CONSTRAINT LOGIC PROGRAMMING

Ivan Bratko
Faculty of Computer and Information Sc.
University of Ljubljana
Slovenia

CONSTRAINT LOGIC PROGRAMMING

- Constraint satisfaction
- Constraint programming
- Constraint Logic Programming (CLP) = Constraint programming + LP

EXAMPLE OF CLP

% Converting between Centigrade and Fahrenheit

```
convert( Centigrade, Fahrenheit)  :-
    Centigrade is (Fahrenheit - 32)*5/9 .
```

EXAMPLE OF CLP, CTD.

```
convert_clp( Centigrade, Fahrenheit)  :-
    { Centigrade = (Fahrenheit - 32)*5/9 }.

convert2_clp( Centigrade, Fahrenheit)  :-
    { 9*Centigrade = (Fahrenheit - 32)*5 }.
```

CONSTRAINT SATISFACTION PROBLEM

- *Given*:

(1) set of *variables*,

(2) *domains* of the variables

(3) *constraints* that the variables have to satisfy

- *Find*:

An assignment of values to the variables,

so that these values satisfy all the given constraints.

- In optimisation problems, also specify optimisation criterion

A SCHEDULING PROBLEM

- tasks a, b, c, d

- durations 2, 3, 5, 4 hours respectively
- precedence constraints

$$\begin{tikzcd}
 & b \arrow[r] & d \\
a \arrow[ur] \arrow[dr] & & \\
 & c &
\end{tikzcd}$$

CORRESPONDING CONSTRAINT SATISFACTION PROBLEM

- **Variables**: *Ta*, *Tb*, *Tc*, *Td*, *Tf*
- **Domains**: All variables are non-negative real numbers

- **Constraints**:

$0 \leq Ta$ (task *a* cannot start before 0)

$Ta + 2 \leq Tb$ (task *a* which takes 2 hours precedes *b*)

$Ta + 2 \leq Tc$ (*a* precedes *c*)

$Tb + 3 \leq Td$ (*b* precedes *d*)

$Tc + 5 \leq Tf$ (*c* finished by *Tf*)

$Td + 4 \leq Tf$ (*d* finished by *Tf*)

- **Criterion**: minimise *Tf*

SET OF SOLUTIONS

$Ta = 0$

$Tb = 2$

$2 \leq Tc \leq 4$

$Td = 5$

$Tf = 9$

APPLICATIONS OF CLP

- scheduling
- logistics
- resource management in production, transportation, placement
- simulation

APPLICATIONS OF CLP

Typical applications involve assigning resources to activities

- machines to jobs,
- people to rosters,
- crew to trains or planes,
- doctors and nurses to duties and wards

SATISFYING CONSTRAINTS

constraint networks:

nodes ~ variables

arcs ~ constraints

For each binary constraint $p(X,Y)$

there are two directed arcs (X,Y) and (Y,X)

$$\begin{tikzcd} X \arrow[r, bend left] & Y \arrow[l, bend left] \end{tikzcd}$$

CONSISTENCY ALGORITHMS

- *Consistency algorithms* operate over constraint networks

- They check consistency of domains of variables with respect to constraints.

- Here we only consider binary constraints.

ARC CONSISTENCY

- arc (X, Y) is *arc consistent*

  if for each value of X in Dx,

  there is some value for Y in Dy

  satisfying the constraint $p(X, Y)$.

- If (X, Y) is not arc consistent,

  then it can be made arc-consistent by deleting the values in Dx for which there is no corresponding value in Dy

ACHIEVING ARC-CONSISTENCY

- Example

  $Dx = 0..10$, $Dy = 0..10$

  $p(X,Y)$: $X+4 \leq Y$.

- arc (X,Y) is not arc consistent

  (for $X = 7$, no corresponding value of Y in Dy)

- To make arc (X,Y) consistent, reduce Dx to 0..6

- To make arc (Y,X) consistent, reduce Dy to 4..10.

ARC CONSISTENCY PROPAGATION

- Domain reductions propagate throughout network, possibly cyclically, until either

  (1) all arcs become consistent, or

  (2) some domain becomes empty
  (constraints unsatisfiable)

- By such reductions no solutions of the constraint problem are possibly lost.

