I N F S Y S
R ESEARCH
R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME
ABTEILUNG WISSENSBASIERTE SYSTEME

PRUNING OPERATORS FOR DISJUNCTIVE LOGIC PROGRAMMING SYSTEMS

Francesco Calimeri Wolfgang Faber
Nicola Leone Gerald Pfeifer

INFSYS RESEARCH REPORT 1843-01-07
MAY 2001; REVISED OCTOBER 2005

Institut für Informationssysteme
Abtg. Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TECHNISCHE UNIVERSITÄT WIEN

INFSYS RESEARCH REPORT
INFSYS RESEARCH REPORT 1843-01-07, MAY 2001; REVISED OCTOBER 2005

PRUNING OPERATORS FOR DISJUNCTIVE LOGIC PROGRAMMING SYSTEMS

Francesco Calimeri[1] Wolfgang Faber[2] Nicola Leone[3]
Gerald Pfeifer[4]

Abstract. Disjunctive Logic Programming (DLP) is an advanced formalism for knowledge representation and reasoning. The language of DLP is very expressive and supports the representation of problems of high computational complexity (specifically, all problems in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$). The DLP encoding of a large variety of problems is often very concise, simple, and elegant. In this paper, we explain the computational process commonly performed by DLP systems, with a focus on search space pruning, which is crucial for the efficiency of such systems. We present two suitable operators for pruning (*Fitting*'s and *Well-founded*), discuss their peculiarities and differences with respect to efficiency and effectiveness. We design an intelligent strategy for combining the two operators, exploiting the advantages of both. We implement our approach in **DLV** – the state-of-the-art DLP system – and perform some experiments. These experiments show interesting results, and evidence how the choice of the pruning operator affects the performance of DLP systems.

[1]Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy. E-mail: calimeri@mat.unical.it
[2]Institut für Informationssysteme, Technische Universität Wien, A-1040 Wien, Austria. E-mail: faber@kr.tuwien.ac.at
[3]Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy. E-mail: leone@mat.unical.it
[4]Institut für Informationssysteme, Technische Universität Wien, A-1040 Wien, Austria. E-mail: pfeifer@dbai.tuwien.ac.at

Acknowledgements: This work was supported by FWF (Austrian Science Funds) under the projects Z29-INF and P14781.

Copyright © 2005 by the authors

1 Introduction

Disjunctive Logic Programming (DLP) is a declarative approach to programming, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. Disjunctive Logic Programming (DLP, which under the stable model semantics is called also Answer Set Programming) is a logic programming alternative to SAT-based programming, which is successfully and widely used in the area of Artificial Intelligence [30]. In SAT-based programming, a given computational problem P is encoded as a propositional CNF formula the models of which correspond to solutions of P; a SAT solver is then used to find such models (and thus solutions of P). In Disjunctive Logic Programming, a given computational problem P is represented by a DLP program whose stable models correspond to solutions; a DLP system is then used to find such solutions to P [37].

The main advantage of Disjunctive Logic Programming over SAT-based programming is the higher expressiveness of its language, which enjoys the knowledge modeling power of logic programming features like variables, negation as failure, and disjunction. Indeed, the knowledge representation language of DLP consists of function-free logic programs where disjunction is allowed in the heads and nonmonotonic negation may occur in the bodies of the rules. The DLP language supports the representation of problems of high computational complexity (specifically, all problems in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$ [20]). Importantly, the DLP encoding of a large variety of problems is often very concise, simple, and elegant [19].

The intended models of a DLP program (i.e., the semantics of the program) are subset-minimal models which are "grounded" in a precise sense; they are called *stable models* [28].

As an example consider the well-known problem of 3-colorability, which is the assignment of three colors to the nodes of a graph such 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 graph given by F.

$$r_1: \ color(X, red) \vee color(X, yellow) \vee color(X, green) \ :- \ node(X).$$
$$r_2: \ :- \ edge(X, Y), \ color(X, C), \ color(Y, C).$$

Rule r_1 states that every node of the graph is colored red or yellow or green, while r_2 forbids the assignment of the same color to any adjacent nodes. The minimality of stable models 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 stable models of $F \cup \{r_1, r_2\}$. The graph is 3-colorable if and only if $F \cup \{r_1, r_2\}$ has some stable model.

The high expressiveness of Disjunctive Logic Programming comes at the price of a high computational cost in the worst case, which makes the implementation of efficient DLP systems a difficult task.

The core of a DLP system is model generation, where a model of the program is produced, which is then subjected to a model check. For the generation of models, DLP systems typically employ procedures which are similar to Davis-Putnam procedures used in SAT solvers. As for SAT solvers, two factors are fundamentally important for the efficiency of model generation in DLP: (i) the *heuristic* (branching rule) for the selection of the branching literal, i.e., the criterion determining the literal to be assumed

true at a given stage of the computation; and (ii) the *pruning operator*, i.e., the operator computing the consequences deterministically derivable from the program rules and the interpretation at hand, which enlarges the set of known facts pruning the search space.

Several efforts have been made in the direction of implementing efficient DLP systems. After some pioneering work on stable model computation [5, 49], a number of modern DLP systems are now available. The most widespread DLP systems are **DLV** [32], `GnT` [29], and recently also Cmodels-3 [34]. Many other systems support various fragments of the DLP language [1, 3, 4, 10, 11, 12, 16, 17, 15, 18, 35, 38, 39, 43, 45, 46, 48].

Nevertheless, much work has to be done to make DLP systems fully satisfactory for modern knowledge-based applications. The design of new optimization techniques and smart algorithms for the computation of DLP programs is of utmost importance. The present paper goes in this direction, focusing on search space pruning, an extremely critical problem for the performance of DLP systems. The main contributions of the paper are as follows:

- We describe the main steps of the computational process performed by DLP systems with a focus on search space pruning, which is crucial for efficiency. We analyze the properties of the disjunctive extensions of two well-known pruning operators for logic programming, *Fitting*'s operator and the *Well-founded* operator. We carry out an in-depth discussion on their strengths and weaknesses w.r.t. efficiency and effectiveness, deriving new properties on these operators which are fundamental for their concrete exploitation in DLP systems.

- We design an intelligent strategy for combining these two pruning operators, which exploits the advantages of both, starting from several known results established in previous works and focusing on modularity properties [36, 20, 33], head-cycle free programs [6], acyclic programs [25], disjunctive unfounded sets and complexity [33], combining these in a smart way.

- We implement our approach in the DLP system **DLV**, taking care of efficiency issues and respecting the known complexity bounds. Indeed, the fixpoints of Fitting's operator are computed in linear time, as are the Greatest Unfounded Sets (which contribute the negative inferences in the Well-founded operator).

- We report experimental results on a number of benchmark problems to assess the impact of our approach. The experiments show that the choice of the pruning operator has a strong influence on the performance of DLP systems, and specifically that our techniques considerably improve the efficiency of the **DLV** system.

To our knowledge, this is the first paper that focuses on pruning the search space for *disjunctive* DLP programs. A number of related works have studied pruning operators in the domain of non-disjunctive programs. The use of the Well-founded operator in the computation of the stable models of non-disjunctive programs has been first proposed in [31] while its concrete implementation in a system was first done in [49]. The Well-founded operator corresponds to the *upper closure* operator (*AtMost*) of the Smodels system [41] on non-disjunctive programs. It has been implemented very efficiently in Smodels, employing a novel optimization technique to localize the computation [47]. Strictly related, in [8] transformation

techniques for non-disjunctive programs have been proposed, for certain combinations of which correspondence to Fitting's and Well-founded operators has been established.

The paper is organized as follows. In Section 2, we introduce Disjunctive Logic Programming and define its syntax and semantics based on the concept of stable model; we then illustrate the usage of DLP for knowledge representation and reasoning by showing a couple of examples. In Section 3, we describe the main procedure for the computation of the stable models semantics. In Section 4, we present two pruning operators for DLP: *Fitting*'s ($\Phi_{\mathcal{P}}$) operator, and the *Well-founded* ($\mathcal{W}_{\mathcal{P}}$) operator. In Section 5, we analyze several interesting properties of these pruning operators on some syntactically restricted classes of DLP programs. In Section 6, we design our new method for the intelligent combination of the pruning operators for DLP, and discuss some key issues for its implementation in **DLV**. In Section 7, we report the results of our experimentation activity on a number of benchmark problems. Finally, Section 8 draws our conclusions. The two appendices provide further details on the experiments and problem encodings.

2 Disjunctive Logic Programming

In this section, we first provide a brief introduction to the syntax and semantics of Disjunctive Logic Programming; then, we show some examples on the usage of DLP for knowledge representation. For further background see [19, 28]. In addition, see [2, 13] for comprehensive surveys on the semantics of disjunction and negation in logic programming.

2.1 Syntax and Notation

Following the convention of Prolog, strings starting with uppercase letters denote variables, while those starting with lower case letters denote constants. A *term* is either a variable or a constant.

An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms.

A literal l is either an atom p (*positive* literal) or its negation $\texttt{not}\ p$ (*negative* literal). Two literals are said to be complementary if they are of the form p and $\texttt{not}\ p$ for some atom p. Given a literal l, $\texttt{not}.l$ denotes its complementary literal. Accordingly, given a set L of literals, $\texttt{not}.L$ denotes the set $\{\texttt{not}.L \mid\ L \in A\}$. A set L of literals is said to be *consistent* if, for every literal $l \in L$, its complementary literal is not contained in L.

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1\ \vee\ \cdots\ \vee\ a_n\ :-\ b_1, \cdots, b_k,\ \texttt{not}\ b_{k+1}, \cdots,\ \texttt{not}\ b_m. \tag{1}$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 1$, $m \geq k \geq 0$. The disjunction $a_1\ \vee\ \cdots\ \vee\ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k,\ \texttt{not}\ b_{k+1}, ...,\ \texttt{not}\ b_m$ is the *body* of r. A rule having precisely one head literal (i.e. $n = 1$) is called a *normal rule*. If the body is empty (i.e. $k = m = 0$), it is called a fact, and the " $:-$ " sign is omitted.

The following notation will be useful for further discussion. If r is a rule of form (1), then $H(r) = \{a_1, \ldots, a_n\}$ is the set of the literals in the head and $B(r) = B^+(r) \cup B^-(r)$ is the set of the body literals, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{\texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m\}$.

A *disjunctive logic program* $\mathcal{P}$ is a finite set of rules. A not-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^-(r) = \emptyset$) is called *positive*, and a $\vee$-free program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : |H(r)| \leq 1$) is called *normal logic program*. A disjunctive logic program is also called *disjunctive datalog program* and *disjunctive deductive database*. A normal logic program is also called *datalog program* and *deductive database*. A program is *ground*, if it does not contain variables, and *propositional*, if all predicate arities are 0.

Sometimes, we simply refer to programs, if they are not restricted to be positive, normal or ground.

2.2 Semantics

The generally accepted semantics of DLP is the (consistent) stable models semantics, originally defined in [28].

Herbrand Universe. For any program $\mathcal{P}$, let $U_\mathcal{P}$ (the Herbrand Universe) be the set of all constants appearing in $\mathcal{P}$. In case no constant appears in $\mathcal{P}$, an arbitrary constant ψ is added to $U_\mathcal{P}$.

Herbrand Literal Base. For any program $\mathcal{P}$, let $B_\mathcal{P}$ be the set of all ground atoms constructible from the predicate symbols appearing in $\mathcal{P}$ and the constants of $U_\mathcal{P}$.

