Constraint Programming

- An overview

- Global Constraints
- Domain-consistency and specialised propagation
- Global Constraints in COMET
- Redundant Constraints

Types of Constraints

- So far, most of the constraints we have been dealing with are arithmetic and logical constraints, using the binary operators

  - \+ (addition)
  - \- (subtraction)
  - * (multiplication)
  - / (integer division)
  - % (modulo)
  - == (equality):

  Sums and multiplications can be specified on an arbitrary number of operands (i.e. $s = \Sigma_{i \in [1..k]} xi$ and $p = \Pi_{i \in [1..k]} xi$) by means of the **sum** and **prod** aggregate operators (and also **min** and **max**). Other more complex constraints can be formed by logically combining these constraints with the operators

  - ! (negation),
  - && (and),
  - || (or),
  - == (equality),
  - => implication)

Global Constraints: Table

- However, arbitrary constraints on an arbitrary number of variables might be not easily formed in this way. In some cases, the most convenient specification is by explicit enumeration of the accepted tuples.

- The table constraint is an example of a constraint given in extension. It constrains three variables (x[1], x[2], x[3]) to take values according to one of the enumerated triples contained in the Table<CP> object given as its parameter.

```
int possibleTriples [1..4,1..3] =
        [[2,4,2], [5,1,6], [2,1,7], [2,3,3]];
var<CP>{int} x[1..3](cp,1..7);
solve<cp> {
  Table<CP> t( all(i in 1..4) possibleTriples[i,1],
               all(i in 1..4) possibleTriples[i,2],
               all(i in 1..4) possibleTriples[i,3]);

cp.post(table(x[1],x[2],x[3],t));
}
```

- The only available consistency is **onDomains** **(generalised arc-consistency)**.

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 $^{n}C_2$ 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_1: 1,2,3	X_2: 1,2,3,4,5,6	X_3: 1,2,3,4,5,6,7,8,9
X_4: 1,2,3,4,5,6	X_5: 1,2,3	X_6: 1,2,3,4,5,6,7,8,9
X_7: 1,2,3,4,5,6,7,8,9	X_8: 1,2,3	X_9: 1,2,3,4,5,6

- 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 X1, X5 and X8 take values 1, 2 and 3 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 X4, X4 and X9 take values 4, 5 and 6, that are pruned from the domain of the other variables. The following pruning should then be “easily” achieved

X_1: 1,2,3	X_2: ~~1,2,3~~,4,5,6	X_3: ~~1,2,3,4,5,6~~,7,8,9
X_4: ~~1,2,3~~,4,5,6	X_5: 1,2,3	X_6: ~~1,2,3,4,5,6~~,7,8,9
X_7: ~~1,2,3,4,5,6~~,7,8,9	X_8: 1,2,3	X_9: ~~1,2,3~~,4,5,6

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

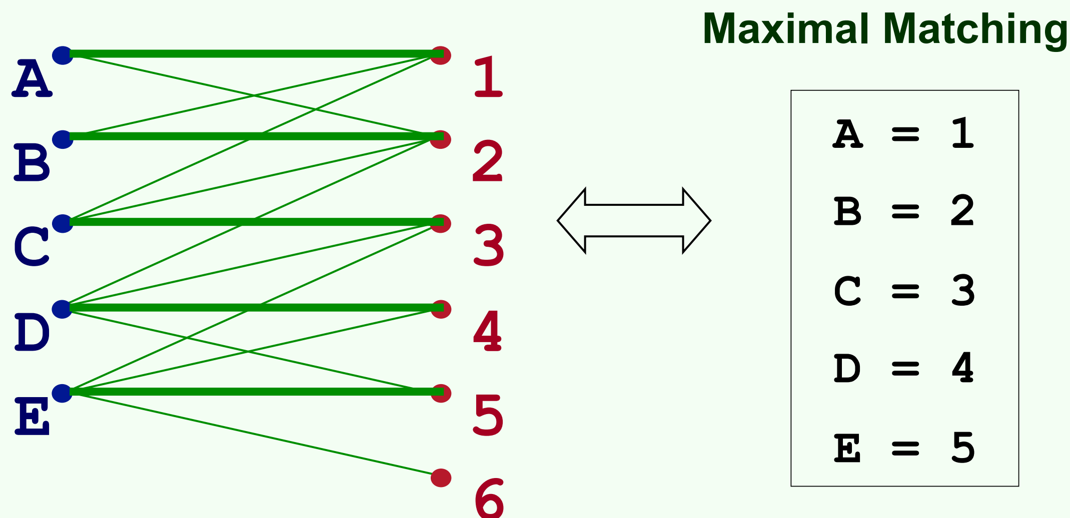
Global Constraints: allDifferent

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

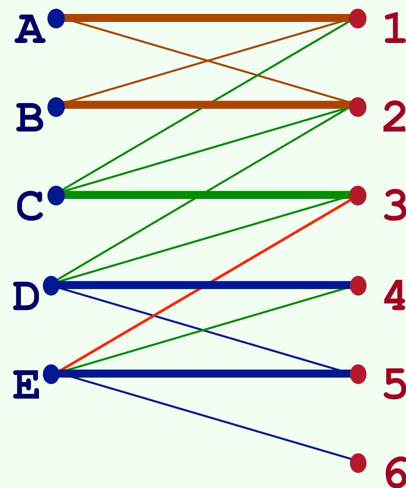
     b) To an **even alternating path**, starting at a free node.

Global Constraints: allDifferent

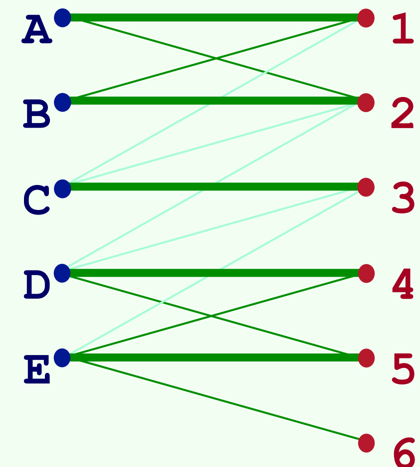
Example:

- **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



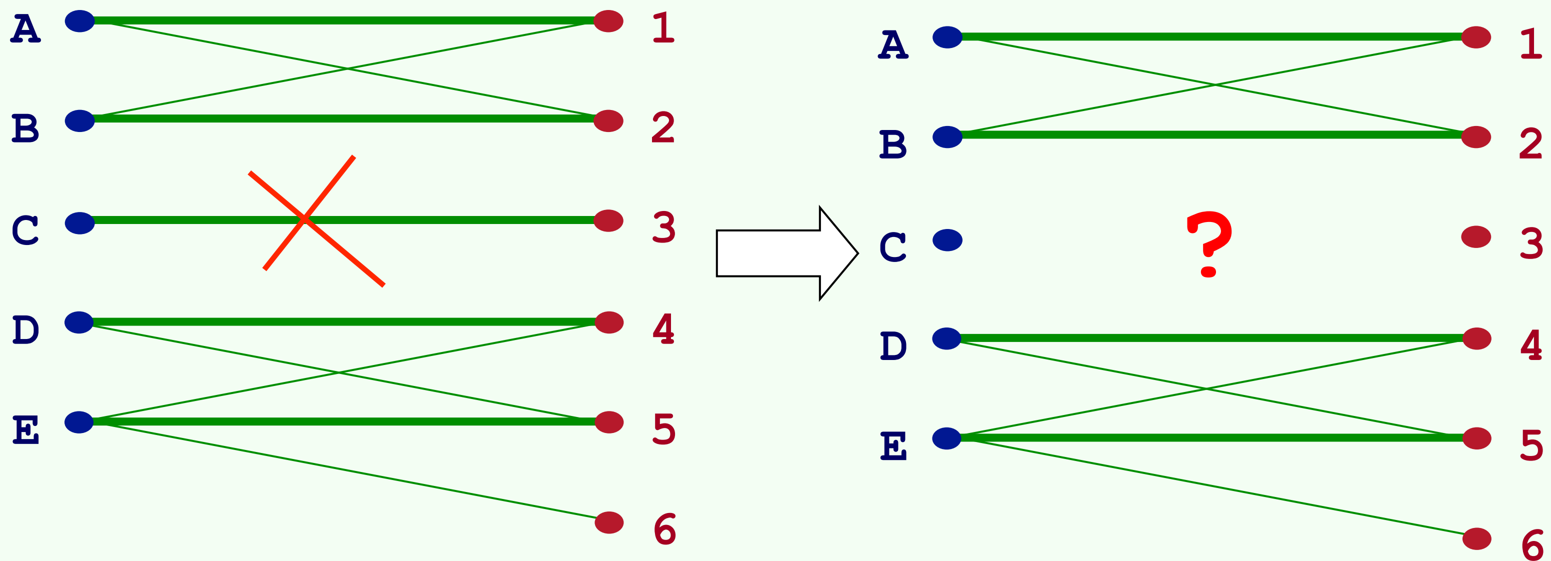
Global Constraints: allDifferent

- 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:

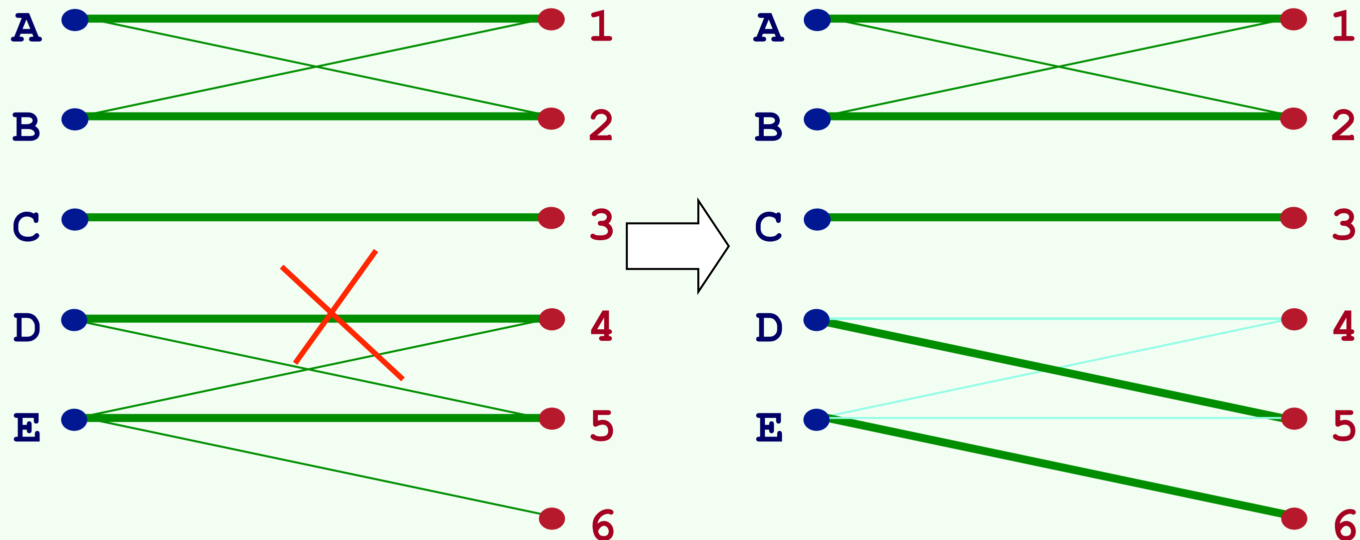
1. Elimination of a vital arc (the only arc connecting a variable node with a value node):

   - The constraint cannot be satisfied.



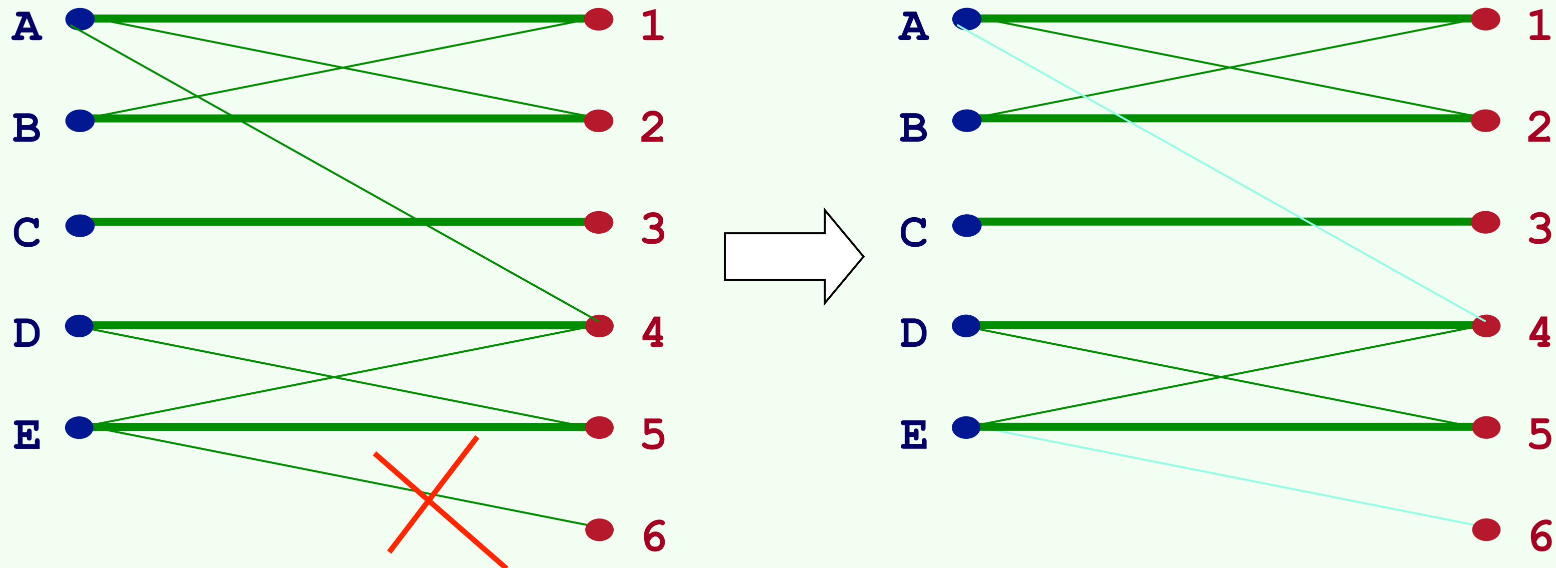
Global Constraints: allDifferent

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 **E-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

- The alldifferent function allows to state that each variable in an array of CP variables takes a different value.

```
Solver<CP> cp();
var<CP>{int} x[1..5](cp,1..6);
solve<cp> {
  cp.post(alldifferent(x), onDomains);
}
```

- The available consistency levels are **onValues**(default) and **onDomains** (the first corresponds to the naïf implementation).

- The effect of the different consistency levels can be observed in the sudoku program applied to the data file presented as auxiliary to these slides.

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: Circuit

- The previous global constraints may be regarded as imposing a certain "permutation" on the variables.

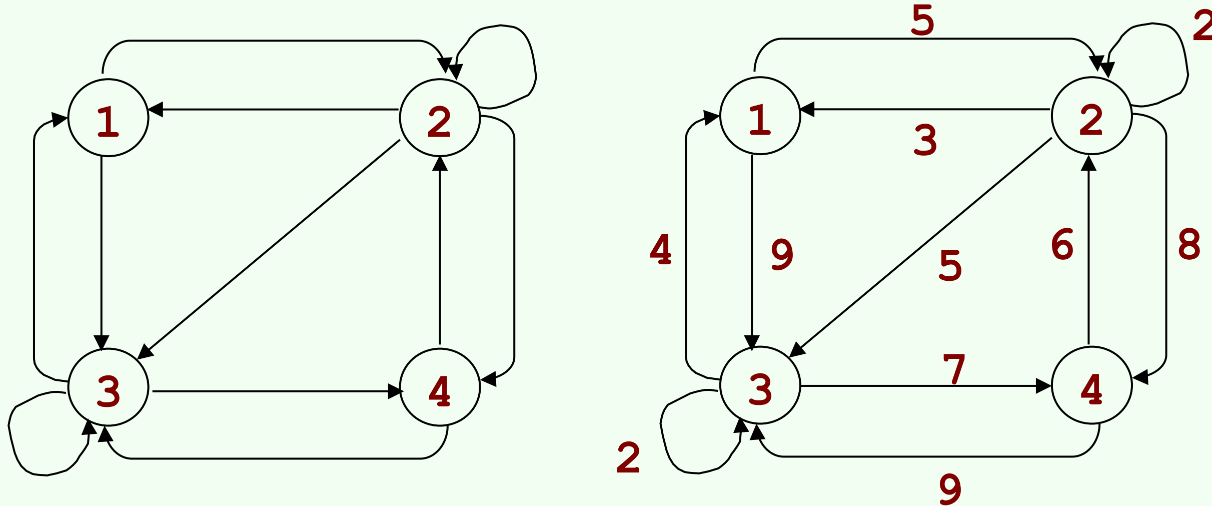
- In many problems, such permutation is not a sufficient constraint. It is necessary to impose a certain "ordering" of the variables.

- A typical situation occurs when there is a sequencing of tasks, with precedences 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 precedences.



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

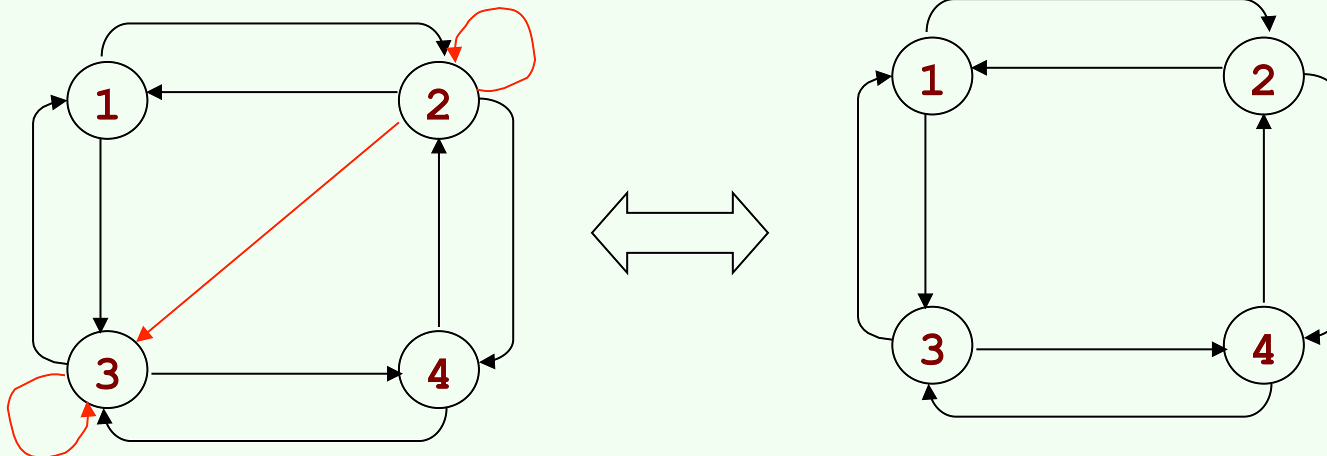
- This is a situation typical of several TSP-like problems (Traveling Salesman).

