A Comparison of Heuristics for Answer Set Programming*

Wolfgang Faber[α] and Nicola Leone[β] and Gerald Pfeifer[α]

Institut für Informationssysteme, TU Wien
Favoritenstr. 9-11, 1040 Wien, Austria
faber@kr.tuwien.ac.at, pfeifer@dbai.tuwien.ac.at

Department of Mathematics, University of Calabria
87030 Rende (CS), Italy
leone@unical.it

Abstract

Answer Set Programming (ASP) is a novel programming paradigm, which allows to solve problems in a simple and highly declarative way. The language of ASP (function-free disjunctive logic programming) is very expressive, and allows to represent even problems of high complexity (every problem in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$).

As for SAT solvers, the heuristic 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) dramatically affects the performance of an ASP system. While heuristics for SAT have received a fair deal of research in AI, only little work in heuristics for ASP has been done so far.

In this paper, we extend to the ASP framework a number of heuristics which have been successfully employed in existing systems, and we compare them experimentally. To this end, we implement such heuristics in the ASP system `DLV`, and we evaluate their respective efficiency on a number of benchmark problems taken from various domains. The experiments show interesting results, and evidence a couple of promising heuristic criteria for ASP, which sensibly outperform the heuristic of `DLV`.

1 Introduction

Answer set programming (ASP) is a declarative approach to programming, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. The knowledge representation language of ASP is very expressive: function-free logic programs with classical negation where disjunction is allowed in the heads of the rules and nonmonotonic negation may occur in the bodies of the rules. The intended models of an ASP program (i.e., the semantics of the program) are subset-minimal models which are "grounded" in a precise sense, they are called *answer sets* [GL91]. The idea of answer set programming is to represent a given computational problem by an ASP

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

program whose answer sets correspond to solutions, and then use an answer set solver to find such a solution [Lif99].

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

$$\begin{array}{ll} r_1: & color(X, r) \vee color(X, y) \vee color(X, g) :- \; node(X) \\ r_2: & :- \; edge(X, Y), color(X, C), color(Y, C) \end{array}$$

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

An advantage of answer set programming over SAT-based programming is that problems can be encoded more easily in the ASP language than in propositional CNF formulas, thanks to the nonmonotonic character of disjunction and negation as failure [Lif99]. Importantly, due to the support for variables, *every* problem in the complexity class Σ_2^P (i.e., in $\mathrm{NP}^{\mathrm{NP}}$) can be **directly** encoded in an ASP program which can then be used to solve all problem instances in a uniform way [EGM97].

The high expressiveness of answer set programming comes at the price of a high computational cost in the worst case, which makes the implementation of efficient ASP systems a difficult task. Nevertheless, some efforts have been done in this direction, and a number of ASP systems are now available. The two best known ASP systems are DLV [EFLP00], and Smodels [Ilk99, Sim00], but also other systems support ASP to some extent, including CCALC [MT98], DCS [ET00], QUIP [EETW00], and XSB [RSS$^+$97].

The heart of the computation of an ASP system is the process of model generation, where a model of the program is produced, which is then subjected to an answer set check. For the generation of models, ASP systems employ procedures which are similar to Davis-Putnam procedures used in SAT solvers. As for SAT solvers, 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) is fundamentally important for the efficiency of a model generation procedure, and dramatically affects the overall performance of an ASP system. While a lot of work has been done in AI developing new heuristics and comparing alternative heuristics for SAT (see, e.g., [HV95, LA97, Fre95]), only little work has been done so far for ASP systems. In particular, we are not aware of any previous comparison between heuristics for ASP.

