ELSEVIER

Artificial Intelligence 112 (1999) 105–146

Artificial Intelligence

www.elsevier.com/locate/artint

Computing with default logic ☆

Paweł Cholewiński [a], Victor W. Marek [b,*], Artur Mikitiuk [c], Mirosław Truszczyński [b]

[a] *Hynomics Corporation, 10632 NE 37th Circle, Bldg. 23, Kirkland, WA 98033-7921, USA*
[b] *Computer Science Department, University of Kentucky, Lexington, KY 40506-0046, USA*
[c] *Department of Computer Technology, Fort Valley State University, 1005 State University Drive, Fort Valley, GA 31030-4313, USA*

Received 2 August 1998

Abstract

Default logic was proposed by Reiter as a knowledge representation tool. In this paper, we present our work on the Default Reasoning System, DeReS, the first comprehensive and optimized implementation of default logic. While knowledge representation remains the main application area for default logic, as a source of large-scale problems needed for experimentation and as a source of intuitions needed for a systematic methodology of encoding problems as default theories we use here the domain of combinatorial problems.

To experimentally study the performance of DeReS we developed a benchmarking system, the TheoryBase. The TheoryBase is designed to support experimental investigations of nonmonotonic reasoning systems based on the language of default logic or logic programming. It allows the user to create parameterized collections of default theories having similar properties and growing sizes and, consequently, to study the asymptotic performance of nonmonotonic systems under investigation. Each theory generated by the TheoryBase has a unique identifier, which allows for concise descriptions of test cases used in experiments and, thus, facilitates comparative studies. We describe the TheoryBase in this paper and report on our experimental studies of DeReS performance based on test cases generated by the TheoryBase.

Keywords: Knowledge representation; Default logic; Nonmonotonic reasoning; Automated reasoning; Constraint satisfaction; Experimental studies; Benchmarking

☆ This paper is a full version of the material presented in two extended abstracts: [17] and [18].

* Corresponding author. Email: marek@cs.engr.uky.edu.

0004-3702/99/$ – see front matter
PII: S0004-3702(99)00053-3

1. Introduction and motivation

In this paper we describe an automated reasoning system, DeReS, based on default logic. We discuss the problem of testing and experimenting with nonmonotonic reasoning. We describe a system, called the TheoryBase, that generates families of default theories for use in experimental studies. We describe results of experiments with DeReS that used as test cases default theories generated by the TheoryBase.

The area of nonmonotonic logics originated in the late 1970s [36,37,49,50] in an effort to build effective knowledge representation formalisms. Since then, solid theoretical foundations of nonmonotonic logics have been established. The efforts of the past two decades culminated in several research monographs [5,11,23,40] describing major nonmonotonic systems: default logic, logic programming with negation as failure, autoepistemic logic and circumscription.

In this paper we focus on default logic—a knowledge representation formalism introduced by Reiter [50] to capture reasoning based on incomplete information. The original motivation of Reiter was to use defaults to derive new information under the assumption of "normality" or "typicality" of a situation. Defaults are inference rules with two types of premises: *prerequisites* and *justifications*. Prerequisites are treated similarly as premises of standard inference rules—they have to have a *proof* in order to allow for the application of a default. Justifications specify the notion of a context-dependent normality under which the default can be applied. To formally describe a semantics for default theories, Reiter introduced the notion of an *extension*. Extensions are theories that model the agent's possible belief sets.

Default logic of Reiter has been widely studied for its potential as a knowledge representation mechanism. Reiter and his collaborators studied default logic as a way to model and investigate the Closed World Assumption [49], inheritance networks with exceptions [22], and situations with conflicting default assumptions [48]. Formalizations of the frame problem and reasoning about action in default logic were extensively studied in [24,27,30,50]. Applications of default logic to diagnosis are discussed in [47,51]. Default logic provides also a semantics for normal logic programs with negation. In [38] we described an encoding of logic programs as default theories, under which there is a straightforward one-to-one correspondence between stable models of a program and extensions of its default interpretation (this application of default logic was independently discovered in [6]).

It is important to notice that, although default logic is a declarative formalism, it is quite different from Horn clause style logic programming. Specifically, extensions of default theories are subsets of the set of formulas, not the elements of that set. For this reason, extensions of default theories correspond to branches of a search tree, rather than to individual nodes of an SLD-tree, which is the case for Horn programs. This "second order" flavor of default logic makes it especially useful in representing problems in which solutions are *subsets* (rather than elements) of some domain. We illustrate the advantage of this property of default logic later in the paper.

It was expected that default logic (and other nonmonotonic systems, too) would have better computational properties than classical logics. Computational complexity results obtained in recent years were discouraging. Decision problems associated with nonmonotonic

reasoning, even when restricted to the propositional case, are computationally complex. For example, in the case of logic programming with the stable model semantics they are NP-complete or co-NP-complete [39]. In the case of default logic, they are Σ_2^{P}-complete or Π_2^{P}-complete [29,57]. We will discuss these results in Section 2.

However, the complexity results do not disqualify nonmonotonic logics as a practical computational knowledge representation mechanism. The results of [14,25] show that higher computational complexity of nonmonotonic logics may be offset by more concise encodings of application problems than those possible with propositional logic. It seems that the only way to establish whether default logic can serve as a computationally practical knowledge representation system is through implementations and systematic experimentations. Recent dramatic improvements in performance of satisfiability algorithms [12,19, 53,54] demonstrate the value of experimental studies.

The progress in understanding default logic resulted in several algorithms for computing extensions and led to first implementation projects [2,3,7,32,40,43]. In the last few years, implementing nonmonotonic reasoning systems became one of the most actively pursued directions in the area of nonmonotonic logics. Several working systems were presented recently at the Fourth Conference on Logic Programming and Nonmonotonic Reasoning [20].

Our goal in this research was to study experimentally properties and performance of default logic as an automated reasoning system. We describe here the Default Reasoning System, DeReS, developed and studied over several years at the University of Kentucky. DeReS supports basic automated reasoning tasks for default logic and for logic programming with the stable model semantics [26]. Our current version of DeReS uses *relaxed stratification* [15,16] as a primary search-space pruning mechanism. [1] A relaxed stratification of a default theory allows us to use a *divide-and-conquer* approach when computing extensions. An original default theory is partitioned into several smaller subtheories, called *strata*. The extensions of the original theory are then reconstructed from the extensions of its strata. The notion of a relaxed stratification considered here is a generalization of the concept of a stratification of a logic program, as introduced in [1]. In particular, a theory (logic program) stratified in our sense may possess no extension (stable model) or, if it does, not necessarily a unique one. In the paper we show that applying relaxed stratification leads to substantial speedups, especially when the strata are small. Relaxed stratification is discussed in Section 3.2.

In the paper we also study the effects of different propositional theorem provers on the efficiency of DeReS. We observe that full theorem provers, which check global consistency when deciding whether a theory proves a formula, result in performing prohibitive amount of redundant computations. A weaker notion of a *local prover*, sound but not complete, can also be used to correctly implement default reasoning and results in significant improvements in time performance. For consistent theories a local prover is complete, and we use this feature of a local prover to limit the size of theories that need to be consulted for provability and satisfiability. Use of a local prover requires modifications in algorithms processing default theories. The details are discussed in Section 3.3.

[1] A similar idea was proposed for logic programs, under the name of "splitting", by Lifschitz and Turner, see [35].

Our results show that there are classes of theories that DeReS can handle very efficiently. However, if relaxed stratification does not yield a partition of an input theory into small strata, the efficiency of DeReS may be poor. In this context, it is interesting to relate our work to that of Niemelä and Simons [45]. Their system, *s-models*, is currently the best implementation of the stable model semantics for logic programs. It is based on the ideas first proposed in [55] that have some common features with the Davis–Putnam approach to satisfiability testing. Namely, *s-models* makes a decision about the membership of an atom in a stable model, propagates the effects of this decision through the program, thus decreasing its size and, then selects the next atom to deal with. As soon as *s-models* establishes that there is no stable model consistent with the decisions made so far, it backtracks. Thus, DeReS and *s-models* attack different aspects of the same problem. While our research focused on techniques to exploit relaxed stratification to reduce the problem to smaller ones (divide-and-conquer), Niemelä and Simons developed techniques to deal with individual strata (*s-models* does not exploit stratification at all). It seems that the next-generation implementations of nonmonotonic systems, in order to be effective in a large range of different applications, must combine techniques developed in both projects.

Systematic implementation and experimentation effort is necessary to provide us with better insights into the computational properties of nonmonotonic logics. Despite importance of experimental studies to the area of nonmonotonic logics, there has been little work reported in the literature. While several algorithms were published and some implementations described [4,8,9,20,42,45], the results are far from conclusive. This state of affairs can be attributed to the lack of systematic experimentation with implemented systems. One possible reason is the absence of commonly accepted benchmarking systems that could generate rich classes of meaningful test data—logic programs and default theories.

Resorting to randomly generated programs and theories, a solution often used in other areas such as graph algorithms or satisfiability testing, is not a viable approach. First, it is difficult to argue that randomly generated data have any correlation with cases that are encountered in practical situations. Second, only a very careful selection of parameters makes randomly generated instances difficult to solve and, hence, useful for benchmarking purposes [12]. Third, no model of a random logic program or random default theory has been proposed yet.

In this paper we describe our approach to the problem of generating logic programs and default theories to test nonmonotonic reasoning systems. Namely, we develop encodings of graph problems as logic programs and default theories. Our approach builds on the work of Knuth [33] in which he presented a graph generating system called The Stanford GraphBase. We apply our encodings of graph problems to graphs generated by The Stanford GraphBase, thus producing a rich variety of programs and theories for testing. We call the resulting system the TheoryBase.

The Stanford GraphBase allows the user to generate *parameterized* families of graphs of similar structure and properties, and of sizes controlled by a numeric parameter. This feature is inherited by the TheoryBase. Thus, the TheoryBase can generate *families* of default theories and logic programs of similar structure and properties, and of growing sizes, which supports studies of scalability of reasoning algorithms.

Each graph generated by The Stanford GraphBase has a unique identifier. This feature greatly facilitates the use of The Stanford GraphBase as a benchmarking system. We

extended the concept of the GraphBase identifier to the case of default theories and logic programs generated by the TheoryBase.

In the paper we demonstrate the usefulness of the TheoryBase in experimental studies of automated reasoning systems by using the TheoryBase generated default theories in our studies of the performance of DeReS.

The paper is organized as follows. In the next section we provide the reader with the formal definition of default logic and its simplified version, logic programming with the stable semantics. We discuss the complexity results for default logic. In Section 3, we describe DeReS, its main components and reasoning algorithms. Section 4 contains descriptions of default encodings of graph problems that are used by the TheoryBase. The TheoryBase itself is described in Section 5. Results of experimenting with DeReS are presented in Section 6. The last section contains conclusions.

2. Default logic—technical introduction

The language of default logic is an extension of the language of first-order logic by new structures called *defaults*. In this paper, we concentrate on the case when the underlying first-order language is propositional. A more general case, of the predicate language without quantifiers and function symbols follows immediately from our presentation.

Let $\mathcal{L}$ be a fixed propositional language over a set of atoms At. A *default* is an expression d of the form

$$\frac{\alpha : \Gamma}{\beta}$$

where α and β are formulas from $\mathcal{L}$, and Γ is a **finite** set of formulas from $\mathcal{L}$. The formula α is called the *prerequisite*, formulas in Γ—the *justifications*, and β—the *consequent* of d. The prerequisite, the set of justifications and the consequent of a default d are denoted by $p(d)$, $j(d)$ and $c(d)$, respectively. If $p(d)$ is a tautology, d is called *prerequisite-free* ($p(d)$ is then usually omitted from the notation of d). This terminology is naturally extended to a set of defaults D. When $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$, we will write d as

$$\frac{\alpha\colon\ \gamma_1, \ldots, \gamma_m}{\beta}.$$

By a *default theory* we mean a pair $\Delta = (D, W)$, where D is a set of defaults and W is a set of formulas from $\mathcal{L}$. The set W is called the *objective part* of (D, W). A default theory $\Delta = (D, W)$ is called *finite* if both D and W are *finite*.

Let T be a set of formulas from $\mathcal{L}$. A default rule $\frac{\alpha:\Gamma}{\beta}$, is T*-applicable* if every formula $\gamma \in \Gamma$ is consistent with T. For a set of defaults D, by D_T we denote the set of defaults from D that are T-applicable.

For a set of defaults D, define

$$Mon(D) = \left\{ \frac{p(d)}{c(d)}\colon\ d \in D \right\}.$$

Thus, $Mon(D)$ consists of standard inference rules obtained from defaults in D by dropping the justification part. By $Cn^{D,T}(W)$ (Reiter used the notation $\Gamma(T)$) we denote

the closure of W under propositional consequence and under all the rules in $Mon(D_T)$. A theory T is called an *extension*[2] of (D, W) if

$$Cn^{D,T}(W) = T.$$

Let T be a theory. A default d is *generating* for T if d is T-applicable and $p(d) \in T$. The set of all defaults in D generating for T is denoted by $GD(D, T)$. The following proposition gathers some well-known properties of default logic [40].

Proposition 2.1. *Let (D, W) be a default theory.*

(1) *If T is an extension of (D, W) then $T = Cn(W \cup c(GD(D, T)))$.*
(2) *If all defaults in D are prerequisite-free then T is an extension of (D, W) if and only if $T = Cn(W \cup c(GD(D, T)))$.*

Part (1) of this proposition is the basis for all algorithms that compute extensions.

A *logic programming clause* (or, simply, a *clause*) is an expression of the form

$$p \leftarrow q_1, \ldots, q_m, \mathbf{not}(r_1), \ldots, \mathbf{not}(r_n)$$

where $p, q_1, \ldots, q_m, r_1, \ldots, r_n$ are atoms. A *logic program* is a finite set of such clauses. When $n = 0$, the clause is called a *Horn* clause. A program P consisting of Horn clauses has a *least model*, that is, a least set $M \subseteq At$ such that for every clause $C \in P$, $C = p \leftarrow q_1, \ldots, q_m$, whenever $q_1, \ldots, q_m \in M$ then also $p \in M$.

Given a set of atoms $M \subseteq At$ and a logic program P, the *reduct* P^M of P with respect to M consists of Horn clauses $p \leftarrow q_1, \ldots, q_m$ such that for some $r_1, \ldots, r_n \notin M$, $p \leftarrow q_1, \ldots, q_m, \mathbf{not}(r_1), \ldots, \mathbf{not}(r_n) \in P$. A *stable model* of a logic program P is a set M of atoms such that M coincides with the least model of P^M. Stable models were introduced by Gelfond and Lifschitz [26].

