Constraint Programming

- An overview

  - Global Constraints
  - Domain-consistency and specialised propagation
  - Global Constraints in Choco (1)

Types of Constraints

- So far, most of the constraints we have been dealing with are arithmetic and logical constraints. Simple constraints use the forms
  - model.arithm/3 or model.arithm/5 using the binary operators

- Arbitrary constraints can be formed (with an awkward syntax) with operators

+	.add()
-	.sub()
*	.mul()
/	.div()
%	.rem()
=	.eq()

These constructs are useful to specify constraints by intension. However, in some cases, the most convenient specification is by extension, i.e. by explicit enumeration of the accepted tuples.

Global Constraints: Table

- The table constraint is an example of a constraint given in extension. It constrains the variables in array X to take values on one of the enumerated tuples given as argument.

```
package choco;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.constraints.extension.Tuples;

public class tabling {
    public static void main(String[] args) {
        Model md = new Model("table test");
        Solver sv = md.getSolver();
        IntVar [] X = md.intVarArray("X", 3, 1, 8);
        int [][] vals = {{1,3,5},{2,4,6},{2,3,7}};
        Tuples t = new Tuples(vals, true);
        md.table(X, t).post();
        while(sv.solve())
            System.out.println(X[0].getValue() + "," + X[1].getValue() + "," + X[2].getValue());
    }
}
```

Global Constraints

- Even when complex constraints can be formed by some form of aggregation of individual constraints (possibly through reification) it is often important to specify these aggregations as a single n-ary "global" constraint, for at least two reasons
  - Simpler modelling of a problem
  - Exploitation of specialised algorithms to achieve better propagation

- The following, very common example, clarifies these issues:

The set of n variables { X_1 ... X_n } must all take different values.

- This problem can be modelled either as
  - A set of $^nC_2$ constraints of difference $X_i \neq X_j$ ($1 \leq i < j \leq n$)
  - A single **all-different({ X_1 ... X_n }) global** constraint.

- Of course, the second option is simpler, but should be more than syntactic sugar and allow a better pruning.

Global Constraints: allDifferent

Example:

X_0: 3,4,5	X_1: 3,4,5,6,7,8	X_2: 1,2,3,4,5,6,7,8,9
X_3: 1,2,3,4,5,6,7,8,9	X_4: 3,4,5	X_5: 3,4,5,6,7,8
X_6: 3,4,5,6,7,8	X_7: 1,2,3,4,5,6,7,8,9	X_8: 3,4,5

- Clearly, the decomposition of the global constraints into binary difference constraints does not lead to any pruning of the domain of the variables.
- Nevertheless, from the “pigeon hole” principle, it is easy to see that …
- variables X_0, X_4 and X_8 take values 3, 4 and 5 among themselves (3 pigeons for 3 holes), values that can be pruned from the domain of the other variables.
- But then, for similar reasons, variables X_1, X_5 and X_6 take values 6, 7 and 8, that are pruned from the domain of the other variables. The following pruning should then be “easily” achieved

X_0: 3,4,5	X_1: ~~3,4,5~~,6,7,8	X_2: 1,2,~~3,4,5,6,7,8~~,9
X_3: 1,2,~~3,4,5,6,7,8~~,9	X_4: 3,4,5	X_5: ~~3,4,5~~,6,7,8
X_6: ~~3,4,5~~,6,7,8	X_7: 1,2,~~3,4,5,6,7,8~~,9	X_8: 3,4,5

Global Constraints: allDifferent

A more realistic example: SUDOKU

- Only a few values (in green) are obtained by naïve all_diff.

the indices show a possible order in which the cuts are made

- Global all_diff performs much more pruning and fixes some variables without backtracking.
- In particular, values may be obtained by application of the "pigeon-hole" principle.

The first cuts are illustrated in the figure.

		8	4		**6**$_4$	3	5	
		6$_7$	**37**$_7$		1		8	
		3	9		8			6
2		**1**$_8$	**258**$_{10}$	9	**347**$_9$	7		
	9	**4**$_2$	**258**$_{10}$	6	**347**$_9$		1	
		5	**258**$_{10}$	1	**347**$_9$			3
6		**7**$_1$	1		2	4		
	4	**2**$_6$	6		**9**$_3$			
	8	9	**37**$_7$	**4**$_5$	5	6		

- More general domain pruning is also obtained in this way, that will eventually lead to complete solving of this board without **any** backtracking

Global Constraints: allDifferent

- How easily can such pruning be achieved?
- Given the widespread use of this and other global constraints, specialised algorithms aim at achieving a pruning close to generalised arc-consistency, at some low cost (recall that a naïve adaptation of AC-3 to GAC-3 would lead to a worst case complexity of **$O(n^4 d^{k+1})$,** clearly too costly to be of any use).
- The algorithm outlined next, maintains generalised arc-consistency, in a network of **n** variables. The algorithm (see [Regi94]), is grounded on graph theory, and uses the notion of **bipartite graph matching**.
- To begin with, a **bipartite** graph is associated to each **all-different** constraint. In such graph,
  - there are nodes of two sorts: one representing variables and the other representing values; and
  - The only arcs in the graph (bipartite) connect variables and values nodes: there is an arc between a variable node and a value node **iff** this value is in the domain of the variable

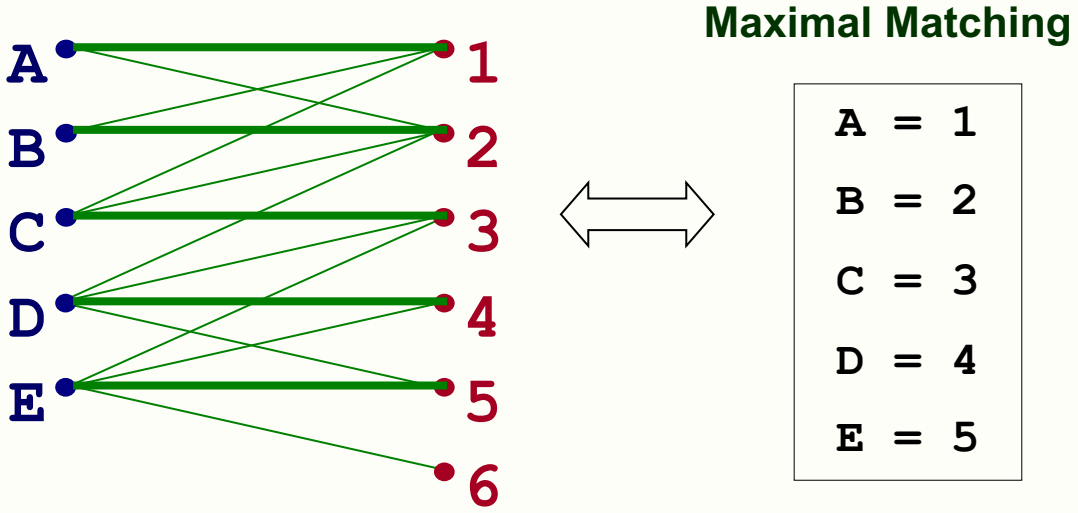
Global Constraints: allDifferent

```
A in 1..2, B in 1..2, C in 1..3, D in 2..5, E in 3..6
```

- In **polinomial time**, it is possible to eliminate, from the graph, all arcs that do not correspond to possible assignments of the variables.

Key Ideas:

- A matching, corresponds to a subset of arcs that link some variable nodes to value nodes, different variables being connected to different values.
- A **maximal matching** is a matching that includes all the variable nodes.



