Computational Logic

Constraint Logic Programming
Constraints

- Constraint: conditions that a solution must satisfy
  - $X + Y = 20$
  - $X \land Y$ is true
  - The third field of the data structure is greater than the second
  - The murderer is one of those who had met the cabaret entertainer

- CLP: LP plus the ability to compute with some form of constraints
  (which are solved by the system during computation)

- Features of a CLP system:
  - Domain of computation (reals, rationals, integers, booleans, structures, ...)
  - *Expressions* that can be built ($+, *, \land, \lor$)
  - *Constraints* allowed: equations, disequations, inequations, etc.
    ($=, \neq, \leq, \geq, <, >$)
  - *Constraint solving algorithms*: simplex, gauss, etc.

- Solutions: assignments to variables, or new constraints among variables.
A comparison with classic LP (I)

- Example (plain Prolog): \( q(X, Y, Z) :- Z = f(X, Y) \).

\[
\begin{align*}
? &- q(3, 4, Z). \\
Z & = f(3,4)
\end{align*}
\]

\[
\begin{align*}
? &- q(X, Y, f(3,4)). \\
X & = 3, Y = 4
\end{align*}
\]

\[
\begin{align*}
? &- q(X, Y, Z). \\
Z & = f(X,Y)
\end{align*}
\]

- Example (plain Prolog): \( p(X, Y, Z) :- Z \text{ is } X + Y \).

\[
\begin{align*}
? &- p(3, 4, Z). \\
Z & = 7
\end{align*}
\]

\[
\begin{align*}
? &- p(X, 4, 7). \\
\{\text{INSTANTIATION ERROR}\} & \leftarrow \text{is/2 not reversible, does not work!}
\end{align*}
\]
A Comparison with classic LP (II)

• Example (*CLP(R)* package):

```
:- use_package(clpr).
p(X, Y, Z) :- Z =. X + Y.

?- p(3, 4, Z).
Z =. 7

?- p(X, 4, 7).
X =. 3

4 ?- p(X, Y, 7).
X =. 7 - Y ← with clpr arithmetic is reversible!
```
A Comparison with classic LP (III)

- Advantages:
  - Helps making programs expressive and flexible.
  - May save much coding.
  - In some cases, more efficient than classic LP programs due to solvers typically being very efficiently implemented.
  - Also, efficiency due to search space reduction:
    - LP: generate-and-test.
    - CLP: constrain-and-generate.

- Disadvantages:
  - Complexity of solver algorithms (simplex, gauss, etc) can affect performance.

- Solutions:
  - better algorithms
  - compile-time optimizations (program transformation, global analysis, etc)
  - parallelism
Example of Search Space Reduction

• Using **plain Prolog** (generate–and–test):

```prolog
% Find three consecutive numbers in the p/1 relation.
solution(X, Y, Z) :-
    p(X), p(Y), p(Z),
    test(X, Y, Z).
```

```prolog
```

```prolog
test(X, Y, Z) :- Y is X + 1, Z is Y + 1.
```

• Query:

```prolog
?- solution(X, Y, Z).
X = 14, Y = 15, Z = 16 ? ;
no
```

• 458 steps (all solutions: 475 steps).
Example of Search Space Reduction

- Using the **CLP(R) package** (generate–and–test):

  ```prolog
  % Find three consecutive numbers in the p/1 relation.
  :- use_package(clpr).
  solution(X, Y, Z) :-
      p(X), p(Y), p(Z),
      test(X, Y, Z).
  test(X, Y, Z) :- Y =. X + 1, Z =. Y + 1.
  ```

  Query:
  ```prolog
  ?- solution(X, Y, Z).
  X =. 14, Y =. 15, Z =. 16 ;
  no
  ```

- 458 steps (all solutions: 475 steps).
Generate–and–test Search Tree
Example of Search Space Reduction

- **Move** test(X, Y, Z) **to the beginning** (constrain–and–generate):

  % Find three consecutive numbers in the p/1 relation.
  :- use_package(clpr).
  solution(X, Y, Z) :-
      test(X, Y, Z),
      p(X), p(Y), p(Z).

- **Using plain Prolog:**

  test(X, Y, Z) :- Y is X + 1, Z is Y + 1.

  ?- solution(X, Y, Z).
  {INSTANTIATION ERROR}

- **Using the CLP(ℜ) package:**

  test(X, Y, Z) :- Y =. X + 1, Z =. Y + 1.

  ?- solution(X, Y, Z).
  X =. 14, Y =. 15, Z =. 16 ? ;
  no

  In **11 steps** (and all solutions in 11 steps)!
Constrain–and–generate Search Tree

```
g
X=11    X=3    X=7    X=16    X=15    X=14

Y=16    Y=15

Z=16
```
The semantics is parameterized by the *constraint domain* $\mathcal{X}$: $\text{CLP}(\mathcal{X})$, where $\mathcal{X} \equiv (\Sigma, D, L, T)$:

- $\Sigma$: set of *predicate* and *function symbols*, together with their arity
- $L \subseteq \Sigma$—formulae: constraints
- $D$: the set of actual elements in the constraint domain
- $D$: meaning of predicate and function symbols (and hence, constraints).
- $T$: a first–order theory (axiomatizes some properties of $D$)

- $(D, L)$ is a *constraint domain*

**Assumptions:**

- $L$ built upon a first–order language
- $\in \subseteq \Sigma$ and $\in$ is *identity* in $D$
- There are identically false and identically true constraints in $L$
- $L$ is closed w.r.t. renaming, conjunction, and existential quantification
Constraint Domains (I)

- \( \Sigma = \{0, 1, +, *, =, <, \leq\} \), \( D = \mathbb{R} \) (the reals), \( D \) interprets \( \Sigma \) as usual, \( \mathcal{R} = (D, \mathcal{L}) \)

  \( \rightarrow \) Arithmetic over the reals ("\( \mathcal{R} \)" domain).