In this paper, we evaluate different heuristics for ASP systems. To this end, we first consider a couple of heuristics which have been very successful in SAT solvers or in non-disjunctive ASP systems, and we extend them to the framework of (disjunctive) ASP. We then perform an experimentation activity to compare the different heuristics. In particular, we implement the heuristics in the ASP system DLV, and we compare their respective efficiency on a number of benchmark problems taken from various domains. The experiments show interesting results, and evidence a couple of promising heuristic criteria for ASP, which sensibly outperform the heuristic of DLV. Nevertheless, this paper is not at all a conclusive work on heuristics for ASP; rather, it is a first step in this field that will hopefully stimulate further works on the design and evaluation of heuristics for ASP, which are strongly needed to build efficient ASP solvers.

2 Answer Set Programming Language

In this section, we provide a formal definition of the syntax and semantics of the answer set programming language supported by DLV: disjunctive datalog extended with strong negation. For further background, see [GL91, EFLP00].

ASP Programs

A *(disjunctive) rule* r is a formula

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals (atoms possibly preceded by the classical negation symbol $\neg$) and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \cdots, b_k, \ \texttt{not} \ b_{k+1}, \cdots, \texttt{not} \ b_m$ is the *body*, $b_1, \cdots, b_k$ the *positive body*, and $\texttt{not} \ b_{k+1}, \cdots, \texttt{not} \ b_m$ the *negative body* of r. Comparison operators (like $=, <, >, <>$) are built-in predicates in ASP systems, and may appear in the bodies of rules.

A *disjunctive datalog program* (also called *ASP program* in this paper) $\mathcal{P}$ is a finite set of rules.

As usual, an object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables.

Answer Sets

We describe the semantics of consistent answer sets, which has originally been defined in [GL91].

Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_{\mathcal{P}}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_{\mathcal{P}}$ be the set of all possible combinations of predicate symbols appearing in $\mathcal{P}$ with constants of $U_{\mathcal{P}}$, possibly preceded by $\neg$.

A set L of literals is said to be *consistent* if, for every literal $\ell \in L$, its complementary literal is not contained in L.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* $Ground(\mathcal{P})$ of $\mathcal{P}$is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

For every program $\mathcal{P}$, we define its answer sets using its ground instantiation $Ground(\mathcal{P})$ in two steps, following [Lif96]: First we define the answer sets of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define answer sets of general programs.

An interpretation I is a set of ground literals. A consistent interpretation $I \subseteq B_{\mathcal{P}}$ is *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive program), if, for every $r \in Ground(\mathcal{P})$, at least one literal in the head is true whenever all literals in the body are true. X is an *answer set* for $\mathcal{P}$ if it is minimal w.r.t. set inclusion and closed under $\mathcal{P}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. a set $X \subseteq B_{\mathcal{P}}$ is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by

- deleting all rules $r \in \mathcal{P}$ whose negative body is false w.r.t. X;

- deleting the negative body from the remaining rules.

An answer set of a general program $\mathcal{P}$ is a set $X \subseteq B_{\mathcal{P}}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

3 Answer Sets Computation

In this section, we describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational engine of the DLV system, which will be used for the experiments, but also other ASP systems, like Smodels, employ a very similar procedure.

An answer set program $\mathcal{P}$ in general contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$.[1] The hard part of the computation is then performed on this ground ASP program generated by the instantiator.

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** IsAnswerSet(I);
    Select an undefined ground atom A according to a heuristic;
    if ModelGenerator($I \cup \{A\}$) **then return** True;
    else return ModelGenerator($I \cup \{\texttt{not}\ A\}$);
end;

Figure 1: Computation of Answer Sets

The heart of the computation is performed by the Model Generator, which is sketched in Figure 1. Roughly, the Model Generator produces some "candidate" answer sets. The stability of each of them is subsequently verified by the function *IsAnswerSet(I)*, which verifies whether the given "candidate" I is a minimal model of the program $Ground(\mathcal{P})^I$ obtained by applying the GL-transformation w.r.t. I.

