität Stuttgart, Germany, 2002.
[25] M. Rebolledo, A combined rough-stochastic approach to process monitoring based on qualitative models, in: IASTED International Conference on Intelligent Systems and Control, Salzburg, Austria, 2003, pp. 69–74.
[26] M. Rebolledo, Situation-based process monitoring in complex systems considering vagueness and uncertainty, PhD Thesis, Universität Stuttgart, Germany, 2004.
[27] P. Katamaneni, Online process supervision of a three tank system, Master Thesis, IAS, Universität Stuttgart, Germany, 2003.
[28] Q. Shen, R. Leitch, Fuzzy qualitative simulation, IEEE Transactions on Systems, Man, and Cybernetics 23 (4) (1993) 1038–1060.
[29] R. Moore, Interval Analysis, Prentice-Hall, Englewood-Cliffs, NJ, 1966.
[30] S. Manz, Online fault detection and diagnosis of complex systems based on hybrid component models, in: Proceedings of the 14th International Congress on Condition Monitoring and Diagnostic Engineering Management—COMADEM 2001, Manchester, UK, 2001, pp. 865–872.
[31] U. Biegert, Computer-aided safety analysis of computer-controlled systems: A case example, in: Proceedings of the 6th IEEE International Conference on Methods and Models in Automation and Robotics (MMAR), Miedzyzdroje, Poland, 2000, pp. 777–782.
[32] U. Keller, T. Wyatt, R. Leitch, FrenSi—a fuzzy qualitative simulation method, in: Proceedings of the MISC Workshop on Application of Interval Analysis to System and Control, Girona, Spain, 1999.
[33] X. Laufenberg, P. Fröhlich, A qualitative model based approach to integrate system and hazard analysis, in: Proceedings of the IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes—SAFEPROCESS 94, Helsinki, Finland, 1994.
[34] Z. Pawlak, Rough sets: A new approach to vagueness, in: L. Zadeh, J. Kacprzyk (Eds.), Fuzzy Logic for the Management of Uncertainty, Whiley Professional Computing, USA, 1982, pp. 105–118.
[35] Z. Pawlak, Rough sets, rough relations and rough functions, ICS Research Report 24/94, Warsaw University of Technology, Warsaw, 1994.

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Artificial Intelligence 170 (2006) 686–713

Artificial Intelligence

www.elsevier.com/locate/artint

Constraint-based optimization and utility elicitation using the minimax decision criterion [☆]

Craig Boutilier [a,*], Relu Patrascu [a,1], Pascal Poupart [b,2], Dale Schuurmans [c,1]

[a] *Department of Computer Science, University of Toronto, Toronto, ON, M5S 3H5, Canada*
[b] *School of Computer Science, University of Waterloo, Waterloo, ON, Canada*
[c] *Department of Computing Science, University of Alberta, Edmonton, AB, T6G 2E8, Canada*

Received 15 August 2005; received in revised form 27 February 2006; accepted 27 February 2006

Available online 30 March 2006

Abstract

In many situations, a set of hard constraints encodes the feasible configurations of some system or product over which multiple users have distinct preferences. However, making suitable decisions requires that the preferences of a specific user for different configurations be articulated or *elicited*, something generally acknowledged to be onerous. We address two problems associated with preference elicitation: computing a best feasible solution when the user's utilities are imprecisely specified; and developing useful elicitation procedures that reduce utility uncertainty, with minimal user interaction, to a point where (approximately) optimal decisions can be made. Our main contributions are threefold. First, we propose the use of minimax regret as a suitable decision criterion for decision making in the presence of such utility function uncertainty. Second, we devise several different procedures, all relying on mixed integer linear programs, that can be used to compute minimax regret and regret-optimizing solutions effectively. In particular, our methods exploit generalized additive structure in a user's utility function to ensure tractable computation. Third, we propose various elicitation methods that can be used to refine utility uncertainty in such a way as to quickly (i.e., with as few questions as possible) reduce minimax regret. Empirical study suggests that several of these methods are quite successful in minimizing the number of user queries, while remaining computationally practical so as to admit real-time user interaction.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Decision theory; Constraint satisfaction; Optimization; Preference elicitation; Imprecise utility; Minimax regret

[☆] Parts of this article appeared in [C. Boutilier, R. Patrascu, P. Poupart, D. Schuurmans, Constraint-based optimization with the minimax decision criterion, in: Proc. of the Ninth Conference on the Principles and Practice of Constraint Programming (CP-2003), Kinsale, Ireland, 2003, pp. 168–182]; and [C. Boutilier, R. Patrascu, P. Poupart, D. Schuurmans, Regret-based utility elicitation in constraint-based decision problems, in: Proc. of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05), Edinburgh, Scotland, 2005, pp. 1293–1299].

[*] Corresponding author.
E-mail addresses: cebly@cs.toronto.edu (C. Boutilier), relu@cs.toronto.edu (R. Patrascu), ppoupart@cs.uwaterloo.ca (P. Poupart), dale@cs.ualberta.ca (D. Schuurmans).
URLs: http://www.cs.toronto.edu/~cebly(C. Boutilier), http://www.cs.toronto.edu/~relu(R. Patrascu), http://www.cs.uwaterloo.ca/~ppoupart(P. Poupart), http://www.cs.ualberta.ca/~dale(D. Schuurmans).

[1] Part of this work was completed while the author was at the School of Computer Science, University of Waterloo.

[2] Part of this work was completed while the author was at the Department of Computer Science, University of Toronto.

0004-3702/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2006.02.003

1. Introduction

The development of automated decision support software is a key focus within decision analysis [21,43,52] and artificial intelligence [9,10,16,17]. In the application of such tools, there are many situations in which the set of decisions and their effects (i.e., induced distribution over outcomes) are fixed, while the *utility functions* of different users vary widely. Developing systems that make or recommend decisions for a number of different users requires accounting for such differences in preferences. Several classes of methods have been employed by decision-support systems to "tune" their behavior appropriately, including inference or induction of user preferences based on observed behavior [28,33] or the similarity of a user's behavior to that of others (e.g., as in collaborative filtering [31]). Such behavior-based methods often require considerable data before strong conclusions can be drawn about a user's preferences (hence before good decisions can be recommended).

In many circumstances, direct *preference elicitation* may be undertaken in order to capture specific user preferences to a sufficient degree to allow an (approximately) optimal decision to be taken. In preference elicitation, the user is queried about her preferences directly. Different approaches to this problem have been proposed, including Bayesian methods that quantify uncertainty about preferences probabilistically [10,17,27], and methods that simply pose constraints on the set of possible utility functions and refine these incrementally [14,50,52].

In this paper, we will focus on direct preference elicitation for constraint-based optimization problems (COPs). COPs provide a natural framework for specifying and solving many decision problems. For example, configuration tasks [42] can naturally be viewed as consisting of a set of hard constraints (options available to a customer) and a utility function (reflecting customer preferences). While much work in the constraint-satisfaction literature has considered indirectly modeling preferences as hard constraints (with suitable relaxation techniques), more direct modeling of utility functions has come to be recognized as both natural and computationally effective. The direct or indirect modeling of multi-attribute utility functions has increasingly been incorporated into constraint optimization software.[3] Soft-constraint frameworks [8,46] that associate values with the satisfaction or violation of various constraints can also be seen as implicitly reflecting a user utility function.

However, the requirement of complete utility information demanded by a COP is often problematic. For instance, users may have neither the ability nor the patience to provide full utility information to a system. Furthermore, in many if not most instances, an optimal decision (or some approximation thereof) can be determined with a very partial specification of the user's utility function. As such, it is imperative that preference elicitation procedures be designed that focus on the relevant aspects of the problem. Preferences for unrealizable or infeasible outcomes are not (directly) relevant to decision making in a particular context; nor are precise preferences needed among outcomes that are provably dominated by others given the partial information at hand. Finally, though one could refine knowledge of a user's utility function with increased interaction, the elicitation effort needed to reach an optimal decision may not be worth the improvement in decision quality: often a near-optimal decision can be made with only a fraction of the information needed to make the optimal choice. Ultimately, it is the impact on decision quality that should guide elicitation effort [10,13,17,50].

The preference elicitation problem lies at the heart of considerable work in multiattribute utility theory [30,35,51] and the theory of consumer choice (such as conjoint analysis [29,48]), though our approach will differ considerably from classic models. Unfortunately, scant attention has been paid to preference elicitation in the constraint-satisfaction and optimization literature. Only recently has the problem of elicitation of objective functions been given due consideration [38,39].[4] While interactive preference elicitation has received little attention, optimizing with respect to a given set of preferences over configurations has been studied extensively. Branch-and-bound methods are commonly used for optimization in conjunction with constraint propagation techniques. A number of frameworks have also been proposed for modeling such systems using "soft constraints" of various types [8,46], each with an associated penalty or value that indirectly represent a user's preferences for different configurations.

In this paper, we adopt a somewhat different perspective from the usual soft constraints formalisms: we assume a user's preferences are represented directly as a utility function over possible configurations. Given a utility function and the hard constraints defining the decision space, we have a standard constraint-based optimization problem.

[3] For example, see www.ilog.com.

[4] Related, but of a decidedly different character is work on constraint acquisition [37]; more closely tied is work on learning soft constraints [41].

However, as argued earlier, it is unrealistic to expect users to express their utility functions with complete precision, nor will we generally require full utility information to make good decisions. Thus we are motivated to consider two problems, namely, the problem of "optimizing" in the presence of partial utility information, and the problem of effectively eliciting the most relevant utility information.

With respect to optimization in the presence of imprecise utility information, we suppose that a set of bounds (or more general linear constraints) on utility function parameters are provided (these constraints will arise as the result of the elicitation procedures we consider). We then consider the problem of finding a feasible solution that minimizes *maximum regret* [4,24,32,34,45] within the space of feasible utility functions. This is the solution we would regret the least should an adversary choose the user's utility function in a way that is consistent with our knowledge of the user's preferences. In a very strong sense, this minimizes the worst-case loss the user could experience as a result of our recommendation. We show that this minimax problem can be formulated and solved using a series of linear integer programs (IPs) and mixed integer programs (MIPs) in the case where utility functions have no structure.

In practice, some utility structure is necessary if we expect to solve problems of realistic size. We therefore also consider problems where utility functions can be expressed using a *generalized additive form* [3,22,23], which includes linear utility functions [30] and factored (or graphical) models like UCP-nets [12] as special cases. We derive two solution techniques for solving such structured problems: the first gives rise to a MIP with fewer variables, combined with an effective constraint generation procedure; the second encodes the entire minimax problem as a single MIP using a cost-network to formulate a compact set of constraints. The former method is shown to be especially effective.

If our knowledge of the utility parameters is loose enough, minimax regret may be unacceptably high, in which case we would like to query the user for additional information about her utility function. In this work we consider *bound queries*—a local form of *standard gamble queries* [24] that provide tighter upper or lower bounds on the utility parameters—and *comparison queries*, that present outcomes to the user for ranking. However, we focus on bound queries in our experiments. We develop several new strategies for eliciting bound information, strategies whose aim is to reduce the worst-case error (i.e., get guaranteed improvement in decision quality) with as few queries as possible. Our first strategy, *halve largest gap* (HLG), provides the best theoretical guarantees—it works by providing uniform uncertainty reduction over the entire utility space. The HLG strategy is similar to heuristically motivated polyhedral methods in conjoint analysis, used in product design and marketing [29,48]. In fact, HLG can be viewed as a special case of the method of [48] in which our polyhedra are hyper-rectangles. Our second strategy, *current solution* (CS), is more heuristic in nature, and focuses attention on *relevant* aspects of the utility function. Our empirical results show that this strategy works much better in practice than HLG, and does indeed distinguish relevant from irrelevant queries. Furthermore, its ability to discern good queries is also largely unaffected by approximation: the anytime nature of minimax computation allows time bounds to be used to ensure real-time response with little impact on the elicitation effort required. We also introduce several additional strategies which capture some of the same intuitions as HLG and CS, but with different computational procedures (and complexity). Among these, the *optimistic-pessimistic (OP)* method works almost as well as CS, having much lower computational demands, but without providing the same strong guarantees on decision quality.

The remainder of the paper is organized as follows. In Section 2 we briefly review constraint-based optimization with factored utility models. We define and motivate minimax regret for decision making with imprecisely specified utility functions in Section 3. In Section 4 we describe several methods for computing minimax regret for COPs, and evaluate one such method empirically. We also suggest several computational shortcuts well-suited to the interactive elicitation context. We define a number of elicitation strategies in Section 5 and provide empirical comparisons of these strategies, both computationally and with respect to number of queries required to reach an optimal solution or an acceptable level of regret. We conclude in Section 6 with a discussion of future research directions.

2. Constraint-based optimization and factored utility models

We begin by describing the basic problem of constraint-based optimization assuming a known utility function and also describe the use of factored utility models in COPs. This will establish background and notation for the remainder of the paper.

2.1. Optimization with known utility functions

We assume a finite set of attributes $\mathbf{X} = \{X_1, X_2, \dots, X_N\}$ with finite domains $Dom(X_i)$. An assignment $\mathbf{x} \in Dom(\mathbf{X}) = \Pi_i Dom(X_i)$ is often referred to as a *state*. For simplicity of presentation, we assume these attributes are Boolean, but nothing important depends on this (indeed, our experiments will involve primarily non-Boolean attributes). We assume a set of hard constraints $\mathcal{C}$ over these attributes. Each constraint $\mathcal{C}_\ell$, $\ell = 1, \dots, L$, is defined over a set $\mathbf{X}[\ell] \subseteq \mathbf{X}$, and thus induces a set of legal configurations of attributes in $\mathbf{X}[\ell]$. More formally, $\mathcal{C}_\ell$ can be viewed as the subset of $Dom(\mathbf{X}[\ell])$ from which all feasible configurations must be constructed. We assume that the constraints $\mathcal{C}_\ell$ are represented in some logical form and can be expressed compactly; for example, we might write

$$(X_1 \wedge X_2) \supset \neg X_3$$

to denote that all legal configurations of Boolean variables X_1, X_2, X_3 are such that X_3 must be false if X_1 and X_2 are both true. We let $Feas(\mathbf{X})$ denote the subset of *feasible states* (i.e., assignments satisfying $\mathcal{C}$). The *constraint satisfaction problem* is that of finding a feasible assignment to a specific set of constraints. While the set of states $Dom(\mathbf{X})$ is exponential in N (the number of variables), logically expressed constraints allow us to specify the feasible set $Feas(\mathbf{X})$ compactly, and makes explicit problem structure that can be exploited to (often) effectively solve CSPs. We refer to Dechter [20] for a detailed overview of models and methods for constraint satisfaction.

The *constraint graph* for a given set of constraints is the undirected graph whose nodes are attributes and whose edges connect any two attributes that occur in the same constraint. This graph has useful properties that can be used to determine the worst-case complexity of various algorithms for constraint-satisfaction and constraint-based optimization [20]. Fig. 1 illustrates a small constraint graph over four variables induced by the logical constraints shown.

Our focus here will not be on constraint satisfaction, but rather on *constraint-based optimization problems* (*COPs*).[5] Suppose we have a known non-negative *utility function* $u : Dom(\mathbf{X}) \to \mathbf{R}^+$ which ranks all states (whether feasible or not) according to their degree of desirability.[6] Our aim is to find an optimal feasible state $\mathbf{x}^*$; that is, any

$$\mathbf{x}^* \in \arg \max_{\mathbf{x} \in Feas(\mathbf{X})} u(\mathbf{x}).$$

Since choices are restricted to feasible states, we sometimes call feasible states *decisions*. Without making any assumptions regarding the nature of the utility function (e.g., with regard to structure, independence, compactness, etc.) we can formulate the COP in an "explicit" fashion as a (linear) 0–1 integer program:

$$\max_{\{I_\mathbf{x}, X_i\}} \sum_\mathbf{x} u_\mathbf{x} I_\mathbf{x} \quad \text{subject to } \mathcal{A} \text{ and } \mathcal{C}, \tag{1}$$

where we have:

- variables $I_\mathbf{x}$: for each $\mathbf{x} \in Dom(\mathbf{X})$, $I_\mathbf{x}$ is a Boolean variable indicating whether $\mathbf{x}$ is the decision made (i.e., the feasible state chosen). In other words, $I_\mathbf{x} = 1$ iff $\mathbf{x}$ is the solution to the COP.

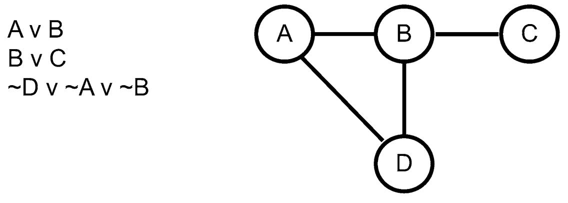


Fig. 1. An example constraint graph induced by the logical constraints shown to the left.

[5] We use the term *constraint-based optimization* (often called constraint optimization) to refer to discrete combinatorial optimization problems with explicit logical constraints over variable instantiations, as in CSPs, to distinguish these from more general *constrained optimization* problems with arbitrary (e.g., continuous) variables and general functional constraints.

[6] For ease of presentation, we assume the utility function has been normalized to be non-negative. Nothing critical depends on this, as such normalization is always possible [23].

- variables X_i: X_i is a Boolean variable corresponding to the ith attribute. In other words, variables $X_i = 1$ iff feature X_i is true in the solution to the COP.[7]
- coefficients $u_{\mathbf{x}}$: for each $\mathbf{x} \in Dom(\mathbf{X})$, constant $u_{\mathbf{x}}$ denotes the (known) utility of state $\mathbf{x}$.
- constraint set $\mathcal{A}$: for each variable $I_{\mathbf{x}}$, we impose a constraint that relates it to its corresponding variable assignment. Specifically, for each X_i: if X_i is true in $\mathbf{x}$, we constrain $I_{\mathbf{x}} \leqslant X_i$; and if X_i is false in $\mathbf{x}$, we constrain $I_{\mathbf{x}} \leqslant 1 - X_i$. We use $\mathcal{A}$ to denote this set of *state definition constraints*.
- constraint set $\mathcal{C}$: we impose each feasibility constraint $\mathcal{C}_\ell$ on the attributes $X_i \in \mathbf{X}[\ell]$. Logical constraints on the variables X_i can be written in a natural way as linear constraints [18], so we omit details.[8]

Note that this formulation ensures that, if there is a feasible solution (given the constraints $\mathcal{A}$ and $\mathcal{C}$), then exactly one $I_{\mathbf{x}}$ will be non-zero.

As an example of the interplay between the constraints in $\mathcal{A}$ and $\mathcal{C}$, consider the example in Fig. 1. The following constraints will be present in the set $\mathcal{A}$ for the state $abc\bar{d}$ (with analogous constraints for the other 15 instantiations of the four Boolean variables):

$$I_{abc\bar{d}} \leqslant A; \quad I_{abc\bar{d}} \leqslant B; \quad I_{abc\bar{d}} \leqslant C; \quad I_{abc\bar{d}} \leqslant 1 - D;$$

while the constraint set $\mathcal{C}$, capturing the three domain constraints in the figure are:

$$A + B \geqslant 1; \quad B + C \geqslant 1; \quad A + B + C \leqslant 2.$$

2.2. Factored utility models

Even with logically specified constraints, solving a COP in the manner above is not usually feasible, since the utility function is not specified concisely. As a result, the IP formulation above is not practical, since there is one $I_{\mathbf{x}}$ variable per state and the number of states is exponential in the number of attributes. With such *flat* utility functions, it is not generally possible to formulate the optimization problem concisely: indeed, if there is no (structural) relationship between the utility of different states, little can be done but to enumerate feasible states to ensure that an optimal solution is found. In contrast, if some structure is imposed on the utility function, say, in the form of a *factored* (or graphical) model, we are then often able to reduce the number of variables to be linear in the number of parameters of the factored model.

We consider here *generalized additive independence* (*GAI*) models [3,22], a natural, but flexible and fully expressive generalization of additive (or linear) utility models.[9] GAI is appealing because of its generality and expressiveness; for instance, it encompasses both linear models [30] and UCP-nets [12] as special cases,[10] but can capture any utility function. The advantage of structured utility models, and GAI specifically, is that the constraint-based optimization can be formulated and (typically) solved without explicit enumeration of states. While we focus on the GAI model, other compact structured utility models can be exploited in similar fashion (e.g., see the many such models proposed by Keeney and Raiffa [30]).

The GAI model assumes that our utility function can be written as the sum of K local utility functions, or *factors*, over small sets of attributes:

$$u(\mathbf{x}) = \sum_{k \leqslant K} f_k(\mathbf{x}[k]). \tag{2}$$

Here each function f_k depends only on a local family of attributes $\mathbf{X}[k] \subset \mathbf{X}$. We denote by $\mathbf{x}[k]$ the restriction of state $\mathbf{x}$ to the attributes in $\mathbf{X}[k]$. Again we assume that each function f_k is non-negative. For example, if $\mathbf{X} = \{A, B, C, D\}$ we might decompose a utility function as follows:

$$u(ABCD) = f_1(A, B) + f_2(B, C).$$

[7] If features are non-Boolean, then X_i simply needs to be a general integer variable ranging over $Dom(X_i)$.

[8] Again, generalizing the form of these constraints to generalized logical constraints involving non-Boolean attributes is straightforward.

[9] Fishburn [22] introduced the model, using the term *interdependent value additivity*; Bacchus and Grove [3] dubbed the same concept GAI, which seems to be more commonly used in the AI literature currently.

[10] For example, UCP-nets encompass GAI with some additional restrictions. Hence any algorithm for GAI models automatically applies to UCP-nets, though one might be able to exploit the structure of UCP-nets for *additional* computational gain.

AB	BC
AB 15	BC 13
$A\bar{B}$ 6	$B\bar{C}$ 15
$\bar{A}B$ 10	$\bar{B}C$ 5
$\bar{A}\bar{B}$ 2	$\bar{B}\bar{C}$ 3

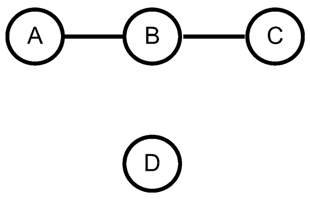


Fig. 2. An example utility graph induced by the two utility factors shown to the left.

Fig. 2 illustrates a possible instantiation of the two factors. Note that a user with such a utility function exhibits no preference for the different values of variable D.

The conditions under which a GAI model provides an accurate representation of a utility function were defined by Fishburn [22,23]. We provide the intuitions here. Let P be some probability distribution over $Dom(\mathbf{X})$, interpreted as a gamble or lottery presented to the decision maker. For instance, P may correspond to the distribution over outcomes induced by some decision she could take. Fishburn showed that the decision maker's utility function u could be written in GAI form over factors f_k iff she is indifferent between any pair of gambles P and Q over $Dom(\mathbf{X})$ whose marginals over each subset of variables $\mathbf{X}[k]$ ($k \leqslant K$) are identical. Note that GAI models are completely general (since any utility function can be represented trivially using a single factor consisting of all variables). Furthermore, linear models are a special case in which we have a singleton factor for each variable.

We refer to a pair $\langle \mathcal{C}, \{f_k\colon k \leqslant K\}\rangle$ as a *structured COP*, where $\mathcal{C}$ is a set of feasibility constraints and $\{f_k\colon k \leqslant K\}$ is a set of utility factors. An IP similar to Eq. (1) can be used to solve for the optimal decision in the case of a GAI model:

$$\max_{\{I_{\mathbf{x}[k]}, X_i\}} \sum_{k \leqslant K} \sum_{\mathbf{x}[k] \in Dom(\mathbf{X}[k])} u_{\mathbf{x}[k]} I_{\mathbf{x}[k]} \quad \text{subject to } \mathcal{A} \text{ and } \mathcal{C}. \tag{3}$$

Instead of one variable $I_{\mathbf{x}}$ per state, we now have a set of *local state variables* $I_{\mathbf{x}[k]}$ for each family k and each instantiation $\mathbf{x}[k] \in Dom(\mathbf{X}[k])$ of the variables in that family. Similarly, we have one associated constant coefficient $u_{\mathbf{x}[k]}$ denoting $f_k(\mathbf{x}[k])$. $I_{\mathbf{x}[k]}$ is true iff the assignment to $\mathbf{X}[k]$ is $\mathbf{x}[k]$. Each $I_{\mathbf{x}[k]}$ is related logically to the attributes $X \in \mathbf{X}[k]$ by (local) state definition constraints $\mathcal{A}$ as before, and the usual feasibility constraints $\mathcal{C}$ are also imposed as above.

Notice that the number of variables and constraints in this IP (excluding the exogenous feasibility constraints $\mathcal{C}$) is now *linear* in the number of parameters of the underlying utility model, which itself is linear in the number of attributes $|\mathbf{X}|$ if we assume that the size of each utility factor f_k is bounded. This compares favorably with the exponential size of the IP for unfactored utility models in Section 2.1.

This formulation of COPs is more or less the same as several other models of COPs found in the constraint satisfaction literature. For example, the notion of a *cost network* is often used to represent objective functions by assuming a similar form of factored decomposition of the objective function [20]. Specifically, let the *utility graph* be defined in the same fashion as the constraint graph, but with edges connecting attributes that occur in the same utility factor. This utility graph can be viewed as a cost network. Fig. 2 illustrates the utility graph over variables $\{A, B, C, D\}$ induced by the utility function decomposition described above (corresponding to the two utility factors AB and BC shown at the left of the figure).

Similarly, certain soft constraint formalisms can be used to represent COPs [8]. Every constraint is assigned a cost corresponding to a penalty incurred if that constraint is not satisfied. Hard constraints in $\mathcal{C}$ are assigned an infinite cost, and constraints corresponding to configurations of local utility factors are given a finite cost corresponding to their (negative) utility. The goal is to find a minimum cost state, with the infinite costs associated with hard constraints ensuring a preference for any feasible solution over any infeasible solution.

The IP formulation of structured COPs in Eq. (3) can be solved using off-the-shelf solution software for general IPs. Various special purposes algorithms that rely on the use of dynamic programming (e.g., the *variable elimination* algorithm) or constraint satisfaction techniques can also be used to solve these problems; we refer to Dechter [20] for a discussion of various algorithmic approaches to COPs developed in the constraint satisfaction literature. While many of these could be adapted to address the problems discussed in the paper, we will focus on IP formulations and their direct solution using standard IP software.

We remark here on the use of utility functions rather than (qualitative) preference rankings in this work. In a deterministic setting, such as in the constraint-based framework adopted here, one does not need utility functions to make decisions. Instead an ordinal preference ranking will suffice [30] since, without uncertainty, strength of preference information is not needed to assess tradeoffs: complete ordinal preference information will dictate which feasible outcome is preferred. However, in large multiattribute domains, even with factored preference models, full elicitation will often be time-consuming and unnecessary. As discussed, our aim is to make decisions with incomplete preference information. If we make a decision that is potentially suboptimal, we cannot be sure of its quality unless we have some information about the strength of preference of this decision (at least relative to the optimal decision). Simply knowing that one decision "may be preferred" to another does not give us enough information to know whether additional preference information should be elicited if we are content with making good, rather than, optimal decisions. It is impossible to say *how good* a non-optimal decision actually is without some quantitative utility information.

3. Minimax regret

In many circumstances, we are faced with a COP in which a user may not have fully articulated her utility function over configurations of attributes. This arises naturally when distinct configurations must be produced for users with different preferences, with some form of utility elicitation used to extract only a partial expression of these preferences. It will frequently be the case that we must make a decision before a complete utility function can be specified. For instance, users may have neither the ability nor the patience to provide full utility information to a system before requiring a decision to be recommended. Furthermore, in many if not most instances, an optimal decision (or some approximation thereof) can be determined with a very partial specification of the user's utility function. This will become evident in our preference elicitation framework and the models we consider in this paper.

If the utility function is unknown, then we have a slightly different problem than the standard COP. We cannot maximize utility (or expected utility in stochastic decision problems) because the utility function is incompletely specified. However, we will often have constraints on the utility function, either initial information about plausible utility values, or more refined constraints based on the results of utility elicitation with a specific user. For example, these might be bounds on the parameters of the utility model, or possibly more general constraints (as we discuss below). Given such a set of possible utility functions (namely those consistent with these constraints), we must adopt some suitable decision criterion for optimization, knowing only that the user's utility function lies within this set.

In the paper we propose the use of *minimax regret* [12,24,32,43,45,50] as a natural decision criterion for imprecise COPs. We first define the notion of minimax regret and then provide some motivation for its use as a suitable criterion in this setting.

3.1. Minimax regret in COPs

Minimax regret is a very natural criterion for decision making with imprecise or incompletely specified utility functions. It requires that one adopt the (feasible) assignment $\mathbf{x}$ with minimum *max regret*, where max regret is the largest amount by which one could *regret* making decision $\mathbf{x}$ (while allowing the utility function to vary within the bounds). It has been suggested as an alternative to classical expected utility theory. Specifically, it has been proposed as a means for accounting for uncertainty over possible states of nature (or the outcomes of decisions) [4,32,34,44], both when probabilistic information is unavailable, and as a descriptive theory of human decision making that explains certain common violations of von Neumann–Morgenstern [49] axioms. However, only recently has it been considered as a means for dealing with the utility function uncertainty a decision system may possess regarding a user's preferences [12,14,43,50]. It is this formulation we present here.

Formally, let $\mathbf{U}$ denote the set of feasible utility functions, reflecting our partial knowledge of the user's preferences. The set $\mathbf{U}$ may be finite; but more commonly it will be continuous, defined by bounds (or constraints) on (sets of) utility values $u(\mathbf{x})$ for various states. We refer to a pair $\langle \mathcal{C}, \mathbf{U}\rangle$ as an *imprecise COP*, where $\mathcal{C}$ is a set of feasibility constraints and $\mathbf{U}$ is the set of feasible utility functions. In the case where $\mathbf{U}$ is defined by a finite set of linear constraints $\mathcal{U}$, we sometimes abuse terminology by speaking of $\langle \mathcal{C}, \mathcal{U}\rangle$ as an imprecise COP.

We define minimax regret in stages:

Definition 1. The *pairwise regret* of state $\mathbf{x}$ with respect to state $\mathbf{x}'$ over feasible utility set $\mathbf{U}$ is defined as

$$R(\mathbf{x}, \mathbf{x}', \mathbf{U}) = \max_{u \in \mathbf{U}} u(\mathbf{x}') - u(\mathbf{x}). \tag{4}$$

Intuitively, $\mathbf{U}$ represents the knowledge a decision support system has of the user's preferences. $R(\mathbf{x}, \mathbf{x}', \mathbf{U})$ is the most the system could regret choosing $\mathbf{x}$ instead of $\mathbf{x}'$, if an adversary could impose any utility function in $\mathbf{U}$ on the user. In other words, if the system were forced to choose between $\mathbf{x}$ and $\mathbf{x}'$, then this corresponds to the worst-case loss associated with choosing $\mathbf{x}$ rather than $\mathbf{x}'$ with respect to possible realizations of $u \in \mathbf{U}$.

Definition 2. The *maximum regret* of decision $\mathbf{x}$ is:

$$\begin{aligned} MR(\mathbf{x}, \mathbf{U}) &= \max_{\mathbf{x}' \in Feas(\mathbf{X})} R(\mathbf{x}, \mathbf{x}', \mathbf{U}) && (5) \\ &= \max_{u \in \mathbf{U}} \max_{\mathbf{x}' \in Feas(\mathbf{X})} u(\mathbf{x}') - u(\mathbf{x}). && (6) \end{aligned}$$

Since the goal of our decision support system is to make the optimal choice with respect to the user's true utility function, $MR(\mathbf{x}, \mathbf{U})$ is the most the system could regret choosing $\mathbf{x}$; that is, it is the worst-case loss associated with choosing $\mathbf{x}$ in the presence of an adversary who could choose the user's utility function to maximize the difference between $\mathbf{x}$ and acting optimally.[11]

Definition 3. The *minimax regret* of feasible utility set $\mathbf{U}$ is:

$$\begin{aligned} MMR(\mathbf{U}) &= \min_{\mathbf{x} \in Feas(\mathbf{X})} MR(\mathbf{x}, \mathbf{U}) && (7) \\ &= \min_{\mathbf{x} \in Feas(\mathbf{X})} \max_{u \in \mathbf{U}} \max_{\mathbf{x}' \in Feas(\mathbf{X})} u(\mathbf{x}') - u(\mathbf{x}). && (8) \end{aligned}$$

A *minimax-optimal* decision $\mathbf{x}^*$ is any decision that minimizes max regret:

$$\mathbf{x}^* \in \arg \min_{\mathbf{x} \in Feas(\mathbf{X})} MR(\mathbf{x}, \mathbf{U}).$$

If the only information we have about a user's utility function is that it lies in the set $\mathbf{U}$, then a decision $\mathbf{x}^*$ that minimizes max regret is very intuitive as we elaborate below. Specifically, without distributional information over the set of possible utility functions, choosing (or recommending) a minimax-optimal decision $\mathbf{x}^*$ minimizes the worst-case loss with respect to possible realizations of the utility function $u \in \mathbf{U}$. Our goal of course is to formulate the minimax regret optimization (Eq. (8)) in a computationally tractable way. We address this in Section 4.

To illustrate minimax regret, consider the example illustrated in Figs. 1 and 2; but suppose that the precise utility values associated with these factors are unknown, and instead replaced with the upper and lower bounds shown in Fig. 3. The problem admits five feasible states, and the pairwise max regret $R(\mathbf{x}, \mathbf{x}', \mathbf{U})$ for each pair of states (with $\mathbf{x}$ along columns and $\mathbf{x}'$ along the rows) is shown in Table 1. The max regret of each feasible state is shown in the final column, from which we see that state $AB\overline{C}$ is the minimax optimal decision.

AB	BC
AB [9,17]	BC [2,4]
$A\bar{B}$ [5,8]	$B\bar{C}$ [0,10]
$\bar{A}B$ [10,11]	$\bar{B}C$ [4,8]
$\bar{A}\bar{B}$ [0,3]	$\bar{B}\bar{C}$ [0,5]

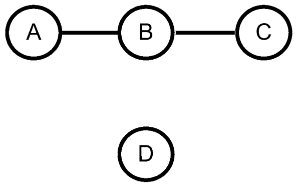


Fig. 3. Imprecisely specified utility factors (with upper and lower bounds provided for each parameter.

[11] Note that the $\mathbf{x}'$ chosen by the adversary for a specific u will always be the optimal decision under u: any other choice would give the adversary lesser utility and thus reduce regret.

Table 1

	ABC	$AB\overline{C}$	$A\overline{B}C$	$\overline{A}BC$	$\overline{A}B\overline{C}$	Max regret
ABC	0	8	5	2	10	10
$AB\overline{C}$	4	0	7	6	2	7
$A\overline{B}C$	12	18	0	6	12	18
$\overline{A}BC$	7	15	6	4	0	15
$\overline{A}B\overline{C}$	15	7	6	4	0	15

3.2. Motivation for minimax regret

As mentioned, minimax regret has been widely studied (and critiqued) as a means for decision making under uncertainty. It has been studied primarily as a means for making decisions when a decision maker is unwilling or unable to quantify her uncertainty over possible states of nature, or as a means of explaining violations of classical axioms of expected utility theory. Savage [44] introduced the notion though could not provide a "categorical" defense of its use. The use of regret, including the incorporation of feelings of regret (and its opposite, "rejoicing") into expected utility theory, has been proposed as a means of accounting for the manner in which people violate the axioms of expected utility theory in practice [4,34]. Difficulties with the use of this decision criterion include the fact that the minimax regret criterion does not satisfy the principle of irrelevant alternatives [24,44], and regret theory fails to satisfy a reasonable notion of stochastic dominance [40]. It can also be argued that, from a Bayesian perspective, a decision maker might as well construct their own subjective assessment of possible states of nature. For this reason, minimax regret is often viewed as too cautious a criterion.

The perspective we adopt here is somewhat different. We do not adopt minimax regret as a means of accounting for a decision maker's personal feelings of regret. Rather we define regret with respect to our *decision system's uncertainty* with respect to the user's true utility function. It seems incontrovertible that there will generally be some utility function uncertainty on the part of any system designed to make decisions on behalf of users. The only issue is how this uncertainty is represented and reasoned with.

Naturally, Bayesian methods may be entirely appropriate in some circumstances: if one can quantify uncertainty over possible utility functions probabilistically, then one can take expectations over this density to determine the *expected* expected utility of a decision [10,11,17]. However, there are many circumstances in which the Bayesian approach is inappropriate. First, it can often be very difficult to develop reasonable priors over the utility functions of a wide class of users. Furthermore, representing priors over such complex entities as utility functions is fraught with difficulty and inevitably requires computational approximations in inference (this is made abundantly clear in recent Bayesian approaches to preference elicitation [10,17]). As a consequence, the value of such "normatively correct" models is undermined in practice. Often bounds on the parameters of utility functions are generally much easier to come by, much easier to maintain, and lend themselves to much more computationally manageable algorithms as we will see in this paper. In addition, max regret provides an upper bound on the expected loss when probabilistic information is known. As we will see below, minimax regret is a very effective driver of preference elicitation, so concerns about its pessimistic nature are largely unfounded. We will see that with relatively few queries, max regret can be reduced to very low levels (often to the point of offering provably optimal solutions). Though we don't pursue this approach here, when probabilistic information is available, it can be combined rather effectively with minimax computation [50].

Finally, it is worth noting that making a recommendation whose utility is near optimal *in expectation*, as is the case in Bayesian models of preference elicitation, is often of cold comfort to a user when the decision made is *actually* very far from optimal. While minimax regret provides a worst-case bound on the loss in decision quality arising from utility function uncertainty (even in cases where distributional information is available), Bayesian methods cannot typically provide such a bound. In some contexts, such as procurement, this has been reported as a source of contention with clients using automated preference elicitation [14]. The argument is often made that users do not want to "leave money on the table" (even if the odds are low); if any money is left on the table, they want guarantees (as opposed to probabilistic assurances) that the amount they could have saved through further preference elicitation is limited.

Recently, a considerable amount of work in *robust optimization* has adopted the minimax regret decision criterion [1,2,32].[12] This work addresses combinatorial optimization problems with data uncertainty (e.g., shortest path problems or facility location with uncertain parameters) and find "robust deviation decisions" that minimize max regret. While the perspective in this work is somewhat different than that adopted in ours, the models and methods are quite similar. Our formulation is specific to the constraint-based optimization setting, but more importantly we focus on how minimax regret can be used to drive the process of elicitation, a problem not addressed systematically in the robust optimization literature. Our techniques for preference elicitation could in fact be adapted for problems in robust optimization as a means to drive the reduction in data uncertainty.

4. Computing minimax regret in COPs

We address the computational problem of computing minimax optimal decisions in several stages. We initially assume upper and lower bounds on utility parameters and discuss procedures for minimax computation for this form of uncertainty. We begin in Section 4.1 by formulating minimax regret in flat (unfactored) utility models to develop intuitions used in the factored case. In Section 4.2 we discuss the computation of maximum regret in factored utility models, and propose two procedures for dealing with minimax regret. We evaluate one of these methods empirically in Section 4.3. Finally, in Section 4.4 we propose a generalization for the minimax problem in the case where the feasible utility set is defined by arbitrary linear constraints on parameters of the utility model.

4.1. Minimax regret with flat utility models

If we make no assumptions about the structure of the utility function, nor any assumptions about the nature of the feasible utility set $\mathbf{U}$, the optimization problem defined in Eq. (8) can be posed directly as a semi-infinite, quadratic, mixed-integer program (MIP):

$$\min_{\{M_{\mathbf{x}}, I_{\mathbf{x}}, X_i\}} \sum_{\mathbf{x}} M_{\mathbf{x}} I_{\mathbf{x}} \quad \text{subject to} \begin{cases} M_{\mathbf{x}} \geqslant u_{\mathbf{x}'} - u_{\mathbf{x}} & \forall \mathbf{x} \in \mathbf{X},\ \forall \mathbf{x}' \in \mathit{Feas}(\mathbf{X}),\ \forall u \in \mathbf{U}, \\ \mathcal{A} \text{ and } \mathcal{C}, \end{cases}$$

where we have:

- variables $M_{\mathbf{x}}$: for each $\mathbf{x}$, $M_{\mathbf{x}}$ is a real-valued variable denoting the max regret when decision $\mathbf{x}$ is made (i.e., when that state is chosen).
- variables $I_{\mathbf{x}}$: for each $\mathbf{x}$, $I_{\mathbf{x}}$ is a Boolean variable indicating whether $\mathbf{x}$ is the decision made.
- coefficients $u_{\mathbf{x}}$: for each $u \in \mathbf{U}$ and each state $\mathbf{x}$, $u_{\mathbf{x}}$ denotes the utility of $\mathbf{x}$ given utility function u.
- state definition constraints $\mathcal{A}$ and feasibility constraints $\mathcal{C}$ (defined as above).

Direct solution of this MIP is problematic, specifically because of the set of constraints on the $M_{\mathbf{x}}$ variables. First, if $\mathbf{U}$ is continuous (the typical case we consider here), then the set of constraints of the form $M_{\mathbf{x}} \geqslant u_{\mathbf{x}'} - u_{\mathbf{x}}$ is also continuous, since it requires that we "enumerate" all utility values $u_{\mathbf{x}}$ and $u_{\mathbf{x}'}$ corresponding to any utility function $u \in \mathbf{U}$. Furthermore, it is critical that we restrict our attention to those constraints associated with $\mathbf{x}'$ in the *feasible* set (i.e., those states satisfying $\mathcal{C}$). Fortunately, we can often tackle this seemingly complex optimization in much simpler stages if we make some very natural assumptions regarding the nature of the feasible utility space and utility function structure.

We begin by considering the case where our imprecise knowledge regarding all utility parameters $u_{\mathbf{x}}$ is independent and represented by simple upper and lower bounds. For example, asking standard gamble queries, as discussed further in Section 5.1, provides precisely such bounds on utility values [24]. Specifically, we assume an upper bound $u_{\mathbf{x}}\!\uparrow$ and a lower bound $u_{\mathbf{x}}\!\downarrow$ on each $u_{\mathbf{x}}$, thus defining the feasible utility set $\mathbf{U}$ to be a hyper-rectangle. This assumption allows us to compute the minimax regret in three simpler stages, which we now describe.[13]

[12] The term "robust optimization" has a number of different interpretations (see for example the work of Ben-Tal and Nemirovski [5]) of which minimax regret is one of the less common.

[13] This transformation essentially reduces the *semi-infinite quadratic* MIP to a *finite linear* IP.

First, we note that the pairwise regret for an ordered pair of states can be easily computed since each $u_{\mathbf{x}}$ is bounded by an upper and lower bound:

$$R(\mathbf{x},\mathbf{x}',\mathbf{U}) = \begin{cases} u'_{\mathbf{x}}\uparrow - u_{\mathbf{x}}\downarrow & \text{when } \mathbf{x}\neq\mathbf{x}', \\ 0 & \text{when } \mathbf{x}=\mathbf{x}'. \end{cases} \tag{9}$$

Let $r_{\mathbf{x},\mathbf{x}'}$ denote this pairwise regret value for each $\mathbf{x}$, $\mathbf{x}'$, which we now assume has been pre-computed for all pairs.

Second, using Eq. (5), we can also compute the max regret $MR(\mathbf{x},\mathbf{U})$ of any state $\mathbf{x}$ based on the pre-computed pairwise regret values $r_{\mathbf{x},\mathbf{x}'}$. Specifically, we can enumerate all feasible states $\mathbf{x}'$, retaining the largest (pre-computed) pairwise regret:

$$MR(\mathbf{x},\mathbf{U}) = \max_{\mathbf{x}'\in Feas(\mathbf{X})} r_{\mathbf{x},\mathbf{x}'}. \tag{10}$$

Alternatively, we can search through feasible states "implicitly" with the following IP:

$$MR(\mathbf{x},\mathbf{U}) = \max_{\{I_{\mathbf{x}'},X'_i\}} \sum_{\mathbf{x}'} r_{\mathbf{x},\mathbf{x}'} I_{\mathbf{x}'} \quad \text{subject to } \mathcal{A} \text{ and } \mathcal{C}. \tag{11}$$

Third, let $m_{\mathbf{x}}$ denote the value of $MR(\mathbf{x},\mathbf{U})$. With the max regret terms $m_{\mathbf{x}} = MR(\mathbf{x},\mathbf{U})$ in hand, we can compute the minimax regret $MMR(\mathbf{U})$ readily. We simply enumerate all feasible states $\mathbf{x}$ and retain the one with the smallest (precomputed) max regret value $m_{\mathbf{x}}$:

$$MMR(\mathbf{U}) = \min_{\mathbf{x}\in Feas(\mathbf{X})} m_{\mathbf{x}}. \tag{12}$$

Again, this enumeration may be done implicitly using the following IP:

$$MMR(\mathbf{U}) = \min_{\{I_{\mathbf{x}},X_i\}} \sum_{\mathbf{x}} m_{\mathbf{x}} I_{\mathbf{x}} \quad \text{subject to } \mathcal{A} \text{ and } \mathcal{C}. \tag{13}$$

In this flat model case, the two IPs above are not necessarily practical, since they require one indicator variable per state. However, this reformulation does show that the original quadratic MIP with a continuous set of constraints can be solved in stages using finite, linear IPs. More importantly, these intuitions will next be applied to develop an analogous procedure for factored utility models.

Note that the strategy above hinges on the fact that we have independently determined upper and lower bounds on the utility value of each state. If utility values are correlated by more complicated constraints, this strategy will not generally work. In particular, *comparison queries* in which a user is asked which of two states is preferred induce linear constraints on the entire set of utility parameters, thus preventing exploitation of independent upper and lower bounds. We discuss formulations that allow us to deal with such feasible utility sets in Section 4.4. However, we initially focus on the case of independent bounds.

4.2. Minimax regret with factored utility models

The optimization for flat models is interesting in that it allows us to get a good sense of how minimax regret works in a constraint-satisfaction setting. From a practical perspective, however, the above model has little to commend it. By solving IPs with one $I_{\mathbf{x}}$ variable per state, we have lost all of the advantage of using a compact and natural constraint-based approach to problem modeling. As we have seen when optimizing with known utility functions, if there is no *a priori* structure in the utility function, there is very little one can do but enumerate (feasible) states. On the other hand, when the problem structure allows for modeling via factored utility functions the optimization becomes more practical. We now show how much of this practicality remains when our goal is to compute the minimax-optimal state given uncertainty in a *factored* utility function represented as a graphical model.

Assume a set of factors f_k, $k \leqslant K$, defined over local families $\mathbf{X}[k]$, as described in Section 2.2. The parameters of this utility function are denoted by $u_{\mathbf{x}[k]} = f_k(\mathbf{x}[k])$, where $\mathbf{x}[k]$ ranges over $Dom(\mathbf{X}[k])$. We use the term *imprecise structured COP* to describe an imprecise COP $\langle\mathcal{C},\mathcal{U}\rangle$ where the feasible utility set $\mathbf{U}$ is defined by a set of constraints $\mathcal{U}$ over the parameters $u_{\mathbf{x}[k]}$ of a factored utility model $\{f_k\colon k\leqslant K\}$.

As in the flat-model case, we assume upper and lower bounds on each of these parameters, which we denote by $u_{\mathbf{x}[k]}\uparrow$ and $u_{\mathbf{x}[k]}\downarrow$, respectively. Hence the range of each utility factor f_k for a given assignment $\mathbf{x}[k]$ corresponds to an interval. By defining $u(\mathbf{x})$ as in Eq. (2), pairwise regret, max regret and minimax regret are all defined in the same manner outlined in Section 3. We now show how to compute each of these quantities in turn by generalizing the intuitions developed for flat models.

4.2.1. Computing pairwise regret and max regret

As in the unfactored case (Section 4.1), it is straightforward to compute the pairwise regret of any pair of states $\mathbf{x}$ and $\mathbf{x}'$. For each factor f_k and pair of local assignments $\mathbf{x}[k]$, $\mathbf{x}'[k]$, we define the *local pairwise regret*:

$$r_{\mathbf{x}[k],\mathbf{x}'[k]} = \begin{cases} u_{\mathbf{x}'[k]}\!\uparrow - u_{\mathbf{x}[k]}\!\downarrow & \text{when } \mathbf{x}[k] \neq \mathbf{x}'[k], \\ 0 & \text{when } \mathbf{x}[k] = \mathbf{x}'[k]. \end{cases}$$

With factored models it is not hard to see from Eqs. (2) and (9) that $R(\mathbf{x}, \mathbf{x}', \mathbf{U})$ is simply the sum of local pairwise regrets:

$$R(\mathbf{x}, \mathbf{x}', \mathbf{U}) = \sum_k r_{\mathbf{x}[k],\mathbf{x}'[k]}. \tag{14}$$

We can compute max regret $MR(\mathbf{x}, \mathbf{U})$ by substituting Eq. (14) into Eq. (5):

$$MR(\mathbf{x}, \mathbf{U}) = \max_{\mathbf{x}' \in Feas(\mathbf{X})} \sum_k r_{\mathbf{x}[k],\mathbf{x}'[k]}, \tag{15}$$

which leads to the following IP formulation:

$$MR(\mathbf{x}, \mathbf{U}) = \max_{\{I_{\mathbf{x}'[k]}, X_i'\}} \sum_k \sum_{\mathbf{x}'[k]} r_{\mathbf{x}[k],\mathbf{x}'[k]} I_{\mathbf{x}'[k]} \quad \text{subject to } \mathcal{A} \text{ and } \mathcal{C}. \tag{16}$$

The above IP differs from its flat counterpart (Eq. (11)) in the use of one indicator variable $I_{\mathbf{x}'[k]}$ per utility parameter rather than one per state, and is thus much more compact and efficiently solvable. Indeed, the size of the IP in terms of the number of variables and constraints (excluding exogenously determined feasibility constraints $\mathcal{C}$) is linear in the size of the underlying factored utility model.

4.2.2. Computing minimax regret: constraint generation

We can compute minimax regret $MMR(\mathbf{U})$ by substituting Eq. (15) into Eq. (7):

$$MMR(\mathbf{U}) = \min_{\mathbf{x} \in Feas(\mathbf{X})} \max_{\mathbf{x}' \in Feas(\mathbf{X})} \sum_k r_{\mathbf{x}[k],\mathbf{x}'[k]} \tag{17}$$

which leads to the following MIP formulation:

$$MMR(\mathbf{U}) = \min_{\{I_{\mathbf{x}[k]}, X_i\}} \max_{\mathbf{x}' \in Feas(\mathbf{X})} \sum_k \sum_{\mathbf{x}[k]} r_{\mathbf{x}[k],\mathbf{x}'[k]} I_{\mathbf{x}[k]} \quad \text{subject to } \mathcal{A} \text{ and } \mathcal{C} \tag{18}$$

$$= \min_{\{I_{\mathbf{x}[k]}, X_i, M\}} M \qquad \text{subject to } \begin{cases} M \geqslant \sum_k \sum_{\mathbf{x}[k]} r_{\mathbf{x}[k],\mathbf{x}'[k]} I_{\mathbf{x}[k]} & \forall \mathbf{x}' \in Feas(\mathbf{X}), \\ \mathcal{A} \text{ and } \mathcal{C}. \end{cases} \tag{19}$$

In Eq. (18), we introduce the variables for the minimization, while in Eq. (19) we transform the minimax program into a min program. The new real-valued variable M corresponds to the max regret of the minimax-optimal solution. In contrast with the flat IP (Eq. (13)), this MIP has a number of $I_{\mathbf{x}[k]}$ variables that is linear in the number of utility parameters. However, this MIP is not generally compact because Eq. (19) has one constraint per feasible state $\mathbf{x}'$. Nevertheless, we can get around the potentially large number of constraints in either of two ways.

The first technique we consider for dealing with the large number of constraints in Eq. (19) is *constraint generation*, a common technique in operations research for solving problems with large numbers of constraints. Our approach can be viewed as a form of Benders' decomposition [6,36]. This approach proceeds by repeatedly solving the MIP in Eq. (19), but using only a subset of the constraints on M associated with the feasible states $\mathbf{x}'$. At the first iteration, all constraints on M are ignored. At each iteration, we obtain a solution indicating some decision $\mathbf{x}$ with purported minimax regret; however, since certain unexpressed constraints may be violated, we cannot be content with this solution. Thus, we look for the unexpressed constraint on M that is maximally violated by the current solution. This involves finding a *witness* $\mathbf{x}'$ that maximizes regret w.r.t. the current solution $\mathbf{x}$; that is, a decision $\mathbf{x}'$ (and, implicitly, a utility function) that an adversary would choose to cause a user to regret $\mathbf{x}$ the most.

Recall that finding the feasible $\mathbf{x}'$ that maximizes $R(\mathbf{x}, \mathbf{x}', \mathbf{U})$ involves solving a single IP given by Eq. (16). We then impose the specific constraint associated with witness $\mathbf{x}'$ and re-solve the MIP in Eq. (19) at the next iteration with this additional constraint. Formally, we have the following procedure:

(1) Let $Gen = \{\mathbf{x}'\}$ for some arbitrary feasible $\mathbf{x}'$.
(2) Solve the MIP in Eq. (19) using the constraints corresponding to states in *Gen*. Let $\mathbf{x}^*$ be the MIP solution with objective value m^*.
(3) Compute the max regret of state $\mathbf{x}^*$ using the IP in Eq. (16), producing a solution with regret level r^* and witness (adversarial state) $\mathbf{x}''$. If $r^* > m^*$, then add $\mathbf{x}''$ to *Gen* and repeat from Step 2; otherwise (if $r^* = m^*$), terminate with minimax-optimal solution x^* (with regret level m^*).

Intuitively, when we solve the MIP in Step 2 using only the constraints in *Gen*, we are computing minimax regret against a *restricted adversary*: the adversary is only allowed to use choices $\mathbf{x}' \in Gen$ in order to make us regret our solution $\mathbf{x}^*$ to the MIP. As such, this solution provides a lower bound on true minimax regret (i.e., the solution that would have been obtained were a completely unrestricted adversary considered).

When we compute the true max regret r^* of $\mathbf{x}^*$ in Step 3, we also obtain an upper bound on minimax regret (since we can always attain max regret of r^* simply by stopping and recommending solution $\mathbf{x}^*$). It is not hard to see that if $r^* = m^*$, then no constraint is violated at the current solution $\mathbf{x}^*$ (and our upper and lower bounds on minimax regret coincide); so $\mathbf{x}^*$ is the minimax-optimal configuration at this point. The procedure is finite and guaranteed to arrive at the optimal solution. The constraint generation routine is not guaranteed to finish before it has the full set of constraints, but it is relatively simple and (as we will see) tends to generate a very small number of constraints. Thus in practice we solve this very large MIP using a series of small MIPs, each with a small number of variables and a set of active constraints that is also, typically, very small.

Since minimax regret will be computed between elicitation queries, it is critical that minimax regret be estimated in a relatively short period of time (e.g., five seconds for certain applications, five minutes for others, possibly several hours for very high stakes applications). With this in mind, several improvements can be made to speed up minimax regret computation. For instance, it is often sufficient to find a feasible (instead of optimal) configuration $\mathbf{x}$ for the MIP in Eq. (19) for each newly generated constraint. Intuitively, as long as the feasible $\mathbf{x}$ allows us to find a violated constraint—constraint generation continues to make progress. Hence, instead of waiting a long time for an optimal $\mathbf{x}$, we can stop the MIP solver as soon as we find a feasible solution for which a violated constraint exists. Of course, at the last iteration, when there are no violated constraints, we have no choice but to wait for the optimal $\mathbf{x}$.

Minimax regret can also be estimated more quickly—to allow for the real-time response needed for interactive optimization—by exploiting the anytime nature of the computation to simply stop early. Since minimax regret is computed incrementally by generating constraints, early stopping has the effect that some violated constraints may not have been generated. As a result the solution provides us with a lower bound on minimax regret. We can terminate early based on a fixed number of iterations (constraints), a fixed amount of computation time, or by terminating when bounds on the solution are tight enough. Apart from this lower bound, we can also obtain an upper bound on minimax regret by computing the max regret of the $\mathbf{x}$ found for the last minimax MIP solved. Note that we may need to explicitly compute this since Step (3) of our procedure may not be invoked if we terminate based only on the number of iterations rather than testing for constraint violation.

Approximation can be very appealing if real-time interactive response is required. The anytime flavor of the algorithm means that these lower and upper bounds are often tight enough to provide elicitation guidance of similar quality to that obtained from computing minimax regret exactly.

Although the full interaction of minimax regret computation with elicitation is explored in Section 5, as a precursor to that discussion, we mention another strategy for accelerating computation which directly influences the querying process. We have observed, unsurprisingly, that the minimax regret problem solved after receiving a response to one query is very similar to that solved before posing the query. As such, one can "seed" the minimax procedure invoked after a query with the constraints generated at the previous step. In this way, typically, only a few extra constraints are generated during each minimax computation. Given that the running time of minimax regret is dominated by constraint generation, this effectively amortizes the cost of minimax computation over a number of queries.

4.2.3. Computing minimax regret: a cost network formulation

A second technique for dealing with the large number of constraints in Eq. (19) is to use a cost network to generate a *compact* set of constraints that effectively summarizes this set. This type of approach has been used recently, for example, to solve Markov decision processes [26]. The main benefit of the cost network approach is that, in principle, it allows us to formulate a MIP with a feasible number of constraints (as elaborated below). We have observed, however, the constraint generation approach described above is usually much faster in practice and much easier to implement, even though it lacks the same worst-case run-time guarantees. Indeed, this same fact has been observed in the context of MDPs [47]. It is for this reason that we emphasize (and only experiment with) the constraint generation algorithm. However, we sketch the cost network formulation for completeness.

To formulate a compact constraint system, we first transform the MIP of Eq. (19) into the following equivalent MIP by introducing penalty terms $\rho_{\mathbf{x}[\ell]}$ for each feasibility constraint $\mathcal{C}_\ell$:

$$\begin{aligned} MMR(\mathbf{U}) &= \min_{\{I_{\mathbf{x}[k]}, X_i, M\}} M \quad \text{s.t.} \quad \begin{cases} M \geqslant \sum_k \sum_{\mathbf{x}[k]} r_{\mathbf{x}[k],\mathbf{x}'[k]} I_{\mathbf{x}[k]} + \sum_\ell \rho_{\mathbf{x}'[\ell]} & \forall \mathbf{x}' \in Dom(\mathbf{X}) \\ \mathcal{A} \text{ and } \mathcal{C} \end{cases} \\ &= \min_{\{I_{\mathbf{x}[k]}, X_i, M\}} M \quad \text{s.t.} \quad \begin{cases} M \geqslant \sum_k R_{\mathbf{x}'[k]} + \sum_\ell \rho_{\mathbf{x}'[\ell]} & \forall \mathbf{x}' \in Dom(\mathbf{X}) \\ R_{\mathbf{x}'[k]} = \sum_{\mathbf{x}[k]} r_{\mathbf{x}[k],\mathbf{x}'[k]} I_{\mathbf{x}[k]} & \forall k, \mathbf{x}'[k] \in Dom(\mathbf{X}[k]) \\ \mathcal{A} \text{ and } \mathcal{C}. \end{cases} \end{aligned} \tag{20}$$

The MIP of Eq. (19) has one constraint on M per feasible state $\mathbf{x}'$, whereas the MIP of Eq. (20) has one constraint per state $\mathbf{x}'$ (whether feasible or not). Therefore, to effectively maintain the feasibility constraints on $\mathbf{x}'$, we add penalty terms $\rho_{\mathbf{x}'[\ell]}$ that make a constraint on M meaningless when its corresponding state $\mathbf{x}'$ is infeasible. This is achieved by defining a local penalty function $\rho^\ell(\mathbf{x}'[\ell])$ for each logical constraint $\mathcal{C}_\ell$ that returns $-\infty$ when $\mathbf{x}'[\ell]$ violates $\mathcal{C}_\ell$ and 0 otherwise.

This transformation has, unfortunately, increased the number of constraints. However, it in fact allows us to rewrite the constraints in a much more compact form, as follows. Instead of enumerating all constraints on M, we analytically construct the constraint that provides the *greatest lower bound*, while simply ignoring the others. This greatest lower bound *GLB* is computed by taking the max of all constraints on M:

$$\begin{aligned} GLB &= \max_{\mathbf{x}'} \sum_k R_{\mathbf{x}'[k]} + \sum_\ell \rho_{\mathbf{x}'[\ell]} \\ &= \max_{x'_1} \max_{x'_2} \ldots \max_{x'_N} \sum_k R_{\mathbf{x}'[k]} + \sum_\ell \rho_{\mathbf{x}'[\ell]}. \end{aligned}$$

This maximization can be computed efficiently by using *variable elimination* [19], a well-known form of non-serial dynamic programming [7]. The idea is to distribute the max operator inward over the summations, and then collect the results as new terms which are successively pulled out. We illustrate its workings by means of an example.

Suppose we have the attributes X_1, X_2, X_3, X_4, a utility function decomposed into the factors $f_1(x_1, x_2)$, $f_2(x_2, x_3)$, $f_3(x_1, x_4)$ and two logical constraints with associated penalty functions $\rho^1(x_1)$ and $\rho^2(x_3, x_4)$. We then obtain

$$\begin{aligned} GLB &= \max_{x'_1} \max_{x'_2} \max_{x'_3} \max_{x'_4} R_{x'_1,x'_2} + R_{x'_2,x'_3} + R_{x'_1,x'_4} + \rho_{x'_1} + \rho_{x'_3,x'_4} \\ &= \max_{x'_1} [\rho_{x'_1} + \max_{x'_2} [R_{x'_1,x'_2} + \max_{x'_3} [R_{x'_2,x'_3} + \max_{x'_4} [R_{x'_1,x'_4} + \rho_{x'_3,x'_4}]]]] \end{aligned}$$

by distributing the individual max operators inward over the summations. To compute the *GLB*, we successively formulate new terms that summarize the result of completing each max in turn, as follows:

Let $A_{x'_1,x'_3} = \max_{x'_4} R_{x'_1,x'_4} + \rho_{x'_3,x'_4}$.

Let $A_{x'_1,x'_2} = \max_{x'_3} R_{x'_2,x'_3} + A_{x'_1,x'_3}$.

Let $A_{x'_1} = \max_{x'_2} R_{x'_1,x'_2} + A_{x'_1,x'_2}$.

Let $GLB = \max_{x'_1} \rho_{x'_1} + A_{x'_1}$.

Notice that this incremental procedure can be substantially faster than enumerating all states $\mathbf{x}'$. In fact the complexity of each step is only exponential in the local subset of attributes that indexes each auxiliary A variable.