Ground Instantiation. For any rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_\mathcal{P}$. For any program $\mathcal{P}$, $Ground(\mathcal{P})$ denotes the following set $Ground(\mathcal{P}) = \bigcup_{r \in Rules(\mathcal{P})} Ground(r)$. For propositional (i.e., variable-free) programs, $\mathcal{P} = Ground(\mathcal{P})$ holds.

Stable Models. For every program $\mathcal{P}$, we define its stable models using its ground instantiation $Ground(\mathcal{P})$ in two steps: First we define the stable models of positive disjunctive programs, then we give a reduction of disjunctive programs containing negation as failure to positive ones and use it to define stable models of arbitrary disjunctive programs, possibly containing negation as failure.

A (partial) *interpretation* I for a program $\mathcal{P}$ is a consistent set of ground literals, i.e. $I \subseteq B_\mathcal{P} \cup \texttt{not}.B_\mathcal{P}$ such that $I \cap \texttt{not}.I = \emptyset$.[1] A ground literal l is *true* (resp. *false*) w.r.t. I if $l \in I$ (resp. $\texttt{not}.l \in I$); l is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. An interpretation I is *total* if each atom in $B_\mathcal{P}$ is either true or false w.r.t. I (i.e., no literal is undefined w.r.t. I). We denote by I^+ and I^-, respectively, the set of positive literals and the set of negative literals occurring in I.

A total interpretation X is called a *model* for $\mathcal{P}$ if, for every $r \in Ground(\mathcal{P})$, $H(r) \cap X \neq \emptyset$ whenever $B(r) \subseteq X$. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N^+ is a proper subset of M^+. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$. For a positive DLP program $\mathcal{P}$, a total interpretation X is a *stable model* if $X \in \mathrm{MM}(\mathcal{P})$.

[1] Even if the stable models are *total* interpretations, we provide the more general notion of *partial* interpretation, because the computation of stable models, described in the next sections, relies heavily on partial interpretations.

Example 1 [2] *For the positive program $\mathcal{P}_1 = \{a \vee b.\}$, the (total) interpretations $\{a,\ \texttt{not}\ b\}$ and $\{b,\ \texttt{not}\ a\}$ are its minimal models (i.e., $\mathrm{MM}(\mathcal{P}_1) = \{\ \{a,\ \texttt{not}\ b\},\ \{b,\ \texttt{not}\ a\}\ \}$), and they are therefore the stable models of $\mathcal{P}_1$.*

For the program $\mathcal{P}_2 = \{a \vee b.,\ b :- a.,\ a :- b.\}$, $\{a, b\}$ is the only stable model.

Program $\mathcal{P}_3 = \{a \vee b.,\ b :- a.\}$ has only the stable model $\{b\}$.

Definition 1 *The* reduct *or* Gelfond-Lifschitz transform *of a ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by*

- *deleting all rules $r \in \mathcal{P}$ for which $B^-(r) \cap \texttt{not}.X \neq \emptyset$ holds (i.e., a negative body literal is false);*
- *deleting the negative body from the remaining rules.*

A stable model of a (general DLP) program $\mathcal{P}$ is a total interpretation X of $\mathcal{P}$ such that X is a stable model of $Ground(\mathcal{P})^X$.

Example 2 *Let $\mathcal{P}_4 = \{a \vee b :- c.,\ b :- \texttt{not}\ a, \texttt{not}\ c.\ a \vee c :- \texttt{not}\ b.\}$. Consider $I = \{b,\ \texttt{not}\ a,\ \texttt{not}\ c\}$. Then, $\mathcal{P}_4^I = \{a \vee b :- c.,\ b.\}$. It is easy to verify that I is a minimal model for $\mathcal{P}_4^I$; thus, I is a stable model for P.*

2.3 Knowledge Representation and Disjunctive Logic Programming

In this section we illustrate how a DLP language can be used for knowledge representation. Encoding problems can be performed in a highly declarative fashion, following a "guess&check" paradigm, as described in [19]. We will recall this technique and will then illustrate how to apply it on a couple of examples.

Many problems, also problems of comparatively high computational complexity (Σ_2^P-complete), can be solved in a natural manner exploiting a DLP system by using this declarative programming technique. The power of disjunctive rules allows for expressing problems which are 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.

Usually, Given a set $\mathcal{F}_I$ of facts that specify an instance I of some problem $\mathbf{P}$, a guess&check program $\mathcal{P}$ for $\mathbf{P}$ consists of the following main parts:

Guessing Part The guessing part $\mathcal{G} \subseteq \mathcal{P}$ of the program defines the search space, in a way such that stable models 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 stable models of $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}_I$ represent the solutions for the problem instance I.

[2] For simplicity, we often use propositional examples, in which the programs coincide with their ground instantiations.

Usually, $\mathcal{G}$ consists of some disjunctive rule, and $\mathcal{C}$ contains some integrity constraints (plus the definition of possible auxiliary predicates). However, in general, both $\mathcal{G}$ and $\mathcal{C}$ might 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); in the extremal case, $\mathcal{G}$ could coincide with the full program and $\mathcal{C}$ could be empty, i.e., all checking is integrated into the guessing part such that solution candidates are always solutions.

It is worth stating that DLP (which is not limited to guess&check) facilitates the formulation of problems arising in many domains, such as planning and combinatorics. Also various forms of abductive reasoning can be expressed, as we will briefly describe in the sequel.

Hamiltonian Cycle

As an example which matches the guess&check scheme, let us consider *Hamiltonian Cycle*, a classical NP-complete problem.

HAMCYCLE: Given an undirected graph $G = (V, E)$, where V is the set of vertices of G, and E is the set of edges, and a node $a \in V$ of this graph, does there exist a cycle of G containing a and passing through each node in V exactly once?

Assuming that the graph G is specified by means of predicates $vertex$ (unary) and arc (binary). Please note that predicate arc is symmetric, since undirected edges are bidirectional directed arcs. The starting node is specified by the predicate $start$ (unary). The following program $\mathcal{P}_{HC}$ solves the problem HAMCYCLE:[3]

$$
\left.\begin{array}{l}
inCycle(X,Y) \vee \ outCycle(X,Y) \ :- \quad start(X), arc(X,Y). \\
inCycle(X,Y) \vee \ outCycle(X,Y) \ :- \quad onCycle(X), arc(X,Y).
\end{array}\right\} \textbf{Guess}
$$

$$
\left.\begin{array}{l}
onCycle(Y) \ :- \quad inCycle(_, Y). \\
\ \% \ At \ most \ one \ ingoing/outgoing \ arc! \\
\ :- \quad inCycle(X,Y), inCycle(X,Y1), Y <> Y1. \\
\ :- \quad inCycle(X,Y), inCycle(X1,Y), X <> X1. \\
\ \% \ Each \ node \ has \ to \ be \ on \ the \ cycle. \\
\ :- \quad vertex(X), \texttt{not} \ onCycle(X).
\end{array}\right\} \textbf{Check}
$$

The guessing part (first two rules) guess a subset of all given arcs, while the rest of the program checks whether it is a Hamiltonian Cycle. The first two constraints in checking part ensure that in the set of arcs S selected by `inCycle` there are not two arcs that start at the same node or end in the same node. The third constraint enforces that all nodes are reached from the starting node in the subgraph induced by S, and ensures that this subgraph is connected (since it is a cycle) through the auxiliary predicate `onCycle` defined by the third rule. Thus, given a set of facts F for $vertex$, arc, and $start$ which specify the problem input, the program $\mathcal{P}_{HC} \cup F$ has a stable model if and only if the input graph has a Hamiltonian Cycle.

[3] In the examples we use also integrity constraints (rules without head). An integrity constraint $:- \ b_1, \ldots, b_m, \texttt{not} \ b_{m+1}, \ldots, \texttt{not} \ b_n.$ is a shorthand for the rule $bad :- \ b_1, \ldots, b_m, \texttt{not} \ b_{m+1}, \ldots, \texttt{not} \ b_n, \texttt{not} \ bad.$, where bad is a *fresh* atom not occurring elsewhere in the program. Intuitively, it forbids that $b_1, \ldots, b_m, \texttt{not} \ b_{m+1}, \ldots, \texttt{not} \ b_n$ are simultaneously true.

Ramsey Numbers

We have seen how a search problem can be encoded in a **DLV** program whose stable models correspond to the problem solutions. We next see another use of the guess&check programming technique. We build a DLP program whose stable models witness that a property does not hold, i.e., the property at hand holds if and only if the program has no stable models. Such a programming scheme is useful to decide co-NPor Π_2^P properties. We next apply the above programming scheme to a well-known problem of number and graph theory.

RAMSEY: The Ramsey number $R(k, m)$ is the least integer n such that, no matter how we color the arcs of the complete graph (clique) with n nodes using two colors, say red and blue, there is a red clique with k nodes (a red k-clique) or a blue clique with m nodes (a blue m-clique).

Ramsey numbers exist for all pairs of positive integers k and m [44]. We next show a program which allows us to determine whether a given integer n is the Ramsey Number $R(3, 4)$, knowing that no integer smaller than n is $R(3, 4)$. Let $\mathcal{F}$ be the collection of facts for input predicates *node* and *arc* encoding a complete graph with n nodes. Consider the following guess&check program $\mathcal{P}_{ramsey}$.

$$blue(X,Y) \vee red(X,Y) \ :- \quad arc(X,Y). \qquad \} \ \texttt{Guess}$$

$$\left.\begin{array}{ll} :- & red(X,Y),\ red(X,Z),\ red(Y,Z). \\ :- & blue(X,Y),\ blue(X,Z),\ blue(Y,Z), \\ & blue(X,W),\ blue(Y,W),\ blue(Z,W). \end{array}\right\} \ \texttt{Check}$$

Intuitively, the disjunctive rule guesses a color for each edge. The first constraint eliminates the colorings containing a red complete graph (i.e., a clique) on 3 nodes, and the second constraint eliminates the colorings containing a blue clique on 4 nodes. The program $\mathcal{P}_{ramsey} \cup \mathcal{F}$ has a stable model if and only if there is a coloring of the edges of the complete graph on n nodes containing no red clique of size 3 and no blue clique of size 4. Thus, if there is a stable model for a particular n, then n is <u>not</u> $R(3, 4)$, that is, $n < R(3, 4)$. The smallest n such that no stable model is found is the Ramsey number $R(3, 4)$.

3 Stable Models Computation

In this section, we describe the main steps of the computational process performed by DLP systems. We will describe the computational engine of the **DLV** system [23, 24] which will be used for the experiments, but also other systems like Smodels [42, 47] employ a very similar procedure.

In general, a disjunctive logic program $\mathcal{P}$ contains variables. The first computational step of a DLP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$ which is a (usually much smaller) subset of all syntactically constructible instances of the rules of $\mathcal{P}$ having precisely the same stable models as $\mathcal{P}$ [23].

The hard part of the computation is then performed on this ground program by the Model Generator, which is sketched in Figure 1. For brevity, $\mathcal{P}$ refers to the (simplified) ground program in the sequel.

```
Function ModelGenerator(var I: Interpretation): Boolean;
var  inconsistency: Boolean;
begin
    DetCons(I,inconsistency);
    if inconsistency then return false;
    if "no atom is undefined in I" then return IsStableModel(I);
    Select an undefined ground atom A according to a heuristic;
    if ModelGenerator(I ∪ {A}) then return true;
    else return ModelGenerator(I ∪ {not A});
end;
```

Figure 1: Computation of Stable Models

Roughly, the Model Generator produces some "candidate" stable models. Each candidate I is then verified by the function *IsStableModel(I)*, which checks whether I is a minimal model of the program $\mathcal{P}^I$ obtained by applying the GL-transformation w.r.t. I.

