APLAI
Other Constraint Programming systems + Modelling issues

Gerda Janssens
Departement computerwetenschappen
A01.26
http://www.cs.kuleuven.be/~gerda/APLAI

Overview

1. Other CLP systems and Industrial Applications
    1. ILOG
    2. OPL
    3. Mozart (Oz)
2. Modeling issues
    1. Viewpoints – channeling constraints
    2. Symmetry

Integration of constraints and search into programming languages

- Declarative constraints to model combinatorial (optimization) problems
- Have easy access to search procedures
- Logic programming: a declarative style + non-determinism
- Constraint technologies: successful!
- Also in other paradigms: broader audience
- Also for modeling concurrency: agents with a shared constraint store
- To design specialized, dedicated constraint solvers

Industrial Applications of CLP

- One of the first applications for the container port of Hong Kong.
- For production scheduling, transport scheduling, financial planning, a host of resource planning and scheduling.
- Now constraint-based technology companies in Europe, Asia, and America.
- Revenue of ILOG: 125Million dollars per annum. In January 2009 acquisition by IBM
- Cisco uses CLP for its internet routing products.
- Cadence – the electronic design automation company – is a major user of constraint technology.

Other constraint programming systems: libraries and glass-box extensibility

- ILOG optimization products : CPLEX + Solver C++ libraries; proprietary
- Artelys Kalis:
  constraint programming as C++ library
  Note: Xpress-MP solves linear and quadratic programs, with integer variables, using various numeric components: primal and dual simplex, interior point, branch-and-bound and branch-and-cut algorithms.
- Other libraries : Gecode(C++), Choco(Java)

Other constraint programming systems: to modeling

- OPL is a rich declarative modeling language
- Comet is an object-oriented language featuring modeling and control abstractions to support constraint-based local search.
- Oz (Mozart programming system) a multi-paradigm language that is designed for advanced, concurrent, networked, soft real-time, and reactive applications

ILOG: SEND+MORE = MONEY

```
#include <ilsolver/ctint.h>

CtInt dummy = CtInit();
CtIntVar S(1, 9), E(0, 9), N(0, 9), D(0, 9), M(1, 9), O(0, 9), R(0, 9), Y(0, 9);
CtIntVar* AllVars[]=  {&S, &E, &N, &D, &M, &O, &R, &Y};

int main(int, char**) {
   CtAllNeq(8, AllVars);
   CtEq(        1000*S + 100*E + 10*N + D
             + 1000*M + 100*O + 10*R + E,
       10000*M + 1000*O + 100*N + 10*E + Y);
   CtSolve(CtGenerate(8, AllVars));

   PrintSol(CtInt, AllVars);
   CtEnd();
   return 0;
}
```

- Since the FD variables are special objects not belonging to the C++ language itself, but defined as part of a class, they cannot be treated in the program in the same way as primary C++ objects: for example, printing them or accessing their values has to be done with special methods provided by the class.

ILOG and search

- ILOG uses an embedded goal interpreter to evaluate an and-or tree data structure
  - Or node expresses non-determinism
  - And node represents a conjunction
- ILOG has predefined search tree methods (macro's) and builds the and-or tree on the fly

ILOG core engines: OPL

- Optimization Programming Language (OPL) is the core of ILOG OPL Studio. OPL lets the user state optimization problems quickly, accurately and naturally without low-level complexities of ordinary programming languages.
- OPL is unique in its support of both mathematical programming and constraint programming.
- OPL is strongly typed.
- Full support for linear and integer programming; support for the modeling and search aspects of constraint programming.

More OPL information

- OPL lets users customize search strategies, a key requirement for successful constraint programming. Scheduling problems can be modeled naturally in terms of activities and resources. OPL supports logical constraints and other concise expressions rarely supported by linear mathematical programming tools.
- OPL provides an abstract connection to databases, allowing integration of external data stored in ODBC, Oracle,Informix and other databases into an optimization application, and to export solutions to those same databases.
- Direct connections to Microsoft Excel spreadsheets allow two-dimensional data and solutions to be rapidly integrated into an OPL application. OPLScript provides a powerful high-level procedural language for solving sequences of OPL models and developing sophisticated solution strategies.

