`dcs`: An Implementation of DATALOG with Constraints

Deborah East and **Mirosław Truszczyński**
Department of Computer Science
University of Kentucky
Lexington KY 40506-0046, USA
email: deast|mirek@cs.uky.edu

Abstract

Answer-set programming (ASP) has emerged recently as a viable programming paradigm. We describe here an ASP system, *DATALOG with constraints* or DC, based on non-monotonic logic. Informally, DC theories consist of propositional clauses (constraints) and of Horn rules. The semantics is a simple and natural extension of the semantics of the propositional logic. However, thanks to the presence of Horn rules in the system, modeling of transitive closure becomes straightforward. We describe the syntax, use and implementation of DC and provide experimental results.

General Info

DC is an answer set programming (ASP) system (MT99) similar to propositional logic but extended to include Horn clauses. The semantics of DC is a natural extension of the semantics of propositional logic. The DC system is implemented in two modules, `ground` and `dcs`, which are written in the 'C' programming language and compiled with `gcc`. There are approximately 2500 lines of code for `ground` and approximately 1500 for `dcs`. DC has been implemented on both SUN SPARC and a PC running linux.

A DC theory (or program) consists of constraints and Horn rules (DATALOG program). This fact motivates our choice of terminology — DATALOG with constraints. We start a discussion of DC with the propositional case. Our language is determined by a set of atoms At. We will assume that At is of the form $At = At_C \cup At_H$, where At_C and At_H are disjoint.

A DC theory (or program) is a triple $T_{dc} = (T_C, T_H, T_{PC})$, where
1. T_C is a set of propositional clauses $\neg a_1 \vee \ldots \vee \neg a_m \vee b_1 \vee \ldots \vee b_n$ such that all a_i and b_j are from At_C,
2. T_H is a set of Horn rules $a_1 \wedge \ldots \wedge a_m \rightarrow b$ such that $b \in At_H$ and all a_i are from At,
3. T_{PC} is a set of clauses over At.

By $At(T_{dc})$, $At_C(T_{dc})$ and $At_{PC}(T_{dc})$ we denote the set of atoms from At, At_C and At_{PC}, respectively, that actually appear in T_{dc}.

With a DC theory $T_{dc} = (T_C, T_H, T_{PC})$ we associate a family of subsets of $At_C(T_{dc})$. We say that a set $M \subseteq At_C(T_{dc})$ *satisfies* T_{dc} (is an *answer set* of T_{dc}) if
1. M satisfies all the clauses in T_C, and
2. the closure of M under the Horn rules in T_H, $M^c = LM(T_H \cup M)$ satisfies all clauses in T_{PC} ($LM(P)$ denotes the least model of a Horn program P).

Intuitively, the collection of clauses in T_C can be thought of as a representation of the constraints of the problem, Horn rules in T_H can be viewed as a mechanism to compute closures of sets of atoms satisfying the constraints in T_C, and the clauses in T_{PC} can be regarded as constraints on closed sets (we refer to them as *post-constraints*). A set of atoms $M \subseteq At_C(T_{dc})$ is a model if it (propositionally) satisfies the constraints in T_C and if its closure (propositionally) satisfies the post-constraints in T_{PC}. Thus, the semantics of DC retains much of the simplicity of the semantics of propositional logic. DC was introduced by the authors in (ET00).

Applicability of the System

DC can be used as a computational tool to solve search problems. We define a search problem Π to be determined by a set of finite *instances*, D_Π, such that for each instance $I \in D_\Pi$, there is a finite set $S_\Pi(I)$ of all *solutions* to Π for the instance I. For example, the problem of finding a hamilton cycle in a graph is a search problem: graphs are instances and for each graph, its hamilton cycles (sets of their edges) are solutions.

A DC theory $T_{dc} = (T_C, T_H, T_{PC})$ *solves* a search problem Π if solutions to Π can be computed (in polynomial time) from answer sets to T_{dc}. Propositional logic and stable logic programming (MT99; Nie98) are used as problem solving formalisms following the same general paradigm.