- Any solution of the all_diff constraint corresponds to **one and only one** maximal matching.

Global Constraints: allDifferent

- The propagation (domain filtering) is done according to the following principles:

  1. If an arc does not belong to any **maximal matching**, then it does not belong to any all_diff solution.
  2. Once determined some maximal matching, it is possible to determine whether an arc belongs or not to any maximal matching.
  3. This is because, given a maximal matching, an arc belongs to any maximal matching iff it belongs:

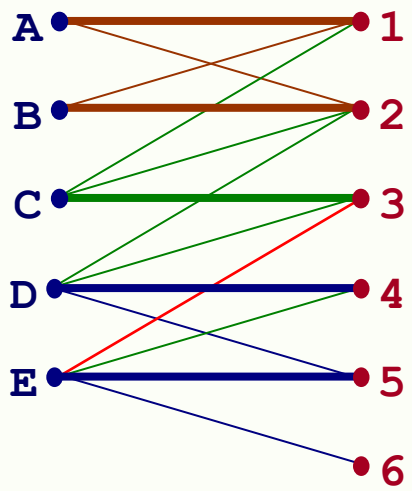
     a) To an **alternating cycle**; or

     b) To an **even alternating path**, starting at a free node.

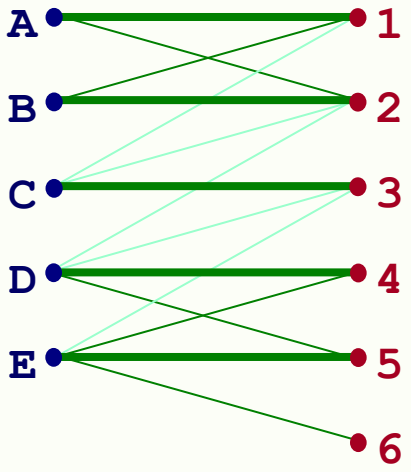
Global Constraints: allDifferent

Example:

- **6** is a free node;
- **6-E-5-D-4** is an **even alternating path**, alternating arcs included in the MM (E-5, D-4) and excluded (D-5, E-6);
- **A-1-B-2-A** is an **alternating cycle**;
- **E-3** does not belong to any **alternating cycle**
- **E-3** does not belong to any **even alternating path** starting in a free node (6)
- **E-3** may be **filtered out**!



Elimination of other arcs by similar reasoning leads to the **pruned** bipartite graph



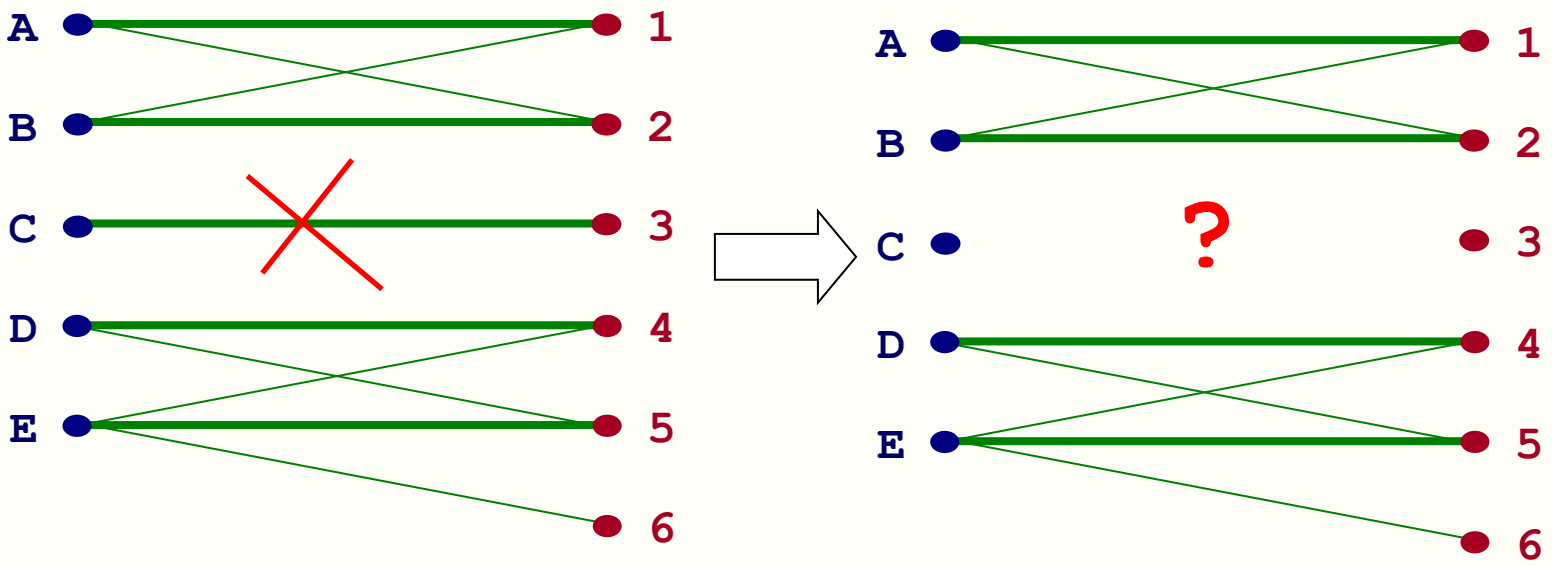
Global Constraints: allDifferent

- Upon elimination of some labels (arcs), possibly due to other constraints, the all_diff constraint propagates such pruning, incrementally.

There are 3 situations to consider:

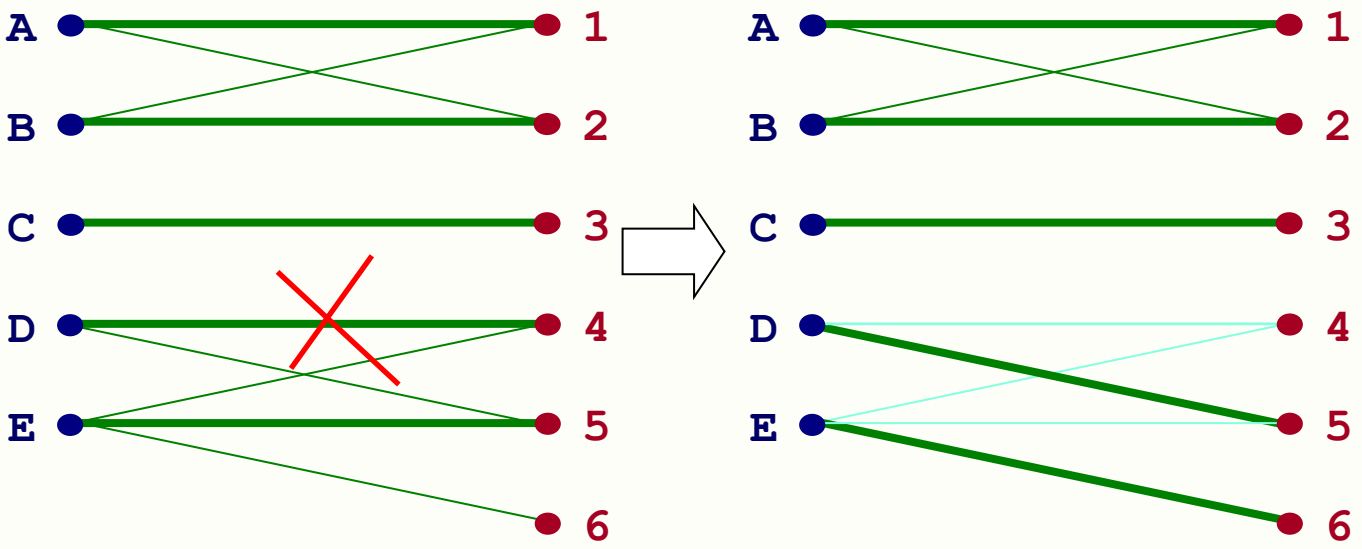
1. Elimination of a vital arc (the only arc connecting a variable node with a value node):

    - The constraint cannot be satisfied.



