Constraint Logic Programming (CLP)

CSE 505 – Computing with Logic

Stony Brook University

http://www.cs.stonybrook.edu/~cse505

Constraints

- ***Constraint***: conjunction of atomic constraints
  - E.g., $4X + 3Y = 10 \land 2X - Y = 0$
  - ***Constraint Solution***: A valuation for the variables in a given constraint problem that satisfies all constraints of the problem. E.g., $X = 1 \land Y = 2$
- Why constraints?
  - Many examples of modelling can be partitioned into two parts:
    - a general description of the object or process, and
    - specific information about the situation at hand (constraints)
      - The programmer should be able to define their own problem specific constraints

(c) Paul Fodor (CS Stony Brook)

Constraint Logic Programming

- ***Constraint logic programming*** is a form of constraint programming, in which logic programming is extended to include concepts from constraint satisfaction
  - A constraint logic program is a logic program that contains ***constraints*** in the body of clauses
    - For example:

      A(X,Y):- X+Y>0, B(X), C(Y).

      - X+Y>0 is a constraint,
      - A(X,Y), B(X) and C(Y) are literals as in regular logic programming

(c) Paul Fodor (CS Stony Brook)

Constraint Logic Programming

- Why CLP?
  - "Generate-and-test" approach is a common methodology for logic programming.
    - Generate possible solutions
    - Test and eliminate non-solutions
  - Disadvantages of "generate-and-test" approach:
    - Passive use of constraints to test potential values
    - Inefficient for combinatorial search problems
  - CLP languages use the global search paradigm.
    - Actively pruning the search space
    - Recursively dividing a problem into sub-problems until its sub-problems are simple enough to be solved

(c) Paul Fodor (CS Stony Brook)

Constraint Logic Programming

- Prolog is inefficient in dealing with numerical values, due to "generate-and-test" paradigm
  - Goal of CLP is to pick numerical values from pre-defined domains for certain variables so that the given constraints on the variables are all satisfied.
  - **Idea: use CLP to define and reason with numerical constraints and assignments**
- Defines a family of programming languages
  - A language CLP(X) is defined by:
    - a constraint domain X,
    - a solver for the constraint domain X
    - a simplifier for the constraint domain X
  - For example: CLP(FD) (finite domains), CLP(R) (reals), …

(c) Paul Fodor (CS Stony Brook)

CLP(FD)

- Constraint Logic Programming over Finite Domains
  - SWI Prolog: library(clpfd)
  - XSB Prolog: library bounds
- SWI:

  :- use_module(library(clpfd)).

  - Two major use cases of this library:
    - Provide ***declarative integer arithmetic***: they implement ***pure relations*** between integer expressions and can be used in all directions, also if parts of expressions are variables.

      ```
      ?- X #> 3, X #= 5+2.
         X=7.
      ```

      In contrast, when using low-level integer arithmetic, we get:

      ```
      ?- X > 3, X is 5+2.
         Error: >/2: Arguments are not sufficiently  instantiated.
      ```


(c) Paul Fodor (CS Stony Brook)

CLP(FD)

- In connection with enumeration predicates and more complex constraints, CLP(FD) is often used to model and solve combinatorial problems such as planning, scheduling and allocation tasks.

- ***Arithmetic constraints*** are relations between arithmetic expressions

integer	Given value
variable	Unknown integer
?(*variable*)	Unknown integer
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr ^ Expr	Exponentiation
`min(Expr,Expr)`	Minimum of two expressions
`max(Expr,Expr)`	Maximum of two expressions
Expr `mod` Expr	Modulo induced by floored division
Expr `rem` Expr	Modulo induced by truncated division
`abs(Expr)`	Absolute value
Expr // Expr	Truncated integer division

Expr1 #>= Expr2	Expr1 is greater than or equal to Expr2
Expr1 #=< Expr2	Expr1 is less than or equal to Expr2
Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #> Expr2	Expr1 is greater than Expr2
Expr1 #< Expr2	Expr1 is less than Expr2

(c) Paul Fodor (CS Stony Brook)

CLP(FD)

- We can write factorial with CLP(FD):

```
n_ factorial(0, 1).
n_factorial (N,F):-
        N #> 0,
        N1 #= N-1,
        F #= N*F1,
        n_factorial(N1, F1).
?- factorial(12, Fact).
        Fact = 479001600.
```