Initially, the ModelGenerator function is invoked with I set to the empty interpretation (all atoms are undefined at this stage). If the program $\mathcal{P}$ has a stable model, then the function returns true and sets I to the computed stable model; otherwise it returns false. The Model Generator is similar to the Davis-Putnam procedure in SAT solvers. It first calls a function DetCons, which extends I with those literals that can be deterministically inferred. This is similar to unit propagation as employed by SAT solvers, but exploits the peculiarities of DLP for making further inferences (e.g., it uses the knowledge that every stable model is a minimal model).

If DetCons does not detect any inconsistency, an atom A is selected according to a heuristic criterion and ModelGenerator is recursively called on both $I \cup \{A\}$ and $I \cup \{\texttt{not}\ A\}$. The atom A corresponds to a branching variable of a SAT solver.

For the performance of a DLP system the implementation of DetCons is crucial in two ways: It has to perform its task as quickly as possible, while pruning the search space as much as possible.

4 Pruning Operators

In this section we review two operators that are useful to implement DetCons (see the previous section). As already mentioned, DetCons has to expand a given interpretation as much as possible to reduce the search space, while ensuring that such an expansion never causes any stable model to be missed. In other words, if an interpretation I is contained in a stable model M, that stable model will also contain the expansion of I computed by DetCons. We can state this "safety" property formally.

Definition 2 *A generic operator $\Gamma_{\mathcal{P}}$ is said to be "safe" if, for each interpretation I, and for each stable model M of a given program $\mathcal{P}$, we have $I \subseteq M$ iff $\Gamma_{\mathcal{P}}(I) \subseteq M$.*

The two operators in question are the *Fitting*'s ($\Phi_{\mathcal{P}}$) operator and the *Well-founded* ($\mathcal{W}_{\mathcal{P}}$) operator. Both have the property described above, and extend the two corresponding operators defined for disjunction-

free programs [26, 51] to the class of disjunctive logic programs. Both $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ consist of two parts: The part drawing positive inferences (which is an extension of the immediate consequence operator $T_{\mathcal{P}}$, defined for three-valued interpretations of normal logic programs [51], to disjunctive programs) is the same for both operators; they only differ in the way they perform negative inferences.

Definition 3 *Let $\mathcal{P}$ be a program, and I be an interpretation.*
$\mathcal{T}_{\mathcal{P}}(I) = \{a \mid \exists r \in \mathcal{P}\ s.t.\ a \in H(r) :\ H(r) - \{a\} \subseteq \texttt{not}.I \wedge B(r) \subseteq I\}$.

Thus, $\mathcal{T}_{\mathcal{P}}(I)$ derives an atom a from a rule r, if the body of r is true w.r.t. I and, apart from a, all other atoms in the head of r are false w.r.t. I. Note that, in order to be a model, any interpretation extending I must necessarily contain a, otherwise rule r is violated.

Example 3 *Consider the following program $\mathcal{P}_1$:*

$$\{a \vee b.,\ c :-\ \texttt{not}\ a.,\ d :-\ e.,\ e :-\ d.,\ k :-\ \texttt{not}\ e.\}$$

Suppose I = $\{\texttt{not}\ a\}$: Then b is derived via the first rule, and c via the second (whose body is contained in I), so $\mathcal{T}_{\mathcal{P}}(I)$ = $\{b, c\}$.

Intuitively, given an interpretation I, $\mathcal{T}_{\mathcal{P}}$ derives a set of atoms that are strictly needed to extend I to a model. Note that $\mathcal{T}_{\mathcal{P}}$ is deterministic, that is, its result is a single set of literals.

4.1 *Fitting*'s ($\Phi_{\mathcal{P}}$) Operator

We extend *Fitting*'s operator, which was originally defined in [26], to the disjunctive case. The way this operator makes negative inferences is described and then combined with the $\mathcal{T}_{\mathcal{P}}$ operator.

Definition 4 *Let $\mathcal{P}$ be a program, and I an interpretation.*

$$\gamma_{\mathcal{P}}(I) \ = \ \{a \in B_{\mathcal{P}} \mid \ \forall r \in ground(\mathcal{P})\ s.t.\ a \in H(r) :\ H(r) - \{a\}\ is\ true\ w.r.t.\ I, \\ or\ B(r)\ is\ false\ w.r.t.\ I\}.$$

Thus, $\gamma_{\mathcal{P}}(I)$ derives an atom a, if each rule with a in the head already has a false body or a true head (the head being true by an atom different from a). Note that all such a rules with a in the head are satisfied in I, and they remain satisfied in all extensions of I even if a is set to false.

Example 4 *Consider the program $\mathcal{P}_1$ of Example 3, and the interpretation I = $\{a\}$. Here $\gamma_{\mathcal{P}}(I) = \{b, c\}$.*

Intuitively, given an interpretation I, $\gamma_{\mathcal{P}}$ computes those atoms that will not appear in any minimal model extending I since there is no rule left that could be used to derive them (and "support" its introduction in the model).

We can now define a single step of Fitting's operator $\Phi_{\mathcal{P}}$ and its least fixpoint as follows:

Definition 5 *Let $\mathcal{P}$ be a program, and I an interpretation.*

$$\Phi_{\mathcal{P}}(I) = \mathcal{T}_{\mathcal{P}}(I) \cup \texttt{not}.\gamma_{\mathcal{P}}(I).$$

Starting from I we define the following sequence F_k:

$$F_0 = I$$
$$F_k = F_{k-1} \cup \Phi_{\mathcal{P}}(F_{k-1}), \quad k > 0.$$

We now have a growing sequence whose n-th *term is the* n-fold *application of $\Phi_{\mathcal{P}}$ to I, and define the least fixpoint $\Phi^{\infty}_{\mathcal{P}}(I)$ of $\Phi_{\mathcal{P}}$ containing I, as the limit to which $\{F_n\}_{n\in\mathcal{N}}$ converges.*

Example 5 *Consider the program $\mathcal{P}_1$ of Example 3, and the interpretation $I = \{a\}$. It is easy to see that $\Phi_{\mathcal{P}}(I) = \emptyset \cup \texttt{not}.\{b, c\} = \{\texttt{not}\ b, \texttt{not}\ c\}$.*
We thus obtain:

$$F_0 = I = \{a\}.$$
$$F_1 = F_0 \cup \{\texttt{not}\ b, \texttt{not}\ c\} = \{a, \texttt{not}\ b, \texttt{not}\ c\}.$$
$$F_2 = F_1 \cup \emptyset = \{a, \texttt{not}\ b, \texttt{not}\ c\} = F_1 = \Phi^{\infty}_{\mathcal{P}}(I)$$

Next we assert the already discussed "safety" property for $\Phi_{\mathcal{P}}$.

Theorem 1 *For every stable model M of a given program $\mathcal{P}$,* if *an interpretation I is contained in M,* then $\Phi^{\infty}_{\mathcal{P}}(I) \subseteq M$.

Proof 1 *It is enough to show that $\Phi_{\mathcal{P}}(I) \subseteq M$ for any $I \subseteq M$; indeed, if this is the case, we can take $I \cup \Phi_{\mathcal{P}}(I)$ as another interpretation contained in M, and then iteratively go on until the fixpoint is reached. We first show that the positive part of $\Phi_{\mathcal{P}}(I)$ is contained in M; then, we show that also its negative part is contained in M.*

By definition, the positive part of $\Phi_{\mathcal{P}}(I)$ is $\mathcal{T}_{\mathcal{P}}(I)$. Let's take a positive *atom $a \in \Phi_{\mathcal{P}}(I)$. Thus, there must exist a rule $r \in \mathcal{P}$ such that $a \in H(r)$ with $H(r) - \{a\} \subseteq \texttt{not}.I$ and $B(r) \subseteq I$, i.e., the rest of the head of r is false while its body is true w.r.t. I. But since $I \subseteq M$, this means that the body of r is true w.r.t. M; since M is a stable model, we must have $a \in M$, otherwise r would be violated w.r.t. M, and M would not be a (stable) model.*

The negative part of $\Phi_{\mathcal{P}}(I)$, again by definition, is $\gamma_{\mathcal{P}}(I)$. Let b be an atom in $\gamma_{\mathcal{P}}(I)$. Suppose, by contradiction, that not *$b \notin M$; then $b \in M$, as M is a total interpretation. Since $b \in \gamma_{\mathcal{P}}(I)$, for each rule having b in the head, we have that either $H(r) - \{b\}$ is true w.r.t. I, or $B(r)$ is false w.r.t. I; since $I \subseteq M$, this means that all such rules are already satisfied (either by a true head or a false body) also w.r.t. M, and cannot give "support" to b. Consequently, M contains atom b which is not supported, contradicting the well-known "supportedness" property of stable models (see, e.g., [27]).*

Importantly, we have the following.

Proposition 1 *Given a propositional program $\mathcal{P}$ and an interpretation I for it, $\Phi^{\infty}_{\mathcal{P}}(I)$ is* linear-time computable.

Proof 2 *$\Phi^{\infty}_{\mathcal{P}}(I)$ is well-known to be linear-time computable for a non-disjunctive $\mathcal{P}$. This result was stated in [7], where it is attributed to "folklore".*

It is easy to see that this result carries over to the disjunctive case by using a suitable data structure for identifying atoms, which are derived by $\gamma_{\mathcal{P}}(I)$, in constant time. One way of achieving this is to keep a counter of potentially supporting rules for each atom b (i.e., rules having b in the head such that the body is not false nor the head is made true by an atom different from b) – whenever such a counter becomes zero, atom b is derived false by $\gamma_{\mathcal{P}}(I)$.

Thus, the $\Phi_{\mathcal{P}}$ operator seems to be a good choice as a pruning operator: it is "*safe*" (Theorem 1), has the capability to perform negative inferences (Definition 5), and its fixpoint $\Phi^{\infty}_{\mathcal{P}}(I)$ is efficiently computable (Proposition 1).

Unfortunately, $\Phi_{\mathcal{P}}$ fails to derive all possible negative consequences. For instance, in Example 5 it fails to derive d and e as false w.r.t. I while the only rules having these atoms in the head will never have a true body. The *Well-founded* operator presented in the following section is "stronger" in this respect.

4.2 *Well-founded* ($\mathcal{W}_{\mathcal{P}}$) Operator

The $\mathcal{W}_{\mathcal{P}}$ operator defined in [33] extends the operator defined in [51] (whose least fixpoint is the Well-founded model) to the disjunctive case. It is defined by an extension of the notion of *unfounded sets* to disjunctive logic programs.

Definition 6 *Let I be an interpretation for a program $\mathcal{P}$. A set $X \subseteq B_{\mathcal{P}}$ of ground atoms is an* unfounded set *for $\mathcal{P}$ w.r.t. I if, for each $a \in X$ and for each rule $r \in ground(\mathcal{P})$ such that $a \in H(r)$, at least one of the following conditions holds:*

1. *$B(r) \cap \texttt{not}.I \neq \emptyset$, that is, the body of r is false w.r.t. I.*
2. *$B^+(r) \cap X \neq \emptyset$, that is, some positive body literal belongs to X.*
3. *$(H(r) - X) \cap I \neq \emptyset$, that is, an atom in the head of r, distinct from a and other elements in X, is true w.r.t. I.*

Example 6 *Considering the program $\mathcal{P}_1$ of Example 3 and the interpretation $I = \{a\}$, we get $GUS_{\mathcal{P}}(I) = \{b, c, d, e\}$. b is added because of the third condition, and c because of the first in Definition 6. Then d and e appear in the head of only a single rule each and for both the second condition of Definition 6 holds. We obtain $\mathcal{W}_{\mathcal{P}}(I) = \{\texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e\}$ and $W_0 = \{a\}, W_1 = \{a, \texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e\}, W_2 = \{a, \texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e, k\}, W_3 = W_2 = \mathcal{W}^{\infty}_{\mathcal{P}}(I)$.*

While for non-disjunctive programs the union of unfounded sets is again an unfounded set for all interpretations, this does not hold, in general, for disjunctive programs.