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
  - 1 → 2 → 4 → 3 → 1 ;
  - 1 → 3 → 4 → 2 → 1

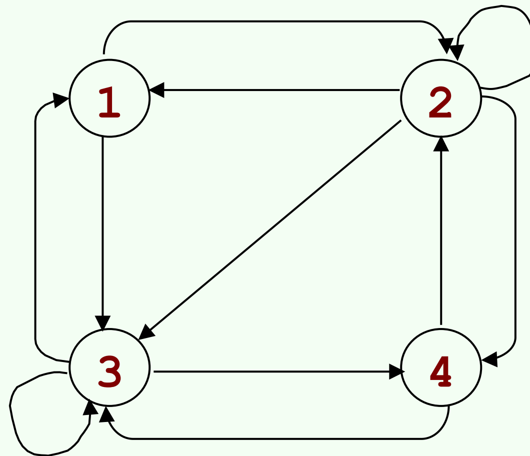
- Certain arcs (e.g. All unary such as 2 → 2, as well as some binary, e.g. 2 → 3), do not belong to any hamiltonian circuit and can (should !) be safely pruned.

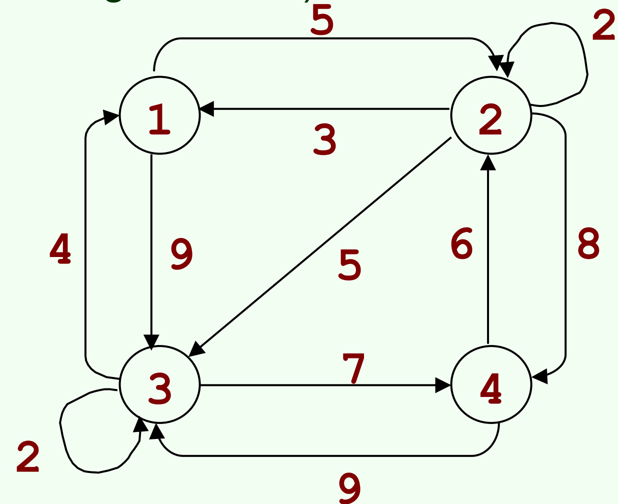


Global Constraints: Circuit

- The pruning of the arcs that do not belong to any circuit is the goal of the global constraint circuit.

- The **circuit** constraint states that an array of variables has to represent a Hamiltonian circuit in a directed graph. Assume a weighted directed graph G = (V, E), where (the third element of the E tuples is the weight or cost)





- V = {1,2,3,4}
- E = {(1,2,5),(1,3,9),(2,1,3),(2,2,2),(2,3,5),(2,4,8),(3,1,4),(3,3,2),(3,4,7),(4,2,6),(4,3,9)}

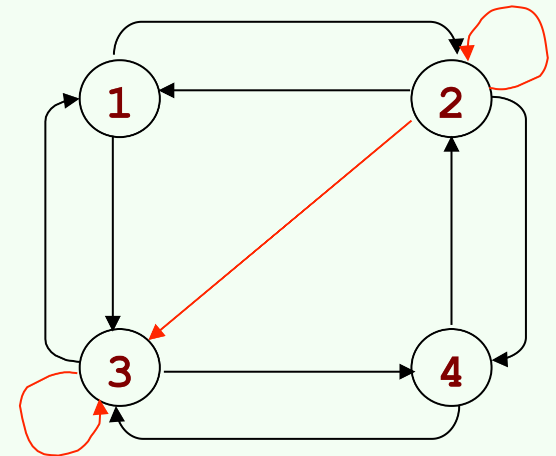
- Representing the graph by the successor array of variables x, the circuit constraint forces x to represent an Hamiltonian circuit on G.

Global Constraints: Circuit

- The circuit constraint states that an array of variables has to represent a Hamiltonian circuit in a directed graph, adopting a representation of a graph by an array of successor nodes.

```
import cotfd;
range Nodes = 1..4;
set{int} succ[Nodes] =
   [{2,3},{1,2,3,4},{1,3,4},{2,3,4}];
Solver<CP> cp();
var<CP>{int} x [s in Nodes](cp,succ[s]);

solveall<cp> {
  cp.post(circuit(x));
} using {
  labelFF(x);
  cout << x << endl;
}
```



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.

- The **Element** (or indexing) constraint allows to index an array of integers or variables by indices that are variables themselves. In the following example, array **a** is indexed by variable **x** to be linked with variable y.

```
Solver<CP> cp();
int a[1..7] = [9,3,4,6,2,7,4];
var<CP>{int} x(cp,1..7);
var<CP>{int} y(cp,{1,4,6});
solve<cp> {
  cp.post(a[x] == y, onDomains); }
} ...
```

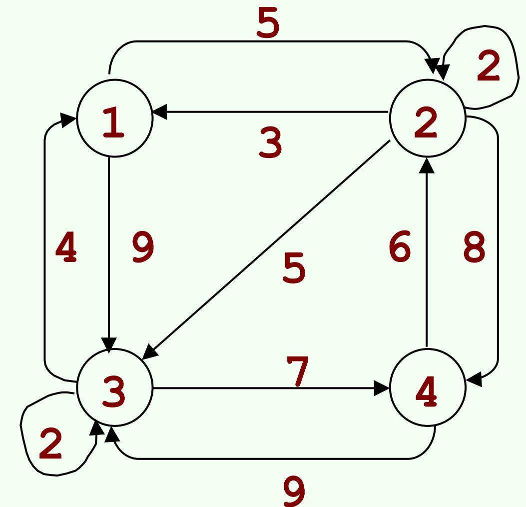
- The available consistency levels are **onBounds**(default) and **onDomains**.

- In the latter case the domain of x is pruned to {3,4,7} and the domain of y to {4,6}.

Global Constraints: Circuit+Element

- The combination of Circuit and Element is shown below to solve the TSP.

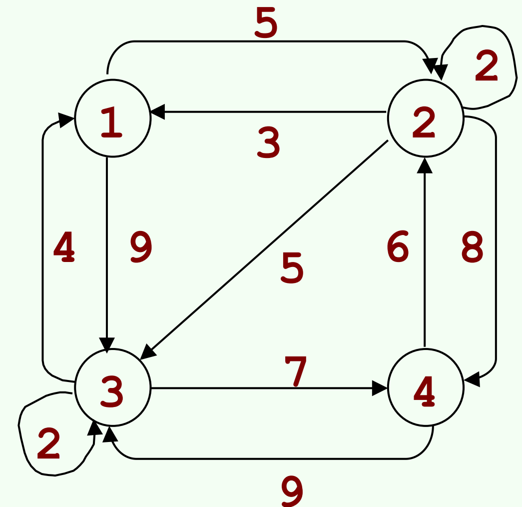
```
import cotfd;
range Nodes = 1..4;
set{int} succ[Nodes] =
   [{2,3},{1,2,3,4},{1,3,4},{2,3,4}];
Solver<CP> cp();
var<CP>{int} x [s in Nodes](cp,succ[s]);
var<CP>{int} cc(cp,0..100);
int costs[Nodes,Nodes] =
   [ [0,5,9,0],
     [3,2,5,8],
     [4,0,2,7],
     [0,6,9,0] ];
minimize<cp> cc subject to {
   cp.post(cc == sum(i in Nodes)
      costs[i, x[i]]);
   cp.post(circuit(x));
} using
   labelFF(x);
```



Global Constraints: minCircuit

- In fact, the minCircuit<CP> constraint is equivalent to the combination of the circuit and Element constraints (but more efficent !) :

```
import cotfd;
range Nodes = 1..4;
set{int} succ[Nodes] =
   [{2,3},{1,2,3,4},{1,3,4},{2,3,4}];
Solver<CP> cp();
var<CP>{int} x [s in Nodes](cp,succ[s]);
var<CP>{int} cc(cp,0..100);
int costs[Nodes,Nodes] =
   [ [0,5,9,0],
     [3,2,5,8],
     [4,0,2,7],
     [0,6,9,0] ];
minimize<cp> cc subject to
  cp.post(minCircuit<CP>(x, costs, cc));
using
  labelFF(x);
```



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.
- Typically slots may be represented by an array of FD variables and for all values some bounds on the cardinality are imposed.
- For example, to guarantee that all values v in some domain Dom (1..3) appear at least twice in an array x with 8 elements, the following constraint can be used

```
Range Rng = 1..8;
Range Dom = 1..3;
var<CP>{int} x[Rng](cp,Dom);
...
forall(v in Dom)
     cp.post ( (sum(i in Rng) (x[i] == v) ) >= 2);
...
```

Global Cardinality Constraints

- Imposing lower and upper bounds on the ocurrence of values in a solution may be imposed by atleast/atmost constraints.

- The **atleast/atmost** constraints guarantee a lower/upper bound on the number of occurrences of each value in an array of variables. The available consistencies are onValues (default) and onDomains.