  - Eg.: \( x^2 + 2xy < \frac{y}{x} \land x > 0 \) (\( \equiv xxx + xxy + xxy < y \land 0 < x \))
  - Question: is 0 needed? How can it be represented?

- \( \Sigma' = \{0, 1, +, =, <, \leq\} \), \( \mathcal{R}_{Lin} = (D', \mathcal{L}') \)

  \( \rightarrow \) Linear arithmetic ("\( \mathcal{R}_{Lin} \)" domain)

  - Eg.: \( 3x - y < 3 \) (\( \equiv x + x + x < 1 + 1 + 1 + y \))

- \( \Sigma'' = \{0, 1, +, =\} \), \( \mathcal{R}_{LinEq} = (D'', \mathcal{L}'') \)

  \( \rightarrow \) Linear equations ("\( \mathcal{R}_{LinEq} \)" domain)

  - Eg.: \( 3x + y = 5 \land y = 2x \)

- A corresponding set of domains can be defined on the rationals ("\( \mathbb{Q} \)" domain)
Constraint Domains (II)

- A very special domain:
  - $\Sigma = \{ \langle \text{constant and function symbols} \rangle, = \}$
  - $D = \{ \text{finite trees} \}$
  - $D$ interprets $\Sigma$ as tree constructors
    * Each $f \in \Sigma$ with arity $n$ maps $n$ trees to a tree with root labeled $f$ and whose subtrees are the arguments of the mapping
  - Constraints: syntactic tree equality
  - $\mathcal{FT} = (D, L)$

$\Rightarrow$ **Equality constraints over the Herbrand domain** ($\mathcal{FT}$ domain)
- Eg.: $g(h(Z), Y) = g(Y, h(a))$

- $LP \equiv CLP(\mathcal{FT})$
  - I.e., classical LP can be viewed as constraint logic programming over *Herbrand terms* with a single *constraint predicate symbol*: $\equiv$. 

13
Constraint Domains (III)

- $\Sigma = \{<\text{constants}>, \lambda, \ldots, \_, =\}$
- $D = \{\text{finite strings of constants}\}$
- $D$ interprets $\_$ as string concatenation, $\_\_$ as string length
  - $\rightarrow$ Equations over strings of constants ($D$ domain)
    - Eg.: $X.A.X = X.A$

- $\Sigma = \{0, 1, \neg, \land, =\}$
- $D = \{\text{true}, \text{false}\}$
- $D$ interprets symbols in $\Sigma$ as boolean functions
- $BOOL = (D, L)$
  - $\rightarrow$ Boolean constraints ($BOOL$ domain)
    - Eg.: $\neg(x \land y) = 1$
CLP(\mathcal{X}) Programs

- Recall that:
  - $\Sigma$ is a set of predicate and function symbols
  - $\mathcal{L} \subseteq \Sigma$–formulae are the constraints

- $\Pi \subseteq \Sigma$: set of predicate symbols definable by a program
  - Atom: $p(t_1, t_2, \ldots, t_n)$, where $p \in \Pi$ and $t_1, t_2, \ldots, t_n$ are terms
  - Primitive constraint: $p(t_1, t_2, \ldots, t_n)$, where
    $t_1, t_2, \ldots, t_n$ are terms and $p \in \Sigma$ is a predicate symbol
  - Constraint: (first–order) formula built from primitive constraints

- The class of constraints will vary (generally only a subset of formulas are considered constraints)

- A **CLP program** is a collection of rules of the form $a \leftarrow b_1, \ldots, b_n$ where $a$ is an atom and the $b_i$’s are atoms or constraints

- A fact is a rule $a \leftarrow c$ where $c$ is a constraint

- A goal (or query) $G$ is a conjunction of constraints and atoms
A case study: CLP(ℜ)

- CLP(ℜ): language based on Prolog + constraint solving over the reals (ℜLin)
  - Same execution strategy as standard Prolog (depth-first, left-to-right)
  - Allows linear equations and disequations over the reals
  - Linear constraints are solved; non-linear constraints are passive: delayed until linear or simple checks:
    * $X \times Y = 7$ becomes linear when $X$ is assigned a definite value
    * $X \times X + 2 \times X + 1 = 0$ becomes a check when $X$ is assigned a definite value
  - Prolog arithmetic is subsumed by constraint solving

- Note: CLP(ℜ) is really CLP((ℜ, ℱT)) — ℱT is often omitted.

- Supported in modern Prologs coexisting with the ISO primitives \texttt{is/2, >/2} etc.

