

THE SECOND INTERNATIONAL IJCAI WORKSHOP ON
GRAPH STRUCTURES FOR
KNOWLEDGE REPRESENTATION AND REASONING
(GKR 2011)

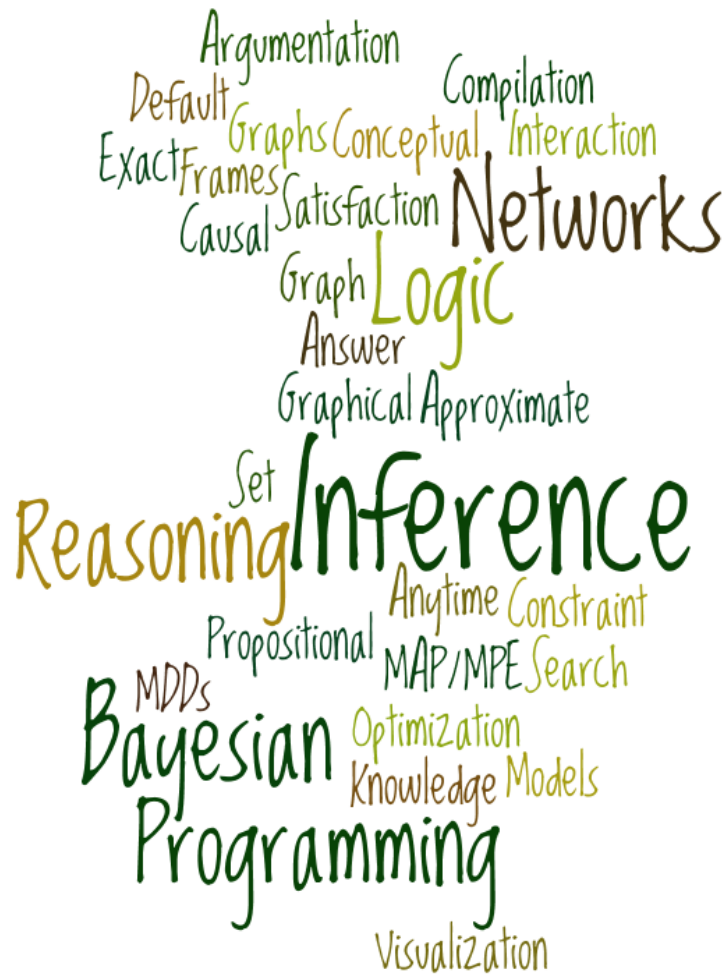


Table of Contents

Local Characterizations of Causal Bayesian Networks	6
<i>Elias Bareinboim, Carlos Brito and Judea Pearl</i>	
Boolean formulas of simple conceptual graphs	12
<i>Olivier Carloni</i>	
Graph Representations of Consistency and Truth-Dependencies in Logic Programs With Answer Set Semantics	16
<i>Stefania Costantini and Alessandro Provetti</i>	
Bucket and mini-bucket Schemes for M Best Solutions over Graphical Models	24
<i>Natalia Flerova, Rina Dechter and Emma Rollon</i>	
Supporting Argumentation Systems by Graph Representation and Computation	32
<i>Jérôme Fortin, Rallou Thomopoulos, Jean-Rémi Bourguet and Marie-Laure Mugnier</i>	
Representing CSPs with Set-labeled Diagrams: A Compilation Map	40
<i>Alexandre Niveau, Helene Fargier and Cédric Pralet</i>	
Anytime Depth-First Search with Problem Decomposition for Optimization in Graphical Models ..	48
<i>Lars Otten and Rina Dechter</i>	
Visually Interacting with a Knowledge Base using Frames, Logic, and Propositional Graphs	56
<i>Daniel Schlegel and Stuart Shapiro</i>	

Keyword Index

Acceptability of Arguments	22
Answer Set Programming	13
Anytime Search	26
Approximate Inference	21
Argumentation	22
Bayesian Networks	21
Branch and Bound	24
Causal Bayesian Networks	6
Causality	6
Computation of Argument Acceptability	23
Conceptual Graphs	12, 22
Constraint Satisfaction	23
Default Reasoning	22
Exact Inference	21
Frame	25
Graph representation of programs	13
Graphical Models	21, 24
Guarded Fragment	12
Interaction	25
Knowledge Compilation	23
Local Characterizations	6
Logic 10	, 25
Logic Programming	13
MAP/MPE Inference	24
MDDs	23
Optimization	21, 24
Program comprehension	13
Propositional Graph	25
Reasoning	13

Visualization	25
Weighted CSPs	26

Author Index

Bareinboim, Elias	6
Bourguet, Jean-Rémi	32
Brito, Carlos	6
Carlioni, Olivier	12
Costantini, Stefania	16
Dechter, Rina	24, 48
Fargier, Helene	40
Flerova, Natalia	24
Fortin, Jérôme	32
Mugnier, Marie-Laure	32
Niveau, Alexandre	40
Otten, Lars	48
Pearl, Judea	6
Pralet, Cédric	40
Provetti, Alessandro	16
Rollon, Emma	24
Schlegel, Daniel	56
Shapiro, Stuart	56
Thomopoulos, Rallou	32

Local Characterizations of Causal Bayesian Networks*

Elias Bareinboim

Cognitive Systems Laboratory
Computer Science Department
University of California Los Angeles
CA 90095
eb@cs.ucla.edu

Carlos Brito

Computer Science Department
Federal University of Ceará
carlos@lia.ufc.br

Judea Pearl

Cognitive Systems Laboratory
Computer Science Department
University of California Los Angeles
CA 90095
judea@cs.ucla.edu

Abstract

The standard definition of causal Bayesian networks (CBNs) invokes a global condition according to which the distribution resulting from any intervention can be decomposed into a truncated product dictated by its respective mutilated subgraph. We provide alternative formulations which emphasize local aspects of the causal process and can serve therefore as more meaningful criterion for testing coherence and network construction. We first examine a definition based on “modularity” and prove its equivalence to the global definition. We then introduce two new definitions, the first interprets the missing edges in the graph, and the second interprets “zero direct effect” (i.e., *ceteris paribus*). We show that these two formulations are equivalent but carry different semantic content.

1 Introduction

Nowadays, graphical models are standard tools for encoding distributional and causal information [Pearl, 1988; Spirtes *et al.*, 1993; Heckerman and Shachter, 1995; Lauritzen, 1999; Pearl, 2000; Dawid, 2001; Koller and Friedman, 2009]. One of the most popular representations is a *causal Bayesian network*, namely, a directed acyclic graph (DAG) G which, in addition to the traditional conditional independencies also conveys causal information, and permits one to infer the effects of interventions. Specifically, if an external intervention fixes any set $\mathbf{X}$ of variables to some constant $\mathbf{x}$, the DAG permits us to infer the resulting post-intervention distribution, denoted by $P_{\mathbf{x}}(\mathbf{v})$,¹ from the pre-intervention distribution $P(\mathbf{v})$.

The standard reading of post-interventional probabilities invokes cutting off incoming arrows to the manipulated variables and leads to a “truncated product” formula [Pearl,

1993], also known as “manipulation theorem” [Spirtes *et al.*, 1993] and “G-computation formula” [Robins, 1986]. A local characterization of CBNs invoking the notion of conditional invariance was presented in [Pearl, 2000, p.24] and will be shown here to imply (and be implied by) the truncated product formula. This characterization requires the network builder to judge whether the conditional probability $P(Y \mid \mathbf{PA}_Y)$ for each parents-child family remains invariant under interventions outside this family. [Tian and Pearl, 2002] provides another characterization with respect to three norms of coherence called Effectiveness, Markov and Recursiveness, and showed their use in learning and identification when the causal graph is not known in advance.

In this paper, we use the concepts of “conditional invariance” and “interventional invariance” to formulate and compare several definitions of CBNs. The first assures invariance of conditional probabilities for each family, while the other two assure the invariance of the distribution of each variable under different interventions. We show that these three definitions are equivalent to the global one, and lead to the same predictions under interventions.

The rest of the paper is organized as follows. In Section 2, we introduce the basic concepts, and present the standard global and local definitions of CBNs together with discussion of their features. In Section 3, we prove the equivalence between these two definitions. In Section 4, we introduce two new definitions which explicitly interpret the missing links in the graph as representing absence of causal influence. In Section 5, we prove the equivalence between these definitions and the previous ones. Finally, we provide concluding remarks in Section 6.

2 Causal Bayesian networks and interventions

A causal Bayesian network (also known as a *Markovian model*) consists of two mathematical objects: (i) a DAG G , called a causal graph, over a set $\mathbf{V} = \{V_1, \dots, V_n\}$ of vertices, and (ii) a probability distribution $P(\mathbf{v})$, over the set $\mathbf{V}$ of discrete variables that correspond to the vertices in G . The interpretation of such a graph has two components, probabilistic and causal.²

*This work was supported in parts by National Institutes of Health #1R01 LM009961-01, National Science Foundation #IIS-0914211 and #IIS-1018922, and Office of Naval Research #N00014-09-1-0665 and #N00014-10-1-0933.

¹[Pearl, 2000] used the notation $P(\mathbf{v} \mid \text{set}(\mathbf{t}))$, $P(\mathbf{v} \mid \text{do}(\mathbf{t}))$, or $P(\mathbf{v} \mid \hat{\mathbf{t}})$ for the post-intervention distribution, while [Lauritzen, 1999] used $P(\mathbf{v} \parallel \mathbf{t})$.

²A more refined interpretation, called functional, is also common [Pearl, 2000], which, in addition to interventions, supports counterfactual readings. The functional interpretation assumes determinis-

The probabilistic interpretation [Pearl, 1988] views G as representing conditional independence restrictions on P : each variable is independent of all its non-descendants given its parents in the graph. This property is known as the *Markov condition* and characterizes the Bayesian network absent of any causal reading. These conditional independence restrictions imply that the joint probability function $P(\mathbf{v}) = P(v_1, \dots, v_n)$ factorizes according to the product:

$$P(\mathbf{v}) = \prod_i P(v_i \mid \mathbf{pa}_i) \quad (1)$$

where $\mathbf{pa}_i$ are (assignments of) the parents of variables V_i in G .

The causal interpretation views the arrows in G as representing potential causal influences between the corresponding variables and, alternatively, the absence of arrows represents no direct causal influence between the corresponding variables. In this interpretation, the factorization of eq. (1) still holds, but the factors are further assumed to represent autonomous data-generation processes, that is, each family conditional probability $P(v_i \mid \mathbf{pa}_i)$ represents a stochastic process by which the values of V_i are assigned in response to the values $\mathbf{pa}_i$ (previously chosen for V_i 's parents), and the stochastic variation of this assignment is assumed independent of the variations in all other assignments in the model.

Moreover, each assignment process remains invariant to possible changes in the assignments processes that govern other variables in the system. This invariance assumption is known as modularity and it enables us to predict the effects of interventions, whenever interventions are described as specific modification of some factors in the product of eq. (1). The most elementary intervention considered is the *atomic* one, where a set $\mathbf{X}$ of variables is fixed to some constant $\mathbf{x} = \mathbf{x}$. The following definitions will facilitate subsequent discussions.

Definition 1 (Interventional distributions). *Let $P(\mathbf{v})$ be a probability distribution on a set $\mathbf{V}$ of variables, and let $P_{\mathbf{x}}(\mathbf{v})$ denote the distribution resulting from the intervention $\text{do}(\mathbf{X} = \mathbf{x})$ that sets a subset $\mathbf{X}$ of variables to constant $\mathbf{x}$. Denote by $\mathbf{P}_*$ the set of all interventional distributions $P_{\mathbf{x}}(\mathbf{v})$, $\mathbf{X} \subseteq \mathbf{V}$, including $P(\mathbf{v})$, which represents no intervention (i.e., $\mathbf{X} = \emptyset$). Additionally, $\mathbf{P}_*$ is such that for all $\mathbf{X} \subseteq \mathbf{V}$, the following property holds:*

- i. [Effectiveness] $P_{\mathbf{x}}(v_i) = 1$, for all $V_i \in \mathbf{X}$ whenever v_i is consistent with $\mathbf{X} = \mathbf{x}$;

Definition 2 (Conditional invariance). *We say that Y is conditional invariant to $\mathbf{X}$ given $\mathbf{Z}$, denoted $(Y \perp\!\!\!\perp_{ci} \mathbf{X} \mid \mathbf{Z})_{\mathbf{P}_*}$, if intervening on $\mathbf{X}$ does not change the conditional distribution of Y given $\mathbf{Z} = \mathbf{z}$, i.e., $\forall \mathbf{x}, y, \mathbf{z}, P_{\mathbf{x}}(y \mid \mathbf{z}) = P(y \mid \mathbf{z})$.*

To capture the intuition behind atomic interventions, [Pearl, 2000] proposed the following local definition of causal Bayesian networks:

tic functional relationships between variables in the model, some of which may be unobserved. Complete axiomatizations of deterministic counterfactual relations are given in [Galles and Pearl, 1998; Halpern, 1998].

Definition 3 (Causal Bayesian network [Pearl, 2000, p.24]). *A DAG G is said to be locally compatible with a set of interventional distributions $\mathbf{P}_*$ if and only if the following conditions hold for every $P_{\mathbf{x}} \in \mathbf{P}_*$:*

- i. [Markov] $P_{\mathbf{x}}(\mathbf{v})$ is Markov relative to G ;
- ii. [Modularity] $(V_i \perp\!\!\!\perp_{ci} \mathbf{X} \mid \mathbf{PA}_i)_{\mathbf{P}_*}$, for all $V_i \notin \mathbf{X}$ whenever $\mathbf{pa}_i$ is consistent with $\mathbf{X} = \mathbf{x}$.³

We shall show (Sec. 3) that modularity permits us to answer queries about the effect of interventions, or causal effects. A *causal effect* of variable X on variable Y written $P_x(y)$, stands for the probability that variable Y attains value y if we enforce uniformly over the population the constraint $X = x$. The standard definition of causal Bayesian networks is based on a global compatibility condition, which makes explicit the joint post-intervention distribution under any arbitrary intervention.

Definition 4 (Global causal Bayesian network [Pearl, 2000]). *A DAG G is said to be globally compatible with a set of interventional distributions $\mathbf{P}_*$ if and only if the distribution $P_{\mathbf{x}}(\mathbf{v})$ resulting from the intervention $\text{do}(\mathbf{X} = \mathbf{x})$ is given by the following expression:*

$$P_{\mathbf{x}}(\mathbf{v}) = \begin{cases} \prod_{\{i \mid V_i \notin \mathbf{X}\}} P(v_i \mid \mathbf{pa}_i) & \mathbf{v} \text{ consistent with } \mathbf{x}. \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

Equation (2) is also known as the *truncated factorization product* of eq. (1), with factors corresponding to the manipulated variables removed. The truncated factorization follows from Definition 3 because, assuming modularity, the post-intervention probabilities $P(v_i \mid \mathbf{pa}_i)$ corresponding to variables in $\mathbf{X}$ are either 1 or 0, while those corresponding to unmanipulated variables remain unaltered.

The two definitions emphasize different aspects of the causal model; Definition 3 ensures that each conditional probability $P(v_i \mid \mathbf{pa}_i)$ (locally) remains invariant under interventions that do not include directly V_i , while Definition 4 ensures that each manipulated variable is not influenced by its previous parents (before the manipulation), and every other variable is governed by its pre-interventional process. Because the latter invokes theoretical conditions on the data-generating process, it is not directly testable, and the question whether a given implemented intervention conforms to an investigator's intention (e.g., no side effects) is discernible only through the testable properties of the truncated product formula (2). Definition 3 provides in essence a series of local tests for Equation (2), and the equivalence between the two (Theorem 1, below) ensures that *all* empirically testable properties of (2) are covered by the local tests provided by Definition 3.

³Explicitly, modularity states: $P(v_i \mid \mathbf{pa}_i, \text{do}(\mathbf{s})) = P(v_i \mid \mathbf{pa}_i)$ for any set $\mathbf{S}$ of variables disjoint of $\{V_i, \mathbf{PA}_i\}$.

3 The equivalence between the local and global definitions

We prove next that the local and global definitions of causal Bayesian networks are equivalent. To the best of our knowledge, the proof of equivalence has not been published before.

Theorem 1 (Equivalence between local and global compatibility). *Let G be a DAG and $\mathbf{P}_*$ a set of interventional distributions, the following statements are equivalent:*

- i. G is locally compatible with $\mathbf{P}_*$
- ii. G is globally compatible with $\mathbf{P}_*$

Proof. (Definition 3 $\Rightarrow$ Definition 4)

Given an intervention $do(\mathbf{X} = \mathbf{x})$, $\mathbf{X} \subseteq \mathbf{V}$, assume that conditions 3:(i-ii) are satisfied. For any arbitrary instantiation $\mathbf{v}$ of variables $\mathbf{V}$, consistent with $\mathbf{X} = \mathbf{x}$, we can express $P_{\mathbf{x}}(\mathbf{v})$ as

$$\begin{aligned}
 P_{\mathbf{x}}(\mathbf{v}) &\stackrel{\text{def.3:(i)}}{=} \prod_i P_{\mathbf{x}}(v_i | \mathbf{pa}_i) \\
 &= \prod_{\{i|v_i \in \mathbf{X}\}} P_{\mathbf{x}}(v_i | \mathbf{pa}_i) \prod_{\{i|v_i \notin \mathbf{X}\}} P_{\mathbf{x}}(v_i | \mathbf{pa}_i) \\
 &\stackrel{\text{effectiveness}}{=} \prod_{\{i|v_i \notin \mathbf{X}\}} P_{\mathbf{x}}(v_i | \mathbf{pa}_i) \\
 &\stackrel{\text{def.3:(ii)}}{=} \prod_{\{i|v_i \notin \mathbf{X}\}} P(v_i | \mathbf{pa}_i)
 \end{aligned}$$

which is the truncated product as desired.

(Definition 4 $\Rightarrow$ Definition 3)

We assume that the truncated factorization holds, i.e., the distribution $P_{\mathbf{x}}(\mathbf{v})$ resulting from any intervention $do(\mathbf{X} = \mathbf{x})$ can be computed as eq. (2).

To prove effectiveness, consider an intervention $do(\mathbf{X} = \mathbf{x})$, and let $v_i \in \mathbf{X}$. Let $Dom(v_i) = \{v_{i1}, v_{i2}, \dots, v_{im}\}$ be the domain of variable V_i , with only one of those values consistent with $\mathbf{X} = \mathbf{x}$. Since $P_{\mathbf{x}}(\mathbf{v})$ is a probability distribution, we must have $\sum_j P_{\mathbf{x}}(V_i = v_{ij}) = 1$. According to eq. (2), all terms not consistent with $\mathbf{X} = \mathbf{x}$ have probability zero, and thus we obtain $P_{\mathbf{x}}(v_i) = 1$, v_i consistent with $\mathbf{X} = \mathbf{x}$.

To show Definition 3:(ii), we consider an ordering $\pi : (v_1, \dots, v_n)$ of the variables, consistent with the graph G induced by the truncated factorization with no intervention $P(\mathbf{v}) = \prod_i P(v_i | \mathbf{pa}_i)$. Now, given an intervention $do(\mathbf{X} = \mathbf{x})$

$$P_{\mathbf{x}}(v_i | \mathbf{pa}_i) = \frac{P_{\mathbf{x}}(v_i, \mathbf{pa}_i)}{P_{\mathbf{x}}(\mathbf{pa}_i)}$$

$$\begin{aligned}
 &\stackrel{\text{marginal.}}{=} \frac{\sum_{v_j \notin \{V_i, \mathbf{PA}_i\}} P_{\mathbf{x}}(\mathbf{v})}{\sum_{v_j \notin \{\mathbf{PA}_i\}} P_{\mathbf{x}}(\mathbf{v})} \\
 &\stackrel{\text{eq.(2)}}{=} \frac{\sum_{v_j \notin \{V_i, \mathbf{PA}_i, \mathbf{X}\}} \prod_{v_k \notin \mathbf{X}} P(v_k | \mathbf{pa}_k)}{\sum_{v_j \notin \{\mathbf{PA}_i, \mathbf{X}\}} \prod_{v_k \notin \mathbf{X}} P(v_k | \mathbf{pa}_k)} \\
 &= P(v_i | \mathbf{pa}_i) \times \frac{\sum_{v_j \notin \{V_i, \mathbf{PA}_i, \mathbf{X}\}} \prod_{v_k \notin \mathbf{X}, k \neq i} P(v_k | \mathbf{pa}_k)}{\sum_{v_j \notin \{\mathbf{PA}_i, \mathbf{X}\}} \prod_{v_k \notin \mathbf{X}} P(v_k | \mathbf{pa}_k)}
 \end{aligned} \tag{4}$$

The last step is due to the fact that variables in $\{V_i, \mathbf{PA}_i\}$ do not appear in the summations in the numerator. Rewriting the numerator, breaking it in relation to variables before and after v_i , we obtain

$$\begin{aligned}
 &\sum_{v_j \notin \{V_i, \mathbf{PA}_i, \mathbf{X}\}} \prod_{\substack{v_k \notin \mathbf{X} \\ k \neq i}} P(v_k | \mathbf{pa}_k) = \\
 &\sum_{\substack{v_j \notin \{\mathbf{PA}_i, \mathbf{X}\} \\ j < i}} \prod_{\substack{v_k \notin \mathbf{X} \\ k < i}} P(v_k | \mathbf{pa}_k) \sum_{\substack{v_j \notin \mathbf{X} \\ j > i}} \prod_{\substack{v_k \notin \mathbf{X} \\ k > i}} P(v_k | \mathbf{pa}_k)
 \end{aligned} \tag{5}$$

Note that $\sum_{\substack{v_j \notin \mathbf{X} \\ j > i}} \prod_{\substack{v_k \notin \mathbf{X} \\ k > i}} P(v_k | \mathbf{pa}_k) = 1$ because all $V_j > V_i$ appear in the summation. Thus, we obtain

$$\begin{aligned}
 &\sum_{v_j \notin \{V_i, \mathbf{PA}_i, \mathbf{X}\}} \prod_{v_k \notin \mathbf{X}} P(v_k | \mathbf{pa}_k) = \\
 &\sum_{\substack{v_j \notin \{\mathbf{PA}_i, \mathbf{X}\} \\ j < i}} \prod_{\substack{v_k \notin \mathbf{X} \\ k < i}} P(v_k | \mathbf{pa}_k)
 \end{aligned} \tag{6}$$

Similarly for the denominator,

$$\begin{aligned}
 &\sum_{v_j \notin \{\mathbf{PA}_i, \mathbf{X}\}} \prod_{v_k \notin \mathbf{X}} P(v_k | \mathbf{pa}_k) = \\
 &\sum_{\substack{v_j \notin \{\mathbf{PA}_i, \mathbf{X}\} \\ j < i}} \prod_{\substack{v_k \notin \mathbf{X} \\ k < i}} P(v_k | \mathbf{pa}_k)
 \end{aligned} \tag{7}$$

Observe that eqs. (6) and (7) are identical, equation (4) reduces to $P_{\mathbf{x}}(v_i | \mathbf{pa}_i) = P(v_i | \mathbf{pa}_i)$ as desired.

To show Definition 3:(i), we first use the truncated factorization

$$\begin{aligned}
 P_{\mathbf{x}}(\mathbf{v}) &\stackrel{\text{eq.(2)}}{=} \prod_{\{i, v_i \notin \mathbf{X}\}} P(v_i | \mathbf{pa}_i) \\
 &\stackrel{\text{def.3:(ii)}}{=} \prod_{\{i, v_i \notin \mathbf{X}\}} P_{\mathbf{x}}(v_i | \mathbf{pa}_i) \\
 &\stackrel{\text{effectiveness}}{=} \prod_i P_{\mathbf{x}}(v_i | \mathbf{pa}_i)
 \end{aligned} \tag{8}$$

Finally, def. 3:(i) follows from the definition of Markov compatibility (definition 1.2.2 in [Pearl, 2000]). $\square$

4 Alternative characterization of Causal Bayesian Networks

We state next a local definition of CBNs which focuses on the absence of edges in the causal graph, i.e., the missing-links representing absence of causal influence.

Definition 5 (Missing-link causal Bayesian network). *A DAG G is said to be missing-link compatible with a set of interventional distributions $\mathbf{P}_*$ if and only if the following conditions hold:*

- i. [Markov] $\forall \mathbf{X} \subseteq \mathbf{V}$, $P_{\mathbf{x}}(\mathbf{v})$ is Markov relative to G ;
- ii. [Missing-link] $\forall \mathbf{X} \subseteq \mathbf{V}$, $Y \in \mathbf{V}$, $\mathbf{S} \subseteq \mathbf{V}$, $P_{\mathbf{x}, \mathbf{s}, \mathbf{pa}_Y}(y) = P_{\mathbf{s}, \mathbf{pa}_Y}(y)$ whenever there is no arrow from $\mathbf{X}$ to Y in G , $\mathbf{pa}_Y$ is consistent with $\{\mathbf{X} = \mathbf{x}, \mathbf{S} = \mathbf{s}\}$ and $\mathbf{X}, \{Y\}, \mathbf{S}$ are disjoint.
- iii. [Parents do/see] $\forall Y \in \mathbf{V}$, $\mathbf{X} \subseteq \mathbf{V}$, $P_{\mathbf{x}, \mathbf{pa}_Y}(y) = P_{\mathbf{x}}(y \mid \mathbf{pa}_Y)$ whenever $\mathbf{pa}_Y$ is consistent with $\mathbf{X} = \mathbf{x}$ and $\mathbf{X}, \{Y\}$ are disjoint.

Condition (ii) requires that when we set $\mathbf{X}$ to some value while keeping the variables $\mathbf{S} \cup \mathbf{pa}_Y$ constant, the marginal distribution of Y remains unaltered, independent of the value of $\mathbf{X}$, whenever there is no edge between $\mathbf{X}$ and Y , i.e., an intervention on $\mathbf{X}$ does not change Y 's distribution while holding constant its parents. In addition to the missing-link condition, 5:(iii) describes the relationship inside each family, i.e., the effect on Y should be the same whether observing (seeing) or intervening (doing) on its parents $\mathbf{pa}_Y$.

Note that the missing-link condition is not sufficient on its own to characterize causal Bayesian networks – condition 5:(iii) is also necessary when there is a link between any two variables. To illustrate, consider a DAG G with only two binary variables A and B , and an edge from A to B . Without condition 5:(iii), the interventional distribution $P_a(b)$ is unconstrained, which allows $P_a(b) \neq P(b \mid a)$. However, Definition 3 implies $P_a(b) = P(b \mid a)$ since A is the only parent of B . Condition 5:(iii) ensures this equality.

To facilitate comparison to previous definitions, we next define the notion of *interventional invariance*:

Definition 6 (Interventional invariance). *We say that Y is interventional invariant (II) to $\mathbf{X}$ given $\mathbf{Z}$, denoted $(Y \perp_{ii} \mathbf{X} \mid \mathbf{Z})_{\mathbf{P}_*}$, if intervening on $\mathbf{X}$ does not change the interventional distribution of Y given $\mathbf{do}(\mathbf{Z} = \mathbf{z})$, i.e., $\forall \mathbf{x}, y, \mathbf{z}$, $P_{\mathbf{x}, \mathbf{z}}(y) = P_{\mathbf{z}}(y)$.*

Note that definitions 2 and 6 represent different types of causal invariance, the former claims invariance given an observation, while the latter claims invariance given an intervention. Interpreting CBNs in these terms, Definition 3 assumes modularity of each family in terms of conditional invariance (i.e., $(Y \perp_{ci} \mathbf{X} \mid \mathbf{pa}_Y)_{\mathbf{P}_*}$, $\forall \mathbf{X}$), while Definition 5 expresses the absence of causal effect in terms of interventional invariance (i.e., $(Y \perp_{ii} \mathbf{X} \mid \mathbf{pa}_Y, \mathbf{S})_{\mathbf{P}_*}$, $\forall \mathbf{S}, \mathbf{X}$).

We believe that Definition 5 is more intuitive because it relies exclusively on causal relationships in terms of which the bulk of scientific knowledge is encoded. We further discuss this intuition in the next section.

Note that conditional independence claims encoded by the CBNs are of the form $(Y \perp \mathbf{ND}_Y \mid \mathbf{pa}_Y)_{\mathbf{P}_*}$, and the II

claims are of the form $(Y \perp_{ii} \mathbf{X} \mid \mathbf{pa}_Y, \mathbf{S})_{\mathbf{P}_*}$, $\forall \mathbf{X}, \mathbf{S}$. In both cases, $\mathbf{pa}_Y$ is required to separate Y from other variables. In the observational case Y is separated from its non-descendants, while in the experimental one it is separated from all other variables. This is so because in the experimental case, an intervention on a descendant of a variable Z cannot influence Z (as is easily shown by d-separation in the mutilated graph).

A characterization based on Zero Direct Effect

The missing-link definition requires advance knowledge about parent sets, which is not necessarily available in the network construction. In this section, we extend the previous definition and propose a new characterization based on the notion of *Zero direct effect*, which is more aligned with our intuition about causal relationships, especially these emanating from typical experiments.

Definition 7 (Zero direct effect). *Let $\mathbf{X} \subseteq \mathbf{V}$, $Y \in \mathbf{V}$ and $\mathbf{S}_{\mathbf{XY}} = \mathbf{V} - \{X, Y\}$.⁴ We say that $\mathbf{X}$ has zero direct effect on Y , denoted $ZDE(\mathbf{X}, Y)$, if*

$$(Y \perp_{ii} \mathbf{X} \mid \mathbf{S}_{\mathbf{XY}})$$

Now, we introduce the definition of CBNs motivated by this notion:

Definition 8 (Zero direct effect (ZDE) causal Bayesian network). *A DAG G is ZDE compatible with a set of interventional distributions $\mathbf{P}_*$ if the following conditions hold:*

- i. [Markov] $\forall \mathbf{X} \subseteq \mathbf{V}$, $P_{\mathbf{x}}(\mathbf{v})$ is Markov relative to G ;
- ii. [ZDE] $\forall X, Y \in \mathbf{V}$, $ZDE(X, Y)$ whenever there is no arrow from X to Y in G ;
- iii. [Additivity] $\forall \mathbf{X} \subseteq \mathbf{V}$, $Z, Y \in \mathbf{V}$, $ZDE(\mathbf{X}, Y)$ and $ZDE(Z, Y) \Rightarrow ZDE(\mathbf{X} \cup \{Z\}, Y)$;
- iv. [Parents do/see] $\forall Y \in \mathbf{V}$, $\mathbf{X} \subseteq \mathbf{V}$, $P_{\mathbf{x}, \mathbf{pa}_Y}(y) = P_{\mathbf{x}}(y \mid \mathbf{pa}_Y)$ whenever $\mathbf{pa}_Y$ is consistent with $\mathbf{X} = \mathbf{x}$ and $\mathbf{X}, \{Y\}$ are disjoint.

The main feature of Definition 8 is condition (ii), which implies that varying X from x to x' while keeping all other variables constant does not change Y 's distribution – this corresponds to an ideal experiment in which all variables are kept constant and the scientist “wiggles” one variable (or set) at a time, and contemplates how the target variable reacts (i.e., *ceteris paribus*).

This condition is supplemented by condition 8:(iii), which has also an intuitive appeal: if experiments show that separate interventions on $\mathbf{X}$ and $\mathbf{Z}$ have no direct effect on Y , then a joint intervention on $\mathbf{X}$ and $\mathbf{Z}$ should also have no direct effect on Y . Conditions (i) and (iv) are the same as in the missing-link definition.

One distinct feature of this new definition emerges when we test whether a given pair $\langle G, \mathbf{P}_* \rangle$ is compatible. First, the modularity condition of Definition 3 requires that each family is invariant to interventions on all subsets of elements “outside” the family, which is combinatorially explosive. In contrast, condition (ii) of Definition 8 involves singleton pairwise experiments which are easier to envision and evaluate.

⁴We use $\{\mathbf{A}, \mathbf{B}\}$ to denote the union of $\mathbf{A}$ and $\mathbf{B}$.

Put another way, when the ZDE condition does not hold, it implies the existence of an edge between the respective pair of nodes thus providing fewer and easier experiments in testing the structure of the graph. Further, one should test the Markov compatibility of P and the new induced graph G .

We now show that all three local definitions of causal Bayesian networks stated so far are equivalent.

Theorem 2. *Let G be a DAG and $\mathbf{P}_*$ a set of interventional distributions, the following statements are equivalent:*

- i. G is locally compatible with $\mathbf{P}_*$
- ii. G is missing-link compatible with $\mathbf{P}_*$
- iii. G is ZDE compatible with $\mathbf{P}_*$

Note that the notion of “parents set”, though less attached to modularity and invariance, is still invoked by the last two compatibility conditions. We believe therefore that it is an essential conceptual element in the definition of causal Bayesian networks.

5 Equivalence between the local definitions of causal Bayesian network

Definition 9 (Strong Markov Condition). *Each variable is interventionally independent of every other variable after fixing its parents. That is, for all $Y \in \mathbf{V}$ and $\mathbf{X} \subseteq \mathbf{V} - \{Y, \mathbf{PA}_Y\}$ we have*

$$P_{\mathbf{x}, \mathbf{pa}_Y}(y) = P_{\mathbf{pa}_Y}(y), \text{ for all } \mathbf{x}, y, \mathbf{pa}_Y \quad (9)$$

5.1 [Zde-CBN] $\Rightarrow$ [local-CBN]

In this subsection, we assume that the four conditions in the definition of the Zero direct effect causal Bayesian network are valid for a given graph G and set $\mathbf{P}_*$.

The first result simply extends the Zero direct effect semantics to subset of variables:

Lemma 1. *Zde($\mathbf{W}, Y$) holds for every $\mathbf{W} \subseteq \mathbf{V} - \{Y, \mathbf{PA}_Y\}$.*

Proof. Note that $\mathbf{W}$ does not contain parents of Y . Then, [Zde] gives that, for every U in $\mathbf{W}$, we have $Zde(U, Y)$. But then, it follows directly by [Additivity], that $Zde(\mathbf{W}, Y)$ holds. $\square$

The next Lemma shows that the strong Markov condition is also valid for G and $\mathbf{P}_*$.

Lemma 2. *For all $Y \in \mathbf{V}$ and $\mathbf{X} \subset \mathbf{V} - \{Y, \mathbf{PA}_Y\}$, the relation $(Y \perp_{ii} \mathbf{X} \mid \mathbf{PA}_Y)$ holds.*

Proof. Let $\mathbf{T}_1 = \mathbf{V} - \{Y, \mathbf{PA}_Y\}$, and note that $S_{Y\mathbf{T}_1} = \mathbf{PA}_Y$. Since $\mathbf{T}_1$ does not have parents of Y , by Lemma 1, we have $Zde(\mathbf{T}_1, Y)$, that is

$$P_{\mathbf{t}_1, s_{y\mathbf{t}_1}}(y) = P_{s_{y\mathbf{t}_1}}(y) = P_{\mathbf{pa}_Y}(y)$$

Now, let $\mathbf{T}_2 = \mathbf{V} - \{Y, \mathbf{X}, \mathbf{PA}_Y\}$, and note that $S_{Y\mathbf{T}_2} = \{\mathbf{X}, \mathbf{PA}_Y\}$. Since $\mathbf{T}_2$ does not have parents of Y , by Lemma 1, we have $Zde(\mathbf{T}_2, Y)$, that is

$$P_{\mathbf{t}_2, s_{y\mathbf{t}_2}}(y) = P_{s_{y\mathbf{t}_2}}(y) = P_{\mathbf{x}, \mathbf{pa}_Y}(y)$$

Since $(\mathbf{T}_1 \cup S_{Y\mathbf{T}_1}) = (\mathbf{T}_2 \cup S_{Y\mathbf{T}_2})$, we obtain

$$P_{\mathbf{x}, \mathbf{pa}_Y}(y) = P_{\mathbf{pa}_Y}(y)$$

$\square$

Lemma 3. *The condition of [Modularity] is valid for G and $\mathbf{P}_*$.*

Proof. Fix a variable Y and $\mathbf{X} \subset \mathbf{V} - \{Y\}$. We need to show that

$$P_{\mathbf{x}}(y \mid \mathbf{pa}_Y) = P(y \mid \mathbf{pa}_Y)$$

Applying the condition [Parents do/see] to both sides in the equation above, we obtain

$$P_{\mathbf{x}, \mathbf{pa}_Y}(y) = P_{\mathbf{pa}_Y}(y)$$

and we immediately recognize here a claim of the strong Markov condition. $\square$

Finally, the observation that the condition [Markov] is present in both definitions, we complete the proof that G is a local causal Bayesian network for $\mathbf{P}_*$.

5.2 [local-CBN] $\Rightarrow$ [Zde-CBN]

In this subsection, we assume that the two conditions in the definition of the local causal Bayesian network are valid for a given graph G and set $\mathbf{P}_*$.

Lemma 4. *For all $Y \in \mathbf{V}$ and $\mathbf{X} \subset \mathbf{V} - \{Y, \mathbf{PA}_Y\}$ we have*

$$P_{\mathbf{x}, \mathbf{pa}_Y}(\mathbf{pa}_Y \mid y) = 1$$

whenever $P_{\mathbf{x}, \mathbf{pa}_Y}(y) > 0$, and $\mathbf{pa}_Y$ is compatible with $\mathbf{x}$.

Proof. This is an immediate consequence of the property of [Effectiveness], in the definition of $\mathbf{P}_*$. $\square$

Lemma 5. *The condition [Parents do/see] is valid for G and $\mathbf{P}_*$.*

Proof. Fix a variable $\mathbf{X} \subset \mathbf{V}$ and consider an arbitrary instantiation $\mathbf{v}$ of variables $\mathbf{V}$, and $\mathbf{pa}_Y$ consistent with $\mathbf{x}$.

Consider the intervention $do(\mathbf{X} = \mathbf{x})$, and given the condition [Modularity], $P_{\mathbf{x}}(y \mid \mathbf{pa}_Y) = P(y \mid \mathbf{pa}_Y)$, $Y \notin \mathbf{X}$. Now consider the intervention $do(\mathbf{X} = \mathbf{x}, \mathbf{PA}_Y = \mathbf{pa}_Y)$, and again by the condition [Modularity] $P_{\mathbf{x}, \mathbf{pa}_Y}(y \mid \mathbf{pa}_Y) = P(y \mid \mathbf{pa}_Y)$. The r.h.s. coincide, therefore

$$\begin{aligned} P_{\mathbf{x}}(y \mid \mathbf{pa}_Y) &= P_{\mathbf{x}, \mathbf{pa}_Y}(y \mid \mathbf{pa}_Y) \\ &\stackrel{\text{Bayes thm.}}{=} \frac{P_{\mathbf{x}, \mathbf{pa}_Y}(\mathbf{pa}_Y \mid y) P_{\mathbf{x}, \mathbf{pa}_Y}(y)}{P_{\mathbf{x}, \mathbf{pa}_Y}(\mathbf{pa}_Y)} \\ &\stackrel{\text{effectiveness}}{=} P_{\mathbf{x}, \mathbf{pa}_Y}(\mathbf{pa}_Y \mid y) P_{\mathbf{x}, \mathbf{pa}_Y}(y) \end{aligned} \quad (10)$$

We consider two cases. If $P_{\mathbf{x}, \mathbf{pa}_Y}(y) > 0$, by lemma 4 $P_{\mathbf{x}, \mathbf{pa}_Y}(\mathbf{pa}_Y \mid y) = 1$, and then substituting back in eq. (10) we obtain $P_{\mathbf{x}}(y \mid \mathbf{pa}_Y) = P_{\mathbf{x}, \mathbf{pa}_Y}(y)$. If $P_{\mathbf{x}, \mathbf{pa}_Y}(y) = 0$, substituting back in eq. (10) we obtain $P_{\mathbf{x}}(y \mid \mathbf{pa}_Y) = P_{\mathbf{x}, \mathbf{pa}_Y}(\mathbf{pa}_Y \mid y) * 0 = 0$, and then $P_{\mathbf{x}}(y \mid \mathbf{pa}_Y) = P_{\mathbf{x}, \mathbf{pa}_Y}(y)$. $\square$

Lemma 6. *The condition [Zde] is valid for G and $\mathbf{P}_*$.*

Proof. Fix $Y, X \in \mathbf{V}$ such that there is no arrow pointing from X to Y . Let $\mathbf{S}_{XY} = \mathbf{V} - \{X, Y\}$. We want to show

$$P_{x, \mathbf{S}_{XY}}(y) = P_{\mathbf{S}_{XY}}(y), \text{ for all } x, y, \mathbf{S}_{XY}$$

Note that $\mathbf{PA}_Y \subseteq \mathbf{S}_{XY}$, and then by the [Parent do/see] condition we have to show

$$P_{x, \mathbf{S}'_{XY}}(y | \mathbf{pa}_Y) = P_{\mathbf{S}'_{XY}}(y | \mathbf{pa}_Y)$$

where $\mathbf{S}'_{XY} = \mathbf{S}_{XY} - \{\mathbf{PA}_Y\}$.

The condition [Modularity] implies that $P_{x, \mathbf{S}'_{XY}}(y | \mathbf{pa}_Y) = P(y | \mathbf{pa}_Y)$. Again by [Modularity], we obtain $P(y | \mathbf{pa}_Y) = P_{\mathbf{S}_{XY}}(y | \mathbf{pa}_Y)$. Applying [Parents do/see], [Zde] follows. $\square$

Lemma 7. *The condition [Additivity] is valid for G and $\mathbf{P}_*$.*

Proof. Fix $\mathbf{X} \subset \mathbf{V}$ and $Z, Y \in \mathbf{V}$. Let $\mathbf{S}_{xzy} = \mathbf{V} - \{\mathbf{X}, Y, Z\}$. Assume $Zde(\mathbf{X}, Y)$ and $Zde(Z, Y)$. For the sake of contradiction, suppose that $Zde(\mathbf{X} \cup \{Z\}, Y)$ is false.