WHEN ALL ARCS CONSISTENT

Two cases:

(1) Each domain has a single value:

a single solution to constraint problem.

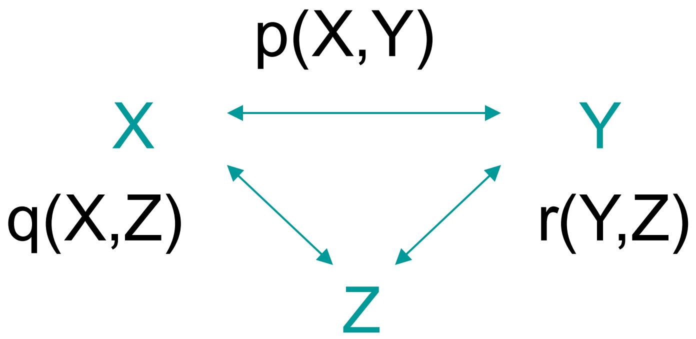
(2) All domains non-empty, and at least one domain has multiple values:

possibly several solutions, possibly no solution; combinatorial search needed over reduced domains

ARC CONSISTENCY AND GLOBAL SOLUTIONS

- Arc consistency does not guarantee that all possible combinations of domain values are solutions to the constraint problem.
- Possibly no combination of values from reduced domains satisfies all the constraints.

EXAMPLE



p(x1,y1). p(x2,y2).

q(x1,z1). q(x2,z2).

r(y1,z2). r(y2,z1).

- Network arc-consistent,

  but no solution to constraint problem.

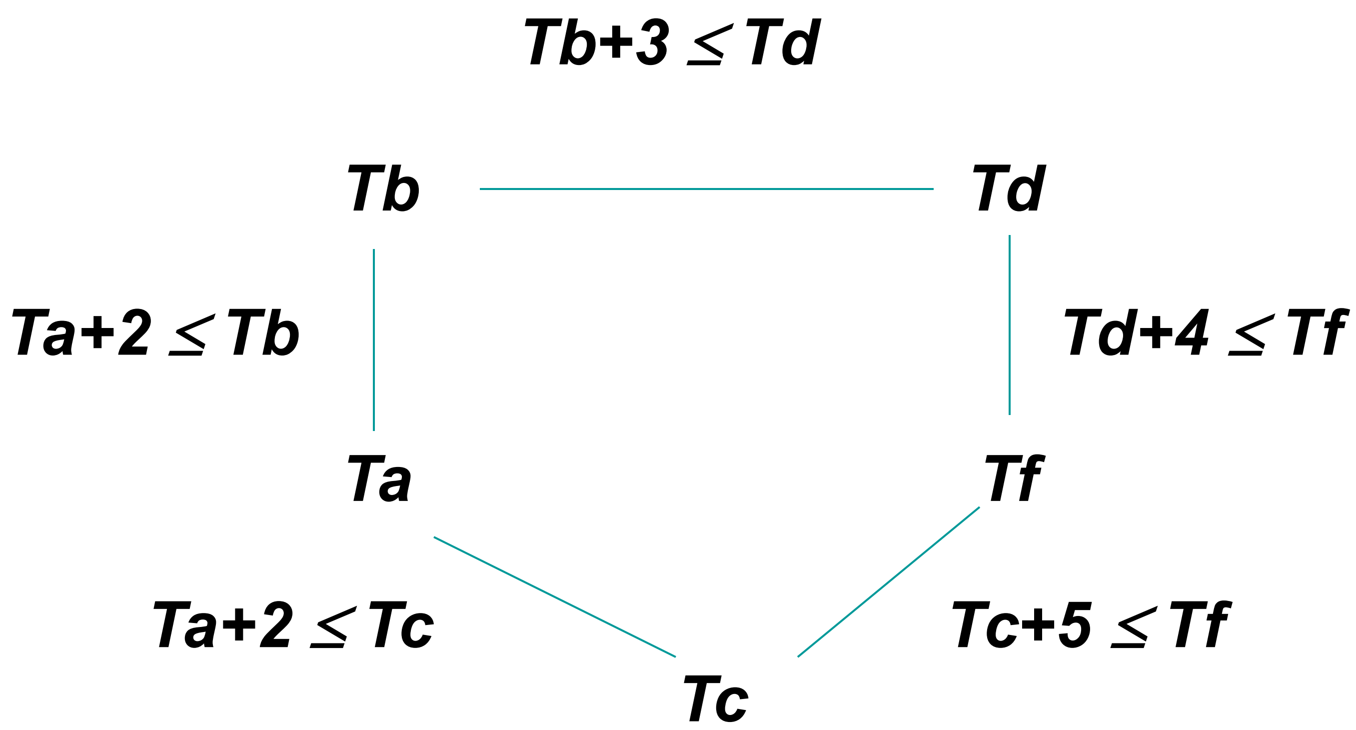
SOLUTION SEARCH IN ARC-CONSISTENT NETWORK