Based on this procedure, we can substitute all the constraints on M in the MIP in Eq. (20) with the following compact set of constraints that analytically encodes the greatest lower bound on M:

$$\begin{aligned}
&A_{x_1',x_3'} \geqslant R_{x_1',x_4'} + \rho_{x_3',x_4'} \quad \forall x_1', x_3', x_4' \in Dom(X_1, X_3, X_4),\\
&A_{x_1',x_2'} \geqslant R_{x_2',x_3'} + A_{x_1',x_3'} \quad \forall x_1', x_2', x_3' \in Dom(X_1, X_2, X_3),\\
&A_{x_1'} \geqslant R_{x_1',x_2'} + A_{x_1',x_2'} \quad \forall x_1', x_2' \in Dom(X_1, X_2),\\
&M \geqslant \rho_{x_1'} + A_{x_1'} \quad \forall x_1' \in Dom(X_1).
\end{aligned}$$

By encoding constraints in this way, the constraint system specified by the MIP in Eq. (20) can be generally encoded with a small number of variables and constraints. Overall we obtain a MIP where: the number of $I_{\mathbf{x}}$ variables is linear in the number of parameters of the utility function; and the number of auxiliary variables (the A variables in our example) and constraints that are added is locally exponential with respect to the largest subset of attributes indexing some auxiliary variable. In practice, since this largest subset is often very small compared to the set of all attributes, the resulting MIP encoding is compact and readily solvable. In particular, let the *joint graph* be the union of the constraint graph and the utility graph (e.g., the union of the graphs in Figs. 1 and 2). The complexity of this algorithm and hence the size of the resultant set of constraints is determined directly by the properties of variable elimination, and as such depends on the order in which the variables in $\mathbf{X}$ are eliminated. More precisely, it is exponential in the *tree width* of the joint graph induced by the elimination ordering [19]. Often this tree width is very small, thus rendering the algorithm only locally exponential [19].

4.3. Empirical results

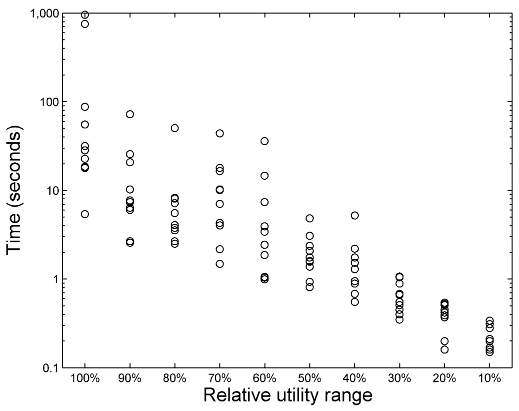
To test the plausibility of minimax regret computation, we implemented the constraint generation strategy outlined above and ran a series of experiments to determine whether factored structure was sufficient to permit practical solution times. We implemented the constraint generation approach outlined in Section 4.2 and used CPLEX 9.0 as the generic IP solver.[14] Our experiments considered two realistic domains—car rentals and real estate—as well as randomly generated synthetic problems. In each case we imposed a factored structure to reduce the required number of utility parameters (upper and lower bounds).

For the real-estate problem, we modeled the domain with 20 (multivalued) variables that specify various attributes of single family dwellings that are normally relevant to making a purchase decision. The variables we used included: square footage, age, size of yard, garage, number of bedrooms, etc. Variables have domains with sizes ranging from two to four values. In total, there were 47,775,744 possible configurations of the variables. We then used a factored utility model consisting of 29 local factors, each defined on only one, two or three variables. In total, there were 160 utility parameters (i.e., utilities for local configurations). Therefore a total of 320 upper and lower bounds had to be specified, a significant reduction over the nearly 10^8 values that would have been required using a unfactored model. The local utility functions represented complementarities and substitutabilities in the utility function, such as requiring a large yard and a fence to allow a pool, sacrificing a large yard if the house happens to be near a park, etc.

The car-rental problem features 26 multi-valued variables encoding attributes relevant to consumers considering a car rental, such as: automobile size and class, manufacturer, rental agency, seating and luggage capacity, safety features (air bags, ABS, etc.), and so on. The size of the domain of the variables varies from 2 to 9. The total number of possible variable configurations is 61,917,360,000. There are 36 local utility factors, each defined on at most five variables, giving rise to 435 utility parameters. Constraints encode infeasible configurations (e.g., no luxury sedans have four-cylinder engines).

For both the car-rental and real-estate problems, we first computed the configuration with minimax regret given manually chosen bounds on the utility functions. The constraint generation technique of Section 4.2 took 15 seconds for the car-rental problem and 0.48 seconds for the real-estate problem. It is interesting to note that only 63 constraints

[14] These experiments were performed on 3.0 GHz PCs.



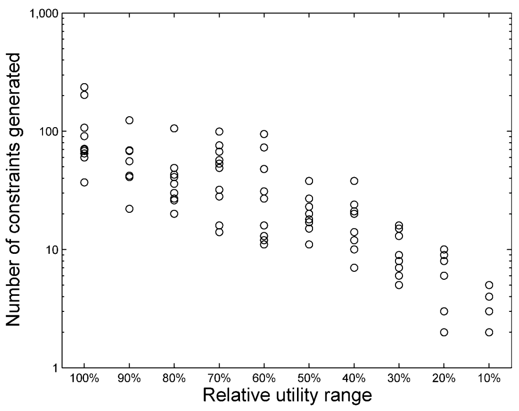
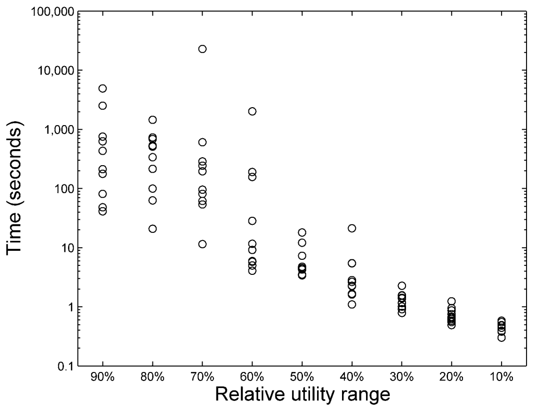


Fig. 4. Computation time (left) and number of constraints generated (right) for minimax regret on real-estate problem (48 million configurations) as a function of the tightness of the utility bounds.



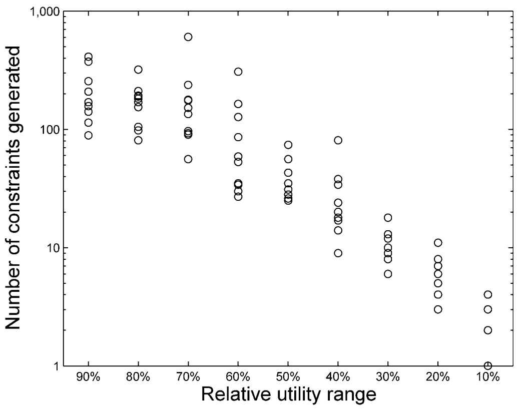


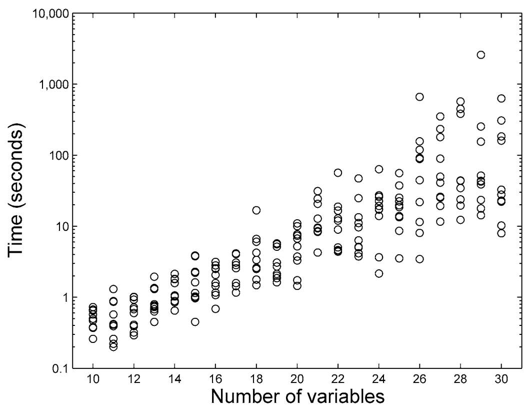
Fig. 5. Computation time (left) and number of constraints generated (right) for minimax regret on car-rental problem (62 billion configurations) as a function of the tightness of the utility bounds.

(out of 61,917,360,000 possible constraints) for the car-rental problem and seven constraints (out of 47,775,744 possible constraints) for the real-estate problem were generated in the search for the minimax optimal configuration. The structure exhibited by the utility functions of each problem is largely responsible for this small number of required constraints.

In practice, minimax regret computation will be interleaved with some preference elicitation technique (as we discuss in Section 5). As the bounds on utility parameters get tighter, we would like to know the impact on the running time of our constraint generation algorithm. To that effect, we carried out an experiment where we randomly set bounds, but with varying degrees of tightness. Initial utility gaps (i.e., difference between upper and lower bounds) ranged from 0 to 100. Figs. 4 and 5 show how tightening the bounds decreases both the running time and the number of constraints generated in an exponential fashion. For this experiment, bounds on utility were generated at random, but the difference between the upper and lower bounds of any utility was capped at a fixed percentage of some predetermined range. Intuitively, as preferences are elicited, the values will shrink relative to the initial range.

Figs. 4 and 5 show scatterplots of computation time and number of constraints for ten random problem instances generated for each of a number of increasingly tight relative utility ranges. As those figures suggest, a significant speed up is obtained as elicitation converges to the true utilities. Intuitively, the optimization required to compute minimax regret benefits from tighter bounds since some configurations emerge as clearly dominant, which in turn requires the generation of fewer constraints.

We carried out a second experiment with synthetic problems. A set of random problems of varying sizes was constructed by randomly setting the utility bounds as well as the variables on which each utility factor depends. Each utility factor depends on at most three variables and each variable has at most five values. Fig. 6 shows the results as



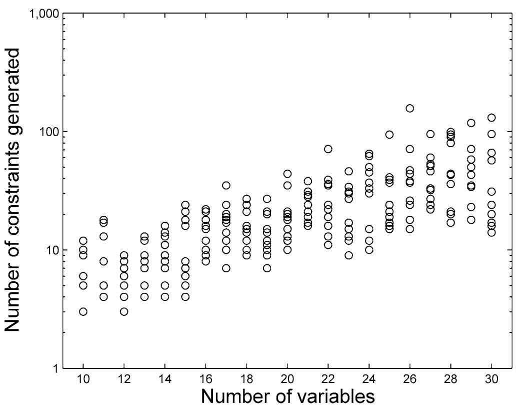
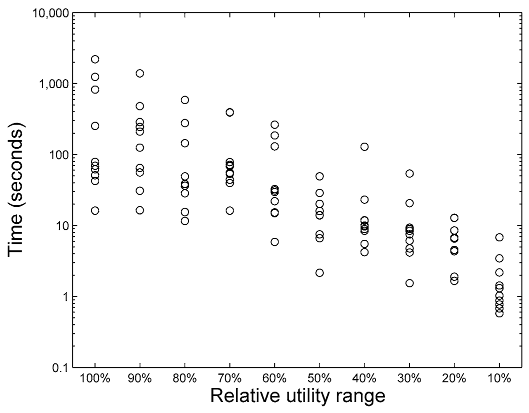


Fig. 6. Computation time (left) and number of constraints generated (right) for artificial random problems as a function of problem size (number of variables and factors).



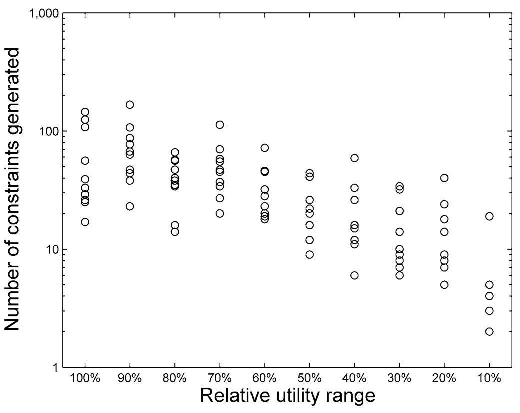


Fig. 7. Computation time (left) and number of constraints generated (right) for minimax regret on artificial random problems (30 variables, 30 factors) as a function of the tightness of the utility bounds.

we vary the number of variables and factors (the number of factors is always the same as the number of variables). The running time and the number of constraints generated increases exponentially with the size of the problem. Note however that the number of constraints generated is still a tiny fraction of the total number of possible constraints. For problems with 10 variables, only 7 constraints were necessary (out of 278,864) on average; and for problems with 30 variables, only 47 constraints were necessary (out of 2.8×10^{16}) on average.

We also tested the impact of the relative tightness of utility bounds on the efficiency of our constraint generation technique, with results shown in Fig. 7. Here, problems of 30 variables and 30 factors were generated randomly while varying the relative range of the utilities with respect to some predetermined range. Each factor has at most three variables chosen randomly and each variable can take at most five values. Once again, as the bounds get tighter, some configurations emerge as clearly dominant, which allows an exponential reduction in the running time as well as the number of required constraints.

Finally, we illustrate the anytime properties of our algorithm. In Fig. 8 we show the lower bound on minimax regret as a function of computation time. Each data point corresponds to one additional generated constraint. As we can see, the constraint generation algorithm has very good anytime properties, approaching the true minimax regret level very quickly as a function of time and number of constraints. This is due to two factors. First, as a function of the number of constraints generated, minimax regret lower bounds increase much more quickly early on, thus exhibiting the desired anytime behavior. Second, solution time with smaller numbers of constraints tends to be considerably less than with larger numbers of constraints. This enhances the anytime profile with respect to time (providing a much steeper increase than one would see if plotting the bound with respect to number of constraints generated). For example, in the rental car problem shown, the first ten constraints are generated in 1.7 s, the first twenty in 4.6 s, the first thirty in

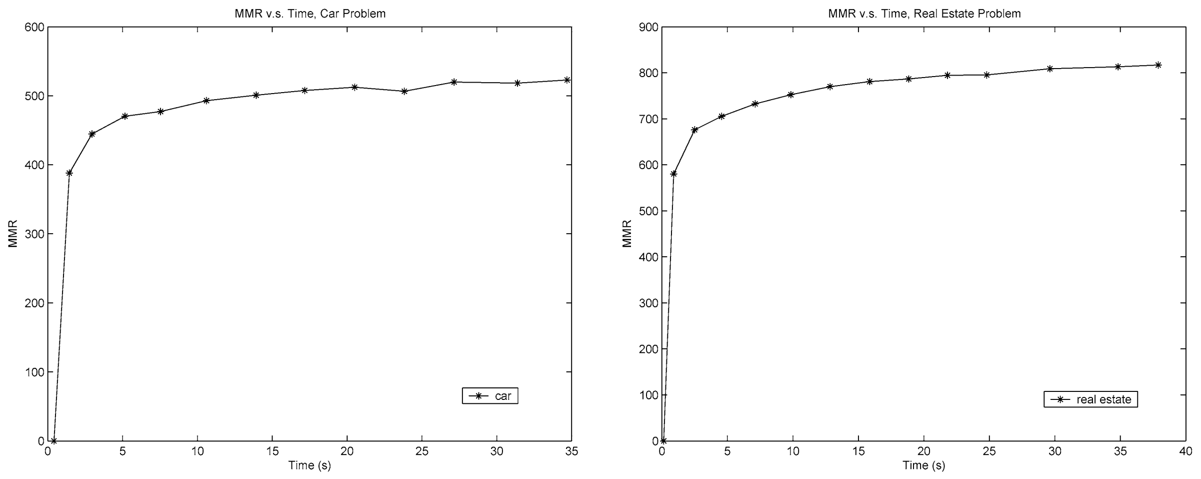


Fig. 8. Lower bound on minimax regret until convergence as a function of computation time. Results for Rental Car (left) and Real Estate (right), both at 70% of utility range. Solution quality is plotted for the solutions generated by adding one additional constraint. Data points are marked for every fifth constraint generated.

8.9 s, and so on until convergence to the true minimax regret after 57 constraints (24.62 s). This anytime property has important implications for real-time preference elicitation as we discuss below.

4.4. *Minimax regret with linear utility constraints*

The computational methods above exploit the existence of upper and lower bounds on utility parameters. While some types of queries used in elicitation allow one to maintain such independent bounds, other forms of queries (e.g., comparison queries) impose arbitrary linear constraints on these utility parameters, demanding new methods for computing minimax regret. We now develop an IP-based procedure for solving COPs with linear constraints on the utility parameters.

Suppose we have an imprecise structured COP $\langle \mathcal{C}, \mathcal{U} \rangle$ where **U** is a polytope defined by a finite set of (arbitrary) linear *utility constraints* $\mathcal{U}$ over the parameters $u_{\mathbf{x}[k]}$ of a factored utility model $\{f_k\colon\ k \leqslant K\}$. Thus we relax the assumption that the constraints $\mathcal{U}$ take the form of bounds.

Computing the max regret of a state **x** can no longer rely on the existence of local pairwise regrets as in Eq. (16). However, we can reformulate the problem somewhat differently to allow this to be solved linearly even when the constraints $\mathcal{U}$ take this more general form. First, we can recast the computation of max regret as a quadratic optimization:

$$MR(\mathbf{x}, \mathbf{U}) = \max_{\{I_{\mathbf{x}'[k]}, X_i', U_{\mathbf{x}[k]}\}} \sum_k \left[\sum_{\mathbf{x}'[k]} U_{\mathbf{x}'[k]} I_{\mathbf{x}'[k]} \right] - U_{\mathbf{x}[k]} \quad \text{subject to } \mathcal{A},\ \mathcal{C},\ \text{and } \mathcal{U}. \tag{21}$$

We have introduced one real-valued variable $U_{\mathbf{x}[k]}$ (denoted $U_{\mathbf{x}'[k]}$ when referring to specific adversarial local states $\mathbf{x}'$) for each utility parameter $u_{\mathbf{x}[k]}$, reflecting their unknown nature; these are constrained by $\mathcal{U}$. The presence of the $U_{\mathbf{x}'[k]}$ variables renders the optimization quadratic. However, this can be reformulated by the introduction of new real-valued variables $Y_{\mathbf{x}'[k]}$ for each such utility variable. Intuitively, $Y_{\mathbf{x}'[k]}$ denotes the product $U_{\mathbf{x}'[k]} I_{\mathbf{x}'[k]}$. This product can be defined by assuming (loose) upper bounds $u_{\mathbf{x}[k]}\!\uparrow$ on the utility parameters. Specifically, we rewrite Eq. (21) as follows:

$$MR(\mathbf{x}, \mathbf{U}) = \max_{\{I_{\mathbf{x}'[k]}, X_i', U_{\mathbf{x}[k]}, Y_{\mathbf{x}'[k]}\}} \sum_k \left(\sum_{\mathbf{x}'[k]} Y_{\mathbf{x}'[k]} \right) - U_{\mathbf{x}[k]} \quad \text{subject to } \begin{cases} Y_{\mathbf{x}'[k]} \leqslant I_{\mathbf{x}'[k]} u_{\mathbf{x}'[k]}\!\uparrow & \forall k, \mathbf{x}'[k], \\ Y_{\mathbf{x}'[k]} \leqslant U_{\mathbf{x}'[k]} & \forall k, \mathbf{x}'[k], \\ \mathcal{A}, \mathcal{C} \text{ and } \mathcal{U}. \end{cases} \tag{22}$$

The constraints on $Y_{\mathbf{x}'[k]}$, together with the fact that the objective aims to maximize its value, ensure that it takes the value zero if $I_{\mathbf{x}'[k]} = 0$ and takes the value $U_{\mathbf{x}'[k]}$ otherwise.[15]

With the ability to compute max regret by solving a MIP, we can use a variant of the constraint generation procedure described in Section 4.2. Notice that the solution to the MIP in Eq. (22) produces a witness $\mathbf{x}'$ as well as a specific utility function in which each variable $U_{\mathbf{x}[k]}$ is set to the value of utility parameter $u_{\mathbf{x}[k]}$ that maximizes the regret of the state in question. Notice also that the state $\mathbf{x}'$ must be the optimal feasible state for the chosen utility function $u \in \mathbf{U}$ (otherwise regret could be made even higher).

We can then express minimax regret in a way similar to Eqs. (18) and (19).

$$\begin{aligned} MMR(\mathbf{U}) &= \min_{\{I_{\mathbf{x}[k]}, X_i\}} \max_{\mathbf{x}' \in Feas(\mathbf{X}'), u \in \mathbf{U}} \sum_k \Bigl(u_{\mathbf{x}'[k]} - \sum_{\mathbf{x}[k]} u_{\mathbf{x}[k]} I_{\mathbf{x}[k]} \Bigr) \quad \text{s.t.} \quad \mathcal{A} \text{ and } \mathcal{C} \\ &= \min_{\{I_{\mathbf{x}[k]}, X_i\}} M \quad \text{s.t.} \quad \begin{cases} M \geqslant \sum_k \bigl(u_{\mathbf{x}'[k]} - \sum_{\mathbf{x}[k]} u_{\mathbf{x}[k]} I_{\mathbf{x}[k]} \bigr) & \forall \mathbf{x}' \in Feas(\mathbf{X}'), u \in \mathbf{U}, \\ \mathcal{A} \text{ and } \mathcal{C}. \end{cases} \end{aligned} \tag{23}$$

We can use the max regret computation described in Eq. (22) to generate constraints iteratively as required for Eq. (23).

5. Elicitation strategies

While the use of minimax regret provides a useful way of handling imprecise utility information, the initial bounds on utility parameters provided by users are unlikely to be tight enough to admit configurations with provably low regret. Instead, we imagine an interactive process in which the decision software queries the user for further information about her utility function—refining bounds or constraints on the parameters—until minimax regret, given the current constraints, reaches an acceptable level τ.[16] We can summarize the general form of the interactive elicitation procedure as follows:

(1) Compute minimax regret *mmr*.
(2) Repeat until $mmr < \tau$:
   (a) Ask query q.
   (b) Update the constraints $\mathcal{U}$ over utility parameters to reflect the response to q.
   (c) Recompute *mmr* with respect to new constraint set $\mathcal{U}$.

We begin by discussing bound queries, the primary type of query that we consider here, then describe a number of elicitation strategies using bound queries. Throughout most of this section we assume some imprecise structured COP problem $\langle \mathcal{C}, \mathcal{U} \rangle$ where $\mathcal{U}$ is specified by a factored utility model $\{f_k\colon k \leqslant K\}$ with upper and lower bounds on its parameters. However, we will also discuss comparison queries, and hence $\mathcal{U}$ in which arbitrary linear constraints are present, in Section 5.6.

There are a number of important issues regarding user interaction that will need to be addressed in the development of any interactive decision support software. The perspective we adopt here is rather rigid and assumes users can (somewhat comfortably) answer the types of queries we pose. We do not consider issues of framing, preference construction or exploration, or other issues surrounding the mode of interaction. Nor we do consider users who may express inconsistent preferences (indeed, none of our strategies will ever ask a query that can be responded to inconsistently). However, we believe the core of our elicitation techniques can certainly be incorporated into the larger context in which these important issues are addressed. For a discussion of some of these issues in the context of constraint-based optimization, see the work of Pu, Faltings, and Torrens [39].

[15] Note that this relies on the fact that each $U_{\mathbf{x}'[k]}$ is non-negative. If we allow negative local utility, these constraints can be generalized by exploiting a (loose) lower bound on $U_{\mathbf{x}'[k]}$ as well.

[16] We could insist that regret reaches zero (i.e., that we have a provably optimal solution), or stop when regret reaches a point where further improvement is outweighed by the cost of additional interaction.

5.1. Bound queries

Bound queries form the primary class of queries we consider, in which we ask the user whether one of her utility parameters lies above a certain value. A positive response raises the lower bound on that parameter, while a negative response lowers the upper bound: in both cases, uncertainty is reduced.

While users often have difficulty assessing numerical parameters, they are typically better at comparing outcomes [24,30]. Fortunately, a bound query can be viewed as a local form of a *standard gamble query* (*SGQ*), commonly used in decision analysis; these in fact ask for comparisons. An SGQ for a specific state $\mathbf{x}$ asks the user if she prefers $\mathbf{x}$ to a gamble in which the best outcome $\mathbf{x}_\top$ occurs with probability l and the worst $\mathbf{x}_\perp$ occurs with probability $1 - l$ [30]. A positive response puts a lower bound on the utility of $\mathbf{x}$, and a negative response puts an upper bound. Calibration is attained by the use of common best and worst outcomes across all queries (and numerical assessment is restricted to evaluating probabilities). Thus a bound query "Is $u(\mathbf{x}) > q$?" can be cast as a standard gamble query: "Do you prefer $\mathbf{x}$ to a gamble in which $\mathbf{x}_\top$ is obtained with probability q and $\mathbf{x}_\perp$ is obtained with probability $1 - q$?"[17]

For instance, ignoring factorization, one might ask in the car rental domain: "Would you prefer CAR27 or a gamble in which your received CARB with probability l and CARW with probability $1 - l$?" Here CAR27 is the specific *complete* outcome of interest, while CARB and CARW are the best and worst possible car configurations, respectively (these need not be feasible in general). Of course, given the factorization of the model, we would prefer not to focus a user's attention on complete outcomes, but rather take advantage of the utility independence inherent in the GAI model to elicit information about local outcomes. As a consequence, we will ask analogous bound queries on local factors.

Our general elicitation procedure when restricted to bound queries takes the following form:

```
(1) Compute minimax regret $mmr$.
(2) Repeat until $mmr < \tau$:
    (a) Ask a bound query "Is $u_{\mathbf{x}[k]} \leqslant q$?" of some utility parameter $u_{\mathbf{x}[k]}$.
    (b) If $u_{\mathbf{x}[k]} \leqslant q$ then reduce upper bound $u_{\mathbf{x}[k]}\uparrow$ to $q$. Otherwise raise lower bound $u_{\mathbf{x}[k]}\downarrow$ to $q$.
    (c) Recompute $mmr$ using the new bounds.
```

While we focus on bound queries, other forms of queries are quite natural. For example, comparison queries ask if one state $\mathbf{x}$ is preferred to another $\mathbf{x}'$ and are discussed further in Section 5.6. Hierarchical structuring of attributes is another avenue that could be considered in posing queries, though we leave this for future research within our model.

The foundations of bound queries can be made precise using results of Fishburn [22]. Roughly speaking, we require calibration across factors in the GAI model in order to be sure that the stated comparisons are meaningful. Gonzales and Perny [25] provide a specific procedure for (full) elicitation in GAI networks that relies on asking queries over complete outcomes, while Braziunas and Boutilier [15] provide a algorithm that allows for local queries (over small subsets of attributes). Bound queries in our framework can be supplemented with a small number of additional calibration queries as suggested in [15] if one requires calibration across factors for the user. Alternatively, if the scales associated with each of the GAI factors is obviously calibrated (e.g., the "utilities" refer to the monetary amount the user is willing to pay for a specific combination of attributes), then bound queries can be used directly without need for additional calibration. We refer to [15,25] for further details.

Several of our bound query strategies rely on the following definitions.

Definition 4. Let $\langle \mathcal{C}, \mathcal{U} \rangle$ be an imprecise COP problem. An *optimistic state* $\mathbf{x}^o$, a *pessimistic state* $\mathbf{x}^p$, and a *most uncertain state* $\mathbf{x}^{mu}$ are any states satisfying (respectively):

$$\mathbf{x}^o \in \arg\max_{\mathbf{x} \in Feas(\mathbf{X})} \max_{u \in \mathbf{U}} u(\mathbf{x}), \qquad \mathbf{x}^p \in \arg\max_{\mathbf{x} \in Feas(\mathbf{X})} \min_{u \in \mathbf{U}} u(\mathbf{x}), \qquad \mathbf{x}^{mu} \in \arg\max_{\mathbf{x} \in Feas(\mathbf{X})} \max_{u, u' \in \mathbf{U}} u(\mathbf{x}) - u'(\mathbf{x}).$$

An optimistic state is a feasible state with the greatest upper bound on utility. A pessimistic state has the greatest lower bound on utility. A most uncertain state has the greatest difference between its upper and lower bounds. Each

[17] If the user is nearly indifferent to the two alternatives, they may be tempted to respond "I don't know". This can be handled by imposing a quantitative interpretation on "near indifference" and imposing a constraint that makes these two utilities "close".

of these states can be computed in a single optimization by setting the parameters of the utility model to their upper bounds, their lower bounds, or their difference, and solving the corresponding (precise) COP problem.

5.2. The halve largest gap strategy

The first query strategy we consider is the *halve largest gap* (*HLG*) *strategy*. It asks a query at the midpoint of the interval of the parameter $\mathbf{x}[k]$ with the largest gap between its upper and lower bounds. This is motivated by theoretical considerations, based on simple worst-case bounds on minimax regret.

Definition 5. Define the *gap* of a utility parameter $u_{\mathbf{x}[k]}$, the *span* of factor f_k and *maxspan* of our utility model as follows:

$$gap(\mathbf{x}[k]) = u_{\mathbf{x}[k]}\uparrow - u_{\mathbf{x}[k]}\downarrow, \tag{24}$$

$$span(f_k) = \max_{\mathbf{x}[k] \in Dom(\mathbf{X}[k])} gap(\mathbf{x}[k]), \tag{25}$$

$$maxspan(\mathbf{U}) = \sum_k span(f_k). \tag{26}$$

The quantity *maxspan* measures the largest difference between the upper and lower utility bound, regardless of feasibility. We can show that this quantity bounds minimax regret:

Proposition 1. *For any* $\langle \mathcal{C}, \mathcal{U} \rangle$, $MMR(\mathbf{U}) \leqslant maxspan(\mathbf{U})$.

Proof. By definition of minimax regret, we have $MMR(\mathbf{U}) \leqslant MR(\mathbf{x}^o, \mathbf{U})$. For any optimistic state $\mathbf{x}^o$ and any alternative state $\mathbf{x}$ we must have that $u\uparrow(\mathbf{x}) - u\downarrow(\mathbf{x}^o) \leqslant u\uparrow(\mathbf{x}^o) - u\downarrow(\mathbf{x}^o) \leqslant maxspan(\mathbf{U})$ (i.e., the difference between the upper and lower bounds of $\mathbf{x}$ and $\mathbf{x}^o$, respectively, is bounded by $maxspan(\mathbf{U})$, since the upper bound of $\mathbf{x}$ cannot exceed that of $\mathbf{x}^o$, and the lower bound of $\mathbf{x}^o$ can be no less than $u\uparrow(\mathbf{x}^o) - maxspan(\mathbf{U})$). Thus, $MR(\mathbf{x}^o, \mathbf{U}) \leqslant maxspan(\mathbf{U})$. The result follows immediately. □

We note that the definition of *maxspan* can be tightened in two ways. (a) One could account for logical consistency across utility factors (e.g., if X occurs in two factors, we cannot have a total utility span for a single state that instantiates the span in one factor with X true, and the span in the other with X false). Computing this tighter definition of span requires some minor optimization to find the logically consistent state with maximum span, but is otherwise straightforward. (b) One could make this tighter still by restricting attention to feasible states (w.r.t. $\mathcal{C}$); in other words, maxspan would be defined as the "span" of any most uncertain state $\mathbf{x}^{mu}$. The result still holds with these tighter definitions. However, the current definition requires no optimization to assess.

The relationship between *maxspan* and minimax regret suggests an obvious query strategy, the HLG method, in which a bound query is asked of the local state $\mathbf{x}[k]$ with the largest utility gap, at the midway point of its interval, $(u_{\mathbf{x}[k]}\uparrow - u_{\mathbf{x}[k]}\downarrow)/2$. This method guarantees reasonably rapid reduction in max regret:

Proposition 2. *Let* $\mathbf{U}$ *be an uncertain utility model with* n *parameters and let* $m = maxspan(\mathbf{U})$. *After* $n \log(m/\varepsilon)$ *queries in the HLG strategy, minimax regret is no greater than* ε.

Proof. Given a utility model with n parameters with a specific initial set of gaps, the largest gap among all states must be reduced by at least half after n queries according to the HLG strategy (with this bound being tight only if the largest initial gap is no more than twice that of the smallest initial gap). Thus after kn queries, leading to an updated feasible utility set $\mathbf{U}'$, we have $maxspan(\mathbf{U}') \leqslant 2^{-k}m$. The result then follows by application of Proposition 1. □

In the worst case, there are classes of utility functions for which the bound is tight, so sets $\mathbf{U}$ and configuration constraints $\mathcal{C}$ exist that ensure regret will never be reduced to zero in finitely many queries. For example, if we have a linear utility function over $X_1, \ldots, X_n$ with

$$u(\mathbf{X}) = f_1(X_1) + \cdots + f_n(X_n),$$

with each local utility parameter having the same gap g and no feasibility constraints, then minimax regret can be reduced no more quickly than this.[18]

This strategy is similar to heuristically motivated polyhedral methods in conjoint analysis used in product design and marketing [29,48]. In fact, HLG can be viewed as a special case of the polyhedral method of [48] in which our polyhedra are hyper-rectangles.

5.3. The current solution strategy

While HLG allows one to provide strong worst-case guarantees on regret improvement, it is "undirected" in that considerations of feasibility play no role in determining which queries to ask. An alternative strategy is to focus attention on parameters that participate in defining minimax regret, namely, the minimax optimal $\mathbf{x}^*$ and the adversarial witness $\mathbf{x}^w$ for the current feasible utility set $\mathbf{U}$ (recall that the witness $\mathbf{x}^w$ maximizes the regret of $\mathbf{x}^*$). The *current solution* (*CS*) *query strategy* asks about the utility parameter in the set $\{\mathbf{x}^*[k]\colon\ k \leqslant K\} \cup \{\mathbf{x}^w[k]\colon\ k \leqslant K\}$ with largest $gap(\mathbf{x}[k])$ and queries the midpoint of the corresponding utility interval. Intuitively, should the answer to a query raise the lower bound on some $u_{\mathbf{x}^*[k]}$ or lower the upper bound on some $u_{\mathbf{x}^w[k]}$, then the pairwise regret $R(\mathbf{x}^*, \mathbf{x}^w, \mathbf{U})$ will be reduced, and usually minimax regret will be reduced as well. Of course, if the answer lowers the upper bound on some $u_{\mathbf{x}^*[k]}$ or raises the lower bound on some $u_{\mathbf{x}^w[k]}$, then pairwise regret $R(\mathbf{x}^*, \mathbf{x}^w, \mathbf{U})$ remains unchanged and minimax regret is not guaranteed to be reduced (though it may).

We have also experimented with a variant of the CS strategy in which regret is computed approximately to ensure fast interactive response in the querying process. This can be done by imposing a time bound on the solution algorithm for computing minimax regret, exploiting the anytime nature of the method described in Section 4.2. While we can't be sure we have the minimax optimal solution with early termination, the solution may be good enough to guide the querying process. Furthermore, since we can compute the max regret of the anytime solution, we have an upper bound on minimax regret which can be used as a natural termination criterion.

5.4. Alternative strategies

Finally, we consider several other strategies, which we describe briefly. The *optimistic query strategy* computes an optimistic state $\mathbf{x}^o$ and queries (at the midpoint of the interval) the utility parameter in $\mathbf{x}^o$ with the largest gap. Intuitively, an optimistic $\mathbf{x}^o$ is a useful adversarial choice, so refining information about it can help reduce regret. The *pessimistic query strategy* is analogous, relying on the intuition that a pessimistic choice is useful in preventing the adversary from making us regret our decision too much. The *optimistic-pessimistic* (*OP*) *strategy* combines the two intuitions: it chooses the parameter with largest gap among both states. These strategies are computationally appealing since they require solving only a standard COP, not a full-fledged minimax optimization.[19]

The *most uncertain state* (*MUS*) *strategy* is a variant of HLG that accounts for feasibility: we compute a most uncertain state $\mathbf{x}^{mu}$ and query (at the midpoint) the parameter in $\mathbf{x}^{mu}$ with the largest gap. Finally, the *second-best* (*SB*) *strategy* is based on the following intuition: suppose we have the optimistic state $\mathbf{x}^o$ and the second-best optimistic state $\mathbf{x}^{2o}$ (i.e., that state with the second-highest upper bound—this is computable with a single optimization). If we could ask a query which reduced the upper bound utility of $\mathbf{x}^o$ to lower than that of $\mathbf{x}^{2o}$, we ensure that regret is reduced (since the adversary can no longer attain this most optimistic value); if the lower bound of $\mathbf{x}^o$ were raised to the level of $\mathbf{x}^{2o}$'s upper bound, then we could terminate—knowing that $\mathbf{x}^o$ is *optimal*. Thus we would like to query $\mathbf{x}^o$ at $\mathbf{x}^{2o}$'s upper bound: a negative response will reduce regret, a positive response ensures $\mathbf{x}^o$ is optimal. Unfortunately, this cannot be implemented directly, since we can only query local parameters, but the strategy can be approximated for factored models by "distributing" this query across the different parameters and asking a set of queries.

The *myopically optimal* (*MY*) *strategy* computes the average regret reduction of the midpoint query for *each* utility parameter by solving the minimax optimization problem for each response to each query; it then asks the query with

[18] The bound is not generally tight if there is overlap in factors. But the bound is tight if *maxspan* is defined to account for logical consistency.

[19] Even termination can be determined heuristically, for example, by computing the max regret of the optimistic state after each query, or doing minimax optimization after every q queries.

the largest regret reduction averaged over both possible answers, *yes* and *no*. For large problems, this approach is computationally infeasible, but we test it on small problems to see how the other methods compare.[20]

5.5. Empirical results

To test the effectiveness of the various query strategies, we ran a series of elicitation experiments on a variety of problems. For each problem we tested the following elicitation strategies: halve largest gap (HLG), current solution (CS), current solution with a computation-time bound of five seconds per query (CS-5), optimistic-pessimistic (OP), second-best (SB), and most uncertain state (MUS). In addition, on problems small enough to permit it, we also compared these strategies to the much more computationally demanding myopically optimal method (MY).

We implemented the constraint generation approach outlined in Section 4.2 and used CPLEX 9.0 as the generic IP solver.[21] Our experiments considered two realistic domains—car rental and real estate—as well as randomly generated synthetic problems, as described in Section 4.3, with a factored structure sufficient to admit practical solution.

First, we experimented with a set of small synthetic problems. We did this to allow comparison of all of our proposed heuristics with the computationally demanding MY strategy. Fig. 9 reports the average minimax regret over 45 small synthetic problems constructed by randomly setting the utility bounds and the variables on which each utility factor depends. Each problem has ten attributes that can take at most four values and ten factors that depend on at most three attributes. We simulate user responses by drawing a random utility function u for each trial, consistent with the bounds, representing a specific user's preferences. Responses to queries are generated using u, assuming that the user accurately answers all queries relative to the specific utility function u.

Results are shown for two cases: first, for utility parameters drawn from a uniform distribution over the corresponding interval; and second, for parameters drawn from a truncated Gaussian distribution centered at the midpoint of the corresponding interval and truncated at the endpoints of that interval. This second regime reflects the fact that, given some initial unquantified uncertainty about a utility parameter, a user is somewhat more likely to have a true parameter value nearer the middle of the range. However, this probabilistic information is used only to generate "simulated users", and is not exploited by the elicitation algorithms.[22]

In the case of both the uniform and truncated Gaussian distributions, we observe that the OP, CS and CS-5 elicitation strategies decrease minimax regret at a rate very close to MY. This suggests that OP, CS and CS-5 are computationally feasible, yet promising alternatives to the computationally prohibitive MY strategy.

We report on further experiments using all strategies except MY (excluded for computational reasons) with larger synthetic problems, the real-estate problem and the car-rental problem. All results are averaged over 45 trials and

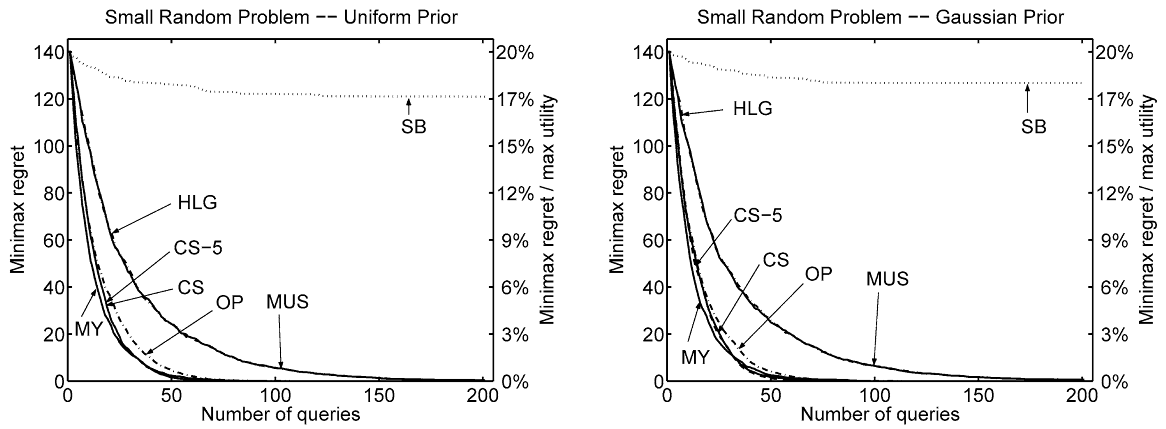
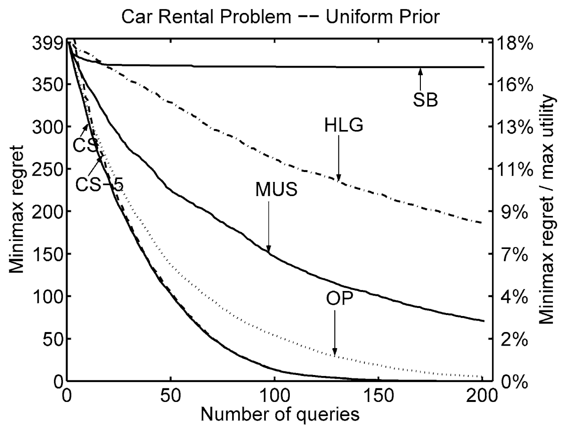


Fig. 9. Average max regret on small random problems (45 instances) as a function of number of queries given uniform (left) and Gaussian (right) distributed utilities.

[20] By doing lookahead of k stages of this type, we could in fact compute the optimal query plan of k-steps; however, doing so is infeasible for all but the smallest problems and small values of k.

[21] These simulations were performed on 3.0 GHz PCs.

[22] All experiments show a reasonably small variance so we exclude error bars for legibility.



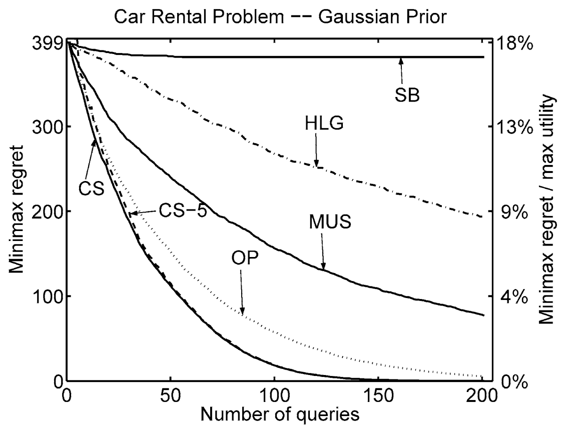


Fig. 10. Average max regret on car-rental problem (45 instances) as a function of number of queries given uniform (left) and Gaussian (right) distributed utilities.

use the same regime described above, involving both uniform and truncated Gaussian priors to generate users. Performance of the various query strategies on the car rental problem is depicted in Fig. 10, showing average minimax regret as a function of the number of queries. Initial utility bounds are set to give minimax regret of roughly 18% of the optimal solution.

Both CS and CS-5 perform extremely well: regret is reduced to almost zero within 160 queries on average. Though this may seem like a lot of queries, recall that the problem is itself large and the utility model has 150 parameters. We intentionally choose problems this large to push the computational boundaries of regret-based elicitation. Furthermore, while 160 queries may be large for typical consumer choice problems, it is more than reasonable for high stakes configuration applications. More importantly, these methods show excellent anytime performance: after only 80 queries, average minimax regret has dropped from 18% to under 2%.

Interestingly, the time bound of five seconds imposed by CS-5, while leading to approximately minimax optimal solutions, does not affect query quality: the approximate solutions give rise to queries that are virtually as effective as those generated by the optimal solutions. This demonstrates the importance of the anytime properties of our constraint generation procedure discussed in the previous section. The CS strategy requires on average 83 seconds per query, compared to the five seconds needed by CS-5. The OP strategy works very well too, and requires less computation time (0.1 s per query) since it does not need to solve minimax problems (except to verify termination "periodically", which is not reflected in the reported query computation time). However, both OP and CS-5 are fast enough to be used interactively on problems of this size. MUS, HLG, and SB do not work nearly as well, with SB essentially stalling because of the slow progress made in reducing the upper bounds of the optimistic state.

Note the HLG performs poorly since it fails to account for the feasibility of options, thus directing its attention to parts of utility space for which no product exists (hence polyhedral methods alone [29,48] will not offer reasonable elicitation in our setting). MUS significantly outperforms HLG for just this reason.

The real-estate problem was also tested, with query performance shown in Fig. 11, using the same regime as above. Again, both CS and CS-5 perform best, and the time bound of CS-5 has no effect on the quality of the CS strategy. Interestingly, OP performs almost identically to these, with somewhat lower computational cost.[23] Each of these methods reduces minimax regret from 40% of optimal to under 5% in about 120 queries. As above, SB fails to make progress, while HLG and MUS provide reasonable performance. Note that HLG requires no optimization nor any significant computation (except to test for termination).

Finally, we tested the query strategies on larger randomly generated problems (with 25 variables of domain size no more than four, and 20 utility factors with no more than three variables each). Results are shown in Fig. 12. The same performance patterns as in the real-estate problem emerge, with CS, CS-5 and OP all performing much better than the others. Although OP performs slightly better than CS/CS-5, the difference is not statistically significant.

[23] CS takes 14 seconds per query, CS-5 takes five seconds, and OP 0.1 seconds. Though we haven't experimented with this, we expect CS would work equally well on this problem with a much tighter time bound than five seconds.

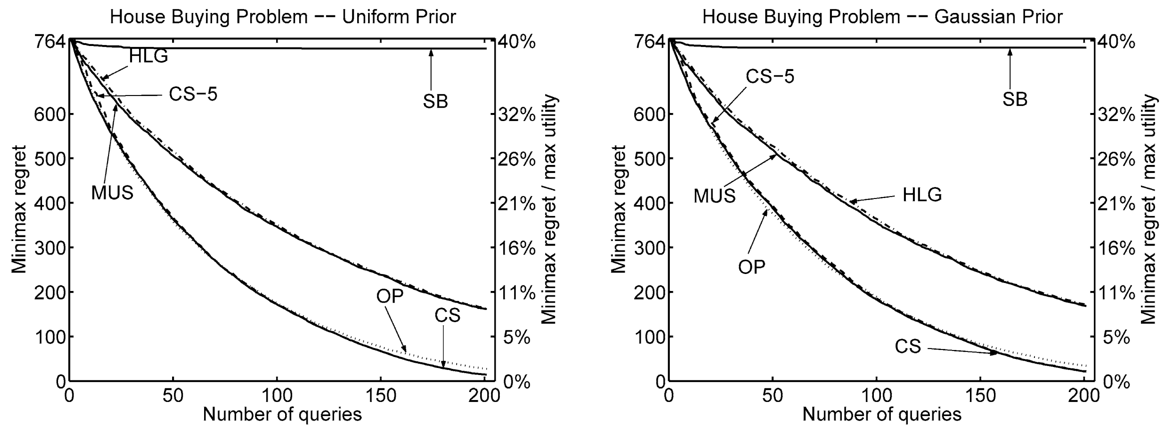


Fig. 11. Average max regret on real-estate problem (45 instances) as a function of number of queries given uniform (left) and Gaussian (right) distributed utilities.

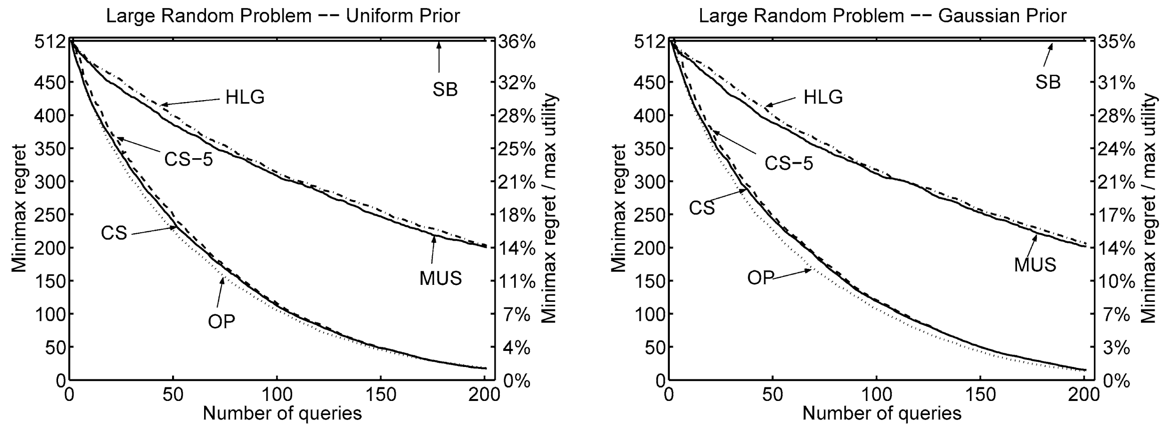


Fig. 12. Average max regret on large random problems (45 instances) as a function of number of queries given uniform (left) and Gaussian (right) distributed utilities.

5.6. Comparison queries

Comparison queries provide a natural alternative to bound queries in many situations. A comparison query takes the form "Do you prefer **x** to **x**′?" A positive response implies that $u_{\mathbf{x}} > u_{\mathbf{x}'}$. If the utility model is factored, this corresponds to the following linear constraint:

$$\sum_k u_{\mathbf{x}[k]} > \sum_k u_{\mathbf{x}'[k]}.$$

A negative response imposes the complementary constraint.

Given a collection of linear constraints $\mathcal{U}$ imposed by responses to a sequence of comparison queries, the minimax optimal decision can be computed using the constraint generation procedure described in Section 4.4. Our generic elicitation algorithm can then be used to ask specific comparison queries until minimax regret reaches an acceptable level. Though we do not experiment with specific comparison query strategies here, we expect that a modification of the current solution (CS) strategy proposed for bound queries would work especially well with comparison queries. More precisely, suppose that given the current constraints $\mathcal{U}$, the minimax optimal solution is computed to be $\mathbf{x}^*$ with adversarial witness $\mathbf{x}^w$. The CS query strategy for comparison queries requires that we ask the user to compare $\mathbf{x}^*$ and $\mathbf{x}^w$. Should the user prefer $\mathbf{x}^*$, this rules out the adversary's chosen utility function from the feasible set $\mathbf{U}$, thus ensuring a reduction in the pairwise regret $R(\mathbf{x}^*, \mathbf{x}^w, \mathbf{U})$ to zero, and usually reducing minimax regret as well. If $\mathbf{x}^w$ is preferred, this does not rule out the adversary's chosen utility function, nor is it guaranteed to reduce regret, but generally imposes a fairly strong constraint on **U**. Given the success of this strategy with respect to bound queries, and the success of related strategies in other domains [13,14], we expect the CS strategy to perform quite well.

Unlike the case of bound queries, where it is quite clear (due to the focus on gaps in specific parameters) that a user cannot provide a response that is inconsistent with prior responses, it is not obvious that a user cannot be inconsistent in responding to comparison queries. However, the CS strategy for bound queries does indeed ensure consistency. Unless minimax regret is zero (in which case the process would terminate), there must be some utility function in the current feasible set $\mathcal{U}$ for which $\mathbf{x}^w$ is preferred to $\mathbf{x}^*$. Furthermore, there must be some utility function for which $\mathbf{x}^*$ is preferred to $\mathbf{x}^w$; otherwise, $\mathbf{x}^w$ would have regret no greater that of $\mathbf{x}^*$ under all $u \in \mathcal{U}$, and would thus be minimax optimal as well (also implying that, since it is the witness, that minimax regret is zero).

6. Concluding remarks

Preference elicitation techniques for constraint-based optimization problems are critical to the development of interactive decision software. We have begun to address several important issues in this regard, specifically, how one should make decisions in the presence of (non-probabilistic) utility function uncertainty, and elicitation strategies that improve decision quality with minimal interaction. We have developed techniques for computing minimax optimal decisions in constraint-based decision problems when a user's utility function is only partially specified in the form of upper and lower bounds on utility parameters, or arbitrary linear constraints on such parameters. While the corresponding optimizations are potentially complex, we derived methods whereby they could be solved effectively using a sequence of MIPs. Furthermore, we showed how structure in the utility model could be exploited to ensure that the resulting IPs are compact or could be solved using an effective constraint generation procedure. Experiments with utility uncertainty specified by parameter bounds demonstrated the practicality of these techniques.

We also developed a number of query strategies for eliciting bounds on the parameters of utility models for the purpose of solving imprecise COPs. The most promising of these strategies, CS and OP, perform extremely well, requiring very few queries (relative to the model size) to provide dramatic reductions in regret. We have shown that using approximation of minimax regret reduces interactive computation time to levels required for real-time response without a noticeable effect on the performance of CS. OP also can be executed in real-time, since it does not require the same intensive minimax computation.

There are a number of directions in which this work can be extended. For example, the use of search and constraint-propagation methods for solving the COPs associated with computing minimax regret is of great interest. Our goal in this paper was to provide a precise formulation of these computational problems as integer programs and use off-the-shelf software to solve them. We expect that constraint-based optimization techniques that are specifically directed toward these problems should prove fruitful. Along these lines, we hope to develop deeper connections to existing work on soft constraints, valued-CSPs, and related frameworks.

Experimental validation of our suggested approach for comparison queries is an important next step, as is the development of new query strategies. In practice, we can often assume or develop prior distributional information over utilities. Rather than asking queries at midpoints of intervals, we could optimize the query point using probabilistic (value of information) computation, while using (distribution-free) regret to make decisions [50]. We are quite interested in the possibility of integrating Bayesian methods for reasoning about uncertain utility functions [10,17,27] with the constraint-based representation of the decision space. Finally, while optimal (non-myopic) strategies could be found (in principle) by solving prohibitively large continuous MDPs, it would nevertheless be interesting to explore non-myopic heuristics, and investigate the extent to which lookahead information can improve the reduction in minimax regret.

Naturally, we would like to consider additional query types, as well as alternative means for structuring outcomes and interactions to ease the cognitive burden on users. Developing means to improve robustness of our methods to the types of errors users typically make is also critical if tools such as those proposed here are to find widespread use. As such, user studies with our models will be required in order to assess the naturalness of such interaction models and to further refine our techniques to make them more understandable and intuitive for users.

An important question left unaddressed is that of eliciting the structure of a GAI model. Our work here assumes that the GAI factorization has been given and elicits only parameters of this model. We are currently exploring the application of techniques from decision analysis and elicitation of graphical models to the automated elicitation of GAI model structure.

Acknowledgements

This research was supported by the Institute for Robotics and Intelligent Systems (IRIS) and the Natural Sciences and Engineering Research Council (NSERC). Poupart was supported by a scholarship provided by Precarn Incorporated through IRIS. Thanks to the anonymous referees for their helpful suggestions.

References

[1] I. Averbakh, Minmax regret solutions for minimax optimization problems with uncertainty, Operations Research Letters 27 (2000) 57–65.

[2] I. Averbakh, V. Lebedev, On the complexity of minmax regret linear programming, European Journal of Operational Research 160 (1) (2005) 227–231.

[3] F. Bacchus, A. Grove, Graphical models for preference and utility, in: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI-95), Montreal, 1995, pp. 3–10.

[4] D.E. Bell, Regret in decision making under uncertainty, Operations Research 30 (1982) 961–981.

[5] A. Ben-Tal, A. Nemirovski, Robust solutions of uncertain linear programs, Operations Research Letters 25 (1999) 1–13.

[6] J.F. Benders, Partitioning procedures for solving mixed-variables programming problems, Numerische Mathematik 4 (1962) 238–252.

[7] U. Bertele, F. Brioschi, Nonserial Dynamic Programming, Academic Press, Orlando, FL, 1972.

[8] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint satisfaction and optimization, Journal of the ACM 44 (2) (1997) 201–236.

[9] J. Blythe, Visual exploration and incremental utility elicitation, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02), Edmonton, 2002, pp. 526–532.

[10] C. Boutilier, A POMDP formulation of preference elicitation problems, in: Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02), Edmonton, 2002, pp. 239–246.

[11] C. Boutilier, On the foundations of *expected* expected utility, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03), Acapulco, 2003, pp. 285–290.

[12] C. Boutilier, F. Bacchus, R.I. Brafman, UCP-Networks: A directed graphical representation of conditional utilities, in: Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI-01), Seattle, WA, 2001, pp. 56–64.

[13] C. Boutilier, R. Das, J.O. Kephart, G. Tesauro, W.E. Walsh, Cooperative negotiation in autonomic systems using incremental utility elicitation, in: Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI-03), Acapulco, 2003, pp. 89–97.

[14] C. Boutilier, T. Sandholm, R. Shields, Eliciting bid taker non-price preferences in (combinatorial) auctions, in: Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04), San Jose, CA, 2004, pp. 204–211.

[15] D. Braziunas, C. Boutilier, Local utility elicitation in GAI models, in: Proceedings of the Twenty-first Conference on Uncertainty in Artificial Intelligence (UAI-05), Edinburgh, 2005, pp. 42–49.

[16] U. Chajewska, L. Getoor, J. Norman, Y. Shahar, Utility elicitation as a classification problem, in: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98), Madison, WI, 1998, pp. 79-88.

[17] U. Chajewska, D. Koller, R. Parr, Making rational decisions using adaptive utility elicitation, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00), Austin, TX, 2000, pp. 363–369.

[18] V. Chandru, J.N. Hooker, Optimization Methods for Logical Inference, Wiley, New York, 1999.

[19] R. Dechter, Bucket elimination: A unifying framework for probabilistic inference, in: Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI-96), Portland, OR, 1996, pp. 211–219.

[20] R. Dechter, Constraint Processing, Morgan Kaufmann, San Francisco, CA, 2003.

[21] J.S. Dyer, Interactive goal programming, Management Science 19 (1972) 62–70.

[22] P.C. Fishburn, Interdependence and additivity in multivariate, unidimensional expected utility theory, International Economic Review 8 (1967) 335–342.

[23] P.C. Fishburn, Utility Theory for Decision Making, Wiley, New York, 1970.

[24] S. French, Decision Theory, Halsted Press, New York, 1986.

[25] C. Gonzales, P. Perny, GAI networks for utility elicitation, in: Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR2004), Whistler, BC, 2004, pp. 224–234.

[26] C. Guestrin, D. Koller, R. Parr, Max-norm projections for factored MDPs, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01), Seattle, WA, 2001, pp. 673–680.

[27] H.A. Holloway, C.C. White III, Question selection for multiattribute decision-aiding, European Journal of Operational Research 148 (2003) 525–543.

[28] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, K. Rommelse, The lumiere project: Bayesian user modeling for inferring goals and needs of software users, in: Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98), Madison, WI, 1998, pp. 256–265.

[29] V.S. Iyengar, J. Lee, M. Campbell, Q-Eval: Evaluating multiple attribute items using queries, in: Proceedings of the Third ACM Conference on Electronic Commerce, Tampa, FL, 2001, pp. 144–153.

[30] R.L. Keeney, H. Raiffa, Decisions with Multiple Objectives: Preferences and Value Trade-offs, Wiley, New York, 1976.

[31] J.A. Konstan, B.N. Miller, D. Maltz, J.L. Herlocker, L.R. Gordon, J. Riedl, Grouplens: Applying collaborative filtering to usenet news, Communications of the ACM 40 (3) (1997) 77–87.

[32] P. Kouvelis, G. Yu, Robust Discrete Optimization and Its Applications, Kluwer, Dordrecht, 1997.

[33] G. Lee, S. Bauer, P. Faratin, J. Wroclawski, Learning user preferences for wireless services provisioning. in: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-04), New York, 2004, pp. 480–487.
[34] G. Loomes, R. Sugden, Regret theory: An alternative theory of rational choice under uncertainty, Economic Journal 92 (1982) 805–824.
[35] A.M. Mármol, J. Puerto, F.R. Fernández, The use of partial information on weights in multicriteria decision problems, Journal of Multicriteria Decision Analysis 7 (1998) 322–329.
[36] G.L. Nemhauser, L.A. Wolsey, Integer Programming and Combinatorial Optimization, Wiley, New York, 1988.
[37] B. O'Sullivan, E. Freuder, S. O'Connell, Interactive constraint acquisition, in: CP-2001 Workshop on User Interaction in Constraint Processing, Paphos, Cyprus, 2001.
[38] P. Pu, B. Faltings, Decision tradeoff using example-critiquing and constraint programming, Constraints 9 (4) (2004) 289–310.
[39] P. Pu, B. Faltings, M. Torrens, User-involved preference elicitation, in: IJCAI-03 Workshop on Configuration, Acapulco, 2003.
[40] J.C. Quiggan, Stochastic dominance in regret theory, The Review of Economic Studies 57 (3) (1990) 503–511.
[41] F. Rossi, A. Sperduti, K.B. Venable, L. Khatib, P.H. Morris, R.A. Morris, Learning and solving soft temporal constraints: An experimental study, in: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming, Ithaca, NY, 2002, pp. 249–263.
[42] D. Sabin, R. Weigel, Product configuration frameworks—a survey, IEEE Intelligent Systems and their Applications 13 (4) (1998) 42–49.
[43] A. Salo, R.P. Hämäläinen, Preference ratios in multiattribute evaluation (PRIME)—elicitation and decision procedures under incomplete information, IEEE Transaction on Systems, Man and Cybernetics 31 (6) (2001) 533–545.
[44] L.J. Savage, The Foundations of Statistics, Wiley, New York, 1954.
[45] L.J. Savage, The theory of statistical decision, Journal of the American Statistical Association 46 (1986) 55–67.
[46] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: Hard and easy problems, in: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, 1995, pp. 631–637.
[47] D. Schuurmans, R. Patrascu, Direct value approximation for factored MDPs, in: Advances in Neural Information Processing Systems 14 (NIPS-2001), Vancouver, 2001, pp. 1579–1586.
[48] O. Toubia, J. Hauser, D. Simester, Polyhedral methods for adaptive choice-based conjoint analysis, Technical Report 4285-03, Sloan School of Management, MIT, Cambridge, 2003.
[49] J. von Neumann, O. Morgenstern, Theory of Games and Economic Behavior, Princeton University Press, Princeton, NJ, 1944.
[50] T. Wang, C. Boutilier, Incremental utility elicitation with the minimax regret decision criterion, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03), Acapulco, 2003, pp. 309–316.
[51] M. Weber, Decision making with incomplete information, European Journal of Operational Research 28 (1987) 44–57.
[52] C.C. White III, A.P. Sage, S. Dozono, A model of multiattribute decisionmaking and trade-off weight determination under uncertainty, IEEE Transactions on Systems, Man and Cybernetics 14 (2) (1984) 223–229.

