Parametric Connectives in
Disjunctive Logic Programming

Nicola Leone and Simona Perri

Department of Mathematics, University of Calabria
I-87030 Rende (CS), Italy
{`leone, perri`}@mat.unical.it

Abstract. Disjunctive Logic Programming (DLP) is an advanced formalism for Knowledge Representation and Reasoning (KRR). DLP is very expressive in a precise mathematical sense: it allows to express every property of finite structures that is decidable in the complexity class Σ_2^P ($\mathrm{NP}^{\mathrm{NP}}$). Importantly, the DLP encodings are often simple and natural.

In this paper, we single out some limitations of DLP for KRR, which cannot naturally express problems where the size of the disjunction is not known "a priori" (like N-Coloring), but it is part of the input. To overcome these limitations, we further enhance the knowledge modelling abilities of DLP, by extending this language by *Parametric Connectives (OR and AND)*. These connectives allow us to represent compactly the disjunction/conjunction of a set of atoms having a given property. We formally define the semantics of the new language, named $\mathrm{DLP}^{\vee,\wedge}$ and we show the usefulness of the new constructs on relevant knowledge-based problems. We analyze the computational complexity of $\mathrm{DLP}^{\vee,\wedge}$, showing that the addition of parametric connectives does not bring a higher cost in that respect.

1 Introduction

Disjunctive logic programs are logic programs where disjunction is allowed in the heads of the rules and negation may occur in the bodies of the rules. Such programs are now widely recognized as a valuable tool for knowledge representation and commonsense reasoning [2, 9, 12, 3, 6, 8, 11, 1]. The most widely accepted semantics for DLP is the *answer sets semantics* proposed by Gelfond and Lifschitz [6] as an extension of the stable model semantics of normal logic programs [5]. According to this semantics, a disjunctive logic program may have several alternative models (but possibly none), called *answer sets*, each corresponding to a possible view of the world. Disjunctive logic programs under answer sets semantics are very expressive. It was shown in [4, 7] that, under this semantics, disjunctive logic programs capture the complexity class Σ_2^P (i.e., they allow us to express, in a precise mathematical sense, *every* property of finite structures over a function-free first-order structure that is decidable in nondeterministic polynomial time with an oracle in NP). As Eiter *et al.* [4] showed, the expressiveness of disjunctive logic programming has practical implications, since relevant practical problems can be represented by disjunctive logic programs, while they cannot be expressed by logic programs without disjunctions, given current complexity beliefs. Importantly,

even problems of lower complexity can be often expressed more naturally by disjunctive programs than by programs without disjunction.

As an example, consider the well-known problem of 3-coloring, which is the assignment of three colors to the nodes of a graph in such a way that adjacent nodes have different colors. This problem is known to be NP-complete. Suppose that the nodes and the edges are represented by a set F of facts with predicates $node$ (unary) and $edge$ (binary), respectively. Then, the following DLP program allows us to determine the admissible ways of coloring the given graph.

$$
\begin{aligned}
r_1 : \quad & color(X, r) \mathbin{\mathbf{v}} color(X, y) \mathbin{\mathbf{v}} color(X, g) \mathbin{:-} node(X) \\
r_2 : \quad & \mathbin{:-} edge(X, Y), color(X, C), color(Y, C)
\end{aligned}
$$

Rule r_1 above states that every node of the graph is colored **r**ed or **y**ellow or **g**reen, while r_2 forbids the assignment of the same color to any adjacent nodes. The minimality of answer sets guarantees that every node is assigned only one color. Thus, there is a one-to-one correspondence between the solutions of the 3-coloring problem and the answer sets of $F \cup \{r_1, r_2\}$. The graph is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some answer set.

Despite the high expressiveness of DLP, there are several problems which cannot be encoded in DLP in a simple and natural manner. Consider, for instance, the generalization of the 3-coloring problem above, where the number of admissible colors is not known "a priori" but it is part of the input. This problem is called *N-Coloring*: Given a graph G and a set of N colors, find an assignment of the N colors to the nodes of G in such a way that adjacent nodes have different colors.

The most natural encoding for this problem would be obtained by modifying rule r_1 in the above encoding of 3-coloring. The head

$$
color(X, r) \mathbin{\mathbf{v}} color(X, y) \mathbin{\mathbf{v}} color(X, g)
$$

should be replaced by a disjunction of N atoms representing the N possible ways of coloring the node at hand. This encoding, however, cannot be done in a uniform way, since the number of colors is not known "a priori"; but it is part of the input (the program should be changed for each number N of colors; while a uniform encoding requires the program to be fixed, and only the facts encoding the input to be varying).