- In Ciao, via the \texttt{clpr} package:
  - Uses \texttt{.=., .>.}, etc. to distinguish the \texttt{clpr} constraints from the ISO-Prolog arithmetic primitives.
  - I.e., $X \ .= . \ Y + 5$, $Y \ .> . \ 1$ vs. $X \ is \ Y + 5$, $Y \ > \ 1$
Linear Equations (CLP(ℜ) package)

- Vector × vector multiplication (dot product):
  \[ \cdot : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R} \]
  \[ (x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) = x_1 \cdot y_1 + \cdots + x_n \cdot y_n \]

- Vectors represented as lists of numbers

```prolog
:- use_package(clpr).
prod([], [], Result) :- Result =. 0.
prod([X|Xs], [Y|Ys], Result) :-
    Result =. X * Y + Rest, prod(Xs, Ys, Rest).
```

- Unification becomes constraint solving!

```prolog
?- prod([2, 3], [4, 5], K).
K =. 23
?- prod([2, 3], [5, X2], 22).
X2 =. 4
?- prod([2, 7, 3], [Vx, Vy, Vz], 0).
Vx =. -1.5*Vz - 3.5*Vy
```

- Any computed answer is, in general, an equation over the variables in the query
Systems of Linear Equations (CLP(R))

- Can we solve systems of equations? E.g.,

\[3x + y = 5\]
\[x + 8y = 3\]

- Write them down at the top level prompt:

```prolog
?- prod([3, 1], [X, Y], 5), prod([1, 8], [X, Y], 3).
X = 1.6087, Y = 0.173913
```

- A more general predicate can be built mimicking the mathematical vector notation \(A \cdot x = b\):

```prolog
system(_Vars, [], []).
system(Vars, [Co|Coefs], [Ind|Indeps]) :-
    prod(Vars, Co, Ind),
    system(Vars, Coefs, Indeps).
```

- We can now express (and solve) equation systems

```prolog
?- system([X, Y], [[3, 1], [1, 8]], [5, 3]).
X = 1.6087, Y = 0.173913
```
Non–linear Equations (CLP(ℜ))

- Non–linear equations are delayed

\[
\begin{align*}
\text{?- } \sin(X) & \text{ .} \ = \ \cos(X).
\sin(X) & \text{ .} \ = \ \cos(X)
\end{align*}
\]

- This is also the case if there exists some procedure to solve them

\[
\begin{align*}
\text{?- } X^2 + 2X + 1 & \text{ .} \ = \ 0.
-2X - 1 & \text{ .} \ = \ X^2
\end{align*}
\]

- Reason: no general solving technique is known. CLP(ℜ) solves only linear (dis)equations.

- Once equations become linear, they are handled properly:

\[
\begin{align*}
\text{?- } X & \text{ .} \ = \ \cos(\sin(Y)), \ Y & \text{ .} \ = \ 2+Y^3.
Y & \text{ .} \ = \ -1, \ X & \text{ .} \ = \ 0.666367
\end{align*}
\]

- Disequations are solved using a modified, incremental Simplex

\[
\begin{align*}
\text{?- } X + Y & \text{ .} \ < \ = \ 4, \ Y & \text{ .} \ > \ = \ 4, \ X & \text{ .} \ > \ = \ 0.
Y & \text{ .} \ = \ 4, \ X & \text{ .} \ = \ 0
\end{align*}
\]
Fibonacci Revisited (Prolog)

- Fibonacci numbers:

\[
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_{n+2} &= F_{n+1} + F_n
\end{align*}
\]

- (The good old) Prolog version:

```prolog
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

- Can only be used with the first argument instantiated to a number
Fibonacci Revisited (CLP(ℜ))

- CLP(ℜ) package version: syntactically similar to the previous one:

```prolog
:- use_package(clpr).
fib(N,N) :- N =. 0.
fib(N,N) :- N =. 1.
fib(N,R) :- N >. 1, F1 >=. 0, F2 >=. 0,
           N1 =. N - 1, N2 =. N - 2,
           fib(N1,F1), fib(N2,F2),
           R =. F1 + F2.
```

- Note all constraints included in program (F1 >=0, F2 >=0) – good practice!
- Only real numbers and equations used (no data structures, no other constraint system): “pure CLP(ℜ)”
- Semantics greatly enhanced! E.g.:

```
?- fib(N, F).
F =. 0, N =. 0 ;
F =. 1, N =. 1 ;
F =. 1, N =. 2 ;
F =. 2, N =. 3 ;
```
Analog RLC circuits (CLP(ℜ))

- Analysis and synthesis of analog circuits
- RLC network in steady state
- Each circuit is composed either of:
  - A simple component, or
  - A connection of simpler circuits
- For simplicity, we will suppose subnetworks connected only in parallel and series
  → Ohm’s laws will suffice (other networks need global, i.e., Kirchoff’s laws)
- We want to relate the current (I), voltage (V) and frequency (W) in steady state
- Entry point: \texttt{circuit(C, V, I, W)} states that:
  across the network \( C \), the voltage is \( V \), the current is \( I \) and the frequency is \( W \)
- \( V \) and \( I \) must be modeled as complex numbers (the imaginary part takes into account the angular frequency)
- Note that Herbrand terms are used to provide data structures
Analog RLC circuits (CLP($\mathbb{R}$))

- Complex number $X + Yi$ modeled as $c(X, Y)$
- Basic operations:

```prolog
:- use_package(clpr).

c_add(c(Re1, Im1), c(Re2, Im2), c(Re12, Im12)) :-
    Re12 =. Re1 + Re2,
    Im12 =. Im1 + Im2.