Logic programs can be represented by default theories. Specifically, a clause

$$C = p \leftarrow q_1, \ldots, q_m, \mathbf{not}(r_1), \ldots, \mathbf{not}(r_n)$$

can be represented as the default

$$dl(C) = \frac{q_1 \wedge \cdots \wedge q_m : \neg r_1, \ldots, \neg r_n}{p}.$$

For this representation we have the following result [6,38].

Proposition 2.2. *Let P be a logic program. Then M is a stable model of P if and only if $Cn(M)$ is an extension of $(\{dl(C)\colon C \in P\}, \emptyset)$.*

Proposition 2.2 tells us that if we are able to compute extensions of default theories then, in particular, we are able to compute stable models of logic programs.

There is an important difference between computing stable models and computing default extensions. Namely, when computing stable models, procedures testing full propositional provability are not needed. DeReS takes advantage of this fact.

[2] Our definition is different from but equivalent to the original definition by Reiter [50].

Reasoning tasks associated with default logic are listed below. In the descriptions we assume that a finite default theory (D, W) and a formula φ form the input.

Existence—decide whether (D, W) has an extension.

In-Some—decide whether (D, W) has an extension containing φ.

In-All—decide whether φ belongs to all extensions of (D, W).

The following result due to Gottlob [29] and Stillman [57] determines the complexity of these problems.

Proposition 2.3. *The problems* **Existence** *and* **In-Some** *are* Σ_2^{P}*-complete. The problem* **In-All** *is* Π_2^{P}*-complete.*

The same reasoning tasks can be formulated for the domain of logic programs and the stable model semantics. In this setting the complexity of the reasoning problems goes down. This is due to the fact that deciding whether an atom follows from a set of atoms is easier (polynomial) than the task of deciding whether a formula follows from a set of formulas (co-NP-complete). Specifically, for logic programs we have the following result [39].

Proposition 2.4. *In the case of logic programs and atoms, the problems* **Existence**, *and* **In-Some** *are* NP*-complete. The problem* **In-All** *is* co-NP*-complete.*

A default theory (D, W) is *disjunction-free* if all formulas in W, and all prerequisites, justifications and consequents of defaults in D are conjunctions of literals. One can show that the same complexity bounds as those given in Proposition 2.4 hold for the class of disjunction-free default theories [34]. Several default theories studied below are disjunction-free.

3. Automated reasoning with default logic

In this section we describe the Default Reasoning System DeReS developed at the University of Kentucky. We provide a general overview of DeReS, describe its main components and the key reasoning algorithms.

3.1. Overview

DeReS is a software package implementing nonmonotonic reasoning and running under all major versions of Unix, including Linux. The focus of DeReS is on automated reasoning with default logic and with logic programming with the stable model semantics. [3]

[3] A detailed information about DeReS and how to use it, as well as the system itself can be obtained from http://www.cs.engr.uky.edu/deres/.

DeReS computes extensions for finite propositional default theories.[4] Given a default theory, DeReS can determine existence of extensions and can compute one of the extensions or all of the extensions. There are no syntactic restrictions on input default theories and formulas.

The user communicates with DeReS via its shell. The DeReS shell provides the user with access to commands specific to DeReS, as well as to system commands. In particular, it reads user queries, initiates appropriate reasoning procedures, and outputs results of the reasoning process. It also outputs statistics such as the amount of the CPU time used to solve a query, the number of calls to the propositional provability procedure and the number of candidates for extensions that were tested. Three main modules of DeReS are:

Default Reasoning Module—a library of routines for reasoning with a given default theory,

Prover Module—a collection of propositional theorem provers that can be called by the Default Reasoning Module,

User Interface—a collection of shell commands for processing input theories and programs, and displaying the progress of the computation and the results.

3.2. Default reasoning module

The key reasoning algorithm of DeReS is based on the observation that every extension of a default theory (D, W) is of the form $Cn(W \cup c(U))$ for some set of defaults $U \subseteq D$. This representation may not be unique. That is, an extension may be generated by W and consequents of different subsets of D. However, every extension T has a unique largest subset of defaults that generates it. This is the set of its generating defaults $GD(D, T)$ (see Proposition 2.1). This observation implies a method, called *generate-and-check*, to construct one (or all) extensions. The idea is to construct all subsets of D and, for each of them, test whether it is the set of generating defaults of an extension.

To accomplish this latter task, DeReS uses a procedure **Is_Extension**(D, W, U). Given a finite default theory (D, W) and a set $U \subseteq D$, it returns value **true** if U is the set of generating defaults of an extension for (D, W), and returns value **false**, otherwise. One such procedure is described in [40]. It is presented here in Fig. 1.

To generate all subsets of D, DeReS generates and searches a full binary tree whose nodes are labeled by subsets of D. This tree is constructed as follows. Let $D = \{d_1, d_2, \ldots, d_n\}$. The root of the tree is labeled by the empty subset of D. If a node a, at depth k in the tree, is labeled by set $U \subseteq D$, then the left child of a is labeled by $U \cup \{d_{k+1}\}$ and the right child of a is labeled by U, again. It is clear that every subset of D appears as a label on at least one node. In the case when $n = 3$, the corresponding binary tree is shown in Fig. 2.

DeReS considers the nodes of the tree according to the depth-first search order. To avoid considering the same subset several times (if it appears as the label on more than one node

[4] To be precise, for each extension T, DeReS computes its *base*, that is, a finite set of formulas B such that $T = Cn(B)$.

Is_Extension(D, W, U)
Input: Finite sets of defaults D and U such that $U \subseteq D$, and a finite set of formulas W;
Output: **true**—if U is the set of generating defaults for an extension of (D, W) and
false—otherwise;

$R := \{d \in D\colon\ W \cup c(U) \not\vdash \neg\beta,\ \text{for } \beta \in j(d)\}$;
if not $(U \subseteq R)$ **then return(false) else**
$B := W$;
$X := \emptyset$;
repeat
$AR := \{d \in R \setminus X\colon\ B \vdash p(d)\}$;
$B := B \cup c(AR)$;
$X := X \cup AR$;
if not $(X \subseteq U)$ **then return(false)**
until $AR = \emptyset$;
if $X = U$ **then return(true) else return(false)**;

Fig. 1. Checking if a given set of defaults is the set of generating defaults for an extension.

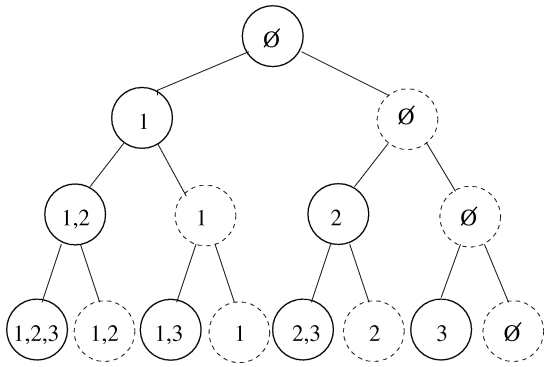


Fig. 2. Generating all subsets of $X = \{1, 2, 3\}$.

of the tree), a set of defaults is checked by the **Is_Extension** procedure only when it is encountered for the first time as the label on a node in the tree. In Fig. 2, the nodes where **Is_Extension** is actually invoked are shown in solid lines.

The sets of generating defaults of extensions form an antichain. This observation yields a method to prune the search space. When the set of defaults represented by a node in the search space is found to be generating for an extension, DeReS prunes all descendants of this node in the search tree. The resulting algorithm to compute all extensions, referred to as **All_Extensions**, is presented in Fig. 3. The variable *backtrack* is set to **true** whenever the currently considered node in the search space is a leaf or represents the set of generating defaults of an extension, causing the algorithm to backtrack.

The algorithm **All_Extensions** is capable of computing extensions for arbitrary finite propositional default theories. However, due to the high computational complexity of default reasoning, computation time can be very long. In many cases this problem can

```
All_Extensions(D, W)
Input: A finite default theory (D, W) and D = {d_1, d_2, ..., d_n};
Output: The list of all extensions of (D, W);

U := ∅;
Build_Extensions(D, W, U, 0);

procedure Build_Extensions(D, W, U, k);
backtrack := (k = |D|);
if k = 0 or d_k ∈ U then
        if Is_Extension(D, W, U) then
                write(W ∪ c(U));
                backtrack := true;
if not backtrack then
        Build_Extensions(D, W, U ∪ {d_{k+1}}, k + 1);
        Build_Extensions(D, W, U, k + 1);
```

Fig. 3. Search for all extensions of (D, W).

be avoided by splitting the input default theory into several strata (clusters) of defaults and dealing with one stratum at a time. This technique, we will refer to it as *relaxed stratification*, was developed in [15,16]. It is the main search space pruning technique used by DeReS.

Relaxed stratification applies to default theories that do not have justification-free defaults and in which formulas in W do not have common propositional variables with the consequents of the defaults. In this method, we first find a finest possible relaxed stratification of D, that is, a partition $\mathcal{D} = \{D_1, \ldots, D_m\}$ such that propositional variables appearing in defaults from D_i do not appear in the consequents of defaults from D_j, for $i < j$, and such that no set D_i can be further partitioned preserving the constraint on variable occurrence. It can be shown that such a relaxed stratification exists. We search for extensions for a single stratum (D_i, W_i) $(W_1 = W)$ using the same approach as in the algorithm **All_Extensions**. However, when an extension, say $Cn(W_i \cup c(U))$, is found we report it only if D_i is the last stratum of the default theory (that is, when $i = m$). Otherwise, we add the formulas from $c(U)$ to W_i to form W_{i+1} $(W_{i+1} := W_i \cup c(U))$, and start the search for extensions of (D_{i+1}, W_{i+1}). If the stratification is fine-grained, then in each step we deal with small sets of defaults and computational savings can be expected. The detailed description of this method can be found in [15,16].

We refer to the algorithm based on the idea described above as **All_Extensions_Stratified**. The pseudocode is given in Fig. 4.

3.3. Prover module

Prover Module of DeReS is used as an oracle by all reasoning procedures. Currently, DeReS is equipped with a prover that implements the propositional tableaux method. However, any other technique based, for instance, on the resolution inference rule or on satisfiability testing procedures could be used in its place.

All_Extensions_Stratified$(\mathcal{D}, n, W)$
Input: A consistent finite propositional theory W and a relaxed stratification $\mathcal{D} = \{D_1, \ldots, D_n\}$ of $(\bigcup_{l=1}^{n} D_l, W)$;
Output: The list of all extensions of $(\bigcup_{l=1}^{n} D_l, W)$;

$U := \emptyset$;
Stratified_Build_Extensions$(\mathcal{D}, n, W, U, 1, 0)$;

procedure Stratified_Build_Extensions$(\mathcal{D}, n, W, U, l, k)$;
(* we assume that $D_l = \{d_1, \ldots, d_m\}$ *)
backtrack := $(k = |D_l|)$;
if $k = 0$ **or** $d_k \in U$ **then**
  if Is_Extension(D_l, W, U) **then**
    if $(l = n)$ **then** (* last stratum extension *)
      write$(W \cup c(U))$;
      backtrack := **true**;
    else
      $W' := W \cup c(U)$; $U' := \emptyset$;
      Stratified_Build_Extensions$(\mathcal{D}, n, W', U', l + 1, 0)$;
if not *backtrack* **then**
  Stratified_Build_Extensions$(\mathcal{D}, n, W, U \cup \{d_{k+1}\}, l, k + 1)$;
  Stratified_Build_Extensions$(\mathcal{D}, n, W, U, l, k + 1)$;

Fig. 4. Search for all extensions of a stratified theory $(\bigcup_{l=1}^{n} D_l, W)$.

Using a sound and complete prover allows DeReS to handle arbitrary default theories. However, it carries a heavy computational cost due to the inefficiency of such provers. Analyzing the performance of sound and complete provers, one can see that substantial amount of time spent to decide whether a theory T proves a formula φ is actually spent to decide consistency of T. Next, when searching for a proof of φ from T, even those parts of T that are irrelevant to φ may be considered by the prover.

Based on these observations, we designed and implemented a method referred to as a *local prover*. This provability testing procedure does not perform consistency checks and, consequently, is sound but not complete. Moreover, the local prover takes into account only the part of T that is relevant to proving φ. We then modified reasoning algorithms in DeReS so that a full prover can be replaced with a local prover without affecting the correctness of DeReS. As expected, we observed substantial computational gains. We will now describe in detail the concept of a local prover.

Let $\mathcal{L}$ be any propositional language. For a formula $\varphi \in \mathcal{L}$, by $Var(\varphi)$ we denote the set of atoms occurring in φ. Similarly, for a theory T, we define $Var(T)$ as the set of all atoms occurring in the formulas from T.

Consider a theory $T \subseteq \mathcal{L}$ and a formula $\varphi \in \mathcal{L}$. The *φ-pertinent fragment* of T, T_φ, is defined recursively as follows:

$$T_\varphi^0 = \{\psi \in T\colon\ Var(\psi) \cap Var(\varphi) \neq \emptyset\},$$

$$T_\varphi^{n+1} = \{\psi \in T\colon\ Var(\psi) \cap Var(T_\varphi^n) \neq \emptyset\},$$

for $n \geqslant 0$, and

$$T_\varphi = \bigcup_{n \geqslant 0} T_\varphi^n.$$

Next, we will introduce the concept of a local provability. The main idea is to capture the expression "*the information in T, pertinent to φ, entails φ*". Thus, φ should not be *locally* provable just because T contains some inconsistent data.

Definition 3.1. A theory T *locally proves* a formula φ (denoted $T \vdash_{loc} \varphi$) if $T_\varphi \vdash \varphi$.

Local provability has the following useful properties.

Proposition 3.1. *Let $T \subseteq \mathcal{L}$ be a theory and let $\varphi \in \mathcal{L}$ be a formula.*

(1) *If $T \vdash_{loc} \varphi$ then $T \vdash \varphi$.*
(2) *If $T \vdash_{loc} \varphi$ and T is consistent then $T \cup \{\varphi\}$ is consistent.*
(3) *If T is consistent then $T \vdash \varphi$ if and only if $T \vdash_{loc} \varphi$.*
(4) *$T \vdash \varphi$ if and only if either T is inconsistent or $T \vdash_{loc} \varphi$.*

All standard propositional routines can be easily modified so that they implement the concept of a local provability. For instance, in order to decide whether $T \vdash_{loc} \varphi$, our tableaux method is modified so that

(1) The root of the tableau is labeled with $\neg\varphi$, and
(2) A branch is never expanded by formulas that have no variables in common with those already appearing on the branch.

