2001

DATALOG WITH CONTRAINTS: A NEW ANSWER-SET PROGRAMMING FORMALISM

Deborah J. East
University of Kentucky, deast@cs.uky.edu

Click here to let us know how access to this document benefits you.

Recommended Citation
East, Deborah J., "DATALOG WITH CONTRAINTS: A NEW ANSWER-SET PROGRAMMING FORMALISM" (2001).
University of Kentucky Doctoral Dissertations. 322.
https://uknowledge.uky.edu/gradschool_diss/322

This Dissertation is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Doctoral Dissertations by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.
ABSTRACT OF DISSERTATION

Deborah Jeanine East

The Graduate School
University of Kentucky
2001
DATALOG WITH CONSTRAINTS: A NEW ANSWER-SET PROGRAMMING FORMALISM

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Kentucky

By
Deborah Jeanine East
Winchester, Kentucky
Director: Dr. Miroslaw Truszczynski, Professor of Computer Science
Lexington, Kentucky
2001
ABSTRACT OF DISSERTATION

DATALOG WITH CONSTRAINTS: A NEW ANSWER-SET PROGRAMMING FORMALISM

Knowledge representation and search are two fundamental areas of artificial intelligence. Knowledge representation is the area of artificial intelligence which deals with capturing, in a formal language, the properties of objects and the relationships between objects. Search is a systematic examination of all possible candidate solutions to a problem that is described as a theory in some knowledge representation formalism.

We compare traditional declarative programming formalisms such as PROLOG and DATALOG with answer-set programming formalisms such as logic programming with stable model semantics. In this thesis we develop an answer-set formalism which we call DC. The logic of DC is based on the logic of propositional schemata and a version of Closed World Assumption. Two important features of the DC logic is that it supports modeling of the cardinalities of sets and Horn clauses. These two features facilitate modeling of search problems.

The DC system includes an implementation of a grounder and a solver. The grounder for the DC system grounds instances of problems retaining the structure of the cardinality of sets. The resulting theories are thus more concise. In addition, the solver for the DC system utilizes the structure of cardinality of sets to perform more efficient search. The second feature, Horn clauses, are used when transitive closure will eliminate the need for additional variables. The semantics of the Horn clauses are retained in the grounded theories. This also results in more concise theories.

Our goal in developing DC is to provide the computer science community with a system which facilitates modeling of problems, is easy to use, is efficient and captures the class of problems in NP-search.
We show experimental results comparing DC to other systems. These results show that DC is always competitive with state-of-the-art answer-set programming systems and for many problems DC is more efficient.

Keywords: knowledge representation, search, declarative programming, answer-set programming, NP-search, constraint satisfaction
DATALOG WITH CONSTRAINTS: A NEW ANSWER-SET PROGRAMMING FORMALISM

By
Deborah Jeanine East

Dr. Miroslaw Truszczynski
Director of Dissertation

Grzegorz Wasilkowski
Director of Graduate Studies

June 14, 2001
RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Doctor’s degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the dissertation in whole or in part also requires the consent of the Dean of The Graduate School of the University of Kentucky.
DISSErTATION

Deborah Jeanine East

The Graduate School
University of Kentucky
2001
DATALOG WITH CONSTRAINTS: A NEW ANSWER-SET PROGRAMMING FORMALISM

DISSERTATION

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the University of Kentucky

By
Deborah Jeanine East
Winchester, Kentucky

Director: Dr. Mirosław Truszczyński, Professor of Computer Science
Lexington, Kentucky
2001
# Table of Contents

List of Tables ................................................................. v  
List of Figures ............................................................... vi  
List of Files ................................................................. vii  

Chapter 1: Introduction .......................................................... 1  
1.1 Motivation ................................................................. 5  
1.2 Contributions ........................................................... 6  
1.3 Organization ............................................................. 6  

Chapter 2: Preliminaries .......................................................... 7  
2.1 Language of Propositional Logic ........................................ 7  
2.2 Language of Predicate Logic ........................................... 8  

Chapter 3: Answer-set Programming ........................................... 15  
3.1 Declarative programming ............................................... 15  
3.2 Goals of answer-set programming systems ........................... 20  

Chapter 4: DATALOG with constraints ........................................ 22  
4.1 DC– in the propositional case .......................................... 22  
4.2 DC– in the predicate case .............................................. 25  

Chapter 5: Environment of DC ............................................... 32  
5.1 Language ................................................................. 32  
5.1.1 Data Input ............................................................ 32  
5.1.2 Program File ........................................................ 33  
5.2 Wire routing – a modeling example ................................... 35  
5.3 Grounding ............................................................... 39  
5.4 DATALOG with constraints solver (des) .............................. 40  

Chapter 6: Implementation .................................................... 41  
6.1 Grounding ............................................................... 41  
6.2 Solving grounded DC theories ......................................... 43  
6.2.1 Preliminary steps .................................................... 44  
6.2.2 Propagation .......................................................... 44  
6.2.3 Branching Atoms .................................................... 45  
6.2.4 Lookahead ........................................................... 46  

Chapter 7: Experimentation .................................................... 47  
7.1 ASP systems ............................................................. 47  
7.2 Generating random graphs ............................................. 48  
7.3 Results for selected problems .......................................... 48  
7.3.1 Hamiltonian Cycle .................................................. 48  
7.3.2 Graph Coloring ...................................................... 49  
7.3.3 N-Queens ........................................................... 50
7.3.4 Pigeonhole Problem ........................................ 51
7.3.5 Schur Numbers ........................................... 52
7.3.6 $3 \times 3$ Magic Square ................................. 53
7.3.7 Wire Routing ............................................. 53
7.4 Summary of Results ........................................ 55

Chapter 8: Conclusions ...................................... 56
8.1 Meeting the goals of answer-set programming .............. 56
8.2 Future work ................................................ 56

Appendix A: Theories ......................................... 58
A.1 Hamiltonian cycle problem .................................. 58
  A.1.1 \textit{DC} theory for Hamiltonian cycle .................. 58
  A.1.2 smodels theory for Hamiltonian cycle ................. 58
  A.1.3 SAT description of Hamiltonian cycle .................. 59
A.2 \textit{k}-coloring of graphs .................................. 60
  A.2.1 \textit{DC} theory for \textit{k}-coloring ....................... 60
  A.2.2 smodels theory for \textit{k}-coloring ...................... 60
  A.2.3 P\textit{S}grnd theory for \textit{k}-coloring .................... 60
A.3 \textit{n}-queens ................................................ 61
  A.3.1 \textit{DC} theory for \textit{n}-queens .......................... 61
  A.3.2 smodels theory for \textit{n}-queens ......................... 61
  A.3.3 P\textit{S}grnd theory for \textit{n}-queens ....................... 62
A.4 Schur numbers .............................................. 62
  A.4.1 \textit{DC} theory for Schur numbers ...................... 63
  A.4.2 smodels theory for Schur numbers ....................... 63
  A.4.3 P\textit{S}grnd theory for Schur numbers .................... 63
A.5 $3 \times 3$ Magic Square .................................... 64
  A.5.1 \textit{DC} theory for $3 \times 3$ Magic Square ............. 64
  A.5.2 smodels theory for $3 \times 3$ Magic Square .......... 64
A.6 Pigeon-Hole Problem ....................................... 65
  A.6.1 \textit{DC} theory for pigeon-hole problem ................ 65
  A.6.2 smodels theory for pigeon-hole problem ................. 66
  A.6.3 P\textit{S}grnd theory for pigeon-hole problem .............. 66
A.7 Wire Routing ............................................... 66
  A.7.1 \textit{DC} theory for Wire Routing ......................... 66
  A.7.2 smodels theory for Wire Routing ......................... 68
  A.7.3 P\textit{S}grnd theory for Wire Routing ...................... 69

Bibliography .................................................. 70
Vita ......................................................... 73
List of Tables

Table 7.1  N-queens All solutions .................................................. 51
Table 7.2  Schur numbers ............................................................... 52
Table 7.3  $3 \times 3$ Magic Square ..................................................... 53
Table 7.4  Wire Routing ................................................................. 54
Table 7.5  Wire Routing ................................................................. 54
List of Figures

Figure 3.1  Graph with 3 vertices and 2 edges. ........................................ 16
Figure 5.1  Example of a $10 \times 10$ grid representing a chip ........................... 36
Figure 5.2  Solution for wire routing example with wandering paths ..................... 37
Figure 5.3  Solution using the additional clauses for eliminating cycles and
restricting search space .............................................................. 38
Figure 6.1  Instantiate .............................................................................. 41
Figure 6.2  ProcessForall ........................................................................ 42
Figure 6.3  ProcessExist .......................................................................... 43
Figure 7.1  Results of computing Hamilton cycles on random graphs ................... 49
Figure 7.2  Results of determining 3-color-ability random graphs .......................... 50
Figure 7.3  N-queens problem; time (seconds) ................................................. 51
Figure 7.4  Pigeonhole problem ................................................................. 52
List of Files

deast-06-01.pdf .pdf 399 kb
Chapter 1

Introduction

Knowledge representation and search are two fundamental areas of artificial intelligence. Due to their importance we have made knowledge representation and search the focus of our research. Knowledge representation is the area of artificial intelligence which deals with capturing, in a formal language, the properties of objects and of the relationships between objects. Knowledge representation requires a formal language in which one can represent domain knowledge in a concise and unambiguous manner. Search is a systematic examination of all possible candidate solutions (search space) to a problem that is described as a theory (program) in some knowledge representation formalism. The conciseness of the knowledge representation formalism is imperative for effective search.

To illustrate what we mean by a knowledge representation formalism we present several examples using a group of objects with properties and relationships. In this example we follow the Prolog conventions of representing constants in lower-case characters, variables in upper-case characters and using the connectives ‘::’ and ‘,’ to mean respectively ‘is implied by’ and ‘and’.

We represent objects (a group of people for example) as constants in the language: john, mary, joe, jane, alice, susan.

We use predicates to describe properties or facts about objects. Here we use the predicates ‘male’ and ‘female’ to represent the gender of each person.

male(john).
female(mary).
male(joe).
female(alice).
female(jane).
female(susan).

We understand these facts as, “John is a male”, “Mary is a female” etc.

Predicates can also be used to describe relationships between objects. We use the predicate ‘parent’ to describe an example of a familial relation.

parent(john,mary).
parent(john,joe).
parent(alice,susan).
parent(joe,susan).
parent(jane,mary).
parent(jane,joe).

These facts describe the relationships, “John is a parent of Mary”, “John is a parent of Joe”, etc.

In addition to representing knowledge as facts, we can use rules or patterns to represent relationships. We introduce the predicate ‘father’ with the rule:

\[
\text{father}(X,Y):- \text{parent}(X,Y), \text{male}(X).
\]

As one can see the rules do not refer to specific objects (constants). Instead, they are templates which can be reused by substituting specific constants for the variables. The rule for the ‘father’ relationship uses the facts for ‘parent’ and ‘male’ to determine that ‘X’ is the father of ‘Y’ if ‘X’ is the parent of ‘Y’ and ‘X’ is male. For example from the facts, ‘male(john)’ and ‘parent(john,mary)’ we can conclude that father(john,mary) or “John is the father of Mary”. By using rules about relationships we avoid having to explicitly list all known relationships.

The predicates ‘aunt’, ‘sibling’ and ‘sister’ illustrate how complex relationships can be defined using previously defined relationships.

\[
\text{sibling}(X,Y):- \text{parent}(Z,X),\text{parent}(Z,Y), X \neq Y.
\]

\[
\text{sister}(W,U):- \text{sibling}(W,U),\text{female}(W).
\]

\[
\text{aunt}(R,S):- \text{sibling}(R,T),\text{parent}(T,S).
\]

There are two important points to note. The first is that when rules are used to define relationships it is not necessary to explicitly enumerate all knowledge. Second, we do not use negative facts in the knowledge base. For example, Mary is not a male but we do not have to include ‘not male(Mary)’. We use the closed world assumption [29] . Under the closed world assumption facts or relationships neither explicitly present in the knowledge base nor derivable from the knowledge base are assumed to be false. The use of rules and the closed world assumption are important for concise knowledge representation. However, the cost of conciseness is processing time.

After a representation has been specified how will the knowledge be used? This is where search comes in. As in Prolog we ask questions using queries. After searching the knowledge base, the query

\[
?- \text{male}(\text{john}).
\]

returns the result ‘yes’ while the query

\[
?- \text{male}(\text{mary}).
\]
returns the result ‘no’. The reason is that the fact ‘male(mary)’ was not in the knowledge base and under the closed world assumption facts not in the knowledge base are false.

We can use queries on relationships in the same manner as on properties. A search on the query

?-parent(john,mary).

returns the result ‘yes’. However, we can also use a search on a query to return specific data. A search on a query with a variable parameter uses substitution. We can make substitutions from less specific to more specific (constants or variables can be substituted for variables but variables can not be substituted for constants). The query

?-parent(joe,X).

will return X=‘susan’.

If one wants to know if Mary is Susan’s aunt a search is performed to determine if this fact can be computed from the facts and rules of the knowledge base. In this instance the answer is yes – we can determine that Mary is Susan’s aunt. In fact, there are several methods for search which can be used. A bottom-up approach starts with the constants and attempts to find a series of substitutions which results in a rule matching the query ‘aunt(mary,susan)’.

This approach can result in unnecessary work.

The top-down approach utilizes matching. Our goal is to determine if the ‘aunt(mary,susan)’ is true. We find that by substituting ‘mary’ for ‘X’ and ‘susan’ for ‘Y’ the rule for ‘aunt’ matches our goal. This gives

\[
aunt(mary,susan):- \text{sister}(mary,T), \text{parent}(T,susan).
\]

Before we can determine if our goal is true we must verify whether the atoms (subgoals) in the rule are true. Therefore, we substitute ‘mary’ and ‘T’ into the ‘sister’ rule.

\[
\text{sister}(mary,T):- \text{sibling}(mary,T), \text{female}(mary).
\]

Next, we substitute ‘mary’ and ‘T’ into the ‘sibling’ rule.

\[
\text{sibling}(mary,T):- \text{parent}(Z,mary), \text{parent}(Z,T), \text{mary} \neq T.
\]

Now, we look for the ‘parent’ relationship with ‘mary’ in the second parameter and find,

\[
\text{parent}(john,mary).
\]

Therefore, ‘john’ is substituted for the variable ‘Z’ and we return to the sibling rule.
sibling(mary,T):-parent(john,mary),parent(john,T), mary ≠ T.

Since the first subgoal has been satisfied, we look for the 'parent' relationship with 'john' in the first parameter and 'T' ≠ mary. We find the fact,

parent(john,joe).

We replace the variable 'T' with the constant 'joe' and return to the 'sibling' rule. Constants have replaced all variables and all subgoals are true.

sibling(mary,joe):-parent(john,mary),parent(john,joe), mary ≠ joe.

Therefore we can return to the 'sister' rule substituting 'joe' for 'T'.

sister(mary,joe):-sibling(mary,joe),female(mary).

Since 'sibling(mary,joe)' is true, we check;

female(mary).

Finding that 'female(mary)' is in the knowledge base we return to the 'aunt' rule.

aunt(mary,susan):- sister(mary,joe),parent(joe,susan).

Last we confirm the relationship

parent(joe,susan).

Since the subgoals for the rule 'aunt(mary,susan)' are true the answer 'yes' is returned. This method computes limits redundant computation what is necessary to determine the answer.

Although it is necessary for the designer of a system to know how answers are computed, we might wonder if it is necessary for the programmer who simply uses the system to be concerned with the method used to answer his query. If we assume that the programmer does not need to know the internal methods of the system, then we can separate the logic from the methods for finding answers to queries. This separation of logic and control [18, 19] was proposed by Kowalski and was instrumental in the development of logic as a programming language. Declarative programming stems from the concept of using logic to declare a meaning for a program or theory without being concerned with the underlying system.

The area of declarative programming has been dominated by PROLOG or PROLOG type logic programming. PROLOG is a top-down approach as illustrated above. Answer-set programming [14, 24, 26] is another approach to using logic as a programming tool. Answer-set programming formalisms can be used for solving search problems and are described in detail in Chapter 3. The answer-set programming formalism DATALOG with constraints is the main topic of this dissertation.
1.1 Motivation

Our motivation for developing answer-set programming formalisms is due to the importance of knowledge representation and search to computer science and the advantages of using declarative programming. Search problems are present in many areas of computer science such as AI, automated reasoning, theorem proving, operations research, planning, VLSI design, etc. Much work has been done on specialized algorithms in these areas. However, recently, efforts have also been directed toward general purpose knowledge representation and search techniques. The success in improving the performance of satisfiability checkers, the emergence of answer-set programming and appeal of declarative programming have been responsible for the current interest in knowledge representation and search techniques.