We will illustrate the use of DC to solve search problems by discussing the problem of finding a hamilton cycle in a directed graph. Consider a directed graph G with the vertex set V and the edge set E. Consider a set of atoms $\{hc(a,b) \colon (a,b) \in E\}$. An intuitive interpretation of an atom $hc(a,b)$ is that the edge (a,b) is in a hamilton cycle. Include in T_C all clauses of the form $\neg hc(b,a) \vee \neg hc(c,a)$, where $a,b,c \in V$, $b \neq c$ and $(b,a),(c,a) \in E$. In addition, include in T_C all clauses of the form $\neg hc(a,b) \vee \neg hc(a,c)$, where $a,b,c \in V$,

$b \neq c$ and $(a, b), (a, c) \in E$. Clearly, the set of propositional variables of the form $\{hc(a, b) \colon (a, b) \in F\}$, where $F \subseteq E$, satisfies all clauses in T_C if and only if no two different edges in F end in the same vertex and no two different edges in F start in the same vertex. In other words, F spans a collection of paths and cycles in G.

To guarantee that the edges in F define a hamilton cycle, we must enforce that all vertices of G are reached by means of the edges in F if we start in some (arbitrarily chosen) vertex of G. This can be accomplished by means of a simple Horn program. Let us choose a vertex, say s, in G. Include in T_H the Horn rules $hc(s, t) \rightarrow vstd(t)$, for every edge (s, t) in G. In addition, include in T_H Horn rules $vstd(t), hc(t, u) \rightarrow vstd(u)$, for every edge (t, u) of G not starting in s. Clearly, the least model of $F \cup T_H$, where F is a subset of E, contains precisely these variables of the form $vstd(t)$ for which t is reachable from s by a *nonempty* path spanned by the edges in F. Thus, F is the set of edges of a hamilton cycle of G if and only if the least model of $F \cup T_H$, contains variable $vstd(t)$ for every vertex t of G. Let us define $T_{PC} = \{vstd(t) \colon t \in V\}$ and $T_{ham}(G) = (T_C, T_H, T_{PC})$. It follows that hamilton cycles of G can be reconstructed (in linear time) from answer sets to the DC theory $T_{ham}(G)$. In other words, to find a hamilton cycle in G, it is enough to find an answer set for $T_{ham}(G)$.

This example illustrates the simplicity of the semantics of DC — it is only a slight adaptation of the semantics of propositional logic to the case when in addition to propositional clauses we also have Horn rules in theories. It also illustrates the power of DC to generate concise encodings. All known propositional encodings of the hamilton-cycle problem require that additional variables are introduced to "count" how far from the starting vertex an edge is located. Consequently, propositional encodings are much larger and lead to inefficient solutions to the problem.

Description of the System

In this section we will discuss general features of DC. First, we will discuss the language for encoding problems and give an example by showing the encoding of the hamiltonicity problem. Second, we will describe how we execute the DC system. Third, we will give some details concerning the implementation of `dcs`, our solver. Last we discuss the expressitivity of DC.

In the previous section we described the logic of DC in the propositional case. Our definitions can be extended to the predicate case (without function symbols). Each constraint is treated as an abbreviation of a set of its ground substitutions. This set is determined by the set of constants appearing in the theory and by additional conditions associated with the constraint (we illustrate this idea later in this section). When constructing predicate *DC*-based solutions to a problem Π we separate the representation of an instance to Π from the constraints that define Π. The representation of an instance of Π is a collection of facts or the extensional database (EDB). The constraints and rules that define Π will be referred to as the intensional database (IDB) and the language for writing the problem descriptions that constitute the IDB will be referred to as L_{dc}. The separation of IDB and EDB means only one predicate description of Π is needed.