In this way, the prover remains restricted to the theory T_φ. This component is often much smaller in size than T.

Replacing a full prover by a local prover may lead, in general, to incorrect results.

Example 3.1. Let (D, W) be a default theory with $W = \{p, \neg p\}$ and $D = \{d_0\}$, where $d_0 = \frac{p:q}{q}$. This theory has a unique extension, $\mathcal{L}$, that is generated by the empty set of generating defaults. Since $W \nvdash_{loc} \neg q$, using a local prover instead of a sound and complete prover will classify d_0 as applicable with respect to the context W. Consequently, the same unique extension $\mathcal{L}$ will be found but the set of generating defaults will be determined incorrectly (d_0 will be returned as generating).

Example 3.2. Let (D, W) be a default theory with $W = \{p\}$ and $D = \{d_0, d_1, d_2\}$, where

$$d_0 = \frac{p:}{\neg p}, \qquad d_1 = \frac{:x}{x}, \qquad d_2 = \frac{:\neg x}{\neg x}.$$

Suppose that we search for extensions by examining subsets of D. For each $U \subseteq D$ we have to check whether $Cn(W \cup c(U))$ is an extension of (D, W). This theory has only one extension $Cn(W \cup c(\{d_0\})) = \mathcal{L}$ with $U = \{d_0\}$ returned by DeReS as the set of generating defaults. Substituting $\vdash$ by $\vdash_{loc}$ in the algorithm **Is_Extension** will result in the algorithm **All_Extensions** returning two extensions generated, respectively, by the sets of defaults $U_1 = \{d_0, d_1\}$ and $U_2 = \{d_0, d_2\}$, as $\{p, \neg p\} \nvdash_{loc} x$ and $\{p, \neg p\} \nvdash_{loc} \neg x$. Both

sets generate the only extension of (D, W), $\mathcal{L}$, but none is, in fact, its set of generating defaults.

Example 3.3. Let (D, W) be a default theory with $W = \{p\}$ and $D = \{d_0, d_1\}$, where

$$d_0 = \frac{:q}{\neg p}, \qquad d_1 = \frac{:p}{\neg q}.$$

The theory (D, W) has a single extension $Cn(\{p, \neg q\})$. However, substituting $\vdash$ by $\vdash_{loc}$ in the algorithm **Is_Extension** will return two theories as extensions: $Cn(\{p, \neg q\})$ and $Cn(\{p, \neg p\}) = \mathcal{L}$.

The algorithm **All_Extensions** outputs sets of generating defaults of extensions of the input default theory. Our examples show that when the local prover is used in **Is_Extension**, the algorithm **All_Extensions** may return additional solutions (sets of defaults). Each of these additional solutions generates the theory $\mathcal{L}$, the entire language. This is the only problem caused by the use of the local prover. Consistent extensions of a default theory will be computed correctly and only once.

We will now describe modifications in the algorithm **All_Extensions** to guarantee correctness when the local prover is used in **Is_Extension** instead of the full propositional prover. These modifications exploit the observation that in the case of a *consistent* theory T, there is no difference between provability and local provability from T (Proposition 3.1).

First, we will decide whether W is inconsistent. To this end, we will start with the empty set of formulas. Then, we will add the formulas from W one by one, each time checking whether consistency is preserved. This can be accomplished by means of a local prover. If W is inconsistent, then (D, W) has a unique extension, which is inconsistent. In this case, the set of generating defaults is the set of all justification-free defaults in D.

If W is consistent, we next check whether an inconsistent extension can be generated out of W and the justification-free defaults (defaults with justifications do not matter in the case of inconsistent extensions). This is done by gradually building the closure of W under the justification-free defaults. Again, each time before a rule is applied, it is checked whether consistency will be preserved (by a single call to the local prover). If a contradiction is detected, (D, W) has an inconsistent extension.

Otherwise, all extensions of (D, W), if they exist, are consistent (and so is W). Before we complete the description of the algorithm, let us notice that the procedure **Is_Extension** with the local prover correctly determines whether $U \subseteq D$ is the set of generating defaults of an extension if $W \cup c(U)$ is consistent. Indeed, if in an iteration of the **repeat** loop the consequents of the defaults in X together with W lead to a contradiction, then the set X is not included in U (as $W \cup c(U)$ is consistent). Hence, the procedure will return **false** and terminate. This is correct, as at this point in the algorithm, only consistent theories may be extensions. Otherwise, all theories involved in provability checks are consistent and the local prover works exactly as the full prover.

Notice that in the algorithms **All_Extensions** and **All_Extensions_Stratified** the space of all subsets U of D is searched by starting with $U = \emptyset$ and then, in each step, a single default is either deleted from or added to U. Assume that the current set of defaults U (the current candidate for the set of defaults generating an extension) is such that $W \cup c(U)$

is consistent (this assumption holds at the beginning of the search as, let us recall, W is consistent). If the next set of defaults, say U', to be considered is obtained by deleting a default then, clearly, $W \cup c(U')$ is consistent, too. Hence, the procedure **Is_Extension** with the local prover can be used to determine whether U' is the set of generating defaults of an extension.

If U' is obtained by adding a default, say d, then we first check whether $W \cup c(U) \vdash_{loc} \neg c(d)$. If the answer is positive, the set $W \cup c(U')$ is inconsistent and does not generate an extension (recall that at this point we know that all extensions are consistent). Thus, the recursive call (second line from the bottom in Fig. 3) is omitted (supersets of U' do not generate an extension, either). Otherwise, $W \cup c(U')$ is consistent. Hence, as before, **Is_Extension** with the local prover can be used to decide whether U' is the set of generating defaults of an extension.

A description of the modified algorithm **All_Extensions**, called **All_Extensions_Loc**, is shown in Fig. 5.

Analogous modifications allow us to use the algorithm **Stratified_Build_Extensions** with a local prover instead of a full propositional prover.

Using a local prover significantly improves the performance of DeReS (see Section 6 for a discussion of our experimental results) and requires no restrictions on the syntax of input theories. Another way to improve the performance of DeReS is to impose syntactic constraints on input theories and exploit these restrictions in the design of even more efficient provers. In particular, DeReS uses special processing methods to deal with disjunction-free theories.

Recall that a default theory (D, W) is disjunction-free if all formulas in W, all prerequisites, justifications and consequents of defaults in D are conjunctions of literals. This condition yields a simple but still very useful class of default theories. In particular, every logic program can be encoded by a disjunction-free default theory.

Recall that every extension of a default theory (D, W) is of the form $Cn(W \cup c(U))$, for some $U \subseteq D$. In the case when (D, W) is disjunction-free, each set of the form $W \cup c(U)$ is a collection of conjunctions of literals and, consequently, can be represented as a set of literals, say L. In the algorithm **Is_Extension** the first task is to compute the set R. Consider a justification β of a default in D. The formula β is a conjunction of literals. The negation of β is logically equivalent to a disjunction of literals, say β'. Deciding whether β' is entailed by the set of literals L can be accomplished as follows:

(1) If L is inconsistent (contains a pair of complementary literals), then L entails β';
(2) If β' is a tautology (that is, contains a pair of complementary literals), then L entails β';
(3) Otherwise, L entails β' if and only if L and β' have at least one literal in common.

The only other time when a propositional prover is called by procedure **Is_Extension** is while computing the set of defaults AR. If B is maintained as a set of literals, then deciding whether a prerequisite α is entailed by B can be accomplished as follows:

(1) If B is inconsistent (contains a pair of complementary literals), then B entails α;
(2) If B is consistent and α is inconsistent (that is, contains a pair of complementary literals), then B does not entail α;
(3) Otherwise, B entails α if and only if every literal occurring in α belongs to B.

All_Extensions_Loc(D, W)
Input: A finite default theory (D, W), where $W = \{w_1, \ldots, w_m\}$ and $D = \{d_1, \ldots, d_n\}$;
Output: The list of all extensions of (D, W);

Set JF to be the set of justification-free defaults in D;
for $i := 1$ **to** m **do**
    if $\{w_1, \ldots, w_{i-1}\} \vdash_{loc} \neg w_i$ **then**
        write$(W \cup c(JF))$;
        return;
(* If the execution goes past this point, W is consistent *)
$B := W$;
$AD := \{d \in JF\colon\ B \vdash_{loc} p(d)\}$;
$JF' := JF \setminus AD$;
while $AD \neq \emptyset$ **do**
    $d :=$ any rule in AD;
    if $B \vdash_{loc} \neg c(d)$ **then**
        write$(W \cup c(JF))$;
        return;
    $B := B \cup \{c(d)\}$;
    $AD := (AD \setminus \{d\}) \cup \{r \in JF'\colon\ B \vdash_{loc} p(r)\}$;
    $JF' := JF' \setminus \{r \in JF'\colon\ B \vdash_{loc} p(r)\}$;
(* If the execution goes past this point, extensions of (D, W), if exist, are consistent *)
$U := \emptyset$;
Build_Extensions_Loc$(D, W, U, 0)$;

procedure Build_Extensions_Loc(D, W, U, k);
backtrack $:= (k = |D|)$;
if $k = 0$ **or** $d_k \in U$ **then**
    if Is_Extension(D, W, U) **then**
        write$(W \cup c(U))$;
        backtrack := **true**;
if not *backtrack* **then**
    if $W \cup c(U) \nvdash_{loc} \neg c(d_{k+1})$ **then Build_Extensions_Loc**$(D, W, U \cup \{d_{k+1}\}, k+1)$;
    Build_Extensions_Loc$(D, W, U, k+1)$;

Fig. 5. Search for all extensions of (D, W) using a local prover.

All the provability tests mentioned above can be accomplished by deciding membership of a literal in a set of literals. This method is implemented in DeReS and referred to as the *table lookup method*. It decides each provability of a literal from a set of literals in a constant time.

In Section 6 we present several examples of the performance of provers on concrete default theories, generated using the TheoryBase.

3.4. Using DeReS

To work with DeReS the user invokes the DeReS shell. The shell allows the user to load files with input default theories, display them, and compute, display and record extensions.

Each default theory to be processed by DeReS is identified by a file *filename1.dt*. This file specifies the names of two other files, *filename2.thc* and *filename3.dc*, by including lines

```
w = filename2
d = filename3
```

The file *filename2.thc* consists of formulas (part W of the default theory). The file *filename3.dc* consists of defaults (part D of the default theory).

The performance of DeReS is substantially improved if the input default theory, say represented by the file *filename.dt*, is stratified and if the strata are possibly small. To take advantage of this feature, the user has to construct an additional file, *filename.str* (the same name as the file identifying the default theory, but different suffix). This file is automatically created by the TheoryBase for all default theories that it produces and which admit nontrivial relaxed stratification. The stratification file defines a partition of input defaults into strata. If the stratification file is not found, DeReS assumes trivial stratification into a single cluster.

The syntax of formulas and defaults is rather straightforward. Symbols `&&`, `||`, `!`, `=>` and `<=>` serve as conjunction, disjunction, negation, implication and equivalence, respectively. Defaults are specified by providing the prerequisite, the list of justifications and the consequent. The prerequisite is separated from the justifications by a colon "`:`". The list of justifications is then followed by `->` and by the consequent.

Example 3.4. Let (D, W) be a default theory defined as:

$$D = \left\{ \frac{:a}{a}, \ \frac{b:c}{c}, \ \frac{d \vee a : e}{e}, \ \frac{c \wedge e : \neg a, d \vee a}{f} \right\},$$

$$W = \{b, c \Rightarrow d \vee a, a \wedge c \Rightarrow \neg e\}.$$

This theory was described in Example 2.4 in [50]. In Fig. 6 we show the three input files which represent the theory (D, W) in the DeReS format.

The user runs DeReS by invoking its shell. The shell provides the user with several commands:

(1) `load` *filename*—loads a default theory (D, W) described in the file *filename.dt*;
(2) `status`—shows the name of the current default theory (the theory loaded by the most recent use of the `load` command) and system settings;
(3) `setprover [-f | -l | -a]`—selects a prover mode; options `-f`, `-l`, `-a` select full, local and table lookup provers, respectively; default setting is `-l`;
(4) `quit`—quits DeReS;
(5) `list [num1 [num2]]`—displays default rules of the current input theory from the default number `num1` to the default number `num2`; the default values for `num1` and `num2` are the first and the last default of the current input;
(6) `pds [num1 [num2]]`—displays strata of the current input theory from the stratum number `num1` to the stratum number `num2`; the default values for `num1` and `num2` are the first and the last stratum of the current input;

```
% File re80.dt
% Example 2.4 from R. Reiter "A logic for default reasoning"

w = re80-2.4-formulas
d = re80-2.4-defaults
```

```
% File re80-2.4-formulas.thc

b;
c => d || a;
a && c => !e;
```

```
% File re80-2.4-defaults.dc

  : a -> a;
b : c -> c;
d || a : e -> e;
c && e : !a, d||a -> f ;
```

Fig. 6. DeReS encoding of a default theory (Example 3.4).

(7) `size`—shows the size of the current input theory;
(8) `ext [-c] [-f] [-h] [-s] [-x] [-timeN] [-lastS] [-llenK]` —computes extensions with terminal output; it has several options that specify whether to halt after first extension is found, compute all extensions, count extensions, store extensions in a file, etc.;
(9) `x11ext`—starts DeReS X11 interface; provides a graphical user interface to DeReS.

A typical session consists of invoking the DeReS shell, loading default theories and starting `ext` or `x11ext`.

4. Programming with default logic

Programming with default logic means reducing a given problem to reasoning tasks of default logic such as deciding of the existence of extensions, finding an extension or finding all extensions. Consider a problem whose solutions are subsets of some domain. Reducing the problem to default logic means *constructing* a default theory whose extensions allow the user to determine all solutions to the original problem. Similarly, in the case of decision problems, solving them by means of default logic means *constructing* a default theory that has an extension if and only if the original problem has a solution. Constructing these default encodings and reconstructing solutions from extensions should be algorithmically easy—polynomial (linear, whenever possible) in the size of the original problem.