One of the most important satisfiability algorithms is the Davis-Putnam algorithm [5, 6]. The introduction of the Davis-Putnam algorithm drew considerable interest in developing satisfiability solvers. However, due to difficulties in scaling, interest in satisfiability solvers waned until the development of probabilistic algorithms such as GSAT [31] and simulated annealing [16, 17, 35]. The success of probabilistic algorithms brought about a renewed interest in satisfiability solvers as general purpose solvers. The renewed interest in satisfiability solvers and increased speed in current computer hardware also led to development of new heuristics for Davis-Putnam algorithms. Several fast implementations of the Davis-Putnam algorithm [8, 20] were also developed. Despite the recent success of satisfiability solvers there are still many drawbacks to their use as general purpose solvers. Implementations based on the Davis-Putnam algorithm have size restrictions and probabilistic algorithms cannot assure us of being able to determine a solution.

Independently of the work on satisfiability solvers, the logic programming community developed other techniques for knowledge representation and search. This work led to declarative programming [19] and the introduction of stable logic programming [14]. Stable logic programming (SLP) proved effective for problems which were often difficult for satisfiability, such as planning, routing etc. The difficulty of modeling problems using stable model semantics is however, a drawback to its use as a general purpose solver.

Satisfiability does not support the use of variables (programming) and the semantics of stable logic programming is not easily grasped which may inhibit its use. Both satisfiability and stable logic programming still have problems with scaling up. More work is needed. Developing better declarative programming tools for answer-set programming is a goal of our work.
1.2 Contributions

The main contribution of this dissertation is the development and implementation of an answer-set programming system, DATALOG with constraints. The DATALOG with constraints system (DC) is a programming environment which includes a formal language with a well-defined syntax and semantics, a parser for the language and a fast solver.

We developed a logic formalism for DC that facilitates the modeling of search problems. Problem descriptions (theories) may be written using a combination of Horn clauses and clauses. The semantics of DC are an extension of the semantics of propositional logic. The logic has been further extended to include special constructs for modeling frequently occurring concepts. In DC simple constraints can be easily specified using clauses or special constructs while more complicated concepts such as transitive closure can be modeled with Horn clauses. The combination of clauses, special constructs and Horn clauses in DC logic is an important contribution of this dissertation.

We implemented a solver for DC. The DC solver is designed to take advantage of the special constructs, Horn clauses and clauses. The search algorithm in DC is an extension of the Davis-Putnam backtracking algorithm [5, 6]. An important element of the Davis-Putnam algorithm is the use of unit propagation. Propagation in the DC solver was extended to effectively handle Horn clauses and special constructs. Heuristics to prune the search space for DC theories were developed and will be presented in Chapter 6.

We show in Chapter 7 that DC outperforms other state-of-the-art solvers. We compare results of experiments with DC to those of a satisfiability solver and an implementation of stable logic programming on a wide range of NP-hard problems. The results show that overall DC is effective.

1.3 Organization

In Chapter 2 we present basic concepts of propositional and predicate logic. We discuss declarative programming, answer-set programming and defines the syntax and semantics of Horn logic programming in Chapter 3. Chapter 4 discusses propositional schemata and describes extensions to the logic for DC. In Chapter 5 we describe the environment of the DC system. Implementation of the DC system is discussed in Chapter 6. Experimental results of DC with comparisons to stable logic programming and satisfiability are presented in Chapter 7. Examples of DC programs and their smodels counterparts are shown in Appendix A.
Chapter 2
Preliminaries

In this chapter, we present basic definitions and concepts from the literature. We introduce satisfiability and Horn theories.

2.1 Language of Propositional Logic

We describe a language $L$ of propositional logic which contains a set of atoms $A$, the connectives ‘$\rightarrow$’ (implies), ‘$\land$’ (and), ‘$\lor$’ (or) and ‘$\neg$’ (negation) and brackets ‘(’ and ‘)’.

The language $L$ is the set of formulas $F$ such that:

1. if $a \in A$ then $a$ is a formula $a \in F$.
2. if $a \in F$ then $(\neg a) \in F$.
3. $a, b \in F$ then $(a \land b) \in F$.
4. $a, b \in F$ then $(a \lor b) \in F$.
5. $a, b \in F$ then $(a \rightarrow b) \in F$.

We define a set of literals $Lit$ such that if $a \in A$ then $a, \neg a \in Lit$.

A clause $C$ is a formula $l_1 \lor \ldots \lor l_n$ where $l_1, \ldots, l_n \in Lit$.

A Horn clause $C$ is a formula $l_1 \lor \ldots \lor l_n$ where at most one of $l_1, \ldots, l_n \in Lit$ is an atom.

A Horn program clause $C$ is a formula $l_1 \lor \ldots \lor l_n$ where exactly one of $l_1, \ldots, l_n \in Lit$ is an atom.

Throughout this thesis we write clauses as implications without explicit negation. The exception to this being descriptions of logic programming with negation.

We write the clause

$$\neg a_1 \lor \ldots \lor \neg a_m \lor b_1 \lor \ldots \lor b_n$$

where $a_1, \ldots, a_m, b_1, \ldots, b_n$ are atoms equivalently as:

$$a_1 \land \ldots \land a_m \rightarrow b_1 \lor \ldots \lor b_n$$

We write the Horn program clause

$$p \lor \neg a_1 \lor \ldots \lor \neg a_n$$

where $p, a_1, \ldots, a_n$ are atoms equivalently using the connective ‘$:\neg$’ (is implied by) as:

$$p :\neg a_1 \land \ldots \land a_n$$
A clausal theory $T$ is a formula \{$C_1 \land \ldots \land C_n$} where all $C_i$'s are clauses.

A model $M \subseteq A$ is a model of an atom $a$ if $a \in M$.

A model $M \subseteq A$ is a model of a clause $C = a_1 \land \ldots \land a_m \rightarrow b_1 \lor \ldots \lor b_n$ if at least one of $b_1, \ldots, b_n \in M$ or at least one of $a_1, \ldots, a_m \notin M$.

A model $M \subseteq A$ is a model of a theory $T = \{C_1 \land \ldots \land C_n\}$ if $M$ is a model of all $C_i$'s.

In the next section, we discuss predicate logic and in particular the semantics of Horn program clauses.

### 2.2 Language of Predicate Logic

Horn logic programming is a good representative of a standard declarative programming formalism and is a basis for much of the work in this dissertation. It is necessary therefore, to describe predicate Horn logic. The definitions given here are well known and can be found in many logic texts [7, 32]. The language of predicate Horn logic is built of:

1. Countable sets of constant, function, predicate and variable symbols.
2. Propositional connectives and constants.
3. The quantifier symbols $\forall$ and $\exists$.
4. Punctuation symbols: (’, ’) and ’,’.

Each function symbol and predicate symbol has an integer arity. The terms of the language are defined by (constructed from) its constant, variable and function symbols along with its punctuation symbols. The atoms of the language are defined by its predicate, constant, variable and function symbols along with its punctuation symbols.

We write predicate Horn program clause as:

$$ p(\overline{X}_0) : - q_1(\overline{X}_1) \land \ldots \land q_m(\overline{X}_m) $$

where $p$ and the $q_i$s are predicates of the language and the $\overline{X}_i$'s are tuples of terms of the appropriate arity. We call $p(\overline{X}_0)$ the consequent and the collection $q_1(\overline{X}_1) \land \ldots \land q_m(\overline{X}_m)$ the antecedent (possibly empty), of the implication. We follow the standard logic programming practice of dropping the universal quantifier.

A Horn logic theory (or logic program) is a set of Horn clauses. We define the semantics for Horn theories by considering a more general class of theories, called universal theories. Horn theories are special universal theories. We extend this approach to DC logic.

**Definition 1** A formula $\phi$ is called a sentence if it is of the form

$$ Q(x_1), \ldots, Q(x_k) \psi. $$
where $\psi$ is a formula in the language without quantifiers, all variables occurring in $\psi$ are among $x_1, \ldots, x_k$ and $Q$ is either $\forall$ or $\exists$. A sentence is a formula which has no free variables. \hfill \triangle

**Definition 2** A sentence $\varphi$ is called a *universal sentence* if it is of the form

$$\forall x_1, \ldots, \forall x_k \psi.$$  

where $\psi$ is a formula in the language without quantifiers and where all variables occurring in $\psi$ are among $x_1, \ldots, x_k$. A *universal theory* is a set of universal sentences. \hfill \triangle

**Definition 3** Let $T$ be a theory. The *Herbrand universe* of $T$, $HU(T)$, is the collection of ground terms (variable-free terms) that can be constructed from the constants and function symbols of $T$. \hfill \triangle

If $T$ contains at least one constant then $HU(T) \neq \emptyset$, and if $T$ contains at least one function symbol then $HU(T)$ is infinite. Throughout this thesis we consider only theories with at least one constant symbol.

**Definition 4** Let $T$ be a theory. The *Herbrand base*, $HB(T)$, is the collection of ground atoms that can be constructed from the ground terms and predicates of $T$. \hfill \triangle

The semantics for universal theories can be obtained by *lifting* propositional semantics to the general case through grounding.

**Definition 5** Let $\varphi$ be a universal sentence of the form

$$\forall x_1, \ldots, \forall x_k \psi.$$  

where $\psi$ is a formula without quantifiers. By a *propositional grounding* of $\varphi$, we mean a collection of all formulas

$$\psi(x_1/t_1, \ldots, x_k/t_k),$$

where $t_1, \ldots, t_k$ range over all ground terms in the Herbrand universe and

$$x_i/t_i$$

is the substitution of a ground term $t_i$ for a variable $x_i$.

For a universal theory $T$, the propositional grounding of $T$, $grnd(T)$, is defined as the union of the propositional groundings of all universal sentences in $T$. \hfill \triangle
**Definition 6** Let $T$ be a universal theory. A collection $M \subseteq HB(T)$ is a Herbrand model of $T$ if $M$ is a propositional model of $grnd(T)$. \[ \triangle \]

This is not a standard definition of a Herbrand model. However, it is equivalent to the standard definition [22]. Herbrand models are important as they are sufficient to establish consistency of universal theories. The following is a well-known theorem [15].

**Theorem 1** If $T$ is a universal theory then $T$ is consistent if and only if there is a Herbrand model of $T$. \[ \triangle \]

A universal theory may have many Herbrand models. A Herbrand model $M$ of a universal theory $T$ is *minimal* if it is inclusion-minimal in the family of subsets of the Herbrand base of the theory that are models of $T$.

**Theorem 2** Every consistent universal theory possesses a minimal Herbrand model. \[ \triangle \]

Since a Horn clause is a special case of a universal sentence, a Horn theory is a special case of a universal theory. Therefore the notions and results for universal theories also apply to Horn theories. Further, for Horn theories a stronger result than Theorem 2 can be proved. For every consistent Horn theory $P$, there is a least Herbrand model. The least Herbrand model of $P$ is a Herbrand model of $P$ that is included in every other Herbrand model of $P$.

**Theorem 3** Every Horn theory $P$ has a least Herbrand model $M_0$.

Proof: Let

$$M_0 = \bigcap \{M : M \text{ is a Herbrand model of } P\}.$$ 

$P$ has at least one model because $HB(P)$ is a model of $P$. By definition, $M_0$ is contained in every model of $P$. Thus it is sufficient to show that $M_0$ itself is a model of $P$ or, equivalently, that $M_0$ is a propositional model of $grnd(P)$.

Consider a clause from $grnd(P)$ say

$$p : -q_1 \land \ldots \land q_n.$$ 

Assume that for every $i, 1 \leq i \leq n$, $q_i \in M_0$. Let $M \subseteq grnd(P)$ be any Herbrand model of $P$. Since $M_0 \subseteq M$, for every $i, 1 \leq i \leq n$, $q_i \in M$. Since $M$ is a model of $P$, $p \in M$. Thus, $p$ belongs to every Herbrand model of $P$ and by the definition of $M_0, p \in M_0$. \[ \square \]
Example 2.1 To illustrate the concepts, we present an example of a Horn logic program $P$, and list its Herbrand universe and Herbrand base.

$$
\begin{align*}
P &= \\
q(X) &::= - p(f(Y)) \\
r(X,Y) &::= - q(X) \land p(Y) \\
p(a) &::= -
\end{align*}
$$

The Horn program $P$ has a single constant $a$, predicate symbols $p, q$ of arity one and a predicate symbol $r$ of arity two, variable symbols $X, Y$, and a function symbol $f$ of arity one. The ground terms constructed from constant symbols and function symbols of $P$ form the Herbrand universe of $P$ and are listed below.

$$
\{a, f(a), f^2(a), \ldots\}.
$$

The Herbrand Base of $P$ is the following set of ground atoms:

$$
\{p(a), q(a), r(a,a), p(f(a)), q(f(a)), r(a,f(a)), r(f(a),a), r(f(a),f(a)), p(f^2(a)), \ldots\}.
$$

The semantics of standard declarative programming systems such as Horn logic and related computational concepts, depend on the notion of a single intended model. We describe one approach to declarative programming using Horn logic. A Horn program $P$ may have many Herbrand models but one of them, the least model, denoted by $LM(P)$, is distinguished as the intended model. We use this model to define the meaning of a Horn program. We say that $P$ expresses a subset $X$ of the Herbrand universe $HU(P)$ if for some predicate symbol $p$ occurring in $P$,

$$
X = \{t \in HU(P) : p(t) \text{ is true in } LM(P)\}.
$$

Intuitively, we can say that Horn programs describe sets of terms where each term is an element of the Herbrand universe and each term is a solution. Terms are complex data structures, thereby, allowing us to recursively define objects. The ability to model recursive definitions is responsible for the expressive power of Horn logic programming. Fundamental to logic programming is the result that every recursively enumerable set can be specified by a Horn program [1, 34]. Thus, Horn programs are as expressive as Turing machines[24].

While Horn programs specify recursively enumerable sets, a mechanism or control is needed to automatically generate these sets. The development of SLD-resolution (Linear resolution with Selection function for Definite clauses) [19] provided such a control.
That is, the mechanism of SLD-resolution can be used for enumerating elements of \( \{ t \in HU(P) : p(t) \text{ is true in } LM(P) \} \). SLD-resolution, therefore, is a key aspect to Horn logic programming. A declarative programming approach is realized using Horn theories and SLD-resolution. The programmer can write a description of a problem using Horn clauses and does not need to be concerned with how the program is processed.

**Example 2.2** We illustrate declarative programming with a Horn logic program that reverses a list using recursion.

1. \( \text{reverse}(L1, L2) : \neg \text{revappend}(L1, [ ], L2) \).
2. \( \text{revappend}([X]L], L2, L3) : \neg \text{revappend}(L, [X]L2, L3) \).
3. \( \text{revappend}([ ], L, L) \).

We use the program “reverse” and the query \(?- \text{reverse}([a, b, c], X)\) where the intended solution is \( X = [c, b, a] \)

By using substitution, the query ‘reverse([a, b, c], X)’ matches rule (1). The subgoal ‘revappend([a, b, c], [ ], X)’ matches rule (2). The subgoal ‘revappend([b, c], [a], X)’ again matches rule (2) as does its subgoal ‘revappend([c], [b, a], X)’. Finally, the subgoal ‘revappend([ ], [c, b, a], X)’ matches rule (3) after the substitution of ‘[c, b, a]’ for ‘X’. Rule (3) has no subgoals and thus is true. The list ‘[c, b, a]’ is returned to the query as a substitution for ‘X’.

Example 2.2 illustrates a program with a single intended model and recursively defined data structures. We use a standard logic programming technique of a query, along with SLD resolution to compute a solution.

We have discussed Horn logic and how it can be used for declarative programming. However, there are limitations. Horn clauses, by definition, do not have negation, but using negation when modeling problems is often a natural way to express concepts. The logic programming and knowledge representation communities considered negation to be an important addition to logic programming. A *logic program with negation* is a set of universally quantified clauses of the form,

\[
p(X_0) : \neg q_1(X_1) \land \ldots \land q_m(X_m) \land \neg r_1(X_1) \land \ldots \land \neg r_n(X_n).
\]

The intuitive interpretation of clause 2.1 is that if all positive subgoals are true and no negative subgoals are provable then the consequent of the clause is true. Determining a formal interpretation turned out to be a difficult problem.