Several possible strategies, e.g.:

- choose one of the multi-valued domains and try repeatedly its values , apply consistency algorithm again
- choose one of the multi-valued domains and split it into two approximately equal size subsets; propagate arc-consistency for each subset, etc.

SCHEDULING EXAMPLE

Constraint network:



TRACE OF CONSISTENCY ALGORITHM

Step	Arc	*Ta*	*Tb*	*Tc*	*Td*	*Tf*
Start		0..10	0..10	0..10	0..10	0..10
1	(*Tb*,*Ta*)		2..10			
2	(*Td*,*Tb*)				5..10	
3	(*Tf*,*Td*)					9..10
4	(*Td*,*Tf*)				5..6	
5	(*Tb*,*Td*)		2..3			
6	(*Ta*,*Tb*)	0..1				
7	(*Tc*,*Ta*)			2..10		
8	(*Tc*,*Tf*)			2..5		

CONSTRAINT LOGIC PROGRAMMING

- Pure Prolog: limited constraint satisfaction language; all constraints are just equalities between terms
- CLP = Constraint solving + Logic Programming
- To extend Prolog to a "real" CLP languag: add other types of constraints in addition to matching

METAINTERPRETER FOR PROLOG WITH CONSTRAINTS

```
solve( Goal) :-
  solve( Goal, [ ], Constr).      % Start with empty constr.

% solve( Goal, InputConstraints, OutputConstraints)

solve( true, Constr0, Constr0).

solve( (G1, G2), Constr0, Constr) :-
  solve( G1, Constr0, Constr1),
  solve( G2, Constr1, Constr).
```

METAINTERPRETER CTD.

```
solve( G, Constr0, Constr)  :-
  prolog_goal( G),                    % G a Prolog goal
  clause( G, Body),                   % A clause about G
  solve( Body, Constr0, Constr).

solve( G, Constr0, Constr)  :-
  constraint_goal( G),                % G a constraint
  merge_constraints( Constr0, G, Constr).
```

MERGE CONSTRAINTS

- Predicate **merge_constraints:**

  constraint-specific problem solver,
  merges old and new constraints,
  tries to satisfy or simplify them

- For example, two constraints $X \leq 3$ and $X \leq 2$ are simplified into constraint $X \leq 2$.

CLP(X)

- Families of CLP techniques under names of form CLP(X), where X is a domain
- CLP(R): CLP over real numbers, constraints are arithmetic equalities, inequalities and disequalities
- CLP(Z) (integers)
- CLP(Q) (rational numbers)
- CLP(B) (Boolean domains)
- CLP(FD) (user-defined finite domains)

CLP(R): CLP over real numbers

- In CLP(R): linear equalities and inequalities typically handled efficiently, nonlinear constr. limited

Conventions from SICStus Prolog

```
?-  use_module( library( clpr)).

    ?- { 1 + X = 5 }.        % Numerical constraint
 X = 4
```

CLP(R) in Sicstus Prolog

- Conjunction of constraints C1, C2 and C3 is written as:
  { C1, C2, C3}
- Each constraint is of form:
- **Expr1 Operator Expr2**
- Operator can be:

=	for equations
=\=	for disequations
<, =<, >, >=	for inequations

CLP(R) in Sicstus Prolog

Example query to CLP(R)

```
?-  { Z =< X-2, Z =< 6-X, Z+1 = 2}.

Z = 1.0
{X >= 3.0}
{X =< 5.0}
```

TEMPERATURE CONVERSION

In Prolog:

```
convert( Centigrade, Fahrenheit)  :-
    Centigrade is (Fahrenheit - 32)*5/9.
```

```
?- convert( C, 95).
C = 35
```

```
?-  convert( 35, F).
Arithmetic error
```

TEMPERATURE CONVERSION, CTD.