- In the example below, the value 1 must appear 0 times, the values 2 at least 2 times, the value 3 at least 1 time, and so on:

```
Solver<CP> cp();
int low[1..5] = [0,2,1,3,0];
var<CP>{int} x[1..6](cp,1..5);
solve<cp>
   cp.post(atleast(low,x));
```

- where the atleast constraint is equivalent to

```
forall(v in 1..5)
     cp.post ( (sum(i in 1..6) x[i] == v ) <= low[v])
```

Global Cardinality Constraints

- Sometimes, a problem might require that a solution has exact number of occurrences of the values in a solution. Such requirement might be specified with a combination of atleast/atmost constraints sharing the same bounds, as in

```
Solver<CP> cp();
int occ[1..5] = [1,2,1,3,2];
var<CP>{int} x[1..9](cp,1..5);
solve<cp>{
   cp.post(atleast(occ,x)); cp.post(atmost(occ,x)); }
```

- The same effect can be obtained with the exact constraint, as shown below.

```
Solver<CP> cp();
int occ[1..5] = [1,2,1,3,2];
var<CP>{int} x[1..9](cp,1..5);
solve<cp>{
   cp.post(exactly(occ,x));
```

- The available consistencies are **onValues** (default) and **onDomains**.

Global Cardinality Constraints

- Many problems may require different bounds on the number of occurrences, e.g.
  - Lower bounds to guarantee that resources are used; and
  - Upper bounds to guarantee that capacities are not exceeded.
- For example, assume a team of 7 people (nurses) where one or two must be assigned the morning shift (m), one or two the afternoon shift (a), one the night shift (n), while the others may be on holliday (h) or stay in reserve (r).
- Again this problem can be modelled with combined atleast/atmost constraints:

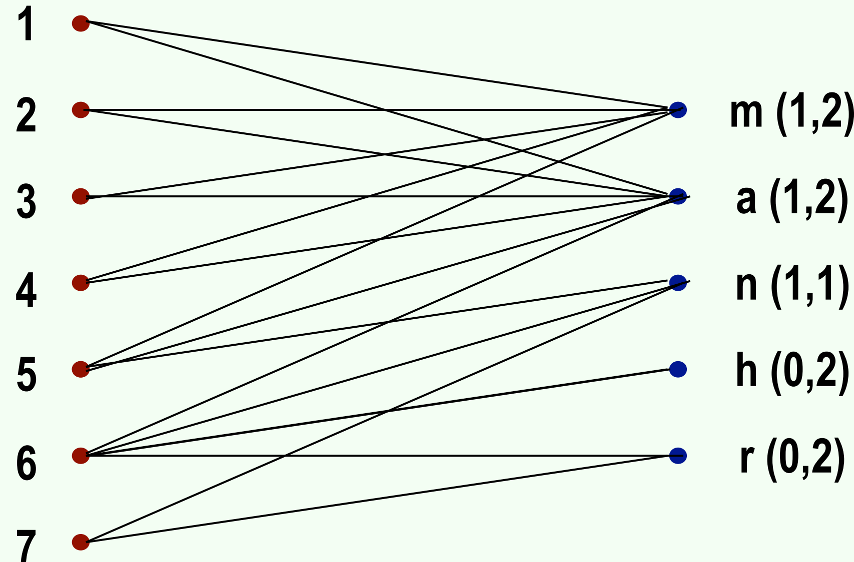
```
range Workers = 1..7;
enum Shifts = {m,a,n,h,r};
set{Shifts} dom[Workers] =
    [{m,a},{m,a},{m,a},{m,a},{m,a,n},{a,n,h,r},{n,r}];
int lower[Shifts] = [1,1,1,0,0];
int upper[Shifts] = [2,2,1,5,5];
Solver<CP> cp();
var<CP>{int} w [i in Workers](cp,dom[i]);
solve<cp> {
  cp.post(atleast(lower,w)); cp.post( atmost(upper,w));
}
```

Global Cardinality Constraints

- 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.

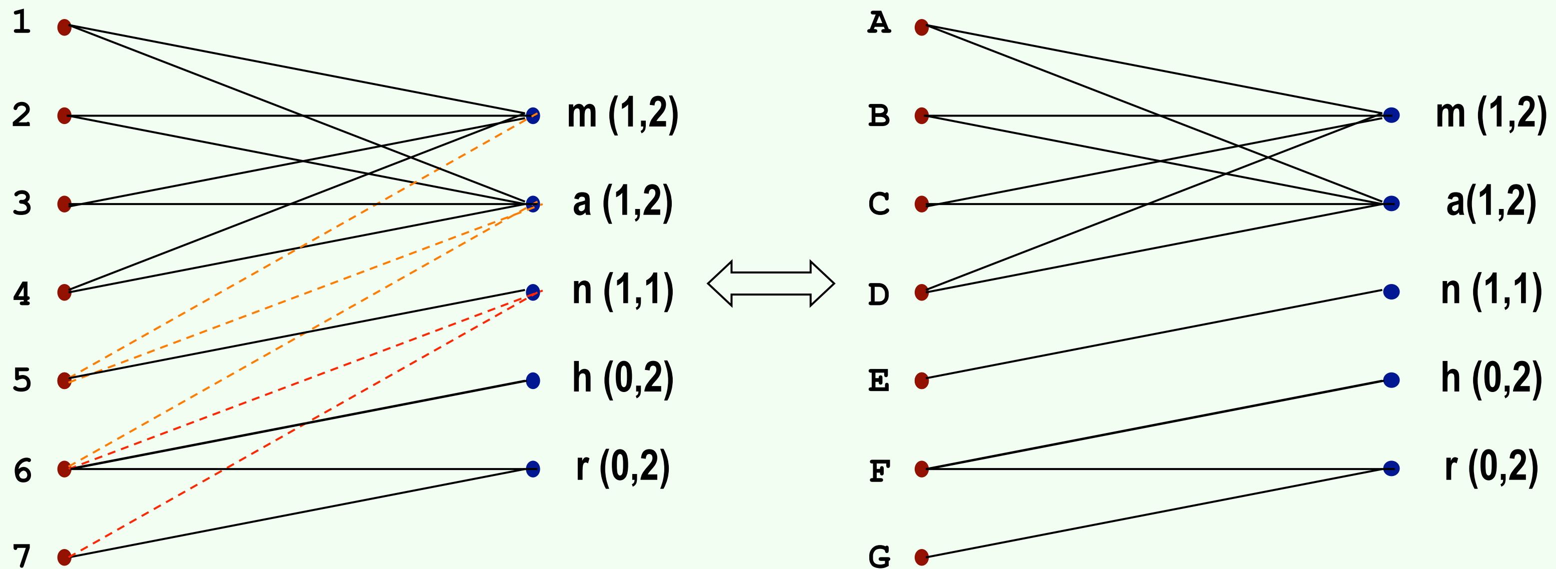
```
1,2,3,4::{m,a}, 5::{m,a,n}, 6::{a,n,h,r}, 7::{n,r}
```

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



Global Cardinality Constraints

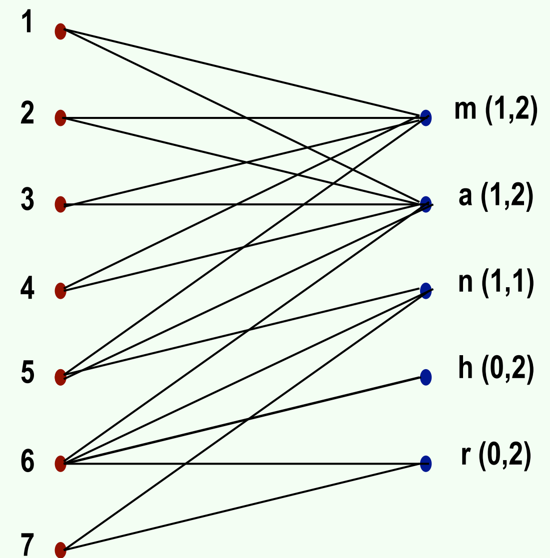
- Workers 1, 2, 3 and 4 may only take values m and a. Since these may only be attributed to 4 people, no one else, namely 5 or 6, may take these values m and a.
- Now worker 5 may now only take value n. Since this must be taken by a single person, no one else (e.g. 6 or 7) may take value n.



Global Cardinality Constraints

- 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,** that combines the atleast and atmost constraint but explores a different propagation algorithm, based on maximal flows in graphs.

```
range Workers = 1..7;
enum Shifts = {m,a,n,h,r};
set{Shifts} dom[Workers] = [{m,a},{m,a},
{m,a},{m,a},{m,a,n},{a,n,h,r},{n,r}];
int lower[Shifts] = [1,1,1,0,0];
int upper[Shifts] = [2,2,1,5,5];
Solver<CP> cp();
var<CP>{int} w [i in Workers](cp,dom[i]);
solve<cp> {
  cp.post(cardinality(lower, w, upper));
  // cp.post(atleast(lower, w));
  // cp.post( atmost(upper, w));
}
```



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 **atLeastNValue** global constraint is a soft version of the alldifferent. It counts the number of different values in a vector of variables.
- The constraint maintains that the variable **numberOfValues** is the number of different values taken by the variables in the array x. In the example below the number of different values x[i] for the 5 variables should be between 3 and 5 (in the last case this corresponds to an all_different).

```
Solver<CP> cp();
var<CP>{int} x[1..5](cp,1..6);
var<CP>{int} n(cp,3..5);
solve<cp>
   cp.post(atLeastNValue(x,n));
```

- The available consistencies are **onValues** (default) and **onDomains**.

Global Constraints: Sequence

- In some applications it is not simply intended to impose cardinality constraints on a set of variables, but rather that the variables are considered as a sequence (i.e. an array is used to model a sequence rather than a set).