We can rewrite it based on the law of total probability,

$$\sum_{\mathbf{pa}_Y} P_{\{\mathbf{x}, z\}, \mathbf{S}_{xzy}}(y | \mathbf{pa}_Y) P_{\{\mathbf{x}, z\}, \mathbf{S}_{xzy}}(\mathbf{pa}_Y) \neq \sum_{\mathbf{pa}_Y} P_{\mathbf{S}_{xzy}}(y | \mathbf{pa}_Y) P_{\mathbf{S}_{xzy}}(\mathbf{pa}_Y)$$

Notice that there is only one configuration of $\mathbf{pa}_Y$ consistent with $\mathbf{S}_{xzy}$ in both sides because $\mathbf{PA}_Y \subseteq \mathbf{S}_{xzy}$ and [Effectiveness]. Then, this equation reduces to

$$P_{\{\mathbf{x}, z\}, \mathbf{S}_{xzy}}(y | \mathbf{pa}_Y) \neq P_{\mathbf{S}_{xzy}}(y | \mathbf{pa}_Y)$$

We reach a contradiction given [Modularity]. $\square$

The proof for the Missing-link CBN is analogous.

6 Conclusions

We first proved the equivalence between two characterizations of Causal Bayesian Networks, one local, based on modularity, and the other global, based on the truncated product formula. We then introduced two alternative characterizations of CBNs, proved their equivalence with the previous ones, and showed that some of their features make the tasks of empirically testing the network structure, as well as judgmentally assessing its plausibility more manageable.

Another way to look at the results of our analysis is in terms of the information content of CBNs, that is, what constraints a given CBN imposes on both observational and experimental findings. For a probabilistic Bayes network the answer is simple and is given by the set of conditional independencies that are imposed by the d-separation criterion. For a CBN, the truncated product formula (2) imposes conditional independencies on any interventional distribution $P_x(\mathbf{v})$. But this does not sum up the entire information content of a CBN. Equation (2) further tells us that the relationship between any two interventional distributions, say $P_x(\mathbf{v})$ and $P_{x'}(\mathbf{v})$, is not entirely arbitrary; the two distributions constrain each other in various ways. For example, the conditional distributions

$P_x(v_i | \mathbf{pa}_i)$ and $P_{x'}(v_i | \mathbf{pa}_i)$ must be the same for any unmanipulated family. Or, as another example, for any CBN we have the inequality: $P_x(y) > P(x, y)$ [Tian *et al.*, 2006].

A natural question to ask is whether there exists a representation that encodes all constraints of a given type. The modularity property of Definition 2 constitutes such a representation, and so do the missing-link and the ZDE definitions. Each encodes constraints of a given type and our equivalence theorems imply that all constraints encoded by one representation can be reconstructed from the other representation without loss of information.

References

- A. P. Dawid. Influence diagrams for causal modelling and inference. *International Statistical Review*, 70(2):161–189, 2001.
- D. Galles and J. Pearl. An axiomatic characterization of causal counterfactuals. *Foundation of Science*, 3(1):151–182, 1998.
- J.Y. Halpern. Axiomatizing causal reasoning. In G.F. Cooper and S. Moral, editors, *Uncertainty in Artificial Intelligence*, pages 202–210. Morgan Kaufmann, San Francisco, CA, 1998. Also, *Journal of Artificial Intelligence Research* 12:3, 17–37, 2000.
- D. Heckerman and R. Shachter. Decision-theoretic foundations for causal reasoning. *Journal of Artificial Intelligence Research*, 3:405–430, 1995.
- D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.
- S. L. Lauritzen. Causal inference from graphical models. In *Complex Stochastic Systems*, pages 63–107. Chapman and Hall/CRC Press, 1999.
- J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Mateo, CA, 1988.
- J. Pearl. Belief networks revisited. *Artificial Intelligence*, 59:49–56, 1993.
- J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, New York, 2000. Second ed., 2009.
- J.M. Robins. A new approach to causal inference in mortality studies with a sustained exposure period – applications to control of the healthy workers survivor effect. *Mathematical Modeling*, 7:1393–1512, 1986.
- P. Spirtes, C.N. Glymour, and R. Scheines. *Causation, Prediction, and Search*. Springer-Verlag, New York, 1993.
- J. Tian and J. Pearl. A new characterization of the experimental implications of causal Bayesian networks. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 574–579. AAAI Press/The MIT Press, Menlo Park, CA, 2002.
- J. Tian, C. Kang, and J. Pearl. A characterization of interventional distributions in semi-Markovian causal models. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1239–1244. AAAI Press, Menlo Park, CA, 2006.

Boolean formulas of simple conceptual graphs (*SGBF*)

Olivier Carloni

Mondeca,

3 cité Nollez, F-75018 Paris - France

<http://www.mondeca.com>

olivier.carloni@mondeca.com

Abstract

This paper presents a conceptual graph formalism called *simple graph boolean formulas* which extends the *SG*-family with boolean connectors. This formalism is used as a query language in a classification service for legal content management. We present the functional architecture of the service, then we define the *SGBF*-model of graph boolean formulas and we conclude by discussing decidability issues for this formalism.

1 Introduction

Mondeca is a software publisher that is developing **Intelligent Topic Manager** (ITM), a knowledge management tool providing several services as: indexing of documents from the thesaurus, creation of annotations controlled by a domain ontology, semi-automatic building of document annotations, semantic navigation and search facilities through the annotation base. The rising number of customers in various fields (as tourism, terminological management, military, etc) requesting knowledge quality control, content enrichment or classification features led Mondeca to provide ITM with several reasoning services. As the knowledge representation formalism in ITM is quite similar to labeled graphs, Mondeca chose the conceptual graph formalism [Sowa, 1984] and especially the *SG*-family [Baget and Mugnier, 2002] to define these reasoning services. The *SG*-family provides ITM with a formal semantics in logic, inference rules and constraints, graph-based reasoning operations sound and complete with respect to the logical deduction, and combinatorial-based efficient algorithms for these graph operations. The reader is referred to [Carloni *et al.*, 2007] for a complete description of ITM, its knowledge representation formalism and its relationships with the *SG*-family.

One feature usually requested by customers is the ability to *classify* knowledge in hierarchically organized categories. This is particularly needed by law publishers, in the field of legal content management. An increasing number of publishers need a tool to provide management facilities to three kinds of user: (1) the end-user, (2) the lawmaker and (3) the publisher itself.

- The end-user refers to the person required to respect a given law. For example, when dealing with the SOLAS

naval law the end-users are the shippers whose fleet must be compliant with the SOLAS regulations. The end-user needs to know which regulations are *applicable* to his current *situation*.

- The lawmaker is the person which produces or updates the law. He needs a tool to manage, search among regulations and compare them together. If a regulation *R* has to be added to the law, he has to evaluate the general impact on the full law content. More precisely, he has to find among the existing regulations:

- (1) the regulations similar to *R*: these regulations and *R* are always applicable to the same situations
- (2) the useless regulations if *R* is added: these regulations are not applicable anymore on any situations
- (3) the regulations applicable each time *R* is applicable,
- (4) and the regulations which applicability lead to the application of *R*.

- On the publisher side, employees need to know which end-user situations are concerned by a given regulation for marketing prospection.

Thus, the tool requested by publishers has to provide (1) a law triggering service to alert the end-user when a regulation is applicable to his situation ; (2) a regulation management service to the lawmaker ; and (3) a search feature for the publisher to know which end-user situations are concerned by a given regulation.

Mondeca designed a classification service based on ITM knowledge base. The figure 1 shows the functional architecture of the service. The knowledge base represented as a box in the center is divided in three parts: (1) the *law hierarchy* is stored on the publisher side, (2) the law editing environment on the lawmaker side and (3) the personal situations in the end-user side. The textual content of the law is not stored in the base. Only chapters and their organisation are represented in a *tree*-structured hierarchy of *regulation objects* providing a link to the real content stored outside ITM.

On each regulation *R*, the lawmaker is expected (1) to define a *query* $Q(R)$ which selects each situation *matching* *R* and (2) if needed, to flag *R* with an *excluded status*. The semantics of the regulations hierarchy is as follows: for each *non-excluded* regulation R_c located below another *non-excluded* regulation R_p , if a situation *matches* R_c ,

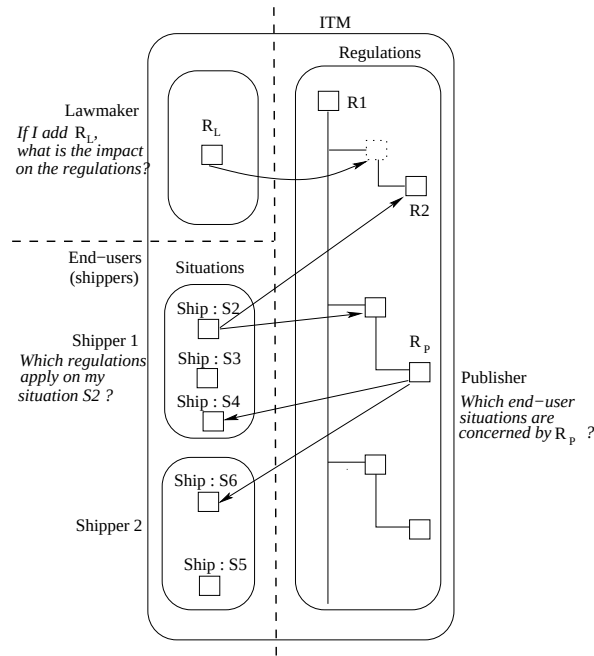


Figure 1: The standard classification architecture for legal content management.

this situation must also *match* R_p . The regulations hierarchy provides an inheritance mechanism which respects this semantics and generates for each non-excluded regulation R , a query $Q_H(R)$ called *inheritance query* combining all criteria defined in $Q(R)$ and the queries of the non-excluded regulations located above in the hierarchy. More precisely, the inheritance query of a non-excluded regulation R is defined as the conjunction of its query $Q(R)$ and the inheritance query $Q_H(R_p)$ of its least non-excluded ancestor R_p . If needed, a regulation R_E may be flagged as *excluded* in order to consider it as being out of the hierarchy. In such a case, R_E is ignored by the inheritance mechanism : it does not inherit from its parent and does not transfer its query content to its children (i.e. $Q_H(R_E) = Q(R_E)$). A regulation R is said to be *applicable* to an end-user situation S iff S answers the query $Q_H(R)$.

Situations are stored in ITM and may be defined on the fly by end-users. The classification service is configured to trigger the end-user when a new regulation R is applicable to his situation (i.e. when the query $Q_H(R)$ matches his situation). On the publisher side, when the publisher needs to know which end-user situations are concerned by a given regulation R , the system performs the query $Q_H(R)$ on the situations and returns the results.

While the problems of the end-users or the publisher may be solved by standard querying, the problem of the lawmaker needs higher level reasoning mechanisms. As the lawmaker needs to compare regulations, the system proceeds by comparing the queries associated to the regulations. For two given regulations R_1 and R_2 :

- (1) R_1 and R_2 are said to be *similar* iff $Q_H(R_1)$ and

$Q_H(R_2)$ are *semantically equivalent* (i.e. for any knowledge in the base, $Q_H(R_1)$ and $Q_H(R_2)$ have same results)

- (2) R_1 is *useless* iff $Q_H(R_1)$ is *inconsistent* (i.e. for any knowledge in the base, $Q_H(R_1)$ has no result)

- (3 and 4) if R_1 is applicable then R_2 is applicable iff $Q_H(R_1)$ is *more specific* than $Q_H(R_2)$ (i.e. for any knowledge in the base, a result of $Q_H(R_1)$ is a result of $Q_H(R_2)$)

Queries associated to regulations may be conjunctive/disjunctive combinations of positive/negative criterias (a criteria may be another query). For example, a typical query is “all ships with electric engine and cooling system or provided with two twined engines with same cooling system but no adjacent tanks”. A formalism has been precisely designed to meet this needed high expressivity while staying the closest possible to the “graph” spirit of the *SG*-family and the ITM tool. This paper is devoted to this formalism, named *SGBF*-model, which stands for *Simple Graph Boolean Formulas*. In the first section we present basic definitions of the standard *SG*-formalism (support and SG) and its relationships with *FOL*. The second section defines the *SGBF*-model and its logical semantics. The third section presents decidability issues for the *SGBF*-model. The paper finish by discussing the integration of the *SGBF*-model in the framework of the classification service.

2 The *SG*-model of conceptual graphs

The *SG*-family defined in [Baget and Mugnier, 2002] is a hierarchy of conceptual graph formalisms organized in layers ranging from the less to the more expressive. The *SG*-model is the bottom layer of this hierarchy and provides the elementary components and operations used in upper layers to define more complex structures and operations (as rules and constraints with their respective inference and validation methods). These components are the *support* and the *simple graph* (SG). A SG is a bipartite graph representing entities of the domain and their shared relations respectively as *concept nodes* edge-connected to *relation nodes*. Each node has a type defined in a structure called the *support* which is a rudimentary ontology. The support and SG definitions may vary through the litterature. The reader is referred to [Chein and Mugnier, 1992] about simple conceptual graphs and [Baget and Mugnier, 2002] for a study of the whole *SG*-family.

2.1 Support

A support is a triple $S = (T_C, T_R, \mathcal{I})$. T_C and T_R are two finite sets respectively of concept types and relation types partially ordered by a relation $\leq_t$ where $x \leq_t y$ means that x is a subtype of y . T_R is partitionned into subsets $T_R^1, T_R^2, \dots, T_R^k$ of relation types of arity 1...k respectively ($k \leq 1$). $\mathcal{I}$ is the set of individual markers. We denote by $*$ the generic marker, where $* \notin \mathcal{I}$ and by $\mathcal{M}$ the set of all markers $\mathcal{I} \cup \{*\}$. A partial order on $\mathcal{M}$ considers elements of $\mathcal{I}$ as pairwise incomparable, and $*$ as its greatest element.

The figure 2 shows a simple support with naval vocabulary. It is said that *tanker* is more specific than *ship* and that, if two entities are *adjacent* then they are *close*.

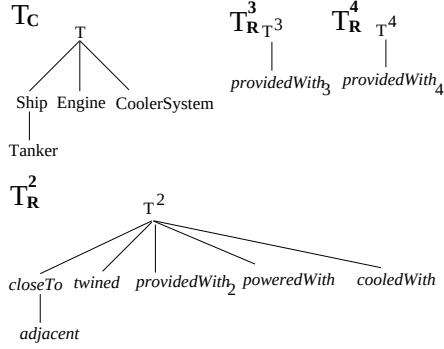


Figure 2: A support with a concept type hierarchy T_C and a binary relation type hierarchy T_R^2 .

2.2 Simple graph

A SG G , defined over a support S is a finite bipartite graph (C, R, E, λ) . Where C and R are the node sets respectively of concept nodes and relation nodes. E is a finite set of labelled edges (r, i, c) where $c \in C$, $r \in R$ and $i \in \mathbb{N}$ is the number of the edge. Edges (r, i, c) incident to a relation node are totally ordered with i ranging from 1 to the degree of the node and for $r \in R$ and $c \in C$ there is at most one $(r, i, c) \in E$.

Each node has a label given by the mapping λ . A relation node r is labelled by $type(r)$ an element of T_R called its type and the degree of r must be equal to the arity of $type(r)$. A concept node c is labelled by a pair $(type(c), marker(c))$ where $type(c)$ is an element of T_C called its type and $marker(c)$ is an element of $\mathcal{M}$ called its marker. The concept is said to be *generic* if $marker(c) = *$; and *individual* otherwise.

The set of node labels is partially ordered by a relation $\leq_\lambda$ such that for two nodes n_1 and n_2 $\lambda(n_1) \leq_\lambda \lambda(n_2)$ iff $type(n_1) \leq_t type(n_2)$ and, when n_1, n_2 are concepts, $marker(n_1)$ must be in addition less or equal than $marker(n_2)$.

The *merge* of two disjoint SGs G and H is the SG obtained by joining the node sets, the edge set and the labelling mapping of G with their homologous in H .

The figure 3 shows three SGs G_1 , G_2 and G_3 meaning respectively that (1) ship x has an electric engine and an electric cooler system, (2) ship x is provided with a cooling system and two twined engines using this cooling system and (3) tanker x has two adjacent tanks. In these figures, the symbol x denotes an individual marker (not a variable).

2.3 Semantics

In the following, the term *logical formula* refers to a $\mathcal{FOL}$ formula where $\mathcal{FOL}$ refers to the classical first order logic without functions, with connectors $\{\wedge, \vee, \rightarrow, \neg\}$ and quantifiers $\{\forall, \exists\}$.

The mapping Φ maps a support as a set of universally quantified logical formulas and SGs to the existential and positive fragment $\mathcal{FOL}(\wedge, \exists)$ of $\mathcal{FOL}$. Given a support S , a constant is assigned to each individual of $\mathcal{I}$. For each k-ary relation

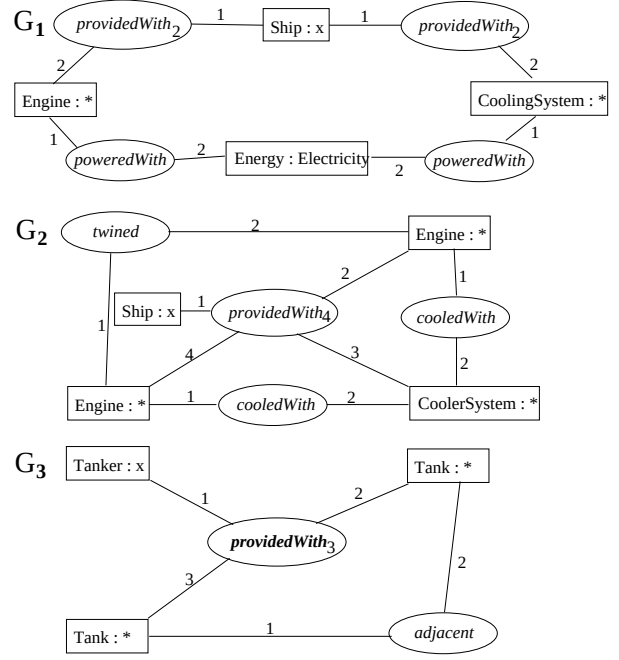


Figure 3: Three SGs.

type, is assigned one k-ary predicate and for each concept type one unary predicate. We consider that each constant or predicate names are identical to their corresponding element in $\mathcal{SG}$ -model.

The set of formulas $\Phi(S)$ assigned to any support S contains all formulas $\forall x_1 \dots x_k (t_1(x_1, \dots, x_k) \rightarrow t_2(x_1, \dots, x_k))$ such that $t_1 \leq_t t_2$ with $k = 1$ when $t_1, t_2 \in T_C$; and k denotes the arity of the relation type when $t_1, t_2 \in T_R$.

For example, the translation of the support shown in figure 2 returns :

- $\forall x (Tanker(x) \rightarrow Ship(x))$
- $\forall x \forall y (adjacent(x, y) \rightarrow closeTo(x, y))$

Given a SG G , in order to build the formula $\Phi(G)$ a term $term(c)$ is assigned to each concept c such that $term(c)$ is the constant corresponding to its marker or a distinct variable if the marker is generic. Let $\Phi^\wedge(G)$ be the conjunction of

- all predicates $p_c(term(c))$ such that $c \in C$ and p_c is the unary predicate assigned to $type(c)$
- and all predicates $p_r(term(c_1), term(c_2), \dots, term(c_k))$ such that $r \in R$, p_r is the k-ary predicate assigned to $type(r)$ and c_i is the concept from $C(G)$ linked to r by an i -numbered edge from $E(G)$

$\Phi(G)$ is the formula $\exists \tilde{x} \Phi^\wedge(G)$ where $\tilde{x}$ is the set of variables appearing in $\Phi^\wedge(G)$.

The SG G_1 shown in figure 3 have the following $\mathcal{FOL}$ semantics, where X denotes here a constant: $\Phi(G_1) = (\exists y \exists z Ship(X) \wedge Engine(y) \wedge CoolingSystem(z) \wedge Energy(Electricity) \wedge providedWith_2(X, y) \wedge providedWith_2(X, z) \wedge poweredWith(y, Electricity) \wedge poweredWith(z, Electricity))$.

2.4 Projection

The projection is the elementary reasoning operation of the SG -family from which other reasoning mechanisms are built. This operation is a kind of homomorphism that preserves the partial order defined on labels. Projection has been proven sound and complete with regard to the semantics Φ in [Chein and Mugnier, 1992]. The projection operation is not presented in this paper which mainly focuses on relationships between $SGBF$ -model and FOL .

3 The $SGBF$ -model of conceptual graphs

The $SGBF$ -model provides the expressivity to combine SGs with boolean connectors into expressions called *graph formulas*.

3.1 Boolean formulas of simple graphs

Introduction of variables

We need to extend the definition of a SG in order to express that two generic concepts from two distinct SGs in a formula represent the same entity. This may be done by using variable symbols. Given a set of variable symbols $\mathcal{V}$ disjoint from $\mathcal{I} \cup \{*\}$ and a SG (C, R, E, λ) , the previous definition of *marker* is now replaced by a total function from C into the set $\mathcal{M} = \mathcal{I} \cup \mathcal{V} \cup \{*\}$ which partial order still considers elements from $\mathcal{I}$ as pairwise incomparable ; and elements from $\mathcal{V} \cup \{*\}$ as pairwise equivalent and greater or equal than those from $\mathcal{I}$. Thus, *marker* assigns to each concept of a graph either a variable, an individual, or a generic marker.

The notion of variable is taken into account in the logical semantics by updating the *term* function in order to assign to a given variable from $\mathcal{V}$ a corresponding FOL variable with the same name.

In the concept nodes $\boxed{Ship : x}$ and $\boxed{Tanker : x}$ appearing in the graphs of figure 3, the symbol x denotes a variable and not an individual marker anymore.

Definition

Given a support s , the language $SGBF(s)$ of *graph formulas* over s is defined by the following rules :

- $SGBF(s)$ contains all SGs definable over the support s
- if A and B are in $SGBF(s)$, then $A \bullet B$ is in $SGBF(s)$ with $\bullet \in \{\wedge, \vee\}$
- if A is in $SGBF(s)$, then $\neg A$ is in $SGBF(s)$

Given a formula $f \in SGBF(s)$, the function $vars : SGBF(s) \rightarrow 2^{\mathcal{V}}$ returns the set of variables involved in the graphs appearing in f .

$$Q(R_1) = G_1 \vee G_2$$

where G_1 and G_2 are the graphs from figure 3.

Figure 4: The graph formula $Q(R_1)$ for the R_1 regulation of the figure 1.

Figures 4 and 5 show the two queries $Q(R_1)$ and $Q(R_2)$ respectively associated to the R_1 regulation and its child R_2 in the hierarchy example shown at figure 1. Each query is

$$Q(R_2) = \boxed{Ship : x} \wedge \neg G_3$$

where G_3 is the graph from figure 3.

Figure 5: The graph formula $Q(R_2)$ for the R_2 regulation of the figure 1.

a boolean combination of graphs from figure 3. $Q(R_1)$ is the disjunction of graphs G_1 and G_2 which stands for all “ships x having either an electric engine or a cooling system and two independent engines using the same cooling system”. $Q(R_2)$ is the conjunction of the graph G_3 and the graph with one concept $\boxed{Ship : x}$. It represents all “ships x which are not a tanker provided with adjacent pair of tanks”. When the inheritance mechanism is applied to the regulation hierarchy, it generates for R_2 the *inheritance query* $Q_H(R_2) = Q(R_1) \wedge Q(R_2)$.

3.2 Relationships with FOL

The next subsections present the $SGBF$ -model logical semantics and the translation from FOL to $SGBF$ -model.

From $SGBF$ -model to FOL

Given a support s , an expression f from $SGBF(s)$ and a subset X of $\mathcal{V}$, the function $g2f$ translates f to the logical formula $g2f(f, X)$ by the following rules :

- if $f = A \bullet B$ then $g2f(f, X) = \exists \tilde{x} g2f(A, X \cup \tilde{x}) \bullet g2f(B, X \cup \tilde{x})$ with $\bullet \in \{\wedge, \vee\}$ and $\tilde{x} = vars(A) \cap vars(B) \setminus X$
- if $f = \neg A$ then $g2f(f, X) = \neg g2f(A, X)$
- if f is a SG^1 then $g2f(f, X) = \exists \tilde{x} \gamma$ where $\tilde{x} = free(\gamma) \setminus X$ contains new variable symbols and $\gamma = \Phi^{\wedge}(f)$ given that for any logical formula α , the function *free* returns the set $free(\alpha)$ of *free* variable symbols in α .

The semantics of a graph formula f from $SGBF(s)$ in FOL is given by the unary function $g2f$ such that $g2f(f) = g2f(f, \emptyset)$.

The FOL semantics for the graph formulas $Q(R_1)$ and $Q(R_2)$ shown in figures 4 and 5 is respectively:

- $g2f(Q_H(R_1)) = \exists x Ship(x) \wedge \neg(\exists y \exists z Tanker(x) \wedge Tank(y) \wedge Tank(z) \wedge providedWith_3(x, y, z) \wedge adjacent(y, z))$
- $g2f(Q_H(R_2)) = (\exists x (\exists y \exists z Ship(x) \wedge Engine(y) \wedge CoolingSystem(z) \wedge Energy(Electricity) \wedge providedWith_2(x, y) \wedge providedWith_2(x, z) \wedge poweredWith(y, Electricity) \wedge poweredWith(z, Electricity)) \vee (\exists p \exists q \exists r Ship(x) \wedge Engine(p) \wedge Engine(q) \wedge CoolingSystem(r) \wedge providedWith_4(x, p, q, r) \wedge twined(p, q) \wedge cooledBy(p, r) \wedge cooledBy(q, r)))$

¹if f is empty, then $g2f(f, X) = \top$ by convention

)

It should be noted that since the $\mathcal{SGBF}$ -model is based on the $\mathcal{SG}$ -model, a SG from $\mathcal{SG}$ -model or $\mathcal{SGBF}$ -model has the same logical semantics.

From $\mathcal{FOL}$ to $\mathcal{SGBF}$ -model

This section presents the mapping $f2g$ from $\mathcal{FOL}$ into $\mathcal{SGBF}$ -model. Given a logical formula f , $f2g(f)$ returns the corresponding graph formula by applying the following rules. Let f^r be the variable renaming of f such that two variables appearing respectively in two distinct quantifiers have different names :

- if $f^r = A \rightarrow B$ then $f2g(f) = f2g(\neg A \vee B)$
- if $f^r = A \bullet B$ then $f2g(f) = f2g(A) \bullet f2g(B)$ with $\bullet \in \{\wedge, \vee\}$
- if $f^r = \neg A$ then $f2g(f) = \neg f2g(A)$
- if $f^r = \forall x A$ then $f2g(f) = f2g(\neg \exists x \neg A)$
- if $f^r = \exists x A$ then $f2g(f) = f2g(A) \wedge (top(x) \vee \neg top(x))$ where $top(x)$ is the SG with only one concept node $\boxed{\top : x}$
- if f^r is a predicate $p(t_1, \dots, t_n)$ then $f2g(f)$ is a SG with one concept $c_i = \boxed{\top : m_i}$ for each term t_i and one relation node r of type p linked to each c_i by an edge (r, i, c_i) . The marker m_i is the individual or variable from $\mathcal{I} \cup \mathcal{V}$ respectively corresponding to the i -th constant or variable t_i in p .

The *fifth* translation step (of the $\exists$ quantifier) adds to the resulting conjunction a graph formula $N = (top(x) \vee \neg top(x))$. This step is necessary to guarantee that every logical formula may be translated into a graph formula. Indeed, if we remove N from this step, the translation rule would be “if $f = \exists x A$ then $f2g(f) = f2g(A)$ ”, and some non-equivalent formulas as $\exists x \neg p(x)$ and $\neg \exists x p(x)$ would have the same corresponding graph formula $\neg f2g(p(x))$.

Logical entailment and equivalence

Given a support s , we say that a graph formula f is *logically satisfiable* (resp. *logically unsatisfiable*) iff the conjunction $g2f(f) \wedge \phi(s)$ of the logical translation of f and the support s is *satisfiable* (resp. *unsatisfiable*). A graph formula f *logically entails* an other graph formula g iff $g2f(f) \wedge \neg g2f(g)$ is logically unsatisfiable. Two graph formulas f and g are *logically equivalent* iff each one *logically entails* the other.

3.3 Language Properties

Algebraic laws

The disjunctive and conjunctive connectors $\{\vee, \wedge\}$ provided by the $\mathcal{SGBF}$ -model respect the usual laws of associativity, commutativity and distributivity. However connectors are no longer idempotent while using negations. For example, a graph formula $\neg A \wedge \neg A$ is not *logically equivalent* to $\neg A$. Indeed $g2f(\neg A \wedge \neg A)$ returns $\exists \tilde{x} \neg g2f(A) \wedge \neg g2f(A)$ with $\tilde{x} = vars(A)$ but not $\neg g2f(A)$ (which is the logical translation of $\neg A$). In the other hand, a graph formula $\neg A \wedge \neg A \wedge \dots \wedge \neg A$ is *logically equivalent* to $\neg A \wedge \neg A$. However, if the same repeated formula A does not contain

any negation, $A \wedge A \wedge \dots \wedge A$ is *logically equivalent* to A (the same holds for $\vee$).

Expressivity

$\mathcal{FOL}$ and $\mathcal{SGBF}$ -model have the same expressive power. Without loss of information: each graph formula can be translated into a logical formula and each logical formula may be translated into a graph formula. Given a logical formula f , $g2f(f2g(f))$ may return a formula different from f . However f and $g2f(f2g(f))$ are always semantically equivalent. Since $f2g$ adds expressions of form $(top(x) \vee \neg top(x))$ at the $\exists$ -step of the translation, expressions of the same form may also appear in the logical formula $g2f(f2g(f))$. Such expressions do not change the semantics of the source formula f as they are always true and appear in a conjunction (a logical formula A being always equivalent to $A \wedge (top(x) \vee \neg top(x))$ whatever the quantification of x).

Universal quantification may be expressed in a graph formula by replacing the universal quantifier by a negated existentially quantified negation as shown in section 3.2. Given two logical formulas A and B , the universal quantified implication $\forall \tilde{x} (A \rightarrow B)$ where $\tilde{x} = free(A) = free(B)$ may be rewritten as $\neg \exists \tilde{x} (A \wedge \neg B)$ and can be translated to a graph formula $\neg (f2g(A) \wedge \neg f2g(B))$. A less trivial case is the formula $\forall x p(x)$, whose corresponding graph formula is $\neg ((top(x) \vee \neg top(x)) \wedge \neg p(x))$ ($top(x)$ is defined in section 3.2).

Relationships with conceptual graph propositions

A *graph formula* is very close to a *conceptual graph proposition* as introduced in [Baader et al., 1998]. A conceptual graph proposition is defined as either a negated graph proposition or a box containing a SG and finitely many graph propositions. The language of a graph proposition is generated by the grammar $p ::= [gp^*] \neg p$ where g denotes a SG. A box [...] is called a *context* and has a conjunctive semantics. *Graph formulas* and *graph propositions* have the same expressivity and a similar $\mathcal{FOL}$ semantics. The fundamental differences are (1) the disjunction operator which does not explicitly exists in graph propositions and (2) the variables which are replaced in graph propositions by equality links named *coreference*. However, these are only syntactic differences since coreference links may correspond to homonymous variables and a disjunction $A \vee B$ may be represented as the semantically equivalent graph proposition $\neg [g_\emptyset \neg A' \neg B']$ provided that $g_\emptyset$ is the empty SG and A' and B' are respectively the graph propositions corresponding to A and B .

3.4 Negated children forests

Using the algebraic laws of the language (which some are presented in subsection 3.3), a graph formula may be turned into a disjunction of conjunctions with negations respecting further specific syntactic properties. Such a rewritten formula may be handled as a set of arborescences called “negated children arborescences” (NCA) in which the nodes are the graphs from the formula. Each subarborescence in a NCA is interpreted as a conjunction of the root graph and the negations of the formulas corresponding to the subarborescences starting at its children. NCAs are collected in sets called “negated children forests” (NCFs). A NCF represents the disjunction

of the formulas corresponding to its NCAs and stands for a kind of normal form for the $SGBF$ -model.

Graph arborescence

A graph “arborescence” T is a couple $(\tau, \mathcal{G})$ such that $\mathcal{G}$ is a set of SGs and $\tau : \mathcal{G} \rightarrow \mathcal{G}$ is the “parent” function of the arborescence such that :

- there exists only one SG $r = \text{root}(T) \in \mathcal{G}$ for which $\tau(r)$ is not defined.
- there does not exist any cycles in τ , i.e. for any $n \in \mathbb{N}$ there does not exist any $g_1, g_2, \dots, g_n \in \mathcal{G}$ such that $\tau(g_1) = g_2, \tau(g_2) = g_3, \dots, \tau(g_n) = g_1$.

For two given $g, g' \in \mathcal{G}$ we consider the following operators:

- $\tau(g)$ returns the parent of g
- $\text{children}(g, T)$ returns the set $\{c \in \mathcal{G} \mid \tau(c) = g\}$
- $\text{subtrees}(g, T)$ returns the set containing each subarborescence T' from T such that $\text{root}(T') \in \text{children}(g, T)$
- $\text{ancestors}(g, T)$ returns the set $\{a \in \mathcal{G} \mid a = \tau(g) \vee a \in \text{ancestors}(\tau(g))\}$ or $\{g\}$ if $g = \text{root}(T)$
- $\text{lca}(g, g', T)$ returns the *least common ancestor* a of g and g' in T : $a \in \mathcal{A}$ such that $\text{children}(a) \cap \mathcal{A} = \emptyset$ and $\mathcal{A} = (\text{ancestors}(a, T) \cup \{g\}) \cap (\text{ancestors}(a', T) \cup \{g'\})$.

Furthermore, we extend vars such that for a given graph arborescence $T = (\tau, \mathcal{G})$, $\text{vars}(T)$ returns the set of variables occurring in $\mathcal{G}$.

Negated children arborescence

A *negated children arborescence* (NCA) is a graph arborescence $T = (\tau, \mathcal{G})$ such that for each SG g of $\mathcal{G}$ and V_a all the variables occurring in its ancestors :

- g does not contain any pair of distinct concepts with same variable
- g does not contain any concept linked to two distinct relations if its marker is a variable from V_a or an individual from $\mathcal{I}$
- if $g = \text{root}(T)$ then g may be empty ; otherwise g is not empty and restricted to one connected component
- let $S_g = \{g\}$ if g is empty and $S_g = \{g\}$ otherwise: $S_g \cup \text{subtrees}(g, T)$ can not be partitioned in two disjoint sets $\mathcal{H}$ and $\mathcal{H}'$ such that for any $h \in \mathcal{H}$ and $h' \in \mathcal{H}'$, $\text{vars}(h) \cap \text{vars}(h') \setminus V_a = \emptyset$.

The last property on $\{g\} \cup \text{subtrees}(g, T) = S_g$ (where g is not empty) means that for any two different elements h and h' from the set S_g containing g and its subarborescences, we can always find a sequence S of elements from S_g starting at h and ending at h' such that each element in S shares with its successor some homonymous variable not appearing in V_a . This means that g and its subarborescences are “stuck” together by their shared variables (excepted if they appear in V_a) and can not be separated one from the other.

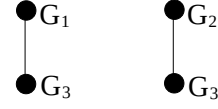


Figure 6: The NCF corresponding to the inheritance query

$$Q_H(R_2) = (G_1 \vee G_2) \wedge (\boxed{\text{Ship} : x} \wedge \neg G_3).$$

Negated children forest

A negated children forest (NCF) is a set of NCAs $(\tau, \mathcal{G})$.

The figure 6 shows the NCF obtained for the graph formula $Q_H(R_2)$. Each graph G_1, G_2 has been distributed over the

$\wedge$ connector and $\boxed{\text{Ship} : x}$ has been merged with each of them.

Semantics in graph formula

The function $t2g$ gives the translation in graph formula of a subarborescence of a given NCA starting at a given root. Given a NCA $T = (\tau, \mathcal{G})$ and a graph $x \in \mathcal{G}$, $t2g(x, T) = x \wedge \neg t2g(c_1, T) \wedge \neg t2g(c_2, T) \wedge \dots \wedge \neg t2g(c_n, T)$ such that $c_i \in \text{children}(x, T)$.

The translation of a NCF $F = \{T_1, T_2, \dots, T_n\}$ in graph formula is the disjunction $\mathcal{F}2g(F) = t2g(\text{root}(T_1), T_1) \vee t2g(\text{root}(T_2), T_2) \vee \dots \vee t2g(\text{root}(T_n), T_n)$

Every graph formula f has a semantically equivalent NCF $g2\mathcal{F}(f)$ computable by an algorithm $g2\mathcal{F}$ which migrates disjunctions as more as possible to the top and negations as deeper as possible in the graph formula. The algorithm always terminates and returns the NCF $F = g2\mathcal{F}(f)$ such that $\mathcal{F}2g(F)$ is logically equivalent to f . For example, the corresponding graph formula for the NCF in figure 6 is $(G_1 \wedge \neg G_3) \vee (G_2 \wedge \neg G_3)$ which is equivalent to the initial

$$\text{graph formula } Q_H(R_2) = (G_1 \vee G_2) \wedge (\boxed{\text{Ship} : x} \wedge \neg G_3).$$

NCAs are used to structure graph formulas in layers of irreducible negations. Given a SG g , all subarborescences starting at one child of g (if any) are considered as negated relatively to g (which may be itself the start of a subarborescence negated relatively to its parent if any). Such negations are said irreducible in the sense that we can not refactor the arborescence into a shorter one without changing its semantics (we can not remove these negations). The deeper a NCA is, the more irreducible negations are nested in the corresponding graph formula, and the more complex is the formula. A NCF may be considered as a disjunctive clause in which NCAs are totally independent in the sense that if we rename variables in one NCA without minding the others, this does not affect the semantics of the NCF.

4 The satisfiability problem

Given a graph formula f , the problem to determine if f is satisfiable is not decidable for the general case since the $SGBF$ -model and $\mathcal{FOL}$ have the same expressivity. If it were, this would lead to the contradiction that the satisfiability problem is decidable in $\mathcal{FOL}$ (which is not). One solution is to restrict the expressivity of the language until there exists a decision

procedure for the resulting fragment. There is different ways to reduce the expressivity, and this lead us to define two different fragments. In the following, these fragments are defined in terms of NCF and not in terms of graph formula (because it is easier to do so). Thus, in order to check if a given graph formula f is satisfiable, the reasoning system described in this paper translates it into a NCF F and selects the decision procedure defined for the fragment to which belongs F . This shows the benefits of the NCF normal form : which is used as a “hub” between several decidable fragments of the graph formula language.

The two decidable fragments of graph formulas are defined in the following :

(1) For a NCF containing only NCAs of height at most 2 the decision procedure is based on the projection mechanism mentioned in subsection 2.4. More precisely, given a NCF F in which each NCA is restricted to one root graph whose children are leaves, the algorithm checks if for each NCA in F one projection² can be found between at least one leaf child and its parent. If such a projection can be found for each NCA, then F is said *unsatisfiable* ; otherwise it is *satisfiable*. This fragment is called the *2-height fragment* of NCFs. As the decision procedure for this fragment is straightforward, we do not present more details of its implementation in order to focus on the second decision procedure which is much more complex.

(2) As the satisfiability problem has been proven decidable for some specific fragments of $\mathcal{FOL}$, the other way to find a decidable fragment for the language of graph formulas is to transfer directly in graph formula the syntactic restrictions a logical formula has to respect in order to belong to one of the decidable fragments of $\mathcal{FOL}$. When dealing with possibly negated propositions of conceptual graphs, [Baader *et al.*, 1998] suggests to restrict SGs such that their logical translation lead to a formula from the $\mathcal{FOL}$ *guarded* and *loosely guarded* fragments. However, the definition of *guarded* graph propositions sets strong restrictions on the syntax of the SGs used in propositions. Roughly speaking, in order for a proposition to be guarded, all concept nodes of a SG must be neighbours of one unic relation (when the graph contains variables not appearing elsewhere in the proposition). It turns out that SGs in propositions are very compact and small in practice.

In the following parts of this paper, after a brief presentation of the guarded fragment of $\mathcal{FOL}$, we present a characterisation of a guarded graph formula in which the previous restriction is relaxed such that SGs may have a much more complex tree-like structure.

4.1 The guarded fragment of $\mathcal{FOL}$

The $\mathcal{FOL}$ *guarded* and *loosely guarded* fragments were introduced in [Andréka *et al.*,] as a generalization of modal logic. It appears that $\mathcal{FOL}$ guarded fragments retain a lot of good properties of modal logics: its satisfiability problem is decidable, it has the finite model property and a sort of tree model property (see [Grädel, 1998]). In the following, we focus on the guarded fragment and do not consider the loosely

guarded fragment (which is a strictly more expressive generalization of the guarded fragment).

The guarded formulas of $\mathcal{FOL}$ are defined by induction as follows.

- an atomic formula is a guarded formula
- if ϕ, ψ are guarded formulas then $\phi \wedge \psi$ and $\neg\psi$ are guarded formulas
- let G be an atom, $\tilde{x} \subseteq \text{free}(G)$ a non-empty set of variables and ψ a guarded formula such that $\text{free}(\psi) \subseteq \text{free}(G)$, then $\exists \tilde{x} G \wedge \psi$ is a guarded formula.