To overcome these limitations, in this paper we enhance the knowledge modeling abilities of DLP, by extending this language by *Parametric Connectives (OR and AND)*. These connectives allow us to represent compactly the disjunction/conjunction of a set of atoms having a given property. For instance, by using parametric OR we obtain a simple and natural encoding of N-Coloring by modifying the above rule r_1 as follows:

$$
\bigvee \{col(X, C) : col(C)\} \leftarrow node(X).
$$

(see Section 4.1).

We formally define the semantics of the new language, named $\mathrm{DLP}^{\vee, \wedge}$, by providing a natural extension of the answer set semantics for $\mathrm{DLP}^{\vee, \wedge}$ programs. We show the usefulness of the new constructs on relevant knowledge-based problems. We analyze the computational complexity of $\mathrm{DLP}^{\vee, \wedge}$. Importantly, it turns out that the addition of

parametric connectives does not increase the computational complexity, which remains the same as for reasoning on DLP programs.

2   The DLP$^{\lor,\land}$ Language

In this section, we provide a formal definition of the syntax and the semantics of the DLP$^{\lor,\land}$ language.

2.1   Syntax

A variable or a constant is a *term*. A *standard atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *standard literal* is either a *standard positive literal p* or a *standard negative literal not p*, where p is a standard atom. A *standard conjunction* is $k_1, \cdots, k_n$ where each $k_1, \cdots, k_n$ is a standard literal. A *symbolic literal set* S is $\{L : Conj\}$ where L is a standard literal and $Conj$ is a standard conjunction; if L is a positive standard literal, S is called *positive symbolic literal set*. A *parametric AND literal* is $\bigwedge S$ where β is a symbolic literal set. A *parametric OR atom* is $\bigvee S$ where S is a positive symbolic literal set. From now on, we refer to both parametric AND literal and parametric OR atom as parametric literals.

Example 1. $\bigvee\{a(X,Y) : q(X,Y), not\ r(Y)\}$ is a parametric OR atom and $\{a(X,Y) : q(X,Y), not\ r(Y)\}$ is the positive symbolic literal set. Intuitively, the above parametric OR atom stands for the disjunction of all instances of $a(X,Y)$ such that the conjunction $q(X,Y), not\ r(Y)$ is true.

An *atom* is either a standard atom or a parametric OR atom. A *literal* is either a standard literal or a parametric AND literal.

A *(disjunctive) rule* r is a syntactic of the following form:

$$a_1 \text{ v } \cdots \text{ v } a_n :- l_1, \cdots, l_m. \qquad n \geq 0, m \geq 0$$

where $a_1, \cdots, a_n$ are atoms and $l_1, \cdots, l_m$ are literals.
The disjunction $a_1 \text{ v } \cdots \text{ v } a_n$ is the *head* of r, while the conjunction $l_1, \cdots, l_m$ is the *body* of r.

We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set $\{l_1, ..., l_m\}$ of the body literals.

An *(integrity) constraint* is a rule with an empty head.

A DLP$^{\lor,\land}$ *program* $\mathcal{P}$ is a finite set of rules. A ¬-free (resp., v -free) program is called *positive* (resp., *normal*). A program where neither parametric atoms nor parametric literals appear is called (*standard*) *DLP program*. A term, an atom, a literal, a rule, or a program are *ground* if no variables appear.

2.2   Syntactic Restrictions and Notation

A variable X appearing solely in a parametric literal of a rule r is a *local variable* of r; otherwise, X is a *global variable* of r.

Example 2. Consider the following rule

$$p(Y, Z) :\!- \bigwedge \{q(X, Y) : a(X, Z))\}, t(Y), r(Z).$$

X is the only local variable, while Y and Z are global variables.

Safety A rule r is *safe* if the following conditions hold:

 (i) each global variable of r appears in a positive standard literal in the body of r;
(ii) each local variable of r appearing in a symbolic set $\{L : Conj\}$, also appears in a positive literal in $Conj$.

A program is *safe* if all of its rules are safe.

