Constraint Programming

- An overview

  - Constraint Networks and Consistency Criteria
  - Node- and Arc-consistency
  - Enforcing Algorithms and their Complexity
  - Constraint Programming by Example (Choco)
  - Constraint Programming Languages
  - An introduction to Choco

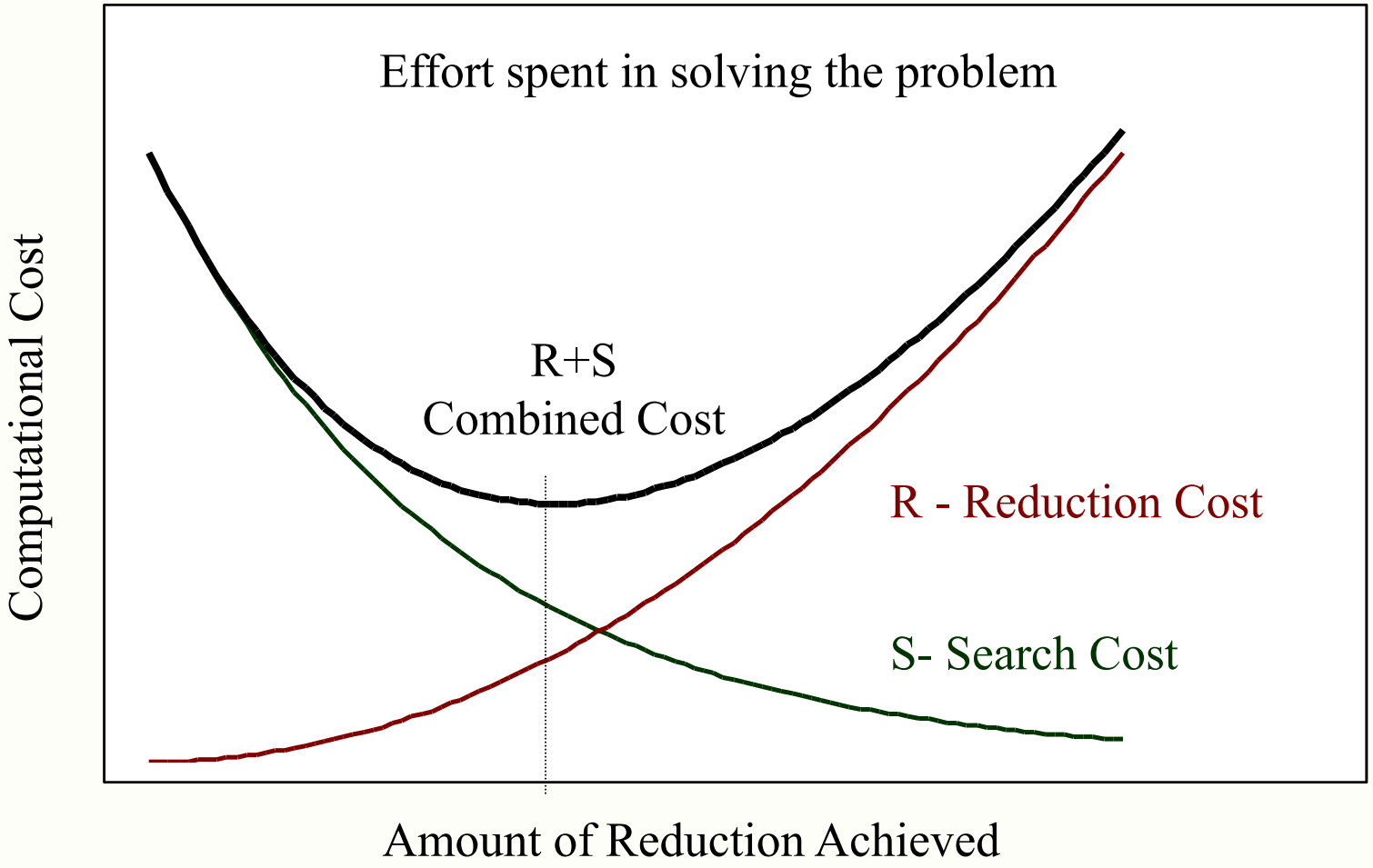
Propagation in Search

- Given a problem with **n** variables x_1 to x_n, and assuming a lexicographical variable/value heuristics, the execution model follows the following pattern to incrementally extend partial solutions until a complete solution is obtained:

```
Declaration of Variables and Domains,
Specification of Constraints,
   propagation,  % initial reduction of the problem
% Labelling of Variables,
   label(x1),    % variable/value selection with backtraking
   propagation,  % reduction of problem {x2 ... xn}
   label(x2),
   propagation,  % reduction of problem {x3 ... xn}
      ...
   label(xn-1)
   propagation,   % reduction of problem {xn}
  label(xn)
```

Complexity of Search

- Qualitatively, this process may be represented by means of the following picture



Propagation: Consistency Criteria

- Consistency criteria enable to establish redundant values (i.e. those that do not appear in any solution) in the variables' domains, requiring no prior knowledge on the set of problem solutions.

- Hence, procedures that maintain these criteria during the "propagation" phases, will eliminate redundant values and so decrease the search space on the variables yet to be enumerated.