4.2 Guarded simple graphs

For a given support $S = (T_C, T_R, \mathcal{I})$, the set $\mathcal{SG}_G(S)$ of guarded SGs over S is defined in induction by the following rules :

- $\mathcal{SG}_G(S)$ contains all SGs with one relation r of type $t_r \in T_R$ and arity k linked to k concepts $c_i = \boxed{t_i : m_i}$ of type $t_i \in T_C$ and marker $m_i \in \mathcal{M}$.
- given a *guarded* SG g , g remains *guarded* after adding a new relation r of type t_r with arity $k = k_n + k_g$ linked to:
 - k_n new concept nodes $\boxed{t_i : m_i}$ of type $t_i \in T_C$ and marker $m_i \in \mathcal{M} \setminus \text{vars}(g)$ and
 - $k_g \geq 1$ concept nodes linked to one unic relation $r' \neq r$ from g

The basic idea standing behind the previous rule is that the guarded property of a given SG is preserved when the added new relation is connected to some of the concept nodes linked to only one target relation in the graph. The new relation may introduce some new concept nodes with new variable names, and may be considered as the next target relation while adding an other new relation (and so on). This inductive construction of the graph organise relation nodes in a tree-like structure. The added relation may be considered as a sort of *child* of the target relation. However, the graph is not exactly a tree. Some concept and relation nodes may form cycles of the form $c_1 r_1 c_2 r_2 \dots c_{n-1} r_{n-1} c_n = c_1$ provided that all concept nodes $c_1 c_2 \dots c_{n-1}$ of the cycle are linked to the same relation node.

In the figure 3, the graphs G_2 and G_3 are guarded. For the graph G_1 to be guarded, we have to split the individual concept $\boxed{\text{Energy} : \text{Electricity}}$ such that it is not anymore linked to two different relation nodes.

4.3 Guarded NCF

A NCA T is *guarded* iff for any graph g in T , g is guarded and there exists a relation node r in g called the *root* of g such that for any graph $g' \neq g$ in T , if $V = \text{vars}(g) \cap \text{vars}(g') \neq \emptyset$ then :

- if g is not an ancestor of g' , then each variable from V appears in a concept node linked to r
- if a is the least common ancestor $\text{lca}(g, g', T)$ of g and g' and $\mathcal{A}$ the set of graphs³ in the arborescence between g (excluded) and a (included), then in each graphs of $\mathcal{A}$

²The projection has to preserve the name of variables.

³ $\mathcal{A} = \text{ancestors}(g, T) \setminus \text{ancestors}(a, T)$

there exists a relation node r_a such that each variable from V appears in a concept node linked to r_a

A NCF is *guarded* iff all its NCAs are *guarded*.

For example, if we consider the two NCAs of the NCF in figure 6, the *root relation* node r of G_3 is the same for both NCAs (in bold in figure 3). The two NCAs are guarded since (1) the three SGs are guarded, (2) the shared variable x between the three SGs appears in a concept node linked to r and (3) we can find a relation node r' in G_1 or G_2 such that x appears in a concept node linked to r' .

4.4 Translation to the guarded fragment

Given a support S , the standard translation $\phi(S)$ already returns a set of logical guarded formulas. Indeed, each formula in $\phi(S)$ is of form $\forall \tilde{x} p(x) \rightarrow q(x)$ and may be written as the equivalent formula $\neg \exists \tilde{x} p(x) \wedge \neg q(x)$ which belongs to the guarded fragment.

Given a SG g of a NCA T , a set X of variables and a relation node x denoting the *root* of the graph, the translation into a guarded logical formula starts at x and follows recursively the inductive structure of the graph in order to translate the next relations located “below” r . While browsing the graph, each relation r is translated in a formula of form $\exists \tilde{y} R \wedge \gamma$ where R is the atom corresponding to r and $\tilde{y} \subseteq \text{free}(R)$ and $\text{free}(\gamma) \subseteq \text{free}(R)$. If there exists a child g' from $\text{children}(g, T)$ such that its root relation x' shares variables from r then the translation is recursively launched on x' in T and the result is added in γ . For a given NCF, the translation returns a disjunction of guarded formulas corresponding to its NCAs (which is a guarded formula).

For example, let F_1 be the guarded formula $\exists y \exists z (\text{providedWith}_3(x, y, z) \wedge \text{Tanker}(x) \wedge \text{Tank}(y) \wedge \text{Tank}(z) \wedge \text{adjacent}(y, z))$ corresponding to G_3 , the translation result for the NCF of the figure 6 is the disjunction of the following two formulas :

- $(\exists x \exists y \text{providedWith}_2(x, y) \wedge \text{Ship}(x) \wedge F_1 \wedge \text{Engine}(y) \wedge \text{Energy}(\text{Electricity}) \wedge \text{poweredWith}(y, \text{Electricity}) \wedge \exists z ((\text{providedWith}_2(x, z) \wedge \text{CoolingSystem}(z) \wedge \text{poweredWith}(z, \text{Electricity})))$
- $(\exists p \exists q \exists r \text{providedWith}_4(x, p, q, r) \wedge \text{Ship}(x) \wedge F_1 \wedge \text{Engine}(p) \wedge \text{Engine}(q) \wedge \text{CoolingSystem}(r) \wedge \text{twined}(p, q) \wedge \text{cooledBy}(p, r) \wedge \text{cooledBy}(q, r))$

5 Conclusion

5.1 Back to the classification service

In the classification system of ITM, queries associated to regulations in the law hierarchy are defined as graph formulas. When a new chapter R_L with query $Q(R_L)$ is *inserted* below an existing chapter R in the hierarchy, R becomes the parent of R_L and some of R children may be transferred as children of R_L . Then, the system is designed to return four sets of regulations in which a regulation R is respectively stated as (1) similar to R_L , (2) useless, (3) entailed by R_L , (4) entailing R_L . If it can not be decided if *yes* or *no* a regulation R_U belongs to one of these sets, then R_U is stored in a fifth set: the

undecidable set. This means that the system could not analyse the impact on R_U of the insertion of R_L in the hierarchy, and that the lawmaker has to check manually if R_L remains compatible with R_U .

More formally, given a new chapter R_L inserted in the law hierarchy the system decides if a regulation R belongs to one of the four previous sets by respectively checking if:

- (1) $(Q_H(R_L) \wedge \neg Q_H(R)) \vee (Q_H(R) \wedge \neg Q_H(R_L))$,
- (2) $Q_H(R)$,
- (3) $Q_H(R_L) \wedge \neg Q_H(R)$
- (4) $Q_H(R) \wedge \neg Q_H(R_L)$

are logically unsatisfiable.

To do so, each of the four previous graph formula g is translated into a NCF $F(g)$. If $F(g)$ is *neither a 2-height nor a guarded* NCF, the system stores R in the *undecidable* set. Otherwise, when $F(g)$ is a *2-height* NCF the decision procedure defined for this kind of NCF is used to check its unsatisfiability. Finally, when $F(g)$ is a *guarded* NCF, it is translated to the corresponding guarded logical formula and a specific guarded fragment reasoning engine is used to check its unsatisfiability.

5.2 Perspectives

The classification service presented in this paper translates - when needed and possible - graph formulas in their corresponding guarded logical formulas, in order to check their unsatisfiability with a logical reasoning engine. Our future work is to transfer existing decision procedures defined for the guarded fragment into the conceptual graph formalism, by making use of the projection elementary operation to build a corresponding “graphical” decision procedure.

References

- [Andréka *et al.*,] H. Andréka, I. Nemeti, and J. van Benthem. Modal languages and bounded fragments of predicate logic. pages 27:217–274.
- [Baader *et al.*, 1998] Franz Baader, Ralf Molitor, and Stephan Tobies. The guarded fragment of conceptual graphs. Technical report, RWTH Aachen, LuFg Theoretische Informatik, 1998.
- [Baget and Mugnier, 2002] J.-F. Baget and M.-L. Mugnier. Extensions of Simple Conceptual Graphs: The Complexity of Rules and Constraints. *JAIR*, 16:425–465, 2002.
- [Carloni *et al.*, 2007] Olivier Carloni, Michel Leclère, and Marie-Laure Mugnier. Introducing graph-based reasoning into an industrial knowledge management tool. *Applied Intelligence*, 2007.
- [Chein and Mugnier, 1992] M. Chéin and M.-L. Mugnier. Conceptual Graphs: Fundamental Notions. *Revue d'Intelligence Artificielle*, 6(4):365–406, 1992.
- [Grädel, 1998] Erich Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64:1719–1742, 1998.
- [Sowa, 1984] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, 1984.

Graph Representations of Consistency and Truth-Dependencies in Logic Programs With Answer Set semantics

Stefania Costantini

Dip. d'Informatica, Università di L'Aquila, Italy

stefania.costantini@univaq.it

Alessandro Provetti

Dip. di Fisica, sez. d'Informatica, Università di Messina, Italy

ale@unime.it

Abstract

In this paper, we propose a formalization of the features that a graph representation of logic programs under the answers set semantics should in our opinion exhibit in order to be a satisfactory and useful representation formalism. We introduce a concept of isomorphism between a program and the corresponding graph. We argue that isomorphic representations can be a good software engineering tool for understanding program behavior, for checking consistency, for being able to create, debug and combine good programs, and for developing program analysis techniques.

1 Introduction

Answer Set Programming (ASP) [Gelfond and Lifschitz, 1988] [Lifschitz, 1999] (also called Stable Logic Programming (SLP) [Marek and Truszczyński, 1999]), is a relatively recent but well-established style of logic programming: each solution to a problem is represented by an answer set (also called stable model), and not by answer substitutions produced in response to a query. Rich literature exists on applications of ASP in many areas, including problem solving, configuration, information integration, security analysis, agent systems, semantic web, and planning (see among many [Baral, 2003; Anger *et al.*, 2004; Leone, 2007; Truszczyński, 2007; Gelfond, 2007] and the references therein).

A hot topic in Answer Set Programming on which quite a lot of work is under way (see e.g. the Proceedings of the SEA - Software Engineering in ASP - Workshop series) is that of defining programming methodologies. In fact, with the growth of practical real-world ASP applications, the need is arising of writing programs with a complicated structure, and of composing existing programs into larger ones. Guidelines should be provided to programmers and system developers, in order to write consistent (or, sometimes, inconsistent on purpose) programs, and to combine them.

Under this perspective, in our opinion there is still the need to investigate basic properties of programs, first of all consistency, but it is also important to be able to reason about programs, debug programs, understand where potential problems arise and how to fix them, and understand what happens when programs are updated. Thus, program representation

and analysis tools should be made available, so as to define and check properties of programs.

These tools should be based upon some kind of representation of logic programs. We believe that the properties of a good representation are at least the following: (i) the relationship between the program and its representation is clearly understandable; (ii) it is easy to shift back and forth from the program to the representation, so that a change in the program reflects into a change in the representation, and vice versa; (iii) it is possible to use the representation for developing techniques to define significant subprograms, study their interactions and determine which program parts can influence specific, critical points, and thus influence relevant program properties.

Graph representations are in our opinion good candidates. In fact, graphs are simple and understandable. Graph-theoretic structures are able to represent properties of programs. Known properties of graphs can help improve the understanding of structural properties of the programs themselves. Especially in the field of imperative programming (but not only), techniques such as *program slicing* [Tip, 1995] based on graph representations are usefully exploited for verifying program properties. It is important to investigate whether similar techniques can be developed for Answer Set Programming as well. Finally, graphs can be shown and manipulated on the screen in a what-you-see-what-you-get fashion.

In this paper we propose a formalization of the features that a graph representation of logic programs should in our opinion exhibit in order to be a satisfactory and useful representation formalism. In order to show that graph representations are able to fulfill the above-mentioned points (i) and (ii), and in perspective also (iii), we introduce a concept of isomorphism between a program and the corresponding graph. We prove that the isomorphism is possible only if a graph representation formalism is able to distinguish the cycles occurring in the program, and the different connections among them. We argue that understanding cycles and their connections is a key point for understanding program behavior and for checking consistency, for being then able to create, debug and combine good programs, and for developing program analysis techniques.

We briefly discuss isomorphism w.r.t. three graph representations: the traditional Dependency Graph DG , the

Extended Dependency Graph *EDG* [Brignoli *et al.*, 1999] and the Rule Graph *RG* [Dimopoulos and Torres, 1996; Linke, 2001; Konczak *et al.*, 2003] (with some of its extensions).

In order to demonstrate the usability of an isomorphic graph representation in program development, we show that properties of a program, first of all consistency, can be characterized (and checked) in terms of *admissible colorings* of an isomorphic graph, and in particular of the *EDG*. We give some suggestions about how to use the *EDG* for fixing inconsistencies. Finally show that, on the basis of the *EDG*, useful classes of consistent programs can be easily defined. We argue that an automated analysis tool might easily verify whether a program belongs to this class, and that program fragments of this kind might act as building blocks for safely constructing larger ones.

The paper is organized as follows. In Section 3 we discuss properties of graph representations and we introduce *isomorphic* graph representations, and their relationship to cycles. In Section 2 we motivate these representations by showing how cycles occurring in a program affect the existence and number of answer sets. In Section 4 we discuss the Extended Dependency Graph and its usability for exploring properties of programs. In Sections 5 and 6 we discuss in particular how answer sets of a program can be characterized in terms of admissible colorings of the corresponding *EDG*. In Section 7 we outline a discussion about how the *EDG*, that turns out to be an isomorphic graph representation formalism, can be useful in program development. Finally, Section 8 concludes the paper. In this paper we build upon previous work, pieces of which we report here for the sake of clarity. However, the material in Sections 3, 6 and 7 is new.

2 The relationship between cycles and answer sets

There is a strong relationship between cycles occurring in a program, and existence and number of answer sets. This motivates the usefulness of defining and studying graph representations, where cycles and their connections become explicit. The material informally presented in this section for the sake of clarity has been formally discussed in depth and at length in [Costantini, 2006b].

As it is well-known, the Dependency Graph is not a satisfactory representation for logic programs under these respects, as shown by the following example.

Example 1 Consider the following programs Π_1 , Π_2 and Π_3 (from left to right):

$p \leftarrow \text{not } p, \text{not } e.$	$p \leftarrow \text{not } p.$	$p \leftarrow \text{not } p, \text{not } e.$
$a \leftarrow \text{not } b.$	$p \leftarrow \text{not } e.$	$a \leftarrow \text{not } b.$
$b \leftarrow \text{not } a.$	$a \leftarrow \text{not } b.$	$b \leftarrow \text{not } a.$
$e \leftarrow \text{not } f.$	$b \leftarrow \text{not } a.$	$e \leftarrow \text{not } f.$
$f \leftarrow \text{not } h.$	$e \leftarrow \text{not } f.$	$f \leftarrow \text{not } h.$
$h \leftarrow \text{not } e.$	$f \leftarrow \text{not } h.$	$h \leftarrow \text{not } e, \text{not } a.$
$h \leftarrow \text{not } a.$	$h \leftarrow \text{not } e, \text{not } a.$	

It is easy to see that the dependency graphs of the three programs coincide. Namely, the *DG* of the three programs is the leftmost graph of Figure 1. However, Π_1 has answer set

$\{b, e, h\}$ while instead Π_2 has answer set $\{a, f, p\}$ and Π_3 has no answer sets at all, i.e., it is inconsistent.

Inconsistency may arise only because of odd cycles: in fact, call-consistent programs, which do not contain odd-cycles, are always consistent. In the *DG* of the above programs we can see two odd cycles: the first one involving atom p , which depends on itself; the second one involving atoms e , f and h . However, the different connections between the odd cycles and the rest of the program, that cannot be seen on the *DG*, influence the existence and the number of answer sets. Moreover, based on the *DG* it is impossible to make predictions about what happens in case of modifications. For instance, if adding fact e , the following happens: (i) program Π_1 remains consistent, with the two answer sets $\{b, e, h\}$ and $\{a, e, f\}$; (ii) Π_2 becomes inconsistent; (iii) Π_3 becomes consistent, with the two answer sets $\{b, e, f\}$ and $\{a, e, f\}$.

Summarizing, the problem with an odd cycle under the answer set semantics is that the corresponding fragment of the overall program, taken alone, is inconsistent. Consider for instance the odd cycle involving atoms e, f, h in the *DG* of Example 1. The corresponding program fragment is $\{e \leftarrow \text{not } f. f \leftarrow \text{not } h. h \leftarrow \text{not } e.\}$. None of the atoms can be either true or false without causing a contradiction (a true atom which depends on the negation of a true atom). If, for instance, we assume f to be false, then e should be concluded to be true (by means of the first rule); consequently, h should be concluded to be false (by means of the third rule, since no other rules for h are available); but, at this point, f should be concluded to be true (by means of the second rule), which is a contradiction.

The only potential way of getting rid of inconsistency is that of connecting cycles. In previous works [Brignoli *et al.*, 1999; Costantini, 2006b] we have identified (and formally defined) two kinds of connections. Let rule ρ of the form $\alpha \leftarrow \text{not } \beta$ be part of a cycle C . I.e., on the *DG* the edge connecting β to α belongs to a cycle C .

First kind of connection:

the head α of rule ρ is defined also by (at least) another rule, for instance of the form $\alpha \leftarrow \text{not } \delta$. In any answer set, if δ is false then α is true. We call $\text{not } \delta$ an *OR handle* for the cycle to which ρ belongs. We also say that this OR handle is relative to atom α . Whenever δ is false and $\text{not } \delta$ is true, we say that the OR handle is *active*.

Second kind of connection:

rule ρ has an additional condition, for instance C is of the form $\alpha \leftarrow \text{not } \beta, \text{not } \gamma$.

In any answer set, if γ is true then α cannot be obtained by this rule (it may, however, from other rules defining α). We call $\text{not } \gamma$ an *AND handle* for the cycle to which ρ belongs. We also say that this AND handle is relative to atom α . Whenever γ is true and $\text{not } \gamma$ is false, we say that the AND handle is *active*.

An active handle gives a truth value to an atom involved in a cycle. Consequently, the truth value of all the other atoms of the cycle follows without contradictions. A necessary condition for consistency is that every odd cycle has at least one handle. This condition is not sufficient, since two handles can be in conflict, in the sense that they cannot be

active simultaneously. For instance, take program fragment $\{v \leftarrow \text{not } v, \text{not } w, t \leftarrow \text{not } t, t \leftarrow \text{not } w\}$: there are two odd cycles, one involving v with AND handle $\text{not } w$, and the other one involving t with OR handle $\text{not } w$. Since the same condition $\text{not } w$ occurs in two handles of different kind, they cannot be simultaneously active, and therefore the cycle, and the overall program, will be inconsistent. Whenever we have an even cycle without handles (*unconstrained* even cycle) we can choose any of the two answer sets of the corresponding program fragment in order to form an answer set of the overall program. For instance, for even cycle $a \leftarrow \text{not } b, b \leftarrow \text{not } a$ any of the two answer sets $\{a\}, \{b\}$ can be selected.

Programs Π_1 and Π_2 of Example 1 are divided into cycles as follows, where OC and EC denote odd and even cycle, respectively, and conditions appearing either in square brackets or in braces correspond to different kinds of handles.

Consider first program Π_1 .

$$\begin{aligned} OC_1 : & \{ p \leftarrow \text{not } p, [\text{not } e]. \\ EC_1 : & \begin{cases} a \leftarrow \text{not } b. \\ b \leftarrow \text{not } a. \end{cases} \\ OC_2 : & \begin{cases} e \leftarrow \text{not } f. \\ f \leftarrow \text{not } h. \\ h \leftarrow \text{not } e. \end{cases} \\ H_1 : & \{ h \leftarrow \{\text{not } a\}. \end{aligned}$$

Condition $\text{not } a$ in H_1 is an OR handle for OC_2 . This handle becomes active by selecting answer set $\{b\}$ for the unconstrained even cycle EC_1 . This handle causes h and e to be true. Then, the AND handle of OC_1 becomes active, making p false. Consequently, Π_1 has answer set $\{b, h, e\}$.

Similar considerations can be made on Π_2 , even though it has a different structure: condition $\text{not } a$ in this case is an AND handle for OC_2 .

$$\begin{aligned} OC_1 : & \{ p \leftarrow \text{not } p. \\ H_2 : & \{ p \leftarrow \{\text{not } e\}. \\ EC_1 : & \begin{cases} a \leftarrow \text{not } b. \\ b \leftarrow \text{not } a. \end{cases} \\ OC_2 : & \begin{cases} e \leftarrow \text{not } f. \\ f \leftarrow \text{not } h. \\ h \leftarrow \text{not } e, [\text{not } a]. \end{cases} \end{aligned}$$

If $\text{not } a$ is false, then h is forced to be false, and consequently OC_2 has answer set $\{f\}$. This means that the OR handle $\text{not } e$ of OC_1 is true, and thus p is true: therefore the contradiction $p \leftarrow \text{not } p$, which could determine the inconsistency, is made harmless. Consequently, Π_2 has answer set $\{a, f, p\}$.

Finally, consider program Π_3 :

$$\begin{aligned} OC_1 : & \{ p \leftarrow \text{not } p, [\text{not } e]. \\ EC_1 : & \begin{cases} a \leftarrow \text{not } b. \\ b \leftarrow \text{not } a. \end{cases} \\ OC_2 : & \begin{cases} e \leftarrow \text{not } f. \\ f \leftarrow \text{not } h. \\ h \leftarrow \text{not } e, [\text{not } a]. \end{cases} \end{aligned}$$

For the AND handle of odd cycle OC_2 to be active, we must select answer set $\{a\}$ for EC_1 . In this way however, we have h false, f true and e false: this means, the (unique) AND handle of odd cycle OC_1 is not active, and therefore Π_3 has no answer sets.

3 Properties of graph representations

The main property of programs one wants in the first place to study under the answer set semantics is consistency, i.e., the existence of answer sets. In general however, one has to actually compute and often examine the answer sets (e.g., in order to check whether some desired conclusion has been actually reached, or if a soft constraint is satisfied). In some situations, for instance for debugging a program or for trying to combine different program components, one may want to be able to investigate the program structure, in order to understand which parts influence which others, and then what ensures consistency, and what can spoil it [Costantini, 2006b]. When modifying and updating programs, it can be useful to reason about how the existence and content of answer sets is affected by modifications and/or updates performed on the given program [Costantini *et al.*, 2003].

Consider again the programs in Example 1. Why do they have such a diverse semantics, even though they contain the same cycles? What the DG does not reveal is in fact the structure of the connections between cycles.

Since we are interested in the syntactic structure of programs, let us characterize those programs that have the same structure, but different names for the atoms occurring in them.

Definition 1 Let Π be a logic program with Herbrand base $B_\Pi = \{a_1, a_2, \dots, a_h\}$, and let B'_Π be a set of atoms with the same cardinality as B_Π . A renaming θ is a bijective function from B_Π to B'_Π , that can be denoted as a set $\{(a_1, a'_1), (a_2, a'_2), \dots, (a_h, a'_h)\}$ where $a_i \neq a_j$ implies $a'_i \neq a'_j$, while for some k we may have $a_k = a'_k$. θ can be applied to Π , thus obtaining a new program $\Pi' = \Pi\theta$, by replacing every occurrence of a_i by a'_i .

Definition 2 Two logic programs Π_1 and Π_2 are renaming-equivalent if and only if there exists a renaming θ from B_{P_1} to B_{P_2} such that $P_2 = P_1\theta$.

Let $\mathcal{GF}$ be any graph representation formalism (for instance the DG). By $G \in \mathcal{GF}$ we mean a graph G which can be constructed for some program according to $\mathcal{GF}$. By $\mathcal{GF}(\diamond)$ we mean the graph which represents program Π according to $\mathcal{GF}$. The following is a straightforward property of any graph representation.

Definition 3 A graph representation formalism $\mathcal{GF}$ is acceptable if and only if for every logic program Π , $\mathcal{GF}(\Pi)$ is unique.

A stronger requirement is that, given a graph, we can reconstruct the program that is represented by this graph, or at least a program, if there is more than one choice. This requirement is related to the usefulness of the representation as a software engineering tool: given a program, one can determine how it is represented; given a graph, one can understand which kind of program it represents.

Definition 4 A graph representation formalism $\mathcal{GF}$ is adequate if and only if for every graph $G \in \mathcal{GF}$, it is possible to construct from G a logic program Π such that $G = \mathcal{GF}(\Pi)$.

We may expect that two programs which are renaming-equivalent are represented by graphs which have the same

structure. Instead, it is reasonable to expect that different programs have different representations.

Definition 5 A graph representation formalism $\mathcal{GF}$ is unambiguous if and only if for any two programs Π_1 and Π_2 that are not renaming-equivalent, we have that, considering the labeling of vertices, $\mathcal{GF}(\Pi_1) \not\approx \mathcal{GF}(\Pi_2)$.

For any graph representation formalism $\mathcal{GF}$ which is acceptable, adequate and unambiguous, we say that $\mathcal{GF}(\diamond)$ is isomorphic to Π . In fact, Π can uniquely be determined from $\mathcal{GF}(\diamond)$. More generally, we will say that the representation $\mathcal{GF}$ is isomorphic to logic programs. For short, we say that $\mathcal{GF}$ is an *isomorphic graph representation*.

Clearly, the Dependency Graph representation formalism is acceptable and adequate, but it is not unambiguous. In the example above in fact, it is impossible to say from the graph which of the three programs is being represented. Consequently, on the basis of the graph it is impossible to characterize consistency under the answer set semantics: we do not know from the graph whether the program under consideration has answer sets, and it is impossible to find these answer sets, and to predict whether they will change after updates.

Finally, we formulate the following proposition, which will be proved formally in the extended version of this work:

Proposition 1 A graph representation formalism $\mathcal{GF}$ which does not distinguish between AND and OR handles of cycles is not an isomorphic representation of logic programs.

In fact, as proved in [Costantini, 2006a], the Rule Graph is acceptable and adequate but cannot distinguish between AND and OR handles, hence it cannot be an isomorphic graph representation formalism. This is not to say that they cannot be of use: in fact, these representations have been adopted as a basis for implementing the noMoRe answer set solver [Anger et al., 2001], and have also been used in order to reason about loop formulas (positive cycles) in [Gebser and Schaub, 2005].

4 The Extended Dependency Graph

The Extended Dependency Graph (*EDG*) has been introduced in [Brignoli et al., 1999]. The *EDG* is similar to the *DG*, except it is more accurate for negative dependencies. The definition is based upon distinguishing among rules defining the same atom, i.e., having the same head. To establish this distinction, we assign to each head an upper index, starting from 1, where upper index 0 is reserved to atoms with no defining rules. E.g., $\{a \leftarrow c, \text{not } b. a \leftarrow \text{not } d. b \leftarrow .\}$ becomes $\{a^{(1)} \leftarrow c^{(1)}, \text{not } b^{(1)}. a^{(2)} \leftarrow \text{not } d^{(0)}. b^{(1)} \leftarrow .\}$.

The following definition of Extended Dependency Graph was given in [Brignoli et al., 1999] as the starting point of the present research programme.

Definition 6 (Extended dependency graph for general programs) (*EDG*)

Given a general logic program Π , its associated Extended Dependency Graph $EDG(\Pi)$ is the directed finite labeled graph $(V, E, \{+, -\})$ where $\{+, -\}$ are the labels associated to the edges, and the sets V and E are defined as follows.

V:1 vertices For each rule in Π there is a vertex $a_i^{(k)}$, $k \geq 1$, where a_i is the name of the head and k is the index of the rule in the definition of a_i .

V:2 vertices For each atom a_i in Π that does not occur in the head of any rule there is a vertex $a_i^{(0)}$.

E:1 edges For each $c_j^{(l)} \in V$, there is an edge $\langle c_j^{(l)}, a_i^{(k)}, + \rangle$, if and only if c_j appears as a positive condition in the k -th rule defining a_i .

E:2 edges For each $c_j^{(l)} \in V$, there is an edge $\langle c_j^{(l)}, a_i^{(k)}, - \rangle$, if and only if not c_j appears as a negative condition in the k -th rule defining a_i .

A simplified definition was later given in [Costantini and Proveti, 2005b] for the special case of negative programs, where only cyclic, negative conditions are allowed in the program. The *EDG* for negative programs can be seen as a shortcut of the *EDG* for general programs, where each edge $\langle c_j^{(l)}, a_i^{(k)} \rangle$ is implicitly labeled with “ $-$ ”, i.e., it corresponds to an edge $\langle c_j^{(l)}, a_i^{(k)}, - \rangle$.

It is important to notice that the definition of *EDG* extends that of *DG* in the sense that for programs where atoms are defined by at most one rule the two representations coincide. Consider in Figure 1 the *EDGs* for the programs in Example 1 (where $h^{(1)}$, $h^{(2)}$, $p^{(1)}$ and $p^{(2)}$ are indicated as h , h' , p and p' , respectively).

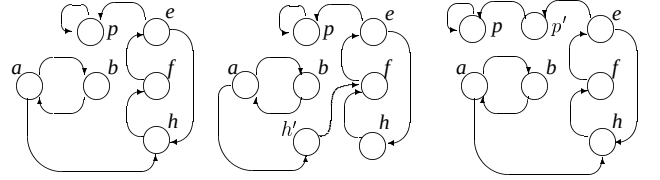


Figure 1: $EDG(\Pi_3)$ (left), $EDG(\Pi_1)$ (center) and $EDG(\Pi_2)$ (right). The leftmost graph coincides with the *DG* of the three programs.

On the *EDG*'s, we clearly see the cycles, and also the handles. Consider $EDG(\Pi_1)$, which is depicted as the center graph of the Figure; rule $\{f \leftarrow \text{not } h.\}$ is represented by the two edges $\langle h, f \rangle$ and $\langle h', f \rangle$, since truth of h may depend on either of its two defining rules. The rule defining h' is auxiliary to the cycle, and corresponds to an OR handle. Therefore, we can see on the *EDG* that the cycle has an OR handle because there is an incoming edge originated in a duplication of one of the atoms of the cycle. In this case, the edge incoming into h' represents the *OR* handle of OC_2 . In the same graph, edge $\langle e, p \rangle$ represents instead the *AND* handle of OC_1 . We can see on the *EDG* that the cycle has an AND handle because there is an incoming edge into that cycle, originated in a duplication of an atom not belonging to the cycle itself. The even cycle involving a and b has no incoming edges and thus it is unconstrained.

In general: given program Π , a cycle in Π corresponds to a cycle in $EDG(\Pi)$. On the *EDG*: (1) An AND handle for a cycle C relative to atom α occurring in C is represented by an edge incoming into α (additional condition of the rule defining α). Atom α (and cycle C) may have either none or

one or several AND handles. (2) An OR handle for a cycle C relative to atom α occurring in C is represented by a sibling node of α (additional rule defining α). Precisely, if C actually includes vertex α^i , another vertex α^j $j \neq i$ is also present in the *EDG*. Atom α (and cycle C) may have either none or one or several OR handles.

Finally, the following result is easily stated.

Proposition 2 *The EDG is an isomorphic graph representation formalism.*

Consequently, the *EDG* conveys enough information for reasoning about answer sets of the program. The usefulness of the *EDG* from a software engineering perspective is discussed in the next section.

In order to argue in favor of the usefulness of isomorphic graph representations for software engineering, in the next sections we give some examples of use of the *EDG* for checking properties of programs, for fixing inconsistencies, and for developing programs which are consistent by construction.

In particular, we show how the *EDG* can be used to study properties of logic programs under the answer set semantics in terms of graph coloring. We then outline the potential usefulness of the *EDG* as a software engineering tool.

Let us define a *coloring* as an assignment of nodes of a graph to colors. Different graph colorings can be in principle defined, corresponding to specific properties of a program. Assume that the property that you want to check is consistency, i.e. the existence of answer sets. With respect to consistency, let the corresponding assignment be $\nu : V \rightarrow \{\text{green}, \text{red}\}$, where intuitively *green* stands for “true” and *red* stands for “false”. Then, an interpretation corresponds to a coloring where all the true atoms are green, and all the others are red. Later on we will define this correspondence in a formal way.

5 Program Analysis: Consistency of Kernel Programs

For the sake of completeness, in this section we start our analysis from a particular kind of negative programs, kernel programs, already studied in [Brignoli *et al.*, 1999], and then extend the analysis to general programs in the next Section. Thus, definitions and results about negative programs reported below are taken from [Brignoli *et al.*, 1999]

Kernel programs are negative programs that are wfs-irreducible (i.e., no atom is undefined under the well-founded semantics [VanGelder *et al.*, 1991]) and all atoms occurring in the program also occur as the head of some rule. In [Costantini and Proveti, 2005a] it is shown that the kernel form is a normal form, in the sense that every logic program can be reduced without loss of generality to a kernel counterpart.

For each atom a_i , the *EDG* contains sibling vertices $a_i^1, \dots, a_i^k$ corresponding to its k defining rules.

In a coloring that is meant to correspond to an answer set of the given program, if any of the siblings, say a_i^r , is colored green, this means to assign truth value “true” to atom a_i . In fact, a_i , may become true by means of any of its rules. Conversely, a coloring implies truth value “false” for atom b_j

represented by sibling vertices $b_j^1, \dots, b_j^h$ only if all of them are colored red.

Referring to the answer set semantics and taking a rule $a_i \leftarrow \text{not } b_j$, for a_i to be true via this rule b_j must be false. This rule on the *EDG* will result in one of the sibling vertices for a_i , say a_i^r , to be connected via (negative) edges to *all* the sibling vertices $b_j^1, \dots, b_j^h$ for b_j . In terms of coloring of the *EDG*, we are allowed to assign green to a_i^r provided that we assign color red to *all* the siblings $b_j^1, \dots, b_j^h$ corresponding to b_j . Assigning red to all the b_j^h 's amounts to assuming that b_j cannot be possibly concluded true via any of its defining rules.

Based on the above-mentioned principles, below we define a concept of *admissible coloring* of the *EDG* of a kernel program. We then prove that admissible colorings characterize the answer sets of programs. We will start by defining which colorings are *not* admissible. Below we specify in fact which colorings we intend to rule out, since they trivially correspond to inconsistencies.

Definition 7 (Kernel non-admissible coloring)

Given kernel logic program Π , a coloring $\nu : V \rightarrow \{\text{green}, \text{red}\}$ is *non-admissible* (or, equivalently, *violates admissibility*) for $EDG(\Pi) = \langle V, E \rangle$ if and only if

1. $\exists i, k. \nu(v_i^{(k)}) = \text{green}$ and
 $\exists j, r. (v_j^{(r)}, v_i^{(k)}) \in E$ such that $\nu(v_j^{(r)}) = \text{green}$, or
2. $\exists i. \nu(v_i^{(k)}) = \text{red}$ and
 $\forall j, r. (v_j^{(r)}, v_i^{(k)}) \in E, \nu(v_j^{(r)}) = \text{red}$.

Taking into account that all dependencies are negative, case (1) states that green nodes cannot be adjacent and case (2) states that whenever edges to a node all come from red nodes, then that node cannot be red.

Then, straightforwardly, a coloring for $EDG(\Pi)$ is admissible unless it is not admissible.

Example 2 What are the admissible colorings for $EDG(\Pi_1)$ in Example 1?

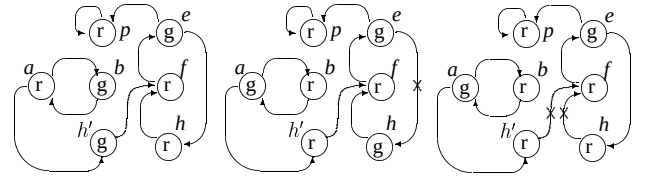


Figure 2: An admissible coloring of $EDG(\Pi_1)$ (on the left) and two not admissible ones (center and right, resp.) of $EDG(\Pi_1)$ with g=green, r=red and X=admissibility violation.

In the center coloring of Figure 2, edge $\langle e, h \rangle$ violates admissibility. In fact, it corresponds to rule $h \leftarrow \text{not } e$ in Π_1 . If e is true/green, then by that rule, h cannot be concluded true/green. As a matter of fact, both e and h are true in an answer set of Π_1 , but the truth of h comes from, intuitively, labeling h' green in the first coloring. In the right coloring

above, admissibility is violated by edges $\langle h', f \rangle$ and $\langle h, f \rangle$ which, together, represent the rule $f \leftarrow \text{not } h$ of Π_1 . When all h s are red, we conclude h false and –by the above rule–, f true/green.

As discussed in [Brignoli *et al.*, 1999], for kernel programs an admissible coloring corresponds to an interpretation which is an answer set and, vice versa, an interpretation which is an answer set corresponds to an admissible coloring. Admissible colorings correspond to *kernels* of the *EDG*.

Other kinds of admissible colorings can be defined, for checking other properties of the program. For instance, we might check whether the program is odd-cycle free, or that cycles have a maximum length, and so on.

6 Program Analysis: Consistency of General Programs

We now extend the notion of admissible colorings to general programs. Differently from kernel programs, there are several issues to be considered in the definition of admissible coloring, again given in terms of non-admissible coloring.

- There are atoms that occur in Π but have no defining rules. Every such atom, say d_i , is false in every answer set and is represented in $EDG(\Pi)$ by a vertex d_i^0 , that must be colored red (cannot be green).
- There can be unit rules (facts) in Π , i.e., rules with empty conditions. The vertex corresponding to the conclusion of a unit rule, say a_i^r , may have siblings, but has no incoming edges. Such a vertex must be colored green.
- Positive conditions: at least one sibling for each positive condition must be green; otherwise, the conclusion of the rule cannot be green.
- Negative conditions: like for kernel programs, all siblings for each negative condition must be red; otherwise, the conclusion of the rule cannot be green.

Definition 8 (Non-admissible coloring)

Given program Π , a coloring $\nu : V \rightarrow \{\text{green}, \text{red}\}$ is non-admissible (or, equivalently, violates admissibility) for $EDG(\Pi) = \langle V, E \rangle$ if and only if

1. $\exists i, k. \nu(v_i^k) = \text{green}$ and one of the following conditions hold:
 - (i) $\exists j, r. (v_j^r, v_i^k, -) \in E$ such that $\nu(v_j^r) = \text{green}$,
 - (ii) $\exists j, s. (v_j^s, v_i^k, +) \in E$ such that $\forall h. \nu(v_j^h) = \text{red}$, or
2. $\exists i, k. \nu(v_i^k) = \text{red}$ and one of the following conditions hold:
 - (i) $\forall j, r. (v_j^r, v_i^k, -) \in E, \nu(v_j^r) = \text{red}$ or
 - (ii) $\forall j, s. (v_j^s, v_i^k, +) \in E, \exists h. \nu(v_j^h) = \text{green}$.
3. $\exists i, k. k \geq 1$ and v_i^k has no incoming edges and $\nu(v_i^k) \neq \text{green}$.
4. $\exists i. \nu(v_i^0) \neq \text{red}$.

As the next issue to be considered, general logic programs have the problem of positive cycles (for short, “loops”). This problem that has been discussed in [Lifschitz and Razborov,

2006], where the number of loops has been shown to be exponential in the worst case. In [Gebser and Schaub, 2005] however, it has been shown that for program analysis it suffices to consider “elementary loops” (that correspond to those that we have called “simple cycles”), as in fact every general loop is a union of elementary ones (though their number can be still exponential). Loops correspond to *unfounded sets* of atoms, i.e., sets of atoms whose truth is unsupported. For instance, atoms a and b form such a set w.r.t. program $a \leftarrow b. b \leftarrow a$.

In fact, assuming each one true would lead one to conclude truth of the other one, while the empty set is an answer set of this program. Recall in fact that an answer set S is a minimal model of the given program and that any atom a in S must be *supported* [Marek and Truszczyński, 1991], i.e., a must be the conclusion of a rule whose conditions are true w.r.t. S .

By exploiting the results of [Gebser and Schaub, 2005] we are able to restrict to simple positive cycles. Following our approach, a simple positive cycle becomes “harmless”, in the sense that some of the composing atoms may be true in some answer set, only if it has an active OR handle.

The program: $\{a \leftarrow b. b \leftarrow a. b \leftarrow \text{not } d.\}$ has answer set $\{a, b\}$ since the second rule has the active OR handle *not d*. In an admissible coloring, an active OR handle means that an atom of the cycle (here b) has a sibling that is colored green. Then, its truth is concluded elsewhere (here from the falsity of d , which corresponds to a red vertex d^0), thus “breaking” the loop.

Definition 9 (Grounded coloring)

Given program Π , a coloring $\nu : V \rightarrow \{\text{green}, \text{red}\}$ is grounded for $EDG(\Pi) = \langle V, E \rangle$ if for each positive cycle C such that $\forall v_i^k \in C, \nu(v_i^k) = \text{green}$, there exists $v_i^r, r \neq k$, such that $\nu(v_i^r) = \text{green}$.

Then, in a grounded coloring: either it does not exist a positive cycle C such that all its composing atoms are colored green, and this amounts to assuming that the coloring provides each positive cycle with an active AND handle; or, if an all-green positive cycle exists, one sibling of some of its vertices must be green as well, thus providing an active OR handle.

A coloring for $EDG(\Pi)$ is admissible if: (i) it is not non-admissible; (ii) it is grounded. All previous results for kernel programs can be rephrased for the general case, except one: for non-kernel programs, an admissible coloring is not a kernel of the *EDG*. This is because we may have chains of positive dependencies that create paths where all vertices have the same color.

Definition 10 Let Π be an arbitrary program with associated extended dependency graph $EDG(P)$. The coloring ν of $EDG(\Pi)$ corresponding to an interpretation $\mathcal{I}$ of Π is defined so that, for each vertex labeled $a_i^{(k)}$, $\nu(a_i^{(k)}) = \text{green}$ if and only if $a \in \mathcal{I}$.

Let us define the opposite correspondence.