OPL constraint programming: queens

```
int    n << "number of queens:";
range  Domain 1..n;
var    Domain queens[Domain];
solve {
      forall(ordered i,j in Domain) {
         queens[i] <> queens[j] ;
         queens[i] + i <> queens[j] + j ;
         queens[i] - i <> queens[j] - j
      };
};
search {
   forall (i in 1..8)
      tryall(v in 1..8)                    % firstfail heuristic
         queens[i] = v;
};
```

The warehouse problem in OPL

The problem consists of assigning stores to warehouses such that each store is assigned to exactly one warehouse, while simultaneously deciding which warehouses to open.

The data given is the cost of assigning each store to each warehouse, a fixed cost that is incurred when the warehouse is opened, and the capacity (in terms of number of stores) of each potential warehouse.

Mathematical Programming

```
1 int fixed = ...;
2 int nbStores = ...;
3 enum Warehouses ...;
4 range Stores 1..nbStores;
5 int capacity[Warehouses] = ...;
6 int supplyCost[Stores,Warehouses] = ...;
7
8 var int open[Warehouses] in 0..1;
9 var int supply[Stores,Warehouses] in 0..1;
```

8: array of binary decision variables : particular warehouse is open

9: whether a store s is assigned to warehouse w.

Mathematical Programming

```
minimize
  sum(w in Warehouses) fixed * open[w] +
  sum(w in Warehouses, s in Stores) supplyCost[s,w] * supply[s,w]
subject to {
   forall(s in Stores)
        sum(w in Warehouses) supply[s,w] = 1;
   forall(w in Warehouses, s in Stores)
         supply[s,w] <= open[w];
   forall(w in Warehouses)
         sum(s in Stores) supply[s,w] <= capacity[w];
};
```

each store assigned to 1 warehouse
a store can only be assigned to an open warehouse: supply[s,w] <= open[w]
limited number of stores assigned to a warehouse

Solving integer programming

- Same branch and bound approach as in ECLiPse
- Linear-programming relaxation (removing the integrality restrictions in the mixed-integer program).
- Picking an integral variable that is still fractional; split the search space and add restrictions.

Constraint Programming

```
int fixed = ...;
int nbStores = ...;
enum Warehouses ...;
range Stores 1..nbStores;
int capacity[Warehouses] = ...;
int supplyCost[Stores,Warehouses] = ...;
int maxCost = max(s in Stores, w in Warehouses) supplyCost[s,w];
% maximum cost assigning any store to any warehouse
Warehouses wlist[w in Warehouses] = w;
% decision variables
var int open[Warehouses] in 0..1;
var Warehouses supplier[Stores];  % which warehouse assigned to which store
var int cost[Stores] in 0..maxCost;  % used in search strategy

```

Constraint Programming

```
14 minimize
15    sum(s in Stores) supplyCost[s, supplier[s]] +
        sum(w in Warehouses) fixed * open[w]
16 subject to {
17    forall(w in Warehouses)
18        sum(s in Stores)   (supplier[s] = w )   <= capacity[w];
19    forall(s in Stores )
20        open[supplier[s]] = 1;
```

18: the number of stores whose supplier is w cannot exceed the capacity of w

20: the warehouse w must be open if it is the supplier of store s

Constraint Programming: variant

```
14 minimize
15 sum(s in Stores) cost[s] + sum(w in Warehouses) fixed * open[w]
16 subject to {
17    forall(s in Stores)
18        cost[s] = supplyCost[s,supplier[s]];
19    forall(s in Stores )
20        open[supplier[s]] = 1;
21    forall( w in Warehouses)
21*       sum(s in Stores) (supplier[s] = w) <= capacity[w];
22 };
24 search {
25    forall(s in Stores ordered by decreasing regretdmin(cost[s]))
26       tryall(w in Warehouses
27            ordered by increasing supplyCost[s,w])
28       supplier[s] = w;
29 };
```

Also : integration between integer and constraint programming model

- Integer programming uses linear relaxation to produce a lower bound at each node of the search tree
- Constraint programming has the advantage of expressing a heuristic search procedure
- Combination!!!