- In this case, not only cardinality constraints are imposed on the whole sequence, but additionally cardinality constraints are also imposed in specific subsequences.

Example (Car sequencing):

The goal is to manufacture in an assembly line 10 cars with different options (1 to 5) shown in the table. Given the assembly conditions of option i, for each sequence of n_i cars, only m_i cars can have that option installed shown in the table.

		cars					
option	capacity	1	2	3 4	5 6	7 8	9 10
1	1 / 2	X				X	X
2	2 / 3			X	X		X
3	1 / 3	X				X	
4	2 / 5	X	X		X		
5	1 / 5			X			
	Configuration	1	2	3	4	5	6

Global Constraints: Global Sequence

- Rather than imposing separate cardinality constraint on the sequence and all the relevant subsequences, a specialised global constraint uses an optimised algorithm to obtain more pruning than that obtained by separate cardinality constraints.

- The sequence constraint has the following format:

```
sequence(var<CP>{int}[] x, int[] demand, int p, int q, set{int} V}
```

- It essentially enforces two conditions:

  - for every value **i** in the range of the integer array **demand**, exactly **demand[i]** variables from the array **x** are assigned to value **i**;

  - at most **p** out of any **q** consecutive variables in the array **x** are assigned values from the given integer set **V**

- The default consistency level is **onDomains** but **onValues** is also available.

Global Constraints: Global Sequence

```
range Cars = 1..10;
range Configurations = 1..6;
int demand[Configurations] = [1,1,2,2,2,2];
range Options = 1..5;
int p[Options] = [1,2,1,2,1];
int q[Options] = [2,3,3,5,5];
set{int} cfs[Options] = [{1,5,6},{3,4,6},{1,5},{1,2,4},{3}];
Solver<CP> cp();
var<CP>{int} x [Cars](cp, Configurations);
solve<cp> {
  forall(o in Options)
    cp.post(sequence(x, demand, p[o], q[o], cfs[o]);
}
```

		cars					
option	capacity	1	2	3 4	5 6	7 8	9 10
1	1 / 2	X				X	X
2	2 / 3			X	X		X
3	1 / 3	X				X	
4	2 / 5	X	X		X		
5	1 / 5			X			
	Configuration	1	2	3	4	5	6

Other Global Constraints

- Many other global constraints have been proposed for specific problems (a list of 200 is maintained in the Global Constraint Catalog

http://www.emn.fr/x-info/sdemasse/gccat/

- Most modern solvers (SICStus Prolog, **GECODE, CHOCO, ZINC, CASPER, ...**) include implementations of some of these global constraints. For example, the current distribution of Zinc /Minizinc (1.6) has implementations of 55 global constraints

- **alldifferent**
- **alldifferent_except_0**
- **all_disjoint**
- **all_equal**
- **among**
- **at_least (atleast)**
- **at_most (atmost)**
- **at_most1 (atmost1)**
- **bin_packing**
- **bin_packing_capa**
- **bin_packing_load**
- **circuit**
- **count_eq (count)**
- **count_geq**
- **count_gt**
- **count_leq**
- **count_lt**
- **count_neq**
- **cumulative**
- **decreasing**
- **diffn**
- **disjoint**
- **distribute**
- **element**
- **exactly**
- **global_cardinality**
- **global_cardinality_closed**
- **global_cardinality_low_up**
- **global_cardinality_low_up_closed**
- **increasing**
- **int_set_channel**
- **inverse**
- **inverse_set**
- **lex_greater**
- **lex_greatereq**
- **lex_less**
- **lex_lesseq**
- **lex2**
- **link_set_to_booleans**
- **maximum**
- **member**
- **minimum**
- **nvalue**
- **partition_set**
- **range**
- **regular**
- **roots**
- **sliding_sum**
- **sort**
- **strict_lex2**
- **subcircuit**
- **sum_pred (sum)**
- **table**
- **value_precede**
- **value_precede_chain**

Other Global Constraints

- In addition to those already addressed, Comet provides other global constraints:
  - **stretch / regular**

    A generalisation of the sequence constraint, that constrain the sequence to have long repetition of values or that the sequence satisfies a regular expression. Useful for roastering problems that should satisfy "contractual" constraints.
  - **knapsack / multipleKnapsack**

    Useful to solve knapsack problems, i.e. to select subsets of objects that do not exceed a certain capacity and have a maximum
  - **cumulative**

    Useful to solve resource allocation problems with a limited number of resources available for a number of tasks, that execute over time periods.
- The knapsack constrains will be briefly addressed and a more detailed discussion will be done on the cumulative constraints.

Knapsack Constraints

- **binaryKnapsack**

- The **binaryKnapsack** is a constraint on the scalar product between a vector of 0-1 variables and a vector of integers. It can be viewed as the decision on which weighted items are placed into a knapsack of variable capacity.

```
Solver<CP> cp();
int w[1..4] = [1,5,3,5];
var<CP>{int} x[1..4](cp,0..1);
var<CP>{int} c(cp,6..7);
solve<cp> {
   cp.post(binaryKnapsack(x,w,c));
}
```

- Only the **onDomains** consistency is available.

- **lightBinaryKnapsack**

- a weaker but more efficient propagation, (strongly polynomial) than the AC version when the capacity is huge.

Knapsack Constraints

- **multiknapsack**

- A natural generalization of binaryknapsack is the multiknapsack constraint, under which a set of weighted items must be placed into several bins:

```
Solver<CP> cp();
int w[1..4] = [1,5,3,5];
var<CP>{int} x[1..4](cp,0..3);
var<CP>{int} l[1..3](cp,3..7);
solve<cp> {
   cp.post(multiknapsack(x,w,l));
}
```

- A solution satisfying the constraint is `x=[1,1,2,3]`, `l=[6,3,5]` because
  - objects 1 and 2, with weights 1 and 5, respectively , are placed into bin 1 `(x[1]=x[2]=1)` raising its total load to 6 `(l[1]=6)`
  - object 3, with weight 3, is placed into bin 2 `(x[3]=2)`, so that the load of bin 2 is 3 `(l[2]=3)`
  - object 4 is placed into bin 3 `(x[4]=3)`, so that the load of bin 3 is 5 `(l[3]=5)`

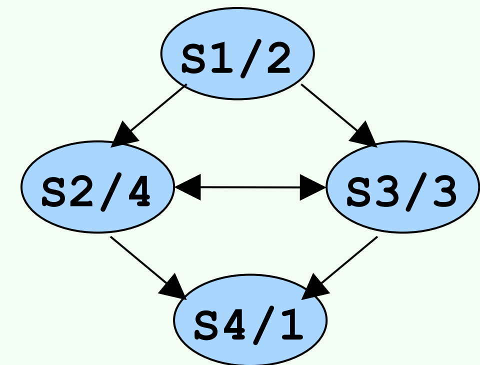
- The only consistency level available for the multiknapsack constraint is Auto (not easily classifiable).

Redundant Constraints

- Before addressing the cumulative global constraint we will discuss a simple scheduling problem and the pitfalls with reasoning with constraints **separately**.

Example:

A project is composed of the four tasks illustrated in the graph, showing precedences between them, as well as mutual exclusion (↔). The tasks durations are shown in the nodes.

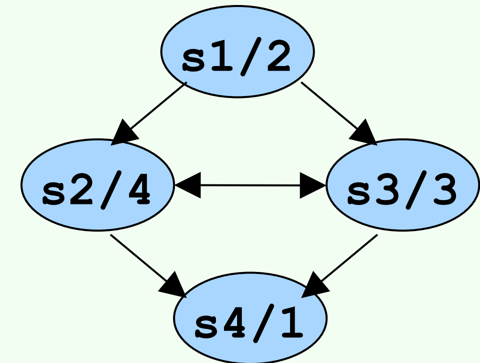


- The problem may be modeled by the following precedence and disjunctive constraints

```
Solver<CP> cp();
var<CP>{int} s[1..4](cp,0..9);
int d[1..4] = [2,4,3,1];
solve<cp> {
  cp.post(s[2] >= s[1]+d[1]);
  cp.post(s[3] >= s[1]+d[1]);
  cp.post(s[4] >= s[2]+d[2]);
  cp.post(s[4] >= s[3]+d[3]);
  cp.post(s3 >= s[2]+d[2] || s2 >= s[3]+d[3]); }
```

Redundant Constraints

```
cp.post(s[2] >= s[1]+d[1]);
cp.post(s[3] >= s[1]+d[1]);
cp.post(s[4] >= s[2]+d[2]);
cp.post(s[4] >= s[3]+d[3]);
cp.post(s[3] >= s[2]+d[2] ||
        s[2] >= s[3]+d[3]);
```



- Because task 1 and **both** tasks 2 and 3 (as well as task1), must be finished before task 4 starts, and because the other tasks have durations 2, 4 and 3 respectively, task 4 may only start at time

  s[4] >= 0+2+3+4 = 9

  which should fix its value to 9. However, because the disjunction is dealt separately from the precedence constraints, the only propagation that is obtained is

  s[4] >= 0+2+4 = 6 and s[4] >= 0+2+3 = 5

  and hence the domain of s[4] is narrowed to 6..9 rather than fixed to 9.

Redundant Constraints

- In general, the interaction of many constraints may not be adequately processed by the corresponding propagators.
- Whenever the pitfalls of such interaction are identified, one technique that might be used is the inclusion of **redundant** constraints.
- Such constraints do not add to the semantics of the programs, i.e. the programs with and without them are equivalent. However, they add to the efficiency of constraint processing, improving its pruning, and therefore leading to a more efficient search.

