Global Constraints: Generalised Arc Consistency

- It is often important to define n-ary “global” constraints, for at least two reasons
  - Ease the modelling of a problem
  - Exploitation of specialised algorithms that take the semantics of the constraint for efficient propagation, achieving generalised arc consistency.

- The **generalised arc consistency** (GAC) criterion sees that no value remains in the domain of a variable with no support in values of each of the other variables participating in the “global” constraint.

Example: all_different ([A_1, A_2, ..., A_n])

Constrain a set of n variables to be all different among themselves

Global Constraints: all_different

- The constraint definition (all_different/1) based on binary difference constraints (≠) does not pose any problems as much as modelling is concerned. For example, it may be defined recursively in CLP.

```
all_different([]).
all_different([H|T]):-
    one_diff(H,T),
    all_different(T).
```

```
one_diff(_,[]).
one_diff(X,[H|T]):-
    X #\= H,
    one_diff(X,T).
```

- However, constraint propagation based on binary constraints **alone** does not provide in general much propagation.
- As seen before, arc consistency is not any better than node consistency, and higher levels of consistency are in general too costly and do not take into account the **semantics** of the all_different constraint.

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

- It is clear that constraint propagation based on maintenance of node-, arc- or even path-consistency would not eliminate any redundant label.

- Yet, it is very easy to infer such elimination with a global view of the constraint!

Global Constraints: all_different

- Variables X_1, X_5 and X_8 may only take values 1, 2 and 3. Since there are 3 values for 3 variables, these must be assigned these values which must then be removed from the domain of the other variables.

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

- Now, variables X2, X4 and X9 may only take values 4, 5 e 6, that must be removed from the other variables domains.

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: all_different

- In this case, these prunings could be obtained, by maintaining (strong) 4-consistency.

- For example, analysing variables X_1, X_2, X_5 and X_8, it would be "easy" to verify that from the d^4 potential assignments of values to them, no assignment would include $X_2=1$, $X_2=2$, nor $X_2=3$, thus leading to the prunning of X_2 domain.

- However, such maintenance is usually very expensive, computationally. For each combination of 4 variables, d^4 tuples would be checked, with complexity $O(d^4)$.

- In fact, in some cases, n-strong consistency would be required, so its naïf maintenance would be exponential on the number of variables, exactly what one would like to avoid in search!

Global Constraints: all_distinct

- However, taking the semantics of this constraint into account, an algorithm based on quite a different approach allows the prunings to be made at a much lesser cost, achieving generalised arc consistency.

- Such algorithm is grounded on graph theory, and uses the notion of graph matching.

- To begin with, a **bipartite graph** is associated to an all_distinct constraint. The nodes of the graphs are the variables and all the values in their domains, and the arcs associate each variable with the values in its domain.

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

Global Constraints: all_distinct

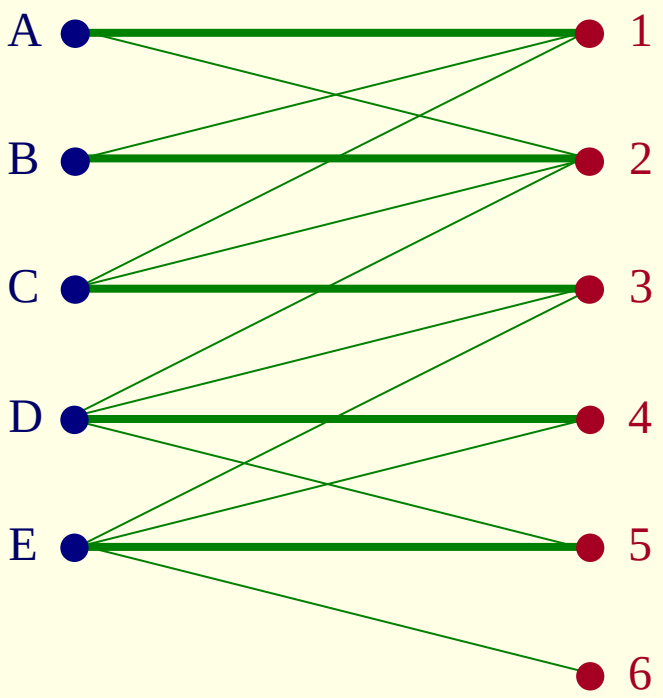
Key Ideas:

- For each variable-value pair, there is an arc in the bipartite graph.
- 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.
- For any solution of the all_distinct constraint there is one and only one maximal matching.