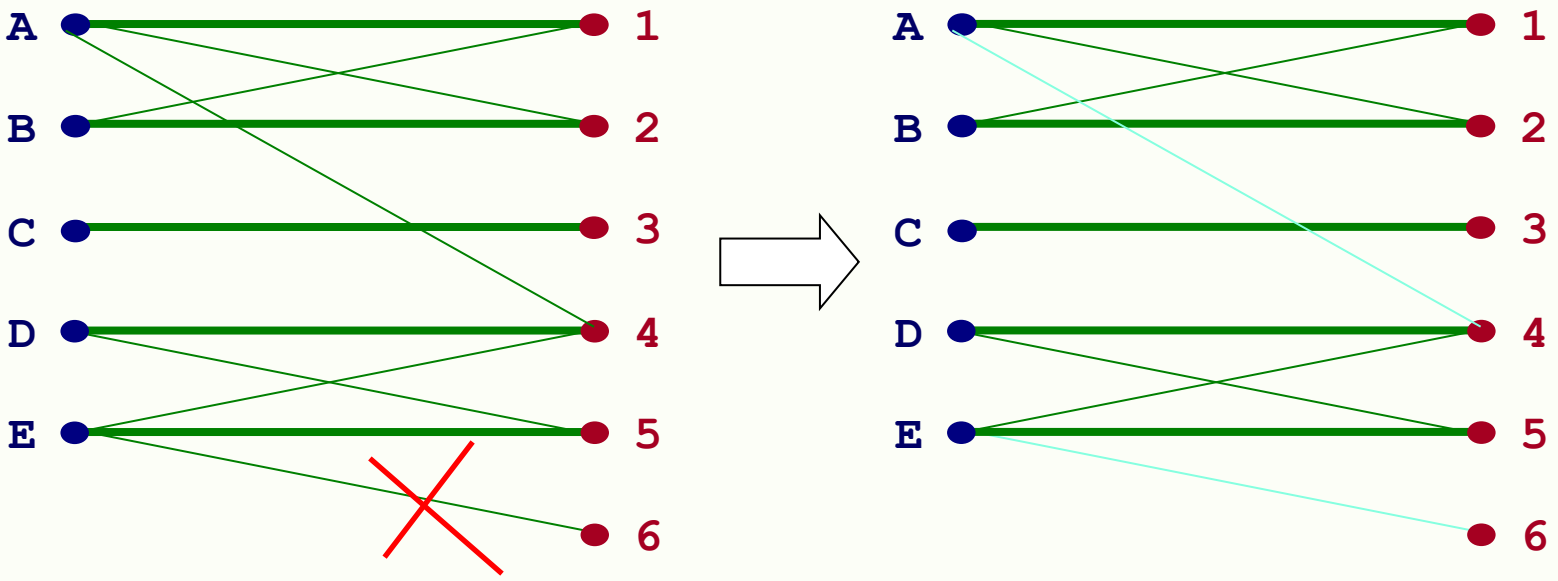
Global Constraints: allDifferent

2. Elimination of a non-vital arc which is a member to the maximal matching
   - Determine a new maximal matching and restart from there.
   - A new maximal matching includes arcs **D-5** and **E-6**. In this matching, arcs **E-4 and E-5** do not belong to even alternating paths or alternating cycle and should be pruned



Global Constraints: allDifferent

3. Elimination of a **non-vital arc which is not a member** to the maximal matching
   - Eliminate the arcs not belonging anymore to an alternating cycle or path.
   - Arc **E-4** was only kept because of the even alternating path **6-E-5-D-4-A.** Once arc **E-6** disappears, so does arc **A-4.**



Global Constraints: allDifferent

Time Complexity: $O(dn^{3/2})$.

Assuming **n** variables, each of which with **d** values, and where **D** is the cardinality of the union of all domains,

1. It is possible to obtain a maximal matching with an algorithm of time complexity **$O(dn^{3/2})$.**
2. Arcs that do not belong to any maximal matching may be removed with time complexity **O(dn+n+D).**
3. Taking into account these results, we obtain complexity of **$O(dn+n+D+dn^{3/2})$.** Since **D < dn**, the total time complexity of the algorithm is dominated by the last term.

Alternatives:

- Other specialised algorithms exist for this constraint, trading efficiency for pruning power. In particular, simpler algorithms exist that only impose bounds consistency on the variables.

Global Constraints: allDifferent

Alternatives:

- Puget designed the first **bounds consistency** algorithm for all-different, which is based on Hall's theorem and runs in **O(n log n)** time.
- Mehlhorn and Thiel later showed that since the matching and SCC computations of Régin's algorithm can be performed faster on convex graphs compared to general graphs, it is possible to achieve bounds consistency for all-different using the matching approach in O(n + n') time, where n' is **the** cardinality of D, plus the time required to sort the variables according to the endpoints of their domains.
- All-different is a special case of generalised cardinality constraint (gcc). Quimper et al. [54] discovered an alternative **bounds consistency** algorithm for gcc, based on the Hall interval approach, that only narrows the domains of the assignment variables. The "previous" algorithm, cf. later, narrows the domains of the assignment variables as well as the count variables, to bound consistency.
- See more in chapter 6 of the Handbook of Contraint Programming, F.Rossi and P. van Beek and T. Walsh, eds., Elsevier, 2006.

Global Constraints: allDifferent

- The previous example may be encoded as follows in Choco, in class alldiff_regin. Firstly the class is specified, together with the libraries/classes to be imported:
  - **Model**: to specify the model
  - **variables.IntVar**: to specify the (integer) variables of the model
  - **util.tools**: to allow some operations on IntVar arrays
  - **Solver**: To solve the problem
  - **Exception**: Here the solver will only propagate, which may throw exceptions

```
package choco;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.util.tools.*;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.exception.ContradictionException;

public class alldiff_regin {
   public static void main(String[] args) throws ContradictionException {
      ....
   }
}
```

Global Constraints: allDifferent

Next

- The model is decred;
- The solver is declared;
- The individual variables are specified;

```
public static void main(String[] args) .... {
   ....
   Model md = new Model("All Diff Demo");
   Solver sv = md.getSolver();
   IntVar a = md.intVar(new int[] {1,2});
   IntVar b = md.intVar(new int[] {1,2});
   IntVar c = md.intVar(1,3,true);
   IntVar d = md.intVar(2,5,true);
   IntVar e = md.intVar(3,6,true);
   ....
}
```

Global Constraints: allDifferent

Next

- The variables are then grouped together in an IntVar array X;

```
public static void main(String[] args) .... {
   ....
   Model md = new Model("All Diff Demo");
    //IntVar [] X = {a,b,c,d,e};
   IntVar [] X = {};
   X = ArrayUtils.concat(X,a);
   X = ArrayUtils.concat(X,b);
   X = ArrayUtils.concat(X,c);
   X = ArrayUtils.concat(X,d);
   X = ArrayUtils.concat(X,e);
   ....
}
```

- Note: they could be grouped together in the initial specification, as shown in comment. In this case the util.tools library needed not to be imported.