- In scheduling problems, these redundant constraints, aim at improving the beginning and ending of the tasks (**edge-finding**).Two simple cases are
- when **k** non-ovelapping tasks $\mathbf{X_i}$ antecede some task Z the following redundant constraint can be added

$$s_z \;\; >= \;\; \min \{ \min(s_1), \min(s_2), ..., \min(s_k) \} \; + d_1 + d_2 + ... + d_k$$

- when **k** non-ovelapping tasks $\mathbf{X_i}$ succeed some task Z the folowing redundant constraint can be added

$$s_z + d_z \;\; <= \;\; \max \{ \max(s_1), \max(s_2), ..., \max(s_k) \} \; - d_1 - d_2 - ... - d_k$$

Cumulative Constraints

- In general, edge finding requires more sophisticated techniques, namely in problems combining scheduling and resource management.
- In fact, if many units of a resource are available, then more than one of the tasks that use these resources may execute simultaneously. All that is needed is that the number of resources required at any given time does not exceed the existing resources.
- This is the semantics of the cumulative constraint, initially introduced in CHIP, and which had an enormous impact in the area of constraint programming.
- Let **S** be the set of starting times of **n** tasks s_i, **D** be the set of their durations d_i and **R** the set of the number of resources of a given type required by the tasks, r_i. Denoting by

$$a = \min_i(s_i) \quad ; \quad b = \max_i(s_i+d_i);$$

$$r_{i,k} = r_i \quad \text{if } s_i =< t_k =< s_i+d_i \text{ or } 0 \text{ otherwise.}$$

then

$$\textbf{cumulative(S,D,R,L)} \Leftrightarrow \forall_{k \in [a,b]} \sum_i r_{i,k} \leq L$$

Cumulative Constraints

- In Comet the cumulative constraint has the following format:

```
cumulative<CP>(int o,int h,int maxCap,var<CP>{int}[] s,
               var<CP>{int}[] d,var<CP>{int}[] cap)
```

- The arguments of the constraint can be interpreted as
  - `int o:` origin of time period
  - `int h:` horizon of time period
  - `int maxCap:` maximum capacity
  - `var<CP>{int}[] s:` start variables
  - `var<CP>{int}[] d:` duration variables
  - `var<CP>{int}[] c:` capacity requirement

- As explained It essentially enforces the following conditions (where R is the range of the arrays s, d and cap):
  - For each i in R, s[i] >= o and s[i] + d[i] <= h.
  - For each t in [o,h], sum(i in R: s[i] <= t < s[i] + d[i]) c[i] <= maxCap.

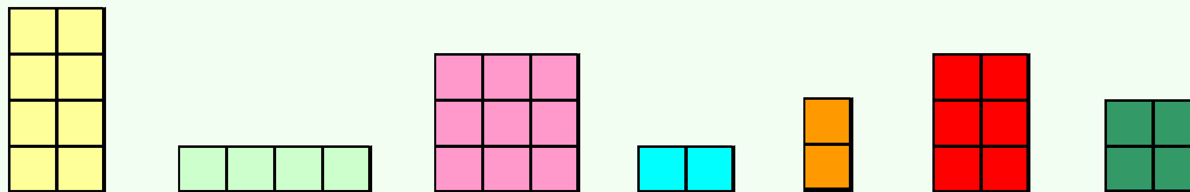
Cumulative Constraints

Example:

Take 7 tasks (A a G) with the duration and resource consumption (e.g. number of workers needed to carry them out) specified in the following arrays

d = [2 , 4 , 3 , 2 , 1 , 2 , 2]   ;   c = [4 , 1 , 3 , 1 , 2 , 3 , 2]

Graphically, the tasks can be viewed as



Goal: Assuming there are Rmax resources (e.g. workers) available at all times

(Sat) Find whether the tasks may all be finished in a given due time **Tmax**;

(Opt) Find the *minimum* due time **Tmax** (make span)

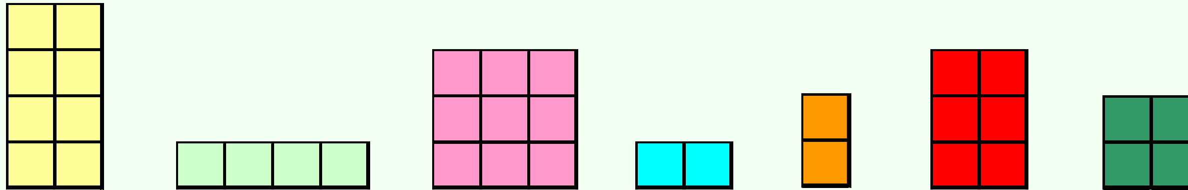
Many instances of the problem may be modelled by a simple constraint

```
cumulative<CP>(0,H,M,s,d,c)
```

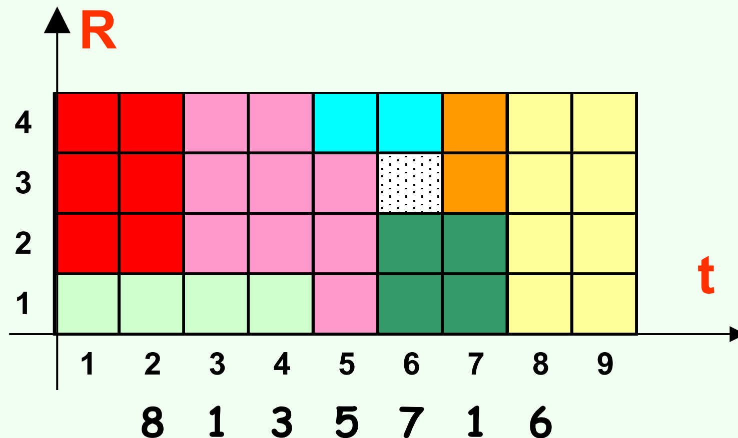
plus some additional constraints regarding the domains of the starting time variables.

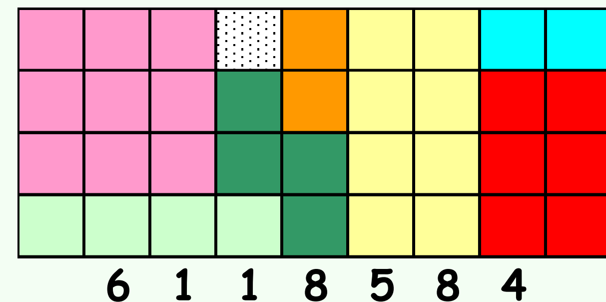
Cumulative Constraints

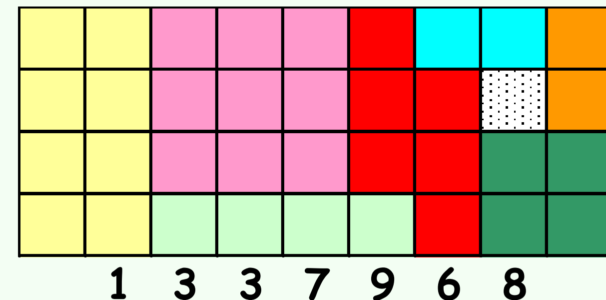
Some Instances: `cumulative<CP>(O,H,M,s,d,c)`



- With M = 4 (4 resource units available) and imposing that all tasks start no earlier than 0 (O = 0) and finish no later than time 9 (H = 9+1) a number of answers are obtained, (allowing one of the 6 workers to rest for one hour) namely

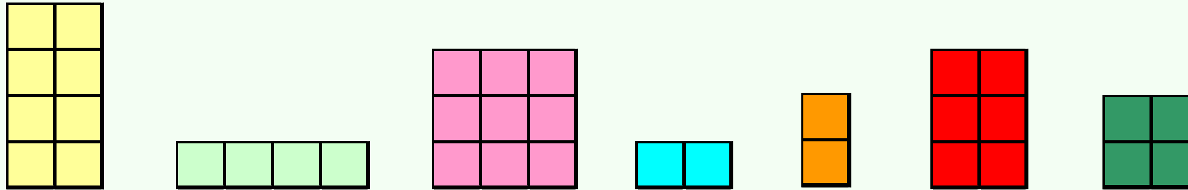




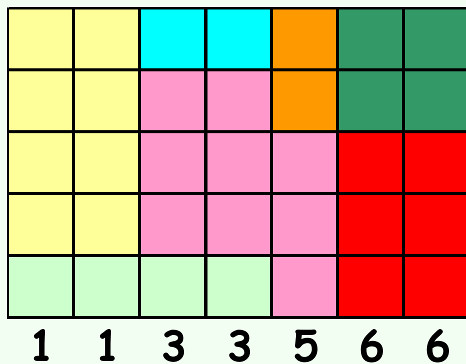


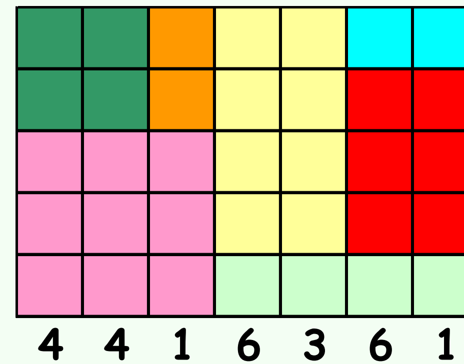
Cumulative Constraints

Some Instances:



- With M = 6 (6 resource units available) and imposing that all tasks are finished at time 6 (H = 6+1) a number of answers are still obtained, (allowing no rest for the workers)