Global Constraints: all_distinct

Example:

```
A,B:: 1..2, C:: 1..3, D:: 2..5, E:: 3..6,
    all_distinct([A,B,C,D,E]).
```



Maximal Matching

```
A = 1
B = 2
C = 3
D = 4
E = 5
```

Global Constraints: all_distinct

- 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_distinct 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:

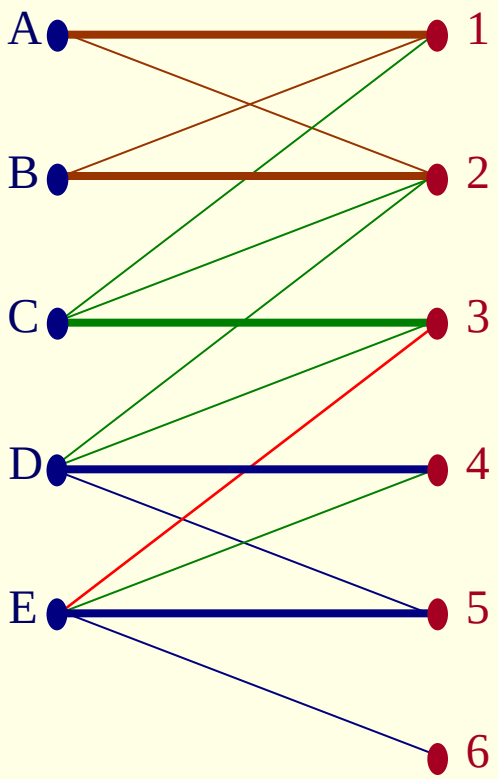
     a) To an **alternating cycle**; or

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

Global Constraints: all_distinct

Example: For the maximal matching (MM) shown

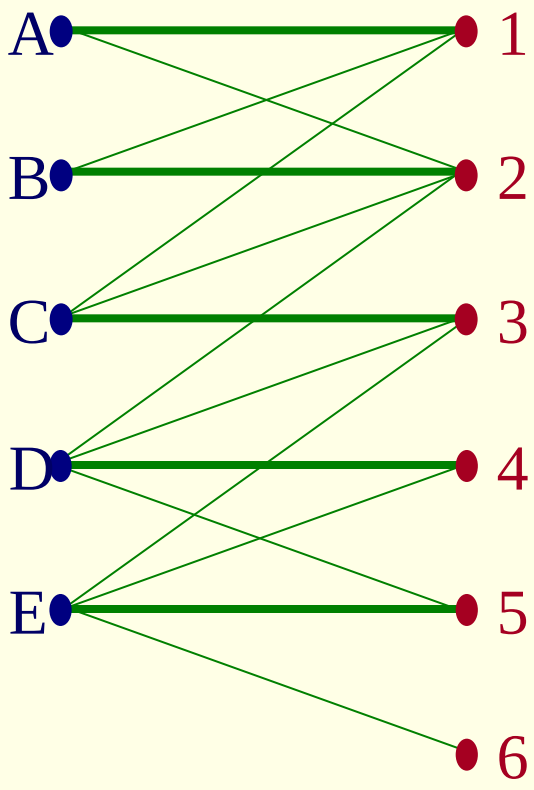
- **6** is a free node;
- **6-E-5-D-4** is an **even alternating path**, alternating arcs from the MM (E-5, D-4) with arcs not in the MM (D-5, E-6);
- **A-1-B-2-A** is an **alternating cycle**;
- **E-3** does not belong to any **alternating cycle**
- **E-3** does not belong to any **even alternating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