c_mult(c(Re1, Im1), c(Re2, Im2), c(Re3, Im3)) :-
    Re3 =. Re1 * Re2 - Im1 * Im2,
    Im3 =. Re1 * Im2 + Re2 * Im1.
```

(equality is $c_equal(c(R, I), c(R, I))$, can be left to [extended] unification)
Analog RLC circuits (CLP(\(\mathbb{R}\)))

- Circuits in series:

```prolog
 circuit(series(N1, N2), V, I, W) :-
    c_add(V1, V2, V),
    circuit(N1, V1, I, W),
    circuit(N2, V2, I, W).
```

- Circuits in parallel:

```prolog
 circuit(parallel(N1, N2), V, I, W) :-
    c_add(I1, I2, I),
    circuit(N1, V, I1, W),
    circuit(N2, V, I2, W).
```
Analog RLC circuits (CLP(ℜ))

Each basic component can be modeled as a separate unit:

- **Resistor:** \[ V = I \times (R + 0i) \]

  \[
  \text{circuit(resistor}(R), V, I, _W) :- \\
  \text{c_mult}(I, c(R, 0), V).
  \]

- **Inductor:** \[ V = I \times (0 + WL_i) \]

  \[
  \text{circuit(inductor}(L), V, I, W) :- \\
  \text{Im} == W \times L, \\
  \text{c_mult}(I, c(0, \text{Im}), V).
  \]

- **Capacitor:** \[ V = I \times (0 - \frac{1}{WC_i}) \]

  \[
  \text{circuit(capacitor}(C), V, I, W) :- \\
  \text{Im} == -1 \times \frac{1}{(W \times C)}, \\
  \text{c_mult}(I, c(0, \text{Im}), V).
  \]
Analog RLC circuits (CLP($\mathbb{R}$))

- Example:

\[
\begin{align*}
R &= \_ \_ \\
C &= \_ \_ \\
V &= 4.5 \\
\omega &= 2400 \\
I &= 0.65 \\
L &= 0.073
\end{align*}
\]

?- circuit(parallel(inductor(0.073),
series(capacitor(C), resistor(R))),
c(4.5, 0), c(0.65, 0), 2400).

R = 6.91229, C = 0.00152546

?- circuit(C, c(4.5, 0), c(0.65, 0), 2400).
The N Queens Problem

- **Problem:**
  place \( N \) chess queens in a \( N \times N \) board such that they do not attack each other

- **Data structure:** a list holding the column position for each row

- **The final solution is a permutation of the list \([1, 2, \ldots, N]\)**

- **E.g.:** the solution is represented as \([2, 4, 1, 3]\)

- **General idea:**
  - Start with partial solution
  - Nondeterministically select new queen
  - Check safety of new queen against those already placed
  - Add new queen to partial solution if compatible; start again with new partial solution
The N Queens Problem in Prolog

```prolog
queens(N, Qs) :- queens_list(N, Ns), % E.g., Ns=[4,3,2,1]
    queens(Ns, [], Qs).

queens([], Qs, Qs). % All queens placed!
queens(Unplaced, Placed, Qs) :-
    select(Unplaced, Q, NewUnplaced), % E.g. Q=4, NewU=[3,2,1]
    no_attack(Placed, Q, 1),
    queens(NewUnplaced, [Q|Placed], Qs). % OK->Choose next q

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-
    Queen =\= Y + Nb, Queen =\= Y - Nb, Nb1 is Nb + 1,
    no_attack(Ys, Queen, Nb1).

select([X|Ys], X, Ys).
select([Y|Ys], X, [Y|Zs]) :- select(Ys, X, Zs).

queens_list(0, []).
queens_list(N, [N|Ns]) :-
    N > 0, N1 is N - 1, queens_list(N1, Ns).
```
The N Queens Problem in Prolog - search space
The N Queens Problem in CLP(\(\mathbb{R}\)) (in Ciao clpr syntax)

```prolog
:- use_package(clpr).
queens(N,Qs) :- constrain_values(N,N,Qs), place_queens(N,Qs).

constrain_values(0, _N, []).  % Constrain before placing
constrain_values(N, Range, [X|Xs]) :-
    N > 0, X > 0, X <= Range, N1 = N - 1,
    constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb).  % Identical to Prolog version
no_attack([Y|Ys], Queen, Nb) :-  % but using constraints
    Queen <> Y + Nb, Queen <> Y - Nb, Nb1 = Nb + 1,
    no_attack(Ys, Queen, Nb1).

place_queens(0, _).
place_queens(N, Q) :-
    N > 0, member(N, Q),
    N1 = N - 1,
    place_queens(N1, Q).
```
This last program can attack the problem in its most general instance:

```prolog
?- queens(N,L).
L = [], N =. 0 ;
L = [1], N =. 1 ;
L = [2, 4, 1, 3], N =. 4 ;
L = [3, 1, 4, 2], N =. 4 ;
L = [5, 2, 4, 1, 3], N =. 5 ;
L = [5, 3, 1, 4, 2], N =. 5 ;
L = [3, 5, 2, 4, 1], N =. 5 ;
L = [2, 5, 3, 1, 4], N =. 5
...
```

- Remark: Herbrand terms used to build the data structures
- But also used as constraints (e.g., length of already built list \( Xs \) in \( \text{no_attack}(Xs, X, 1) \))
- Note that in fact we are using both \( \mathbb{R} \) and \( \mathcal{FT} \)
The N Queens Problem in CLP(ℜ) – search space
The N Queens Problem in CLP(ℜ)

- CLP(ℜ) generates internally a set of equations for each board size
  
  ```prolog
  ?- constrain_values(4, 4, Qs).
  Qs = [_A, _B, _C, _D],
  nonzero(_E), _A=<.4.0, _E.=.3.0+_A-_D,
  nonzero(_F), _A>.0, _F.=.-3.0+_A-_D,
  nonzero(_G), _B=<.4.0, _G.=.2.0+_A-_C,
  nonzero(_H), _B>.0, _H.=.-2.0+_A-_C,
  nonzero(_I), _C=<.4.0, _I.=.1+_A-_B,
  nonzero(_J), _C>.0, _J.=.-1+_A-_B,
  nonzero(_K), _D=<.4.0, _K.=.2.0+_B-_D,
  nonzero(_L), _D>.0, _L.=.-2.0+_B-_D,
  nonzero(_M), _M.=.1+_B-_C,
  nonzero(_N), _N.=.-1+_B-_C,
  nonzero(_O), _O.=.1+_C-_D,
  nonzero(_P), _P.=.-1+_C-_D
  ```

The N Queens Problem in CLP(\(\mathbb{R}\))

- Constraints are (incrementally) simplified as new queens are added

```prolog
?- constrain_values(4, 4, Qs), Qs = [3,1|_].
Qs = [_A,_B,_C,_D],
nonzero(_E),  _A.=.3.0,  _E.=.6.0--D,
nonzero(_F),  _B.=.1.0,  _F.=.--D,
nonzero(_G),  _C.=<.4.0,  _G.=.5.0--C,
nonzero(_H),  _C.+>.0,  _H.=.1.0--C,
nonzero(_I),  _D.=<.4.0,  _I.=.3.0--D,
nonzero(_J),  _D.+>.0,  _J.=.--1.0--D,
nonzero(_K),  _K.=.2.0--C,
nonzero(_L),  _L.=.--C,
nonzero(_M),  _M.=.1+-C--D,
nonzero(_N),  _N.=.--1+-C--D ?
```

- Bad choices are rejected using constraint consistency:

```prolog
?- constrain_values(4, 4, Qs), Qs = [3,2|_].
no
```
Finite Domains (I)

- A *finite domain* constraint solver associates each variable with a finite subset of \( \mathbb{Z} \)

- Example: \( E \in \{-123, -10..4, 10\} \)
  
  Can be represented as, e.g., \[ E :: [-123, -10..4, 10] \] [Eclipse notation]
  
or as \[ E \in -123 \lor (-10..4) \lor 10 \] [Ciao notation]

- We can:
  
  ◦ Establish the *domain* of a variable (\texttt{in}).
  
  ◦ Perform arithmetic operations (\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}) on the variables
  
  ◦ Establish linear relationships among arithmetic expressions (\texttt{#=}, \texttt{#<}, \texttt{#=<})

- These operations / relationships narrow the domains of the variables

- **Note:** In Ciao this functionality is loaded with a

  \[ :- \texttt{use}_{\text{package}}(\text{clpfd}). \]

  directive in the source code –or, equivalently, adding in the module declaration:

  \[ :- \texttt{module}(_\ldots, [\text{clpfd}]). \]
Finite Domains (II)

Examples:

?- X #= A + B, A in 1..3, B in 3..7.
X in 4..10, A in 1..3, B in 3..7

- The respective minimums and maximums are added
- There is no unique solution

?- X #= A - B, A in 1..3, B in 3..7.
X in -6..0, A in 1..3, B in 3..7

- The min value of X is the min value of A minus the max value of B
- (Similar for the maximum values)

?- X #= A - B, A in 1..3, B in 3..7, X #>= 0.
   A = 3, B = 3, X = 0

- Putting more constraints results in a unique solution.
Finite Domains (III)

Some useful primitives in finite domains:

- **domain(Variables, Min, Max)**: A shorthand for several in constraints

- **labeling(Options, VarList)**:
  
  - instantiates variables in \( \text{VarList} \) to values in their domains
  - \( \text{Options} \) dictates the search order

```prolog
?- domain([X, Y, Z],1,1000), X*X+Y*Y #= Z*Z, X #>= Y, labeling([], [X,Y,Z]).
X = 4, Y = 3, Z = 5,
X = 8, Y = 6, Z = 10,
X = 12, Y = 5, Z = 13,
...
```

- **minimize(G, X)**: solve \( G \) minimizing the value of variable \( X \)

- This can be used to minimize (c.f., maximize) a solution
A classic example: SEND MORE MONEY

% SEND
% + MORE
% _________
% MONEY

:- use_package(clpfd).

smm([S,E,N,D,M,O,R,Y]) :-
  domain([S,E,N,D,M,O,R,Y], 0, 9), % All digits 0..9
  0 #< S, 0 #< M, % No leftmost zeros
  all_different([S,E,N,D,M,O,R,Y]), % All digits different
  S*1000 + E*100 + N*10 + D + %
  M*1000 + O*100 + R*10 + E #= % Arith. constr.
  M*10000 + O*1000 + N*100 + E*10 + Y, %
  labeling([], [S,E,N,D,M,O,R,Y]). % Instantiate variables
A Project Management Problem (I)

- The job whose dependencies and task lengths are given by this graph...

... should be finished in 10 time units or less.

- Constraints:

\[
\text{pn1}(A,B,C,D,E,F,G) :\neg \\
\text{domain}([A,B,C,D,E,F,G], 0, 10), \\
A \#>= 0, G \#=< 10, \\
B \#>= A, C \#>= A, D \#>= A, \\
E \#>= B + 1, E \#>= C + 2, \\
F \#>= C + 2, F \#>= D + 3, \\
G \#>= E + 4, G \#>= F + 1.
\]
A Project Management Problem (II)

• Query:

```prolog
?- pn1(A, B, C, D, E, F, G).
A in 0..4, B in 0..5, C in 0..4,
D in 0..6, E in 2..6, F in 3..9, G in 6..10.
```

• Note the slack of the variables

• Some additional constraints must be respected as well, but are not shown by default

• Minimize the total project time:

```prolog
?- minimize(pn1(A, B, C, D, E, F, G), G).
A = 0, B in 0..1, C = 0, D in 0..2,
E = 2, F in 3..5, G = 6
```

• Variables without slack represent critical tasks
A Project Management Problem (III)

- An alternative setting:

- We can accelerate task F at some cost

\[
\text{pn2} (A, B, C, D, E, F, G, X) :-
\text{domain}([A,B,C,D,E,F,G,X], 0, 10),
A \#>= 0, G \#=< 10,
B \#>= A, C \#>= A, D \#>= A,
E \#>= B + 1, E \#>= C + 2,
F \#>= C + 2, F \#>= D + 3,
G \#>= E + 4, G \#>= F + X.
\]

- We do not want to accelerate it more than needed!

\[\rightarrow \text{minimize } G \text{ and maximize } X.\]

A = 0, B in 0..1, C = 0, D = 0,
E = 2, F = 3, G = 6, X = 3.
A Project Management Problem (IV)

- We have two independent tasks \( B \) and \( D \) whose lengths are not fixed:

- We can finish any of \( B, D \) in 2 time units at best
- Some shared resource disallows finishing *both* tasks in 2 time units: they will take 6 time units
A Project Management Problem (V)

- Constraints describing the net:

\[
\text{pn3}(A,B,C,D,E,F,G,X,Y) : - \\
\quad \text{domain([A,B,C,D,E,F,G,X,Y], 0, 10),} \\
\quad A \#>= 0, G \#=< 10, \\
\quad X \#>= 2, Y \#>= 2, X + Y \#= 6, \\
\quad B \#>= A, C \#>= A, D \#>= A, \\
\quad E \#>= B + X, E \#>= C + 2, \\
\quad F \#>= C + 2, F \#>= D + Y, \\
\quad G \#>= E + 4, G \#>= F + 1.
\]

- Query:

\[
? - \text{minimize(pn3(A,B,C,D,E,F,G,X,Y),G).}
\]

\[
A = 0, B = 0, C = 0, D \text{ in } 0..1, E = 2, \\
F \text{ in } 4..5, X = 2, Y = 4, G = 6
\]

- I.e., we must devote more resources to task B

- All tasks but F and D are critical now

- Sometimes, \text{minimize/2} not enough to provide best solution (pending constr.):

\[
? - \text{minimize(pn3(A,B,C,D,E,F,G,X,Y),G), labeling([], [D,F]).}
\]
The N-Queens Problem Using Finite Domains (in Ciao clpfd syntax)

- By far, the fastest implementation

```prolog
:- use_package(clpfd).
queens(N, Qs, Type) :- % Type is labeling strategy
  constrain_values(N, N, Qs), % Constrain before placing
  all_different(Qs), % Using built-in constraint
  labeling(Type, Qs). % Labeling places the queens

constrain_values(0, _N, []).
constrain_values(N, Range, [X|Xs]) :-
  N > 0, N1 is N - 1, X in 1 .. Range, % Limits X values
  constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb). % Same as CLP(R) version
no_attack([Y|Ys], Queen, Nb) :- % but using clpfd primitives
  Queen #!= Y + Nb, Queen #!= Y - Nb, Nb1 is Nb + 1,
  no_attack(Ys, Queen, Nb1).
```

- Query: `?- queens(20, Q, [ff]).` (Type is the type of labeling desired.)

  Q = [1,3,5,14,17,4,16,7,12,18,15,19,6,10,20,11,8,2,13,9]
CLP($\mathcal{F}T$) (a.k.a. Logic Programming)

- Equations over Finite Trees

- Check that two trees are isomorphic (same elements in each level)

```prolog
iso(Tree, Tree).
iso(t(R, I1, D1), t(R, I2, D2)) :-
    iso(I1, D2),
    iso(D1, I2).

