Constraint Programming

- An overview

- Examples of decision (making) problems
- Declarative Modelling with Constraints
- Finite and Continuous Domains
- An Introduction to CHOCO

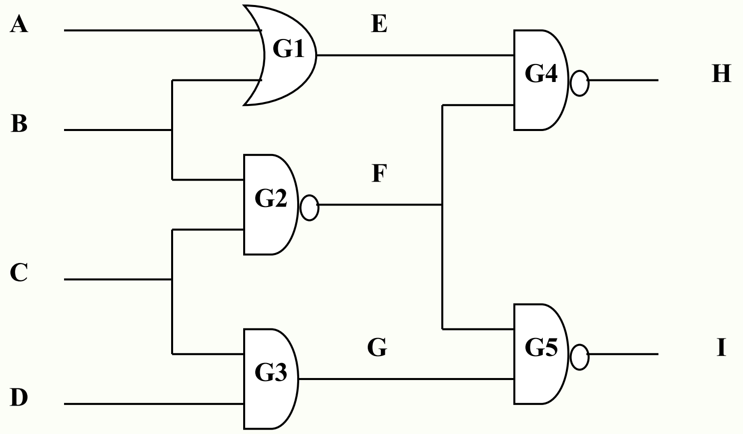
Constraint Problems: Examples

- Decision Making Problems include:
  - Modelling of Digital Circuits
  - Production Planning
  - Network Management
  - Scheduling
  - Assignment (Colouring, Latin/ Magic Squares, Sudoku, Circuits, ...)
  - Assignment and Scheduling (Timetabling, Job-shop)
  - Filling and Containment
- Typically a problem may be represented by different models, some of which may be more adequate (ease of modelling, efficiency of solving in a given solver, etc)

Modeling of Digital Circuits

Goal (Example): Determine a test pattern that detects some faulty gate

- Variables:
  - Signals in the circuit
- Domain:
  - Booleans: 0/1 (or True/False, or High/Low)
- Constraints:
  - Equality constraints between the output of a gate and its “boolean operation” (e.g. and, or, not, nand, ...)



```
E = or(A,B)      % G1
F = nand(B,C)    % G2
G = and(B,C)     % G3
H = nand(E,F)    % G4
I = nand(F,G)    % G5
```

Production Planning

Goal (Example): Determine a production plan

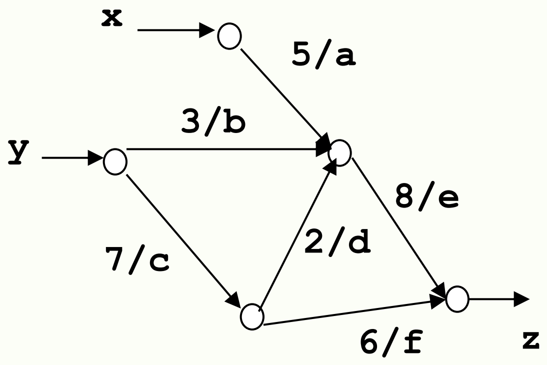
- Variables:
  - Quantities of goods to produce
- Domain:
  - Rational/Reals or Integers
- Constraints:
  - Equality and Inequality (linear) constraints to model resource limitations, minimal quantities to produce, costs not to exceed, balance conditions, etc...

```
Find x, y and z such that
 4x+ 3y + 6z ≤ 1500    % resources used do not exceed 1500
  x + y + z >= 300     % production not less than 300 units
  x ≤ z + 20           % x units within z ± 20 units
  x ≥ z - 20
  x, y, z ≥ 0          % non negative production
```

Network Management

Goal (Example): Determine acceptable traffic on a netwok

- Variables:
  - Flows in each edge
- Domain:
  - Rational/Reals (or Integers)
- Constraints:
  - Equality and Inequality (linear) constraints to model capacity limitations, flow maintenance, costs, etc...

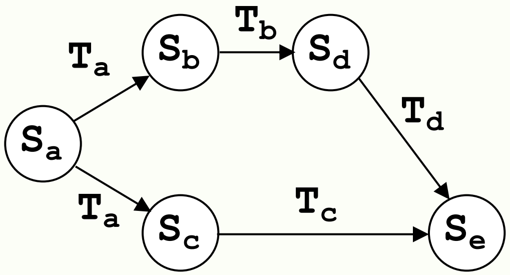


```
Find x,y,z, a,b,c,d,e such that
   x ≥ 6, z ≥ 10          % minimum flow
   a ≤ 5, ... , f ≤ 6   % capacity
% flow maintenance
  x = a, y = b + c, a + b + d = e,
  c = d + f, e + f = z
  x, y ,z, a, b, c, d, e, f ≥ 0
```

Schedulling

Goal (Example): Assign timing/precedence to tasks

- Variables:
  - Start Timing of Tasks, Duration of Tasks
- Domain:
  - Rational/Reals or Integers
- Constraints:
  - Precedence Constraints, Non-overlapping constraints, Deadlines, etc...



Find S_a ,...., S_e, such that

$S_b \geq S_a+T_a$, % precedence

....

$(S_c \geq S_b+T_b) \vee (S_b \geq S_c+T_c)$ % non overlap

...,

$6 \geq Sa+Ta$ % deadline

S_a,..., $S_e \geq 0$

Assignment

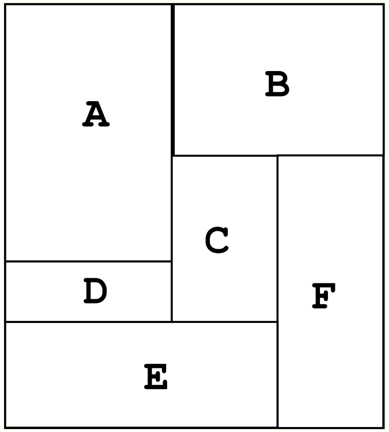
Many constraint problems can be classified as assignment problems. In general all that can be stated is that these problems follow a general CSP goal :

Assign values to the variables to satisfy the relevant constraints.