The ModelGenerator function is first called with parameter I set to the empty interpretation.[2] If the program $\mathcal{P}$ has an answer set, then the function returns True setting I to the computed answer set; otherwise it returns False. The Model Generator is similar to the Davis-Putnam procedure employed by SAT solvers. It first calls a function DetCons(), which extends I with the literals that can be deterministically inferred. This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If DetCons does not detect any inconsistency, an atom A is selected according to a heuristic criterion and ModelGenerator is called on $I \cup \{A\}$ and on $I \cup \{\texttt{not}\ A\}$. The atom A plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of

[1] Note that $ground(\mathcal{P})$ is not the full set of all syntactically constructible instances of the rules of $\mathcal{P}$; rather, it is a subset (often much smaller) of it having precisely the same answer sets as $\mathcal{P}$ [FLMP99].

[2] Observe that the interpretations built during the computation are 3-valued, that is a literal can be True, False or Undefined (if its value has not been set yet) w.r.t. to an interpretation I.

a "good" atom A is crucial for the performance of an ASP system. In the next section, we describe a number of heuristic criteria for the selection of such branching atoms.

4 Heuristics

Throughout this section, we assume that a ground ASP program $\mathcal{P}$ and an interpretation I have been fixed. Here, we describe the heuristic criteria that will be compared in Section 7. We consider "dynamic heuristics" (the ASP equivalent of UP heuristics for SAT), that is, branching rules where the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. Given a literal Q, $ext(Q)$ will denote the intepretation resulting from the application of DetCons (see previous section) on $I \cup \{Q\}$; w.l.o.g., we assume that $ext(Q)$ is consistent, otherwise Q is automatically set to false and the heuristic is not evaluated on Q at all.

Heuristic $\mathbf{h_1}$. This is an extension of the branching rule adopted in the system SATZ [LA97] – one of the most efficient SAT solvers – to the framework of ASP.

The *length* of a rule r (w.r.t. an interpretation J), is the number of undefined literals occurring in r. Let $Unsat_k(Q)$ denote the number of unsatisfied rules[3] of length k w.r.t. $ext(Q)$, which have a greater length w.r.t. I. In other words, $Unsat_k(Q)$ is the number of unsatisfied rules whose length shrinks to k if the truth of Q is assumed and propagated in the interpretation I. The weight $w_1(Q)$ is

$$w_1(Q) = \Sigma_{k>1}\, Unsat_k(Q) * 5^{-k}$$

Thus, the weight function w_1 prefers literals introducing a higher number of short unsatisfied rules. Intuitively, the introduction of a high number of short unsatisfied rules is preferred because it creates more and stronger constraints on the intepretation so that a contradiction can be found earlier [LA97]. We combine the weight of an atom Q with the weight of its complement $\texttt{not}\ Q$ to favour Q such that $w_1(Q)$ and $w_1(\texttt{not}\ Q)$ are roughly equal, to avoid that a possible failure leads to a very bad state. To this end, as in SATZ, we define the combined weight *comb*-$w_1(Q)$ of an atom Q as follows:

$$comb\text{-}w_1(Q) = w_1(Q) * w_1(\texttt{not}\ Q) * 1024 + w_1(Q) + w_1(\texttt{not}\ Q).$$

Given two atoms A and B, heuristic h_1 prefers B over A ($A <_{h1} B$) iff *comb*-$w_1(A) <$ *comb*-$w_1(B)$. Once the $<_{h1}$-maximum atom Q is selected, heuristic h_1 takes Q if $w_1(Q) > w_1(\texttt{not}\ Q)$, $\texttt{not}\ Q$ else.