- For constraint satisfaction problems with binary constraints, the most usual criteria are, in increasingly complexity order,

  - **Node Consistency**
  - **Arc Consistency**
  - **Path Consistency**
  - **i-Consistency**

Node - Consistency

Definition (**Node Consistency**):

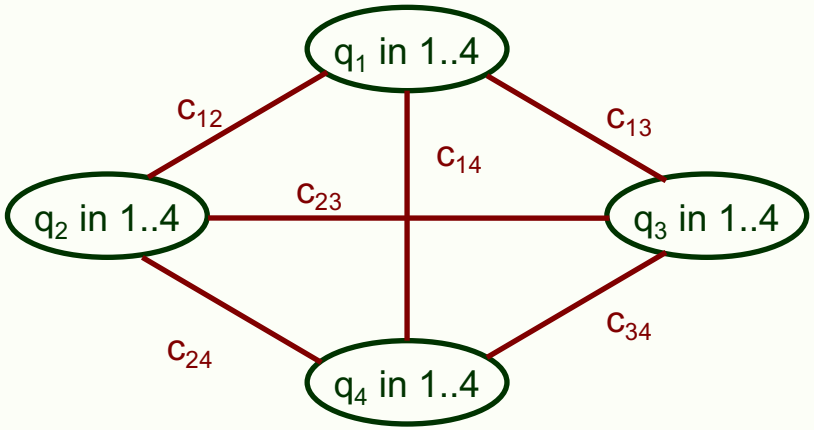
- A constraint satisfaction problem is **node-consistent** if no value in the domain of its variables violates the **unary** constraints.

- This criterion may seem both obvious and useless. After all, who would specify a domain that violates the unary constraints ?!
- However, this criterion must be regarded within the context of the execution model that incrementally completes partial solutions.
  - Constraints that were not unary in the initial problem become so when one (or more) variables are enumerated.

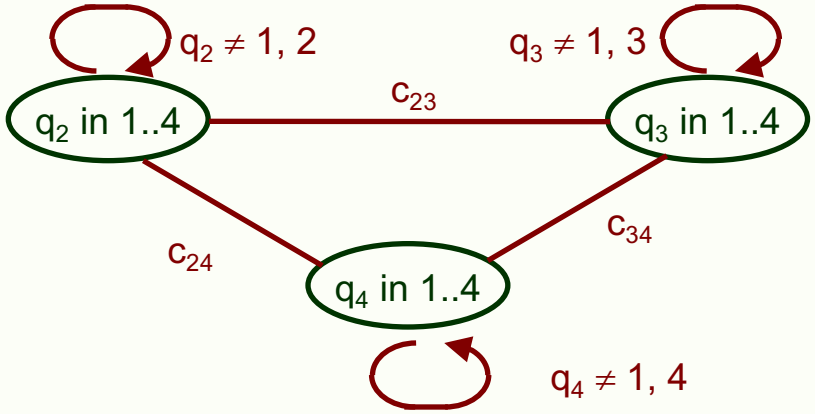
Node - Consistency

Example:

- After the initial posting of the constraints, the constraint network model at the right represents the 4-queens problem.

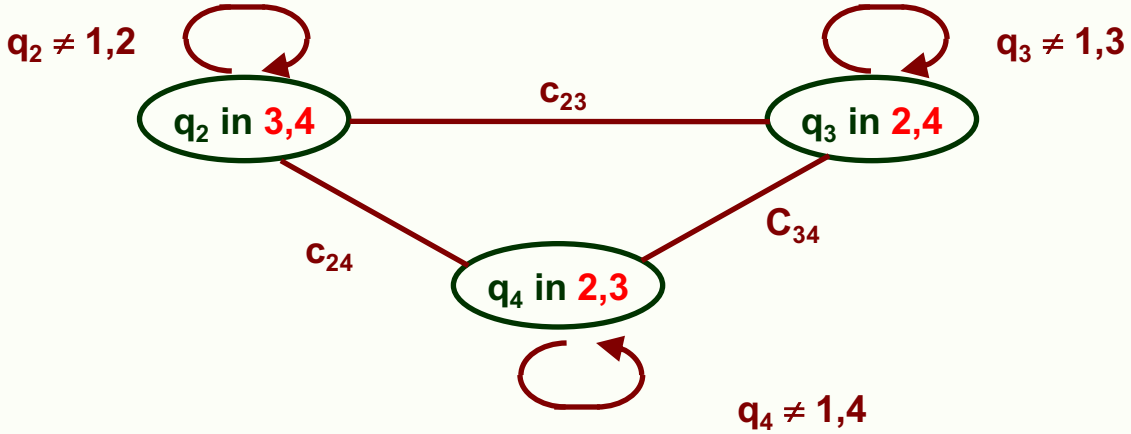


- After enumeration of variable q_1, i.e. q_1=1, constraints c_{12}, c_{13} and c_{14} become **unary** !!



Node - Consistency

- An algorithm that maintains node consistency should remove from the domains of the “future” variables the appropriate values.



- Maintaining node consistency thus achieves the following domain reduction.

●			
1	1		
1		1	
1			1

$q_2 \neq 1,2$

$q_3 \neq 1,3$

$q_4 \neq 1,4$

Enforcing Node-Consistency

Definition (**Node Consistency**):

- A constraint satisfaction problem is **node-consistent** if no value in the domain of its variables violates the **unary** constraints.

Enforcing node consistency: Algorithm NC-1

- Node-consistency can be enforced by the very simple algorithm shown below:

```
procedure NC-1(V, D, C);
   for x in V
      for v in Dx do
         for Cx in {C: Vars(Cx) = {x}} do
            if not satisfy(x-v, Cx) then
              Dx <- Dx \ {v}
         end for
      end for
   end for
end procedure
```

Enforcing Node-Consistency

Space Complexity of NC-1: O(nd).