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Artificial Intelligence 170 (2006) 714–738

Artificial Intelligence

www.elsevier.com/locate/artint

Cut-and-solve: An iterative search strategy for combinatorial optimization problems

Sharlee Climer *, Weixiong Zhang

Department of Computer Science and Engineering, Washington University, One Brookings Drive, St. Louis, MO 63130-4899, USA

Received 26 August 2005; received in revised form 13 February 2006; accepted 23 February 2006

Available online 17 April 2006

Abstract

Branch-and-bound and *branch-and-cut* use search trees to identify optimal solutions to combinatorial optimization problems. In this paper, we introduce an iterative search strategy which we refer to as *cut-and-solve* and prove optimality and termination for this method. This search is different from traditional tree search as there is no branching. At each node in the search path, a relaxed problem and a sparse problem are solved and a constraint is added to the relaxed problem. The sparse problems provide incumbent solutions. When the constraining of the relaxed problem becomes tight enough, its solution value becomes no better than the incumbent solution value. At this point, the incumbent solution is declared to be optimal. This strategy is easily adapted to be an *anytime* algorithm as an incumbent solution is found at the root node and continuously updated during the search.

Cut-and-solve enjoys two favorable properties. Since there is no branching, there are no "wrong" subtrees in which the search may get lost. Furthermore, its memory requirement is negligible. For these reasons, it has potential for problems that are difficult to solve using depth-first or best-first search tree methods.

In this paper, we demonstrate the cut-and-solve strategy by implementing a generic version of it for the Asymmetric Traveling Salesman Problem (ATSP). Our unoptimized implementation outperformed state-of-the-art solvers for five out of seven real-world problem classes of the ATSP. For four of these classes, cut-and-solve was able to solve larger (sometimes substantially larger) problems. Our code is available at our websites.

Keywords: Search strategies; Branch-and-bound; Branch-and-cut; Anytime algorithms; Linear programming; Traveling Salesman Problem

1. Introduction

Life is full of optimization problems. We are constantly searching for ways to minimize cost, time, energy, or some other valuable resource, or maximize performance, profit, production, or some other desirable goal, while satisfying the constraints that are imposed on us. Optimization problems are interesting as there are frequently a very large number of *feasible* solutions that satisfy the constraints; the challenge lies in searching through this vast solution space and identifying an optimal solution. When the number of solutions is too large to explicitly look at each one, two search strategies, *branch-and-bound* [4] and *branch-and-cut* [28], have been found to be exceptionally useful.

* Corresponding author.
E-mail addresses: sharlee@climer.us (S. Climer), zhang@cse.wustl.edu (W. Zhang).

0004-3702/$ – see front matter © 2006 Published by Elsevier B.V.
doi:10.1016/j.artint.2006.02.005

Branch-and-bound uses a search tree to pinpoint an optimal solution. (Note there may be more than one optimal solution.) If the entire tree were generated, every feasible solution would be represented by at least one leaf node. The search tree is traversed and a relaxed variation of the original problem is solved at each node. When a solution to the relaxed subproblem is also a feasible solution to the original problem, it is made the *incumbent* solution. As other solutions of this type are found, the incumbent is updated as needed so as to always retain the best feasible solution found thus far. When the search tree is exhausted, the current incumbent is returned as an optimal solution.

If the number of solutions is too large to allow explicitly looking at each one, then the search tree is also too large to be completely explored. The power of branch-and-bound comes from its *pruning* rules, which allow pruning of entire subtrees while guaranteeing optimality. If the search tree is pruned to an adequately small size, the problem can be solved to optimality.

Branch-and-cut improves on branch-and-bound by increasing the probability of pruning. At some or all of the nodes, *cutting planes* [28] are added to tighten the relaxed subproblem. These cutting planes remove a set of solutions for the relaxed subproblem. However, in order to ensure optimality, these cutting planes are designed to never exclude any feasible solutions to the current unrelaxed subproblem.

While adding cutting planes can substantially increase the amount of time spent at each node, these cuts can dramatically reduce the size of the search tree and have been used to solve a great number of problems that were previously insoluble.

Branch-and-bound and branch-and-cut are typically implemented in depth-first fashion due to its linear space requirement and other favorable features [49]. However, depth-first search can suffer from the problem of exploring subtrees with no optimal solution, resulting in a large search cost. A wrong choice of a subtree to explore in an early stage of a depth-first search is usually difficult to rectify. The Artificial Intelligence community has invested a great deal of effort in addressing this issue. Branching techniques [4], heuristics investigations [42], and search techniques such as limited discrepancy [24] and randomization and restarts [19] have been developed in an effort to combat this persistent problem.

In this paper, we introduce an iterative search strategy which overcomes the problem of making wrong choices in depth-first branch-and-bound, while keeping memory requirements nominal. We refer to this search strategy as *cut-and-solve* and demonstrate it on integer linear programs. Being an iterative strategy, there is no search tree, only a search path that is directly traversed. In other words, there is only one child for each node, so there is no need to choose which child to traverse next. At each node in the search path, two relatively easy subproblems are solved. First, a relaxed solution is found. Then a *sparse* problem is solved. Instead of searching for an optimal solution in the vast solution space containing every feasible solution, a very sparse solution space is searched. An incumbent solution is found at the first node and updated as needed at subsequent nodes. When the search terminates, the current incumbent solution is guaranteed to be an optimal solution. In this paper, we prove optimality and termination of the cut-and-solve strategy.

The paper is organized as follows. In the next section, branch-and-bound and branch-and-cut are discussed in greater detail. In the following section, the cut-and-solve strategy is described and compared with these prevalent techniques. Next, we illustrate this strategy by applying it to a simple linear programming problem. Then a generic procedure for using cut-and-solve is presented. This generic procedure is demonstrated by implementing an algorithm for the Asymmetric Traveling Salesman Problem (ATSP). (The ATSP is the NP-hard problem of finding a minimum-cost Hamiltonian cycle for a set of cities in which the cost from city i to city j may not necessarily be equal to the cost from city j to city i.) We have tested a preliminary, unoptimized implementation of this algorithm and compared it with branch-and-bound and branch-and-cut solvers. Our tests show that cut-and-solve is not always as fast as these state-of-the-art solvers for relatively simple problem instances. However, it is faster than these solvers on the largest instances for five out of seven real-world problem classes, and solves larger (sometimes substantially larger) instances for four of these classes. This paper is concluded with a discussion of this technique and related work. A preliminary version of this paper appeared in [11].

2. Background

In this section, we define several terms and describe branch-and-bound and branch-and-cut in greater detail, using the Asymmetric Traveling Salesman Problem (ATSP) as an example.

Branch-and-bound and branch-and-cut have been used to solve a variety of optimization problems. The method we present in this paper can be applied to any such problem. However, to make our discussion concrete, we will narrow our focus to integer linear programs (IPs). An IP is an optimization problem that is subject to a set of linear constraints. IPs have been used to model a wide variety of problems, including Traveling Salesman Problems (TSP) [22,36], Constraint Satisfaction Problems (CSP) [15], and network optimization problems [45]. Moreover, a wealth of problems can be cast as one of these more general problems. TSP applications include a vast number of scheduling, routing, and planning problems such as the no-wait flowshop, stacker crane, tilted drilling machine, computer disk read head, robotic motion, and pay phone coin collection problems [30]. Furthermore, the TSP can be used to model surprisingly diverse problems, such as the shortest common superstring problem, which is of interest in genetics research. CSPs are used to model configuration, design, diagnosis, spatio-temporal reasoning, resource allocation, graphical interfaces, and scheduling problems [15]. Finally, examples of network optimization problems include delay-tolerant network routing, cellular radio network base station locations, and the minimum-energy multicast problem in wireless ad hoc networks. There exist interesting research problems that can be cast as IPs in virtually every field of computer science. Furthermore, there are a staggering number of commercial applications that can be cast as IPs.

A general IP can be written in the following form:

$$Z = \min \text{ (or max)} \sum_i c_i x_i \tag{1}$$

$$\text{subject to:} \quad \text{a set of linear constraints} \tag{2}$$

$$x_i \in I \tag{3}$$

where the c_i values are instance-specific constants and the set of x_i represents the *decision variables*. Constraints (2) are linear equalities or inequalities composed of constants, decision variables, and possibly some auxiliary variables. Constraints (3) enforce integrality of the decision variables. A *feasible* solution is one that satisfies all of the constraints (2) and (3). The set of all feasible solutions is the *solution space*, *SS*, for the problem. The solution space is defined by the given problem. In contrast, the search space is defined by the algorithm used to solve the problem. An *optimal* solution is a feasible solution with the least (or greatest) value, as defined by the *objective function* (1).

Linear programs (LPs) are similar to IPs except the former allow real numbers for the decision variables. A general LP has the same form as the general IP given above with the omission of constraints (3). LPs can be solved in polynomial time using the ellipsoid method [33]. However, in practice the simplex method [13] is commonly used, despite its exponential worst-case running time [31]. The simplex algorithm moves along the edges of the polyhedron defined by the constraints, always in a direction that improves the solution. Usually this approach is very effective.

In contrast, most IPs cannot be solved in polynomial time in the worst case. For example, the ATSP is a NP-hard problem and can be defined by the following IP:

$$ATSP(G) = \min\left(\sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}\right) \tag{4}$$

subject to:

$$\sum_{i \in V} x_{ij} = 1, \quad \forall j \in V \tag{5}$$

$$\sum_{j \in V} x_{ij} = 1, \quad \forall i \in V \tag{6}$$

$$\sum_{i \in W} \sum_{j \in W} x_{ij} \leqslant |W| - 1, \quad \forall W \subset V,\ W \neq \emptyset \tag{7}$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j \in V \tag{8}$$

for directed graph $G = (V, A)$ with vertex set $V = \{1, \ldots, n\}$ (where n is the number of cities), arc set $A = \{(i, j) \mid i, j = 1, \ldots, n\}$, and cost matrix $c_{n \times n}$ such that $c_{ij} \geqslant 0$ and $c_{ii} = \infty$ for all i and j in V. Each decision variable, x_{ij}, corresponds to an arc (i, j) in the graph. Constraints (8) require that either an arc (i, j) is traversed (x_{ij} is equal to 1) or is not traversed (x_{ij} is equal to 0). Constraints (5) and (6) require that each city is entered exactly once and departed from exactly once. Constraints (7) are called *subtour elimination constraints* (SECs) as they require that no more than one cycle can exist in the solution. Note that there are an exponential number of SECs. It is common practice to initially omit these constraints and add only those that are necessary as the problem is solved. Finally, the objective function (4) requires that the sum of the costs of the traversed arcs is minimized. In this problem, the solution space is the set of all permutations of the cities and contains $(n-1)!$ distinct solutions.

2.1. Using bounds

Without loss of generality, we only discuss minimization problems in the remainder of this paper.

An IP can be *relaxed* by relaxing one or more of the constraints. This relaxation is a *lower-bounding* modification as an optimal solution for the relaxation cannot exceed the optimal solution of the original problem. Furthermore, the solution space of the relaxed problem, SS_r, contains the entire solution space of the original problem, SS_o, however, the converse is not necessarily true.

An IP can be *tightened* by tightening one or more of the constraints or adding additional constraints. This tightening is an *upper-bounding* modification as an optimal solution for the tightened problem cannot have a smaller value than the optimal solution of the original problem. Furthermore, the solution space of the original problem, SS_o, contains the solution space of the tightened problem, SS_t, however, the converse is not necessarily true. In summary, $SS_t \subseteq SS_o \subseteq SS_r$.

For example, the ATSP can be relaxed by completely omitting constraints (7). This relaxation allows any number of subtours to exist in the solution. This relaxed problem is simply the Assignment Problem (AP) [39]. The AP is the problem of finding a minimum-cost matching on a bipartite graph constructed by including all of the arcs and two nodes for each city, where one node is used for the tail of all its outgoing arcs and one is used for the head of all its incoming arcs. Fig. 1 depicts a solution for the AP relaxation of a 49-city symmetric TSP (STSP). STSPs are a special class in which the cost from city i to city j is equal to the cost from city j to city i for all cities i and j.

Another relaxation can be realized by relaxing the integrality requirement of constraints (8). This can be accomplished by replacing constraints (8) with the following constraints:

$$0 \leqslant x_{ij} \leqslant 1, \quad \forall i, j \in V \tag{9}$$

This relaxation transforms the IP into an LP and is referred to as the *Held–Karp* relaxation [25,26]. Fig. 2 depicts a solution for the Held–Karp relaxation of the 49-city STSP. At first glance it appears that subtour elimination constraints are also violated. However, constraints (7) are not violated due to the fact that some of the x_{ij} variables have values less than one.

One way the ATSP can be tightened is by adding constraints that set the values of selected decision variables. For example, adding $x_{ij} = 1$ forces the arc (i, j) to be included in all solutions.

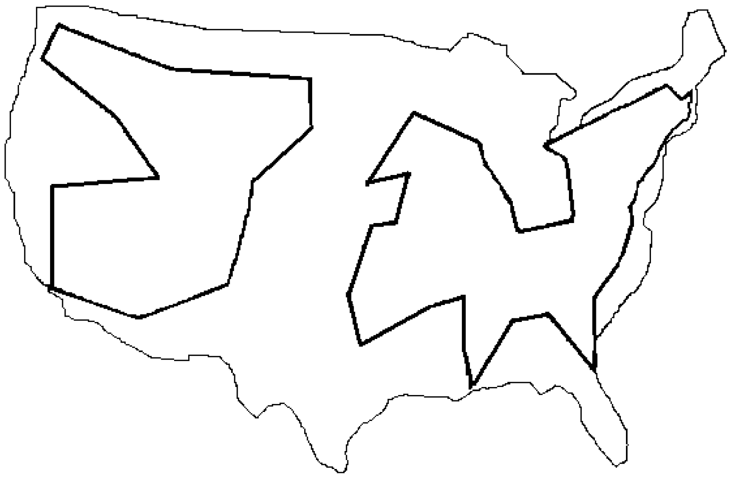


Fig. 1. AP relaxation of a 49-city TSP. Two subtours occurred in this solution.

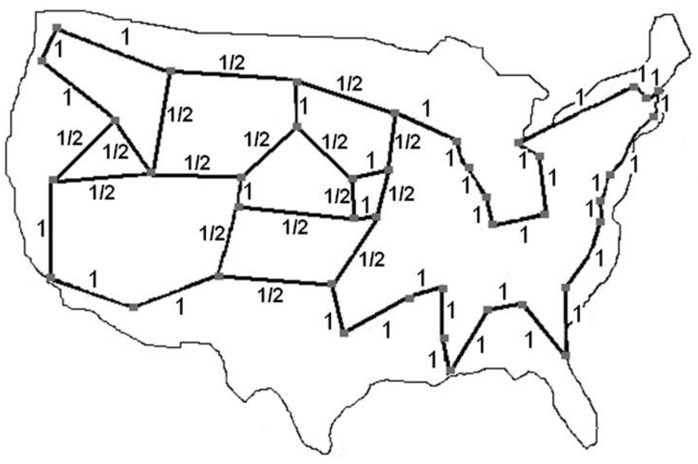


Fig. 2. Held–Karp relaxation of a 49-city TSP. The values shown on the arcs are their corresponding x_{ij} values. This example appeared in [14].

2.2. Branch-and-bound search

In 1958, several papers appeared that used branch-and-bound search [5,12,16,43]. This method organizes the search space into a tree structure. At each level of the tree, branching rules are used to generate child nodes. The children are tightened versions of their parents as constraints are added that set values of decision variables. Each node inherits all of the tightening constraints that were added to its ancestors. These tightened problems represent subproblems of the parent problem and the tightening may reduce the size of their individual solution spaces.

At each node a relaxation of the original problem is solved. This relaxation may enlarge the size of the node's solution space. Thus, at the root node, a relaxation of the problem is solved. At every other node, a doubly-modified problem is solved; one that is simultaneously tightened and relaxed. The solution space of one of these doubly-modified problems may contain extra solutions that are not in the solution space of the original problem and may be missing solutions from the original problem as illustrated by the following example.

Consider the Carpaneto, Dell'Amico, and Toth (CDT) implementation of branch-and-bound search for the ATSP [7] as depicted in Fig. 3. For this algorithm, the AP is used for the relaxation, allowing any number of subtours in the solution. The branching rule dictates forced inclusions and exclusions of arcs. Arcs that are not forced in this way are referred to as *free* arcs. The branching rule selects the subtour in the AP solution that has the fewest free arcs and each child node forces the exclusion of one of these free arcs. Furthermore, each child node after the first forces the inclusion of the arcs excluded by their elder siblings. More formally, given a parent node, let E denote its set of excluded arcs, I denote its set of included arcs, and $\{a_1, \ldots, a_t\}$ be the free arcs in the selected subtour. In this case, t children would be generated with the kth child having $E_k = E \cup \{a_k\}$ and $I_k = I \cup \{a_1 \ldots a_{k-1}\}$. Thus, child k is tightened by adding the constraints that the decision variables for the arcs in E are equal to zero and those for the arcs in I are equal to one. When child k is processed, the AP is solved with these additional constraints. The solution space of this doubly-modified problem is missing all of the tours containing an arc in E_k and all of the tours in which any arc in I_k is absent. However, it is enlarged by the addition of all the AP solutions that are not a single cycle, do not contain an arc in E_k, and contain all of the arcs in I_k.

The CDT algorithm is experimentally compared with cut-and-solve in the Results section of this paper.

2.3. Gomory cuts

In the late fifties, Gomory proposed an iterative search strategy in which *cutting planes* were systematically derived and applied to a relaxed problem [20]. An example of a cutting plane follows. Assume we are given three binary decision variables, x_1, x_2, x_3, a constraint $10x_1 + 16x_2 + 12x_3 \leqslant 20$, and the integrality relaxation ($0 \leqslant x_i \leqslant 1$ is substituted for the binary constraints). It is observed that the following cut could be added to the problem: $x_1 + x_2 + x_3 \leqslant 1$ without removing any of the solutions to the original problem. However, solutions would be removed from the relaxed problem (such as $x_1 = 0.5$, $x_2 = 0.25$, and $x_3 = 0.5$).

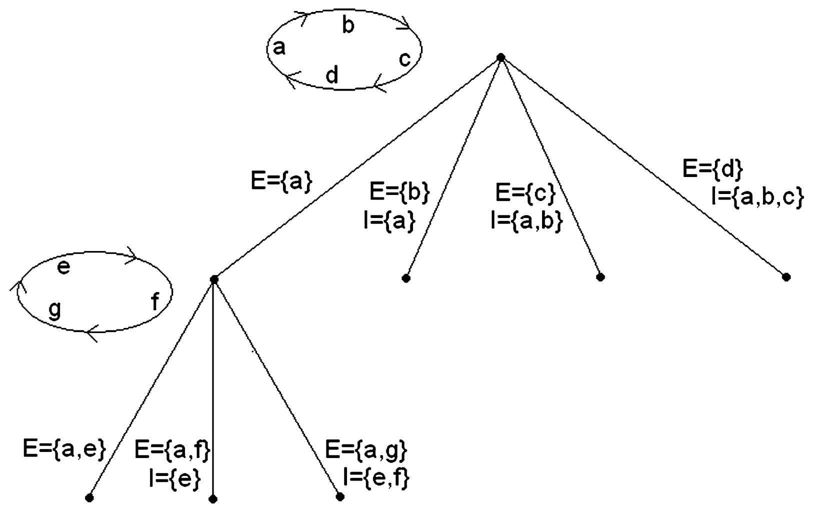


Fig. 3. An example of the first few nodes of a CDT search tree for an ATSP. The cycles shown are the subtours with the fewest free arcs for the AP solutions of the corresponding nodes. Arcs that are forced to be excluded are in sets E and arcs that are forced to be included are in sets I.

Gomory cuts tighten the relaxed problem by removing part of its solution space. These cuts do not tighten the original (unrelaxed) problem, as none of the solutions to the original problem are removed. However, the removal of relaxed solutions tends to increase the likelihood that the next relaxed solution found is also a solution to the unrelaxed problem. Such solutions may be used to establish or update the incumbent solution. Cuts are added and relaxations are solved iteratively until the tightening on the relaxed problem becomes so constrictive that its solution is equal to the current incumbent. At this point, the search is terminated and the incumbent solution is declared optimal.

2.4. Branch-and-cut search

Branch-and-cut search is essentially branch-and-bound search with the addition of the application of cutting planes at some or all of the nodes. These cutting planes tighten the relaxed problem and increase the pruning potential in two ways. First, the value of the solution to this subproblem may be increased (and cannot be decreased) by this tightening. If such increase causes the value to be greater than or equal to the incumbent solution value, then the entire subtree can be pruned. Second, forcing out a set of the relaxed solutions may increase the possibility that a feasible solution to the unrelaxed problem is found. If this feasible solution has a value that is less than the current incumbent solution, it will replace the incumbent and increase future pruning potential.

The number of nodes at which cutting planes are applied is algorithm-specific. Some algorithms only apply cuts at the root node, while others apply cuts at many or all of the nodes.

`Concorde` [1,2] is an award-winning branch-and-cut algorithm designed for solving the symmetric TSP (STSP). This code has been used to solve many very large STSP instances, including a 24,978-city instance corresponding to cities in Sweden [2]. This success was made possible by the design of a number of clever cutting planes custom tailored for this problem.

2.5. Branch-and-bound and branch-and-cut design considerations

When designing an algorithm using branch-and-bound or branch-and-cut, a number of policies must be determined. These include determining a relaxation to be used and an algorithm for solving this relaxation, branching rules, and a search method, which determines the order in which the nodes are explored.

Since a relaxed problem is solved at every node, it must be substantially easier to solve than the original problem. However, it is desirable to use the tightest relaxation possible in order to increase the pruning capability.

Branching rules determine the structure of the search tree. They determine the depth and breadth of the tree. Since branching rules tighten the subproblem, strong rules can increase pruning potential.

Finally, a search method must be selected. *Best-first* search selects the node with the best heuristic value to be explored first. A* search is a best-first strategy that uses the sum of the cost of the path up to a given node and an estimate of the cost from this node to a goal node as the heuristic value [23]. This strategy ensures that the least number of nodes are explored for a given search tree and heuristic. Unfortunately, identifying the best current node requires storing all active nodes and even today's vast memory capabilities can be quickly exhausted. For this reason, *depth-first* search is commonly employed. While this strategy solves the memory problem and is asymptotically optimal [49], it introduces a substantial new problem. Heuristics used to guide the search can lead in the wrong direction, resulting in large subtrees being fruitlessly explored.

Unfortunately, even when a combination of policies is fine-tuned to get the best results, many problem instances remain insoluble. This is usually due to inadequate pruning. On occasion, the difficulty is due to the computational demands of solving the relaxed problem or finding cutting planes. For instance, the simplex method is commonly used for solving the relaxation for IPs, despite the fact that it has exponential worst-case performance.

3. Cut-and-solve search strategy

3.1. Basic algorithm

Unlike the cutting planes in branch-and-cut search, cut-and-solve uses cuts that intentionally cut out solutions from the original solution space. We use the term *piercing cut* to refer to a cut that removes at least one feasible solution from the original (unrelaxed) problem solution space. The cut-and-solve algorithm is presented in Fig. 4.[1]

In this algorithm, each iteration corresponds to one node in the search path. First, a piercing cut is selected. Let SS_{sparse} be the set of feasible solutions in the solution space removed by the cut. Then the best solution in SS_{sparse} is found. This problem tends to be relatively easy to solve as a sparse solution space, as opposed to the vast solution space of the original problem, is searched for the best solution. At the root node, this solution is referred to as the incumbent, or *best*, solution. If later iterations find a solution that is better than *best*, *best* is replaced by this new solution.

In the next step, the piercing cut is added to the IP. This piercing cut excludes all of the solutions in SS_{sparse} from the IP. Thus, the piercing cut tightens the IP and reduces the size of its solution space. The lower bound for this tightened IP is then found. If this lower bound is greater than or equal to *best*, then the search is terminated and *best* is an optimal solution.

At subsequent nodes, the process is repeated. The incumbent solution is updated as needed. The piercing cuts accumulate with each iteration. When the tightening due to these piercing cuts becomes constrictive enough, the solution to this doubly-modified problem will become greater than or equal to the incumbent solution value. When this occurs, the incumbent solution is returned as optimal.

Theorem 1. *When the cut-and-solve algorithm terminates, the current incumbent solution must be an optimal solution.*

Proof. The current incumbent is the optimal solution in the solution spaces that were cut away by the piercing cuts. The solution space of the final doubly-modified problem contains all of the solutions for the original problem except

```
algorithm cut_and_solve (IP)
     select cut
     find optimal feasible solution in space removed by cut
        update best if necessary
     add cut to problem
     find lower bound
     if (lower bound >= best) return best
     otherwise, repeat
```

Fig. 4. Cut-and-solve algorithm.

[1] We thank an anonymous reviewer for suggesting this pseudo code.

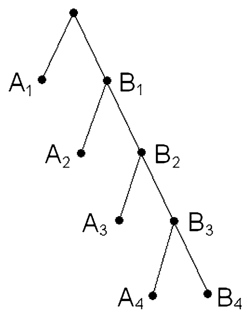


Fig. 5. Cut-and-solve depicted as a binary search tree. At each of the A_i nodes, the optimal solution is found for the small chunk of the solution space that is cut away by the corresponding piercing cut. At each of the B_i nodes, a relaxation is solved over the remaining solution space.

those that were cut away by the piercing cuts. If the relaxation of this reduced solution space has a value that is greater than or equal to the incumbent value, then this reduced solution space cannot contain a solution that is better than the incumbent. □

Termination of the algorithm is summarized in the following theorem:

Theorem 2. *If the solution space for the original problem, SS_o, is finite, and both the relaxation algorithm and the algorithm for selecting and solving the sparse problem are guaranteed to terminate, then the cut-and-solve algorithm is guaranteed to terminate.*

Proof. The number of nodes in the search path must be finite as a non-zero number of solutions are removed from SS_o at each node. Therefore there are a finite number of problems solved, each of which are guaranteed to terminate. □

Cut-and-solve can be cast as a binary search tree with highly disproportionate children as shown in Fig. 5. At each node, a relaxation is solved and a cut is chosen. This cut is used for the branching rule. The left child's solution space is small as it only contains the solutions that are cut away by the piercing cut. The right child has the huge solution space that is remaining after removing the left child's solutions. The left child's solution space is easily explored and produces potential incumbent solutions. It is immediately solved and the right child is subdivided into another set of disproportionate children. When the relaxed solution of the right child is greater than or equal to the current incumbent, it is pruned and the search is finished. Thus, the final configuration of the search tree is a single path at the far right side with a single branch to the left at each level. This search strategy is essentially linear: consistently solving the easy problems immediately and then redividing.

3.2. Using cut-and-solve as a complete anytime algorithm

The cut-and-solve algorithm is easily adapted to be an *anytime* algorithm [6]. Anytime algorithms allow the termination of an execution at any time and return the best approximate solution that has been found thus far. Many anytime algorithms are based on local search solutions. This type of anytime algorithm does not guarantee that the optimal solution will eventually be found and cannot determine whether the current solution is optimal. *Complete* anytime algorithms [35,46,47] overcome these two restrictions. These algorithms will find an optimal solution if given adequate time and optimality is guaranteed. Cut-and-solve finds an incumbent solution at the root node so there is an approximate solution available any time after the root node is solved. This solution improves until the optimum is found or the execution is terminated, making it a complete anytime algorithm.

Cut-and-solve offers a favorable benefit when used as an anytime solver. Many approximation algorithms are unable to provide a bound on the quality of the solution. This can be especially problematic when the result is to be used in further analysis and an estimated error needs to be propagated. Upon termination of cut-and-solve, the value of the most recently solved doubly-modified problem is a lower bound on the optimal solution, yielding the desired information. Furthermore, the rate of decrease of the gap between the incumbent and the doubly-modified problem solutions can be monitored and used to determine when to terminate the anytime solver.

3.3. Parallel processing

Cut-and-solve can be computed using parallel processing. Each of the sparse problems can be solved independently on different processors. It is not necessary that every sparse problem runs to completion. Whenever a sparse problem is solved, it is determined which iteration had the *smallest* lower bound that is greater than or equal to the solution of the sparse problem. It is only necessary that the sparse problems prior to that iteration run to completion to ensure optimality.

Consider the search depicted in Fig. 5. The solutions for the B_i nodes are non-decreasing as i increases. Suppose that A_4 completes first with a solution of x. Suppose further that B_1 has a solution that is less than x and B_2 has a solution that is equal to x. (In this case, B_2's solution can't be greater than x as it is a lower bound on the solution space that contains the solution found at A_4.) Only A_1 and A_2 need to run to completion and the search at A_3 can be terminated as it can't contain a solution that is better than x.

More generally, let y equal the cost of a solution found at a node A_p. Let B_q be the node with the smallest cost that is also greater than or equal to y. All of the searches at nodes A_i where $i > q$ can be terminated.

4. A simple example

In this section, we present a simple example problem and step through the search process using cut-and-solve. Consider the following IP:

$$\min Z = y - \frac{4}{5}x \tag{10}$$

subject to:

$$x \geqslant 0 \tag{11}$$

$$y \leqslant 3 \tag{12}$$

$$y + \frac{3}{5}x \geqslant \frac{6}{5} \tag{13}$$

$$y + \frac{13}{6}x \leqslant 9 \tag{14}$$

$$y - \frac{5}{13}x \geqslant \frac{1}{14} \tag{15}$$

$$x \in I \tag{16}$$

$$y \in I \tag{17}$$

This IP has only two decision variables, allowing representation as a 2D graph as shown in Fig. 6. Every x, y pair that is feasible must obey the constraints. Each of the first five linear constraints (11) to (15) corresponds to an edge of the polygon in Fig. 6. All of the feasible solutions must lie inside or on the edge of the polygon. Constraints (16) and (17) require that the decision variables assume integral values. Therefore, the feasible solutions for this IP are shown by the dots located inside and on the edge of the polygon.

The terms of the objective function (10) can be rearranged into the slope-intercept form as follows: $y = \frac{4}{5}x + Z$. Therefore, the objective function of this IP represents an infinite number of parallel lines with the slope of $\frac{4}{5}$. In this example, each value of Z is equal to its corresponding y-intercept. (For general 2D IPs, the y-intercept is equal to a constant times Z.) Fig. 6(b) shows one of the lines in this family. The feasible solution at this point has an x value of zero and a y value of 3, yielding a Z value of 3. Clearly, this is not the optimal solution. By considering all of the lines with a slope of $\frac{4}{5}$, it is apparent that the line with the smallest y-intercept value that also intersects a feasible solution will identify the optimal Z value. This line can be found by inspection, and is shown in Fig. 6(d). The optimal solution is $x = 2$ and $y = 1$, yielding $Z = -0.6$.

For this IP, the solution space of the original problem, SS_o, contains the nine points that lie within and on the polygon. Each of these nine points are feasible solutions as they satisfy all of the constraints. To apply cut-and-solve, a relaxation must be chosen. In this example, we relax constraints (16) and (17). The solution space for this

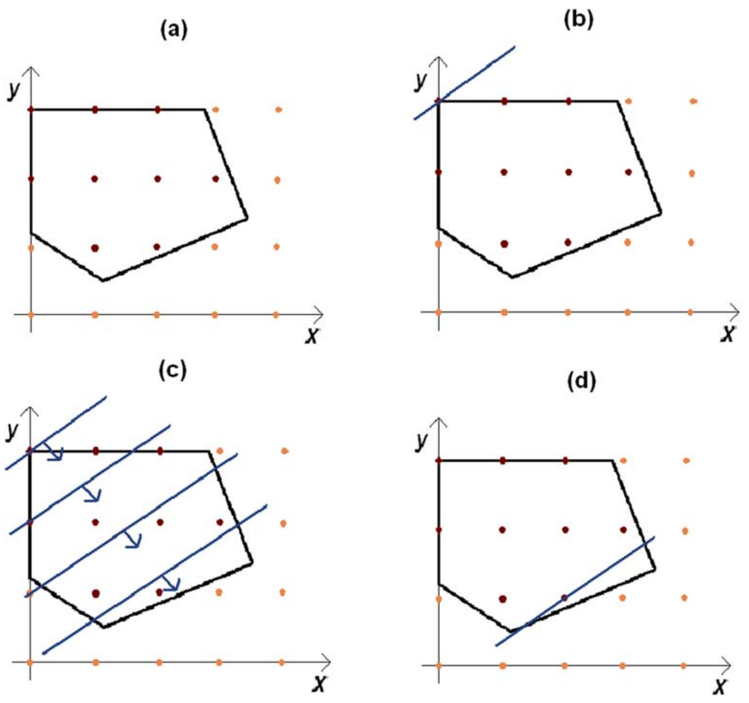


Fig. 6. Graphical solution of the example IP. (a) The feasible solution space. Dots within and on the polygon represent all of the feasible solutions. (b) One of the lines representing the objective function where $Z = 3$. It intersects the feasible solution $x = 0$, $y = 3$. (c) Lines representing the objective function for several values of Z. Arrows show the direction in which Z decreases. (d) The optimal solution of $x = 2$, $y = 1$, and $Z = -0.6$.

relaxed problem, SS_r, contains every point of real values of x and y inside and on the polygon—an infinite number of solutions.

Fig. 7 shows the steps taken when solving this IP using cut-and-solve. First, the relaxed solution is found. The solution to this relaxation is shown in Fig. 7(a). The value of x is 3.5, y is 1.4, and the solution value of this relaxed subproblem is equal to -1.4. This is a lower bound on the optimal solution value.

Next, a piercing cut is selected as shown in Fig. 7(b). The shaded region contains the solution to the relaxed problem as well as a feasible solution to the original problem. It also contains an infinite number of feasible solutions to the relaxed problem. This sparse problem is solved to find the best *integral* solution. There is only one integral solution in this sparse problem, so it is the best. Thus the incumbent solution is set to $x = 3$, $y = 2$, and the objective function value is -0.4. *Best* is set to -0.4.

The line that cuts away the shaded region from the polygon in Fig. 7(b) can be represented by a linear constraint: $y - \frac{17}{3}x \geqslant -14$. This constraint is added to the IP. Now the feasible region of the IP is reduced and its current solution space contains eight points as shown in Fig. 7(c). The relaxation of the current IP as shown in Fig. 7(d). The value of this relaxed solution is -1.1. Notice that this lower bound is less than *best*, so the search continues to the second iteration.

At the second node of the search, we select a piercing cut as shown in Fig. 7(e). The sparse problem represented by the shaded area in Fig. 7(e) is solved yielding a value of -0.6. This value is less than the current incumbent, so this solution becomes the new incumbent and *best* is set to -0.6.

The linear constraint corresponding to the current piercing cut is added to the IP, resulting in a reduced feasible solution space as shown in Fig. 7(f). The relaxed problem is solved on the current IP as shown in Fig. 7(g). The new lower bound is -0.2. This value is greater than the *best* of -0.6, so the search is finished.

The current incumbent must be an optimal solution. The incumbent is the best solution in the union of the solution spaces of all of the sparse problems that have been solved. -0.2 is a lower bound on the best possible solution that is in the remaining solution space. Since it is greater than the incumbent, there cannot be a solution in this space that is better than the incumbent. Therefore, optimality is ensured.

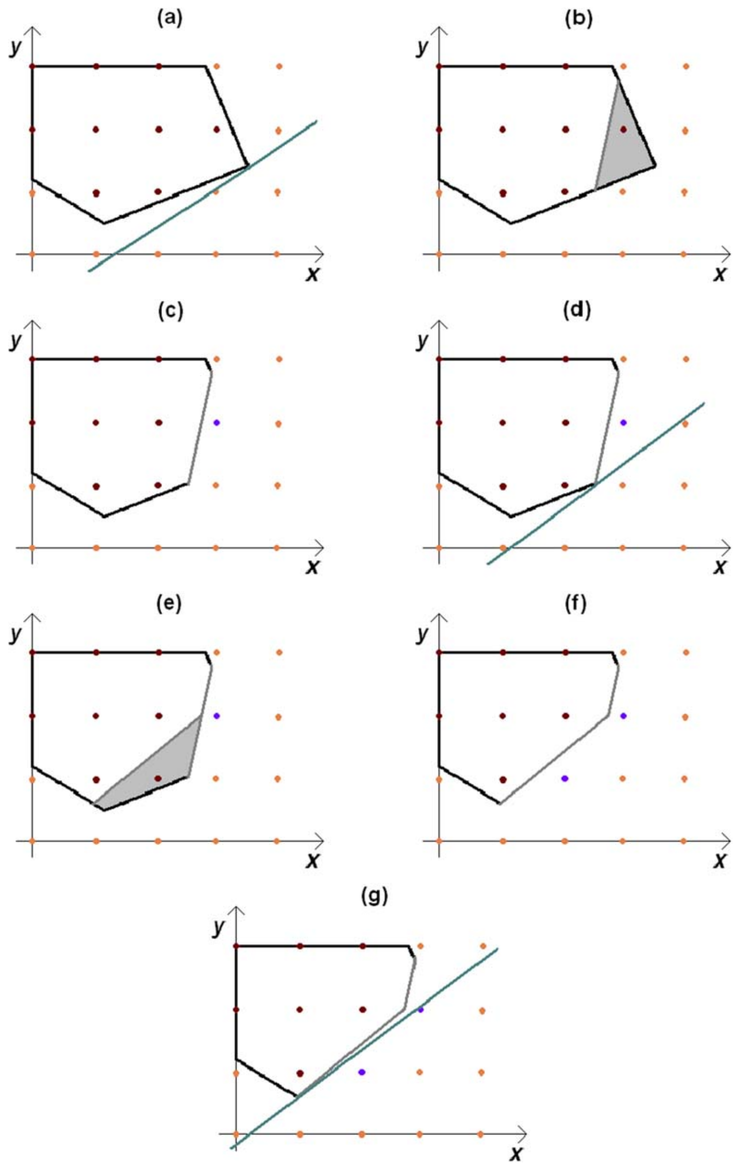


Fig. 7. Solving the example IP using cut-and-solve. (a) The relaxed solution of the original problem. (b) The first piercing cut. (c) The solution space of the IP with the piercing cut added. (d) The solution of the doubly-modified problem. (e) The second piercing cut. (f) The solution space of the IP with both piercing cuts added. (g) The solution of the doubly-modified problem, resulting in a value that is worse than the current incumbent.

The example problem presented in this section is easily solved by inspection. However, problems of interest may contain many thousands of decision variables, yielding solution spaces that are defined by convex polyhedrons in a correspondingly high-dimensional space. For example, a 100-city ATSP has 9,900 decision variables.

The piercing cuts chosen for the example problem in this section were customized for this particular instance. In the next section, a generic procedure for deriving piercing cuts is presented.

5. A generic cut-and-solve procedure

The cut-and-solve algorithm requires three selections. A relaxation must be chosen, a method for deriving piercing cuts must be devised, and a technique for solving the sparse problem must be selected. For best results, each of these choices should be determined by careful evaluation of the particular problem that is being addressed. For many problems of interest, relaxation techniques and algorithms for solving sparse instances may have been previously researched. However, the derivation of piercing cuts has not been previously explored. In this section, we summarize favorable characteristics of piercing cuts and present a generic procedure for deriving them. Then we present a generic cut-and-solve algorithm with all three selections specified. This general procedure can be used for any IP. While a more customized approach should yield better results, this generic approach is used in our implementation for the ATSP with very good results.

Every IP can be cast as a binary IP (BIP) [27], where the decision variables are restricted to the values of 0 and 1. For a given BIP, the decision variables can be partitioned into two sets: a small set S and a large set L. The sparse problem is solved over the small set S. This is essentially the same as setting the values of all of the variables in L to zero. After the sparse problem is solved, the constraint that set all of the variables in L to zero is removed. Then the following piercing cut is added to the BIP:

$$\sum_{x_i \in L} x_i \geqslant 1 \tag{18}$$

Using this cut partitions the solution space. Either the sum of the variables in L is equal to zero or it is greater than or equal to one. The solutions corresponding to the first case are precisely the solutions for the sparse problem. All of the other solutions in the solution space are contained in the original problem with the addition of piercing cut (18). The question that remains is how to partition the variables into sets S and L, producing effective piercing cuts. The answer to this question is believed to be problem dependent. We offer some guidelines here.

Following are desirable properties of piercing cuts:

(1) Each piercing cut should remove the solution to the current relaxed problem so as to prevent this solution from being found in subsequent iterations.
(2) The space that is removed by the piercing cut should be adequately sparse, so that the optimal solution can be found relatively easily.
(3) The piercing cuts should attempt to capture an optimal solution for the original problem. The algorithm will not terminate until an optimal solution has been cut away and consequently made the incumbent.
(4) In order to guarantee termination, each piercing cut should contain at least one feasible solution for the original, unrelaxed, problem.

Sensitivity analysis is extensively studied and used in Operations Research [44]. We borrow a tool from that domain to devise a totally generic approach to partitioning the decision variables into sets S and L as follows. First, the integrality constraints are relaxed and the BIP is solved as an LP. An LP solution defines a set of values referred to as *reduced costs*. Each decision variable x_i has a reduced cost value, which is a lower bound on the increase of the LP solution cost if the value of the variable is increased by one unit. Each of the decision variables with non-zero values in the LP solution has a reduced cost of zero. If another decision variable, x_i, has a reduced cost of zero, it may be possible to increase the value of x_i without increasing the cost of the LP solution. On the other hand, if x_i has a large reduced cost, then an equally large, or larger, increase of the LP solution cost would occur if there is a unit increase of the value of x_i.

We use this concept in our generic algorithm. Set S can be composed of all the decision variables with reduced costs less than a parameter α, where $\alpha > 0$. α should be set small enough that the resulting sparse problem is relatively easy to solve—satisfying property (2). On the other hand, it should be set as large as possible to increase the likelihood that set S contains all of the decision variables necessary for an optimal solution of the original problem. Property (1) is satisfied as the LP solution must be included in the sparse solution space because the decision variables with non-zero values in the LP solution have reduced costs of zero. Intuitively, decision variables with small reduced costs seem more likely to appear in an optimal solution. Thus, property (3) is addressed by using reduced costs as a guide for selecting variables for set S and by making α as large as practical. If guaranteed termination is required, property (4) can be satisfied by setting α to an adequately large value or adding variables to S that will guarantee at least one feasible solution for the original problem.

Having determined a method for partitioning the decision variables, we now present a completely generic cut-and-solve program that can be used for any BIP (and consequently any IP), as outlined in Fig. 8. The relaxation used is

```
algorithm generic_cut_and_solve (BIP)
     relax integrality and solve LP
     if (LP solution >= best) return best
     let S = {variables with reduced costs <= alpha}
     find optimal feasible solution in S
        update best if necessary
        if (LP solution >= best) return best
     add (sum of variables not in S >= 1) to BIP
     repeat
```

Fig. 8. A generic cut-and-solve algorithm.

the omission of the integrality constraints. The sparse problem can be solved using any BIP or IP solver. The piercing cut that is added to the BIP is $\sum_{x_i \notin S} x_i \geqslant 1$.

Notice that the performance of this generic approach could be improved by customizing it for the problem at hand. Tighter lower bounds, more effective piercing cuts, and/or use of a solver that is designed to solve sparse instances of the problem of interest may yield dramatic increases in performance. Yet, in the next two sections we show how a simple implementation of this generic approach can produce impressive results.

6. Cutting traveling salesmen down to size

We have implemented the cut-and-solve algorithm for solving real-world instances of the ATSP. The ATSP can be used to model a host of planning, routing, and scheduling problems in addition to a number of diverse applications as noted in Section 1. Many of these real-world applications are very difficult to solve using conventional methods and as such are good candidates for this alternative search strategy.

Seven real-world problem classes of the ATSP have been studied in [8]. Instance generators for these seven problem classes are available on David Johnson's webpage at: http://www.research.att.com/~dsj/chtsp/atsp.html. A brief description of the seven problems follows. The first class is the approximate shortest common superstring (`super`). This problem pertains to genome reconstruction, in which a given set of strings are combined to form the shortest possible superstring. The second class is tilted drilling machine, with an additive norm (`rtilt`). This class corresponds to the problem of scheduling the drilling of holes into a tilted surface. The third class (`stilt`) corresponds to the same problem, but with a different norm. The fourth class is random Euclidean stacker crane (`crane`), which pertains to planning the operation of a crane as it moves a number of items to different locations. The next class is disk drive (`disk`). This class represents the problem of scheduling a computer disk read head as it reads a number of files. The sixth class is pay phone collection (`coin`), corresponding to the problem of collecting coins from pay phones. The phones are located on a grid with two-way streets, with the perimeter of the grid consisting of one-way streets. Finally, the no-wait flow shop (`shop`) class represents the scheduling of jobs that each require the use of multiple machines. In this problem, there cannot be any delay for a job between machines.

In the generic algorithm, lower bounds are found by relaxing integrality. For the ATSP, this is the Held–Karp lower bound. Then arcs with reduced costs less than α are selected for set S and a sparse graph composed of these arcs is solved. The best tour in this sparse graph becomes our first incumbent solution. The original problem is then tightened by adding the constraint that the sum of the decision variables for the arcs in S is less than or equal to $n - 1$, where n is equal to the number of cities. This has the same effect as the piercing cut presented in the generic algorithm ($\sum_{x_i \notin S} x_i \geqslant 1$) as there are n arcs in an optimal solution for the ATSP. If all of the arcs needed for an optimal solution are present in the selected set of arcs, this solution will be made the incumbent. Otherwise, at least one arc that is not in this set is required for an optimal tour. This constraint is represented by the piercing cut.

The process of solving the Held–Karp lower bound, solving a sparse problem, and adding a piercing cut to the problem repeats until the Held–Karp value of the deeply-cut problem is greater than or equal to the incumbent solution. At this point, the incumbent must be an optimal tour.

The worst-case complexities of solving the Held–Karp lower bound (using the simplex method) and solving the sparse problem are both exponential. However, in practice these problems are usually relatively easy to solve. Furthermore, after the first iteration, the incumbent solution can be used as an upper bound for the sparse problem solver, potentially improving its performance.

Selection of an appropriate value for α may be dependent on the distribution of reduced cost values. In our current implementation, we simply select a number of arcs, m_{cut}, to be in the initial cut. At the root node, the arcs are sorted by their reduced costs and the m_{cut} lowest arcs are selected. α is set equal to the maximum reduced cost in this set. At subsequent nodes, α is used to determine the selected arcs. The choice of the value for m_{cut} is dependent on the problem class and the number of cities. We believe that determining α directly from the distribution of reduced costs would enhance this implementation as *a priori* knowledge of the problem class would not be necessary and α could be custom tailored to suit variations of instances within a class of problems.

If a cut does not contain a single feasible solution, it can be enlarged to do so. However, in our experiments this check was of no apparent benefit. The problems were solved using very few iterations, so guaranteeing termination was not of practical importance.

We used an LP and IP solver, `Cplex` [29], to solve both the relaxation and the sparse problem. All of the parameters for this solver were set to their default modes and no "warm starts" were used. For some of the larger instances, this generic solver became bogged down while solving the sparse problem. Performance could be improved by the substitution of an algorithm designed specifically for solving sparse ATSPs. We were unable to find such code available. We are investigating three possible implementations for this task: (1) adapting a Hamiltonian circuit enumerative algorithm to exploit ATSP properties, (2) enhancing the `Cplex` implementation by adding effective improvements such as the Padberg and Rinaldi shrinking procedures, external pricing, cutting planes customized for sparse ATSPs, heuristics for node selection, and heuristics for determining advanced bases, or (3) implementing a hybrid algorithm that integrates constraint programming with linear programming. However, despite the crudeness of our implementation, it suffices to demonstrate the potential of the cut-and-solve method. We compare our solver with branch-and-bound and branch-and-cut implementations in the next section.

7. Computational results for the ATSP

In this section, we compare cut-and-solve with branch-and-bound and branch-and-cut solvers. We used the seven real-world problem generators [8] that were discussed in Section 6 to conduct large scale experiments. For each problem class and size, we ran 500 trials, revealing great insight as to the expected behavior of each solver for each problem class. We started with $n = 25$ and incremented by 25 cities at a time until the solver was unable to solve the instances in an average time of ten minutes or less; thus allowing at most 5,000 minutes per trial.

With the exception of the `super` problem generator, the generators use a parameter which has an effect on the number of significant digits used for arc costs. We set this parameter to 100,000. The `crane` class uses a second parameter which was increased for larger numbers of cities in [8]. We set this parameter to 10 for $n \leqslant 100$, 13 for $n = 125$, and 15 for $n = 150$.

We compare cut-and-solve (`CZ`) with the branch-and-bound algorithm introduced by Carpaneto, Dell'Amico, and Toth (`CDT`) [7], which was discussed in Section 2. We also make comparisons with two branch-and-cut solvers: `Concorde` [1,2] and `Cplex` [29].

`Concorde` was introduced in Section 2. It is used for solving symmetric TSPs (STSPs). ATSP instances can be transformed into STSP instances using a *2-node* transformation [32]. David Johnson has publicly-available code for this 2-node transformation at http://www.research.att.com/~dsj/chtsp/atsp.html. When using the 2-node transformation, the number of arcs is increased to $4n^2 - 2n$. However, the number of arcs that have neither zero nor (essentially) infinite cost is $n^2 - n$, as in the original problem.

`Concorde` allows adjustment of its "chunk" size. This parameter controls the degree to which *project-and-lift cuts* [3] are utilized. Project-and-lift cuts are dynamically produced to suit the particular instance being solved. If the chunk size is set to zero, no project-and-lift cuts are generated and `Concorde` behaves as a pure STSP solver. It was shown in [17] that the default chunk size of 16 produced the best results for real-world ATSP instances. We used this default value in all of our trials.

The only publically available branch-and-cut code for specifically solving TSPs that we could find was `Concorde`. However, `Cplex` is a robust branch-and-cut solver for general IPs, so we also included it in comparisons. This solver dynamically determines search strategies and cuts for each instance. For instance, the variables chosen for branching and the direction of the search are customized to the given instance. Furthermore, `Cplex` uses nine different classes of cutting planes (Gomory cuts comprise one of the classes) and automatically determines the frequency of each class of cuts to use. Finally, `Cplex` uses various heuristics to find integer feasible solutions while traversing the search tree.

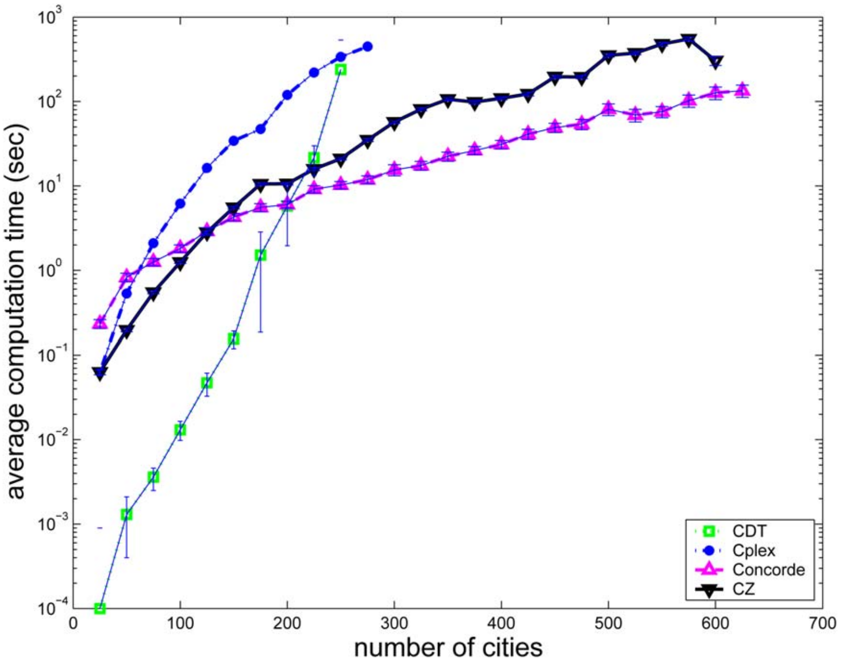


Fig. 9. Average normalized computation times with 95% confidence interval shown for the `super` problem class.

Generally speaking, `CZ` is as generic as our `Cplex` implementation. Both algorithms identify subtours and add appropriate subtour elimination constraints. Other than this, both algorithms are generic IP solvers.

Our code was run using `Cplex` version 8.1. We used Athlon 1.9 MHz dual processors with two gigabytes shared memory for our tests (although we did not use parallel processing). In order to identify subtours in the relaxed problem, we used Matthew Levine's implementation of the Nagamochi and Ibaraki minimum cut code [37], which is available at [38]. Our code is available at [10].

In the experiments presented here, we found that the search path was quite short. Typically only one or two sparse problems and two or three relaxed problems were solved. This result indicates that the set of arcs with small reduced costs is likely to contain an optimal solution.

The average normalized computation times and the 95% confidence intervals for the real-world problem classes are shown in Figs. 9 through 15. One may be interested in the "typical" time that is required for each of these solvers, so the median times for all of the trials are given in Figs. 16 through 22. Table 1 lists the largest problem instance size that is solved by each algorithm within the execution time limit.

`Concorde` outperformed all of the other solvers for the `super` class as shown in Figs. 9 and 16. It was able to solve instances with $n = 625$, `CZ` solved 600 cities, `Cplex` solved 275 cities, and `CDT` solved 250 cities.

`Concorde` also outperformed the other solvers for the `rtilt` problem class as shown in Figs. 10 and 17. It solved 125-city instances, `CZ` solved 100 cities, `Cplex` solved 75 cities, and `CDT` solved 25 cities.

We solved some of the `super` and `rtilt` instances for all solutions and found they both had many optimal solutions.

For the other five problem classes, `CZ` outperformed the other solvers. For the `crane` class, `CZ`, `Concorde`, and `Cplex` were able to solve up to 125 cities as shown in Figs. 12 and 19. `CZ` was the fastest for this number of cities, followed by `Concorde`.

For the four other problem classes, `CZ` was able to solve larger instances than any of the other solvers. For `stilt`, `CZ` solved 100 cities within the allotted time, while `Concorde` and `Cplex` solved 75 cities as shown in Figs. 11 and 18. For `disk`, `CZ` solved 775 cities, while `Cplex` followed far behind with 325 cities as shown in Figs. 13 and 20. `CZ` solved 100-city `coin` instances and `Cplex` solved 75 cities as shown in Figs. 14 and 21. Finally, `CZ` was able to solve 550-city `shop` instances, with `CDT` following with 350 cities as shown in Figs. 15 and 22.

`CDT` performed moderately well for `super` and `shop`. However, it was only able to solve 50 cities for `crane` and 25 cities for the other four problem classes. We let the computations run past the time limit and found that many

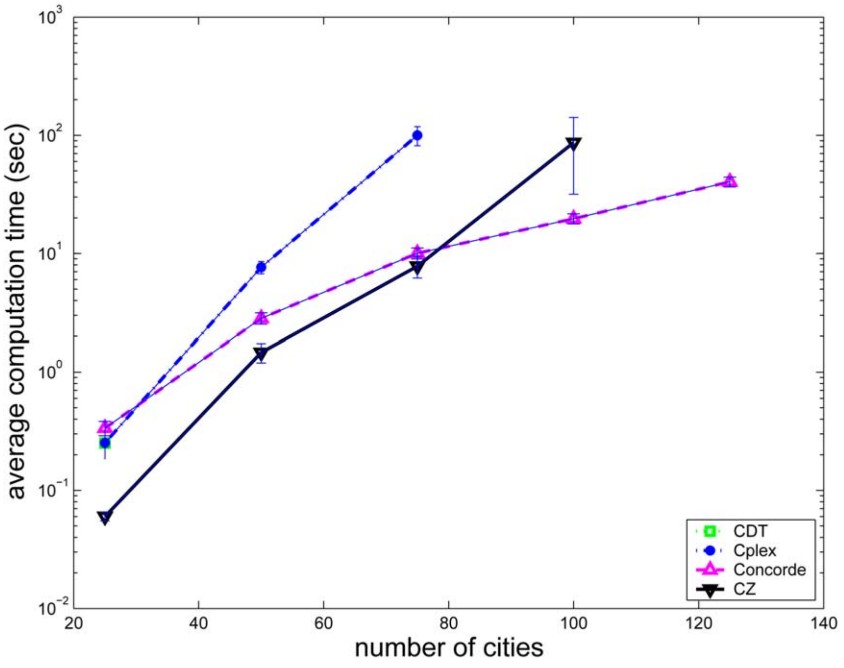


Fig. 10. Average normalized computation times with 95% confidence interval shown for the `rtilt` problem class.

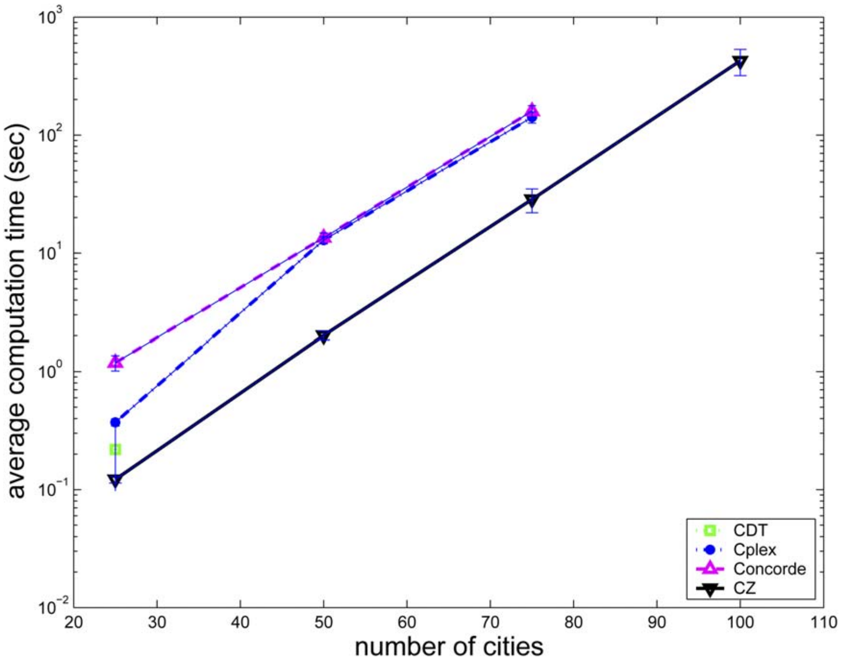


Fig. 11. Average normalized computation times with 95% confidence interval shown for the `stilt` problem class.

of these trials required substantial amounts of time. The worst case was for the 50-city `disk` class, which we finally killed at 25,900 minutes.

In summary, for the problem instances we tested, CDT was not very robust. `Cplex` was robust, but had moderate performance in general. `Concorde` was the fastest for the two problem classes that had a large number of optimal solutions. `CZ` was robust and had the best performance for the other five classes.

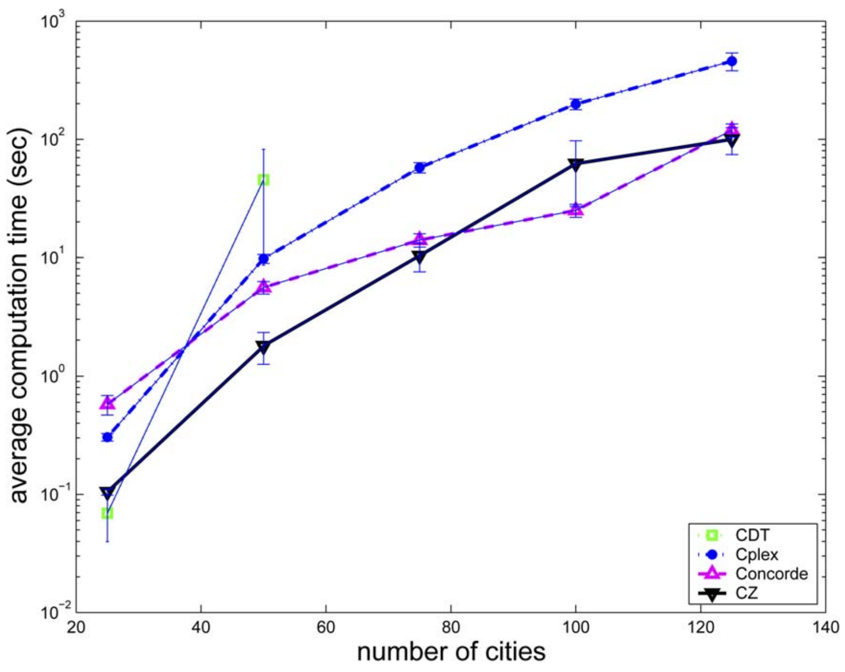


Fig. 12. Average normalized computation times with 95% confidence interval shown for the `crane` problem class.

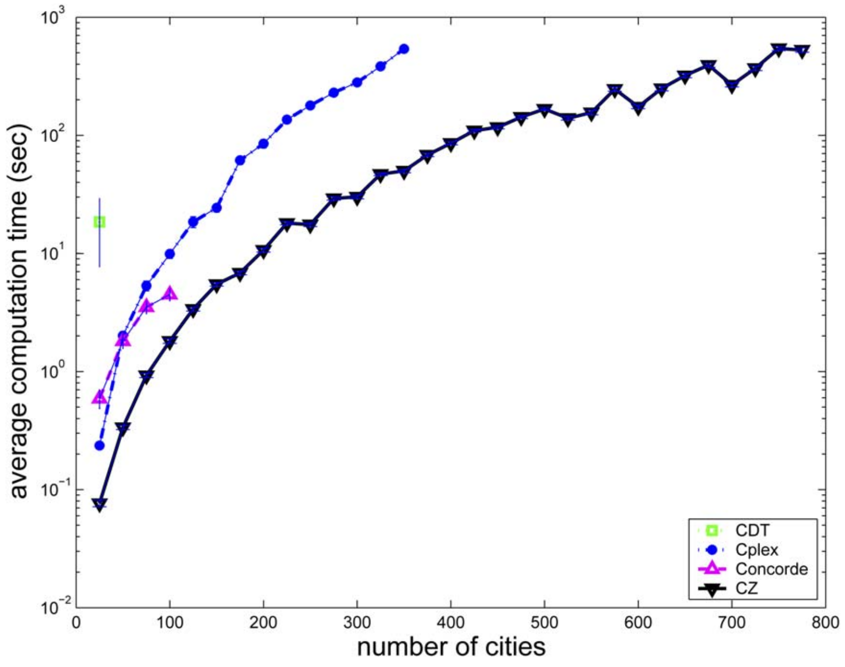


Fig. 13. Average normalized computation times with 95% confidence interval shown for the `disk` problem class.