Global Constraints: allDifferent

- The allDifferent constraint is posted, with one of 4 possible levels of consistency:
  - "FC" (Forward checking) : only propagates when a variable is grounded. The same pruning of naïve decomposition;
  - "FC" (Arc Consistency) : performs the optimal pruning shown before;
  - ”BC" (Bounds Consistency) : Aims at arc consistency but only on the bounds of the variables’ domains;
  - ”DEFAULT" () : The solver “decides”.
- Finally, the solver method propagate is executed.

```
public static void main(String[] args) .... {
   ....
   String cons_level = "AC"; // "DEFAULT", "AC", "BC", "FC"
   md.allDifferent(X,cons_level).post();
   sv.propagate();
   ....
}
```

Global Constraints: allDifferent

- The results can be observed with the following snippet

```
public static void main(String[] args) .... {
   ....
     for (int i = 0; i < 5; i++){
     int v = X[i].getDomainSize()
     System.out.print(String.valueOf(i) + " (" + v + "): ");
     for (int j = 1; j < 7; j++){
       if (X[i].contains(j)) {
              System.out.print(" " + j);
       }
     }
     System.out.println();
}
```

Global Constraints: allDifferent

X_0: 3,4,5	X_1: 3,4,5,6,7,8	X_2: 1,2,3,4,5,6,7,8,9
X_3: 1,2,3,4,5,6,7,8,9	X_4: 3,4,5	X_5: 3,4,5,6,7,8
X_6: 3,4,5,6,7,8	X_7: 1,2,3,4,5,6,7,8,9	X_8: 3,4,5

- For the previous example, the following pruning is achieved:

```
md.allDifferent(V), "FC").post();
```

X_0: 3,4,5	X_1: 2,3,4,5,6,7	X_2: 1,2,3,4,5,6,7,8,9
X_3: 1,2,3,4,5,6,7,8,9	X_4: 3,4,5	X_5: 2,3,4,5,6,7
X_6: 2,3,4,5,6,7	X_7: 1,2,3,4,5,6,7,8,9	X_8: 3,4,5

- No pruning is achieved, since none of the variables is ground.

Global Constraints: allDifferent

X_0: 3,4,5	X_1: 2,3,4,5,6,7	X_2: 1,2,3,4,5,6,7,8,9
X_3: 1,2,3,4,5,6,7,8,9	X_4: 3,4,5	X_5: 2,3,4,5,6,7
X_6: 2,3,4,5,6,7	X_7: 1,2,3,4,5,6,7,8,9	X_8: 3,4,5

- For the previous example, the following pruning is achieved:

```
md.allDifferent(V), "AC").post();
```

X_0: 3,4,5	X_1: ~~3,4,5~~,6,7,8	X_2: 1,2,~~3,4,5,6,7,8~~,9
X_3: 1,2,~~3,4,5,6,7,8~~,9	X_4: 3,4,5	X_5: ~~3,4,5~~,6,7,8
X_6: ~~3,4,5~~,6,7,8	X_7: 1,2,~~3,4,5,6,7,8~~,9	X_8: 3,4,5

- As expected the full pruning discussed is achieved, as obtained with the algorithm discussed earlier.

Global Constraints: allDifferent

X_0: 3,4,5	X_1: 2,3,4,5,6,7	X_2: 1,2,3,4,5,6,7,8,9
X_3: 1,2,3,4,5,6,7,8,9	X_4: 3,4,5	X_5: 2,3,4,5,6,7
X_6: 2,3,4,5,6,7	X_7: 1,2,3,4,5,6,7,8,9	X_8: 3,4,5

- For the previous example, the following pruning is achieved:

```
md.allDifferent(V), "BC").post();
```

X_0: 3,4,5	X_1: ~~3,4,5~~,6,7,8	X_2: 1,2,3,4,5,6,7,8,9
X_3: 1,2,3,4,5,6,7,8,9	X_4: 3,4,5	X_5: ~~3,4,5~~,6,7,8
X_6: ~~3,4,5~~,6,7,8	X_7: 1,2,3,4,5,6,7,8,9	X_8: 3,4,5

- The lower bound of the domains of variables X_1, X_5 and X_6 change to 6, since all the values in the domain are wiped out as in "AC".
- However, 1 and 9 are possible values of variables X_2, X_3 and X_7. Bounds-consistency, does not "see" inside the domains, thus keeps the domains of these variables with no pruning.

Global Constraints: allDifferent

X_0: 3,4,5	X_1: 2,3,4,5,6,7	X_2: 1,2,3,4,5,6,7,8,9
X_3: 1,2,3,4,5,6,7,8,9	X_4: 3,4,5	X_5: 2,3,4,5,6,7
X_6: 2,3,4,5,6,7	X_7: 1,2,3,4,5,6,7,8,9	X_8: 3,4,5

- For the previous example, the following pruning is achieved:

```
md.allDifferent(V), "DEFAULT").post();
```

X_0: 3,4,5	X_1: ~~3,4,5~~,6,7,8	X_2: 1,2,~~3,4,5,6,7,8~~,9
X_3: 1,2,~~3,4,5,6,7,8~~,9	X_4: 3,4,5	X_5: ~~3,4,5~~,6,7,8
X_6: ~~3,4,5~~,6,7,8	X_7: 1,2,~~3,4,5,6,7,8~~,9	X_8: 3,4,5

- In this case, the solver applies the AC algorithm achieving the same pruning.
- The effect of the different types of consistency may be observed in the alldiff_regin and alldif_triples programs that are appended to the slides.
- Notice that the programs do not solve the problems, they simply specify the propagation with the solver method **propagate()**.
  - **Note**: It requires the specification of "*throws ContradictionException*".

Global Constraints: Circuit

- The previous global constraint may be regarded as imposing a certain “permutation” on the variables.

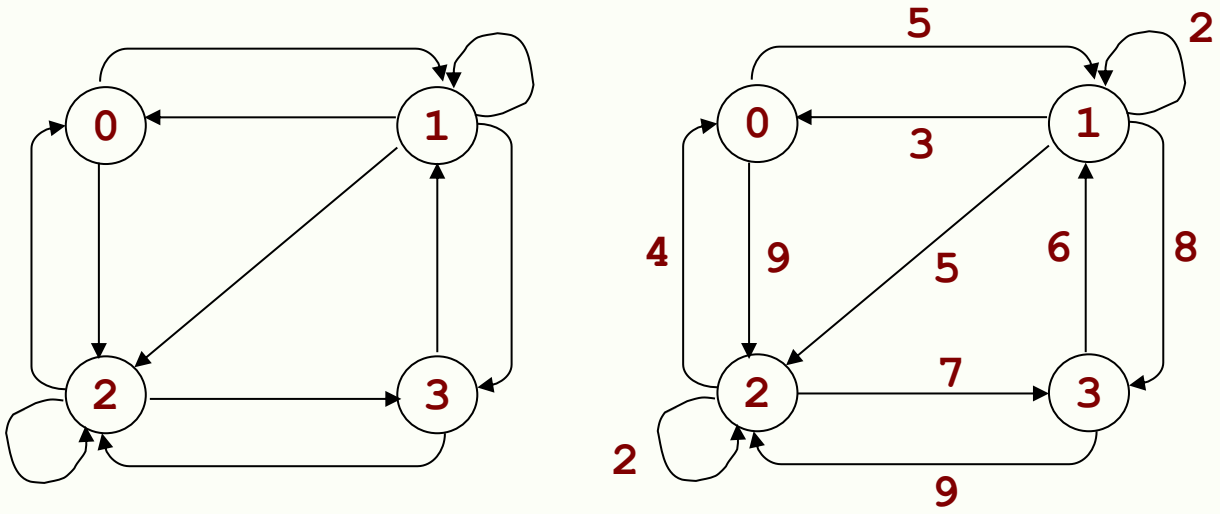
- In many problems, such permutation is not a sufficient constraint. It is necessary to impose a certain “ordering” of the variables.