Heuristic $\mathbf{h_2}$. The second heuristic we consider is inspired to the branching rule of Smodels – a well known ASP system [Sim00]. Let $|J|$ denote the number of atoms which are either true or false in a (three-valued) intepretation J. Then, define

$$w_2(Q) = |ext(Q)|.$$

Since w_2 maximizes the size of the resulting intepretation, it minimizes the atoms which are left undefined. Intuitively, this minimizes the size of the remaining search space (which is 2^u, where u is the number of undefined atoms in $ext(Q)$) [Sim00]. Similar to Smodels, the heuristic h_2 cautiously maximizes the minimum of $w_2(Q)$ and $w_2(\texttt{not}\ Q)$. More precisely, the preference relationship $<_{h2}$ of h_2 is defined as

[3]Recall that a rule r is satisfied w.r.t. J if the body of r is false w.r.t. J or the head of r is true w.r.t. J.

follows. Given two atoms A and B, $A <_{h2} B$ holds if $min(h_2(A), h_2(\texttt{not } A)) < min(h_2(B), h_2(\texttt{not } B))$; otherwise, $A <_{h2} B$ holds if $min(h_2(A), h_2(\texttt{not } A)) = min(h_2(B), h_2(\texttt{not } B))$ and $max(h_2(A), h_2(\texttt{not } A)) < max(h_2(B), h_2(\texttt{not } B))$. Once the $<_{h2}$-maximum atom Q is selected, heuristic h_2 takes either Q or $\texttt{not } Q$, depending on the same selection principle.

Remark. It is worthwhile noting that the heuristic of Smodels, while following the above intuition, is more advanced and sophisticated than h_2. Unfortunately, it is defined for non-disjunctive programs, and centered around properties of unstratified negation, which does not play an important role in our framework (and does not appear at all in any of the benchmark programs considered here). We do not see any immediate extension of Smodels' heuristic to the framework of disjunctive ASP programs. ■

Heuristic $\mathbf{h_3}$. Let us consider now the heuristic used in the DLV system. Even if this is more "naive" than the previous heuristics, we will benchmark it in order to evaluate the impact of changing the branching rule on the test system.

A peculiar property of answer sets is *supportedness*: For each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r. Since an ASP system must eventually converge to a supported interpretation, ASP systems try to keep the interpretations "as much supported as possible" during the intermediate steps of the computation. To this end, the DLV system counts the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule (further details on UTs can be found in [FLP99] where they are called MBTs). For instance, the rule $:- \ not\ x$ implies that x must be true in every answer set of the program; but it does not give a "support" for x. Thus, in the DLV system x is taken true to satisfy the the rule, and it is added to the set of UnsupportedTrue; it will be removed from this set once a supporting rule for x will be found (e.g., $x \vee b :- c$ is a supporting rule for x in the interpretation $I = \{x, \texttt{not } b, c\}$). Given a literal Q, let $UT(Q)$ be the number of UT atoms in $ext(Q)$. Moreover, let $UT_2(Q)$ and $UT_3(Q)$ be the number of UT atoms occurring, respectively, in the heads of 2 and 3 unsatisfied rules w.r.t. $ext(Q)$. The heuristic h_3 of DLV considers $UT(Q)$, $UT_2(Q)$ and $UT_3(Q)$ in a prioritized way, to favor atoms yielding interpretations with fewer $UT/UT_2/UT_3$ atoms (which should more likely lead to a supported model). If all UT counters are equal, then the heuristic considers the total number $Sat(Q)$ of rules which are satisfied w.r.t. $ext(Q)$. More precisely, given two atoms A and B, $A <_{h3} B$ if $UT(A) > UT(B)$; otherwise, $A <_{h3} B$ if $UT(A) = UT(B)$ and $UT_2(A) > UT_2(B)$; otherwise, $A <_{h3} B$ if $UT_2(A) = UT_2(B)$ and $UT_3(A) > UT_3(B)$; otherwise, $A <_3 B$ if $UT_3(A) = UT_3(B)$ and $Sat(A) < Sat(B)$. The $<_{h3}$-maximum atom is selected by the heuristic h_3 of DLV.

Thus, unlike the previous heuristics, the heuristic h_3 of DLV considers only atoms (instead of literals), and it does not take into account what happens when the selected atom Q leads to a failure (i.e., $ext(\texttt{not } Q)$ is not considered in the heuristic).