?- iso(t(a, b, t(X, Y, Z)), t(a, t(u, v, W), L)).
L=b, X=u, Y=v, Z=W ? ;
L=b, X=u, Y=W, Z=v ? ;
L=b, W=t(_C,_B,_A), X=u, Y=t(_C,_A,_B), Z=v ? ;
L=b, W=t(_E,t(_D,_C,_B),_A), X=u, Y=t(_E,_A,t(_D,_B,_C)), Z=v ?
```

CLP(\mathcal{W}\mathcal{E})

- Equations over finite strings
- Primitive constraints: concatenation (.), string length (::)
- Find strings meeting some property:

  |- "123".Z = Z."231", Z::0.

    no

  |- "123".Z = Z."231", Z::1.

    Z = "1"

  |- "123".Z = Z."231", Z::2.

    no

  |- "123".Z = Z."231", Z::3.

    no

  |- "123".Z = Z."231", Z::4.

    Z = "1231"

- These constraint solvers are very complex
- Often incomplete algorithms are used
CLP((\mathcal{WE}, \mathcal{Q}))

- Word equations plus arithmetic over \(\mathcal{Q}\) (rational numbers)
- Prove that the sequence \(x_{i+2} = |x_{i+1}| - x_i\) has a period of length 9 (for any starting \(x_0, x_1\))
- Strategy: describe the sequence, try to find a subsequence such that the period condition is violated
- Sequence description (syntax is Prolog III slightly modified):

\[
\begin{align*}
\text{seq}(\langle Y, X \rangle). & \quad \text{abs}(Y, Y) :- Y \geq 0. \\
\text{seq}(\langle Y1 - X, Y, X \rangle. U) :- \quad \text{abs}(Y, -Y) :- Y < 0. \\
& \quad \text{seq}(\langle Y, X \rangle. U) \quad \text{abs}(Y, Y1).
\end{align*}
\]

- Query: Is there any 11–element sequence such that the 2–tuple initial seed is different from the 2–tuple final subsequence (the seed of the rest of the sequence)?

```prolog
?- seq(U.V.W), U::2, V::7, W::2, U#W.
fail
```
Summarizing

- **In general:**
  - Data structures (Herbrand terms) for free
  - Each logical variable may have constraints associated with it (and with other variables)

- **Problem modeling:**
  - Rules represent the problem at a high level
    - Program structure, modularity
    - Recursion used to set up constraints
  - Constraints encode problem conditions
  - Solutions also expressed as constraints

- **Combinatorial search problems:**
  - CLP languages provide backtracking: enumeration is easy
  - Constraints keep the search space manageable

- **Tackling a problem:**
  - Keep an open mind: often new approaches possible
Complex Constraints

• Some complex constraints allow expressing simpler constraints
• May be operationally treated as passive constraints
• E.g.: cardinality operator \( \#(L, [c_1, \ldots, c_n], U) \) meaning that the number of true constraints lies between \( L \) and \( U \) (which can be variables themselves)
  ◦ If \( L = U = n \), all constraints must hold
  ◦ If \( L = U = 1 \), one and only one constraint must be true
  ◦ Constraining \( U = 0 \), we force the conjunction of the negations to be true
  ◦ Constraining \( L > 0 \), the disjunction of the constraints is specified

• Disjunctive constructive constraint: \( c_1 \lor c_2 \)
  ◦ If properly handled, avoids search and backtracking
  ◦ E.g.: 
    \[
    \begin{align*}
    \text{nz}(X) & \leftarrow X > 0. \\
    \text{nz}(X) & \leftarrow X < 0. \\
    \end{align*}
    \]
    \[
    \begin{align*}
    \text{nz}(X) & \leftarrow X < 0 \lor X > 0. 
    \end{align*}
    \]
Other Primitives

- CLP(\(\mathcal{X}\)) systems usually provide additional primitives
- E.g.:
  - \texttt{enum(X)} enumerates \(X\) inside its current domain
  - \texttt{maximize(X)} (c.f. \texttt{minimize(X)}) works out maximum (minimum value) for \(X\) under the active constraints
  - \texttt{delay Goal until Condition} specifies when the variables are instantiated enough so that \texttt{Goal} can be effectively executed
    * Its use needs deep knowledge of the constraint system
    * Also widely available in Prolog systems
    * Not really a constraint: control primitive
Implementation Issues: Satisfiability

- Algorithms must be *incremental* in order to be practical
- Incrementality refers to the performance of the algorithm
- It is important that algorithms to decide satisfiability have a good average case behavior
- Common technique: use a *solved form* representation for satisfiable constraints
- Not possible in every domain
- E.g. in $\mathcal{FT}$ constraints are represented in the form $x_1 = t_1(\tilde{y}), \ldots, x_n = t_n(\tilde{y})$, where
  - each $t_i(\tilde{y})$ denotes a term structure containing variables from $\tilde{y}$
  - no variable $x_i$ appears in $\tilde{y}$

(i.e., idempotent substitutions, guaranteed by the unification algorithm)
Implementation Issues: Backtracking in CLP(\(\mathcal{X}\))

- Implementation of backtracking more complex than in Prolog
- Need to record changes to constraints
- Constraints typically stored as an association of variable to expression
- Trailing expressions is, in general, costly: cannot be stored at every change
- Avoid trailing when there is no choice point between two successive changes
- A standard technique: use time stamps to compare the age of the choice point with the age of the variable at the time of last trailing

\[
X < Y + Z, \quad Y = Z + W
\]

\[
X < Y + 4, \quad Y = 4 + W, \quad Z = 4
\]

\[
X < 9, \quad Y = 5, \quad Z = 4, \quad W = 1
\]

trail \(W\), timestamp it

trail \(X, Y, Z\), timestamp them

timestamp \(X, Y, Z, W\)
Implementation Issues: Extensibility

- Constraint domains often implemented now in Prolog-based systems using:
  - Attributed variables [Neumerkel,Holzbaur]:
    * Provide a hook into unification.
    * Allow attaching an attribute to a variable.
    * When unification with that variable occurs, user-defined code is called.
    * Used to implement constraint solvers (and other applications, e.g.,
      distributed execution).
  - Constraint handling rules (CHR):s:
    * Higher-level abstraction.
    * Allows defining propagation algorithms (e.g., constraint solvers) in a
      high-level way.
    * Often translated to attributed variable-based low-level code.
Attributed Variables Example: Freeze

- Primitives:
  - `attach_attribute(X,C)`
  - `get_attribute(X,C)`
  - `detach_attribute(X)`
  - `update_attribute(X,C)`
  - `verify_attribute(C,T)`
  - `combine_attributes(C1,C2)`

- Example: Freeze