In CLP(R) this works in both directions:

```
convert( Centigrade, Fahrenheit)  :-
   { Centigrade = (Fahrenheit - 32)*5/9 }.

?-  convert( 35, F).
F = 95

?-  convert( C, 95).
C = 35
```

TEMPERATURE CONVERSION, CTD.

Even works with neither argument instantiated:

```
?-  convert( C, F).
{ F = 32.0 + 1.8*C }
```

LINEAR OPTIMISATION FACILITY

- Built-in CLP(R) predicates:

  minimize(Expr)

  maximize(Expr)

- For example:

```
?-  { X =< 5}, maximize(X).
X = 5.0
?- { X =< 5, 2 =< X}, minimize( 2*X + 3).
X = 2.0
```

LINEAR OPTIMISATION FACILITY, CTD.

```
?-  {X >=2, Y >=2, Y =< X+1, 2*Y =< 8-X, Z = 2*X +3*Y},
      maximize(Z).

X = 4.0
Y = 2.0
Z = 14.0

?- { X =< 5}, minimize( X).
no
```

LINEAR OPTIMISATION FACILITY, CTD.

- CLP(R) predicates to find the supremum (least upper bound) or infimum (greatest lower bound) of an expression:

sup(Expr, MaxVal)

inf(Expr, MinVal)

Expr is a linear expression in terms of linearly constrained variables. Variables in **Expr** do not get instantiated to the extreme points.

SUP, INF

```
?-  { 2 =< X, X =< 5}, inf( X, Min), sup( X, Max).
Max = 5.0
Min = 2.0
{X >= 2.0}
{X =< 5.0}
```

OPTIMISATION FACILITIES

```
?-  {X >=2, Y >=2,
     Y =< X+1,
     2*Y =< 8-X,
     Z = 2*X +3*Y},
    sup(Z,Max), inf(Z,Min), maximize(Z).

X = 4.0
Y = 2.0
Z = 14.0
Max = 14.0
Min = 10.0
```

SIMPLE SCHEDULING

```
?-  { Ta + 2 =< Tb,          % a precedes b
      Ta + 2 =< Tc,          % a precedes c
      Tb + 3 =< Td,          % b precedes d
      Tc + 5 =< Tf,          % c finished by finishing time Tf
      Td + 4 =< Tf},         % d finished by Tf
      minimize( Tf).

Ta = 0.0, Tb = 2.0, Td = 5.0, Tf = 9.0
{Tc =< 4.0}
{Tc >= 2.0}
```

FIBONACCI NUMBERS WITH CONSTRAINTS

fib(N,F): F is the N-th Fibonacci number

F(0)=1, F(1)=1, F(2)=2, F(3)=3, F(4)=5, etc.

For N>1, F(N)=F(N-1)+F(N-2)

FIBONACCI IN PROLOG

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

FIBONACCI IN PROLOG

- Intended use:

?- fib(6,F).

F=13

- A question in the opposite direction:

?- fib(N, 13).

Error

- Goal N > 1 is executed with N uninstantiated

FIBONACCI IN CLP(R)

```
fib( N, F)  :-
    { N = 0, F = 1}
    ;
    { N = 1, F = 1}
    ;
    { N > 1, F = F1 + F2, N1 = N - 1, N2 = N - 2} ,
    fib( N1, F1),
    fib( N2, F2).
```

FIBONACCI IN CLP(R)

- This can be executed in the opposite direction:

?- fib(N, 13).

N = 6

- However, still gets into trouble when asked an unsatisfiable question:

?- fib(N, 4).

FIBONACCI IN CLP(R)
?- fib(N, 4).

The program keeps trying to find two Fibonacci numbers F1 and F2 such that F1+F2=4. It keeps generating larger and larger solutions for F1 and F2, all the time hoping that eventually their sum will be equal 4. It does not realise that once their sum has exceeded 4, it will only be increasing and so can never become equal 4. Finally this hopeless search ends in a stack overflow.