  - We can also use it in reverse:

```
?- factorial(N, 479001600).
        N = 12.
```

  - We can find out all the possible outputs:

```
?- factorial(N, F).
        N = 0,
        F = 1 ;
        N = 1,
        F = 1 ; …
```


(c) Paul Fodor (CS Stony Brook)

CLP(FD)

- Domains:
  - Each CLP(FD) variable has an associated set of admissible integers which we call the variable's domain.
  - Initially, the domain of each CLP(FD) variable is the set of all integers.
  - The constraints in/2 and ins/2 are the primary means to specify tighter domains of variables.

```
?- X #>3.
        X in 4..sup
?- [X,Y ,Z] ins 0..3.
        X in 0..3,
        Y in 0..3,
        Z in 0..3
```


(c) Paul Fodor (CS Stony Brook)

Example: Send More Money

- Crypto-arithmetic Puzzle
  - Replace distinct letters by distinct digits, numbers have no leading zeros.
  - The variables are the letters S, E, N, D, M, O, R and Y.
  - Each letter represents a digit between 0 and 9.
  - Assign a value to each digit, such that SEND + MORE equals MONEY.

$$
\begin{array}{cccccc}
 & & S & E & N & D \\
+ & & M & O & R & E \\
\hline
= & M & O & N & E & Y
\end{array}
$$

(c) Paul Fodor (CS Stony Brook)

Example: Send More Money

- Crypto-arithmetic Puzzle

$$
\begin{array}{cccccc}
 & & S & E & N & D \\
 & & 9 & 5 & 6 & 7 \\
 & & M & O & R & E \\
+ & & 1 & 0 & 8 & 5 \\
\hline
 & M & O & N & E & Y \\
= & 1 & 0 & 6 & 5 & 2
\end{array}
$$

(c) Paul Fodor (CS Stony Brook)

Example: Send More Money

```
:- use_module(library(clpfd)).

send([S,E,N,D,M,O,R,Y]) :-
    gen_domains([S,E,N,D,M,O,R,Y],0..9),
    S #\= 0,
    M #\= 0,
    all_distinct([S,E,N,D,M,O,R,Y]),
    1000*S + 100*E + 10*N + D+ 1000*M
         + 100*O + 10*R + E #= 10000*M
         + 1000*O + 100*N + 10*E + Y,
    labeling([],[S,E,N,D,M,O,R,Y]).
gen_domains([],_).
gen_domains([H|T],D) :-
    H in D,
    gen_domains(T,D).
```


(c) Paul Fodor (CS Stony Brook)

Labeling

- Labeling procedure or enumeration procedure: try possible values for a variable $X=v\_1 \lor \ldots \lor X=v\_n$
- labeling(+Options, +Vars): assign a value to each variable in *Vars*.
  - labeling procedure will use heuristics to choose the next variable and value for labeling
  - variable ordering: chosen sequence of variables
  - first-fail principle: choose the most constrained variable first; will often lead to failure quickly, thus pruning the search tree early
  - value ordering: next value for labeling a variable must be chosen

(c) Paul Fodor (CS Stony Brook)

Labeling

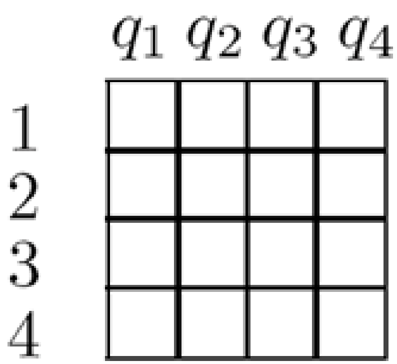
- labeling(+Options, +Vars): *Options* is a list of options that let you exhibit some control over the search process.
  - leftmost = Label the variables in the order they occur in Vars. This is the default.
  - ff = First fail. Label the leftmost variable with smallest domain next, in order to detect infeasibility early. This is often a good strategy.
  - ffc = Of the variables with smallest domains, the leftmost one participating in most constraints is labeled next.
  - min = Label the leftmost variable whose lower bound is the lowest next.
  - max = Label the leftmost variable whose upper bound is the highest next.

    ?- [X,Y] ins 10..20, labeling([max(X),min(Y)],[X,Y]).