Example 3. Consider the following rules:

$$\bigvee \{p(X, Y) : q(Y)\} :\!- r(X).$$
$$p(X, Z) :\!- \bigwedge \{q(X, Y) : a(X))\}, s(X, Z).$$
$$p(X) :\!- \bigwedge \{q(X, Y) : a(X)\}, t(Y).$$

The first rule is safe, while the second is not, since the local variables Y violates condition (ii). The third rule is not safe either, since the global variable X violates condition (i).

From now on, throughout this paper, we assume that all rules of a $DLP^{\vee, \wedge}$ $\mathcal{P}$ are safe.

2.3   Semantics

Program Instantiation. Given a DLP$^{\vee, \wedge}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$, and $B_{\mathcal{P}}$ the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$.

A *substitution* is a mapping from a set of variables to the set $U_{\mathcal{P}}$ of the constants appearing in the program $\mathcal{P}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for S*. Given a symbolic set without global variables $S = \{L : Conj\}$, the *instantiation of set S* is the following ground set of pairs $S' = \{\langle \gamma(L) : \gamma(Conj) \rangle \mid \gamma$ is a local substitution for $S\}$[1]; S' is called *ground literal set*.

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

---

[1] Given a substitution σ and a DLP$^{\vee, \wedge}$ object Obj (rule, conjunction, set, etc.), with a little abuse of notation, we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

Example 4. Consider the following program $\mathcal{P}_1$:

$$q(1) \lor p(2,2). \qquad q(2) \lor p(2,1).$$
$$t(X) :\!\!-\, q(X), \bigwedge\{a(Y) : p(X,Y)\}.$$

The instantiation $Ground(\mathcal{P}_1)$ is the following:

$$q(1) \lor p(2,2). \qquad q(2) \lor p(2,1).$$
$$t(1) :\!\!-\, q(1), \bigwedge\{\langle a(1) : p(1,1)\rangle, \langle a(2) : p(1,2)\rangle\}.$$
$$t(2) :\!\!-\, q(2), \bigwedge\{\langle a(1) : p(2,1)\rangle, \langle a(2) : p(2,2)\rangle\}.$$

Interpretation and models. An *interpretation* for a $\text{DLP}^{\lor,\land}$ program $\mathcal{P}$ is a set of standard ground atoms $I \subseteq B_{\mathcal{P}}$. The truth valuation $I(A)$, where A is a standard ground literal is defined in the usual way. Besides assigning truth values to the standard ground literals, an interpretation provides the meaning also to (ground)literal sets, and to (the instantiation of) parametric literals. Let S be a (ground) literal set. The valuation $I(S)$ of set S w.r.t. I is the set

$$\{L \mid (L : conj \in S) \land (conj\ is\ true\ w.r.t\ I)\}.$$

Given a symbolic literal set, let S' be the instantiation of S. Then a *parametric AND literal* $\bigwedge S$ is true w.r.t I if all the standard literals in $I(S')$ are true w.r.t I. Similarly, a *parametric OR atom* $\bigvee S$, is true w.r.t I if at least one of the standard literals in $I(S')$ is true w.r.t I.

Example 5. Let $U_{\mathcal{P}}$ be the set $\{1,2\}$ and I the interpretation $\{p(1),p(2), a(1,2), a(2,1), b(1), b(2)\}$. Consider the parametric AND atom

$$\bigwedge S = \bigwedge\{p(X) : a(X,Y), b(X)\}$$

Then the instantiation of S is

$$S' = \{\langle p(1) : a(1,1), b(1)\rangle, \langle p(1) : a(1,2), b(1)\rangle,$$
$$\langle p(2) : a(2,1), b(2)\rangle, \langle p(2) : a(2,2), b(2)\rangle\}$$

and its value w.r.t I is $I(S') = \{p(1), p(2)\}$. $\bigwedge S$ is true w.r.t. I because both $p(1)$ and $p(2)$ are true w.r.t I.

Using the above notion of truth valuation for parametric literals, the notion of models, minimal models and answer sets for $\text{DLP}^{\lor,\land}$ are an immediate extension of the corresponding notions in standard DLP [6].

3   Declarative Programming in Standard DLP

3.1   The GC Declarative Programming Methodology

The standard DLP language can be used to encode problems in a highly declarative fashion, following a "**GC**" (Guess/Check) paradigm. In this section, we will describe

this technique and we then illustrate how to apply it on a number of examples. Many problems, also problems of comparatively high computational complexity (that is, even Σ_2^P-complete problems), can be solved in a natural manner with DLP by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are even more complex than NP, and the (optional) separation of a fixed, non-ground program from an input database allows to do so uniformly over varying instances.

Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem **P**, a **GC** program $\mathcal{P}$ for **P** consists of the following two main parts:

Guessing Part The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, in a way such that answer sets of $\mathcal{G} \cup \mathcal{F}_I$ represent "solution candidates" for I.

Checking Part The checking part $\mathcal{C} \subseteq \mathcal{P}$ of the program tests whether a solution candidate is in fact an admissible solution, such that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the solutions for the problem instance I.

The two layers above can also use additional auxiliary predicates, which can be seen as a background knowledge.

In general, we may allow both $\mathcal{G}$ and $\mathcal{C}$ to be arbitrary collections of rules in the program, and it may depend on the complexity of the problem which kinds of rules are needed to realize these parts (in particular, the checking part); we defer this discussion to a later point in this chapter.

Without imposing restrictions on which rules $\mathcal{G}$ and $\mathcal{C}$ may contain, in the extremal case we might set $\mathcal{G}$ to the full program and let $\mathcal{C}$ be empty, i.e., all checking is integrated into the guessing part such that solution candidates are always solutions. However, in general the generation of the search space may be guarded by some rules, and such rules might be considered more appropriately placed in the guessing part than in the checking part. We do not pursue this issue any further here, and thus also refrain from giving a formal definition of how to separate a program into a guessing and a checking part.

For many problems, however, a natural **GC** program can be designed, in which the two parts are clearly identifiable and have a simple structure:

- The guessing part $\mathcal{G}$ consists of some disjunctive rules which "guess" a solution candidate S.
- The checking part $\mathcal{C}$ consists of integrity constraints which check the admissibility of S.

All two layers may also use additional auxiliary predicates, which are defined by normal stratified rules.Such auxiliary predicates may also be associated with the guess for a candidate, and defined in terms of other guessed predicates, leading to a more "educated guess" which reduces blind guessing of auxiliary predicates; this will be seen in some examples below.

Thus, the disjunctive rules define the search space in which rule applications are branching points, while the integrity constraints prune illegal branches.

Remark 1. The **GC** *programming methodology has positive implications also from the Software Engineering viewpoint. Indeed, the modular program structure in* **GC** *allows us to develop programs incrementally providing support for simpler testing and debugging activities. Indeed, one first writes the Guess module $\mathcal{G}$ and tests that $\mathcal{G} \cup \mathcal{F}_I$ correctly defines the search space. Then, one deals with the Check module and verifies that the answer sets of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ are the admissible problem solutions.*

3.2 Applications of the GC Programming Technique

In this section, we illustrate the declarative programming methodology described in Section 3.1 by showing its application on a couple of standard problems from graph theory.

Hamiltonian Path Let us consider now a classical NP-complete problem in graph theory, namely *Hamiltonian Path*.

Definition 1 (HAMPATH). *Given a directed graph $G = (V, E)$ and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?*

Suppose that the graph G is specified by using predicates $node$ (unary) and arc (binary), and the starting node is specified by the predicate $start$ (unary). Then, the following **GC** program $\mathcal{P}_{hp}$ solves the problem HAMPATH.

$$
\begin{aligned}
&inPath(X, Y) \text{ v } outPath(X, Y) :\!- start(X), arc(X, Y). \\
&inPath(X, Y) \text{ v } outPath(X, Y) :\!- reached(X), arc(X, Y).
\end{aligned}
\quad \Big\} \textbf{ Guess}
$$

$$
\begin{aligned}
&:\!- inPath(X, Y), inPath(X, Y1), Y <> Y1. \\
&:\!- inPath(X, Y), inPath(X1, Y), X <> X1. \\
&:\!- node(X), \text{not } reached(X), \text{not } start(X).
\end{aligned}
\quad \Big\} \textbf{ Check}
$$

$$
reached(X) :\!- inPath(Y, X).
\quad \Big\} \begin{aligned}&\textbf{Auxiliary}\\&\textbf{Predicate}\end{aligned}
$$

The two disjunctive rules guess a subset S of the given arcs to be in the path, while the rest of the program checks whether that subset S constitutes a Hamiltonian Path. Here, an auxiliary predicate $reached$ is used, which is associated with the guessed predicate $inPath$ using the last rule.