- A typical situation occurs when there is a sequencing of tasks, with precedence between tasks, possibly with non-adjacency constraints between some of them.

- In these situations, in addition to the permutation of the variables, one must ensure that the ordering of the tasks makes a single cycle, i.e. there must be no sub-cycles.

Global Constraints: Circuit

- These problems may be described by means of directed graphs, whose nodes represent tasks and the directed arcs represent precedence.



- The arcs may even be labelled by “features” of the transitions, namely their ***costs***.

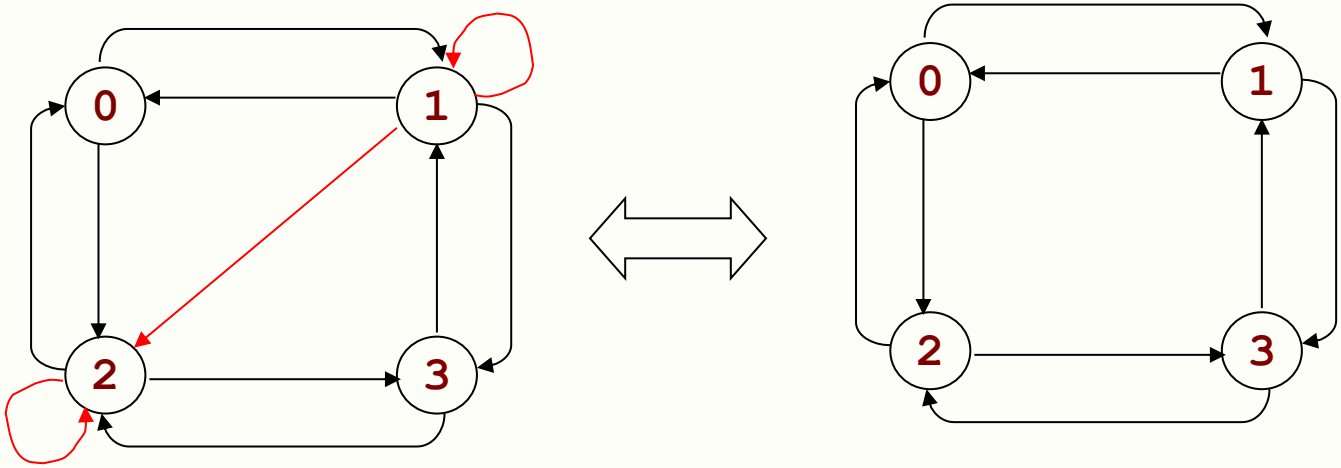
- This is a situation typical of several **TSP-like problems** (Traveling Salesman).

Global Constraints: Circuit

- **Filtering**: For these type of problems, the arcs that do not belong to any Hamiltonian circuit should be eliminated.

- In the graph, it is easy to check that the only possible circuits are
  - $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$;
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 0$

- Certain arcs (e.g. all unary such as $2 \rightarrow 2$, as well as some binary, e.g. $1 \rightarrow 2$), do not belong to any Hamiltonian circuit and can (should !) be safely pruned.



Global Constraints: Circuit

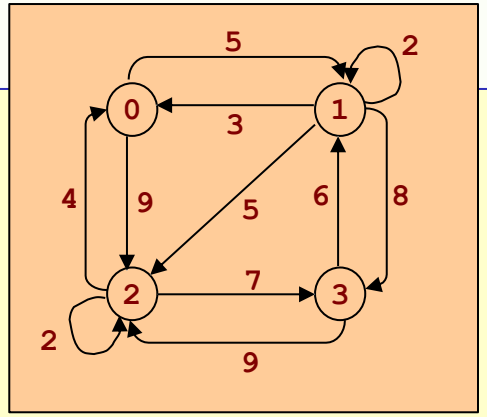
- These problems may be modelled in (at least) two different forms:
- A variable array, **`rank`**, may be used to specify the order in which the nodes are visited.
  - Hence, **`rank[i] = j`**, means that the i^{th} node is the j^{th} node to be visited.
- This modelling has some pros and cons.
- It allows many **symmetries**, in the sense that the important issue is, for each node, what is the node that follows in the tour.
  - This symmetries may be eliminated by imposing some (arbitrary) node to be the first. For example, node 0 might be made the first (and last) of the tour.
- More importantly, it is difficult to propagate the constraint. Since a node may be many ranking positions, the following positions may be quite arbitrary.
- However (and this is an advantage w.r.t. to the next model), a permutation of the values of the rank vector is always a solution.
- Nonetheless, the cons of limited propagation favour an alternative modelling technique.

Global Constraints: Circuit

- Alternatively, a variable array, **`next`**, indicates which node follows the "current" node.
  - Hence, **`next[i] = j`**, means that node **`j`** is visited immediately after node **`i`**.
- This model facilitates the propagation, since for each node a propagation algorithm may more easily eliminate nodes to be visited "next".
- However, the second model does not guarantee that a permutation of the values in the rank vector is a true "solution".
- For example, the permutation rank = [1,0,3,2] is a "pseudo-solution", since it encodes two independent sub-circuits:
  - 0->1->0; and
  - 2->3->2,
- The efficient detection of these sub-circuits is not trivial, so a global constraint, circuit, is typically available in constraint solvers.
- This is the case of Choco, as seen next.

Global Constraints: Circuit

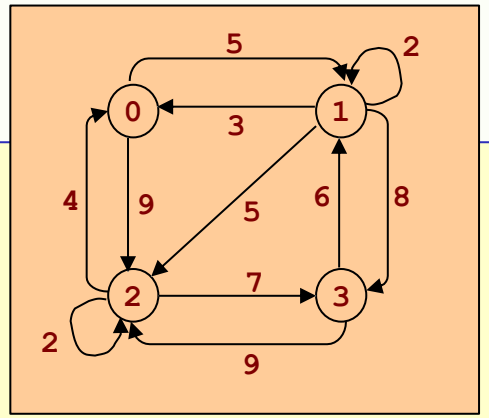
- The following program solves a TSP problem with the “next model”.
- It assumes the graph is specified by its adjacency matrix, **dist.** In this matrix, a 0 represents a non-existent arc.




```
package choco;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.Solver;
import org.chocosolver.util.tools.*;

public class tsp_0 {
   public static void main(String[] args) {
      int [][] dist = {{0,5,9,0},{3,2,5,8},{4,0,2,7},{0,6,9,0}};
      int n = dist.length;
      Model md = new Model(" TSP ");
      Solver s = md.getSolver();
      IntVar [] next = {};
      ....
   }
}
```

Global Constraints: Circuit

- First the domain of each of the variables **next[i]** is obtained from the **dist** matrix, with function **create_domain**.
- For each row **i** of the **dist** matrix, this function returns the domain of variable **dist[i]**, i.e. the nodes to which node **i** is connected to.
- Once this domain is obtained, a IntVar variable, **one**, is created to model node **i**.
- And the variable is appended to the IntVar Array **next**.