8. Discussion and related work

Many optimization problems of interest are difficult to solve to optimality, so approximations are employed. Sacrificing optimality can be costly in several situations. Approximations can be problematic when the solution is important for the progress of research, such as biological applications. Approximations can be costly when the solution is an ex-

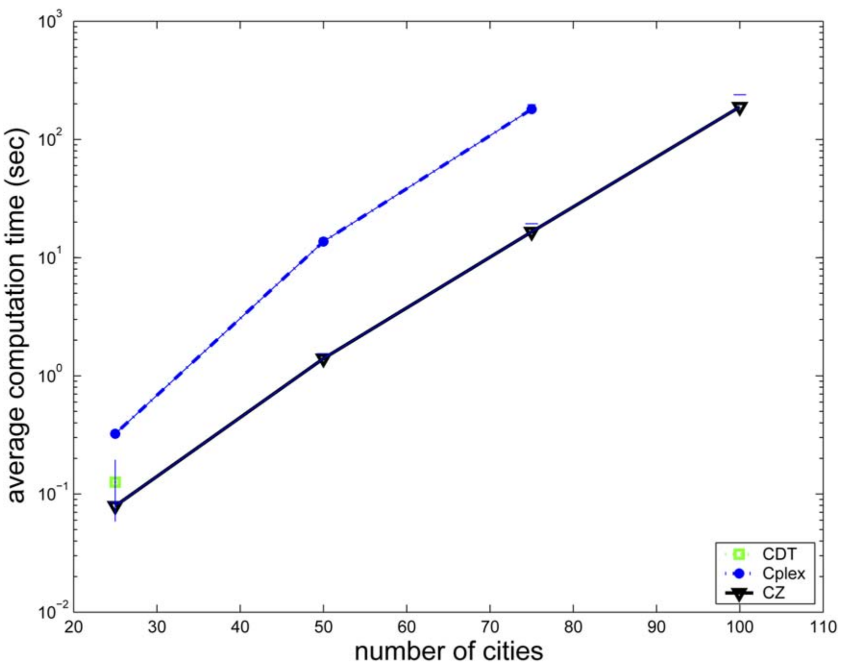


Fig. 14. Average normalized computation times with 95% confidence interval shown for the `coin` problem class.

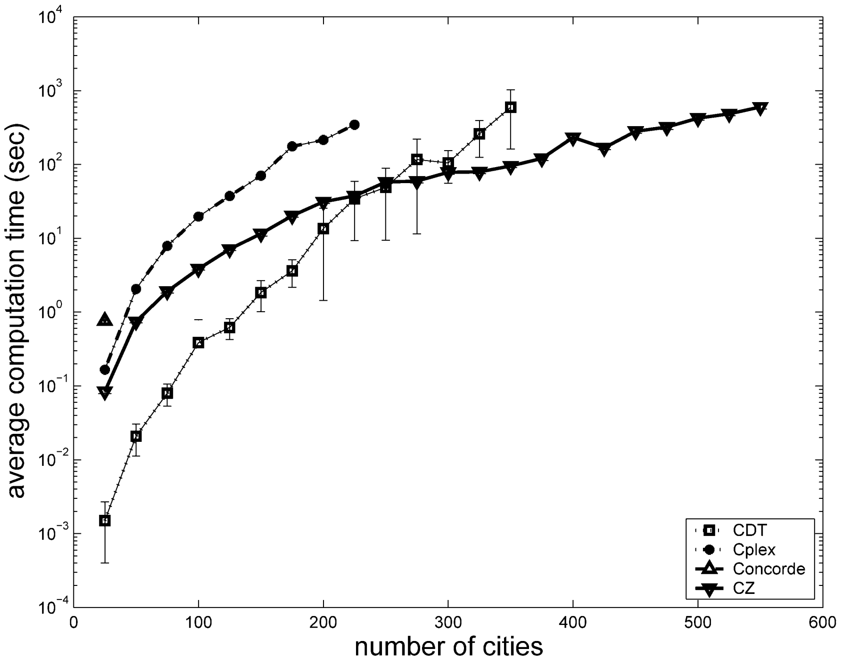


Fig. 15. Average normalized computation times with 95% confidence interval shown for the `shop` problem class.

pensive procedure, such as routing a spacecraft, or when the solution is reused a number of times, as in manufacturing applications. Moreover, it can be deleterious to use an approximation in any application if the approximation is poor. Cut-and-solve offers an alternative to branch-and-bound and branch-and-cut when optimality is desired.

In our experiments, we observed that the performance of cut-and-solve diminished for problems with large numbers of optimal solutions. We monitored a few of these trials and noticed that cut-and-solve found an optimal solution early

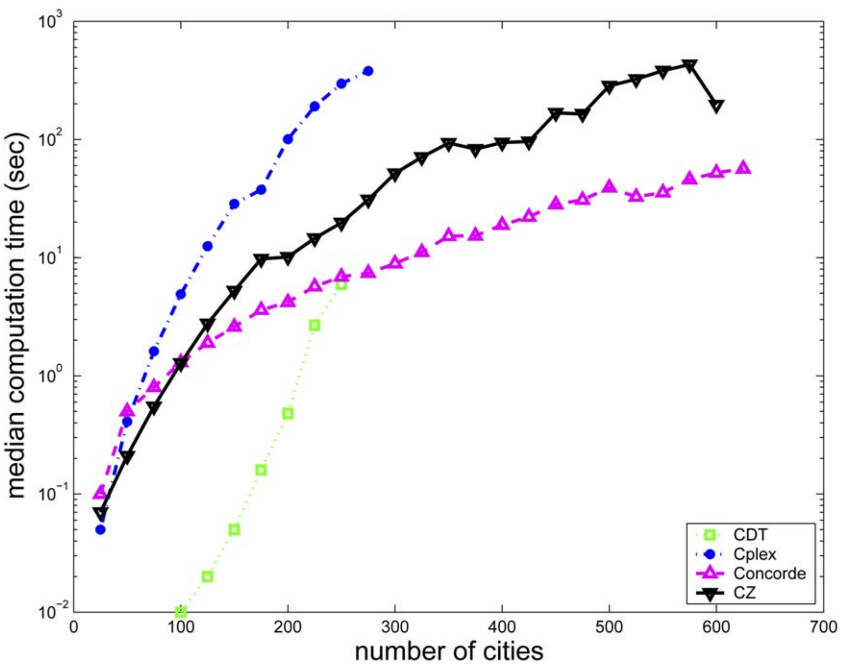


Fig. 16. Median computation times for the super problem class.

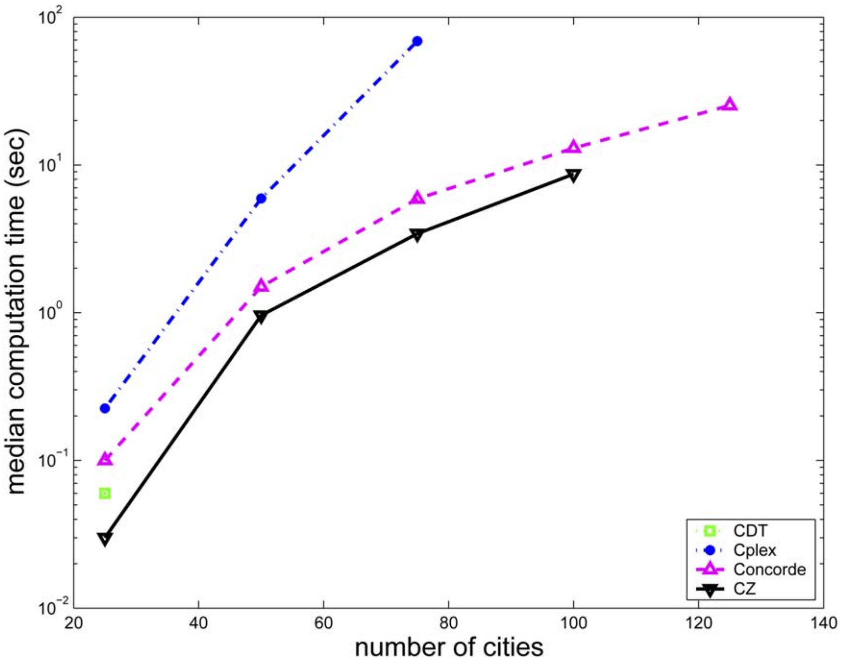


Fig. 17. Median computation times for the rtilt problem class.

in the search, but required a substantial amount of computation time to prove optimality. We suspect that many optimal solutions remained in the doubly-modified problem, making it difficult for the relaxation to be equal to the optimal solution value. It may be that most of the optimal solutions had to be cut away before the computation could complete.

One concern about the viability of cut-and-solve might be that choosing a poor cut may be similar to choosing the wrong path in a depth-first search tree. These two actions are very different. When choosing the wrong child in depth-

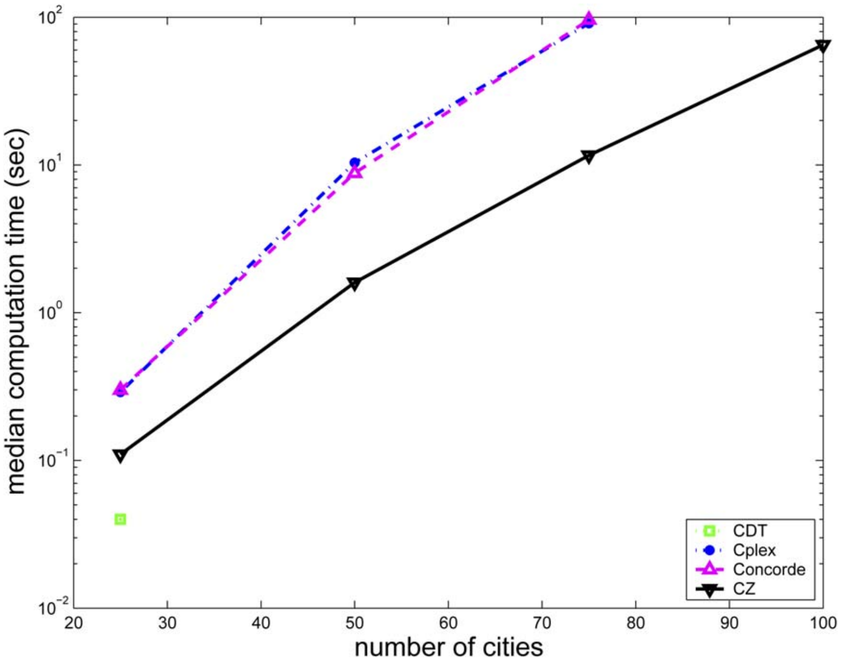


Fig. 18. Median computation times for the `stilt` problem class.

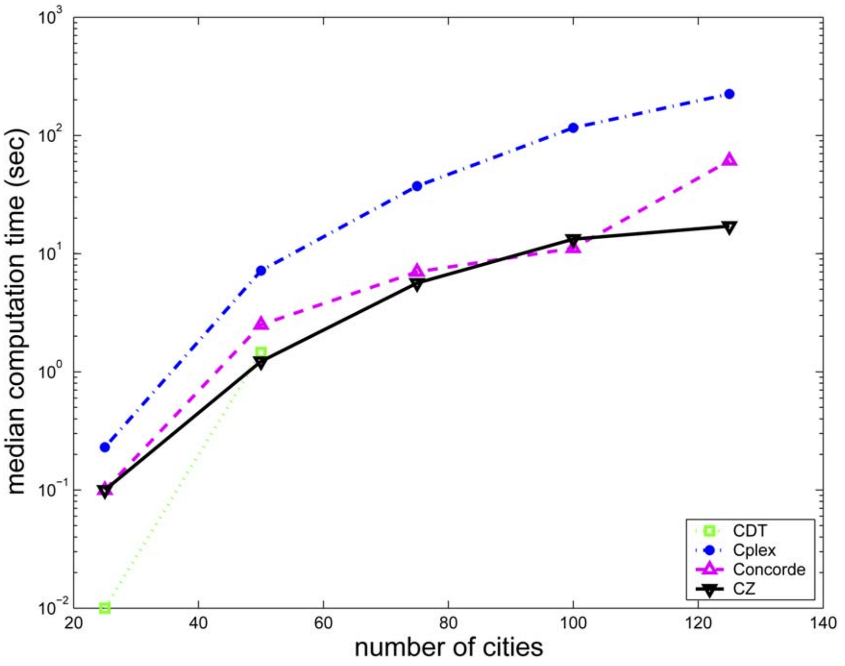


Fig. 19. Median computation times for the `crane` problem class.

first search, the decision is permanent for all of the descendants in the subtree. For example, if a variable appears in every optimal solution (a *backbone* variable [40,48]), then setting its value to zero is calamitous. Conversely, forcing the inclusion of a variable that doesn't appear in any feasible solution (a *fat* variable [9]) results in a similar dire situation. Every node in the subtree is doomed. On the other hand, when a poor cut is chosen in cut-and-solve, the only consequence is that the time spent exploring that single node was relatively ineffective. It wasn't completely

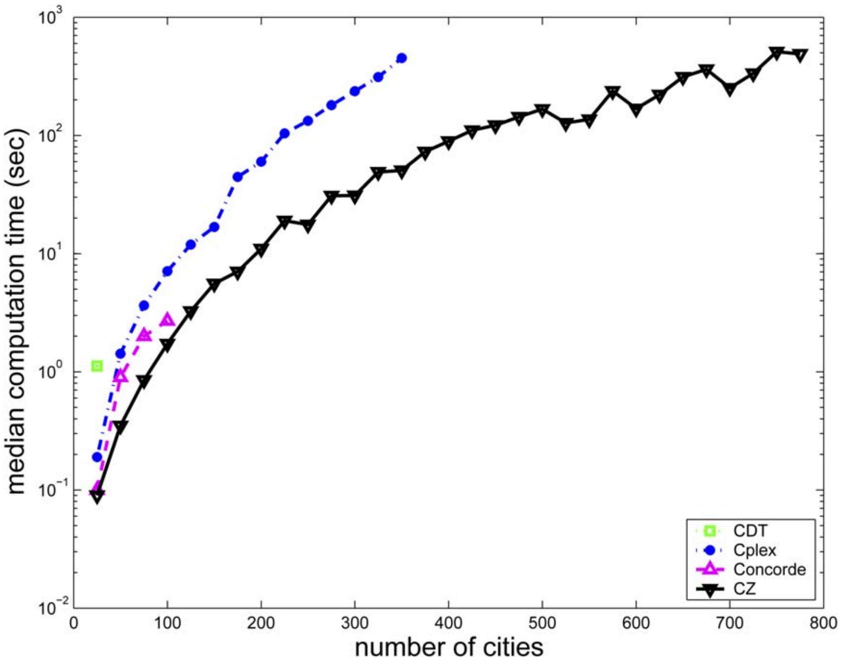


Fig. 20. Median computation times for the `disk` problem class.

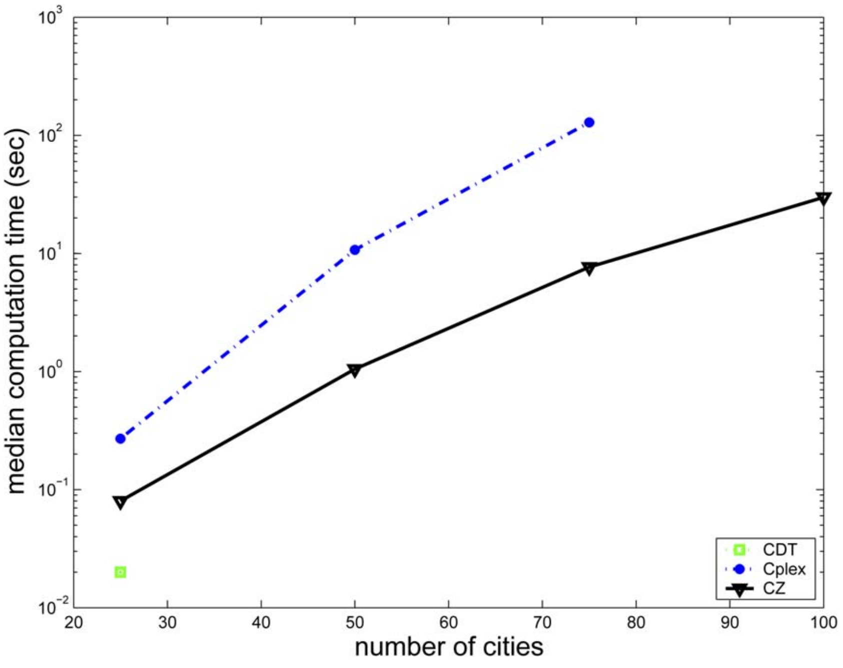


Fig. 21. Median computation times for the `coin` problem class.

wasted as even a poor cut helps to tighten the problem to some degree and once it is made, this cut cannot be repeated in future iterations. Moreover, subsequent nodes are not "locked in" by the choice made for the current cut.

Search tree methods such as branch-and-bound and branch-and-cut must choose between memory problems or the risk of fruitlessly searching subtrees containing no optimal solutions. Cut-and-solve is free from these difficulties. Its memory requirement is insignificant as only the current incumbent solution and the current doubly-modified problem

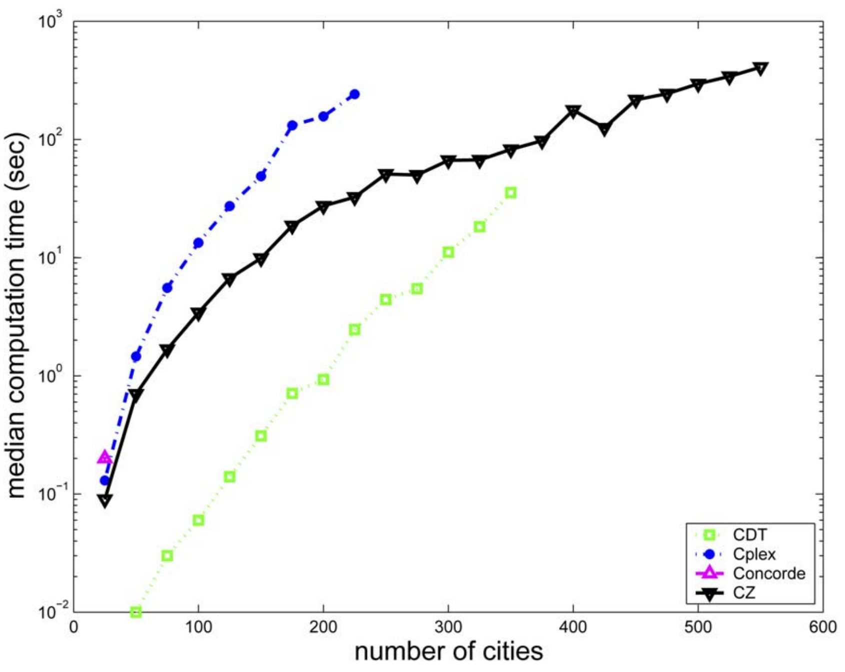


Fig. 22. Median computation times for the shop problem class.

Table 1
Largest problem size that was solved within the allotted time for each algorithm and each problem class

	CDT	Concorde	Cplex	CZ
super	250	625	275	600
rtilt	25	125	75	100
stilt	25	75	75	100
crane	50	125	125	125
disk	25	100	325	775
coin	25	–	75	100
shop	350	25	250	550

(Concorde terminated with errors for the coin class.)

need to be saved as the search path is traversed. Furthermore, being an iterative search, there are no subtrees in which to get lost. A great number of techniques have been devised in an effort to overcome this problem for depth-first search. Among these are iterative deepening [34], limited discrepancy [24], and randomization and restarts [19].

Another interesting comparison is that branch-and-bound and branch-and-cut must compute at least one relaxed solution that is not only feasible for the original problem, but it must also be optimal for the original problem. In contrast, cut-and-solve is never required to find a relaxed solution that is optimal or even feasible for the original problem. Consider, for example, generic cut-and-solve. Each of the right-hand children in Fig. 5 may have non-integral solutions. Yet, the search is terminated as soon as the value of the relaxation is greater than or equal to the incumbent value.

We now discuss several algorithms that are similar to cut-and-solve. Cut-and-solve is similar to Gomory's algorithm in that cuts are used to constrain the problem and a linear path is searched. Gomory's algorithm is sometimes referred to as "solving the problem at the root node" as this is essentially its behavior when comparing it to branch-and-cut. However, cutting planes are applied and relaxed problems are solved in an iterative manner until the search is terminated, suggesting an iterative progression of the search. Cut-and-solve can be thought of as an extension of Gomory's method. Like Gomory's technique, cuts are applied, one at a time, until an optimal solution is found.

The major difference between the two methods is that Gomory's cuts are not *piercing* cuts as they do not cut away any feasible solutions to the original problem. Piercing cuts are deeper than Gomory cuts as they are not required to trim around feasible solutions for the unrelaxed problem. Cutting deeply yields two benefits. First, it provides a set of solutions from which the best is chosen for a potential incumbent. Second, these piercing cuts tighten the relaxed problem in an aggressive manner, and consequently tend to increase the solution of the doubly-modified problem more expeditiously.

Cut-and-solve is also similar to an algorithm for solving the Orienteering Problem (OP) as presented in [18]. In this work, *conditional cuts* remove feasible solutions to the original problem. These cuts are used in conjunction with traditional cuts and are used to tighten the problem. When a conditional cut is applied, an enumeration of all of the feasible solutions within the cut is attempted. If the enumeration is not solved within a short time limit, the cut is referred to as a *branch cover cut* and the sparse graph associated with it is stored. This algorithm attempts to solve the OP in an iterative fashion, however, branching occurs after every five branch cover cuts have been applied. After this branch-and-cut tree is solved, a second branch-and-cut tree is solved over the union of all of the graphs stored for the branch cover cuts.

Cut-and-solve differs from this OP algorithm in several ways. First, incumbent solutions are forced to be found early in the cut-and-solve search. These incumbents provide useful upper bounds as well as improve the *anytime* performance. Second, the approach used in [18] stores sparse problems and combines and solves them as a single, larger problem after the initial branch-and-cut tree is explored. Finally, this OP algorithm is not truly iterative as branching is allowed.

In a more broad sense, cut-and-solve is similar to divide-and-conquer [41] in that both techniques identify small subproblems of the original problem and solve them. While divide-and-conquer solves all of these subproblems, cut-and-solve solves a very small percentage of them. When comparing these two techniques, cut-and-solve might be thought of as divide-and-conquer with powerful pruning rules.

We conclude this section with a note about the generality of piercing cuts. Piercing cuts are used in an iterative fashion for cut-and-solve, but they could also be applied in other search strategies such as branch-and-cut. As long as the sparse problem defined by the cut is solved, piercing cuts could be added to an IP. For example, Fischetti and Toth's branch-and-cut algorithm [17] solves a number of sparse problems at each node of the search tree in order to improve the upper bound. Piercing cuts corresponding to these sparse problems could be added to the IP without loss of optimality. These piercing cuts may tighten the problem to a greater degree than the tightening due to traditional cuts and may reduce the time required to solve the instance.

9. Conclusions

In this paper, we presented a search strategy which we referred to as cut-and-solve. We showed that optimality and termination are guaranteed despite the fact that no branching is used. Being an iterative strategy, this technique is immune to some of the pitfalls that plague search tree methods such as branch-and-bound and branch-and-cut. Memory requirements are negligible, as only the incumbent solution and the current tightened problem need be saved as the search path is traversed. Furthermore, there is no need to use techniques to reduce the risks of fruitlessly searching subtrees void of any optimal solution.

We demonstrated cut-and-solve for integer programs and have implemented this strategy for solving the ATSP. In general, our generic implementation is robust and outperforms state-of-the-art solvers for difficult real-world instances that don't have many optimal solutions. For five of the seven problem classes, cut-and-solve outperformed the other solvers. The two remaining classes had many optimal solutions.

In the future, we plan to improve our implementation by designing a custom solver for sparse ATSPs. As for the general cut-and-solve search strategy, we are looking into dynamically determining α so that it is customized to each particular instance. Finally, we are investigating the use of cut-and-solve for other IPs. We are currently implementing it for a biological problem, haplotype inferencing [21].

Life is full of optimization problems. Many such problems arise in the field of Artificial Intelligence and a number of search strategies have emerged to tackle them. Yet, many of these problems stubbornly defy resolution by current methods. It is our hope that the unique characteristics of cut-and-solve may prove to be useful when addressing some of these interesting problems.

Acknowledgements

This research was supported in part by NDSEG and Olin Fellowships, by NSF grants IIS-0196057, ITR/EIA-0113618, and IIS-0535257, and in part by DARPA Cooperative Agreement F30602-00-2-0531. We are grateful to the anonymous reviewers of this paper, as well as a preliminary conference paper, who provided a number of valuable insights. We thank Matthew Levine, for the use of his minimum cut code, and David Applegate, Robert Bixby, Vašek Chvátal, and William Cook for the use of Concorde. We also extend thanks to Matteo Fischetti, who generously shared his expertise in several discussions.

References

[1] D. Applegate, R. Bixby, V. Chvátal, W. Cook, TSP cuts which do not conform to the template paradigm, in: M. Junger, D. Naddef (Eds.), Computational Combinatorial Optimization, Springer, New York, 2001, pp. 261–304.
[2] D. Applegate, R. Bixby, V. Chvátal, W. Cook, Concorde—A code for solving Traveling Salesman Problems, 15/12/99 Release, http://www.tsp.gatech.edu//concorde.html, web.
[3] D. Applegate, R. Bixby, V. Chvátal, W. Cook, On the solution of Traveling Salesman Problems, in: Documenta Mathematica, Extra volume ICM III, 1998, pp. 645–656.
[4] E. Balas, P. Toth, Branch and bound methods, in: The Traveling Salesman Problem, John Wiley & Sons, Essex, England, 1985, pp. 361–401.
[5] F. Bock, An algorithm for solving 'traveling-salesman' and related network optimization problems, Technical report, Armour Research Foundation, 1958. Presented at the Operations Research Society of America 14th National Meeting, St. Louis, October 1958.
[6] M. Boddy, T. Dean, Solving time-dependent planning problems, in: Proceedings of IJCAI-89, Detroit, MI, 1989, pp. 979–984.
[7] G. Carpaneto, M. Dell'Amico, P. Toth, Exact solution of large-scale, asymmetric Traveling Salesman Problems, ACM Transactions on Mathematical Software 21 (1995) 394–409.
[8] J. Cirasella, D.S. Johnson, L. McGeoch, W. Zhang, The asymmetric traveling salesman problem: Algorithms, instance generators, and tests, in: Proc. of the 3rd Workshop on Algorithm Engineering and Experiments, 2001.
[9] S. Climer, W. Zhang, Searching for backbones and fat: A limit-crossing approach with applications, in: Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02), Edmonton, Alberta, July 2002, pp. 707–712.
[10] S. Climer, W. Zhang, Cut-and-solve code for the ATSP, http://www.climer.us, or http://www.cse.wustl.edu/~zhang/projects/tsp/cutsolve, 2004.
[11] S. Climer, W. Zhang, A linear search strategy using bounds, in: Proc. 14th International Conference on Automated Planning and Scheduling (ICAPS'04), Whistler, Canada, June 2004, pp. 132–141.
[12] G.A. Croes, A method for solving traveling-salesman problems, Operations Research 6 (1958) 791–812.
[13] G. Dantzig, Maximization of linear function of variables subject to linear inequalities, in: T.C. Koopmans (Ed.), Activity Analysis of Production and Allocation, Wiley, New York, 1951.
[14] G.B. Dantzig, D.R. Fulkerson, S.M. Johnson, Solution of a large-scale traveling-salesman problem, Operations Research 2 (1954) 393–410.
[15] R. Dechter, F. Rossi, Constraint satisfaction, in: Encyclopedia of Cognitive Science, March 2000.
[16] W.L. Eastman, Linear programming with pattern constraints, PhD thesis, Harvard University, Cambridge, MA, 1958.
[17] M. Fischetti, A. Lodi, P. Toth, Exact methods for the Asymmetric Traveling Salesman Problem, in: G. Gutin, A. Punnen (Eds.), The Traveling Salesman Problem and its Variations, Kluwer Academic, Norwell, MA, 2002.
[18] M. Fischetti, J.J. Salazar, P. Toth, The generalized traveling salesman and orienteering problems, in: G. Gutin, A. Punnen (Eds.), The Traveling Salesman Problem and its Variations, Kluwer Academic, Norwell, MA, 2002, pp. 609–662.
[19] C.P. Gomes, B. Selman, H. Kautz, Boosting combinatorial search through randomization, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98), New Providence, RI, 1998, pp. 431–438.
[20] R.E. Gomory, Outline of an algorithm for integer solutions to linear programs, Bulletin of the American Mathematical Society 64 (1958) 275–278.
[21] D. Gusfield, S.H. Orzack, Haplotype inference, in: S. Aluru (Ed.), Handbook on Bioinformatics, CRC, 2005, in press.
[22] G. Gutin, A.P. Punnen, The Traveling Salesman Problem and its Variations, Kluwer Academic, Norwell, MA, 2002.
[23] T.P. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems, Science and Cybernetics 4 (1968) 100–107.
[24] W.D. Harvey, M.L. Ginsberg, Limited discrepancy search, in: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95), vol. 1, Montreal, Canada, August 1995.
[25] M. Held, R.M. Karp, The traveling salesman problem and minimum spanning trees, Operations Research 18 (1970) 1138–1162.
[26] M. Held, R.M. Karp, The traveling salesman problem and minimum spanning trees: Part ii, Mathematical Programming 1 (1971) 6–25.
[27] F. Hillier, G. Lieberman, Introduction to Operations Research, sixth ed., McGraw-Hill, Boston, 2001.
[28] K.L. Hoffman, M. Padberg, Improving LP-representations of zero-one linear programs for branch-and-cut, ORSA Journal on Computing 3 (1991) 121–134.
[29] Ilog, http://www.cplex.com, web.
[30] D.S. Johnson, G. Gutin, L.A. McGeoch, A. Yeo, W. Zhang, A. Zverovich, Experimental analysis of heuristics for the ATSP, in: G. Gutin, A. Punnen (Eds.), The Traveling Salesman Problem and its Variations, Kluwer Academic, Norwell, MA, 2002.
[31] D.S. Johnson, L.A. McGeoch, E.E. Roghberg, Asymptotic experimental analysis for the Held–Karp Traveling Salesman bound, in: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, 1996, pp. 341–350.

[32] R. Jonker, T. Volgenant, Transforming asymmetric into symmetric traveling salesman problems, Operations Research Letters 2 (1983) 161–163.
[33] L.G. Khaciyan, A polynomial algorithm for linear programming, Doklady Akademii Nauk SSSR 244 (1979) 1093–1096.
[34] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artificial Intelligence 27 (1985) 97–109.
[35] R.E. Korf, A complete anytime algorithm for number partitioning, Artificial Intelligence 105 (1998) 133–155.
[36] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys, The Traveling Salesman Problem, John Wiley & Sons, Essex, England, 1985.
[37] M.S. Levine, Experimental study of minimum cut algorithms, PhD thesis, Computer Science Dept., Massachusetts Institute of Technology, Massachusetts, May 1997.
[38] M.S. Levine, Minimum cut code, http://theory.lcs.mit.edu/~mslevine/mincut/index.html, web.
[39] S. Martello, P. Toth, Linear assignment problems, Annals of Discrete Mathematics 31 (1987) 259–282.
[40] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, L. Troyansky, Determining computational complexity from characteristic 'phase transitions', Nature 400 (1999) 133–137.
[41] Z.G. Mou, P. Hudak, An algebraic model of divide-and-conquer and its parallelism, Journal of Supercomputing 2 (1988) 257–278.
[42] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1984.
[43] M.J. Rossman, R.J. Twery, A solution to the travelling salesman problem, Operations Research 6 (1958) 687. Abstract E3.1.3.
[44] A. Saltelli, S. Tarantola, F. Campolongo, M. Ratto, Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models, John Wiley & Sons, Ltd., West Sussex, England, 2004.
[45] H.P. Williams, Model Building in Mathematical Programming, second ed., John Wiley and Sons, Chichester, 1985.
[46] W. Zhang, Complete anytime beam search, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98), Madison, WI, July 1998, pp. 425–430.
[47] W. Zhang, Depth-first branch-and-bound vs. local search: A case study, in: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), Austin, TX, July 2000, pp. 930–935.
[48] W. Zhang, Phase transitions and backbones of the asymmetric Traveling Salesman, Journal of Artificial Intelligence Research 20 (2004) 471–497.
[49] W. Zhang, R.E. Korf, Performance of linear-space search algorithms, Artificial Intelligence 79 (1995) 241–292.

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Artificial Intelligence 170 (2006) 739–778

Artificial Intelligence

www.elsevier.com/locate/artint

Solving logic program conflict through strong and weak forgettings ☆

Yan Zhang [a,*], Norman Y. Foo [b]

[a] *Intelligent Systems Laboratory, School of Computing and Mathematics, University of Western Sydney, Penrith South DC, NSW 1797, Australia*
[b] *School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia*

Received 6 September 2005; received in revised form 23 February 2006; accepted 23 February 2006

Available online 29 March 2006

Abstract

We consider how to forget a set of atoms in a logic program. Intuitively, when a set of atoms is forgotten from a logic program, all atoms in the set should be eliminated from this program in some way, and other atoms related to them in the program might also be affected. We define notions of strong and weak forgettings in logic programs to capture such intuition, reveal their close connections to the notion of forgetting in classical propositional theories, and provide a precise semantic characterization for them. Based on these notions, we then develop a general framework for conflict solving in logic programs. We investigate various semantic properties and features in relation to strong and weak forgettings and conflict solving in the proposed framework. We argue that many important conflict solving problems can be represented within this framework. In particular, we show that all major logic program update approaches can be transformed into our framework, under which each approach becomes a specific conflict solving case with certain constraints. We also study essential computational properties of strong and weak forgettings and conflict solving in the framework.

Keywords: Conflict solving; Knowledge representation; Answer set semantics; Logic program update; Computational complexity

1. Introduction

1.1. Motivation

One promising approach in the research of reasoning about knowledge dynamics is to represent agents' knowledge bases as logic programs on which necessary updates/revisions are conducted as a way of modeling agents' knowledge evolution. A key issue in this study is to solve various conflicts and inconsistencies in logic programs, e.g. [15].

We observe that some typical conflict solving problems in applications are essential in reasoning about agents' knowledge change, but they may not be properly handled by traditional logic program updates. Let us consider a

☆ Some results presented in this paper were published in IJCAI-2005 and AAAI-2005 [Y. Zhang, N.Y. Foo, K. Wang, Solving logic program conflict through strong and weak forgettings, in: Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), 2005, pp. 627–632; Y. Zhang, N.Y. Foo, A unified framework for representing logic program updates, in: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05), 2005, pp. 707–712].

* Corresponding author.
E-mail addresses: yan@cit.uws.edu.au (Y. Zhang), norman@cse.unsw.edu.au (N.Y. Foo).

0004-3702/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2006.02.002

scenario. John wants Sue to help him to complete his assignment. He knows that Sue will help him if she is not so busy. Tom is a good friend of John and wants John to let him copy John's assignment. Then John learns that Sue hates Tom, and will not help him if he lets Tom copy his assignment, which will be completed under Sue's help. While John does not care whether Sue hates Tom or not, he has to consider Sue's condition to offer him help. What is John going to do? We formalize this scenario in a logic programming setting. We represent John's knowledge base Π_J:

$$r_1\colon\ \mathit{complete}(\mathit{John}, \mathit{Assignment}) \leftarrow \mathit{help}(\mathit{Sue}, \mathit{John}),$$

$$r_2\colon\ \mathit{help}(\mathit{Sue}, \mathit{John}) \leftarrow \mathit{not}\ \mathit{Busy}(\mathit{Sue}),$$

$$r_3\colon\ \mathit{goodFriend}(\mathit{John}, \mathit{Tom}) \leftarrow,$$

$$r_4\colon\ \mathit{copy}(\mathit{Tom}, \mathit{Assignment}) \leftarrow \mathit{goodFriend}(\mathit{John}, \mathit{Tom}),\ \mathit{complete}(\mathit{John}, \mathit{Assignment}),$$

and Sue's knowledge base Π_S:

$$r_5\colon\ \mathit{hate}(\mathit{Sue}, \mathit{Tom}) \leftarrow,$$

$$r_6\colon\ \leftarrow \mathit{help}(\mathit{Sue}, \mathit{John}),\ \mathit{copy}(\mathit{Tom}, \mathit{Assignment}).$$

In order to take Sue's knowledge base into account, John may update his knowledge base Π_J in terms of Sue's Π_S. In this way, John obtains a solution: $\Pi_J^{\mathit{final}} = \{r_1, r_2, r_3, r_5, r_6\}$ or its stable model, from which we know that Sue will help *John* to complete the assignment and John will not let Tom copy his assignment. Although the conflict between Π_J and Π_S has been solved by updating, the result is somehow not always satisfactory. For instance, while John wants Sue to help him, he may have no intention to contain the information that Sue hates Tom into his new knowledge base.

As an alternative, John may just weaken his knowledge base by *forgetting* atom $\mathit{copy}(\mathit{Tom}, \mathit{Assignment})$ from Π_J in order to accommodate Sue's constraint on help. Then John will have a new program $\Pi_J^{\mathit{final}'} = \{r_1, r_2, r_3\}$—John remains a maximal knowledge subset which is consistent with Sue's condition without being involved in Sue's personal feeling about Tom.

The formal notion of forgetting in propositional theories was initially considered by Lin and Reiter from a cognitive robotics perspective [18] and has recently received a great attention in KR community. It has been shown that the theory of forgetting has important applications in solving knowledge base inconsistencies, belief update and merging, abductive reasoning, causal theories of actions, and reasoning about knowledge under various propositional (modal) logic frameworks, e.g. [13,14,19,24]. Then a natural question is: whether can we develop an analogous theory of forgetting in logic programs and apply it as a foundational basis for various conflict solving in logic programs? This paper provides an answer to this question.

1.2. Summary of contributions of this paper

The main contributions of this paper can be summarized as follows.

(1) We define two notions of strong and weak forgettings in logic programs under answer set programming semantics. We reveal their close connections to the notion of forgetting in classical propositional theories, and provide a precise semantic characterization for them.
(2) Based on these notions, we develop a general framework for conflict solving called *logic program contexts*. Under this framework, conflicts can be solved by strongly or/and weakly forgetting certain sets of atoms from corresponding programs. We show that our framework is general enough to represent many important conflict solving problems. In particular, for the first time we demonstrate that all major logic program update approaches can be transformed into our framework.
(3) We investigate essential computational properties in relation to strong and weak forgettings and conflict solving in the proposed framework. Specifically, we show that under the answer set programming with no disjunction in the head, the associated inference problem for strong and weak forgettings is coNP-complete, and the irrelevance problem related to strong and weak forgettings and conflict solving is coDP-complete. We also study other computational problems related to the computation of strong and weak forgetting and conflict solving.

1.3. Structure of the paper

The rest of this paper is organized as follows. We first present preliminary definitions and concepts in Section 2. In Section 3, we give formal definitions of strong and weak forgettings in logic programs, and present their essential properties. Based on notions of strong and weak forgettings, in Section 4 we propose a framework called logic program contexts for general conflict solving in logic programs. In Section 5, we investigate various semantic properties and features in relation to strong and weak forgettings and conflict solving in the proposed framework. In Section 6, we show that our conflict solving framework is general enough to represent all major logic program update approaches. In Section 7, we study essential computational properties of strong and weaking forgettings and conflict solving. Finally, in Section 8 we conclude the paper with some discussions.

2. Preliminaries

We consider finite propositional normal logic programs in which each rule is of the form:

$$a \leftarrow b_1, \ldots, b_m, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_n, \tag{1}$$

where a is either a propositional atom or empty, $b_1, \ldots, b_m, c_1, \ldots, c_n$ are propositional atoms, and *not* presents the negation as failure. From (1) we know that a normal logic program does not contain classical negation and has no disjunction in the head. When a is empty, rule (1) is called a *constraint*. Given a rule r of the form (1), we denote $\mathit{head}(r) = \{a\}$, $\mathit{pos}(r) = \{b_1, \ldots, b_m\}$, $\mathit{neg}(r) = \{c_1, \ldots, c_n\}$, and $\mathit{body}(r) = \mathit{pos}(r) \cup \mathit{neg}(r)$. Therefore, rule (1) may simply be represented as the form:

$$\mathit{head}(r) \leftarrow \mathit{pos}(r), \mathit{not}\ \mathit{neg}(r), \tag{2}$$

here we denote $\mathit{not}\ \mathit{neg}(r) = \{\mathit{not}\ c_1, \ldots, \mathit{not}\ c_n\}$. We also use $\mathit{atom}(r)$ to denote the set of all atoms occurring in rule r. For a program Π, we define notions $\mathit{head}(\Pi) = \bigcup_{r\in\Pi} \mathit{head}(r)$, $\mathit{pos}(\Pi) = \bigcup_{r\in\Pi} \mathit{pos}(r)$, $\mathit{neg}(\Pi) = \bigcup_{r\in\Pi} \mathit{neg}(r)$, $\mathit{body}(\Pi) = \bigcup_{r\in\Pi} \mathit{body}(r)$, and $\mathit{atom}(\Pi) = \bigcup_{r\in\Pi} \mathit{atom}(r)$. Given sets of atoms P and Q, we may use notion

$$r'\colon\ \mathit{head}(r) \leftarrow \big(\mathit{pos}(r) - P\big), \mathit{not}\big(\mathit{neg}(r) - Q\big)$$

to denote rule r' obtained from r by removing all atoms occurring in P and Q in the positive and negation as failure parts respectively.

The stable model of a program Π is defined as follows. Firstly, we consider Π to be a program in which each rule does not contain negation as failure *not*. A finite set S of propositional atoms is called a *stable model* of Π if S is the smallest set such that for each rule $a \leftarrow b_1, \ldots, b_m$ from Π, if $b_1, \ldots, b_m \in S$, then $a \in S$. Now let Π be an arbitrary normal logic program. For any set S of atoms, program Π^S is obtained from Π by deleting (1) each rule from Π that contains $\mathit{not}\ c$ in the body if $c \in S$; and (2) all subformulas of $\mathit{not}\ c$ in the bodies of the remaining rules. Then S is a stable model of Π if and only if S is a stable model of Π^S [7]. We also call Π^S is the result of Gelfond–Lifschitz transformation on Π with S. It is easy to see that a program may have one, more than one, or no stable models at all. A program is called *consistent* if it has a stable model. We say that an atom *a is entailed* from program Π, denoted as $\Pi \models a$ if a is in every stable model of Π.

Two programs Π_1 and Π_2 are *equivalent* if Π_1 and Π_2 have the same stable models. Π_1 and Π_2 are called *strongly equivalent* if for every program Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ are equivalent [17]. The concept of strong equivalence can be used to simplify a program. For example, if two programs are strongly equivalent, then whenever one program is contained in a particular program, it can be replaced by the other program safely. The following observation gives two instances for this case which will be useful for our later formalization.

Observation 1. *Let Π be a logic program Π. Then Π is strongly equivalent to the empty set iff each rule r in Π is of one of the following two forms*: (1) $\mathit{head}(r) \neq \emptyset$ *and* $\mathit{head}(r) \subseteq \mathit{pos}(r)$, *or* (2) $\mathit{pos}(r) \cap \mathit{neg}(r) \neq \emptyset$.[1]

[1] This result can be viewed as a special case of more general results proved in [9] and [20] respectively.

For convenience in the later reference in this paper, we call the two types of rules mentioned above *valid rules*.

Let Π be a logic program. We use $[\Pi]^C$ to denote the conjunctive normal form obtained from Π by translating each rule of the form (1) in Π into the clause: $a \vee \neg b_1 \vee \cdots \vee \neg b_m \vee c_1 \vee \cdots \vee c_m$. Note that this is not a translation in a classical sense since here we replace negation as failure *not* with classical negation $\neg$. For instance, if $\Pi = \{a \leftarrow \mathit{not}\ b, c \leftarrow a\}$, then we have $[\Pi]^C = (a \vee b) \wedge (c \vee \neg a)$. In general, we may write $[\Pi]^C = \{C_1, \ldots, C_n\}$ where each C_i is a conjunct of $[\Pi]^C$. If C is a clause, we call any subformula of C a *subclause* of C.

Now we introduce the notion of forgetting in a classical propositional theory [18,19]. Let T be a propositional theory. We use $T[p/\mathit{true}]$ (or $T[p/\mathit{false}]$, resp.) to denote the theory obtained from T by substituting all occurrences of propositional atom p with *true* (or *false*, resp.). For instance, if $T = \{p \supset q, (q \wedge r) \supset s\}$, then $T[q/\mathit{true}] = \{r \supset s\}$ and $T[q/\mathit{false}] = \{\neg p\}$.[2] Then we can define the notion of forgetting in terms of a propositional theory. For a given propositional theory T and a set of propositional atoms P, the result of *forgetting* P in T, denoted as $\mathit{Forget}(T, P)$, is defined inductively as follows:

$$\mathit{Forget}(T, \emptyset) = T,$$
$$\mathit{Forget}\big(T, \{p\}\big) = T[p/\mathit{true}] \vee T[p/\mathit{false}],$$
$$\mathit{Forget}\big(T, P \cup \{p\}\big) = \mathit{Forget}\big(\mathit{Forget}(T, p), P\big).$$

It is easy to see that the ordering in which atoms in P are considered does not affect the final result of forgetting P from T. Consider $T = \{p \supset q, (q \wedge r) \supset s\}$ again. From the above definition, we have $\mathit{Forget}(T, \{q\}) = \{(r \supset s) \vee \neg p\}$.

3. Strong and weak forgettings in logic programs

3.1. Definitions

Let us consider how to forget a set of atoms from a logic program. Intuitively, we would expect that after forgetting a set of atoms, all occurrences of these atoms in the underlying program should be eliminated in some way. Those atoms having certain connections to forgotten atoms through rules in the program might or might not be affected depending on the situation, while all other atoms should not be affected. We observe that the forgetting definition in propositional theories cannot be directly used for logic programs as logic programs themselves cannot be disjuncted together. Further, different ways of handling negation as failure in forgetting may also lead to different resulting programs.

For example, suppose we have a program Π containing two rules:

$$a \leftarrow b,$$
$$b \leftarrow c.$$

Now if we want to forget atom b, we can simply remove the second rule and replace the first rule with $a \leftarrow c$. In this case, forgetting b is just to remove b through the rule replacement. However, things become not so simple if we change the program to:

$$a \leftarrow \mathit{not}\ b,$$
$$b \leftarrow c,$$

and we still want to forget atom b. In this case, the method of replacement mentioned above seems not working because replacing the first rule with $a \leftarrow \mathit{not}\ c$ will change the entire semantics of the program. One way we can do is to completely remove the second rule since b is forgotten, and the first rule may be either reduced to $a \leftarrow$ or completely removed depending on whether we assume b true or false. These two examples actually reflect our intuition of defining forgetting notions in logic programs.

To formalize our idea of forgetting in logic programs, we first introduce a program transformation called *reduction*. The intuition behind reduction may be easily illustrated as follows. Given a program $\Pi = \{p \leftarrow q, p' \leftarrow p, \mathit{not}\ q'\}$, performing a reduction on Π with respect to atom p will result in a new program $\Pi' = \{p' \leftarrow q, \mathit{not}\ q'\}$. The formal definition is presented as follows.

[2] For convenience, we may consider a finite set of formulas as a single conjunction of all elements in the set.

Definition 1 *(Program reduction).* Let Π be a program and p an atom. We define the *reduction* of Π with respect to p, denoted as $Reduct(\Pi, \{p\})$, to be a program obtained from Π by (1) for each rule r with $head(r) = \{p\}$ and each rule r' with $p \in pos(r')$, replacing r' with a new rule r'': $head(r') \leftarrow (pos(r') - \{p\}), pos(r), not\ (neg(r) \cup neg(r'))$; (2) if there is such rule r' in Π and has been replaced by r'' in (1), then removing rule r from the remaining program. Let P be a set of propositional atoms. Then the reduction of Π with respect to P is inductively defined as follows:

$$Reduct(\Pi, \emptyset) = \Pi,$$
$$Reduct\big(\Pi, P \cup \{p\}\big) = Reduct\big(Reduct\big(\Pi, \{p\}\big), P\big).$$

Note that in our program reduction definition, Step (1) is the same as Sakama and Seki's [23] and Brass and Dix's [4] *unfolding* in logic programs. While unfolding is to eliminate positive middle occurrences of an atom in a logic program, the reduction, on other hand, is further to remove those rules with heads of this atom. Now let us consider a program $\Pi = \{a \leftarrow b, b \leftarrow a, d \leftarrow not\ e\}$. Then

$$Reduct\big(Reduct\big(\Pi, \{a\}\big), \{b\}\big) = \{b \leftarrow b, d \leftarrow not\ e\}, \quad \text{and}$$
$$Reduct\big(Reduct\big(\Pi, \{b\}\big), \{a\}\big) = \{a \leftarrow a, d \leftarrow not\ e\}.$$

A brief glimpse of this example seems to indicate that the program reduction is not well defined since these two programs look different. However, it is easy to see that they are strongly equivalent, and both can be simplified to $\{d \leftarrow not\ e\}$. The following proposition actually shows that our program reduction is well defined under the strong equivalence.

Proposition 1. *Let Π be a logic program and p, q two propositional atoms. Then $Reduct(Reduct(\Pi, \{p\}), \{q\})$ is strongly equivalent to $Reduct(Reduct(\Pi, \{q\}), \{p\})$.*

Proof. To prove this result, we need to consider a general case of iterated reductions which captures all possible features. For this purpose, it is sufficient to deal with a program $\Pi = \Pi_1 \cup \Pi_2$, where all possible reductions related to atoms p and q are only happened within Π_1. That is, we can assume Π_1 consists of six parts: $\Pi_{11} \cup \Pi_{12} \cup \Pi_{13} \cup \Pi_{14} \cup \Pi_{15} \cup \Pi_{16}$:

Π_{11}:

r_1: $p \leftarrow pos(r_1), not\ neg(r_1)$,

$\ldots$,

r_h: $p \leftarrow pos(r_h), not\ neg(r_h)$,

Π_{12}:

r_{h+1}: $p \leftarrow q, pos(r_{h+1}), not\ neg(r_{h+1})$,

$\ldots$,

r_k: $p \leftarrow q, pos(r_k), not\ neg(r_k)$,

Π_{13}:

r_{k+1}: $q \leftarrow pos(r_{k+1}), not\ neg(r_{k+1})$,

$\ldots$,

r_l: $q \leftarrow pos(r_l), not\ neg(r_l)$,

Π_{14}:

r_{l+1}: $q \leftarrow p, pos(r_{l+1}), not\ neg(r_{l+1})$,

$\ldots$,

r_m: $q \leftarrow p, pos(r_m), not\ neg(r_m)$,

Π_{15}:

r_{m+1}: $a_{m+1} \leftarrow p, pos(r_{m+1}), not\ neg(r_{m+1})$,

$\dots$,

r_n: $a_n \leftarrow p, pos(r_n), not\ neg(r_n)$,

Π_{16}:

r_{n+1}: $b_{n+1} \leftarrow q, pos(r_{n+1}), not\ neg(r_{n+1})$,

$\dots$,

r_s: $b_s \leftarrow q, pos(r_s), not\ neg(r_s)$,

where $a_i \neq p$, $a_i \neq q$, $b_j \neq p$, and $b_j \neq q$ for all a_i and b_j, and also p, q do not occur in all $pos(r_i)$ ($i = 1, \dots, s$). We assume that p, q are not in $head(\Pi_2)$ and $pos(\Pi_2)$, i.e. no reduction related to p or q will occur in Π_2.

It is not hard to see that the above Π covers all possible cases of reductions of Π with respect to atoms p and q. In order to avoid a tedious proof, without loss of generality, we may consider a simplified version of program Π as follows. $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 contains the following rules:

r_1: $p \leftarrow pos(r_1), not\ neg(r_1)$,

r'_1: $p \leftarrow q, pos(r'_1), not\ neg(r'_1)$,

r_2: $q \leftarrow pos(r_2), not\ neg(r_2)$,

r'_2: $q \leftarrow p, pos(r'_2), not\ neg(r'_2)$,

r_3: $a \leftarrow p, pos(r_3), not\ neg(r_3)$,

r_4: $b \leftarrow q, pos(r_4), not\ neg(r_4)$.

We assume p and q do not occur in $pos(r_1), pos(r'_1), pos(r_2)$ and $pos(r'_2)$. Also, all rules in Π_2 do not contain p or q in their heads and positive bodies. We should mention that our following proof can be extended to the general case of Π as constructed earlier.

Firstly, we have $Reduct(\Pi, \{p\}) = \Pi'_1 \cup \Pi_2$ consists of the following rules:

r_2: $q \leftarrow pos(r_2), not\ neg(r_2)$,

r'_2: $q \leftarrow (pos(r_1) \cup pos(r'_2)), not(neg(r_1) \cup neg(r'_2))$,

r''_2: $q \leftarrow q, (pos(r'_1) \cup pos(r'_2)), not\ (neg(r'_1) \cup neg(r'_2))$,

r'_3: $a \leftarrow (pos(r_1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r''_3: $a \leftarrow q, (pos(r'_1) \cup pos(r_3)), not(neg(r'_1) \cup neg(r_3))$,

r_4: $b \leftarrow q, pos(r_4), not\ neg(r_4)$.

From Observation 1, we know that $\{r''_2\}$ is strongly equivalent to the empty set. So we have $\Pi'_1 = \{r_2, r'_2, r'_3, r''_3, r_4\}$. Then by the reduction of $\Pi'_1 \cup \Pi_2$ with respect to $\{q\}$, we have the following result: $Reduct(Reduct(\Pi, \{p\}), \{q\}) = \Pi''_1 \cup \Pi_2$, where Π''_1 contains the following rules:

r'_3: $a \leftarrow (pos(r1) \cup pos(r_3)), not(neg(r_1) \cup neg(r_3))$,

r^*: $a \leftarrow (pos(r1') \cup pos(r_2) \cup pos(r_3)), not(neg(r'_1) \cup neg(r_2) \cup neg(r_3))$,

$r^{*'}$: $a \leftarrow (pos(r1) \cup pos(r1') \cup pos(r'_2) \cup pos(r_3)), not\ (neg(r_1) \cup neg(r'_1) \cup neg(r'_2) \cup neg(r_3))$,

r'_4: $b \leftarrow (pos(r_2) \cup pos(r_4)), not(neg(r_2) \cup neg(r_4))$,

r''_4: $b \leftarrow (pos(r_1) \cup pos(r'_2) \cup pos(r_4)), not(neg(r_1) \cup neg(r_2) \cup neg(r_4))$.

It is easy to see that programs $\{r'_3, r^{*'}\}$ and $\{r'_3\}$ are strongly equivalent because $pos(r'_3) \subseteq pos(r^{*'})$ and $neg(r'_3) \subseteq neg(r^{*'})$. Therefore, rule $r^{*'}$ can be removed. So finally, we have $\Pi''_1 = \{r'_3, r^*, r'_4, r''_4\}$.

Now we consider $\mathit{Reduct}(\Pi, \{q\})$. It is easy to see that $\mathit{Reduct}(\Pi, \{q\}) = \Pi^\dagger \cup \Pi_2$, where $\Pi^\dagger$ consists of the following rules:

$$\begin{aligned}
&r_1\colon\ p \leftarrow pos(r_1), not\ neg(r_1),\\
&r_1''\colon\ p \leftarrow \big(pos(r_1') \cup pos(r_2)\big), not\big(neg(r_1') \cup neg(r_2)\big),\\
&r_1'''\colon\ p \leftarrow p, \big(pos(r_1') \cup pos(r_2')\big), not\big(neg(r_1') \cup neg(r_2')\big),\\
&r_3\colon\ a \leftarrow p, pos(r_3), not\ neg(r_3),\\
&r_4'\colon\ b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),\\
&r_4''\colon\ b \leftarrow p, \big(pos(r_2') \cup pos(r_4)\big), not\big(neg(r_2') \cup neg(r_4)\big).
\end{aligned}$$

Also, rule r_1''' can be removed from $\Pi^\dagger$. So we have $\Pi^\dagger = \{r_1, r_1'', r_3, r_4', r_4''\}$. Then $\mathit{Reduct}(\mathit{Reduct}(\Pi, \{q\}), \{p\}) = \Pi^\ddagger \cup \Pi_2$, where $\Pi^\ddagger$ consists of the following rules:

$$\begin{aligned}
&r_3'\colon\ a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),\\
&r^*\colon\ \leftarrow \big(pos(r_1') \cup pos(r_2) \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_2) \cup neg(r_3)\big),\\
&r_4'\colon\ b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),\\
&r_4''\colon\ b \leftarrow \big(pos(r_1) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2') \cup neg(r_4)\big),\\
&r^\dagger\colon\ b \leftarrow \big(pos(r_1) \cup pos(r_2) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2) \cup neg(r_2') \cup neg(r_4)\big).
\end{aligned}$$

Since $pos(r_4'') \subseteq pos(r_\dagger)$, we know that programs $\{r_4'', r^\dagger\}$ and $\{r^\dagger\}$ are strongly equivalent. So $r^\dagger$ can be removed from $\Pi^\ddagger$. Therefore, $\Pi^\ddagger = \{r_3', r^*, r_4', r_4''\} = \Pi_1''$. This proves our result. □

Example 1. Let $\Pi_1 = \{a \leftarrow not\ b,\ a \leftarrow d,\ c \leftarrow a, not\ e\}$, $\Pi_2 = \{a \leftarrow c, not\ b,\ c \leftarrow not\ d\}$, and $\Pi_2 = \{a \leftarrow b,\ b \leftarrow not\ d,\ c \leftarrow a, not\ e\}$. Then $\mathit{Reduct}(\Pi_1, \{a\}) = \{c \leftarrow not\ b, not\ e,\ c \leftarrow d, not\ e\}$, $\mathit{Reduct}(\Pi_2, \{a\}) = \Pi_2$, and $\mathit{Reduct}(\Pi_3, \{a, b\}) = \{c \leftarrow not\ d, not\ e\}$.

Definition 2 *(Strong forgetting).* Let Π be a logic program, and p a propositional atom. We define a program to be the result of *strongly forgetting* p in Π, denoted as $\mathit{SForgetLP}(\Pi, \{p\})$, if it is obtained from the following transformation:

(1) $\Pi' = \mathit{Reduct}(\Pi, \{p\})$;
(2) $\Pi' = \Pi' - \{r \mid r$ is a valid rule$\}$;
(3) $\Pi' = \Pi' - \{r \mid head(r) = \{p\}\}$;
(4) $\Pi' = \Pi' - \{r \mid p \in pos(r)\}$;
(5) $\Pi' = \Pi' - \{r \mid p \in neg(r)\}$;
(6) $\mathit{SForgetLP}(\Pi, \{p\}) = \Pi'$.

Let us take a closer look at Definition 2. Step 1 is just to perform reduction on Π with respect to atom p. This is to replace those *positive middle occurrences* of p in rules with other rules having p as the head. Step 2 is to remove all valid rules which may be introduced by the reduction of Π with respect to p. From Observation 1, we know that this does not change anything in the program. Steps 3 and 4 are to remove those rules which have p as the head or in the positive body. Note that after reduction, there does not exist any pair of rules r and r' such that $head(r) = \{p\}$ and $p \in pos(r')$. The intuitive meaning of these two steps is that after forgetting p, any atom's information in rules having p as their heads or positive bodies will be lost because they are all relevant to p, i.e. these atoms either serve as a support for p or p is in part of the supports for these atoms. On the other hand, Step 5 states that any rule containing p in its negation as failure part will be also removed. The consideration for this step is as follows. If we think $neg(r)$ as a part of support of $head(r)$, then when $p \in neg(r)$ is forgotten, $head(r)$'s entire support is lost as well. Clearly, such treatment of negation as failure in forgetting is quite strong in the sense that more atoms may be lost together with *not* p. Therefore we call this kind of forgetting *strong forgetting*.

Definition 2 can be easily extended to the case of strongly forgetting a set of atoms:

$$SForgetLP(\Pi, \emptyset) = \Pi,$$
$$SForgetLP\big(\Pi, P \cup \{p\}\big) = SForgetLP\big(SForgetLP\big(\Pi, \{p\}\big), P\big).$$

With a different way of dealing with negation as failure, we have a weak version of forgetting as defined below.

Definition 3 *(Weak forgetting).* Let Π be a logic program, and p a propositional atom. We define a program to be the result of *weakly forgetting* p in Π, denoted as $WForgetLP(\Pi, \{p\})$, if it is obtained from the following transformation:

(1) $\Pi' = Reduct(\Pi, \{p\})$;
(2) $\Pi' = \Pi' - \{r \mid r \text{ is a valid rule}\}$;
(3) $\Pi' = \Pi' - \{r \mid head(r) = \{p\}\}$;
(4) $\Pi' = \Pi' - \{r \mid p \in pos(r)\}$;
(5) $\Pi' = \Pi' - \Pi^* \cup \Pi^\dagger$, where $\Pi^* = \{r \mid p \in neg(r)\}$ and
$\Pi^\dagger = \{r' \mid r'\colon\ head(r) \leftarrow pos(r), not(neg(r) - \{p\}) \text{ where } r \in \Pi^*\}$;
(6) $WForgetLP(\Pi, \{p\}) = \Pi'$.

$WForgetLP(\Pi, \{p\})$ is defined in the same way as $SForgetLP(\Pi, \{p\})$ except Step 5. Suppose we have a rule like $r\colon\ b \leftarrow pos(r), not\ neg(r)$ where $p \in neg(r)$. Instead of viewing $neg(r)$ as part of the support of $head(r)$, we may treat it as a default evidence of $head(r)$, i.e. under the condition of $pos(r)$, if all atoms in $neg(r)$ are not presented, then $head(r)$ can be derived. Therefore, forgetting p will result in the absence of p in any case. So r may be replaced by $r'\colon\ b \leftarrow pos(r), not(neg(r) - \{p\})$. The notion of weakly forgetting a set of atoms, denoted as $WForgetLP(\Pi, P)$, is defined accordingly:

$$WForgetLP(\Pi, \emptyset) = \Pi,$$
$$WForgetLP\big(\Pi, P \cup \{p\}\big) = WForgetLP\big(WForgetLP\big(\Pi, \{p\}\big), P\big).$$

The following proposition ensures that our strong and weak forgettings in logic programs are well defined under strong equivalence.

Proposition 2. *Let Π be a logic program and p, q two propositional atoms. Then*

(1) *$SForgetLP(SForgetLP(\Pi, \{p\}), \{q\})$ is strongly equivalent to $SForgetLP(SForgetLP(\Pi, \{q\}), \{p\})$; and*
(2) *$WForgetLP(WForgetLP(\Pi, \{p\}), \{q\})$ is strongly equivalent to $WForgetLP(WForgetLP(\Pi, \{q\}), \{p\})$.*

Proof. We only prove Result 1, as Result 2 is proved in a similar way. Similar to the proof of Proposition 1, without loss of generality, we consider a program $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 contains the following rules:

$$r_1\colon\ p \leftarrow pos(r_1), not\ neg(r_1),$$
$$r_1'\colon\ p \leftarrow q, pos(r_1'), not\ neg(r_1'),$$
$$r_2\colon\ q \leftarrow pos(r_2), not\ neg(r_2),$$
$$r_2'\colon\ q \leftarrow p, pos(r_2'), not\ neg(r_2'),$$
$$r_3\colon\ a \leftarrow p, pos(r_3), not\ neg(r_3),$$
$$r_4\colon\ b \leftarrow q, pos(r_4), not\ neg(r_4).$$

We assume p and q do not occur in $pos(r_1), pos(r_1'), pos(r_2)$ and $pos(r_2')$. Also, all rules in Π_2 do not contain p or q in their heads and positive bodies, but may contain $not\ p$ or $not\ q$.