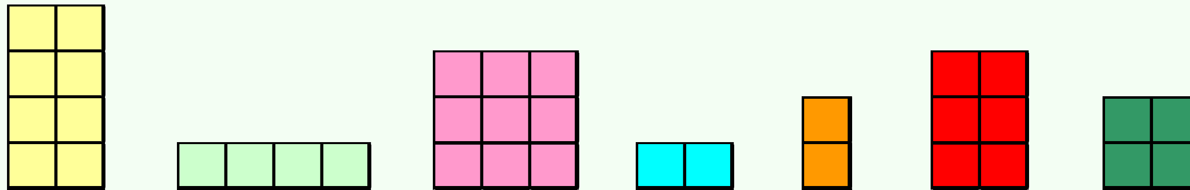
Cumulative Constraints

- In some applications, tasks are flexible, in the sense that time may be traded for resources.
- For example, a flexible task might require either 2 workers working for 3 hours, or 3 workers working for 2 hours. It may even be executed by a single worker during 6 hours, or by 6 workers in 1 hour.
- Flexible tasks may be more easily accomodated within the resources (and time) available.
- Scheduling of this type of tasks may be specified as before.
- However, whereas in the previous case, the durations **d** and resources **c** used by each task **i** were constants, now they are variables **a** and **b** constrained by

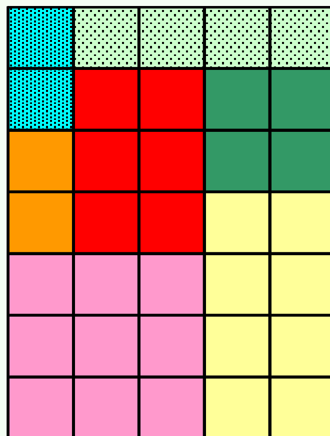
```
cp.post(s[i] * d[i] = a[i] * b[i])
```

Cumulative Constraints

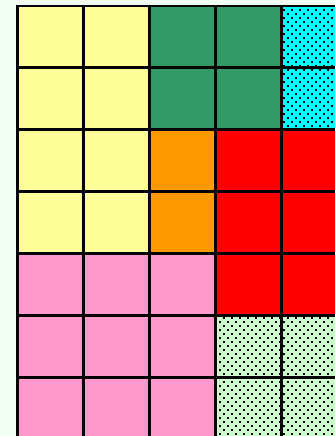
Results



With H = 5+1 and M = 7 (previously impossible) there are now several solutions. (Notice the “deeper” transformation in task 2, from (4*1 ⇒ 2*2), in addition to a “rotation”).



T	2	4	3	1	1	2	2
R	4	1	3	2	2	3	2
S	4	2	1	1	1	2	4

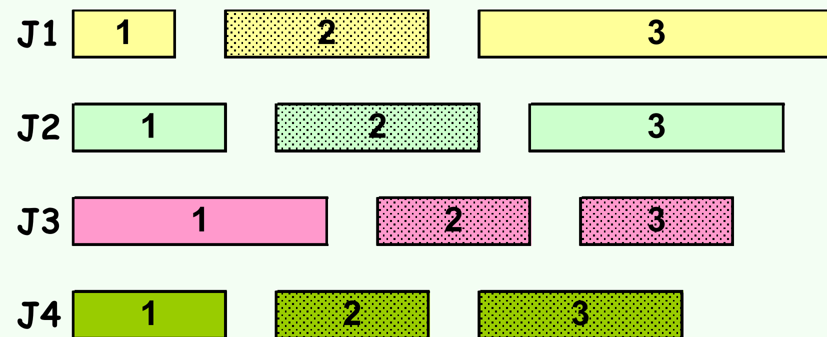


T	2	2	3	1	1	2	2
R	4	2	3	2	2	3	2
S	1	4	1	5	3	4	3

Cumulative Constraints: Job Shop

- The job shop problem consists of executing the different tasks of several jobs without exceeding the available resources.
- Within each job, there are several tasks, each with a duration. Within each job, the tasks have to be performed in sequence, possibly respecting mandatory delays between the end of a task and the start of the following task.
- Tasks of different jobs are independent, except for the sharing of common resources (e.g. machines). Each task must be executed in one machine of a certain type. The number of machines of each type is limited.
- A simple instance of the problem (with 2 machines) is given in the table below (with the corresponding graphic representation).

		Tasks Y		
	Z, D	1	2	3
Jobs X	1	1 , 2	2 , 4	1 , 7
	2	1 , 3	2 , 4	1 , 5
	3	1 , 5	2 , 3	2 , 3
	4	1 , 3	2 , 3	2 , 4



Cumulative Constraints: Job Shop

- This instance was proposed in the book Industrial Scheduling [MuTh63]. For 20 years no solution was found that optimised the "makespan", i.e. the fastest termination of all tasks.
- Around 1980, the best solution was 935 (time units). In 1985, the optimum was lower bounded to 930. In 1987 the problem was solved with a highly specialised algorithm, that found a solution with makespan 930.
- With the cumulative/4 constraint, in the early 1990's, the problem was solved in 1506 seconds (in a SUN/SPARC workstation).

		Tasks Y									
	Z, D	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**
Jobs X	1	1, 29	2, 78	3, 9	4, 36	5, 49	6, 11	7, 62	8, 56	9, 44	a, 21
	2	1, 43	3, 90	5, 75	a, 11	4, 69	2, 28	7, 46	6, 46	8, 72	9, 30
	3	2, 91	1, 85	4, 39	3, 74	9, 90	6, 10	8, 12	7, 89	a, 45	5, 33
	4	2, 81	3, 95	1, 71	5, 99	7, 9	9, 52	8, 85	4, 98	a, 22	6, 43
	5	3, 14	1, 6	2, 22	6, 61	4, 26	5, 69	9, 21	8, 49	a, 72	7, 53
	6	3, 84	2, 2	6, 52	4, 95	9, 48	a, 72	1, 47	7, 65	5, 6	8, 25
	7	2, 46	1, 37	4, 61	3, 13	7, 32	6, 21	a, 32	9, 89	8, 30	5, 55
	8	3, 31	1, 86	2, 46	6, 74	5, 32	7, 88	9, 19	a, 48	8, 36	4, 79
	9	1, 76	2, 69	4, 76	6, 51	3, 85	a, 11	7, 40	8, 89	5, 26	9, 74
	a	2, 85	1, 13	3, 61	7, 7	9, 64	a, 76	6, 47	4, 52	5, 90	8, 45

Placement Problems

- Several applications of great (economic) importance require the satisfaction of placement constraints, i.e. the determination of where to place a number of components in a given space, without overlaps.

Some of these applications include:

- Wood boards cuttings: a number of smaller pieces should be cut from large boards:
- Placement of items into a large container.

In the first 2 problems the space to consider is 2D, whereas the third problem is a typical 3D application. We will focus on 2D problems.

- An immediate parallelism can be drawn between these 2D problems and those of scheduling, if the following correspondences are made:
  - Time ↔ the X dimension;
  - Resources ↔ the Y dimension;
  - A task duration ↔ the item X size (width);
  - A task resource ↔ the item Y size (height).

Placement Problems

Example:

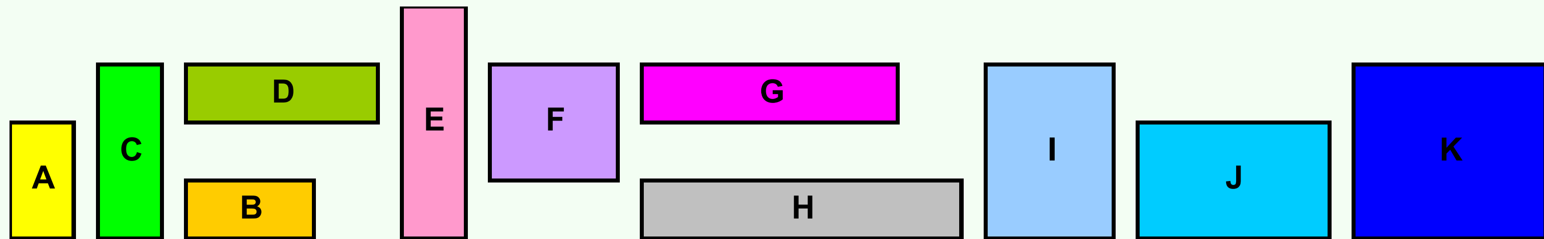
Find the appropriate cuts to be made on a wood board of dimensions W * H so as to obtain 11 rectangular pieces (A a K).