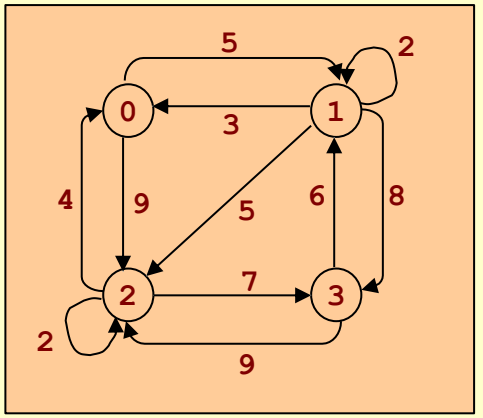



```
      ....
      for (int i = 0; i < n; i++){
         int [] dom = create_domain(dist[i]);
         IntVar one = md.intVar(dom);
         next = ArrayUtils.concat(next,one);
      }
      ....
   }
}
```

Global Constraints: Circuit

- Function create_domain creates a domain by appending the values j, for which the input vector d as a non-null value.

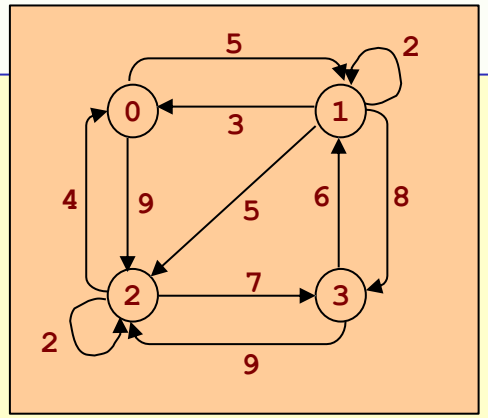
```
public static int [] create_domain(int [] d){
   int [] dom = {};
   int n = d.length;
   for (int i = 0; i < n; i++){
      if (d[i] > 0){
         int m = dom.length;
         int [] dom_x = new int[m + 1];
         for (int j = 0; j < m; j++){
                dom_x [j] = dom[j];
         }
         dom_x[m] = i;
         dom = dom_x;
      }
   return dom;
};
```




```
d0 = {0,5,9,0} -> dom = {1,2}
d1 = {3,2,5,8} -> dom = {0,1,2,3}
d2 = {4,0,2,7} -> dom = {0,2,3}
d3 = {0,6,9,0} -> dom = {1,2}
```

Global Constraints: Circuit

- Now, the circuit constraint is applied, and (in this case) all the solutions are printed.
- From all the available permutations, the global constraint only allows those representing the cycles:
  - $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$; and
  - $0 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0$;




```
      ....
      md.circuit(next).post();
      int k = 0;
      while (s.solve()){
         k = k + 1;
         System.out.print(" Solution " + k + ": ");
         for(int i = 0; i< n; i++) {
            System.out.print(" " + next[i].getValue());
         }
      System.out.println();
      }
   }
}
```

```
Solution 1:  1 3 0 2
Solution 2:  2 0 3 1
```

Global Constraints: Element

- The minimization of the cost of a circuit (e.g. the TSP: Travelling Salesman problem) can be obtained with the application of the **circuit** and the **element** constraints.
- In fact, what is needed is to consider an array of costs, where for each node I, costs[i] would denote the cost of the arc leaving it in the circuit. Hence, it should be

  `costs[i] = dist[next[i]]`
- However, an array cannot be indexed by a IntVar (i.e. dist cannot be indexed by next[i]).
- This problem is addressed by constraint element. In fact the element constraint

  `element(value, Vector, index, offset)`

  guarantees that

  `value = Vector[index-offset]`

  where **value** and **index** are IntVar variables, **Vector** is an IntVar vector and **offset** is an integer.

Global Constraints: Circuit

- The minimisation the cost of a circuit can be programmed by
  i. specifying the costs with an element constraint;
  ii. sum all the costs; and
  iii. setting the objective to be the minimization of the costs.

```
....
md.circuit(next).post();
int costUB = 100;
IntVar cost = md.intVar(1,n*costUB);
IntVar [] costs = md.intVarArray(n, 0, costUB);
for(int i = 0; i < n; i++)
   md.element(costs[i], dist[i], next[i],0).post();
md.sum(costs, "=", cost).post();
md.setObjective(Model.MINIMIZE, cost);
int k = 0;
while (s.solve()){
   k = k + 1;
   // print solution k
}
```

```
Solution 1:  2 0 3 1 with cost 25
Solution 2:  1 3 0 2 with cost 26
```

Global Constraints: Circuit

- For larger graphs, the minimization process can be monitored to access efficiency. For example, with a graph from the “Bavaria set” (see downloads), namely bavaria20.txt the following output is obtained.

```
Solution 1:   5 12 15 19 1 11 18 0 4 2 14 8 16 6 3 7 17 13 10 9 with cost 2264 in 146.40936 ms
Solution 2:   5 12 15 19 1 11 18 0 4 2 14 8 17 16 3 7 6 13 10 9 with cost 2250 in 150.01392 ms
Solution 3:   5 12 15 19 1 11 18 0 4 2 16 8 6 14 3 7 17 13 10 9 with cost 2169 in 152.86569 ms
Solution 4:   5 2 12 19 1 11 18 0 4 15 16 8 6 14 3 7 17 13 10 9 with cost 2047 in 156.94257 ms
Solution 5:   5 2 6 19 1 11 18 0 4 12 16 8 15 14 3 7 17 13 10 9 with cost 2041 in 159.15411 ms
Solution 6:   5 2 17 19 1 11 18 0 4 12 14 8 15 16 3 7 6 13 10 9 with cost 2039 in 163.0663 ms
Solution 7:   5 2 17 19 1 11 18 0 4 12 14 8 6 16 3 7 10 13 15 9 with cost 2022 in 172.46585 ms
Solution 8:   5 2 12 19 1 11 18 0 4 17 6 8 14 16 3 7 10 13 15 9 with cost 2021 in 175.47702 ms
Solution 9:   5 2 17 19 1 11 18 0 4 12 6 8 15 16 3 7 10 13 14 9 with cost 1975 in 179.40015 ms
Solution 10:  5 2 17 19 1 11 18 0 4 12 6 8 15 10 3 7 13 16 14 9 with cost 1947 in 208.06018 ms
Solution 11:  5 17 1 19 2 11 18 0 4 12 6 8 15 10 3 7 13 16 14 9 with cost 1873 in 281.39963 ms
Solution 12:  5 19 1 17 2 11 18 0 4 14 6 8 15 10 3 7 13 16 12 9 with cost 1805 in 1074.5154 ms
Solution 13:  5 19 1 17 2 11 15 0 4 12 6 8 18 10 3 7 13 16 14 9 with cost 1796 in 6432.6704 ms
Solution 14:  5 19 1 12 2 11 10 0 4 18 13 8 15 16 3 7 17 14 6 9 with cost 1793 in 16173.714 ms
Solution 15:  7 4 1 9 8 0 10 15 11 19 13 5 18 16 3 12 17 14 6 2 with cost 1791 in 21538.502 ms
Solution 16:  7 2 4 9 8 0 10 15 11 19 13 5 18 16 3 12 17 14 6 1 with cost 1737 in 21657.334 ms

!!! No (more) solutions in 52660.45 ms
```

- Several solutions are obtained and subsequently improved. The best was obtained in 21.657 secs but was only proved optimal in 52.660 secs!