The predicate $reached$ influences through the second rule the guess of $inPath$, which is made somehow inductively: Initially, a guess on an arc leaving the starting node is made by the first rule, and then a guess on an arc leaving from a reached node by the second rule, which is repeated until all reached nodes are treated.

In the Checking Part, the first two constraints check whether the set of arcs S selected by $inPath$ meets the following requirements, which any Hamiltonian Path must satisfy: (i) there must not be two arcs starting at the same node, and (ii) there must not be two arcs ending in the same node. The third constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. This constraint also ensures that this subgraph is connected.

It is easy to see that any set of arcs S which satisfies all three constraints must contain the arcs of a path $v_0, v_1, \ldots, v_k$ in G that starts at node $v_0 = a$, and passes through distinct nodes until no further node is left, or it arrives at the starting node a again. In the latter case, this means that the path is a Hamiltonian Cycle, and by dropping the last arc, we have a Hamiltonian Path.

Thus, given a set of facts $\mathcal{F}$ for $node$, arc, and $start$, specifying the problem input, the program $\mathcal{P}_{hp} \cup \mathcal{F}$ has an answer set if and only if the input graph has a Hamiltonian Path. Thus, the above program correctly encodes the decision problem of deciding whether a given graph admits an Hamiltonian Path or not.

This encoding is very flexible, and can be easily adapted to solve both the *search problems* Hamiltonian Path and Hamiltonian Cycle (where the result is to be a tour, i.e., a closed path). If we want to be sure that the computed result is an *open* path (i.e., it is not a cycle), then we can easily impose openness by adding a further constraint $:- start(Y), inPath(\_, Y).$ to the program (like in Prolog, the symbol '$\_$' stands for an anonymous variable, whose value is of no interest). Then, the set S of selected arcs in an answer set of $\mathcal{P}_{hp} \cup \mathcal{F}$ constitutes a Hamiltonian Path starting at a. If, on the other hand, we want to compute a Hamiltonian Cycle, then we have just to strip off the literal not $start(X)$ from the last constraint of the program.

N-Coloring Now we consider another classical NP-complete problem from graph theory, namely *N-Coloring*.

Definition 2 (N-COLORING). *Given a graph $G = (V, E)$, a N-Coloring of G is an assignment of one, among N colors, to each vertex in V, in such a way that every pair of vertices joined by an edge in E have different colors.*

Let us suppose that the graph G is represented by a set of facts with predicates $vertex$ (unary) and $edge$ (binary), respectively. Then, the following DLP program $\mathcal{P}_{col}$ determines the admissible ways of coloring the given graph.

$$col(X, I) \text{ v } not\_col(X, I) :- vertex(X), color(I). \quad \rbrace \textbf{ Guess}$$

$$:- col(X, I), col(Y, I), edge(X, Y).$$
$$:- col(X, I), col(X, J), I <> J.$$
$$:- vertex(X), \text{not } colored(X).$$

$$colored(X) :- col(X, I).$$

$\rbrace$ **Check**

$\rbrace$ **Auxiliary Predicate**

The disjunctive rule guesses a graph coloring; $col(X, I)$ says that vertex X is assigned to color I and $not\_col(X, I)$ that it is not. The constraints in the checking part verify that the guessed coloring is a legal N-Coloring. In particular the first constraint assert that two joined vertices cannot have the same color, while the other two constraints impose that each vertex is assigned to exactly one color.

The answer sets of $\mathcal{P}_{col}$ are all the possible legal N-Colorings of the graph. That is, there is a one-to-one correspondence between the solutions of the N-Coloring problem and the answer sets of $\mathcal{P}_{col}$. The graph is N-colorable if and only if there exists one of such answer sets.

Maximal Independent Set Another classical problem in graph theory is the independent set problem.

Definition 3 (Maximal Independent Set). *Let $G = (V, E)$ be an undirected graph, and let $I \subseteq V$. The set I is* independent *if whenever $i, j \in I$ then there are no edges between i and j. An independent set I is* maximal *if no one among supersets of I is an independent set.*

Suppose that the graph G is represented by a set of facts F with predicates $node$ (unary) and $edge$ (binary). The following program $\mathcal{P}_{IndSet}$ computes the maximal independent sets of G:

$$(r_1) \quad in(X) \vee out(X) :- node(X).$$

$\left.\vphantom{\begin{matrix}a\end{matrix}}\right\}$ **Guess**

$$(c_1) \quad :- in(X), in(Y), edge(X, Y).$$
$$(c_2) \quad :- out(X), toBeExcluded(X).$$

$\left.\vphantom{\begin{matrix}a\\b\end{matrix}}\right\}$ **Check**

$$(r_2) \quad toBeExcluded(X) :- in(Y), edge(X, Y).$$

$\left.\vphantom{\begin{matrix}a\end{matrix}}\right\}$ **Auxiliary Predicate**

The rule r_1 guesses a set of vertices; $in(X)$ means that node X belongs to the set while $out(X)$ means that it does not. Then, the integrity constraint c_1 verifies that the guessed set is independent. In particular, it says that it is not possible that two nodes joined by an edge belong to the set.