The various pieces to obtain have the following dimensions (width-w and height-h)

w = [1, 2, 1, 3, 1, 2, 4, 5, 2, 3, 3]

h = [2, 1, 3, 1, 4, 2, 1, 1, 3, 2, 3]

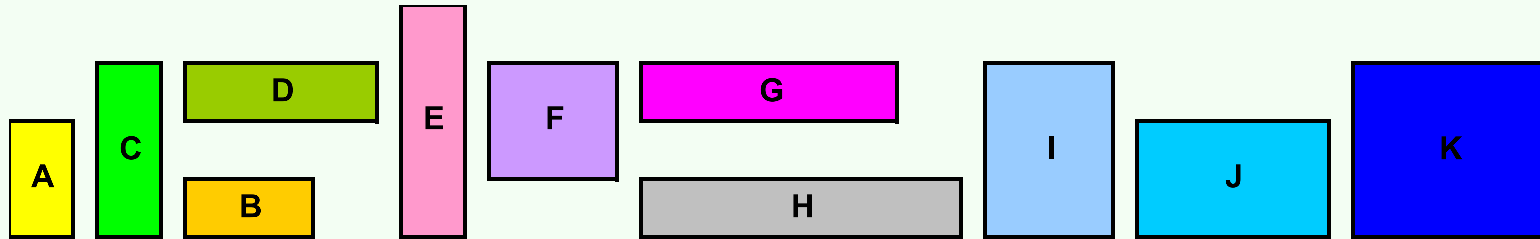
Graphically



and the constraint used, adapting durations to widths and resources to heights is

cumulative<CP>(1,W+1,H,x,w,h)

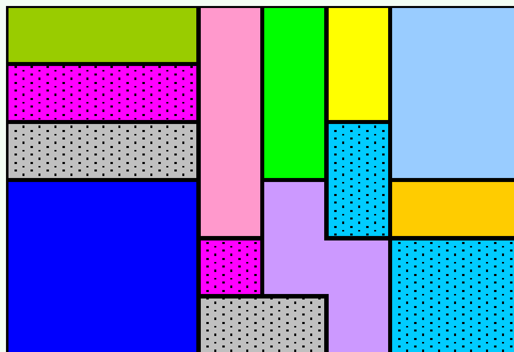
Placement Problems

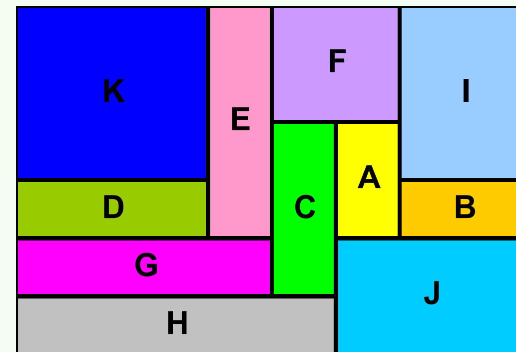


- Unfortunately, the results obtained have not a direct reading. For example, one of the solutions obtained with an 8*6 rectangle is

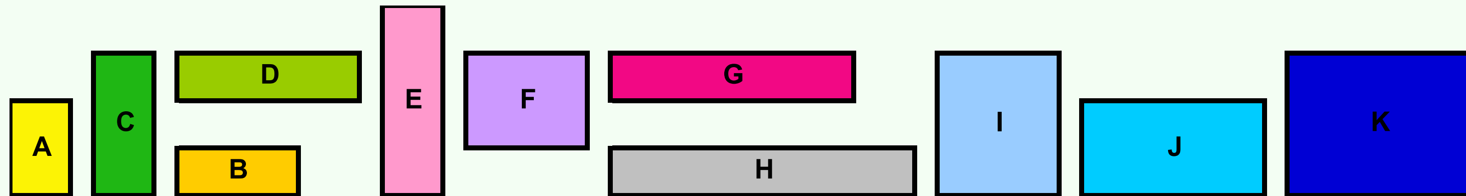
x = [6, 7, 5, 1, 4, 5, 1, 1, 7, 6, 1]

That can be read as (???) or as





Placement Problems



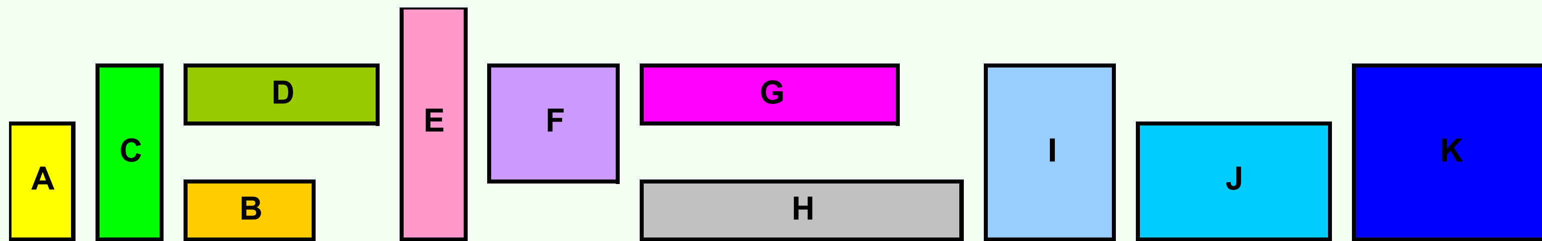
- To avoid this ambiguity, one should explicitely compute, not only the X-origin of the rectangles, but also its Y-origin.
- Such computation can easily be made, taking into account that all that is needed is considering a rotation of 90º in the viewing perspective, changing the X with the Y axes.
- Hence, all that is required is a “duplication” of the previous program, considering not only X variables, but also Y variables for explicit control over the Y-origins of the rectangles.

```
cumulative<CP>(1,W+1,H,x,w,h)
cumulative<CP>(1,H+1,W,y,h,w)
```

where

```
w = [1,2,1,3,1,2,4,5,2,3,3]
h = [2,1,3,1,4,2,1,1,3,2,3]
```

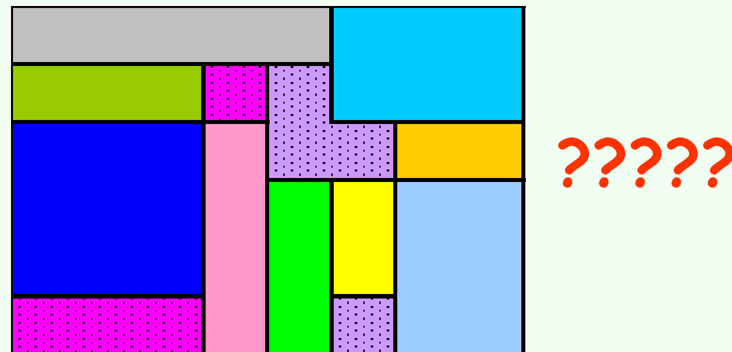
Placement Problems



- Yet, the results still aren't what they should be. For example, the first solution obtained is

```
x-y = [7-4,6-2,5-1,1-5,4-1,5-4,1-1,1-6,7-1,6-5,1-2]
```

corresponding to



?????

- Analysing the problem, it becomes clear that its cause is the fact that no non-overlaping constraint was imposed on the rectangles!

Placement Problems

- Several applications of great (economic) importance require the satisfaction of placement constraints, i.e. the determination of where to place a number of components in a given space, without overlaps.
- The non overlapping of the rectangles defined by their x and y origins and their widths w (x-sizes) and heights h (y-sizes) is guaranteed, as long as one of the constraints below is satisfied (for rectangles i and j)

`x[i]+w[i] <= x[j]` rectangle `i` is left of rectangle `j`

`x[j]+w[j] <= x[i]` rectangle `i` is rigth of rectangle `j`

`y[i]+h[i] <= y[j]` rectangle `i` is below rectangle `j`

`y[j]+h[j] <= y[i]` rectangle `i` is above rectangle `j`

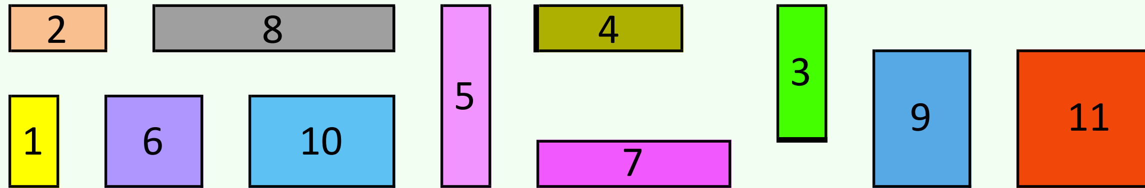
- As explained before, rather than commiting to one of these conditions, and change the commitment by backtracking, a better option is to adopt a least commitment approach, for implementing such disjunctive constraint.

Placement Problems

Important points to stress

- The enumeration should be made jointly on both the X_i and the Y_j, hence their merging into a single list Z.
- Several heuristics could possibly be used for variable enumeration. The heuristic chosen, ffc, is the classical choice.
- Alternatively, one could possibly start placing the "largest" rectangles in the corners, so as to make room for the others.
- The cumulative constraints are not strictly necessary, given the overlapping and the maximum constraints applied in both dimensions.
- Yet, they are extremely useful. Without them, the program would "hardly" work!

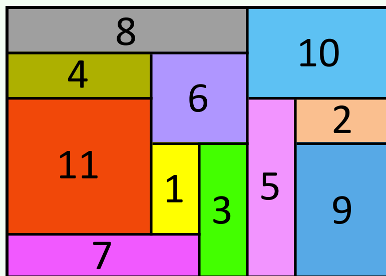
Placement Problems



- The cumulative constraints are not strictly necessary, given the overlapping and the maximum constraints applied in both dimensions.
- Nevertheless, the results below show that the use of redundant cumulative constraints, may speed up execution very significantly.

without cumulative:

x[4,7,5,1,6,4,1,1,7,6,1]
y[2,4,1,5,1,4,1,6,1,5,2]



2800 ms / 105453 fails

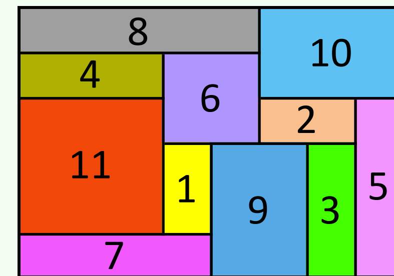
Speedup

Run time : > 1000

Backtracks: > 10000

with cumulative:

x[4,6,7,1,8,4,1,1,5,6,1]
y[2,4,1,5,1,4,1,6,1,5,2]



1 ms / 14 fails