- Variables:
  - Objects / Properties of objects
- Domain:
  - Finite Discrete /Integer or Infinite Continuous /Real or Rational Values
    - colours, numbers, duration, load
  - Booleans for decisions
- Constraints:
  - Compatibility (Equality, Difference, No-attack, Arithmetic Relations)

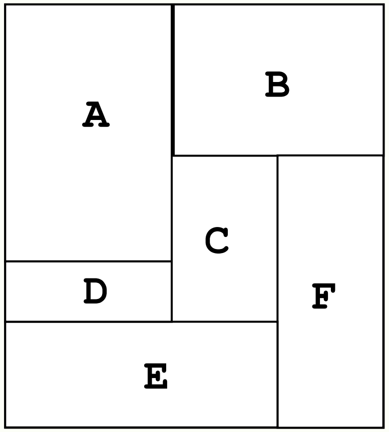
Some examples may help to illustrate this class of problems

Assignment (2)



Graph Colouring (Finite Domains)

```
Assign values to A, .., F,
  s.t.  A, B, .., F ∈ {red, blue, green}
        A ≠ B, A ≠ C, A ≠ D,
        B ≠ C, B ≠ F, C ≠ D, C ≠ E, C ≠ F
        D ≠ E, E ≠ F
```



Graph Colouring (0/1 or Booleans – but not SAT)

```
Assign values to A1,A2, .., F1,F2
  s.t.  Ar, Ab, Ag, .., Fr, Fb, Fg ∈ {0,1}
% one and only one colour for A, B, ..., F
        Ar + Ab + Ag = 1;
        ....
% different colours for A and B, ...
Ar + Br <= 1; Ab + Bb <= 1; Ag + Bg <= 1;
        ....
```

Assignment (3)

Q1		●		
Q2				●
Q3	●			
Q4			●	

N-queens (Finite Domains):

Assign Values to Q1,..., Qn $\in$ {1,.., n}

s.t. $\forall_{i\neq j}$ noattack (Qi, Qj)

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

Latin Squares (similar to **Sudoku**)**:**

Assign Values to X11,..., X33 $\in$ {1,.., 3}

s.t. $\forall_k \; \forall_i \; \forall_{j\neq i} \; X_{ki} \neq X_{kj}$ % same row

$\forall_k \; \forall_i \; \forall_{j\neq i} \; X_{ik} \neq X_{jk}$ % same column

X_{11}	X_{12}	X_{13}
X_{21}	X_{22}	X_{23}
X_{31}	X_{32}	X_{33}

Magic Squares:

Assign Values to X11,..., X33 $\in$ {1,..,9}

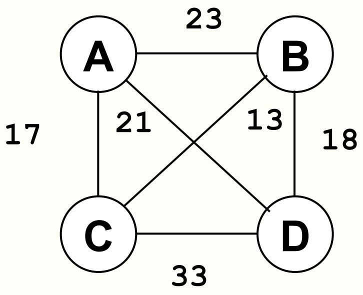
s.t. $\forall_i \; \forall_{j\neq i} \; \Sigma_k \; X_{ki} = \Sigma_k \; X_{kj} = M$ % same rows sum

$\forall_i \; \forall_{j\neq i} \; \Sigma_k \; X_{ik} = \Sigma_k \; X_{jk} = M$ % same cols sum

$\Sigma_k \; X_{kk} = \Sigma_k \; X_{k,n-k+1} = M$ % diagonals

$\forall_{i\neq k} \vee \forall_{j\neq l} \; X_{ij} \neq X_{kl}$ % all different

Assignment (3)



Travelling Salesperson (Finite Domains)

Find values for A, B, C, D $\in$ {1,..,4}

s.t. A $\neq$ B, ..., C $\neq$ D

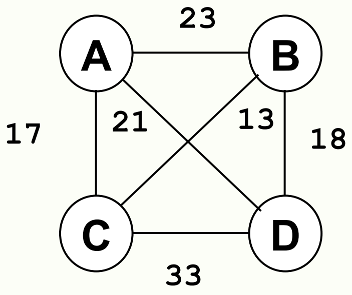
% a permutation of [A,B,C,D]

if A = B+1 then $X_A = L_{ba}$,

...

if D = C+1 then $X_D = L_{cd}$

$X_A + X_B + X_C + X_D \leq k$



Travelling Salesperson (0/1 or Booleans – but not SAT)

Find decision values for $X_{ab} \ldots X_{dc} \in \{0,1\}$

s.t. $\forall_a \Sigma_k \ X_{ak} = 1$

$\forall_a \Sigma_k \ X_{ka} = 1$

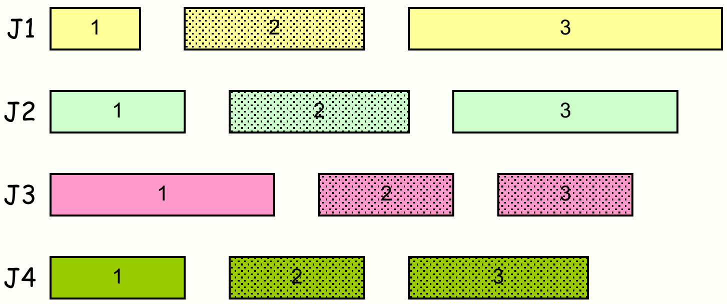
... no subcycle constraints

$\Sigma_a \Sigma_b \ X_{ab} \ L_{ab} < k$

Mixed: Assignment and Scheduling

Goal (Example): Assign values to variables

- Variables:
  - Start Times, Durations, Resources used
- Domain:
  - Integers (typicaly) or Rationals/Reals
- Constraints:
  - Compatibility (Conditional, Disjunctive, Difference, Arithmetic Relations)