    - generates solutions in descending order of X, and for each binding of X, solutions are generated in ascending order of Y.
    - To obtain the incomplete behaviour that other systems exhibit with "maximize(Expr)" and "minimize(Expr)", use once/1, e.g.: once(labeling([max(Expr)], Vars))

Example: n-Queens

- Place n queens q1, . . . , qn on an n x n chess board, such that they do not attack each other.



$$q_1, \ldots, q_n \in \{1, \ldots, n\}$$

$$\forall\ i \neq j.\ q_i \neq q_j \wedge |q_i - q_j| \neq |i - j|$$

- No two queens are in the same row, column and diagonal
  - each row and each column has exactly one queen
  - each diagonal has at most one queen
- qi: row position of the queen i in the i-th column

(c) Paul Fodor (CS Stony Brook)

Example: n-Queens

```
:- use_module(library(clpfd)).

n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,
        safe_queens(Qs).
safe_queens([]).
safe_queens([Q|Qs]) :-
      safe_queens(Qs, Q, 1),
      safe_queens(Qs).
safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
        Q0 #\= Q,
        abs(Q0 - Q) #\= D0,
        D1 #= D0 + 1,
        safe_queens(Qs, Q0, D1).

?- N = 8, n_queens(N, Qs), labeling([ff], Qs).
      Qs = [1, 5, 8, 6, 3, 7, 2, 4] .
```


(c) Paul Fodor (CS Stony Brook)

CLP(R)

- This library provides Constraint Logic Programming over real numbers.
  - Elements are trees containing real constants with operator in $\{=, \neq, <, \leq, >, \geq\}$.
  - SWI Prolog:

```
:- use_module(library(clpr))
```

  - Example:

```
:- use_module(library(clpr)).
p(X,Y) :-
        {X = Y * 3},
        q(X,Y).
q(X,Y) :-
        {X - 2 = Y}.
```

  - constraints are marked with {…}.

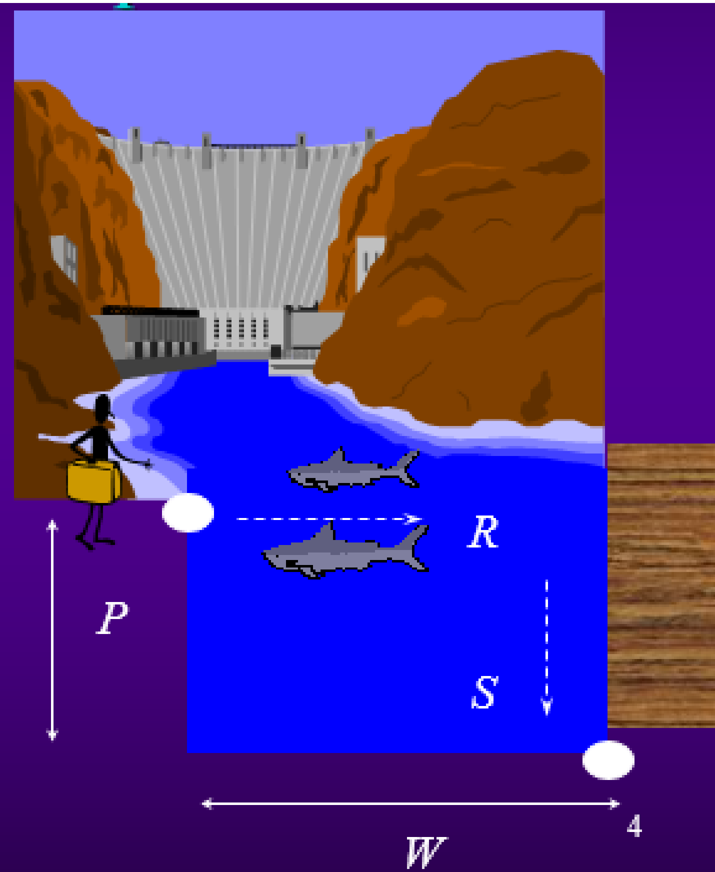
(c) Paul Fodor (CS Stony Brook)

CLP(R)

- Example:

A traveller wishes to cross a shark infested river as quickly as possible. Reasoning the fastest route is to row straight across and drift downstream, where should she set off

width of river: *W*
speed of river: *S*
set of position: *P*
rowing speed: *R*



CLP(R)

- Example:

```
:- use_module(library(clpr))
river(W, S, R, P):-
    {T = W/R},
    {P= S*T}.
```

  - Suppose she rows at 1.5m/s, river speed is 1m/s and width is 24m.

```
?- river(24, 1, 1.5, P).
```

    - Has unique answer P = 16.