- Assuming **n** variables in the problem, each with **d** values in its domain, and assuming that the variable's domains are represented by extension, a space **nd** is required to keep explicitely the domains of the variables.
- Algorithm **NC-1** does not require additional space, so its space complexity is **O(nd)**.

Time Complexity of NC-1: O(nd).

- Assuming **n** variables in the problem, each with **d** values in its domain, and taking into account that each value is evaluated one single time, it is easy to conclude that algorithm NC-1 has time complexity **O(nd)**.

- The low complexity, both temporal and spatial, of algorithm NC-1, makes it suitable to be used in virtual all situations by a solver, despite the low pruning power of node-consistency.

Arc - Consistency

- A more demanding and complex criterion of consistency is that of arc-consistency

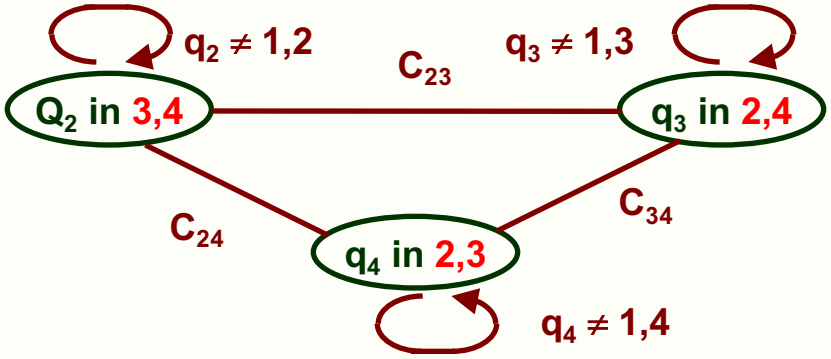
Definition (**Arc Consistency**):

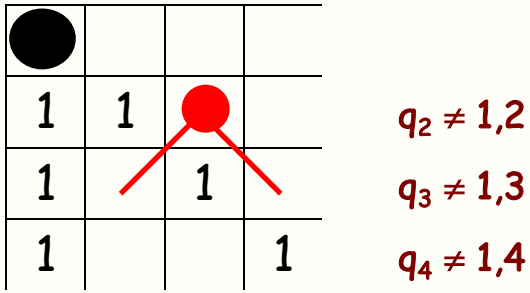
- A constraint satisfaction problem is arc-consistent if,
  - It is node-consistent; and
  - For every label x_i-v_i of every variable x_i, and for all constraints c_{ij}, defined over variables x_i and x_j, there must exist a value v_j that supports v_i, i.e. such that the compound label $\{x_i\text{-}v_i, x_j\text{-}v_j\}$ satisfies constraint c_{ij}.

Arc - Consistency

Example:

- After enumeration of variable q_1=1, and making the network node-consistent, the 4 queens problem has the following constraint network:





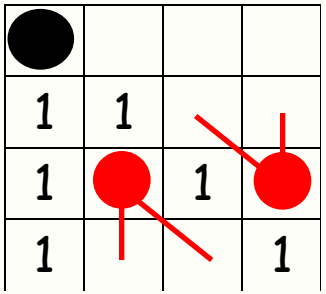
- However, label q_2-3 has **no support** in variable q_3, since neither the compound label {q_2-3 , q_3-2} nor {q_2-3 , q_3-4} will satisfy constraint C_{23}.

- Therefore, value 3 can be safely removed from the domain of q_2.

Arc - Consistency

Example (cont.):

- In fact, none (!) of the values of q3 has support in variables q2 and q4, as shown below:



$q_2 \neq 1,2$

$q_3 \neq 1,3$

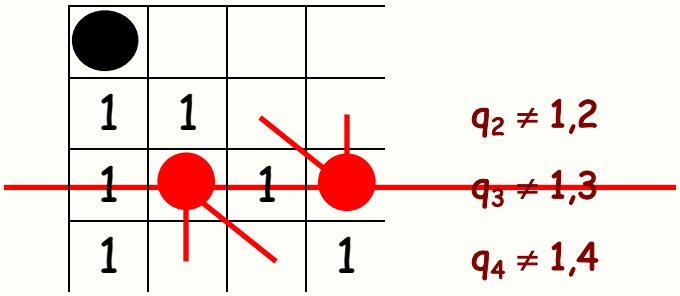
$q_4 \neq 1,4$

- Label q_3-4 has no support in variable q_2, since none of the compound labels {q_2-3, q_3-4} and {q_2-4, q_3-4} satisfy constraint c_{23}.
- Label q_3-2 has no support in variable q_4, since none of the compound labels {q_3-2, q_4-2} and {q_3-2, q_4-3} satisfy constraint c_{34}.

Arc - Consistency

Example (cont.):

- Since none of the values from the domain of q_3 has support in variables q_2 and q_4, maintenance of arc-consistency **empties** the domain of q_3!



- Hence, maintenance of arc-consistency not only prunes the domain of the variables but also antecipates the detection of unsatisfiability in variable q_3 !
- In this case, backtracking of q_1=1 may be started even before the enumeration of variable q_2.
- Given the good trade-of between pruning power and simplicity of arc-consistency, a number of algorithms have been proposed to maintain it.