Example 7 *Given $\mathcal{P} = \{a \vee b\}$ and $I = \{a, b\}$, both $\{a\}$ and $\{b\}$ are unfounded sets w.r.t. I; but their union $\{a, b\}$ is not.*

We thus denote by $\mathbf{I}_{\mathcal{P}}$ the set of all interpretations of $\mathcal{P}$ for which the union of all unfounded sets for $\mathcal{P}$ w.r.t. I is an unfounded set for $\mathcal{P}$ w.r.t. I as well. In analogy with traditional logic programming, given $I \in \mathbf{I}_{\mathcal{P}}$, we call the union of all unfounded sets for $\mathcal{P}$ w.r.t. I the *greatest unfounded set* of $\mathcal{P}$ w.r.t. I, and denote it by $GUS_{\mathcal{P}}(I)$.

Because the existence of the *greatest unfounded set* is not guaranteed in general, the question of how to decide whether an interpretation I is in $\mathbf{I}_{\mathcal{P}}$ naturally comes up (also from the viewpoint of complexity, which we will deal with later on). There is a class of interpretations, called *unfounded-free interpretations*, which always have the greatest unfounded set.

Definition 7 *Let I be an interpretation for a program $\mathcal{P}$. I is* unfounded-free *if $I \cap X = \emptyset$ for each unfounded set X for $\mathcal{P}$ w.r.t. I.*

Unfounded-free interpretations have a nice semantic property, that, as we will see in the next sections, has also a practical impact on the computation.

Proposition 2 *[33] Let I be an unfounded-free interpretation for a program $\mathcal{P}$. Then*

$\mathcal{P}$ has the greatest unfounded set $GUS_{\mathcal{P}}(I)$ (i.e., $I \in \mathbf{I}_{\mathcal{P}}$).

Thus, an unfounded-free interpretation always admits the greatest unfounded set, and this set is efficiently computable. We can now introduce the *Well-founded* operator.

Definition 8 *Let $\mathcal{P}$ be a program, and $I \in \mathbf{I}_{\mathcal{P}}$ be an* unfounded-free *interpretation. We define the $\mathcal{W}_{\mathcal{P}}$ operator as follows:*

$$\mathcal{W}_{\mathcal{P}}(I) \;=\; \mathcal{T}_{\mathcal{P}}(I) \cup \texttt{not}.GUS_{\mathcal{P}}(I).$$

This definition extends the $\mathcal{W}_{\mathcal{P}}$ operator defined in [51] (whose least fixpoint is the Well-founded model) to the disjunctive case. Note that, since $\mathcal{W}_{\mathcal{P}}$ is defined on the domain $\mathbf{I}_{\mathcal{P}}$, each fixpoint of $\mathcal{W}_{\mathcal{P}}$ by definition admits the greatest unfounded set (since each fixpoint of $\mathcal{W}_{\mathcal{P}}$ must belong to the domain $\mathbf{I}_{\mathcal{P}}$ of $\mathcal{W}_{\mathcal{P}}$).

Example 8 *Consider the program $\mathcal{P}_1$ of Example 3, and the interpretation I = $\{a\}$. Then, $\mathcal{W}_{\mathcal{P}}(I) = \emptyset \cup \texttt{not}.\{b, c, d, e\} = \{\texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e\}$.*

Observe that, as Example 8, clearly shows, thanks to $GUS_{\mathcal{P}}$, $\mathcal{W}_{\mathcal{P}}$ derives more negative information than $\Phi_{\mathcal{P}}$, and therefore ensures a better pruning of the search space.

Let us now define the least fixpoint $\mathcal{W}_{\mathcal{P}}^{\infty}$ of the $\mathcal{W}_{\mathcal{P}}$ operator.

Definition 9 *Given a program $\mathcal{P}$ and an interpretation I, we define the following sequence W_k:*

$$W_0 = I$$
$$W_k = W_{k-1} \cup \mathcal{W}_{\mathcal{P}}(W_{k-1}), \quad k > 0$$

We have a growing sequence whose n-th *term is the* n-fold *application of $\mathcal{W}_{\mathcal{P}}$ to I, and define the least fixpoint $\mathcal{W}_{\mathcal{P}}^{\infty}(I)$ of $\mathcal{W}_{\mathcal{P}}$ containing I, as the limit to which $\{W_n\}_{n \in \mathcal{N}}$ converges.*

Example 9 *Again considering the program $\mathcal{P}_1$ from Example 3 and starting from the interpretation I = $\{a\}$, we have:*

$$W_0 = I = \{a\}.$$
$$W_1 = W_0 \cup \{\texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e\} = \{a, \texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e\}.$$
$$W_2 = W_1 \cup \{k\} = \{a, \texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e, k\}.$$
$$W_3 = W_2 \cup \emptyset = W_2 = \mathcal{W}_{\mathcal{P}}^{\infty}(I).$$

Next we state that the $\mathcal{W}_{\mathcal{P}}$ operator has the "*safety*" property previously discussed.

Proposition 3 [33] *Let I be an interpretation for a program $\mathcal{P}$, and let M be a stable model for $\mathcal{P}$. If $I \subseteq M$, then*

(a) I belongs to the domain $\mathbf{I}_{\mathcal{P}}$ of $\mathcal{W}_{\mathcal{P}}$, and

(b) $\mathcal{W}_{\mathcal{P}}(I) \subseteq M$.

The $\mathcal{W}_{\mathcal{P}}$ operator appears to be a good pruning operator: it is "safe", and performs more negative inferences than $\Phi_{\mathcal{P}}$. A negative point of $\mathcal{W}_{\mathcal{P}}$ is that it is applicable only on unfounded-free interpretations. According to the following proposition, we cannot efficiently test whether I is indeed unfounded-free (unless $\mathrm{P} = \mathrm{NP}$).

Proposition 4 [33] Let $\mathcal{P}$ be a propositional program and I be an interpretation for $\mathcal{P}$. Deciding whether I is unfounded-free is co-NP-complete.

5 Pruning Operators on Syntactically Restricted Classes of Programs

In this section, we explore several interesting properties of the pruning operators on some syntactically restricted classes of programs. To this end, we introduce *dependency graphs* which represent the dependencies of head predicates on the positive body predicates of rules.

Definition 10 *With every program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (\mathcal{N}, E)$, called the* dependency graph *of $\mathcal{P}$, where (i) each predicate of $\mathcal{P}$ is a node in $\mathcal{N}$, and (ii) there is an arc in E directed from node a to node b if there is a rule r in $\mathcal{P}$ such that two predicates a and b appear in $B^+(r)$ and $H(r)$, respectively.*

The dependency graph allows us to single out the recursive parts of the program, and split the program into subprograms having different properties.

Definition 11 *A* component *C of $DG_{\mathcal{P}}$ is a maximal strongly connected subset of nodes of $DG_{\mathcal{P}}$. The* subprogram *of C is the set of rules in $\mathcal{P}$ having a head predicate in C, denoted by $\mathcal{P}_C$. The set of all components of $DG_{\mathcal{P}}$ is denoted by $Comp(\mathcal{P})$.*

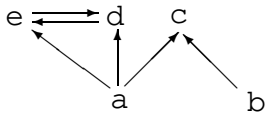

Figure 2: *The dependency graph $DG_{\mathcal{P}_1}$ of program $\mathcal{P}_1$*

Since there is a one-to-one correspondence between nodes in $DG_{\mathcal{P}}$ and predicates in $\mathcal{P}$, we will often refer to *components of* $\mathcal{P}$ (meaning components of $DG_{\mathcal{P}}$), and identify $Comp(\mathcal{P})$ as the *components of* $\mathcal{P}$.

It is worthwhile noting that the same disjunctive rule may occur in the subprograms of two different components. For instance, the program consisting of the single rule $a \vee b.$, has two components: $C_1 = \{a\}$ and $C_2 = \{b\}$. Rule $a \vee b.$ belongs to both subprograms $\mathcal{P}_{C_1}$ and $\mathcal{P}_{C_2}$.

We are now in the position to define the concepts of (a)cyclicity and head-cycle freeness, which play a very important role in our computational strategy.

Definition 12 *Given a program $\mathcal{P}$ and its dependency graph $DG_{\mathcal{P}}$, we say that:*

- *a component C is* cyclic *if the related subprogram $\mathcal{P}_C$ of $\mathcal{P}$ contains at least one recursive rule (i.e., a rule r such that a head predicate and a positive body predicate of r are in C); C is* acyclic *if it is not cyclic.*

- *a component C is* head-cycle-free *(HCF) iff the related subprogram $\mathcal{P}_C$ of $\mathcal{P}$ contains no rule r such that two predicates occurring in the head of r belong to C.*

$DG_{\mathcal{P}}$ and $\mathcal{P}$ are cyclic *if there is* at least one cyclic *component, otherwise they are acyclic. They are HCF if* all *components are HCF.*

Observe that acyclicity obviously implies head-cycle freeness; while an HCF component might be cyclic (e.g., if it is not disjunctive) or acyclic.

Example 10 *Consider the following program $\mathcal{P}_1$:*

$$\{\ a \vee b.\ ,\ \ c\ :-\ a.\ ,\ \ c\ :-\ b.\ ,\ \ d \vee e\ :-\ a.\ ,\ \ d\ :-\ e.,\ \ e\ :-\ d, \texttt{not}\ b.\ \}$$

The dependency graph $DG_{\mathcal{P}_1}$ of $\mathcal{P}_1$ is depicted in Figure 2. There are four components: $\{a\}$, $\{b\}$, $\{c\}$, $\{d, e\}$. All of them are acyclic except for the last which is also the only non-HCF component, as the head of $d \vee e\ :-\ a.$ contains two predicates belonging to the same cycle. The whole graph, and thus the program, is cyclic but not HCF.

We next present a well-known theorem in DLP community about a class of programs for which the operators $\Phi^{\infty}_{\mathcal{P}}(I)$ and $\mathcal{W}^{\infty}_{\mathcal{P}}(I)$ are equivalent. The importance of this theorem will become clear in the next section.

Theorem 2 *Given a program $\mathcal{P}$ and an unfounded-free interpretation I, if $\mathcal{P}$ is* acyclic *then $\mathcal{W}_\mathcal{P}(I) = \Phi_\mathcal{P}(I)$.*

Proof 3 *Let I be an unfounded-free interpretation of an acyclic program $\mathcal{P}$. The positive parts of $\mathcal{W}_\mathcal{P}(I)$ and $\Phi_\mathcal{P}(I)$ coincide by definition, since both of them are obtained by $\mathcal{T}_\mathcal{P}(I)$ (see Definition 5 and Definition 8).*

We have to show that also $GUS_\mathcal{P}(I) = \gamma_\mathcal{P}(I)$ holds for acyclic programs. Let $A = \gamma_\mathcal{P}(I)$ and $B = GUS_\mathcal{P}(I)$. For each atom $a \in A$, the set $\{a\}$ is an unfounded set for $\mathcal{P}$ w.r.t. I, as, by Definition of $\gamma_\mathcal{P}$, all ground rules with a in the head satisfy Condition 1 or Condition 3 of Definition 6. Therefore, for each $a \in A$, we have that $\{a\}$ is contained in the Greatest Unfounded Set B, that is, $A \subseteq B$. On the other hand, we know that also B is an unfounded set for $\mathcal{P}$ w.r.t. I. Since $\mathcal{P}$ is acyclic, we obtain that Condition 2 is superfluous for the unfoundedness of B, the ground rules having an element from B in the head satisfy either Condition 1 or Condition 3 of Definition 6. Consequently, all elements from B belong to $\gamma_\mathcal{P}(I)$, that is, $B \subseteq A$.