Then we have $Reduct(\Pi, \{p\}) = Reduct(\Pi_1, \{p\}) \cup \Pi_2$, where, according to the proof of Proposition 1, $Reduct(\Pi_1, \{p\})$ consists of the following rules:

$$r_2\colon\ q \leftarrow pos(r_2), not\ neg(r_2),$$
$$r_2'\colon\ q \leftarrow \big(pos(r_1) \cup pos(r_2')\big), not\big(neg(r_1) \cup neg(r_2')\big),$$

$$\begin{aligned}
&r_2''\colon\ q \leftarrow q, \big(pos(r_1') \cup pos(r_2')\big), not\big(neg(r_1') \cup neg(r_2')\big),\\
&r_3'\colon\ a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),\\
&r_3''\colon\ a \leftarrow q, \big(pos(r_1') \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_3)\big),\\
&r_4\colon\ b \leftarrow q, pos(r_4), not\ neg(r_4).
\end{aligned}$$

Then after Step 2 (removing valid rules), rule r_2'' is removed. So we can write $SForgetLP(\Pi, \{p\}) = \Pi_1' \cup \Pi_2'$, where $\Pi_1' = \{r_2, r_2', r_3', r_3'', r_4\}$, and $\Pi_2' \subseteq \Pi_2$ in which all rules containing *not p* are removed. Note that rules in Π_1' may be removed if they contain *not p*, according to Step 5 in the transformation.

Now we consider $SForgetLP(\Pi_1' \cup \Pi_2', \{q\})$. Since Π_2' does not contain any rule having q in its head or positive body, $Reduct(\Pi_1' \cup \Pi_2', \{q\}) = Reduct(\Pi_1', \{q\}) \cup \Pi_2'$. By ignoring the details, we will have the final resulting program: $SForgetLP(\Pi_1' \cup \Pi_2', \{q\}) = \Pi_1'' \cup \Pi_2''$, where Π_1'' consists of the following rules:

$$\begin{aligned}
&r_3'\colon\ a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),\\
&r^*\colon\ a \leftarrow \big(pos(r_1') \cup pos(r_2) \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_2) \cup neg(r_3)\big),\\
&r_4'\colon\ b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),\\
&r_4''\colon\ b \leftarrow \big(pos(r_1) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2) \cup neg(r_4)\big),
\end{aligned}$$

and $\Pi_2'' \subseteq \Pi_2'$ in which all rules containing *not q* are removed. Again, rules among $\{r_3', r^*, r_4', r_4''\}$ will be removed if they contain *not q*. Let us denote the resulting program after such elimination as $\Pi_1^{*'}$, i.e. $\Pi_1^{*'} \subseteq \Pi_1''$ where each rule in Π_1'' containing *not p* or *not q* is removed from $\Pi_1^{*'}$.

Let us examine the result of $SForgetLP(SForgetLP(\Pi, \{q\}), \{p\})$. Firstly, we have $Reduct(\Pi, \{q\}) = Reduct(\Pi_1, \{q\}) \cup \Pi_2$, where $Reduct(\Pi_1, \{q\})$ consists of the following rules:

$$\begin{aligned}
&r_1\colon\ p \leftarrow pos(r_1), not\ neg(r_1),\\
&r_1''\colon\ p \leftarrow \big(pos(r_1') \cup pos(r_2)\big), not\big(neg(r_1') \cup neg(r_2)\big),\\
&r_1'''\colon\ p \leftarrow p, \big(pos(r_1') \cup pos(r_2')\big), not\big(neg(r_1') \cup neg(r_2')\big),\\
&r_3\colon\ a \leftarrow p, pos(r_3), not\ neg(r_3),\\
&r_4'\colon\ b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),\\
&r_4''\colon\ b \leftarrow p, \big(pos(r_2') \cup pos(r_4)\big), not\big(neg(r_2') \cup neg(r_4)\big).
\end{aligned}$$

Again, after Step 2, rule r_1''' is removed. So we can write $SForgetLP(\Pi, \{q\}) = \Pi_1^* \cup \Pi_2^*$, where $\Pi_1^* = \{r_1, r_1'', r_3, r_4', r_4''\}$, and $\Pi_2^* \subseteq \Pi_2$ in which all rules containing *not q* are removed. Also rules in Π_1^* will be removed if they contain *not q*.

Now we consider $SForgetLP(\Pi_1^* \cup \Pi_2^*, \{p\})$. Since Π_2^* does not contain any rule having p in its head or positive body, $Reduct(\Pi_1^* \cup \Pi_2^*, \{p\}) = Reduct(\Pi_1^*, \{p\} \cup \Pi_2^*)$. Then we have $Reduct(\Pi_1^*, \{p\}) = \Pi_1^{*'}$, which has the following rules:

$$\begin{aligned}
&r_3'\colon\ a \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big),\\
&r^*\colon\ \leftarrow \big(pos(r_1') \cup pos(r_2) \cup pos(r_3)\big), not\big(neg(r_1') \cup neg(r_2) \cup neg(r_3)\big),\\
&r_4'\colon\ b \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),\\
&r_4''\colon\ b \leftarrow \big(pos(r_1) \cup pos(r_2') \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r_2') \cup neg(r_4)\big).
\end{aligned}$$

This program is Π_1'' as we have shown above. Also note that rules in $\{r_3', r^*, r_4', r_4''\}$ will be removed if they contain *not p* according to Step 5. Then clearly, such resulting program is $\Pi_1^{*'}$ as mentioned above.

So after performing Steps 2–5, we finally have $SForgetLP(\Pi_1^* \cup \Pi_2^*, \{p\}) = \Pi_1^{*'} \cup \Pi_2^{*'}$, where $\Pi_2^{*'} \subseteq \Pi_2^*$, in which all rules containing *not p* are removed. Obviously, $\Pi_2^{*'} = \Pi_2''$. This proves our result. $\square$

Example 2. Let $\Pi_1 = \{a \leftarrow not\ b, b \leftarrow not\ a\}$, and $\Pi_2 = \{a \leftarrow b, not\ c, a \leftarrow not\ e, d \leftarrow a, e, e \leftarrow not\ a\}$. Then

$$\begin{aligned}
&SForgetLP\big(\Pi_1, \{a\}\big) = \emptyset,\\
&SForgetLP\big(\Pi_1, \{a, b\}\big) = \emptyset,\\
&WForgetLP\big(\Pi_1, \{a\}\big) = \{b \leftarrow\},\\
&WForgetLP\big(\Pi_1, \{a, b\}\big) = \emptyset,\\
&SForgetLP\big(\Pi_2, \{a\}\big) = \{d \leftarrow b, e, not\ c\},\\
&WForgetLP\big(\Pi_2, \{a\}\big) = \{d \leftarrow b, e, not\ c, e \leftarrow\}.
\end{aligned}$$

3.2. Relationship to forgetting in propositional theories

As we argued earlier, the notion of forgetting in propositional theories is not applicable to logic programs generally. However, as we will show next, there are close connections between forgetting in propositional theories and strong and weak forgettings in logic programs. Let us first consider the following example.

Example 3. Let $\Pi = \{b \leftarrow a, c, d \leftarrow not\ a, e \leftarrow not\ f\}$. Then we have

$$\begin{aligned}
&SForgetLP\big(\Pi, \{a\}\big) = \{e \leftarrow not\ f\}, \quad \text{and}\\
&WForgetLP\big(\Pi, \{a\}\big) = \{d \leftarrow, e \leftarrow not\ f\}.
\end{aligned}$$

Now we consider $Forget([\Pi]^C, \{a\})$, which is logically equivalent to formula $(b \vee \neg c \vee d) \wedge (f \vee e)$. Then it is clear that

$$\begin{aligned}
&\models Forget\big([\Pi]^C, \{a\}\big) \supset \big[SForgetLP\big(\Pi, \{a\}\big)\big]^C, \quad \text{and}\\
&\models \big[WForgetLP\big(\Pi, \{a\}\big)\big]^C \supset Forget\big([\Pi]^C, \{a\}\big).
\end{aligned}$$

The above example motivates us to examine deeper connections between strong and weak forgettings in logic programs and forgetting in propositional theories. To begin with, we introduce a useful notion. Let Π be a program and L a clause, i.e. $L = l_1 \vee \cdots \vee l_k$ where each l_i is a propositional literal. We say that L is *Π-coherent* if there exists a subset Π' of Π and a set of atoms $P \subseteq atom(\Pi)$ (P could be empty) such that $[Reduct(\Pi', P)]^C$ is a single clause and L is a subclause of $[Reduct(\Pi', P)]^C$. Intuitively, the coherence notion tries to specify those clauses that are parts of clauses generated from program Π through reduction.

Consider program $\Pi = \{a \leftarrow b, d \leftarrow a, not\ c, e \leftarrow not\ d\}$. Clause $d \vee \neg b$ is Π-coherent, where clause $\neg d \vee e$ is not. Obviously, for each rule $r \in \Pi$, $[\{r\}]^C$ is Π-coherent. The following proposition provides a semantic account for Π-coherent clauses.

Proposition 3. *Let Π be a program and L a Π-coherent clause. Then either $\models [\Pi]^C \supset L$ or $\models L \supset \Phi$ for some clause Φ where $\models [\Pi]^C \supset \Phi$.*

Proof. Note that if L is Π-coherent, then we can find a subset Π' of Π and a set of atoms $P \subseteq atom(\Pi)$, such that $Reduct(\Pi', P)$ only contains one rule r and L is a subclause of $[\{r\}]^C$. Recall that the reduction $Reduct(\Pi', P)$ is just to eliminate positive middle occurrences of P in rules of Π' and remove the rules with heads of P if such positive middle occurrences exist in Π'. Then it is easy to observe that $\models [\Pi]^C \supset [Reduct(\Pi', P)]^C$. If $L = [Reduct(\Pi', P)]^C$, then $\models [\Pi]^C \supset L$. If L is a proper subclause of $[Reduct(\Pi', P)]^C$, then $\models L \supset [Reduct(\Pi', P)]^C$. This proves our result. □

Definition 4. Let Π be a logic program, φ, φ_1 and φ_2 three propositional formulas where φ_1 and φ_2 are in conjunctive normal forms.

(1) φ_1 is called a *consequence* of φ *with respect to Π* if $\models \varphi \supset \varphi_1$ and each conjunct of φ_1 is Π-coherent. φ_1 is a *strongest consequence* of φ *with respect to Π* if φ_1 a consequence of φ with respect to Π and there does not exist another consequence φ_1' of φ ($\varphi_1' \not\equiv \varphi_1$) with respect to Π such that $\models \varphi_1' \supset \varphi_1$.

(2) φ_2 is called a *premiss* of φ *with respect to* Π if $\models \varphi_2 \supset \varphi$ and each conjunct of φ_2 Π-coherent. φ_2 is a *weakest premiss* of φ *with respect to* Π if φ_2 a premiss of φ with respect to Π and there does not exist another premiss φ_2' of φ ($\varphi_2' \not\equiv \varphi_2$) with respect to Π such that $\models \varphi_2 \supset \varphi_2'$.

Example 4. (*Example* 3 *continued*) It is easy to verify that $[\mathit{SForgetLP}(\Pi, \{a\})]^C$ is a strongest consequence of $\mathit{Forget}([\Pi]^C, \{a\})$ and $[\mathit{WForgetLP}(\Pi, \{a\})]^C$ is a weakest premiss of $\mathit{Forget}([\Pi]^C, \{a\})$. In fact, the following theorem confirms that this is always true.

Theorem 1. *Let Π be a logic program and P a set of atoms. Then $[\mathit{SForgetLP}(\Pi, P)]^C$ is a strongest consequence of $\mathit{Forget}([\Pi]^C, P)$ with respect to Π and $[\mathit{WForgetLP}(\Pi, P)]^C$ is a weakest premiss of $\mathit{Forget}([\Pi]^C, P)$ with respect to Π.*

Proof. We only prove the first part of the result, while the second part is proved in a similar way. To simplify our proof, we consider set P to be a singleton, i.e. $P = \{p\}$. The general case can be proved by induction on the size of P. Without loss of generality, we assume program Π is of the following form: $\Pi = \Pi_1 \cup \Pi_2 \cup \Pi_3$, where Π_1 only contains rules which are related to the process of the reduction of Π with respect to p, Π_2 does not contain any rules containing p in heads or positive bodies (i.e. Π_2 is irrelevant to the reduction process) but contains rules having p in their negative bodies, and Π_3 does not contain any rules having p in their heads, positive or negative bodies. Obviously, Π_3 is irrelevant to the process of strongly forgetting p in Π. In particular, we assume Π_1 and Π_2 have the following forms:

Π_1:

$r_1\colon\ p \leftarrow \mathit{pos}(r_1), \mathit{not}\ \mathit{neg}(r_1),$

$\ldots,$

$r_k\colon\ p \leftarrow \mathit{pos}(r_k), \mathit{not}\ \mathit{neg}(r_k),$

$r_{k+1}\colon\ q_{k+1} \leftarrow p, \mathit{pos}(r_{k+1}), \mathit{not}\ \mathit{neg}(r_{k+1}),$

$\ldots,$

$r_m\colon\ q_m \leftarrow p, \mathit{pos}(r_m), \mathit{not}\ \mathit{neg}(r_m),$

Π_2:

$r_{m+1}\colon\ q_{m+1} \leftarrow \mathit{pos}(r_{m+1}), \mathit{not}\ p, \mathit{not}\ \mathit{neg}(r_{m+1}),$

$\ldots,$

$r_n\colon\ q_n \leftarrow \mathit{pos}(r_n), \mathit{not}\ p, \mathit{not}\ \mathit{neg}(r_n).$

In Π_1, we may assume that p is not in $\mathit{pos}(r_i)$ for $i = 1, \ldots, m$ (otherwise, those rules having p as heads can be omitted from Π_1 according to Observation 1). For Π_2, on the other hand, p is not in $\mathit{pos}(r_j)$ for $j = m+1, \ldots, n$.

Then according to Definition 1, we have $\mathit{Reduct}(\Pi, \{p\}) = \Pi_1' \cup \Pi_2 \cup \Pi_3$, where Π_1' is as follows:

$r_{1,k+1}\colon\ q_{k+1} \leftarrow \mathit{pos}(r_1), \mathit{pos}(r_{k+1}), \mathit{not}\ \mathit{neg}(r_1), \mathit{not}\ \mathit{neg}(r_{k+1}),$

$\ldots,$

$r_{1,m}\colon\ q_m \leftarrow \mathit{pos}(r_1), \mathit{pos}(r_m), \mathit{not}\ \mathit{neg}(r_1), \mathit{not}\ \mathit{neg}(r_m),$

$\ldots,$

$r_{k,k+1}\colon\ q_{k+1} \leftarrow \mathit{pos}(r_k), \mathit{pos}(r_{k+1}), \mathit{not}\ \mathit{neg}(r_k), \mathit{not}\ \mathit{neg}(r_{k+1}),$

$\ldots,$

$r_{k,m}\colon\ q_m \leftarrow \mathit{pos}(r_k), \mathit{pos}(r_m), \mathit{not}\ \mathit{neg}(r_k), \mathit{not}\ \mathit{neg}(r_m).$

Note that p may occur in negative bodies of some rules in Π_1'. However, to simplify our proof, we may consider that no p occurs in negative bodies in all rules of Π_1' because p's occurrences in negative bodies have been presented in the case of Π_2.

Now we consider $SForgetLP(\Pi, \{p\})$. Clearly, $SForgetLP(\Pi, \{p\}) = \Pi_1' \cup \Pi_3$, where Π_2 is removed from Step 5 in Definition 2. Then we conclude that

$$\big[SForgetLP\big(\Pi, \{p\}\big)\big]^C = [\Pi_1']^C \wedge [\Pi_3]^C,$$

where $[\Pi_1']^C$ consists of the following clauses:

$$\Big(q_{k+1} \vee \neg pos(r_1) \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_1) \vee \bigvee neg(r_{k+1})\Big),^{3}$$
$$\ldots$$
$$\Big(q_m \vee \neg pos(r_k) \vee \neg pos(r_m) \vee \bigvee neg(r_k) \vee \bigvee neg(r_m)\Big).$$

Obviously, each clause of $SForgetLP(\Pi, \{p\})$ is Π-coherent.

Now we consider $Forget([\Pi]^C, \{p\})$. Firstly, it is to observe that

$$Forget\big([\Pi]^C, \{p\}\big) = \Phi \wedge [\Pi_3]^C,$$

where Φ is formula $([\Pi_1]^C[p/true] \wedge [\Pi_2]^C[p/true]) \vee ([\Pi_1]^C[p/false] \wedge [\Pi_2]^C[p/false])$. $[\Pi_1]^C[p/true] \wedge [\Pi_2]^C[p/true]$ consists of the following clauses:

$$q_{k+1} \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_{k+1}),$$
$$\ldots,$$
$$q_m \vee \neg pos(r_m) \vee \bigvee neg(r_m),$$

and $[\Pi_1]^C[p/false] \wedge [\Pi_2]^C[p/false]$ contains the following clauses:

$$\neg pos(r_1) \vee \bigvee neg(r_1),$$
$$\ldots,$$
$$\neg pos(r_k) \vee \bigvee neg(r_k),$$
$$q_{m+1} \vee \neg pos(r_{m+1}) \vee \bigvee neg(r_{m+1}),$$
$$\ldots,$$
$$q_n \vee \neg pos(r_n) \vee \bigvee neg(r_n).$$

Then by translating Φ into CNF, say $Con(\Phi)$, it is easy to see that all clauses of $[\Pi_1']^C$ are contained in $Con(\Phi)$. So $[SForgetLP(\Pi, \{p\})]^C$ is a consequence of $Forget([\Pi]^C, \{p\})$ with respect to Π.

Observing $Con(\Phi)$'s structure, we know that $Con(\Phi)$ also contains the following clauses:

$$q_{k+1} \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_{k+1}) \vee q_{m+1} \vee \neg pos(r_{m+1}) \vee \bigvee neg(r_{m+1}),$$
$$\ldots,$$
$$q_{k+1} \vee \neg pos(r_{k+1}) \vee \bigvee neg(r_{k+1}) \vee q_n \vee \neg pos(r_n) \vee \bigvee neg(r_n),$$
$$\ldots,$$
$$q_m \vee \neg pos(r_m) \vee \bigvee neg(r_m) \vee q_n \vee \neg pos(r_n) \vee \bigvee neg(r_n).$$

According to the structure of Π, none of these clauses is Π-coherent. Therefore, there does not exist another consequence φ' of $Forget([\Pi]^C, \{p\})$ with respect to Π such that $\models \varphi' \supset [SForgetLP(\Pi, \{p\})]^C$. This proves our result. □

Theorem 1 actually states that under a certain set of propositional atoms P, the conjunctive normal form of the strong forgetting of P in program Π is the strongest formula which is implied by the forgetting of P in the corresponding propositional theory, while the conjunctive normal form of the weak forgetting of P in Π is the weakest

[3] Here $\neg pos(r)$ presents the disjunction of all negative atoms whose atoms occur in $pos(r)$ and $\bigvee neg(r)$ presents the disjunction of all atoms in $neg(r)$.

formula that implies it. So semantically, our notions of strong and weak forgettings in logic programs are strongest necessary and weakest sufficient conditions respectively for the forgetting in the corresponding propositional theory.

3.3. A semantic characterization

From previous presentation, we can see that our strong and weak forgettings are defined in a syntactic way. This is one of the major differences comparing with the forgetting notion in propositional theories, where an equivalent model theoretic semantics is provided for the resulting theory after forgetting some atoms [19]. Although we do not have corresponding model theoretic definitions for strong and weak forgettings, the following property precisely characterizes the stable models of strong and weak forgettings.

Firstly, we observe that the consistency of program Π does not necessarily imply a consistent $\mathit{SForgetLP}(\Pi, P)$ or $\mathit{WForgetLP}(\Pi, P)$ for some set of atoms P, and *vice versa*. For example, consider program $\Pi = \{a \leftarrow, b \leftarrow \mathit{not}\ a, \mathit{not}\ b\}$, then weakly forgetting a in Π will result in an inconsistent program $\{b \leftarrow \mathit{not}\ b\}$. Similarly, strongly forgetting a from an inconsistent program $\Pi = \{b \leftarrow \mathit{not}\ a,\ c \leftarrow b, \mathit{not}\ c\}$ will get a consistent program $\{c \leftarrow b, \mathit{not}\ c\}$. Theorem 2 explains how this happens.

Given program Π and a set of atoms P, we specify two programs X and Y. Program X is a subset of Π containing three types of rules in Π: (1) for each $p \in P$, if $p \notin \mathit{head}(\Pi)$, then rule $r \in \Pi$ with $p \in \mathit{pos}(r)$ is in X; (2) for each $p \in P$, if $p \notin \mathit{pos}(\Pi)$, then rule $r \in \Pi$ with $\mathit{head}(r) = \{p\}$ is in X; and (3) rule $r \in \Pi$ with $\mathit{neg}(r) \cap P \neq \emptyset$ but not of the types (1) and (2) is also in X. Clearly, X contains those rules of Π satisfying $\mathit{atom}(r) \cap P \neq \emptyset$ but will not be affected by $\mathit{Reduct}(\Pi, P)$. On the other hand, program Y is obtained as follows: for each rule r in X of the type (3), a replacement of r of the form: r': $\mathit{head}(r) \leftarrow \mathit{pos}(r), \mathit{not}(\mathit{neg}(r) - P)$ is in Y. It should be noted that both X and Y can be obtained in linear time in terms of the sizes of Π and P. Then we have the following result.

Theorem 2. *Let Π be a program and P a set of atoms. A set of atoms S is a stable model of $\mathit{SForgetLP}(\Pi, P)$ (or $\mathit{WForgetLP}(\Pi, P)$ resp.) iff program $\Pi - X$ (or $(\Pi - X) \cup Y$ resp.) has a stable model S' such that $S = S' - P$.*

Proof. From the definition of X, we can see that X contains exactly all those rules of Π that are not affected by $\mathit{Reduct}(\Pi, P)$ but have to be removed from $\mathit{SForgetLP}(\Pi, P)$. So we have $\mathit{SForgetLP}(\Pi, P) = \mathit{Reduct}(\Pi, P) - X = \mathit{Reduct}((\Pi - X), P)$ (we suppose that no valid rule is presented here as it does not influence the result). So it is easy to see that $\mathit{SForgetLP}(\Pi, P)$ has a stable model S iff $\Pi - X$ has a stable model S' where $S = S' - P$. Similarly, we can observe that $\mathit{WForgetLP}(\Pi, P) = \mathit{Reduct}((\Pi - X) \cup Y, P)$. $\square$

It is interesting to note that given program Π and set of atoms P, although computing $\mathit{SForgetLP}(\Pi, P)$ or $\mathit{WForgetLP}(\Pi, P)$ may need exponential time (see Section 7), its stable models can be computed through some program that is obtained from Π in linear time.

4. Logic program contexts—A framework for conflict solving

In this section, we define a general framework called logic program contexts to represent a knowledge system which consists of multiple agents' knowledge bases. We consider the issue of conflicts occurring in the reasoning within the underlying logic program context. As will be shown, notions of strong and weak forgettings that we proposed earlier will provide a basis for solving such conflicts.

Definition 5 *(Logic program context).* A *logic program context* is an n-ary tuple $\Sigma = (\Phi_1, \dots, \Phi_n)$, where each Φ_i is a triplet $(\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$ – Π_i and $\mathcal{C}_i$ are two logic programs, and $\mathcal{F}_i \subseteq \mathit{atom}(\Pi_i)$ is a set of atoms. We also call each Φ_i the ith *component* of Σ. A logic program context Σ is *consistent* if for each i, $\Pi_i \cup \mathcal{C}_i$ is consistent. Σ is *conflict-free* if for any i and j, $\Pi_i \cup \mathcal{C}_j$ is consistent.

In Definition 5, each component Φ_i in Σ represents agent i's local situation, where Π_i is agent i's knowledge base, $\mathcal{C}_i$ is a set of constraints that agent i should comply and will not change in any case, and $\mathcal{F}_i$ is a set of atoms that agent i may forget if necessary. Now the problem of conflict solving under this setting can be stated as follows: given a logic program context $\Sigma = (\Phi_1, \dots, \Phi_n)$, which may not be consistent or conflict-free, how can we find an alternative logic program context $\Sigma' = (\Phi'_1, \ldots, \Phi'_n)$ such that Σ' is conflict-free *and* is *closest* to the original Σ in some sense.

We first present formal definitions about the solution that solves conflicts in a logic program context.

Definition 6 *(Solution).* Given a logic program context $\Sigma = (\Phi_1, \ldots, \Phi_n)$, where each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$. We call a logic program context Σ' a *solution* that solves conflicts in Σ, if Σ' satisfies the following conditions:

(1) Σ' is conflict-free;
(2) For each Φ'_i in Σ', $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i)$, where $\Pi'_i = \mathit{SForgetLP}(\Pi_i, P_i)$ or $\Pi'_i = \mathit{WForgetLP}(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$.

We denote the set of all solutions of solving conflicts in Σ as $\mathit{Solution}(\Sigma)$.

Definition 7 *(Ordering on solutions).* Given three logic program contexts Σ, Σ' and Σ'' where $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$. We say that Σ' is *closer* or *as close to* Σ *as* Σ'', denoted as $\Sigma' \preceq_\Sigma \Sigma''$, if for each i, $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma'$ and $\Phi''_i = (\Pi''_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma''$, where $\Pi'_i = \mathit{SForgetLP}(\Pi_i, P_i)$ or $\Pi'_i = \mathit{WForgetLP}(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$, and $\Pi''_i = \mathit{SForgetLP}(\Pi_i, Q_i)$ or $\Pi''_i = \mathit{WForgetLP}(\Pi_i, Q_i)$ for some $Q_i \subseteq \mathcal{F}_i$ respectively, we have $P_i \subseteq Q_i \subseteq \mathcal{F}_i$. We denote $\Sigma' \prec_\Sigma \Sigma''$ if $\Sigma' \preceq_\Sigma \Sigma''$ and $\Sigma'' \not\preceq_\Sigma \Sigma'$.

Proposition 4. *$\preceq_\Sigma$ is a partial ordering.*

Proof. From the definition of $\preceq_\Sigma$, it is easy to see that $\preceq_\Sigma$ is reflexive and antisymmetric. So we only need to show $\preceq_\Sigma$ is transitive, and this is obvious according to Definition 7. □

Definition 8 *(Preferred solution).* Given two logic program contexts Σ and Σ'. We say that Σ' is a *preferred solution* that solves conflicts in Σ, if $\Sigma' \in \mathit{Solution}(\Sigma)$ and there does not exist another $\Sigma'' \in \mathit{Solution}(\Sigma)$ such that $\Sigma'' \prec_\Sigma \Sigma'$.

It should be noted that in order to achieve a preferred solution, both strong and weak forgettings may have to apply alternatively. Consider the following simple example.

Example 5. Let $\Sigma = (\Phi_1, \Phi_2)$, where

$$\begin{array}{llll}
\Phi_1: & & \Phi_2: & \\
\Pi_1: & a \leftarrow, & \Pi_2: & c \leftarrow, \\
& b \leftarrow a, \mathit{not}\ c, & & d \leftarrow \mathit{not}\ e, \\
& d \leftarrow a, \mathit{not}\ e, & & e \leftarrow, \\
& f \leftarrow d, & & f \leftarrow d, \\
\mathcal{C}_1: & \leftarrow d, \mathit{not}\ f, & \mathcal{C}_2: & \leftarrow b, \mathit{not}\ c, \\
& \leftarrow f, \mathit{not}\ d, & & \\
\mathcal{F}_1: & \{a, b, c\}, & \mathcal{F}_2: & \{a, b, c, d, e, f\}.
\end{array}$$

It is easy to see that Σ is consistent but not conflict-free because neither $\Pi_1 \cup \mathcal{C}_2$ nor $\Pi_2 \cup \mathcal{C}_1$ is consistent. Now consider two logic program contexts $\Sigma_1 = (\Phi'_1, \Phi'_2)$ and $\Sigma_2 = (\Phi''_1, \Phi''_2)$, where

$$\begin{aligned}
&\Phi'_1 = \big(\mathit{SForgetLP}\big(\Pi_1, \{c\}\big), \mathcal{C}_1, \mathcal{F}_1\big), \\
&\Phi'_2 = \big(\mathit{WForgetLP}\big(\Phi_2, \{e\}\big), \mathcal{C}_2, \mathcal{F}_2\big), \quad \text{and} \\
&\Phi''_1 = \big(\mathit{WForgetLP}\big(\Pi_1, \{b, c\}\big), \mathcal{C}_1, \mathcal{F}_1\big), \\
&\Phi''_2 = \big(\mathit{WForgetLP}\big(\Phi_2, \{e\}\big), \mathcal{C}_2, \mathcal{F}_2\big).
\end{aligned}$$

It can be verified that both Σ_1 and Σ_2 are solutions that solve the conflict in Σ, but only Σ_1 is a preferred solution.

From the above example, we can see that to solve conflicts in a logic program context, the agent may apply strong forgettings, weak forgettings, or both to obtain a (preferred) solution. In this sense, the agent has a freedom to choose the ways of conflict solving if no specific constraint is taken into account. It is noted that sometimes solving conflict through strong forgetting will loose more atoms than weak forgetting, or *vice versa*. Therefore, in order to minimally forget atoms from a logic program, the agent can apply strong and weak forgettings alternatively in different components. However, in practice, it may be more desirable for an agent to use a unified approach in conflict solving. Our approach provided here can certainly accommodate this requirement by simply re-defining the solution of a logic program context by applying strong or weak forgetting only.

Example 6. We consider a conflict solving scenario. A couple John and Mary are discussing their family investment plan. They consider to invest four types of different shares *shareA, shareB, shareC* and *shareD*, where *shareA* and *shareB* are of high risk but also have high returns, and *shareC* and *shareD* are property investment shares and hence are of lower risk and may be suitable for a long term investment. John is very interested in *shareA* and wants to buy it definitely. He also tends to invest *shareB* if they invest neither *shareC* nor *shareD*. However, if they do not invest *shareB*, John may consider to invest *shareC* or *shareD* if the house price will keep growing, which John is actually not sure yet. But John does not consider to invest both of them. On the other hand, Mary is more conservative. She prefers to invest both *shareC* and *shareD* because she believes that the house price will continue growing as she is confident that the government has no plan to increase the Reserve Bank interest. Mary definitely does not consider to invest both *shareA* and *shareB*. At most, she may consider to buy some *shareB* if they invest neither *shareA* nor *shareC*. But Mary insists that they should invest at least one of *shareC* and *shareD* in any case. Now how can John and Mary negotiate to achieve a common agreement?

We first represent John and Mary's investment preferences as the following programs respectively:

Π_J:

r_1: $\mathit{shareA} \leftarrow$,

r_2: $\mathit{shareB} \leftarrow \mathit{not}\ \mathit{shareC}, \mathit{not}\ \mathit{shareD}$,

r_3: $\mathit{shareC} \leftarrow \mathit{houseIncrease}, \mathit{not}\ \mathit{shareB}, \mathit{not}\ \mathit{shareD}$,

r_4: $\mathit{shareD} \leftarrow \mathit{houseIncrease}, \mathit{not}\ \mathit{shareB}, \mathit{not}\ \mathit{shareC}$,

Π_M:

r_5: $\mathit{shareC} \leftarrow \mathit{houseIncrease}$,

r_6: $\mathit{shareD} \leftarrow \mathit{houseIncrease}$,

r_7: $\mathit{shareB} \leftarrow \mathit{not}\ \mathit{shareA}, \mathit{not}\ \mathit{shareC}$,

r_8: $\mathit{houseIncrease} \leftarrow \mathit{not}\ \mathit{interestUp}$.

To negotiate with each other, John and Mary set up their conditions respectively that they do not want to compromise:

$\mathcal{C}_J$:

$\leftarrow \mathit{not}\ \mathit{shareA}$,

$\leftarrow \mathit{shareC}, \mathit{shareD}$, and

$\mathcal{C}_M$:

$\leftarrow \mathit{shareA}, \mathit{shareB}$,

$\leftarrow \mathit{not}\ \mathit{shareC}, \mathit{not}\ \mathit{shareD}$.

John and Mary then specify a logic program context to solve the conflict about their family investment plan: $\Sigma_{JM} = ((\Pi_J, \mathcal{C}_J, F_J), (\Pi_M, \mathcal{C}_M, F_M))$, where $F_J = \{\mathit{shareB}, \mathit{shareC}, \mathit{shareD}\}$ (note that *shareA* is not a forgettable atom for John as he definitely wants to buy it) and $F_M = \{\mathit{shareA}, \mathit{shareB}, \mathit{shareC}, \mathit{shareD}\}$.

Unfortunately, it is easy to check that Σ_{JM} has no (preferred) solution. That means, it is impossible for John and Mary to solve their conflict by just weakening their own belief sets. So John and Mary realize that they have to make further compromise that both of them should not only weaken their own belief sets, but also take the other's beliefs into account. However, their strategy is to take the other's beliefs as little as possible. To this end, John and Mary specify a new logic program context as follows: $\Sigma_{JM}^{New} = ((\Pi_J \cup \Delta_M, C_J, F'_J), (\Pi_M \cup \Delta_J, C_M, F'_M))$, where

Δ_M:

r'_5: $shareC \leftarrow houseIncrease, not\ l_{r'_5}$,

r'_{51}: $l_{r'_5} \leftarrow not\ h_{r'_5}$,

r'_6: $shareD \leftarrow houseIncrease, not\ l_{r'_6}$,

r'_{61}: $l_{r'_6} \leftarrow not\ h_{r'_6}$,

r'_7: $shareB \leftarrow not\ shareA, not\ shareC, not\ l_{r'_7}$,

r'_{71}: $l_{r'_7} \leftarrow not\ h_{r'_7}$,

r'_8: $houseIncrease \leftarrow not\ interestUp, \quad not\ l_{r'_8}$,

r'_{81}: $l_{r'_8} \leftarrow not\ h_{r'_8}$,

Δ_J:

r'_1: $shareA \leftarrow not\ l_{r'_1}$,

r'_{11}: $l_{r'_1} \leftarrow not\ h_{r'_1}$,

r'_2: $shareB \leftarrow not\ shareC, not\ shareD, not\ l_{r'_2}$,

r'_{21}: $l_{r'_2} \leftarrow not\ h_{r'_2}$,

r'_3: $shareC \leftarrow houseIncrease, not\ shareB, not\ shareD, not\ l_{r'_3}$,

r'_{31}: $l_{r'_3} \leftarrow not\ h_{r'_3}$,

r'_4: $shareD \leftarrow houseIncrease, not\ shareB, not\ shareC, not\ h_{r'_4}$,

r'_{41}: $l_{r'_4} \leftarrow not\ h_{r'_4}$,

and $F'_J = F_J \cup \{h_{r'_i}, l_{r'_i} \mid i = 5, \ldots, 8\}$ and $F'_M = F_M \cup \{h_{r'_i}, l_{r'_i} \mid i = 1, \ldots, 4\}$ ($h_{r'_i}, l_{r'_i}$ ($i = 1, \ldots, 8$) are newly introduced atoms).

Let us take a closer look at Δ_M. During the conflict solving, if none of $h_{r'_i}, l_{r'_i}$ ($i = 5, \ldots, 8$) has been strongly or weakly forgotten, then all rules r'_i ($i = 5, \ldots, 8$) in Δ_M equipped with the corresponding rules from Π_M will be defeated. In this case, John does not need to take any of Mary's beliefs into his consideration. On the other hand, if for some j ($5 \leqslant j \leqslant 8$) $h_{r'_j}$ is strongly forgotten (or $l_{r'_j}$ is weakly forgotten), then rules r'_j in Δ_M will be initiated and hence will affect John's decision for conflict solving. As only a minimal number of $h_{r'_i}$ (or $l_{r'_i}$) ($i = 5, \ldots, 8$) will be strongly forgotten (or weakly forgotten, resp.) in the conflict solving, John just takes a minimal number of Mary's rules for his consideration. The same explanation applies for Δ_J.

Σ_{JM}^{New} has a unique preferred solution $((\Pi'_J, C_J, F'_J), (\Pi'_M, C_M, F'_M))$, where

$$\Pi'_J = WForgetLP(\Pi_J \cup \Delta_M, \{shareB, l_{r'_8}\}), \quad \text{and}$$
$$\Pi'_M = WForgetLP(\Pi_M \cup \Delta_J, \{shareC, l_{r'_1}\}).$$

Π'_J has two stable models which include $\{shareA, shareC\}$ and $\{shareA, shareD\}$ respectively, and Π'_M has one stable model including *shareA* and *shareD*. Therefore, John has two options: either to invest *shareA* and *shareC*, or to invest *shareA* and *shareD*, while Mary will only consider to invest *shareA* and *shareD*. Finally, John and Mary can reach an agreement to invest *shareA* and *shareD*.

Example 6 presents an application of our approach to solve complex logic program conflicts involving negotiation and belief merging that most of current methods have difficulties to deal with.

5. Semantic properties

In this section, we study important semantic properties in relation to strong and weak forgettings and logic program contexts.

5.1. Irrelevance

Irrelevance is an important issue related to forgetting [18]. Basically, if we are able to answer an query q against a logic program Π, i.e. $\Pi \models q$, then we are interested in knowing whether we still can answer this query in the resulting program after strongly or weakly forgetting a set of atoms from Π, because this will enable us to significantly simplify the inference problem in the resulting logic program. We first give a formal definition of irrelevance in relation to strong and weak forgetting.

Definition 9 *(Irrelevance).* Let Π be a logic program and P a set of atoms. We say that atom a is *irrelevant* to the strong forgetting (or weak forgetting) of P from Π, or simply say that a is s-*irrelevant* (or w-*irrelevant*, resp.) to P in Π, if $\Pi \models a$ iff $\mathit{SForgetLP}(\Pi, P) \models a$ (or $\mathit{WForgetLP}(\Pi, P) \models a$ resp.). We say that a is *irrelevant* to P in Π if a is either s-irrelevant or w-irrelevant to P in Π.

Trivially, if Π is inconsistent, then a is s-irrelevant (w-irrelevant) to any P in Π iff $\mathit{SForgetLP}(\Pi, P) \models a$ (or $\mathit{WForgetLP}(\Pi, P) \models a$, resp.). Also if for some $P \subseteq \mathit{atom}(\Pi)$, $\mathit{SForgetLP}(\Pi, P)$ ($\mathit{WForgetLP}(\Pi, P)$) is inconsistent, then a is s-irrelevant (or w-irrelevant, resp.) to P in Π iff $\Pi \models a$. To provide a general characterization result for irrelevance, we need a notion of support.

Definition 10. Let Π be a program and a an atom. We define a's *support* with respect to Π to be a set of atoms $\mathit{Support}(a)$ specified as follows:

$$S_0 = \{p \mid p \in \mathit{body}(r) \text{ where } r \in \Pi \text{ and } \mathit{head}(r) = \{a\}\};$$
$$S_{i+1} = S_i \cup \{p \mid p \in \mathit{body}(r) \text{ where } r \in \Pi \text{ and } \mathit{head}(r) \subseteq S_i\};$$
$$\mathit{Support}(a) = \bigcup_{i=0}^{\infty} S_i.$$

An atom $p \in \mathit{Support}(a)$ is called a *positive* (or *negative*) *support* of a if $p \in \mathit{pos}(r)$ (or $\in \mathit{neg}(r)$, resp.) for some rule r occurring in defining $\mathit{Support}(a)$.[4]

Basically, $\mathit{Support}(a)$ contains all atoms that occur in those rules related to a's derivation in program Π. Therefore, changing or removing any rules which contain atoms in $\mathit{Support}(a)$ may affect atom a. It turns out that the notion of support plays an important role in deciding the irrelevance.

Theorem 3. *Let Π be a logic program, P a set of atoms and a an atom. Suppose Π, $\mathit{SForgetLP}(\Pi, P)$ and $\mathit{WForgetLP}(\Pi, P)$ are consistent. Then the following results hold.*

(1) *If $a \notin \mathit{head}(\Pi)$, then a is irrelevant to P in Π;*
(2) *If $a \in P$, then a is irrelevant to P in Π iff $\Pi \not\models a$;*
(3) *If $a \notin P$ and $P \cap \mathit{Support}(a) = \emptyset$, then a is irrelevant to P in Π.*

Proof. Proofs for Results 1 and 2 are trivial. Here we only prove Result 3. To prove this result, we need a result about program splitting from [26]. Before we present this program splitting result, we introduce a notion. Given a program Π and a set of atoms S, we use $e(\Pi, S)$ to denote the program obtained from Π by deleting: (1) each rule in Π having a form *not* a in its body with $a \in S$; and (2) all atoms a in the bodies of the remaining rules with $a \in S$. Intuitively,

[4] Note that an atom in $\mathit{Support}(a)$ could be both positive and negative supports of a.

$e(\Pi, S)$ can be viewed as a simplified form of Π given those atoms in S to be true. Then we can re-state Theorem 5 in [26] under the normal logic program setting:

> A set of atoms S is a stable model of program Π if and only if $\Pi = \Pi_1 \cup \Pi_2$ such that $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, and $S = S_1 \cup S_2$, where S_1 is a stable model of Π_1 and S_2 is a stable model of program $e(\Pi_2, S_1)$.

From the definition of *Support*(a), we can see that Π can be expressed as $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 is the subset of Π containing all rules mentioned in *Support*(a). So we have $\Pi_1 \cap \Pi_2 = \emptyset$. Also, it is observed that $body(\Pi_1) \cap head(\Pi_2) = \emptyset$. Because if this is not true, then there must be some rule $r \in \Pi_2$ such that $body(r) \cap body(\Pi_1) \neq \emptyset$. According to Π_1's construction, this leads to $r \in \Pi_1$ as well. That is, $\Pi_1 \cap \Pi_2 \neq \emptyset$. This is a contradiction. Since $P \cap Support(a) = \emptyset$, it is clear that all rules containing some atoms in P are in Π_2. We may use $\Pi(P)$ to denote this set of rules of Π.

From $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, we know that each stable model S of Π can be expressed as $S = S_1 \cup S_2$, where S_2 is a stable model of program $e(\Pi_2, S_1)$. Also, since rule $r^a \in \Pi_1$, this implies that $\Pi \models a$ iff $\Pi_1 \models a$.

Now from the definitions of strong and weak forgettings and condition $\Pi(P) \subseteq \Pi_2$, we know that both strong and weak forgettings only influence rules in Π_2. So we have

$$SForgetLP(\Pi, P) = \Pi_1 \cup \Pi^{\dagger}, \quad \text{and}$$
$$WForgetLP(\Pi, P) = \Pi_1 \cup \Pi^{\ddagger},$$

where $head(\Pi^{\dagger}) \subseteq head(\Pi_2)$ and $head(\Pi^{\ddagger}) \subseteq head(\Pi_2)$. This follows:

$$\Pi_1 \cap \Pi^{\dagger} = \emptyset, body(\Pi_1) \cap head(\Pi^{\dagger}) = \emptyset, \quad \text{and}$$
$$\Pi_1 \cap \Pi^{\ddagger} = \emptyset, body(\Pi_1) \cap head(\Pi^{\ddagger}) = \emptyset.$$

By the result stated above, we have that each stable model S^s of $SForgetLP(\Pi, P)$ can be expressed as $S^s = S_1 \cup S^{\dagger}$, and each stable model S^w of $WForgetLP(\Pi, P)$ can be expressed as $S^w = S_1 \cup S^{\ddagger}$, where S_1 is a stable model of Π_1, $S^{\dagger}$ and $S^{\ddagger}$ are stable models of $\Pi^{\dagger}$ and $\Pi^{\ddagger}$ respectively.

Finally, from the observation that $\Pi \models a$ iff $\Pi_1 \models a$, we have ($\Pi \models a$ iff $SForgetLP(\Pi, P) \models a$) and ($\Pi \models a$ iff $WForgetLP(\Pi, P) \models a$). This proves our result. □

Theorem 3 provides common conditions under which atom a is both s-irrelevant and w-irrelevant to P in Π. However, we should note that in general, an atom's s-irrelevance does not imply its w-irrelevance, and *vice versa*. Usually we need to deal with these two types of irrelevances separately. The following theorem illustrates different sufficient conditions to ensure these irrelevances respectively.

Theorem 4. *Let Π be a logic program, P a set of atoms and a an atom where $a \notin P$. Suppose that Π, $SForgetLP(\Pi, P)$ and $WForgetLP(\Pi, P)$ are consistent. Then the following results hold*:

(1) *If for each $p \in P \cap Support(a)$, p is a negative support of a and $\Pi \not\models p$, then a is w-irrelevant to P in Π*;
(2) *If for each $p \in P \cap Support(a)$, p is a negative support of a and $\Pi \models p$, then a is s-irrelevant to P in Π.*

Proof. We only prove Result 1, while Result 2 can be proved in a similar way. From the proof of Theorem 3, given *Support*(a), program Π can be expressed as $\Pi = \Pi_1 \cup \Pi_2$, where $\Pi_1 \cap \Pi_2 = \emptyset$, Π_1 contains all rules used in computing *Support*(a), and $\Pi \models a$ iff $\Pi_1 \models a$. Now let us consider $WForgetLP(\Pi, P)$. We will show that $WForgetLP(\Pi, P)$ can be also expressed as $WForgetLP(\Pi, P) = \Pi_1' \cup \Pi_2'$, such that $body(\Pi') \cap head(\Pi_2') = \emptyset$, and $\Pi_1' \models a$ iff $\Pi_1 \models a$.

To simplify our presentation, we may assume $P = \{p\}$ where the proof for the general case can be easily extended from this special case. Without loss of generality, we can consider that $\Pi = \Pi_1 \cup \Pi_2$, where Π_1 includes the following rules in relation to P (note that Π_1 may also contain other rules):

$$r_1\colon\ head(r_1) \leftarrow pos(r_1), not\ p, not\ neg(r_1),$$
$$r_2\colon\ p \leftarrow pos(r_2), not\ neg(r_2),$$
$$r_3\colon\ head(r_3) \leftarrow p, pos(r_3), not\ neg(r_3),$$

and Π_2 includes the following rules related to P (again, Π_2 may contain other rules):

$$r_4\colon\ head(r_4) \leftarrow p, pos(r_4), not\ neg(r_4),$$
$$r_5\colon\ head(r_5) \leftarrow pos(r_5), not\ p, not\ neg(r_5),$$

we should indicate that Π_2 does not contain a rule with head of p, because this rule will be contained in Π_1 as a rule used for computing $Support(a)$.

Clearly, by weakly forgetting $\{p\}$ in Π, only rules r_1–r_5 will be affected, and other rules do remain unchanged. Therefore, we have $WForgetLP(\Pi, \{p\}) = \Pi'_1 \cup \Pi'_2$, where the only difference between Π_1 and Π'_1 are following rules in Π'_1:

$$r'_1\colon\ head(r_1) \leftarrow pos(r_1), not\ neg(r_1),$$
$$r'_3\colon\ head(r_3) \leftarrow \big(pos(r_2) \cup pos(r_3)\big), not\big(neg(r_2) \cup neg(r_3)\big),$$

and the only difference between Π_2 and Π'_2 are the following rules in Π'_2:

$$r'_4\colon\ head(r_4) \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big),$$
$$r'_5\colon\ head(r_5) \leftarrow pos(r_5), not\ neg(r_5).$$

This concludes that $body(\Pi'_1) \cap head(\Pi'_2) = \emptyset$. Now we show that $\Pi'_1 \models a$ iff $\Pi_1 \models a$. Observing that in Π'_1, weakly forgetting p actually does not affect the derivation of $head(r_3)$, while $head(r_1)$'s derivation might be affected since *not p* has been removed from r'_1. However, note that $\Pi \not\models a$, in the original rule r_1 in Π_1, formula *not p* does not play any role. So removing *not p* has no any effect on a's derivation. This follows that $\Pi'_1 \models a$ iff $\Pi_1 \models a$. So a is w-irrelevant to $\{p\}$ in Π. □

Example 7. Consider the following program Π:

$$a \leftarrow not\ b,$$
$$c \leftarrow d,$$
$$e \leftarrow c,$$
$$b \leftarrow not\ c.$$

It is easy to see that a is w-irrelevant to $\{c\}$ in Π. This is because $\Pi \not\models a$ and $WForgetLP(\Pi, \{c\}) = \{a \leftarrow not\ b, e \leftarrow d, b \leftarrow\} \not\models a$. Indeed, since $Support(a) = \{b, c\}$ where c is a negative support and $\Pi \not\models c$, the condition of Result 1 of Theorem 4 holds. We can also verify that a is not s-irrelevant to $\{c\}$ in Π.

Now suppose we add an extra rule into Π: $\Pi' = \Pi \cup \{d \leftarrow\}$. Here we still have $Support(a) = \{b, c\}$ where c is a negative support. However, since $\Pi' \models c$, according to Result 2 in Theorem 4, a is s-irrelevant to $\{c\}$ in Π'. It is also observed that a is *not* w-irrelevant to $\{c\}$ in Π'.

We can generalize the notion of irrelevance to the logic program context. Formally, let Σ be a logic program context and a an atom, we say that a is derivable from Σ's ith component, denoted as $\Sigma \models_i a$, if $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ and $\Pi_i \models a$.

Definition 11 *(Irrelevance wrt logic program contexts).* Let Σ and Σ' be two logic program contexts where $\Sigma' \in Solution(\Sigma)$, and a an atom. We say that a is *irrelevant* with respect to Σ and Σ' on their ith components, or simply say that a is $(\Sigma, \Sigma')^i$-irrelevant, if $\Sigma \models_i a$ iff $\Sigma' \models_i a$.

Given a logic program context Σ and an atom a, we would like to know whether there is a preferred solution Σ' of Σ such that a is $(\Sigma, \Sigma')^i$-irrelevant. To answer this question, we need to consider the *preservation of irrelevance* along the preferred ordering $\preceq_\Sigma$ on solutions of Σ. That is, if $\Sigma', \Sigma'' \in Solution(\Sigma)$, $\Sigma' \preceq_\Sigma \Sigma''$ and a is $(\Sigma, \Sigma'')^i$-irrelevant, then under what conditions a is also $(\Sigma, \Sigma')^i$-irrelevant. If for each of those more preferred solutions, a's irrelevance is preserved, then eventually, we can obtain a's irrelevance with respect to Σ and its preferred solution.

We formalize this idea as follows. Let Σ, Σ', Σ'' be three logic program contexts and $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$. We say that Σ' and Σ'' are *forgetting-congruent* on their ith components with respect to Σ, denoted as $\Sigma' \sim^i_\Sigma \Sigma''$, if for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$,

$$\Phi'_i = \big(\mathit{SForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma',$$
$$\Phi''_i = \big(\mathit{SForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma'',$$

or

$$\Phi'_i = \big(\mathit{WForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma',$$
$$\Phi''_i = \big(\mathit{WForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i\big) \in \Sigma'',$$

where $P', P'' \subseteq \mathcal{F}_i$. In other words, if two solutions of Σ are forgetting-congruent on their ith components, it means that both of their ith components are obtained by performing either strong forgettings or weaking forgettings on some sets of atoms from Σ's ith component. We say that two solutions Σ' and Σ'' of Σ are *forgetting-congruent*, denoted as $\Sigma' \sim_\Sigma \Sigma''$, if $\Sigma' \sim^i_\Sigma \Sigma''$ for each i. The following theorem shows that forgetting-congruence is a sufficient condition for preserving irrelevance in terms of the preferred ordering on solutions.

Theorem 5. *Let Σ, Σ', Σ'' be three logic program contexts and $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$, a an atom. Suppose $\Sigma' \preceq_\Sigma \Sigma''$ and a is $(\Sigma, \Sigma'')^i$-irrelevant. Then a is $(\Sigma, \Sigma')^i$-irrelevant if $\Sigma' \sim^i_\Sigma \Sigma''$.*

Proof. To prove this theorem, we need to show that for $\Sigma', \Sigma'' \in \mathit{Solution}(\Sigma)$, if $\Phi'_i = (\Pi'_i = \mathit{SForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i)$ and $\Phi''_i = (\Pi''_i = \mathit{SForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i)$, or $\Phi'_i = (\Pi'_i = \mathit{WForgetLP}(\Pi_i, P'), \mathcal{C}_i, \mathcal{F}_i)$ and $\Phi''_i = (\Pi''_i = \mathit{WForgetLP}(\Pi_i, P''), \mathcal{C}_i, \mathcal{F}_i)$, where Π_i is in some $\Phi_i \in \Sigma$, $\Phi'_i \in \Sigma'$, $\Phi''_i \in \Sigma''$, $P' \subseteq P'' \subseteq \mathcal{F}_i$, and $\Pi_i \models a$ iff $\Pi''_i \models a$, then $\Pi_i \models a$ iff $\Pi'_i \models a$. Recall that we do not consider invalid strong and weak forgettings, so here we assume that all Π_i, Π'_i and Π''_i are consistent programs.

In order to avoid unnecessary tediousness in our proof, we consider a simplified case in our proof where $P' = \{p\}$ and $P'' = \{p, q\}$. Note that the proof for the general case of $P' \subseteq P''$ can be obtained in a similar way of this proof. Under the assumption of $P' = \{p\}$ and $P'' = \{p, q\}$, program Π_i may be simplified as a form of $\Pi_i = \Pi_{i1} \cup \Pi_{i2} \cup \Pi_{i3}$, where Π_{i1} contains the following rules:

$$r_1\colon\ p \leftarrow \mathit{pos}(r_1), \mathit{not}\ \mathit{neg}(r_1),$$
$$r'_1\colon\ p \leftarrow q, \mathit{pos}(r'_1), \mathit{not}\ \mathit{neg}(r'_1),$$
$$r_2\colon\ q \leftarrow \mathit{pos}(r_2), \mathit{not}\ \mathit{neg}(r_2),$$
$$r'_2\colon\ q \leftarrow p, \mathit{pos}(r'_2), \mathit{not}\ \mathit{neg}(r'_2),$$
$$r_3\colon\ \mathit{head}(r_3) \leftarrow p, \mathit{pos}(r_3), \mathit{not}\ \mathit{neg}(r_3),$$
$$r_4\colon\ \mathit{head}(r_4) \leftarrow q, \mathit{pos}(r_4), \mathit{not}\ \mathit{neg}(r_4).$$

We assume p and q do not occur in anywhere else in Π_{i1}. Π_{i2} contains the rules not having p and q in their heads and positive bodies, but only having p and q in their negative bodies:

$$r_5\colon\ \mathit{head}(r_5) \leftarrow \mathit{pos}(r_5), \mathit{not}\ p, \mathit{not}\ q, \ldots,$$
$$r_6\colon\ \mathit{head}(r_6) \leftarrow \mathit{pos}(r_6), \mathit{not}\ p, \ldots,$$
$$r_7\colon\ \mathit{head}(r_7) \leftarrow \mathit{pos}(r_7), \mathit{not}\ q, \ldots.$$

Finally, Π_{i3} consists of rules not containing p and q in anywhere.

Case 1. Suppose $\Pi'_i = \mathit{SForgetLP}(\Pi_i, \{p\})$ and $\Pi''_i = \mathit{SForgetLP}(\Pi_i, \{p, q\})$. In this case, Π'_i and Π''_i are as follows:

Π'_i:

$$r_2\colon\ q \leftarrow \mathit{pos}(r_2), \mathit{not}\ \mathit{neg}(r_2),$$
$$r'_{21}\colon\ q \leftarrow \big(\mathit{pos}(r_1) \cup \mathit{pos}(r'_2)\big), \mathit{not}\big(\mathit{neg}(r_1) \cup \mathit{neg}(r'_2)\big),$$

r_{31}: $head(r_3) \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big)$,

r'_{31}: $head(r_3) \leftarrow q, \big(pos(r'_1) \cup pos(r_3)\big), not\big(neg(r'_1) \cup neg(r_3)\big)$,

r_4: $head(r_4) \leftarrow q, pos(r_4), not\ neg(r_4)$,

r_7: $head(r_7) \leftarrow pos(r_7), not\ q, \ldots,$

Π_{i3},

Π''_i:

r_{31}: $head(r_3) \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big)$,

r_{32}: $head(r_3) \leftarrow \big(pos(r_2) \cup pos(r'_1) \cup pos(r_3)\big), not\big(neg(r_2) \cup neg(r'_1) \cup neg(r_3)\big)$,

r'_{32}: $head(r_3) \leftarrow \big(pos(r_1) \cup pos(r'_2) \cup pos(r'_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r'_2) \cup neg(r'_1) \cup neg(r_3)\big)$,

r_{33}: $head(r_4) \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big)$,

r'_{33}: $head(r_4) \leftarrow \big(pos(r_1) \cup pos(r'_2) \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r'_2) \cup neg(r_4)\big)$,

Π_{i3}.

Now we assume that for some atom a, $\Pi_i \models a$ iff $\Pi''_i \models a$. From the proof of Theorem 3, we know that $\Pi_i \models a$ iff $\Pi^*_i \models a$, where $\Pi^*_i = \{r \mid r \in \Pi_i$ and occurs in the definition of *Support*$(a)\}$. Let $\Pi_i^{'*} = \{r \mid r \in \Pi'_i$ and occurs in the definition of *Support*$(a)\}$ and $\Pi_i^{''*} = \{r \mid r \in \Pi''_i$ and occurs in the definition of *Support*$(a)\}$. From $\Pi_i \models a$ iff $\Pi''_i \models a$, we have $\Pi^*_i \models a$ iff $\Pi_i^{''*} \models a$. Then we will show that $\Pi_i^{'*} \models a$ iff $\Pi_i^{'*} \models a$, this will follow $\Pi_i \models a$ iff $\Pi''_i \models a$.

Comparing structures of programs Π_i and Π''_i, it is clear that rules r_5, r_6 and r_7 do not play any role in deriving a even if they are in Π^*_i because these rules are removed from Π''_i. Consequently, rule r_7 does not play any role in deriving a in Π'_i even if it is in $\Pi_i^{'*}$. On the other hand, for all rules in $\Pi_i^{'*}$, they are either in $\Pi_i^{''*}$ or have been replaced in $\Pi_i^{''*}$ by the corresponding rules after reduction on $\{q\}$. Then we have the fact that $\Pi_i^{''*} \models b$ iff $\Pi_i^{'*} \models b$ for all atoms which are not q. Now consider that $a = q$. since $\Pi''_i \not\models q$, and q is $(\Sigma, \Sigma'')^i$-irrelevant, we have $\Pi_i \not\models q$. Then we can conclude that $\Pi_i^{''*} \not\models q$ as well because if this is not the case, we will have $\Pi_i \models q$ (observing that rules r_1, r_2 and r'_2 used to derive q can be replaced by r_2 and r'_{21} in Π''_i), which contradicts with $\Pi''_i \not\models q$. So the result holds.

Case 2. Suppose $\Pi'_i = WForgetLP(\Pi_i, \{p\})$ and $\Pi''_i = WForgetLP(\Pi_i, \{p, q\})$. In this case, we have:

Π'_i:

r_2: $q \leftarrow pos(r_2), not\ neg(r_2)$,

r'_{21}: $q \leftarrow \big(pos(r_1) \cup pos(r'_2)\big), not\big(neg(r_1) \cup neg(r'_2)\big)$,

r_{31}: $head(r_3) \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big)$,

r'_{31}: $head(r_3) \leftarrow q, \big(pos(r'_1) \cup pos(r_3)\big), not\big(neg(r'_1) \cup neg(r_3)\big)$,

r_4: $head(r_4) \leftarrow q, pos(r_4), not\ neg(r_4)$,

r'_5: $head(r_5) \leftarrow pos(r_5), not\ q, \ldots,$

r'_6: $head(r_5) \leftarrow pos(r_5), \ldots,$

r_7: $head(r_7) \leftarrow pos(r_7), not\ q, \ldots,$

Π_{i3},

Π''_i:

r_{31}: $head(r_3) \leftarrow \big(pos(r_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r_3)\big)$,

r_{32}: $head(r_3) \leftarrow \big(pos(r_2) \cup pos(r'_1) \cup pos(r_3)\big), not\big(neg(r_2) \cup neg(r'_1) \cup neg(r_3)\big)$,

r'_{32}: $head(r_3) \leftarrow \big(pos(r_1) \cup pos(r'_2) \cup pos(r'_1) \cup pos(r_3)\big), not\big(neg(r_1) \cup neg(r'_2) \cup neg(r'_1) \cup neg(r_3)\big)$,

r_{33}: $head(r_4) \leftarrow \big(pos(r_2) \cup pos(r_4)\big), not\big(neg(r_2) \cup neg(r_4)\big)$,

$$r'_{33}\colon\ head(r_4) \leftarrow \big(pos(r_1) \cup pos(r'_2) \cup pos(r_4)\big), not\big(neg(r_1) \cup neg(r'_2) \cup neg(r_4)\big),$$
$$r''_5\colon\ head(r_5) \leftarrow pos(r_5), \ldots,$$
$$r''_6\colon\ head(r_6) \leftarrow pos(r_6), \ldots,$$
$$r''_7\colon\ head(r_7) \leftarrow pos(r_7), \ldots,$$
$$\Pi_{i3}.$$

In a similar way as described above, we can show that $\Pi''_i \models a$ iff $\Pi'_i \models a$. □

Corollary 1. *Let $\Sigma', \Sigma'' \in Solution(\Sigma)$, where Σ'' is a preferred solution of Σ, and a an atom. Then a is $(\Sigma, \Sigma'')^i$-irrelevant if a is $(\Sigma, \Sigma')^i$-irrelevant and $\Sigma' \sim^i_\Sigma \Sigma''$.*

Example 8. Let us consider a logic program context $\Sigma = (\Phi_1, \Phi_2, \Phi_3)$, where

Φ_1:	Φ_2:	Φ_3:
Π_1: $a \leftarrow not\ b$,	Π_2: $d \leftarrow$,	Π_3: $b \leftarrow not\ a$,
$c \leftarrow a$,	$b \leftarrow not\ c$,	$c \leftarrow not\ a$,
$d \leftarrow not\ e$,		$d \leftarrow not\ c$,
$\mathcal{C}_1$: $\emptyset$,	$\mathcal{C}_2$: $\leftarrow not\ d$,	$\mathcal{C}_3$: $\leftarrow c, d$,
	$e \leftarrow c$,	
$\mathcal{F}_1$: $\{a, b, c, d, e\}$,	$\mathcal{F}_2$: $\{b, c, d\}$,	$\mathcal{F}_3$: $\{a, b, c, d\}$.