Definition 11 An interpretation $\mathcal{I}$ of program Π corresponds to a coloring of $EDG(\Pi)$ where: (i) for each atom $a \in \mathcal{I}$ and for each rule, say the k -th, defining a in Π , vertex $a_i^{(k)} \in EDG(\Pi)$ corresponding to this rule is colored green

only if for every atom c occurring in the positive conditions of this rule (if any), $c \in \mathcal{I}$ and for every atom c occurring in the negative conditions of this rule (if any), $c \notin \mathcal{I}$; (ii) all the other vertices are colored red.

Proposition 3 An interpretation $\mathcal{I}$ is an answer set of program Π if and only if it corresponds to an admissible coloring of $EDG(\Pi)$.

7 Program Debugging and Development

In this section we aim at convincing the reader about the potential usefulness of isomorphic graph representations as a software engineering tool for debugging and modifying programs. At the moment, we are able to make our point by means of some basic cases. A more deep analysis and the development of a practical methodology is a subject of future work.

For the sake of conciseness, let us consider kernel logic programs and their EDG 's. However, the discussion could be rephrased to general logic programs and their EDG 's, at the expense of coping with (chains of) positive dependencies.

Let us first of all present an example related to debugging. Consider again program Π_3 of Example 1, and its EDG which is on the left of Figure 1. The rule for p is $p \leftarrow \text{not } p, \text{not } e$ with inconsistency on p , and AND handle $\text{not } e$. The handle is represented by edge $\langle e, p \rangle$. If this handle were active, it would force p to be false.

On the EDG , it is easy to see that consistency can be regained for instance by replacing edge $\langle e, p \rangle$ with edge $\langle a, p \rangle$. This means replacing the existing rule $p \leftarrow \text{not } p, \text{not } e$ with $p \leftarrow \text{not } p, \text{not } a$, thus obtaining a modified program Π'_3 . By choosing answer set $\{a\}$ for the even cycle, handle $\text{not } a$ becomes active for both the odd cycles, removing inconsistencies and determining answer set $\{a, f\}$.

Observe that Π'_3 belongs to the very simple class of programs that we formalize below.

In the definition that follows, two handles are of the same kind if they both are either AND or OR handles. In an even cycle composed of atoms $a_1, \dots, a_n$ with n even, atoms a_1, a_3 etc. on the one hand and atoms a_2, a_4 , etc. on the other hand are of the same kind. Notice that the two kinds of atoms correspond to the two answer sets that the even cycle has whenever it is *unbounded*, i.e., it has no incoming handle. Incoming handles, if active, “transform” the cycle into a locally stratified program with a single answer set, depending upon the handles (i.e., not necessarily coinciding with one of the two above).

Definition 12 A kernel logic program Π is called even-bounded if the following conditions hold: (i) every even cycle in Π is unbounded, i.e. it has no incoming handle; (ii) every odd cycle has an incoming handle from an even cycle; (iii) if there are two odd cycles whose incoming handles come from the same even cycle, then the two handles are either of the same kind (AND w.r.t. OR) and originate in the same kind of atom of the even cycle, or of opposite kind (AND/OR) and originate in different kinds of atoms of the even cycle.

We can now prove the following:

Proposition 4 Every even-bounded program Π is consistent, i.e., has answer sets.

Simple as it is, this is the first class of non-call-consistent programs ever appeared in the literature, which is guaranteed to have answer sets. This class is easy understandable by programmers, and can be further generalized.

Static analysis automated tools might check whether a program is even-bounded, and might be able to suggest how to fix inconsistent program fragments. These tools might, in perspective, return the possible “slicings” of the program into even-bounded subprograms. The edges connecting the slices will represent the critical points where consistency gets lost, or can be regained. A similar reasoning can be performed for other more general forms of slicing.

Automated tools might help construct programs that are consistent by construction, by composing program fragments. We give below some simple examples of sufficient conditions for obtaining consistent programs by composition. A deeper discussion of this topic is impossible for lack of space, however many more general and more subtle conditions may be defined.

For the sake of simplicity, let us assume here not to modify the rules of the fragments we intend to compose. This means that the composition is a union of the rules, and can only add new OR handles to cycles. This happens if one of the fragments has rules whose head is an atom which occurs in the other fragment. This kind of composition cannot instead add new AND handles, as this would require to add new conditions to rules. Let the two fragments be F_1 and F_2 . The conditions below make statements about the consistency of $F_3 = F_1 \cup F_2$.

- C1:** F_1 and F_2 are both even-bounded. Their union either adds no new handles or it adds new handles to some of the odd cycles, and (possibly) to some of the even cycles without out-coming handles. Then, F_3 is even-bounded.
- C2:** F_1 is even-bounded and F_2 is inconsistent and composed of unbounded odd cycles, to which F_1 provides handles. Then, F_3 is consistent and even-bounded.
- C3:** F_1 is even-bounded and F_2 is inconsistent. Their union adds new OR handles to every unconstrained odd cycle, and no new handles to even cycles with out-coming handles. Then, F_3 is consistent and even-bounded.
- C4:** F_1 and F_2 are both even-bounded. Their union adds new handles also to some even cycle C with out-coming handles. Then in general F_3 is not even-bounded and not necessarily consistent.

Similar conditions can be defined if we allow not only union of the two program, but also the introduction of new AND handles.

8 Conclusions

Graph representations of logic programs can be useful from a software engineering perspective. We have shown how, on the basis of a suitable graph representation, classes of programs with certain useful properties can be defined, and the

properties easily checked on the graph. Topics of further research are that of demonstrating effective usefulness for program analysis and construction, by defining more general classes of programs, and of experimenting the approach for developing techniques similar to program slicing with constraint solving and path conditions for answer set programming.

Acknowledgement

We much appreciated the detailed comments and suggestions given by the anonymous referees for GKR 2011.

References

- [Anger *et al.*, 2001] Christian Anger, Kathrin Konczak, and Thomas Linke. NoMoRe : A system for non-monotonic reasoning under answer set semantics. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *LP-NMR*, volume 2173 of *Lecture Notes in Computer Science*, pages 406–410. Springer, 2001.
- [Anger *et al.*, 2004] C. Anger, T. Schaub, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004. See <http://asparagus.cs.uni-potsdam.de>.
- [Baral, 2003] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
- [Brignoli *et al.*, 1999] G. Brignoli, S. Costantini, O. D’Antona, and A. Proveti. Characterizing and computing stable models of logic programs: the non-stratified case. In *Proceedings of the 1999 Conference on Information Technology, Bhubaneswar, India*, 1999.
- [Costantini and Proveti, 2005a] S. Costantini and A. Proveti. Normal forms for answer sets programming. *J. on Theory and Practice of Logic Programming*, 5(6), 2005.
- [Costantini and Proveti, 2005b] Stefania Costantini and Alessandro Proveti. Normal forms for answer sets programming. *TPLP*, 5(6):747–760, 2005.
- [Costantini *et al.*, 2003] S. Costantini, B. Intrigila, and A. Proveti. Coherence of updates in answer set programming. In G. Brewka and P. Peppas, editors, *Proceedings of the IJCAI-2003 Workshop on Nonmonotonic Reasoning, Action and Change, NRAC03*, 2003.
- [Costantini, 2006a] S. Costantini. Comparing different graph representations of logic programs under the answer set semantics. Technical Report, University of L’ Aquila, 2006.
- [Costantini, 2006b] S. Costantini. On the existence of stable models of non-stratified logic programs. *J. on Theory and Practice of Logic Programming*, 6(1-2), 2006.
- [Dimopoulos and Torres, 1996] Y. Dimopoulos and A. Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170:209–244, 1996.
- [Gebser and Schaub, 2005] M. Gebser and T. Schaub. Loops: Relevant or redundant? Number 3662 in *LNAI*. Springer Verlag, Berlin, 2005.
- [Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proc. of the 5th Intl. Conference and Symposium on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
- [Gelfond, 2007] Michael Gelfond. Answer sets. In *Handbook of Knowledge Representation, chapter 7*. Elsevier, 2007.
- [Konczak *et al.*, 2003] K. Konczak, T. Schaub, and T. Linke. Graphs and colorings for answer set programming: Abridged report. In M. De Vos and A. Proveti, editors, *Answer Set Programming: Advances in Theory and Implementation, ASP03*, Volume 78 of The CEUR Workshop Proceedings Series, 2003. <http://eur-ws.org/Vol-78/>.
- [Leone, 2007] Nicola Leone. Logic programming and non-monotonic reasoning: From theory to systems and applications. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007*, page 1, 2007.
- [Lifschitz and Razborov, 2006] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7, 2006.
- [Lifschitz, 1999] V. Lifschitz. Answer set planning. In *Proc. of the 16th Intl. Conference on Logic Programming*, pages 23–37, 1999.
- [Linke, 2001] T. Linke. Graph theoretical characterization and computation of answer sets. In *Proceedings of IJ-CAI’01*, 2001.
- [Marek and Truszczyński, 1991] V. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38(3):587–618, 1991.
- [Marek and Truszczyński, 1999] V. W. Marek and M. Truszczyński. *Stable logic programming - an alternative logic programming paradigm*, pages 375–398. Springer-Verlag, Berlin, 1999.
- [Tip, 1995] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
- [Truszczyński, 2007] M. Truszczyński. Logic programming for knowledge representation. In Verónica Dahl and Ilkka Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007*, pages 76–88, 2007.
- [VanGelder *et al.*, 1991] A. VanGelder, K.A. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Bucket and mini-bucket Schemes for M Best Solutions over Graphical Models

Natalia Flerova, Emma Rollon and Rina Dechter

Abstract

The paper focuses on finding the m best solutions of a combinatorial optimization problem defined over a graphical model (e.g., the m most probable explanations for a Bayesian network). We describe *elim-m-opt*, a new bucket elimination algorithm for solving the m -best task, provide efficient implementation of its defining combination and marginalization operators, analyze its worst-case performance, and compare it with that of recent related algorithms. An extension to the mini-bucket framework, yielding a collection of bounds for each of the m -best solutions is discussed and empirically evaluated. We also formulate the m -best task as a regular reasoning task over general graphical models defined axiomatically, which makes all other inference algorithms applicable.

1 Introduction

Given an optimization problem, the objective typically is to find an optimal solution, i.e., a solution that provides the best value of the objective function. However, in many applications it is desirable to obtain not just a single optimal solution but a set (of a given size m) of the best possible solutions. Such a set can be useful, for example, in assessing the sensitivity of the optimal solution to variation of the parameters of the problem, or when a set of diverse assignments with approximately the same cost is wanted.

Lawler [Lawler, 1972] provided a general scheme for using any optimization algorithm to solve the m -best task. Its main idea is to compute the m -best solutions by successively computing the best solution, each time using a slightly different reformulation of the original problem. This approach has been extended and improved over the years and is still one of the primary strategies to date for finding the m -best solutions. The approach used in this paper is to develop direct algorithms that avoid the repeated computation inherent in Lawler's scheme. The main idea is to integrate the m -best task into existing optimization schemes such as inference or search. In the paper we focus on graphical models and show how the well-known Bucket Elimination (BE) framework can be extended to compute the m -best solutions by a relatively simple modification of its underlying combination

and marginalization operators [Dechter, 1999] yielding algorithm *elim-m-opt*.

We analyze the complexity of *elim-m-opt*, compare it to previously developed schemes, and discuss extensions to Mini-Bucket Elimination to compute bounds on each of the m -best solutions, yielding algorithm *mbe-m-opt*. We then show that the m -best task can be formalized more broadly as a reasoning task over a general graphical model as defined axiomatically by [Shenoy and Shafer, 1990]. As a consequence, any inference algorithm for solving a reasoning task over this representation (including Bucket Elimination) is immediately applicable as a sound and complete algorithm for the m -best task. In particular, *elim-m-opt* is therefore applicable to a wide range of graphical models.

We also provide empirical analysis for *mbe-m-opt* demonstrating its effectiveness both as an exact scheme as well as for approximation.

2 Background

We consider problems expressed as graphical models. The graphical model framework provides a common formalism to model a broad spectrum of problems, and a collection of general algorithms to efficiently solve them. Examples of graphical models are Markov and Bayesian networks [Pearl, 1988], constraint networks and influence diagrams. We next follow definitions as in [Kask *et al.*, 2005].

Let $\mathbf{X} = (X_1, \dots, X_n)$ be an ordered set of variables and $\mathbf{D} = (\mathbf{D}_1, \dots, \mathbf{D}_n)$ an ordered set of domains. Domain $\mathbf{D}_i$ is a finite set of potential values for X_i . The assignment of variable X_i with $a \in \mathbf{D}_i$ is noted $(X_i = a)$. A tuple is an ordered set of assignments to different variables $(X_{i_1} = a_{i_1}, \dots, X_{i_k} = a_{i_k})$. A *complete assignment* to all the variables in $\mathbf{X}$ is called a *solution*. Let t and s be two tuples having the same instantiations to the common variables. Their join, noted $t \cdot s$, is a new tuple which contains the assignments of both t and s (this notation is also used for multiplication, so we assume the meaning will be clear from the context). If t is a tuple over a set $T \subseteq X$ and $\mathbf{S}$ is a set of variables, then $t_{[\mathbf{S}]}$ is a relational projection of t on $\mathbf{S}$.

A valuation set $\mathbf{A}$ admits two binary operations over valuations: $\otimes : \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{A}$ called combination and $\oplus : \mathbf{A} \times \mathbf{A} \rightarrow \mathbf{A}$ called addition. Both operators are associative and commutative. Typical combination operators are sum and product over numbers, and logical AND (i.e., $\wedge$) over booleans. Typical

addition operators are min, max and sum over numbers and logical OR (i.e., $\vee$) over booleans.

We denote by $\mathbf{D}_Y$ the set of tuples over a subset of variables Y , also called the *domain of Y* . Functions are defined on subsets of variables of $\mathbf{X}$, called scopes and their range is a set $\mathbf{A}$ whose elements are called *valuations*. If $f : \mathbf{D}_Y \rightarrow \mathbf{A}$ is a function the scope of f , denoted $\text{var}(f)$, is Y . In the following, we will use $\mathbf{D}_f$ as a shorthand for $\mathbf{D}_{\text{var}(f)}$.

DEFINITION 1 (combination operator). Let $f : \mathbf{D}_f \rightarrow \mathbf{A}$ and $g : \mathbf{D}_g \rightarrow \mathbf{A}$ be two functions. Their combination, noted $f \otimes g$ is a new function with scope $\text{var}(f) \cup \text{var}(g)$, s.t.

$$\forall t \in \mathbf{D}_{\text{var}(f) \cup \text{var}(g)}, (f \otimes g)(t) = f(t) \otimes g(t)$$

DEFINITION 2 (marginalization operator). Let $f : \mathbf{D}_f \rightarrow \mathbf{A}$ be a function and $W \subseteq X$ be a set of variables. The marginalization of f over W , noted $\downarrow_W f$, is a function whose scope is $\text{var}(f) - W$, s.t.

$$\forall t \in \mathbf{D}_{\text{var}(f) - W}, (\downarrow_W f)(t) = \oplus_{t' \in D_W} (t \cdot t')$$

DEFINITION 3 (graphical model). A **graphical model** is a tuple $\mathcal{M} = (X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes)$, where: $X = \{X_1, \dots, X_n\}$ is a set of variables; $\mathbf{D} = \{D_1, \dots, D_n\}$ is the set of their finite domains of values; $\mathbf{A}$ is a set of valuations ($\mathbf{A}, \otimes, \oplus$); $\mathbf{F} = \{f_1, \dots, f_r\}$ is a set of discrete functions, where $\text{var}(f_j) \subseteq X$ and $f_j : \mathbf{D}_{f_j} \rightarrow \mathbf{A}$; and $\otimes$ is the combination operator over functions (see Definition 1). The graphical model $\mathcal{M}$ represents the function $C(X) = \otimes_{f \in \mathbf{F}} f$.

DEFINITION 4 (reasoning task). A **reasoning task** is a tuple $\mathcal{P} = (X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes, \downarrow)$, where $(X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes)$ is a graphical model and $\downarrow$ is a marginalization operator (Definition 2). The reasoning task is to compute $\downarrow_X C(X)$.

For a reasoning task $\mathcal{M} = (X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes, \downarrow)$ the choice of $(\mathbf{A}, \otimes, \oplus)$ determines the combination $\otimes$ and marginalization $\downarrow$ operators over functions, and thus the nature of the graphical model and its reasoning task. For example, if $\mathbf{A}$ is the set of non-negative reals and $\otimes$ is product, the graphical model is a Markov network or a Bayesian network. If $\downarrow$ is max, the task is to compute the Most Probable Explanation (MPE), while if $\downarrow$ is sum, the task is to compute the Probability of the Evidence.

It was shown in several works (e.g. [Shenoy and Shafer, 1990]) that, when the operators of a graphical models satisfy certain axioms, inference algorithms, such as variable elimination and join tree schemes, are sound and complete for the reasoning task. In this paper we assume graphical models obeying the Shenoy-Shafer axioms.

DEFINITION 5 (A proper reasoning task). Given a graphical model $\mathcal{M} = (X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes)$, a reasoning task $\mathcal{P} = (\mathcal{M}, \downarrow)$ is **proper** if its $\otimes$ and $\downarrow$ operators satisfy the following 3 axioms:

Axiom A1. Order of marginalization does not matter. Namely, $\downarrow_{X,Y} f(x, y) = \downarrow_{Y,X} f(x, y)$.

Axiom A2. $\otimes$ is associative and commutative.

Axiom A3. $\downarrow$ distributes over $\otimes$. Namely, $\downarrow_{X,Y} f(x) \otimes f(y) = \downarrow_X f(x) \otimes \downarrow_Y f(y)$.

An important parameter of a graphical model, characterizing its complexity, is the *induced width*.

DEFINITION 6 (induced graph, induced width). The **induced graph** of a graphical model relative to ordering o is an undirected graph that has variables as its vertices. The edges of the graph are added by: 1) connecting all variables that are in the scope of the same function, 2) processing nodes from last to first in o , recursively connecting preceding neighbours of each node. The **induced width** $w(o)$ relative to ordering o is the maximum number of preceding neighbours across all variables in the induced graph. The **induced width of the graphical model** w^* is the minimum induced width of all orderings.

DEFINITION 7 (bucket elimination). Bucket elimination (BE) is an algorithmic framework that generalizes dynamic programming for many reasoning tasks [Dechter, 1999]. The input of BE (see Algorithm 1) is a reasoning task $\mathcal{P} = (X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes, \downarrow)$ and an ordering $o = (X_1, X_2, \dots, X_n)$, dictating an elimination order for BE, from last to first. Each function from $\mathbf{F}$ is placed in the bucket of its latest variable in o . The algorithm processes the buckets from X_n to X_1 , computing for each Bucket_{X_i} , noted $\mathbf{B}_i$, $\downarrow_{X_i} \otimes_{j=1}^n \lambda_j$, where λ_j are the function in the $\mathbf{B}_i$, some of which are original f_i 's and some are earlier computed messages. The result of the computation is a new function, also called message, that is placed in the bucket of its latest variable in the ordering o .

For example, algorithm *elim-opt*, which solves the optimization task, is obtained by substitution of the operators $\downarrow_X f = \max_{S-X} f$ and $\otimes_j = \prod_j$.

The message passing between buckets follows a bucket-tree structure.

DEFINITION 8 (bucket tree). Bucket elimination defines a bucket tree, where the bucket of each X_i is linked to the destination bucket of its message (called the parent bucket). A node of the bucket is associated with its bucket variable.

Algorithm 1 Bucket elimination

Input: A reasoning task $\mathcal{P} = (X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes, \downarrow)$; An ordering of variables $o = \{X_1, \dots, X_n\}$;

Output: A zero-arity function $\lambda_1 : \emptyset \rightarrow \mathbf{A}$ containing the solution of the reasoning task.

- 1: **Initialize:** Generate an ordered partition of functions in buckets $\mathbf{B}_1, \dots, \mathbf{B}_n$, where $\mathbf{B}_i$ contains all the functions whose highest variable in their scope is X_i .
 - 2: **Backward:**
 - 3: **for** $i \leftarrow n$ **down to** 1 **do**
 - 4: Generate $\lambda_i = (\otimes_{f \in \mathbf{B}_i} f) \downarrow_{X_i}$
 - 5: Place λ_i in the bucket $\mathbf{B}_j$ where j is the largest-index variable in $\text{var}(\lambda_i)$
 - 6: **end for**
 - 7: **Return:** λ_1
-

Theorem 1. [Dechter, 1999] Given a graphical model and a proper reasoning task $\mathcal{P} = (X, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes, \downarrow)$ BE is sound and complete. Given an ordering o , the time and space complexity of $\text{BE}(\mathcal{P})$ is exponential in the induced width of the ordering.

The m -best task over a graphical model is defined formally next.

DEFINITION 9 (m -best task). *The m -best task over a graphical model $\mathcal{M}$ is to find m complete assignments $\mathbf{T} = \{t_1, \dots, t_m\}$, such that $\forall t' \notin \mathbf{T} \forall t \in \mathbf{T}, C(t') \leq C(t)$. The solution is the set of valuations $\{C(t_1), \dots, C(t_m)\}$, called m -best solutions.*

Most proofs are omitted for lack of space.

3 Algorithm *elim-m-opt*

This section provides the main contribution of our paper, presenting BE algorithm for the m -best task. We derive the algorithm through an example, having in mind the MPE (most probable explanation) task in probabilistic networks.

3.1 Deriving the algorithm using an example

Consider a graphical model with four variables $\{X, Y, Z, T\}$ having the following functions (for simplicity we use unnormalized functions):

x	z	$f_1(z, x)$	y	z	$f_2(z, y)$	z	t	$f_3(t, z)$
0	0	2	0	0	6	0	0	1
0	1	2	0	1	7	0	1	2
1	0	5	1	0	2	1	0	4
1	1	1	1	1	4	1	1	3
2	0	4	2	0	8			
2	1	3	2	1	2			

Finding the m best solutions to $P(t, z, x, y) = f_3(t, z) \cdot f_1(z, x) \cdot f_2(z, y)$ can be expressed as finding Sol , defined by:

$$Sol = \underset{t, x, z, y}{\text{sort}}^m \left(f_3(t, z) \cdot f_1(z, x) \cdot f_2(z, y) \right) \quad (1)$$

where operator $\underset{s}{\text{sort}}^m f(s)$ returns the first m elements of the set $\{f(s)_{s \in D_s}\}$ or the entire set, if its size is less than m . When the argument of the operator is a subset of the function's scope, like Y in $\underset{y}{\text{sort}}^m f(x, y)$, then $\underset{y}{\text{sort}}^m$ yields a set of the m best values of $f(x, y)$ for every fixed x (by default kept ordered) $\langle f(x, y^1(x)), \dots, f(x, y^m(x)) \rangle$ where $f(x, y^j)$ is the j^{th} largest f value relative to D_Y , for a fixed $X = x$.

Since the output of the sort operators are vectors or sets, we will use *vector function* to denote a function whose range is a set (ordered set in our exposition and implementation). The *dimension of a vector function* is the size of its range vector.

Operator $\underset{x}{\text{sort}}^m f(x)$ can be viewed as applying operator $\max f(x)$ m times, while at each step generating and then removing the best element from $\{f(s)_{s \in D_s}\}$, and it inherits, therefore, its distributive properties over multiplication. Due to this distributivity, we can apply symbolic manipulation and migrate each of the functions to the left of the $\underset{x}{\text{sort}}^m$ operator over variables that are not in its scope. In our example we rewrite as:

$$Sol = \underset{t}{\text{sort}}^m \underset{z}{\text{sort}}^m \left(f_3(t, z) (\underset{x}{\text{sort}}^m f_1(z, x)) (\underset{y}{\text{sort}}^m f_2(z, y)) \right) \quad (2)$$

However, since the output of $\underset{x}{\text{sort}}^m$ is a set, to make equation 2 well defined, we need to extend the multiplication operator to vector functions as a Cartesian product between all

pair-wise elements. This yields a new binary *combination* operator denoted by $\bar{\otimes}$ between vector functions (generalizing combination with a scalar function in the obvious way), with which we rewrite expression 2 as

$$Sol = \underset{t}{\text{sort}}^m \underset{z}{\text{sort}}^m (f_3(t, z) \bar{\otimes}_x (\underset{x}{\text{sort}}^m f_1(z, x)) \bar{\otimes}_y (\underset{y}{\text{sort}}^m f_2(z, y))) \quad (3)$$

The right to left computation in the order dictated by expression 3 can be carried out by a BE algorithm (see Figure 1) as follows. We assume that original input functions extends to vector functions e.g., $\bar{f}_i$ as $\bar{f}_i(t) = \{f_i(t)\}$. Then, the algorithm processes bucket $\mathbf{B}_Y$ generating vector function $\bar{\lambda}_Y(z)$ and bucket $\mathbf{B}_X$ generating vector function $\bar{\lambda}_X(z)$:

z	$\bar{\lambda}_X(z)$	$\bar{x}$	$\bar{\lambda}_Y(z)$	$\bar{y}$
0	$\{5, 4, 2\}$	$\{1, 2, 0\}$	$\{8, 6, 2\}$	$\{2, 0, 1\}$
1	$\{3, 2, 1\}$	$\{2, 0, 1\}$	$\{7, 4, 2\}$	$\{0, 1, 2\}$

When processing $\mathbf{B}_Z$, we compute $\bar{\lambda}_Z(t) = \underset{z}{\text{sort}}^m [f_3(t, z) \bar{\otimes} \bar{\lambda}_X(z) \bar{\otimes} \bar{\lambda}_Y(z)]$. The result is a new vector function that has m^2 elements for each tuple (t, z) as shown below.

t	z	$f_3(t, z) \bar{\otimes} \bar{\lambda}_X(z) \bar{\otimes} \bar{\lambda}_Y(z)$
0	0	$\{40, 32, 30, 16, 24, 12, 10, 8, 4\}$
0	1	$\{84, 56, 48, 32, 28, 24, 16, 16, 8\}$
1	0	$\{80, 64, 60, 48, 32, 24, 20, 16, 8\}$
1	1	$\{63, 42, 36, 24, 21, 18, 12, 12, 6\}$

Applying $\underset{z}{\text{sort}}^m$ to the resulting combination we get vector function $\bar{\lambda}_Z(t)$ along with its variable assignments:

t	$\bar{\lambda}_Z(t)$	$\langle \bar{x}, \bar{y}, \bar{z} \rangle$
0	$\{84, 56, 48\}$	$\{(2, 0, 1), (0, 0, 1), (2, 1, 1)\}$
1	$\{80, 64, 63\}$	$\{(1, 2, 0), (2, 2, 0), (2, 0, 1)\}$

Finally, processing the last bucket yields the vector of m best solution costs for the entire problem and the corresponding assignments.

$\bar{\lambda}_Z(t)$	$\langle \bar{t}, \bar{z}, \bar{x}, \bar{y} \rangle$
$\{84, 80, 64\}$	$\{(0, 1, 2, 0), (1, 0, 1, 2), (1, 0, 2, 2)\}$

Since we are interested in recovering at least one complete assignment for each m -best solution, the algorithm propagates the variable assignments along with the vector messages when processing each bucket. These variable assignments are generated using the *argsort* operator defined as follows.

DEFINITION 10. *Operator $\underset{X_i}{\text{argsort}}^m f$ returns a vector function $\bar{x}_i(t)$ such that $\forall t \in D_{\text{var}(f) \setminus X_i}$, where $\langle f(t \cdot x_i^1), \dots, f(t \cdot x_i^m) \rangle$, are the m -best costs extending t to X_i .*

In words, $\bar{x}_i(t)$ is the vector of assignments to X_i that yields the m -best extensions to t .

3.2 The Algorithm definition

We next more formally define the generalized operators to be used in the bucket-elimination algorithm *elim-m-opt*. Let $\mathcal{M} = (\mathbf{X}, \mathbf{D}, \mathbf{A}, \mathbf{F}, \bar{\otimes})$ be a graphical model, over which we want to solve the m -best task. Let $2^{\mathbf{A}}$ be the set of subsets of $\mathbf{A}$, and let us call a function with range in $2^{\mathbf{A}}$ a vector function.

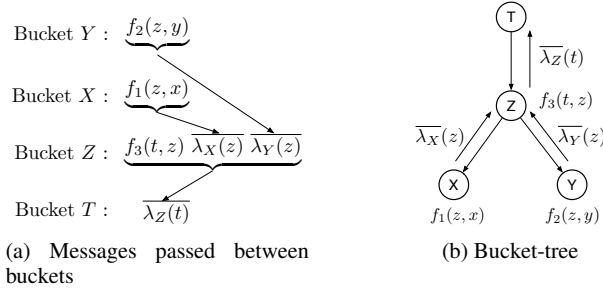


Figure 1: Example of applying *elim-m-opt*

DEFINITION 11 (combination and addition over sets). Let $S, T \in 2^A$. Their combination, noted $S \otimes T$, is the set $\{a \otimes b \mid a \in S, b \in T\}$. Their addition, noted $\text{sort}\{S, T\}$, is the set of the m -best elements in the set $S \cup T$.

DEFINITION 12 (combination and marginalization over vector functions). Let $f : D_f \rightarrow 2^A$ and $g : D_g \rightarrow 2^A$ be two vector functions. Their **combination**, noted $f \otimes g$, is a new vector function defined on scope $\text{var}(f) \cup \text{var}(g)$ s.t. $\forall t \in D_{\text{var}(f) \cup \text{var}(g)}$, $f \otimes g(t) = f(t|_{\text{var}(f)}) \otimes g(t|_{\text{var}(g)})$. The **marginalization** of f over $X_i \in \text{var}(f)$, noted $\text{sort}_{X_i}^m f$, is a function over scope $\text{var}(f) - \{X_i\}$ such that $\forall t \in D_{\text{var}(f) - X_i}$, $(\text{sort}_{X_i}^m f)(t) = \text{sort}^m \{ \bigcup_{x \in D_{X_i}} f(t \cdot x) \}$.

The bucket-elimination algorithm *elim-m-opt* is described in Algorithm 2 using the two new combination and marginalization operators of $\otimes$ and sort^m . The algorithm processes the buckets from last to first as usual. The bucket computation of a message-function (step 2) is detailed in the next subsection. The message-function associates each tuple in its domain with the m -best costs-to-go restricted to the subproblem below the bucket variable in the bucket tree. For clarity we omit the generation of actual m -best solution assignments.

While the correctness of the algorithm can be given directly, we defer to Section 4, where we show that the combination and marginalization operators just defined are an instance of a more general formulation of the m -best task as a *proper reasoning task over a proper graphical model* [Shenoy and Shafer, 1990].

3.3 The bucket processing algorithm

We will next show that the messages computed in a bucket can be obtained more efficiently than through a brute-force application of $\otimes$ followed by sort^m . Consider processing $\mathbf{B}_Z$ (see Figure 1a). A brute-force computation of $\bar{\lambda}_Z(t) = \text{sort}_z^m(f_3(z, t) \otimes \bar{\lambda}_Y(z) \otimes \bar{\lambda}_X(z))$ for each t combines $f_3(z, t)$, $\bar{\lambda}_Y(z)$ and $\bar{\lambda}_X(z)$ for $\forall z \in D_Z$ first. This results in a vector function with scope $\{T, Z\}$ having m^2 elements that we call *candidate elements* and denote by $E(t, z)$. The second step is to apply $\text{sort}_z^m E(t, z)$ yielding the desired m best elements $\bar{\lambda}_Z(t)$.

However, since $\bar{\lambda}_Y(z)$ and $\bar{\lambda}_X(z)$ can be kept sorted, we can generate only a small subset of these m^2 candidates as

Algorithm 2 elim-m-opt algorithm

Input: A set of functions $\mathbf{F} = \{f_1, \dots, f_n\}$ over scopes $\{S_1, \dots, S_n\}$; An ordering of variables $o = \{X_1, \dots, X_n\}$,
Output: A zero-arity function $\lambda_1 : \emptyset \rightarrow 2^A$ containing the solution of the m -best optimization task.

- 1: **Initialize:** Transform each function $f \in \mathbf{F}$ into a singleton vector function $h(t) = \{f(t)\}$; Generate an ordered partition of vector functions h in buckets $\mathbf{B}_1, \dots, \mathbf{B}_n$, where $\mathbf{B}_i$ contains all the functions whose highest variable in their scope is X_i .
- 2: **Backward:**
- 3: **for** $i \leftarrow n$ down to 1 **do**
- 4: Generate $\lambda_i = \text{sort}_{X_i}^m(\bigotimes_{f \in \mathbf{B}_i} f)$
- 5: Generate assignment $\bar{x}_i = \text{argsort}_{X_i}^m(\bigotimes_{f \in \mathbf{B}_i} f)$, concatenate with relevant elements of the previously generated assignment messages.
- 6: Place λ_i and corresponding assignments in the bucket of the largest-index variable in $\text{var}(\lambda_i)$
- 7: **end for**
- 8: **Return:** λ_1

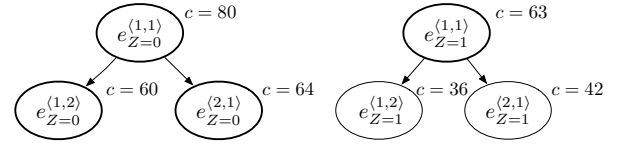


Figure 2: The explored search space for $T = 0$ and $m = 3$. The resulting message is $\bar{\lambda}_Z(1) = \{80, 64, 63\}$.

follows. We denote by $e_z^{(i,j)}(t)$ the candidate element obtained by the product of the scalar function value $f_3(t, z)$ with the i^{th} element of $\bar{\lambda}_Y(z)$ and j^{th} element of $\bar{\lambda}_X(z)$, having cost $c_z^{(i,j)}(t) = f_3(t, z) \cdot \lambda_Y^i(z) \cdot \lambda_X^j(z)$. We would like to generate the candidates $e_z^{(i,j)}$ in decreasing order of their costs while taking their respective indices i and j into account.

The *child elements* of $e_z^{(i,j)}(t)$, $\text{children}(e_z^{(i,j)}(t))$ are obtained by replacing in the product either an element $\lambda_Y^i(z)$ with $\lambda_Y^{i+1}(z)$, or $\lambda_X^j(z)$ with $\lambda_X^{j+1}(z)$, but not both.

This leads to a forest-like search graph whose nodes are the candidate elements, where each search subspace corresponds to a different value of z denoted by $G_{Z=z}$ and rooted in $e_{Z=z}^{(1,1)}(t)$. Clearly, the cost along any path from a node to its descendants is non-increasing. It is easy to see that the m best elements $\bar{\lambda}_Z(t)$ can then be generated using a greedy best-first search across the forest search space $G_{Z=0} \cup G_{Z=1}$. It is easy to show that we do not need to keep more than m nodes on the OPEN list (the fringe of the search) at the same time. The general algorithm is described in Algorithm 3. The trace of the search for the elements of cost message $\bar{\lambda}_Z(t = 1)$ for our running example is shown in Figure 2.

Proposition 1. Given a bucket of a variable X over scope S having j functions $\{\lambda_1, \dots, \lambda_j\}$ of dimension m , where m is the number of best-solutions sought and k bounds the domain size, the complexity of bucket processing is $O(k^{|S|} \cdot m \cdot j \log m)$, where $|S|$ is the scope size of S .

Complexity of elim-m-opt: Given n buckets, one for each

Algorithm 3 Bucket processing

Input: $\mathbf{B}_X$ of variable X containing a set of ordered m -vector functions $\{\lambda_1(S_1, X), \dots, \lambda_d(S_d, X)\}$

Output: m -vector function $\bar{\lambda}_X(S)$, where $S = \cup_{i=1}^d S_i - X$.

```

1: for all  $t \in D_S$  do
2:   for all  $x \in D_X$  do
3:      $OPEN \leftarrow e_{X=x}^{(1, \dots, 1)}(t)$ ; Sort OPEN;
4:   end for
5:   while  $j \leq m$ , by +1 do
6:      $n \leftarrow$  first element  $e_{X=x}^{(i_1, \dots, i_d)}(t)$  in OPEN. Remove  $n$  from OPEN;
7:      $\lambda_X^j(s) \leftarrow n$ ; {the  $j^{th}$  element is selected}
8:      $C \leftarrow$  children( $n$ ) =  $\{e_{X=x}^{(i_1, \dots, i_r+1, \dots, i_d)}(t) | r = 1..d\}$ ;
9:     Insert each  $c \in C$  into OPEN maintaining order based on its computed value. Check for duplicates; Retain the  $m$  best nodes in OPEN, discard the rest.
10:  end while
11: end for

```

variable X_i , $\mathbf{B}_i$ containing deg_i (i.e., the degree of the respective node in the bucket-tree) functions and at most w^* different variables, the total time complexity of *elim-m-opt* is $\sum_{i=1}^n O(k^{w^*} m \cdot deg_i \log m)$. Assuming $deg_i \leq deg$ and since $\sum_{i=1}^n deg_i \leq 2n$, we get complexity expression:

$$O(k^{w^*} m \cdot \log m \cdot 2n) = O(nmk^{w^*} \log m)$$

The space complexity is dominated by the size of the messages between buckets, each containing m costs-to-go for each of $O(k^{w^*})$ tuples. Having at most n such message yields the total space complexity of $O(mnk^{w^*})$. In summary:

Theorem 2 (complexity of *elim-m-opt*). *Given a graphical model $(X, D, F, \otimes)$ having n variables, an ordering o , induced-width of w^* , a bucket-tree degree bounded by deg and the domain size bounded by k , and an operator $\Downarrow = \max$, the time complexity of *elim-m-opt* is $O(k^{w^*} nm \cdot \log m)$ and its space complexity is $O(mnk^{w^*})$.*

Note that *elim-m-opt* is superior to applying Lawler's scheme [Lawler, 1972] to BE, which would lead to complexity of $O(mn^2 k^{w^*})$.

4 The M-best reasoning task

The m -best task defined via an optimization reasoning task is not itself in the form of a proper reasoning task over its graphical model. However, by some formal manipulation we can phrase it as a proper optimization reasoning task $\tilde{\mathcal{P}}(m)$ over an associated graphical model $\tilde{\mathcal{M}}(m)$. This formal exercise is worthwhile, because by doing so all inference algorithms, and in particular BE, are immediately applicable. We use definition of combination given in Definition 12.

DEFINITION 13 (m-best reasoning task). *Let $\mathcal{M} = (X, D, A, F, \otimes)$ be a graphical model and $\mathcal{P} = (\mathcal{M}, \Downarrow)$ be an optimization reasoning task. The m -best graphical model relative to $\mathcal{M}$ is $\tilde{\mathcal{M}}(m) = (X, D, A_m, \bar{F}, \otimes)$, where each function $f : D_f \rightarrow A$ in F is transformed into vector function: $\bar{f}(t) = \{f(t)\}$. The m -best reasoning task over $\tilde{\mathcal{M}}(m)$*

is $\tilde{\mathcal{P}}(m) = (\tilde{\mathcal{M}}(m), \text{sort}^m)$, where $\bar{\otimes}$ and sort^m are as in Definition 12.

We now can show that our original m -best task over a graphical model $\mathcal{M}$ can be captured by $\tilde{\mathcal{P}}(m)$ over $\tilde{\mathcal{M}}(m)$. Namely,

Theorem 3. *Let $\mathcal{M} = (X, D, A, F, \otimes)$ be a graphical model and let $\mathcal{P} = (\mathcal{M}, \max)$ be its proper optimization reasoning task. Then, the reasoning task $\tilde{\mathcal{P}}(m) = (\tilde{\mathcal{M}}(m), \text{sort}^m)$ is proper and it coincides with the m -best task over $\mathcal{M}$.*

We can now conclude that the m -best task can be solved via traditional inference algorithms for reasoning tasks, and, in particular, by Bucket Elimination.

Theorem 4. *The m -best task over a graphical model $\mathcal{M}$ can be solved by bucket-elimination as defined by $BE(\tilde{\mathcal{P}}(m))$ and it coincides with $\text{elim-m-opt}(\mathcal{M})$.*

Proof. By Theorem 3, the m -best reasoning task $\tilde{\mathcal{P}}(m)$ computes the m -best solutions of $\mathcal{M}$ and since $\tilde{\mathcal{P}}_m$ is a proper reasoning task, it can be solved exactly by BE. It is easy to see that $\text{elim-m-opt}(\mathcal{M}) = BE(\tilde{\mathcal{P}})$. $\square$

5 Mini-Bucket Elimination for m-best Task

We now demonstrate the power of *elim-m-opt* in yielding useful bounds via mini-bucket-elimination.

5.1 The Mini-Bucket for the m-best

Mini-bucket Elimination (MBE) [Dechter and Rish, 2003] is an approximation designed to avoid the space and time complexity of BE. Consider a bucket $\mathbf{B}_i$ and an integer bounding parameter z . MBE creates a z -partition $Q = \{Q_1, \dots, Q_p\}$ of $\mathbf{B}_i$, where each set $Q_j \in Q$, called *mini-bucket*, includes no more than z variables. Then, each mini-bucket is processed separately, thus computing a set of messages $\{\lambda_{ij}\}_{j=1}^p$, where $\lambda_{ij} = \Downarrow_{X_i} (\bigotimes_{f \in Q_j} f)$. In general, greater values of z increase the quality of the bound.

Theorem 5. [Dechter and Rish, 2003] *Given a reasoning task $\mathcal{P}$, MBE computes a bound on $\mathcal{P}$. Given an integer control parameter z , the time and space complexity of MBE is exponential in z .*

Algorithm *m-best MBE* (*mbe-m-opt*) (given in Figure 4) is a straightforward extension of MBE to m -best reasoning task, where the combination and marginalization operators are the ones defined over vector functions. Algorithm *mbe-m-opt* solves m -best reasoning task $\tilde{\mathcal{P}}(m)$ and its output is a m -best bound on the m -best solutions.

