

# Search and Optimisation

---

## - An overview

- Examples of decision (making) problems
- Complexity and need for search
- Declarative Modelling with Constraints
  - Constraint Programming
- Finite and Continuous Domains

# Constraint Problems: Examples

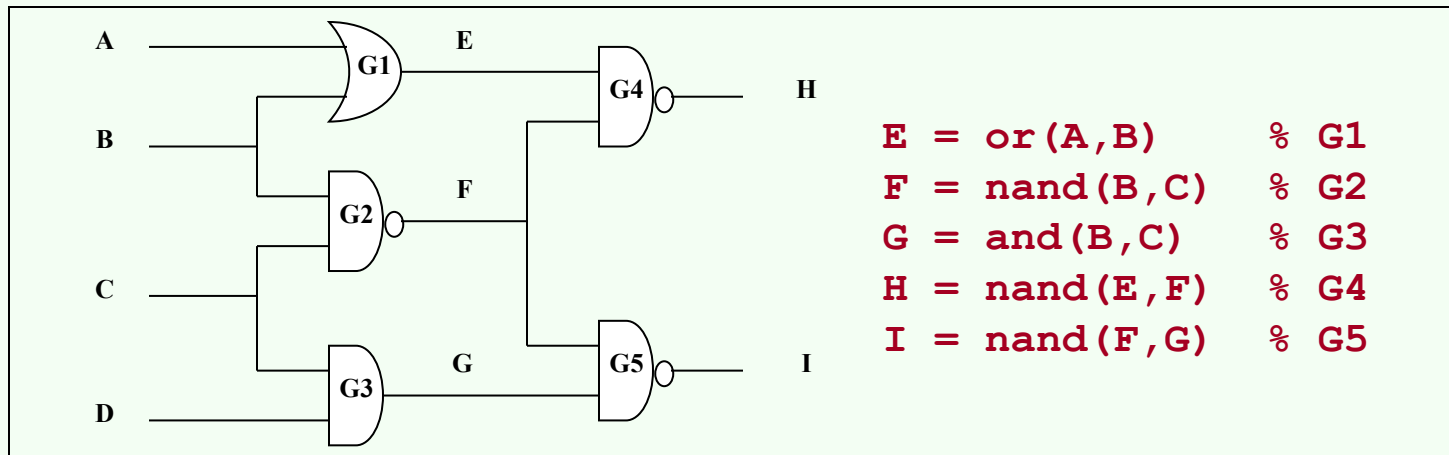
---

- 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 modeling, 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, ...)



# 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...

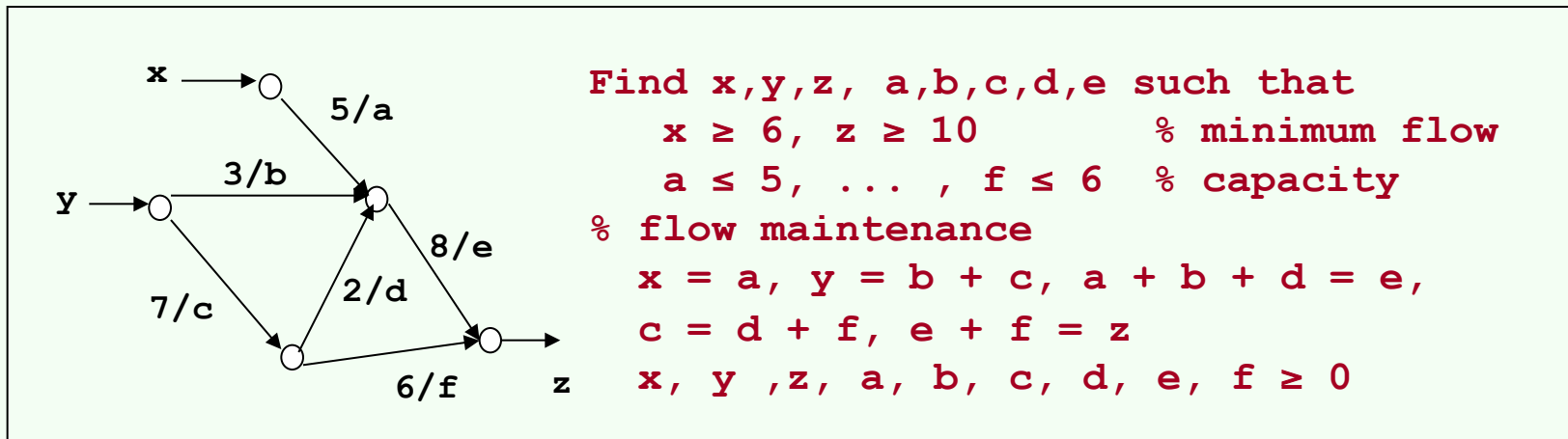
**Find  $x$ ,  $y$  and  $z$  such that**