Heuristic $\mathbf{h_4}$. Finally, we have considered a simple "balanced version" h_4 of the heuristic h_3 of DLV, where also the complement of an atom is evaluated for the heuristic. Given an atom A, let $UT'(A) = UT(A) + UT(\texttt{not } A)$, $UT'_2(A) = UT_2(A) + UT_2(\texttt{not } A)$, $UT'_3(A) = UT_3(A) + UT_3(\texttt{not } A)$, and $Sat'(A) = Sat(A) + Sat(\texttt{not } A)$. The heuristic h_4 works precisely as h_3, but considers the primed counters. Once the best atom has been selected, it is taken positive or negative, depending on h_3.

5 Benchmark Programs

To evaluate the different heuristics presented in the previous section, we chose a couple of benchmark problems: 3SAT, Blocksworld Planning, Hamiltonian path, and Strategic Companies.

3SAT is one of the best researched problems in AI and generally used for solving many other problems by translating them to 3SAT, solving the 3SAT problem and transforming the solution back to the original domain:

> Let $\Phi = \bigwedge_{i=1}^{n}(d_{i,1} \vee \ldots \vee d_{i,3})$ be a propositional formula in conjunctive normal form (3CNF) where the $d_{i,j}$ are literals over the propositional variables $x_1, \ldots, x_m$.
>
> Φ is satisfiable, iff there exists a consistent conjunction I of literals such that $I \models \Phi$.

3SAT is a classical NP-complete problem and can be easily represented in our formalism as follows:

For every propositional variable x_i ($1 \leq i \leq$m), we add the following rule which ensures that we either assume that variable x_i or its complement nx_i true:

$$x_i \vee nx_i.$$

And for every clause $d_1 \vee \ldots \vee d_3$ in Φ we add the constraint

$$:- \texttt{not}\ \bar{d}_1, \ldots, \texttt{not}\ \bar{d}_3.$$

where $\bar{d}_i$ ($1 \leq i \leq 3$) is x_j if d_i is a positive literal x_j, and nx_j if d_i is a negative literal not x_j.

Hamiltonian Path (HAMPATH) is another classical NP-complete problem from the area of 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?

Suppose that the graph G is specified by using two predicates $node(X)$ and $arc(X, Y)$[4], and the starting node is specified by the predicate $start$ (unary) which contains only a single tuple. Then, the following program solves the problem HAMPATH.

% Each node has to be reached.
$:- node(X), \texttt{not}\ reached(X).$
$reached(X) :- start(X).$
$reached(X) :- inPath(Y, X).$
% Guess whether to take a path or not.
$inPath(X, Y) \vee outPath(X, Y) :- reached(X), arc(X, Y).$
% At most one incoming/outgoing arc!
$:- inPath(X, Y), inPath(X, Y1), Y{<>}Y1.$
$:- inPath(X, Y), inPath(X1, Y), X{<>}X1.$

Blocksworld (BW) is a classic problem from the planning domain, and one of the oldest problems in AI:

[4]Predicate arc is symmetric, since undirected edges correspond to bidirectional arcs.

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.

Figure 2 shows a simple example 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.

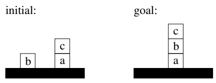


Figure 2: Simple BW Example

Due to space restrictions we refer to [Erd99, FLMP99] for a complete encoding.

Strategic Companies (STRATCOMP) finally, is a Σ_2^P-complete problem, which has been first described in [CEG97]:

> A holding owns companies $C(1), \ldots, C(c)$, each of which produces some goods. Some of these companies may jointly control another one.
>
> This is modelled by means of predicates $produced\_by(P, C1, C2)$ — product P is produced by companies $C1$ and $C2$ — and $controlled\_by(C, C1, C2, C3)$ — company C is jointly controlled by $C1$, $C2$ and $C3$.
>
> Now, some companies should be sold, under the constraint that all goods can be still produced, and that no company is sold which would still be controlled by the holding afterwards. A company is strategic, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints.