(c) Paul Fodor (CS Stony Brook)

More Constraint Handling

- Constraint Simplification
- Optimization
- Implication and Equivalence

(c) Paul Fodor (CS Stony Brook)

More Constraint Handling

- Constraint Simplification
  - Two equivalent constraints represent the same information, but one may be simpler than the other

$$
\begin{aligned}
& X \geq 1 \wedge X \geq 3 \wedge 2 = Y + X \\
& \quad \leftrightarrow X \geq 3 \wedge 2 = Y + X \\
& \quad \leftrightarrow 3 \leq X \wedge X = 2 - Y \\
& \quad \leftrightarrow X = 2 - Y \wedge 3 \leq X \\
& \quad \leftrightarrow X = 2 - Y \wedge 3 \leq 2 - Y \\
& \quad \leftrightarrow X = 2 - Y \wedge Y \leq -1
\end{aligned}
$$

Removing redundant constraints, rewriting a primitive constraint, changing order, substituting using an equation all preserve equivalence

(c) Paul Fodor (CS Stony Brook)

Redundant Constraints

- One constraint C1 implies another C2 if the solutions of C1 are a subset of those of C2
  - C2 is said to be redundant wrt C1
    - It is written C1 → C2
    - For example:

$$X \geq 3 \rightarrow X \geq 1$$

$$Y \leq X + 2 \wedge Y \geq 4 \rightarrow X \geq 1$$

$$cons(X, X) = cons(Z, nil) \rightarrow Z = nil$$

(c) Paul Fodor (CS Stony Brook)

Solved Form Solvers

- Since a solved form solver creates equivalent constraints it can be a simplifier

$$cons(X, X) = cons(Z, nil) \land Y = succ(X) \land succ(Z) = Y \land Z = nil$$
$$\leftrightarrow X = nil \land Z = nil \land Y = succ(nil)$$

- Gaussian elimination:

$$X = 2 + Y \land 2Y + X - T = Z \land X + Y = 4 \land Z + T = 5$$
$$\leftrightarrow X = 3 \land Y = 1 \land Z = 5 - T$$

(c) Paul Fodor (CS Stony Brook)

Optimization

- Often given some problem which is modelled by constraints we don't want just any solution, but a "**best**" solution
  - This is an optimization problem
  - We need an ***objective function*** so that we can ***rank*** solutions, that is a mapping from solutions to a real value
    - An ***optimization problem*** (C,f) consists of a constraint C and objective function f
    - A valuation v1 is ***preferred*** to valuation v2 if f(v1) < f(v2)
    - An ***optimal solution*** is a solution of C such that no other solution of C is preferred to it.

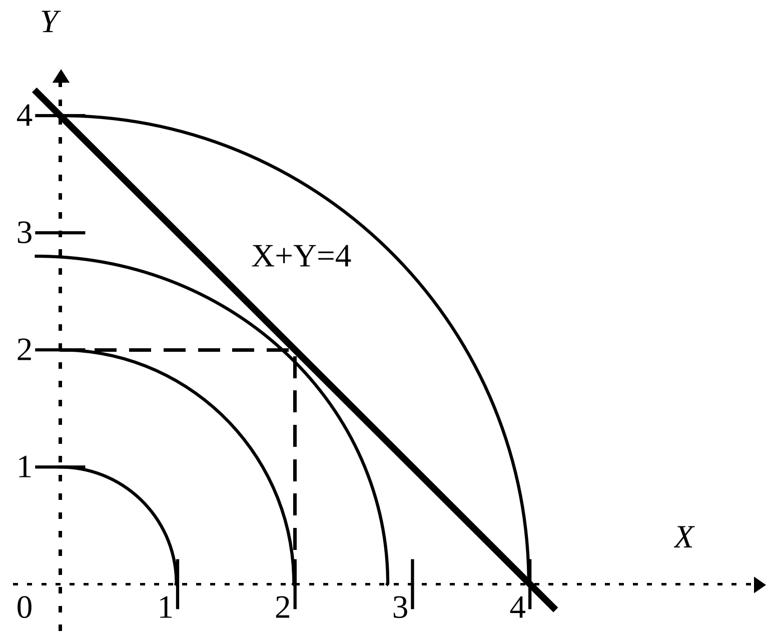
(c) Paul Fodor (CS Stony Brook)