Note that the answer sets of $F \cup \{r_1, c_1\}$ correspond exactly to the independent sets of G.

The maximality of the set is enforced by constraint c_2 using the auxiliary predicate $toBeExcluded$. A node X has to be excluded by the set because a node connected to it is already in the set. Then c_2 says that it is not possible that a node is out of the set if there is no reason to exclude it.

4   Knowledge Representation by DLP$^{\vee, \wedge}$

In this section, we show how DLP extended with parametric connectives can be used to encode relevant problems in a natural and elegant way.

4.1   N-Coloring

In the previous section we showed an encoding for the N-Coloring problem, following the **GC** paradigm. Now, we show how the extension of DLP with parametric connectives allows us to represent the N-Coloring problem in a much more intuitive way by simply modifying the elegant encoding of 3-colorability described in the Introduction.

Suppose again that the graph in input is represented by predicates $vertex$ (unary) and $edge$ (binary) and the set of N admissible colors is provided by a set of facts $color(c_1), \cdots, color(c_N)$. Then, the following simple DLP$^{\vee, \wedge}$ program computes the N-Colorings of the graph.

$$\bigvee\{col(X,C) : color(C)\} \text{ :- } vertex(X).$$
$$\text{:- } col(X,C), col(Y,C), edge(X,Y), X \neq Y.$$

The first rule guesses all possible N-Colorings. It contains in the head a parametric atom representing the disjunction of all the atoms $col(X,c_1), \cdots, col(X,c_N)$, where $c_1, \cdots, c_N$ are the N colors (i.e. the disjunction of all the atoms representing the possible way to color X). For each vertex v, the following ground rule belongs to the instantiation of the program:

$$\bigvee\{\langle col(v,c_1) : color(c_1)\rangle, \cdots, \langle col(v,c_N) : color(c_N)\rangle\} \text{ :- } vertex(v).$$

Since vertex v and $color(c_1), \cdots color(c_N)$ are always true, the above rule stands for the following disjunction

$$col(v,c_1) \vee \cdots \vee col(v,c_N)$$

The constraint simply checks that the N-Coloring is correct, that is, adjacent nodes must always have different colors.

4.2   Maximal Independent Set

Another problem which can be easily encoded in a more intuitive way by DLP$^{\vee,\wedge}$ is maximal independent set. Indeed, this problem can be represented by the following simple *one-rule* encoding.

$$in(X) \text{ :- } node(X), \bigwedge\{\text{not } in(Y) : arc(X,Y)\}.$$

As usual, the graph in input is encoded by predicates $node$ and arc and the atom $in(X)$ means that node X belongs to the set. Intuitively, such rule says that node X belongs to the independent set if for each node Y which is connected to it, Y does not belong to the set. In particular, the parametric AND literal $\bigwedge\{\text{not } in(Y) : arc(X,Y)\}$ is the conjunction of all the literals not $in(Y)$ such that there exists an edge between X and Y.

Note that, differently from the **GC** encoding shown in the previous section this formulation does not need the predicate $out(X)$ and the auxiliary predicate $toBeExcluded(X)$ used to mark the nodes that have to be excluded by the set.

It is worth noting that we do not need further rules to express maximality property, which, indeed, comes for free.

4.3   N-Queens

DLP$^{\vee,\wedge}$ allows to obtain another intuitive and elegant encoding also with respect to the problem of *N-Queens*.