In this section, we discuss techniques to systematically encode problems as default theories. Since extensions of default theories form *subsets* of the language, default theories can be used to represent those problems whose solutions are *subsets* of some domain. These solutions are usually defined as subsets of the domain satisfying certain constraints. With these insights, we propose an approach to programming with default logic that has two main components:

(1) Techniques to construct default theories representing collections of basic objects such as sets and functions.
(2) Techniques for modifying these default theories to eliminate extensions representing those objects that do not satisfy constraints implied by the original problem specification.

Although the target of default logic is knowledge representation, large test cases are needed for both experimentation and for studies of the methodology of representing problems as default theories. In our research, we chose the domain of combinatorics as the source of large and meaningful examples. In this domain it is easy to generate parameterized families of test cases needed for performance evaluation. Further, combinatorial problems are often specified in terms of constraints. Consequently, the domain of combinatorics can provide useful insights into modelling constraints as defaults or sets of defaults.

In what follows, we will be introducing techniques to impose constraints (item (2)) on default theories representing collections of sets and functions (item (1)). However, these techniques can be used in any application domain where constraints can be specified by means of default theories.

While in our discussion we focus on the propositional case, DATALOG-style encodings of some of the problems discussed below have been considered in [21,41,44].

4.1. Subsets

In this section we will present default theories whose extensions encode all subsets of a given set. For a propositional variable p let us define defaults

$$s^+(p) = \frac{:p}{p} \quad \text{and} \quad s^-(p) = \frac{:\neg p}{\neg p}.$$

Consider the default theory

$$\big(\{s^+(p), s^-(p)\}, \emptyset\big).$$

It is clear that this default theory has exactly two extensions, $Cn(\{p\})$ and $Cn(\{\neg p\})$. Consequently, it can be used to decide whether p is *in* or *out*.

Consider now a set X. Define a set of defaults $S_1(X)$ as follows:

$$S_1(X) = \big\{s^+(p)\colon\ p \in X\big\} \cup \big\{s^-(p)\colon\ p \in X\big\}.$$

Since, for $p \neq p'$, there are no interactions between defaults in $\{s^+(p), s^-(p)\}$ and $\{s^+(p'), s^-(p')\}$, we have the following observation.

Observation 4.1. *Let X be a set and let $Y \subseteq X$. A theory T is an extension of the default theory $(S_1(X), Y)$ if and only if*

$$T = Cn\big(\{p\colon\ p \in U\} \cup \{\neg p\colon\ p \in X \setminus U\}\big),$$

for some set $U \subseteq X$ such that $Y \subseteq U$.

It follows that there is a one-to-one correspondence between extensions of $(S_1(X), Y)$ and all subsets of X that contain Y. In other words, the default theory $(S_1(X), Y)$ can be used to represent all subsets of X containing Y.

Observe that elements $p \in X$ are treated in the definition of $S_1(X)$ as propositional variables. We will often use elements of combinatorial structures (for instance, vertices and edges of graphs) as propositional variables to indicate their membership in sets.

Another straightforward form of encoding all subsets of X is to introduce for every element p of X two propositional variables: $in(p)$ and $out(p)$. Consider the following two defaults:

$$t^+(p) = \frac{:\neg out(p)}{in(p)} \quad \text{and} \quad t^-(p) = \frac{:\neg in(p)}{out(p)}.$$

Consider the default theory

$$\big(\{t^+(p), t^-(p)\}, \emptyset\big).$$

This default theory has two extensions: $Cn(\{in(p)\})$ and $Cn(\{out(p)\})$. Hence, as before, this theory can be used to decide whether p is *in* or *out*.

Define a set of defaults $S_2(X)$ by:

$$S_2(X) = \big\{t^+(p)\colon\ p \in X\big\} \cup \big\{t^-(p)\colon\ p \in X\big\}.$$

The same argument as before yields the following observation, establishing a one-to-one correspondence between subsets of a set X, containing a prespecified subset $Y \subseteq X$, and extensions of the default theory $(S_2(X), \{in(p)\colon\ p \in Y\})$.

Observation 4.2. *Let X be a set and let $Y \subseteq X$. A theory T is an extension of the default theory $(S_2(X), \{in(p)\colon\ p \in Y\})$ if and only if*

$$T = Cn\big(\{in(p)\colon\ p \in U\} \cup \{out(p)\colon\ p \in X \setminus U\}\big),$$

for some set $U \subseteq X$ such that $Y \subseteq U$.

Let us observe that the theories $(S_1(X), Y)$ and $(S_2(X), \{in(p)\colon\ p \in Y\})$ are disjunction-free. Moreover, the theory $(S_2(X), \{in(p)\colon\ p \in Y\})$ has a straightforward translation into a logic program. Namely, the default $t^+(p)$ can be represented by the clause

$$in(p) \leftarrow \mathbf{not}(out(p)),$$

the default $t^-(p)$ can be represented by the clause

$$out(p) \leftarrow \mathbf{not}(in(p)),$$

whereas the atom $in(p)$ can be represented by the clause

$$in(p) \leftarrow$$

(see Section 2).

4.2. Maximal conflict-free sets

Often solutions to problems are specified as maximal *conflict-free* subsets. Let X be a set and let C be a function from X to $\mathcal{P}(X)$. If

(1) for every $x, y \in X$, $x \in C(y)$ if and only if $y \in C(x)$, and

(2) for every $x \in X$, $x \notin C(x)$,

then C is called a *conflict* function.

A subset Y of X is *conflict-free* if for every $x \in Y$, $C(x) \cap Y = \emptyset$. For every $x \in X$, define a default $select(x)$ by

$$select(x) = \frac{: \{\neg y\colon\ y \in C(x)\}}{x}.$$

The intuition behind the default $select(x)$ is as follows: if none of the elements in conflict with x is included in the solution, then include x.

Define now a set of defaults $SELECT(X, C)$ by

$$SELECT(X, C) = \{select(x)\colon\ x \in X\}.$$

Observation 4.3. *Let X be a set and let C be a conflict function from X to $\mathcal{P}(X)$. Let $Y \subseteq X$ be conflict-free. Then a theory T is an extension of $(SELECT(X, C), Y)$ if and only if $T = Cn(U)$, for some maximal* (*with respect to inclusion*) *conflict-free subset U of X such that $Y \subseteq U$.*

Clearly, Observation 4.3 establishes a one-to-one correspondence between maximal conflict-free subsets of X and extensions for $(SELECT(X, C), \emptyset)$.

Observe that the theory $(SELECT(X, C), Y)$ is disjunction-free. This theory can also be represented as a logic program by means of the translation described in Section 2.

4.3. Maximal independent subsets

A common type of a combinatorial structure appearing in practical applications is an *independent set*. Consider a *finite* collection $\mathcal{H}$ of *finite* subsets of a set X. A subset $Y \subseteq X$ is called *independent* for $\mathcal{H}$ if there is no $H \in \mathcal{H}$ such that $H \subseteq Y$. We will construct now a default theory that represents all maximal independent subsets for a family of sets $\mathcal{H} \subseteq \mathcal{P}(X)$.

For a finite set $H \subseteq X$, define a clause $\varphi(H)$ by

$$\varphi(H) = \bigvee \{\neg h\colon\ h \in H\}.$$

(Observe that, as before, we treat elements of X as propositional variables.) Let $x \in X$ and let $H_1, \ldots, H_k$ be all the sets in $\mathcal{H}$ containing x (recall that $\mathcal{H}$ is finite). Define

$$ind(x) = \frac{: \varphi(H_1 \setminus \{x\}), \ldots, \varphi(H_k \setminus \{x\})}{x}.$$

Consider a set X and a finite collection $\mathcal{H}$ of finite subsets of X. Define a set of defaults as follows:

$$MS(\mathcal{H}, X) = \big\{ind(x)\colon\ x \in X\big\}.$$

Observation 4.4. *Let $\mathcal{H}$ be a finite collection of finite subsets of a set X. Let $Y \subseteq X$ be an independent set for $\mathcal{H}$. Then, a theory T is an extension for the default theory $(MS(\mathcal{H}, X), Y)$ if and only if $T = Cn(U)$, for some maximal independent subset U of X such that $Y \subseteq U$.*

Observation 4.4 establishes a one-to-one correspondence between maximal independent sets for $\mathcal{H}$ and extensions of $(MS(\mathcal{H}, X), \emptyset)$.

Default theories $(MS(\mathcal{H}, X), Y)$ are not, in general, disjunction-free (unless $|H| = 2$ for all sets $H \in \mathcal{H}$). However, the existence of an extension problem for such theories is still only NP-complete.

The concept of a maximal independent set is a very general one. In particular, it is possible to represent maximal conflict-free sets as maximal independent sets in a suitably defined family $\mathcal{H}$.

4.4. Functions

In this section we will use the results of Section 4.2 to construct a default theory whose extensions correspond to all functions from a finite set X to a finite set Y. First, for every $x \in X$ and $y \in Y$, let us introduce a propositional variable $f_{x,y}$. This variable will represent the fact that y is assigned to x. The set of all these new variables will be denoted by $F(X, Y)$. For each new atom $f_{x,y}$, define its conflict set, $C(f_{x,y})$, by

$$C(f_{x,y}) = \{f_{x,z}\colon z \in Y, z \neq y\}. \tag{1}$$

Clearly, a subset F of $\{f_{x,y}\colon x \in X, y \in Y\}$ is a maximal conflict-free set if and only if there is a function $g : X \to Y$ such that $F = \{f_{x,g(x)}\colon x \in X\}$. Let us define the set of defaults $MAP(X, Y)$ as follows:

$$MAP(X, Y) = SELECT(F(X, Y), C),$$

where C is given by Eq. (1). Observation 4.3 implies the following corollary.

Corollary 4.1. *Let X and Y be finite sets, let $Z \subseteq X$ and let $h : Z \to Y$. A theory T is an extension for the default theory $(MAP(X, Y), \{f_{z,h(z)}\colon z \in Z\})$ if and only if $T = Cn(\{f_{x,g(x)}\colon x \in X\})$, for some function $g : X \to Y$ such that $g|Z = h$.*

Observe that the default theory $(MAP(X, Y), \{f_{z,h(z)}\colon z \in Z\})$ is disjunction-free.

4.5. Constraints

In this section, we will present a method to impose constraints that can be expressed by propositional formulas. That is, we will show how to modify a default theory so that the extensions of the resulting default theory are precisely those extensions of the original theory that satisfy the constraints.

Let φ be a propositional formula and let aux_φ be a new atom. Define the following defaults:

$$d_\varphi = \frac{: \neg\varphi, \neg aux_\varphi}{aux_\varphi}$$

and

$$d'_\varphi = \frac{\varphi\colon \neg aux_\varphi}{aux_\varphi}.$$

Theorem 4.2. *Let (D, W) be a default theory in a propositional language $\mathcal{L}$, let $\varphi \in \mathcal{L}$ and let aux_φ be a new propositional variable (not in $\mathcal{L}$). Let d_φ and d'_φ be defaults defined as above. Then*:

(1) *The theory (D, W) has an inconsistent extension if and only if the theory $(D \cup \{d_\varphi\}, W)$ has an inconsistent extension. Similarly, the theory (D, W) has an inconsistent extension if and only if the theory $(D \cup \{d'_\varphi\}, W)$ has an inconsistent extension.*

(2) *Every consistent extension of $(D \cup \{d_\varphi\}, W)$ is a subset of $\mathcal{L}$. Moreover, a consistent theory $E \subseteq \mathcal{L}$ is an extension of the default theory $(D \cup \{d_\varphi\}, W)$ if and only if E is an extension of (D, W) and $\varphi \in E$.*

(3) *Every consistent extension of $(D \cup \{d'_\varphi\}, W)$ is a subset of $\mathcal{L}$. Moreover, a consistent theory E is an extension of the default theory $(D \cup \{d'_\varphi\}, W)$ if and only if E is an extension of (D, W) and $\varphi \notin E$.*

Proof. The proof of (1) is straightforward. We leave it to the reader.

(2) Define $D' = D \cup \{d_\varphi\}$, and assume that E is a consistent extension of the default theory (D', W). We have

$$E = Cn^{D',E}(W).$$

Assume that d_φ is E-applicable. Then, since d_φ is prerequisite-free, $aux_\varphi \in E$. On the other hand, E-applicability of d_φ implies that $E \nvdash \neg(\neg aux_\varphi)$. Since E is closed under propositional consequence, we obtain a contradiction. Thus, d_φ is not E-applicable. It follows that $E \subseteq \mathcal{L}$ and that

$$Cn^{D',E}(W) = Cn^{D,E}(W).$$

Consequently,

$$E = Cn^{D,E}(W).$$

Hence, E is an extension of (D, W). Since E is consistent, d_φ is not E-applicable, and aux_φ occurs only in d_φ, it follows that $E \vdash \neg(\neg\varphi)$. Thus, $\varphi \in E$.

Conversely, assume that E is a consistent extension of (D, W) and that $\varphi \in E$. The latter fact implies that d_φ is not E-applicable. So, as before,

$$Cn^{D,E}(W) = Cn^{D',E}(W)$$

and, consequently,

$$E = Cn^{D',E}(W).$$

The proof of (3) is similar and we omit it. □

Theorem 4.2 shows that defaults d_φ and d'_φ can be used to enforce constraints expressed by propositional formulas. Enforcing means selecting those extensions that entail the

constraints. Defaults that act as such selection filters (for instance, d_φ and d'_φ) will be referred to as *selection defaults*. Observe also that when constructing the selection defaults, a formula φ can be replaced by a logically equivalent one (cf. [40], Theorem 5.3) without changing the selection properties of the default. We will often take advantage of this observation.

In general, we can use the same atom aux in all selection defaults. However, to decrease the number of dependencies between defaults and obtain finer stratification, it is better to use different auxiliary atoms in different selection defaults. Thus, in this section and throughout the paper we use a new auxiliary atom aux_φ for each selection default.

There are other classes of defaults that act as selection defaults. For instance, $\frac{:\neg\varphi}{\varphi}$ eliminates all extensions not containing φ (similarly to d_φ). However, the default $\frac{:\neg\varphi}{\varphi}$ may interact with other defaults and introduce cyclic dependencies that lead to larger strata.

4.6. Kernels in directed graphs

In the remainder of this section, we will present several default theories that encode problems in graph theory. They are constructed by first using our results about representing all subsets (or functions) and then by imposing constraints.

We will start by constructing default theories that represent the problem of existence of kernels in directed graphs. Given a directed graph $G = (V, A)$ (V stands for the set of vertices and A for the set of directed edges of G), a set $K \subseteq V$ is called a *kernel* if:

(K1) The set K is an independent set, that is, for every edge $(u, v) \in A$, $u \in V \setminus K$ or $v \in V \setminus K$.