DEFINITION 14 (m-best bound). *Let $S = \{a_1, \dots, a_j\}$ and $T = \{b_1, \dots, b_k\}$ be two sets (i.e., $S, T \in 2^A$). S is a m -best bound of T iff $\forall 1 \leq i \leq |T|$, $b_i \leq a_i$.*

Theorem 6 (*mbe-m-opt* bound and complexity). *Given an m -best reasoning task $\tilde{\mathcal{P}}(m)$, *mbe-m-opt* computes an m -best bound on $\tilde{\mathcal{P}}(m)$. Given an integer control parameter z , the time and space complexity of *mbe-m-opt* is $O(mnk^z \log m)$ and $O(mnk^z)$, respectively, where k is the maximum domain size and n is the number of variables.*

Algorithm 4 MBE-m-opt algorithm

Input: An m -best reasoning task $\tilde{\mathcal{P}}(m) = (\mathbf{X}, \mathbf{D}, \mathbf{A}, \mathbf{F}, \otimes, \text{sort}^m)$; An ordering of variables $o = \{X_1, \dots, X_n\}$; parameter z .

Output: bounds on each of the m -best solution costs and the corresponding assignments for the expanded set of variables (i.e., node duplication).

- 1: **Initialize:** Generate an ordered partition of functions $\bar{f}(t) = \{f(t)\}$ into buckets $\mathbf{B}_1, \dots, \mathbf{B}_n$, where $\mathbf{B}_i$ along o .
- 2: **Backward:**
- 3: **for** $i \leftarrow n$ down to 1 (Processing bucket B_i) **do**
- 4: Partition functions in bucket B_i into $\{Q_{i_1}, \dots, Q_{i_l}\}$, where each Q_{i_j} has no more than z variables.
- 5: Generate cost messages $\lambda_{i_j} = \text{sort}_{X_i}^m(\otimes_{f \in Q_{i_j}} f)$ and place each in the largest index variable in $\text{var}(Q_{i_j})$
- 6: **end for**
- 7: **Return:** The set of all buckets, and the vector of m -best costs bounds in the first bucket.

5.2 Using the m -best bound to tighten the first-best bound

Here is a simple, but quite fundamental observation: whenever upper or lower bounds are generated by solving a relaxed version of a problem, the relaxed problem's solution set contains all the solutions to the original problem. We next discuss the ramification of this observation.

Proposition 2. *Given the m -best solutions costs generated by mbe-m-opt (for clarity we consider MPE problem, the results can be extended for other reasoning tasks) $\tilde{C} = \{\tilde{p}_1 \geq \tilde{p}_2 \geq \dots, \geq \tilde{p}_m\}$, let p^{opt} be the optimal value (the probability of the most probable explanation) and let j_0 be the first index such that $\tilde{p}_{j_0} = p^{\text{opt}}$, or else we assign $j_0 = m + 1$. Then, if $j_0 > m$, $\tilde{p}_m$ is an upper bound on p^{opt} , which is as tight or tighter than all other $\tilde{p}_1, \dots, \tilde{p}_{m-1}$. In particular $\tilde{p}_m$ is tighter than the bound $\tilde{p}_1$.*

Proof. Let $\tilde{C} = \{\tilde{p}_1 \geq \tilde{p}_2 \geq \dots, \geq \tilde{p}_{N_1}\}$ be an ordered set of probabilities of all tuples over the relaxed problem (with duplicate variables). By the nature of any relaxation, $\tilde{C}$ must also contain all the probability values associated with solutions of the original problem denoted by $C = \{p_1 \geq \dots \geq p_{N_2}\}$. Therefore, if j_0 is the first index such that $\tilde{p}_{j_0}$ coincides with p^{opt} , then clearly for all $i < j_0$, $p^{\text{opt}} \leq \tilde{p}_i$ with $\tilde{p}_{j_0-1}$ being the tightest upper-bound. Also, when $j_0 > m$ we have $\tilde{p}_m \geq p^{\text{opt}}$ $\square$

In other words if $j \leq m$, we already have optimal value, otherwise we can use $\tilde{p}_m$ as our better upper bound. Such tighter bounds would be useful during search algorithm such as A*. It is essential therefore to decide efficiently if a bound coincides with the exact optimal cost. Luckily the nature of the MBE relaxation supplies us with an efficient decision scheme.

Proposition 3. *Given a set of bounds produced by mbe-m-opt $\tilde{p}_1 \geq \tilde{p}_2 \geq \dots \geq \tilde{p}_m$, deciding if $\tilde{p}_j = p^{\text{opt}}$ can be done efficiently.*

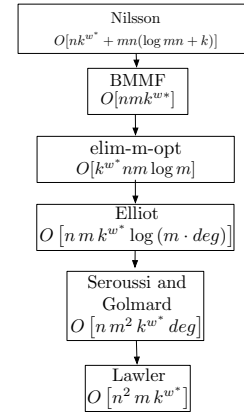


Figure 3: Complexity comparison. A parent node in the graph has a better complexity than its children.

Proof. mbe-m-opt provides both the bounds on the m -best costs and for each bound a corresponding tuple, where assignments to duplicated variables is maintained. The first assignment from these m -best bounds (going from largest to smallest) that corresponds to a tuple whose duplicate variables are assigned identical value, is optimal. And, if no such tuple is observed the optimal value is smaller than $\tilde{p}_m$. Since the above tests require just $O(nm)$ steps applied to m -best assignments already obtained in polytime, the claim follows. $\square$

6 Related work

Comparing with exact schemes. Lawler's approach, whose complexity is $O(nmT(n))$, where $T(n)$ is the complexity of finding a single best solution, was applied by [Nilsson, 1998] to a join-tree. Instead of solving reformulated problems from scratch in each iteration, as Lawler does, Nilsson utilizes the results from previous computations, achieving worst case complexity of $O(mT(n))$. If applied to a bucket-tree Nilsson's algorithm is superior to all other schemes mentioned here, with run time of $O(nk^{w*} + mn \log(mn) + mnk)$. More recently Yanover and Weiss [Yanover and Weiss, 2004] developed a belief propagation approximation scheme for loopy graphs, called BMMF which also finds solutions iteratively. At each iteration BMMF uses loopy Belief Propagation to solve two new problems obtained by restricting the values of certain variables. When applied to junction tree it can function as an exact algorithm with complexity $O(mnk^{w*})$.

Two algorithms based on dynamic programming, similar to elim-m-opt, are [Seroussi and Golmard, 1994] and [Elliott, 2007]. Seroussi and Golmard extract the m solutions directly, by propagating the m best partial solutions along a junction tree. Given a junction tree with p cliques and branching degree deg , the complexity of the algorithm is $O(m^2 p \cdot k^{w*} \text{deg})$. Elliott [Elliott, 2007] propagates the m best partial solutions along a representation called Valued And-Or Acyclic Graph, i.e., smooth deterministic decomposable negation normal form (sd-DNNF) [Darwiche, 2001]. The

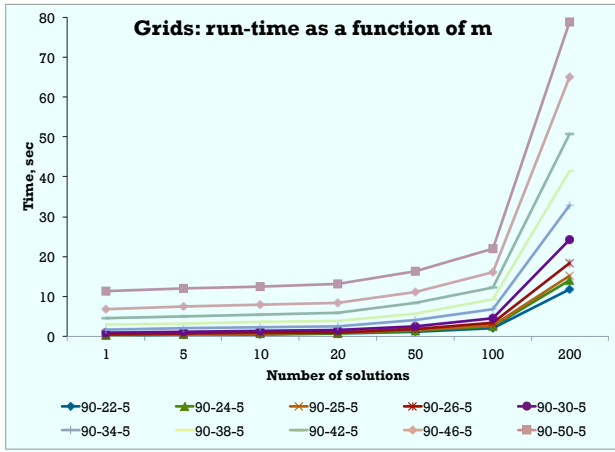


Figure 4: Run time as a function of number of solutions on grid instances

complexity of Elliot’s algorithm is $O(nk^{w^*}m \log(mdeg))$.

Figure 3 depicts the dominance relationships between the time complexities of the exact algorithms discussed, when specialized to a bucket tree. Clearly our *elim-m-opt* algorithm does not boast the best complexity, being dominated by the algorithms [Nilsson, 1998; Yanover and Weiss, 2004]. However, it demonstrates the direct applicability of established inference schemes to the generalized formulation of the m best solution problem as the *m-best reasoning problem*. Moreover, the main significance *elim-m-opt* is in the natural extension to an approximation scheme with guarantees on the solution quality that provides flexible trade off between accuracy and complexity.

Comparing with approximation schemes. In addition to BMMF, another extension of Nilsson’s and Lawler’s idea that yields an approximation scheme is an algorithm called STRIPES by [Fromer and Globerson, 2009]. They focus on m -MAP problem over binary Markov networks, solving each new subproblem by an LP relaxation. The algorithm solves the task exactly if the solutions to all LP relaxations are integral, and provides an upper bound of each m MAP assignments otherwise. In contrast, our algorithm *mbe-m-opt* can compute bounds over any graphical model (not only binary) and over a variety of m -best optimization tasks.

7 Empirical demonstrations

All experiments assume solving m -best MPE task. We evaluated empirically algorithm *mbe-m-opt* with $m = \{1, 5, 10, 20, 50, 100, 200\}$ and with z -bound 10 on two sets of instances. The first set contained grid instances with a hundred to 2.5 thousand variables and tree-width from 12 to 50, the second - pedigree instances with several hundred variables and tree width from 15 to 30. Those instances were taken from the UAI 2008 evaluation. For clarity and space reasons we present only a subset of instances illustrating typical behaviour.

Figures 4 and 6 present the dependence of the run-time on

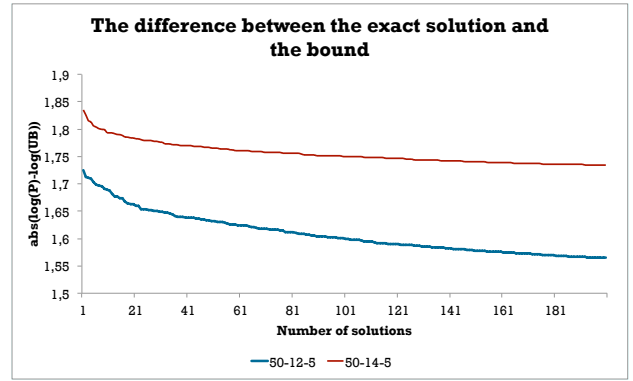


Figure 5: Accuracy of upper bound for different solutions on grid instances

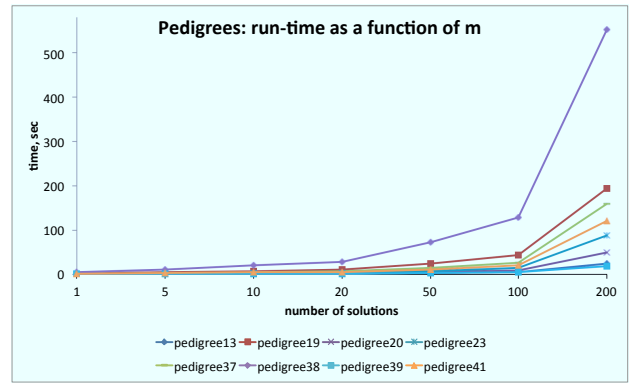


Figure 6: Run-time performance of *mbe-m-opt* on pedigrees

m , for a few selected instances. Figure 5 shows the change in accuracy as a function of m . Accuracy is measured by the absolute values of the difference between the optimal solution and the bounds on the j^{th} solution up to $m = 200$ solutions. For these grid instances as j increases, the bound on the cost of the j^{th} solution slowly approaches the exact best solution. This demonstrates that there is a potential of improving the bound on the optimal assignment using the m -best bounds as discussed in Section 5.2.

We carried some comparison with BMMF by [Yanover and Weiss, 2004] on randomly generated 10 by 10 grids. The run times of the algorithms are not comparable since our algorithm is implemented in C and BMMF in Matlab, which is inherently slower. For most instances that *mbe-m-opt* can solve exactly in under a second, BMMF takes more than 5 minutes. The algorithms also differ in the nature of the outputs: BMMF provides approximate solutions with no guarantees while *mbe-m-opt* generates bounds on all the m -best solutions. Still some information can be learned from viewing the two algorithms side by side as is demonstrated by a typical result in Figure 7. We know that in this case the solutions obtained with z -bound equal to 1000 are exact, while

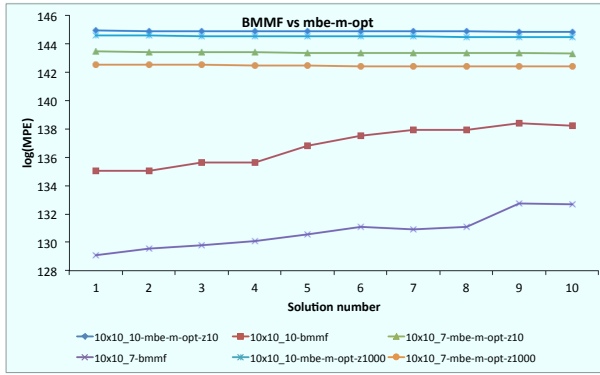


Figure 7: Comparison of *mbe-m-opt* with *z*-bounds 10 and 1000 vs. BMMF on random 10 by 10 grids

z-bound equal to 10 yields an upper bound. BMMF outputs significantly less accurate results than *mbe-m-opt* with even a low *z*-bound. Admittedly, these experiments are quite preliminary and not conclusive.

8 Conclusions

We presented a new bucket-elimination algorithm for solving the *m*-best task over a graphical model, analyzed its performance and related it to other approaches in the literature. We proved the algorithm’s correctness by formally showing that the *m*-best task can be formulated as a reasoning task satisfying the Shenoy-Shafer axioms and, therefore, all inference algorithms and in particular our algorithm, *elim-m-opt*, are immediately applicable, sound and complete.

The significance of the proposed algorithm is primarily in providing an inference framework for the *m*-best task that can both suggest approximation schemes and yield heuristic advice. Indeed, optimization tasks that seek a single optimal solution are solved far more effectively by search (e.g., branch and bound and best-first search), than by variable elimination, because they can benefit from the bounding power of the guiding cost function. It is also likely that search will be more effective for *m*-best task. The promise of the *elim-m-opt* inference algorithm is in its potential to yield viable lower- and upper-bounds for the *m*-best solutions via the mini-bucket algorithm, as we discussed.

Furthermore, it could also lead to loopy propagation message-passing schemes that are now the most common way for approximations in graphical models, since those schemes are relaxation of exact message-passing schemes such as bucket-elimination. In particular, our algorithm can be extended into a loopy max-prod for the *m*-best task. This approach will yield a direct loopy-propagation for the *m*-best reasoning problem, while the approach by Yanover and Weiss uses loopy max-prod for solving a sequence of optimization problems in the style of Lawler’s approach. Moreover, all such approximation extensions would be applicable to the broad range of graphical models captured by the unifying framework of Shenoy and Shafer. Future work will focus on such extensions and on empirical evaluations of the emerging

schemes.

The empirical analysis we provided is only preliminary. Yet it shows that *mbe-m-opt* scales even better than worst-case predict as a function of *m*. Comparison with other exact and approximation algorithms is left for future work.

Acknowledgments

This work was partially supported by NSF grants IIS-0713118 and IIS-1065618 and NIH grant 5R01HG004175-03. Thanks to the reviewers for their helpful feedback.

References

- [Darwiche, 2001] A. Darwiche. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4):608–647, 2001.
- [Dechter and Rish, 2003] R. Dechter and I. Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM (JACM)*, 50(2):107–153, 2003.
- [Dechter, 1999] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1):41–85, 1999.
- [Elliott, 2007] P.H. Elliott. Extracting the K Best Solutions from a Valued And-Or Acyclic Graph. Master’s thesis, Massachusetts Institute of Technology, 2007.
- [Fromer and Globerson, 2009] M. Fromer and A. Globerson. An LP View of the M-best MAP problem. *Advances in Neural Information Processing Systems*, 22:567–575, 2009.
- [Kask et al., 2005] Kalev Kask, Rina Dechter, Javier Larrosa, and Avi Dechter. Unifying cluster-tree decompositions for automated reasoning. *Artificial Intelligence Journal*, 2005.
- [Lawler, 1972] E.L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.
- [Nilsson, 1998] D. Nilsson. An efficient algorithm for finding the M most probable configurations in probabilistic expert systems. *Statistics and Computing*, 8(2):159–173, 1998.
- [Pearl, 1988] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.
- [Seroussi and Golmard, 1994] B. Seroussi and JL Golmard. An algorithm directly finding the K most probable configurations in Bayesian networks. *International Journal of Approximate Reasoning*, 11(3):205–233, 1994.
- [Shenoy and Shafer, 1990] P. Shenoy and G. Shafer. Axioms for probability and belief-function propagation. *Uncertainty in Artificial Intelligence*, 4:169–198, 1990.
- [Yanover and Weiss, 2004] C. Yanover and Y. Weiss. Finding the M Most Probable Configurations Using Loopy Belief Propagation. In *Advances in Neural Information Processing Systems 16*. The MIT Press, 2004.

Supporting Argumentation Systems by Graph Representation and Computation

Jérôme Fortin

Université Montpellier II
fortin@supagro.inra.fr

Rallou Thomopoulos

INRA
rallou.thomopoulos@supagro.inra.fr

Jean-Rémi Bourguet

Université Montpellier III
bourguet@lirmm.fr

Marie-Laure Mugnier

Université Montpellier II
mugnier@lirmm.fr

Abstract

Argumentation is a reasoning model based on arguments and on attacks between arguments. It consists in evaluating the acceptability of arguments, according to a given semantics. Due to its generality, Dung's framework for abstract argumentation systems, proposed in 1995, is a reference in the domain. Argumentation systems are commonly represented by graph structures, where nodes represent arguments and edges attacks between arguments. However beyond this graphical support, graph operations have not been considered as reasoning tools in argumentation systems. This paper proposes a conceptual graph representation of an argumentation system and a computation of argument acceptability relying on conceptual graph defaults.

1 Introduction

Argumentative reasoning is based on the construction and the evaluation of interacting arguments. Most of the existing argumentation models are grounded on the abstract argumentation framework proposed by Dung in [Dung, 1995]. It consists of a set of arguments and a binary relation on this set, expressing conflicts among arguments. Argumentation systems are commonly represented by graph structures, where nodes represent arguments and edges attacks between arguments.

Two main tools are currently available for argument visualization: Araucaria [Araucaria, website] and Carneades [Carneades, website]. These tools facilitate the visual display of arguments and in particular their structure (e.g. premisses and conclusion), however they are not reasoning tools. This paper deals with the reasoning issue, moreover it regards a different level, since it does not focus on argument structure, but on the representation of a whole argument base and on the computation of maximal acceptable sets of arguments from it.

The chosen formalism is the conceptual graph (CG) model [Chein and Mugnier, 2009]. Indeed, it is the only AI formalism that combines a graphical representation and graph-based operations, together with an equivalent logical interpretation, providing a well-founded graphical and logical reasoning model. A recent study [de Moor *et al.*, 2009] has considered the representation of "argument maps" in the concep-

tual graph model. It designs an architecture to map argument maps and conceptual graphs, to allow visualizing the arguments put forward by different actors and examining them through queries. The authors present the advantages of using conceptual graphs, and highlight that, beyond this first step, further reasoning features remain to be explored.

In this paper, we focus on this reasoning issue. Non-monotonic reasoning is used as a computational tool to determine maximal acceptable sets of arguments. It is instantiated by conceptual graph defaults [Baget *et al.*, 2009; Baget and Fortin, 2010]. The paper is organized as follows. Section 2 presents the background on argumentation systems and conceptual graphs. Section 3 introduces a representation of argumentation systems in the conceptual graph formalism. Section 4 proposes a way of computing four kinds of acceptable sets of arguments, namely the naive, the admissible, the preferred, and the stable sets, according to the semantics of argumentation systems.

2 Background

This section introduces fundamental notions of argumentation systems and the basic conceptual graph formalism as well as one of its extensions, namely CG defaults.

2.1 Argumentation systems

Three main steps can be distinguished in an argumentation process: 1) constructing *arguments* and counter-arguments, 2) evaluating the collective *acceptability* of sets of arguments, and 3) drawing *justified conclusions*. In [Dung, 1995], an abstract argumentation framework is defined as follows.

Definition 1 (Dung's argumentation framework). *An argumentation framework is a pair $AF = \langle A, R \rangle$ where A is a set of arguments and $R \subseteq A \times A$ is an attack relation. An argument α attacks an argument β iff $(\alpha, \beta) \in R$.*

In the above definition, arguments are abstract entities, whose origin and structure are left unknown. With each argumentation system is naturally associated a directed graph whose nodes are the arguments and edges represent the attack relation between them.

Definition 2 (Conflict-free, Defense). *Let $B \subseteq A$.*

- *B is conflict-free iff $\nexists \alpha, \beta \in B$ such that $(\alpha, \beta) \in R$.*

- B defends an argument $\alpha \in B$ iff for each argument $\beta \in A$, if $(\beta, \alpha) \in R$, then $\exists \gamma \in B$ such that $(\gamma, \beta) \in R$.

Among all the conflicting arguments, one has to select acceptable subsets of arguments for inferring conclusions and making decision. In [Dung, 1995; Bondarenko *et al.*, 1997], several semantics for the notion of acceptability have been proposed. For the purpose of this paper, we only recall naive, admissible, preferred, and stable semantics. Other semantics (e.g. complete, grounded) are not presented here.

Definition 3 (Acceptability semantics). Let $B \subseteq A$.

- B is a naive set iff it is a maximal (w.r.t. set-inclusion) conflict-free set.
- B is an admissible set iff it is a conflict-free set that defends all its elements.
- B is a preferred set iff it is a maximal (w.r.t. set-inclusion) admissible set.
- B is a stable set iff it is a maximal (w.r.t. set-inclusion) conflict-free set such that every element of $(A \setminus B)$ is attacked by an element in B .

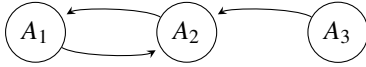


Figure 1: Graph of attacks

Figure 1 presents a graph of attacks, where A_1 and A_2 attack each other and argument A_3 attacks A_2 . There are two naive sets $S_1 = \{A_1, A_3\}$ and $S_2 = \{A_2\}$. S_1 is admissible, preferred and stable, while S_2 has none of these properties.

Note that a stable set is *a fortiori* a preferred set (which is itself an admissible set). Given an argumentation framework, there is always a naive (resp. admissible, preferred) set of arguments, possibly equal to the empty set. This is not true for the stable semantic. For example, consider an argumentation framework with three arguments A_1, A_2, A_3 , such that A_1 attacks A_2 , A_2 attacks A_3 and A_3 attacks A_1 (hence a cycle of attacks). It does not have any stable set of arguments because, for any conflict-free set, there is at least one argument outside this set that is not attacked by any argument of the set.

2.2 The Conceptual Graph Formalism

The conceptual graph (CG) formalism [Sowa, 1984; Chein and Mugnier, 2009] is a knowledge representation and reasoning formalism based on labelled graphs. In its simplest form, a CG knowledge base is composed of two parts: the *support*, which encodes terminological knowledge –and constitutes a part of the represented domain ontology– and *basic conceptual graphs* built on this support, which express assertional knowledge, or *facts*. The knowledge base can be further enriched by other kinds of knowledge built on the support. Here, we will consider two kinds of *rules*: “usual rules” and CG defaults, which lead to non-monotonic reasoning.

The support. It provides the ground vocabulary used to build the knowledge base, i.e., the types of concepts, denoted

by $\mathcal{T}_C$ and the types of relations that can link concept instances, denoted by $\mathcal{T}_R$. $\mathcal{T}_C$ is partially ordered by a *kind of relation*, with $\top$ being its greatest element. $\mathcal{T}_R$ is also partially ordered by a *kind of relation*, with any two comparable relation types having necessarily the same arity (i.e., number of arguments). Each relation has a signature that specifies its arity and the maximal concept type of each of its arguments.

Figure 2 shows the sets of concept types and of relation types (each with their signature) used in the application. $\mathcal{T}_R$ is partitioned into two sets, $\mathcal{T}_{R_1}$ and $\mathcal{T}_{R_2}$. $\mathcal{T}_{R_1}$ is a set of unary relations representing properties of sets (of arguments): being a naive set (Acc^{na}), an admissible set (Acc^{ad}), a preferred set (Acc^{pr}), a stable set (Acc^{st}) or a non-stable set ($nonAcc^{pr}$). $\mathcal{T}_{R_2}$ is a set of binary relations, that contains for instance the attack relation $\mathcal{R}$ (with signature (Arg, Arg)).

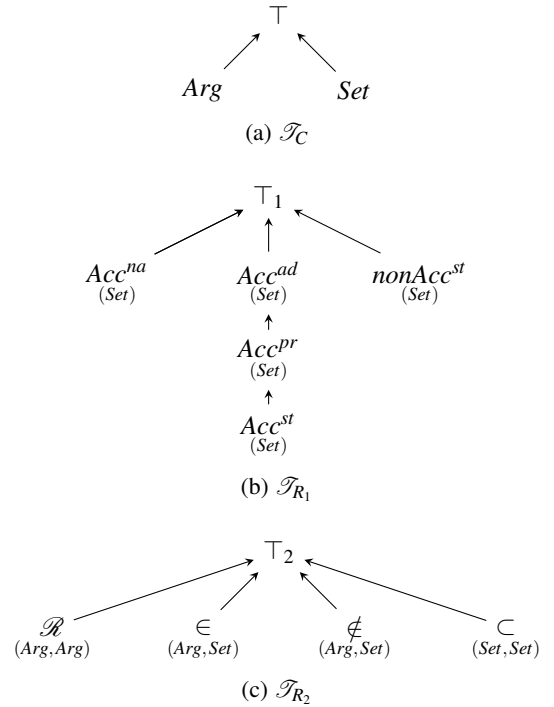


Figure 2: CG Support with concept and relation types for representing argumentation systems

Basic conceptual graphs. A basic CG is a bipartite graph composed of: (i) a set of *concept nodes* (pictured as rectangles), which represent entities, attributes, states or events; (ii) a set of *relation nodes* (pictured as ovals), which express the nature of relationships between concept nodes; (iii) a set of *edges* linking relation nodes to concept nodes; (iv) a *labelling* function, which labels each node or edge: the label of a concept node is a pair $t : m$, where t is a concept type and m is a marker, the label of a relation node is a relation type, and the label of an edge is its rank in the total order on the arguments of the incident relation node; furthermore, the relation type signatures have to be satisfied: the number of edges incident to a relation node is equal to the arity of its type r , and the

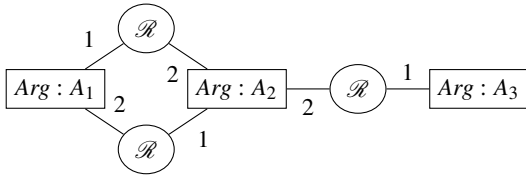


Figure 3: Basic conceptual graph

concept type assigned to its neighbor by an edge labelled i is less or equal to the i^{th} element of the signature of r . The marker of a concept node can be either an identifier referring to a specific individual (for instance A_1 of type Arg in Figure 3) or the generic marker (noted $*$) referring to an unspecified instance of the associated concept type (see for instance in Figure 4). A basic CG without occurrence of the generic marker is said to be totally instantiated. Figure 3 shows a totally instantiated basic CG built on the support of Figure 2, which encodes the attack graph of Figure 1.

Logical translation. Conceptual graphs have a logical translation in first-order logic, which is given by a mapping classically denoted by ϕ . ϕ assigns a formula $\phi(S)$ to a support S , and a formula $\phi(G)$ to any basic CG G on this support. First, each concept or relation type is translated into a predicate (a unary predicate for a concept type, and a predicate with the same arity for a relation type) and each individual marker occurring on the graphs is translated by a constant. Then, the *kind of* relation between types of the support is translated by logical implications. For example, the fact that Acc^{st} is a specialization of Acc^{pr} (Figure 2b) is translated by: $\forall x, Acc^{st}(x) \rightarrow Acc^{pr}(x)$

Then, given a basic conceptual graph G on S , $\phi(G)$ is built as follows. A distinct variable is assigned to each node with a generic marker. An atom of the form $t(e)$ is assigned to each concept node with label $t : m$, where e is the variable assigned to this node if $m = *$, otherwise $e = m$. An atom of the form $r(e_1, \dots, e_k)$ is assigned to each relation node with label r , where e_i is the variable or the constant corresponding to the i^{th} neighbor of the relation. $\phi(G)$ is then the existential closure of the conjunction of all atoms assigned to its nodes. For instance, the logical translation of the conceptual graph represented in Figure 3 is the following: $Arg(A_1) \wedge Arg(A_2) \wedge Arg(A_3) \wedge \mathcal{R}(A_1, A_2) \wedge \mathcal{R}(A_2, A_1) \wedge \mathcal{R}(A_3, A_2)$. Note that in this case, the graph is totally instantiated, thus its logical translation has no variable.

Specialization relation, homomorphism. Any set of conceptual graphs is partially preordered by a *specialization relation*, which can be computed by a *homomorphism* (allowing the restriction of the node labels) also called projection in the conceptual graph community. The specialization relation, and thus homomorphism, between two graphs, corresponds to the logical entailment between the corresponding formulas, i.e., there is a homomorphism from G to H (both built on a support S) if and only if $\phi(G)$ is entailed by $\phi(H)$ and $\phi(S)$

(e.g. [Chein and Mugnier, 2009])¹.

Basic CG rules. Basic CG rules [Salvat and Mugnier, 1996] are an extension of basic CGs. A CG rule (notation: $R = (H, C)$) is of the form “if H then C ”, where H and C are two basic CG (respectively called the *hypothesis* and the *conclusion* of the rule), which may share some concept nodes. Formally, it can be defined as a single bicolored basic CG, as illustrated in Figure 4: the hypothesis is composed of the white nodes; the conclusion is induced by the black nodes and the white concept nodes that are linked to a black relation node; these latter nodes that belong both to the hypothesis and the conclusion are called *frontier* nodes. Intuitively, this rule says that “if an argument x attacks an argument y that attacks an argument z , then x defends z ”.

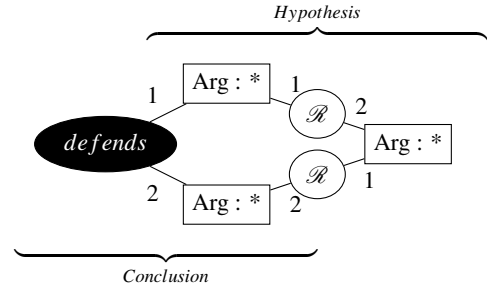


Figure 4: CG rule

A rule R is applicable to a basic CG G if there is a homomorphism from its hypothesis to G . Let π be such a homomorphism. Then, the *application of R on G according to π* produces a basic CG obtained from G by adding the conclusion of R according to π , i.e., merging each frontier node c of the added conclusion with the node of G that is image of c by the homomorphism π . For instance, the rule in Figure 4 can be applied twice to the basic CG of Figure 3, which allows to infer that A_3 defends A_1 and that A_1 defends itself.

The mapping ϕ to first-order logic is extended to CG rules. Let $R = (H, C)$ be a CG rule, and let $\phi'(R)$ denotes the conjunction of atoms associated with the basic CG underlying R (all variables are kept free). Then, $\phi(R) = \forall x_1 \dots \forall x_k (\phi'(H) \rightarrow (\exists y_1 \dots \exists y_q \phi'(C)))$, where $\phi'(H)$ and $\phi'(C)$ are the restrictions of $\phi'(R)$ to the nodes of H and C respectively, $x_1, \dots, x_k$ are all the variables appearing in $\phi(H)$ and $y_1, \dots, y_q$ are all the variables appearing in $\phi(C)$ but not in $\phi(H)$. For example, the rule of Figure 4 is translated as follows: $\forall x \forall y \forall z Arg(x) \wedge Arg(y) \wedge Arg(z) \wedge \mathcal{R}(x, y) \wedge \mathcal{R}(y, z) \rightarrow defends(x, z)$.

The rule application mechanism is logically sound and complete: given a set of rules $\mathcal{R}$, basic CGs G and H (representing for instance a query and a set of facts), G is entailed by the logical formulas assigned to $\mathcal{R}$, H and S (the support) if and only if there is a sequence of rule applications with rules of $\mathcal{R}$ leading from H to a basic CG H' such that there is

¹ Note that, for the completeness part, H has to be in normal form: each *individual* marker appears at most once in it.

a homomorphism from G to H' (in other words, by applying rules to H , it is possible to obtain H' which entails G).

When a rule is applied, it may create new individuals (one for each generic concept node in its conclusion, i.e., one for each existential variable y_i in the logical translation of the rule). In the following, we will assume that all facts (represented as basic CGs) are completely instantiated. Then, when a rule is applied, we will instantiate each new generic concept node created, by replacing its generic marker with a new individual marker (which can be seen as a skolem function, moreover without variable in this case). This will allow us to present CG defaults in a simpler way.

2.3 Conceptual Graph Defaults

A Brief Introduction to Reiter's Default logic

Let us recall some basic definitions of Reiter's default logics. For a more precise description and examples, the reader should refer to [Reiter, 1980; Brewka and Eiter, 1999].

Definition 4 (Reiter's Default theory). *A Reiter's default theory is a pair (Δ, W) where W is a set of FOL formulae and Δ is a set of defaults of form $\delta = \frac{\alpha(\vec{x}); \beta_1(\vec{x}), \dots, \beta_n(\vec{x})}{\gamma(\vec{x})}$, $n \geq 0$, where $\alpha(\vec{x})$, $\beta_i(\vec{x})$ and $\gamma(\vec{x})$ are FOL formulae for which each free variable is in the tuple of variable $\vec{x} = (x_1, \dots, x_k)$.*

The intuitive meaning of a default δ is "For all individuals $\vec{x} = (x_1, \dots, x_k)$, if $\alpha(\vec{x})$ is believed and each of $\beta_1(\vec{x}), \dots, \beta_n(\vec{x})$ can be consistently believed, then one is allowed to believe $\gamma(\vec{x})$ ". $\alpha(\vec{x})$ is called the *prerequisite*, the $\beta_i(\vec{x})$ are called the *justifications* and $\gamma(\vec{x})$ is called the *consequent*. A default is said *closed* if $\alpha(\vec{x})$, $\beta_i(\vec{x})$ and $\gamma(\vec{x})$ are all closed FOL formulae. A default theory (Δ, W) is said to be *closed* if all its defaults are closed.

Intuitively, an *extension* of a default theory (Δ, W) is a set of formulae that can be obtained from (Δ, W) while being consistently believed. More formally, an extension E of (Δ, W) is a minimal deductively closed set of formulae containing W such that for any $\frac{\alpha; \beta}{\gamma} \in \Delta$, if $\alpha \in E$ and $\neg\beta \notin E$, then $\gamma \in E$. The following theorem [Reiter, 1980] provides an equivalent characterization of extensions that we use here as a formal definition.

Theorem 1 (Extension). *Let (Δ, W) be a closed default theory and E be a set of FOL formulae. We inductively define $E_0 = W$ and for all $i \geq 0$, $E_{i+1} = Th(E_i) \cup \{\gamma \mid \frac{\alpha; \beta_1, \dots, \beta_n}{\gamma} \in \Delta, \alpha \in E_i \text{ and } \neg\beta_1, \dots, \neg\beta_n \notin E_i\}$, where $Th(E_i)$ is the deductive closure of E_i . Then E is an extension of (Δ, W) iff $E = \bigcup_{i=0}^{\infty} E_i$.*

Note that this characterization is not effective for computational purposes since both E_i and $E = \bigcup_{i=0}^{\infty} E_i$ are required for computing E_{i+1} (for more details on generating extensions, see in [Reiter, 1980; G. Brewka, 2007]). Moreover, Theorem 1 is only useful to deal with a closed default theory, which is less expressive than the general case, since no variable can be shared between the hypothesis, the conclusion or a justification. When we want to apply some non-closed default, we first have to instantiate each free variable by all the constants that may appear in the knowledge base, which yields a set of closed defaults.

Default Rules in the Conceptual Graph Formalism

We now present an extension of CG rules, which has been introduced in [Baget *et al.*, 2009; Baget and Fortin, 2010] and allows for default reasoning. It can be seen as a graphical implementation of a subset of Reiter's default logic: indeed, we restrict the kind of formulae that can be used in the three components of a default. On the other hand, we can deal directly with non-closed defaults, i.e., without instantiating free variables before processing the defaults. In Reiter's logic, the application of a default is subject to a consistency check with respect to current knowledge: each justification J has to be consistent with the current knowledge, i.e., $\neg J$ should not be entailed by it. In CG defaults, justifications are replaced by graphs called *constraints*; a constraint C can be seen as the negation of a justification: C should not be entailed by current knowledge.

Definition 5 (CG defaults). *A CG default is a tuple $D = (H, C, C_1, \dots, C_k)$ where H is called the hypothesis, C the conclusion and $C_1, \dots, C_k$ are called the constraints of the default; all components of D are themselves basic CGs and may share some concept nodes.*

Briefly said, H , C and each C_i respectively correspond to the prerequisite, the consequent and the negation of a justification in a Reiter's default.

In this paper, we will represent CG defaults by a multi-colored basic CG. As in a CG rule, the hypothesis is represented by the white nodes. The conclusion is represented by the black nodes and frontier nodes. Each constraint is represented by a different level of gray.

The intuitive meaning of a CG default is rather simple: "for all individuals $x_1 \dots x_k$, if $H[x_1 \dots x_k]$ holds true, then $C[x_1 \dots x_k]$ can be inferred provided that no $C_i[x_1 \dots x_k]$ holds true". If we can map by homomorphism the hypothesis H to a fact graph G , then we can add the conclusion of the default according to this homomorphism (as in a standard rule application), unless this homomorphism can be extended to map one of the constraints. As already pointed out, while the negation of a justification in a Reiter's default should not be entailed, in a CG default the constraint itself should not be entailed.

The entailment mechanism is based on the construction of a default derivation tree.

Let $\mathcal{K} = ((\mathcal{S}, G, \mathcal{R}), \mathcal{D})$ be a knowledge base, where G is a basic CG, $\mathcal{R}$ is a set of CG rules, and $\mathcal{D}$ is a set of CG defaults, all defined over the support $\mathcal{S}$. As previously mentioned, we will assume that G is completely instantiated. Then the rule application mechanism ensures that all derived facts are also completely instantiated.

A node of the default derivation tree $DDT(\mathcal{K})$ is labelled by a basic CG called fact and a set of basic CGs called constraints. A node of $DDT(\mathcal{K})$ with label $(G, \mathcal{C})$ is said to be *valid* if there is no homomorphism from a constraint in $\mathcal{C}$ or a constraint occurring in the label of one of its ancestors to G . We now define inductively the tree $DDT(\mathcal{K})$:

- the root is labelled by $(G, \emptyset)$
- if x is a valid node of $DDT(\mathcal{K})$ with label $(F, \mathcal{C})$

for every default $D = (H, C, C_1, \dots, C_k)$ in $\mathcal{D}$, for every homomorphism π from H to a basic CG F' $\mathcal{R}$ -derived from F ,

x has a successor whose fact is obtained by the application of D as a classical rule $R = (H, C)$ without considering it as a default, and whose constraints are the $\pi(C_i)$, iff that successor is valid.

In the above definition, $\pi(C_i)$ is obtained from C_i by replacing the labels of concept nodes that belong to the domain of π (thus are shared with H) with their image by π . This allows to bind some nodes of C_i to nodes of the current fact. Let us consider for instance the DDT in Figure 6, obtained with the set of rules in Figure 5 and the initial fact G in Figure 3. The successor of the root is obtained by applying the rule R_N , which has an empty hypothesis and no constraints: hence, the conclusion is simply added to G , after instantiation of the generic concept node $[\text{Set}:*]$ with a new individual marker (denoted by E in the figure). The left most successor of this tree node (let us note it n_3) is obtained by applying the rule D_N with the concept nodes $[\text{Set}:*]$ and $[\text{Arg}:*]$ from the rule hypothesis being respectively mapped to $[\text{Set}:E]$ and $[\text{Arg}:A_1]$. Both constraints of D_N are instantiated accordingly. It is checked that n_3 is valid: indeed, the instantiated constraints do not map to the newly built fact. Thus n_3 is actually added to the tree. Note that all descendants of n_3 will have to satisfy the constraints labelling n_3 .

The leaves of $\text{DDT}(\mathcal{K})$ exactly encode extensions of a default theory (see [Baget *et al.*, 2009; Baget and Fortin, 2010]).

3 Argumentation System Modelling

This section proposes a representation of an abstract argumentation system in the CG formalism, as previously introduced. The associated reasoning mechanisms that computes acceptable sets of arguments, will be presented in Section 4.

3.1 Support Description

To encode an argumentation system, independently from a given application domain, the elements that constitute an argumentation framework (arguments, sets of arguments), their properties (admissibility semantics) and the relations between them (attack relation, membership and inclusion relations) have to be introduced in the support of the CG model.

Figure 2 shows a support that represents these elements. The fact that stable sets of acceptable arguments are specializations of preferred sets, which are themselves specializations of admissible sets, is directly encoded in the support through the “kind of” relation.

3.2 Graph of Attacks

Based on the generic support of Figure 2, a graph of attacks is defined as follows.

Definition 6 (Graph of attacks). *A graph of attacks is a basic CG such that:*

- *concept nodes are labelled by pairs $\text{Arg} : i$, where i denotes an instance of an argument;*

- *relation nodes are labelled by R , which represents the attack relation.*

Figure 3 shows an example of a graph of attacks in which A_1 attacks A_2 , A_2 attacks A_1 and A_3 attacks A_2 .