It is easy to see that conflicts occur in Σ. That is, $\Pi_1 \cup \mathcal{C}_2$, $\Pi_1 \cup \mathcal{C}_3$, and $\Pi_3 \cup \mathcal{C}_2$ are inconsistent. By performing strong and weak forgettings, we obtain a solution of Σ: $\Sigma' = (\Phi'_1, \Phi_2, \Phi'_3)$, where $\Phi'_1 = (\Pi'_1, \mathcal{C}_1, \mathcal{F}_1)$, $\Phi'_3 = (\Pi'_3, \mathcal{C}_3, \mathcal{F}_3)$, $\Pi'_1 = WForgetLP(\Pi_1, \{a, c, e\}) = \{d \leftarrow\}$ and $\Pi'_3 = SForgetLP(\Pi_3, \{a\}) = \{d \leftarrow \texttt{not}\, c\}$. We can verify that atom a is $(\Sigma, \Sigma')^i$-irrelevant for all $i = 1, 2, 3$.

On the other hand, by weakly forgetting only $\{c, e\}$ in Π_1, we further obtain a more preferred solution of Σ: $\Sigma'' = (\Phi''_1, \Phi_2, \Phi'_3)$, where $\Phi''_1 = (\Pi''_1, \mathcal{C}_1, \mathcal{F}_1)$, and $\Pi''_1 = WForgetLP(\Pi_1, \{c, e\}) = \{a \leftarrow not\ b, d \leftarrow\}$. In fact, Σ'' is also a preferred solution of Σ. Since $\Sigma' \sim_\Sigma \Sigma''$, according to Corollary 1, we know that a is also $(\Sigma, \Sigma'')^i$-irrelevant ($i = 1, 2, 3$).

5.2. Characterizing solutions for conflict solving

In this subsection, we focus our study on the semantic characterization on conflict solving solutions, because such characterizations are useful to optimize the procedure of conflict solving in logic program contexts. To begin with, we give a general result for the existence of preferred solutions for arbitrary logic program context.

Theorem 6. *Let Σ be a logic program context. Σ has a preferred solution iff $Solution(\Sigma) \neq \emptyset$.*

Proof. Obviously, if Σ has a preferred solution, then $Solution(\Sigma) \neq \emptyset$. Now we assume that $Solution(\Sigma) \neq \emptyset$. In this case, we only need to show that for each $\Sigma' \in Solution(\Sigma)$, a new solution Σ'' can always be generated from Σ' such that $\Sigma'' \preceq_\Sigma \Sigma'$. If no such solution can be generated from Σ', then Σ' itself is a preferred solution. We present the following algorithm for this purpose.

```
Algorithm: **Solution-Generation**
**Input**: $\Sigma = (\Phi_1, \ldots, \Phi_n)$ and $\Sigma' = (\Phi'_1, \ldots, \Phi'_n)$, where
    $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$ and $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i)$;
**Output**: $\Sigma'' = (\Phi''_1, \ldots, \Phi''_n)$;
**for** $i = 1$ to $n$
    **let** $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma'$ and
    $\Pi'_i = SForgetLP(\Pi, P)$ or $\Pi'_i = WForgetLP(\Pi, P)$ $(P \subseteq \mathcal{F}_i)$;
```

```
  while $Q \subset P$
      testing the consistency of $SForgetLP(\Pi, Q) \cup \mathcal{C}_j$
      for all $j = 1, \dots, n$;
      if consistency holds, then $\Pi_i'' = SForgetLP(\Pi, Q)$;
      if consistency does not hold, then
          testing the consistency of $WForgetLP(\Pi, Q) \cup \mathcal{C}_j$
          for all $j = 1, \dots, n$;
          if consistency holds, then
              $\Pi_i'' = WForgetLP(\Pi, Q)$, otherwise $\Pi_i'' = \Pi_i'$;
return $\Sigma'' = ((\Pi_1'', \mathcal{C}_1, \mathcal{F}_1), \dots, (\Pi_n'', \mathcal{C}_n, \mathcal{F}_n))$.
```

It is easy to see that algorithm **Solution-Generation** terminates as the procedures of computing $SForgetLP(\Pi_i, Q)$ and $WForgetLP(\Pi_i, Q)$, and consistency testing for a program can always finish in finite steps respectively. Furthermore, the output Σ'' is either the same as Σ' or $\Sigma'' \preceq_\Sigma \Sigma'$. This proves our result. □

The proof of Theorem 5 actually provides a method to generate a preferred solution for a logic program context. That is, once we have an initial solution for a logic program context, we can always generate a more preferred solution from the current one. We continue the process until a preferred solution is finally achieved. However, not every logic program has a solution. For instance, a logic program context $\Sigma = (\Phi_1, \Phi_2) = (\emptyset, \{\leftarrow a, not\ b\}, \emptyset), (\{a \leftarrow not\ b\}, \emptyset, \emptyset))$ has no solution.

Proposition 5. *Let $\Sigma = (\Phi_1, \dots, \Phi_n)$ be a logic program context. If for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i)$, Π_i does not contain a constraint rule (a rule with empty head), $\mathcal{C}_i$ is consistent, and for each $r \in \Pi_i$, $atom(r) \cap \mathcal{F}_i \neq \emptyset$, then $Solution(\Sigma) \neq \emptyset$.*

Proof. We show that $\Sigma' = (\Phi_1', \dots, \Phi_n')$, where $\Phi_i' = (\emptyset, \mathcal{C}_i, \mathcal{F}_i)$ $(1 \leqslant i \leqslant n)$ is a solution of Σ. Since for each i, $\mathcal{F}_i \cap atom(r) \neq \emptyset$ for each $r \in \Pi_i$, we have $\Pi_i' = SForgetLP(\Pi_i, \mathcal{F}_i) = \emptyset$ (note that this is because we already assumed that Π_i does not contain any rules with empty heads. Instead, this type of rule is contained in $\mathcal{C}_i$). This follows that $\Pi_i' \cup \mathcal{C}_j = \mathcal{C}_j$ for all $j = 1, \dots, n$ are consistent. So $((\emptyset, \mathcal{C}_1, \mathcal{F}_1), \dots, (\emptyset, \mathcal{C}_n, \mathcal{F}_n))$ is a solution of Σ. □

We should indicate that many conflict solving scenarios can be represented in the type of logic program context in Proposition 5. For example, the negotiation scenario discussed in Example 6 and most logic program update approaches (see Section 6) can be specified under logic program contexts with this form. Therefore, solving conflicts for this particular type of logic program context has a special interest in various applications. This motivates us to study more detailed properties related to the solution of this type of logic program contexts.

We first introduce some useful concepts. A logic program Π's *dependency graph* [1], denoted as $G(\Pi)$, is a directed graph $(atom(\Pi), E)$, where $atom(\Pi)$ is the set of vertices, and E is the set of edges. An edge $(a, b) \in E$ iff there is a rule $r \in \Pi$ such that $a \in pos(r) \cup neg(r)$ and $\{b\} = head(r)$. Edge (a, b) is labelled "positive" if $a \in pos(r)$ and "negative" if $a \in neg(r)$. Then a logic program is called *call-consistent* [12] if it does not contain a constraint (i.e. a rule with empty head) and its dependency graph has no simple cycles with odd number of negative edges.[5]

Lemma 1. *Let Π_1 and Π_2 be two logic programs and Π_1 be consistent. Then program $\Pi_1 \cup \Pi_2$ is consistent if $body(\Pi_1) \cap head(\Pi_2) = \emptyset$ and Π_2 is call-consistent.*

Proof. Similar to the proof of Theorem 3, To prove this lemma, we need a result about program splitting from [26]. To remain a completeness of the proof, we present this result again. Before we present this program splitting result, we introduce a notion. Given a program Π and a set of atoms S, we use $e(\Pi, S)$ to denote the program obtained from Π by deleting: (1) each rule in Π having a form *not a* in its body with $a \in S$; and (2) all atoms a in the bodies of the remaining rules with $a \in S$. Intuitively, $e(\Pi, S)$ can be viewed as a simplicity of Π giving those atoms in S to be true. Then we can re-state Theorem 5 in [26] under the normal logic program setting:

[5] A simple cycle is the one that does not contain any other cycles.

A set of atoms S is a stable model of program Π if and only if $\Pi = \Pi_1 \cup \Pi_2$ such that $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, and $S = S_1 \cup S_2$, where S_1 is a stable model of Π_1 and S_2 is a stable model of program $e(\Pi_2, S_1)$.

From this result, we can see that under the condition that Π_1 is consistent, $\Pi_1 \cup \Pi_2$ is consistent if $body(\Pi_1) \cap head(\Pi_2) = \emptyset$, and for each stable model S_1 of Π_1, $e(\Pi_2, S_1)$ is also consistent.

Since a call-consistent program is also consistent [25], to prove our result, we will prove that if Π_2 is call-consistent, then $e(\Pi_2, S_1)$ is also call-consistent for any set of atoms S_1. From the definition of call-consistency, it is clear that if Π_2 is call-consistent, its dependency graph does not contain a simple cycle with odd number of negative edges. Observing that for any set of atoms S_1, program $e(\Pi_2, S_1)$'s dependency graph $G(e(\Pi_2, S_1))$ can be obtained from $G(\Pi_2)$ by removing more edges and nodes from $G(\Pi_2)$. That is, $G(e(\Pi_2, S_1))$ is a subgraph of $G(\Pi_2)$. This concludes that $G(e(\Pi_2, S_1))$ does not contain a simple cycle with odd number of negative edges. So $e(\Pi_2, S_1)$ is also call-consistent. □

We need to mention that in Lemma 1, the call-consistency condition for program Π_2 is important. It is easy to see that Π_2's consistency does not imply the consistency of $\Pi_1 \cup \Pi_2$ even if the other conditions of Lemma 1 remain the same. For example, consider two programs $\Pi_1 = \{b \leftarrow\}$ and $\Pi_2 = \{a \leftarrow b, not\ a\}$. Both Π_1 and Π_2 are consistent and $body(\Pi_1) \cap head(\Pi_2) = \emptyset$. But $\Pi_1 \cup \Pi_2$ has no stable model. The following theorem states that the procedure of generating a more preferred solution may be simplified under certain conditions.

Theorem 7. *Let $\Sigma = ((\Pi_1, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi_n, \mathcal{C}_n, \mathcal{F}_n))$ be a logic program context satisfying the conditions stated in Proposition 5. Suppose $\Sigma' = ((\Pi'_1, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi'_n, \mathcal{C}_n, \mathcal{F}_n))$ is a solution of Σ, where each Π'_i is of the form SForgetLP(Π_i, P_i) or WForgetLP(Π_i, P_i) $(P_i \subseteq \mathcal{F}_i)$.[6] Then a logic program context $\Sigma'' = ((\Pi''_1, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi''_n, \mathcal{C}_n, \mathcal{F}_n))$ is a solution of Σ and $\Sigma'' \preceq_\Sigma \Sigma'$, if for each i either $\Pi''_i = \Pi'_i$, or Π''_i is of the form $\Pi''_i = $ SForgetLP(Π_i, Q_i) or $\Pi''_i = $ WForgetLP(Π_i, Q_i) for some $Q_i \subseteq P_i$ such that $body(\bigcup_{i=1}^n \mathcal{C}_i) \cap head(\Pi''_i) = \emptyset$ and Π''_i is call-consistent.*

Proof. From Lemma 1, it follows that if for each i, $body(\bigcup_{i=1}^n \mathcal{C}_i) \cap head(\Pi''_i) = \emptyset$ and Π''_i is call-consistent, then all programs $\Pi''_i \cup \mathcal{C}_1, \ldots, \Pi''_i \cup \mathcal{C}_n$ are consistent. So Σ'' is a solution of Σ. On the other hand, since for each i, $Q_i \subseteq P_i$, this concludes that $\Sigma'' \preceq_\Sigma \Sigma'$. □

In Theorem 7, the condition that $body(\bigcup_{i=1}^n \mathcal{C}_i) \cap head(\Pi'_i) = \emptyset$ and Π'_i is call-consistent ensures that Σ' is a solution of Σ, while the minimal subset P_i of $atom(\Pi_i)$ implies that Σ' is a preferred solution. The following Example 9 illustrates how a preferred solution can be obtained under the condition of Theorem 7.

Example 9. Consider a logic program context $\Sigma = (\Phi_1, \Phi_2, \Phi_3)$, where

Φ_1:	Φ_2:	Φ_3:
Π_1: $a \leftarrow not\ b,$	Π_2: $d \leftarrow,$	Π_3: $a \leftarrow not\ b,$
$c \leftarrow a, not\ d,$	$f \leftarrow not\ b,$	$c \leftarrow not\ b,$
	$e \leftarrow not\ d,$	
$\mathcal{C}_1$: $e \leftarrow d,$	$\mathcal{C}_2$: $\leftarrow a, c,$	$\mathcal{C}_3$: $f \leftarrow d,$
$\mathcal{F}_1 = \{a, b, c, d\},$	$\mathcal{F}_2 = \{b, d, e, f\},$	$\mathcal{F}_3 = \{a, b, c\}.$

Clearly, Σ is not conflict free since $\Pi_1 \cup \mathcal{C}_2$, $\Pi_2 \cup \mathcal{C}_1$, $\Pi_2 \cup \mathcal{C}_3$ and $\Pi_3 \cup \mathcal{C}_2$ are not consistent. We can verify that a logic program context $\Sigma_1 = (\Phi'_1, \Phi'_2, \Phi'_3)$ is a solution of Σ, where

$$\Phi'_1 = (SForgetLP(\Pi_1, \{c\}), \mathcal{C}_1, \mathcal{F}_1),$$
$$\Phi'_2 = (SForgetLP(\Pi_2, \{d, e, f\}), \mathcal{C}_2, \mathcal{F}_2),$$
$$\Phi'_3 = (WForgetLP(\Pi_3, \{a\}), \mathcal{C}_3, \mathcal{F}_3).$$

[6] Note that from Proposition 5, a solution of Σ always exists. In the initial case, Π'_i could be $\emptyset$.

Now we consider a program $WForgetLP(\Pi_2, \{d\})$:

$$f \leftarrow not\ b,$$

$$e \leftarrow .$$

Since $\{e, f\} \cap body(\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3) = \emptyset$ and $WForgetLP(\Pi_2, \{d\})$ is call-consistent, according to Theorem 7, we know that $\Sigma_1'' = (\Phi_1', \Phi_2'', \Phi_3')$, where $\Phi_2' = (WForgetLP(\Pi_2, \{d\}), \mathcal{C}_2, \mathcal{F}_2)$ is also a solution of Σ and $\Sigma'' \preceq_\Sigma \Sigma'$. In fact Σ_1'' is a preferred solution of Σ.

6. Representing logic program updates

Logic program updates have been considerably studied in recent years. While similarities and differences among these different approaches have been addressed by many researchers, it is believed that comparing different types of update approaches at some formal level is generally difficult (discussions on this topic are referred to [5,6,15,28]). In this section, we show that four major logic program update approaches can be transformed into the framework of logic program contexts, in which all these update approaches become special cases of conflict solving problems with different types of constraints.

6.1. Representing causal rejection based approach

Eiter et al.'s update approach is based on a principle called *causal rejection* where a sequence of logic program updates is allowed [5]. Let $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$, where $\Pi_1, \ldots, \Pi_1$ are extended logic programs, be an (extended logic program) *update sequence* and $\mathcal{A}$ a set of atoms. We say that $\mathbf{P}$ is *over* $\mathcal{A}$ iff $\mathcal{A}$ represents the set of all atoms occurring in the rules in $\Pi_1, \ldots, \Pi_n$. We use $Lit_\mathcal{A}$ to denote the set of all literals whose corresponding atoms are in $\mathcal{A}$. We assume a set $\mathcal{A}^*$ of atoms extending $\mathcal{A}$ by new and pairwise distinct atoms $rej(r)$ and a_i, for each rule r occurring in $\Pi_1, \ldots, \Pi_n$ and each atom $a \in \mathcal{A}$. Then Eiter et al.'s update process is defined by the following two definitions (here we only consider ground extended logic programs in our investigation).

Definition 12. [5] Given an update sequence $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ over a set of atoms $\mathcal{A}$, the *update program* $\mathbf{P}_\triangleleft = \Pi_1 \triangleleft \cdots \triangleleft \Pi_n$ over $\mathcal{A}^*$ consisting of the following items:

(1) all constraints in $\Pi_1, \ldots, \Pi_n$ (recall that a constraint is a rule with an empty head);
(2) for each r in Π_i $(1 \leqslant i \leqslant n)$:

$$l_i \leftarrow body(r), not\ rej(r) \quad \text{if } head(r) = \{l\};$$

(3) for each $r \in \Pi_{i-1}$ $(2 \leqslant i \leqslant n)$:

$$rej(r) \leftarrow body(r), \neg l_i \quad \text{if } head(r) = \{l\};$$

(4) for each literal l occurring in $\Pi_1 \cup \cdots \cup \Pi_n$:

$$l_{i-1} \leftarrow l_i\ (1 < i \leqslant n), \quad l \leftarrow l_1.$$

A set $S \subseteq Lit_\mathcal{A}$ is an *update answer set* of $\mathbf{P}$ iff $S = S' \cap Lit_\mathcal{A}$ for some answer set S' of $\mathbf{P}_\triangleleft$.

As an example, consider an update sequence $\mathbf{P} = (\Pi_1, \Pi_2, \Pi_3)$, where Π_1, Π_2 and Π_3 consist of the following rules respectively [5],

Π_1:

r_1: $sleep \leftarrow not\ tv\_on$,

r_2: $night \leftarrow$,

r_3: $tv\_on \leftarrow$,

r_4: $watch\_tv \leftarrow tv\_on$;

Π_2:

r_5: $\neg tv\_on \leftarrow power\_failure$,

r_6: $power\_failure \leftarrow$,

Π_3:

r_7: $\neg power\_failure \leftarrow$.

According to Definition 12, it is easy to see that $\mathbf{P} = (\Pi_1, \Pi_2, \Pi_3)$ has a unique update answer set $S = \{\neg power\_failure, tv\_on, watch\_tv, night\}$, which is consistent with our intuition.

In order to transform this update approach into our framework of logic program context, we first re-formulate this approach in a normal logic program setting. In particular, given an update sequence $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ over $\mathcal{A}$, we extend the set $\mathcal{A}$ to $\overline{\mathcal{A}}$ by adding atom $\bar{a}$ to $\mathcal{A}$ for each $a \in \mathcal{A}$. Then by replacing each negative atom $\neg a$ occurring in Π_i with $\bar{a}$, and adding constraint $\leftarrow a, \bar{a}$ for each $a \in \mathcal{A}$, we obtain a translated (normal logic program) update sequence $\overline{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ over $\overline{\mathcal{A}}$.

We also extend set $\overline{\mathcal{A}}$ to $\overline{\mathcal{A}}^*$ by including new atoms $rej(r)$, a_i and $\bar{a}_i$ for each rule r in $\overline{\Pi}_1, \ldots, \overline{\Pi}_n$ and each pair of atoms $a, \bar{a} \in \overline{\mathcal{A}}$. Then following Definition 12, we can obtain the corresponding update program $\overline{\mathbf{P}}_\triangleleft$ which is also a normal logic program. We also call a stable model of $\overline{\mathbf{P}}_\triangleleft$ *update stable model* of $\overline{\mathbf{P}}$.

Proposition 6. *Let $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ be an update sequence, $\mathbf{P}_\triangleleft$ the update program of $\mathbf{P}$, and $\overline{\mathbf{P}}$ and $\overline{\mathbf{P}}_\triangleleft$ the corresponding translations of $\mathbf{P}$ and $\mathbf{P}_\triangleleft$ respectively as described above. $S \subseteq Lit_{\mathcal{A}}$ is an update answer set of $\mathbf{P}$ iff there is an update stable model $\overline{S}$ of $\overline{\mathbf{P}}$ such that $S = (\overline{S} \cap \mathcal{A}) \cup \{\neg a \mid \bar{a} \in \overline{S}\}$.*[7]

Having Proposition 6, we only need to consider a transformation from a normal logic program update sequence $\overline{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$, where $\overline{\mathbf{P}}$ is translated from an extended logic program update sequence $\mathbf{P}$ as described above, to a conflict solving problem under the framework of logic program contexts.

Definition 13. Let $\overline{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ $(n > 1)$ be a normal logic program update sequence over $\overline{\mathcal{A}}$. We specify a sequence of logic program contexts $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$[8] over the set of atoms $\mathcal{B} = \overline{\mathcal{A}}^* \cup \{l_{a_i}, l_{\bar{a}_i} \mid a_i, \bar{a}_i \in \overline{\mathcal{A}}^*, i = 1, \ldots, n\}$ where l_{a_i} and $l_{\bar{a}_i}$ are newly introduced atoms:

(1) $\Sigma_1 = ((\overline{\Pi}_1^*, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where
  (a) $\overline{\Pi}_1^*$ consists of the following rules:
    (i) all constraints in $\overline{\Pi}_1, \ldots, \overline{\Pi}_n$;
    (ii) for each $r \in \overline{\Pi}_i$: $a \leftarrow body(r)$ or $\bar{a} \leftarrow body(r)$ $(i = 1, \ldots, n)$, $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$ respectively,
    (iii) for each $a, \bar{a}$ in $\overline{\mathcal{A}}$,
      $a_{i-1} \leftarrow a_i, \bar{a}_{i-1} \leftarrow \bar{a}_i \quad (i = 1, \ldots, n)$,
      $a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.
  (b) $\mathcal{F}_1 = \{l_{a_{n-1}}, l_{\bar{a}_{n-1}} \mid \forall a \in \mathcal{A}\}$,
  (c) $\mathcal{C}_1 = \{\leftarrow a_{n-1}, \bar{a}_n, \leftarrow \bar{a}_{n-1}, a_n \mid \forall a \in \mathcal{A}\}$;
(2) $\Sigma_i = ((\overline{\Pi}_i^*, \emptyset, \mathcal{F}_i), (\emptyset, \mathcal{C}_i, \emptyset))$ $(i = 1, \ldots, n)$, where
  (a) $\overline{\Pi}_i^* = \overline{\Pi}_{i-1}^\dagger$, and $\overline{\Pi}_{i-1}^\dagger$ is in a preferred solution of Σ_{i-1}:
    $\Sigma'_{i-1} = ((\overline{\Pi}_{i-1}^\dagger, \emptyset, \mathcal{F}_{i-1}), (\emptyset, \mathcal{C}_{i-1}, \emptyset))$,
  (b) $\mathcal{F}_i = \{l_{a_{n-i}}, l_{\bar{a}_{n-i}} \mid \forall a \in \mathcal{A}\}$,
  (c) $\mathcal{C}_i = \{\leftarrow a_{n-i}, \bar{a}_{n-i+1}, \leftarrow \bar{a}_{n-i}, a_{n-i+1} \mid \forall a \in \mathcal{A}\}$.

[7] Note that S is reduced to $Lit_{\mathcal{A}}$ if both a and $\neg a$ are in S for some $a \in \mathcal{A}$.

[8] Note that when $n = 1$ our transformation becomes trivial since we can simply specify Ω_{CR} to consist of a single logic program context $\Sigma = ((\overline{\Pi}_1, \emptyset, \emptyset), (\emptyset, \emptyset, \emptyset))$. In this case Σ has a (preferred) solution iff $\overline{\Pi}_1$ is consistent. So in the rest of the paper we will only consider the case $n > 1$.

A subset $S \subseteq \mathcal{B}$ is called a *model* of Ω_{CR} if S is a stable model of $\overline{\Pi}^{\dagger}_{n-1}$, where $\overline{\Pi}^{\dagger}_{n-1}$ is in a preferred solution of Σ_{n-1}: $\Sigma'_{n-1} = ((\overline{\Pi}^{\dagger}_{n-1}, \emptyset, \mathcal{F}_{n-1}), (\emptyset, \mathcal{C}_{n-1}, \emptyset))$.

Let us take a closer look at Definition 13. Given an update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$, Definition 13 specifies a sequence of logic program contexts $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$, where each Σ_i solves certain conflicts embedded in $\bar{\mathbf{P}}$. Σ_1 represents the first level of conflict solving, where $\overline{\Pi}^*_1$ is similar to $\bar{\mathbf{P}}_{\lhd}$ except that the possible conflict between a_{n-1} and $\bar{a}_n$ (or $\bar{a}_{n-1}$ and a_n) has been reformulated as a constraint $\leftarrow a_{n-1}, \bar{a}_n$ (or $\leftarrow \bar{a}_{n-1}, a_n$ resp.) in $\mathcal{C}_1$. Note that in rules specified in (ii) of Definition 13: $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$, formulas $not\ l_{\bar{a}_{n-1}}$ and $not\ l_{a_{n-1}}$ (here $i = n-1$) are introduced to solve the conflict between a_{n-1} and $\bar{a}_n$ (or $\bar{a}_{n-1}$ and a_n resp.).

Observe that Σ_1 *only* solves conflicts between atoms at level $n-1$. For example, if both a_{n-1} and $\bar{a}_n$ can be derived from $\overline{\Pi}^*_1$, then rule $a_{n-1} \leftarrow body(r), not\ l_{\bar{a}_{n-1}}$ will be eliminated from Π_1 by strongly forgetting atom $l_{\bar{a}_{n-1}}$ under the constraint $\leftarrow a_{n-1}, \bar{a}_n$ in $\mathcal{C}_1$.

In the sequence $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$, conflicts are solved in a *downwards* manner with respect to the update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$, where each Σ_i $(i > 1)$ is specified for the purpose of solving conflicts between atoms a_{n-i} and $\bar{a}_{n-i+1}$ (or $\bar{a}_{n-i}$ and a_{n-i+1}).

Example 10. Consider the TV example mentioned earlier, where $\mathbf{P} = (\Pi_1, \Pi_2, \Pi_3)$ is an update sequence. It is easy to translate $\mathbf{P}$ to the corresponding normal logic program update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \overline{\Pi}_2, \overline{\Pi}_3)$, where $\neg tv\_on$ and $\neg power\_failure$ are replaced by atoms $\overline{tv\_on}$ and $\overline{power\_failure}$ respectively. According to Definition 13, we then specify a sequence of logic program contexts $\Omega_{CR} = (\Sigma_1, \Sigma_2)$ to solve the conflict occurring in $\bar{\mathbf{P}}$. $\Sigma_1 = ((\overline{\Pi}^*_1, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where $\overline{\Pi}^*_1$ consists of the following rules[9]:

$$sleep_1 \leftarrow not\ tv\_on, not\ l_{\overline{sleep_1}},$$
$$night_1 \leftarrow not\ l_{\overline{night_1}},$$
$$tv\_on_1 \leftarrow not\ l_{\overline{tv\_on_1}},$$
$$watch\_tv_1 \leftarrow tv\_on, not\ l_{\overline{watch\_tv_1}},$$
$$\overline{tv\_on_2} \leftarrow power\_failure, not\ l_{tv\_on_2},$$
$$power\_failure_2 \leftarrow not\ l_{\overline{power\_failure_2}},$$
$$\overline{power\_failure_3} \leftarrow not\ l_{power\_failure_3},$$
$$night \leftarrow night_1,$$
$$tv\_on \leftarrow tv\_on_1,$$
$$watch\_tv \leftarrow watch\_tv_1,$$
$$\overline{tv\_on_1} \leftarrow \overline{tv\_on_2},$$
$$\overline{tv\_on} \leftarrow \overline{tv\_on_1},$$
$$\overline{power\_failure_2} \leftarrow \overline{power\_failure_3},$$
$$\overline{power\_failure_1} \leftarrow \overline{power\_failure_2},$$
$$\overline{power\_failure} \leftarrow \overline{power\_failure_1},$$
$$power\_failure_1 \leftarrow power\_failure_2,$$
$$power\_failure \leftarrow power\_failure_1,$$
$$\mathcal{F}_1 = \{l_{power\_failure_2}, l_{\overline{power\_failure_2}}\}, \quad \text{and}$$
$$\mathcal{C}_1 = \{\leftarrow power\_failure_2, \overline{power\_failure_3}\}.$$

[9] To avoid unnecessarily tedious details, here we omit some irrelevant rules and atoms from $\overline{\Pi}^*_1$, $\mathcal{F}_1$ and $\mathcal{C}_1$. The same for Σ_2.

It is easy to see that Σ_1 is not conflict free since $\overline{\Pi}_1^* \cup \mathcal{C}_1$ is not consistent (i.e it has no stable model). To specify Σ_2, we first need to obtain a preferred solution of Σ_1. In fact Σ_1 has a unique preferred solution $\Sigma_1' = ((\overline{\Pi}_1^\dagger, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where

$$\overline{\Pi}_1^\dagger = SForgetLP\big(\overline{\Pi}_1^*, \{l_{\overline{power\_failure_2}}\}\big) = \overline{\Pi}_1^* - \big\{power\_failure_2 \leftarrow not\ l_{\overline{power\_failure_2}}\big\}.$$

Now we specify $\Sigma_2 = ((\overline{\Pi}_1^\dagger, \emptyset, \mathcal{F}_2), (\emptyset, \mathcal{C}_2, \emptyset))$, where $\mathcal{F}_2 = \{l_{tv\_on_1}, l_{\overline{tv\_on}_1}\}$ and $\mathcal{C}_2 = \{\leftarrow tv\_on_1, \overline{tv\_on}_2\}$. Note that Σ_2 is already conflict free. So by ignoring those atoms with subscripts, Ω_{CR} has a unique model $\{\overline{power\_failure}$, *tv_on*, *watch_tv*, *night*$\}$, which is the same as the update stable model of update sequence $\bar{\mathbf{P}}$.

Theorem 8. *Let* $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ $(n > 1)$ *be a normal logic program update sequence over the set of atoms* $\overline{\mathcal{A}}$. *A subset* $\bar{S}$ *of* $\overline{\mathcal{A}}$ *is an update stable model of* $\bar{\mathbf{P}}$ *iff there is a sequence of logic program contexts* $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$ *constructed from* $\bar{\mathbf{P}}$ *as specified in Definition* 13 *such that* Ω_{CR} *has a model* S *satisfying* $\bar{S} = S \cap \overline{\mathcal{A}}$.

Proof. We prove this result by induction on the length n of normal logic program update sequence $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$.

Case 1. We first consider the case $n = 2$, i.e. $\bar{\mathbf{P}} = (\overline{\Pi}_1, \overline{\Pi}_2)$. In this case, $\Omega_{CR} = (\Sigma_1)$, where $\Sigma_1 = ((\overline{\Pi}_1^*, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$ is formed as follows:

(a) $\overline{\Pi}_1^*$ consists of the following rules:
  (i) all constraints in $\overline{\Pi}_1$ and $\overline{\Pi}_2$;
  (ii) for each $r \in \overline{\Pi}_i$: $a \leftarrow body(r)$ or $\bar{a} \leftarrow body(r)$ $(i = 1, 2)$,
  $a_i \leftarrow body(r), not\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow body(r), not\ l_{a_i}$ respectively,
  (iii) for each $a, \bar{a}$ in $\overline{\mathcal{A}}, \ldots, \overline{\Pi}_n$,
  $a_1 \leftarrow a_2, \bar{a}_1 \leftarrow \bar{a}_2$,
  $a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.
(b) $\mathcal{F}_1 = \{l_{a_1}, l_{\bar{a}_1} \mid \forall a \in \mathcal{A}\}$,
(c) $\mathcal{C}_1 = \{\leftarrow a_1, \bar{a}_2, \leftarrow \bar{a}_1, a_2 \mid \forall a \in \mathcal{A}\}$.

Note that in above (ii), for rule $r \in \overline{\Pi}_2$, $a_2 \leftarrow body(r), not\ l_{\bar{a}_2}$, or $\bar{a}_2 \leftarrow body(r), not\ l_{a_2}$ can be simplified as $a_2 \leftarrow body(r)$, or $\bar{a}_2 \leftarrow body(r)$ respectively since atom $l_{\bar{a}_2}$ or l_{a_2} is not forgettable.

Now we consider the update program $\bar{\mathbf{P}}_\lhd$ built upon $\bar{\mathbf{P}}$ (see Definition 12), which consists of the following rules:

(1) all constraints in $\overline{\Pi}_1$ and $\overline{\Pi}_2$;
(2) $a_1 \leftarrow body(r), not\ rej(r)$ or $\bar{a}_1 \leftarrow body(r), not\ rej(r)$ for $r \in \overline{\Pi}_1$, and $a_2 \leftarrow body(r)$ or $\bar{a}_2 \leftarrow body(r)$ for $r \in \overline{\Pi}_2$;
(3) $rej(r) \leftarrow body(r), \bar{a}_2$ if $head(r) = \{a_1\}$ or $rej(r) \leftarrow body(r), a_2$ if $head(r) = \{\bar{a}_1\}$ for $r \in \overline{\Pi}_1$;
(4) for all $a \in \overline{\mathcal{A}}$, $a_1 \leftarrow a_2, \bar{a}_1 \leftarrow \bar{a}_2, a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.

Now suppose $\bar{S}$ is an update stable model of $\bar{\mathbf{P}}$. Then we can extend $\bar{S}$ to $\bar{S}^*$ over set $\overline{\mathcal{A}}^*$ so that $\bar{S}^*$ is a stable model of program $\bar{\mathbf{P}}_\lhd$, which contains atoms $rej(r)$ for some $r \in \overline{\Pi}_1$. Note that those rules in item (2) above with $rej(r) \in \bar{S}^*$ actually play no roles and hence viewed as been removed from $\bar{\mathbf{P}}$. Then we specify a set $P \subseteq \mathcal{F}_1$ which includes those l_{a_1} or $l_{\bar{a}_1}$ whose corresponding rules $r \in \overline{\Pi}_1$ in (ii) are removed from $\bar{\mathbf{P}}$ as indicated above. Then it can be verified that S where $\bar{S} = S \cap \overline{\mathcal{A}}$ must be a stable model of program $SForgetLP(\overline{\Pi}_1^*, P)$, and P is a minimal such set to make $SForgetLP(\overline{\Pi}_1^*, P)$ consistent. That is, S is a model of Ω_{CR}.

On the other hand, consider a stable model S of $SForgetLP(\overline{\Pi}_1^*, P)$, where $SForgetLP(\overline{\Pi}_1^*, P)$ is in a preferred solution of Σ_1. Let $\bar{S} = S \cap \overline{\mathcal{A}}$. Similarly, for each l_{a_1} or $l_{\bar{a}_1}$ in P, we extend $\bar{S}$ to $\bar{S}^*$ to contain atoms $rej(r)$ in $\bar{S}^*$. Note that for each $rej(r)$, such $r \in \overline{\Pi}_1$ corresponds to $a_1 \leftarrow body(r), not\ l_{\bar{a}_1}$ or $\overline{a_1} \leftarrow body(r), not\ l_{a_1}$ in (ii) specified above. Now we do a Gelfond–Lifschitz transformation on program $\overline{\mathcal{P}}_\lhd$ in terms of set $\bar{S}^*$: $\overline{\mathcal{P}}_\lhd^{\bar{S}^*}$. By avoiding tedious checkings, we can show that $\bar{S}^*$ is a stable model of $\overline{\mathcal{P}}_\lhd^{\bar{S}^*}$.

Case 2. Suppose for all $n < k$, $\bar{S}$ is an update stable model of $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_n)$ iff there is a $\Omega_{CR} = (\Sigma_1, \ldots, \Sigma_{n-1})$ such that Ω_{CR} has a model S satisfying $\bar{S} = S \cap \overline{\mathcal{A}}$. Now we consider the case of $n = k$.

($\Rightarrow$) Let $\bar{S}$ be an update stable model of $\bar{\mathbf{P}} = (\overline{\Pi}_1, \ldots, \overline{\Pi}_k)$. We will show that we can generate a sequence of logic program contexts Ω_{CR} with length of $k - 1$ such that Ω_{CR} has a model S satisfying $\bar{S} = S \cap \overline{\mathcal{A}}$.

We first specify a new normal logic program update sequence with length of $k-1$: $\bar{\mathbf{P}}' = (\overline{\Pi}_1, \dots, \overline{\Pi}'_{k-1})$, where $\overline{\Pi}'_{k-1} = \overline{\Pi}^*_{k-1} \cup \overline{\Pi}_k$, and $\overline{\Pi}^*_{k-1} = \overline{\Pi}_{k-1} - \{r \mid \mathit{rej}(r) \in \bar{S}^*\}$.[10] Then from Definition 12, we can see that $\bar{S}$ is also an update stable model of $\bar{\mathbf{P}}'$. Now suppose $\Omega'_{CR} = (\Sigma_1, \dots, \Sigma_{k-2})$ is a sequence of logic program contexts constructed from $\bar{\mathbf{P}}'$ according to Definition 13. From the induction assumption, we know that Ω'_{CR} has a model S satisfying $\bar{S} = S \cap \overline{\mathcal{A}}$.

Now we show that $\Omega'_{CR} = (\Sigma_1, \dots, \Sigma_{k-2})$ actually can be extended to another $\Omega_{CR} = (\Sigma'_1, \Sigma_1, \dots, \Sigma_{k-2})$ with a length of $k-1$, which eventually is constructed from $\bar{\mathbf{P}} = (\overline{\Pi}_1, \dots, \overline{\Pi}_k)$.

Observe $\overline{\Pi}'_{k-1}$ in $\bar{\mathbf{P}}'$, we can see that those a_{k-1} or $\bar{a}_{k-1}$ cannot be derived if $\bar{a}_k$ or a_k is already presented in $\bar{S}$. That is, no conflict between a_{k-1} and $\bar{a}_k$ (or $\bar{a}_{k-1}$ and a_k) exists in $\overline{\Pi}'_{k-1}$. So the first logic program context Σ_1 in Ω'_{CR} is specified as $\Sigma_1 = ((\overline{\Pi}^*_1, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$:

(1) $\overline{\Pi}^*_1$ consists of the following rules:
  (a) all constraints in $\overline{\Pi}_1, \dots, \overline{\Pi}'_{k-1}$;
  (b) for each r: $a \leftarrow \mathit{body}(r)$ or $\bar{a} \leftarrow \mathit{body}(r)$ in $\overline{\Pi}_i$ $(i = 1, \dots, k-2)$ or in $\overline{\Pi}'_{k-1}$: $a_i \leftarrow \mathit{body}(r), \mathit{not}\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow \mathit{body}(r), \mathit{not}\ l_{a_i}$ respectively,
  (c) for each $a, \bar{a}$ in $\overline{\mathcal{A}}$, $a_{i-1} \leftarrow a_i, \bar{a}_{i-1} \leftarrow \bar{a}_i$ $(i = 1, \dots, n)$,
  $a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.
(2) $\mathcal{F}_1 = \{l_{a_{k-2}}, l_{\bar{a}_{k-2}} \mid \forall a \in \mathcal{A}\}$,
(3) $\mathcal{C}'_1 = \{\leftarrow a_{k-2}, \bar{a}_{k-1}, \leftarrow \bar{a}_{k-2}, a_{k-1} \mid \forall a \in \mathcal{A}\}$.

Thus, we can view Σ_1 in Ω'_{CR} represents a preferred solution of logic program context $\Sigma'_1 = ((\overline{\Pi}^{*'}_1, \emptyset, \mathcal{F}'_1), (\emptyset, \mathcal{C}'_1, \emptyset))$,[11] where

(1) $\overline{\Pi}^{*'}_1$ consists of the following rules:
  (a) all constraints in $\overline{\Pi}_1, \dots, \overline{\Pi}_k$;
  (b) for each $r \in \overline{\Pi}_i$: $a \leftarrow \mathit{body}(r)$ or $\bar{a} \leftarrow \mathit{body}(r)$ $(i = 1, \dots, k)$, $a_i \leftarrow \mathit{body}(r), \mathit{not}\ l_{\bar{a}_i}$, or $\bar{a}_i \leftarrow \mathit{body}(r), \mathit{not}\ l_{a_i}$ respectively,
  (c) for each $a, \bar{a}$ in $\overline{\mathcal{A}}$,
  $a_{i-1} \leftarrow a_i, \bar{a}_{i-1} \leftarrow \bar{a}_i$ $(i = 1, \dots, n)$,
  $a \leftarrow a_1, \bar{a} \leftarrow \bar{a}_1$.
(2) $\mathcal{F}'_1 = \{l_{a_{k-1}}, l_{\bar{a}_{k-1}} \mid \forall a \in \mathcal{A}\}$,
(3) $\mathcal{C}'_1 = \{\leftarrow a_{k-1}, \bar{a}_k, \leftarrow \bar{a}_{k-1}, a_k \mid \forall a \in \mathcal{A}\}$.

Now we form a new $\Omega_{CR} = (\Sigma'_1, \Sigma_1, \dots, \Sigma_{k-2})$. Obviously S is model of Ω_{CR} iff S is a model of Ω'_{CR}. On the other hand, According to Definition 13, it turns out that Ω_{CR} can be viewed as such a sequence of logic program contexts formed from $\bar{\mathbf{P}} = (\overline{\Pi}_1, \dots, \overline{\Pi}_k)$.

($\Leftarrow$) Given $\bar{\mathbf{P}} = (\overline{\Pi}_1, \dots, \overline{\Pi}_k)$ and $\Omega_{CR} = (\Sigma_1, \dots, \Sigma_{k-1})$ which is specified as in Definition 13. Suppose S is a model of Ω_{CR}. We show that $S \cap \overline{\mathcal{A}}$ is an update stable model of $\bar{\mathbf{P}}$. Now we consider a subsequence of $\Omega'_{CR} = (\Sigma_2, \dots, \Sigma_{k-1})$, where $\Sigma_2 = ((\overline{\Pi}^*_2, \emptyset, \mathcal{F}_2), (\emptyset, \mathcal{C}_2, \emptyset))$, which is a preferred solution of Σ_1 in Ω_{CR}. So we can represent $\overline{\Pi}_2 = \mathit{SForgetLP}(\overline{\Pi}^*_1, P)$, where $P \subseteq \mathcal{F}_1 = \{l_{a_{k-1}}, l_{\bar{a}_{k-1}} \mid \forall a \in \mathcal{A}\}$, and $\overline{\Pi}^*_1$ is in Σ_1. Now we define a program based on $\bar{\mathbf{P}}_\triangleleft$:

$$\bar{\mathbf{P}}'_\triangleleft = \bar{\mathbf{P}}_\triangleleft - \big(\big\{r\colon a_{k-1} \leftarrow \mathit{body}(r), \mathit{not}\ \mathit{rej}(r) \mid l_{a_{k-1}} \in P\big\} \cup \big\{r\colon a_{k-1} \leftarrow \mathit{body}(r), \mathit{not}\ \mathit{rej}(r) \mid l_{\bar{a}_{k-1}} \in P\big\}\big).$$

Equivalently, we can view $\bar{\mathbf{P}}'_\triangleleft$ as the update program of a new sequence $\bar{\mathbf{P}}' = (\overline{\Pi}_1, \dots, \overline{\Pi}^*_{k-1})$ where $\overline{\Pi}^*_{k-1} = \overline{\Pi}'_{k-1} \cup \overline{\Pi}_k$, and $\overline{\Pi}'_{k-1} = \overline{\Pi}_{k-1} - \{r \mid$ those corresponding rules removed in $\bar{\mathbf{P}}'_\triangleleft\}$. Also, it is easy to verify that Ω'_{CR} can be

[10] Here we denote $\bar{S}^*$ to be the extension of $\bar{S}$ containing atoms from $\overline{\mathcal{A}}^*$.

[11] Note the difference between $\overline{\Pi}^*_1$ and $\overline{\Pi}^{*'}_1$.

generated from $\overline{\Pi}'_{k-1}$ following Definition 13. According to the induction assumption, we know that $S \cap \overline{\mathcal{A}}$ is an update stable model of $\bar{\mathbf{P}}'$.

On the other hand, since $\bar{S} = S \cap \overline{\mathcal{A}}$ is an update stable model of $\bar{\mathbf{P}}'$, we can extend $\bar{S}$ to $\bar{A}^*$ containing those atoms in $\overline{\mathcal{A}}^*$. Therefore, for each rule r: $a_{k-1} \leftarrow body(r), not\ rej(r)$ or r: $a_{k-1} \leftarrow body(r), not\ rej(r)$ removed from $\bar{\mathbf{P}}_{\triangleleft}$ (see the definition for $\bar{\mathbf{P}}'_{\triangleleft}$ above), atom $rej(r)$ should be in $\bar{A}^*$. Otherwise, this will violate the induction assumption. This follows that $\bar{S}$ must be an update model for $\bar{\mathbf{P}}$ too. This completes our proof. □

6.2. Representing dynamic logic program approach

Logic program update based on dynamic logic programs (DLP) (or simply called DLP update approach) was proposed by Alferes, Leite, Pereira, et al. [2], and then extended for various purposes [15]. DLP deals with *generalized logic programs* in which negation as failure *not* is allowed to occur in the head of a rule while classical negation $\neg$ is excluded from the entire program. Let $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ be a sequence of generalized logic programs over set of atoms $\mathcal{A}$, we extend $\mathcal{A}$ to $\mathcal{A}_D$ by adding pairwise distinct atoms $\bar{a}, a_i, \bar{a}_i, a_{P_i}, \bar{a}_{P_i}$, for each $a \in \mathcal{A}$.

Definition 14. [15] Given a update sequence $\mathbf{P} = (\Pi_1, \ldots, \Pi_n)$ over $\mathcal{A}$, where each Π_i is a generalized logic program, the corresponding *dynamic update program* $\mathbf{P}_\oplus = \Pi_1 \oplus \cdots \oplus \Pi_n$ over $\mathcal{A}_D$ is a generalized logic program consisting of the following rules:

(1) for each $r \in \Pi_i$: $head(r) \leftarrow pos(r), not\ neg(r)$,
  $a_{P_i} \leftarrow pos(r), not\ neg(r)$ if $head(r) = \{a\}$ or
  $\bar{a}_{P_i} \leftarrow pos(r), not\ neg(r)$, if $head(r) = \{not\ a\}$;
(2) for each a occurring $\mathbf{P}$ and each $i = 1, \ldots, n$,
  $a_i \leftarrow a_{P_i}$ and $\bar{a} \leftarrow \bar{a}_{P_i}$;
(3) for each a occurring $\mathbf{P}$ and each $i = 1, \ldots, n$,
  $a_i \leftarrow a_{i-1}, not\ \bar{a}_{P_i}$,
  $\bar{a}_i \leftarrow \bar{a}_{i-1}, not\ a_{P_i}$;
(4) for each a occurring $\mathbf{P}$, $\bar{a}_0 \leftarrow$, $a \leftarrow a_n$, $\bar{a} \leftarrow \bar{a}_n$, $not\ a \leftarrow \bar{a}_n$.

The semantics of DLP is defined in terms of the dynamic stable model semantics [15]. However, it is easy to characterize this through the original stable model semantics.

Proposition 7. *Given a dynamic update program* $\mathbf{P}_\oplus = \Pi_1 \oplus \cdots \oplus \Pi_n$, *we define* $\mathbf{P}^*_\oplus = \mathbf{P}_\oplus - \{not\ a \leftarrow \bar{a}_n \mid a \in \mathcal{A}\}$.[12] *Then S is a dynamic stable model of* $\mathbf{P}_\oplus$ *iff* $S = S' \cup \{not\ a \mid \bar{a}_n \in S'\}$, *where* S' *is a stable model of* $\mathbf{P}^*_\oplus$.

Now we can represent a transformation from $\mathbf{P}^*_\oplus$ to a sequence of logic program contexts which captures the dynamic logic programming update approach.

Definition 15. Given a dynamic update program $\mathbf{P}_\oplus = \Pi_1 \oplus \cdots \oplus \Pi_n$ over $\mathcal{A}_D$ (see Definition 14), and let $\mathbf{P}^*_\oplus = \mathbf{P}_\oplus - \{not\ a \leftarrow a_n^- \mid a \in \mathcal{A}\}$. We specify a sequence of logic program contexts $\Omega_{DLP} = (\Sigma_1, \ldots, \Sigma_n)$ over the set of atoms $\mathcal{A}^*_D = \mathcal{A}_D \cup \{h_{a_i}, h_{\bar{a}_i}, l_{a_i}, l_{\bar{a}_i} \mid a_i, \bar{a}_i \in \mathcal{A}_D, i = 0, \ldots, n\}$ where $h_{a_i}, h_{\bar{a}_i}, l_{a_i}, l_{\bar{a}_i}$ are newly introduced atoms:

(1) $\Sigma_1 = ((\Pi^*_1, \emptyset, \mathcal{F}_1), (\emptyset, \mathcal{C}_1, \emptyset))$, where
  (a) Π^*_1 consists of the following rules:
    (i) all rules in $\mathbf{P}^*_\oplus$ except the following rules ($i = 1, \ldots, n$):
      $a_i \leftarrow a_{i-1}, not\ \bar{a}_{p_i}$, and
      $\bar{a}_i \leftarrow \bar{a}_{i-1}, not\ a_{p_i}$,
    (ii) for each pair of rules in $\mathbf{P}^*_\oplus$ ($i = 1, \ldots, n$):
      $a_i \leftarrow a_{i-1}, not\ \bar{a}_{p_i}$, and

[12] Clearly, $\mathbf{P}^*_\oplus$ is a normal logic program.

$\bar{a}_i \leftarrow \bar{a}_{i-1}, not\ a_{p_i}$,
replace them with the following rules in Π_1^*:
$a_i \leftarrow a_{i-1}, not\ l_{a_i}$, $\bar{a}_i \leftarrow \bar{a}_{i-1}, not\ l_{\bar{a}_i}$,
$l_{a_i} \leftarrow not\ h_{a_i}$, $l_{\bar{a}_i} \leftarrow not\ h_{\bar{a}_i}$,
$h_{a_i} \leftarrow \bar{a}_{P_i}$, $h_{\bar{a}_i} \leftarrow a_{P_i}$,

(b) $\mathcal{F}_1 = \{h_{a_1}, h_{\bar{a}_1} \mid \forall a \in \mathcal{A}\}$,
(c) $\mathcal{C}_1 = \{\leftarrow a_1, \bar{a}_{P_1}, \leftarrow \bar{a}_1, a_{P_1} \mid \forall a \in \mathcal{A}\}$;

(2) $\Sigma_i = ((\Pi_i^*, \emptyset, \mathcal{F}_i), (\emptyset, \mathcal{C}_i, \emptyset))$, where
(a) $\Pi_i^* = \Pi_{i-1}^\dagger$, and $\Pi_{i-1}^\dagger$ is in a preferred solution of Σ_{i-1}:
$\Sigma'_{i-1} = ((\Pi_{i-1}^\dagger, \emptyset, \mathcal{F}_{i-1}), (\emptyset, \mathcal{C}_{i-1}, \emptyset))$,
(b) $\mathcal{F}_i = \{h_{a_i}, h_{\bar{a}_i} \mid \forall a \in \mathcal{A}\}$,
(c) $\mathcal{C}_i = \{\leftarrow a_i, \bar{a}_{P_i}, \leftarrow \bar{a}_i, a_{P_i} \mid \forall a \in \mathcal{A}\}$.

A subset $S \subseteq \mathcal{A}_D^*$ is called a *model* of Ω_{DLP} if S is a stable model of $\Pi_n^\dagger$, where $\Pi_n^\dagger$ is in a preferred solution of Σ_n: $\Sigma'_n = ((\Pi_n^\dagger, \emptyset, \mathcal{F}_n), (\emptyset, \mathcal{C}_n, \emptyset))$.

In Definition 15, the sequence of logic program contexts $\Omega_{DLP} = (\Sigma_1, \ldots, \Sigma_n)$ represents a way of solving conflicts between atoms in an *upwards* manner. Starting from $i = 1$, for each i Σ_i solves conflicts between atoms a_i and $\bar{a}_{P_i}$ (or $\bar{a}_i$ and a_{P_i} resp.) through *weakly* forgetting h_{a_i} or $h_{\bar{a}_i}$. For instance, if both a_{i-1} and $\bar{a}_{P_i}$ are derived from Π_i^*, then both a_i and $\bar{a}_i$ can be derived from Π_i^* as well. Therefore a conflict would occur. Σ_i solves such conflict by weakly forgetting h_{a_i}. In particular, after weakly forgetting h_{a_i}, rule $h_{a_i} \leftarrow \bar{a}_{P_i}$ in Π_i^* will be removed, atom l_{a_i} is then derived from $l_{a_i} \leftarrow$ (note that formula $not\ h_{a_i}$ is deleted from rule $l_{a_i} \leftarrow not\ h_{a_i}$). Consequently rule $a_i \leftarrow a_{i-1}, not\ l_{a_i}$ is defeated so that atom a_i cannot be derived from a_{i-1} via the corresponding inertia rule. This process continuous until all conflicts among atoms from level 1 to level n are solved.

Theorem 9. *Let $\mathbf{P}_\oplus^*$ be specified as above over set of atoms $\mathcal{A}_D$. A subset $S^* \subseteq \mathcal{A}_D$ is a stable model of $\mathbf{P}_\oplus^*$ iff there is a sequence of logic program contexts $\Omega_{DLP} = (\Sigma_1, \ldots, \Sigma_n)$ constructed from $\mathbf{P}_\oplus^*$ as specified in Definition* 15 *such that Ω_{DLP} has a model S satisfying $S^* = S \cap \mathcal{A}_D$.*

Since the proof for this theorem is tedious but similar to the proof of Theorem 8, we skip it here.

6.3. Representing syntax based approach

Sakama and Inoue's update approach is viewed as a typical syntax based logic program update approach [22], which solves conflicts between two programs on a basis of syntactic coherence.

To simplify our discussion, we restrict Sakama and Inoue's approach from an extended logic program setting to a normal logic program setting. Note that this restriction does not affect the result presented in this subsection. In fact, we may use the method described in last subsection to translate an extended logic program update into a normal logic program update by introducing new atoms in the underlying language.

Definition 16. [22] Let Π_1 and Π_2 be two consistent logic programs. Program Π' is a *SI-result* of a theory update of Π_1 by Π_2 if (1) Π' is consistent, (2) $\Pi_2 \subseteq \Pi' \subseteq \Pi_1 \cup \Pi_2$, and (3) there is no other consistent program Π'' such that $\Pi' \subset \Pi'' \subseteq \Pi_1 \cup \Pi_2$.

Now we transform Sakama and Inoue's theory update into a logic program context. First, for each rule $r \in \Pi_1$, we introduce a new atom l^r which does not occur in $atom(\Pi_1 \cup \Pi_2)$. Then we define a program Π'_1: for each $r \in \Pi_1$, rule r': $head(r) \leftarrow pos(r), not\ (neg(r) \cup \{l^r\})$ is in Π'_1. That is, for each $r \in \Pi_1$, we simply extend its negative body with a unique atom l^r. This will make each r' in Π'_1 be removable by strongly forgetting atom l^r without influencing other rules. Finally, we specify $\Sigma_{SI} = (\Phi_1, \Phi_2)$, where $\Phi_1 = (\Pi'_1, \emptyset, \{l^r \mid r \in \Pi_1\})$ and $\Phi_2 = (\emptyset, \Pi_2, \emptyset)$.

For convenience, we also use Π^{-notP} to denote a program obtained from Π by removing all occurrences of atoms in P from the negative bodies of all rules in Π. For instance, if $\Pi = \{a \leftarrow b, not\ c, not\ d\}$, then $\Pi^{-not\{c\}} = \{a \leftarrow b, not\ d\}$. Now we have the following characterization result.

Theorem 10. *Let Π_1 and Π_2 be two consistent programs, and Σ_{SI} as specified above. Π' is a SI-result of updating Π_1 by Π_2 iff $\Pi' = \Pi^{-not\{l^r \mid r\in\Pi_1\}} \cup \Pi_2$, where $\Sigma' = ((\Pi, \emptyset, \{l^r \mid r \in \Pi_1\}), (\emptyset, \Pi_2, \emptyset))$ is a preferred solution of Σ_{SI}.*

Proof. From the specifications of Σ_{SI} and Σ', we know that $\Pi = SForgetLP(\Pi', P)$, where P is a minimal subset of $\{l^r \mid r \in \Pi_1\}$ such that $\Pi \cup \Pi_2$ is consistent. Note that each rule $r \in \Pi$ is of the form: $head(r) \leftarrow pos(r), not(neg(r) \cup \{l^r\})$, which can actually be simplied as $head(r) \leftarrow pos(r), not\ neg(r)$ since atom l^r does not play any role in the program evaluation. That is, $\Pi' \cup \Pi_2$ is equivalent to $\Pi^{-not\{l^r \mid r\in\Pi_1\}} \cup \Pi_2$, which is a *SI*-result of the update of Π_1 with Π_2. □

6.4. Representing integrated update approach

Different from both model based and syntax based approaches, Zhang and Foo's update approach integrated both desirable semantic and syntactic features of (extended) logic program updates [27]. Their approach also solves default conflicts caused by negation as failure in logic programs by using a prioritized logic programming language. Consequently, Zhang and Foo's update approach can generate an explicit resulting program for a logic program update and also avoid some undesirable solutions embedded in Sakama–Inoue's approach [28].

Since we do not consider default conflict solving in this paper, we will only focus on the transformation from first part of Zhang–Foo's update approach, that is, the conflict (contradiction) elimination, into a logic program context.

Let Π_1 and Π_2 be two extended logic programs. Updating Π_1 with Π_2 consists of two stages. Step (1): Simple fact update—updating an answer set S of Π_1 by program Π_2. The result of this update is a collection of sets of literals, denoted as $Update(S, \Pi_2)$. Step (2): Select a $S' \in Update(S, \Pi_2)$, and extract a maximal subset Π^* of Π_1 such that program $\Pi^* \cup \{l \leftarrow \mid l \in S'\}$ (or simply represented as $\Pi^* \cup S'$) is consistent. Then $\Pi^* \cup \Pi_2$ is called a *resulting program* of updating Π_1 with Π_2.

Note that in Step (1), the simple fact update is achieved through a prioritized logic programming [27]. Recently, Zhang proved an equivalence relationship between the simple fact update and Sakama and Inoue's program update [28]:

$$Update(S, \Pi_2) = \bigcup \mathcal{S}\big(SI\text{-}Update\big(\Pi(S), \Pi_2\big)\big),$$

where $\Pi(S) = \{l \leftarrow \mid l \in S\}$, and $\bigcup \mathcal{S}(SI\text{-}Update(\Pi(S), \Pi_2))$ is the class of all answer sets of resulting programs after updating $\Pi(S)$ by Π_2 using Sakama–Inoue's approach.

Example 11. Consider two extended logic programs Π_1 and Π_2 as follows:

Π_1:	Π_2:
$a \leftarrow,$	$b \leftarrow a,$
$c \leftarrow b,$	$\neg c \leftarrow b.$
$d \leftarrow not\ e.$	

Π_1 has a unique answer set $\{a, d\}$. Then Step (1) Zhang–Foo's simple fact update of $\{a, d\}$ by Π_2, $Update(\{a, d\}, \Pi_2)$, which is equivalently to update $\{a \leftarrow, d \leftarrow\}$ with Π_2 using Sakama–Inoue's approach, will contain a single set $\{a, b, \neg c, d\}$. Applying Step (2), we obtain the final update result $\{a \leftarrow, d \leftarrow not\ e\} \cup \Pi_2$.

As we have already provided a transformation from Sakama–Inoue's approach to a logic program context, to show that Zhang–Foo's update approach can also be represented within our framework, it is sufficient to only transform Step (2) above into a conflict solving problem under certain logic program context.

As before, given two extended logic programs Π_1 and Π_2 over the set of atoms $\mathcal{A}$, we extend $\mathcal{A}$ to $\overline{\mathcal{A}}$ with new atom $\bar{a}$ for each $a \in \mathcal{A}$. Then by replacing each $\neg a$ in S' and Π_2 with $\bar{a}$, we obtain the corresponding normal logic programs $\overline{\Pi}_1$ and $\overline{\Pi}_2$ respectively. Suppose $Update(\overline{S}, \overline{\Pi}_2)$ is the result of the simple fact update, where $\overline{S}$ is a stable model of $\overline{\Pi}_1$.

Definition 17. Let $\overline{\Pi}_1$, $\overline{\Pi}_2$, and $Update(\overline{S}, \overline{\Pi}_2)$ be defined as above, and $\overline{S'} \in Update(\overline{S}, \overline{\Pi}_2)$. We specify a logic program context $\Sigma_{ZF} = ((\overline{\Pi'}_1, \emptyset, \mathcal{F}), (\emptyset, \mathcal{C}, \emptyset))$ over the set of atoms $\overline{A} \cup \{l_r \mid r \in \overline{\Pi}_1\}$ where l_r are newly introduced atoms:

(1) $\overline{\Pi'}_1$ consists of rules: (a) for each rule r: $head(r) \leftarrow pos(r), not\ neg(r)$ in $\overline{\Pi}_1$, $head(r) \leftarrow body(r), not\ l_r$ is in $\overline{\Pi'}_1$, and (b) $\overline{S'} \subseteq \overline{\Pi'}_1$,
(2) $\mathcal{F} = \{l_r \mid r \in \overline{\Pi}_1\}$,
(3) $\mathcal{C} = \{\leftarrow a, \bar{a} \mid a, \bar{a} \in \overline{A}\}$.

The following theorem shows that Step (2) in Zhang–Foo's approach can be precisely characterized by a logic program context specified in Definition 17.

Theorem 11. *Let $\overline{\Pi}_1$, $\overline{\Pi}_2$, Σ_{ZF}, and $Update(\overline{S}, \overline{\Pi}_2)$ be defined as above, and $\overline{S'} \in Update(\overline{S}, \overline{\Pi}_2)$. $\overline{\Pi}^*$ is a maximal subset of $\overline{\Pi}_1$ such that $\overline{\Pi'} = \overline{\Pi}^* \cup \overline{S'}$ is consistent iff $\overline{\Pi''}$ is in a preferred solution of Σ_{ZF}: $\Sigma'_{ZF} = ((\overline{\Pi''}, \emptyset, \mathcal{F}), (\emptyset, \mathcal{C}, \emptyset))$, where $\overline{\Pi''} = \{r\colon head(r) \leftarrow pos(r), not\ neg(r), not\ l^r \mid r \in \overline{\Pi}^*\}$.*

The proof of Theorem 11 is similar to that of Theorem 10.

6.5. Further discussions: Updates, constraints, and expressiveness

From previous descriptions, we observe that the key step to transform an update approach into a sequence of logic program contexts (or one logic program context like the case of SI approach) is to construct the underlying constraints for conflict solving. In both Eiter et al.'s causal rejection and DLP approaches, constraints are specified based on atoms, e.g. $\leftarrow a_{n-i}, \bar{a}_{n-i+1}$ in Ω_{CR}, and $\leftarrow a_i, a_{P_i^-}$ in Ω_{DLP}.

For SI approach, on the other hand, the underlying constraints are specified as the entire update program. For instance, consider the update of Π_1 by Π_2 using SI approach, the corresponding logic program context for this update is of the form $\Sigma = ((\Pi, \emptyset, \mathcal{F}), (\emptyset, \Pi_2, \emptyset))$, in which program Π_2 serves as constraints for conflict solving.

Finally, since Zhang and Foo's integrated update approach combined both model and syntax based approaches, the transformation of this approach into logic program context framework consists of two steps: an equivalent SI transformation with program based constraints, followed by another transformation with atoms based constraints (see Definition 17).