Enforcing Arc-Consistency: AC-1

Definition (**Arc Consistency**):

- A constraint satisfaction problem is arc-consistent if it is node-consistent and for every label x_i-v_i of every variable x_i, and for all constraints c_{ij}, defined over variables x_i and x_j, there must exist a value v_j that supports v_i, i.e. such that the compound label $\{x_i\text{-}v_i, x_j\text{-}v_j\}$ satisfies constraint c_{ij}.

Enforcing arc-consistency: Algorithm AC-1

- The following simple (and inefficient) algorithm enforces arc-consistency:

```
procedure AC-1(V, D, C);
   NC-1(V,D,C);          % node consistency
   Q = {a_ij | c_ij ∈ C ∨ c_ji ∈ C }; % see note
   repeat
     changed <- false;
     for a_ij in Q do
           changed <- changed or revise_dom(a_ij,V,D,C)
     end for
   until not change
end procedure
```

Enforcing Arc-Consistency: AC-1

Revise-Domain

- Algorithm AC-1 (and others) uses predicate **revise-domain** on some arc a_{ij}, that succeeds if some value is removed from the domain of variable x_i (a side-effect of the predicate).

```
predicate revise_dom(a_ij,V,D,C): Boolean;
    success <- false;
    for v in dom(x_i) do
        if ¬ ∃v_j in dom(x_j): satisfies({x_i-v,x_j-v_j},c_ij) then
            dom(x_i) <- dom(x_i) \ {v};
            success <- true;
        end if
    end for
    revise_dom <- success;
end predicate
```

Enforcing Arc-Consistency: AC-1

Space Complexity of AC-1: $O(ad^2)$

- AC-1 must maintain a queue **Q**, with maximum size **2a**. Hence the inherent spacial complexity of AC-1 is **O(a)**.
- To this space, one has to add the space required to represent the domains **O(nd)** and the constraints of the problem. Assuming **a** constraints and **d** values in each variable domain the space required is $\mathbf{O(ad^2)}$, and the total space requirement of

$$O(nd + ad^2)$$

  which dominates **O(a)**.
- For "dense" constraint networks", $\mathbf{a \approx n^2/2}$. This is then the dominant term, and the space complexity becomes

$$O(ad^2) = O(n^2d^2)$$

Enforcing Arc-Consistency: AC-1

Time Complexity of AC-1: $O(nad^3)$

- Assuming **n** variables in the problem, each with **d** values in its domain, and a total of **a** arcs, in the worst case, predicate revise_dom, checks $\mathbf{d^2}$ pairs of values.
- The number of arcs $\mathbf{a_{ij}}$ in queue **Q** is **2a** (2 directed arcs $\mathbf{a_{ij}}$ and $\mathbf{a_{ji}}$ are considered for each constraint $\mathbf{c_{ij}}$). For each value removed from one domain, revise_dom is called **2a** times.
- In the worst case, only one value from one variable is removed in each cycle, and the cycle is executed **nd** times.
- Therefore, the worst-case time complexity of AC-1 is $O(d^2 *2a*nd)$, i.e.

$$\mathbf{O(nad^3)}$$

Enforcing Arc-Consistency: AC-3

Enforcing node consistency: Algorithm AC-3

- In AC-1, whenever a value $\mathbf{v_i}$ is removed from the domain of some $\mathbf{x_i}$, all arcs are re-examined. However, only the arcs $\mathbf{a_{ki}}$ (for $k \neq i$) should be re-examined.
- This is because the removal of $\mathbf{v_i}$ may eliminate the support from some value $\mathbf{v_k}$ of some variable $\mathbf{x_k}$ for which there is a constraint $\mathbf{c_{ik}}$ (or $\mathbf{c_{ki}}$).
- Such inefficiency of AC-1 is avoided in AC-3 below

```
procedure AC-3(V, D, C);
   NC-1(V,D,C);          % node consistency
   Q = {a_ij | c_ij ∈ C ∨ c_ji ∈ C };
   while Q ≠ ∅ do
      Q = Q \ {a_ij}     % removes an element from Q
      if revise_dom(a_ij,V,D,C) then    % revised x_i
         Q = Q ∪ {a_ki | (c_ik ∈ C ∨ c_ki ∈ C )∧  k ≠ i}
      end if
   end while
end procedure
```

Enforcing Arc-Consistency: AC-3

Space Complexity of AC-3: $O(ad^2)$

- AC-3 has the same requirements than AC-1, and the same worst-case space complexity of $\mathbf{O(ad^2)} \approx \mathbf{O(n^2d^2)}$, due to the representation of constraints by extension.

Time Complexity of AC-3: $O(ad^3)$

- Each arc a_{ki} is only added to **Q** when some value $\mathbf{v_i}$ is removed from the domain of $\mathbf{x_i}$.
- In total, each of the **2a** arcs may be added to **Q** (and removed from **Q**) **d** times.
- Every time that an arc is removed, predicate revise_dom is called, to check at most $\mathbf{d^2}$ pairs of values.
- All things considered, and in contrast with AC-1, with temporal complexity $\mathbf{O(nad^3)}$, the time complexity of AC-3, in the worst case, is $\mathbf{O(2ad * d^2)}$, i.e.

$$O(ad^3)$$

Enforcing Arc-Consistency: AC-4

Counting Supports: AC-4

- Every time a value $\mathbf{v_i}$ is removed from the domain of some variable $\mathbf{x_i}$, all arcs $\mathbf{a_{ki}}$ ($k \neq i$) leading to that variable are re-examined.
- Nevertheless, only some of these arcs should be examined.
- Although the removal of vi may eliminate one support for some value $\mathbf{v_k}$ of another variable $\mathbf{x_k}$ (given constraint c_{ki}), other values in the domain of $\mathbf{x_i}$ may support the pair $\mathbf{x_k}$-$\mathbf{v_k}$!