Hence, we have that $A = B$, that is, $GUS_\mathcal{P}(I) = \gamma_\mathcal{P}(I)$ holds.

Thus, on the class of acyclic programs, one can conveniently use Fitting's pruning operator, which is efficiently computable and equivalent to $\mathcal{W}_\mathcal{P}$ on these programs. The Well-founded operator, however, has a stronger inference power than Fitting's in the general case (see, e.g., Example 8, and the subsequent observation). To be able to exploit the $\mathcal{W}_\mathcal{P}$ operator in practice, we have to: (1) be able to efficiently detect whether it is applicable or not (i.e., if the interpretation at hand is unfounded-free or not – in general a co-NP-complete task, cf. Proposition 4), and (2) provide a concrete method for computing $GUS_\mathcal{P}(I)$ efficiently. The $\mathcal{R}_{\mathcal{P},I}$ operator, defined next, will serve this purpose.

Definition 13 *Let $\mathcal{P}$ be a program and I an interpretation. Then we define an operator $\mathcal{R}_{\mathcal{P},I}$ as follows:*

$$\begin{array}{rccl} \mathcal{R}_{\mathcal{P},I}: & 2^{B_\mathcal{P}} & \rightarrow & 2^{B_\mathcal{P}} \\ & X & \mapsto & \{a \in X \mid \forall r \in ground(\mathcal{P}) \text{ with } a \in H(r), \\ & & & \quad B(r) \cap (\texttt{not}.I \cup X) \neq \emptyset \text{ or } (H(r) - \{a\}) \cap I \neq \emptyset\} \end{array}$$

Given a set $X \subseteq B_\mathcal{P}$, the sequence $R_0 = X$, $R_n = \mathcal{R}_{\mathcal{P},I}(R_{n-1})$ decreases monotonically and converges finitely to a limit that we denote by $\mathcal{R}^\omega_{\mathcal{P},I}(X)$.

We next prove a lemma, which was not known so far, and is fundamental for the concrete exploitation of the $\mathcal{W}_\mathcal{P}$ operator in DLP systems.

Lemma 1 *Let $\mathcal{P}$ be a HCF program and I an interpretation. $\mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})$ is equal to the union of all unfounded sets w.r.t. $\mathcal{P}$ and I.*

Proof 4 *Consider an arbitrary unfounded set X w.r.t. $\mathcal{P}$ and I. It is easy to see that $X \subseteq \mathcal{R}_{\mathcal{P},I}(X)$ and also $X \subseteq \mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})$, as also for any $Y \supseteq X$ we have $Y \subseteq \mathcal{R}_{\mathcal{P},I}(Y)$, so $X \subseteq \mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})$.*

On the other hand, we can show that any $a \in \mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})$ is contained in some unfounded set. Observe that for each rule $r \in \mathcal{P}$ such that $a \in H(r)$, $B(r) \cap (\neg.I \cup \mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P})) \neq \emptyset$ or $(H(r) \setminus \{a\}) \cap I \neq \emptyset$ holds by definition of $\mathcal{R}_{\mathcal{P},I}$. So in particular $B(r) \cap \neg.I \neq \emptyset$ or $B(r) \cap \mathcal{R}^\omega_{\mathcal{P},I}(B_\mathcal{P}) \neq \emptyset$ hold. The

only reason why $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})$ itself might not be an unfounded set is if $B(r) \cap (\neg.I \cup \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})) = \emptyset$ and $(H(r) \setminus \{a\}) \cap I \subseteq \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})$ holds for some rule, because then $(H(r) \setminus \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})) \cap I = \emptyset$ and consequently none of the three conditions of Definition 6 hold for r.

If this is the case, we can construct an unfounded set, starting from $X \setminus \{b \mid b \in H(r) \setminus \{a\}, a \in H(r)\}$. This however, may invalidate condition 2 of Definition 6 for some rule r_1 with $c \in H(r_1)$ and $c \in X_1$. Note that then $c \neq a$, as $\mathcal{P}$ is HCF. Eliminating all such cs where also conditions 1 and 3 do not hold, may again entail that condition 2 of Definition 6 becomes invalidated. However, this process can be iterated. In this process, a is never eliminated (as the program is HCF, so in the worst case we arrive at $\{a\}$).

More formally, we create a sequence as follows: Start at

$$\begin{aligned} X_0 &= X \setminus Y_0 \textit{where} \\ Y_0 &= \{b \mid b \in H(r) \setminus \{a\}, a \in H(r)\} \end{aligned}$$

Subsequently, for $i > 0$:

$$\begin{aligned} X_i &= X_{i-1} \setminus Y_i \textit{where} \\ Y_i &= \{c \mid c \in H(r) \cap X_{i-1}, B^+(r) \cap X \subseteq Y_{i-1}, B(r) \cap \texttt{not}.I = \emptyset, (H(r) \setminus X_{i-1}) \cap I = \emptyset\} \end{aligned}$$

Obviously this sequence converges, and the set which is the limit, is an unfounded set w.r.t. I and $\mathcal{P}$.

The properties shown in the following theorem guarantee that the $\mathcal{W}_{\mathcal{P}}$ operator can be efficiently used on head cycle free programs.

Theorem 3 *Let $\mathcal{P}$ be a HCF ground program and I be an interpretation, then*

1. *detecting whether I is unfounded free is feasible in* linear time,
2. *if I is unfounded-free, $GUS_{\mathcal{P}}(I)$ can be computed in* linear time,
3. *$\mathcal{W}^{\infty}_{\mathcal{P}}(I)$ (if it is defined) is computable in* quadratic time,

all in the size of $\mathcal{P}$.

Proof 5 *From Lemma 1 it follows that I is unfounded-free iff $\mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}}) \cap I = \emptyset$. We refer to Section 6.2, in which a linear time implementation of $\mathcal{R}^{\omega}_{\mathcal{P},I}(X)$ is described, so item 1 follows.*

If I is unfounded-free, then $GUS_{\mathcal{P}}(I)$ exists (cf. Proposition 2), and in fact from Lemma 1 we get that $GUS_{\mathcal{P}}(I) = \mathcal{R}^{\omega}_{\mathcal{P},I}(B_{\mathcal{P}})$, obtaining item 2.

Item 3 follows from item 2; a linear number of iterations is sufficient, each of which consumes at most linear time, so in total at most quadratic time is spent.

These theorems suggest us a direction to follow in order to improve the capability of pruning the search space in DLP systems.

6 Efficient Combination of Pruning Operators

We now show how to combine the $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ operators, resulting in an efficient implementation of DetCons.

6.1 A Pondered Choice

From the previous sections, given a program $\mathcal{P}$ and an interpretation I, we know that:

- the computation of $\Phi_{\mathcal{P}}^{\infty}(I)$ is always very efficient (*linear time computable*);
- $\mathcal{W}_{\mathcal{P}}$ is "stronger" than $\Phi_{\mathcal{P}}$ (i.e., $\Phi_{\mathcal{P}}(I) \subseteq \mathcal{W}_{\mathcal{P}}(I)$ for any interpretation I);
- the computation of $\mathcal{W}_{\mathcal{P}}$ is intractable in the general case (since deciding whether an interpretation belongs to its domain is co-NP-hard);
- the computation of $\mathcal{W}_{\mathcal{P}}^{\infty}(I)$ is tractable (*quadratic*) when $\mathcal{P}$ belongs to the restricted class of *head-cycle-free* programs;
- $\Phi_{\mathcal{P}}$ is equivalent to $\mathcal{W}_{\mathcal{P}}$ when $\mathcal{P}$ belongs to the restricted class of *acyclic* programs.

Based on these observations, we have designed an approach which exploits the positive aspects of both operators, including the efficiency of *Fitting*'s operator wherever we are sure that it is equivalent to the *Well-founded* operator or that the computation of the latter is intractable. On the other hand, our approach takes advantage of the (potentially) stronger pruning of the *Well-founded* operator where feasible. In particular, we treat each program component differently, and apply to each component the most appropriate pruning operator.

Our implementation of DetCons is sketched in Figure 3. Functions *ComputeFittingFixpoint(I, inconsistency)* and *ComputeWellFoundedFixpoint(I, inconsistency)* compute $\Phi_{\mathcal{P}}^{\infty}(I)$ and $\mathcal{W}_{\mathcal{P}}^{\infty}(I)$, respectively. They set the boolean variable *inconsistency* to true if they detect a contradiction (e.g., a branching variable previously assumed true is proven to be false). Depending on the syntactical structure of each component, we choose the more suitable of the two operators (Fitting's and Well-founded). In particular, we apply $\mathcal{W}_{\mathcal{P}}$ on *cyclic and HCF* components, where it is stronger than $\Phi_{\mathcal{P}}$ but efficiently computable. On the other hand, we apply $\Phi_{\mathcal{P}}$ on *acyclic* components, where it is equivalent to $\mathcal{W}_{\mathcal{P}}$ and more efficiently computable, and on *cyclic and not-HCF* components, where $\mathcal{W}_{\mathcal{P}}$ is intractable.

Thus, if the input program is acyclic, we always apply the linear operator $\Phi_{\mathcal{P}}$ without any loss in pruning strength. If the program is cyclic, we limit the application of $\mathcal{W}_{\mathcal{P}}$ to those components (*cyclic and HCF*) where it has potential for pruning the search space and is efficiently computable.

6.2 Implementation Issues

We conclude this section with some relevant implementation remarks. For better understanding, DetCons has been presented in a simplified manner; for details we refer to [22].

At the beginning of the DLP computation, we classify the components of the program w.r.t. acyclicity and head-cycle freeness, since the *DetCons* procedure needs this information. This classification is performed efficiently: We first build the dependency graph $DG_{\mathcal{P}}$ of $\mathcal{P}$ (in linear time); then, we compute the strongly connected components of $DG_{\mathcal{P}}$ applying the linear-time Tarjan algorithm [50], and we finally scan the components, checking whether they are acyclic or HCF, also in linear time.

```
Procedure DetCons(var I: Interpretation, var inconsistency: Boolean);
begin
    inconsistency := false;
    for each component C ∈ Comp(P) do
    begin
        switch classOf(C)
            case acyclic: ComputeFittingFixpoint(I, inconsistency);
            case cyclic-notHCF: ComputeFittingFixpoint(I, inconsistency);
            case cyclic-HCF: ComputeWellFoundedFixpoint(I, inconsistency);
        end;
        if inconsistency then break;
    end;
end;
```

Figure 3: *The DetCons Procedure*

To implement the *Well-founded* operator, we have designed an algorithm computing its negative part, i.e., $GUS_{\mathcal{P},C}(I)$ for an interpretation I and a cyclic HCF component C of a program $\mathcal{P}$ ($GUS_{\mathcal{P},C}(I)$ denotes the union of all unfounded sets of $\mathcal{P}$ w.r.t. I which are contained in component C). The efficient implementation of the positive part of the Well-founded operator is straightforward and has already been implemented within DetCons, cf. [22]. The sketch of the algorithm is depicted in Figure 4 (all implementation details can be found in [9]).