4 Computing acceptable sets of arguments using CG defaults

In this section, we use CG defaults to compute acceptable sets of arguments.

4.1 Computing naive sets of acceptable arguments

We will show how to compute all the naive sets of arguments using two rules (a “classic” CG rule and a CG default, see Figure 5). The way we compute the naive sets is thus purely declarative. Using these rules, the default computation mechanism calculates the default extensions. Each of these extensions encodes in a graphical manner a naive set of arguments.

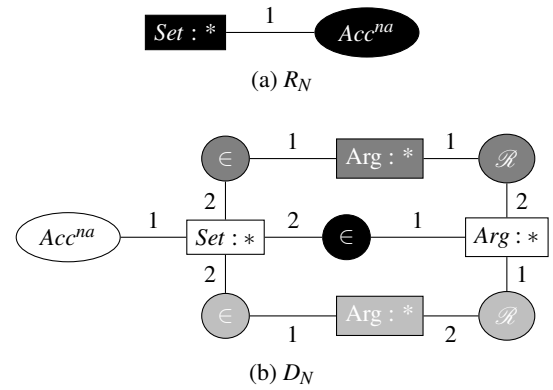


Figure 5: Rules generating naive sets of acceptable arguments

The first rule encodes the following information: “A naive set of arguments exists”. Indeed, as mentioned in section 2.1, for any graph of attacks, a naive set of arguments always exists. This rule (denoted by R_N) is a classic CG rule (see Figure 5(a)).

Given a conflict-free set of arguments E , a naive set of arguments (which has to be maximal) can be built by iteratively adding some arguments. An argument a may be added to E if $E \cup \{a\}$ is still conflict-free. The CG default D_N given in Figure 5(b) is designed in such a way that the argument a is added to the set E only if it is not in conflict with any argument of E . This is guaranteed by the two constraints of D_N :

- the first one (in dark gray) ensures that a does not attack any argument of E ;
- the second one (in light gray) ensures that a is not attacked by any argument of E .

Therefore, applying this rule to a graph of attacks preserves the property that the group of arguments linked to the set E by the relation $\in$ is conflict-free.

The deduction mechanism based on the default deviation tree ensures that when a default extension is computed, the rule D_N cannot be applied any more. Hence the extension

node labelled by E is a maximal (w.r.t. set inclusion) conflict-free set of arguments, i.e. a naive set.

Figure 6 shows the default derivation tree that is computed to obtain the default extensions encoding the naive sets. The default extensions are encoded in the leaves of the tree. Note that two of the leaves of the tree are identical, so we obtain only two different default extensions: $Acc_1^{na} = \{A_2\}$ $Acc_1^{na} = \{A_1, A_2\}$.

4.2 Computing preferred sets of acceptable arguments

One way to obtain the preferred sets from the naive ones is thus to iteratively remove the non-defended arguments. For that, we use the two CG defaults shown in Figure 7. Given a set of arguments, if there is an argument in this set that is not defended, the CG default DCP_1 creates a new set, and the non-defended argument is declared as not belonging to this set. The CG default DCP_2 adds all the arguments to the new set of arguments, unless they have been declared as not belonging to this set. By applying these two rules until obtaining of the default extensions, the resulting sets of arguments have the property of being:

- conflict-free, since they are subsets of naive sets;
- without non-defended arguments.

Among them, the maximal ones (w.r.t. set-inclusion) are the preferred sets of acceptable arguments. Note that, to select maximal sets, one would have to compare extensions, which is not possible in our framework. Thus, tagging such sets by the Acc^{pr} relation, indicating they are preferred sets, is done outside the framework.

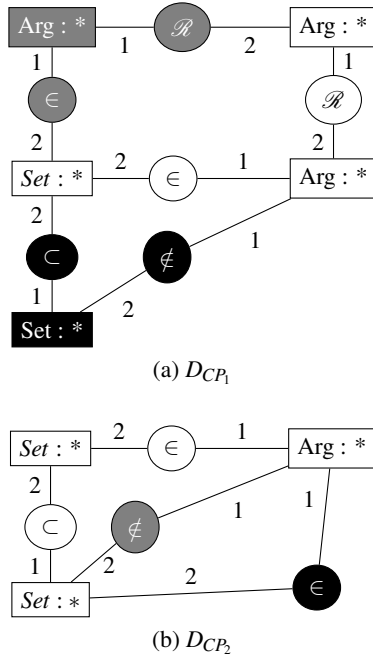


Figure 7: Rules generating preferred sets of acceptable arguments

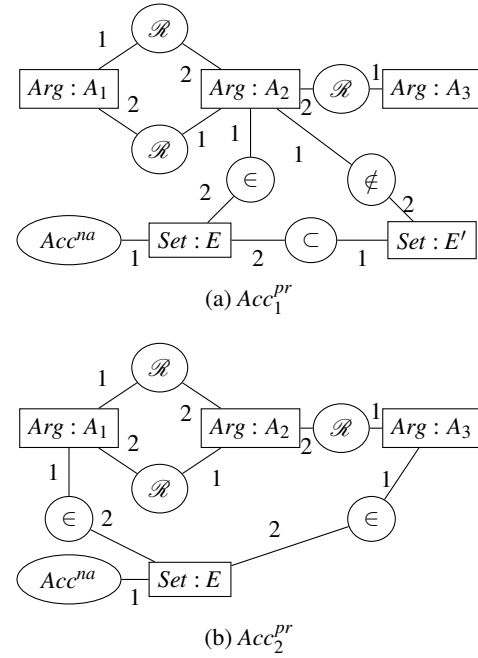


Figure 8: Extensions computed from the leaves of Figure 6, with the CG defaults of Figure 7

Figure 8 represents the preferred sets of our running example (subject to maximality test). The first set, derived from Acc_1^{na} , is the empty set (since there is no $\in$ relation linking E' to any argument). As it is included in the set $Acc_2^{pr} = \{A_1, A_3\}$, derived from Acc_2^{na} , it is not maximal and thus not tagged as a preferred set.

4.3 Computing stable sets of acceptable arguments

A stable set of arguments is *a fortiori* a preferred set. To be stable, a preferred set of arguments has to attack all the arguments that are not in the set. It turns out that once a set of preferred arguments is computed, it is easy to check whether it is also a stable set of arguments. This can be done using the CG default represented in Figure 9, which starts from a preferred set and expresses a condition to reject it if it is not a stable set; it is tagged as “not stable” (denoted by $notAcc^{st}$) unless each argument belongs to it (dark gray constraint) or is attacked by an argument that belongs to it (light gray constraint). Then, when a default extension is computed, a preferred extension is identified as a stable extension iff it is not tagged as “not stable”.

In our running example, the unique preferred set of arguments $Acc_2^{pr} = \{A_1, A_3\}$ is also a stable set, since the only argument that is not in the set, A_2 , is attacked by A_1 and A_3 .

5 Conclusion

In this paper, we have shown how an argumentation framework can be represented in the CG formalism. This formalism also allows to compute different kinds of acceptable sets of arguments. However, it does not capture the notion of maximality in the definition of preferred sets. Therefore, in order

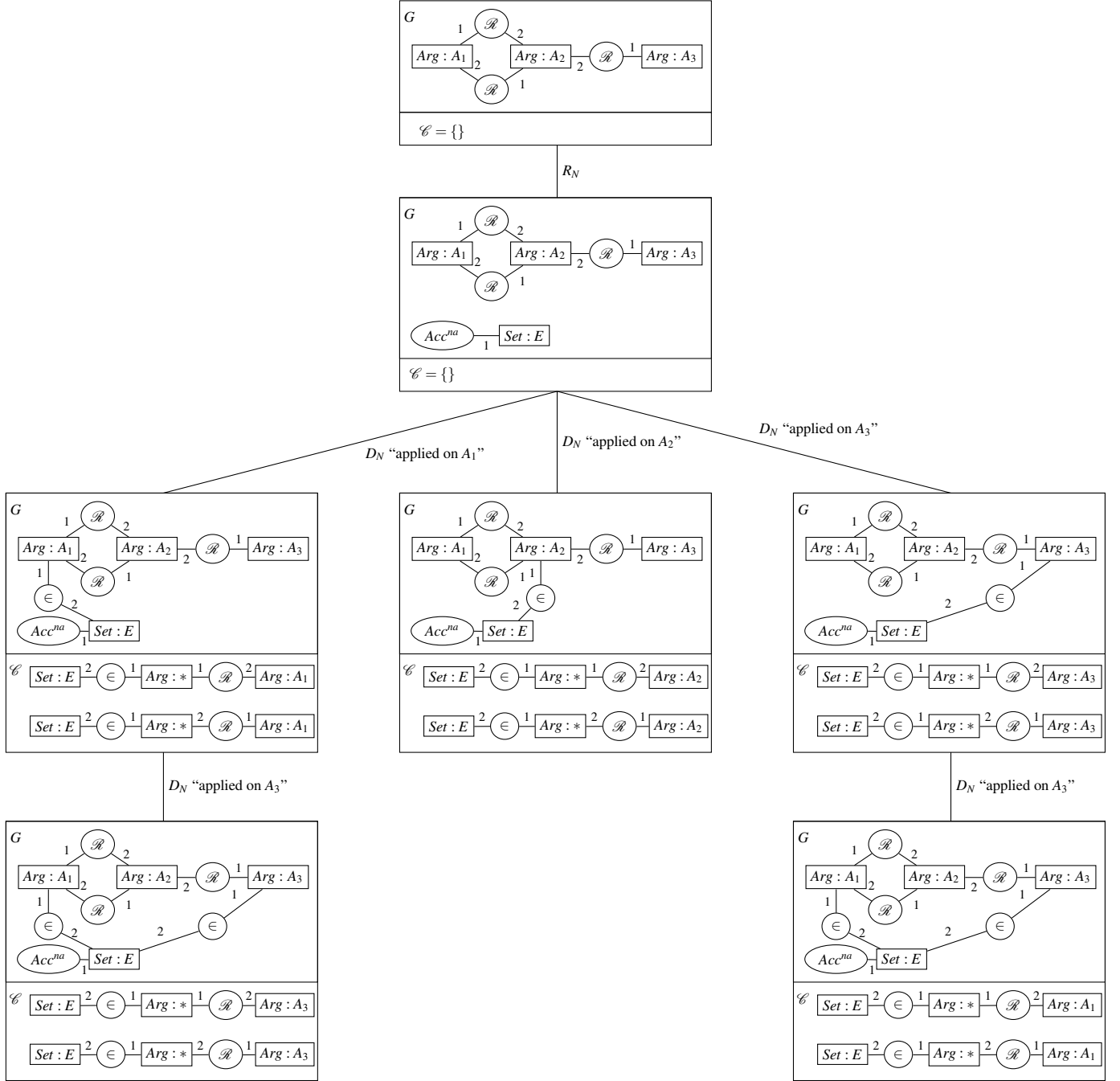


Figure 6: DDT

Representing CSPs with Set-labeled Diagrams: A Compilation Map

Alexandre Niveau

ONERA/DCSD, France

alexandre.niveau@onera.fr

Hélène Fargier

IRIT/CNRS, France

fargier@irit.fr

Cédric Pralet

ONERA/DCSD, France

cedric.pralet@onera.fr

Abstract

Constraint Satisfaction Problems (CSPs) offer a powerful framework for representing a great variety of problems. Unfortunately, most of the operations associated with CSPs are NP-hard. As some of these operations must be addressed online, compilation structures for CSPs have been proposed, *e.g.* finite-state automata and Multivalued Decision Diagrams (MDDs).

The aim of this paper is to draw a compilation map of these structures. We cast all of them as fragments of a more general framework that we call Set-labeled Diagrams (SDs), as they are rooted, directed acyclic graphs with variable-labeled nodes and set-labeled edges; contrary to MDDs and Binary Decision Diagrams, SDs are not required to be deterministic (the sets labeling the edges going out of a node are not necessarily disjoint), ordered nor even read-once.

We study the relative succinctness of different subclasses of SDs, as well as the complexity of classically considered queries and transformations. We show that a particular subset of SDs, satisfying a focusing property, has theoretical capabilities very close to those of Decomposable Negation Normal Forms (DNNFs), although they do not satisfy the decomposability property *stricto sensu*.

1 Introduction

Constraint Satisfaction Problems (CSPs) [Rossi *et al.*, 2006] offer a powerful framework for representing a great variety of problems, *e.g.* planning or configuration problems. Different kinds of operations can be posted on a CSP, such as extraction of a solution (the most classical query), strong consistency of the domains, addition or retraction of new constraints (dynamic CSP), counting of the number of solutions, and even combinations of these operations. For instance, the interactive solving of a configuration problem amounts to a series of (unary) constraints additions and retractions while maintaining the strong consistency of the domains, *i.e.* each value in a domain is involved in at least one solution.

Most of these operations are NP-hard, but must sometimes be addressed online. A possible way of solving this contra-

diction is to use knowledge compilation, which consists in transforming the problem offline in such a way that its online resolution becomes tractable. As a matter of fact, Multivalued Decision Diagrams (MDDs) [Srinivasan *et al.*, 1990; Vempaty, 1992; Kam *et al.*, 1998; Andersen *et al.*, 2007] have been proposed as a way to “compile” CSPs, and successfully used in product configuration [Amilhastre *et al.*, 2002]. Figure 1 shows how an MDD can represent the set of solutions of a CSP.

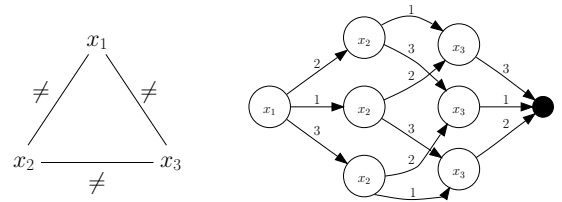


Figure 1: This figure shows the constraint graph (on the left) of the 3-coloring problem (the domain of the variables is $\{1, 2, 3\}$), and an MDD (on the right) representing the set of solutions of this CSP.

In the present paper, we investigate this landscape by capturing these existing compilation structures as subsets of a more general framework called “set-labeled diagrams”. The latter also covers new structures relaxing the requirements of determinism and ordering, which, as we show, can lead to exponentially more compact graphs without losing much in efficiency. In particular, we identify a subset of set-labeled diagrams that has theoretical capabilities very close to those of DNNFs (Decomposable Negation Normal Forms) [Darwiche and Marquis, 2002], although it does not satisfy decomposability *stricto sensu*. Moreover, while most of the operations considered in classical knowledge compilation maps deal with reasoning problems, we introduce in the present map a few new operations, that are motivated by the use of the CSP framework for some more decision-oriented applications, such as planning and configuration.

Some proofs are gathered in Appendix A, but many are omitted for the sake of brevity. An extended version including detailed proofs can be found at

<http://www.irit.fr/PERSONNEL/ADRIA/Fargier/GKR11.pdf>.

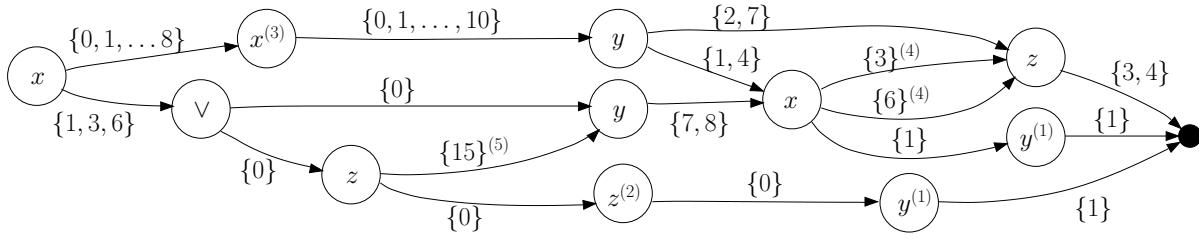


Figure 2: An example of non-reduced SD. Variable domains are all $\{0, 1, \dots, 10\}$. The two nodes marked ⁽¹⁾ are isomorphic; node ⁽²⁾ is stammering; node ⁽³⁾ is undecisive; the edges marked ⁽⁴⁾ are contiguous; edge ⁽⁵⁾ is dead.

2 Set-labeled Diagrams

Let us first formally define set-labeled diagrams, their interpretation, and their place among other knowledge compilation languages.

2.1 Structure and Semantics

The definition of set-labeled diagrams is similar to the one of classical decision diagram structures:

Definition 2.1 (Set-labeled diagram). A set-labeled diagram (SD) is a couple $\phi = \langle X, \Gamma \rangle$, with

- X (also denoted $\text{Var}(\phi)$) a finite and totally ordered set of variables whose domains are finite sets of integers;
- Γ a directed acyclic graph¹ with at most one root and at most one leaf (the sink). Non-leaf nodes are labeled by a variable of X or by the disjunctive symbol $\vee$. Each edge is labeled by a finite subset of $\mathbb{N}$.

This definition allows the graph to be empty (no node at all, only case when there is not exactly one root and one leaf) or to contain only one node (together root and sink). It does not prevent edges' labels to be empty, and ensures that every edge belongs to at least one path from the root to the sink. Figure 2 gives an example of SD.

We will now introduce useful notations: For $x \in X$, $\text{Dom}(x) \subseteq \mathbb{N}$ denotes the *domain* of x . By convention, $\text{Dom}(\vee) = \{0\}$. For $Y = \{y_1, \dots, y_k\} \subseteq X$, such that the y_i are sorted in ascending order, $\text{Dom}(Y)$ denotes $\text{Dom}(y_1) \times \dots \times \text{Dom}(y_k)$, and $\vec{y}$ denotes a Y -assignment of variables from Y , i.e. $\vec{y} \in \text{Dom}(Y)$. When $Y \cap Z = \emptyset$, $\vec{y}, \vec{z}$ is the concatenation of $\vec{y}$ and $\vec{z}$. Last, $\vec{y}(y_i)$ denotes the value assigned to y_i in $\vec{y}$ (by convention, $\vec{y}(\vee) = 0$). Let $\phi = \langle X, \Gamma \rangle$ be a set-labeled diagram, N a node and E an edge in Γ ; let us use the following notations:

- $\text{Root}(\phi)$ the root of Γ and $\text{Sink}(\phi)$ its sink;
- $|\phi|$ the *size* of ϕ , i.e. the sum of the cardinalities of all labels in ϕ plus the cardinalities of the variables' domains;
- $\text{Out}_\phi(N)$ (resp. $\text{In}_\phi(N)$) the set of outgoing (resp. incoming) edges of N ;
- $\text{Var}_\phi(N)$ the variable labeling N (not defined for $\text{Sink}(\phi)$);

¹Actually, depending on the definition Γ may not strictly be a graph, but rather a *multigraph*, since we allow two edges to go in parallel from one node to another (see e.g. Figure 2): the set of edges is a subset of $N \times N \times 2^{\mathbb{N}}$, N being the set of nodes.

- $\text{Src}_\phi(E)$ the node from which E comes and $\text{Dest}(E)$ the node to which E points;
- $\text{Lbl}_\phi(E)$ the set labeling E ;
- $\text{Var}_\phi(E) = \text{Var}_\phi(\text{Src}(E))$ the variable associated to E .

We shall drop the ϕ subscript whenever there is no ambiguity.

An SD can be seen as a compact representation of a Boolean function over discrete variables. This function is the *interpretation function* of the set-labeled diagram:

Definition 2.2 (Semantics of a set-labeled diagram). A set-labeled diagram ϕ on $X = \text{Var}(\phi)$ represents a function $I(\phi)$ from $\text{Dom}(X)$ to $\{\top, \perp\}$, called the *interpretation function* of ϕ , and defined as follows: for a given X -assignment $\vec{x}$, $I(\phi)(\vec{x}) = \top$ if and only if there exists a path p from the root to the sink of ϕ such that for each edge E along p , $\vec{x}(\text{Var}(E)) \in \text{Lbl}(E)$.

We say that $\vec{x}$ is a model of ϕ whenever $I(\phi)(\vec{x}) = \top$. $\text{Mod}(\phi)$ denotes the set of models of ϕ .

ϕ is said to be equivalent to another SD ψ (denoted $\phi \equiv \psi$) iff $\text{Mod}(\phi) = \text{Mod}(\psi)$.

Note that the interpretation function of the empty SD always returns $\perp$, since it contains no path from the root to the sink. Conversely, the interpretation function of the one-node SD always returns $\top$, since in the one-node SD, the only path from the root to the sink contains no edge.

From these two definitions it follows that SDs are strongly related to ordered binary decision diagrams [Bryant, 1986] and multivalued decision diagrams [Srinivasan *et al.*, 1990; Vempaty, 1992; Amilhastre *et al.*, 1999; Andersen *et al.*, 2007] as a way to represent a set of assignments of discrete variables (or typically, the set of solutions of a CSP). They actually generalize these data structures twofold. First, there is no restriction on the order in which the variables are encountered along a path, and variables can be repeated along a path. Second, SDs are not necessarily deterministic: the sets labeling edges going out of a node are not due to be pairwise disjoint, and thus a single model can be captured by several paths. SDs even support pure non-deterministic “OR” nodes (the $\vee$ -nodes) that allow the unrestricted union of several subgraphs. Putting away these two restrictions is valuable both theoretically, to generalize a large class of data structures, and practically, since SDs can be more compact than their ordered and deterministic variants (see Section 3.1). Let us define determinism formally and then introduce useful concepts.

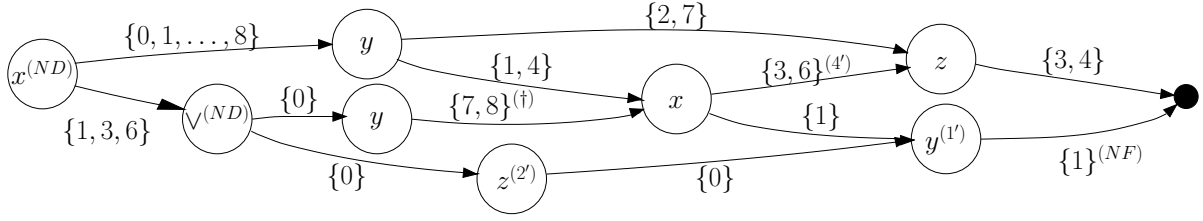


Figure 3: In this SD, all edges are focusing but the one marked $^{(NF)}$ (it is not included in the one marked $^{(\dagger)}$), and all nodes are deterministic but the ones marked $^{(ND)}$. This SD is the reduced form of the SD presented in Figure 2: isomorphic nodes marked $^{(1)}$ have been merged into node $^{(1')}$, stammering node $^{(2)}$ has been collapsed into node $^{(2')}$, contiguous edges marked $^{(4)}$ have been merged into edge $^{(4')}$, and undecisive node $^{(3)}$ and dead edge $^{(5)}$ have been removed.

Definition 2.3 (Deterministic set-labeled diagrams). A node N in a set-labeled diagram ϕ is deterministic if the sets labeling its outgoing edges are pairwise disjoint.

A deterministic set-labeled diagram (dSD) is an SD containing only deterministic nodes.

The notion of determinism is illustrated on Figure 3.

Definition 2.4 (Consistency, validity, context). Let ϕ be a set-labeled diagram on X .

ϕ is said to be consistent (resp. valid) if and only if $\text{Mod}(\phi) \neq \emptyset$ (resp. $\text{Mod}(\phi) = \text{Dom}(X)$).

A value $v \in \mathbb{N}$ is said to be consistent for a variable $y \in X$ in ϕ if and only if there exists an X -assignment $\vec{x}$ in $\text{Mod}(\phi)$ such that $\vec{x}(y) = v$.

The set of all consistent values for y in ϕ is called the context of y in ϕ and denoted $\text{Ctxt}_\phi(y)$.

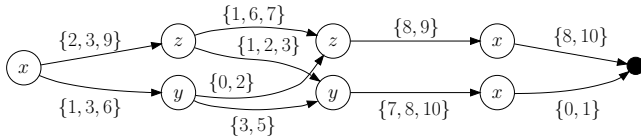


Figure 4: Before an SD is proven inconsistent, every path must be checked. Here is an example of SD whose every path is inconsistent.

We will see in the following that deciding whether an SD is consistent is not tractable. One of the reasons is that the sets restricting a variable along a path can be conflicting, hence in the worst case all paths must be explored before a consistent one is found. Figure 4 shows an example of SD with no consistent path at all. To avoid this, we will consider SDs in which labeling sets can only shrink from the root to the sink, thus preventing conflicts:

Definition 2.5 (Focusing set-labeled diagrams). A focusing edge in a set-labeled diagram ϕ is an edge E such that all edges E' on a path from the root of ϕ to $\text{Src}(E)$ such that $\text{Var}(E) = \text{Var}(E')$ verify $\text{Lbl}(E) \subseteq \text{Lbl}(E')$.

A focusing set-labeled diagram (FSD) is an SD containing only focusing edges.

The notion of focusing edge is illustrated in Figure 3. It is sufficient for the consistency query to be polynomial, but for some other operations (such as obtaining the conjunction

of two SDs), a stricter restriction is necessary. An interesting one, very common in knowledge compilation, is to impose an order on the variables encountered along the paths; applying to SDs, we recover Multivalued Decision Diagrams (MDDs) in their practical acception² [Srinivasan *et al.*, 1990; Vempaty, 1992; Andersen *et al.*, 2007].

Definition 2.6 (Ordered diagrams). Let $<$ be a total order on the variables of X . A set-labeled diagram is said to be ordered w.r.t. $<$ iff, for any two nodes N and M , if N is an ancestor of M then $\text{Var}(N) < \text{Var}(M)$.

A dSD ordered w.r.t. $<$ is called an $\text{MDD}_<$. The language MDD is the union of all $\text{MDD}_<$ languages.³

Obviously, if there are not two occurrences of the same variable in a path, all edges are focusing. Hence: $\text{MDD}_< \subseteq \text{MDD} \subseteq \text{dFSD} \subseteq \text{FSD} \subseteq \text{SD}$ and $\text{dFSD} \subseteq \text{dSD} \subseteq \text{SD}$. We will study in the next sections the main properties of the SD family, and their relationships with classical Boolean decision diagrams. But before that, let us show how to reduce an SD in order to make it as compact as possible — and save space.

2.2 Reduction

Like a BDD, an SD can be reduced in size without changing its semantics. Reduction is based on several operations; some of them are straightforward generalizations of those introduced in the context of BDDs [Bryant, 1986], namely merging isomorphic nodes (that are equivalent) and collapsing undecisive edges (that are always crossed), while others are specific to set-labeled diagrams, namely suppressing dead edges (that are never crossed), merging contiguous edges (that have the same source and the same destination) and collapsing stammering nodes (successive decisions that pertain to the same variable). All these notions are illustrated in the SD of Figure 2, and the reduced form of this SD is shown on Figure 3. Formally:

Definition 2.7. • Two edges E_1, E_2 are contiguous iff $\text{Src}(E_1) = \text{Src}(E_2)$ and $\text{Dest}(E_1) = \text{Dest}(E_2)$.

²The original definition of MDDs does not require determinism, nor introduces an order on the variables. Nevertheless, the papers resorting to these structures work only with ordered and deterministic MDDs; that is why we abusively designate ordered dSDs as MDDs.

³A language is a set of graph structures, fitted up with an interpretation function. We denote SD the language of SDs, dSD the language of dSDs, and so on.

- Two nodes N_1, N_2 are isomorphic iff $\text{Var}(N_1) = \text{Var}(N_2)$ and there exists a bijection σ from $\text{Out}(N_1)$ onto $\text{Out}(N_2)$, such that $\forall E \in \text{Out}(N_1), \text{Lbl}(E) = \text{Lbl}(\sigma(E))$ and $\text{Dest}(E) = \text{Dest}(\sigma(E))$.
- An edge E is dead iff $\text{Lbl}(E) \cap \text{Dom}(\text{Var}(E)) = \emptyset$.
- A node N is undecisive iff $|\text{Out}(N)| = 1$ and $E \in \text{Out}(N)$ is such that $\text{Dom}(\text{Var}(E)) \subseteq \text{Lbl}(E)$.
- A non-root node N is stammering iff all parent nodes of N are labeled by $\text{Var}(N)$, and either $\sum_{E \in \text{Out}(N)} |E| = 1$ or $\sum_{E \in \text{In}(N)} |E| = 1$.

Definition 2.8 (Reduced form). A set-labeled diagram ϕ is said to be reduced iff no node of ϕ is isomorphic to another, stammering, or undecisive; and no edge of ϕ is dead or contiguous to another.

In the following, we can consider only reduced SDs since reduction can be done in time polynomial in their size; indeed, each reduction step (removal of isomorphic nodes, contiguous edges, etc.) is polytime and removes more nodes and edges than it adds, hence even if we have to traverse the graph several times, the global complexity remains polynomial.

Proposition 2.9 (Reduction). Let L be a sublanguage of SD among $\{\text{SD}, \text{FSD}, \text{dSD}, \text{dFSD}, \text{MDD}, \text{MDD}_<\}$. There exists a polytime algorithm that transforms any ϕ in L into an equivalent reduced ϕ' in L such that $|\phi'| \leq |\phi|$.

We have seen that SDs are strongly related to BDDs and MDDs; we will now detail these relations.

2.3 SDs and the decision diagram family

Binary Decision Diagrams (BDDs, [Bryant, 1986]) are rooted, directed acyclic graphs that represent Boolean functions of Boolean variables. They have two leaves, respectively labeled $\perp$ and $\top$; their non-leaf nodes are labeled by a Boolean variable and have two outgoing edges, respectively labeled $\perp$ and $\top$. A free BDD (FBDD) is a BDD that satisfies the read-once property (each path contains at most one occurrence of each variable). Whenever a same order is imposed on the variables along every path, we get an ordered BDD (OBDD). OBDDs have been extended to enumerated domains as MDDs by [Srinivasan *et al.*, 1990; Vempaty, 1992; Amilhastre *et al.*, 1999] and later on, worked out by [Andersen *et al.*, 2007].

SDs are obviously not decision diagrams in the sense of Bryant since they do not have a $\perp$ sink, but classical MDDs do not either. Adding or not such a sink is actually harmless, and does not represent a real difference. The first main difference between decision diagrams and SDs is that SDs can be non-deterministic. Relationships between SDs and their Boolean counterparts are formally provided thereafter.

Definition 2.10 (Polynomial translatability). A sublanguage L_2 of SD is polynomially translatable into another sublanguage L_1 of SD, which we denote $L_1 \leq_P L_2$, if and only if there exists a polytime algorithm mapping any element from L_2 to an equivalent element from L_1 .

For any subclass L of SD, any $D \subseteq \mathbb{N}$, let L_D be the sublanguage of L for which all domains are included in D . We

will consider in particular classes $\text{dSD}_{\{0,1\}}$, $\text{dFSD}_{\{0,1\}}$, and $\text{FSD}_{\{0,1\}}$, that generalize BDD, FBDD, and DNF respectively.

When an order is imposed on the variables, $\text{MDD}_{\{0,1\}}$ and OBDD are equivalent representation languages, up to the existence of a $\perp$ sink, which, once again, is harmless. More generally, [Srinivasan *et al.*, 1990] have shown that a log encoding of the domains allow to transform any MDD into an equivalent OBDD, providing by the way a convenient way to implement a MDD package on top of a BDD package.

Let us put the emphasis on the new languages, namely focusing SDs:

Proposition 2.11 ($\text{FSD}_{\{0,1\}} \leq_P \text{DNF}$). Any formula in the DNF language can be expressed in the form of a $\text{FSD}_{\{0,1\}}$ in linear time.

Proposition 2.12 ($\text{dFSD}_{\{0,1\}} \leq_P \text{FBDD}$). Any FBDD (and thus any OBDD) can be expressed in the form of an equivalent $\text{dFSD}_{\{0,1\}}$ in time linear in the FBDD's size.

dFSD actually generalizes FBDD, and we will see that it reaches the same performances as this fragment, except for the counting query. But it is worth noticing that, contrary to FBDD, dFSD allows a variable to be met twice on a path. $\text{dFSD}_{\{0,1\}}$ is thus a proper superset of FBDD.

FSD are more general than usual MDD compilations of CSPs, since they do not require any order nor even determinism; we will see in the following that this can lead to exponential savings in space.

Last, but not least, it should be noticed that dFSDs and FSDs are not decomposable structures in the sense of Negation Normal Forms (NNFs). Indeed, the definition of decomposability [Darwiche, 2001], when applied to a decision diagram, implies that variables cannot be repeated along a path. Since they are not decomposable, dFSDs do not define a subclass of AND/OR MDDs [Mateescu and Dechter, 2006], of structured DNNFs [Pipatsrisawat and Darwiche, 2008], nor of tree automata [Fargier and Vilarem, 2004], that are decomposable (and ordered) structures.

3 A Compilation Map of SDs

In the following, we put forward the properties of the SD language and its sublanguages, according to their spatial efficiency (succinctness), and to their capacity to make queries and transformations tractable.

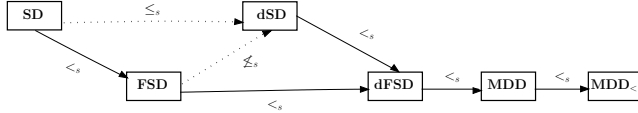
3.1 Succinctness

Definition 3.1 (Succinctness). A sublanguage L_1 of SD is at least as succinct as another sublanguage L_2 of SD, which is denoted $L_1 \leq_s L_2$, if and only if there exists a polynomial $P(\cdot)$ s.t. for each element ϕ of L_2 , there exists an equivalent ψ of L_1 verifying $|\psi| \leq P(|\phi|)$.

Of course, $\leq_s$ is a preorder: let us denote $\sim_s$ its symmetric part and $<_s$ its asymmetric part.

Proposition 3.2 (Succinctness). The results of Table 1 hold.

This shows that $\text{SD} <_s^* \text{FSD}$, $\text{FSD} <_s^* \text{dFSD}$, and $\text{dFSD} <_s^* \text{MDD}$, which means that imposing focusingness, determinism, or ordering may lead to an exponential increase in space.



L	SD	dSD	FSD	dFSD	MDD	MDD<
SD	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$
dSD	?	$\leq_s$	?	$\leq_s$	$\leq_s$	$\leq_s$
FSD	$\not\leq_s^*$	$\not\leq_s^*$	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$
dFSD	$\not\leq_s^*$	$\not\leq_s^*$	$\not\leq_s^*$	$\leq_s$	$\leq_s$	$\leq_s$
MDD	$\not\leq_s^*$	$\not\leq_s^*$	$\not\leq_s^*$	$\not\leq_s^*$	$\leq_s$	$\leq_s$
MDD<	$\not\leq_s^*$	$\not\leq_s^*$	$\not\leq_s^*$	$\not\leq_s^*$	$\not\leq_s^*$	$\leq_s$

Table 1: Results about succinctness. * means that the result is true unless the polynomial hierarchy PH collapses. The graph above illustrates those results, dotted lines meaning that we lack of information to prove both directions (*i.e.* it is proven that one of the languages is at least as succinct as the other, but it isn't known whether the converse is true or false).

3.2 Operations on Set-labeled Diagrams

We now need to extend to enumerated domains the queries and transformations considered in classical compilation maps. The importance of most of them is discussed in depth in [Darwiche and Marquis, 2002], so we refrain from recalling it here. We add to these operations two new ones: The CX, “Context Extraction” query aims at providing the user with all possible values of some variable of interest. The $\wedge tC$ “conjunction with a term” transformation restricts the possible values of some variables to a subset of their domains. These two operations are widely used in configuration, where the user iteratively looks for the possible values of the next configuration variables and restricts its values according to her preferences [Amilhastre *et al.*, 2002]. For space reasons, we do not include here the most straightforward operation extensions; they can however be found in the long version of this paper.

Definition 3.3 (Queries). *Let L denote a subset of the SD language.*

- L satisfies *clausal entailment* (CE) (resp. *implication* (IM)) iff there exists a polynomial $P(\cdot)$ and an algorithm that maps every SD ϕ from L , any set of variables $\{y_1, \dots, y_k\} \subseteq \text{Var}(\phi)$, and any sequence $(A_1, \dots, A_k)$ of sets of integers, to 1 if $I(\phi) \models f_{y_1, A_1} \vee \dots \vee f_{y_k, A_k}$ (resp. $f_{y_1, A_1} \wedge \dots \wedge f_{y_k, A_k} \models I(\phi)$) and to 0 otherwise, in time $P(|\phi|; n_A)$, where $n_A = \max_{1 \leq i \leq k} |A_i|$ and $f_{x, A}$ is the function defined on $Y = \{x\}$ by $f_{x, A}(\vec{y}) = \top \Leftrightarrow \vec{y}(x) \in A$.
- L satisfies *model extraction* (MX) iff there exists a polytime algorithm that maps every SD ϕ in L to one model of ϕ if there is one, and stops without returning anything otherwise.
- L satisfies *context extraction* (CX) iff there exists a polytime algorithm that outputs, for any ϕ in L and any $y \in \text{Var}(\phi)$, $\text{Ctx}_\phi(y)$.

Definition 3.4. *Let I, J be the interpretation functions on $\text{Var}(I), \text{Var}(J)$ of some SDs.*

- The universal projection of I on $Y \subseteq \text{Var}(I)$ is the function $I^{\downarrow Y}$ on the variables of Y defined by: $I^{\downarrow Y}(\vec{y}) = \top$ iff for any Z -assignment $\vec{z}$ (with $Z = \text{Var}(I) \setminus Y$), $I(\vec{z}, \vec{y}) = \top$. The “ensuring” operation is the dual one: $\text{ensure}(I, Y) = I^{\downarrow \text{Var}(I) \setminus Y}$.

Definition 3.5 (Transformations). *Let L denote a subset of the SD language.*

- L satisfies *ensuring* (EN) iff there exists a polytime algorithm that maps every SD ϕ from L and every $Y \subseteq \text{Var}(\phi)$ to an SD ϕ' in L such that $I(\phi') = \text{ensure}(I(\phi), Y)$. It satisfies *simple ensuring* (SEN) iff it satisfies EN when limited to a single variable (*i.e.* $\text{Card}(Y) = 1$).
- L satisfies *term conjunction* ($\wedge tC$) iff there exists a polynomial $P(\cdot)$ and an algorithm that outputs, for every SD ϕ from L , any set of variables $\{y_1, \dots, y_k\} \subseteq \text{Var}(\phi)$ and any sequence $(A_1, \dots, A_k)$ of sets of integers, an SD ϕ' in L such that $I(\phi') = I(\phi) \wedge f_{y_1, A_1} \wedge \dots \wedge f_{y_k, A_k}$ in time $P(|\phi|; n_A)$, where $n_A = \max_{1 \leq i \leq k} |A_i|$.

Proposition 3.6. *The results of Table 2 hold.*

The results pertaining to MDDs are generally known or follows directly from [Srinivasan *et al.*, 1990; Kam *et al.*, 1998]: in short, since MDDs (i) are a generalization of OBDDs to non-Boolean domains and (ii) can be encoded as OBDDs, they obviously have the same capabilities than OBDDs, except for SEN and SFO, for which the domain size has a more important role.

Table 2 puts forward the attractivity of the new dFSD and FSD classes: their performances are comparable to those of the d-DNNF (resp. DNNF) fragments, although, once again, dFSD and FSD do not satisfy decomposability but a distinct property also guaranteeing CO: focusingness.

From a more practical point of view, when one needs to use queries on compiled forms, but no transformation, like in configuration applications, dFSDs are much more interesting than MDDs — they can be much more compact, while satisfying almost the same queries. It is worthwhile imposing a variable order (*i.e.* going from dFSDs to MDDs) when one of the $\neg C$, SEN, SFO transformation is required.

Relaxing the requirement of determinism, we get the FSD fragment which moreover satisfies FO and $\vee C$; FSD fits particularly applications such as planning, where one needs to often check consistency, forget variables and extract models.

3.3 Capturing Continuous Variables

For the sake of simplicity, we chose to represent labeling sets as enumerations (the size of a set is defined as its cardinality). However, they could be represented in a more compact way, as a union of integer intervals. In this case, the effective size of a label would be the number of bounds necessary to represent this union, and this could lead to space savings linear in the size of the domains.

This idea of grouping elements raises another thought: there can be, in an SD, subsets whose elements are never separated, *i.e.* values that are interchangeable. One could then define meta-values, each one corresponding to one of these

L	CO	VA	MC	CE	IM	EQ	SE	MX	CX	CT	ME
SD	○	○	✓	○	○	○	○	○	○	○	○
dSD	○	○	✓	○	○	○	○	○	○	○	○
FSD	✓	○	✓	✓	○	○	○	✓	✓	○	✓
dFSD	✓	✓	✓	✓	✓	?	○	✓	✓	?	✓
MDD	✓	✓	✓	✓	✓	✓	○	✓	✓	✓	✓
MDD _{<}	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DNNF	✓	○	✓	✓	○	○	○	✓	✓	○	✓
d-DNNF	✓	✓	✓	✓	✓	?	○	✓	✓	✓	✓

L	CD	SCD	△C	FO	SFO	EN	SEN	△C	△BC	△C	△BC	△C
SD	✓	✓	✓	○	✓	○	✓	✓	✓	✓	✓	?
dSD	✓	✓	✓	○	✓	○	✓	✓	✓	✓	✓	✓
FSD	✓	✓	✓	✓	✓	○	○	○	○	✓	✓	○
dFSD	✓	✓	✓	○	○	○	○	○	○	○	○	?
MDD	✓	✓	✓	●	●	●	●	●	○	●	○	✓
MDD _{<}	✓	✓	✓	●	●	●	●	●	✓	●	✓	✓
DNNF	✓	✓	✓	✓	✓	○	○	○	○	✓	✓	○
d-DNNF	✓	✓	✓	○	○	○	○	○	○	○	○	?