The answer sets of the following natural program correspond one to one to the strategic sets. Checking whether any given company C is strategic is done by brave reasoning: "Is there *any* answer set containing C?"

$$\begin{array}{l} strategic(C1) \vee strategic(C2) :- produced\_by(P, C1, C2). \\ strategic(C) :- controlled\_by(C, C1, C2, C3), \\ \qquad\qquad\qquad strategic(C1), strategic(C2), strategic(C3). \end{array}$$

As in [CEG97] we assume that each product is produced by at most two companies and each company is jointly controlled by at most three companies to allow for an easier representation.

6 Benchmark Data

For 3SAT, we have randomly generated 3CNF formulas over n variables (where n denotes the size as plotted on the x-axis of the graphs in Section 7) using a tool by Selman and Kautz [SK97]. For each size we generated 8 such instances, where we kept the ratio between the number of clauses and the number of variables at the cross-over point of 4.3.

The instances for HAMPATH were generated using a tool by Patrik Simons which has been used to compare Smodels against SAT solvers (cf. [Sim00]), and is available at `http://tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`.

	h_1	h_2	h_3	h_4
3SAT	270	270	250	260
HAMPATH	80	40	70	> 100
BW	$>$ P6	$>$ P6	P5	$>$ P6
STRATCOMP	> 1200	> 1200	< 700	> 1200

Table 1: Largest size where all instances were solved in less than 600 CPU secs

Again, for each problem size n we generated 8 instances, always assuming node 1 as the starting node.

The blocksworld problems P3 and P4 have been employed in in [Erd99] to compare ASP systems, and can be solved in 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 input.

For STRATCOMP, finally, we randomly generated 8 instances for each problem size n, with n companies and n products. Each company O is controlled by one to five companies (two groups of companies, where each of these groups controls the same company O, must have at least one member in common), where the actual number of companies is uniform randomly chosen. On average there are 1.5 $controlled\_by$ relations per company.

All experiments were performed on an Athlon/750 machine with 256MB of main memory running FreeBSD 4.2. The binaries were produced with GCC 2.95.2.

7 Experimental Results and Conclusion

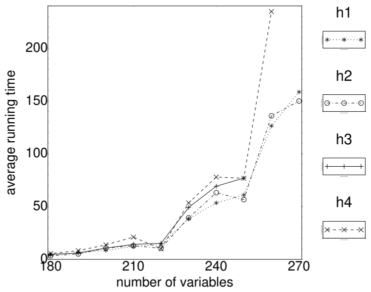
The results of our experiments are displayed in the graphs of Figures 7–7. In each graph, the horizontal axis reports a parameter representing the size of the instance. On the vertical axis, we report the average running time (expressed in seconds) over the 8 instances of the same size we have run (see previous section).

Remark. All heuristics have been implemented in a straightforward way, without optimizations. The running times reported in the graph are meaningful for comparing the relative efficiencies of the heuristics, but (all of them) can be drastically improved by optimizing the implementation. ∎

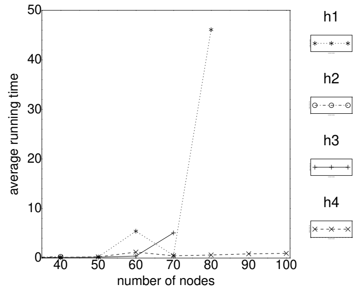
We have allowed a running time of 600 seconds for each problem instance. In the graphs, the line of an heuristic stops whenever some problem instance was not solved in the maximum allowed time. Table 1 displays, for each heuristic, the maximum instance-size where the heuristic could solve all problem instances in the maximum allowed time.

As expected, heuristic h_3, the "native" heuristic of the DLV system, which does not combine the heuristic values of complementary atoms, is the worst in most cases. It does not terminate on the instance P6 of BW, it could not solve any of the benchmark instances of STRATCOMP (it does not appear at all in Figure 7), and stopped earlier than the others on 3SAT.