The modules of DC, `ground` and `dcs` provide a complete system for describing and finding solutions to problems. An IDB in L_{dc} along with a specific EDB are the input to `ground`. A grounded propositional theory T_{dc} is output by `ground` and used as input to `dcs`.

A problem description in L_{dc} defines predicates, declares variables, and provides a description of the problem using rules. The predicates are defined using types from the EDB. Similarly, the variables are declared using types from the EDB. The rules consist of constraints, Horn rules and post-constraints. The constraints and post-constraints use several constructs to allow a more natural modeling. These constructs could be directly translated to clauses. (We use them as shorthands to ensure the conciseness of encodings.)

We present here a brief discussion of the constraints, Horn rules and post-constraint in L_{dc} and their meanings. Let $PRED$ be the set of predicates occurring in the IDB. For each variable X declared in the IDB the range $R(X)$ of X is determined by the EDB.

Select$(n, m, \vec{Y}; p_1(\vec{X}), \ldots, p_i(\vec{X}, \vec{Y}))q(\vec{X}, \vec{Y})$**,** where n, m are nonnegative integers such that $n \leq m, q \in PRED$ and $p_1, \ldots, p_i$ are EDB predicates or logical conditions (logical conditions can be comparisons of arithmetic expressions or string comparisons). The interpretation of this constraint is as follows: for every $\vec{x} \in R(\vec{X})$ at least n atoms and at most m atoms in the set $\{q(\vec{x}, \vec{y}) : \vec{y} \in R(\vec{Y})\}$ are true.

Select$(n, m, \vec{Y})q(\vec{X}, \vec{Y})$**,** where n, m are nonnegative integers such that $n \leq m, q \in PRED$. The interpretation of this constraint is as follows: for every $\vec{x} \in R(\vec{X})$ at least n atoms and at most m atoms in the set $\{q(\vec{x}, \vec{y}) : \vec{y} \in R(\vec{Y})\}$ are true.

Select$(n, m)q_1(\vec{X}), \ldots, q_j(\vec{X})$**,** where n, m are nonnegative integers such that $n \leq m, q_1, \ldots, q_j \in PRED$. The interpretation of this constraint is as follows: for every $\vec{x} \in R(\vec{X})$ at least n atoms and at most m atoms in the set $\{q_1(\vec{x}), \ldots, q_j(\vec{x})\}$ are true.

Certain choices of n, m in any of the Select constraints allow construction of even more specific constraints in T_{dc}:
$n = m$ exactly n atoms must be true.
$n = 0$ at most m atoms must be true.
$m = 999$ at least n atoms must be true.

NOT $q_1(\vec{X}), \ldots, q_i(\vec{X})$**,** where $q_1, \ldots, q_i \in PRED$. For every $\vec{x} \in R(\vec{X})$ at least one atom in the set $\{q_1(\vec{x}), \ldots, q_i(\vec{x})\}$ must be false.

$q_1(\vec{X})| \ldots |q_i(\vec{X})$**,** where $q_1, \ldots, q_i \in PRED$. For every $\vec{x} \in R(\vec{X})$ at least one atom in the set

```
An example of a graph file used as EDB
vtx(1).
vtx(2).
vtx(3).
edge(1,3).
edge(3,2).
```

Figure 1: Format for EDB.

$\{q_1(\vec{x}), \ldots, q_i(\vec{x})\}$ must be true.

$p_1(\vec{X}), \ldots, p_i(\vec{X}) \rightarrow q_1(\vec{X}) | \ldots | q_j(\vec{X})$**,** where $p_1, \ldots, p_i, q_1, \ldots, q_j \in PRED$. For every $\vec{x} \in R(\vec{X})$ if all atoms in the set $\{p_1(\vec{x}), \ldots, p_i(\vec{x})\}$ are true then at least one atom in the set $\{q_1(\vec{x}), \ldots, q_j(\vec{x})\}$ must be true. (This constraint represents standard propositional logic clauses.)