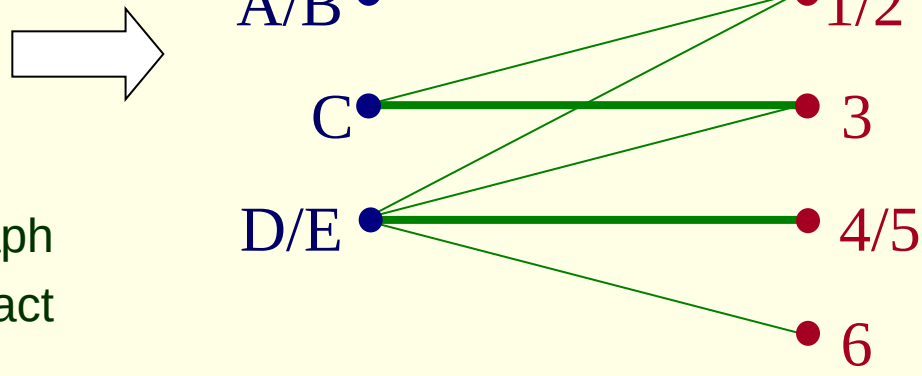
Global Constraints: all_distinct

- **Compaction**
  - Before this analysis, the graph may be “compacted”, aggregating, into a single node, “equivalent nodes”, i.e. those belonging to alternating cycles.
  - Intuitively, for any solution involving these variables and values, a different solution may be obtained by permutation of the corresponding assignments.
  - Hence, the filtering analysis may be made based on any of these solutions, hence the set of nodes can be grouped in a single one.

Global Constraints: all_distinct



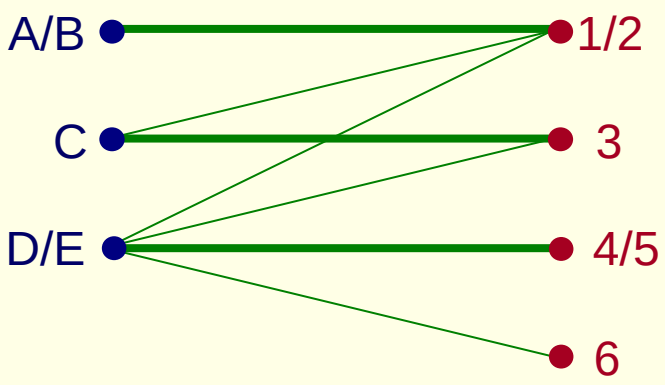
- A-1-B-2-A is an **alternating cycle**;
- By permutation of variables A and B, the solution <A,B,C,D,E> = <1,2,3,4,5> becomes <A,B,C,D,E> = <2,1,3,4,5>
- Hence, nodes A e B, as well as nodes 1 and 2 may be grouped together (as may the nodes D/E and 4/5).



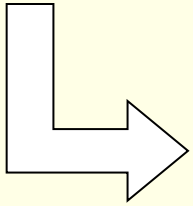
With these grouping the graph becomes much more compact

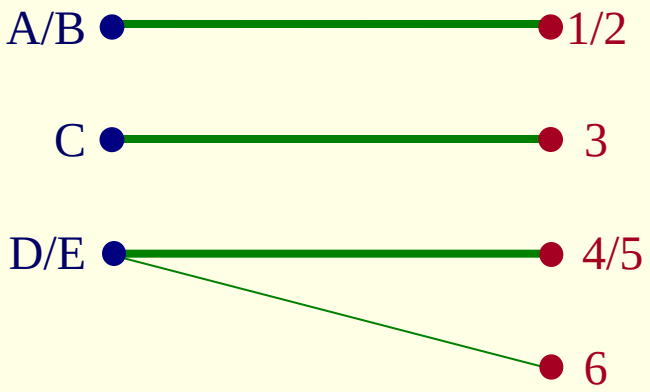
Global Constraints: all_distinct

Analysis of the compacted graph shows that



- Arc **D/E - 3** may be filtered out (notice that despite belonging to cycle **D/E - 3 - C - 1/2 - D/E**, this cycle is not alternating.
- Arcs **D/E - 1/2** and **C - 1/2** may also be filtered.