- This idea is exploited in algorithm AC-4, that uses a number of new data-structures to **count** supporting values, which contrary to AC-3, with time complexity of $O(ad^2)$, achieves a time-complexity of

$$\mathbf{O(ad^2)}$$

- This is in fact an optimal **assymptotical worst-case complexity**, since checking all the pairs of values in all the binary constraints require $\mathbf{ad^2}$ operations.

Enforcing Arc-Consistency: AC-4

Detecting last Supports: AC-6

- Whereas AC-4 maintains a number of counters to check whether there are support for variables in the domain, the same effect can be achieved by simply maintaining one value that witnesses this support.
- Every time this witness is removed, a next witness is sought.
- Hence there is no need to maintain expensive counters, and the goal of maintaining arc-consistency can be made more efficiently.

- This is the idea exploited in algorithm AC-6, that uses “lighter” data-structures to detect **next** supporting values which, like AC-4, has a time complexity of

$$\mathbf{O(ad^2)}$$

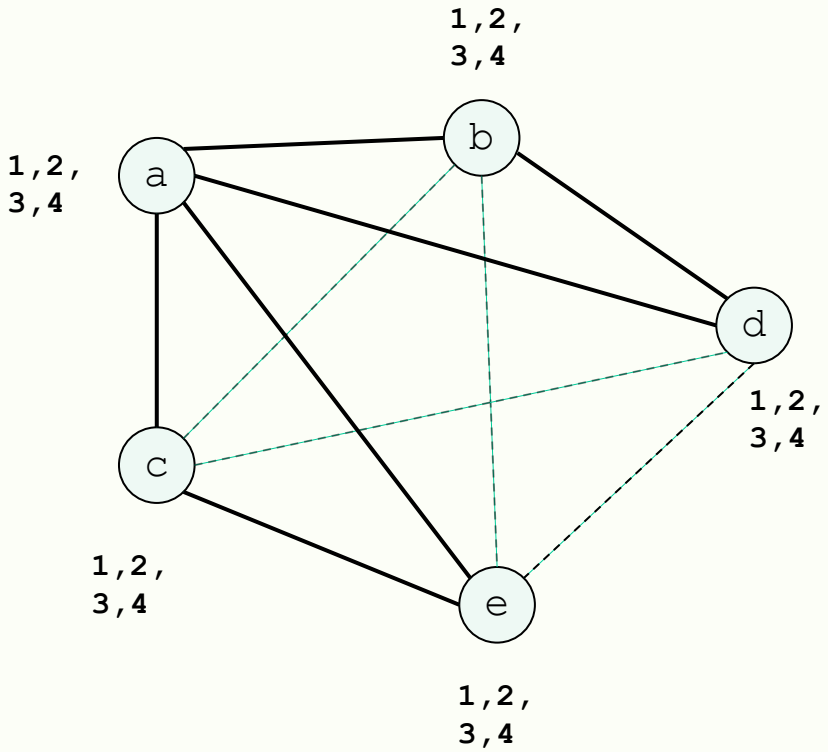
- Again this was already seen as the optimal assymptotical worst-case complexity, and AC-6 could not beat it.
- However …

Assessing Typical Complexity

Typical complexity of AC-x algorithms

- The **worst-case** time complexity that can be inferred from the algorithms that maintain arc-consistency do not give a precise idea of their average behaviour in typical situations. For such study, either one tests the algorithms in:
  - A set of “benchmarks”, i.e. problems that are supposedly representative of everyday situations (e.g. N-queens); or
  - Randomly generated instances parameterised by
    - their **size** (number of variables and cardinality of the domains) ; and
    - their **difficulty** measured by
      - density of the constraint network - % existing/ possible constraints; and
      - tightness of the constraints - % of allowed / all tuples.
- The study of these issues has led to the conclusion that constraint satisfaction problems often exhibit a phase transition, which should be taken into account in the study of the algorithms.