Optimization Example

$(C \equiv X + Y \geq 4, \quad f \equiv X^2 + Y^2)$

- Find the closest point to the origin satisfying the C.
- Some solutions and f value

$\{X \mapsto 0, Y \mapsto 4\}$	16
$\{X \mapsto 3, Y \mapsto 3\}$	18
$\{X \mapsto 2, Y \mapsto 2\}$	8



- Optimal solution:

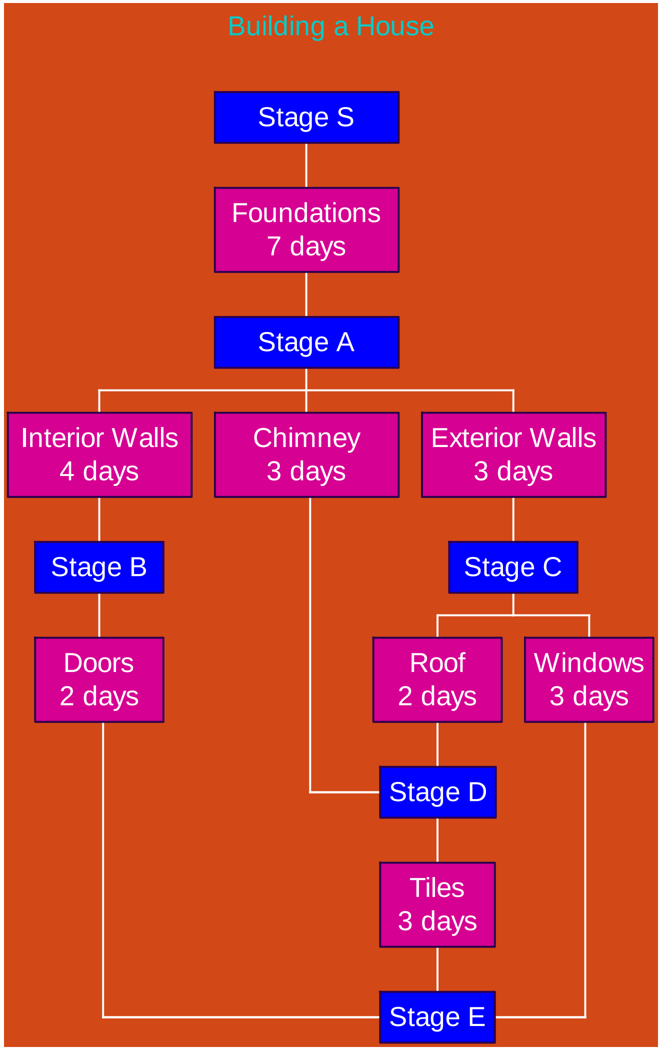
$\{X \mapsto 2, Y \mapsto 2\}$

(c) Paul Fodor (CS Stony Brook)

Implication and Equivalence

- Other important operations involving constraints are:
  - **implication**: test if C1 implies C2
    - impl(C1, C2) answers true, false or unknown
  - **equivalence**: test if C1 and C2 are equivalent
    - equiv(C1, C2) answers true, false or unknown

Implication Example



- For the house constraints *CH*, will stage B have to be reached after stage C?

$$CH \rightarrow T_B \geq T_C$$

- For this question the answer if *false*, but if we require the house to be finished in 15 days the answer is *true*

$$CH \wedge T_E = 15 \rightarrow T_B \geq T_C$$

(c) Paul Fodor (CS Stony Brook)

Application Domains

- Modeling
- Executable Specifications
- Solving combinatorial problems
  - Scheduling, Planning, Timetabling
  - Configuration, Layout, Placement, Design
  - Analysis: Simulation, Verification, Diagnosis of software, hardware and industrial processes.
- Artificial Intelligence
  - Machine Vision
  - Natural Language Understanding
  - Qualitative Reasoning, etc.

Applications in Research

- Computer Science: Program Analysis, Robotics, Agents
- Molecular Biology, Biochemistry, Bio-informatics: Protein Folding, Genomic Sequencing
- Economics: Scheduling
- Linguistics: Parsing
- Medicine: Diagnosis Support
- Physics: System Modeling
- Geography: Geo-Information-Systems

(c) Paul Fodor (CS Stony Brook)