Computational Logic

Logic Programming

Georg Moser

Institute of Computer Science @ UIBK

Summer 2015

Expert System with Certification Factor

solve4/1

```
solve4(true,1,_Treshold) :-
        !.
solve4((A,B),C,Treshold) :-
        !,
        solve4(A,C1,Treshold),
        solve4(B,C2,Treshold),
        minimum(C1,C2,C).
solve4(A,C,Treshold) :-
        clause_cf(A,B,C1),
        C1 > Treshold,
        Treshold1 is Treshold / C1,
        solve4(B,C2,Treshold1),
        C is C1 * C2.
```

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming, constraint logic programming, answer set programming

Outline of the Lecture

Logic Programs

introduction, basic constructs, database and recursive programming, theory of logic programs

The Prolog Language

programming in pure prolog, arithmetic, structure inspection, meta-logical predicates, cuts, extra-logical predicates, how to program efficiently

Advanced Prolog Programming Techniques

nondeterministic programming, incomplete data structures, definite clause grammars, meta-programming, constraint logic programming, answer set programming

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \mathrm{S\ E\ N\ D} \\ \mathrm{M\ O\ R\ E} \\ \hline \mathrm{M\ O\ N\ E\ Y} \end{array} \quad +$$

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \mathrm{S\ E\ N\ D} \\ \mathrm{M\ O\ R\ E} \\ \hline \mathrm{1\ O\ N\ E\ Y} \end{array} \quad +$$

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \mathsf{S\ E\ N\ D} \\ \mathsf{1\ O\ R\ E} \\ \hline \mathsf{1\ O\ N\ E\ Y} \end{array} \quad +$$

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} \mathsf{S\ E\ N\ D} \\ \mathsf{1\ 0\ R\ E} \\ \hline \mathsf{1\ 0\ N\ E\ Y} \end{array} +$$

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9\,E\,N\,D \\ 1\,0\,R\,E \\ \hline 1\,0\,N\,E\,Y \end{array} +$$

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9\ \mathrm{E}\ \mathrm{N}\ \mathrm{D} \\ 1\ 0\ 8\ \mathrm{E} \\ \hline 1\ 0\ \mathrm{N}\ \mathrm{E}\ \mathrm{Y} \end{array} \quad +$$

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9\ 5\ N\ D \\ 1\ 0\ 8\ 5 \\ \hline 1\ 0\ N\ 5\ Y \end{array} +$$

Cryptarithmetic

Definition

- a cryptarithmetic problem is a puzzle in which each letter represents a unique digit $\leqslant 9$
- the object is to find the value of each letter
- first digit cannot be 0

Example

$$\begin{array}{r} 9\,5\,6\,7 \\ 1\,0\,8\,5 \\ \hline 1\,0\,6\,5\,2 \end{array} +$$

First Attempt

generate and test

```
solve([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]]) :-
        Digits = [D, E, M, N, O, R, S, Y],
        Carries = [C1,C2,C3,C4],
        selects(Digits, [0,1,2,3,4,5,6,7,8,9]),
        members(Carries, [0,1]),
        M           =:=          C4,
        O + 10 * C4 =:= S + M + C3,
        N + 10 * C3 =:= E + O + C2,
        E + 10 * C2 =:= N + R + C1,
        Y + 10 * C1 =:= D + E,
        M > 0, S > 0.

:- solve(X),
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]].
```

Discussion

very inefficient

```
?- time(solve(X)).
% 133,247,057 inferences,
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

Discussion

very inefficient

```
?- time(solve(X)).
% 133,247,057 inferences,
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it's purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

Discussion

very inefficient

```
?- time(solve(X)).
% 133,247,057 inferences,
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it's purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

improvement

- move testing into generating
- destroys clean structure of program

Discussion

very inefficient