Randomly Generated Problem



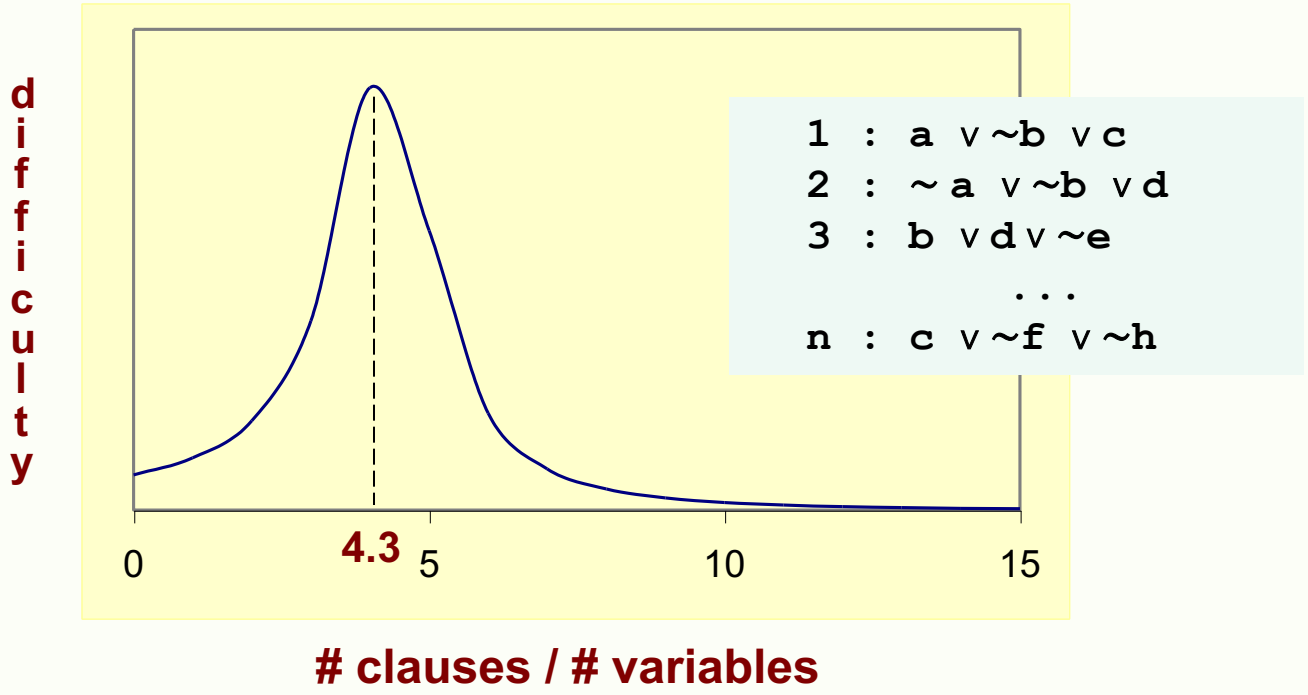
n = 5
d = 4
Density = 6 / 10 = 60%
Tightness = 10 / 16 = 62.5%

C_{ij} =

1 1	3 1
1 2	3 2
1 3	3 3
1 4	3 4
2 1	4 1
2 2	4 2
2 3	4 3
2 4	2 4

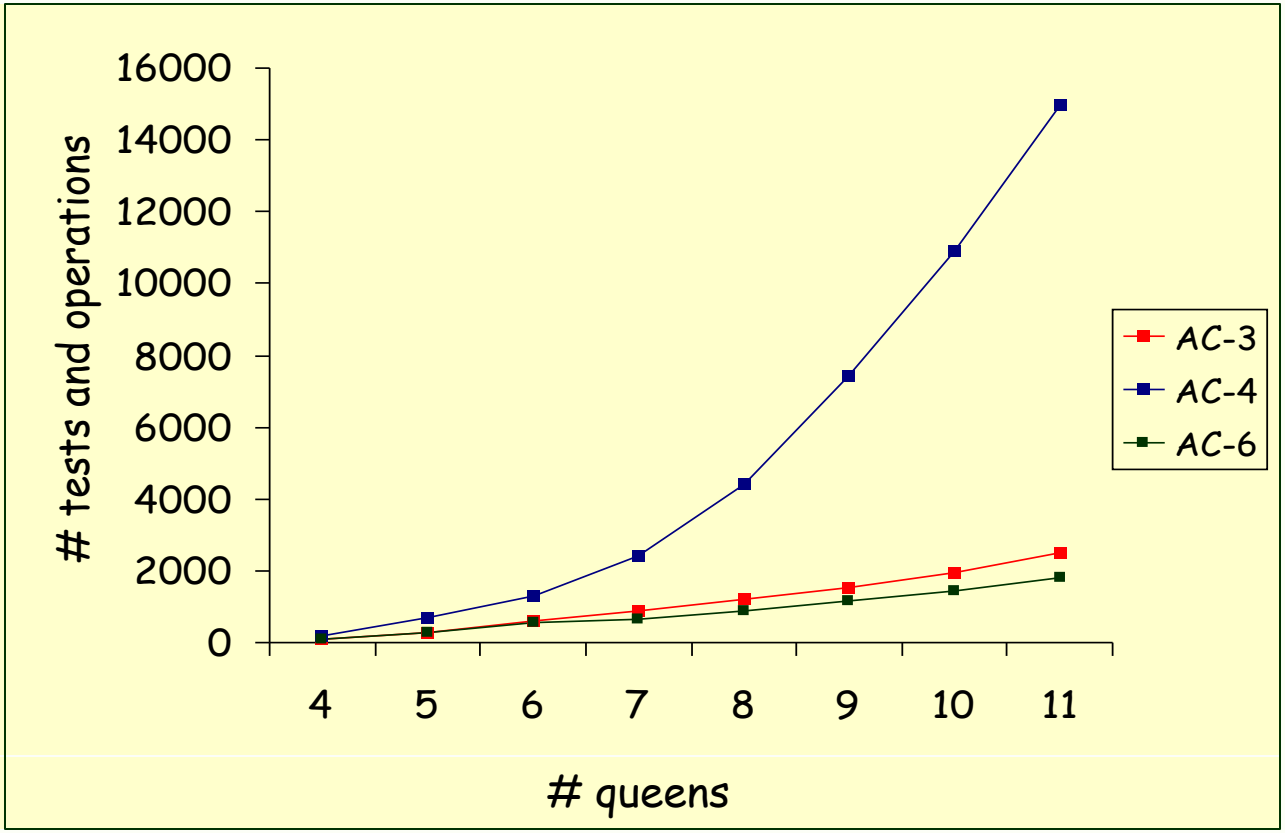
Assessing Typical Complexity: Phase Transition

- This phase transition typically contains the most difficult instances of the problem, and separates the instances that are trivially satisfied from those that are trivially insatisfiable.
- For example, in **SAT** problems, it has been found that the phase transition occurs when the ratio of clauses to variables is around 4.3.