Table 2: Results about queries and transformations; ✓ means “satisfies”, ● means “does not satisfy”, and ○ means “does not satisfy, unless $P = NP$ ”. Results for DNNF, d-DNNF are from Darwiche and Marquis’ map are given here as a baseline; in a similar way results for MDD, MDD_< are obtained or follow from Srinivasan et Al’s work.

subsets. This would preserve the structural properties (determinism, focusingness...) of the considered SD.

It is also worth noticing that these principles provide us with a way to capture continuous variables, and to transform “continuous SDs” into discrete SDs. Classically, a continuous variable is “discretized” by partitioning its domain arbitrarily and replacing it by an integer variable, each value of which is associated with a unique element of the partition. Here, given a continuous SD, obtained for example by compiling the trace of a continuous CSP solver [Niveau *et al.*, 2010], it is easy to compute a discrete labeling that fits exactly the configuration of the graph. The idea is to recover, for each variable x , the set B of bounds of all intervals pertaining to x in the continuous SD. The partition chosen for $\text{Dom}(x)$ is then the set containing all singleton intervals $[a, a]$ where $a \in B$, and all open intervals $]a, b[$ where a and b are successive bounds in B . Thus, all continuous intervals associated with x in the continuous SD and all intersections of such intervals can be represented as the exact union of elements in the partition.

Doing so, we can build an SD representing a CSP with real variables without resorting to an arbitrary (and spatially costly) discretization of the domains. The resulting SD can be embarked along with a translation table, allowing to make queries and transformations on the discrete structure and translating inputs (resp. results) from (resp. to) real numbers.

4 Conclusion

This paper draws the compilation map of set-labeled diagrams, a new knowledge compilation language able to represent solution sets of CSPs. We identified a structural property, focusingness, which is not equivalent to decomposability, yet when imposed on SDs allows the same queries and transformations. On the practical side, having considered CSPs rather than Boolean logic as a source language for compilation also raised the need for specific operations, that weren’t covered by classical compilation maps (context extraction, conjunction with a term); it is noticeable that focusingness is sufficient to allow them in polytime. Finally, we sketched a way to use SDs to compile continuous CSPs.

This work introduces the first brick of a complexity map of CSP compilation; it is dedicated to “linear” decision diagrams. The next step is obviously to extend it to parallel

focusing structures, the introduction of “AND” nodes. This will explicitly raise the question of a property capturing both the notions of focusingness and decomposability.

Future work also involves studying of how to build SDs, notably the experimentation of a compiler based on the trace of a search-like algorithm, and the comparison of various *dynamic* search heuristics w.r.t. SDs’ size.

References

- [Amilhastre *et al.*, 1999] J. Amilhastre, P., and M.-C. Vilarem. FA Minimisation Heuristics for a Class of Finite Languages. In *WIA*, pages 1–12, 1999.
- [Amilhastre *et al.*, 2002] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSPs — application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.
- [Andersen *et al.*, 2007] H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A constraint store based on multivalued decision diagrams. In *CP*, pages 118–132, 2007.
- [Bryant, 1986] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [Darwiche and Marquis, 2002] A. Darwiche and P. Marquis. A Knowledge Compilation Map. *JAIR*, 17:229–264, 2002.
- [Darwiche, 2001] Adnan Darwiche. Decomposable Negation Normal Form. *Journal of the ACM*, 48(4):608–647, 2001.
- [Fargier and Vilarem, 2004] H. Fargier and M.-C. Vilarem. Compiling CSPs into tree-driven automata for interactive solving. *Constraints*, 9:263–287, 2004.
- [Kam *et al.*, 1998] T. Kam, T. Villa, R.K. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued Decision Diagrams: Theory and Applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.
- [Kautz and Selman, 1992] H.A. Kautz and B. Selman. Forming concepts for fast inference. In *Proc. of AAAI’92*, pages 786–793, San Jose (CA), 1992.
- [Mateescu and Dechter, 2006] R. Mateescu and R. Dechter. Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams. In *CP*, pages 329–343, 2006.
- [Niveau *et al.*, 2010] A. Niveau, H. Fargier, C. Pralet, and G. Verfaillie. Knowledge compilation using interval automata and applications to planning. In *ECAI*, pages 459–464, 2010.

[Pipatsrisawat and Darwiche, 2008] K. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. In *AAAI'08*, 2008. 517-522.

[Rossi et al., 2006] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[Srinivasan et al., 1990] A. Srinivasan, T. Ham, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *ICCAD-90*, pages 92–95, November 1990.

[Vempaty, 1992] N. R. Vempaty. Solving Constraint Satisfaction Problems Using Finite State Automata. In *AAAI*, pages 453–458, 1992.

A Some Important Proofs

In all of these proofs, we consider for short that for an interpretation function I with $Y = \text{Var}(I)$, a set of variables Z s.t. $Z \cap Y = \emptyset$, $\vec{y}$ a Y -assignment and $\vec{z}$ a Z -assignment, $I(\vec{y}, \vec{z})$ is simply defined as $I(\vec{y})$.

Let ϕ be an SD on X and $\vec{x}$ an X -assignment. A path p from the root to the sink of ϕ is said to be *compatible with $\vec{x}$* if and only if for each edge E along p , $\vec{x}(\text{Var}(E)) \in \text{Lbl}(E)$.

For any I , we denote $\vec{x} \models I$ the fact that $\vec{x}$ is a model of I , and $\vec{x} \models \phi$ iff $\vec{x} \models I(\phi)$. We also denote $\vec{x} \models p$ if p is a path compatible with $\vec{x}$.

A.1 Succinctness

Lemma A.1. *Let γ be a clause; a $\text{dFSD}_{\{0,1\}}$ equivalent to $\Sigma \vee l$ can be constructed in polytime in the size of Σ .*

Proof. Let Σ be a $\text{dFSD}_{\{0,1\}}$; let $l_1, \dots, l_n$ the literals that appear in γ .

$\Sigma \vee l_1 \equiv (\Sigma_{|\neg l_1} \wedge \neg l_1) \vee l_1$. Since dFSD satisfies **CD**, a $\text{dFSD}_{\{0,1\}}$ β representing $\Sigma_{|\neg l_1}$ can be built in polytime. A $\text{dFSD}_{\{0,1\}}$ equivalent to $(\beta \wedge \neg l_1) \vee l_1$ can be built, whose root is labeled by $\text{var}(l_1)$ and has two outgoing edges: the first one is labeled by $\{1\}$ if l_1 is a positive literal, by $\{0\}$ otherwise; the second one is labeled by $\{0\}$ if l_1 is a positive literal, by $\{1\}$ otherwise, and links the root to β . The procedure is polynomial and the result is linear in the size of Σ (the size $\Sigma_{|\neg l_1}$ is lower than the one of Σ ; then one node and two edges are added).

Iterating the operation for $i = \{1, \dots, n\}$, we get a $\text{dFSD}_{\{0,1\}}$ equivalent to $\Sigma \vee \gamma$; its number of nodes (and thus, of edges) is bounded by $n + |\text{nodes}(\Sigma)|$. $\square$

Many proofs of succinctness rely on the following lemma, due to [Kautz and Selman, 1992]:

Lemma A.2. *It is impossible to find a polysize compilation function comp such that for any CNF Σ and any clause γ , checking whether $\Sigma \models \gamma$ using $\text{comp}(\Sigma)$ can be done in polytime, unless the polynomial hierarchy PH collapses at the second level.*

In particular, it is impossible, unless the polynomial hierarchy PH collapses at the second level, to find a target language at least as succinct as CNF that supports **CE**.

Proof of Proposition 3.2 (succinctness table).

$\text{FSD} \not\leq_s \text{SD}$. Any clause can be expressed as in polytime as a $\text{dSD}_{\{0,1\}}$ and thus as an element of $\text{SD}_{\{0,1\}}$. Making the conjunction of two SDs is linear (juste let the source of the first one be the sink of the second one, see also Prop. 3.6). Hence $\text{SD}_{\{0,1\}} \leq_{\mathcal{P}} \text{CNF}$.

Suppose we had $\text{FSD}_{\{0,1\}} \leq_s \text{SD}_{\{0,1\}}$: by transitivity it would be true that $\text{FSD}_{\{0,1\}} \leq_s \text{CNF}$.

Yet, **FSD** supports **CE** (Prop. 3.6), and it is impossible to find a target language at least as succinct as CNF and supporting **CE**, unless the polynomial hierarchy collapses (Lemma A.2).

As a result, $\text{FSD}_{\{0,1\}} \not\leq_s \text{SD}_{\{0,1\}}$ and thus $\text{FSD} \not\leq_s \text{SD}$, unless the polynomial hierarchy collapses.

The same proof can be used to show that $\text{dFSD} \not\leq_s \text{dSD}$ and that $\text{FSD} \not\leq_s \text{dSD}$.

$\text{dFSD} \not\leq_s \text{FSD}$. The proof is close to the one of $\text{d-DNNF} \not\leq_s \text{DNF}$ in [Darwiche and Marquis, 2002].

Suppose we had $\text{dFSD}_{\{0,1\}} \leq_s \text{FSD}_{\{0,1\}}$: Since $\text{FSD}_{\{0,1\}} \leq_{\mathcal{P}} \text{DNF}$ (Proposition 2.11), by transitivity it would be true that $\text{dFSD}_{\{0,1\}} \leq_s \text{DNF}$. Then any DNF Δ can be compiled into an equivalent polysize $\text{dFSD}_{\{0,1\}}$ $(\Delta)^*$.

Now, checking whether a clause γ is entailed by a CNF sentence Σ is equivalent to checking whether the sentence $\neg \Sigma \vee \gamma$ is valid, where $\neg \Sigma$ is a DNF. As a DNF, $\neg \Sigma$ can be compiled into a $\text{dFSD}_{\{0,1\}}$ $(\neg \Sigma)^*$ in time and size polynomial. Thanks to Lemma A.1, we can get in polytime a $\text{dFSD}_{\{0,1\}}$ equivalent to $(\neg \Sigma)^* \vee \gamma$, and, since dFSD satisfy **VA**, we can check in polytime whether $(\neg \Sigma)^* \vee \gamma$ is valid.

In summary $(\neg \Sigma)^*$ is a polysize compilation of Σ , allowing clausal entailment to be achieved in polytime. The existence of such a $(\neg \Sigma)^*$ for every CNF Σ implies the collapse of the polynomial hierarchy (Lemma A.2).

Hence, $\text{dFSD}_{\{0,1\}} \not\leq_s \text{FSD}_{\{0,1\}}$ and thus $\text{dFSD} \not\leq_s \text{FSD}$, unless the polynomial hierarchy collapses. $\square$

A.2 Conjunction with a Term and Context Extraction

FSD and dFSD support $\wedge \text{tC}$. To conjunct an SD ψ with $x \in A$, compute the restriction $\psi_{|x \in A}$ by simply removing from the x -edges the values that do not belong to A . The procedure is linear in the size of ψ and A (bounded by $\mathcal{O}(|A| \cdot |\phi|)$). The graph of ψ and $\psi_{|x \in A}$ are the same — only the labels on the edges change. Then add before the root of $\psi_{|x \in A}$ a node labeled by x , with a single outgoing edge labeled by A and pointing to the former root (since there may be paths that do not mention x).

The operation is linear and does not increase the size of that graph. Moreover, it preserves the properties of focusingness ($E \subseteq F \implies E \cap A \subseteq F$) and determinism ($E \cap F = \emptyset \implies (E \cap A) \cap (F \cap A) = \emptyset$).

Moreover, $I(\psi') = I(\psi) \wedge f_{x,A}$: By construction, for any $\vec{z}$, if $\vec{z}$ is a counter model for ψ , it is also a counter model for ψ' . Moreover any $\vec{z}$, when $\vec{z}(x)$ belongs to A , is supported by the very same path in ψ and ψ' : $\vec{z}$ is a model of $I(\psi)$ and satisfies $x \in A$, it is a model of $I(\psi_{|x \in A})$. Hence $I(\psi') = I(\psi) \wedge f_{x,A}$.

To conjunct ϕ by a term $x_1 \in A_1, \dots, x_k \in A_k$, repeat the operation for each $x_i \in A_i$. The graph of ψ and $\psi|_{x \in A}$ are the same — only the labels on the edges change. Since each elementary restriction does not increase the size of that graph, the complexity of the full procedure is bounded by $\mathcal{O}(k \cdot |\phi| \cdot \max_{1 \leq i \leq k} |A_i|)$.

As a consequence, FSD and dFSD satisfy $\wedge \mathbf{tC}$. $\square$

FSD supports CX. The following polynomial algorithm (each edge is encountered once) computes the context of y in ϕ :

```

1: reduce  $\phi$ 
2: let  $\mathcal{C} := \emptyset$ 
3: mark the sink of  $\phi$ 
4: for all node  $N$  in  $\phi$ , ordered from the sink to the root do
5:   if  $N$  is marked then
6:     for all  $E \in \text{In}(N)$  do
7:       if  $\text{Var}(E) = y$  then
8:         add  $\text{Lbl}(E)$  to  $\mathcal{C}$ 
9:       else
10:        mark  $\text{Src}(E)$ 
11: if the root of  $\phi$  is marked then
12:    $\mathcal{C} := \text{Dom}(y)$ 
13: return  $\mathcal{C} \cap \text{Dom}(y)$ 

```

The idea is to find the y -frontier of the sink (e.g. the set of the y -labeled nodes N such that there exists a path from a child of N to the sink not mentioning y), by pulling up a mark meaning that no y -labeled node has been encountered. If a mark reaches the root, there is at least one path from the root to the sink on which there is no y -labeled node, so the context of y in ϕ is $\text{Dom}(y)$ (because ϕ is reduced, so the path is trivially satisfiable, in the same way that a non-empty reduced FSD is satisfiable). If not, the context of y is the union $\mathcal{C}$ of the intervals labeling edges by which the y -frontier access the sink, intersected with the domain of y (this intersection being polytime w.r.t. the number of intervals in $\mathcal{C}$, and thus w.r.t. $|\phi|$). $\square$

A.3 Ensuring

Lemma A.3. $\text{ensure}(\mathbf{I}, \{x\}) = \bigwedge_{\vec{x} \in \text{Dom}(\{x\})} \mathbf{I}_{|\vec{x}}$.

Proof. Straightforward from the definition: let $W = X \setminus \{x\}$. $\text{ensure}(\mathbf{I}, \{x\}) = \mathbf{I}^{\downarrow W}$. Then $\text{ensure}(\mathbf{I}, \{x\})(\vec{w}) = \top$ iff $\forall \vec{x} \in \text{Dom}(\{x\})$ it holds that $\mathbf{I}(\vec{w}, \vec{x}) = \top$. Hence $\text{ensure}(\mathbf{I}, \{x\}) = \bigwedge_{\vec{x} \in \text{Dom}(\{x\})} \mathbf{I}_{|\vec{x}}$. $\square$

SD and dSD do not support EN. Given any SD ϕ , ϕ is valid iff $\text{ensure}(\mathbf{I}(\phi), \text{Var}(\phi)) \equiv \top$. Again, the only reduced SDs without any variable are the empty and the sink-only graphs. If SD were satisfying EN, we would have a polytime algorithm for deciding the validity of any SD, yet SD does not support VA unless $P = NP$. The proof is the same for dSDs. $\square$

SD and dSD support SEN. From Lemma A.3, it holds that $\text{ensure}(\mathbf{I}, \{x\}) = \bigwedge_{\vec{x} \in \text{Dom}(\{x\})} \mathbf{I}_{|\vec{x}}$. Let $n = |\text{Card}(\text{Dom}(x))|$; to obtain an SD whose interpretation function is equal to $\text{ensure}(\mathbf{I}, \{x\})$, it is sufficient to make n simple conditionings and $n - 1$ conjunctions. These two operations are linear, and n is linear in $|\phi|$, thus SEN is feasible in time polynomial in $|\phi|$. The proof is the same for dSDs. $\square$

FSD and dFSD do not support SEN, EN. Let ϕ_1 and ϕ_2 be two FSDs.

Let $Z = \text{Var}(\phi_1) \cup \text{Var}(\phi_2)$, and a variable $x \notin Z$ of domain $\{0, 1\}$.

We build the graph ψ by merging the sinks of ϕ_1 and ϕ_2 , and adding a node which will be the root of ψ , labeled by x , with one outgoing edge labeled by $\{0\}$ and pointing to the root of ϕ_1 , and a second outgoing edge labeled by $\{1\}$ and pointing to the root of ϕ_2 .

ψ is obviously focusing, since ϕ_1 and ϕ_2 are, and x is mentioned in one node only.

Let $\vec{x}$ be the 0 assignment of $\{x\}$ and $\vec{x}'$ be the 1 assignment of $\{x\}$.

By construction, $\mathbf{I}(\phi_1) = \mathbf{I}(\psi|_{\vec{x}})$ (i.e. $\vec{z}$ is a model of $\mathbf{I}(\phi_1)$ iff $\vec{z}(x) = 0$). Similarly, $\mathbf{I}(\phi_2) = \mathbf{I}(\psi|_{\vec{x}'})$ (i.e. $\vec{z}$ is a model of $\mathbf{I}(\phi_2)$ iff $\vec{z}(x) = 1$).

From Lemma A.3, it holds that $\text{ensure}(\mathbf{I}, \{x\}) = \bigwedge_{\vec{x} \in \text{Dom}(\{x\})} \mathbf{I}_{|\vec{x}}$.

Hence $\text{ensure}(\mathbf{I}(\psi), \{x\}) = \mathbf{I}(\psi|_{\vec{x}}) \wedge \mathbf{I}(\psi|_{\vec{x}'})$, that is to say: $\text{ensure}(\mathbf{I}(\psi), \{x\}) = \mathbf{I}(\phi_1) \wedge \mathbf{I}(\phi_2)$.

It is possible to build ψ in time linear in the size of ϕ_1 and ϕ_2 . Therefore, if FSD supported SEN, we would have a polytime algorithm allowing to build an FSD equivalent to the conjunction of two FSDs. Yet it is impossible, unless $P = NP$ (see $\wedge \mathbf{BC}$). Hence FSD does not support SEN, and *a fortiori* EN, unless $P = NP$.

The proof is similar for dFSDs, with two dFSDs ϕ_1 and ϕ_2 — the previous construction of ψ is a dFSD. $\square$

MDD_< and MDD do not support SEN, EN. Again, the proof is similar, building ψ from k MDDs_< with a root variable having a domain of size k . ψ is obviously an MDD bearing on the same variable order (with the root variable being inferior to all other variables). If MDD_< supported SEN, we would have a polytime algorithm allowing to build an MDD_< equivalent to the conjunction of k MDDs_<, which is impossible. Thus MDD_< does not support SEN, and *a fortiori* EN.

This also proves the result for MDD, as only one MDD is considered for this transformation. $\square$

A.4 Forgetting

MDD_< and MDD do not support SFO. We use the same mechanism as for the previous proof. If MDD_< supported SFO, we would have a polytime algorithm allowing to build an MDD_< equivalent to the disjunction of k MDDs_<, which is impossible. Thus MDD_< does not support SFO, and thus FO. This also proves the result for MDD, as only one MDD is considered for this transformation. $\square$

Anytime Depth-First Search with Problem Decomposition for Optimization in Graphical Models

Lars Otten and Rina Dechter
Department of Computer Science
University of California, Irvine, U.S.A.
{lotten,dechter}@ics.uci.edu

Abstract

One popular and efficient scheme for solving *exactly* MPE/MAP and related problems over graphical models is depth-first Branch and Bound. However, when the algorithm exploits problem decomposition using AND/OR search spaces, its anytime behavior breaks down. This paper 1) analyzes and demonstrates this inherent conflict between effective exploitation of problem decomposition (through AND/OR search spaces) and the anytime behavior of depth-first search (DFS), 2) presents a first scheme to address this issue while maintaining desirable DFS memory properties, 3) analyzes and demonstrates its effectiveness. Our work is applicable to *any* problem that can be cast as search over an AND/OR search space.

1 Introduction

Max-product problems over graphical models, generally known as MPE or MAP, have many applications with practical significance, ranging from computational biology and genetics to scheduling tasks and coding networks. One established and efficient class of algorithms for solving these problems exactly is depth-first Branch and Bound over AND/OR search spaces. Developed in the past decade within the probabilistic reasoning and constraint communities, these methods are effective because they use sophisticated lower bound schemes such as soft arc-consistency [8] or the Mini Bucket heuristic [4; 9], because they avoid redundant computation using caching schemes, and most significantly, because they take advantage of problem decomposition by exploring an AND/OR search space [11] or an equivalent representation. The efficiency of these algorithms was established in several evaluations, including recent UAI competitions [5], and their properties (for exact computation) are well documented [6; 9; 10].

A principled alternative is presented by best-first schemes, but while provably superior in terms of number of node expansions, these often fail when a problem has large induced width; moreover, they can only provide a solution at termination [10]. Depth-first search is therefore often preferred because of its flexibility in working with bounded memory and because of its *anytime behavior*. Namely, when finding

a feasible solution is easy but an optimal one is hard, depth-first Branch and Bound generates solutions that get better and better over time, until it eventually discovers an optimal one. Thus it can function also as an approximation scheme for otherwise infeasible problems or when time is limited [12].

Indeed, in the 2010 UAI challenge participating Branch and Bound solvers performed competitively wrt. approximation (placing 1st and 3rd in some categories). But we also observed an inability to produce even a single solution on some instances, especially when the time bound was small. Thus motivated, this paper will demonstrate that the issue is rooted in the underlying AND/OR search space.

Originally introduced to graphical models to facilitate problem decomposition during search (e.g. [3]), these search spaces can be explored by any search strategy. When traversed depth-first, however, all but one decomposed subproblem will be *fully solved* before a single overall solution can be composed, voiding the algorithm’s anytime characteristics.

This paper’s main contribution is a new Branch and Bound scheme over AND/OR search spaces, called *Breadth-Rotating AND/OR Branch and Bound (BRAOBB)* that addresses the anytime issue in a principled way, while maintaining the favorable complexity guarantees of depth-first search. The algorithm combines depth-first and breadth-first exploration by periodically rotating over the different subproblems, each of which is processed depth-first.

Experimental evaluation on a variety of benchmark domains compares BRAOBB against one of the best variants of AND/OR branch and Bound search, AOBB [9], and against an “ad hoc” fix that we suggest, which relies on a heuristic to quickly find a solution to each subproblem before reverting to depth-first search. The empirical results demonstrate superior anytime behavior of BRAOBB, especially over problematic cases where standard AOBB and its ad hoc fix fail. In particular, we obtained good performance on two very hard instances from the 2010 UAI challenge (Figure 5).

The work presented in this paper is focused on optimization problems defined over graphical models. As such our results are also relevant for related schemes like recursive conditioning [2] and value elimination [1] in the area of probabilistic reasoning, or BTD (Backtracking Tree Decomposition [6]) in constraint optimization. We note, however, that the presented concepts carry over to combinatorial AND/OR search spaces in general.

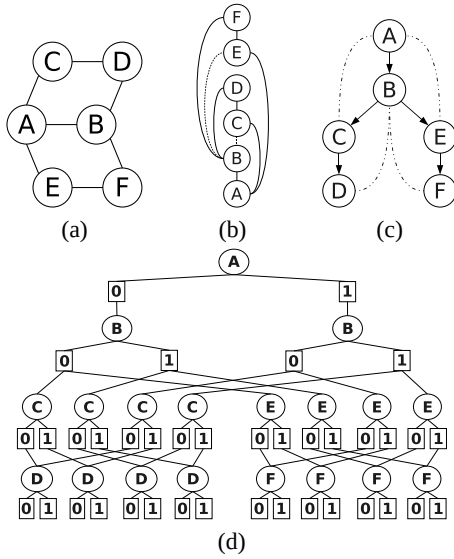


Figure 1: (a) Example primal graph of a graphical model with six variables, (b) its induced graph along ordering $d = A, B, C, D, E, F$, (c) a corresponding pseudo tree, and (d) the resulting context-minimal AND/OR search graph.

The remainder of this paper is structured as follows: Section 2 introduces the underlying concepts while Section 3 identifies the central issue and provides empirical evidence. The new algorithm BRAOBB is proposed in Section 4 and its properties analyzed. Section 5 presents experimental results and analysis before Section 6 concludes.

2 Background

We consider a MPE (Most Probable Explanation, sometimes also MAP, Maximum A Posteriori assignment) problem over a graphical model, $(X, F, D, \max, \prod)$. $F = \{f_1, \dots, f_r\}$ is a set of functions over variables $X = \{X_1, \dots, X_n\}$ with discrete domains $D = \{D_1, \dots, D_n\}$, we aim to compute $\max_X \prod_i f_i$, the probability of the most likely assignment. The set of function scopes implies a primal graph and, given an ordering of the variables, an *induced graph* (where, from last to first, each node's earlier neighbors are connected) with a certain *induced width*. Another closely related combinatorial optimization problem is the *weighted constraint problem*, where we aim to minimize the sum of all costs, i.e. compute $\min_X \sum_i f_i$. These tasks have many practical applications but are known to be NP-hard.

The concept of **AND/OR search spaces** has recently been introduced to graphical models to better capture the structure of the underlying graph during search [3]. The search space is defined using a *pseudo tree* of the graph, which captures problem decomposition:

DEFINITION 1. A pseudo tree of an undirected graph $G = (X, E)$ is a directed, rooted tree $\mathcal{T} = (X, E')$, such that every arc of G not included in E' is a back-arc in $\mathcal{T}$, namely it connects a node in $\mathcal{T}$ to an ancestor in $\mathcal{T}$. The arcs in E' may not all be included in E .

AND/OR Search Trees and Graphs : Given a graphical model instance with variables X and functions F , its primal graph (X, E) , and a pseudo tree $\mathcal{T}$, the associated *AND/OR search tree* consists of alternating levels of OR and AND nodes. Its structure is based on the underlying pseudo tree $\mathcal{T}$: the root of the AND/OR search tree is an *OR node* labeled with the root of $\mathcal{T}$. The children of an OR node $\langle X_i \rangle$ are *AND nodes* labeled with assignments $\langle X_i, x_j \rangle$ that are consistent with the assignments along the path from the root; the children of an AND node $\langle X_i, x_j \rangle$ are OR nodes labeled with the children of X_i in $\mathcal{T}$, representing conditionally independent subproblems.

Identical subproblems, identified by their context (the partial instantiation that separates the subproblem from the rest of the network), can be merged, yielding the *context-minimal AND/OR search graph* [3]. It was shown that, given a pseudo tree $\mathcal{T}$ of height h , the size of the AND/OR search tree based on $\mathcal{T}$ is $O(n \cdot k^h)$, where k bounds the variables' domain size. The context-minimal AND/OR search graph has size $O(n \cdot k^{w^*})$, where w^* is the induced width of the problem graph along a depth-first traversal of $\mathcal{T}$ [3]. Note that in Figure 1(a) the AND nodes for B have two children each, representing independent subproblems and thus demonstrating problem decomposition.

Given an AND/OR search space $S_{\mathcal{T}}$, a *solution subtree* $Sol_{S_{\mathcal{T}}}$ is a tree such that (1) it contains the root of $S_{\mathcal{T}}$; (2) if a nonterminal AND node $n \in S_{\mathcal{T}}$ is in $Sol_{S_{\mathcal{T}}}$ then all its children are in $Sol_{S_{\mathcal{T}}}$; (3) if a nonterminal OR node $n \in S_{\mathcal{T}}$ is in $Sol_{S_{\mathcal{T}}}$ then exactly one of its children is in $Sol_{S_{\mathcal{T}}}$.

AND/OR Branch and Bound (AOBB) is a state-of-the-art algorithm for solving optimization problems such as max-product over graphical models. The edges of the AND/OR search graph can be annotated by weights derived from the set of cost functions F in the graphical model; finding the optimal-cost solution subtree solves the stated optimization task. Assuming a maximization query, AOBB traverses the weighted context-minimal AND/OR graph in a depth-first manner while keeping track of the current lower bound on the maximal solution cost. A node n will be pruned if this lower bound exceeds a heuristic upper bound on the solution to the subproblem below n , often obtained by solving a relaxed problem (e.g. through Mini Buckets [7]). The algorithm interleaves forward node expansion with a backward cost revision or propagation step that updates node values (capturing the current best solution to the subproblem rooted at each node), until search terminates and the optimal solution has been found. [9].

3 Anytime versus AND/OR

We will use AOBB to denote the algorithm above in its specific graphical models context as well as a generic name for any depth-first Branch and Bound scheme over an AND/OR search space. As a depth-first scheme one would expect AOBB to quickly produce a non-optimal solution and then gradually improve upon it, maintaining the current best one throughout the search. However this ability is compromised in the context of AND/OR search.

Specifically, in AND/OR search spaces depth-first traversal

of a set of independent subproblems will solve to completion all but one subproblem before the last one is even considered. As a consequence, the first generated overall non-optimal solution contains conditionally optimal solutions to all subproblems but the last one. Furthermore, depending on the problem structure and the complexity of the independent subproblems, the time to return even this first non-optimal overall solution can be significant, practically negating the anytime behavior of depth-first search (DFS).

3.1 Subproblem Ordering

In certain cases, the above suggests a simple remedy: if decomposition yields only one large subproblem and several smaller ones, the latter can be solved depth-first in relatively little time, to be then combined with the incrementally improving solutions of the larger subproblem. Thus for anytime behavior an AOBB algorithm would need to process independent subproblems from “easy” to “hard”.

To demonstrate the practical impact of subproblem orderings, we use a simple heuristic that takes the induced width as a measure of subproblem hardness (motivated by its exponential role in the asymptotic complexity), i.e. we modify AOBB such that subproblems with smaller induced width will be processed first (in the general description of AOBB the subproblem ordering is left unspecified).

Figure 2 contrasts the anytime performance of AOBB using this “increasing” subproblem order against the inverse one (“decreasing”) by plotting the probability of the best assignment found over time on a set of example problems; all other aspects of the algorithm remain constant. Pedigree30x1 in particular features exactly one single complex subproblem and a number of relatively simple ones; in this case processing subproblems by increasing induced width right away produces a non-optimal solution that improves rapidly. The inverse order yields the first solution only after about 90 minutes – the one complex subproblem has been fully solved and the overall solution is already optimal. Pedigree41x1 has a similarly advantageous structure and thus yields similar results – with the distinction that the inverse subproblem order does not produce any solution at all within 24 hours.

In case of pedigree34x2, however, decomposition yields two complex subproblems: the increasing subproblem order still outperforms its inverse, yet it returns the initial solution only after about 1,000 seconds. In fact, no possible subproblem ordering can lead to acceptable anytime behavior in this case due to the structure of subproblems, clearly highlighting the limits of this approach.

Independent of anytime behavior, we point out that incorporating different subproblem orderings impacts the algorithm’s overall efficiency (i.e., the time to finding and proving an optimal solution): knowing the solution to one subproblem can aid the pruning of Branch and Bound in the next one to varying degrees. However, this issue has not been treated systematically in the literature for graphical models, with sporadic experiments also suggesting an easy-to-hard order, using some heuristic to determine subproblem complexity [9]. This general problem is outside the scope of the present paper, however.

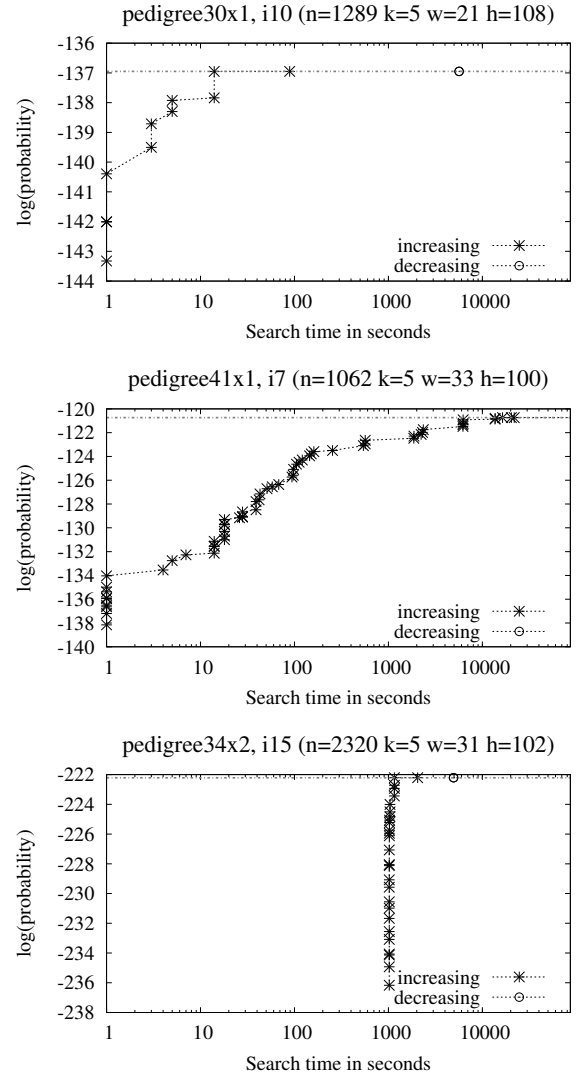


Figure 2: Impact of subproblem ordering on AOBB. The number of variables n , the max. domain size k , the induced width w along the chosen ordering, as well as the height of the corresponding pseudo tree h are specified for each network.

3.2 Greedy Subproblem Dive

Another relatively straightforward remedy that can be viewed as an “ad hoc” fix is the following: Every time decomposition is encountered within the search space, we will try to greedily find a single initial solution to each independent subproblem before successively solving each of them to completion depth-first, through normal AOBB. To obtain this initial solution the algorithm can perform a greedy “dive” into each subproblem by only considering one value for each variable along the path (in case of the Mini Bucket heuristic, it is easy to see that this is equivalent to a forward pass over the bucket structure [7]).

Clearly, the choice of the dive path is crucial for the algorithm’s performance. Namely, if the chosen path leads to a dead end (zero probability), the dive will be futile and

not yield a subproblem solution. And in fact experiments in Section 5 will demonstrate that the resulting performance depends heavily on the quality of the heuristic, which often prevents satisfactory anytime behavior. In the next section we will therefore propose a new search strategy that addresses the anytime issue over AND/OR search spaces in a principled manner.

4 Breadth-Rotating AOBB

In the following we develop a new search scheme called *Breadth-Rotating AND/OR Branch and Bound (BRAOBB)* that addresses the issue of anytime performance over AND/OR search spaces. It combines depth-first exploration with the notion of “rotating” through different subproblems in a breadth-first manner. Namely, node expansion still occurs depth-first as in standard AOBB, but the algorithm takes turns in processing subproblems, each up to a given number of operations at a time, round-robin style.

To motivate this approach, consider again that a solution is represented by a *solution tree* over an AND/OR search space, guided by a pseudo tree. A pure DFS scheme will construct the different branches of a solution tree one by one, ensuring optimality for each branch before moving to the next. To restore anytime behavior, we instead aim to develop all branches of the solution tree “simultaneously”, which we emulate by rotating through them.

More systematically, the algorithm repeats the following high-level steps until completion:

1. Move breadth-first to next open subproblem P .
2. Process P depth-first, until either:
 - P is solved optimally,
 - P decomposes into child subproblems, or
 - a predefined threshold of operations is reached.

The threshold is needed to ensure the algorithm does not get stuck in one large subproblem (where the other two conditions do not occur for a long time). Furthermore, in order to focus on a single solution tree at a time, a subproblem is only considered “open” if it does not currently have any open child subproblems, as illustrated below.

Algorithm 1 gives more detailed pseudo code for the scheme (with some details from standard AOBB omitted, cf. [9]). The key element lies in rotating over the different subproblems of the search space; by organizing these in a global first-in-first-out queue (*GLOBAL*), we emulate breadth-first exploration across the different branches of the solution tree. The input parameter Z determines how many nodes should be processed before moving on to the next subproblem (if none of the other conditions are satisfied first). Each subproblem is itself explored depth-first (via a local last-in-first-out stack of nodes, *LOCAL*); whenever a new level of decomposition is encountered, as captured by the pseudo tree, the resulting child subproblems are pushed to the end of the global queue. Finally, subproblems are only considered in the rotation if they don’t currently have any open child subproblems. The following example illustrates:

Example : Figure 3 demonstrates the scheme’s application ($Z = 2$) to the AND/OR search graph in Figure 1 (assuming

Algorithm 1 Breadth-Rotating AOBB

Given: Graphical model $(X, F, D, \max, \prod)$ and pseudo tree $\mathcal{T}$ with root X_o , rotation threshold Z

Output: cost of optimal solution

```

1:  $ROOT \leftarrow \{X_o\}$  // generate root subproblem
2: push  $ROOT$  to end of  $GLOBAL$ 
3: while  $GLOBAL \neq \emptyset$ 
4:    $LOCAL \leftarrow \text{front}(GLOBAL)$  // next subproblem in queue
5:   for  $z \leftarrow 1$  to  $Z$  or until  $LOCAL = \emptyset$ 
     or until  $\text{childSubprob}(LOCAL) \neq \emptyset$ 
6:      $n \leftarrow \text{top}(LOCAL)$  // top node from current subproblem
7:     ... // caching and pruning as in AOBB
8:     if  $n = \langle X_i \rangle$  is OR node
9:       for  $x_j \in D_i$ 
10:        create AND child  $\langle X_i, x_j \rangle$ 
11:        add  $\langle X_i, x_j \rangle$  to top of  $LOCAL$ 
12:     else if  $n = \langle X_i, x_j \rangle$  is AND node
13:        $Y_1, \dots, Y_m \leftarrow \text{children}_{\mathcal{T}}(X_i)$ 
14:       generate OR children  $\langle Y_1 \rangle, \dots, \langle Y_r \rangle$ 
15:       if  $m=1$  // no decomposition
16:         push  $\langle Y_1 \rangle$  to top of  $LOCAL$ 
17:       else if  $m > 1$  // problem decomposition
18:         for  $r \leftarrow 1$  to  $m$ 
19:            $NEW \leftarrow \{ \langle Y_r \rangle \}$  // new child subproblem
20:           push  $NEW$  to back of  $GLOBAL$ 
21:       if  $\text{children}(n) = \emptyset$  //  $n$  is leaf
22:         propagate( $n$ ) // upwards in search space
23:       if  $LOCAL \neq \emptyset$  // subproblem not yet solved
24:         push  $LOCAL$  to end of  $GLOBAL$ 
25: return value( $\langle X_o \rangle$ ) // root node contains optimal solution
```

no pruning). Part (a) shows the first 12 nodes expanded during the first seven iterations of the outer while loop as follows: (1) Taking the overall problem as subproblem P0, expand $\langle A \rangle$ and $\langle A, 0 \rangle$ before reaching the threshold $Z = 2$. (2) With no decomposition so far rotation returns to subproblem P0. Expand $\langle B \rangle$ and $\langle B, 0 \rangle$, yielding subproblems P1 and P2 rooted at $\langle C \rangle$ and $\langle E \rangle$, respectively, which are added to the queue. (3) Rotate to subproblem P1 and expand $\langle C \rangle$ and $\langle C, 0 \rangle$. (4) Rotate to subproblem P2. Expand $\langle E \rangle$ and $\langle E, 0 \rangle$. (5) Rotate to subproblem P0 but skip it at this point, since its child subproblems P1 and P2 are still open. (6) Rotation moves to subproblem P1. Expand $\langle D \rangle$ and $\langle D, 0 \rangle$, discover a leaf and propagate. (7) Rotate to subproblem P2, expand $\langle F \rangle$ and $\langle F, 0 \rangle$ – which, as a leaf, is propagated to yield the first overall solution.

Figure 3(b) illustrates how the search then proceeds to take turns solving subproblems P1 and P2 to completion (nodes 13–22) before reopening subproblem P0. Expansion 23 yields two new independent subproblems P3 and P4; their solution is depicted by nodes 24–41. After that subproblem P0 gets reopened, where expanding nodes 42–44 again yields two new subproblems P5 and P6, and so forth.