Recall Definition 6 where three conditions account for cases in which a set of atoms cannot be derived. Conditions (i) and (iii) basically correspond to rule satisfaction (w.r.t. I and $I - X$, respectively), while condition (ii) is used to detect positive cycles without foundation. The basic idea, given a component C, is to compute $C - GUS_{\mathcal{P},C}(I)$ by incrementally deriving atoms in C which are "founded", i.e., which do not belong to $GUS_{\mathcal{P},C}(I)$. This means that we want to build a finite sequence $Y_0, \ldots, Y_n$, where $Y_0 = \emptyset$ and $Y_n = C - GUS_{\mathcal{P},C}(I)$. To this end, we look for rules which do not satisfy any of the three conditions of Definition 6 (the conditions are checked w.r.t. X set to Y_i and the interpretation I). Once one such a rule r is found, we derive that $H(r) \cap C$ (which is a single atom, since C is HCF) is "founded" (more accurately, "not unfounded"), that is, it does not belong to $GUS_{\mathcal{P},C}(I)$ and can thus be added to Y_{i+1}. The "foundedness" of an atom may imply the foundedness of further atoms; we proceed until a fixpoint is reached.

There is yet more room for optimization. Observing Definition 6, one can see that condition (1) does not involve the unfounded set X: it is therefore "static" with respect to the narrowing process, and can be checked once before computing the Y_is. In addition, for condition (3) a similar, less straightforward, argument holds: if we take for each r the set $\{r \mid (H(r) - C) \cap I = \emptyset\}$, then for each i we have that $(H(r) - (C - Y_i)) \cap I \neq \emptyset$ only if some $a \in I$ and $a \in Y_i$, but then there is some other rule r_1 satisfying (2) or (3), otherwise $a \in Y_i \subseteq C - GUS_{\mathcal{P},C}(I)$ would not hold (as the component C is HCF). So for creating the sequence $Y_0, \ldots, Y_n$ it is sufficient to consider only rules for which (1) and (3) (with $X = C$) do not hold (let us call these rules "active"). Thus, for $i \geq 0$, we compute $Y_{i+1} = Y_i \cup \{a \mid$

$a \in H(r), (B^+(r) \cap C) \subseteq Y_i\}$ where r is an "active" rule. We have implemented this computation by a linear-time algorithm using a propagation queue and counters which store $|B^+(r) \cap Y_i|$ for each "active" rule r.

At the end of the computation, all atoms in $C - Y_n$ are known to be unfounded, and we set them to false in I. This can result in inconsistency if $I \cap (C - Y_n) \neq \emptyset$, i.e., if an unfounded atom was set to true in I.

It is worthwhile noting that procedure *computeGUS* can be seen as a (linear time) implementation of the computation of the fixpoint of the $\mathcal{R}$ operator for component C (i.e., $\mathcal{R}^{\omega}_{\mathcal{P}_C,I}(C)$). At each step, instead of explicitly computing $\mathcal{R}_{\mathcal{P}_C,I}(X)$, the procedure computes its complement $C - \mathcal{R}_{\mathcal{P}_C,I}(X)$. The i-th element Y_i of the above sequence corresponds to the element $X_i = C - Y_i$. Thus, in terms of the $\mathcal{R}_{\mathcal{P}_C,I}$ operator, the procedure computes the sequence $X_0 = C$, $X_1 = \mathcal{R}_{\mathcal{P}_C,I}(X_0)$ (X_1 is the set of atoms of C which are not in *FoundedAtoms* after the initialization phase of the procedure in Figure 4),..., $X_n = \mathit{GUS}_{\mathcal{P},C}(I)$.

We have designed a further optimization to the above algorithm, that we have also incorporated in our actual implementation of DetCons in the **DLV** system.[4] Frequently, all atoms in $\mathit{GUS}_{\mathcal{P},C}(I)$ happen to be already false w.r.t. I, and its computation is completely useless. We would like to identify cases where this condition can be recognized without actually computing $\mathit{GUS}_{\mathcal{P},C}(I)$. To this end, at each step of DetCons (Figure 3), we propagate the deterministic consequences over all components by means of *Fitting*'s operator, and subsequently invoke the *Well-founded* operator only on *some selected* cyclic and HCF components instead of all, as described below.

At the very beginning of the computation, the GUS-computation is applied on each cyclic and HCF component. Later, we invoke the GUS-computation only on components where some atom may have become unfounded by the most recent propagation step. In order to do that, we store some further information during the *Fitting* propagation.

Basically, an atom A can become unfounded if it has lost a "potential support", as some rule r containing A in the head has become satisfied during the last propagation, i.e., either the body of r has become false or a head atom of r, different from A, has become true (recall that a (disjunctive) rule can support only one atom in its head). If no atom of a component C has lost any potential support, then we know that $\mathit{GUS}_{\mathcal{P},C}(I)$ is unaltered, and its computation is superfluous. To automatically recognize such superfluous computations, when a rule becomes satisfied, we push the component of each atom that loses a potentially supporting rule into a queue. We eventually launch the GUS-computation only for the components stored in this queue, i.e. only for those cyclic and HCF components in which at least one head atom lost a potentially supporting rule. We thus avoid a lot of useless GUS computations.

It is worthwhile noting that the algorithms for computing both the *Greatest Unfounded Set* (as described above) and *Fitting*'s operator are *linear-time* algorithms; they use propagation queues and suitable counters à la Dowling and Gallier [14, 40].

7 Comparisons and Benchmarks

In order to evaluate our intuitions, we have implemented two new pruning operators, based on the conclusions drawn in the previous section in the DLP system **DLV** [32] and experimentally compared the new

[4] A similar technique, for the (smaller) class of disjunction-free programs, is implemented also in Smodels [47].

```
Procedure computeGUS (var C:Component, var I: Interpretation, var inconsistency: Boolean)
var a,b : Atom;
var FoundedAtoms : Interpretation;  % Stores the set of atoms of C which are proven to be "founded" (not unfounded).
var GUSqueue : Queue;               % Stores the atoms whose "foundedness" is to be propagated; controls the fixpoint computation.
var r.counter : Integer; (∀r)       % Stores the number of atoms of C in B+(r) which are not proved to be founded.
                                    % If r.counter becomes zero, then the head of r gets founded.
  inconsistency := false;
  % Initialize the rules counters and the queue.
  For each atom a ∈ C do
      For each rule r such that (r is active and a ∈ H(r)) do
          r.counter := |{b : b ∈ B+(r) ∩ C}|;
          If r.counter = 0 then
              FoundedAtoms.Add(a);
              GUSqueue.Push(a);
          EndIf;
      EndFor;
  EndFor;
  % Fixpoint Computation.
  While not GUSqueue.empty() do
      a := GUSqueue.Pop();
      For each rule r such that (r is active and a ∈ B+(r)) do
          r.counter := r.counter - 1;
          If r.counter = 0 then
              Let b be the atom of C in H(r);
              FoundedAtoms.Add(b);
              GUSqueue.Push(b);
          EndIf;
      EndFor;
  EndWhile;
  % Set to false all atoms of C which are not in FoundedAtoms.
  For each atom a ∈ C do
      If a ∉ FoundedAtoms then
          If a ∈ I then
              inconsistency := true;
              return;
          else
              I := I ∪ {not a};
          EndIf;
      EndIf;
  EndFor;
EndProcedure;
```

Figure 4: *The computeGUS procedure*

pruning operators against the original pruning operator employed by **DLV**. Next, we describe the compared methods, the benchmark problems and instances, and then discuss the results of the experiments.

7.1 Overview of the Compared Methods

To evaluate our proposed operators, we have implemented the following three approaches on top of **DLV** and compared them by means of various benchmarks:

Old. The method originally employed by **DLV**. It always uses the generalization of Fitting's operator $\Phi_{\mathcal{P}}$ introduced in Section 4.1, which is efficiently computable (a fixpoint is reached in linear time), but does not prune the search space as much as $\mathcal{W}_{\mathcal{P}}$.

ifPoss. Based on the generalized Well-founded operator $\mathcal{W}_{\mathcal{P}}$ introduced in Section 4.2, and exploiting observations from Section 6, this method avoids the use of $\mathcal{W}_{\mathcal{P}}$ on those components where its computation is very expensive (since deciding its applicability is intractable). It employs $\mathcal{W}_{\mathcal{P}}$ on all head-cycle free components, while resorting to the generalization of Fitting's operator $\Phi_{\mathcal{P}}$ on the remaining (i.e., non-HCF) components.

ifNeed. This is the method described in Figure 3. It fully implements the theoretical results from Section 6, using both $\Phi_{\mathcal{P}}$ and $\mathcal{W}_{\mathcal{P}}$ where appropriate, and is a refinement of method **ifPoss**. $\mathcal{W}_{\mathcal{P}}$ is only used for cyclic head-cycle free components, whereas $\Phi_{\mathcal{P}}$ is applied on all acyclic components.

7.2 Benchmark Problems

To evaluate the pruning techniques described in the previous sections, we chose three benchmark problems: Hamiltonian path, Blocksworld Planning, and Sokoban.

For the sake of readability, the full encodings used for the benchmarks are reported in Appendix A.

Hamiltonian Path (HAMPATH) is a classical NP-complete problem from graph theory: Given an undirected graph $G = (V, E)$, where V is the set of vertices of G and E is the set of edges, 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?

This is almost the same problem as described in Section 2.3, but the path does not have to be cyclic. The full encoding is reported in Appendix A.1.

Blocksworld (BW) is a classic problem from the planning domain, and one of the oldest problems in AI: Given a table and a number of blocks in a (known) initial state and a desired goal state, try to reach that goal state by moving one block at a time such that each block is either on top of another block or the table at any given time step. The encoding is reported in Appendix A.2.

Figure 5 shows a simple instance that can be solved in three time steps: First we move block c to the table, then block b on top of a, and finally c on top of b.

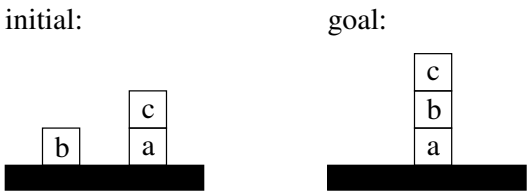

Figure 5: A Blocksword Instance

Sokoban (SOKO) is a game puzzle developed by the Japanese company *Thinking Rabbit, Inc.* in 1982. *Sokoban* means "warehouse-keeper" in Japanese. Each puzzle consists of a room layout (a number of square fields representing walls or parts of the floor, some of which are marked as storage space) and a starting situation (one sokoban and a number of boxes, all of which must reside on some floor location, where one box occupies precisely one location and each location can hold at most one box). The goal is to move all boxes onto storage locations. To this end, the sokoban can walk on floor locations (unless occupied by some box), and push single boxes onto unoccupied floor locations. Figure 6 shows a typical configuration involving two boxes, where grey fields are storage fields and black fields are walls.

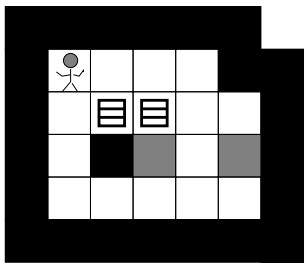

Figure 6: A Sokoban Instance

We have written a **DLV** program, reported in Appendix A.3, which finds solutions with a given number of push actions (where one push action can move a box over a number of fields, but always in the same direction) for a given puzzle together with a script which iteratively runs that **DLV** program with increasing numbers of push actions (starting at one) until some solution is found. This finds solutions with a minimal number of push actions.