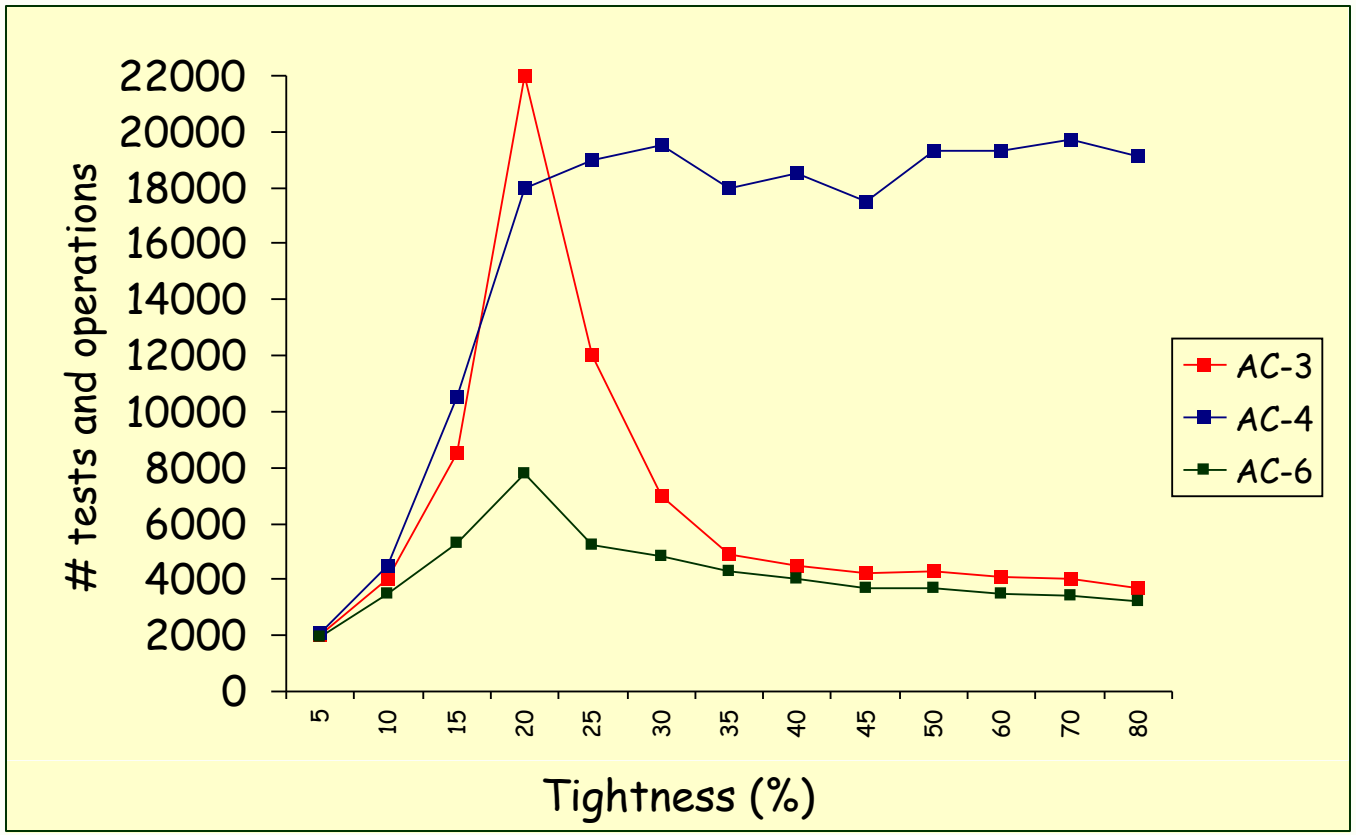
Assessing Typical Complexity

- **Typical Complexity** of algorithms AC-3, AC-4 e AC-6
- (N-queens)



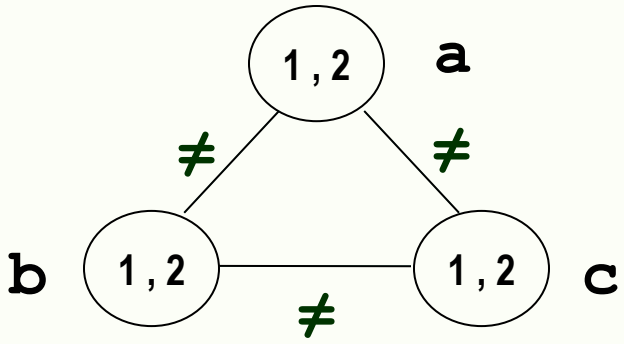
Assessing Typical Complexity

Typical Complexity of algorithms AC-3, AC-4 e AC-6
(randomly generated problems)
n = 12 variables, d= 16 values, density = 50%



Path-Consistency

- The following constraint network is obviously inconsistent:



- Nevertheless, it is arc-consistent: every binary constraint of difference (≠) is arc-consistent whenever the constraint variables have at least 2 elements in their domains.
- However, is is not path-consistent: no label {<**a-v$_a$**>, <**b-v$_b$**>} that is consistent (i.e. does not violate any constraint) can be extended to the third variable **c**.