4.1 Analysis of Breadth-Rotating AOBB

Complexity : We assume a graphical model with n variables whose domain size is bounded by k . Let w^* be the induced width of the problem along a given ordering and h the height of the corresponding pseudo tree $\mathcal{T}$. Despite the breadth-first component the algorithm maintains the asymptotic complex-

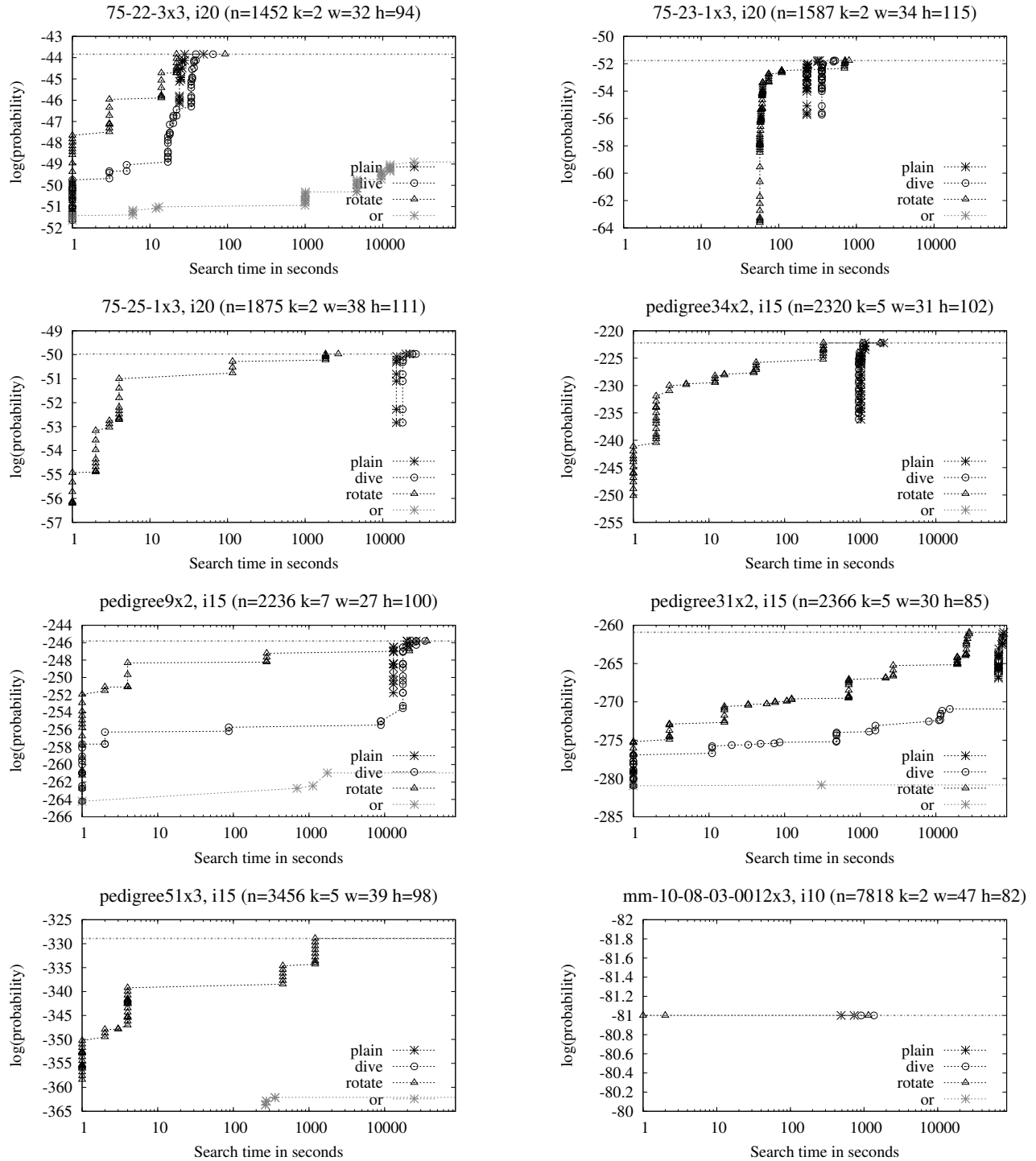


Figure 4: Anytime profiles of plain AOBB, AOBB with subproblem dive, BRAOBB, and OR Branch and Bound on selected instances with more than one hard subproblem (3 grids, 4 pedigree, 1 mastermind network).

which any solution was found, for which the optimal solution was found, and for which optimality was proven (i.e. the algorithm terminated). The results confirm that BRAOBB yields superior anytime performance: for example, within 1

second it provides an initial solution on 293 instances (out of 345), compared to just 98 for plain AOBB and 131 for the dive extension; this lead is maintained for higher time bounds. BRAOBB also finds the optimal solution quicker

Table 1: Summary statistics over 345 instances for each scheme: given are the number of cases for which, within the respective time bound, (1) any solution was found, (2) the optimal solution was found, (3) optimality was proven.

	Time bound						
	1 sec	5 sec	10 sec	1 min	5 min	1 hour	24 hours
Pedigree networks (171 total)							
plain	52 / 19 / 6	70 / 36 / 17	75 / 42 / 24	87 / 56 / 48	101 / 76 / 68	111 / 90 / 86	129 / 117 / 108
dive	76 / 16 / 5	86 / 29 / 13	94 / 38 / 20	105 / 53 / 48	116 / 69 / 64	127 / 89 / 86	135 / 114 / 105
rotate	153 / 26 / 2	157 / 40 / 15	160 / 47 / 24	162 / 59 / 48	164 / 74 / 60	165 / 98 / 84	167 / 127 / 102
or	73 / 6 / 1	76 / 7 / 5	77 / 10 / 5	79 / 10 / 9	82 / 12 / 11	87 / 16 / 15	90 / 22 / 21
Grid networks (150 total)							
plain	38 / 10 / 0	48 / 19 / 0	58 / 32 / 4	84 / 62 / 52	101 / 82 / 76	128 / 120 / 113	149 / 148 / 147
dive	47 / 6 / 0	52 / 12 / 0	55 / 24 / 1	82 / 54 / 37	97 / 78 / 71	121 / 111 / 104	147 / 147 / 146
rotate	122 / 16 / 0	128 / 27 / 0	129 / 35 / 1	136 / 69 / 38	143 / 86 / 73	146 / 126 / 110	149 / 149 / 147
or	45 / 0 / 0	45 / 1 / 0	46 / 1 / 0	53 / 2 / 0	57 / 4 / 2	64 / 10 / 9	74 / 21 / 21
Mastermind networks (24 total)							
plain	8 / 8 / 1	8 / 8 / 3	8 / 8 / 3	10 / 10 / 4	13 / 13 / 7	17 / 17 / 12	24 / 24 / 24
dive	8 / 8 / 1	8 / 8 / 3	8 / 8 / 3	11 / 11 / 5	12 / 12 / 6	21 / 21 / 19	24 / 24 / 24
rotate	18 / 18 / 1	18 / 18 / 3	18 / 18 / 3	18 / 18 / 3	21 / 21 / 4	24 / 24 / 19	24 / 24 / 24
or	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0	0 / 0 / 0

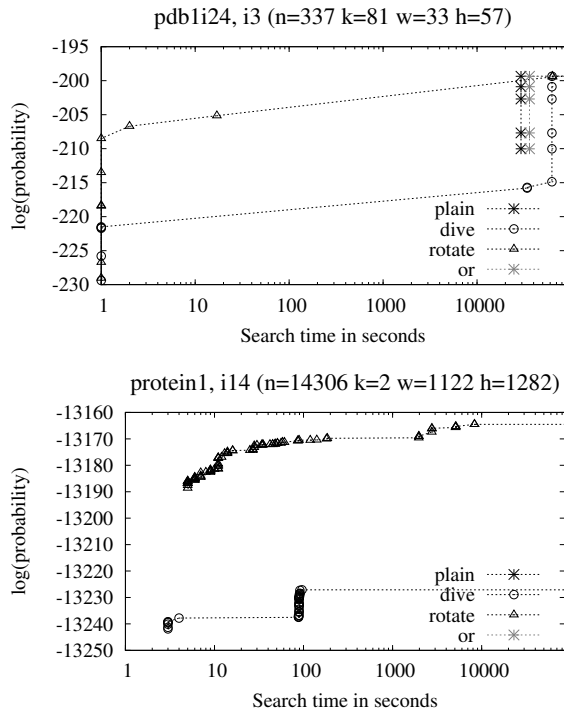


Figure 5: Anytime profiles on two example instances from the UAI'10 challenge (top: protein folding, bottom: protein-protein interaction).

than the other schemes, e.g., for 100 instances after 10 seconds (versus 80 instances for plain). Finally, we see that plain AOBB has a slight edge in terms of proving optimality, confirming that exploring subproblems concurrently can slightly impair the pruning (cf. Section 4.1).

Notably, BRAOBB did indeed restore anytime performance on example problems from the UAI 2010 inference

challenge where plain AOBB failed [5]. Out of the ten instances made available, eight were solved in seconds by standard AOBB and hence not further considered. Anytime results on the remaining two, modeling protein folding and protein-protein interaction, are shown in Figure 5, once more demonstrating superiority of BRAOBB (note the large domain size of pdb1i24, the massive induced width of protein1, and that optimality for either problem could not be proved within 24 hours).

To investigate to what extent the different schemes depend on the heuristic's accuracy, Figure 6 (top) contrast plain AOBB, dive, and BRAOBB each with two different heuristics, parametrized by the Mini Bucket i -bound (where higher is better) [7]. Plain AOBB fails or does very poorly due to problem decomposition; AOBB with dive depends very much on the heuristic and fails with the weaker one. BRAOBB, however, exhibits acceptable anytime behavior even with the weaker heuristic.

Going back to Section 3.1, Figure 6 (middle) compares the performance of BRAOBB with subproblems ordered by increasing and decreasing width. In contrast to AOBB (included for reference) our new scheme is very robust and delivers nearly the same performance in both cases. Finally, experiments with different values for the rotation threshold $Z \in 10, 1000, 100000, 10000000$ in Algorithm 1 showed no significant difference in performance as exemplified by Figure 6 bottom, confirming the analysis in Section 4.1.

6 Summary

Exploiting problem decomposition in search methods has been proven to yield significantly better overall complexity in many cases. Yet this paper has demonstrated how it can be in direct conflict with the depth-first nature of Branch and Bound, thus impairing the important anytime properties of this class of algorithms.

We devised a “quick fix” that employs an initial greedy subproblem dive, but whose performance was lacking due to

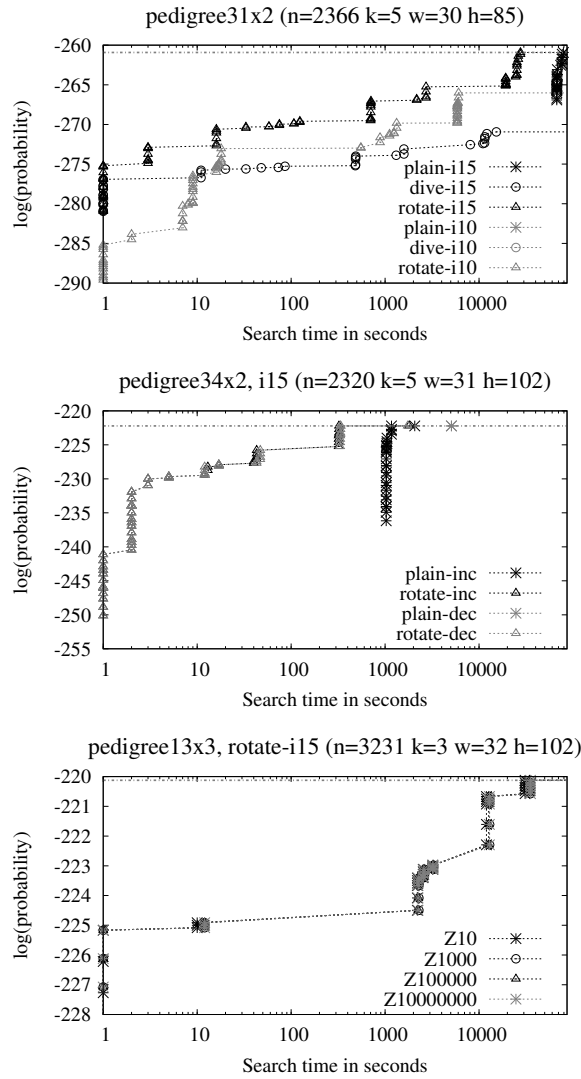


Figure 6: *Top*: impact of heuristic accuracy on anytime performance. *Middle*: impact of subproblem ordering. *Bottom*: impact of rotation threshold Z .

heavy dependence on the underlying heuristic.

The main contribution of this work is the new scheme *Breadth-Rotating AND/OR Branch and Bound (BRAOBB)*, which periodically iterates over the different subproblems in a “breadth-first” manner but was shown to retain many desirable properties of the depth-first strategy. In particular, its memory complexity remains linear in the number of variables (not accounting for caching).

We presented a large set of successful experiments that confirmed vastly improved anytime performance, especially in cases where standard depth-first Branch and Bound and its “ad hoc” extensions fail, including two hard instances from the UAI 2010 challenge. Through analysis and experiments we also showed our new scheme to be robust with respect to the order of subproblems as well as the accuracy of the guiding heuristic.

Acknowledgements

This work was partially supported by NSF grants IIS-0713118, IIS-1065618 and NIH grant 5R01HG004175-03.

References

- [1] F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *UAI*, pages 20–28, 2003.
- [2] A. Darwiche. Recursive conditioning. *Artif. Intell.*, 126(1-2):5–41, 2001.
- [3] R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artif. Intell.*, 171(2-3):73–106, 2007.
- [4] R. Dechter and I. Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM*, 50(2):107–153, 2003.
- [5] G. Elidan and A. Globerson. UAI 2010 approximate inference challenge. <http://www.cs.huji.ac.il/project/UAI10/>.
- [6] P. Jégou and C. Terrioux. Decomposition and good recording for solving max-CSPs. In *ECAI*, pages 196–200, 2004.
- [7] K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. *Artif. Intell.*, 129(1-2):91–131, 2001.
- [8] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artif. Intell.*, 159(1-2):1–26, 2004.
- [9] R. Marinescu and R. Dechter. AND/OR Branch-and-Bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17):1457–1491, 2009.
- [10] R. Marinescu and R. Dechter. Memory intensive AND/OR search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17):1492–1524, 2009.
- [11] N. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.
- [12] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.

Visually Interacting with a Knowledge Base Using Frames, Logic, and Propositional Graphs

Daniel R. Schlegel and Stuart C. Shapiro

University at Buffalo

Buffalo, New York, USA

{drschleg | shapiro}@buffalo.edu

Abstract

The knowledge base of a knowledge representation and reasoning system can simultaneously be thought of as being logic, frame, and graph-based. We present a method for naturally extending this three-fold view to methods for visual interaction with the knowledge base in the context of SNePS 3 and its newly developed user interface. Assertions to, and search of, the knowledge base are tasks well suited to a frame or logical representation. Visualization on the other hand is best done through the use of propositional graphs. We show how these interaction techniques, which are extensions of the underlying knowledge base representation, augment each other to allow knowledge engineers to manipulate and view large knowledge bases.

1 Introduction

The knowledge base (KB) of a knowledge representation and reasoning (KR) system can be conceived of as a set of logical expressions, as a set of frames, or as a graph. The choice determines how the user interface presents the KB to the user or knowledge engineer. The KB of the SNePS 3 KR system [Shapiro, 2000] can be thought of as all three simultaneously, and the recently developed SNePS 3 user interface makes use of all three paradigms. In this paper, we focus on the presentation of a SNePS 3 KB as a graph, and on the way that conceiving of it as a set of frames and logical expressions influences the visualization of the graph.

The SNePS 3 graphical user interface (GUI, Figure 1) highlights the visualization of a SNePS 3 KB as a propositional graph. It allows the user to add to the KB using either the logic or frame paradigm, and to select a part of the KB for graphical display using frames in a relational database-esque visual query by example (QBE) way. The user can also hide or show specific instances of a relation through interactions with the graph itself. Binary relations in the graph can be collapsed to provide a less cluttered display without changing the semantics of the graph. In this way, most of the interaction with the knowledge base in terms of assertion and search is carried out in a frame-based way, but visualization of the result is largely graph-based.

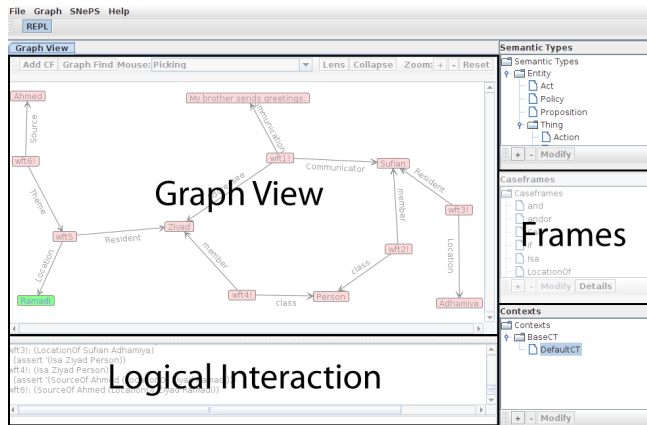


Figure 1: The SNePS 3 GUI supports graph, logic, and frame interactions.

In Section 2 we introduce SNePS 3, its three-fold knowledge base representation, and the relationships between those three views. We discuss frame-based interactions with the knowledge base in Section 3 and collapsing binary relations in the graph in Section 4. Finally the usefulness of the system will be discussed along with a comparison with ontology editing software in Section 5.

2 SNePS 3

2.1 Logical Expressions

A SNePS 3 KB may be thought of as a set of logical expressions. For example, the first eight lines of Figure 2 show a user using the logical interaction pane of the SNePS 3 GUI to assert four atomic propositions, and the GUI echoing them back, each labeled with an internal name of the form `wfti!`. The choice of “wft” to begin the name will be explained shortly. The “!” is appended to the name to indicate that the proposition is asserted (considered to be True in the KB). The syntax used for logical expressions is a version of CLIF [ISO/IEC, 2007], and includes: `not`; `if`; the set-oriented connectives `and`, `or`, `nand`, `nor`, `xor`, `iff`, `andor`, and `thresh` [Shapiro, 2010]; and arbitrary and indefinite terms

```

: (assert '(Call Sufian Ziyad "My brother sends greetings.))
wft1!: (Call Sufian Ziyad |My brother sends greetings.|)
: (assert '(Isa Sufian Person))
wft2!: (Isa Sufian Person)
: (assert '(LocationOf Sufian Adhamiya))
wft3!: (LocationOf Sufian Adhamiya)
: (assert '(Isa Ziyad Person))
wft4!: (Isa Ziyad Person)
: (assert '(SourceOf Ahmed (LocationOf Ziyad Ramadi)))
wft6!: (SourceOf Ahmed (LocationOf Ziyad Ramadi))

```

Figure 2: A set of assertions in the text-based logical interface to SNePS 3.

[Shapiro, 2004].¹

To facilitate metaknowledge—knowledge about knowledge [Shapiro *et al.*, 2007], propositions are considered to be first-class objects in the semantic domain, and all well-formed SNePS logical expressions, including those that look like formulas, are terms [Shapiro, 1993]. So what look like predicates (for example, “Call”, “Isa”, “LocationOf”, and “SourceOf” in Figure 2) are actually proposition-valued functions. Even “logical connectives” are function symbols. The internal names of SNePS 3 expressions start with “wft” as a reminder that they are “well-formed terms.” This is illustrated in the last assertion of Figure 2, which is intended to represent the proposition that Ahmed is the source of the proposition that the location of Ziyad is Ramadi. Taken together, the assertions of Figure 2 are intended to mean that “*Sufian, a person in Adhamiya, called Ziyad, a person who, according to Ahmed, is in Ramadi, saying ‘My brother sends greetings.’*”

2.2 Frames

Before a function symbol can be used, it must be associated with a caseframe. A caseframe has a *semantic type* t , specifies a *slot* s_i for each argument x_i of the function, and is associated with one or more function symbols. *Isa* is predefined in SNePS 3 as associated with a caseframe whose slots are *member* and *class* and whose semantic type is *Proposition*. The slots in a caseframe are ordered as they are upon its definition, corresponding to the order of the arguments of the function it is meant to represent.

The user interface we have developed allows the graphical definition of such a caseframe. Figure 3 shows the graphical definition of the *Isa* caseframe. The user selects the slots they are interested in and moves them to the right column. The order they are in from top to bottom represents the ordering of the slots in the caseframe. This graphical view is automatically converted to its appropriate representation in the logical interface to SNePS 3 and displayed in the user interface (see Figure 4). Also displayed in the user interface is the current listing of defined caseframes, which in Figure 5 now includes the *Isa* caseframe.

A *frame* is an instance of a caseframe. That is, the term $F(x_1, \dots, x_n)$ is represented by the frame of type t whose

¹SNePS 3 examples shown in this paper will be limited to ground atomic propositions.

Figure 3: The user interface for defining a caseframe, populated with the slots *member* and *class* for defining the *Isa* caseframe.

```

: (defineCaseframe 'Proposition '(Isa member class))
#<caseframe: 'Isa
type:    #<standard-class Proposition>
slots:   member
         class
>

```

Figure 4: The logical interface displaying the execution of the input in Figure 3 in the logical language.

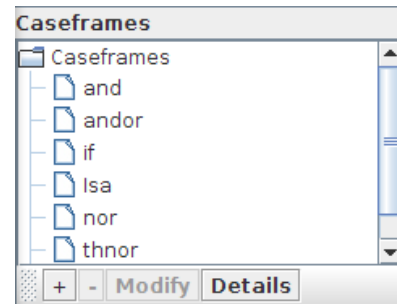


Figure 5: The list of defined caseframes, including the *Isa* caseframe.

slots are $s_1, \dots, s_n$ filled by the representations of $x_1, \dots, x_n$, respectively. So *wft2* is a frame whose *member* slot contains the *filler* *Sufian*, and whose *class* slot contains the filler *Person*. In this example, *Call* is associated with a frame whose slots are *Communicator*, *Addressee*, and *Communication* (eg. *wft1*), *LocationOf* is associated with a frame whose slots are *Resident* and *Location* (eg. *wft3*), and *SourceOf* is associated with a frame whose slots are *Source* and *Theme* (eg. *wft6*).²

²The *Call*, *LocationOf* and *SourceOf* frames are based

Caseframes, motivated by Fillmore’s case theory [Fillmore, 1968], are comparable to relational database schemas, with slot names corresponding to attributes, and frames to rows of a relational database table. There are several differences, however: a SNePS frame slot may be filled with another frame; and a SNePS frame slot may have multiple fillers. Multiple fillers in a slot are interpreted conjunctively. If one frame has two fillers in one slot and three in another, the frame is semantically equivalent to six frames with only one filler in each of the two slots.

2.3 Propositional Graphs

A SNePS 3 KB may be conceived of as a graph. Every term, whether an individual constant, an arbitrary or indefinite term,³ or a functional term (whether or not denoting a proposition), corresponds to a node in the graph. Slot names label directed arcs that go from nodes corresponding to frames (functional terms) to the nodes that correspond to the fillers of that slot in that frame (arguments of the function). This is illustrated in Figure 6, which shows the same KB created in Figure 2. Recall that an exclamation mark, “!”, is appended to the *wft* names indicating those propositions that are asserted in the KB. For example, *wft6!* is asserted to indicate that it is true that Ahmed was the source of the information that Ziyad is in Ramadi, but *wft5* is not asserted indicating that Ahmed is not (yet?) believed. The graph is rendered within the GUI by the JUNG (Java Universal Network/Graph) system [The JUNG Framework Development Team, 2010].

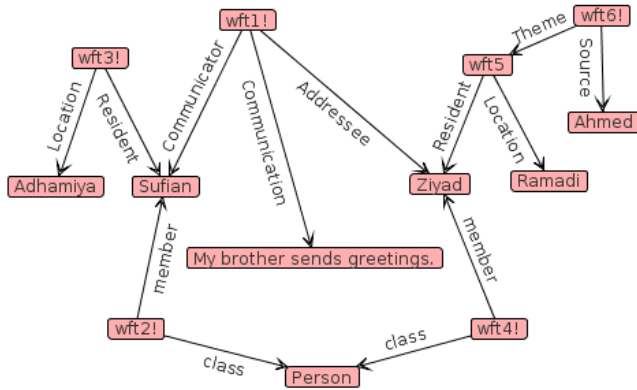


Figure 6: A graphical view of the SNePS 3 KB created in Figure 2, meaning “*Sufian, a person in Adhamiya, called Ziyad, a person who, according to Ahmed, is in Ramadi, saying ‘My brother sends greetings.’*”

A SNePS 3 graph may also be considered a hypergraph with labeled nodes and edges. The *wft* nodes correspond to hyperedges, with the benefit, however, that they can have arcs pointing into them (for example, *wft5* of Figure 6). We believe that visualizing the graphs as in Figure 6 is clearer than

on the Contacting, Residence, and Source_of_getting frames of FrameNet [Fillmore, 1976; Ruppenhofer *et al.*, 2006], but simplified for this paper.

³Arbitrary and indefinite terms are not illustrated in this paper.

replacing the *wft* nodes with contours. However, a simplification is discussed and illustrated in §4 below.

3 Frame-Based Interaction

The addition of new knowledge to the KB requires that all slots of the desired caseframe are filled with the proper number of fillers, specified during the definition of the slots themselves. This number can vary between instances of a single caseframe. This is difficult to enforce graphically; the caseframe would have to be instantiated one piece at a time, leading to periods where the graph being edited would be syntactically invalid and inconsistent with the KB that supports it. Relaying in what ways the graph is incorrect and how the issues should be resolved to the user is very difficult. Our solution to this problem is for the user to add knowledge using either the traditional logic based interface or a frame based interface. As you can see in the logical interface as described above, information is added to the KB using logical representations of frames, so the visual interface we have developed is a conceptual extension of the logical language. In fact, when assertions are added to the KB using the frame-based user interface, they are converted to the logical language first and displayed in the logical interface portion of the GUI.

The frame-based interface we have devised for assertions can be seen in Figure 7. The user selects the caseframe which they wish to add an instance of to the KB. This automatically provides a frame based view of the slots which require fillers. Conceptually frames are often seen as tables of slots and fillers, and this is the metaphor we use here. As previously mentioned, the slots in a SNePS 3 caseframe may be configured such that they may exist in the frame a variable number of times. For example, a single instance of the caseframe Isa can be used to say that both Toto and Fido are instances of both Dog and Pet. In the interface the user may add instances of slots to the frame which then must be assigned values by the user. In using an interface such as this, the inputs required by the user become obvious and easily machine verifiable. No changes to the graph, and hence KB, are made until the change is guaranteed to result in a valid assertion to the KB.

Queries can be performed on the KB and the results displayed in the graph using Query By Example [Zloof, 1975]. This is the same as the visual QBE interface designed for database systems wherein the user is shown an empty row for a table and can enter values into fields they are interested in in order to filter results. This speaks to the general likeness of a KB with a database system wherein frames are like rows in a database table and the caseframe defines the table itself.⁴

Using our method, the knowledge engineer can display only the parts of a graph matching certain parts of a relation. In Figure 8 the interface we use for the QBE process is shown. The engineer first chooses the relation (caseframe) they are interested in and chooses to provide fillers for none, some, or all of the slots shown. All areas not completed in

⁴The analogy begins to break down when we consider two abilities which differentiate a caseframe from a database table: the ability to have a variable number of instances of a slot act as the fillers of a frame, and the ability to fill a slot with an instance of another frame.

Select Caseframe: **Isa**

Assertion Type: **Definite**

Slot Fillers

Slot Name	Value
member	
class	

Add slot instance: **member** **Add**

OK **Cancel**

Figure 7: A frame-based interface for assertions to the KB.

the query interface will automatically be converted to variables $\{?w_1, \dots, ?w_n\}$. The system converts the text entered into the logical language and queries the KB. The resulting matches are displayed on the graph and all others are hidden.

In this example, the user is interested in only instances of the Isa caseframe in which the slot for class has filler Person. We will apply this frame-based query to the KB containing the knowledge about the call Sufian made to Ziyad discussed earlier. The system filters the graph, as seen in Figure 9, to contain only the graphical representations of Isa frames in which the slot for class has the filler Person.

Select Caseframe: **Isa**

Slot Fillers

Slot Name	Value
member	
class	Person

Add slot instance: **member** **Add**

OK **Cancel**

Figure 8: The QBE dialog box indicating a query for instances of the Isa caseframe with the class Person.

In addition, the user may themselves use variables in their query by prepending a question mark, “?”, to the variable symbol, a feature useful for querying metaknowledge. For example, a user may view the graph with all instances of the SourceOf caseframe where the Source slot contains Ahmed and the Theme slot contains (LocationOf Ziyad ?x) to display all relations representing locations

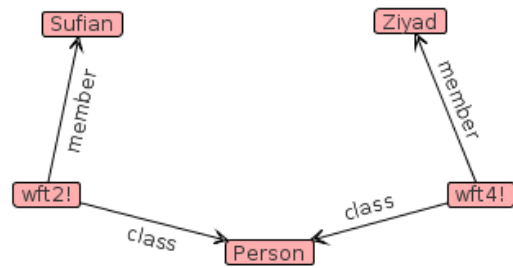


Figure 9: The result of finding instances of the Isa caseframe with the class Person in the calling example discussed earlier.

Ahmed has said Ziyad has been. At this time the user must enter (LocationOf Ziyad ?x) into the filler of the Theme slot manually, though it would be advantageous to have an interface allowing for embedded queries.

The graph the user is interacting with may contain relations not of interest, or perhaps not all the relations of interest. In this case the user may show or hide the relations attached to a node by right clicking a node and selecting the appropriate option. They may choose to show or hide the remainder of the relations with arrows either in to or out of the selected node. This shows or hides entire frames from the graph view. That is, it is never the case that a wft node with only some of the arrows out of it will be visible. One place this has particular use is once a QBE filter has been applied. Consider the example shown in Figure 9. We can choose to show all of the relations attached to the node Sufian if we are interested in knowledge associated with him, resulting in Figure 10. In this way the knowledge engineer can begin with a QBE-based query and only expand nodes of interest.

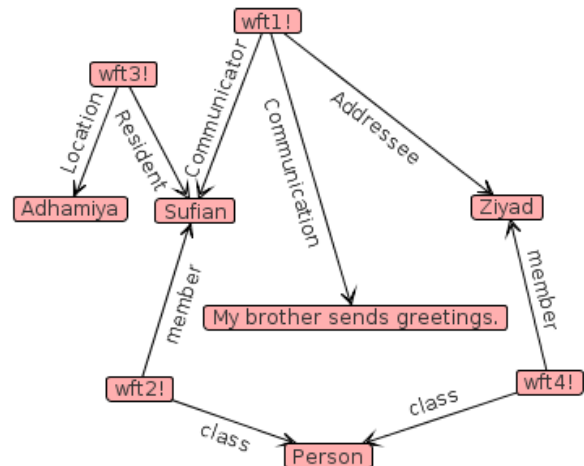


Figure 10: An expansion of the graph shown in Figure 9 wherein relations attached to the node Sufian are shown.

This example is of course quite trivial given the size of the graph being represented. In a much larger KB this becomes extremely useful as a method to visualize specific portions of a KB. The SNePS 3 user interface is currently being used

to visualize large propositional graphs for soft information fusion [Prentice *et al.*, 2010]. The graphs being visualized are many thousands of nodes.

In order to see the usefulness of QBE on a larger scale, we consider a knowledge base from this project containing 2,075 terms representing place names and various information about the places from the NGA GEOnet Names Server [National Geospatial-Intelligence Agency, 2011]. When these terms are added to the knowledge base and visualized, the graph shown in Figure 11 is produced. Large KBs are very difficult to work with conceptually as there is a significant burden on the knowledge engineer to understand the complex structure. The purpose of visualizing a KB though, should be to give the knowledge engineer an idea of the contents of the KB while reducing this burden as much as possible. As you can see, just visualizing the graphical version of the KB is unhelpful - it's impossible to pick out even a single relation.

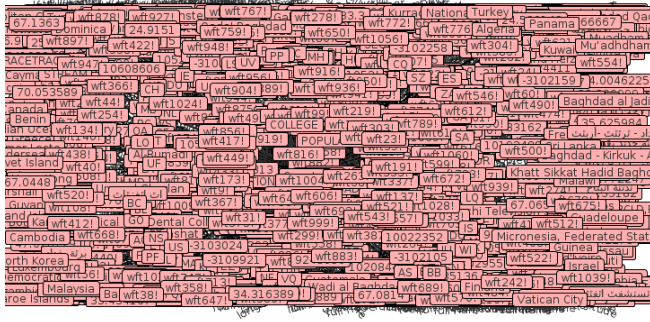


Figure 11: A large network of places and knowledge about them from the NGA GEOnet Names Server.

Very rarely is a knowledge engineer really interested in the entire KB at one time though, and constraining the display would be useful. Assume we are really only concerned with places that are countries, and we'd like to see the nodes for all of those. We can do this with a simple QBE-based query (by filling Country in to the class slot of the Isa caseframe in the user interface, and leaving the filler of the member slot empty). The graph is then restricted to showing only the relevant knowledge, as shown in Figure 12.

As you can see the resulting graph is much easier to understand; an underlying structure is visible and a node of interest is easier to pick out. The knowledge engineer could easily find and expand a country node if she were interested in the relations attached to it. You will notice though that the propositional graph contains many wft nodes which may not be helpful to the user of the graph. Under some situations these can be removed by redrawing, or collapsing the graph.

4 Collapsing the Graph

It is important to display the wft nodes in the graph either if the function has more than two arguments, as is the case for wft1 in our calling example, or if the term is, itself, an argument of another function, as is the case for wft5. However, in the case of functional terms with just two slots that are not fillers of other slots, such as wft2, wft3, wft4 and

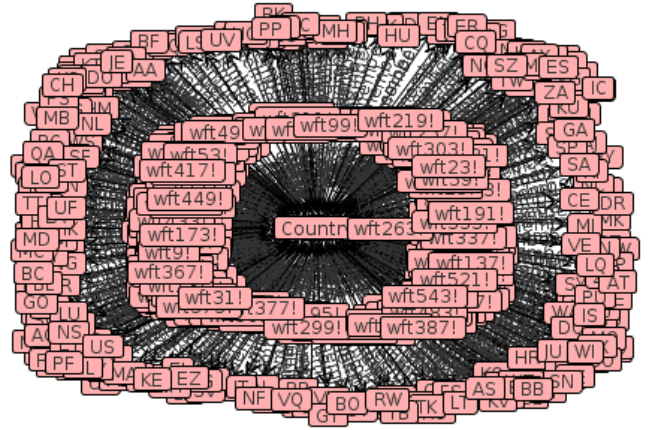


Figure 12: The KB shown in Figure 11 filtered by QBE to show only places which are countries.

wft6, the wft nodes just clutter the display. In this case, the wft node with its two outgoing arcs can be converted into a single arc labeled by the function symbol. This is referred to as “collapsing” the graph. The resulting collapsed graph is semantically equivalent to the uncollapsed version, it is only an alternate representation of the knowledge in the KB. Figure 13 is a collapsed version the graph shown in Figure 6.

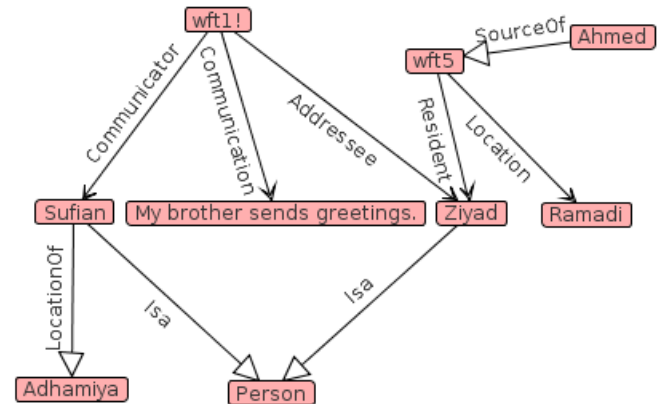


Figure 13: A collapsed version of the graph shown in Figure 6 in which binary relations are redrawn without their wft nodes.

Because the graph representation is backed by frames with ordered slots, the GUI can automatically determine which parts of the graph can be collapsed, the direction of the new arcs, and their labels. Any frame with only two slots can be collapsed. The arc is drawn from the filler of the slot containing the first argument of the function, to the filler of the slot containing the second argument of the function, and the label of the new arc is the function symbol associated with the frame. The arrow head used on the new arcs is a different style from that used on uncollapsed arcs as a visual reminder to the user that it is a collapsed arc.

Notice that, in contrast to some other systems, for exam-

ple those assumed by [Bodenreider, 2002], not all SNePS 3 `wft` nodes can be collapsed in this way. This is due to the increased expressibility of SNePS 3: the ability to represent n -ary relations for $n > 2$, and the ability to express propositions about propositions.

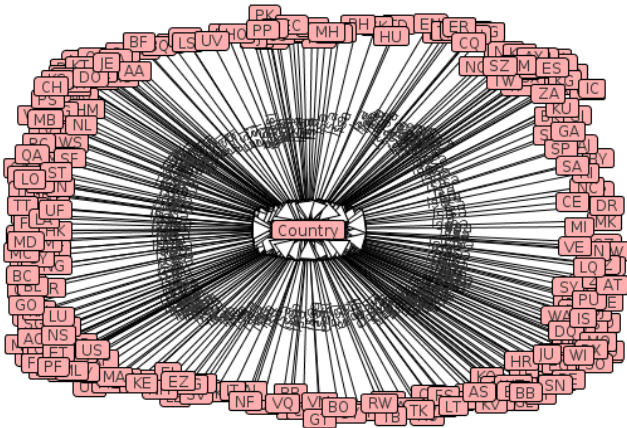


Figure 14: A collapsed version of the graph shown in Figure 12 to show places which are countries with binary relations collapsed.

5 Evaluation

ontology editors [Stanford Center for Biomedical Informatics Research, 2011; Harris, 2011; Revelytix, Inc., 2011] and found that all of them provide the same basic functionality in mostly similar ways, so we will discuss this in the context of Protégé, which appears to be the most popular.

The methods of interaction with an ontology in Protégé are somewhat similar to what we have discussed here. The addition of classes and instances to an ontology occurs outside of any graph interface, and the graphs are used for visualization purposes only. The ontology editors surveyed also have basic reasoning abilities and the ability to present a query to the reasoner. Querying a graph by performing reasoning is something which we wish to incorporate into a future version of the SNePS 3 interface. Currently QBE only filters the graph itself (that is, the current contents of the KB). It would be useful for the system to, at least optionally, perform inference at query time. When doing this, we will want QBE to become more expressive as well - allowing, minimally, for conjunctive queries. As discussed earlier, we also would like a graphical interface for filling a slot with another frame.

SNePS 3 and its user interface can be (and have been [Prentice and Shapiro, 2011]) used for interaction with ontology structures, providing for the most part a superset of the visualization and interaction features provided by ontology editors.

user friendly way. The combination of frame and graph based techniques in QBE and binary relation collapsing respectively has resulted in a highly usable method for interacting with KBs. It is now possible to view, modify, and understand a large KB without the need to sift through complex text outputs or graphs which are too large to be viewed easily. We believe the techniques given here are ideal for visualizing and interacting with the data associated with a KR system.

7 Acknowledgments

We would like to thank the SNePS Implementation Group for their consistent feedback on the user interface. This work has been supported in part by a Multidisciplinary University Research Initiative (MURI) grant (Number W911NF-09-1-0392) for "Unified Research on Network-based Hard/Soft Information Fusion", issued by the US Army Research Office (ARO) under the program management of Dr. John Lavery.

References

- [Bodenreider, 2002] Olivier Bodenreider. Experiences in visualizing and navigating biomedical ontologies and knowledge bases. In *Proceedings of the ISMB'2002 SIG meeting: Bio-ontologies*, pages 29–32, 2002.
- [Fillmore, 1968] Charles J. Fillmore. The case for case. In E. Bach and R. T. Harms, editors, *Universals in Linguistic Theory*, pages 1–88. Holt, Rinehart and Winston, New York, 1968.
- [Fillmore, 1976] Charles J. Fillmore. Frame semantics and the nature of language. In *In Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech*, volume 280, pages 20–32, 1976.
- [Harris, 2011] Nomi Harris. OBO-Edit. <http://oboedit.org/>, 2011.
- [ISO/IEC, 2007] ISO/IEC. *Information technology — Common Logic (CL): a framework for a family of logic-based languages, ISO/IEC 24707:2007(E)*. ISO/IEC, Switzerland, First edition, October 2007. available from <http://standards.iso/itft/license.html>.
- [National Geospatial-Intelligence Agency, 2011] National Geospatial-Intelligence Agency. NGA GEOnet names server (GNS). <http://earth-info.nga.mil/gns/html/>, 2011.
- [Prentice and Shapiro, 2011] Michael Prentice and Stuart C. Shapiro. Using propositional graphs for soft information fusion. In *Proceedings of the 14th International Conference on Information Fusion*, 2011.
- [Prentice et al., 2010] Michael Prentice, Michael Kandefer, and Stuart C. Shapiro. Tractor: A framework for soft information fusion. In *Proceedings of the 13th International Conference on Information Fusion*, 2010.
- [Revelytix, Inc., 2011] Revelytix, Inc. Knoodl. <http://www.knoodl.com/>, 2011.
- [Ruppenhofer et al., 2006] J. Ruppenhofer, M. Ellsworth, M. R. L. Petruck, C. R. Johnson, and J. Scheczyk. FrameNet II: Extended theory and practice. Unpublished manuscript, 2006.
- [Shapiro et al., 2007] Stuart C. Shapiro, William J. Rapa-
port, Michael Kandefer, Frances L. Johnson, and Albert Goldfain. Metacognition in SNePS. *AI Magazine*, 28:17–31, Spring 2007.
- [Shapiro, 1993] Stuart C. Shapiro. Belief spaces as sets of propositions. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 5(2&3):225–235, April–September 1993.
- [Shapiro, 2000] Stuart C. Shapiro. An introduction to SNePS 3. In Bernhard Ganter and Guy W. Mineau, editors, *Conceptual Structures: Logical, Linguistic, and Computational Issues, Lecture Notes in Artificial Intelligence 1867*, pages 510–524. Springer-Verlag, Berlin, 2000.
- [Shapiro, 2004] Stuart C. Shapiro. A logic of arbitrary and indefinite objects. In D. Dubois, C. Welty, and M. Williams, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, pages 565–575, Menlo Park, CA, 2004. AAAI Press.
- [Shapiro, 2010] Stuart C. Shapiro. Set-oriented logical connectives: Syntax and semantics. In Fangzhen Lin, Ulrike Sattler, and Mirosław Truszczyński, editors, *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR2010)*, pages 593–595, Menlo Park, CA, 2010. AAAI Press.
- [Stanford Center for Biomedical Informatics Research, 2011] Stanford Center for Biomedical Informatics Research. The Protégé ontology editor and knowledge acquisition system. <http://protege.stanford.edu/>, 2011.
- [The JUNG Framework Development Team, 2010] The JUNG Framework Development Team. JUNG - Java universal network/graph framework. <http://jung.sourceforge.net/>, 2010.
- [Zloof, 1975] Mosh M. Zloof. Query by example. *AFIPS*, 44, 1975.