(K2) For every vertex $w \in V \setminus K$, there exists a vertex $v \in K$ such that $(w, v) \in A$.

The first, rather *ad-hoc* representation of the kernel problem as a default theory appeared in [39]. Let $G = (V, A)$ be a directed graph. For every edge $e = (x, y) \in A$, define

$$r(e) = \frac{:\neg y}{x}.$$

Denote by $KER_1(G)$ the default theory $(\{r(e)\colon e \in A\}, \emptyset)$. It was shown in [39] that $K \subseteq V$ is a kernel of a directed graph $G = (V, A)$ if and only if $Cn(M)$, where $M = V \setminus K$, is an extension of $KER_1(G)$. In other words, extensions of this default theory are precisely the *complements* of kernels. Note that the theory $KER_1(G)$ is disjunction-free.

We will now construct another encoding of the kernel problem, systematically utilizing the results from the preceding sections. Consider the default theory $(S_1(V), \emptyset)$. Its extensions represent the collection of all subsets of V. More precisely, they are all of the form $\{x\colon x \in K\} \cup \{\neg x\colon x \in V \setminus K\}$, for some $K \subseteq V$. We will denote a set of this form, determined by $K \subseteq V$, by $\overline{K}$.

To represent kernels, we need to enforce kernel conditions (K1) and (K2) on such sets. To enforce (K1), for every directed edge $e = (x, y)$ define

$$\varphi(e) = \neg(x \wedge y).$$

Clearly, K satisfies condition (K1) if and only if $\overline{K}$ entails $\varphi(e)$, for every $e \in A$.

To enforce condition (K2), for every vertex v define a formula

$$\psi(v) = \neg v \supset v_1 \vee \cdots \vee v_k,$$

where $v_1, \ldots, v_k$ are all the vertices connected to v by an edge starting in v. Observe that a set of vertices K satisfies condition (K2) if and only if $\overline{K}$ entails $\psi(v)$, for every $v \in V$.

Formulas $\varphi(e)$ and $\psi(v)$ give rise to selection defaults

$$d_{\varphi(e)} = \frac{: x \wedge y, \neg aux_{\varphi(e)}}{aux_{\varphi(e)}},$$

for $e \in A$, $e = (x, y)$, and

$$d_{\psi(v)} = \frac{: \neg v \wedge \neg v_1 \wedge \cdots \wedge \neg v_k, \neg aux_{\psi(v)}}{aux_{\psi(v)}},$$

for $v \in V$ (where $v_1, \ldots, v_k$ are all the vertices connected to v by an edge starting in v). Notice that when defining $d_{\psi(v)}$, we replaced $\neg\psi(v)$ by an equivalent formula (using Theorem 5.3 from [40]).

Let $G = (V, A)$ be a directed graph. Let us denote by $KER_2(G)$ the default theory obtained by adding all defaults $d_{\varphi(e)}$, $e \in A$, and $d_{\psi(v)}$, $v \in V$, to the set of defaults $S_1(V)$ and setting $W = \emptyset$. Observe that the theory $KER_2(G)$ is disjunction-free.

Observation 4.5. *Let $G = (V, A)$ be a directed graph. A set $K \subseteq V$ is a kernel of G if and only if $Cn(\overline{K})$ is an extension of $KER_2(G)$. Moreover, every extension of $KER_2(G)$ is of the form $Cn(\overline{K})$, for some kernel K of G.*

Yet another approach is to encode complements of kernels, as it is easy to decode a set from its complement (this approach was used in [39]).

4.7. Maximal independent sets in graphs, matchings and perfect matchings

Let $G = (V, E)$ be an undirected graph. A set of vertices $I \subseteq V$ is *independent* if for every edge $e \in E$, at least one of its endvertices is not in I. Let us recall that an edge in an undirected graph can be identified with the *set* of its endvertices. Hence, it is clear that I is an independent set in G if and only if it is independent for E in the sense of Section 4.3. Let us denote $MIS(G) = (MS(E, V), \emptyset)$.

Observation 4.6. *Let $G = (V, E)$ be an undirected graph. A set $Y \subseteq V$ is a maximal independent subset of G if and only if $Cn(Y)$ is an extension of $MIS(G)$. Moreover, every extension of $MIS(G)$ is of the form $Cn(Y)$, for some maximal independent set Y in G.*

It is also easy to see that if $U \subseteq V$ is independent, then the default theory $(MS(E, V), U)$ describes all maximal independent sets in an undirected graph $G = (V, E)$ that contain U. Since all sets in E have only two elements, the theory $(MS(E, V), U)$ is disjunction-free.

An alternative encoding is implied by an observation that undirected graphs can be regarded as directed graphs (each undirected edge $\{x, y\}$ is treated as a pair of two directed edges (x, y) and (y, x)). It is easy to see that a set of vertices K is a kernel of an undirected graph G (regarded as a directed graph in the sense described above) if and only if K is a maximal independent set. Thus, extensions of the theory $KER_2(G)$, where $A = \{(x, y), (y, x) : \{x, y\} \in E\}$ correspond precisely to maximal independent sets of the (undirected) graph $G = (V, E)$.

Next, we will construct default theories representing all maximal matchings and perfect matchings in an undirected graph. Let $G = (V, E)$ be an undirected graph. A set of edges M is called a *matching* if no two different edges from M share an endvertex. A matching M is called *maximal* if there is no matching in G that would *properly* contain M. A matching M is called *perfect* if it covers all vertices of the graph.

Let $G = (V, E)$ be an undirected graph. Observe that $M \subseteq E$ is a matching if and only if M is independent for $\mathcal{E}(G) = \{\{e, f\}\colon\ e, f \in E, e \neq f, e \text{ and } f \text{ share an endvertex}\}$. Consequently, the default theory $(MS(\mathcal{E}(G), E), \emptyset)$ represents (through its extensions) all maximal matchings in G.

We will now add to $(MS(\mathcal{E}(G), E), \emptyset)$ selection defaults to weed out those maximal matchings that are not perfect. To this end, for every vertex $v \in V$ define the formula

$$cov(v) = e_1 \vee \cdots \vee e_k,$$

where $e_1, \ldots, e_k$ are all the edges with endvertex v. Clearly, a matching M is perfect if and only if M entails $cov(v)$, for every vertex $v \in V$. Each formula $cov(v)$ gives rise to the selection default $d_{cov(v)}$. Adding all these defaults to the set of defaults in $(MS(\mathcal{E}(G), E), \emptyset)$ yields a default theory, called $PM(G)$, whose extensions are those extensions of $(MS(\mathcal{E}(G), E), \emptyset)$ that entail all formulas $cov(v)$, that is, those extensions of $(MS(\mathcal{E}(G), E), \emptyset)$ that represent perfect matchings.

Observation 4.7. *Let $G = (V, E)$ be an undirected graph. A set of edges $M \subseteq E$ is a perfect matching of G if and only if $Cn(M)$ is an extension of $PM(G)$. Moreover, every extension of $PM(G)$ is of the form $Cn(M)$, for some perfect matching M of G.*

Since all sets in $\mathcal{E}(G)$ contain two elements and, since while constructing the selection default $d_{cov(v)}$ we can use $\neg e_1 \wedge \cdots \wedge \neg e_k$ instead of $\neg(e_1 \vee \cdots \vee e_k)$, the default theories $(MS(\mathcal{E}(G), E), \emptyset)$ and $PM(G)$ are disjunction-free.

If M' is a matching in a graph G, then extensions of the default theory $(MS(\mathcal{E}(G), E), M')$ represent all maximal matchings in G that contain M'. Theory $PM(G)$ can be modified in the same way. This yields a default theory representing all perfect matchings in the graph G containing M'.

4.8. Graph coloring

Let $G = (V, E)$ be an undirected graph. Let us denote by I_k the set $\{1, \ldots, k\}$. A function $f : V \to I_k$ is a *k-coloring* of G if for every edge $\{u, v\} \in E$, $f(u) \neq f(v)$. A graph G is *k-colorable* if there is a k-coloring of G. Since a coloring is a function from V to I_k which satisfies certain conditions, we can encode all k-colorings of a graph as a default theory using the results given in Sections 4.5 and 4.4. By Corollary 4.1, extensions of the default theory $(MAP(V, I_k), \emptyset)$ encode all functions from V to I_k.

We will now define propositional formulas that describe a violation of the condition that the endvertices of the same edge are assigned different colors. For every edge $e = \{x, y\} \in E$ and every $i \in I_k$, define

$$cl(e, i) = f_{x,i} \wedge f_{y,i}$$

(recall that $f_{v,p}$ is a new atom used in the construction of the default theory $MAP(X, Y)$ to represent the fact that $v \in X$ is assigned $p \in Y$). Hence, $cl(e, i)$ states that the endvertices of e are assigned color i.

It is easy to see that a function $c : V \to I_k$ is a coloring if for every $e \in E$ and every $i \in I_k$, $\{f_{x,c(x)}\colon x \in V\}$ *does not* entail $cl(e, i)$. Weeding out extensions that entail formulas $cl(e, i)$ can be accomplished by adding to $MAP(V, I_k)$ the selection defaults $d'_{cl(e,i)}$, $e \in E$, $i \in I_k$. Let us denote the resulting default theory by $COL_1(G, k)$.

Observation 4.8. *Let $G = (V, E)$ be an undirected graph. A function $c : V \to I_k$ is a k-coloring of G if and only if $Cn(\{f_{x,c(x)}\colon x \in V\})$ is an extension of $COL_1(G, k)$. Moreover, every extension of $COL_1(G, k)$ is of the form $Cn(\{f_{x,c(x)}\colon x \in V\})$, for some coloring c of G.*

Note that the theory $COL_1(G, k)$ is disjunction-free.

Another approach to encoding of the coloring problem was given in [45]. This encoding, $COL_2(G, k)$, can be constructed, using our approach, as follows. For every $x \in V$ and $i \in I_k$, define the conflict set $C(f_{x,i})$ by:

$$C(f_{x,i}) = \{f_{x,j}\colon j \in I_k, j \neq i\} \cup \{f_{y,i}\colon y \in V \text{ is a neighbor of } x\}. \tag{2}$$

It is clear that maximal conflict-free subsets of $\{f_{x,i}\colon x \in V, i \in I_k\}$ are maximal partial k-colorings of the graph G (a *partial coloring* is an assignment of colors to some of the vertices of the graph so that no edge has the same color assigned to its endvertices). Thus, maximal partial k-colorings of G are encoded (in a one-to-one fashion) by extensions of the default theory $(SELECT(F, C), \emptyset)$, where $F = \{f_{x,i}\colon x \in V, i \in I_k\}$ and C is defined by (2).

Next, for each vertex v, define a formula $s(v)$:

$$s(v) = f_{v,1} \vee \cdots \vee f_{v,k}.$$

Clearly, a subset of F entails $s(v)$ if and only if it contains at least one element of the form $f_{v,i}$. Thus, by Theorem 4.2, adding to $(SELECT(F, C), \emptyset)$ the selection defaults $d_{s(v)}$ leaves as extensions only those that encode *complete* k-colorings of G (colorings assigning a color to every vertex of the graph). Let us define

$$COL_2(G, k) = \big(SELECT(F, C) \cup \{d_{s(v)}\colon v \in V\}, \emptyset\big).$$

Observation 4.9. *Let $G = (V, E)$ be an undirected graph. A function $c : V \to I_k$ is a k-coloring of G if and only if $Cn(\{f_{x,c(x)}\colon x \in V\})$ is an extension of $COL_2(G, k)$. Moreover, every extension of $COL_2(G, k)$ is of the form $Cn(\{f_{x,c(x)}\colon x \in V\})$, for some coloring c of G.*

By using $\neg f_{v,1} \wedge \cdots \wedge \neg f_{v,k}$ instead of $\neg(f_{v,1} \vee \cdots \vee f_{v,k})$ when constructing $d_{s(v)}$, we can ensure that the theory $COL_2(G, k)$ is disjunction-free.

As in the previous cases, by modifying the objective part of the theories $COL_1(G, k)$ and $COL_2(G, k)$ one can encode the collection of those colorings that assign prespecified colors to prespecified vertices.

4.9. Cycles and hamiltonian cycles

Let $G = (V, A)$ be a directed graph such that $|V| \geqslant 3$. For an edge $e = (x, y) \in A$ let us define the conflict set

$$C(e) = \{(x, z) \in A\colon\ z \neq y\} \cup \{(z, y) \in A\colon\ z \neq x\}.$$

Let us observe that $H \subseteq A$ is a maximal conflict-free subset of A if and only if H is a maximal subset of edges in G with the following two properties:

(C1) no vertex is the tail of two different edges in H,

(C2) no vertex is the head of two different edges in H.

Consequently, the default theory $(SELECT(A, C), \emptyset)$ has as its extensions precisely the sets of the form $Cn(H)$, where $H \subseteq A$ is a maximal set satisfying conditions (C1) and (C2).

For every edge $e = (x, y) \in A$, let us define a default $move(e)$ by

$$move(e) = \frac{x \wedge e\colon}{y}.$$

The default $move(e)$ is justification-free. It is used like a standard inference rule. If $e = (x, y)$ and x are in an extension of a default theory that contains default $move(e)$, then y is in this extension as well. Let us define the default theory $\Delta(G)$ by:

$$\Delta(G) = \big(SELECT(A, C) \cup \{move(e)\colon\ e \in A\}, \{v_s\}\big),$$

where $v_s \in V$ is a fixed vertex. One can show that extensions of $\Delta(G)$ are precisely the theories of the form $Cn(X \cup H)$, where $H \subseteq A$ is a maximal subset of edges of G satisfying conditions (C1) and (C2) and X is the set of vertices reachable from v_s by means of the edges in H.

To leave only those extensions that correspond to hamiltonian cycles, it is enough to enforce two constraints:

(1) An extension must entail formulas v, for every $v \in V$ (in other words, all vertices must be reachable from v_s by means of edges in the extension),

(2) an extension must contain an edge with the head v_s.

To enforce the first constraint, the selection defaults d_v, $v \in V$ are added to $\Delta(G)$. To enforce the second constraint, the selection default

$$\frac{:\{\neg f\colon\ f \in A, f = (x, v_s)\}, \neg aux}{aux}$$

must be added. Let us denote the resulting theory by $HAM_1(G)$.