The puzzle in Figure 6 is solvable with 6 push actions, so the script uses **DLV** to prove that no solutions with 1 to 5 push actions exist, and then to compute a solution with 6 push actions.

7.3 Benchmark Data

We created random graph instances for HAMPATH using a tool[5] by Patrik Simons which has been used to compare Smodels against SAT solvers [47]. For each problem size n we generated *ten* instances, always

[5] `http://www.tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`

assuming node 0 as the starting node, and for each instance we stopped after the first solution had been found.

The blocksworld problems P1 to P4 have been employed in [21] to compare DLP systems, and can be solved in 4, 6, 8 and 9 steps, respectively. We augmented these by problems P5 and P6 which require 11 and 12 steps, respectively. For each of these problems, we generated 8 random permutations of the full input (including the encoding). In addition, we also tried to solve each of these problems with one step less than required, which fails to produce any plan but shows the minimality of the regular solutions. These instances are labeled P1-1, P2-1 and so forth in Figure 8.

A vast amount of Sokoban puzzles is available on the Internet in a simple ANSI text format. The examples we used for benchmarks are results of efforts to automatically generate hard puzzles. One set has been created by Yoshio Murase[6], the other set is due to Jacques Duthen[7]. The puzzle in Figure 6 is number 2 of Duthen's instances.

7.4 Experimental Results

All experiments have been carried out on a Pentium III/1GHz machine with 512MB of main memory, running SuSE GNU/Linux (kernel 2.4.21). The different **DLV** executables have been built with the GCC compiler (version 3.2.2).

For each invocation of **DLV** we allowed a maximum run-time of 600 seconds. For SOKO there may be several invocations per problem instance, so the total reported time may be more than 600 seconds.

		Average			Maxima	
	Old	**ifPoss**	**ifNeed**	**Old**	**ifPoss**	**ifNeed**
10	0.02	0.02	0.04	0.02	0.03	0.10
20	0.05	0.06	0.05	0.07	0.08	0.05
30	0.08	0.09	0.09	0.15	0.12	0.09
40	0.12	0.14	0.13	0.13	0.16	0.14
50	55.59	0.20	0.19	443.58	0.23	0.20
60	11.52	0.29	0.27	86.54	0.32	0.29
70	-	0.38	0.35	-	0.42	0.37
80	-	0.50	0.47	-	0.54	0.50
90	-	0.66	0.60	-	0.80	0.65
100	-	0.83	0.85	-	1.09	1.11
110	-	1.02	1.01	-	1.25	1.22
120	-	17.77	16.86	-	116.62	110.32

Figure 7: Hamiltonian Path Running Times

Results for HAMPATH are shown in Figure 7. It is easy to see that both **ifPoss** and **ifNeed** perform similarly to **Old** for very small problem instances, but scale tremendously better and are able to efficiently

[6] `http://www.ne.jp/asahi/ai/yoshio/sokoban/auto52/`

[7] `http://hem.passagen.se/awl/ksokoban/sokogen-990602.skm`

	Old	**ifPoss**	**ifNeed**
P1 -1	0.03	0.03	0.03
P2 -1	0.05	0.05	0.06
P3 -1	2.35	2.37	2.40
P4 -1	1.28	1.31	1.32
P5 -1	15.86	15.90	15.94
P6 -1	262.64	263.45	263.65
P1	0.04	0.04	0.04
P2	0.07	0.08	0.08
P3	5.97	5.99	5.99
P4	14.53	14.68	14.67
P5	259.65	261.40	261.96
P6	234.62	235.00	235.45

Figure 8: Blocksworld - Average Running Times

deal with graphs of 120 nodes, whereas **Old** is not able to solve almost all problems with more than 60 nodes within the allowed time. **ifPoss** and **ifNeed** scale much better, their behavior is similar here, with **ifNeed** being slightly faster than **ifPoss**.

The programs for HAMPATH have highly cyclic HCF dependency graphs. Thus, **ifPoss** and **ifNeed** can exploit the pruning power of the Well-founded operator, significantly outperforming **Old** which employs only Fitting's operator. On the other hand, the dependency graphs of these programs usually have one big component containing nearly all atoms. Therefore, there are only few differences (but still noticeable) between **ifPoss** and **ifNeed**, as the latter cannot avoid many calls to the Well-founded operator.

For BW, **Old**, **ifPoss** and **ifNeed** are nearly equivalent, and all three approaches seem to scale similarly. We explain this as follows: These programs have only few cyclic HCF components while most components are acyclic. Moreover, these few cyclic components are also very small, and the Well-founded operator does not bring a relevant gain in terms of pruning compared to Fitting's operator, so the three implementations show essentially the same behavior.

SOKO, finally, shows that both **ifPoss** and **ifNeed** perform significantly better than **Old**, which fails to solve more than 60% of all problems instances and usually takes one or two orders of magnitude longer to solve the remaining ones:

	Yoshio Murase		Jacques Duthen	
	solved	unsolved	solved	unsolved
Old	9	43	36	42
ifPoss	40	12	68	10
ifNeed	40	12	68	10

As can be verified on the data reported in Appendix B, both for the Yoshio Murase and the Jacques Duthen sets, **ifNeed** yields an average speedup of about 6% over **ifPoss**, the maximum speedup being

about 10% (instances 14 and 33 for Yoshio Murase and Jacques Duthen, respectively). For this class of benchmarks, the potential gain brought about by avoiding useless calls to $\mathcal{W}_\mathcal{P}$ is evident.

In summary, the experiments show that both **ifPoss** and **ifNeed** are strictly preferable to **Old**; and that of these two, **ifNeed** shows a measurable speedup on a wide range of examples. Therefore, recent **DLV** releases employ **ifNeed** by default, even if a command-line option easily allows the user to switch it off, thus making **DLV** use only the original implementation of *Fitting*'s operator.

8 Conclusions

We have addressed some key issues for the computation of disjunctive logic programs. In particular, we have focused on search space pruning, which is crucial for the efficiency of DLP systems. We have carried out an in-depth analysis of two main pruning operators for DLP, namely *Fitting*'s operator and the *Well-founded* operator. We have proposed a new strategy for the intelligent combination of the two pruning operators, and we have implemented it in the **DLV** system. We have carried out experiments, which confirm the strong impact of the pruning operators on the efficiency of DLP systems, and assess the importance of our results. Interestingly, even if the Well-founded operator is computationally more expensive than Fitting's operator (quadratic time versus linear time in the propositional case), its stronger pruning power often pays off and reduces the computation time by an order of magnitude in some cases.

Future work will focus on tuning the actual implementations of the pruning operators, and on singling out new classes of DLP programs where they are efficiently computable.

References

[1] Christian Anger, Kathrin Konczak, and Thomas Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 406–410. Springer Verlag, September 2001.

[2] K. Apt and N. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[3] Chandrabose Aravindan, Jürgen Dix, and Ilkka Niemelä. DisLoP: A Research Project on Disjunctive Logic Programming. *AI Communications – The European Journal on Artificial Intelligence*, 10(3/4):151–165, 1997.

[4] Yulia Babovich. Cmodels homepage, since 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.

[5] Colin Bell, Anil Nerode, Raymond T. Ng, and V.S. Subrahmanian. Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases. *Journal of the ACM*, 41:1178–1215, 1994.

[6] R. Ben-Eliyahu and R. Dechter. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 12:53–87, 1994.

[7] Kenneth A. Berman, John S. Schlipf, and John V. Franco. Computing the well-founded semantics faster. In *Proceedings of the 3rd International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'95)*, volume 928 of *Lecture Notes in AI (LNAI)*, pages 113–125. Springer, 1995.

[8] Stefan Brass, Jürgen Dix, Burkhardt Freitag, and Ulrich Zukowski. Transformation-Based Bottom-Up Computation of the Well-Founded Model. *Journal of the Theory and Practice of Logic Programming*, 1(5):497–538, 2001.

[9] Francesco Calimeri. Progettazione e Sviluppo di Tecniche Di Ottimizzazione per Sistemi di Basi di Conoscenza. Master's thesis, D.E.I.S., Università degli Studi della Calabria, Rende (CS), Italy, 2001. Supported by Nicola Leone.

[10] Weidong Chen and David Scott Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.

[11] Paweł Cholewiński, V. Wiktor Marek, and Mirosław Truszczyński. Default Reasoning System DeReS. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR '96)*, pages 518–528, Cambridge, Massachusetts, USA, 1996. Morgan Kaufmann Publishers.

[12] Paweł Cholewiński, Victor W. Marek, Artur Mikitiuk, and Mirosław Truszczyński. Computing with Default Logic. *Artificial Intelligence*, 112(2–3):105–147, 1999.

[13] J. Dix. Semantics of Logic Programs: Their Intuitions and Formal Properties. An Overview. In *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pages 241–329. DeGruyter, 1995.

[14] W. F. Dowling and J. H. Gallier. Linear-time Algorithms for Testing the Satisfability of Propositional Horn Formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[15] D. East and M. Truszczyński. Propositional Satisfiability in Answer-set Programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, pages 138–153. Springer Verlag, LNAI 2174, 2001.

[16] Deborah East and Mirosław Truszczyński. dcs: An Implementation of DATALOG with Constraints. In Chitta Baral and Mirosław Truszczyński, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning (NMR'2000)*, Breckenridge, Colorado, USA, April 2000.

[17] Deborah East and Mirosław Truszczyński. System Description: aspps – An Implementation of Answer-Set Programming with Propositional Schemata. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings*, number 2173 in Lecture Notes in AI (LNAI), pages 402–405. Springer Verlag, September 2001.

[18] Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00), July 30 – August 3, 2000, Austin, Texas USA*, pages 417–422. AAAI Press / MIT Press, 2000.

[19] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative Problem-Solving Using the DLV System. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[20] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[21] Esra Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[22] Wolfgang Faber. *Enhancing Efficiency and Expressiveness in Answer Set Programming Systems*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, 2002.

[23] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In INAP Organizing Committee, editor, *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139. Prolog Association of Japan, September 1999.

[24] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with Heuristics for Answer Set Programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001*, pages 635–640, Seattle, WA, USA, August 2001. Morgan Kaufmann Publishers.

[25] François Fages. Consistency of Clark's Completion and Existence of Stable Models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.

[26] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[27] M. Gelfond and N. Leone. Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence*, 138(1-2):3–38, 2002.

[28] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[29] Tomi Janhunen, Ilkka Niemelä, Dietmar Seipel, Patrik Simons, and Jia-Huai You. Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM Transactions on Computational Logic*, 2005. To appear.

[30] Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 359–363, 1992.

[31] N. Leone, M. Romeo, P. Rullo, and D. Saccà. Effective Implementation of Negation in Database Logic Query Languages. In *LOGIDATA+: Deductive Database with Complex Objects*, LNCS 701, pages 159–175, 1993.

[32] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 2005. To appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.

[33] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. *Information and Computation*, 135(2):69–112, June 1997.

[34] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Proceedings of the 8th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'05)*, LNCS, pages 447–451. Springer, September 2005.

[35] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7)*, LNCS, pages 346–350. Springer, January 2004.

[36] V. Lifschitz and H. Turner. Splitting a Logic Program. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming (ICLP'94)*, pages 23–37, Santa Margherita Ligure, Italy, June 1994. MIT Press.

[37] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[38] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[39] Norman McCain and Hudson Turner. Satisfiability Planning with Causal Theories. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 212–223. Morgan Kaufmann Publishers, 1998.

[40] Michel Minoux. LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters*, 29:1–12, 1988.

[41] Ilkka Niemelä. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[42] Ilkka Niemelä and Patrik Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In Michael J. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming (ICLP'96)*, pages 289–303, Bonn, Germany, September 1996. MIT Press.