$p_1(\vec{X}), \ldots, p_i(\vec{X}) \rightarrow q_1(\vec{X}), \ldots, q_j(\vec{X})$**,** where $p_1, \ldots, p_i, q_1, \ldots, q_j \in PRED$. For every $\vec{x} \in R(\vec{X})$ if all atoms in the set $\{p_1(\vec{x}), \ldots, p_i(\vec{x})\}$ are true then all atoms in the set $\{q_1(\vec{x}), \ldots, q_j(\vec{x})\}$ must be true.

Horn $p_1(\vec{X}), \ldots, p_i(\vec{X}) \rightarrow q_1(\vec{X}), \ldots, q_j(\vec{X})$**,** where $p_1, \ldots, p_i, q_1, \ldots, q_j \in PRED$.

Methodology

Here we show the encoding of a problem in DC. We will use the example of hamiltonicity of a graph which we discussed previously. Figure 1 shows the EDB format used by `ground`. This format is compatible to that used by `smodels` and others. However, we *require* that the EDB be in separate files from the IDB. The format for the EDB allows data to be entered as sets, ranges, or individual elements and constant values can be entered on the command line.

The IDB provides a definition of the problem in L_{dc}. The IDB file has three parts. First a definition of the predicates, next the declaration of variables and last a set of constraints and Horn clauses. The types used in the IDB must be in the data file(s). For example, the only data types in the graph file (Fig. 1) are **vtx** and **edge**. Thus the only data types which can be used in the IDB are **vtx** and **edge**.

In the **idbpred** section of Fig. 2, we define two predicates, the *vstd* predicate and the *hc* predicate. The *vstd* predicate has one parameter of type *vtx* and *hc* has two parameters both of type *vtx*.

The **idbvar** section of Fig. 2 declares two variables X, Y both of type *vtx*.

The section containing the constraints, Horn clauses, and post-constraints is proceeded by the keyword **idbrules** (see Fig.2). The order in which the rules are entered is not important. The first constraint, $Select(1, 1, Y; edge(X, Y))hc(X, Y).$, ensures that each vertex has exactly one outgoing edge. The second constraint, $Select(1, 1, X; edge(X, Y))hc(X, Y).$, requires that each vertex has exactly one incoming edge.

```
An example of a file used as IDB
% comments begin with percent sign
idbpred % section for defining predicates
vstd(vtx).
hc(vtx,vtx).
idbvar % section for declaring variables
vtx X,Y.
idbrules % rule section
% constraints
Select(1,1,Y;edge(X,Y)) hc(X,Y).
Select(1,1,X;edge(X,Y)) hc(X,Y).
% Horn rules
Horn Forall(X,Y;X!=i,edge(X,Y))
vstd(X), hc(X,Y) → vstd(Y).
Horn Forall(X,Y;X==i,edge(X,Y))
hc(X,Y) → vstd(Y).
% post-constraints
vstd(X).
```

Figure 2: File showing IDB for the hamiltonicity of a graph. The types are from Fig. 1

The first Horn rule ranges over all X, Y such that $edge(X, Y) \in EDB$ and $X \neq i$ where i is a constant used to initialize $vstd(i)$. This rule requires both $vstd(X)$ and $hc(X, Y)$ to be true before we can assign the value true to $vstd(Y)$. The second Horn rule only requires $hc(X, Y)$ to be true before $vstd(Y)$ is assigned value true; however, in this rule X is restricted to the value of i.

The last line is a post-constraint that requires $vstd(X)$ to be true for all $X \in R(X)$ ensuring that the cycle is closed.

Running the system

Here we will describe the steps for execution of the DC system. The first module of DC, `ground`, has as input a data file(s), a rule file and command line arguments. Output is the theory file which will be used as input to `dcs`. The theory is written to a file whose name is the catenation of the constants and file names given as command line arguments. The extension **.tdc** is then appended to the output file name. The command line arguments for `ground`:

ground -r rf -d df [-c label=v] [-V]

Required arguments

-r **rf** is the file describing the problem. There must be exactly one rule file.