$4x + 3y + 6z \leq 1500$	% resources used do not exceed 1500
$x + y + z \geq 300$	% production not less than 300 units
$x \leq z + 20$	% $x$ units within $z \pm 20$ units
$x \geq z - 20$	
$x, y, z \geq 0$	% non negative production

# Network Management

Goal (Example): Determine acceptable traffic on a network

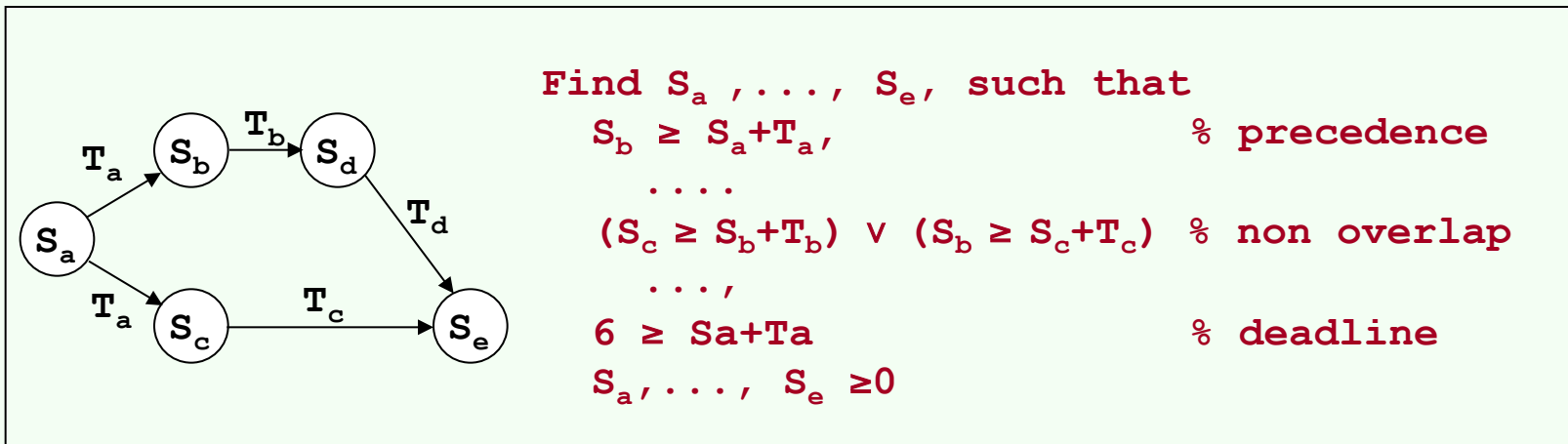
- Variables:
  - Flows in each edge
- Domain:
  - Rational/Reals (or Integers)
- Constraints:
  - Equality and Inequality (linear) constraints to model capacity limitations, flow maintenance, costs, etc...



# Scheduling

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...



# Assignment

---

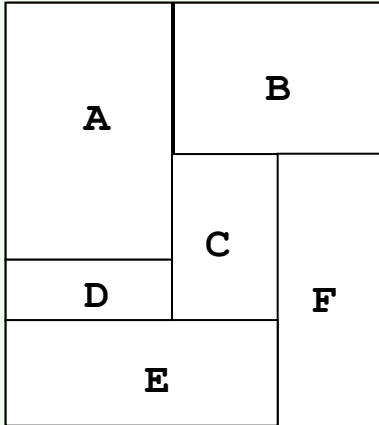
Many constraint problems can be classified as assignment problems. In general all that can be stated is that these problems follow a general CSP goal :

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)

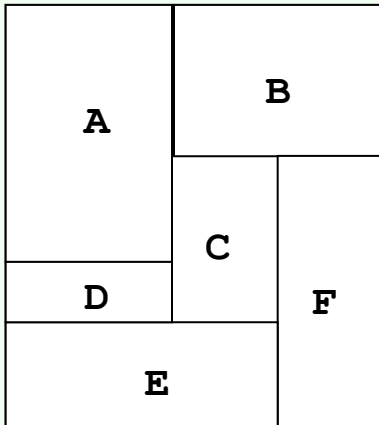
Assign values to  $A, \dots, F,$

s.t.  $A, B, \dots, F \in \{\text{red, blue, green}\}$

$A \neq B, A \neq C, A \neq D,$

$B \neq C, B \neq F, C \neq D, C \neq E, C \neq F$

$D \neq E, E \neq F$



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

Assign values to  $A_1, A_2, \dots, F_1, F_2$

s.t.  $A_r, A_b, A_g, \dots, F_r, F_b, F_g \in \{0,1\}$

% one and only one colour for  $A, B, \dots, F$

$A_r + A_b + A_g = 1;$

.....