Definition 4 (N-QUEENS). *Place N queens on a N*N chess board such that the placement of no queen constitutes an attack on any other. A queen attacks another if it is in the same row, column, or on a diagonal.*

Let's suppose that rows and columns are represented by means of facts $row(1)., \cdots, row(N).$ and $column(1)., \cdots, column(N).$ Then the following $\text{DLP}^{\vee, \wedge}$ problem solves the N-Queens problem.

$$\bigvee\{q(X,Y) : column(Y)\} :\text{--} \, row(X).$$
$$(c_1) \quad :\text{--} \, q(X,Y), q(Z,Y), X \neq Z.$$
$$(c_2) \quad :\text{--} \, q(X1,Y1), q(X2,Y2), X2 = X1 + K, Y2 = Y1 + K, K > 0.$$
$$(c_3) \quad :\text{--} \, q(X1,Y1), q(X2,Y2), X2 = X1 + K, Y1 = Y2 + K, K > 0.$$

We represent queens with atoms of the form $q(X,Y)$. $q(X,Y)$ is true if a queen is placed in the chess board at row X and column Y. The disjunctive rule guesses the position of the queens; in particular, for each row X, we guess the column where the queen has to be placed. Then the constraints assert that two queens cannot stay in the same column (constraint c_1) and in the same diagonal (from top left to bottom right (constraint c_2) and from top right to bottom left (constraint c_3)).

5   Computational Complexity of $\text{DLP}^{\vee, \wedge}$

As for the classical nonmonotonic formalisms [10], two important decision problems, corresponding to two different reasoning tasks, arise in $\text{DLP}^{\vee, \wedge}$:

(Brave Reasoning) Given a $\text{DLP}^{\vee, \wedge}$ program $\mathcal{P}$ and a ground literal L, is L true in some answer set of $\mathcal{P}$?

(Cautious Reasoning) Given a $\text{DLP}^{\vee, \wedge}$ program $\mathcal{P}$ and a ground literal L, is L true in all answer sets of $\mathcal{P}$?

The following theorems report on the complexity of the above reasoning tasks for propositional (i.e., variable-free) $\text{DLP}^{\vee, \wedge}$ programs. Importantly, it turns out that reasoning in $\text{DLP}^{\vee, \wedge}$ does not bring an increase in computational complexity, which remains exactly the same as for standard DLP.

Theorem 1. *Brave Reasoning on ground* $\text{DLP}^{\vee, \wedge}$ *programs is* Σ_2^P*-complete.*

Theorem 2. *Cautious Reasoning on ground* $\text{DLP}^{\vee, \wedge}$ *programs is* Π_2^P*-complete.*

6   Conclusions

We have proposed $\text{DLP}^{\vee, \wedge}$, an extension of DLP by parametric connectives. This formalism enhances the knowledge modelling abilities of DLP allowing to represent in a natural and elegant way problems that cannot be simply encoded using standard DLP. Indeed, these connectives allow us to represent compactly the disjunction/conjunction of a set of atoms having a given property.

We have formally defined the semantics of the new language, and we have shown the usefulness of $\text{DLP}^{\vee, \wedge}$ on relevant knowledge-based problems.

Ongoing work concerns the implementation of parametric literals in the **DLV** system. In order to guarantee an efficient instantiation of parametric literals we impose a syntactic restriction on the domain predicates (i.e. on the predicates appearing in the conjunction on the right side of symbolic set). In particular, we require that such predicates are normal (disjunction-free) and stratified. Moreover, the instantiation process

takes care of instantiating the domain predicates before dealing with parametric literals where they occur. In this way, the parametric literals can be completely solved during the instantiation process and transformed into standard disjunctions or standard conjunctions. Further work concerns an experimentation activity devoted to the evaluation of the impact of parametric connectives on system efficiency. We believe that the conciseness of the encoding obtained through parametric literals in some cases, like for instance N-Coloring and N-Queens, should bring a positive gain on the efficiency of the evaluation.

Acknowledgments

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2002.
2. C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.
3. T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The DLV System. In J. Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC*, College Park, Maryland, June 1999. Computer Science Department, University of Maryland. Workshop Notes.
4. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.
5. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.
6. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
7. G. Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS '94)*, pages 23–42, Ithaca NY, 1994. MIT Press.
8. V. Lifschitz. Foundations of Logic Programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.
9. J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. The MIT Press, Cambridge, Massachusetts, 1992.
10. V. W. Marek and M. Truszczyński. Autoepistemic Logic. *Journal of the ACM*, 38(3):588–619, 1991.
11. J. Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.
12. B. Wolfinger, editor. Workshop: *Disjunctive Logic Programming and Disjunctive Databases*, Berlin, August 1994. German Society for Computer Science (GI), Springer. 13th IFIP World Computer Congress, Hamburg, Germany.