Global Constraints: Circuit

- The circuit constraint can be configured to apply different filtering algorithms. In particular the same problem can be run with the “LIGHT” configuration and achieve much better “speeds”. Instead of the previous result,

```
....
Solution 16:  7 2 4 9 8 0 10 15 11 19 13 5 18 16 3 12 17 14 6 1 with cost 1737 in 21657.334 ms

!!! No (more) solutions in 52660.45 ms
```

- we have now

```
....
Solution 16:  7 2 4 9 8 0 10 15 11 19 13 5 18 16 3 12 17 14 6 1 with cost 1737 in 9603.029 ms

!!! No (more) solutions in 27424.746 ms
```

- Thus achieving an approximate 2 fold speed-up in execution. More precisely,
  - 2.25 (21657 /9603) speed up to obtain the optimal solution; and
  - 1.92 (52660 / 27424) speed up to prove optimality the optimal solution; and

Global Constraints: Circuit

- The configure the circuit constraint, it is necessary:
  i. To import the circuit configuration class;
  ii. To declare the intended configuration (
    - One of ALL | FIRST | LIGHT | RD
  iii. To use the extended circuit constraint,
    - replacing the simple one;

```
....
import org.chocosolver.solver.constraints.nary.circuit.CircuitConf;

  ....
  CircuitConf conf = CircuitConf.ALL; // ALL, FIRST, LIGHT, RD
  md.circuit(next, 0, conf).post();
  //md.circuit(next).post();
  ....
```

Global Constraints: Global Cardinality

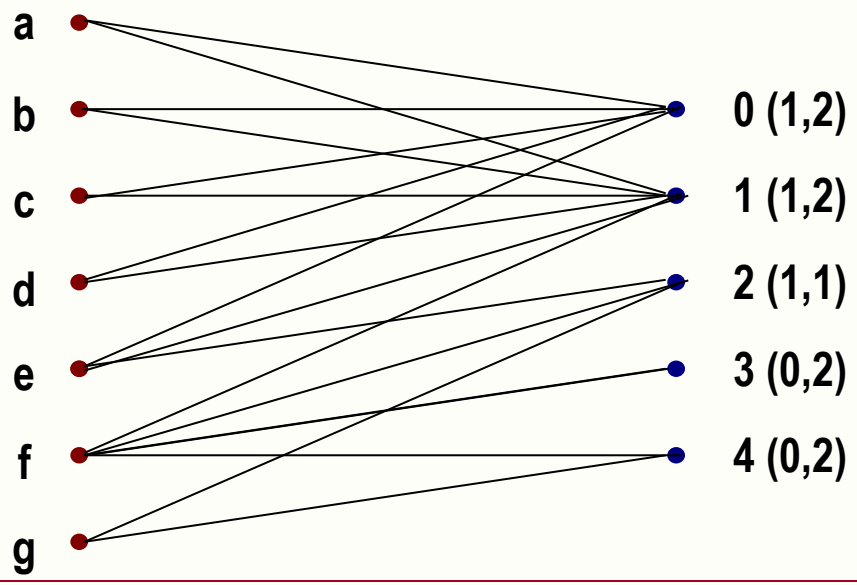
- Many scheduling and timetabling problems, have quantitative requirements such as

  in these **n** "slots" **m** must have value **v**

- This type of requirements must be modelled by **cardinality** constraints, that **count** the number of occurrences of value v in a solution of the problem. For example,

  `a,b,c,d::{0,1}, e::{0,1,2}, 6::{1,2,3,4}, 7::{2,4}`

- The workers (nurses) and their domains can be represented by a graph, where the pairs in the values represent the lower and upper bounds in a solution.



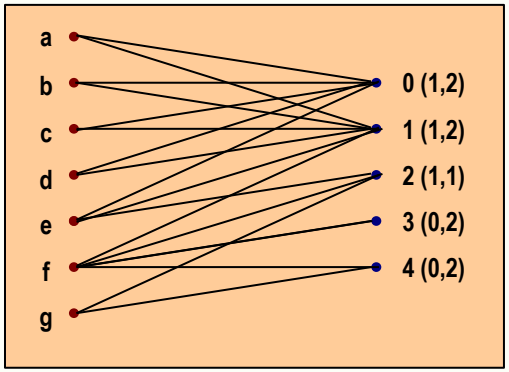
Global Constraints: Count

- Imposing lower and upper bounds on the ocurrence of values in a solution may be imposed by a global constraint: **count**.

- The **count** constraints guarantees that the number of occurrences of **any value** in an array of decision variables is constrained by a counting decision variable .

- To model the previous example, a count constraint may be posted for each of the values that the variables can take.

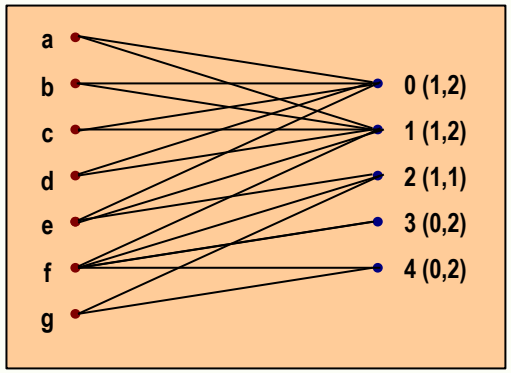
```
package choco;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.Solver;

public class nurses {
   public static void main(String[] args) {
      ....
   }
}
```



Global Constraints: Count

- First the variables **a, b, …, g,** that represent the nurses are declared within a model, and grouped together in an IntVar array, **nurses**.

```
public static void main(String[] args) {
   Model md = new Model(" TSP ");
   Solver sv = md.getSolver();
   IntVar b = md.intVar(0,1);
   IntVar c = md.intVar(0,1);
   IntVar d = md.intVar(0,1);
   IntVar e = md.intVar(0,2);
   IntVar f = md.intVar(1,4);
   IntVar g = md.intVar(new int[] {2,4});
   IntVar [] nurses = {a,b,c,d,e,f,g};
   ....
}
```



Global Constraints: Count

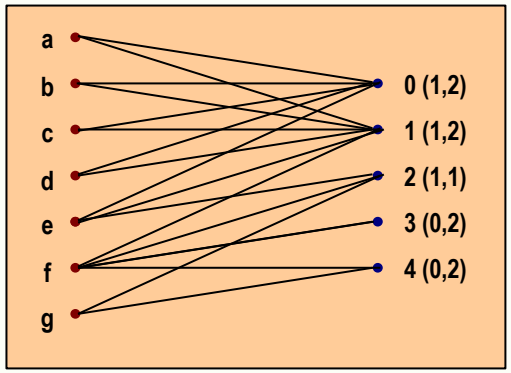
- Then, for each shift that can take the values

  0: morning; **1**: afternoon; **2**: night; **3**: reserve; **4**: holliday

- A counter is specified with a domain consistent with the time that shift can be taken by the nurses,

- and the counters are grouped together in a IntVar array, **shifts**.