From the above observation, we can see that the main difference between model based and syntax based update approaches is to solve conflicts under different types of constraints, namely atoms based and program based constraints respectively.

While we have shown that our conflict solving approach provides a unified framework to represent different kinds of logic program updates, we should indicate that our approach does not give specific computational advantages over these logic program update approaches. As we will see in Section 7, conflict solving under our framework is generally intractable. From previous definitions, we also observe that transforming model based logic program updates into a sequence of logic program contexts may need exponential time because it involves the computation of solutions of logic program contexts, although transforming syntax based logic program updates can always be done in polynomial time.

Nevertheless, the most significant feature of using our logic program contexts to represent logic program updates is to provide an expressive framework that unifies many different logic program update approaches. Under the unified framework, it becomes possible to analyze and compare syntactic and semantic properties of these different approaches.

7. Computational issues

In this section, we study related computational issues. In particular, we consider two major computational problems concerning (1) irrelevance in reasoning with respect to strong and weak forgettings and conflict solving, and (2) general decision problems for conflict solving under the framework of logic program contexts.

We first introduce basic notions from complexity theory and refer to [21] for further details. Two important complexity classes are P and NP. The class P includes all languages recognizable by a polynomial-time deterministic Turing machine. The class NP, on the other hand, consists of those languages recognizable by a polynomial-time nondeterministic Turing machine. The class of coNP is the complements of class NP. The class of DP contains all languages L such that $L = L_1 \cap L_2$ where L_1 is in NP and L_2 is in coNP. The class coDP is the complement of class DP. The class $\Sigma_2^P = NP^{NP}$ includes all languages recognizable in polynomial time by a nondeterministic Turing machine with an NP oracle, where the class Π_2^P is the complement of Σ_2^P, i.e. $\Pi_2^P = co\Sigma_2^P$. It is well known that $P \subseteq NP \subseteq DP \subseteq \Sigma_2^P$, and these inclusions are generally believed to be proper.

7.1. Complexity results on irrelevance

By definitions, we can see that the main computation of strong and weak forgettings relies on the procedure of reduction that further inherits the computation of the conventional program unfolding. Hence, it is easy to observe that in the worst case, the size of the resulting program after strong (or weak) forgetting could be exponentially larger than the original program. This means that in general computing strong and weak forgettings in logic programs is hard. However, the following result shows that this actually does not increase the complexity of the associated inference problem.

Theorem 12. *Let Π be a logic program, P a set of atoms, and a an atom. Deciding whether $SForgetLP(\Pi, P) \models a$ (or $WForgetLP(\Pi, P) \models a$) is coNP-complete.*

Proof. The hardness is obvious when $P = \emptyset$. To prove the membership, we first specify two transformations on Π with respect to P. The program $STrans(\Pi, P)$ is obtained from Π by removing some rules in Π: (1) for each $p \in P$, if $p \notin head(\Pi)$, then removing rules r in Π with $p \in pos(r)$; (2) if $p \notin pos(\Pi)$, then removing rules r in Π with $head(r) = p$; and (3) removing rules r in Π with $p \in neg(r)$. The program $WTrans(\Pi, P)$, on the other hand, is obtained from Π in the same way as program $STrans(\Pi, P)$ except (3): for rules r in Π having $p \in neg(r)$, change it to be of the form: r': $head(r) \leftarrow pos(r), not(neg(r) - \{p\})$. Now we prove the following two results:

Result 1. *$SForgetLP(\Pi, P)$ is consistent if and only if program $STrans(\Pi, P)$ is consistent, and each of $SForgetLP(\Pi, P)$'s stable models S' can be expressed as $S' = S - P$, where S is a stable model of $STrans(\Pi, P)$.*

Result 2. *$WForgetLP(\Pi, P)$ is consistent if and only if program $WTrans(\Pi, P)$ is consistent, and each of $WForgetLP(\Pi, P)$'s stable models S' can be expressed as $S' = S - P$, where S is a stable model of $WTrans(\Pi, P)$.*

Here we give the proof of Result 1, while Result 2 can be proved in a similar way. Firstly, we assume that $SForgetLP(\Pi, P)$ is consistent and S' is a stable model of $SForgetLP(\Pi, P)$. Then we show that $STrans(\Pi, P)$ must have a stable model S such that $S' = S - P$. Observing the construction of the structure of $STrans(\Pi, P)$, we can see that for each $p \in P$ occurring in $STrans(\Pi, P)$, there are two rules r_1 and r_2 in $STrans(\Pi, P)$ of the forms:

$$r_1\colon\ p \leftarrow pos(r_1), not\ neg(r_1),$$

$$r_2\colon\ head(r_2) \leftarrow p, pos(r_2), not\ neg(r_2),$$

and furthermore, we also have $P \cap neg(STrans(\Pi, P)) = \emptyset$. Now we present an algorithm to construct a set S of atoms as follows:

```
Algorithm: **Generating** $S$
**Input**: $STrans(\Pi, P)$ and $S'$ where $S'$ is a stable model of $SForgetLP(\Pi, P)$;
**Output**: a set $S$ of atoms;
**let** $S = S'$;
selecting a rule $r$ from $STrans(\Pi, P)$ of the form:
    $r$: $p \leftarrow pos(r), not\ neg(r)$, where $p \in P$ and $pos(r) \cap P = \emptyset$;
**if** no such rule exists in $Strans(\Pi, P)$, **then return** $S$;
**else**
    **if** each $a \in pos(r)$ is in $S'$ and each $b \in neg(r)$ is not in $S'$,
        **then** $S = S \cup \{p\}$;
**repeat** the following two steps until $S$ no longer changes
    selecting a rule $r'$ from $STrans(\Pi, P)$ of the form:
        $r'$: $p \leftarrow pos(r'), not\ neg(r')$ where $p \in P$;
    if each $a \in pos(r')$ is in $S$ and each $b \in neg(r')$ is not in $S$,
        then $S = S \cup \{p\}$;
return $S$.
```

We need to show that S generated from the above algorithm is a stable model of *STrans*(Π, P). We perform Gelfond–Lifschitz transformation on *STrans*(Π, P) with S, and obtain program *STrans*$(\Pi, P)^S$. First, we prove that for each rule r: $head(r) \leftarrow pos(r)$ in *STrans*$(\Pi, P)^S$, if $pos(r) \subseteq S$, then $head(r) \in S$.

Case 1. If $pos(r) \subseteq S'$, then $head(r) \subseteq S$ according to the algorithm.

Case 2. Suppose r is of the form: r: $head(r) \leftarrow p, pos(r)$, where $p \in P$, $\{p\} \cup pos(r) \subseteq S$ and $pos(r) \subseteq S'$. In this case, we show $head(r) \in S$. This is true if $head(r) \in P$ according to the above algorithm. Now suppose $head(r) \not\subseteq P$. Consider r's original form in *STrans*(Π, P): r': $head(r) \leftarrow p', pos(r), not\ neg(r')$ (i.e. the part $not\ neg(r')$ is removed in *STrans*$(\Pi, P)^S$). Recall the structure of *STrans*(Π, P), in which there exists a rule r'': $p \leftarrow pos(r''), not\ neg(r'')$. By performing proper reduction, eventually we can replace r'' with a new rule: r^*: $p \leftarrow pos(r^*), not\ neg(r^*)$ such that $P \cap pos(r^*) = \emptyset$ (note that if we can not reach this form of rule r^*, for instance, $P \cap pos(r^*) \neq \emptyset$, we will have $p \notin S$ according to the above algorithm). As $p \in S$, we must have $pos(r^*) \subseteq S$, and hence $pos(r^*) \subseteq S'$. On the other hand, it is not hard to observe that a rule of the form is in *SForgetLP*$(\Pi, P)^{S'}$: $head(r) \leftarrow pos(r), pos(r^*)$. Since we already know that $pos(r) \cup pos(r^*) \subseteq S'$ and S' is a stable model of *SForgetLP*(Π, P), it follows that $head(r) \in S'$ and hence $head(r) \in S$ as $S' \subseteq S$.

On the other hand, it is also easy to show that S' generated from the above algorithm is the smallest set to have the above property for program *STrans*(Π, P). This proves that S is a stable model of *STrans*(Π, P).

Now we assume that *STrans*(Π, P) is consistent and S is a stable model of *STrans*(Π, P). In this case, we simply prove that $S' = S - P$ is a stable model of *SForgetLP*(Π, P). We omit the proof as it is easy to verify.

Having these results, the membership is proved as follows. For the case of strong forgetting, we consider the complement of the problem. Clearly, it is easy to see that the *STrans*(Π, P) can be obtained from Π in polynomial time. Guessing a S stable model of *STrans*(Π, P), verifying it, and checking whether $a \notin S - P$ can be done in polynomial time. So the complement of the problem is in NP. Consequently, the problem is in coNP. Proof for the case of weak forgetting is the same. □

From the above result, we can show the complexity of irrelevance in relation to strong and weak forgettings.

Theorem 13. *Let Π be a logic program, P a set of atoms and a an atom. Deciding whether a is irrelevant to P in Π is coDP-complete.*

Proof. To prove this theorem, we need to show deciding whether $\Pi \models a$ iff *SForgetLP*$(\Pi, P) \models a$ (s-irrelevant) is coDP-complete, and deciding whether $\Pi \models a$ iff *WForgetLP*$(\Pi, P) \models a$ (w-irrelevant) is coDP-complete. Here we only give the proof of the first statement, and the second can be proved in a similar way.

Membership. To decide whether $\Pi \models a$ iff *SForgetLP*$(\Pi, P) \models a$, we need to show $\Pi \models a$ and *SForgetLP*$(\Pi, P) \models a$, or $\Pi \not\models a$ and *SForgetLP*$(\Pi, P) \not\models a$. Clearly, given Π, P and a, deciding whether $\Pi \models a$ and *SForgetLP*$(\Pi, P) \models a$ is in coNP, and deciding whether $\Pi \not\models a$ and *SForgetLP*$(\Pi, P) \not\models a$ is in NP (see Theorem 12). So the problem is in coDP.

Hardness. We consider a pair (Φ_1, Φ_2) of CNFs and from which we polynomially construct a program Π, a set of atoms P and an atom a, and prove that Φ_1 is satisfiable or Φ_2 is unsatisfiable iff $\Pi \models a$ and *SForgetLP*$(\Pi, P) \models a$, or $\Pi \not\models a$ and *SForgetLP*$(\Pi, P) \not\models a$.

Let $\Phi_1 = \{C_1, \dots, C_m\}$ and $\Phi_2 = \{C'_1, \dots, C'_n\}$, where each C_i and C'_j ($1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant n$) are sets of propositional literals respectively. We also assume that Φ_1 and Φ_2 do not share any propositional atoms. Now we construct a program Π based on propositional atoms $atom(\Phi_1) \cup atom(\Phi_2) \cup \hat{X} \cup \hat{Y} \cup \{l_1, \dots, l_n, p, a, sat^{\Phi_1}, unsat^{\Phi_1}, unsat^{\Phi_2}\}$, where any two sets of atoms are disjoint and $|\hat{X}| = |atom(\Phi_1)|$ and $|\hat{Y}| = |atom(\Phi_2)|$. Program Π consists of four groups of rules:

Π_1:

for each $x \in atom(\Phi_1)$, we have:

$x \leftarrow not\ \hat{x}$,

$\hat{x} \leftarrow not\ x$,

for each $y \in atom(\Phi_2)$, we have:

$y \leftarrow not\ \hat{y}$,

$\hat{y} \leftarrow not\ y$,

Π_2:

$unsat^{\Phi_1} \leftarrow \overline{C_1}$,

$\ldots$,

$unsat^{\Phi_1} \leftarrow \overline{C_m}$,

$unsat^{\Phi_2} \leftarrow \overline{C'_1}$,

$\ldots$,

$unsat^{\Phi_2} \leftarrow \overline{C'_n}$,

where for each clause C_i (or C'_j), if $b \in C_i$ (or C'_j resp.), then $not\ b \in \overline{C_i}$ (or $\overline{C'_j}$ resp.), and if $\neg b \in C_i$ (or C'_j resp.) then $b \in \overline{C_i}$ (or $\overline{C'_j}$ resp.),

Π_3:

$l_1 \leftarrow unsat^{\Phi_2}, not\ l_2, \ldots, not\ l_n$,

$\ldots$,

$l_n \leftarrow unsat^{\Phi_2}, not\ l_1, \ldots, not\ l_{n-1}$,

$pos(\overline{C'_j}) \leftarrow l_j\ (1 \leqslant j \leqslant n)$,

where $pos(\overline{C'_j}) \leftarrow l_j$ represents a group of rules: for all atoms $b \in \overline{C'_j}$, we have $b \leftarrow l_j$ (note that if $not\ b \in \overline{C'_j}$, no rule is needed),

Π_4:

$sat^{\Phi_1} \leftarrow not\ unsat^{\Phi_1}$,

$a \leftarrow sat^{\Phi_1}$,

$unsat^{\Phi_2} \leftarrow not\ a$,

$p \leftarrow .$

Let us look at the intuition behind this program. Clearly, Π_1 generates all truth assignments for Φ_1 and Φ_2 (recall that $atom(\Phi_1) \cap atom(\Phi_2) = \emptyset$). This ensures that there is a correspondence between stable models of Π and truth assignments of Φ_1 and Φ_2. Π_2 indicates that if Φ_1 (or Φ_2) is unsatisfiable, then atom $unsat^{\Phi_1}$ (or $unsat^{\Phi_2}$ resp.) will be derived. Rules in Π_3 are used to force Φ_2 to be unsatisfiable. That is, if atom $unsat^{\Phi_2}$ is derived from through rule $unsat^{\Phi_2} \leftarrow \texttt{not}\ a$ in Π_4, then the corresponding truth assignment of Φ_2 in each stable model of Π must make some $\overline{C'_j}$ to be true.

Now we prove that Φ_1 is satisfiable or Φ_2 is unsatisfiable if and only if $\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$; or $\Pi \not\models a$ and $SForgetLP(\Pi, \{p\}) \not\models a$. We observe that $SForgetLP(\Pi, \{p\}) = \Pi - \{p \leftarrow\}$, which implies that if $\Pi \models a$ then $SForgetLP(\Pi, \{p\}) \models a$ and if $\Pi \not\models a$ then $SForgetLP(\Pi, \{p\}) \not\models a$.

Suppose that Φ_1 is satisfiable or Φ_2 is unsatisfiable. We consider the following cases. (1) If Φ_1 is satisfiable, then it is easy to see that none of rules in Π_2 with head $unsat^{\Phi_1}$ is applicable and hence atoms sat^{Φ_1} and a can be derived from Π. In this case, no matter if Φ_2 is satisfiable or unsatisfiable, we always have $\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$.

(2) If Φ_2 is unsatisfiable. In this case one of rules in Π_2 having $unsat^{\Phi_2}$ as heads is applicable and hence atom $unsat^{\Phi_2}$ is derivable from Π. In this case, if Φ_1 is satisfiable, then a is derived from Π. Otherwise, a is not derivable from Π. The same for $SForgetLP(\Pi, \{p\})$. So we have the statement: if Φ_1 is satisfiable or Φ_2 is unsatisfiable, then $\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$, or $\Pi \not\models a$ and $SForgetLP(\Pi, \{p\}) \not\models a$.

Suppose $\Pi \models a$ and $SForgetLP(\Pi, \{p\}) \models a$; or $\Pi \not\models a$ and $SForgetLP(\Pi, \{p\}) \not\models a$. (1) If $\Pi \models a$ and hence $SForgetLP(\Pi, \{p\}) \models a$. From the construction of Π, we know that the only way to derive a from Π is that rule $a \leftarrow sat^{\Phi_1}$ in Π_4 is applicable. This implies that none of rules in Π_2 having $unsat^{\Phi_1}$ as heads is applicable. Consequently, one of truth assignments generated from Π_1 for Φ_1 must satisfy Φ_1. So Φ_1 is satisfiable.

(2) If $\Pi \not\models a$ and hence $SForgetLP(\Pi, \{p\}) \not\models a$. In this case, sat $unsat^{\Phi_2}$ can be derived from rule $unsat^{\Phi_2} \leftarrow not\, a$. Then from rule in Π_3, we know that in each stable model of Π, the corresponding truth assignment of Φ_2 must not satisfy Φ_2. Since all truth assignments of Φ_2 have been represented in Π's stable models, this concludes that Φ_2 is unsatisfiable. This proves our result. □

The following complexity result of irrelevance with respect to logic program contexts is inherited from Theorem 13.

Theorem 14. *Let Σ and Σ' be two logic program contexts where $\Sigma' \in Solution(\Sigma)$, and a an atom. Deciding whether a is $(\Sigma, \Sigma')^i$-irrelevant is coDP-complete.*

7.2. Complexity results on conflict solving

Proposition 8. *Let Σ be a logic program context. Deciding whether Σ has a preferred solution is NP-hard.*

Proof. We consider a special form of logic program context $\Sigma = ((\Pi_1, \emptyset, \emptyset), \ldots, (\Pi_n, \emptyset, \emptyset))$. Clearly, Σ has a solution iff each Π_i has a stable model, and we know checking whether a program has stable is NP-hard. On the other hand, from Theorem 6, we know that Σ has a preferred solution iff $Solution(\Sigma) \neq \emptyset$. Then the result directly follows. □

We observe that computing a solution for a logic program context consists of two major stages: (1) computing strong and weak forgettings, and (2) consistency testing for all $\Pi_i \cup \mathcal{C}_j$ in the resulting logic program context (see Definition 6). While many existing results may be used for efficient consistency testing of a logic program (e.g. see Section 5.2 and Chapter 3 in [3]), it is important to investigate possible optimizations for computing strong and weak forgettings in logic programs.

For this purpose, we first introduce a useful notion. Let Π be a logic program, a an atom in $atom(\Pi)$, and $G(\Pi)$ the dependency graph of Π. In $G(\Pi)$, we call a positive path[13] without cycles starting from a the *inference chain* starting from a. We define the *inference depth* of a, denoted as $i\text{-}depth(a)$, to be the length of the longest inference chain starting from a in $G(\Pi)$. Intuitively, $i\text{-}depth(a)$ represents the maximal number of rules that may be used to derive any other atoms starting from a in program Π. We denote the *inference depth* of Π as

$$i\text{-}depth(\Pi) = Max\big(i\text{-}depth(a)\colon\ a \in atom(\Pi)\big).$$

It turns out that the inference depth plays a key role in characterizing the computation of strong and weak forgettings in logic programs.

Theorem 15. *Let Π be a logic program. If Π has a bounded inference depth, i.e. $i\text{-}depth(\Pi) \leqslant c$ for some constant c, then for any set of atoms $P \subseteq atom(\Pi)$, $SForgetLP(\Pi, P)$ and $WForgetLP(\Pi, P)$ can be computed in polynomial time.*

Proof. To prove this theorem, we only need to show that under the condition of bounded inference depth, $Reduct(\Pi, P)$ is polynomially achievable for any $P \subseteq atom(\Pi)$. Without loss of generality, for $P = \{p_1, \ldots, p_k\}$, we may assume that Π consists of three components:

Π_1:

$r_{11}\colon\ p_1 \leftarrow pos(r_{11}), not\ neg(r_{11}),$

$\ldots,$

$r_{1l_1}\colon\ p_1 \leftarrow pos(r_{1l_1}), not\ neg(r_{1l_1}),$

13 That is, a path does not contain any negative edges.

r_{21}: $p_2 \leftarrow pos(r_{21}), not\ neg(r_{21})$,

$\ldots$,

r_{2l_2}: $p_2 \leftarrow pos(r_{2l_2}), not\ neg(r_{2l_2})$,

$\ldots$,

r_{k1}: $p_k \leftarrow pos(r_{k1}), not\ neg(r_{k1})$,

$\ldots$,

r_{kl_k}: $p_k \leftarrow pos(r_{kl_k}), not\ neg(r_{kl_k})$,

Π_2[14]:

r_1: $head(r_1) \leftarrow p_1, pos(r_1), not\ neg(r_1)$,

r_2: $head(r_2) \leftarrow p_2, pos(r_2), not\ neg(r_2)$,

$\ldots$,

r_k: $head(r_k) \leftarrow p_k, pos(r_k), not\ neg(r_k)$,

Π_3,

where the reduction only occurs among rules in $\Pi_1 \cup \Pi_2$, and Π_3 contains all rules irrelevant to the reduction process. Now we show that if $i\text{-}depth(\Pi) \leqslant c$ for some constant c, the size of $Reduct(\Pi, P)$ will be at most polynomial times of the size of Π. Indeed, since $i\text{-}depth(\Pi) \leqslant c$, it follows that for each $p_i \in P$, $i\text{-}depth(p_i) \leqslant c$ in program Π_1. This implies that during the reduction, for each p_i's occurrence in other rule's positive body, at most only $h_1 \times \cdots \times h_{c+1}$, where $\{h_1, \ldots, h_{c+1}\} \subseteq \{l_1, \ldots, l_k\}$, new rules will be introduced due to the inference chain in Π_1 starting from a. This number of rules is bounded by $|\Pi|^{c+1}$. If p_i occurs in all other rules' positive bodies in Π_1, the total number of new rules possibly introduced through reduction via p_i is bounded by $|P| \times |\Pi|^{c+1}$. Therefore, the number of all new rules introduced through the entire reduction via P is bounded by $\mathcal{O}(|P|^2 \times |\Pi|^{c+1})$. In other words, to perform $Reduct(\Pi, P)$, the number of all operations on rule substitutions and replacements is bounded by $\mathcal{O}(|P|^2 \times |\Pi|^{c+1})$. □

Theorem 16. *Let $\Sigma = (\Phi_1, \ldots, \Phi_n)$ and $\Sigma' = (\Phi'_1, \ldots, \Phi'_n)$ be two logic program contexts, where for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ $(1 \leqslant i \leqslant n)$, $\Phi'_i \in \Sigma'$ is of the form $\Phi'_i = (\Pi'_i, \mathcal{C}_i, \mathcal{F}_i)$, where $\Pi'_i = SForgetLP(\Pi_i, P_i)$ or $\Pi'_i = WForgetLP(\Pi_i, P_i)$ for some $P_i \subseteq \mathcal{F}_i$. Then the following results hold*:

(1) *Deciding whether Σ' is a solution of Σ is NP-complete*;
(2) *Deciding whether Σ' is a preferred solution of Σ is in Π_2^P provided that strong and weak forgettings in Σ can be computed in polynomial time*;[15]
(3) *For a given atom a, deciding whether for all $\Sigma'' \in Solution(\Sigma)$, $\Sigma'' \models_i a$ is in Π_2^P provided that strong and weak forgettings in Σ can be computed in polynomial time.*

Proof. Result 1 is easy to prove. To check if Σ' is a solution of Σ, we only need to check whether $\Pi'_i \cup C_j$ is consistent for all i and j, and altogether we need to do n^2 such consistency checkings. On the other hand, we know that checking the consistency of $\Pi'_i \cup C_j$ is in NP. So the problem is in NP. For the hardness, just consider a special case where $n = 1$, then Σ' is a solution of Σ iff $\Pi'_1 \cup C_1$ is consistent, and this is NP-hard.

To prove Result 2, we consider the complement of the problem. If Σ' is not a preferred solution of Σ, then there must exist Σ'' such that $\Sigma'' \in Solution(\Sigma)$ and $\Sigma'' \prec_\Sigma \Sigma'$. This equals to that there are $P''_1, \ldots, P''_n$ where $P''_i \subseteq P_i$ and for some k we have $P''_k \subset P_k$ such that (1) $\Sigma'' = ((\Pi''_n, \mathcal{C}_1, \mathcal{F}_1), \ldots, (\Pi''_n, \mathcal{C}_n, \mathcal{F}_n))$, and each Π''_i is of the form $SForgetLP(\Pi_i, P''_i)$ or $WForgetLP(\Pi_i, P''_i)$; and (2) $\Sigma'' \in Solution(\Sigma)$. Clearly, guessing such $P''_1, \ldots, P''_n$ and computing each $SForgetLP(\Pi_i, P''_i)$ and $WForgetLP(\Pi_i, P''_i)$ can be done in polynomial time. Then we can

[14] In Π_2, there may be more than one rules having p_i in their positive bodies. But this simplified case does not affect our proof.

[15] Computing strong and weak forgettings in Σ, we mean that for each $\Phi_i = (\Pi_i, \mathcal{C}_i, \mathcal{F}_i) \in \Sigma$ and $P \subseteq \mathcal{F}_i$, we compute $SForgetLP(\Pi_i, P)$ and $WForgetLP(\Pi_i, P)$.

construct a Σ'' in polynomial time, where Σ'' is of the form $\Sigma'' = ((\Pi_1'', \mathcal{C}_i, \mathcal{F}_1), \ldots, (\Pi_n'', \mathcal{C}_n, \mathcal{F}_n))$, in which for each i, Π_i'' can be either *SForgetLP*(Π_i, P_i'') or *WForgetLP*(Π_i, P_i''). Then checking whether Σ'' is a solution of Σ can be achieved with number of n^2 calls for an NP oracle. So the problem is in Σ_2^P. Consequently, the original problem is in Π_2^P.

We prove Result 3 as follows. We guess a set of atoms S_i, and n sets of atoms $P_1, \ldots, P_n$ such that $P_i \subseteq \mathcal{F}_i$ for each $1 \leqslant i \leqslant n$. Then similarly to the proof of Result 2, we can construct a logic program context Σ in polynomial time. Checking whether $\Sigma' \in Solution(\Sigma)$ can be achieved with one call to an NP oracle. Then checking whether S_i is a stable model of a particular Π_i', where $\Phi_i' \in \Sigma'$ and $\Phi_i' = (\Pi', \mathcal{C}_i, \mathcal{F}_i)$, and $a \notin S_i$ can be done in polynomial time as well. So the complement of the problem is in Σ_2^P, and thus the original problem is in Π_2^P. □

8. Conclusions

In this paper, we defined notions of strong and weak forgettings in logic programs, which may be viewed as an analogy of forgetting in propositional theories. Based on these notions, we developed a framework of logic program contexts. We then studied the irrelevance property related to strong and weak forgettings and conflict solving and provided various solution characterizations for logic program contexts. We showed that our approach presented in this paper is quite general and unified all major logic program update approaches. We also analyzed the computational complexity of strong and weak forgettings in logic programs and conflict solving in logic programs contexts.

We noted that there were other methods for solving the inconsistency of logic programs in the literature, especially the work involving abductive reasoning in logic programs. For instance, Inoue's method of deletion and addition of names of rules [8], where certain atoms can be blocked from derivation by removing/adding some rules in the program. In this case, these atoms are still presented in the program. As we have shown in Section 6.3, by introducing new atom such as l^r in the language, our approach can simply model this method to solve program inconsistency. The main difference between our approach and others is that we presented a very general framework based on strong and weak forgettings, and this framework can handle many different types of conflict solving scenarios including logic program updates, negotiation and belief merging, that seem to be difficult for any other single method in the literature (e.g. see Example 6 in Section 4).

Our work presented in this paper can be further extended. One interesting issue is to integrate dynamic preference orderings on forgettable atoms into the current framework of logic program contexts, so that the extended framework can represent domain-dependent conflict solving cases. This is particularly important when we use this approach to represent complex belief merging (e.g. [10,11,16]) and negotiations under the setting of logic programming, in which each agent usually has different preferences on the atoms that she may forget for a final agreement.

Acknowledgements

The authors thank Fangzhen Lin for useful discussions on this topic and valuable comments on an early draft of this paper. The research of the first author was supported in part by Australian Research Council under grant DP0666540. The authors also thank the anonymous reviewers for their insightful comments and criticisms which were helpful in revising and improving the paper.

References

[1] K.R. Apt, H.A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Database and Logic Programming, Morgan Kaufmann, 1988, pp. 293–322.
[2] J.J. Alferes, J.A. Leite, L.M. Pereira, et al., Dynamic logic programming, in: Proceedings of KR-98, 1998, pp. 98–111.
[3] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University, 2002.
[4] S. Brass, J. Dix, A general framework for semantics of disjunctive logic programs based on partial evaluation, Journal of Logic Programming 38 (3) (1998) 167–213.
[5] T. Eiter, M. Fink, G. Sabbatini, H. Tompits, On properties of update sequences based on causal rejection, Theory and Practice of Logic Programming 2 (2002) 711–767.
[6] T. Eiter, M. Fink, G. Sabbatini, H. Tompits, Reasoning about evolving nonmonotonic knowledge base, ACM Transaction on Computational Logic 6 (2005) 389–440.
[7] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the International Conference on Logic Programming, The MIT Press, 1988, pp. 1070–1080.

[8] K. Inoue, A simple characterization of extended abduction, in: Proceedings of the First International Conference on Computational Logic (CL-2000), 2000, pp. 718–732.
[9] K. Inoue, C. Sakama, Update of equivalence of logic programs, in: Proceedings of JELIA 2004, 2004.
[10] S. Konieczny, R. Pino Pérez, On the logic of merging, in: Proceedings of the 6th International Conference on Knowledge Representation and Reasoning (KR-98), 1998, pp. 488–498.
[11] S. Konieczny, R. Pino Pérez, Propositional belief base merging or how to merge beliefs/goal coming from several sources and some links with social choice theory, European Journal of Operational Research 160 (3) (2005) 785–802.
[12] K. Kunen, Signed data dependencies in logic programs, Journal of Logic Programming 7 (3) (1989) 231–245.
[13] J. Lang, P. Marquis, Resolving inconsistencies by variable forgetting, in: Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR-2002), Morgan Kaufmann Publishers, 2002, pp. 239–250.
[14] J. Lang, P. Liberatore, P. Marquis, Propositional independence—Formula-variable independence and forgetting, Journal of Artificial Intelligence Research 18 (2003) 391–443.
[15] J.A. Leite, Evolving Knowledge Bases: Specification and Semantics, IOS Press, 2003.
[16] P. Liberatore, M. Schaerf, A system for the integration of knowledge bases, in: Proceedings of the 7th International Conference on Knowledge Representation and Reasoning (KR-2000), Morgan Kaufmann Publishers, 2000, pp. 145–152.
[17] V. Lifschitz, D. Pearce, A. Valverde, Strongly equivalent logic programs, ACM Transactions on Computational Logic 2 (4) (2001) 426–541.
[18] F. Lin, R. Reiter, Forget it!, in: Working Notes of AAAI Fall Symposium on Relevance, 1994, pp. 154–159.
[19] F. Lin, On the strongest necessary and weakest sufficient conditions, Artificial Intelligence 128 (2001) 143–159.
[20] F. Lin, Y. Chen, Discovering classes of strongly equivalent logic programs, in: Proceedings of IJCAI-2005, 2005.
[21] C.H. Papadimitriou, Computational Complexity, Addison Wesley, 1995.
[22] C. Sakama, K. Inoue, Updating extended logic programs through abduction, in: Proceedings of LPNMR'99, 1999, pp. 2–17.
[23] C. Sakama, H. Seki, Partial deduction in disjunctive logic programming, Journal of Logic Programming 32 (3) (1997) 229–245.
[24] K. Su, G. Lv, Y. Zhang, Reasoning about knowledge by variable forgetting, in: Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning (KR-2004), Morgan Kaufmann Publishers, 2004, pp. 576–586.
[25] J.-H. You, L. Yuan, A three-valued semantics for deductive databases and logic programs, Journal of Computer and System Sciences 49 (2) (1994) 334–361.
[26] Y. Zhang, Two results for prioritized logic programming, Theory and Practice of Logic Programming 3 (2) (2003) 223–242.
[27] Y. Zhang, N. Foo, Updating logic programs, in: Proceedings of ECAI-1998, 1998, pp. 403–407.
[28] Y. Zhang, Logic program based updates, ACM Transaction on Computational Logic, submitted for publication http://www.acm.org/pubs/tocl/accepted.html, 2006.

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Artificial Intelligence 170 (2006) 779–801

Artificial Intelligence

www.elsevier.com/locate/artint

Automated reformulation of specifications by safe delay of constraints [☆]

Marco Cadoli, Toni Mancini [*]

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, I-00198 Roma, Italy

Received 14 September 2004; received in revised form 25 January 2006; accepted 25 January 2006

Abstract

In this paper we propose a form of reasoning on *specifications* of combinatorial problems, with the goal of reformulating them so that they are more efficiently solvable. The reformulation technique highlights constraints that can be safely "delayed", and solved afterwards. Our main contribution is the characterization (with soundness proof) of safe-delay constraints with respect to a criterion on the specification, thus obtaining a mechanism for the automated reformulation of specifications applicable to a great variety of problems, e.g., graph coloring, bin-packing, and job-shop scheduling. This is an advancement with respect to the forms of reasoning done by state-of-the-art-systems, which typically just detect linearity of specifications. Another contribution is an experimentation on the effectiveness of the proposed technique using six different solvers, which reveals promising time savings.

Keywords: Modelling; Reformulation; Second-order logic; Propositional satisfiability; Constraint satisfaction problems

1. Introduction

Current state-of-the-art languages and systems for constraint modelling and programming (e.g., AMPL [22], OPL [48], XPRESS$^{\mathrm{MP}}$,[1] GAMS [9], DLV [31], SMODELS [39], ESRA [21], PS [18] and NP-SPEC [8]) exhibit a strong separation between a problem *specification* (e.g., Graph 3-coloring) and its *instance* (e.g., a graph), usually adopting a two-level architecture for finding solutions: the specification is firstly instantiated (or grounded) against the instance, and then an appropriate solver is invoked (cf. Fig. 1). Such a separation leads to several advantages: obviously declarativeness increases, and the solver is completely decoupled from the specification. Ideally, the programmer can focus only on the combinatorial aspects of the problem specification, without committing *a priori* to a specific solver. In

[☆] This paper is an extended and revised version of [M. Cadoli, T. Mancini, Automated reformulation of specifications by safe delay of constraints, in: Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2004), Whistler, BC, Canada, AAAI Press/The MIT Press 2004, pp. 388–398].

[*] Corresponding author.

E-mail addresses: cadoli@dis.uniroma1.it (M. Cadoli), tmancini@dis.uniroma1.it (T. Mancini).

[1] Cf. http://www.dashoptimization.com.

0004-3702/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2006.01.008

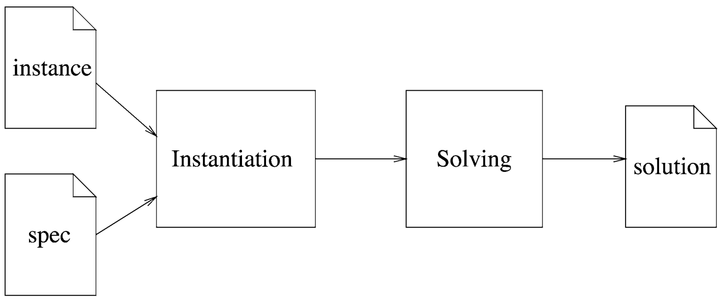


Fig. 1. Two-level architecture of current problem solving systems.

fact, some systems, e.g., AMPL, are able to translate—at the request of the user—a specification in various formats, suitable for different solvers, e.g., among the others, CPLEX, MINOS,[2] LANCELOT.[3]

Nonetheless, many existing techniques proposed in the literature for optimizing the solution of constraint satisfaction problems apply after the commitment to the instance: notable examples are, e.g., symmetry detection and breaking (cf., e.g., [4,16,37]), the development of techniques for imposing various local consistency notions and of heuristics during search (cf., e.g., [17]), the development of algorithms that deal with dependent variables, e.g., those added to SAT instances during the clausification of non-CNF formulae [27], and with the so-called "equivalence clauses" [32].

However, in many cases, properties that are amenable to be optimized derive from the problem structure, rather than the particular instance considered. Optimization techniques that act on the problem structure have been proposed. They include the addition of implied constraints (cf., e.g., [47]), the deletion or abstraction of some of the constraints (cf., e.g., [28]), the use of redundant models, i.e., multiple viewpoints synchronized by channelling constraints, in order to increase constraint propagation [12,20,29].

Our research follows the latter approach, with the aim of systematize the process of finding useful reformulations by performing a *symbolic reasoning* on the specification. In general, for many properties, symbolic reasoning can be more natural and effective than making such "structural" aspects emerge after instantiation, when the structure of the problem has been hidden.

An example of system that performs a sort of reasoning on the specification is OPL, which is able to automatically choose the most appropriate solver for a problem. However, the kind of reasoning offered is very primitive: OPL only checks (syntactically) whether a specification is linear, in this case invoking a linear—typically more efficient—solver, otherwise a general constraint programming one.

Conversely, our research aims to the following long-term goal: the *automated reformulation* of a declarative constraint problem specification, into a form that is more efficiently evaluable by the solver at hand. The ultimate goal is to handle all properties suitable for optimization that derive from the problem structure at the specification level, leaving at the subsequent instance level the handling of the remaining ones, i.e., those that truly depend on the instance. In fact, it is worthwhile to note that focusing on the specification does not rule out the possibility of additionally applying existing optimization techniques at the instance level.

The approach we follow is similar, in a sense, to the one used in the database research community for attacking the query optimization problem in relational databases. A query planner, whose task is to reformulate the query posed by the user in order to improve the efficiency of the evaluation, takes into account the query and the database schema only, not its current content, i.e., the instance (cf., e.g., [1]).

In general, reformulating a constraint problem specification is a difficult task: a specification is essentially a formula in second-order logic, and it is well known that the equivalence problem is undecidable already in the first-order case [3]. For this reason, research must focus on controlled and restricted forms of reformulation.

Moreover, the effectiveness of a particular reformulation technique is expected to depend both on the problem and on the solver, even if it is possible, in principle, to find reformulations that are good for all solvers (or for solvers of a certain class, e.g., linear, or SAT-based ones). To this end, in related work (cf. Section 6), we present different reformulation strategies that have been proposed in order to speed-up the process of solving a constraint problem.

[2] Cf. http://www.sbsi-sol-optimize.com/.

[3] Cf. http://www.cse.clrc.ac.uk/nag/lancelot/lancelot.shtml.

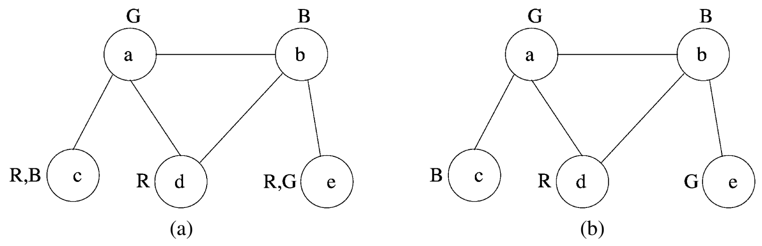


Fig. 2. Delaying the disjointness constraint in 3-coloring. (a) 1st stage: covering and good coloring; (b) 2nd stage: disjointness.

In this paper, we propose a technique that allows us to select constraints in a problem specification that can be ignored in a first step (regardless of the instance), and efficiently reinforced once a solution of the simplified problem has been found. We call such constraints *safe-delay*. Moreover, we experimentally show how reformulating problem specifications by safe-delay improves performances of different (but not all) solvers. On one hand, this gives evidence that problem reformulation can be effective in many cases, and on the other, it confirms the intuition that a single reformulation technique may have positive effects for some classes of solvers, but negative ones for others, and that a portfolio of different and complementary reformulation strategies has to be considered, in general (cf. Section 6 for related work).

The NP-complete graph k-coloring problem offers a simple example of a safe-delay constraint. The problem amounts to find an assignment of nodes to k colors such that:

- Each node has at least one color (*covering*);
- Each node has at most one color (*disjointness*);
- Adjacent nodes have different colors (*good coloring*).

For each instance of the problem, if we obtain a solution neglecting the disjointness constraint, we can always choose for each node one of its colors in an arbitrary way at a later stage (cf. Fig. 2). It is interesting to note that the deletion of the disjointness constraints in graph k-coloring has been already proposed as an ad-hoc technique in [46] (cf. also [42]), and implemented in, e.g., the standard DIMACS formulation in SAT of k-coloring.

Of course not all constraints are safe-delay: as an example, both the covering and the good coloring constraints are not. Intuitively, identifying the set of constraints of a specification which are safe-delay may lead to several advantages:

- The instantiation phase (cf. Fig. 1) will typically be faster, since safe-delay constraints are not taken into account. As an example, let's assume we want to use (after instantiation) a SAT solver for the solution of k-coloring on a graph with n nodes and e edges. The SAT instance encoding the k-coloring instance—in the obvious way, cf., e.g., [25]—has $n \cdot k$ propositional variables, and a number of clauses which is n, $n \cdot k \cdot (k-1)/2$, and $e \cdot k$ for covering, disjointness, and good coloring, respectively. If we delay disjointness, $n \cdot k \cdot (k-1)/2$ clauses need not to be generated.
- Solving the simplified problem, i.e., the one without disjointness, might be easier than the original formulation for some classes of solvers, since removing constraints makes the set of solutions larger. For each instance it holds that:

  $$\{\text{solutions of original problem}\} \subseteq \{\text{solutions of simplified problem}\}.$$

  In our experiments, using six different solvers, including SAT, integer linear programming, and constraint programming ones, we obtained fairly consistent (in some cases, more than one order of magnitude) speed-ups for hard instances of various problems, e.g., graph coloring and job-shop scheduling. On top of that, we implicitly obtain several good solutions. Results of the experimentation are given in Section 5.
- Ad hoc efficient methods for solving delayed constraints may exist. As an example, for k-coloring, the problem of choosing only one color for the nodes with more than one color is $\mathrm{O}(n)$.

The architecture we propose is illustrated in Fig. 3 and can be applied to any system which separates the instance from the specification. It is in some sense similar to the well-known *divide and conquer* technique, cf., e.g., [14], but

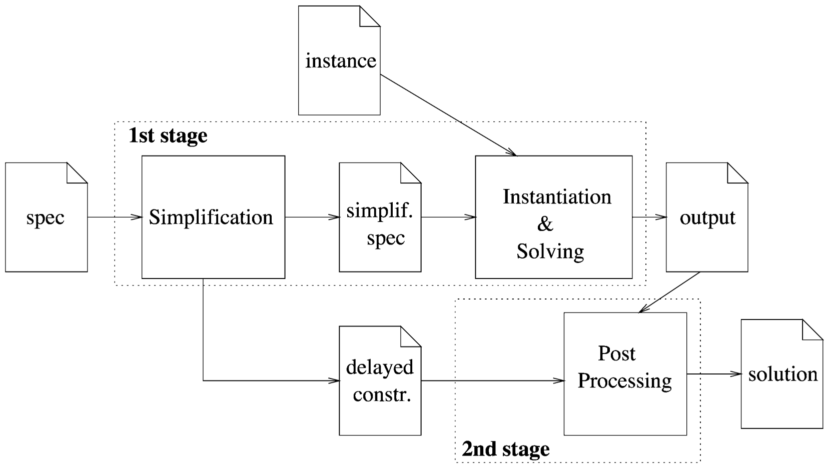


Fig. 3. Reformulation architecture.

rather than dividing the instance, we divide the constraints. In general, the first stage will be more computationally expensive than the second one, which, in our proposal, will always be doable in polynomial time.

The goal of this paper is to understand in which cases a constraint is safe-delay. Our main contribution is the characterization of safe-delay constraints with respect to a semantic criterion on the specification. This allows us to obtain a mechanism for the automated reformulation of a specification that can be applied to a great variety of problems, including the so-called *functional* ones, i.e., those in which the search space is a (total) function from a finite domain to a finite codomain.

The outline of the paper is as follows: after recalling some preliminaries in Section 2, we present our reformulation technique in Section 3, and a discussion on the adopted methodology in Section 4. Afterwards, experimentation on the effectiveness of the approach is described in Section 5, on both benchmark and randomly generated instances, using SAT and state-of-the-art linear and constraint programming solvers. Finally, conclusions, future and related work are presented in Section 6.

2. Preliminaries

The style used for the specification of a combinatorial problem varies a lot among different languages for constraint programming. In this paper, rather than considering procedural encodings such as those obtained using libraries (in, e.g., C++ or PROLOG), we focus on highly declarative languages. Again, the syntax varies a lot among such languages: AMPL, OPL, XPRESS^{MP} and GAMS allow the representation of constraints by using algebraic expressions, while DLV, SMODELS, and NP-SPEC are rule-based languages. Anyway, from an abstract point of view, all such languages are extensions of *existential second-order logic* (ESO) over finite databases, where the existential second-order quantifiers and the first-order formula represent, respectively, the *guess* and *check* phases of the constraint modelling paradigm. In particular, in all such languages it is possible to embed ESO queries, and the other way around is also possible, as long as only finite domains are considered.

To this end, in this paper we use ESO for the specification of problems, mainly because of its simplicity and because it allows to represent all search problems in the complexity class NP [19,40]. In particular, as we show in the remainder of this section, ESO can be considered as the formal basis for virtually all available languages for constraint modelling. Intuitively, the relationship between ESO and real modelling languages is similar to that holding between Turing machines or assembler, and high-level programming languages. We claim that studying the simplified scenario is a mandatory starting point for more complex investigations, and that our results can serve as a basis for reformulating specifications written in higher-level languages. In Section 4 we discuss further our choice, showing how our reformulation technique can be easily lifted in order to deal with problem specifications written in other and richer languages, e.g., AMPL.

Coherently with all state-of-the-art systems, we represent an instance of a problem by means of a *relational database*. All constants appearing in a database are *uninterpreted*, i.e., they don't have a specific meaning.

An ESO specification describing a search problem π is a formula

$$\psi_\pi \doteq \exists \vec{S}\, \phi(\vec{S}, \vec{R}), \tag{1}$$

where $\vec{R} = \{R_1, \ldots, R_k\}$ is the input relational schema (i.e., a fixed set of relations of given arities denoting the schema for all input instances for π), and ϕ is a closed first-order formula on the relational vocabulary $\vec{S} \cup \vec{R} \cup \{=\}$ ("=" is always interpreted as identity), with no function symbols.

An instance $\mathcal{I}$ of the problem is given as a relational database over the schema $\vec{R}$, i.e., as an extension for all relations in $\vec{R}$. Predicates (of given arities) in the set $\vec{S} = \{S_1, \ldots, S_n\}$ are called *guessed*, and their possible extensions (with tuples on the domain given by constants occurring in $\mathcal{I}$ plus those occurring in ϕ, i.e., the so called Herbrand universe) encode points in the search space for problem π on instance $\mathcal{I}$.

Formula ψ_π correctly encodes problem π if, for every input instance $\mathcal{I}$, a bijective mapping exists between solutions to $\pi(\mathcal{I})$ and extensions of predicates in $\vec{S}$ which verify $\phi(\vec{S}, \mathcal{I})$. More formally, the following must hold:

$$\text{For each instance } \mathcal{I}\text{:} \quad \Sigma \text{ is a solution to } \pi(\mathcal{I}) \quad \Longleftrightarrow \quad \{\Sigma, \mathcal{I}\} \models \phi.$$

It is worthwhile to note that, when a specification is instantiated against an input database, a constraint satisfaction problem (in the sense of [17]) is obtained.

Example 1. (*Graph* 3-*Coloring* [26, Prob. GT4]) In this NP-complete decision problem the input is a graph, and the question is whether it is possible to give each of its nodes one out of three colors (red, green, and blue), in such a way that adjacent nodes (not including self-loops) are never colored the same way. The question can be easily specified as an ESO formula ψ over a binary relation *edge*:

$$\exists RGB \quad \forall X \quad R(X) \vee G(X) \vee B(X) \ \wedge \tag{2}$$

$$\forall X \quad R(X) \rightarrow \neg G(X) \ \wedge \tag{3}$$

$$\forall X \quad R(X) \rightarrow \neg B(X) \ \wedge \tag{4}$$

$$\forall X \quad B(X) \rightarrow \neg G(X) \ \wedge \tag{5}$$

$$\forall XY \ X \neq Y \wedge R(X) \wedge R(Y) \rightarrow \neg edge(X, Y) \wedge \tag{6}$$

$$\forall XY \ X \neq Y \wedge G(X) \wedge G(Y) \rightarrow \neg edge(X, Y) \wedge \tag{7}$$

$$\forall XY \ X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X, Y), \tag{8}$$

where clauses (2), (3–5), and (6–8) represent the covering, disjointness, and good coloring constraints, respectively. Referring to the graph in Fig. 2, the Herbrand universe is the set $\{a, b, c, d, e\}$, the input database has only one relation, i.e., *edge*, which has five tuples (one for each edge).

In what follows, the set of tuples from the Herbrand universe taken by guessed predicates will be called their *extension* and denoted with *ext*(). By referring to the previous example, formula ψ is satisfied, e.g., for $ext(R) = \{d\}$, $ext(G) = \{a, e\}$, $ext(B) = \{b, c\}$ (cf. Fig. 2(b)). The symbol *ext*() will be used also for any first-order formula with one free variable. An interpretation will be sometimes denoted as the aggregate of several extensions.

Finally, we observe that in this paper we consider basic ESO. Nonetheless, it is known (cf., e.g., [35]) that much syntactic sugar can be added to ESO in order to handle types, functions, bounded integers and arithmetics, without altering its expressing power. In Section 3.3 we give some examples of the enriched language.

3. Reformulation

In this section we show sufficient conditions for constraints of a specification to be safe-delay. We refer to the architecture of Fig. 3, with some general assumptions:

Assumption 1. As shown in Fig. 2, the output of the first stage of computation may—implicitly—contain *several* solutions. As an example, node c can be assigned to either green or blue, and node e to either red or green. In the second stage we do not want to compute all of them, but just to arbitrarily select one. In other words, we focus on search problems, with no objective function to be optimized.

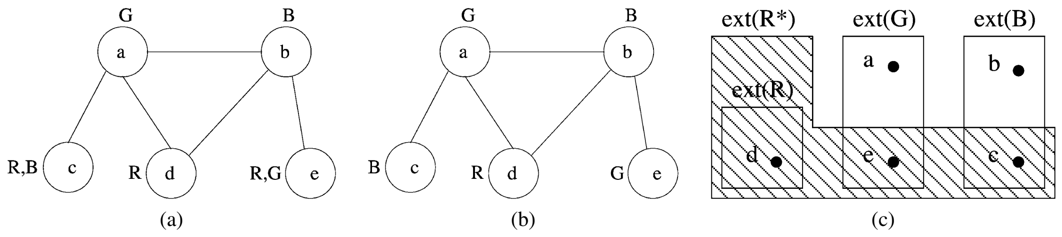


Fig. 4. (a) A solution of the first (a) and second (b) stage of a graph 3-coloring instance. (c) An alternative view showing how the extension for R shrinks when moving from the first (dashed area) to the second stage.

Assumption 2. The second stage of computation can only *shrink* the extension of a guessed predicate. Fig. 4 represents a solution of the first (a) and second (b) stage on the graph 3-coloring instance in Fig. 2. Fig. 4(c) gives further evidence about how the extension for predicate R shrinks when moving from the first ($ext(R^*)$) to the second stage ($ext(R)$) ($ext(B)$ and $ext(G)$ are unchanged).

This assumption is coherent with the way most algorithms for constraint satisfaction operate: each variable has an associated *finite domain*, from which values are progressively eliminated, until a satisfying assignment is found. Nonetheless, in Section 3.4 we give examples of problem specifications which are amenable to safe-delay, although with a second stage of different nature.

Identification of safe-delay constraints requires reasoning on the whole specification, taking into account relations between guessed and database predicates. For the sake of simplicity, in Section 3.1 we will initially focus our attention on *a single monadic* guessed predicate, trying to figure out which constraints concerning it can be delayed. Afterwards, in Section 3.2 we extend our results to *sets* of monadic guessed predicates, then, in Section 3.3, to *binary* predicates.

3.1. Single monadic predicate

We refer to the 3-coloring specification of Example 1, focusing on one of the guessed predicates, R, and trying to find an intuitive explanation for the fact that clauses (3–4) can be delayed. We immediately note that clauses in the specification can be partitioned into three subsets: NO_R, NEG_R, and POS_R with—respectively—no, only negative, and only positive occurrences of R.

Neither NO_R nor NEG_R clauses can be violated by shrinking the extension of R. Such constraints will be called *safe-forget* for R, because if we decide to process (and satisfy) them in the first stage, they can be safely ignored in the second one (which, by Assumption 2 above, can only shrink the extension for R). We note that this is just a possibility, and we are not obliged to do that: as an example, clauses (3–4) will *not* be evaluated in the first stage.

Although in general POS_R clauses are not safe-forget—because shrinking the extension of R can violate them—we now show that clause (2) is safe-forget. In fact, if we equivalently rewrite clauses (2) and (3–4), respectively, as follows:

$$\forall X \quad \neg B(X) \wedge \neg G(X) \rightarrow R(X) \tag{2$'$}$$

$$\forall X \quad R(X) \rightarrow \neg B(X) \wedge \neg G(X), \tag{3–4$'$}$$

we note that clause (2)′ sets a lower bound for the extension of R, and clauses (3–4)′ set an upper bound for it; both the lower and the upper bound are $ext(\neg B(X) \wedge \neg G(X))$. If we use—in the first stage—clauses (2, 5–8) for computing $ext(R^*)$ (in place of $ext(R)$), then—in the second stage—we can safely define $ext(R)$ as $ext(R^*) \cap ext(\neg B(X) \wedge \neg G(X))$, and no constraint will be violated (cf. Fig. 4). The next theorem (all proofs are delayed to Appendix A) shows that is not by chance that the antecedent of (2)′ is semantically related to the consequence of (3–4)′.

Theorem 1. *Let ψ be an ESO formula of the form:*

$$\exists S_1, \ldots, S_h, S \ \Xi \wedge \forall X \ \alpha(X) \rightarrow S(X) \wedge \forall X \ S(X) \rightarrow \beta(X),$$

in which S is one of the (all monadic) guessed predicates, Ξ is a conjunction of clauses, both α and β are arbitrary formulae in which S does not occur and X is the only free variable, and such that the following hypotheses hold:

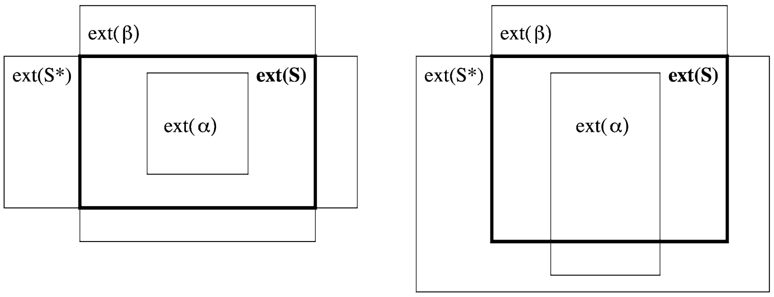


Fig. 5. Extensions with and without Hyp 2.

Hyp 1: *S either does not occur or occurs negatively in Ξ*;
Hyp 2: $\models \forall X\ \alpha(X) \rightarrow \beta(X)$.

Let now ψ^s be:

$$\exists S_1, \ldots, S_h, S^* \quad \Xi^* \wedge \forall X\ \alpha(X) \rightarrow S^*(X),$$

where S^* is a new monadic predicate symbol, and Ξ^* is Ξ with all occurrences of S replaced by S^*, and let ψ^d be:

$$\forall X\ S(X) \leftrightarrow S^*(X) \wedge \beta(X).$$

For every input instance $\mathcal{I}$, and every extension M^* for predicates $(S_1, \ldots, S_h, S^*)$ such that $(M^*, \mathcal{I}) \models \psi^s$, it holds that:

$$\big(\big(M^* - ext(S^*)\big) \cup ext(S), \mathcal{I}\big) \models \psi$$

where $ext(S)$ is the extension of S as defined by M^* and ψ^d.

A comment on the relevance of the above theorem is in order. Referring to Fig. 3, ψ is the specification, $\mathcal{I}$ is the instance, ψ^s is the "simplified specification", and $\forall X\ S(X) \rightarrow \beta(X)$ is the "delayed constraint" (belonging to NEG_S). Solving ψ^s against $\mathcal{I}$ produces—if the instance is satisfiable—a list of extensions M^* (the "output"). Evaluating ψ^d against M^* corresponds to the "PostProcessing" phase in the second stage. The structure of the delayed constraint ψ^d clearly reflects Assumption 2 above, i.e., that extensions for guessed predicates can only be shrunk in the second stage. Moreover, since the last stage amounts to the evaluation of a first-order formula against a fixed database, it can be done in logarithmic space (cf., e.g., [1]), thus in polynomial time.

In other words, Theorem 1 says that, for each satisfiable instance $\mathcal{I}$ of the simplified specification ψ^s, each solution M^* of ψ^s can be translated, via ψ^d, to a solution of the original specification ψ; we can also say that $\Xi \wedge \forall X\ \alpha(X) \rightarrow S(X)$ is safe-forget, and $\forall X\ S(X) \rightarrow \beta(X)$ is safe-delay.

Referring to the specification of Example 1, the distinguished guessed predicate is R, Ξ is the conjunction of clauses (5–8), and $\alpha(X)$ and $\beta(X)$ are both $\neg B(X) \wedge \neg G(X)$, cf. clauses (3–4)$'$. Fig. 4 represents possible extensions of the red predicate in the first (R^*) and second (R) stages, for the instance of Fig. 2, and Fig. 5 (left) gives further evidence that, if Hyp 2 holds, the constraint $\forall X\ \alpha(X) \rightarrow S(X)$ can never be violated in the second stage.

We are guaranteed that the two-stage process preserves at least one solution of ψ by the following theorem.

Theorem 2. *Let $\mathcal{I}$, ψ, ψ^s and ψ^d as in Theorem* 1. *For every instance $\mathcal{I}$, if ψ is satisfiable, ψ^s and ψ^d are satisfiable.*

To substantiate the reasonableness of the two hypotheses of Theorem 1, we play the devil's advocate and consider the following example.

Example 2. (*Graph* 3*-Coloring with red self-loops* (*Example* 1 *continued*)) In this problem, which is a variation of the one in Example 1, the input is the same as for graph 3-Coloring, and the question is whether it is possible to find a coloring of the graph with the additional constraint that all self-loops insist on red nodes.

A specification for this problem can be easily derived from that of Example 1 by adding the following constraint:

$$\forall X\ edge(X, X) \rightarrow R(X). \tag{9}$$

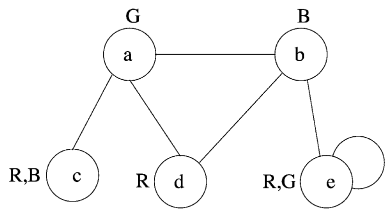


Fig. 6. An instance of the graph 3-coloring with red self-loops problem along with a solution of the first stage, obtained by delaying only the disjointness constraints.

We immediately notice that now clauses (3–4) are not safe-delay: intuitively, after the first stage, nodes may be red either because of (2) or because of (9), and (3–4) are not enough to set the correct color for a node. Now, if—on top of (5–8)—Ξ contains also the constraint (9), Hyp 1 is clearly not satisfied. Analogously, if (9) is used to build $\alpha(X)$, then $\alpha(X)$ becomes $edge(X, X) \vee (\neg B(X) \wedge \neg G(X))$, and Hyp 2 is not satisfied. Fig. 5, right, gives further evidence that the constraint $\forall X\ \alpha(X) \rightarrow S(X)$ can be violated if $ext(S)$ is computed using ψ^d and $ext(\alpha)$ is not a subset of $ext(\beta)$. An instance of the graph 3-coloring with red self-loops problem is given in Fig. 6, along with a solution of the first stage, in the case Ξ contains also the constraint (9). It can be observed that constraints (3–4) are not enough to set the correct color for node e.

Summing up, a constraint with a positive occurrence of the distinguished guessed predicate S can be safely forgotten (after being evaluated in the first stage) only if there is a safe-delay constraint which justifies it.

Some further comments about Theorem 1 are in order. As it can be observed (cf. Appendix A), the theorem proof does not formally require Ξ to be a conjunction of clauses; actually, it can be any formula such that, from any structure M such that $M \models \Xi$, by shrinking $ext(S)$ and keeping everything else fixed we obtain another model of Ξ. As an example, Ξ may contain the conjunct $\exists X\ S(X) \rightarrow \gamma(X)$ (with $\gamma(X)$ a first-order formula in which S does not occur). Secondly, although Hyp 2 calls for a tautology check—which is not decidable in general—we will see in what follows that many specifications satisfy it *by design*.

3.2. Set of monadic predicates

Theorem 1 states that we can delay some constraints of a specification ψ, by focusing on one of its monadic guessed predicates, hence obtaining a new specification ψ^s, and a set of delayed constraints ψ^d. Of course, the same theorem can be further applied to the specification ψ^s, by focusing on a different guessed predicate, in order to obtain a new simplified specification $(\psi^s)^s$ and new delayed constraints $(\psi^s)^d$. Since, by Theorem 2, satisfiability of such formulae is preserved, it is afterwards possible to translate, via $(\psi^s)^d$, each solution of $(\psi^s)^s$ to a solution of ψ^s, and then, via ψ^d, to a solution of ψ.

The procedure REFORMULATE in Fig. 7 deals with the general case of a set of guessed predicates: if the input specification ψ is satisfiable, it returns a simplified specification $\overline{\psi^s}$ and a list of delayed constraints $\overline{\psi^d}$. Algorithm SOLVEBYDELAYING gets any solution of $\overline{\psi^s}$ and translates it, via the evaluation of formulae in the list $\overline{\psi^d}$—with LIFO policy—to a solution of ψ.

As an example, by evaluating the procedure REFORMULATE on the specification of Example 1, by focusing on the guessed predicates in the order R, G, B, we obtain as output the following simplified specification $\overline{\psi^s}$, that omits all disjointness constraints (i.e., clauses (3–5)):

$$\begin{aligned}
\exists R^* G^* B \quad & \forall X \quad R^*(X) \vee G^*(X) \vee B(X) \wedge \\
& \forall XY\ X \neq Y \wedge R^*(X) \wedge R^*(Y) \rightarrow \neg edge(X, Y) \ \wedge \\
& \forall XY\ X \neq Y \wedge G^*(X) \wedge G^*(Y) \rightarrow \neg edge(X, Y) \ \wedge \\
& \forall XY\ X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X, Y),
\end{aligned}$$

and the following list $\overline{\psi^d}$ of delayed constraints:

$$\forall X\ R(X) \leftrightarrow R^*(X) \wedge \neg G(X) \wedge \neg B(X); \tag{10}$$

$$\forall X\ G(X) \leftrightarrow G^*(X) \wedge \neg B(X). \tag{11}$$

```
Algorithm SOLVEBYDELAYING
Input:  a specification $\Phi$, a database $D$;
Output: a solution of $\langle D, \Phi\rangle$, if satisfiable, ‘unsatisfiable’ otherwise;

begin
    $\langle \overline{\Phi^s}, \overline{\Phi^d}\rangle$ = REFORMULATE($\Phi$);
    if ($\langle \overline{\Phi^s}, D\rangle$ is satisfiable) then
      begin
          let $M$ be a solution of $\langle \overline{\Phi^s}, D\rangle$;
          while ($\overline{\Phi^d}$ is not empty) do
          begin
              Constraint $d = \overline{\Phi^d}$.pop();
              $M = M \cup$ solution of $d$; // cf. Theorem 1
          end;
          return $M$;
      end;
    else return ‘unsatisfiable’;
end;

Procedure REFORMULATE
Input:  a specification $\Phi$;
Output: the pair $\langle \overline{\Phi^s}, \overline{\Phi^d}\rangle$, where $\overline{\Phi^s}$ is a simplified specification, and $\overline{\Phi^d}$
        a stack of delayed constraints;
begin
    Stack $\overline{\Phi^d}$ = the empty stack;
    $\overline{\Phi^s} = \Phi$;
    for each monadic guessed pred. $S$ in $\overline{\Phi^s}$ do
      begin
          partition constraints in $\overline{\Phi^s}$ according to Thm 1, in:
            $\langle \Xi;\ \ \forall X\ \alpha(X) \to S(X);\ \ \forall X\ S(X) \to \beta(X)\rangle$;
          if the previous step is possible with $\forall X\ \beta(X) \neq$ TRUE then
            begin
                $\overline{\Phi^d}$.push(‘$\forall X\ S(X) \leftrightarrow S^*(X) \wedge \beta(X)$’);
                $\overline{\Phi^s} = \Xi^* \wedge \forall X\ \alpha(X) \to S^*(X)$;
            end;
      end;
    return $\langle \overline{\Phi^s}, \overline{\Phi^d}\rangle$;
end;
```

Fig. 7. Algorithm for safe-delay in case of a set of monadic predicates.