We have stated that the intended model for a Horn logic program is the *least model*. However, if we allow negation in the antecedent of the clauses, we can no longer be assured
that the program has a least Herbrand model. What then is the semantics of logic programs with negation? Several approaches were suggested to maintain the concept of a single intended model. Among these approaches are stratified programs and well-founded semantics, which required restricting the syntax or weakening the semantics. Stratified programs [28] restricted the syntax by requiring that priorities be assigned to atoms, while well-founded semantics [37] produced a three valued model. Neither of these approached were entirely satisfactory.

Another approach to logic programs with negation is logic programming with stable model semantics (stable logic programming) [14]. Stable logic programming does not restrict the syntax and its models are 2-valued. However, the stable logic programming approach does not maintain the concept of a single intended model, but rather specifies a set of intended models. This approach was embraced by the logic programming community and is discussed further in this dissertation.

Let us recall that semantics for universal theories (and as special cases Horn theories) can be obtained by lifting propositional semantics through grounding. The same approach is used to define the stable model semantics of logic programs with negation.

**Definition 7** Let $P$ be a propositional logic program and let $M$ be a subset of the Herbrand base of $P$. The reduct of $P$ with respect to $M$ is obtained from $P$ by:

(i) Removing each clause that contains a subgoal not $r_i$ where $r_i \in M$.

(ii) Removing each subgoal not $r_i$ from the remaining clauses.

The reduct of $P$ with respect to $M$, denoted $P^M$, is a Horn program. Therefore, $P^M$ has a least model.

**Definition 8** [14] Let $P$ be a propositional logic program and let $M$ be a subset of the Herbrand base of $P$. The set $M$ is a stable model of $P$ if $M$ coincides with the least Herbrand model of $P^M$.

**Theorem 4** [14] Let $P$ be a Horn program. Then $P$ has a unique stable model and it is the least Herbrand model of $P$. $\triangle$

Theorem 4 shows that stable model semantics can be regarded as a generalization of the least model semantics for Horn logic programs.

**Example 2.3** We use logic program $P$ with negation to illustrate the concepts we have defined.
\[ P = \\
q :\neg \, \text{not } p \\
p :\neg \, \text{not } q \]

The Herbrand base of \( P \) is \{p, q\} The candidates for stable models of \( P \) are the subsets of \( HB(P) \). We use each candidate \( M \) to compute the reduct of \( P \) and determine the least model of the resulting Horn program. If the candidate model used for the reduct coincides with the least model then it is a stable model.

(1) The candidate model \( M_1 = \emptyset \). Then \( LM(P^{M_1}) = \{p, q\} \neq M_1 \). Thus, \( M_1 \) is not a stable model.

(2) The candidate model \( M_2 = \{p, q\} \). Then \( LM(P^{M_2}) = \emptyset \neq M_2 \). Thus, \( M_2 \) is not a stable model.

(3) The candidate model \( M_3 = \{p\} \). Then \( LM(P^{M_3}) = \{p\} = M_3 \). Thus, \( M_3 \) is a stable model.

(4) The candidate model \( M_4 = \{q\} \). Then \( LM(P^{M_4}) = \{q\} = M_4 \). Thus, \( M_4 \) is a stable model.

In Example 2.3 both \( M_3 = \{p\} \) and \( M_4 = \{q\} \) are stable models of \( P \). This illustrates how stable model semantics differs from the least model semantics for Horn programs. Stable model semantics defines multiple intended models while the least model semantics for Horn programs specifies a single intended model.

Stable logic programming with both function symbols and negation in the bodies of clauses increases the expressiveness beyond what we consider feasibly computable \[30\]. To limit the expressiveness we can either prohibit negation or eliminate function symbols from the language. If we eliminate negation then stable logic programming reduces to Horn programming. This direction is, then, not interesting. By disallowing function symbols we ensure that the Herbrand universe and Herbrand base are finite. Thus searching for stable models becomes feasible. This is the direction we adopted in our thesis.

A stable logic program can have multiple intended models thus it does not fit into the standard logic programming concept of a single-intended model. The notion of multiple solution sets led to the answer-set programming \[21, 24\] paradigm. In the answer-set programming paradigm, multiple models correspond to different solutions to the problem described by the program as opposed to traditional logic programming where a single intended model is enumerated. In the next section, we present a discussion of answer-set programming.
Chapter 3
Answer-set Programming

Answer-set programming (ASP) is an emerging paradigm for declarative programming. In this chapter we present the basics of answer-set programming and contrast it with traditional logic programming.

We discuss propositional logic, stable logic programming and disjunctive logic programming as examples of answer-set programming. Lastly, we discuss criteria for evaluating answer-set programming systems.

3.1 Declarative programming

An answer-set programming [21, 24] formalism is a declarative programming formalism in which programs are built as theories whose models represent solutions. A formalism, $\mathcal{F}$, in the answer-set programming paradigm is a formalism based on an underlying formal language and having a well-defined semantics. The semantics of $\mathcal{F}$ support definition of a mapping that assigns to a program $P$ of $\mathcal{F}$ a collection of sets $AS(P)$. The elements of $AS(P)$ are called answer sets for $P$.

Why are we interested in answer-set programming? One of our goals is to study search techniques and answer-set programming seems to be a framework well suited to search problems.

Definition 9 [13] A search problem $\Pi$ is a set $D_\Pi$ of finite instances such that for each instance $I \in D_\Pi$ there exist a finite set $S_\Pi(I)$ of solutions for $I$.

We may solve a search problem by finding solutions or answer sets.

Definition 10 [13] An algorithm $\mathcal{A}$ solves a search problem $\Pi$ if, for each instance $I \in D_\Pi$, $\mathcal{A}$ returns “no” when $S_\Pi(I)$ is empty and any $s \in S_\Pi(I)$ otherwise.

A search problem can often be solved by reducing it to the problem of finding answer sets to a program in some answer-set programming formalism $\mathcal{F}$.

Definition 11 Let $\mathcal{F}$ be an answer-set programming formalism. Then $\mathcal{F}$ solves a search problem $\Pi$ if for every $I \in D_\Pi$ there is theory, $T(\Pi, I) \in \mathcal{F}$ and a polynomial time function $sol_\Pi$ that maps answer sets of $T(\Pi, I)$ to solutions for $\Pi$.

As an example of answer-set programming formalism, let us look at propositional logic. Let $\Pi$ be a search problem. If for every instance $I$ of $\Pi$ we can construct a theory $T(\Pi, I)$
such that models of $T(\Pi, I)$ have a polynomial mapping to solutions of $\Pi$ then $\Pi$ can be solved by means of satisfiability checkers. To illustrate one use of propositional logic as an answer-set programming system, we look at the graph coloring problem. Given an undirected graph $G = (V, E)$ (where $V$ is the set of vertices of $G$ and $E$ is the set of edges of $G$), a valid coloring is given by an assignment of exactly one color to each vertex such that vertices connected by an edge are not assigned the same color. The following example shows how the graph 3-coloring problem can be solved by (a reduction to) propositional satisfiability.

**Example 3.1** Let us consider the graph shown in Figure 3.1. We construct a propositional theory encoding the graph and the constraints of 3-color-ability. We assume that there are 3 colors available for coloring: red, green and blue. We use connectives ‘$\rightarrow$’ (implies), ‘$\lor$’ (or) and ‘$\land$’ (and). An empty consequent requires that the antecedent be false and an empty antecedent requires the consequent to be true. The graph and the colors are formally expressed by the following facts:

$$D =$$

$\rightarrow \ vtx(a)$
$\rightarrow \ vtx(b)$
$\rightarrow \ vtx(c)$
$\rightarrow \ edge(a,b)$
$\rightarrow \ edge(a,c)$
$\rightarrow \ clr(\text{red})$
$\rightarrow \ clr(\text{green})$
$\rightarrow \ clr(\text{blue})$

The constraints are modeled by the clauses:

$$T =$$

$(C1) \quad \rightarrow \ \text{color}(a,\text{red}) \lor \text{color}(a,\text{green}) \lor \text{color}(a,\text{blue})$

$(C2) \quad \rightarrow \ \text{color}(b,\text{red}) \lor \text{color}(b,\text{green}) \lor \text{color}(b,\text{blue})$
\[(C3) \quad \rightarrow \quad \text{color}(c, \text{red}) \lor \text{color}(c, \text{green}) \lor \text{color}(c, \text{blue})\]

Clauses (C1)-(C3) ensure that each vertex is assigned at least one color.

\[(C4) \quad \text{color}(a, \text{red}) \land \text{color}(a, \text{green}) \rightarrow \]
\[(C5) \quad \text{color}(a, \text{red}) \land \text{color}(a, \text{blue}) \rightarrow \]
\[(C6) \quad \text{color}(a, \text{green}) \land \text{color}(a, \text{blue}) \rightarrow \]
\[(C7) \quad \text{color}(b, \text{red}) \land \text{color}(b, \text{green}) \rightarrow \]
\[(C8) \quad \text{color}(b, \text{red}) \land \text{color}(b, \text{blue}) \rightarrow \]
\[(C9) \quad \text{color}(b, \text{green}) \land \text{color}(b, \text{blue}) \rightarrow \]
\[(C10) \quad \text{color}(c, \text{red}) \land \text{color}(c, \text{green}) \rightarrow \]
\[(C11) \quad \text{color}(c, \text{red}) \land \text{color}(c, \text{blue}) \rightarrow \]
\[(C12) \quad \text{color}(c, \text{green}) \land \text{color}(c, \text{blue}) \rightarrow \]

Clauses (C4)-(C12) ensure that no vertex is assigned more than one color.

\[(C13) \quad \text{color}(a, \text{red}) \land \text{color}(b, \text{red}) \rightarrow \]
\[(C14) \quad \text{color}(a, \text{blue}) \land \text{color}(b, \text{blue}) \rightarrow \]
\[(C15) \quad \text{color}(a, \text{green}) \land \text{color}(b, \text{green}) \rightarrow \]
\[(C16) \quad \text{color}(a, \text{red}) \land \text{color}(c, \text{red}) \rightarrow \]
\[(C17) \quad \text{color}(a, \text{blue}) \land \text{color}(c, \text{blue}) \rightarrow \]
\[(C18) \quad \text{color}(a, \text{green}) \land \text{color}(c, \text{green}) \rightarrow \]

Clauses (C13)-(C18) ensure that vertices with an edge between them are not assigned the same color.

The propositional theory \(D \cup T\) is satisfiable for 12 answer sets. They determine all valid 3-colorings of \(G\). One such coloring is given by:
\[
\{\text{color}(a, \text{red}), \text{color}(b, \text{green}), \text{color}(c, \text{blue})\}.
\]

In Example 3.1 we present a propositional theory that describes 3-color-ability of a specific graph. We now show a generalization of the propositional theory for representing \(k\)-color-ability of an arbitrary graph.

**Example 3.2** Let \(G = (V, E)\) be an undirected graph with a set of vertices \(V = \{v_1, \ldots, v_n\}\) and a set of edges \(E\). Let \(c_i \in C\) be a set of colors. The graph \(G\), has a valid coloring if each vertex is assigned a color and vertices joined by an edge are not assigned the same color. We use the propositional atoms \(\text{color}(v, c) \in At\) to denote that vertex \(v\) is assigned color \(c\).
\[ T_{dr}(G) \] consists of the clauses

(1) \[ \rightarrow \text{color}(v_i, c_i) \vee \ldots \vee \text{color}(v_i, c_k) \]
where \( v_i \in V \) and \( c_i \in C \) for \( 1 \leq i \leq k \)

(2) \[ \text{color}(v_i, c_i) \wedge \text{color}(v_i, c_j) \rightarrow \]
where \( v_i \in V \) and \( c_i, c_j \in C \) \( i \neq j \)

(3) \[ \text{color}(v_i, c) \wedge \text{color}(v_j, c) \rightarrow \]
where \( (v_i, v_j) \in E \) and \( c \in C \)

Clause (1) ensures that each vertex is assigned a color. Clause (2) ensures that each vertex has at most one color. Clause (3) ensures that vertices joined by an edge are not assigned the same color.

**Theorem 5** Let \( G = (V, E) \) be an undirected graph and let \( k \) be a positive integer. An assignment \( f : V \rightarrow \{1, \ldots, k\} \) is a \( k \)-coloring of \( G \) if and only if \( M = \{\text{color}(v, f(v)) : v \in V\} \) is a model of \( T \).

Proof: \( (\Rightarrow) \) Let us assume that \( f : V \rightarrow \{1, \ldots, k\} \) is a \( k \)-coloring of \( G \). We will show that \( M = \{\text{color}(v, f(v)) : v \in V\} \) is a model of \( T \). We will now show that all clauses in \( T \) are satisfied by \( M \). First consider an instance of clause (1), requires that each vertex be assigned a color. Clause (2) restricts assignment of colors such that a vertex can be assigned at most one color. Clause (3) disallows assignment of the same color to two vertices if there is an edge between them. Thus the assignment of colors to vertices \( A \in At \) where \( \{\text{color}(v, c) : \text{color}(v, c) \in A\} \) represents a valid coloring.

\( (\Leftarrow) \) A valid coloring has one color assigned to each vertex which satisfies Clauses (1) and (2). A valid coloring will not have two vertices which are joined by an edge assigned the same color. This satisfies Clause (3). Thus a valid coloring is a model of \( T \). \( \square \)

The propositional theory in Example 3.1 decides \( k \)-color-ability of a specific graph where \( k = 3 \). Using this technique a different theory is required for each graph and value of \( k \). It is obvious, however, that we can define \( k \)-color-ability of a graph without referring to a specific graph or value of \( k \). In fact we did this informally in Example 3.2 with the generalization to \( k \)-color-ability of an arbitrary graph. That is we described the constraints of a problem without referring to a specific instance. By separating the constraints or theory from an instance or data we write the theory only once. The separation of theory from data requires a formalism in which constraints can be given a fixed (independent of data instance) description.
Stable logic programming is a formalism which supports separation of theory from data. We use the example of $k$-coloring of graphs to show how stable logic programming can be used to write theories independently of data.

**Example 3.3** Let $G = (V, E)$ be an undirected graph with a set of vertices $V = \{v_1, \ldots, v_n\}$ and a set of edges $E$. Let $C = \{c_1, \ldots, c_k\}$ be a set of colors. The graph $G$, has a valid coloring if each vertex is assigned a color and 4 vertices joined by an edge are not assigned the same color. We use the propositional atoms $\text{color}(v, c) \in At$ to denote that vertex $v$ is assigned color $c$ and the propositional atoms $\text{othercolor}(v, c) \in At$ to denote that vertex $v$ has been assigned a color other than $c$. We use the connective $:\neg$ (is implied by) to emphasis the different semantics.

\[
P_{\text{clr}}(G) = \]

\[
(1) \quad \text{color}(X, A) : \neg \text{othercolor}(X, A) \land \text{vtx}(X) \land \text{clr}(A).
\]

\[
(2) \quad \text{othercolor}(X, A) : \neg \text{color}(X, B) \land A \neq B \land \text{vtx}(X) \land \text{clr}(A) \land \text{clr}(B).
\]

\[
(3) \quad : \neg \text{color}(X, C) \land \text{color}(Y, C) \land \text{edge}(X, Y) \land \text{clr}(C).
\]

Clause (1) assigns a color to a vertex only if the vertex has not been assigned a different color. Clause (2) together with Clause (1) ensure that each vertex is assigned exactly one color. Clause (3) prohibits vertices joined by an edge from being assigned the same color.

Example 3.3 presents a theory $P_{\text{clr}}(G)$ for $k$-coloring of a graph in the formalism of stable logic programming. The theory $P_{\text{clr}}(G)$ illustrates the separation of theory and data. The separation of theory and data is fundamental to the concept of uniformly solving a search problem. Next, we define what is meant by a formalism uniformly solving a search problem.

**Definition 12** An answer-set programming formalism $\mathcal{F}$ uniformly solves a search problem, $\Pi$ if there is

1. an effective encoding, $E_{D_{\Pi}}$, for every $I \in D_{\Pi}$ in the language of $\mathcal{F}$ and

2. a finite theory $T_{\Pi}$ defining $\Pi$,

such that there is a one-to-one polynomial time mapping from answer sets of $T_{\Pi} \cup E_{D_{\Pi}}(I)$ to solutions of $\Pi$.