Job-Shop

Assign values to $S_{ij} \in \{1,..,n\}$ % time slots

and to $M_{ij} \in \{1,..,m\}$ % machines available

% precedence within job

$$\forall_j \ \forall_{i < k} \ S_{ij} + D_{ij} \leq S_{kj}$$

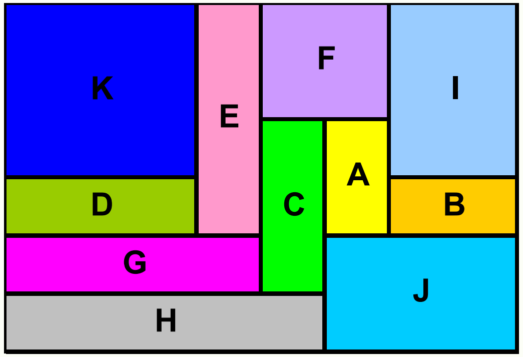
% either no-overlap or different machines

$$\forall_{i,j,k,l} \ (M_{ij} = M_{kl}) \rightarrow (S_{ij} + D_{ij} \leq S_{kl}) \vee (S_{kl} + D_{kl} \leq S_{ij})$$

Filling and Containment

Goal (Example): Assign values to variables

- Variables:
  - Point Locations
- Domain:
  - Integers (typicaly) or Rationals/Reals
- Constraints:
  - Non-overlapping (Disjunctive, Inequality)



Fiiling

```
Assign values to $X_i \in \{1,..,\ Xmax\}$ % X-dimension
                 $Y_i \in \{1,..,\ Ymax\}$ % Y-dimension
% no-overlapping rectangles
        $\forall_{i,j}$    $(X_i+Lx_i \leq X_j)$   % I to the left of J
                $(X_j+Lx_j \leq X_i)$   % I to the right of J
                $(Y_i+Ly_i \leq Y_j)$   % I in front of J
                $(Y_j+Lx_j \leq X_i)$   % I in back of J
```

Constraint Satisfaction Problems

- Other Examples (from CP-16):
  - Finding Patterns for DataMining
    - Rather than finding rules (as in ID3 /CS4.5) whole sets must be obtained
    - e.g. sequences of letters in ADN / Protein searches
  - Hospital Residence Problem (with pairs)
    - Kind of Stable Marriage Problem but pairings make it NP-Hard
    - Both Hospitals and Residents (junior doctors) have a list of preferences
    - Pairs of Residents have joint preferences

Constraint Satisfaction Problems

- Formally a constraint satisfaction problem (CSP) can be regarded as a tuple <X, D, C>, where
  - $X = \{ X_1, \ldots , X_n\}$ is a set of variables
  - $D = \{ D_1, \ldots , D_n\}$ is a set of domains (for the corresponding variables)
  - $C = \{ C_1, \ldots , C_m\}$ is a set of constraints (on the variables)

- Solving a constraint problem consists of determining values $x_i \in D_i$ for each variable X_i, satisfying all the constraints C.

- Intuitively, a constraint C_i is a limitation on the values of its variables.

- More formally, a constraint C_i (with arity k) over variables $X_{i1}, \ldots, X_{ik}$ ranging over domains $D_{i1}, \ldots, D_{ik}$ is a subset of the cartesian cartesian $D_{j1}\times \ldots \times D_{jk}$.

$$C_i \subseteq D_{j1}\times \ldots \times D_{jk}$$

Constraints and Optimisation Problems

- In many cases, one is interested not only in satisfying some set of constraints but also in finding among all solutions those that optimise a certain objective function (minimising a cost or maximising some positive feature).

- Formally a constraint (satisfaction and) optimisation problem (CSOP or COP) can be regarded as a tuple <V, D, C, F>, where
  - $X = \{ X_1, \dots , X_n\}$ is a set of variables
  - $D = \{ D_1, \dots , D_n\}$ is a set of domains (for the corresponding variables)
  - $C = \{ C_1, \dots , C_m\}$ is a set of constraints (on the variables)
  - F is a function on the variables

- Solving a constraint satisfaction and optimisation problem consists of determining values $x_i \in D_i$ for each variable X_i, satisfying all the constraints C and that optimise the objective function.

Decision Problems are NP-complete

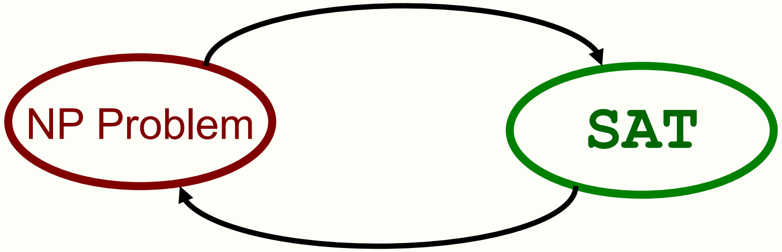
- All the problems presented are decision problems in that a decision has to be made regarding the value to assign to each variable.

- Non-trivial decision making problems are untractable, i.e. they lie in the class of NP problems.

- Formally, these are the problems that can be solved in polinomial time by a non-deterministic machine, i.e. one that “guesses the right answer”.

- For example, in the graph colouring problem (n nodes, k colours), if one has to assign colours to n nodes, a non-deterministic machine could guess a solution in O(n) steps.

- As a class, NP-complete problems may be converted in polinomial time onto other NP-complete problems (SAT, in particular).



Decision Problems are NP-complete

- No one has already found a polynomial algorithm to solve SAT (or any other NP problem), and hence the conjecture P ≠ NP (perhaps one of the most challenging open problems in computer science) is regarded as true.

- Hence, with real machines and non trivial problems, one has to guess the adequate values for the variables and make mistakes. In the worst case, one has to test $O(k^n)$ potential solutions.