From: IBM-ILOG website

- A car manufacturer increased productivity by 30%
- Chile's two largest forest-products companies reduced their truck fleets by 30%
- A semiconductor manufacturer cut wafer-processing cycle time in half, to just 30 days
- A major airline responded to unexpected delays with efficient crew rescheduling, saving $40 million in one year
- A package-delivery company cut costs by $87 million
- A television network increased annual advertising revenue by $50 million
- An investment firm cut transaction costs by $100 million
- A major consumer packaged goods (CPG) manufacturer dramatically increased the direct loading of trucks off its packaging lines

Mozart Programming system

- The Mozart system provides state-of-the-art support in two areas: open distributed computing and constraint-based inference.
- Mozart implements Oz, a concurrent object-oriented language with dataflow synchronization.
- Oz combines concurrent and distributed programming with logical constraint-based inference, making it a unique choice for developing multi-agent systems.
- Mozart is an ideal platform for both general-purpose distributed applications as well as for hard problems requiring sophisticated optimization and inferencing abilities.

Oz: SEND+MORE =MONEY

```
proc {Money Root}
    S E N D M O R Y
  in
    Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)     % 1
    Root ::: 0#9                                     % 2
    {FD.distinct Root}                               % 3
    S \=: 0                                          % 4
    M \=: 0
              1000*S + 100*E + 10*N + D              % 5
    +          1000*M + 100*O + 10*R + E
    =: 10000*M + 1000*O + 100*N + 10*E + Y
    {FD.distribute ff Root}
  end
```

Explanation

The script first declares a local variable for every letter. Then it posts the following constraints:

1. Root = sol(s:S e:E n:N d:D m:M o:O r:R y:Y)   (basic constraint)
   Root is a record that has a field for every letter. The fields of Root are the digits for the corresponding letters.
2. Root ::: 0#9   (basic)
   The fields of Root are integers in the domain 0#9.
3. {FD.distinct Root}   (nonbasic constraint)
   The fields of Root are pairwise distinct.
4. S \=: 0   M \=: 0   (nonbasic)
   The values of the variables S and M are different from zero (no leading zeros).
5. The digits for the letters satisfy the equation SEND+MORE=MONEY.
6. {FD.distribute ff Root}   uses first-failure heuristic in search

Oz: queens ….

```
fun {Queens N}
    proc {$ Row}
      L1N ={MakeTuple c N}
      LM1N={MakeTuple c N}
    in
      {FD.tuple queens N 1 #N Row}
      {For 1 N 1 proc {$ I}
                 L1N.I=I LM1N.I=~I
               end}
      {FD.distinct Row}
      {FD.distinctOffset Row LM1N}
      {FD.distinctOffset Row L1N}
      {FD.distribute generic(value:mid) Row}
    end
  end
```

Warehouses

http://www.mozart-oz.org/documentation/fdt/node45.html#program.warehouse

Further topics

- Global constraints: `:- library(ic_global)`
  `alldifferent/1` with stronger propagation behaviour.
- Constraint based scheduling, route planning
  `:- library(ic_cumulative)`
  `cumulative(Starttimes, Durations, Resources, ResourceLimit)`
- Local search
- Hard and soft constraints
- Modeling issues

- Concurrent constraint (logic) programming

2. Modeling issues: CSP program for a problem P

- The entities in the problem P have to represented in CSP by variables and values.
- The rules and restrictions in P have to be expressed in constraints.
- CSP uses the notion of a viewpoint (X,D) with
  X a set of variables and
  D the corresponding domains
- The viewpoint should allow us to express the constraints easily and concisely.
- Moreover the constraints should force propagation!!

2.1 Typically: several possible viewpoints

- For n-queens <X,D>
- X: set of columns D: set of rows
- Dual one: switch variables and values, thus
- X: set of rows D: set of columns
- Other ones?
- X: the n queens D: {1,2,…., n^2} the squares
- X: b_ij (expressing that x_i has value j)
  with as domain D: {0,1}
- Boolean viewpoint!!