[43] Ilkka Niemelä and Patrik Simons. Smodels – An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in AI (LNAI)*, pages 420–429, Dagstuhl, Germany, July 1997. Springer Verlag.

[44] Stanislaw P. Radziszowski. Small Ramsey Numbers. *The Electronic Journal of Combinatorics*, 1, 1994. Revision 9: July 15, 2002.

[45] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David S. Warren, and Juliana Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 2–17, Dagstuhl, Germany, July 1997. Springer Verlag.

[46] Dietmar Seipel and Helmut Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. In Antoni Olivé, editor, *Proceedings International Workshop on the Deductive Approach to Information Systems and Databases (DAISD'94)*, pages 325–343. Universitat Politecnica de Catalunya (UPC), 1994.

[47] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[48] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[49] V.S. Subrahmanian, Dana Nau, and Carlo Vago. WFS + Branch and Bound = Stable Models. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):362–377, June 1995.

[50] R. Tarjan. Depth-First Search and Linear Graph Algorithm. *SIAM Journal on Computing*, 1(2), June 1972.

[51] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

APPENDIX

A Encodings of the Problems

Note that some of the encodings reported here employ true negation (denoted using "$-$" in front of atoms), which has not been introduced in the syntax definition in this paper. However, **DLV** reduces such programs to equivalent ones without true negations, substituting each truly negated occurrence of an atom a by a new atom na and adding a constraint $:- a, na.$ for each atom a that occurs both with and without true negation.

A.1 Hamiltonian Path

Suppose that the graph G is specified by two predicates $node(X)$ and $arc(X, Y)$, and the starting node is specified by the predicate $start$ which contains only a single tuple. Then, the following program solves the problem HAMPATH:

```
% Each node has to be reached.
reached(X) :- start(X).      reached(X) :- inPath(Y,X).      :- node(X), not reached(X).
% Guess whether a given arc is in the path or not.
inPath(X,Y) v outPath(X,Y) :- reached(X), arc(X,Y).
% At most one incoming/outgoing arc!
:- inPath(X,Y), inPath(X,Y1), Y<>Y1.      :- inPath(X,Y), inPath(X1,Y), X<>X1.
```

A.2 Blocksworld

For blocksworld, we used an encoding of the problem domain which is derived from an encoding in an action language. The reader can refer to [21, 23] for further details.

```
% specification of the move action
move(B,L,T) v -move(B,L,T) :- block(B), location(L), actiontime(T), B <> L.
% the effects of moving a block
on(B,L,T1) :- move(B,L,T), #succ(T,T1).
-on(B,L,T1) :- move(B,_,T), on(B,L,T), #succ(T,T1).
% move preconditions
% a block can be moved only when it's clear
:- move(B,L,T), on(B1,B,T).
% if a block is moved onto another block, the latter must be clear
:- move(B,B1,T), on(B2,B1,T), block(B1).
% concurrent actions are not allowed
:- move(B,_,T), move(B1,_,T), B <> B1.
:- move(_,L,T), move(_,L1,T), L <> L1.
% inertia
on(B,L,T1) :- on(B,L,T), not -on(B,L,T1), #succ(T,T1).
% time at which actions can be initiated
actiontime(T) :- T < #maxint, #int(T).
% location definition (blocks are defined in the problem instances)
location(t).      location(B) :- block(B).
```

A.3 Sokoban

The encoding solving SOKO puzzles follows.

```
% Timesteps etc.
time(T) :- #int(T).      actiontime(T) :- #int(T), T! = #maxint.
% define left and bottom for simplicity
left(L1, L2) :- right(L2, L1).      bot(L1, L2) :- top(L2, L1).
% define the adjacent squares
adj(L1, L2) :- right(L1, L2).      adj(L1, L2) :- left(L1, L2).
adj(L1, L2) :- top(L1, L2).      adj(L1, L2) :- bot(L1, L2).
% all the locations
location(L) :- adj(L, _).
% It is possible to push a box if the Sokoban can move to the square
% in front the box and the box can be pushed in the desired direction.
push(B, right, B1, T) ∨ -push(B, right, B1, T) :-
    reachable(L, T), right(L, B), box(B, T), pushable_right(B, B1, T), good_pushlocation(B1), actiontime(T).
push(B, left, B1, T) ∨ -push(B, left, B1, T) :-
    reachable(L, T), left(L, B), box(B, T), pushable_left(B, B1, T), good_pushlocation(B1), actiontime(T).
push(B, up, B1, T) ∨ -push(B, up, B1, T) :-
    reachable(L, T), top(L, B), box(B, T), pushable_top(B, B1, T), good_pushlocation(B1), actiontime(T).
push(B, down, B1, T) ∨ -push(B, down, B1, T) :-
    reachable(L, T), bot(L, B), box(B, T), pushable_bot(B, B1, T), good_pushlocation(B1), actiontime(T).
% reachable represents the locations which are reachable at some
% timestep from the location of the Sokoban in that timestep.
reachable(L, T) :- sokoban(L, T).      reachable(L, T) :- reachable(L1, T), adj(L1, L), notbox(L, T).
% The following rules define the possible pushes during some timestep.
pushable_right(B, D, T) :- box(B, T), right(B, D), notbox(D, T), actiontime(T).
pushable_right(B, D, T) :- pushable_right(B, D1, T), right(D1, D), notbox(D, T).
pushable_left(B, D, T) :- box(B, T), left(B, D), notbox(D, T), actiontime(T).
pushable_left(B, D, T) :- pushable_left(B, D1, T), left(D1, D), notbox(D, T).
pushable_top(B, D, T) :- box(B, T), top(B, D), notbox(D, T), actiontime(T).
pushable_top(B, D, T) :- pushable_top(B, D1, T), top(D1, D), notbox(D, T).
pushable_bot(B, D, T) :- box(B, T), bot(B, D), notbox(D, T), actiontime(T).
pushable_bot(B, D, T) :- pushable_bot(B, D1, T), bot(D1, D), notbox(D, T).
% Effects of pushing.
sokoban(L, T1) :- push(_, right, B1, T), #succ(T, T1), right(L, B1).
sokoban(L, T1) :- push(_, left, B1, T), #succ(T, T1), left(L, B1).
sokoban(L, T1) :- push(_, up, B1, T), #succ(T, T1), top(L, B1).
sokoban(L, T1) :- push(_, down, B1, T), #succ(T, T1), bot(L, B1).
-sokoban(L, T1) :- push(_, _, _, T), #succ(T, T1), sokoban(L, T).
box(B, T1) :- push(_, _, B, T), #succ(T, T1).
-box(B, T1) :- push(B, _, _, T), #succ(T, T1).
```

```
% Inertia. Unless changes are caused, things remain as they were.
box(LB,T1) :- box(LB,T),#succ(T,T1),not -box(LB,T1).
sokoban(LS,T) :- sokoban(LS,T),#succ(T,T1),not -sokoban(LS,T1).
% Unique actions per timestep.
:- push(B,_,_,T),push(B1,_,_,T),B! = B1.      :- push(B,D,_,T),push(B,D1,_,T),D! = D1.
:- push(B,D,B1,T),push(B,D,B11,T),B1! = B11.
% Auxiliary definitions.
good_pushlocation(L) :- right(L,_),left(L,_).      good_pushlocation(L) :- top(L,_),bot(L,_).
good_pushlocation(L) :- solution(L).
```

B Sokoban Detailed Results

In Tables 1 and 2 you find detailed results for the Sokoban puzzle benchmarks. The reported numbers are seconds for runtime (user + system time).

	Old	**ifPoss**	**ifNeed**
1	738.04	21.39	19.70
2	-	-	-
3	-	7.48	6.99
4	-	7.74	7.29
5	-	77.59	71.42
6	-	173.68	162.39
7	-	195.27	178.51
8	548.24	12.27	11.62
9	264.85	16.04	14.97
10	-	16.01	14.77
11	-	-	-
12	-	54.76	50.20
13	-	23.90	22.18
14	-	271.24	242.87
15	-	19.25	17.54
16	-	627.05	575.95
17	-	20.13	18.58
18	-	45.31	41.43
19	-	487.70	441.67
20	-	32.13	30.09
21	-	99.02	92.08
22	-	-	-
23	255.83	21.41	20.10
24	-	13.81	12.79
25	-	31.69	29.24
26	321.81	20.69	19.26

	Old	**ifPoss**	**ifNeed**
27	-	7.54	7.13
28	-	199.58	183.73
29	421.22	4.07	3.86
30	-	-	-
31	124.32	14.42	13.23
32	-	-	-
33	-	219.63	202.54
34	-	62.37	57.57
35	-	199.52	182.09
36	-	-	-
37	549.44	17.63	16.68
38	-	56.75	51.69
39	-	-	-
40	-	16.62	15.07
41	-	-	-
42	-	-	-
43	-	27.04	25.15
44	-	295.75	276.50
45	-	-	-
46	-	-	-
47	-	9.44	8.76
48	-	-	-
49	385.71	13.89	12.90
50	-	25.35	23.18
51	-	217.76	203.39
52	-	246.47	230.39

Table 1: Detailed results for the Yoshio Murase SOKO instances.

	Old	ifPoss	ifNeed
1	0.98	0.83	0.82
2	1.90	1.67	1.60
3	0.65	0.68	0.67
4	2.99	1.60	1.56
5	61.65	2.22	2.16
6	152.54	21.19	19.28
7	3.86	1.67	1.60
8	100.86	3.87	3.73
9	1.95	1.43	1.37
10	1.16	0.90	0.90
11	3.78	1.59	1.55
12	83.54	2.27	2.20
13	27.30	5.05	4.75
14	445.60	4.86	4.65
15	48.18	2.92	2.74
16	0.59	0.60	0.60
17	53.47	9.74	9.10
18	361.54	5.98	5.58
19	8.21	2.14	2.06
20	-	8.72	8.07
21	1.55	1.01	0.99
22	565.06	7.54	7.01
23	44.30	1.72	1.68
24	-	27.28	25.18
25	82.02	11.41	10.57
26	125.81	7.29	6.76
27	3.87	1.55	1.50
28	-	13.63	12.52
29	-	23.90	22.46
30	13.81	2.22	2.18
31	37.72	1.92	1.87
32	3.86	1.21	1.19
33	-	39.44	35.61
34	-	63.17	58.07
35	189.21	10.37	9.44
36	-	302.66	281.64

	Old	ifPoss	ifNeed
37	-	589.14	533.16
38	789.52	81.20	75.19
39	132.97	2.59	2.52
40	-	389.44	354.95
41	8.37	1.71	1.70
42	-	104.60	96.22
43	403.12	2.51	2.42
44	224.12	3.61	3.40
45	-	31.23	28.88
46	-	6.43	6.10
47	-	8.67	8.20
48	-	221.16	201.23
49	-	33.38	30.79
50	-	-	-
51	-	-	-
52	-	349.29	328.23
53	-	10.98	10.26
54	-	-	-
55	37.24	2.77	2.62
56	-	10.87	10.04
57	-	79.42	71.75
58	-	64.82	58.67
59	-	8.31	7.73
60	-	5.62	5.29
61	-	260.75	241.98
62	-	116.11	106.23
63	-	688.73	638.46
64	257.59	21.11	19.60
65	-	375.09	341.36
66	-	40.46	37.40
67	-	695.20	629.97
68	-	589.31	533.12
69	-	233.59	214.51
70	-	30.53	28.05
71	-	14.55	13.34

Table 2: Detailed results for the Jacques Duthen SOKO instances.