-d **df** must be one or more files containing data that will be used to instantiate the theory. It is often convenient to use more than one file for data.

Optional arguments

-c This option allows use of constants in both data and rule files. When **label** is found while reading either file it is replaced by **v**. **v** can be any string that is valid for the data type. If **label** is to be used in a range then **v** must be an integer. For example, if the data file contains the entry **queens[1..q].** then we can define the constant **q** with the option **-c q=8**. If more than one constant is needed then **-c b=3 n=14** defines both constants **b,n** using the **-c** option.

-V The verbose options sends output to stdout during the execution of **ground**. This output may be useful for debugging of the data or rule files.

For the example of hamiltonicity we could have a data file (see Fig. 1) named **1.gph** and an IDB file (see Fig. 2) named **hcp**. The constant **i** is needed in the IDB file to initialize the first vertex in the graph. The command line argument would be:

ground -r hcp -d 1.gph -c i=1

The theory file produced would be named **1_1.gph_hcp.tdc**.

The second module of the DC system, `dcs`, has as input the theory file produced by `ground`. A file named dcs.stat is created or appended with statistics concerning the results of executing `dcs` on the theory. The command line arguments are:

dcs -f filename [-A] [-P] [-C] [-V]

Required arguments

-f **filename** is the name of the file containing a theory produced by `ground`.

Optional arguments

-A Prints the positive atoms for solved theories in readable form.

-P Prints the input theory and then exits.

-C Counts the number of solutions. This information is recorded in the statistics file.

-V Prints information during execution (branching, backtracking, etc). Useful for debugging.

Discussion of `dcs`

The DC solver uses a Davis-Putnam type approach, with backtracking, propagation and LookAHead to deal with constraints represented as clauses, *select* constraints and Horn rules, and to search for answer sets. The LookAHead in DC is similar to local processing performed in `csat` (DABC96). However, we use different methods to determine how many literals to consider in the LookAHead phase. Other techniques, especially propagation and search heuristics, were designed specifically for the case of DC as they must take into account the presence of Horn rules in programs.

Propagation consists of methods to reduce the theory. Literals which appear in the heads of Horn rules, $l_h \in At_H$ require different interpretations. A literal is a l_h if and only if it appears in the head of a Horn rule. We can not guess an assignment for l_h rather it must be computed. We can only assign value true to l_h if it appears in the head of a Horn rule for which all literals in the body of the Horn rule have been assigned the value true. If one or more literals in the body of a Horn rule have been assigned the value false then that rule is removed. If l_h has not been assigned a value and all Horn rules in which l_h appeared in the head have been removed then l_h is assigned the value false. If we have a post-constraint that required a value be assigned to l_h and the value cannot be computed then we must backtrack.

Non Horn rules are constraints which must be satisfied. These rules are identified by tags which indicate which method is needed to evaluate the constraint during propagation.

The LookAHead procedure tests a number of literals not yet assigned a value. The LookAHead procedure is similar to the local processing procedure used for `csat` (DABC96). A literal is chosen, assigned the value true, then false using propagation to reduce and evaluate the resulting theories. If both evaluations of assignments result in conflicts then we return false and backtracking will result. If only one evaluation results in conflict then we can assign the literal the opposite value and continue the LookAHead procedure. If neither evaluation results in conflict we cannot make any assignments, but we save information (number of forced literals and satisfied constraints) computed during the evaluations of the literal.

The number of literals to be tested has been determined empirically. It is obvious that if all unassigned literals were tested during each LookAHead it would greatly increase the time. However, if only a small number of literals are to be tested during each LookAHead then they must be chosen to provide the best chance of reducing the theory. To choose the literals with the best chance of reducing the theory, we order the unassigned literals based on a sum computed by totaling the weights of the unsatisfied constraints in which they appear. The constraint weight is based on both the length and type of constraint. The shorter the constraint the larger the weight and when literals are removed the weight is recomputed. The literals are tested in descending order of the sum of constraints weights. Using this method we need to test only a very small number of literals during each LookAHead to obtain good results.