We can see from Example 3.3 that stable logic programming is an answer-set formalism which can be used to uniformly solve $k$-coloring. This was not the case with the encoding present in Example 3.1.
Decision problems can be viewed as special search problems. For the class of decision problems the following theorem on the expressive power of DATALOG~ (logic programs with negation) was shown by Schlipf [30].

**Theorem 6** A decision problem $\Pi$ can be solved by a stable logic program if and only if $\Pi$ is in NP. △

A search problem $\Pi$ is in the class $NP$-search [23] if there is a nondeterministic Turing Machine $TM$ such that

1. $TM$ runs in polynomial time;
2. for every instance $I \in D_\Pi$, the set of strings left on the tape when accepting computations for $I$ terminate is precisely the set of solutions $S_\Pi(I)$.

We now have the following theorem that determines the expressive power of stable logic programming [23].

**Theorem 7** A search problem $\Pi$ can be solved by a stable logic program if and only if $\Pi \in NP$-search.

We can see that stable logic programming and propositional logic fit the answer-set programming paradigm. Besides stable logic programming and propositional logic, default logic [2, 10] and disjunctive logic programming [11] also fit the answer-set programming paradigm. Although propositional logic fits the answer-set programming paradigm, each instance of a problem requires a different theory. Our goal is to develop a formalism to uniformly solve search problems.

### 3.2 Goals of answer-set programming systems

Answer-set programming is a paradigm for declarative programming. Since there are many formalisms for answer-set programming and several emerging implementations, we need meaningful criteria for comparisons.

**expressive power** – The expressiveness of a particular systems relates to the class of problems which it is able to capture.

**time performance** – Empirical results on the performance of an answer-set programming system provide a metrics for the comparison of efficiency. Performance can vary greatly depending on the problem and even on specific instances. This indicates that for an accurate picture of efficiency of a system empirical testing over large number of instances from different problem domains is necessary. Care must be taken when doing
empirical studies to ensure that experiments are executed fairly, i.e. the same machine and benchmarks. Even then modeling of problems can greatly affect efficiency.

simplicity of semantics – Semantics based on monotonic logic, such as that for propositional logic, are well suited for defining constraints. However, there are concepts that are difficult to describe in propositional logic. On the other hand, complex semantics based on non-monotonic logics, such as SLP, were designed to handle incomplete and indefinite information. The drawback is the difficulty of understanding the semantics which makes errors more likely and may inhibit its use.

ease of modeling – It is important for a programmer to be able to easily and correctly model a problem. This is why the intuitiveness of the semantics is important. Implementations of SAT solvers (propositional logic) with some notable exceptions [25] do not provide tools for modeling problems. The current implementations of answer-set programming systems for stable logic programming and disjunctive logic do include explicit means to model frequently occurring constraints.

conciseness of theories – Having semantics and constructs which lead to concise theories improves efficiency.

We have discussed answer-set programming in some detail because this was the approach we decided to use when designing a formalism to uniformly solve search problems. The next chapter presents a discussion of the DC formalism and illustrates how it fits the answer-set programming paradigm.
Chapter 4
DATALOG with constraints

In this chapter we present DATALOG with constraints (DC) a new answer-set programming formalism. DATALOG with constraints was developed to provide the simplicity of propositional logic while extending the semantics to include cardinality constraints and Horn clauses. We can use Horn clauses to model transitive closure. Modeling transitive closure without Horn clauses requires using additional variables. However, Horn clauses are not required for DC theories. We show examples of problems that can be efficiently modeled without the use of Horn clauses. It is our belief that DC logic not only facilitates modeling and produces concise theories, but also leads to efficient search heuristics.

We present a discussion of DC in the propositional case and then show how this extends to the predicate case. Further, we show that DC is non-monotonic and has the same expressive power as stable logic programming.

4.1 DC— in the propositional case

We begin a discussion of DC with the propositional case. The language of propositional DC is determined by a set of propositional atoms At. We assume that At is of the form 
\[ At = At_C \cup At_H, \]
where \( At_C \) and \( At_H \) are disjoint. A cardinality atom is an expression of the form:
\[ m\{p_1, \ldots, p_k\}n \]
where all \( p_i \)'s are atoms in \( At \) and \( m, n \) are non-negative integers.

Logical connectives used in propositional DC are ’|’, ’|’, ’\rightarrow’, ’:\rightarrow’ and mean respectively ’and’, ’or’, ’implication’ and ’Horn implication’.

A normal clause is an expression of the form:
\[ b_1, \ldots, b_n \rightarrow a_1 | \ldots | a_m, \]
where each \( b_i, a_j \) is a propositional atom or a cardinality atom.

A Horn clause in propositional DC is an expression of the form:
\[ h : \neg a_1, \ldots, a_m \]
where \( h \in At_H \) and each \( a_i \in At \) or is a cardinality atom built of atoms from \( At_C \).

We call a clause a constraint if all \( b_i, a_j \) appearing in the clause belong to the set \( At_C \). Otherwise, we call a clause a post-constraint.
A DC theory (in the language given by the set of atoms $At = At_C \cup At_H$), is a triple $T = (T_C, T_H, T_{PC})$, where $T_C$ is a set of constraints, $T_H$ is a set of Horn clauses and $T_{PC}$ is a set of post-constraints. By $At(T)$, $At_C(T)$ and $At_{PC}(T)$ we denote the set of atoms from $At$, $At_C$ and $At_{PC}$, respectively, that actually appear in $T$.

**Definition 13** Let $M$ be a set of atoms, $M$ satisfies (is a model of) an atom $a \in At$ if $a \in M$.

A cardinality atom $m \{p_1, \ldots, p_k\} n$ is satisfied by $M$ if $m \leq |M \cap \{p_1, \ldots, p_k\}| \leq n$, that is if at least $m$ and at most $n$ of the atoms $p_i$ are in $M$.

Let $C$ be a clause or a Horn clause. Let $M \subseteq At$ be a set of atoms. If $M$ satisfies at least one atom or cardinality atom in the consequent of $C$ or if at least one atom or cardinality atom in the antecedent of $C$ is false in $M$ then $M$ satisfies $C$.

A set of atoms $M$ that satisfies an atom, cardinality atom or a clause is called a model of the atom, cardinality atom or the clause. The notion of a least model which we discuss in Chapter 3 extends to a collection of Horn clauses in DC as explained by the following result.

**Theorem 8** Let $T$ be a collection of Horn clauses. Let $A$ be a set of atoms from $At_C$. The collection of formulas $A \cup T$ has a least model.

Proof: By definition an atom $a \in At_C$ can appear only in the antecedent of clauses in $T$. Therefore a reduct of $T$ with respect to $A$ is a Horn theory and every Horn theory has a least model $3$. \qed

We call the least model of $A \cup T$ the closure of $A$ under $T$. We again use the symbol $LM(T)$ to denote the least model of a Horn theory $T$.

With a DC theory $T = (T_C, T_H, T_{PC})$ we associate a family of subsets of $At_C(T)$. We say that a set $M \subseteq At_C(T)$ is an answer set for $T$ if

1. $M$ satisfies all the clauses in $T_C$, and
2. the closure of $M$ under the Horn clauses in $T_H$, $M^c = LM(T_H \cup M)$ satisfies all clauses in $T_{PC}$

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

**Example 4.1** We illustrate the semantics of propositional $DC$ using the $DC$ theory $T = (T_C, T_H, T_{PC})$, where the set of constraint atoms is given by $At_C = \{a, b, c\}$ and the set of Horn atoms is given by $At_H = \{x\}$.

\begin{align*}
T_C \text{ (constraints)} & \quad c \rightarrow b \mid a \\
\quad a \rightarrow b
\end{align*}

\begin{align*}
T_H \text{ (Horn clauses)} & \quad b \rightarrow x
\end{align*}

\begin{align*}
T_{PC} \text{ (post-constraints)} & \quad x \rightarrow
\end{align*}

We first try to satisfy the constraints, $T_C$. We see that the sets $\emptyset$, $\{b\}$, $\{b, c\}$, $\{a, b, c\}$, $\{a, b\}$ are models of $T_C$. Computing the closure of the clauses in $T_H$ results in the sets $\emptyset$, $\{b, x\}$, $\{b, c, x\}$, $\{a, b, c, x\}$, $\{a, b, x\}$. Checking the post-constraint eliminates all sets containing $x$ and therefore $\emptyset$ is the only model which satisfies $T$.

This example illustrates the simplicity of the semantics — 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 clauses in theories to compute closure.

For propositional logic $DC$ we have the following complexity result.

**Theorem 9** The problem of existence of an answer set for a finite propositional $DC$ theory $T$ is $NP$-complete.

Proof: Consider a $DC$ theory $T = (T_C, T_H, T_{PC})$, where $T_H = T_{PC} = \emptyset$. Clearly, $M$ is an answer set for $T$ if and only if $M$ is a model of the collection of clauses $T_C$. Thus, the problem of existence of an answer set is at least as hard as the propositional satisfiability problem. On the other hand, for every $DC$ theory $T$ and for every set $M \subseteq At_C(T)$, it can be checked in linear time whether $M$ is an answer set for $T$.

It follows that every problem in $NP$ can be polynomially reduced to the problem of existence of an answer set for a propositional $DC$ theory. Thus, given a problem $\Pi$ in $NP$, for every instance $I$ of $\Pi$, $\Pi$ can be decided by deciding the existence of an answer set for the $DC$ theory corresponding to $\Pi$ and $I$.

Theorem 9 implies that every decision problem in the class $NP$ can be reduced in polynomial time to the problem of deciding the existence of a model for a propositional $DC$ theory. Thus, propositional $DC$ can be used as an answer-set programming system in the very same way as propositional logic. In Chapter 3 one of the criteria for comparison of answer-set
programming is simplicity of semantics. Here we show that the semantics of propositional DC is comparable to that of propositional logic.

In the next section we extend propositional DC to the predicate case. We show that this extension provides a basis for an answer-set programming system similar to that of SLP. In particular, it supports uniform representation of search problems.

4.2 DC— in the predicate case

Let us recall that in Section 3.1 we discuss grounding of universal theories. By using the same approach we can obtain the semantics in the case of predicate DC. We can lift the semantics of propositional DC to the predicate case through the concept of grounding. Extending DC to the predicate case is important as it significantly simplifies the task of developing programs for solving problems with DC. Each time we change input, a different theory has to be used. However, when constructing predicate DC-based solutions to a problem Π, it is often possible to separate the representation of an instance (input) to Π from that of the constraints that define Π. As a result only one (predicate) program describing the constraints of Π needs to be written. Specific input for the program, say I, can be described separately as a collection of facts (according to some uniform schema). Both parts together can be combined to yield a DC program whose answer sets determine solutions to Π for the input I. Such an approach, we refer to it as uniform, is often used in the context of DATALOG, DATALOG× or logic programming to study expressive power of these systems as query languages.

The language of DC is determined by sets of constant, variable and predicate symbols. The set of predicates of the DC language is partitioned into built-in predicates, program predicates and data predicates. Built-in predicates which are currently implemented in DC are: =, ≠, <, >, ≤, ≥ and have the standard meaning. Additional built-in predicates can easily be implemented.

In predicate DC we maintain a separation of the program and the data. Therefore, a predicate DC theory is a pair \( T = (P, D) \) where \( D \) is a set of ground atoms based on the data predicates and \( P \) is a collection of clauses.

An atom is an expression of one of the following three forms:

\[
\begin{align*}
(4.1) & \quad p(\vec{X}), \\
(4.2) & \quad p(\vec{X}, \vec{Y}) : q(\vec{Y}),
\end{align*}
\]

where \( \vec{X} \) is a set of variables or constants of the correct arity, \( p \) is a program predicate, data predicate or built-in predicate;
where $p$ is a program predicate and $q$ is a data predicate;

$$m\{p(\bar{X}, \bar{Y}) : q_1(\bar{Z}_1) : \ldots : q_k(\bar{Z}_k)\}n,$$

where $p$ is a program predicate and $q_i$’s are either data predicates or built-in predicates.

A normal clause in predicate $DC$ is an expression of the form:

$$a_1, \ldots, a_n \rightarrow b_1 \mid \ldots \mid b_m$$

where $n, m \geq 0$, and $a_i$’s, $b_j$’s are atoms each having of one of the three forms listed above.

A Horn clause in predicate $DC$ is an expression of the form:

$$h : - a_1, \ldots, a_n$$

where $n \geq 0$, $a_j$’s are atoms and $h$ is an atom of form (4.1), constructed from a program predicate. We distinguish $h$ as a Horn atom.

The collection of clauses $P$ is partitioned into Horn clauses $P_H$ and normal clauses. We further partition the set of normal clauses. A clause which contains no Horn atoms is called a constraint. The collection of constraints is denoted as $P_C$. A clause which is not a Horn clause but does contain at least one Horn atom is called a postconstraint. The collection of post-constraints is denoted as $P_{PC}$.

In Section 3.1 we discuss grounding of universal theories. By using the same approach we can obtain the semantics of $DC$ theories. We lift the semantics of propositional $DC$ theories through the concept of grounding.

Let us begin a discussion of grounding in $DC$ by emphasizing that a $DC$ theory does not extend the data. Since there are logic programming systems that do extend the data, it is an important to note that $DC$ does not. That is, for a theory $T = (P, D)$ we evaluate a ground atom constructed from a data predicate $dp(a)$ such that $dp(a)$ is true if $dp(a) \in D$. Otherwise $dp(a)$ is false. Ground atoms constructed from data predicates and built-in predicates are evaluated. We can remove a ground clause if a ground atom in the antecedent is evaluated as false or if a ground atom in the consequent is evaluated as true. Otherwise we can simplify the ground clause by removing the evaluated ground atom. Thus the set of ground clauses consists of ground atoms constructed from program predicates.

We continue the discussion of grounding in $DC$ by considering the grounding of a clause $C$ where all atoms are expressions of the form of 4.1. The grounding of $C$ generates a set of ground clauses constructed by replacing every variable in $C$ by a constant from $D$ (different occurrences of a variable are replaced with the same constant).
Example 4.2 Given a set of data $D = \{dp_1(1), dp_1(2), dp_1(3)\}$, program predicates $p, q$ and a clause

$$C = p(X, Y) \rightarrow q(Y)$$

$grnd(C)$ is the set of clauses:

$$\{p(1, 1) \rightarrow q(1),
\quad p(1, 2) \rightarrow q(2),
\quad p(1, 3) \rightarrow q(3),
\quad p(2, 1) \rightarrow q(1),
\quad p(2, 2) \rightarrow q(2),
\quad p(2, 3) \rightarrow q(3),
\quad p(3, 1) \rightarrow q(1),
\quad p(3, 2) \rightarrow q(2),
\quad p(3, 3) \rightarrow q(3)\}$$

An atom, $p(\tilde{X}, \tilde{Y}) : q(\tilde{Y})$, of the form (4.2) is understood as a disjunction

$$p(\tilde{X}, q_1) \mid \ldots \mid p(\tilde{X}, q_k)$$

where all $q_i$’s are tuples of constants specified by the extension of the data predicate $q$. When such an atom appears in a clause the grounding of the clause depends on whether the atom appears in the antecedent or in the consequent.

In the case where a clause $C$ has in its consequent an atom $A$ which is an expression of the form given by 4.2, $A$ is interpreted as a disjunction in the consequent of $C$.