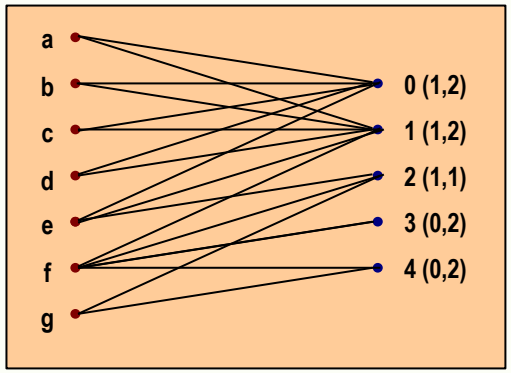
```
public static void main(String[] args) {
   ....
   IntVar c0 = md.intVar(1,2);
   IntVar c1 = md.intVar(1,2);
   IntVar c2 = md.intVar(1,1);
   IntVar c3 = md.intVar(0,5);
   IntVar c4 = md.intVar(0,5);
   IntVar [] shifts = {c0,c1,c2,c3,c4};
   ....
}
```



Global Constraints: Count

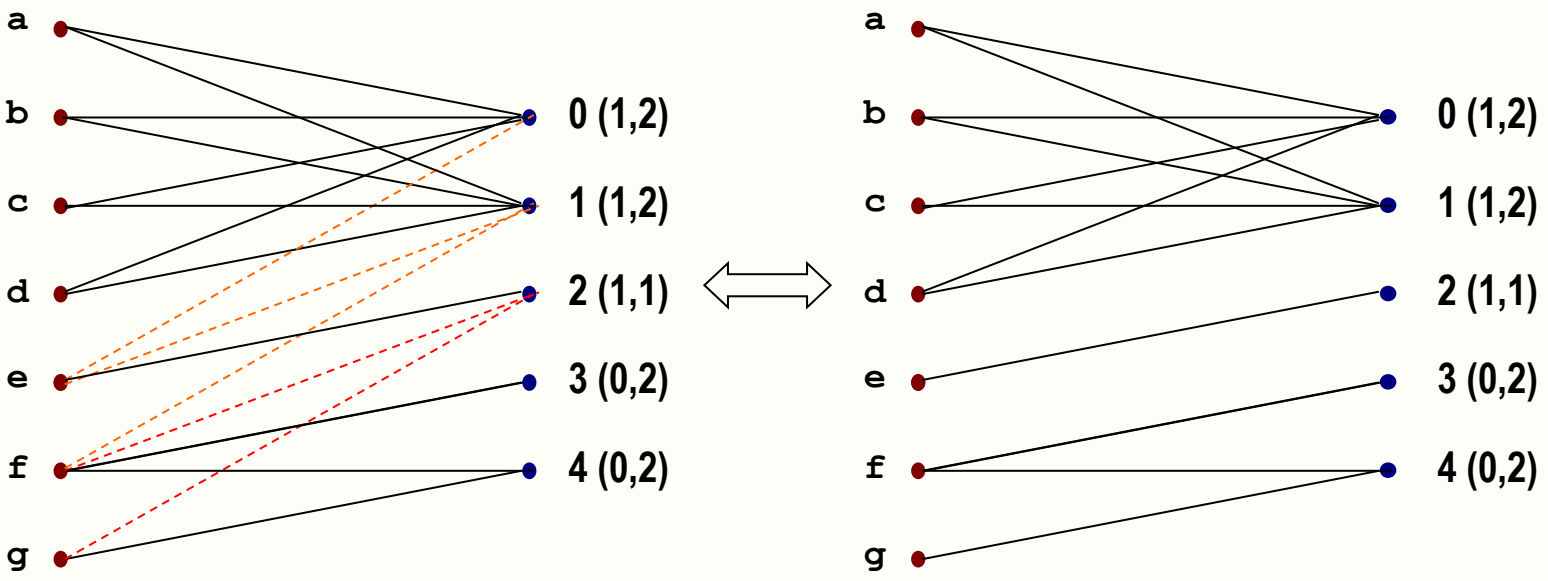
- Finally the count constraints are posted and the problem solved.

```
public static void main(String[] args) {
   ....
   for (int i = 0; i < ns; i++){
     md.count(i, nurses, shifts[i]).post();
   if (sv.solve()){
      // show solutions
   }
}
```



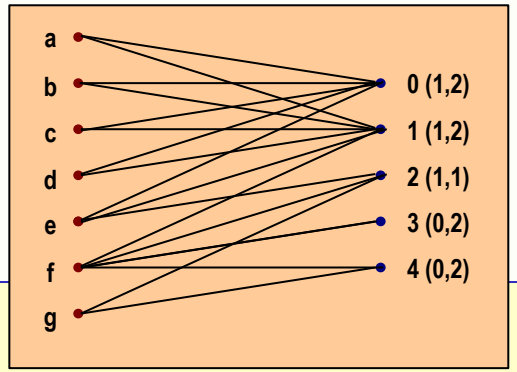
Global Constraints: GCC

- Nevertheless, the separate, or local, handling of each of these constraints, does not detect all the pruning opportunities for the variables domains, as can be seen in the problem described.
- Workers **a**, **b**, **c** and **d** may only take values 0 and 1. Since these may only be attributed to 4 people, no one else, namely **e** or **f**, may take these values 0 and 1.
- Now, worker **e** may now only take value 2. Since this must be taken by a single person, no one else (e.g. **f** or **g**) may take value 2.



Global Constraints: GCC

- This filtering, that could not be found in each constraint alone, can be obtained with an algorithm that uses analogy with results in maximum network flows.
- Such filtering is obtained in the **global cardinality constraint**, based on maximal flows in graphs.
- To use it, the individual count constraints are replaced by a single **gcc** constraint.




```
public static void main(String[] args) {
   ....
   //for (int i = 0; i < ns; i++) md.count(i, nurses, shifts[i]).post();
   new int[] values = {0,1,2,3,4},
   md.globalCardinality(nurses, values, shifts, false).post();
   ....
   }
}
```

Global Cardinality: NValue

- Sometimes there are no constraints over number of occurrences of specific values in a solution, but rather on the number of different values occurring in a solution (whatever the values may be).
- The **NValue** global constraints are a soft generalisation of the alldifferent constraint, as they constrain the number of different values in a vector of decision variables, **vars**, as well as a decision variable, **lim**, such that
  - **atLeastNValue(vars,lim)**: the different values are **at least** lim;
  - **atMostNValue(vars,lim)**: the different values are **at most** lim;
  - **NValue(vars,lim)**: the different values are **exactly** lim.
- These constraints can help solving set covering type problems. For example to find (minimal) sets of variables that cover a set of values.
- Or they may guarantee a certain diversity on the values that the variables take.
- In the extreme case that the number of different values is the same as the number of variables, the constraint is (semantically) equivalent to the alldifferent, although the latter is more “efficient” in this special case.

Global Constraints: Count

```
public class nvalues {
   public static void main(String[] args){
   Model md = new Model(" nvalues ");
      Solver sv = md.getSolver();
      IntVar a = md.intVar(new int [] {0,2});
      IntVar b = md.intVar(new int [] {1,3});
      IntVar c = md.intVar(new int [] {2,5});
      IntVar d = md.intVar(new int [] {0,3});
      IntVar e = md.intVar(new int [] {2,4});
      IntVar f = md.intVar(new int [] {1,3});
      IntVar [] vars = {a,b,c,d,e,f};
      IntVar ndifs = md.intVar(2,3);
      //md.atMostNValues(vars, ndifs, false).post();
      //md.atLeastNValues(vars, ndifs, true).post();
      md.nValues(vars, ndifs).post();
      if (sv.solve()) {
         // show solutions
      }
   }
}
```

	0	1	2	3	4	5
a	x		x			
b		x		x		
c			x			x
d	x			x		
e			x		x	
f		x		x		

```
(1,2):
2 :: 2 3 2 3 2 3
```

```
(3,4):
3 :: 0 1 2 0 2 1
```

```
(4,4):
4 :: 0 1 2 0 4 1
```

```
(6,6):
6 :: 2 1 5 0 4 3
```