- Just to have an idea of the complexity, the table below shows the time needed to check kn solutions, assuming one solution is examined in 1 μsec (times in secs).

k^n		n					
		10	20	30	40	50	60
k	2	1.0E-03	1.0E+00	1.1E+03	1.1E+06	1.1E+09	1.2E+12
	3	5.9E-02	3.5E+03	2.1E+08	1.2E+13	7.2E+17	4.2E+22
	4	1.0E+00	1.1E+06	1.2E+12	1.2E+18	1.3E+24	1.3E+30
	5	9.8E+00	9.5E+07	9.3E+14	9.1E+21	8.9E+28	8.7E+35
	6	6.0E+01	3.7E+09	2.2E+17	1.3E+25	8.1E+32	4.9E+40

1 hour = $3.6 * 10^3$ sec   1 year = $3.2 * 10^7$ sec   TOUniv = $4.7 * 10^{17}$ sec

Decision Problems are NP-complete

- Still, constraint solving problems are NP-complete problems (as SAT is).
- If a non-deterministic machine (that guesses correctly) can solve a problem in polynomial time, then a real deterministic machine can check in polinomial time whether a potential solution satisfies all the constraints.
- More important: with an appropriate search strategy, many instances of NP-complete problems can be solved in quite acceptable times.
- Hence, search plays a fundamental role in solving this kind of problems. Adequate search methods and appropriate heuristics can often solve large instances of these problems in very acceptable time.

Search Strategies

- There are two main types of search strategies that have been adopted to solve combinatorial problems:

Complete Backtrack Search Methods:

- Solutions are incrementally **completed**, by assigning values to "undecided" variables and backtrack whenever any constraint is violated;
- These methods are complete: if a solution exists it is found in finite time.
- More importantly, they can proof non-satisfiability.

Incomplete Local Search Methods:

- Complete "solutions" are incrementally **repaired**, by changing the values assigned to some of the variables until a "real solution" is found;
- These local search methods are not guaranteed to avoid revisiting the same solutions time and again and are therefore incomplete.
- They are often very efficient to find very good solutions (local optima)

Optimisation Problems are NP-hard

- Optimisation problems are typically NP-hard problems in that solving them is at least as difficult as solving the corresponding decision problem.

- In practice these problems cannot be solved in polynomial time by a non-deterministic machine, nor can they be checked by a deterministic machine.

- In fact, to find an optimal solution it is not enough to find it ... It is necessary to show that it is better than all other solutions!

- Being harder than the decision problems, optimisation problems also require adequate search strategies, if larger instances are to be solved.
    - In complete search, detection of failure and subsequent backtracking may be imposed if the partial solution can be proved to be no better than one already found (branch & bound).

Declarative Programming

- Programming a combinatorial problem thus requires
  - The specification of the constraints of the problem; and
  - The specification of a search algorithm
- The separation of these two aspects has for a long time been advocated by several programming paradigms, namely functional programming and logic programming.
- Logic programming in particular has a built-in mechanism for search (backtracking) that makes it easy to extend into constraint (logic) constraint programming, by “replacing” its underlying **resolution** to **constraint propagation.** A number of Constraint Logic Programming languages have been proposed (CHIP, ECLiPSE, GNU Prolog, SICStus) to explore this extension of logic programming.
- More recently, other declarative languages such as Comet (OO-like), Choco (Java Library) and Zinc, provide more convenient data structures for modelling, maintaining a declarative approach.

Constraint Programming

Constraint Programming (and Languages) is driven by a number of goals

- Expressivity
    - Constraint Languages should be able to easily specify the variables, domains and constraints (e.g. conditional, global, etc...);
- Declarative Nature
    - Ideally, programs should specify the constraints to be solved, not the algorithms used to solve them
- Efficiency
    - Solutions should be found as efficiently as possible, i.e. with the minimum possible use of resources (time and space).

These goals are partially conficting goals and have led to the various developments in this research and development area.

Search Methods – Pure Backtracking

- In this course we will focus on these two aspects of Constraint Programming:
  - **Declarative Modelling**
    - How to specify as naturally as possible the problem we want to solve
  - **Efficient Execution**
    - How to solve the problems thus specified as efficiently as possible, combining, as we should study, Heuristics with constraint propagation.
- These topics will be studied in the context of two types of domains
  - **Finite Domains**
    - discrete domains, basically integer intervals)
  - **Continuous Domains**
    - in principle, the difference between two values can be as small as we may want.

Constraint Programming – Finite Domains

- In Finite domains we will see that the the efficiency obtained in solving a problem with CP depends on many issues that will be addressed in the course:

  1. Formalization of Constraint Propagation
  2. Types of constraints and their main features
  3. Alternative models
     a. Redundant Constraints
     b. Symmetry Breaking Constraints
  4. Heuristics that are most commonly used
  5. Testing these techniques with Choco in several non-trivial examples

- These aspects will be studied in the first part of the course (first 6 weeks).

Constraint Programming - Continuous Domains

Continuous constraints require somewhat different methods for constraint propagation as well as enumeration. The main differences to consider are:

1. In a domain **lo .. hi** there are infinite values to consider. Hence enumeration cannot be a simple test of the alternative values, backtracking if necessary.

2. Constraints should consider variables whose domains are intervals, and adapt standard arithmetic to consider such domains – interval arithmetic.

3. Advanced methods can be used to propagate constraints, more sophisticated than naïve methods adapted from the finite domains (e.g. interval Newton).

4. Approximations are often necessary (e.g. rounding off arithmetic operations) and care must be taken that errors are not made (so as to loose solutions).

- Constraints in these continuous domains will be covered in the second part of the course, by Prof. Jorge Cruz.