Example 4.3 We show the ground clauses in the case where the atom appears in the consequent of the clause for a set of data $D = \{dp_1(1), dp_1(2), dp_2(a), dp_2(b)\}$, program predicates $p, q$ and a clause

$$C = p(X) \rightarrow q(X, Y) : dp_2(Y).$$

The variable $X$ is universally quantified therefore we have a separate clause for each $X$ and the atom, $q(X, Y) : dp_2(Y)$ in the consequent becomes a disjunction. $grnd(C)$ is the set of clauses:

$$\{p(1) \rightarrow q(1, a) \mid q(1, b),
\quad p(2) \rightarrow q(2, a) \mid q(2, b),
\quad p(a) \rightarrow q(a, a) \mid q(a, b),
\quad p(b) \rightarrow q(b, a) \mid q(b, b)\}$$
Propositional DC clauses do not have disjunction in their bodies. An expression with a disjunction in its antecedent can be written equivalently as a set of clauses. The expression
\[ p \lor q \rightarrow r \]
is logically equivalent to
\[ p \rightarrow r, q \rightarrow r. \]
Therefore, in the case where a clause has in its antecedent an atom of the form 4.2, we ground it to a set of ground clauses

**Example 4.4** Given a set of data \( D = \{dp_1(1), dp_1(2), dp_2(a), dp_2(b)\} \), program predicates \( p, q \) and a clause
\[ C = q(X, Y) : dp_2(Y) \rightarrow p(X) \]
The variable \( X \) is universally quantified therefore we have a separate clause for each \( X \) and the atom, \( q(X, Y) : dp_2(Y) \) in the antecedent becomes a disjunction which is written as separate clauses. \( \text{grnd}(C) \) is the set of clauses:
\[
\{q(1, a) \rightarrow p(1),
q(1, b) \rightarrow p(1),
q(2, a) \rightarrow p(2),
q(2, b) \rightarrow p(2),
q(a, a) \rightarrow p(a),
q(a, b) \rightarrow p(a),
q(b, a) \rightarrow p(b),
q(b, b) \rightarrow p(b) \} 
\]
A cardinality atom is an expression of the form (4.3). The expression within the braces is replaced by the set of ground atoms it represents. The set of ground atoms depends on which variables are universally quantified and which are not. Built-in predicates can be used to restrict constant substitutions. We use examples to demonstrate the construction of cardinality atoms.
Example 4.5 Given a set of data \( D = \{dp_1(1), dp_1(2), dp_2(a), dp_2(b)\} \), program predicate \( p, q \) and a clause
\[
C = p(X) \rightarrow 1\{q(X, Y) : dp_2(Y)\}\]

The variable \( X \) is universally quantified therefore we have a separate clause for each \( X \). The set of atoms inside the braces depends on \( X \) and \( dp_2(Y) \). \( grnd(C) \) is the set of clauses:
\[
\{p(1) \rightarrow 1\{q(1, a), q(1, b)\}\},
\]
\[
p(2) \rightarrow 1\{q(2, a), q(2, b)\}\}
\]
\[
p(a) \rightarrow 1\{q(a, a), q(a, b)\}\}
\]
\[
p(b) \rightarrow 1\{q(b, a), q(b, b)\}\}
\]

Let us continue this example by assuming we want to restrict the grounding of variable \( X \) to the constants specified by data predicate \( dp_1 \). Then we add an atom constructed from \( X \) and \( dp_1 \) to the antecedent of \( C \).
\[
C = dp_1(X), p(X) \rightarrow 1\{q(X, Y) : dp_2(Y)\}\]

The grounding evaluates \( dp_1(a) \) and \( dp_1(b) \) as false and the clauses with these ground atoms in the antecedent are removed. The data predicates \( dp_1(1) \) and \( dp_1(2) \) are evaluated as true therefore they are removed from the antecedent of the clause. Thus \( C \) is ground as the set of clauses:
\[
\{p(1) \rightarrow 1\{q(1, a)q(1, b)\}\},
\]
\[
p(2) \rightarrow 1\{q(2, a)q(2, b)\}\}
\]

We can restrict constants used to ground the cardinality atoms by using built-in predicates. Given a set of data \( D = \{dp_1(1), dp_1(2), dp_2(a), dp_2(b)\} \), program predicates \( p, q \) and a clause
\[
C = p(X) \rightarrow 1\{q(X, Y) : dp_2(Y) : X \neq Y\}\]

\( grnd(C) \) is the set of clauses:
\[
\{p(1) \rightarrow 1\{q(1, a)q(1, b)\}\},
\]
\[
p(2) \rightarrow 1\{q(2, a)q(2, b)\}\}
\]
\[
p(a) \rightarrow 1\{q(a, b)\}\}
\]
\[
p(b) \rightarrow 1\{q(b, a)\}\}
\]

Horn clauses are grounded with the restrictions:
1. Cardinality predicates cannot contain Horn predicates.

2. Horn predicates cannot be cardinality predicates or disjunctive predicates.

Given a theory $T = (P, D)$ we define the ground atoms of $T$ as $At(T)$. We define the set of ground atoms in the consequent of ground Horn clauses in $T$ as $At_H(T)$ and all other ground atoms of $T$ as $At_C(T)$. Therefore $At(T) = At_H(T) \cup At_C(T)$ are the ground atoms of $T = (P, D)$. The propositional $DC$ theory $grnd(T)$ is defined as the set of the ground clauses of $T$ determined by the language of $At(T)$.

**Definition 14** Let $T = (P, D)$ be a $DC$ theory. A set of atoms $M \subseteq At_C(T)$ is a model of $T$ if $M$ is a model of the propositional $DC$ theory $grnd(T)$. \(\triangle\)

We show how the logic of $DC$ can be used to a represent search problem. We again use the $k$-coloring problem as an example.

**Example 4.6** The data for 3-color-ability includes the vertices and edges of the graph (Fig. 3.1) and a set of three colors.

$I = 
\{color(red), color(blue), color(green),
vtx(a), vtx(b), vtx(c),
edge(a, b), edge(a, c)\}$

By $clr(X, C)$ we mean is vertex $X$ is assigned color $C$.

$P =
\begin{align*}
(C1) & \quad clr(X, C) \rightarrow vtx(X). \\
(C2) & \quad clr(X, C) \rightarrow color(C). \\
(C3) & \quad \rightarrow 1\{clr(X, C) : color(C)\}1. \\
(C5) & \quad clr(X, C), clr(Y, C) \rightarrow edge(X, Y). \\
\end{align*}$

Clause (C1) ensures that constants replacing $X$ are vertices. Likewise clause (C2) ensures that constants replacing $C$ are colors. In clause (C3) we use a cardinality atom which requires exactly one color to be assigned to each vertex. Clause (C4) restricts vertices which form an edges from being assigned the same color.

In Chapter 3 we list expressive power as one of the criteria for comparison of answer-set programming systems. We present here a theorem concerning the expressive power of stable logic programming and $DC$. The proof for Theorem 10 can be found in [9].
Theorem 10  The expressive power of $DC$ is the same as that of stable logic programming.

Corollary 1  A search problem $\Pi$ can be solved by a $DC$ program if and only if $\Pi \in NP$-search.

Decision problems can be viewed as special search problems. For the class of decision problems, Theorem 1 implies the following corollary analogous to an old result on the expressive power of DATALOG$^-$ [30].

Corollary 2  A decision problem $\Pi$ can be solved by a $DC$ program if and only if $\Pi$ is in $NP$. 

Chapter 5
Environment of \( DC \)

In this chapter we describe the syntax and semantics of the language of \( DC \). Examples of data and problem encodings will be shown and explained. We will provide instructions for executing both \texttt{dcgrnd} and \texttt{dcs}. Required input and available options will be presented.

The \( DC \) system consists of two modules, \texttt{dcgrnd} and \texttt{dcs}. The module, \texttt{dcgrnd} instantiates or grounds a problem, with encoded data for an instance of the problem. A machine readable grounded theory is constructed to be used as input to the solver \texttt{dcs}.

5.1 Language

In Chapter 3 we list criteria for comparison of answer-set programming systems. One of these criteria was ease of modeling. The formal language of \( DC \) is declarative and was developed to facilitate modeling of search problems. Our modification of propositional schemata allows the use of cardinality constraints and Horn clauses. These extensions provide a unique modeling environment for search problems.

5.1.1 Data Input

The syntax for encoding data is similar to that used by other programs, such as \texttt{smodels}. However, we require that encoded data and rules be in separate files. Data can be encoded as sets, ranges, or as individual elements. Each entry is required to be on a separate line and end with a period. Data encoded as a single element has an identifier or predicate name and the values is given within parenthesis.

\textbf{Example 5.1}

\[
\text{vtx}(1).
\]
\[
\text{vtx}(2).
\]
\[
\text{edge}(1, 2).
\]

This illustrates a data encoding which is often used for graphs.

Shortcuts for encoding data:

\textbf{Example 5.2}

\[
\text{clr[red, green, blue]}.
\]
$$clr[1..3].$$
$$clr[1..k].$$

We see here three different data encodings of a set of colors.

### 5.1.2 Program File

Program predicates and variables in $DC$ are typed. This allows for concise grounding, better error checking by the grounder and facilitates modeling. In Example 4.6 clauses (C1) and (C2) were needed to ensure the intended meaning of the program. We eliminate the need for these clauses by using typing. This also reduces the size of the grounded theory.

The program file consists of a preamble and a set of predicate clauses. Program predicates are defined and variables declared in the preamble. The definition of a program predicate begins with the keyword `pred` and is followed by the predicate name with data predicate names as its arguments.

**Example 5.3** A program predicate $q$ with $n$ arguments is defined by:

$$\text{pred } q(dp_1, \ldots, dp_n).$$

where $dp_1, \ldots, dp_n$ are unary data predicates. We can limit grounding of a program atom to values of a data predicate.

$$\text{pred } q(dp_1, \ldots, dp_n) : dp_m,$$

where $dp_1, \ldots, dp_n, dp_m$ are unary data predicates and $q, dp_m$ are both $n$-ary predicates. Each predicate definition ends with a period.

Variable declarations begin with the keyword `var` then the data predicate name and last the variable name.

**Example 5.4** A variable declaration is of the form:

$$\text{var } dp \ X.$$

where $dp$ is an unary data predicate. Several variables can be declared for a data predicate on a single line:

$$\text{var } dp \ X, Y, Z.$$

However, variables for different data predicate must be declared on separate lines:

$$\text{var } dp_1 \ A, B, C.$$

$$\text{var } dp_2 \ X, Y, Z.$$
Predicate definitions and variable declarations end with a period.

All program predicates and variables that will be used in program clauses must first be defined or declared. Syntax for the program file allows blank lines for readability. Comments are allowed. They begin with ‘%’ and continue to the end of the line. Clauses may be split across lines and each clause ends with a period.

Arithmetic expressions can be used in built-in predicate and program predicate arguments. The data value resulting from the evaluation of the arithmetic expression must be valid for the predicate. The arithmetic at this time is limited to integers and the operators $\, +, -, *, /, \text{abs()} \,$ (absolute value), $\text{mod}(N, b) \,$ and $\text{max}(X, Y) \,$ and $\text{min}(X, Y)$ . Standard order of operation is followed and parentheses can be used in arithmetic expressions.

Atoms constructed from built-in predicates $==, !=, <, >, <= \,$ and $>= \,$ are evaluated and removed during grounding as are atoms constructed from data predicates. If the built-in or data atom is true and in the antecedent of the clause then the atom is removed. If it is true and in the consequent of the clause the entire clause is removed. If the built-in or data atom is false and in the antecedent of the clause then the clause is removed. If it is false and in the consequent of the clause then the atom is removed.

Program atoms are simple, existential or cardinality atoms. Existential atoms are written in the form:

$$p(X, Y) : dp(Y)$$

where $p$ is a binary program predicate and $dp$ is an unary data predicate. $X$ is universally quantified unless otherwise distinguished ($Y$ is is distinguished by : $dp(Y)$). The colon indicates that a condition is being placed on the grounding of the program atom. In this example $Y$ is limited to the range given by $dp$. Since program predicates are typed, we can alternately use the shorthand notation:

$$p(X, \_).$$

Cardinality atoms are of the form:

$$m \ \{p_1(\bar{X}, Y_{i,1}, \ldots, Y_{i,i}) : dp_{i,1}(Y_{i,1}) : \ldots : dp_{i,i}(Y_{i,i}) : q_1(\bar{X}, \bar{Y}_1) : \ldots : q_k(\bar{X}, \bar{Y}_1),$$

$$\ldots, p_j(\bar{X}, Y_{j,1}, \ldots, Y_{j,i}) : dp_{j,1}(Y_{j,1}) : \ldots : dp_{j,i}(Y_{j,i}) : q_1(\bar{X}, \bar{Y}_j) : \ldots : q_k(\bar{X}, \bar{Y}_j) \} \ n$$

which is a generalization of the previous form except that we have upper and lower bounds on the number of grounded atoms which are used to decide the truth value of the cardinality atom. The individual grounded atoms together with the upper and lower bounds make up the cardinality atom. As in the previous case we can use the shorthand notation:

$$m \ \{p(X, \_) \} \ n$$
The shorthand notation can be used when the predicate argument is not restricted. We restrict assignments to variables during grounding by using a ‘:’ (colon) followed by a either a data predicate or a built-in predicate. An example using a data predicate and a built-in predicate is:

\[ m \{ p(X, Y) : dp(Y) : Y! = X \} \ n \]

We can also explicitly list atoms:

\[ m \{ p(X, Y), q(X, Y), r(X, Y) \} \ n \]

where \( p, q \) and \( r \) are program predicates and \( X, Y \) are universally quantified variables.

Non-Horn clauses are of the form:

\[ q_1(X_1), \ldots, q_n(X_n) \rightarrow q_{n+1}(X_{n+1}) \ | \ \ldots \ | q_m(X_m). \]

where the \( q_i \)'s can be program, data, cardinality or built-in predicates.

Horn clauses are of the form:

\[ q(Y) : -p_1(X_1), \ldots, p_n(X_n). \]

where the \( p_i \)'s can be program, data, cardinality or built-in predicates and \( q \) is a Horn predicate. Horn atoms cannot be used in cardinality atoms. Horn clauses are written differently than normal clauses because they have a different semantics. The grounded Horn clauses are maintained as Horn clauses.

Post-constraints have the same form as non-Horn clauses, but have a Horn atom in either the antecedent or consequent of the clause.

Appendix A presents theories for selected problems. These theories are written in the syntax of psgrnd, parse and CNF syntax for psgrnd. Theories are presented for graph problems, combinatorial problems and planning. The theories can be used as benchmarks for comparing the efficiency of solvers.

5.2 Wire routing – a modeling example

In this section, we demonstrate how DC is used to model problems. We look at a simplified version of the wire routing problem which could also be applied to other routing problems. The physical layout is the last phase in the design of VLSI circuits. In this phase, the components are placed on the circuit and wires are laid out. In our model, we assume the components have been placed on the circuit and we define the requirements:

1. Wires do not intersect
2. Wires cannot touch components

3. Each wire has a pair of points on the circuit which must be connected

Data for the problem consists of a set of wires, coordinates for a grid, two terminal points for each wire and a set of coordinates that are unavailable for use (have components).

Figure 5.1 is a $10 \times 10$ grid for representing a chip. Gray areas are blocked (components) and terminal points are indicated by their wire number. An example of the encoded data for the grid is given as:

wire(1). wire(2). wire(3). wire(4).

coord(1). coord(2). coord(3). coord(4). coord(5).

terminal(6,4,1). terminal(10,8,1).
terminal(7,9,2). terminal(9,4,2).
terminal(4,9,3). terminal(9,9,3).
terminal(8,6,4). terminal(3,4,4).

block(4,3). block(4,4). block(4,5). block(4,6).
block(5,3). block(5,4). block(5,5). block(5,6).
block(7,1). block(7,2). block(8,1). block(8,2).

The encoding $wire(W)$ identifies wire $W$. A coordinate on the grid is given $coord(I)$. The encoding $terminal(I,J,W)$ is read as wire $W$ has a terminating point at $I,J$. The
Figure 5.2: Solution for wire routing example with wandering paths

encoding \( \text{block}(I, J) \) indicates that point \( I, J \) is unavailable.

We begin our program by writing clauses describing a path connecting the terminal points. First, each wires terminal points are on the path. \( \text{terminal}(I, J, W) - > \text{path}(I, J, W) \).

Second, each terminal point has exactly one adjacent point on the path. This clause enforces a certain optimization since requirements for a path could be enforced by at least one adjacent point is on the path. \( \text{terminal}(I, J, W) - > \)

\[ \{ \text{path}(L, M, W) : \text{coord}(L) : \text{coord}(M) : \text{abs}(I - L) + \text{abs}(J - M) = 1 \} \]

Next, all other points on the path connecting the terminal points must have exactly two adjacent points on the path. Again, this clause is stronger than a clause simply enforcing path requirements. \( \text{path}(I, J, W) - > \)

\[ \{ \text{terminal}(I, J, W) \mid 2 \{ \text{path}(L, M, W) : \text{coord}(L) : \text{coord}(M) : \text{abs}(I - L) + \text{abs}(J - M) = 1 \} \} \]

The next clause restricts each point from having more than one wire passing through it. \( \{ \text{path}(I, J, W) \} \).