Class of permutation problems

- N- queens is a permutation problem:
  the union of the domains
  has the same number of elements
  as there are variables
  and each variable must be assigned a different value.
- Every variable is assigned exactly one value

column	row
row	column

- Switch role of variables and values

Another permutation problem

- Finding an nxn magic square containing the numbers 1 .. n^2 so that the sum of every row and column is the same.
- X? D?
- X: cells/squares   D:{1,…, n^2}
- X: {1,…, n^2}   D: set of cells
- Constraints on the row/column sums easier for ???

See example code

```
% N by N grid of cells: array Square
% sum of row I
   Sum #= sum(Square[I,1..N]),
```

sum(+ExprList,-Result) evaluates the arithmetic expressions in ExprList and unifies their sum with Result.

Combining viewpoints

- (X1,D1,C1)   and (X2,D2,C2)
- Combined CSP
  - X1 U X2
  - C1 U C2 U Cc
- Cc are channeling constraints that express the relationship between sets X1 and X2 aiming at when a variable in X1(X2) is assigned a value, it has an impact on X2(X1).

Channeling constraints for permutation problems

- x1, x2, …, xk with domain {1,2,…, k}
- Dual vars: d1,…, dk with domain {1,2,…, k}
- (xi = j) ≡ (dj = i)
- …If xi takes value j then dj has to be i…
- How to write it?
- (with reified constraints)

Setting up the channeling constraints

```
q(N) :-
   length(Xs,N), length(Ds,N),
   Xs::1..N, Ds::1..N,

   channel(N,Xs,Ds),
   labeling(Ds),
   write(now(Xs,Ds)),nl.

channel(N,Xs,Ds):-
   ( count(I,1,N),
     param(Xs,Ds,N)
   do nthelement(I,Xs,Xi),
      ( count(J,1,N),
        param(Ds,Xi,I)
      do
        nthelement(J,Ds,Dj),
        #=(Xi,J,B), #=(Dj,I,B)
      )
   ).
```

Selecting constraints: C1 U C2??

- Combined CSP allows the constraints to be expressed in the most convenient viewpoint.
- Often unnecessary to include all C1 U C2
- Permutation problem: channeling constraints are sufficient to ensure that all the values assigned to x1, …, xk are distinct.
  No need anymore for alldifferent!!

Viewpoints for sudoku?

- Trivial approach: uses as finite domain variables the (r,c) squares
- Which can take the values 1 .. 9
- Another approach: numbers in a block, (b,n)
- Which can take positions …..
- ….
- Channeling constraints?
- ….
- Always include sources/references!!!

More on viewpoints & channeling

- Smith, Barbara M. "Modelling for constraint programming." *Lecture Notes for the First International Summer School on Constraint Programming. Available at: http://www. math. unipd. it/frossi/cp-school* (2005).
- also in section 1.9 ofhttp://www-module.cs.york.ac.uk/copr/handbook-modelling.pdf
- http://4c.ucc.ie/~hsimonis/ELearning/index.htm Chapter 9: Choosing the Model: Sports Scheduling

2.2 Symmetry

- As a property of the solution set
- As a property of the statement of the problem (of the constraints)

- In terms of values (values can be permuted), or variables, or variable-value pairs.

- What is the point about it?

Breaking symmetry: reduce search space

- Never explore two search states that are symmetric to each other.
  1. By reformulation of the problem
  2. By adding symmetry breaking constraints before search: making symmetric solutions unacceptable, but leaving at least one in each symmetric equivalence class
  3. By adapting the search algorithm to break symmetry

2.2.1 Reformulation

- X: queens D: the n^2 squares
  Now an unnecessary notion of the first queen, the second queen, …
  Different solutions to the CSP can correspond to the same layout on the board

- The usual viewpoint does not have this symmetry!!

- Reformulation can reduce the symmetry!!!