Observation 4.10. *Let $G = (V, A)$ be a directed graph. A set H of edges spans a hamiltonian cycle in G if and only if $Cn(V \cup H)$ is an extension of $HAM_1(G)$. Moreover, every extension of $HAM_1(G)$ is of the form $Cn(V \cup H)$, for some set $H \subseteq A$ spanning a hamiltonian cycle in G.*

Clearly, the theory $HAM_1(G)$ is disjunction-free.

We will now describe an alternative encoding. Let, as before, $G = (V, A)$ be a directed graph. For an edge $e = (x, y)$ define the default

$$move'(e) = \frac{x\colon \{\neg f\colon\ f = (x, z) \in A, z \neq y\}}{y \wedge e}.$$

The intuitive meaning of $move'(e)$ is: if x has been reached and it is possible to select an outgoing edge $e = (x, y)$ (none of the other outgoing edges from x is known to have been selected), then select e and visit y. Define $\Delta'(G)$ by:

$$\Delta'(G) = \big(\{move'(e)\colon\ e \in A\}, \{v_s\}\big),$$

where $v_s \in V$ is a fixed vertex. One can show that extensions of $\Delta'(G)$ are precisely the theories of the form $Cn(X \cup H)$, where H is a sequence of edges starting in v_s with each next edge starting where the previous one ended and X is the set of vertices of the edges in H. The sequence H ends when for the first time the head of an edge coincides with one of the vertices visited earlier.

Note that the sequence H need not to end in v_s and it is not guaranteed that all vertices are visited (that is, X may be a proper subset of V). To construct a default theory such that its extensions represent hamiltonian cycles, let us observe that to guarantee that all vertices are visited, we must require that the extensions entail the formulas v (v is treated here as a propositional variable), for all $v \in V$. Similarly, to guarantee that the sequence H ends up back in v_s we must ensure that the extensions entail the formula $\alpha = \bigvee\{e \in A\colon\ e \text{ ends in } v_s\}$. Both objectives can be accomplished by adding the selection defaults d_v, $v \in V$, and d_α to $\Delta'(G)$. Let us denote the resulting theory by $HAM_2(G)$.

Observation 4.11. *Let $G = (V, A)$ be a directed graph. A set H of edges spans a hamiltonian cycle in G if and only if $Cn(V \cup H)$ is an extension of $HAM_2(G)$. Moreover, every extension of $HAM_2(G)$ is of the form $Cn(V \cup H)$, for some set $H \subseteq A$ spanning a hamiltonian cycle in G.*

Note that $HAM_2(G)$ is disjunction-free.

5. TheoryBase

We believe that the lack of significant experimental studies of the performance of nonmonotonic reasoning systems can be, in large part, attributed to the absence, in the past, of large sets of test cases of varying difficulty and structure. This problem is not unique to automated theorem proving. It appears in all areas of experimental research [31].

To test and experiment with software systems we need easily generated, realistic and meaningful test instances. A possible approach is to produce a collection of real-life problems. Such benchmarks are now used in several areas of experimental research in computer science. The benefits of this approach are evident. The problems are *real* and, thus, *meaningful*. In addition, they can easily be disseminated. But, there are also drawbacks. The data often does not provide enough flexibility to allow full-fledged testing. In particular, a comprehensive study of performance scalability cannot be easily conducted,

as databases of benchmarks rarely contain *families* of test cases of similar structure and growing sizes that would allow good extrapolation of the running time.

The other approach frequently used in experimental research is to generate data randomly. This method offers an unlimited number of test cases and often the user has control over at least some parameters of data generated. For example, when generating random graphs, we can request a specific number of vertices and edges. However, the data generated randomly often has properties that rarely occur in real-life examples. It is well known that (under appropriate technical assumptions) almost every connected random graph is hamiltonian [10]. Similarly, it is now believed that random 3-SAT problems do not provide an adequate model for problems likely to occur in real-life applications [13,28].

None of these two approaches has been fully developed for experimenting with logic programming and nonmonotonic reasoning. In logic programming, the set of benchmark programs is very small. Two programs most commonly used in testing are the "naive reverse" program [56], and the "win" program [46,52]. The situation is even worse with generating logic programs and default theories randomly. In fact, up to now, no random model of a logic program or a default theory has been proposed.

In this section, we will describe a system that generates logic programs and default theories. Our approach is based on the work by Knuth on methods to generate graphs [33], and on the results from the previous section providing encodings of graph problems in terms of default theories and logic programs.

Knuth argues that random graphs do not constitute an adequate tool for testing graph algorithms. Instead, Knuth develops a graph generation system, The Stanford GraphBase. This system is publicly available (see [33] for details) and, thus, can be used as a "common denominator" for work requiring experimenting with graphs. The Stanford GraphBase is a collection of datasets and graph generating procedures. It allows the user to generate *families* of directed, undirected, weighted, unweighted, bipartite, planar, regular and random graphs. An important feature of The Stanford GraphBase is that every graph generated gets a unique label (or identifier). It is essential for storing and easy reconstruction of test cases generated.

The core of The Stanford GraphBase is formed by several procedures to generate *basic* graphs (other graphs can be obtained by applying graph operations implemented in The Stanford GraphBase). These procedures root the graphs they generate in objects such as maps and dictionaries in an effort to ensure some correlation of the graphs generated to real-life problems. For instance, an interesting family of graphs in The Stanford GraphBase is generated from a table of highway distances between 128 North American cities.

In our work, we extended The Stanford GraphBase to a system, called the TheoryBase, [5] that generates logic programs and default theories. It was developed to facilitate experimenting with DeReS. Our idea is to apply the encodings presented in Section 4 to graphs which are the outputs of The Stanford GraphBase.

The TheoryBase shell provides the user with two main classes of commands: to generate graphs, and to generate default theories encoding graph problems. The graph generating

[5] A detailed description of the TheoryBase commands and features, as well as the executable code can be obtained from ftp://ftp.cs.engr.uky.edu/cs/software/logic/TheoryBase.tar.gz.

commands rely on The Stanford GraphBase procedures. They allow the user to generate *families* of graphs of similar structure but increasing sizes.

The graph generating commands must be followed by invoking encoding generating commands. The encoding commands allow the user to specify a graph (or a range of graphs) generated before, a graph problem and a version of an encoding to use (they are minor modifications of the encodings presented in Section 4). Currently, the TheoryBase supports the following commands (together with available options, these commands allow the user to generate nine different encodings):

(1) `kernel`—this command produces the theory $KER_2(G)$ (to be precise, its slight modification) encoding the existence of a kernel for G; by selecting appropriate options two other encodings can also be generated,
(2) `color`—this command, invoked with the parameter k, generates the theory $COL_1(G, k)$ to encode the existence of a k-coloring problem for G,
(3) `hamilton`—produces the theory $HAM_1(G)$ to encode the existence of a hamiltonian cycle problem for G,
(4) `maxind`—generates the default theory $MIS(G)$, whose extensions identify all maximal independent sets in G,
(5) `maxmatch`—generates the default theory $(MS(\mathcal{E}(G), E), \emptyset)$ (see Section 4.3), whose extensions identify all maximal matchings in G.

Each of these commands generates: the header file (suffix *.dt*), the file of propositional formulas (suffix *.thc*), the file of defaults (suffix *.dc*), the stratification file (suffix *.str*).

The TheoryBase provides a unique identifier for each theory it allows the user to construct. The concept is an extension of a unique identifier of a graph in The Stanford GraphBase. Combining the name of the encoding generating command (possibly appended by strings representing a selection of options) with The Stanford GraphBase identifier of a graph for which the encoding is applied yields the identifier of the resulting default theory. For instance, if `kernel` command is applied to a graph with The Stanford GraphBase identifier *board*(5, 5, 0, 0, 5, 3, 1) (see Fig. 7) the resulting default theory is denoted by *kernel*.*board*_5, 5, 0, 0, 5, 3, 1_.[6] Similarly, applying the command *color* to the same graph, to produce a default theory encoding the existence of 3-colorings, yields the default theory with the identifier *color*3.*board*_5, 5, 0, 0, 5, 3, 1_.

The TheoryBase encoding generating commands also generate two additional files: the *graph description file* and the *display actions file*. These two additional files play no role in the reasoning but they support graphical presentation of the results by the TheoryBase and DeReS X11 graphical user interfaces. For instance, the graphical user interface for DeReS, `x11ext`, allows the user to display the underlying graph, identifies the graph problem to be solved, provides the user with several command buttons and displays the results of the computation. Fig. 7 presents the state of the interface after the first extension was computed for the theory encoding the existence of a kernel problem for the graph with The Stanford GraphBase identifier *board*(5, 5, 0, 0, 5, 3, 1).

Although the present focus in the TheoryBase is on test theories for experimentation with nonmonotonic reasoning, our method has wider implications. By encoding graph problems by means of propositional theories or 3-SAT data instances, one can obtain a

[6] For technical reasons, the parentheses in The Stanford GraphBase identifier are replaced by _ symbols.

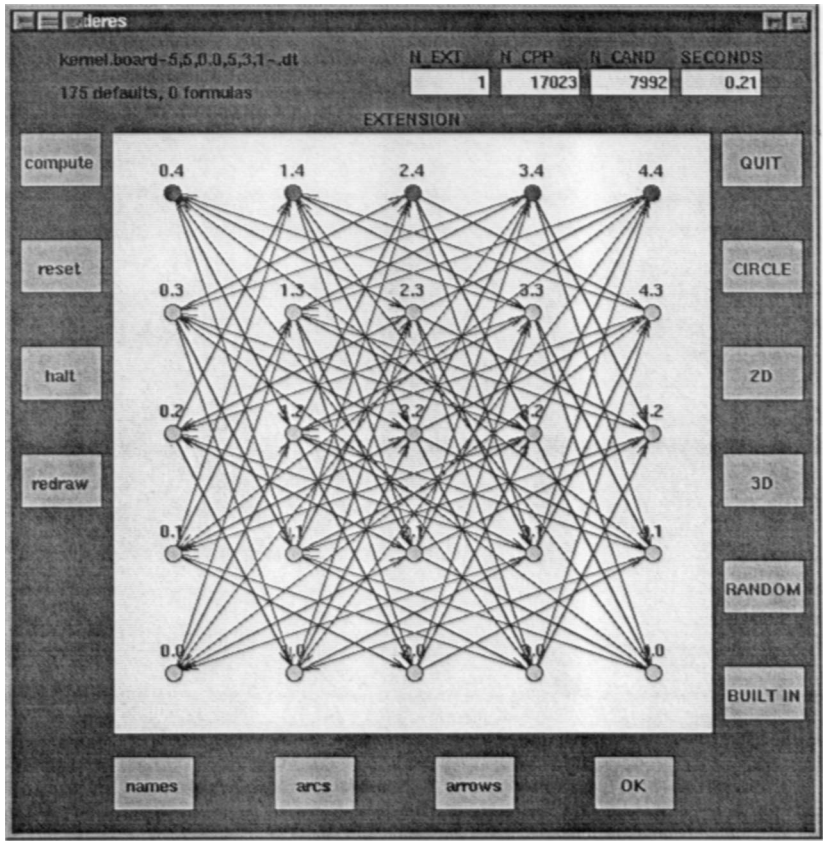


Fig. 7. A kernel in graph *board*(5, 5, 0, 0, 5, 3, 1).

benchmarking system for testing propositional theorem proving techniques. There is an obvious need for such a system (see [28] for additional discussion of the subject), especially in view of recent work on new satisfiability testing methods: GSAT [54], TABLEAU [12], WSAT [53], CSAT [19] and other.

6. Using TheoryBase, experimenting with DeReS

In this section we present the results of our experiments with DeReS and demonstrate usefulness of the TheoryBase in experimental studies of nonmonotonic reasoning systems. When studying DeReS, we were interested in the following three main questions:

(1) How does the performance of DeReS scale up with the growth of the size of input default theories?
(2) How the selection of a prover (recall that DeReS offers three choices) influences the performance of DeReS?
(3) What is the effect of stratification on the performance of DeReS?

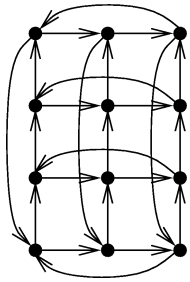


Fig. 8. A 3×4-grid wrapped around a torus.

In order to obtain meaningful and reliable results, testing must be extensive and the test cases must cover a wide spectrum of default theories with diverse properties.

The TheoryBase was designed to support this type of studies. Let us recall that the TheoryBase allows the user to produce *parameterized* families of default theories. The size of default theories in such a parameterized family grows as a function of the parameters and all the default theories in the family share similar properties. Several such families were constructed for our experiments.

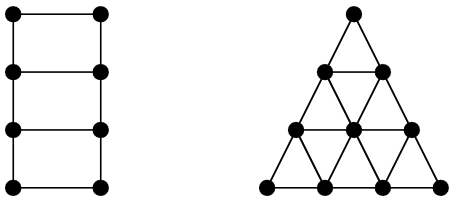
We will first discuss those families of default theories that are constructed by means of the TheoryBase `kernel` and `kernel -b` commands. These commands produce encodings of the existence of a kernel problem (through encodings KER_2 and KER_1, respectively). We applied these commands to several families of directed graphs, called $n \times m$*-tori*, whose vertices form an $n \times m$-grid wrapped on a torus, edges connect vertices at distance one in the grid, with the direction determined by the lexicographic ordering of the endpoints (see Fig. 8 for the 3×4-torus):

(1) $3 \times (3m - 1)$-tori, $m \geqslant 1$; The Stanford GraphBase labels $board(3, 3m - 1, 0, 0, 1, 3, 1)$,

(2) $4 \times 2m$-tori, $m \geqslant 1$; The Stanford GraphBase labels $board(4, 2m, 0, 0, 1, 3, 1)$.

We also applied these commands to the graphs with the vertex set representing squares on an $8 \times m$ chessboard, in which two vertices are connected if one can be reached from the other by a knight's move (with wraparound allowed along both dimensions). These graphs have The Stanford GraphBase labels $board(8, m, 0, 0, 5, 3, 1)$.

As a result, we obtained several families of default theories with labels *kernel.board_*p, $q, 0, 0, s, 3, 1\_$ and *kernel.b.board_*$p, q, 0, 0, s, 3, 1\_$, for appropriate values of p, q and s. All these theories are disjunction-free. Consequently, all three provers can be used by DeReS when processing them. The theories in the families with the prefix `kernel` have a relaxed stratification into small strata. The theories in the families with the prefix `kernel.b` have no nontrivial relaxed stratification. The theories obtained from graphs $board(4, 2m, 0, 0, 1, 3, 1)$ have exactly two extensions (it is easy to see that the corresponding graphs have exactly two kernels) and the theories obtained from graphs $board(3, 3m - 1, 0, 0, 1, 3, 1)$ have no extensions. Finally, the number of extensions for the theories *kernel.board_*$8, m, 0, 0, 5, 3, 1\_$ is a slowly growing function of m.