% different colours for  $A$  and  $B, \dots$

$A_r + B_r \leq 1; A_b + B_b \leq 1; A_g + B_g \leq 1;$

.....



# Assignment (3)

Q1		●		
Q2				●
Q3	●			
Q4			●	

## N-queens (Finite Domains):

Assign Values to  $Q_1, \dots, Q_n \in \{1, \dots, n\}$

s.t.  $\forall_{i \neq j}$  noattack ( $Q_i, Q_j$ )

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

## Latin Squares (similar to Sudoku):

Assign Values to  $x_{11}, \dots, x_{33} \in \{1, \dots, 3\}$

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  $x_{11}, \dots, x_{33} \in \{1, \dots, 9\}$

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

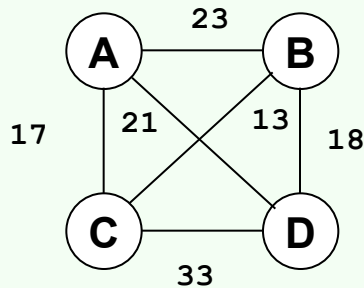
$\forall_i \forall_{j \neq i} \sum_k X_{ik} = \sum_k X_{jk} = M$  % same cols sum

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

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

# Assignment (3)

## Travelling Salesperson (Finite Domains)



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

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

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

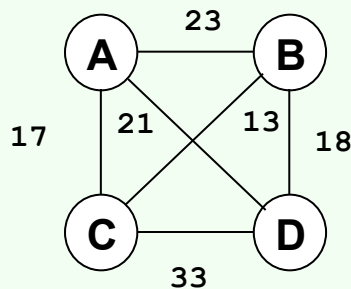
if  $A = B+1$  then  $X_A = L_{ba}$ ,

...

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  $X_{ab} \dots X_{dc} \in \{0, 1\}$

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

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

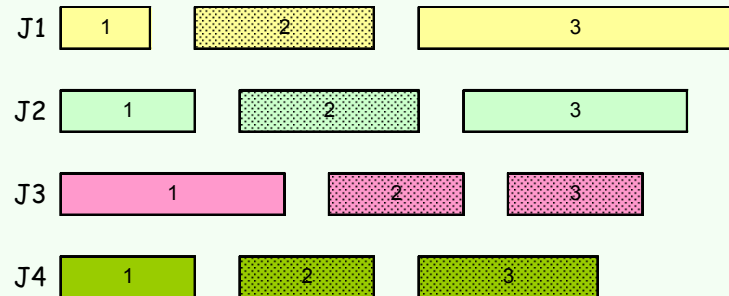
... no subcycle constraints

$\sum_a \sum_b X_{ab} L_{ab} < k$

# Mixed: Assignment and Scheduling

Goal (Example): Assign values to variables

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



## Job-Shop

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

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

% precedence within job

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

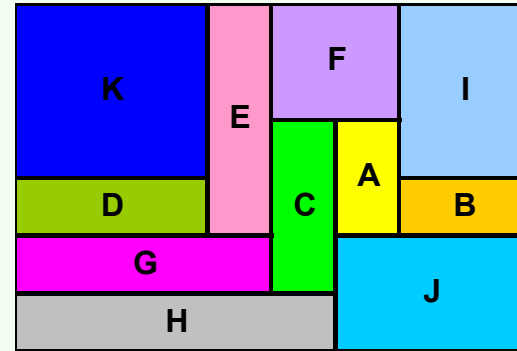
% either no-overlap or different machines

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

# Filling and Containment

Goal (Example): Assign values to variables

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



## Filling

Assign values to  $X_i \in \{1, \dots, X_{\max}\}$  % X-dimension

$Y_i \in \{1, \dots, Y_{\max}\}$  % 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

$(Y_j + Ly_j \leq Y_i)$  % I in back of J

# Constraint Satisfaction Problems

---

- Formally a constraint satisfaction problem (CSP) can be regarded as a tuple  $\langle X, D, C \rangle$ , 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)
- 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  $X_{i1}, \dots, X_{ik}$  ranging over domains  $D_{i1}, \dots, D_{ik}$  is a subset of the cartesian cartesian  $D_{j1} \times \dots \times D_{jk}$ .