Next, we prohibit any wire from using an unavailable point. \( \text{block}(I, J), \text{path}(I, J, W) - > . \)

These five clauses are sufficient to ensure that the wire routing conditions are met. After grounding with the example data the \( DC \) solver finds the solution shown in Fig. 5.2. We see that paths “wander” and that there are cycles independent to the path connecting the terminals.

We can eliminate the independent cycles which consist of only four points with the following clause:
Figure 5.3: Solution using the additional clauses for eliminating cycles and restricting search space

\[ \text{path}(I, J, W), \text{path}(I + 1, J, W), \text{path}(I + 1, J + 1, W), \text{path}(I, J + 1, W) \rightarrow . \]

This restriction does not in general eliminate cycles. However, along with the following set of clauses which limits the search space, the possibly of having independent cycles decreases considerably, since the next smallest independent cycle would have to have at least eight points in its path.

\[ \text{terminal}(I, J, W), \text{terminal}(M, N, W), \text{path}(K, L, W), \]
\[ K + k < I, K + k < M, I! = M \rightarrow . \]
\[ \text{terminal}(I, J, W), \text{terminal}(M, N, W), \text{path}(K, L, W), \]
\[ K - k > I, K - k > M, I! = M \rightarrow . \]
\[ \text{terminal}(I, J, W), \text{terminal}(M, N, W), \text{path}(K, L, W), \]
\[ L + k < J, L + k < N, J! = N \rightarrow . \]
\[ \text{terminal}(I, J, W), \text{terminal}(M, N, W), \text{path}(K, L, W), \]
\[ L - k > J, L - k > N, J! = N \rightarrow . \]

This set of clauses limits the search space for each wire to a rectangle formed by its terminal points. The constant \( k \) is also used here to relax this constraint if paths cannot be found within the rectangle. The value for \( k \) can be set on the command line when grounding (this is explained in the next section). Figure 5.3 shows the wire routing example using these additional constraints.

The user may also want to restrict the path length. This can be handled by adding a clause which allows at most \( m \) points in each path. Like \( k \) in the previous set of clauses \( m \) is given a value when grounding. The two following clauses limit path length and search space.

\[ \{ \text{path}(I, J, W) : \text{coord}(I) : \text{coord}(J) \} m. \]
terminal \((I, J, W), \text{path}(K, L, W), \abs(I - K) + \abs(J - L) > m \rightarrow \).

We see here how easy it is to describe problems in the language of \(DC\) and how the use of cardinality predicates facilitates modeling of problems. This is one of the goals of answer-set programming which we discuss in Section 3.2.

### 5.3 Grounding

The required input to execute \texttt{dcgrnd} is a single program file, one or more data files and optional constants. If no errors are found while reading the files and during grounding, an output file is constructed. The output file is a machine readable file whose name is a catenation of the constants and file names with the extension \texttt{tdc}.

\texttt{dcgrnd -r rulefile -d datafile list [-c name=value]}

**Required arguments**

- \texttt{-r rulefile} is the file describing the problem. There must be exactly one rule file.

- \texttt{-d datafile list} is 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. For example, you may have several files each containing the description of a graph. Additional data such as the number of colors would be used with each file and can be contained in a separate file.

**Example 5.5** \texttt{dcgrnd -r color -d 1.gph clrfie}

**Optional arguments**

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

Error messages are identified as either read errors or compile errors. The read errors usually indicate a syntax error in the data or rule file. The compile errors tend to refer to the formation of the rules although they could possibly be the result of syntax errors which were not recognized as such.
5.4 DATALOG with constraints solver (dcs)

dcs is used to solve the grounded theory constructed by dcgrnd. The name of the file containing the theory is input on the command line. A file named dcs.stat is created or appended with statistics concerning the solving of the problem.

dcs -f filename [-A] [-P] [-C [x]] [-L [x]] [-S name]

Required arguments

-f filename is the name of the file containing a theory produced by dcgrnd.

Optional arguments

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

-P Prints the input theory and then exits.

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

Optional: x is an integer such that dcs finds x number of solutions then exits.

-L [x] Options for number of literals to process in lookahead.

Optional: x is an integer indicating the number of literals to process in each call to lookahead. (default = off)

-S name Show positive atoms with predicate name.
Chapter 6
Implementation

In this chapter, we describe the implementation of $DC$. For a search problem $Π$, a $DC$ theory $T_Π$ consists of encoded data for an instance $I_Π$ and a program $P$. To obtain $T_Π = (I_Π \cup P)$ we ground $P$ with $I_Π$. We can then solve $T_Π$.

We implemented the $DC$ system in two modules. The grounding module $d\text{cgrnd}$ reads data and program files and generates a concise propositional $DC$ theory. The solver $d\text{cs}$ uses a backtracking algorithm and propagation. We designed the solver to take advantage of cardinality atoms and Horn clauses in $DC$ theories.

6.1 Grounding

The program, $d\text{cgrnd}$, grounds a problem description using encoded data for a specific instance. The files containing the $DC$ program and encoded data for a specific instance are read by $d\text{cgrnd}$. During input of the files, checks are performed to ensure that parameters used to define predicates and types used to declare variables are valid. Checking of valid variables, predicate types and conditions is performed during parsing of $DC$ clauses. Each clause in a $DC$ theory is grounded independently.

Variables in a clause, not designated as existential variables or cardinality atoms, are assumed to be universally quantified. All grounding is limited to correct type and values of data predicates.

During grounding, universally quantified variables are processed before existentially quantified variables. The language of $DC$ also contains built-in predicates and function symbols such as the equality operator and arithmetic comparators and operations. We assign to these symbols their standard interpretation. We evaluate all expressions involving built-in function symbols and all atoms involving built-in relation symbols in the grounding process. If any

```plaintext
Instantiate(C,NextVariable)
1  if(NextVariable==Exist)
2      ProcessExist(NextVariable,C)
3  else ProcessForall(NextVariable,C)
4  endInstantiate
```

Figure 6.1: Instantiate

41
ProcessForall(C,V)
1   while(data==MoreData(V))
2       good:=Replace(V,data,C)
3           if(good)
4                   if(NextVariable)
5                       Instantiate(C,NextVariable)
6                   else WriteClause(C)
7               Remove(V,data,C)
8           endwhile
9       endProcessForall

Figure 6.2: ProcessForall

argument of a built-in relation is not of the appropriate type, we interpret the corresponding
atom as false. If a function yields as a result a constant that does not appear in the theory
or if one of its arguments is not of the required type, we also interpret the corresponding
atom as false. We then eliminate tautologies and simplify the remaining clauses by removing
true “built-in” atoms from the antecedents and false “built-in” atoms from the consequents.

Figure 6.1 shows the procedure Instantiate. After parsing a clause, the procedure
Instantiate is called with the clause and the first variable in the variable list. Instantiate
then calls either ProcessForall or ProcessExist depending on quantification of the variable.

The procedure ProcessForall is called with the clause structure and V (see Fig. 6.2). A
while loop processes data values for V. A data value is obtained (line 2) and every occurrence
of V in the clause is replaced (line 3). If all replacements are valid according to clause
restrictions then a check is made to see if there are more unprocessed variables (line 5). If
there are more variables to be processed Instantiate is called with the clause structure and
the NextVariable (line 6). If there were no more variables then the clause is written to a
theory file (line 7). For the last statement in the while loop we retract the current data value
for V (line 8). When the while loop has processed all data values for V then ProcessForall
returns.

The procedure ProcessExist is called with the clause structure and V (see Fig. 6.3). A
while loop processes data values for V. A data value is obtained (line 2) and every occurrence
of V in the clause is replaced (line 3). If all replacements are valid according to clause
restrictions then a check is made to see if there are more unprocessed variables (line 5).
ProcessExist(C,V)
1     while(data==MoreData(V))
2         good:=Replace(V,data,C)
3             if(good)
4                 if(NextVariable)
5                     ProcessExist(NextVariable,C)
6                 else AddAtom(V,data,C)
7             Remove(V,data,C)
8         endwhile
9     WriteClause(C)
10    endProcessExist

Figure 6.3: ProcessExist

If there are more variables to be processed they must also be select variables therefore ProcessExist is called recursively (line 6). If there were no more variables then the valid atom or atoms are add to the clause (line 7). The last statement within the while loop is the retraction of the current data value for V (line 8). When the while loop has processed all data values for V then the clause is written to the theory file (line 10).

The cardinality atoms are processed last using a routine similar to ProcessExist. In Chapter 3 one of the criteria listed for comparison of answer-set programming systems is conciseness of theories. The grounder simplifies DC theories during grounding, retains the structure of the cardinality constraints and the semantics of the Horn clauses which leads to more concise grounded theories.

6.2 Solving grounded DC theories

The solver, dcs, uses a depth first search with backtracking, propagation and lookahead techniques. The lookahead in dcs is similar to techniques used for satisfiability solvers (csat [8]) and stable logic programming (smodels [27]). However, we use different methods to determine how many atoms to consider in the lookahead phase. Other techniques, especially propagation and search heuristics, were designed specifically for the dcs as they must take into account Horn clauses, constraints and post-constraints.
6.2.1 Preliminary steps

The first step in dcs is setting up referencing structures that are used throughout the execution of the program. The referencing structures are essential to the efficiency of the algorithm. The atom reference maintains a list of clauses and cardinality atoms for each atom. The reference indicates whether the atom appears in the antecedent or consequent of the clause and how often it appears in each. This data is used during propagation and when choosing branch atoms.

Next, an array for assigning weights to clauses is initialized. The values for the weight array are computed using, N, the number of atom in the longest clause and C, an empirically determined constant.

\[ wgt[i] = C^{N-i} \text{ for } 0 \leq i \leq N \]

The weight array, and current length of a clause are used to dynamically revise weights.

6.2.2 Propagation

The propagation routine takes a partial assignment of values to atoms and evaluates the theory based on the partial assignment. When a value is assigned to an atom, each clause in which the atom appears may be:

1. satisfied,
2. reduced such that there is a single atom in the clause and there is only one assignment of values to the atom which satisfies the clause,
3. unsatisfied (backtracking is required) or
4. can not yet be determined.

In the case of (2) the remaining atom is called a forced atom since there is only one assignment which satisfies the clause. Not all forced atoms in DC are propagated. Horn atoms must be computed by Horn clauses and cardinality atoms must have its individual atoms appropriately assigned. These atoms are marked as forced atoms with the required truth value indicated.

A cardinality atom is an atom of the form:

\[ m\{a_{t_1},\ldots,a_{t_k}\}n, \]

where \( m, n \) are non-negative integers, \( m \leq n \) and \( a_{t_i}'s \) are atoms. A cardinality atom is “true” if at least \( m \) and at most \( n \) of the \( a_{t_i}'s \) are “true”. The atoms \( (a_{t_i}'s) \) of a forced cardinality atom can be propagated by the cardinality atom when there is only one assignment of values to the atoms which satisfies the required truth value of the forced cardinality atom.
Example 6.1 Let \( C \) be a forced cardinality atom

\[
C = 1\{a, b, c\}1
\]

where \( a, b, c \) are atoms, \( C \) is required to be “true” and \( a \) is assigned “true”. The only assignment to the unassigned atoms that satisfies \( C \) is both \( b, c \) assigned “false”. Both \( b \) and \( c \) are propagated.

The value “true” is assigned to a Horn atom if it appears in the consequent of a Horn clause for which all atoms in the antecedent of the Horn clause have been assigned the value “true”. If one or more atoms in the antecedent of a Horn clause have been assigned the value “false”, the clause is removed. If a Horn atom has not been assigned a value and all Horn clauses in which it appeared in the consequent have been removed, it is assigned the value “false”.

Cardinality atoms can be assigned “true” only if the truth assignment to the individual atoms in the set of the cardinality atom are within the upper and lower limits of the cardinality atom and “false” otherwise. We can force individual atoms if the cardinality atom is a forced atom.

If the assigned value for either a Horn atom or a cardinality atom contradicts a forced value then backtracking is required.

6.2.3 Branching Atoms

Branching atoms are those atoms which are assigned values and propagated. The most constrained atoms are chosen for branching. In general, atoms which appear most often and in the shortest clauses in the reduced theory are the best candidates. This data is available readily from the reference and weight structures. We use this data to compute a weight for each atom that is neither a Horn atom nor a cardinality atom. The atom with the maximum weight becomes a candidate for branching.

Horn atoms are not used for branching atoms. However, some cardinality atoms are used for branching. A cardinality atom is an atom of the form:

\[
m\{at_1, \ldots, at_k\}n,
\]

where \( m, n \) are non-negative integers, \( m \leq n \) and \( at_i \)'s are atoms. The requirements for a cardinality atom to be a candidate for branching are:

1. \( m, n = 1 \)

Where exactly one atom from \( at_1 \ldots a_k \) can be “true”, there are exactly \( k \) possible
assignments that can make the cardinality atom true. Other cardinality atoms, in
general, have too many possible assignments to make branching on them practical.

2. The candidate cardinality atom is a forced atom where the required value is “true”.

The weight of a cardinality candidate $C$ is the sum of the weights of atoms $at_1, \ldots, at_k$
from $C$. If there are no cardinality candidates, the atom candidate is the branching atom
otherwise we choose from among the cardinality candidates as follows:

1. Fewest individual atoms.
2. The greatest weight.

The branching atom is returned to the calling function.

6.2.4 Lookahead

Lookahead tests for contradictions and forced atoms among atoms not assigned a value. An
atom is chosen, assigned the value true, evaluated using only propagation, and then assigned
the value false and evaluated. If both evaluations of assignments result in conflicts then
backtracking is required. If only one evaluation results in conflict, the atom is assigned
the opposite value and lookahead is continued. If neither evaluation results in conflict no
assignment is made. Because of the high overhead only a few atoms are tested during looka-
head. The same function used for determining the branching atoms is used for determining
lookahead atoms. Lookahead is implemented only as a option.
Chapter 7
Experimentation

One of the criteria for comparison of answer-set programming systems listed in Chapter 3 is time performance. Our goal in developing \( DC \) was to provide an answer-set programming system that is fast for a wide range of problems. Different problems and even different instances of the same problem may vary greatly in size and difficulty. Additionally, heuristics that work well for some problems may work poorly for other problems. The best means of determining efficiency seems to be to execute the system on a wide range of problems and compare the results with other ASP systems.

The theories used for experiments with \texttt{smodels} and \( DC \) are presented in Appendix A. We have attempted to write theories as efficiently as possible.

The theories for SAT problems have been produced either by the \( DC \) grounder or by a utility program to encode a specific problem. Explanations of these problems along with the complete \( DC \) and SLP theories are presented in Appendix A.

7.1 ASP systems

We have discussed SLP and propositional logic as ASP systems. Therefore, to test the efficiency of \( DC \) as an ASP system comparisons of \( DC \) will be made with SLP and a SAT solver. Many implementations of SLP and SAT solver are available. We wanted to use efficient implementations of SLP and SAT solvers for our comparisons with \( DC \). We compared \texttt{smodels} and \texttt{satz} with other implementations of SLP and SAT solvers by executing them on a set of difficult problems. Based on our experiments we found \texttt{smodels} and \texttt{satz} performed efficiently therefore these are the implementations we use in comparisons with \( DC \). Both implementations are available on the Internet.

The SAT solver \texttt{satz} [20] was implemented by Chu Min Li and is based on the Davis-Putnam-Loveland-Logemann approach [5, 6]. The input theory for \texttt{satz} is a propositional theory in DIMACS format for SAT.

The SLP implementation \texttt{smodels} [27, 33] was implemented by Ilkka Niemelä and Patrik Simons. Also available is \texttt{1parse} [36] a stable model semantics grounder implemented by Tommi Syrjänen that produces propositional theories for \texttt{smodels}.

Comparisons between the \( DC \) solver \texttt{dcs}, \texttt{smodels} and \texttt{satz} are discussed in this chapter. All results are from executions on a Pentium III running under Linux.
7.2 Generating random graphs

We use randomly generated graphs for experiments with the Hamilton cycle problem and graph coloring. The graphs generated for Hamilton cycle problem are directed graphs and for graph coloring we use undirected graphs. The random graph routine uses as inputs the number of vertices and the number of edges. Two vertices are generated randomly. If an edge between them does not already exist then that edge is added to the graph. Input parameters are the number of vertices $v$, and the number of edges $e$.