```
?- time(solve(X)).
% 133,247,057 inferences,
% 7.635 CPU in 7.667 seconds (100% CPU, 17452690 Lips)
X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]]
```

explanation

- generate-and-test in it's purest form
- all guesses are performed before the constraints are checked
- arithmetic checks cannot deal with variables

improvement

- move testing into generating
- destroys clean structure of program
- any other ideas?

Constraint Logic Programming

Definitions (CLP on finite domains)

- `use_module(library(clpfd))` loads the clpfd library
- `Xs ins N .. M` specifies that all values in *Xs* must be in the given range
- `all_different(Xs)` specifies that all values in *Xs* are differnt
- `label(Xs)` all variables in *Xs* are evaluated to become values
- `#=`, `#\=`, `#>`, ... like the arithmetic comparison operators, but may contain (constraint) variables

Constraint Logic Programming

Definitions (CLP on finite domains)

- `use_module(library(clpfd))` loads the clpfd library
- `Xs ins N .. M` specifies that all values in *Xs* must be in the given range
- `all_different(Xs)` specifies that all values in *Xs* are differnt
- `label(Xs)` all variables in *Xs* are evaluated to become values
- `#=`, `#\=`, `#>`, ... like the arithmetic comparison operators, but may contain (constraint) variables

standard approach

- load the library
- specify all constraints
- call `label` to start efficient computation of solutions

Second Attempt

constraint logic program

```
solve([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]]) :-
        Digits = [D, E, M, N, O, R, S, Y],
        Carries = [C1,C2,C3,C4],
        Digits  ins 0 .. 9, all_different(Digits),
        Carries ins 0 .. 1,
       M           #=          C4,
       O + 10 * C4 #= S + M + C3,
       N + 10 * C3 #= E + O + C2,
       E + 10 * C2 #= N + R + C1,
       Y + 10 * C1 #= D + E,
       M #> 0, S #> 0,
        label(Digits).
```

8 queens (as before)

```
queens(Xs) :- template(Xs),solution(Xs).

template([1/_Y1,2/_Y2,3/_Y3,4/_Y4,
          5/_Y5,6/_Y6,7/_Y7,8/_Y8]).

solution([]).
solution([X/Y|Others]) :-
        solution(Others),
        member(Y, [1,2,3,4,5,6,7,8]),
        noattack(X/Y, Others).

noattack(_,[]).
noattack(X/Y,[X1/Y1|Others]) :-
        Y =\= Y1,
        Y1 - Y =\= X1 - X,
        Y1 - Y  =\= X - X1,
        noattack(X/Y,Others).
```

n-queens (using clp)

```
nqueens(N,Qs) :-
        length(Qs,N),
        Qs ins 1 .. N, all_different(Qs),
        constraint_queens(Qs),
        label(Qs).

constraint_queens([]).
constraint_queens([Q|Qs]) :-
        noattack(Q,Qs,1),
        constraint_queens(Qs).

noattack(_,[],_).
noattack(X,[Q|Qs],N) :-
        X #\= Q+N,
        X #\= Q-N,
        M is N+1,
        noattack(X,Qs,M).
```

Definition

- Sudoku is a well-known logic puzzle; usually played on a 9×9 grid
- $\forall$ *cells*: *cells* $\in \{1, \ldots, 9\}$
- $\forall$ *rows*: all entries are different
- $\forall$ *colums*: all entries are different
- $\forall$ *blocks*: all entries are different

Definition

- Sudoku is a well-known logic puzzle; usually played on a 9×9 grid
- $\forall$ *cells*: *cells* $\in \{1, \ldots, 9\}$
- $\forall$ *rows*: all entries are different
- $\forall$ *colums*: all entries are different
- $\forall$ *blocks*: all entries are different

Main Loop (using clp)

```
sudoku(Puzzle) :-
        show(Puzzle),
        flatten(Puzzle,Cells),
        Cells ins 1 .. 9,
        rows(Puzzle),
        cols(Puzzle),
        blocks(Puzzle),
        label(Cells),
        show(Puzzle).
```

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

row/1

```
rows([]).
rows([R|Rs]) :-
        all_different(R), rows(Rs).
```

auxiliary predicates

- *flatten*/2 flattens a list
- *show*/1 prints the current puzzle

row/1

```
rows([]).
rows([R|Rs]) :-
        all_different(R), rows(Rs).
```

row/1 (alternative)