FIBONACCI: EXTRA CONSTRAINTS

- Fix this problem by adding constraints
- Easy to see: for all N: $F(N) \geq N$
- Therefore variables N1, F1, N2 and F2 must always satisfy the constraints:

F1 >= N1, F2 >= N2.

FIBONACCI: EXTRA CONSTRAINTS

```
fib( N, F)  :-
  .....
  ;
  { N > 1, F = F1+F2, N1 = N-1, N2 = N-2,
    F1 >= N1, F2 >= N2},    % Extra constraints
  fib( N1, F1),
  fib( N2, F2).
```

FIBONACCI: EXTRA CONSTRAINTS

?- fib(N, 4).

no

- The recursive calls of **fib** expand the expression for F in the condition **F = 4**:

4 = F = F1 + F2 =

F1' + F2' + F2 =

F1'' + F2'' + F2' + F2

FIBONACCI: EXTRA CONSTRAINTS

- The recursive calls of **fib** expand the expression for F in the condition **F = 4**:

4 = F = F1 + F2 =

F1' + F2' + F2 =

F1'' + F2'' + F2' + F2

- Additional constraints that make the above unsatisfiable:

F1’ >= N1’ > 1, F2’’ >= N2’’ > 1,

F2’ >= N2’ > 1, F2 >= N2 > 1

FIBONACCI: EXTRA CONSTRAINTS

- Each time this expression is expanded, new constraints are added to the previous constraints. At the time that the four-term sum expression is obtained, the constraint solver finds out that the accumulated constraints are a contradiction that can never be satisfied.

CLP(Q): CLP OVER RATIONAL NUMBERS

- Real numbers represented as quotients between integers

- Example:

?- { X = 2*Y, Y = 1-X }.

- A CLP(Q) solver gives:

X = 2/3, Y = 1/3

- A CLP(R) solver gives something like:

X = 0.666666666, Y = 0.333333333

SCHEDULING

Scheduling problem considered here is given by:

- A set of tasks *T1*, ..., *Tn*
- Durations *D1*, ..., *Dn* of the tasks
- Precedence constraints prec(*Ti*, *Tj*)

  Ti has to be completed before *Tj* can start
- Set of *m* processors available for executing the tasks
- Resource constraints:

  which tasks may be executed by which processors

SCHEDULING

- Schedule assigns for each task:
  processor + start time

- Respect:
  - precedence constraints
  - resource constraints:
    processor suitable for task
    one task per processor at a time

VARIABLES IN CONSTRAINT PROBLEM

For each task **Ti:**

Si start time

Pj processor name

FinTime finishing time of schedule (to be minimised)

SPECIFICATION OF A SCHEDULING PROBLEM

By predicates:

tasks([Task1/Duration1, Task2/Duration2, ...])
gives the list task names and their durations

prec(Task1, Task2)
Task1 precedes **Task2**

resource(Task, [Proc1, Proc2, ...])
Task can be done by any of processors **Proc1**, ...

SCHEDULING WITHOUT RESOURCE CONSTRAINTS

This is an easy special case

1. Construct inequality constraints between starting times of tasks, corresponding to precedences among the tasks.

2. Minimise finishing time within the constructed inequality constraints.

As all constraints are linear inequalities,

so this is linear optimisation (built-in facility in CLP(R))

FORMULATING PRECEDENCE CONSTR.

Tasks a, b

Start times: **Ta**, **Tb**

Durations: **Da**, **Db**

Constraint **prec(a,b)** translates into numerical inequality:

{ Sa + Da =< Sb }

All start times **Si** positive, all tasks finished by **FinTime**:

{ Si >= 0, Si + Di =< FinTime }

PREDICATE SCHEDULE

schedule(Schedule, FinTime)

Schedule is a best schedule for problem specified by predicates **tasks** and **prec**

FinTime is the finishing time of this schedule.

Representation of a schedule is:

Schedule = [Task1/Start1/Duration1,
Task2/Start2/Duration2, ...]