(a) (b)

Fig. 9. (a) A ladder graph; (b) A simplex graph.

We obtained especially encouraging results on DeReS performance for theories encoding the existence of k-colorings of graphs. We applied the TheoryBase `color` command, that implements the translation COL_1, to the following families of graphs:

(1) ladder graphs (see Fig. 9(a) for an example of a ladder graph), with The Stanford GraphBase labels $board(n, 2, 0, 0, 1, 0, 0)$,
(2) simplex graphs with the *side* of size n (see Fig. 9(b)), with The Stanford GraphBase labels $simplex(n, n, -2, 0, 0, 0, 0)$.

For graphs in these families, we generated theories encoding the existence of a 3-coloring. As a result, we obtained the following families of default theories:

(1) $color3.board\_n, 2, 0, 0, 1, 0, 0\_$, $n \geqslant 2$,
(2) $color3.simplex\_n, n, -2, 0, 0, 0, 0\_$, $n \geqslant 2$.

All these default theories are disjunction-free and have a good relaxed stratification. The theories $color3.board\_n, 2, 0, 0, 1, 0, 0\_$ have a large number of extensions (ladder graphs have exponentially many 3-colorings). The theories $color3.simplex\_n, n, -2, 0, 0, 0, 0\_$ have exactly six extensions (each graph $simplex(n, n, -2, 0, 0, 0, 0)$ has exactly six 3-colorings).

The effects of a fine relaxed stratification are perhaps best illustrated by the theories encoding the existence of a hamiltonian cycle problem. So far, no encoding with good stratification is known. It is easy to see that ladder graphs $board(n, 2, 0, 0, 1, 0, 0)$ have a hamiltonian cycle. We applied the command `hamilton` to the ladder graphs to produce the family $hamilton.board\_n, 2, 0, 0, 1, 0, 0\_$. Default theories in this family are disjunction-free and do not have a nontrivial relaxed stratification. Moreover, each has exactly two extensions (there are two directed hamiltonian cycles in the directed symmetric representation of a ladder graph).

This collection of test families demonstrates that the TheoryBase allows the user to generate a wide range of examples that can be used to test nonmonotonic reasoning systems. Some of the families we generated and used consist of theories which have a relaxed stratification into small clusters and others had only a trivial, one-cluster, relaxed stratification. Some families had no extensions, some other had very few extensions, and yet other had large numbers of extensions. Additional diversification was ensured by the fact that the families generated encode several graph problems and by the diversity of the underlying families of graphs.

In the remainder of this section we present experimental results on the performance of DeReS on the default theories described above. In all the tables we give, we use the following notation:

(1) $time_f$ denotes the CPU time for queries processed with the full propositional tableaux prover;
(2) $time_l$ denotes the CPU time for queries processed with the local propositional tableaux prover;
(3) $time_a$ denotes the CPU time for queries processed with the table lookup prover;
(4) NCPP stands for the number of calls to a prover;
(5) EXT stands for the total number of extensions for the input theory.

All times are measured in seconds.

The results were obtained on a 166 MHz Pentium PC under Linux 2.0.18 operating system. The time was measured using the `time` routine and is presented as the sum of the CPU time used while executing instructions in the user space of the calling process and the CPU time used by the system on behalf of the calling process. To capture the reasoning time we measure the CPU time from the point when an input default theory is already stored together with its stratification in DeReS data structures to the point when the answer is returned.

6.1. Provers, efficiency of DeReS processing and scalability

DeReS offers a choice of three propositional provers. Recall that these are: a full tableaux prover, a local tableaux prover (sound, but not complete), and a table lookup prover (applicable to disjunction-free theories only). All our experiments, perhaps not surprisingly, demonstrate that the local prover significantly and *uniformly* outperforms the full prover and that the lookup prover, whenever applicable, performs better than tableaux provers. In particular, this is illustrated in Table 1, which summarizes DeReS performance for the family of theories *kernel.board*_$8, m, 0, 0, 5, 3, 1$_ in the case when only one solution was needed, and in Table 2 that reports time needed to compute all extensions for these default theories.

In both cases time grows exponentially with the size of the underlying default theory. Nevertheless, both experiments show that DeReS can deal, in the matter of seconds, with default theories containing hundreds of defaults and encoding nontrivial problems.

The results from the tables can be used to extrapolate the behavior of the performance of DeReS for theories *kernel.board*_$8, m, 0, 0, 5, 3, 1$_ and obtain quantitative insights on the savings possible due to the choice of a prover. For instance, the time $time_a(m)$ (in μs) to compute all extensions using the table lookup prover satisfies the inequalities

$$C_1\, 3^m \leqslant time_a(m) \leqslant C_2\, 3^m,$$

for some small constants C_1 and C_2. Hence, the time grows exponentially and has order $\Theta(3^{|D|/56})$ (where, recall, D stands for the set of defaults of the theory). That is, the time grows at a much smaller rate than the theoretical bound $O(|D|^2 \times 2^{|D|})$ [40].

When tableaux provers are used times are larger because more time is needed for each call to the propositional provability procedure. For instance, the local prover needs to scan the input theory to find all formulas which have common propositional variables with the

Table 1
Searching for a kernel in $board(8, m, 0, 0, 5, 3, 1)$

$kernel.board\_8, m, 0, 0, 5, 3, 1\_$, one solution					
m	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
4	224	14804	14.32	1.04	0.06
5	280	34377	52.91	3.04	0.14
6	336	121249	291.27	12.28	0.49
7	392	105548	302.70	11.97	0.42
8	448	308910	1389.91	39.65	1.24
9	504	557398	2924.17	78.11	2.21
10	560	1982796	14327.56	316.29	7.86

Table 2
Computing all kernels in $board(8, m, 0, 0, 5, 3, 1)$

$kernel.board\_8, m, 0, 0, 5, 3, 1\_$, all solutions						
m	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$	EXT
4	224	65704	72.89	4.65	0.26	6
5	280	114709	208.79	10.23	0.48	15
6	336	421082	1039.76	42.77	1.65	5
7	392	1255383	4214.01	146.72	5.02	147
8	448	4130579	$>$ 2 hrs	541.35	16.29	134
9	504	10760494	$>$ 2 hrs	1603.21	42.53	120
10	560	31630658	$>$ 2 hrs	5204.96	124.24	267

query formula and then decide provability. From our results, it can be estimated that the time $time_l(m)$ (in μs) for computing all extensions by means of the local prover satisfies

$$C_1' \, m3^m \leqslant time_l(m) \leqslant C_2' \, m3^m,$$

that is, it is of the order $\Theta(|D| \times 3^{|D|/56})$. Finally, similar considerations for the full prover show that, in this case, the time needed to find all extensions is of the order $\Theta(|D|^2 \times 3^{|D|/56})$. Thus, for the default theories $kernel.board\_8, m, 0, 0, 5, 3, 1\_$, using the local prover saves a factor of $|D|$ over the full prover, and using the table lookup prover saves an additional factor of $|D|$.

The results were similar for several other families of default theories. In some cases, savings due to the choice of the prover were even more dramatic and led to excellent scalability. Table 3 summarizes running times of DeReS for all three provers for the family of default theories $color3.board\_n, 2, 0, 0, 1, 0, 0\_$.

In this case, due to a large number of solutions, we only computed the first extension (computing all would clearly take exponential time). As before, full and local provers are not practical while the table lookup prover performs very well. Even for very large default theories from this family, with tens of thousands of defaults, the table lookup version of DeReS computes an extension in less than a second. This excellent performance is due to two factors: relaxed stratification and a large number of extensions these theories have,

Table 3
Finding a 3-coloring for *board*$(n, 2, 0, 0, 1, 0, 0)$

color3.board_n, 2, 0, 0, 1, 0, 0_, one solution					
n	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
300	4494	11988	1343.57	10.75	0.08
400	5994	15988	3385.35	19.20	0.09
500	7494	19988	$>$ 2 hrs	30.27	0.11
600	8994	23988	$>$ 2 hrs	45.68	0.13
700	10494	27988	$>$ 2 hrs	62.74	0.14
800	11994	31988	$>$ 2 hrs	82.77	0.16
900	13494	35988	$>$ 2 hrs	108.05	0.18
1000	14994	39988	$>$ 2 hrs	137.02	0.20

Table 4
Finding a 3-coloring for *simplex*$(n, n, -2, 0, 0, 0, 0)$

color3.simplex_n, n, -2, 0, 0, 0, 0_, one solution					
n	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
6	270	806	0.26	0.05	0.01
7	360	1020	0.53	0.08	0.01
8	459	1845	1.10	0.19	0.01
9	570	1649	1.35	0.18	0.01
10	693	1950	2.18	0.27	0.01
11	828	3294	4.61	0.52	0.02
12	975	2789	5.44	0.50	0.02
13	1134	3177	8.16	0.67	0.02
14	1305	5160	16.21	1.29	0.03
15	1488	4226	17.74	1.19	0.03

which makes it easy to stumble upon them. Table 4 presents the performance results of DeReS for theories *color3.simplex_n*, $n, -2, 0, 0, 0, 0$_ (they encode 3-colorings of the simplex graphs). Each such theory has exactly six extensions corresponding to six 3-colorings of the graph *simplex*$(n, n, -2, 0, 0, 0, 0)$.

Finally, DeReS exhibits similar scalability and prover performance results for theories with no extensions. Table 5 summarizes our experiments with the family of theories *kernel.board_*$3, 3m-1, 0, 0, 1, 3, 1$_. Since these theories have no extensions, DeReS can terminate execution only after it scans through a portion of the search space that is large enough to allow it to conclude that indeed no extensions exist. Consequently, in this case, the performance of DeReS is worse than in the previous two cases.

All these results demonstrate the magnitude of savings possible with the appropriate choice of the propositional prover in DeReS. Significant savings were observed for theories encoding both existence of kernels and 3-colorings, and for theories with very many, moderately many, few and no extensions. They also show that the performance of DeReS, even in the current implementation, scales up very well for several nontrivial families of

Table 5
Searching for a kernel in $board(3, 3m-1, 0, 0, 1, 3, 1)$

*kernel.board*_3, 3*m* − 1, 0, 0, 1, 3, 1_						
m	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$	EXT
2	75	2,170	0.58	0.09	0.01	0
3	120	12,626	8.77	0.68	0.06	0
4	165	66,740	79.14	4.39	0.27	0
5	210	339,032	667.74	26.81	1.36	0
6	255	1,673,382	4890.73	154.85	6.79	0
7	300	8,093,622	32620.66	819.07	31.82	0

Table 6
Searching for a kernel in $board(4, 2m, 0, 0, 1, 3, 1)$, nonstratified encoding

*kernel.b.board*_4, 2*m*, 0, 0, 1, 3, 1_, one solution					
m	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
1	16	30,284	1.04	0.40	0.08
2	32	36,371,891	–	589.13	78.76
3	48	36,743,185,961	–	–	76,191.31

default theories. Our results point to the importance of encoding problems as disjunction-free theories as this allows the user to select the table lookup prover in DeReS.

6.2. Effects of relaxed stratification

Currently, the main pruning mechanism of DeReS is relaxed stratification. We will now discuss how it influences the performance of DeReS. In particular, we report experiments with theories that are equivalent (in the sense that they possess precisely the same extensions) but differ in the quality of relaxed stratification.

The times took by DeReS to find a single extension for the theories *kernel.b.board*_4, 2*m*, 0, 0, 1, 3, 1_ are shown in Table 6. Each of these theories has exactly two extensions. None of them has a good relaxed stratification. In general, in the encoding $KER_1(G)$, the strata correspond to the strong components of the underlying graph G. The size of each stratum is equal to the number of edges in G starting in the corresponding strong component of G. In particular, for strongly connected graphs, there is a single stratum of size $|D| = |E(G)|$. The graphs $board(4, 2m, 0, 0, 1, 3, 1)$ are strongly connected and have two edges originating in each of $8m$ vertices. Hence, the encoding $KER_1(G)$ has a single stratum of size $16m$.

Significantly better performance of DeReS is obtained if the theories *kernel.board*_4, 2*m*, 0, 0, 1, 3, 1_ are used. They encode the same problem, the existence of kernels, and for the same family of graphs, $board(4, 2m, 0, 0, 1, 3, 1)$, as theories *kernel.b.board*_4, 2*m*, 0, 0, 1, 3, 1_. However, as opposed to *kernel.b.board*_4, 2*m*, 0, 0, 1, 3, 1_, they have a relaxed stratification into small strata. The results are summarized in Table 7.

Table 7
Searching for a kernel in $board(4, 2m, 0, 0, 1, 3, 1)$, stratified encoding

*kernel.board*_4, 2*m*, 0, 0, 1, 3, 1_, *all solutions*					
m	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
1	36	484	0.05	0.02	0.01
2	80	989	0.45	0.08	0.01
3	120	6,674	3.51	0.34	0.04

Table 8
Searching for a kernel in $board(4, 2m + 1, 0, 0, 1, 3, 1)$, non-stratified encoding

*kernel.b.board*_4, 2*m* + 1, 0, 0, 1, 3, 1_					
m	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
1	24	914,523	41.95	13.70	2.01
2	40	1,153,615,536	–	–	2,438.99

Table 9
Searching for a kernel in $board(4, 2m + 1, 0, 0, 1, 3, 1)$, stratified encoding

*kernel.board*_4, 2*m* + 1, 0, 0, 1, 3, 1_					
m	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
1	60	671	0.13	0.03	0.01
2	100	3,157	1.35	0.15	0.02

Tables 6 and 7 show that the same problem can be represented in DeReS in an efficient way and in an inefficient manner. The difference is dramatic (7 orders of magnitude) and it points to the importance of good programming in DeReS. Whenever possible, one should encode problems by means of theories that have a good relaxed stratification.

Similarly significant speedups were observed for theories which have no extensions. Table 8 shows the timing results for the task of computing extensions for the theories *kernel.b.board*_4, 2*m* + 1, 0, 0, 1, 3, 1_ (they do not have extensions).

Again, once a stratified encoding was used, DeReS performance improved dramatically, as reported in Table 9.