Constraint Programming - Continuous Domains

A summary of this second part:

1. Continuous Constraint Satisfaction Problems
2. Continuous Constraint Reasoning
   a. Representation of Continuous Domains
   b. Pruning and Branching
3. Solving Continuous CSPs
   a. Constraint Propagation
   b. Consistency Criteria
4. Practical Examples

Constraint Programming - Continuous Domains

A major concern of dealing with continuous constraints regards constraint propagation.

For these part of the course some topics will be dealt more formally, namely:

1. Interval Constraints Overview
2. Intervals, Interval Arithmetic and Interval Functions
3. Interval Newton Method
4. Associating Narrowing Functions to Constraints
5. Constraint Propagation and Consistency Enforcement

Assessment

- Evaluation consists of the following components
  - Project 1 – Finite Domains Problem
  - Mini-Test 1 – Finite Domains Concepts
  - Project 2 – Continuous Domains Problem
  - Mini-Test 2 – Continuous Domains Concepts
- Projects are made in team work (2 students per group) and the tests assess the students individually.
- All components have the same weight for the final grade.
- Students that do not get the minimum grade, are allowed to do a repetition exam if they get at least an average grade of 8/20 in the two projects.
- Exact dates to be announced –
  - Project 1 and Mini-test 1 at mid-term (end October)
  - Project 2 and Mini-test 2 at the end of semester (mid December)

Constraint Programming by Example

First example: SEND+MORE = MONEY

- Find the digits encoded by letters, where different letters stand for different digits, and the symbolic sum below stands (the leftmost digits are not zero):

```
  S E N D
+ M O R E
---------
M O N E Y
```

- Similarly to all combinatorial problems, a declarative approach (as taken by Constraint Logic Programming) solves this problem by separating the two components:
  - **Model**: What are the variables that will be chosen for the problem unknowns, and the constraints that must be satisfied
  - **Search**: What strategies are used to assign values to variables

Constraint Programming by Example

Modelling

- There are two main steps in modelling a problem:

  1. Choose variables to represent the unknowns
     - What are the variables
     - What values can they take
  2. Select the constraints that these variables must satisfy according to the conditions of the problem;
     - How to constrain the variables
     - Are there alternative (more efficient?) sets of constraints?

- These decisions are often interdependent as illustrated in this problem.

```
    S E N D
  + M O R E
  ---------
  M O N E Y
```

Constraint Programming by Example

Model 1 :

- Variables Adopted:
  - One variable for each letter (we use the letter as the name of the variable)
  - Each variable takes values in 0 to 9
- Constraints to be Satisfied:
  - All variables must be different;
  - The sum must be correct
  - No leading zeros

```
    S E N D
  + M O R E
  ---------
  M O N E Y
```

Constraint Programming by Example

Model 1 : In Choco, this model may be specified as follows

```
    S  E  N  D
 +  M  O  R  E
 -------------
 M  O  N  E  Y
```

```
package choco;

import org.chocosolver.solver.Model;
import org.chocosolver.solver.*;
import org.chocosolver.solver.variables.IntVar;

public class sendmory {
   public static void main(String[] args) {
      Model model = new Model("send + more = money");
      // Declaration of variables
      // Specification of constraints
      // execute
      // show results
   }
}
```

Constraint Programming by Example

Model 1 : In Choco, this model may be specified as follows

```
// Declaration of variables
IntVar s = model.intVar("S", 0, 9);
IntVar e = model.intVar("E", 0, 9);
IntVar n = model.intVar("N", 0, 9);
IntVar d = model.intVar("D", 0, 9);
IntVar m = model.intVar("M", 0, 9);
IntVar o = model.intVar("O", 0, 9);
IntVar r = model.intVar("R", 0, 9);
IntVar y = model.intVar("Y", 0, 9);
IntVar op1 = model.intVar("S", 0, 10000);
IntVar op2 = model.intVar("S", 0, 10000);
IntVar res = model.intVar("S", 0, 100000);
// Specification of constraints
// execute
// show results
```

```
    S E N D
  + M O R E
  ---------
  M O N E Y
```

Constraint Programming by Example

Model 1 : In Choco, this model may be specified as follows

```
    S  E  N  D
 +  M  O  R  E
 -------------
 M  O  N  E  Y
```

```
      // Declaration of variables
      // Specification of constraints
        model.arithm(m, ">", 0).post();
        model.arithm(s, ">", 0).post();
        model.arithm(res, "=", op1, "+", op2).post();

        // op1 = 1000s + 100e +10n + d
        op1.eq(s.mul(1000).add(e.mul(100)).add(n.mul(10)).add(d)).post();
        op2.eq(m.mul(1000).add(o.mul(100)).add(r.mul(10)).add(e)).post();
        res.eq(m.mul(10000).add(o.mul(1000)).add(n.mul(100)).add(e.mul(10)
).add(y)).post();

        model.allDifferent(new IntVar[]{s, e, n, d, m, o, r, y}).post();

      // execute
      // show results
```

Constraint Programming by Example

Model 2 :

- There is an alternative modelling, that represents the total sum as it is usually operated with “carries”
  - One variable for each letter (we use the letter as the name of the variable)
    - Each variable takes values in 0 to 9
  - 4 Carries
    - Each carry takes value 0 or 1

```
C4  C3  C2  C1
     S   E   N   D
 +   M   O   R   E
 ─────────────────
 M   O   N   E   Y
```

- Constraints to be Satisfied:
  - All variables must be different;
  - All the sums (digit by digit, including carries) must be correct
  - No leading zeros