SCHEDULING, UNLIMITED RES.

```
% Scheduling with CLP with unlimited resources

schedule( Schedule, FinTime)  :-
  tasks( TasksDurs),
  precedence_constr( TasksDurs, Schedule, FinTime),
           % Construct precedence constraints
  minimize( FinTime).
```

SCHEDULING, UNLIMITED RES., CTD.

```
precedence_constr( [ ], [ ], FinTime).

precedence_constr( [T/D | TDs], [T/Start/D | Rest], FinTime)  :-
  { Start >= 0,                          % Earliest start at 0
    Start + D =< FinTime},               % Must finish by FinTime
  precedence_constr( TDs, Rest, FinTime),
  prec_constr( T/Start/D, Rest).
```

SCHEDULING, UNLIMITED RES., CTD.

```
% prec_constr( TaskStartDur, OtherTasks):
%     Set up precedence constr. between Task and other tasks


prec_constr( _, [ ]).


prec_constr(  T/S/D,  [T1/S1/D1 | Rest])  :-
  (  prec( T, T1), !, { S+D =< S1}            % T precedes T1
     ;
     prec( T1, T), !, { S1+D1 =< S}           % T1 precedes T
     ;
     true ),
  prec_constr( T/S/D, Rest).
```

SCHEDULING, UNLIMITED RES., CTD.

```
% List of tasks to be scheduled
tasks( [ t1/5, t2/7, t3/10, t4/2, t5/9]).

% Precedence constraints
prec( t1, t2).   prec( t1, t4).   prec( t2, t3).   prec( t4, t5).

?-  schedule( Schedule, FinTime).
FinTime = 22,
Schedule = [t1/0/5,t2/5/7,t3/12/10,t4/S4/2,t5/S5/9],
{S5 =< 13}   {S4 >= 5}   {S4–S5 =< -2}
```

SCHEDULING WITH RESOURCE CONSTRAINTS

- Schedule also has to assign processors to tasks:

  Schedule = [Task1/Proc1/Start1/Dur1,
  Task2/Proc2/Start2/Dur2, ...]

- Handling precedence constraints: similar as before

- Handling resource constraints: requires combinatorial search among possible assignments

ASSIGNING PROCESSORS

- To search among possible assignments:
  keep track of best finishing time so far

- Whenever assigning a suitable processor to a task,
  add constraint:

  { FinTime < BestFinTimeSoFar }

- This is branch-and-bound principle

```
% Scheduling with limited resources

schedule( BestSchedule, BestTime)  :-
  tasks( TasksDurs),
  precedence_constr( TasksDurs, Schedule, FinTime),
              % Set up precedence inequalities
  initialise_bound,              % Initialise bound on finishing time
  assign_processors( Schedule, FinTime),      % Assign proc. to tasks
  minimize( FinTime),
  update_bound( Schedule, FinTime),
  fail                                      % Backtrack to find more schedules
  ;
  bestsofar( BestSchedule, BestTime).            % Final best
```

```
% assign_processors( Schedule, FinTime):
%   Assign processors to tasks in Schedule

assign_processors( [ ], FinTime).

assign_processors( [T/P/S/D | Rest], FinTime)  :-
  assign_processors( Rest, FinTime),
  resource( T, Processors),               % Suitable processors for task T
  member( P, Processors),                 % Choose one of processors
  resource_constr( T/P/S/D, Rest),        % Impose resource constraints
  bestsofar( _, BestTimeSoFar),
  { FinTime < BestTimeSoFar }.            % New schedule better all previous
```

```
% resource_constr( ScheduledTask, TaskList):
%   ensure no resource conflict between ScheduledTask and
     TaskList

resource_constr( _, [ ]).

resource_constr( Task, [Task1 | Rest])  :-
  no_conflict( Task, Task1),
  resource_constr( Task, Rest).
```

NO CONFLICT BETWEEN PROCESSOR ASSIGNMENTS

```
no_conflict( T/P/S/D, T1/P1/S1/D1)  :-
  P \== P1, !              % Different processors
  ;
  prec( T, T1), !          % Already constrained
  ;
  prec( T1, T), !          % Already constrained
  ;
  {  S+D =< S1             % Same processor, no time overlap
     ;
     S1+D1 =< S }.
```