At the completion of the LookAHead procedure, we use the information computed during the evaluation of literals to choose the next branching literal and its initial truth assignment.

Expressitivity

The expressive power of DC is the same as that of logic programming with the stable-model semantics. The following theorem is presented in (ET00)

Theorem 1 *The expressive power of* DC *is the same as that of stable logic programming. In particular, a decision problem* Π *can be solved uniformly in* DC *if and only if* Π *is in the class* NP.

B-N	csat	dcs	smodel
b-n	sec	sec	sec
3-13	0.03	0.00	0.12
3-14	0.05	0.00	0.16
4-14	0.05	0.01	0.23
4-43	0.59	1.91	5.23
4-44	1.95	51.04	5.55
4-45	1599.92	226.44	12501.00

Table 1: Schur problem; times

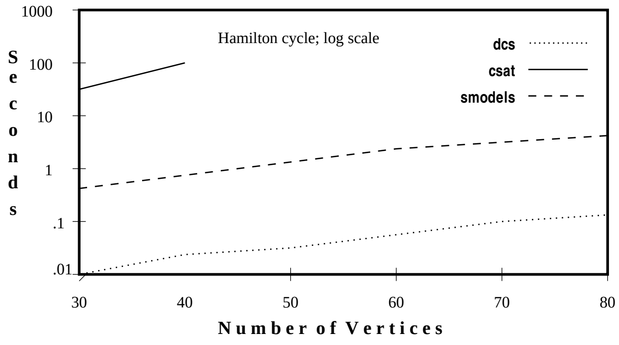

Figure 3: Results of computing hamilton cycles;log scale

Evaluating the System

The DC system provides a language, L_{dc}, which facilitates writing problem descriptions. Once an IDB is written in L_{dc} it can be used for any instance of the problem for which data, in the EDB format, is available or can be generated. It is possible to add constraints to IDB for a given problem when new requirements or constraints occur. The constructs used in L_{dc} allow for a natural description of constraints. Users need only know enough about a specific problem to be able to describe the problem in L_{dc} (there is a user's manual with examples). To help with programming in DC `ground` provides error messages and compiling information that are useful for debugging the IDB.

Benchmarks

The DC system has been executed using problems from NP, combinatorics, and planning. In particular, it has been used to compute hamilton cycles and colorings in graphs, to solve the N-queens problem, to prove that the pigeonhole problem has no solution if the number of pigeons exceeds the number of holes, and to compute Schur numbers.

The instances for computing hamilton cycles were obtained by randomly generating one thousand directed graphs with $v = 30, 40 \ldots, 80$ and density such that $\approx 50\%$ of the graphs contained hamilton cycles.

The graphs for instances of coloring were one hundred randomly generated graphs for $v = 50, 100, \ldots, 300$ with density such that $\approx 50\%$ had solutions when encode as 3-coloring.