Constraint Programming by Example

Model 2 : This alternative model can also be expressed in Choco

```
// Declaration of variables
IntVar s = model.intVar("S", 0, 9);
IntVar e = model.intVar("E", 0, 9);
IntVar n = model.intVar("N", 0, 9);
IntVar d = model.intVar("D", 0, 9);
IntVar m = model.intVar("M", 0, 9);
IntVar o = model.intVar("O", 0, 9);
IntVar r = model.intVar("R", 0, 9);
IntVar y = model.intVar("Y", 0, 9);
IntVar c1 = model.intVar("C1", 0, 1);  //carries
IntVar c2 = model.intVar("C2", 0, 1);
IntVar c3 = model.intVar("C3", 0, 1);
IntVar c4 = model.intVar("C4", 0, 1);
// Specification of constraints
// execute
// show results
```

C4	C3	C2	C1	
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Constraint Programming by Example

Model 2 : This alternative model can also be expressed in Choco

```
C4 C3 C2 C1
   S  E  N  D
+  M  O  R  E
M  O  N  E  Y
```

```
// Declaration of variables
// Specification of constraints
  model.arithm(c4,"=", m).post();
  model.arithm(m, ">", 0).post();
  model.arithm(s, ">", 0).post();

  // d + e = y + 10 c1
  d.add(e).eq(y.add(c1.mul(10))).post();
  // c1 + n + r = e + 10 c2
  c1.add(n).add(r).eq(e.add(c2.mul(10))).post();
  c2.add(e).add(o).eq(n.add(c3.mul(10))).post();
  c3.add(s).add(m).eq(o.add(c4.mul(10))).post();

  model.allDifferent(new IntVar[]{s, e, n, d, m, o, r, y}).post();
// execute
// show results
```

Constraint Programming by Example

Enumeration :

- Once the variables are declared and the constraints posted, the constraint solver should find values for the variables in some efficient way.
- This is because the underlying constraint propagation process does not guarantee that the problem has a solution!
- It simply removes values from the domain of variables that guaranteedely do not belong to any solution.
- The enumeration is typically achieved in Choco with method **solve()**, that assigns values to the input variables and backtracks when this is impossible.
- The labelling process may be more or less efficient, depending on the heuristics used. A fairly good heuristic is the fail-first that assigns values to the variables with less values in their domains. In Choco, that may be expressed by setting the search policy
    - slv.setSearch(minDomLBSearch(vars));
- More sophisticated heuristics may nevertheless be programmed by the user.

Constraint Programming by Example

Model 1 : In Choco, this model may be specified as follows

```
 C4 C3 C2 C1
     S  E  N  D
 +   M  O  R  E
 ---------------
 M   O  N  E  Y
```

```
       // Declaration of variables
       // Specification of constraints

       // execute
       Solver slv = model.getSolver();
       if (slv.solve()){
          System.out.println("  " + Integer.toString(1000*s.getValue()+
100*e.getValue()+10*n.getValue()+d.getValue()));
          System.out.println("+" + Integer.toString(1000*m.getValue()+
100*o.getValue()+10*r.getValue()+e.getValue()));
          System.out.println("-----");
          System.out.println(10000*m.getValue()+1000*o.getValue()+
100*n.getValue()+10*e.getValue()+y.getValue());
       } else {
          System.out.println("no solutions");
       }
```

Constraint Programming

- An Introduction to **CHOCO**

Constraint Programming Languages

- A number of (pedagogical) reasons might justify Comet:
  - It is stand-alone
    - not a library of Java or C++, as is the case of Choco and Gecode.
  - It includes solvers for both
    - Constraint Programming; and
    - Constrained Local Search
    - As a full fledged language, it allows the full programming of heuristics.
      - in Zinc, heuristics cannot be fully specified (a number of annotations are available but they are not sufficient for some problems).
  - Nevertheless, **Comet** has a major problem in that it has been discontinued, and replaced by Objective-CP (designed by the same authors – Pascal Van Hentenryck and Laurent Michel Modelling.

Constraint Programming Languages

- **Choco** is a set of Java libraries that supports CP (Complete Backtrack Search) and is thus adopted in the course, although not exclusively.
- As mentioned, the alternative language, **Comet**, previously used in the course, has been discontinued (although it may still be used).
- Meanwhile, a language that is becoming a standard, for CP alone, is Zinc / MiniZinc.
- In particular, it provides an interface (Flat-Zinc) that almost all existing CP solvers can support (Gecode, Choco, SICStus, … CaSPER).
- This makes it possible to test solvers in a competition held annually with the CP conferences.

- Given the above said, we will use Choco in this course (but Comet may be used alternatively).