Input: The number of vertices $v$ and the number of edges $e$
Output: Graph $G$

random-graph$(v, e)$

Set $E = \emptyset$

For $i = 1$ to $e$ do

Randomly generate $(w, u) \notin E$

if undirected $(w, u) = (u, w)$

Add $(w, u)$ to $E$

End For

Return $G$

End random-graph

7.3 Results for selected problems

The problems used in comparisons are search problems, problems based on graphs, problems from combinatorics and planning problems. These problems have corresponding decision problems which are in the class NP, hence these problems are NP-hard.

7.3.1 Hamiltonian Cycle

The Hamiltonian cycle problem cast as a search problem is that of finding cycle in a graph such that passes through each vertex exactly once. The corresponding decision problem is NP-complete [13].

Not only is the Hamiltonian cycle problem in itself of interest, many other problems (routing, planning, etc) can be encoded as Hamilton cycle problems.

The instances for computing Hamilton cycles were obtained by randomly generating one thousand directed graphs with $v = 20, 30 \ldots, 100$ and density such that $\approx 50\%$ of the graphs
were expected to contain Hamiltonian cycles. We use instance from the probability space where \( \approx 50\% \) have Hamiltonian cycles for two reasons. First, we want to experiment with both satisfiable and unsatisfiable instance. Second, the “hardest” instances usually occur at the phase transition [3, 4].

The Hamiltonian cycle problem demonstrates the effectiveness of systems which can easily model transitive closure. Systems which can model transitive closure use a reachability predicate to ensure all edges are included in the Hamiltonian cycle. Modeling the Hamiltonian cycle problem as satisfiability requires some notion of time to simulate transitive closure. The addition of time increases the size of the theories – frequently to such an extent that problems solved very quickly with \( DC \) or \( \text{smo}d\ell s \) are not practical for satisfiability solvers. The results in Fig. 7.1 show that for \( \text{sat}z \) instances with more than 40 vertices were on average impractical while \( \text{smo}d\ell s \) and \( \text{dcs} \) on the other hand had little difficulty even on large instances.

The spikes shown in Fig. 7.1 for \( V = 50 \) is the result of two instances from the 1000 instances averaged for that data point taking a very long time and for \( V = 90 \) it is the result of one instance from 1000 instances.

### 7.3.2 Graph Coloring

The problem of graph coloring is that of assigning colors to vertices such that vertices joined by an edge are not assigned the same color. Deciding if a graph is \( k\)-color-able for \( k \geq 3 \) is
NP-complete [13]. We look at the corresponding search problem of finding a legal assignment of colors to vertices.

The graphs for instances of coloring were one thousand randomly generated graphs for each of $v = 50, 100, \ldots, 300$ with density such that $\approx 50\%$ had solutions. Again, we choose this probability space so that we test both satisfiable and unsatisfiable instances in the “hard” region.

The graph coloring problem does not require the use of Horn clauses in DC theories for encoding. The size of SAT encodings for coloring are comparable to grounded DC theories unlike those for Hamilton cycle problems where DC uses Horn clauses. For smodels of course modeling is done as SLP. Figure 7.2 shows the rate of growth as the sizes of the problems increase being better for satz than for dcs. Scaling, in the current version of dcs, seems to be a problem.

7.3.3 N-Queens

The $n$-queens problem is that of placing $n$ queens on an $n \times n$ board such that no queen can remove another (two queens cannot be on the same row, column or diagonal). The $n$-queen problems are satisfiable for $3 > n$. The $n$-queens problem has many solutions and we can experiment solving for the first solution or for all solutions.

The $n$-queens problem was executed for $n = 8, 10, \ldots, 24$. The theories used for all three systems can be found in Appendix A. The results in Fig. 7.3 show a similar scaling for
all three systems with DC performing slightly better than satz and smodels worse. We believe the cardinality atoms in DC are responsible for the lower times because the grounded theories are smaller and the DC solver takes advantage of the cardinality atoms.

We also looked at the times for DC and smodels to find all solutions. (The satisfiability checker satz finds only the first solution.) DC performed better than smodels for all solutions. This indicates DC heuristics for pruning the search space and choosing branching atoms are effective.

### 7.3.4 Pigeonhole Problem

The pigeonhole problem requires that we place $p$ pigeons into $h$ holes such that there is at most one pigeon in each hole. The problem is always satisfiable if $h \geq p$ and unsatisfiable
otherwise. In this study we cast the problem as a search problem where $h < p$ thus unsatisfiable. We use the pigeonhole problem to compare the performance of systems when an exhaustive search is required to determine unsatisfiability.

We see from Fig. 7.4 that all three implementations had a similar increase in time as the size of the problem increased and \texttt{dcs} performed slightly more efficiently.

### 7.3.5 Schur Numbers

The Schur numbers problem is to place numbers $1 \ldots n$ into $b$ bins such that if $X + Y = Z$ then $Z$ cannot be in a bin with both $X, Y$.

We show the times for the largest $n$ for $b = 3, 4$ which is satisfiable and the smallest $n$ which is unsatisfiable. The Schur numbers are another problems where symmetry is impor-
Table 7.3: 3 × 3 Magic Square

<table>
<thead>
<tr>
<th>solutions</th>
<th>satz</th>
<th>dcs</th>
<th>smodels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>atoms</td>
<td>clauses</td>
<td>sec</td>
</tr>
<tr>
<td>1</td>
<td>81</td>
<td>6334</td>
<td>***</td>
</tr>
<tr>
<td>8</td>
<td>81</td>
<td>6334</td>
<td>***</td>
</tr>
</tbody>
</table>

***Stopped after one hour

tant. satz does much better than either DC or smodels for Schur numbers. We believe this is due to satz being able to effectively reduce problems with symmetry.

7.3.6 3 × 3 Magic Square

The 3 × 3 magic square problem requires the numbers 1, ..., 9 to be placed in a 3 × 3 grid such that each number is used only once and the numbers in each row, column and diagonal sum to 15.

The sizes of the grounded theories for DC, satz and smodels were comparable. Despite this DC performed much better not only for finding the first solution but also for finding all solutions.

7.3.7 Wire Routing

This problem comes from VLSI design where the object is to route multiple wires on a chip. Each wire has two terminal points. There are also components on the chip. Routes for each wire must be determined such that wires do not intersect or touch the components.

The small differences in the times between the 10 × 10 grids and 15 × 15 grids is very encouraging. Much more testing is needed to determine how well this modeling in DC and smodels scales. Current applications for VLSI design use non-deterministic algorithms thus work in this direction for deterministic methods is extremely important.

The results for wire routing indicates a lack of robustness in DC. Although DC performs as well or better than smodels on the majority of the instances there is a very large variance for a few instances. The lack of robustness in DC is an area that we will focus on in future work.
Table 7.4: Wire Routing

<table>
<thead>
<tr>
<th>instances</th>
<th>relaxation</th>
<th>dcs</th>
<th>smodels</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>$t = 1$</td>
<td>0.30</td>
<td>0.06</td>
</tr>
<tr>
<td>10.2</td>
<td>$t = 0$</td>
<td>0.00</td>
<td>0.13</td>
</tr>
<tr>
<td>10.3</td>
<td>$t = 0$</td>
<td>0.01</td>
<td>0.12</td>
</tr>
<tr>
<td>10.4</td>
<td>$t = 0$</td>
<td>0.02</td>
<td>0.09</td>
</tr>
<tr>
<td>10.5</td>
<td>$t = 1$</td>
<td>0.01</td>
<td>0.10</td>
</tr>
<tr>
<td>10.6</td>
<td>$t = 1$</td>
<td>0.00</td>
<td>0.08</td>
</tr>
<tr>
<td>10.7</td>
<td>$t = 1$</td>
<td>0.05</td>
<td>0.08</td>
</tr>
<tr>
<td>10.8</td>
<td>$t = 1$</td>
<td>0.02</td>
<td>0.10</td>
</tr>
<tr>
<td>10.9</td>
<td>$t = 2$</td>
<td>11.76</td>
<td>0.20</td>
</tr>
<tr>
<td>10.10</td>
<td>$t = 0$</td>
<td>0.01</td>
<td>0.09</td>
</tr>
</tbody>
</table>

Table 7.5: Wire Routing

<table>
<thead>
<tr>
<th>instance</th>
<th>relaxation</th>
<th>dcs</th>
<th>smodels</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1</td>
<td>$t = 0$</td>
<td>170.92</td>
<td>0.26</td>
</tr>
<tr>
<td>15.2</td>
<td>$t = 0$</td>
<td>0.04</td>
<td>0.25</td>
</tr>
<tr>
<td>15.3</td>
<td>$t = 0$</td>
<td>0.04</td>
<td>0.23</td>
</tr>
<tr>
<td>15.4</td>
<td>$t = 2$</td>
<td>0.08</td>
<td>0.38</td>
</tr>
<tr>
<td>15.5</td>
<td>$t = 0$</td>
<td>0.03</td>
<td>0.22</td>
</tr>
<tr>
<td>15.6</td>
<td>$t = 0$</td>
<td>0.07</td>
<td>0.86</td>
</tr>
<tr>
<td>15.7</td>
<td>$t = 0$</td>
<td>5.70</td>
<td>0.16</td>
</tr>
<tr>
<td>15.8</td>
<td>$t = 0$</td>
<td>0.03</td>
<td>0.24</td>
</tr>
<tr>
<td>15.9</td>
<td>$t = 0$</td>
<td>0.03</td>
<td>0.22</td>
</tr>
<tr>
<td>15.10</td>
<td>$t = 0$</td>
<td>0.02</td>
<td>0.20</td>
</tr>
</tbody>
</table>
7.4 Summary of Results

We experimented with $DC$ using problems based on graphs, combinatorial problems and planning. The results show that $DC$ was more efficient overall with respect to time than either $satz$ or $smodels$. Both $satz$ and $smodels$ are state of the art implementation that have been under development and in use for several years.

The $k$-coloring problem was an example where $satz$ performed better than $DC$. It is one problem where the CNF theories are very close in size to the grounded $DC$ theories. In this case, $DC$ does not have the advantage of much more concise theories. For the $n$-queens and the pigeonhole problem as well as the $k$-coloring problem $satz$ uses the symmetry of the problem to decrease the search space. Also the implementation is optimized for finding the first solution.

Results of comparisons with $smodels$ show that $DC$ does better for most problems. The wire routing problem being the most notable example of $smodels$ performing better than $DC$. Even in this instance, we believe that the large variance in the results for $DC$ will decrease as the robustness of the implementation is improved.

The results of the experiments show that $DC$ performs very well in comparison to other implementations. We believe that $DC$ has applicability to a large range of practical problems.
Chapter 8
Conclusions

In this thesis we present a new answer-set programming system \textit{DC}. We designed \textit{DC} keeping in mind the criteria for comparison of answer-set programming systems in Chapter 3. The next section reiterates how we meet our goals for an answer-set programming.

8.1 Meeting the goals of answer-set programming

The language of \textit{DC} which is an extension of propositional schemata facilitates modeling of problems. First, constraints are easily modeled in predicate logic. Second, Horn clauses can be used for problems where transitive closure is an intuitive method for modeling some concepts of the problem. Cardinality constraints are easily understood and are useful in modeling problems where requiring or limiting the number of atoms is needed.

The semantics of \textit{DC} is based on propositional logic and despite the addition of Horn clauses and cardinality constraints simplicity is maintained. The simple semantics is useful when modeling problems and during solving.

We show in Chapter 4 that the expressive power of \textit{DC} is equivalent to that of stable logic programming. We can use \textit{DC} to uniformly solve search problems.

The grounded \textit{DC} theories are concise. By retaining the structure of the cardinality constraints we greatly reduce the size of the grounded theories. The semantics of the Horn clauses are also retained in the grounded theories. Thus there is no need for additional atoms and clauses to be used for translating the grounded Horn clauses to normal clauses.

The results of comparisons between executions of \textit{smo}dels, \textit{sat}z and \textit{dcs} show that the \textit{DC} system has time performance equal to or faster than current state-of-the-art implementations used as general solvers.

8.2 Future work

The \textit{DC} system is a viable alternative to current systems. However, there are several issues which require further investigation. First, efforts to make the \textit{DC} solver more stable are needed. We can see evidence of instability when looking at the results of Hamiltonian cycles (Fig. 7.1) experiments where a very few outliers have an affect on average values. Heuristics to resolve these anomalies are needed.

Planning problems which include the concept of time may be solved more efficiently if there is a integration of the grounder and solver. This would allow partial grounding of theories.
We are interested in search problems in the class NP. These problems are exponential time problems in the worst case, therefore it is often not practical to try to solve them using complete algorithms. Adding a probabilistic option to $DC$ is a possible way to maximize the benefits of the concise $DC$ theories along with the advantages of restarts used in conjunction with probabilistic methods.
Appendix A

Theories

We present both smodels and DC theories which were used for experimentation. The CNF formulas for SAT problems were encoded by either utility programs written to encode specific problems or PSgrnd which grounds theories similar to dcgrnd, but with the output in CNF format. PSgrnd can be used only when the theory is written with neither cardinality atoms nor Horn clauses.

A.1 Hamiltonian cycle problem

Let $G = (V, E)$ be a directed graph with a set of vertices $V = \{v_1, \ldots, v_n\}$ and a set of edges $E$. The graph $G$, has a Hamiltonian cycle $C$ if there is a path that passes through each vertex exactly once and is closed.

A.1.1 DC theory for Hamiltonian cycle

% Assumes graph vertices defined as vtx(a) and edges as edge(a,b).
pred visit(vtx).
pred hc(vtx,vtx):edge.
var vtx X,Y.
%Exactly one outgoing edge
  1 \{hc(X,_)\} 1.
%Exactly one incoming edge
  1 \{hc(_,Y)\} 1.
%Transitive closure
  visit(Y) :- visit(X), hc(X,Y).
%Initialization of first vertex
  visit(Y) :- hc(1,Y).
%Post-constraint--all vertices must be visited
  visit(X).

A.1.2 smodels theory for Hamiltonian cycle

% Assumes graph vertices defined as vtx(a) and edges as edge(a,b).
% Atoms hc(X,Y) in the stable models provide the circuit.
%Exactly one incoming edge
  1 \{ hc(V1,V2) : vtx(V1) \} 1 :- vtx(V2).
% Exactly one outgoing edge
1 { hc(V1,V2) : vtx(V2) } 1 :- vtx(V1).

% Transitive closure
reached(V2) :- edge(V1,V2), hc(V1,V2), reached(V1), not initialnode(V1).
reached(V2) :- edge(V1,V2), hc(V1,V2), initialnode(V1).

% Initialization of first vertex
initialnode(1).

% All vertices must be visited
:- vtx(V), not reached(V).

A.1.3 SAT description of Hamiltonian cycle

The DC theory for the Hamilton cycle problem problem uses Horn clauses and our grounded
does not ground Horn clauses into CNF format. Therefore, we use a utility program for
coding graphs to determine if there is a Hamiltonian cycle. A brief explanation of the
coding is given here.

The graph, G(V, E), contains a Hamiltonian cycle if there is a path that passes through
each vertex exactly once and is closed. Since we must have a path through all vertices, we
can choose any vertex v₀ as our starting and ending vertex. We must have exactly n edges in
the path, one of each distance 1, ..., n from v₀. The path variable, path(vᵢ, vⱼ, k) represents
the edge(vᵢ, vⱼ) where vⱼ is distance k on the path from v₀ and k ∈ {1, 2, ..., n}.

Let T be a theory constructed from G. We define rules, HamiltonCycle(G) for the sets
of clauses:

\[ \bigvee \{ path(v₀, w, 1) : w \in \Gamma^+(v₀) \} \]
\[ \bigvee \{ path(w, v₀, n) : w \in \Gamma^-(v₀) \} \]

For each edge(v, w) ∈ E and for each l ∈ {1, 2, ..., n}:

\[ path(v, w, l) \iff \bigwedge \{ \neg path(v, w', l') : w' \neq w, l' = l, w' \in \Gamma^+(v), \text{ or } l' \neq l w' \in \Gamma^+(v) \} \]

For each edge(v, w) ∈ E and for each l ∈ {1, 2, ..., n}:

\[ path(v, w, l) \iff \bigwedge \{ \neg path(v', w, l') : v' \neq v, l' = l, v' \in \Gamma^-(w), \text{ or } l' \neq l v' \in \Gamma^-(w) \} \]

For each w ≠ v₀ ∈ G and k ∈ {1, 2, ..., n − 1}:

\[ \bigvee \{ path(v, w, k) : v \in \Gamma^-(w) \} \iff \bigvee \{ path(w, u, k + 1) : u \in \Gamma^+(w) \} \]
A.2 \textit{k-coloring of graphs}

The problem of \textit{k-coloring} of a graph is that of assigning \textit{k} colors to vertices such that each vertices is assigned exactly one color and vertices joined by an edge are not assigned the same color. Deciding if a graph is \textit{k-color-able} for \( k \geq 3 \) is NP-complete [13]. We look at the corresponding search problem of finding a valid assignment of colors to vertices.