COMPLEXITY

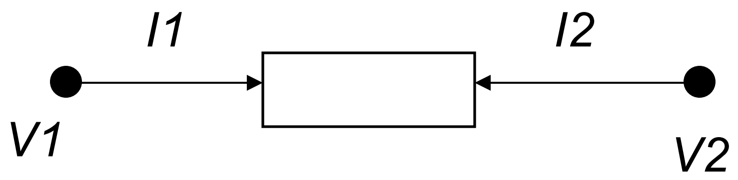
- This process is combinatorially complex - exponential number of possible assignments of processors to tasks
- Bounding a partial schedule by **BestTimeSoFar** leads to abandoning sets of bad schedules *before* they are completely built
- Savings in computation time depend on how good the upper bound is
- The tighter upper bound, the sooner bad schedules are recognised and abandoned
- The sooner some good schedule is found, the sooner a tight upper bound is applied

SIMULATION WITH CONSTRAINTS

- Elegant when system consists of components and connections among components

- Example: electric circuits

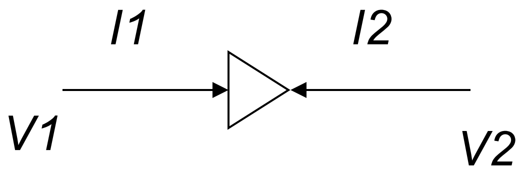
ELECTRIC CIRCUITS IN CLP



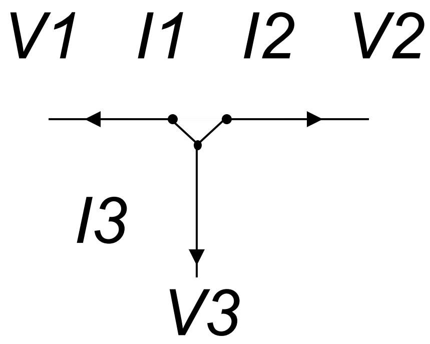

```
% resistor( T1, T2, R):
%  R=resistance; T1, T2 its terminals
%   T1 = (I1, V1),   T2 = (I2, V2)

resistor( (V1, I1), (V2, I2), R)  :-
  { I1 = -I2, V1-V2 = I1*R }.
```

ELECTRIC CIRCUITS IN CLP

```
% diode( T1, T2): T1, T2 terminals of a diode
%   Diode open in direction from T1 to T2

diode( (V1,I1), (V2,I2) )  :-
  { I1 + I2 = 0},
  { I1 > 0, V1 = V2
    ;
    I1 = 0, V1 =< V2}.

battery( (V1,I1), (V2,I2), Voltage)  :-
  { I1 + I2 = 0, Voltage = V1 - V2 }.
```



CONNECTIONS



Constraints:

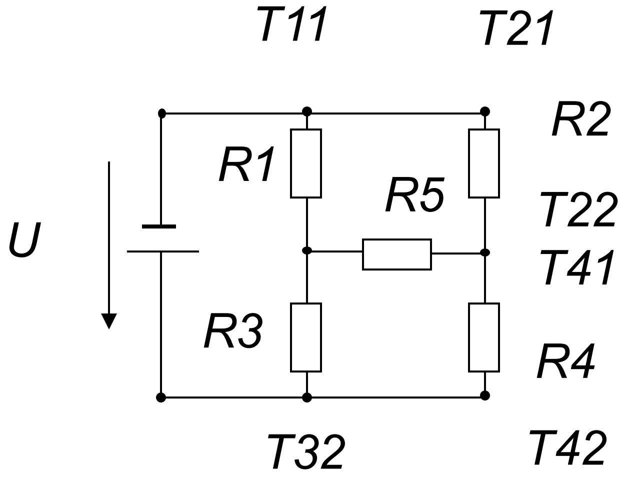
V1 = V2 = V3

I1 + I2 + I3 = 0

ELECTRIC CIRCUITS IN CLP

```
% conn( [T1,T2,...]): Terminals T1, T2, ... connected
%  Therefore all el. potentials equal, sum of currents = 0

conn( Terminals)  :-
  conn( Terminals, 0).

conn( [ (V,I) ], Sum)  :-
  { Sum + I = 0 }.

conn( [ (V1,I1), (V2,I2) | Rest], Sum)  :-
  { V1 = V2, Sum1 = Sum + I1},
  conn( [ (V2, I2) | Rest ], Sum1).
```