```
rows(Rs) :- maplist(all_distinct,Rs).
```

cols/1

```
cols([[]|_]).
cols([
        [X1|R1],
        [X2|R2],
        [X3|R3],
        [X4|R4],
        [X5|R5],
        [X6|R6],
        [X7|R7],
        [X8|R8],
        [X9|R9]]) :-
          all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
          cols([R1,R2,R3,R4,R5,R6,R7,R8,R9]).
```

cols/1

```
cols([[]|_]).
cols([
        [X1|R1],
        [X2|R2],
        [X3|R3],
        [X4|R4],
        [X5|R5],
        [X6|R6],
        [X7|R7],
        [X8|R8],
        [X9|R9]]) :-
          all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
          cols([R1,R2,R3,R4,R5,R6,R7,R8,R9]).
```

cols/1 (alternative)

use *maplist*/2

blocks/1

```
blocks([]).
blocks([[],[],[]|Rs]) :- blocks(Rs).
blocks([[X1,X2,X3|R1],
        [X4,X5,X6|R2],
        [X7,X8,X9|R3] |Rs]) :-
        all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
        blocks([R1,R2,R3|Rs]).
```

blocks/1

```
blocks([]).
blocks([[],[],[]|Rs]) :- blocks(Rs).
blocks([[X1,X2,X3|R1],
        [X4,X5,X6|R2],
        [X7,X8,X9|R3] |Rs]) :-
        all_different([X1,X2,X3,X4,X5,X6,X7,X8,X9]),
        blocks([R1,R2,R3|Rs]).
```

Example

```
:- sudoku([[1,_,_,_,_,_,_,_,_],
           [_,_,2,7,4,_,_,_,_],
           [_,_,_,5,_,_,_,_,4],
           [_,3,_,_,_,_,_,_,_],
           [7,5,_,_,_,_,_,_,_],
           [_,_,_,_,_,9,6,_,_],
           [_,4,_,_,_,6,_,_,_],
           [_,_,_,_,_,_,_,7,1],
           [_,_,_,_,_,1,_,3,_]]).
```

Demo

The New Kid on the Block

Answer Set Programming

- novel approach to modeling and solving search and optimisation problems
- ¬ programming, but a specification language
- ¬ Turing complete
- purely declarative
- restricted to finite models

The New Kid on the Block

Answer Set Programming

- novel approach to modeling and solving search and optimisation problems
- ¬ programming, but a specification language
- ¬ Turing complete
- purely declarative
- restricted to finite models

Success Stories

- team building for cargo at Gioia Tauro Seaport

The New Kid on the Block

Answer Set Programming

- novel approach to modeling and solving search and optimisation problems
- ¬ programming, but a specification language
- ¬ Turing complete
- purely declarative
- restricted to finite models

Success Stories

- team building for cargo at Gioia Tauro Seaport
- expert system in space shuttle

The New Kid on the Block

Answer Set Programming

- novel approach to modeling and solving search and optimisation problems
- $\neg$ programming, but a specification language
- $\neg$ Turing complete
- purely declarative
- restricted to finite models

Success Stories

- team building for cargo at Gioia Tauro Seaport
- expert system in space shuttle
- natural language processing
- ...

Propositional Setting

Definitions

- atoms, facts, rules are defined as before

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure
- disjunctions may appear in the head

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure
- disjunctions may appear in the head
- an answer set is a set of atoms corresponding to the minimal model of the program

Propositional Setting

Definitions

- atoms, facts, rules are defined as before
- only constants (= propositions) are allowed as atoms
- negation is negation as failure
- disjunctions may appear in the head
- an answer set is a set of atoms corresponding to the minimal model of the program

Example (Negation as Failure)

```
light_on :- power_on, not broken.
power_on.
```

answer set: $\{power\_on, light\_on\}$

Example (Disjunctive Heads)

```
open | closed :- door.
```

answer sets: $\{open\}$, $\{closed\}$

Example (Disjunctive Heads)

```
open | closed :- door.
```

answer sets: $\{open\}$, $\{closed\}$

Example

```
a | b.
a | c.
```

answer sets: $\{a\}$ and $\{b, c\}$

Example (Disjunctive Heads)