A.2.1 \textit{DC} theory for graph coloring

\%For an undirected graph with vertices as \texttt{vtx(a)} and edges as \texttt{edge(a,b)}
\% and colors are color[1..k]
\quad \text{pred clr(vtx,color)}.
\quad \text{var vtx X,Y}.
\quad \text{var color K}.
\%Exactly one color for each vertex
\quad 1\{\text{clr}(X,\_)}1.
\%Vertices of an edge can not be the same color
\quad (\text{clr}(X,K), \text{clr}(Y,K), \text{edge}(X,Y) \rightarrow).

A.2.2 \textit{smo}dels theory for graph coloring

\%For an undirected graph with vertices as \texttt{vtx(a)} and edges as \texttt{edge(a,b)}
\% set of colors
\quad \text{color(1..n)}.
\%Exactly one color for each vertex
\quad 1 \{ \text{node_color}(N, C) : \text{color(C)} \} 1 :- \text{vtx(N)}.
\%Vertices of an edge can not be the same color
\quad :- \text{node_color}(X, C), \text{node_color}(Y, C), \text{edge}(X,Y), \text{color}(C).

A.2.3 \textit{PS}grnd theory for graph color-ability

The theory for color-ability can be grounded using a version of \texttt{dcgrnd} for CNF. This version does not allow Horn clauses or cardinality clauses. The theory therefore is written slightly different than that for \textit{DC}.

\%For an undirected graph with vertices as \texttt{vtx(a)} and edges as \texttt{edge(a,b)}
\% and colors are color[1..k]
\quad \text{pred clr(vtx,color)}.
\quad \text{var vtx X,Y}.
var color K, C.
%At least one color for each vertex
    clr(X, _).
%At most one color for each vertex
    clr(X, K), clr(X, C), K != C -> .
%Vertices of an edge can not be the same color
    clr(X, K), clr(Y, K), edge(X, Y) -> .

A.3 n-queens

The n-queens problem requires the placement of n queens on an \( n \times n \) board in such a manner that no queen can remove another (two queens cannot be on the same row, column or diagonal).

A.3.1 DC theory for n-queens

The data file consists of only one line as shown below. The data values of number is a range from 1 to \( q \). The constant value for \( q \) will be entered on the command line with the constant option. The data file \( \text{qd} \):

number[1..q].

The rule file \( \text{queen} \):

%predicate indicates the column and row position.
    pred queen(number, number).
    var number C, R, I.
%Exactly one queen must be in each column
    1{queen(C, _)}1.
%Exactly one queen must be in each row
    1{queen(_, R)}1.
%Two queens must not be in diagonal to each other
    queen(C, R), queen(C+I, R+I) -> .
    queen(C, R), queen(C+I, R-I) -> .

A.3.2 smodels theory for n-queens

%Data for smodels theories can be placed in program file.
number(1..q).
%predicate indicates the column and row position.
%Exactly one queen must be in each column
 1{queen(C,R):number(R)}1 :- number(C).
%Exactly one queen must be in each row
 1{queen(C,R):number(C)}1 :- number(R).
%Two queens must not be in diagonal to each other
  :- queen(C,R), queen(C+I,R+I), number(C;R;I).
  :- queen(C,R), queen(C+I,R-I), number(C;R;I).

A.3.3 PSgrnd theory for n-queens

The data file consists of only one line as shown below. The data values of number is a range from 1 to q. The constant value for q will be entered on the command line with the constant option. The data file qd:

number[1..q].

The rule file queen:

%predicate indicates the column and row position.
  pred queen(number,number).
  var number C, R, I.
%Exactly one queen must be in each column
%These two clauses replace the single clause with a cardinality atom
  queen(C,\_).
  queen(C,R), queen(C,I), R!=I ->.
%Exactly one queen must be in each row
%These two clauses replace the single clause with a cardinality atom
  queen(\_,R).
  queen(C,R), queen(I,R), C!=I ->.
%Two queens must not be in diagonal of each other
  queen(C,R), queen(C+I,R+I) -> .
  queen(C,R), queen(C+I,R-I) -> .

A.4 Schur numbers

The problem is to place numbers 1...n into b bins such that if X + Y = Z then Z cannot be in the same bin as X,Y.
A.4.1 DC theory for Schur numbers

The data file for Schur numbers:

\[
\text{bin}[1..b]. \\
\text{number}[1..n].
\]

where \( b, n \) are constants given on the dcgrnd command line.

The rule file \texttt{schur}:

\[
\text{pred in(bin, number).} \\
\text{var bin B.} \\
\text{var number X,Y,Z.} \\
\%\text{Each number is in exactly one bin}  \\
\text{1\{in(_,X)\}1.} \\
\text{in(B,X), in(B,Y), in(B,Z), X+Y==Z, X<=Y ->.} \\
\%\text{Optional clause to reduce symmetry of the problem}  \\
\text{in(B,X), B>X ->.}
\]

A.4.2 smodels theory for Schur numbers

\%The data for Schur numbers can be placed in the program file:

\[
\text{bin(1..b).} \\
\text{number(1..n).} \\
\%\text{Each number is in exactly one bin}  \\
\text{1\{in(B,X):bin(B)\}1 :- number(X).} \\
\text{:- in(B,X), in(B,Y), in(B,Z), X+Y==Z, X<=Y.} \\
\%\text{Optional clause to reduce symmetry of the problem}  \\
\text{:- in(B,X), B>X.}
\]

A.4.3 PScrnd theory for Schur numbers

The data file for Schur numbers is the same as for \textit{DC}. The rule file \texttt{schur}:

\[
\text{pred in(bin, number).} \\
\text{var bin B.} \\
\text{var number X,Y,Z.} \\
\%\text{Each number is in at least one bin}  \\
\text{in(_,X).} \\
\%\text{Each number is in at least one bin}
\]
in(B,X), in(C,X), C!=B -> .
in(B,X), in(B,Y), in(B,Z), X+Y==Z, X<=Y -. 
%Reduces symmetry of the problem
in(B,X), B>X -. 

A.5 3 x 3 Magic Square

The 3 x 3 magic square problem requires the numbers 1,..., 9 to be placed in a 3 x 3 grid such that each number is used only once and the numbers in each row, column and diagonal sum to 15.

A.5.1 DC theory for 3 x 3 Magic Square

The data file for 3 x 3 magic square:

size[1..3].
number[1..9].
diagonal[1,2].

The theory for magic square:

pred square(size,size,number).

var number X,Y,Z.
%Each position is assigned exactly one number
1{square(I,J,_)}1.
%Each number is placed in exactly one position
1{square(_,_,X)}1.
%The sum of the numbers in each row is 15
square(I,1,X),square(I,2,Y),square(I,3,Z),X+Y+Z!=15 -. 
%The sum of the numbers in each column is 15
square(1,I,X),square(2,I,Y),square(3,I,Z),X+Y+Z!=15 -. 
%Diagonal sums must also be 15
square(1,1,X),square(2,2,Y),square(3,3,Z),X+Y+Z!=15 -. 
square(1,3,X),square(2,2,Y),square(3,1,Z), X+Y+Z!=15 -. 

A.5.2 smodels theory for 3 x 3 Magic Square

The data is included in the program file for 3 x 3 magic square:
size(1..3).
number(1..9).

Each position is assigned exactly one number
1{square(I,J,N):number(N)}1 :- size(I,J).

Each number is placed in exactly one position
1{square(I,J,N):size(I;J)}1 :- number(I;J).

A row is completed when
The sum of the numbers in each row is 15
:- square(I,1,X),square(I,2,Y),square(I,3,Z),X+Y+Z!=15.

A column is completed when
The sum of the numbers in each column is 15
:- square(1,I,X),square(2,I,Y),square(3,I,Z),X+Y+Z!=15.

Diagonal sums must also be 15
:- square(1,1,X),square(2,2,Y),square(3,3,Z),X+Y+Z!=15.

:- square(1,3,X),square(2,2,Y),square(3,1,Z),X+Y+Z!=15.

A.6 Pigeon-Hole Problem

The pigeon-hole problem requires that we place \( p \) pigeons into \( h \) holes such that there is at most one pigeon in each hole. The problem is always unsatisfiable if \( h < p \).

A.6.1 DC theory for pigeon-hole problem

The data file \texttt{ph}:

\begin{verbatim}
pigeon[1..p].
hole[1..h].
\end{verbatim}

The rule file \texttt{pigeon}:

\begin{verbatim}
pred inhole(hole,pigeon).
var hole H.
var pigeon P.

At most one pigeon can be in each hole
{inhole(H,\_)}1.

Each pigeon must be in exactly one hole
1{inhole(\_,P)}1.
\end{verbatim}
A.6.2 smodels theory for pigeon-hole problem

%Data for pigeon-hole problem
  pigeon(1..p).
  hole(1..h).
%At most one pigeon can be in each hole
  \{inhole(H,\_)}1.
%Each pigeon must be in exactly one hole
  1\{inhole(\_,P)}1.

A.6.3 PSgrnd theory for pigeon-hole problem

The same data file used by DC. The rule file pigeon:

  pred inhole(hole,pigeon).
  var hole H.
  var pigeon P, Q.
%Each pigeon must be a hole
  inhole(\_,P).
%At most one pigeon can be in each hole
  inhole(H,P), inorder(H,Q), P!=Q ->.

A.7 Wire Routing

This problem comes from VLSI design where the object is to route multiple wires on a chip. Each wire has source and sink points. There are also components on the chip. Routes for each wire must be determined such that wires do not intersect or touch the components.

This modeling of wire routing differs from that given by Erdem et al.[12] where the problem is described in terms of robots traveling paths. Each robot represents a wire which can move one position in each time step and does not cross another robots path. Each robot moves from a starting position to a goal position. The starting and goal positions are used to represent the terminal points of the wire. Our program for determining wire routes sets constraints on the choice of points that can be included in the path for each wire.

A.7.1 DC theory for Wire Routing

The data predicate block gives the coordinates for a point on grid that cannot be included in a path. The data predicate terminal is the coordinates for the original points of each wire. Each wire W has two terminal points represented by terminal(I,J,W) where I, J are the
coordinates of the terminal points. The constants \( m, k \) are entered on the `dcsgnrd` command line. We use \( m \) to limit the number of points allowed on each path and to eliminated points of distance \( m \) from either terminal point. The constant \( k \) is used to relax restrictions on the rectangular search space.

The rule file `wire`:

```plaintext
pred path(coord, coord, wire).
var coord I, J, L, M, P, Q.
var wire W, X, Y.
% Each point can have at most one wire.
% Prevents intersection of wire.
{path(I, J, _)}1.
% This rule can be commented out or
% uses to limit the length of each path
{path(_, _, W)}m.
% If the path length is limited then this
% clause is used to restrict search space.
terminal(I, J, W), path(L, M, W), abs(I-L)+abs(J-M) > m -> .
% Each wires terminal points must be one the wires path
terminal(I, J, W) -> path(I, J, W).
% Each terminal point has exactly one adjacent point on the path
terminal(I, J, W) ->
1{path(L, M, W): coord(L): coord(M): abs(I-L)+abs(J-M)==1}1.
% Each non terminal point on the path has
% exactly two adjacent points on the path
path(I, J, W) -> terminal(I, J, W) |
% Blocked points cannot be on any path
block(I, J), path(I, J, W) ->.
% Eliminates independent four point cycle
path(I, J, W), path(I+1, J, W), path(I+1, J+1, W), path(I, J+1, W) ->.
% Limits search space for each wire to
% rectangle formed by it terminal points
terminal(L, M, W), terminal(P, Q, X), path(I, J, Y), W==X, X==Y, I+k < L, I+k<P, L!=P -> .
terminal(L, M, W), terminal(P, Q, X), path(I, J, Y), W==X, X==Y, I-k > L, I-k>P, L!=P -> .
```
terminal(L,M,W), terminal(P,Q,X), path(I,J,Y), W==X,X==Y, J+k < M, J+k<Q, M!Q ->.
terminal(L,M,W), terminal(P,Q,X), path(I,J,Y), W==X,X==Y, J-k > M, J-k>Q, M!Q ->.

A.7.2 smodels theory for Wire Routing

%Each point may have at most one wire.
%Prevents intersection of wire.
{ path(I,J,W) : wire(W) } 1:- coord(I;J).
%This rule can be commented out
%or used to limit the length of each path
{ path(I,J,W) : coord(I):coord(J) } k:- wire(I;J).
%If the path length is limited
%then this clause is used to restrict search space.
:- path(I,J,W), terminal(M,N,W), coord(I;J), abs(I-M)+abs(J-N) > k.
%Each wires terminal points must be on the wires path
path(I,J,W) :- terminal(I,J,W).
%Each terminal point has exactly one adjacent point on the path
1{ path(M,N,W):coord(M):coord(N):(abs(I-M)+abs(J-N))==1}1 :-
terminal(I,J,W).
%Each non terminal point on the path has
%exactly two adjacent points on the path
2{ path(M,N,W):coord(M):coord(N):(abs(I-M)+abs(J-N))==1}2 :-
path(I,J,W), not terminal(I,J,W), coord(I;J), wire(W).
%Blocked points cannot be on any path
:- path(I,J,W), block(I,J), coord(I;J), wire(W).
%Eliminates independent four point cycle
:- path(I,J,W), path(I+1,J,W), path(I+1,J+1,W), path(I,J+1,W),
  coord(I;J), wire(W).
%Limits search space for each wire to
%rectangle formed by it terminal points
:- path(I,J,W),terminal(L,M,W), terminal(K,N,W), I + k <= L,
  I +k <=K, L!=K, coord(I;J;K;L;M;N), wire(W).
:- path(I,J,W),terminal(L,M,W), terminal(K,N,W), I - k >= L,
  I-k>=K, L!=K, coord(I;J;K;L;M;N), wire(W).
\[\text{:- path}(I,J,W), \text{terminal}(L,M,W), \text{terminal}(K,N,W), J + k \leq M, J + k \leq N, M! = N, \text{coord}(I;J;K;L;M;N), \text{wire}(W).\]

\[\text{:- path}(I,J,W), \text{terminal}(L,M,W), \text{terminal}(K,N,W), J - k \geq M, J - k \geq N, M! = N, \text{coord}(I;J;K;L;M;N), \text{wire}(W).\]

A.7.3 Psgrnd theory for Wire Routing

Theory for wire routing in Psgrnd is not given.
Bibliography


Vita

Deborah East
DOB 1-20-52

Education

Ph.D. candidate in Computer Science, University of Kentucky.

B.S. in Computer Science, University of Kentucky, 1992. Departmental Honors.

Professional Experience

Research Assistant: (May 1997 - Present), University of Kentucky.

Teaching Assistant: (Jan 1994 - May 1997), University of Kentucky.


Research Interests

Developing declarative languages for modeling constraint satisfaction problems and designing constraint satisfaction solvers.
Software simulations and modeling.

Teaching Interests

Artificial intelligence, logic, expert systems, logic programming, algorithms, constraint satisfaction.

Skills

Languages: Pascal, Fortran, C, C++, Java, HTML, Perl, Prolog
Operating Systems: Unix, DOS, Windows, Prime, Linux
Utilities, etc: C-scape, ODBC, SQL, MS WORD office, LaTex

Awards and Distinctions

Member of UPE National Honor Society for Computing Sciences, Gamma Chapter of Kentucky
Thaddeus B. Curtz Memorial Scholarship Award in Computer Science, Spring 1994
Diploma in Computer Sciences with honors, University of Kentucky, 1992
Professional Organizations

American Association for Artificial Intelligence
Association for Computing Machinery
Association for Logic Programming

Publications


Presentations


Software

**dcsgrnd** – grounding program for DATALOG with Constraints.
**dcs** – solver for DATALOG with Constraints.

**psgrnd** – grounding program for propositional schemata.