It is worth noting that the check that $\forall X\ \beta(X)$ is not a tautology prevents the (useless) delayed constraint $\forall X\ B(X) \leftrightarrow B^*(X)$ to be pushed in $\overline{\psi^d}$.

From any solution of $\overline{\psi^s}$, a solution of ψ is obtained by reconstructing first of all the extension for G by formula (11), and then the extension for R by formula (10) (synthesized, respectively, in the second and first iteration of the algorithm). Since each delayed constraint is first-order, the whole second stage is doable in logarithmic space (thus in polynomial time) in the size of the instance.

We also observe that the procedure REFORMULATE is intrinsically non-deterministic, because of the partition that must be applied to the constraints.

3.3. Binary predicates

In this subsection we show how our reformulation technique can be extended in order to deal with specifications with binary (and, in general, n-ary) guessed predicates. This can be formally done by unfolding non-monadic guessed predicates into monadic ones, exploiting the finiteness of the Herbrand domain.

To illustrate the point, we consider the specification of the k-coloring problem using a binary predicate *Col*—the first argument being the node and the second the color, which is as follows (the input schema in this case is given by

$\vec{R} = \{node(\cdot), color(\cdot), edge(\cdot,\cdot)\}$ encoding the set of nodes, colors, and the graph edges, respectively, and constraints force *Col* to be correctly typed, and to satisfy conditions for covering, disjointness, and good coloring):

$$\begin{aligned}
\exists Col\ & \forall XY\ Col(X,Y) \rightarrow node(X) \wedge color(Y) \wedge \\
& \forall X\ \exists Y node(X) \rightarrow Col(X,Y) \wedge \\
& \forall XYZ\ Col(X,Y) \wedge Col(X,Z) \rightarrow Y = Z \wedge \\
& \forall XYZ\ X \neq Y \wedge Col(X,Z) \wedge Col(Y,Z) \rightarrow \neg edge(X,Y).
\end{aligned}$$

Since the number of colors is finite, it is always possible to unfold the above constraints with respect to the second argument of *Col*. As an example, if $k = 3$, we replace the binary predicate *Col* with three monadic guessed predicates Col_1, Col_2, Col_3, one for each value of the second argument (i.e., the color), with the meaning that, if tuple $\langle n, c\rangle$ belongs to *Col*, then $\langle n\rangle$ belongs to Col_c. Constraints of the specification must be unfolded accordingly. The output of the unfolding process for $k = 3$—up to an appropriate renaming of Col_1, Col_2, Col_3 into R, G, B—-is exactly the specification of Example 1.

The above considerations imply that we can use the architecture of Fig. 3 for a large class of specifications, including the so called *functional specifications*, i.e., those in which the search space is a (total) function from a finite domain to a finite codomain. A safe-delay functional specification is an ESO formula of the form

$$\exists P\ \Xi \wedge \forall X\ \exists Y\ P(X,Y) \wedge \forall XYZ\ P(X,Y) \wedge P(X,Z) \rightarrow Y = Z,$$

where Ξ is a conjunction of clauses in which P either does not occur or occurs negatively. In particular, the disjointness constraints are safe-delay, while the covering and the remaining ones, i.e., Ξ, are safe-forget. Formally, soundness of the architecture on safe-delay functional formulae is guaranteed by Theorem 1.

Safe-delay functional specifications are quite common; apart from graph coloring, notable examples are Job-shop scheduling and Bin packing, that we consider next.

Example 3. (*Job-shop scheduling* [26, Prob. SS18]) In the Job-shop scheduling problem we have sets (sorts) J for jobs, K for tasks, and P for processors. Jobs are ordered collections of tasks and each task has an integer-valued length (encoded in binary relation L) and the processor that is requested in order to perform it (in binary relation *Proc*). Each processor can perform a task at the time, and tasks belonging to the same job must be performed in their order. Finally, there is a global deadline D that has to be met by all jobs.

An ESO specification for this problem is as follows. For simplicity, we assume that relation *Aft* contains all pairs of tasks $\langle k', k''\rangle$ of the same job such that k' comes after k'' in the given order (i.e., it encodes the transitive closure), and that relation *Time* encodes all time points until deadline D (thus it contains exactly D tuples). Moreover, we assume that predicate "$\geqslant$" and function "$+$" are correctly defined on constants in *Time*. It is worth noting that these assumptions do not add any expressive power to the ESO formalism, and can be encoded in ESO with standard techniques.

$$\begin{aligned}
\exists S\ \ & \forall k,t\ S(k,t) \rightarrow K(k) \wedge T(t)\ \wedge && (12)\\
& \forall k\ \exists t\ S(k,t)\ \wedge && (13)\\
& \forall k,t',t''\ S(k,t') \wedge S(k,t'') \rightarrow t' = t''\ \wedge && (14)\\
& \forall k',k'',j,t',t'',l'\ Job(k',j) \wedge Job(k'',j) \wedge && \\
& \quad k' \neq k'' \wedge Aft(k'',k') \wedge S(k',t') \wedge S(k'',t'') \wedge && (15)\\
& \quad L(k',l')\ \rightarrow\ t'' \geqslant t' + l'\ \wedge && \\
& \forall k',k'',p,t',t'',l',l'' && \\
& \quad Proc(k',p) \wedge Proc(k'',p) \wedge k' \neq k'' \wedge L(k',l') \wedge && \\
& \quad L(k'',l'') \wedge S(k',t') \wedge S(k'',t'') \rightarrow && (16)\\
& \quad \big[(t' \geqslant t'' \rightarrow t' \geqslant t'' + l'') \wedge (t' \leqslant t'' \rightarrow t'' \geqslant t' + l')\big] \wedge && \\
& \forall k,t,l\ \ T(k) \wedge S(k,t) \wedge L(k,l) \rightarrow Time(t+l). && (17)
\end{aligned}$$

Constraints (12–14) force a solution to contain a tuple $\langle k, t\rangle$ (t being a time point) for every task k, hence to encode an assignment of exactly a starting time to every task (in particular, (14) assigns at most one starting time to each task). Moreover, constraint (15) forces tasks that belong to the same job to be executed in their order without overlapping, while (16) avoids a processor to perform more than one task at each time point. Finally, (17) forces the scheduling to terminate before deadline D.

It is worth noting that additional syntactic sugar may be added to ESO in order to better deal with scheduling problems. As an example, constructs like those commonly found in richer modelling languages for such problems (cf., e.g., the part of OPL concerning scheduling) can be made available. However, such enhancements are out of the scope of this paper, and will not be taken into account. □

As an example, Fig. 8(a) and (b) show, respectively, an instance and a possible solution of the Job-shop scheduling problem. The instance consists of 3 jobs (J1, J2, and J3) of, respectively, 4, 3, and 5 tasks each. The order in which tasks belonging to the same job have to be performed is given by the letter in parentheses (a, b, c, d, e). Tasks have to be executed on 3 processors, P1, P2, and P3, which are denoted by different borderlines. Hence, the processor needed to perform a given task is given by the task borderline.

To reformulate the Job-shop scheduling problem, after unfolding the specification in such a way to have one monadic guessed predicate S_t for each time point t, we focus on a time point $\bar{t}$ and partition clauses in the specification in which $S_{\bar{t}}$ does not occur, occurs positively, or negatively, in order to build Ξ, $\alpha(k)$, and $\beta(k)$. The output of this phase is as follows:

- $\alpha(k) \doteq \bigwedge_{t\neq\bar{t}} \neg S_t(k)$ (obtained by unfolding (13));
- $\beta(k) \doteq \bigwedge_{t\neq\bar{t}} \neg S_t(k)$ (obtained by unfolding (14)).

α and β above clearly satisfy Hyp 2 of Theorem 1. Moreover, according to the algorithm in Fig. 7, by iteratively focusing on all predicates S_t, we can delay all such (unfolded) constraints. It is worth noting that the unfolding of guessed predicates is needed only to formally characterize the reformulation with respect to Theorem 1, and must not be performed in practice.

Intuitively, the constraint we delay, i.e. (14), forces each task to have at most one starting time: thus, by delaying it, we allow a task to have multiple starting times, i.e., the task does not overlap with any other task at any of its start times. Again, in the second stage, we can arbitrarily choose one of them. We observe that a similar approach has been used in [15] for an optimized ad-hoc translation of this problem into SAT, where propositional variables represent the encoding of *earliest starting times* and *latest ending times* for all tasks, rather than their exact scheduled times. As an example, Fig. 8(c) shows a solution of the first stage of the reformulated problem for the instance in Fig. 8(a), which subsumes the solution in Fig. 8(b).

Example 4. (*Bin packing* [26, Prob. SR1]) In the Bin packing problem (cf. also [36]), we are asked to pack a set I of items, each one having a given size, into a set B of bins, each one having a given capacity. Under the assumption that input instances are given as extensions for relations I, S, B, and C, where I encodes the set of items, B the set of bins, S the size of items (a tuple $\langle i, s\rangle$ for each item i), and C the capacity of bins (a tuple $\langle b, c\rangle$ for each bin b), an ESO specification for this problem is as follows:

$$\exists P \quad \forall i, b\ P(i,b) \rightarrow I(i) \wedge B(b)\ \wedge \tag{18}$$
$$\forall i\ \exists b\ I(i) \rightarrow P(i,b)\ \wedge \tag{19}$$
$$\forall i, b, b'\ P(i,b) \wedge P(i,b') \rightarrow b = b' \tag{20}$$
$$\forall b, c\ C(b,c) \rightarrow sum\big(\{s \mid P(i,b) \wedge S(i,s)\}\big) \leqslant c \tag{21}$$

where, to simplify notations, we assume bounded integers to encode the size of items and capacity of bins, and the existence of a function *sum* that returns the sum of elements that belong to the set given as argument. We remind that bounded integers and arithmetic operations over them do not add expressive power to ESO.

In the above specification, a solution is a total mapping P from items to bins. Constraints force the mapping to be, respectively, over the right relations (18), total (19), mono-valued (20), and satisfying the capacity constraint for every bin (21).

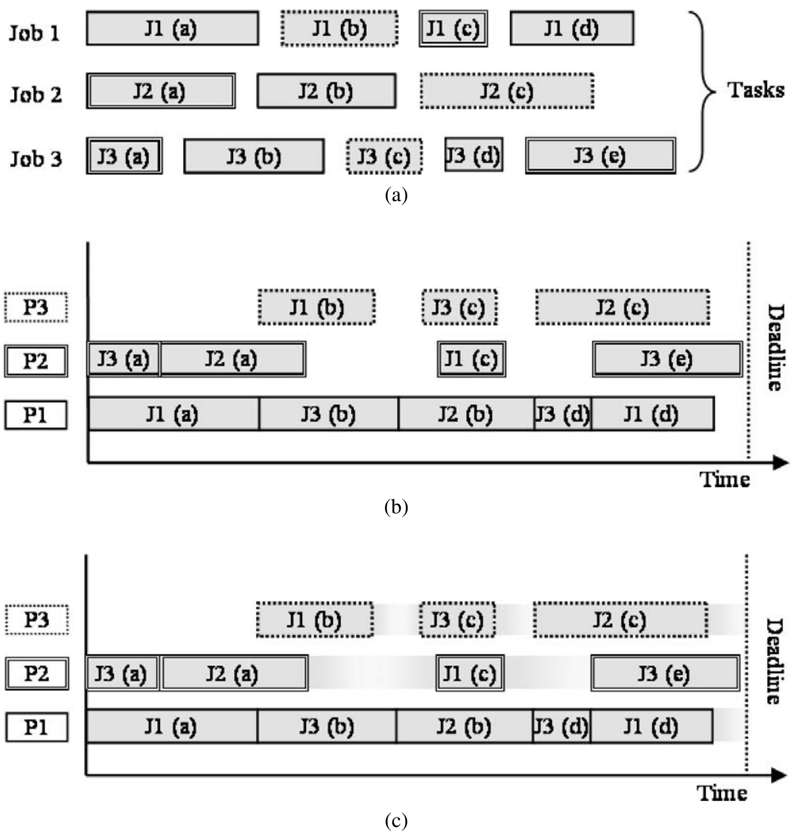


Fig. 8. (a) An instance of the Job-shop scheduling problem, consisting in 3 jobs of, respectively, 4, 3, and 5 tasks each, to be executed on 3 processors. (b) A possible solution of the whole problem for the instance in (a). (c) A solution of the first stage of the reformulated problem, that subsumes that in (b). Shades indicate multiple good starting times for tasks.

In particular, by unfolding the guessed predicate P to $|I|$ monadic predicates P_i, one for every item i, and, coherently, the whole specification, the constraints that can be delayed are the unfolding of (20), that force an item to be packed in exactly one bin. Thus, by iteratively applying Theorem 1 by focusing on all unfolded guessed predicates, we intuitively allow an item to be assigned to *several* bins. In the second stage, we can arbitrarily choose one bin to obtain a solution of the original problem.

Other problems that can be reformulated by safe-delay exist. Some examples are Schur's Lemma (www.csplib.org, Prob. 15) and Ramsey problem (www.csplib.org, Prob. 17). A short discussion on how these reformulations can be addressed is given in [7].

As showed in the previous examples, it is worth noting that arithmetic constraints do not interfere with our reformulation technique. As an instance, in the last example, the "$\leqslant$" predicate leads to clauses that remain satisfied if the extension of the selected guessed predicate is shrunk, while keeping everything else fixed.

3.4. Non-shrink second stages

As specified at the beginning of Section 3, in this paper we have focused on second stages in which the extension of the selected guessed predicate can only be shrunk, while those for the other ones remain fixed.

Actually, there are other specifications which are amenable to be reformulated by safe-delay, although with a different kind of second stages. As an example, we show a specification for the Golomb ruler problem.

Example 5. (*Golomb ruler* (www.csplib.org, Prob. 6)) In this problem, we are asked to put m marks $M_1, \ldots, M_m$ on different points on a ruler of length l in such a way that:

(1) Mark i is put on the left (i.e., before) mark j if and only if $i < j$, and
(2) The $m(m-1)/2$ distances among pairs of distinct marks are all different.

By assuming that input instances are given as extensions for unary relations M (encoding the set of marks) and P (encoding the l points on the ruler), and that the function "$+$" and the predicate "$<$" are correctly defined on tuples in M and on those in P, a specification for this problem is as follows:

$$\exists G \quad \forall m, i\ G(m,i) \to M(m) \wedge P(i)\ \wedge \tag{22}$$

$$\forall m\ \exists i\ M(m) \to P(m,i)\ \wedge \tag{23}$$

$$\forall m, i, i'\ G(m,i) \wedge G(m,i') \to i = i'\ \wedge \tag{24}$$

$$\forall m, m', i, i'\ G(m,i) \wedge G(m',i') \wedge m < m' \to i < i' \wedge \tag{25}$$

$$\begin{aligned} &\forall m, m', i, i', n, n', j, j' \\ &\quad G(m,i) \wedge G(m',i') \wedge G(n,j) \wedge G(n',j') \wedge \\ &\quad m < m' \wedge n < n' \wedge (m < n \vee (m = n \wedge m' < n')) \to \\ &\qquad (i'-i) \neq (j'-j). \end{aligned} \tag{26}$$

A solution is thus an extension for the guessed predicate G which is a mapping (22–24) assigning a point in the ruler to every mark, such that the order of marks is respected (25) and distances between two different marks are all different (26).

Here, the constraint that can be delayed is (25), which forces the ascending ordering among marks. By neglecting it, we extend the set of solutions of the original problem with all their permutations. In the second stage, the correct ordering among marks can be enforced in polynomial time.

By unfolding the binary guessed predicate G, we obtain $|M|$ monadic predicates G_m, one for each mark m. Once a solution of the simplified specification has been computed, by focusing on all of them, in order to reinforce the $m(m-1)/2$ unfolded constraints derived from (25), we possibly have to *exchange* tuples among pairs of predicates G_m and $G_{m'}$, for all $m \neq m'$, and not to shrink the extensions of single guessed predicates. Hence, Theorem 1 does not apply. Furthermore, a modification of some of the other constraints may be needed to ensure the correctness of the reformulation. In particular, in constraint (26) differences must be replaced by their absolute values.

As for the effectiveness of such a reformulation, it can be objected that the constraints delayed actually break the permutation symmetry, and removing them is likely to be not a good choice. However, even if this is likely to be true for CP solvers, like those based on backtracking, it is not obvious for others, e.g., SAT ones. In [7] we show how delaying such constraints significantly drops down the instantiation time needed by the NP-SPEC SAT compiler, without major variations in the solving times of the best SAT solver (ZCHAFF).

Another class of problems that are amenable to be reformulated by safe-delay is an important subclass of permutation problems, that includes, e.g., Hamiltonian path (HP), Permutation flow-shop, and Tiling. Some preliminary results on how these problems can be reformulated appear in [34]. As an example, HP can be reformulated by looking for (small) cliques in the graph, and viewing them as single nodes. If we find an HP of the reduced graph, we can, in polynomial time, obtain a valid solution of the original problem, since cliques can be traversed in any order.

We are currently investigating the formal aspects of such a generalization, and for which class of solvers this kind of reformulations are effective in practice.

4. Methodological discussion

In this section we make a discussion on the methodology we adopted in this work, in particular the use of ESO as a modelling language, and the choice of the solvers for the experimentation.

As already claimed in Section 1, and as the previous examples show, using ESO for specifying problems wipes out many aspects of state-of-the-art languages which are somehow difficult to take into account (e.g., numbers, arithmetics, constructs for functions, etc.), thus simplifying the task of finding criteria for reformulating problem specifications. However, it must be observed that ESO, even if somewhat limited, is not too far away from the modelling languages provided by some commercial systems.

A good example of such a language is AMPL [22], which admits only linear constraints: in this case, the reformulation technique described in Theorem 1 can often be straightforwardly applied; as an instance, a specification of the k-coloring problem in such a language is as follows:

```
param n_nodes;
param n_colors integer, > 0;
set NODES := 1..n_nodes;
set EDGES within NODES cross NODES;
set COLORS := 1..n_colors;

# Coloring of nodes as a 2-ary predicate
var Coloring {NODES,COLORS} binary;

s.t. CoveringAndDisjointness {x in NODES}:
    # nodes have exactly one color
    sum {c in COLORS} Coloring[x,c] = 1;

s.t. GoodColoring {(x,y) in EDGES, c in COLORS}:
    # nodes linked by an edge have diff. colors
    Coloring[x,c] + Coloring[y,c] <= 1;
```

The reformulated specification can be obtained by simply replacing the "CoveringAndDisjointness" constraint with the following one:

```
s.t. Covering {x in NODES}
    sum {c in COLORS} Coloring[x,c] >= 1;
```

thus leading exactly to the reformulated specification of Example 1.

As for languages that admit non-linear constraints, e.g., OPL, it is possible to write a different specification using integer variables for the colors and inequality of colors between adjacent nodes. In this case it is not possible to separate the disjointness constraint from the other ones, since it is implicit in the definition of the domains. Of course, study of safe-delay constraints is relevant also for such languages, because we can always specify in OPL problems such as the one of Example 5, which has such constraints, or we may want to write a linear specification for a given problem, in order to use a linear solver, more efficient in many cases (cf. Section 5).

For what concerns the experimentation, it must be observed that a specification written in ESO naturally leads to a translation into a SAT instance. For this reason, we have chosen to use, among others, SAT solvers for the experimentation of the proposed technique. Moreover we considered also the impressive improving in performances recently shown by state-of-the art SAT solvers.

As already claimed in Section 1, the effectiveness of a reformulation technique is expected to strongly depend on the particular solver used. To this end, we solved the same set of instances with SAT solvers of very different nature (cf. Section 5). Finally, since it is well-known that state-of-the-art linear and constraint programming systems may perform better than SAT on some problems, we repeated the experimentation by using commercial systems CPLEX (linear) and SOLVER (non-linear), invoked by OPLSTUDIO.[4]

5. Experimental results

We made an experimentation of our reformulation technique on 3-coloring (randomly generated instances), k-coloring (benchmark instances from the DIMACS repository[5]), and job-shop scheduling (benchmark instances from OR library[6]), using both SAT-based solvers (and the NP-SPEC SAT compiler [8] for the instantiation stage), and the constraint and linear programming system OPL [48], obviously using it as a pure modelling language, and omitting search procedures.

[4] Cf. http://www.ilog.com.

[5] Cf. ftp://dimacs.rutgers.edu/pub/challenge.

[6] Cf. http://www.ms.ic.ac.uk/info.html.

Table 1
Experimental results for 3-coloring on random instances with 500 nodes (100 instances for each fixed number of edges). Solving times (in seconds, with timeout of 1 hour) are relative to each set of 100 instances

Und. edges	e/n	ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
		$\sum$ No delay	$\sum$ Delay	% sav.	$\sum$ No delay	$\sum$ Delay	% sav.	$\sum$ No delay	$\sum$ Delay	% sav.	$\sum$ No delay	$\sum$ Delay	% sav.
500	2.0	0.26	0.26	0.00	0.92	0.60	34.55	2.43	1.85	23.97	10.17	7.17	29.50
600	2.4	0.13	0.22	−69.23	2.22	1.82	18.05	3.63	3.00	17.43	8.97	5.08	43.37
700	2.8	0.22	0.12	45.45	8.48	7.58	10.61	9.92	8.60	13.28	7210.96	10804.02	−49.83
800	3.2	0.2	0.1	50.00	9.33	8.58	8.04	10.95	9.82	10.35	7208.19	7240.53	−0.45
900	3.6	0.18	0.14	22.22	18.18	16.52	9.17	19.88	17.87	10.14	14408.3	14403.53	0.03
1000	4.0	0.16	0.07	56.25	48.77	42.82	12.20	48.62	42.97	11.62	18787.4	21603.45	−14.99
1100	4.4	11.25	12.07	−7.29	119.70	105.50	11.86	119.47	105.47	11.72	11825.44	8772.81	25.81
1200	4.8	80537.81	86843.41	−7.83	129.35	112.05	13.37	129.75	113.78	12.31	16686.21	12279.19	26.41
1300	5.2	1497.69	1441.24	3.77	134.50	115.80	13.90	138.37	117.58	15.02	700.14	544.35	22.25
1400	5.6	60.4	59.77	1.04	145.32	119.93	17.47	145.53	124.68	14.33	95.09	78.27	17.69
1500	6.0	18.77	24.1	−28.40	147.68	125.25	15.19	152.82	128.97	15.61	31.56	26.95	14.61

As for the SAT-based experimentation, we used four solvers of very different nature: the DPLL-based complete systems ZCHAFF [38] and SATZ [33], and the local-search based incomplete solvers WALKSAT [45] and BG-WALKSAT [51] (the last one being guided by "backbones"). We solved all instances both with and without delaying constraints. As for OPL, we wrote both a linear and a non-linear specification for the above problems, and applied our reformulation technique to the linear one (cf. Section 4). All solvers have been used with their default parameters, without any heuristic or tuning that would possibly alter their performances. This is coherent with the declarative approach we adopted in this paper.

Experiments were executed on an Intel 2.4 GHz Xeon bi-processor computer. The size of instances was chosen so that our machine is able to solve (most of) them in more than a few seconds, and less than one hour. In this way, both instantiation and post-processing, i.e., evaluation of delayed constraints, times are negligible, and comparison can be done only on the solving time.

In what follows, we refer to the *saving percentage*, defined as the ratio:

$$(\mathit{time\_no\_delay} - \mathit{time\_delay})/\mathit{time\_no\_delay}$$

3-*coloring*. We solved the problem on 1500 randomly generated graph instances with 500 nodes each. The number of edges varies, and covers the phase transition region [11]: the ratio (# of directed edges/# of nodes) varies between 2.0 and 6.0. In particular, we considered sets of 100 instances for each fixed number of edges, and solved each set both with and without delaying disjointness constraints (timeout was set at 1 hour). Table 1 shows overall solving times for each set of instances, for all the SAT solvers under consideration.

As it can be observed, the saving percentage depends both on the edges/nodes ratio, and on the solver. However, we have consistent time savings for many classes of instances. In particular, ZCHAFF seems not to be positively affected by safe-delay, and, for some classes of instances, e.g., those with 1500 edges, it seems to be negatively affected (−28.40%). On the other hand, both local-search based SAT solvers, i.e., WALKSAT and BG-WALKSAT show a consistent improvement with our reformulation technique, saving between 13% and 17% for hard instances. This behavior is consistent with the observation that enlarging the set of solutions can be profitable for this kind of solvers, and will be discussed at the end of this section, since it has been observed in all our experiments. Even if these are incomplete solvers, they have been always able to find a solution for satisfiable instances, except for the class of input graphs with 1100 edges: in this case, they found a solution on the 40% (WALKSAT) and 50% (BG-WALKSAT) of positive instances. Delaying disjointness constraints does not alter this percentage in a significant way. Also SATZ benefits from safe-delay, with savings between 22% and 26% for hard instances, even if underconstrained instances (e.g., those with 700 edges) highlight poorer performances (−49.83%).

It is worth noting, from the experiments described above, that the effectiveness of the reformulation may depend also on some instance-dependent parameters like, e.g., the edges/nodes ratio. However, this does exclude that some classes of solvers often benefit from the technique. In particular, it is interesting to note that the best technology, i.e., local search, is always improved.

For what concerns CPLEX instead, experimental results do not highlight significant variations in performances, since, in many cases, for the same set of 1500 instances, either both solving times were negligible, or a timeout occurred both with and without disjointness constraints. Finally, by solving the linear specification with SOLVER we observed the following mixed evidence: (i) About 15% of the instances were not solved, regardless of safe-delay; (ii) Another 15% of the instances were successfully solved with the original specification, but performing safe-delay on them prevented the system from terminate within the time limit; (iii) As for the remaining instances, average savings in time were often appreciable.

k-coloring. We solved the k-coloring problem on several benchmark instances of various classes of the DIMACS repository, with k close to the optimum, in order to have non-trivial instances, both positive and negative.

Results of our SAT-based experiments are shown in Table 2. As it can be observed, also here the effectiveness of the reformulation technique varies among the different solvers. In particular, ZCHAFF benefits by safe-delay on several instances, both positive and negative, but not on all of them. On the other hand, for local search solvers WALKSAT and BG-WALKSAT, delaying disjointness constraints always (except for very few cases) speeds-up the computation (usually by 20–30%). The same happens when using SATZ with even higher savings, even if this solver timeouts for several instances.

As for OPL instead, we have mixed evidence, since it is not the case that the linear specification (solved using CPLEX) is always more efficient than the non-linear one (solved using SOLVER), or vice versa. Indeed, the linear specification, when solved with CPLEX, often, but not always, benefits from safe-delay. Table 3 shows results obtained on graphs of various classes of the benchmark set. Differently from Table 2, in this case, due to the higher number of instances solved, we opted for showing aggregate results, grouping together instances of the same class. In partic-

Table 2
Solving times (seconds) for k-coloring (SAT solvers)

Instance	Colors	Sol-vable?	ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
			No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.
anna	10	N	24.87	15.02	39.61	6.48	5.71	11.83	6.2	4.0	4.57	–	–	–
anna	11	Y	0.01	0.01	0.00	0.08	0.05	40.00	0.1	0.06	33.33	0.29	0.09	68.97
david	10	N	15.15	13.04	13.93	5.58	4.78	14.33	5.52	4.66	15.41	–	–	–
david	11	Y	0.1	0.1	0.00	0.07	0.05	25.00	0.07	0.05	25.00	0.63	0.08	87.30
DSJC125.5	8	N	41.42	4.04	90.25	mem	mem	–	mem	mem	–	–	–	–
DSJC250.5	10	N	–	52.69	>98.54	mem	mem	–	mem	mem	–	–	–	–
DSJC500.1	5	N	11.14	1.55	86.09	mem	mem	–	mem	mem	–	158.67	145.33	8.41
le450_5a	5	Y	17.23	0.91	94.72	5.42	4.91	9.23	5.51	4.97*	9.97	96.83	77.16	20.31
le450_5b	5	Y	27.77	1.36	95.10	6.17*	5.10	17.30	6.15*	5.48	10.84	104.62	93.33	10.79
le450_5c	5	Y	0.02	0.02	0.00	10.00*	8.67	13.33	9.65	9.23*	4.32	22.45	20.02	10.82
le450_5c	9	Y	11.20	1.81	83.84	1.20	1.20	0.00	1.47	1.35	7.95	–	–	–
le450_5d	5	Y	0.09	1.20	−1233.33	8.52	8.15	4.31	8.42	8.40	0.20	6.00	5.20	13.33
miles500	9	N	80.19	54.55	31.97	9.08	8.51	6.24	9.70	7.38	23.92	–	–	–
miles500	20	Y	0.01	0.01	0.00	0.53	0.36	31.25	0.63	0.46	26.32	8.36	3.76	55.02
mulsol.i.2	30	N	–	–	–	28.83	22.85	20.75	30.18	24.16	19.93	–	–	–
mulsol.i.2	31	Y	0.01	0.01	0.00	4.25	2.36	44.31	4.21	2.80	33.60	5.10	2.67	47.65
mulsol.i.3	30	N	–	–	–	29.20	23.03	21.12	31.47	24.38	22.51	–	–	–
mulsol.i.3	31	Y	0.01	0.01	0.00	4.22	2.48	41.11	4.60	3.05	33.70	5.04	2.70	46.43
mulsol.i.4	30	N	–	–	–	29.40	23.58	19.78	30.93	24.81	19.77	–	–	–
mulsol.i.4	31	Y	0.01	0.01	0.00	4.03	2.38	40.91	4.98	2.76	44.48	5.08	2.73	46.26
mulsol.i.5	30	N	–	–	–	26.70	21.38	19.91	28.48	22.63	20.54	–	–	–
mulsol.i.5	31	Y	0.01	0.01	0.00	4.63	2.53	45.32	5.42	2.88	46.77	5.14	2.74	46.69
myciel5	5	N	413.99	1714.26	<−314.08	1.33	1.16	12.50	1.33	1.15	13.75	52.89	39.25	25.79
myciel5	6	Y	0.01	0.01	0.00	0.02	0.02	0.00	0.01	0.01	0.00	0.10	0.04	60.00
queen8_8	9	Y	1397.23	2144.76	−53.50	5.23	3.95	24.52	5.25	4.13	21.27	0.27	0.22	18.52
queen9_9	10	Y	–	–	–	8.22*	7.31*	10.95	8.43*	7.61*	9.68	56.91	45.51	20.03
queen11_11	13	Y	553.46	863.14	−55.95	15.13	9.71	35.79	14.47	11.63	19.59	205.66	175.62	14.61
queen14_14	17	Y	17.75	114.14	−543.04	8.27	11.82	−42.94	7.40	9.08	−22.75	943.30	708.92	24.86
queen8_12	12	Y	0.13	1.78	−1269.23	5.27	4.68	11.08	6.03	5.71	5.25	0.35	0.25	28.57

('–' means that the solver did not terminate in one hour, while 'mem' that an out-of-memory error occurred. A '*' means that the local search solver did not find a solution.)

Table 3
Aggregate results (sum of solving times in seconds) for k-coloring (CPLEX)

Class	Instances		CPLEX			
	Solvable	Number	No delay	Delay	Saving	Saving %
Leighton	Y	3	4776.52	6660.07	−1883.55	−39.43%
(le graphs)	N	6	110.64	40.42	70.22	63.47%
	Total	9	4887.16	6700.49	−1813.33	−37.10%
Random	Y	1	19.22	18.15	1.07	5.57%
(DSJC graphs)	N	4	5026.08	4112.10	913.98	18.18%
	Total	5	5045.30	4130.25	915.05	18.14%
Reg. alloc.	Y	9	12423.85	11179.64	1244.21	10.01%
(fpsol, mulsol, zeroin graphs)	N	1	22.79	14.24	8.55	37.52%
	Total	10	12446.64	11193.88	1252.76	10.07%
SGB book	Y	4	2.16	2.27	−0.11	−5.09%
(anna, david, huck, jean graphs)	N	4	1.91	1.71	0.20	10.47%
	Total	8	4.07	3.98	0.09	2.21%
SGB miles	Y	3	138.24	197.23	−58.99	−42.67%
	N	2	2.98	2.07	0.91	30.54%
	Total	5	141.22	199.30	−58.08	−41.13%
SGB games	Y	1	0.97	0.88	0.09	9.28%
	N	1	0.97	1.05	−0.08	−8.25%
	Total	2	1.94	1.93	0.01	0.52%
SGB queen	Y	11	11806.53	6319.20	5487.33	46.48%
	N	4	8.93	9.33	−0.40	−4.48%
	Total	15	11815.46	6328.53	5486.93	46.44%
Mycielsky	Y	5	7.08	4.97	2.11	29.80%
	N	2	6.08	5.57	0.51	8.39%
	Total	7	13.16	10.54	2.62	19.91%
All	Y	37	29174.57	24382.41	4792.16	16.43%
	N	24	5180.38	4186.49	993.89	19.19%
	Total	61	34354.95	28568.90	5786.05	16.84%

ular, for each class, we write the number of instances solved, and the time needed by CPLEX for both positive and negative ones, with and without safe-delay (instances that couldn't be solved in one hour by both specifications have been removed). It can be observed that the reformulated specification is more efficient, on the average, especially on negative instances. Actually, safe-delay is really deleterious in only two cases: positive instances of "Leighton" and "SGB miles" classes.

Job shop scheduling. We considered 40 benchmark instances known as LA01, ..., LA40, with the number of tasks ranging between 50 and 225, number of jobs between 10 and 15, and number of processors ranging between 5 and 15. However, in order to make our solvers (especially the SAT ones) able to deal with such large instances, we reduced (and rounded) all task lengths and the global deadline by a factor of 20 (original lengths were up to 100). In this way, we obtained instances that are good approximations of the original ones, but with much smaller time horizons, hence fewer propositional variables need to be generated.

SAT solving times are listed in Table 4 for different values for the deadline. Again, we have a mixed evidence for what concerns ZCHAFF, which benefits from safe-delay on many but not all instances, while savings in time are always positive when using local search solvers WALKSAT and BG-WALKSAT (even when they are not able to find a solution for positive instances, delaying constraints makes them terminate earlier). As for SATZ, savings in performances are often very high, even if this solver is able to solve only a small portion of the instance set. Interestingly, the blow-down in the number of clauses due to safe-delay, in some cases makes the solver able to handle some large instances (cf. Table 4, e.g., instance LA06 with deadline 50), preventing the system from running out of memory (even if, in many cases, the out-of-memory error changes to a timeout).

Table 4
Solving times (seconds) for job shop scheduling

Instance	Proc	Tasks	Jobs	Deadline	Solvable?	ZCHAFF			WALKSAT			BG-WALKSAT			SATZ		
						No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.	No delay	Delay	% sav.
la01	5	50	10	33	N	0.69	0.85	−23.19	39.83	31.13	21.84	39.55	31.21	21.07	–	–	–
la01	5	50	10	34	Y	0.13	1.24	−853.85	40.77*	31.42*	22.94	40.72*	31.95*	21.53	154.49	1.76	98.86
la02	5	50	10	32	N	1.75	1.61	8.00	36.63	28.63	21.84	37.95	29.92	21.17	–	–	–
la02	5	50	10	33	Y	2.34	1.07	54.27	37.37*	28.80*	22.93	38.23*	29.10*	23.89	–	–	–
la03	5	50	10	31	N	2.77	2.10	24.19	32.78	25.37	22.62	32.92	25.11	23.70	–	–	–
la03	5	50	10	32	Y	1.92	0.31	83.85	32.85*	25.53*	22.27	32.67*	26.31*	19.44	–	–	–
la04	5	50	10	29	N	1.11	1.06	4.50	33.22	26.10	21.42	33.48	26.06	22.15	904.38	189.44	79.05
la04	5	50	10	30	Y	4.05	2.83	30.12	33.25*	26.22*	21.15	34.47*	26.02*	24.52	699.77	295.12	57.83
la05	5	50	10	28	N	1.33	0.99	25.56	28.92	22.83	21.04	29.25	23.10	21.03	–	–	–
la05	5	50	10	29	Y	0.29	0.57	−96.55	26.42	18.40	30.35	27.73*	19.46	29.81	0.89	0.58	34.83
la06	5	75	15	46	N	–	–	–	78.10	62.92	19.44	79.98	63.00	21.23	mem	–	–
la06	5	75	15	50	Y	0.44	1.89	−329.55	77.32	60.20*	22.14	78.77*	60.37*	23.36	mem	4.87	100.00
la07	5	75	15	43	N	–	–	–	78.35	62.21	20.59	80.10	62.55	21.91	–	–	–
la07	5	75	15	50	Y	1.27	0.98	22.83	78.62*	60.78*	22.68	78.08*	59.40*	23.93	mem	–	–
la08	5	75	15	42	N	–	–	–	77.02	60.72	21.16	76.18	60.17	21.02	–	–	–
la08	5	75	15	43	Y	262.44	181.53	30.83	76.83*	62.02*	19.28	77.25*	61.07*	20.95	–	–	–
la09	5	75	15	46	N	–	–	–	86.58	67.20	22.39	86.57	67.15	22.43	mem	–	–
la09	5	75	15	50	Y	0.31	5.76		85.73*	65.40*	23.72	83.65*	63.30	24.33	mem	3108.10	100.00
la10	5	75	15	46	N	–	–	–	80.80	64.86	19.72	81.42	63.07	22.54	mem	–	–
la10	5	75	15	50	Y	2.23	15.44		80.10*	61.90*	22.72	80.97*	61.05*	24.60	mem	–	–
la16	10	100	10	47	N	155.26	90.71	41.58	69.33	51.73	25.38	67.47	51.11	24.23	mem	–	–
la16	10	100	10	48	Y	151.64	18.71	87.66	69.87*	51.55*	26.22	70.70*	51.53*	27.11	mem	–	–
la17	10	100	10	39	N	5.42	3.72	31.37	57.03	44.46	22.03	56.23	43.78	22.14	–	–	–
la17	10	100	10	40	Y	3.68	4.19	−13.86	57.82*	44.20*	23.55	58.80*	43.71*	25.65	–	–	–
la18	10	100	10	41	N	6.31	4.21	33.28	62.30	47.53	23.70	61.12	45.48	25.58	–	–	–
la18	10	100	10	42	Y	5.75	3.27	43.13	63.72*	47.50*	25.45	62.17*	46.85*	24.64	–	–	–
la19	10	100	10	42	N	50.04	33.09	33.87	63.28	47.17	25.47	63.97	47.4	25.90	–	–	–
la19	10	100	10	43	Y	120.68	154.67	−28.17	64.83*	48.53*	25.14	63.52*	47.58*	25.09	–	3524.02	>2.11
la20	10	100	10	44	N	6.04	5.33	11.75	66.52	50.68	23.80	66.22	49.53	25.20	mem	–	–
la20	10	100	10	45	Y	20.86	13.91	33.32	68.87*	50.60*	26.52	67.47*	49.40*	26.78	mem	–	–
la22	10	150	15	48	N	–	–	–	mem	mem	–	mem	mem	–	mem	mem	–
la22	10	150	15	50	Y	1173.04	619.5	47.19	mem	mem	–	mem	mem	–	mem	mem	–
la23	10	150	15	50	N	–	934.79	>48.07	mem	mem	–	mem	mem	–	mem	mem	–
la23	10	150	15	53	Y	–	1028.56	42.86	mem	mem	–	mem	mem	–	mem	mem	–
la24	10	150	15	46	N	138.15	123.37	10.70	mem	mem	–	mem	mem	–	mem	mem	–
la24	10	150	15	48	Y	–	1083.31	>39.82	mem	mem	–	mem	mem	–	mem	mem	–
la25	10	150	15	48	N	356.56	344.02	3.52	mem	mem	–	mem	mem	–	mem	mem	–
la25	10	150	15	50	Y	438.27	510.21	−16.41	mem	mem	–	mem	mem	–	mem	mem	–
la29	10	200	20	50	N	705.09	216.74	69.26	mem	mem	–	mem	mem	–	mem	mem	–
la29	10	200	20	75	Y	–	1387.73	>22.90	mem	mem	–	mem	mem	–	mem	mem	–
la36	15	225	15	62	N	1238.04	782	36.84	mem	mem	–	mem	mem	–	mem	mem	–
la36	15	225	15	65	Y	–	527.86	>70.67	mem	mem	–	mem	mem	–	mem	mem	–
la38	15	225	15	58	N	–	844.58	>53.08	mem	mem	–	mem	mem	–	mem	mem	–
la38	15	225	15	75	Y	1329.74	–	<−35.36	mem	mem	–	mem	mem	–	mem	mem	–
la39	15	225	15	61	N	1193.76	1773.00	−48.52	mem	mem	–	mem	mem	–	mem	mem	–
la39	15	225	15	62	Y	787.54	918.24	−16.60	mem	mem	–	mem	mem	–	mem	mem	–
la40	15	225	15	59	N	989.98	712.96	27.98	mem	mem	–	mem	mem	–	mem	mem	–
la40	15	225	15	75	Y	1154.18	1256.54	−8.87	mem	mem	–	mem	mem	–	mem	mem	–

('–' means that the solver did not terminate in one hour, while 'mem' that an out-of-memory error occurred. A '*' means that the local search solver did not find a solution.)

For what concerns the experiments in OPL instead, both CPLEX and SOLVER (run on the linear specification), seem not to be much affected by delaying constraints (or even affected negatively), and anyway are typically slower than SAT. For this reason, detailed results are omitted.

Summing up, we solved several thousands of instances. On the average, delaying constraints seems to be useful when using a SAT solver. In particular, local search solvers like WALKSAT and BG-WALKSAT almost always benefit from the reformulation. The same happens when using SATZ. As for ZCHAFF, we have mixed evidence, since in some cases it seems not to be affected by safe-delay (cf., e.g., results in Table 1), or affected negatively, while in others it benefits from the reformulation (cf., e.g., Tables 2 and 4). The behavior of SAT solvers can be explained by considering the two phenomena that safe-delay produces: (i) The reduction of the number of clauses, and (ii) The enlargement of the set of solutions. The latter phenomenon is particularly beneficial when dealing with local search, as already observed in, e.g., [41,43] where symmetry-breaking (a technique that reduces the solution density) has been experimentally proven to be an obstacle for these algorithms. Of course, also the reduced number of clauses in general helps the solver. Nonetheless, it is worth noting that clauses derived from disjointness constraints are short, and this intuitively explains why the most efficient complete solver, ZCHAFF, that has powerful algorithms to efficiently deal with short clauses (with respect to SATZ), not always benefits from safe-delay.

As far as CPLEX and SOLVER are concerned, we have mixed evidence. First of all, as already observed, it is not always the case that the linear specification solved with CPLEX is always faster than the non-linear one solved with SOLVER, or vice versa. However, in those cases in which CPLEX is faster, delaying constraints often speeds-up the computation. This is consistent with the observation that, from a theoretical point of view, finding a partitioning is much harder than finding a covering, and in practice this often holds also in presence of additional constraints. Moreover, as for the beneficial behavior highlighted on negative instances, deeper analyses on the number of iterations and branches show that (i) More iterations are needed, on the average, by the simplex algorithm when invoked on the specification with no delay, and (ii) The number of branches is often unaffected when performing safe-delay. In particular, the latter issue suggests that each branch-and-bound subproblem is often solved more efficiently on the specification with safe-delay. Unfortunately, since CPLEX is not an open-source system, it is hard to analyze and explain its behavior in greater detail.

Finally, as for SOLVER when invoked on the linear specification, we observed that safe-delay has often (but not always) a negative effect, and this behavior is in line with the common observation, cf., e.g., the literature about symmetry-breaking and the addition of implied constraints, that *restricting* the set of solutions and adding tighter constraints helps when dealing with solvers based on backtracking, since this can significantly increase the amount of propagation, and consequently leads to a better pruning of the search space.

6. Conclusions, related and future work

In this paper we have shown a simple reformulation architecture and proven its soundness for a large class of problems. The reformulation allows to delay the solution of some constraints, which often results in faster solving. In this way, we have shown that reasoning on a specification can be very effective, at least on some classes of solvers.

Although Theorem 1 calls for a tautology checking (cf. Hyp 2), we have shown different specifications for which this test is immediate. Furthermore, we believe that, in practice, an automated theorem prover (ATP) can be used to reason on specifications, thus making it possible to automatically perform the task of choosing constraints to delay. Actually, in another paper [6] we have shown that state-of-the-art ATPs usually perform very well in similar tasks (i.e., detecting and breaking symmetries and detecting functional dependencies on problem specifications).

Related work. Several researchers addressed the issue of reformulation of a problem *after the instantiation phase*: as an example, in [50] it is shown how to translate an instantiated CSP into its Boolean form, which is useful for finding different reformulations, while in [13] the proposed approach is to generate a conjunctive decomposition of an instantiated CSP, by localizing independent subproblems. Furthermore, in [24], the system CGRASS is presented, allowing for the automated breaking of symmetries and the generation of useful implied constraints in a CSP. Finally, in [23] it is shown that abstracting problems by simplifying constraints is useful for finding more efficient reformulations of the original problem; the abstraction may require backtracking for finding solutions of the original problem. In our work, we focus on reformulation of the specification, i.e., regardless of the instance, and, differently from other techniques, the approach is backtracking-free: once the first stage is completed, a solution will surely be found by evaluating the delayed constraints.

Other papers investigate the best way to encode an instance of a problem into a format adequate for a specific solver. As an example, many different ways for encoding graph coloring or permutation problems into SAT have been figured out, cf., e.g., [25,49]. In particular, the idea of looking for “multivalued” functions has been already implemented in ad-hoc techniques for encoding various problems into SAT, like Graph coloring [42,46] and Job shop scheduling [15]. Conversely, in our work we take a specification-oriented approach, giving a formal justification of why some of the constraints can be safely delayed, and presenting sufficient conditions that can be effectively used in order to isolate such constraints.

Finally, we point out that a logic-based approach has also been successfully adopted in the '80s to study the query optimization problem for relational DBs. Analogously to our approach, the query optimization problem has been attacked relying on the query (i.e., the specification) only, without considering the database (i.e., the instance), and it was firstly studied in a formal way using first-order logic (cf., e.g., [2,10,30,44]). In a later stage, the theoretical framework has been translated into rules for the automated rewriting of queries expressed in real world languages and systems.

Since the effectiveness of a particular reformulation technique is expected to depend both on the problem and on the solver (even if this does not rule out, in principle, the possibility to find reformulations that are good for all solvers, or for solvers of a certain class), our research investigates the reformulation problem in different and complementary directions: in particular, in related work, we study how to detect symmetries [35] and functional dependencies [5] among predicates in specifications, and how specifications can be rewritten by exploiting these properties: symmetries can be broken by appropriate symmetry-breaking constraints added to the problem specification, and dependencies can be exploited by automatically synthesizing suitable search strategies. Experimental analysis shows how these approaches are effective in practice, for different classes of solvers. We have also shown [6] how automatic tools, such as first-order theorem provers and finite model finders can be exploited, in practical circumstances, to make this kind of reasoning by computer.

Future work. In this paper we have focused on a form of reformulation which partitions the first-order part of a specification. This basic idea can be generalized, as an example by evaluating in both stages of the computation a constraint (e.g., (9)), or to allow non-shrink second stages (cf., e.g., the specification for the Golomb ruler problem in Example 5), in order to allow reformulation for a larger class of specifications. Even more generally, the second stage may amount to the evaluation of a second-order formula. In the future, we plan—with a more extensive experimentation—to check whether such generalizations are effective in practice.

Another important issue is to understand the relationships between delaying constraints and other techniques, e.g., symmetry breaking. In fact, it is not always the case that delaying constraints, and so making the set of solutions larger, improves the solving process. Adding, e.g., symmetry-breaking or implied constraints are well known techniques that may reach the same goal with the opposite strategy, i.e., reducing the set of solutions. Currently, it is not completely clear in which cases *removing* constraints results in better performances with respect to *adding* more constraints to the specification itself, even if it seems that an important role is played by the nature of constraints we remove or add, e.g., by their amenability to propagation in the search tree, together with the nature of the solver used, e.g., backtracking, linear, or based on local search. As for the latter class of solvers, it is known that enlarging the set of solutions can significantly speed-up performances (cf., e.g., [41,43]). Our experiments on WALKSAT and BG-WALKSAT confirm this thesis.

Finally, it is our goal to rephrase the theoretical results into rules for automatically reformulating problem specifications given in more complex languages, e.g. AMPL and OPL, which have higher-level built-in constructs.

Acknowledgements

This research has been supported by MIUR (Italian Ministry for Instruction, University, and Research) under the FIRB project ASTRO (Automazione dell'Ingegneria del Software basata su Conoscenza), and under the COFIN project "Design and development of a software system for the specification and efficient solution of combinatorial problems, based on a high-level language, and techniques for intensional reasoning and local search". The authors are grateful to Carlo Mannino and Stefano Smriglio for useful discussions about CPLEX, and to the anonymous reviewers for their comments and suggestions.

Appendix A. Proofs of results

Proof of Theorem 1. Let $\mathcal{I}$ be an instance, M^* be a list of extensions for $(S_1, \ldots, S_h, S^*)$ such that $(M^*, \mathcal{I}) \models \psi^s$, and $ext(S)$ be an extension for S such that $(M^*, ext(S), \mathcal{I}) \models \psi^d$.

From the definition of ψ^d, it follows that:

$$\bigl(M^*, ext(S), \mathcal{I}\bigr) \models \forall X\ S(X) \rightarrow S^*(X),$$

and so, since clauses in Ξ^* contain at most negative occurrences of S^*, that:

$$\big(M^*, ext(S), \mathcal{I}\big) \models \Xi. \tag{A.1}$$

Furthermore, from the definition of ψ^s it follows that:

$$\big(M^*, \mathcal{I}\big) \models \forall X\ \alpha(X) \rightarrow S^*(X),$$

and from Hyp 2 that:

$$(M^*, \mathcal{I}) \models \forall X\ \alpha(X) \rightarrow S^*(X) \wedge \beta(X).$$

This implies, by the definition of ψ^d, that:

$$\big(M^*, ext(S), \mathcal{I}\big) \models \forall X\ \alpha(X) \rightarrow S(X). \tag{A.2}$$

Moreover, by the same definition, it is also true that:

$$\big(M^*, ext(S), \mathcal{I}\big) \models \forall X\ S(X) \rightarrow \beta(X). \tag{A.3}$$

From (A.1–A.3), and from the observation that S^* does not occur in any of the right parts of them, the thesis follows. □

Proof of Theorem 2. Let $\mathcal{I}$ be an input instance, and M be a list of extensions for $(S_1, \ldots, S_h, S)$ such that $(M, \mathcal{I}) \models \psi$. Let S^* be defined in such a way that $ext(S^*) = ext(S)$.

Since solutions for ψ are also solutions for ψ^s, and since $ext(S^*) = ext(S)$, it follows that $((M - ext(S)) \cup ext(S^*))$ is a solution of ψ^s.

As for ψ^d, we observe that since M (which is a solution for the whole specification ψ) is also a solution for one of its constraints, namely $\forall X\ S(X) \rightarrow \beta(X)$ (the delayed constraint), then, from $ext(S^*) = ext(S)$, and in particular from the fact that $\forall X\ S(X) \rightarrow S^*(X)$ holds, it follows that $(M, ext(S^*)) \models \forall X\ S(X) \rightarrow S^*(X) \wedge \beta(X)$.

Conversely, from $ext(S^*) = ext(S)$, and in particular from the fact that $\forall X\ S^*(X) \rightarrow S(X)$ holds, it follows that $(M, ext(S^*)) \models \forall X\ S^*(X) \wedge \beta(X) \rightarrow S(X)$.

Hence, the thesis follows. □

References

[1] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley Publishing Company, Reading, MA, 1995.
[2] A.V. Aho, Y. Sagiv, J.D. Ullman, Equivalences among relational expressions, SIAM Journal on Computing 2 (8) (1979) 218–246.
[3] E. Börger, E. Gräedel, Y. Gurevich, The Classical Decision Problem, Perspectives in Mathematical Logic, Springer, Berlin, 1997.
[4] C.A. Brown, L. Finkelstein, P.W. Purdom, Backtrack searching in the presence of symmetry, in: T. Mora (Ed.), Proceedings of the Sixth International Conference on Applied Algebra, Algebraic Algorithms and Error Correcting Codes, Rome, Italy, in: Lecture Notes in Computer Science, vol. 357, Springer, Berlin, 1988, pp. 99–110.
[5] M. Cadoli, T. Mancini, Exploiting functional dependencies in declarative problem specifications, in: Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA 2004), Lisbon, Portugal, in: Lecture Notes in Artificial Intelligence, vol. 3229, Springer, Berlin, 2004.
[6] M. Cadoli, T. Mancini, Using a theorem prover for reasoning on constraint problems, in: Proceedings of the Ninth Conference of the Italian Association for Artificial Intelligence (AI*IA 2005), Milano, Italy, in: Lecture Notes in Artificial Intelligence, vol. 3673, Springer, Berlin, 2005, pp. 38–49.
[7] M. Cadoli, T. Mancini, F. Patrizi, SAT as an effective solving technology for constraint problems, in: Proceedings of the Twentieth Convegno Italiano di Logica Computazionale (CILC 2005), Roma, Italy, 2005.
[8] M. Cadoli, A. Schaerf, Compiling problem specifications into SAT, Artificial Intelligence 162 (2005) 89–120.
[9] E. Castillo, A.J. Conejo, P. Pedregal, R. García, N. Alguacil, Building and Solving Mathematical Programming Models in Engineering and Science, John Wiley & Sons, 2001.
[10] A.K. Chandra, P.M. Merlin, Optimal implementation of conjunctive queries in relational databases, in: Proceedings of the Ninth ACM Symposium on Theory of Computing (STOC'77), Boulder, CO, ACM Press, 1977, pp. 77–90.
[11] P. Cheeseman, B. Kanefski, W.M. Taylor, Where the really hard problem are, in: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI'91), Sydney, Australia, Morgan Kaufmann, Los Altos, CA, 1991, pp. 163–169.
[12] B.M.W. Cheng, K.M.F. Choi, J.H.-M. Lee, J.C.K. Wu, Increasing constraint propagation by redundant modeling: an experience report, Constraints 4 (2) (1999) 167–192.
[13] B.Y. Choueiry, G. Noubir, On the computation of local interchangeability in discrete constraint satisfaction problems, in: Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98), Madison, WI, AAAI Press/The MIT Press, 1998, pp. 326–333.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, 1990.

[15] J.M. Crawford, A.B. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, in: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94), Seattle, WA, AAAI Press/The MIT Press, 1994, pp. 1092–1097.

[16] J.M. Crawford, M.L. Ginsberg, E.M. Luks, A. Roy, Symmetry-breaking predicates for search problems, in: Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, MA, Morgan Kaufmann, Los Altos, CA, 1996, pp. 148–159.

[17] R. Dechter, Constraint networks (survey), in: Encyclopedia of Artificial Intelligence, second ed., John Wiley & Sons, 1992, pp. 276–285.

[18] D. East, M. Truszczyǹski, Predicate-calculus based logics for modeling and solving search problems ACM Transactions on Computational Logic, 2004, submitted for publication.

[19] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, in: R.M. Karp (Ed.), Complexity of Computation, American Mathematical Society, Providence, RI, 1974, pp. 43–74.

[20] P. Flener, Towards relational modelling of combinatorial optimisation problems, in: C. Bessière (Ed.), Proceedings of the International Workshop on Modelling and Solving Problems with Constraints, in conjunction with the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, 2001.

[21] P. Flener, J. Pearson, M. Ågren, Introducing ESRA, a relational language for modelling combinatorial problems, in: Proceedings of the Eighteenth IEEE Symposium on Logic in Computer Science (LICS 2004), Uppsala, Sweden, Springer, Berlin, 2003, pp. 214–232.

[22] R. Fourer, D.M. Gay, B.W. Kernigham, AMPL: A Modeling Language for Mathematical Programming, International Thomson Publishing, 1993.

[23] E.C. Freuder, D. Sabin, Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction, in: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97), Providence, RI, AAAI Press/The MIT Press, 1997, pp. 191–196.

[24] A. Frisch, I. Miguel, T. Walsh, CGRASS: A system for transforming constraint satisfaction problems, in: Proceedings of the Joint Workshop of the ERCIM Working Group on Constraints and the CologNet area on Constraint and Logic Programming on Constraint Solving and Constraint Logic Programming (ERCIM 2002), Cork, Ireland, in: Lecture Notes in Artificial Intelligence, vol. 2627, Springer, Berlin, 2002, pp. 15–30.

[25] A.M. Frisch, T.J. Peugniez, Solving non-boolean satisfiability problems with stochastic local search, in: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, Morgan Kaufmann, Los Altos, CA, 2001, pp. 282–290.

[26] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, San Francisco, CA, 1979.

[27] E. Giunchiglia, R. Sebastiani, Applying the Davis–Putnam procedure to non-clausal formulas, in: Proceedings of the Sixth Conference of the Italian Association for Artificial Intelligence (AI*IA'99), Bologna, Italy, in: Lecture Notes in Artificial Intelligence, vol. 1792, Springer, Berlin, 2000, pp. 84–94.

[28] F. Giunchiglia, T. Walsh, A theory of abstraction, Artificial Intelligence 57 (1992) 323–389.

[29] T. Hnich, T. Walsh, Why Channel? Multiple viewpoints for branching heuristics, in: Proceedings of the Second International Workshop on Modelling and Reformulating CSPs: Towards Systematisation and Automation, in conjunction with the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003), Kinsale, Ireland, 2003.

[30] A. Klug, On conjunctive queries containing inequalities, Journal of the ACM 1 (35) (1988) 146–160.

[31] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV System for knowledge representation and reasoning, in: ACM Transactions on Computational Logic, submitted for publication.

[32] C.M. Li, Integrating equivalency reasoning into Davis–Putnam procedure, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000), Austin, TX, AAAI Press/The MIT Press, 2000, pp. 291–296.

[33] C.M. Li, Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), Nagoya, Japan, Morgan Kaufmann, Los Altos, CA, 1997, pp. 366–371.

[34] T. Mancini, Reformulation techniques for a class of permutation problems, in: Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP 2003), Kinsale, Ireland, in: Lecture Notes in Computer Science, vol. 2833, Springer, Berlin, 2003, p. 984.

[35] T. Mancini, M. Cadoli, Detecting and breaking symmetries by reasoning on problem specifications, in: Proceedings of the Sixth International Symposium on Abstraction, Reformulation and Approximation (SARA 2005), Airth Castle, Scotland, UK, in: Lecture Notes in Artificial Intelligence, vol. 3607, Springer, Berlin, 2005, pp. 165–181.

[36] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementation, John Wiley & Sons, 1990.

[37] B.D. McKay, Nauty user's guide (version 2.2). Available at http://cs.anu.edu.au/~bdm/nauty/nug.pdf, 2003.

[38] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, in: Proceedings of the Thirty Eighth Conference on Design Automation (DAC 2001), Las Vegas, NV, ACM Press, 2001, pp. 530–535.

[39] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, Annals of Mathematics and Artificial Intelligence 25 (3,4) (1999) 241–273.

[40] C.H. Papadimitriou, Computational Complexity, Addison Wesley Publishing Company, Reading, MA, 1994.

[41] S.D. Prestwich, Supersymmetric modeling for local search, in: Proceedings of the Second International Workshop on Symmetry in Constraint Satisfaction Problems, in conjunction with the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002), Ithaca, NY, 2002.

[42] S.D. Prestwich, Local search on SAT-encoded colouring problems, in: Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Santa Margherita Ligure, Genova, Italy, in: Lecture Notes in Computer Science, vol. 2919, Springer, Berlin, 2004, pp. 105–119.

[43] S.D. Prestwich, A. Roli, Symmetry breaking and local search spaces, in: R. Barták, M. Milano (Eds.), Proceedings of the Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2005), Prague, CZ, in: Lecture Notes in Computer Science, vol. 3524, Springer, Berlin, 2005, pp. 273–287.
[44] Y. Sagiv, M. Yannakakis, Equivalence among relational expressions with the union and difference operations, Journal of the ACM 4 (27) (1980) 633–655.
[45] B. Selman, H.A. Kautz, B. Cohen, Local search strategies for satisfiability testing, in: M. Trick, D.S. Johnson (Eds.), Proceedings of the Second DIMACS Challenge on Cliques, Coloring, and Satisfiability, Providence, RI, 1993.
[46] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability instances, in: Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92), San Jose, CA, AAAI Press/The MIT Press, 1992.
[47] B.M. Smith, K. Stergiou, T. Walsh, Using auxiliary variables and implied constraints to model non-binary problems, in: Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI 2000), Austin, TX, AAAI Press/The MIT Press, 2000, pp. 182–187.
[48] P. Van Hentenryck, The OPL Optimization Programming Language, The MIT Press, 1999.
[49] T. Walsh, Permutation problems and channelling constraints, in: R. Nieuwenhuis, A. Voronkov (Eds.), Proceedings of the Eighth International Conference on Logic for Programming and Automated Reasoning (LPAR 2001), Havana, Cuba, in: Lecture Notes in Computer Science, vol. 2250, Springer, Berlin, 2001, pp. 377–391.
[50] R. Weigel, C. Bliek, On reformulation of constraint satisfaction problems, in: Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI'98), Brighton, UK, John Wiley & Sons, 1998, pp. 254–258.
[51] W. Zhang, A. Rangan, M. Looks, Backbone guided local search for maximum satisfiability, in: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, Morgan Kaufmann, Los Altos, CA, 2003, pp. 1179–1186.