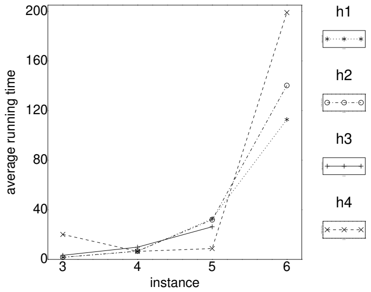
Heuristic h_1, the extension of SATZ heuristic to ASP, behaves very well on average. On 3SAT, BW, and STRATCOMP, h_1 could solve all benchmark instances we have run. It is the fastest on BW and one of the two fastest on 3SAT. It shows a negative behaviour only on HAMPATH. In this problem, considering UnsupportedTrue literals



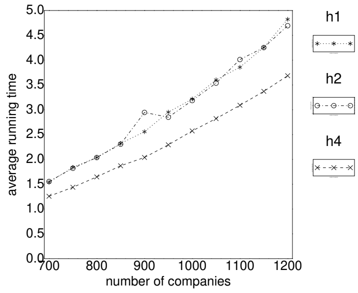
3SAT problems



Hamiltonian Path problems



Blocksworld problems



Strategic Companies

Figure 3: Average running times for benchmarks

seems to be crucial for the efficiency.

Heuristic h_4 is surprisingly good compared to h_3. It is a simple "balanced version" of heuristic h_3 (the heuristic values of the positive and of the negative literal are combined by sum). This simple extension to h_3 improves dramatically the performance. Indeed, heuristic h_4 solves nearly all instances we ran (only on 3SAT it stopped a bit earlier than other heuristics). It is the best heuristic on STRATCOMP and, importantly, on HAMPATH, where it beats all other heuristics of a relevant factor.

The behaviour of heuristic h_2, based on the minimization of the undefined atoms, is rather controversial. It behaves very well on 3SAT and BW, but it is extremely bad on HAMPATH, where it stops at 40 nodes already and is beated even from the "naive" heuristic h_3. This confirms that further studies are needed to find a proper extension of the heuristic of Smodels to the framework of disjunctive ASP.

Concluding, we observe that both heuristic h_1 and heuristic h_4, significantly improve the efficiency of the native heuristic h_3 of the `DLV` system. The dramatic improvement obtained by the simple change from h_3 to h_4, confirms even more the importance of a careful study of branching rules in ASP systems. This paper is only a first step in this field, we hope that further works will follow, for proposing new heuristics for ASP and for better understanding the existing ones, in order to improve the efficiency of ASP systems.

References

[CEG97] Marco Cadoli, Thomas Eiter, and Georg Gottlob. Default Logic as a Query Language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):448–463, May/June 1997.

[EETW00] Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proceedings National Conference on AI (AAAI'00)*, pages 417–422, Austin, Texas, July 30 – August 3 2000. AAAI Press.

[EFLP00] 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.

[EGM97] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[Erd99] Esra Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[ET00] 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.

[FLMP99] 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.

[FLP99] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pushing Goal Derivation in DLP Computations. In Michael Gelfond, Nicola Leone, and Gerald Pfeifer, editors, *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, number 1730 in Lecture Notes in AI (LNAI), pages 177–191, El Paso, Texas, USA, December 1999. Springer Verlag.

[Fre95] J.W. Freeman. *Improvements on Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[HV95] J.N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[Ilk99] Ilkka Niemelä. Logic programming with stable model semantics as constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[LA97] C.L. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fourteen International Joint Conference on Artificial Intelligence (IJCAI) 1997*, pages 366–371, Nagoya, Japan, August 1997.

[Lif96] Vladimir Lifschitz. Foundations of logic programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, Stanford, 1996.

[Lif99] 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.

[MT98] 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.

[RSS+97] 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.

[Sim00] Patrik Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[SK97] Bart Selman and Henry Kautz, 1997. `ftp://ftp.research.att.com/dist/ai/`.