Finally, the same poor performance of DeReS on theories without good relaxed stratification is observed for the default theories *hamilton.board_n*, 2, 0, 0, 1, 0, 0_ that encode the existence of hamiltonian cycles in ladder graphs $board(n, 2, 0, 0, 1, 0, 0)$ (Tables 10 and 11). It is worth noting that, to the best of our knowledge, these theories do not possess equivalent theories with small strata.

The results in this section demonstrate, on one hand, the importance of good search space pruning techniques and, on the other, the need for the programmer to understand them and to take full advantage of them. In particular, when solving problems by means of default logic, an effort should be made to always encode the problems by means of theories which admit relaxed stratification into strata of small sizes.

Table 10
Finding a hamiltonian cycle in *board*$(n, 2, 0, 0, 1, 0, 0)$

hamilton.board_n, 2, 0, 0, 1, 0, 0_, one solution					
n	$\|D\|$	NCPP	$time_f$	$time_l$	$time_a$
2	13	260	0.02	0.01	0.01
3	21	5248	0.55	0.11	0.01
4	29	121371	16.92	2.56	0.30
5	37	2598270	488.29	65.32	5.67
6	45	52139039	$>$ 2 hrs	1365.10	111.67

Table 11
Finding all hamiltonian cycles in *board*$(n, 2, 0, 0, 1, 0, 0)$ (There are two solutions for each of these theories.)

hamilton.board_n, 2, 0, 0, 1, 0, 0_, all solutions									
n	$\|V\|$	$\|E\|$	$\|D\|$	K	NCPP	CAND	$time_f$	$time_l$	$time_a$
2	4	4	13	8	1027	129	0.06	0.02	0.01
3	6	7	21	14	33239	1719	3.74	0.66	0.08
4	8	10	29	20	809973	26278	129.29	17.17	1.93
5	10	13	37	26	17478917	417441	4030.99	413.73	39.88
6	12	16	45	32	352170869	6672528	$>$ 2 hrs	$>$ 2 hrs	789.35

7. Conclusions and future work

We described a comprehensive environment for computation with default logic of Reiter. The implementation, the Default Reasoning System (DeReS) is capable of handling large default theories, often with thousands of defaults. Our paper reports the results of the past five years when DeReS has been implemented and experimented with.

DeReS performs significantly better if the programmer writes a program (a default theory) that is disjunction-free and possesses a fine relaxed stratification. This implies that good programming practices in DeReS require that the programmer submits (if possible) a theory with these desirable properties. From this perspective, DeReS is not much different from other declarative languages such as Prolog or LDL [58]. That is, the programmer writes a declarative program, but the ease with which DeReS is able to solve the problem depends on the syntactic form of the theory (i.e., of DeReS program).

In order to demonstrate that DeReS can handle large and diverse examples, we implemented a benchmarking environment for nonmonotonic reasoning, the TheoryBase. Building on the work of Knuth (The Stanford GraphBase) and the systematic technique for implementing constraints as defaults (outlined in Section 4.5) we were able to construct large examples of default theories. These examples can be used as benchmark problems for DeReS. Moreover, by using families of similar graphs as underlying structures, we were able to construct parameterized families of default theories, thus creating families of benchmarks. Those families allow us to extrapolate the behavior of the algorithms underlying DeReS.

Although our benchmarking system was implemented expressly to facilitate experimentation with DeReS, the TheoryBase can be used alone—without DeReS. All nonmonotonic reasoning systems can now use the TheoryBase as a tool for benchmarking.

Currently we are working on several improvements to DeReS that, we expect, will lead to a better performance. Those improvements can, roughly, be categorized in three main thrusts. First, we need better cluster-handling techniques. Those are necessary especially in the situation when the program does not admit a fine relaxed stratification. Second, the natural parallelism implied by the structure of the search tree associated with the default theory makes it possible to apply tools such as PVM (Parallel Virtual Machine) or DIB (Distributed Implementation of Backtracking) for speeding up DeReS performance. Third, a natural structure of the acyclic graph of clusters associated with the relaxed stratification, allows for a better control of backtracking (in effect, backjumping). We expect that the cumulative effect of all these techniques will result in significant improvements over the current performance of DeReS.

Acknowledgement

This work was partially supported by the NSF grants IRI-9400568, CDA-9502645 and IRI-9619233.

References

[1] K. Apt, H.A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, CA, 1988, pp. 89–142.
[2] G. Antoniou, E. Langetepe, V. Sperschneider, New proofs in default logic theory, Ann. Math. Artificial Intelligence 12 (1994) 215–230.
[3] R. Ben-Eliyahu, R. Dechter, Default logic, propositional logic and constraints, in: Proc. AAAI-91, Anaheim, CA, Morgan Kaufmann, Los Altos, CA, 1991.
[4] R. Ben-Eliyahu, L. Palopoli, Reasoning with minimal models: Efficient algorithms and applications, in: Proc. 4th Internat. Conference on the Principles of Knowledge Representation and Reasoning (KR-94), Bonn, Germany, Morgan Kaufmann, San Francisco, CA, 1994.
[5] P. Besnard, An Introduction to Default Logic, Springer, Berlin, 1989.
[6] N. Bidoit, C. Froidevaux, Negation by default and unstratifiable logic programs, Theoret. Comput. Sci. 78 (1991) 85–112.
[7] S. Brass, U.W. Lipeck, Bottom-up query evaluation with partially ordered defaults, in: Proc. 3rd Internat. Conference on Deductive and Object-Oriented Databases (DOOD-93), Lecture Notes in Computer Science, Vol. 760, Springer, Berlin, 1993, pp. 253–266.
[8] C. Bell, A. Nerode, R. Ng, V.S. Subrahmanian, Implementing stable semantics by linear programming, in: A. Nerode, L. Pereira (Eds.), Logic Programming and Non-Monotonic Reasoning, MIT Press, Cambridge, MA, 1993.
[9] C. Bell, A. Nerode, R. Ng, V.S. Subrahmanian, Implementing deductive databases by mixed integer programming, ACM Trans. Database Systems 21 (1996) 238–269.
[10] B. Bollobás, Random Graphs, Academic Press, New York, 1985.
[11] G. Brewka, Nonmonotonic Reasoning: Logical Foundations of Commonsense, Cambridge University Press, Cambridge, UK, 1991.
[12] J.M. Crawford, L.D. Auton, Experimental results on the crossover point in random 3-SAT, Artificial Intelligence 81 (1996) 31–57.

[13] J.M. Crawford, A.B. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, in: Proc. AAAI-94, Seattle, WA, 1994.
[14] M. Cadoli, F.M. Donini, M. Schaerf, Is intractability of nonmonotonic reasoning a real drawback?, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 946–951.
[15] P. Cholewiński, Reasoning with stratified default theories, in: Proc. LPNMR-95, Lecture Notes in Computer Science, Vol. 928, Springer, Berlin, 1995, pp. 273–286.
[16] P. Cholewiński, Stratified default theories, in: Proc. CSL-94, Lecture Notes in Computer Science, Vol. 933, Springer, Berlin, 1995, pp. 456–470.
[17] P. Cholewiński, W. Marek, A. Mikitiuk, M. Truszczyński, Experimenting with nonmonotonic reasoning, in: Proc. 12th International Conference on Logic Programming, MIT Press, Cambridge, MA, 1995, pp. 267–281.
[18] P. Cholewiński, W. Marek, M. Truszczyński, Default reasoning system deres, in: Proc. 5th Internat. Conference on the Principles of Knowledge Representation and Reasoning (KR-96), Cambridge, MA, 1996, pp. 518–528.
[19] O. Dubois, P. Andre, Y. Boufkhad, J. Carlier, Sat versus unsat, in: Cliques, Coloring and Satisfiability, Second DIMACS Implementation Challenge, American Mathematical Society, Providence, RI, 1996, pp. 415–436.
[20] J. Dix, U. Furbach, A. Nerode (Eds.), Proc. 4th Internat. Conference on Logic Programming and Non-Monotonic Reasoning, Springer, Berlin, 1997.
[21] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello, The KR System dlv: Progress Report, Comparisons, and Benchmarks, in: A.G. Cohn, L. Schubert, S.C. Shapiro (Eds.), Proc. 6th Internat. Conference on Principles of Knowledge Representation and Reasoning (KR-98), Trento, Italy, 1998, pp. 406–417.
[22] D.W. Etherington, R. Reiter, On inheritance hierarchies with exceptions, in: Proc. AAAI-83, Washington, DC, 1983, pp. 104–108.
[23] D.W. Etherington, Reasoning with Incomplete Information, Pitman, London, 1988.
[24] M.L. Ginsberg, Counterfactuals, Artificial Intelligence 30 (1986) 35–79.
[25] G. Gogic, H. Kautz, Ch. Papadimitriou, B. Selman, Compactness of knowledge representation: A comparative analysis, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 862–869.
[26] M. Gelfond, V. Lifschitz, The stable semantics for logic programs, in: R. Kowalski, K. Bowen (Eds.), Proc. 5th International Symposium on Logic Programming, MIT Press, Cambridge, MA, 1988, pp. 1070–1080.
[27] M. Gelfond, V. Lifschitz, Representing actions in extended logic programming, in: Proc. Internat. Joint Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, 1992, pp. 559–573.
[28] M.L. Ginsberg, D.A. McAllester, Gsat and dynamic backtracking, in: J. Doyle, E. Sandewall, P. Torasso (Eds.), Principles of Knowledge Representation and Reasoning (KR-94), Bonn, Germany, Morgan Kaufmann, San Francisco, CA, 1994, pp. 226–237.
[29] G. Gottlob, Complexity results for nonmonotonic logics, J. Logic Comput. 2 (1992) 397–425.
[30] S. Hanks, D. McDermott, Default reasoning, nonmonotonic logics and frame problem, in: Proc. AAAI-86, Philadelphia, PA, 1986, pp. 328–333.
[31] R.W. Hockney, The Science of Computer Benchmarking, SIAM, Philadelphia, PA, 1996.
[32] U. Junker, K. Konolige, Computing the extensions of autoepistemic and default logics with a truth maintenance system, in: Proc. AAAI-90, Boston, MA, 1990.
[33] D.E. Knuth, The Stanford GraphBase: A Platform for Combinatorial Computing, Addison-Wesley, Reading, MA, 1993.
[34] H.A. Kautz, B. Selman, Hard problems for simple default logics, in: Proc. 1st Internat. Conference on Principles of Knowledge Representation and Reasoning (KR-89), Toronto, Ontario, Morgan Kaufmann, San Francisco, CA, 1989, pp. 189–197.
[35] V. Lifschitz, H. Turner, Splitting a logic program, in: P. Van Hentenryck (Ed.), Proc. 11th Internat. Conference on Logic Programming, 1994, pp. 23–37.
[36] J. McCarthy, Circumscription—A form of nonmonotonic reasoning, Artificial Intelligence 13 (1980) 27–39.
[37] D. McDermott, J. Doyle, Nonmonotonic logic I, Artificial Intelligence 13 (1980) 41–72.
[38] W. Marek, M. Truszczyński, Stable semantics for logic programs and default theories, in: E. Lusk, R. Overbeek (Eds.), Proc. North American Conference on Logic Programming, MIT Press, Cambridge, MA, 1989, pp. 243–256.
[39] W. Marek, M. Truszczyński, Autoepistemic logic, J. ACM 38 (1991) 588–619.

[40] W. Marek, M. Truszczyński, Nonmonotonic Logics; Context-Dependent Reasoning, Springer, Berlin, 1993.
[41] W. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: K.R. Apt, W. Marek, M. Truszczyński, D.S. Warren (Eds.), The Logic Programming Paradigm: A 25-Year Perspective, Springer, Berlin, 1999, pp. 375–398.
[42] D. Maier, D.S. Warren, Computing with Logic. Logic Programming with Prolog, Benjamin/Cummings, Menlo Park, CA, 1988.
[43] I. Niemelä, On the decidability and complexity of autoepistemic reasoning, Fundamenta Informaticae 17 (1992) 117–155.
[44] I. Niemelä, Logic programs with stable model semantics as a constraint programming paradigm, in: I. Niemelä, T. Schaub (Eds.), Proc. Workshop on Computational Aspects of Nonmonotonic Reasoning, 1998, pp. 72–79.
[45] I. Niemelä, P. Simons, Evaluating an algorithm for default reasoning, in: Proc. IJCAI-95, Montreal, Quebec, 1995.
[46] I. Niemelä, P. Simons, Efficient implementation of the well-founded and stable model semantics, in: Proc. JICSLP-96, MIT Press, Cambridge, MA, 1996.
[47] D. Poole, Normality and faults in logic-based diagnosis, in: Proc. IJCAI-89, Detroit, MI, Morgan Kaufmann, San Mateo, CA, 1989, pp. 1206–1212.
[48] R. Reiter, G. Criscuolo, On interacting defaults, in: Proc. IJCAI-81, Vancouver, BC, 1981, pp. 270–276.
[49] R. Reiter, On closed world data bases, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Plenum Press, New York, 1978, pp. 55–76.
[50] R. Reiter, A logic for default reasoning, Artificial Intelligence 13 (1980) 81–132.
[51] R. Reiter, A theory of diagnosis from first principles, Artificial Intelligence 32 (1987) 57–95.
[52] P. Rao, I.V. Ramskrishnan, K. Sagonas, T. Swift, D.S. Warren, J. Freire, XSB: A system for efficiently computing well-founded semantics, in: Proc. LPNMR-97, Lecture Notes in Computer Science, Vol. 1265, Springer, Berlin, 1997, pp. 430–440.
[53] B. Selman, H.A. Kautz, B. Cohen, Local search strategies for satisfiability testing, in: Cliques, Coloring and Satisfiability, Second DIMACS Implementation Challenge, American Mathematical Society, Providence, RI, 1996, pp. 521–531.
[54] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proc. AAAI-92, San Jose, CA, Morgan Kaufmann, Los Altos, CA, 1992, pp. 440–446.
[55] V.S. Subrahmanian, D. Nau, C. Vago, Wfs + branch bound = stable models, IEEE Trans. Knowledge and Data Engineering 7 (1995) 362–377.
[56] L. Sterling, E. Shapiro, The Art of Prolog, MIT Press, Cambridge, MA, 1986.
[57] J. Stillman, The complexity of propositional default logics, in: Proc. AAAI-92, San Jose, CA, Morgan Kaufmann, Menlo Park, CA, 1992, pp. 794–799.
[58] C. Zaniolo, Design and implementation of logic based language for data intensive applications, in: Proc. International Conference on Logic Programming, MIT Press, Cambridge, MA, 1988.