```
open | closed :- door.
```

answer sets: $\{open\}$, $\{closed\}$

Example

```
a | b.
a | c.
```

answer sets: $\{a\}$ and $\{b, c\}$

```
a | b.
a :- b.
```

answer set: $\{a\}$, but not $\{b\}$ nor $\{a, b\}$

Definition

constraints are negative assertions, representing fact that must not occur in any model of the program

Definition

constraints are negative assertions, representing fact that must not occur in any model of the program

Example

```
a :- not a, b.
```

any answer set must not contain b and simplifies to

```
:- b.
```

Definition

constraints are negative assertions, representing fact that must not occur in any model of the program

Example

```
a :- not a, b.
```

any answer set must not contain b and simplifies to

```
:- b.
```

Additional Features

- finite choice functions: $\{fact_1, fact_2, fact_3\}$.
- choice and counting: $1\{fact_1, fact_2, fact_3\}2$.
  "1" or "2" may be missing

First-Order Setting

Definition

- extension of first-order language
- no function symbols

First-Order Setting

Definition

- extension of first-order language
- no function symbols

Example (3-colouring)

```
red(X) | green(X) | blue(X).
:- red(X), red(Y), edge(X,Y).
:- green(X), green(Y), edge(X,Y).
:- blue(X), blue(Y), edge(X,Y).
```

First-Order Setting

Definition

- extension of first-order language
- no function symbols

Example (3-colouring)

```
red(X) | green(X) | blue(X).
:- red(X), red(Y), edge(X,Y).
:- green(X), green(Y), edge(X,Y).
:- blue(X), blue(Y), edge(X,Y).
```

Example ((part of) 8-queens problem)

```
:- not (1 = count(Y : queen(X,Y))), row(X)
```

expresses that exactly one queen appears in every row and column

Grounders and Solvers

Grounders and Solvers

Grounders

- DLV (DLV Systems, Calabria)
- Gringo (University of Potsdam)
- lparse (University of Helsinki)

Grounders and Solvers

Grounders

- DLV (DLV Systems, Calabria)
- Gringo (University of Potsdam)
- lparse (University of Helsinki)

Solvers

- clasp (University of Potsdam)
- cmodels (University of Austin)
- smodels (University of Helsinki)

Prolog and Answer Set Programming

- proof search
- Turing complete
- control
- efficiency

- model search
- finite domain
- specification language
- generality

Prolog and Answer Set Programming

- proof search
- Turing complete
- control
- efficiency

- model search
- finite domain
- specification language
- generality

Example

```
hanoi(0,_,_,_,[]).
hanoi(N,X,Y,Z,Ls) :-
        N > 0, M is N - 1,
        hanoi(M,X,Z,Y,Ls0),
        append(Ls0,[move(N,X,Z)],Ls1),
        hanoi(M,Y,X,Z,Ls2),
        append(Ls1,Ls2,Ls).
```

Example

```
disk(1..n).                                peg(a;b;c).
transition(0..pathlength-1).               situation(0..pathlength).
location(Peg) :- peg(Peg).                 location(Disk) :- disk(Disk).
#domain disk(X;Y).                  #domain peg(P;P1;P2).
#domain transition(T).              #domain situation(I).
#domain location(L;L1).

on(X,L,T+1) :- on(X,L,T), not otherloc(X,L,T+1).
otherloc(X,L,I) :- on(X,L1,I), L1!=L.
:- on(X,L,I), on(X,L1,I), L!=L1.
inpeg(X,P,I) :- on(X,L,I), inpeg(L,P,I).    inpeg(P,P,I).
top(P,L,I) :- inpeg(L,P,I), not covered(L,I).
covered(L,I) :- on(X,L,I).
:- on(X,Y,I), X>Y.
on(X,L,T+1) :- move(P1,P2,T), top(P1,X,T), top(P2,L,T).
:- move(P1,P2,T), top(P1,P1,T). movement(P1,P2) :- P1 != P2.
1 {move(A,B,T) : movement(A,B) } 1.
on(n,a,0).                                 on(X,X+1,0) :- X<n.
onewrong :- not inpeg(X,c,pathlength).
:- onewrong.
```

Thank You for Your Attention!