WHEATSTONE CIRCUIT



WHEATSTONE CIRCUIT

```
circuit_wheat( U, T11, T21, T31, T41, T51, T52)  :-
    T2 = ( 0, _),                    % Terminal T2 at potential 0
    battery( T1, T2, U),
    resistor( T11, T12, 5),          % R1 = 5
    resistor( T21, T22, 10),         % R2 = 10
    resistor( T31, T32, 15),         % R3 = 15
    resistor( T41, T42, 10),         % R4 = 10
    resistor( T51, T52, 50),         % R5 = 50
    conn( [T1, T11,T21]),
    conn( [T12, T31, T51]),
    conn( [T22, T41, T52]),
    conn( [T2, T32, T42]).
```

QUERY TO SIMULATOR

- Given the battery voltage 10 V, what are the electrical potentials and the current at the "middle" resistor R5?

```
?- circuit_wheat(10, _, _, _, _, T51, T52).
```

```
T51 = ( 7.3404..., 0.04255...)
T52 = ( 5.2127..., -0.04255...)
```

- So the potentials at the terminals of R5 are 7.340 V and 5.123 V respectively, and the current is 0.04255 A.

CLP over finite domains: CLP(FD)

- In Sicstus: Domains of variables are sets of integers

- Constraints:

X in Set

where **Set** can be:

{Integer1, Integer2, ...}

Term1..Term2	set between **Term1** and **Term2**
Set1 \/ Set2	union of **Set1** and **Set2**
Set1 /\ Set2	intersection of **Set1** and **Set2**
\ Set1	complement of **Set1**

ARITHMETIC CONSTRAINTS

Arithmetic constraints have the form:

Exp1 Relation Exp2

Exp1, Exp2 are arithmetic expressions

Relation can be:

#=	equal
#\=	not equal
#<	less than
#>	greater than
#=<	less or equal
etc	

EXAMPLE

```
?-  X in 1..5, Y in 0..4,
    X #< Y, Z #= X+Y+1.
```

```
X in 1..3
Y in 2..4
Z in 3..7
```

indmain

?- X in 1..3, indomain(X).

X = 1;

X = 2;

X = 3

domain, all_different

domain(L, Min, Max)

all the variables in list L have domains **Min**..**Max**.

all_different(L)

all the variables in L must have different values.

labeling

labeling(Options, L)

generates concrete possible values of the variables in list L.

Options is a list of options regarding the order in which the variables in L are "labelled".

If Options = [] then by default the variables are labelled from left to right

CRPTARITHMETIC PUZZLE

% Cryptarithmetic puzzle DONALD+GERALD=ROBERT in CLP(FD)

```
solve( [D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T])  :-
  Vars = [D,O,N,A,L,G,E,R,B,T],             % All variables in the puzzle
  domain( Vars, 0, 9),                       % They are all decimal digits
  all_different( Vars),                      % They are all different
  100000*D + 10000*O + 1000*N + 100*A + 10*L + D  +
  100000*G + 10000*E + 1000*R + 100*A + 10*L + D  #=
  100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
  labeling( [ ], Vars).
```

EIGHT QUEENS

```
% 8 queens in CLP(FD)

solution( Ys)  :-                  % Ys is list of Y-coordinates of queens
  Ys = [ _,_,_,_,_,_,_,_ ],        % There are 8 queens
  domain( Ys, 1, 8),               % All the coordinates have domains 1..8
  all_different( Ys),              % All different to avoid horizontal attacks
  safe( Ys),                       % Constrain to prevent diagonal attacks
  labeling( [ ], Ys).              % Find concrete values for Ys
```

QUEENS, CTD.

```
safe( [ ]).
safe( [Y | Ys])  :-
  no_attack( Y, Ys, 1),    % 1 = horizontal distance between queen Y and Ys
  safe( Ys).

% no_attack( Y, Ys, D): %   queen at Y doesn't attack any queen at Ys;
%   D is column distance between first queen and other queens

no_attack( Y, [ ], _).

no_attack( Y1, [Y2 | Ys], D)  :-
  D #\= Y1-Y2,
  D #\= Y2-Y1,
  D1 is D+1,
  no_attack( Y1, Ys, D1).
```