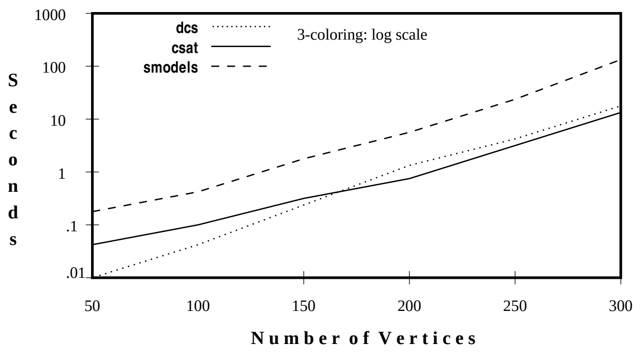

Figure 4: Results on encodings of coloring problems;log scale

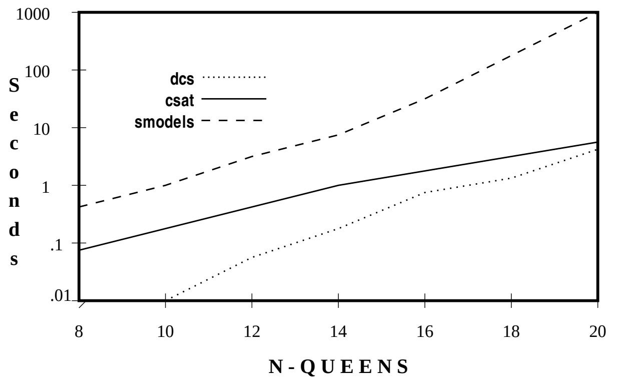

Figure 5: N-queens problem;log scale

Comparison

We have used the benchmarks to compare DC with systems based on stable model semantics and satisfiability. The performance of DC solver `dcs` was compared with `smodels`, a system for computing stable models of logic programs (NS96), and `csat`, a systems for testing propositional satisfiability (DABC96). In the case of `smodels` we used version 2.24 in conjunction with the grounder `lparse`, version 0.99.41 (Nie98). The extended rules (Sim99) allowed in the newer versions of `smodels` and `lparse` were used where applicable. The programs were all executed on a Sun SPARC Station 20. For each test we report the cpu user times for processing the corresponding propositional program or theory. We tested all three system using the benchmarks discussed in the previous section.

Note that `csat` performs comparable to DC for pigeonhole (see Fig. 6), N-queens (see Fig. 5) and coloring (see Fig. 4). These are problems where DC en-

Theory	Vertices	Edges	Atoms	Clauses
DC	30	130	160	220
smodels	30	130	1212	621
SAT	30	130	900	27960

Table 2: Difference in size of theories for hamilton cycle

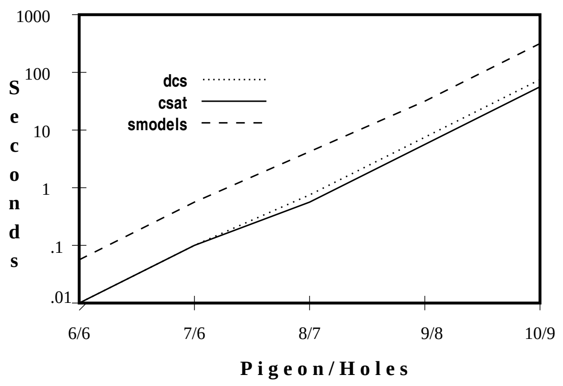

Figure 6: Pigeonhole problem ;log scale

codings do not use Horn rules (in the examples here only encodings for computing hamilton cycles use Horn rules). The closure properties of Horn rules allow for much smaller theories as shown in Table 2. The satisfiability theories for computing hamilton cycles are so large that they were not practical to execute for over 40 vertices (see Fig. 3). `smodels` performs much better than satisfiability solvers for computing the hamilton cycles although not as well as DC. The results for computing Schur numbers (see Table 1) also show much better results for DC.

Experimental results show that `dcs` often outperforms systems based on satisfiability as well as systems based on non-monotonic logics, and that it constitutes a viable approach to solving problems in AI, constraint satisfaction and combinatorial optimization. We believe that our focus on short encodings (see Fig. 2) is the key to the success of `dcs`.

References

[DABC96] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. Sat versus unsat. *DIMACS Cliques, Coloring and Satisfiability*, 26, 1996.

[ET00] D. East and M. Truszczyński. DATALOG with Constraints. In *Proccedings of the Seventeenth National Conference on Artificial Intelligence(AAAI-2000)*, July 2000.

[MT99] V.W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Nie98] I. Niemela. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, 1998.

[NS96] I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of JICSLP-96.* MIT Press, 1996.

[Sim99] P. Simons. Extending the stable model semantics with more expressive rules. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of 5th International Conference, LPNMR '99*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 305–316. Springer Verlag, 1999.