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

# Constraints and Optimisation Problems

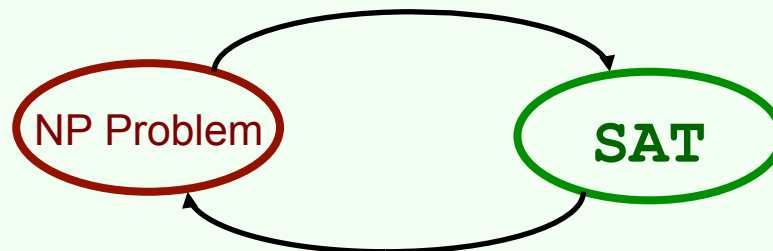
---

- 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  $\langle V, D, C, F \rangle$ , 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  $P \neq NP$  (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  $1 \mu\text{sec}$  (times in secs).

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

1 hour =  $3.6 * 10^3 \text{ sec}$

1 year =  $3.2 * 10^7 \text{ sec}$

TOUniv =  $4.7 * 10^{17} \text{ sec}$



# 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 polynomial 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.
  - In incomplete local search, the number of constraint violations in a (potential) solution can be counted, and local moves are made during search to aim at reaching one such solution with no violations.

# 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).

# 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 conflicting goals and have led to the various developments in this research and development area.

# 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)

# Declarative Programming

---

- Programming a combinatorial problem thus requires
  - the specification of the constraints of the problem
  - 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 ) and Zinc, provide more convenient data structures for modelling, maintaining a declarative approach.

# Constraint Programming by Example

---

## An 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):

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

- 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.

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$

# 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

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array}$$



# Constraint Programming by Example

**Model 1** : In Comet, this model is specified as follows

```
% // import cotfd;
enum Letters = {s,e,n,d,m,o,r,y};
range Rng = 0 .. 9;
Solver<CP> cp();
var<CP>{int} q[Letters] (cp,Rng);
solve<cp>{
  cp.post(q[s] != 0);           % No leading zeros
  cp.post(q[m] != 0);
  cp.post(alldifferent(q));    % All variables are different
                                % The sum must be correct
  cp.post(1000*q[s]+100*q[e]+10*q[n]+q[d]
          + 1000*q[m]+100*q[o]+10*q[r]+q[e]
          == 10000*q[m]+1000*q[o]+100*q[n]+10*q[e]+q[y]);
} using
  labelFF(q);
```

C4	C3	C2	C1	
	S	E	N	D
+	M	O	R	E
<hr/>				
M	O	N	E	Y

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

	<b>C4</b>	<b>C3</b>	<b>C2</b>	<b>C1</b>	
		<b>S</b>	<b>E</b>	<b>N</b>	<b>D</b>
<b>+</b>	<b>M</b>	<b>O</b>	<b>R</b>	<b>E</b>	
	<b>M</b>	<b>O</b>	<b>N</b>	<b>E</b>	<b>Y</b>

# Constraint Programming by Example

**Model 2** : This alternative model can also be expressed in Comet

```
enum Letters = {s,e,n,d,m,o,r,y};
range Rng = 0 .. 9;
Solver<CP> cp();
  var<CP>{int} q[Letters] (cp,Rng);
  var<CP>{int} c[1..4] (cp,0..1);
solve<cp>{
  cp.post(q[s] != 0);
  cp.post(q[m] != 0);
  cp.post(alldifferent(q));
  cp.post(q[d] + q[e] == q[y] +10 * c[1]);
  cp.post(q[n] + q[r] + c[1] == q[e] +10 * c[2]);
  cp.post(q[e] + q[o] + c[2] == q[n] +10 * c[3]);
  cp.post(q[s] + q[m] + c[3] == q[o] +10 * c[4]);
  cp.post(c[4] == q[m]);
} using { labelFF(q); labelFF (c) }
```

C4	C3	C2	C1		
	S	E	N	D	
	<u>+</u>	<u>M</u>	<u>O</u>	<u>R</u>	<u>E</u>
M	O	N	E	Y	

% No leading zeros

% All variables are different

% The sum must be correct

# Constraint Programming by Example

---

## Model 3 :

- Even the constraints might often be expressed in alternative ways. The constraint that all letters are different can be expressed by the primitive global constraint **alldifferent(q)**, or obtained by conjunction of the individual pairwise different (!=) constraints:

```
// cp.post(alldifferent(q));  
  
forall(i in letters, j in letters: i!=j)  
    cp.post(q[i] != q[j]);
```

- As will be seen later, when available, global constraints typically lead to much more efficient execution.

# 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 guaranteedly do not belong to any solution.
- The enumeration is typically achieved in Comet with function **label/1**, 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 Comet, that may be expressed by function **labelFF/1**.
- More sophisticated heuristics may nevertheless be programmed by the user.

# Constraint Programming - Finite Domains

---

- 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 Comet in several non-trivial examples

# 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 lose 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 (mid November)
  - Project 2 and Mini-test 2 at the end of semester (mid December)