8-queens problem

$Q_1 = 1$

$Q_2 = 5$

$Q_3 = 8$

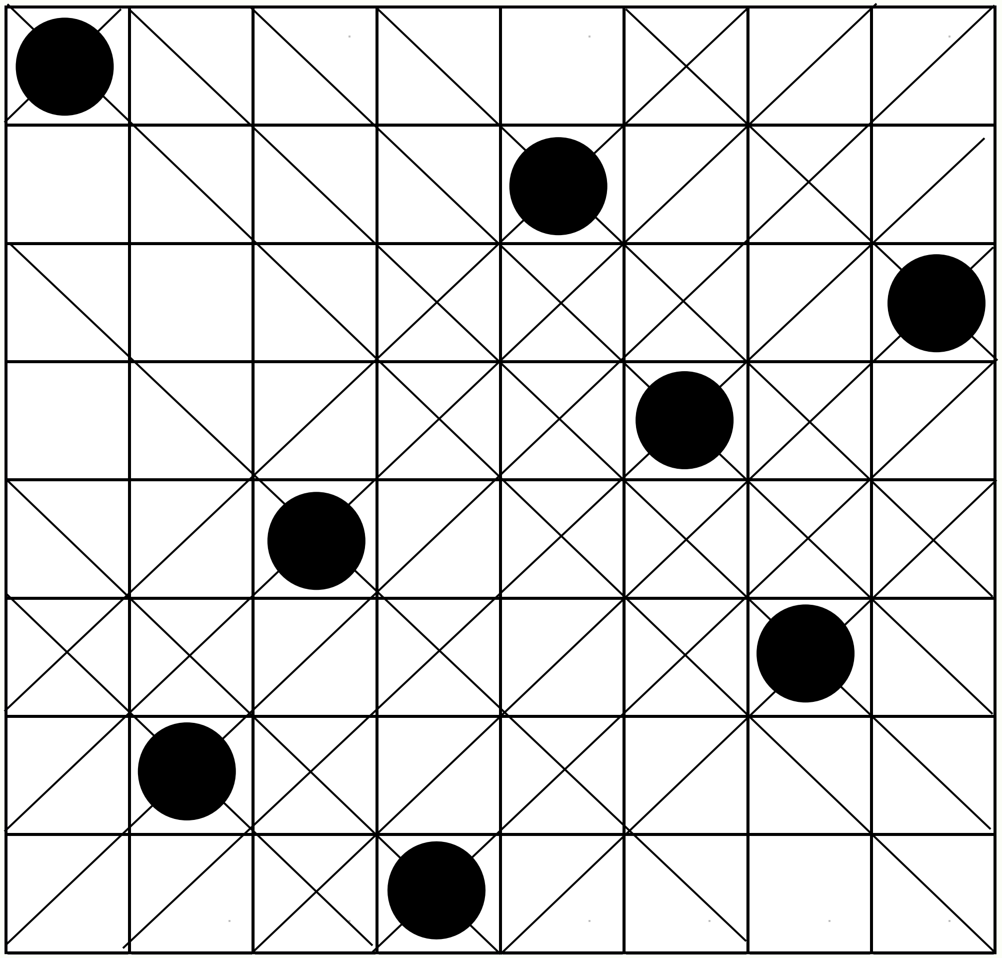
$Q_4 = 6$

$Q_5 = 3$

$Q_6 = 7$

$Q_7 = 2$

$Q_8 = 4$



Constraint Programming Languages

- The declarative nature of ZINC is easily illustrated with the n-queens problem:

```
int: n = 8;

array [1..n] of var 1..n: q;

include "alldifferent.mzn”;

constraint alldifferent(q);                           % rows
constraint alldifferent(i in 1..n)(q[i] + i-1);     % / diagonal
constraint alldifferent(i in 1..n)(q[i] + n-i);     % \ diagonal


solve   :: int_search( q, first_fail,indomain_min, complete)
  satisfy;

output   ["8 queens, CP version:\n"] ++
         [        if fix(q[i]) = j then "Q " else ". " endif ++
                  if j = n then "\n" else "" endif
         |        i, j in 1..n
         ];
```

Constraint Programming Languages

… which can be compared with the Comet version…

```
import cotfd;
int t0 = System.getCPUTime();

int n = 8; range S = 1..n;

Solver<CP> cp();
   var<CP>{int} q[i in S](cp,S);

solve<cp> {
   cp.post(alldifferent(q));
   cp.post(alldifferent(all(i in S) q[i] + i));
   cp.post(alldifferent(all(i in S) q[i] - i));
}
using {
   forall(i in S) by(q[i].getSize())
      tryall<cp>(v in S) cp.label(q[i],v);
}

int t1 = System.getCPUTime();
cout << q << endl;
cout << " cpu time (ms) = " << t1-t0 <<endl;
cout << " number of fails = " << cp.getNFail() << endl;
```

Constraint Programming Languages

… and the Choco version (to be done interactively in class):

```
public class n_queens {
   public static void main(String[] args) {
   int n = 24;
   Model model = new Model(n + "-queens problem");
   Solver s = model.getSolver();
// Use fail-first Heuristics
   s.setSearch(minDomLBSearch(queens));

   IntVar[] queens = model.intVarArray("Q", n, 1, n, false);
   IntVar[] diag1 = new IntVar[n];
   IntVar[] diag2 = new IntVar[n];
   for(int i = 0 ; i < n; i++){
      diag1[i] = q[i].sub(i).intVar();
      diag2[i] = q[i].add(i).intVar();}
   m.post(
      m.allDifferent(q),
      m.allDifferent(diag1),
      m.allDifferent(diag2));

// Solve and show statistics
   Solution solution = s.findSolution();
   System.out.println(solution.toString());
   model.getSolver().printStatistics();
}}
```

Introduction to Choco

- Before addressing concepts and definitions we will informally see how these features are addressed in the constraint programming language **Choco**.
- Choco is an Object Oriented language, implemented as a set of libraries of JAVA, with special classes and methods to deal with Constraint Programming.
- To install Choco, download the Choco Solver from the website
  - http://www.choco-solver.org
- You will get a zip file (4.10.1.zip) which contains the following files:
  - **choco-solver-4.10.0.jar:** A ready-to-use jar file including dependencies; it provides tools to declare a Model, the vari- ables, the constraints, the search strategies, etc. In a few words, it enables modeling and solving CP problems.
  - **apidocs-4.10.0.zip**: Javadoc of Choco-4.10.0

Introduction to Choco

- In **Choco**, a CSP (Constraint Satisfaction Problem) is typically solved in **CP** with a program with the following structure

```
import libraries;
// declare the variables
// post the constraints
// non deterministic search
// show results
```

- Any Choco program requires a model. Model is a class with methods to associate variables and constraints as well as nondeterministic search.

- To declare it the Model library must be imported;

```
import org.chocosolver.solver.Model;
Model model = new Model(n + "-queens problem");
```

Introduction to Choco

- Variables are objects, declared by identifying their
  - Name (for reporting results)
  - Type
  - Domain

- We will be mostly concerned with Finite Domain (FD) variables, whose type is **IntVar**, and have a domain that restricts the values that can appear in a solution of the problem.