```prolog
freeze( X, Goal) :-
    attach_attribute( V, frozen(V,Goal)),
    X = V.

verify_attribute( frozen(Var,Goal), Value) :-
    detach_attribute( Var),
    Var = Value,
    call(Goal).

combine_attributes( frozen(V1,G1), frozen(V2,G2)) :-
    detach_attribute( V1),
    detach_attribute( V2),
    V1 = V2,
    attach_attribute( V1, frozen(V1,(G1,G2))).
```
Programming Tips

- Over-constraining:
  - Seems to be against general advice “do not perform extra work”, but can actually cut more search space
  - Specially useful if infer is weak
  - Or else, if constraints outside the domain are being used

- Use control primitives (e.g., cut) very sparingly and carefully

- Determinacy is more subtle, (partially due to constraints in non–solved form)

- Choosing a clause does not preclude trying other exclusive clauses (as with Prolog and plain unification)

- Compare:

  \[
  \begin{align*}
  \text{max}(X, Y, X) & :\!- \ X & >. & \ Y. & \quad \text{?- max}(X, Y, Z). \\
  \text{max}(X, Y, Y) & :\!- \ X & \leq & \ Y. & \quad \text{Z & =. & X, Y & <. & X} \\
  \end{align*}
  \]

  with

  \[
  \begin{align*}
  \text{max}(X, Y, X) & :\!- \ X & >. & \ Y, & !. & \quad \text{?- max}(X, Y, Z). \\
  \text{max}(X, Y, Y) & :\!- \ X & \leq & \ Y. & \quad \text{Z & =. & X, Y & <. & X} \\
  \end{align*}
  \]
CLP Systems

• As mentioned before, CLP defines a class of languages obtained by
  ◊ Specifying the particular constraint system(s)
  ◊ Specifying the *Computation* and *Selection* rules

• Most practical systems include also the Herbrand domain with “=”, but then add
different domains and/or solver algorithms

• Most use the *Computation* and *Selection* rules of Prolog
Some Classic CLP Systems

- **CLP(R):**
  - Linear arithmetic over reals (\(=, \leq, >\)) – CLP(R)
  - Incremental Gaussian elimination and incremental Simplex

- **PrologIII:**
  - CLP(R)
  - Boolean (\(=\)), 2-valued Boolean Algebra – CLP(B)
  - Infinite (rational) trees (\(=, \neq\))
  - Equations over finite strings – CLP(WE)

- **CHIP (and its successor: the ILOG library):**
  - CLP(FD), CLP(B), CLP(Q)
  - User–defined constraints and solver algorithms

- **BNR-Prolog / CLP(BNR):**
  - Arithmetic over reals (closed intervals); CLP(FD), CLP(B).

- **RISC–CLP:**
  - Arithmetic constraints over reals, also non-linear
    (using Presburger arithmetic)
Some Current CLP Systems

- **clp(FD)/gprolog:**
  - CLP(FD).

- **SICStus:**
  - CLP(R), CLP(Q), CLP(FD)
  - Attributed variables and CHR for adding domains.

- **ECLiPS:**
  - CLP(R), CLP(Q), CLP(FD).

- **SWI:**
  - CLP(R), CLP(Q), CLP(FD), CLP(B).
  - Attributed variables and CHR for additional domains.

- **Ciao:**
  - CLP(R), CLP(Q), CLP(FD).
  - Attributed variables and CHR for additional domains.
  - Different domains can be activated on a per-module basis (packages).

→ Most Prolog systems now support constraints!
Some origins and other instances

• Ancestors:
  ◊ SKETCHPAD (1963), Waltz’s algorithm (1965?), THINGLAB (1981), MACSYMA (1983), ...

• Constraints in logic languages: – the origin of “constraint programming”:
  ◊ General theory developed (Jaffar and Lassez ’97).
  ◊ First, standalone systems developed: clpr, CHIP, ...
  ◊ Later, included in mainstream Prolog implementations.
  ◊ Has given to a whole

• Constraints in imperative languages:
  ◊ Equation solving libraries (ILOG, GECODE, ...)
  ◊ Timestamping of variables: \[ x := x + 1 \leftrightarrow x_{i+1} := x_i + 1 \]
    (similar to iterative methods in numerical analysis)

• Constraints in functional languages, via extensions:
  ◊ Evaluation of expressions including free variables.
  ◊ *Absolute Set Abstraction.*