{<a-1>, <b-2>} $\rightarrow$ c ≠ 1, 2 ; {<a-1>, <b-2>} $\rightarrow$ c ≠1, 2

- This property is captured by the notion of path-consistency (next class)

Constraint Programming

- Modelling in Choco

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:

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

  1. Choose variables to represent the unknowns
     - What are the variables
     - What values they can 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.

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

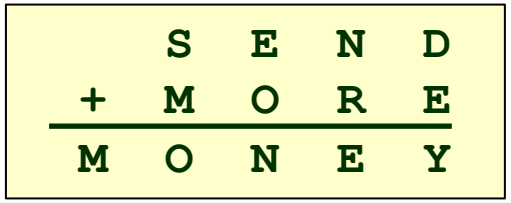
```
package choco;

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 and 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

```
// 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 and show results
```



Constraint Programming by Example

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

```
// 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 and show results
```

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

```
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 and 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 and 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

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


      // Execute and show results
      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");
      }
```

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

Constraint Programming

- Modelling Languages
  - Comet
  - Zinc
  - 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.

8-queens problem

$Q_1 = 1$

$Q_2 = 5$

$Q_3 = 8$

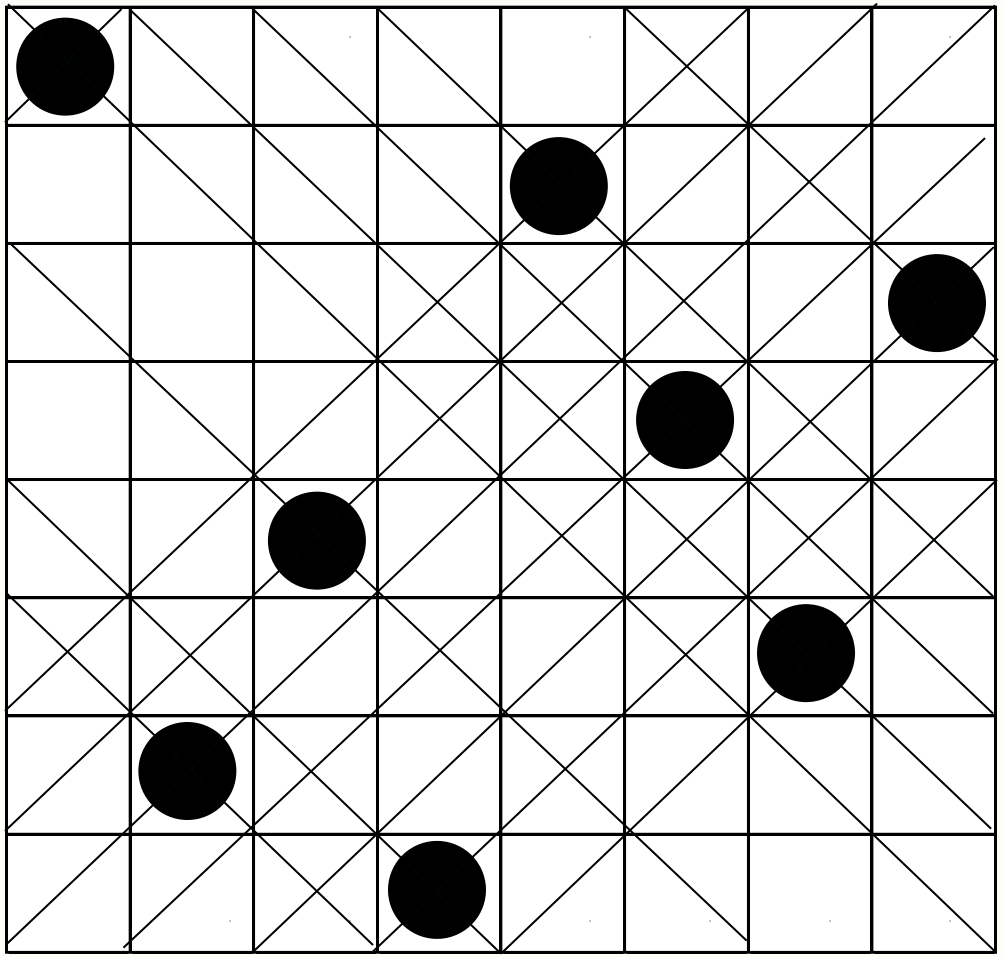
$Q_4 = 6$

$Q_5 = 3$

$Q_6 = 7$

$Q_7 = 2$

$Q_8 = 4$



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();

    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(queens),
        m.allDifferent(diag1),
        m.allDifferent(diag2));
// Use fail-first Heuristics
    s.setSearch(minDomLBSearch(queens));

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

Constraint Programming

- An Introduction to Choco

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).

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 .jar files:
  - **choco-parsers-4.10.4-jar-with-dependencies**
  - **choco-parsers-4.10.4-sources**
  - **choco-parsers-4.10.4**
  - **choco-solver-4.10.4-jar-with-dependencies**
  - **choco-solver-4.10.4-sources**
  - **choco-solver-4.10.4-sources**

  from the Choco Solver website:
  - http://www.choco-solver.org
- (or directly from:
  - https://github.com/chocoteam/choco-solver/releases/tag/4.10.4

You can find user guide and tutorials in the wiki:
- https://github.com/chocoteam/choco-solver/wiki

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));
```