Reformulation

- Symmetry inherent to a problem:
  set of persons in a crew.
- Symmetry: ???
  Among the crew members p1 p2 p3
  Among the crews themselves c1 c2
- Constraint solver that supports set variables.
  :- library(ic_sets).
  Set variable for a crew:
  breaks permutations of people
  Still permutation of crews:
  card(set_crew1) >= card(set_crew2) >= …

Reformulation

- Symmetry inherent to a problem:
  set of persons in a crew.
- Constraint solver that supports set variables.
  :- library(ic_sets).
- But propagation is not easy for sets!!
- Reformulation: no known general technique

2.2.2 Symmetry breaking constraints

- X: queens D: the n^2 squares
- Now an unnecessary notion of the first queen, the second queen, …
- X1 =< X2 =< ….

- Typically used for the elements of a set: e.g. the n queens in the set.
- All elements are different: use </2.

1..19 : every diagonal adds up to 38

```
    A,B,C,
   D,E,F,G,
  H,I,J,K,L,
   M,N,O,P,
    Q,R,S
```

% rotated solutions:
A must be the smallest corner

```
A #< C, A #< H, A #< L,
 A #< S, A #< Q
```

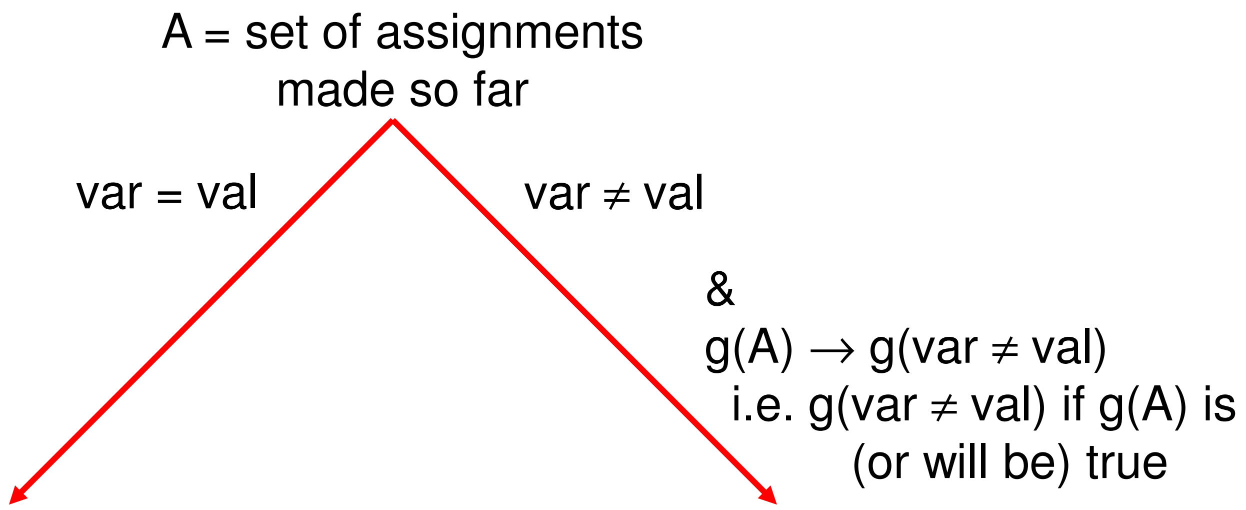
% solutions mirrored:
on the A-S diagonal

```
C #< H
```

Lex-Leader (for variable symmetries)

- Define a canonical solution and add constraints to choose it.
- Assume an ordering on variables
- Define an ordering on the solutions
- Compare 2 solutions by **lexicographic ordering**
- A canonical solution is smaller than any of its symmetric variants

2.2.3 Symmetry Breaking on Backtracking


A = set of assignments made so far
var = val
var ≠ val
&
g(A) → g(var ≠ val)
i.e. g(var ≠ val) if g(A) is (or will be) true

With g(A) a symmetric variant of A

The end

- Some ECLiPSe programming
- Lots of libraries
- Research topics

- Principles also useful in other systems
- You!!!