- Typically the domain is defined as a range of integers, as in

```
// Variable taking its value in [1, 3] (the value is 1, 2 or 3)
IntVar v1 = model.intVar("v1", 1, 3);
```

- Alternatively, the domain can be a set of integers

```
// Variable taking its value in {1, 3} (the value is 1 or 3)
IntVar v2 = model.intVar("v2", new int[]{1, 3});
```

Introduction to Choco

- Variables may also be grouped together in arrays, specifying the size of the arrays and the bounds of the individual elements, as in

```
IntVar[] dst = model.intVarArray("D", n_elements, 0, 9);
```

- Variables may also be grouped together in matrices, specifying the size of the arrays and the bounds of the individual elements, as

```
IntVar[][] pos = model.intVarMatrix("T", nrows, ncols, 0, 9);
```

- To declare the variables, individually or in arrays the variable library must be imported

```
import org.chocosolver.solver.variables.IntVar;
```

Introduction to Choco

- Many types of constraints are defined in the language as primitives. They belong to the class ***constraint*** and are declared with post method of the solver.

- The most common constraints are arithmetic constraints, imposing a relation (==, !=, >, >=, <, <=) on arithmetic expressions built over CP and basic variables and values with the arithmetic operators +, -, *, /.

- Simple Relational constraint with up to 3 arguments can be posted with the arithm method, as in

```
model.arithm(res, "=", op1, "+", op2).post();
```

- Constraint involving expressions with more than three arguments must be specified with a “cumbersome” syntax, as seen before

```
// c1 + n + r = e + 10 c2
c1.add(n).add(r).eq(e.add(c2.mul(10))).post();
```

Introduction to Choco

- As a library of Java Choco inherits all its control structures (**IF**, **FOR**, **WHILE**) that can be used to specify the constraints.

- For example, to impose all variables in a vector to be different one may use:

```
Model model = new Model(n + "-queens problem");
IntVar[] q = model.intVarArray("Q", n, 1, n, false);

for(int i  = 0; i < n; i++){
   for(int j  = i+1; j < n; j++){
      model.arithm(q[i], ”!=",q[j]).post();
   }
}
```

… although the same effects can be achieved with the alldifferent constraint

```
model.allDifferent(q).post()
```

Introduction to Choco

- Other useful constraints are not easy to decompose into simpler arithmetic and logical constraints.

- Even when they are, there are some specialised algorithms that achieve better propagation.

- These are usually known as Global Constraints, and **Choco** supports a number of those that have been proposed in the literature:
  - Element
  - **Alldifferent**
  - Cardinality
  - Knapsack
  - Circuit
  - Sequence
  - Stretch
  - Regular
  - Cumulative

Introduction to Choco

- Nondeterministic search is specified in **CHOCO** with a solver, associated to the model previously defined, and asdequately imported.

```
import org.chocosolver.solver.Solver;
Solver s = model.getSolver();
s.solve()
// Use fail-first Heuristics
s.setSearch(minDomLBSearch(queens));
```

- By default, a non-deterministic search is imposed, where alternative values for the value of the variables are explored in some order and backtracked if they lead to failure. This is achieved by method solve(), as in

```
s.solve()
```

- Some predefined heuristics can be specified to direct the search. For example the first-fail heuristics can be specified as

```
import static org.chocosolver.solver.search.strategy.Search.minDomLBSearch;
.....
s.setSearch(minDomLBSearch(queens));
```

Introduction to Choco

- Solutions can also be obtained with a special class, Solution, that includes all the declared decision variables, as in.

```
import org.chocosolver.solver.Solution;

Solver s = model.getSolver();
Solution solution = s.findSolution();
```

- A solution, if any, may be displayed converting it to a string, as in

```
if(solution != null){
   System.out.println(solution.toString());}
```

- Alternatively, the value of some decision variable **v**, may be shown, by obtaining its value (with method getValue(), and converting it to a string

```
    System.out.print(String.valueOf(v.getValue())
```

Introduction to Choco

- One solution that satisfies the problem is obtained with method solve(). When more than the first solution is sought, then the **solve()** method may be used in a while cycle, as in

```
while (s.solve()){
    s.findSolution();
    System.out.println(solution.toString());}
```

- Notice that the solution must be reported **inside** the cycle (after leaving the the cycle cycle the solver has no solution!).

- A similar technique should be adopted when aiming the optimization of some variable v, after setting the model objective

```
model.setObjective(Model.MINIMIZE, v);
s.setSearch(minDomLBSearch(vars));
....
while (s.solve()){
    s.findSolution();
    System.out.println(solution.toString());}
```

Introduction to Choco

- We finish this brief introduction to **CHOCO** with some useful tips to measure performance in program execution. The simplest way to obtain a number of performance indicators of program execution, is with the statistics method:

```
model.getSolver().printStatistics();
```

- obtaining a complete statistics of execution, as in

```
- Model[24-queens problem] features:
   Variables : 96
   Constraints : 51
   Building time : 0.064s
   User-defined search strategy : yes
   Complementary search strategy : no
- Complete search - 1 solution found.
   Model[24-queens problem]
   Solutions: 1
   Building time : 0.064s
   Resolution time : 0.054s
   Nodes: 24 (445.9 n/s)
   Backtracks: 6
   Backjumps: 0
   Fails: 4
   Restarts: 0
```

Introduction to Choco

- Individual performance indicators can be obtained by specific methods of the model and the solver, namely the number of failures, backtracks and elapsed CPU time

```
Model model = new Model(n + "-queens problem");
Solver s = model.getSolver();
...
float t0 = s.getTimeCount()*1000;
...
float t1 = s.getTimeCount()*1000;

// show model name
System.out.println(model.getName());
// number of failures
System.out.println(String.valueOf(s.getFailCount()));
// number of backtracks
System.out.println(String.valueOf(s.getBackTrackCount()));
// execution time
System.out.println(String.valueOf(t1-t0));
```