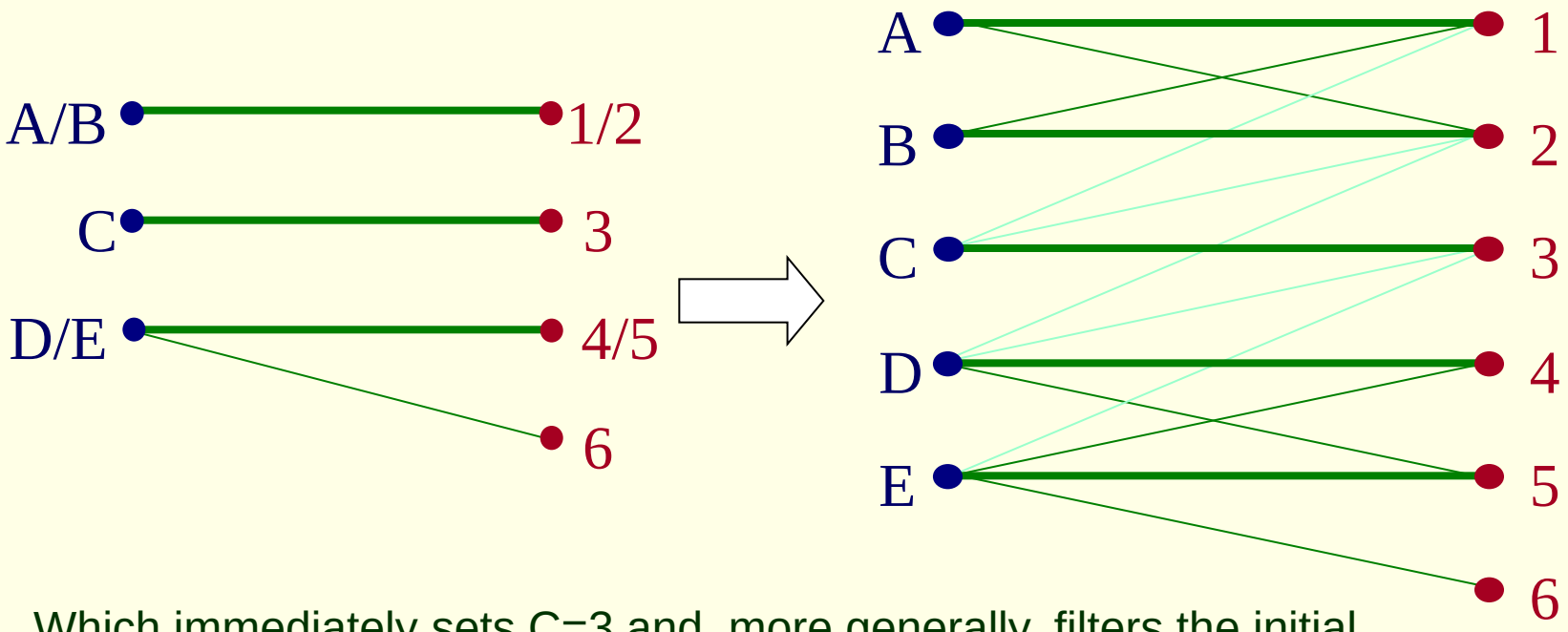




The compact graph may thus be further simplified to

Global Constraints: all_distinct

By expanding back the simplified compact graph, one gets the graph on the right



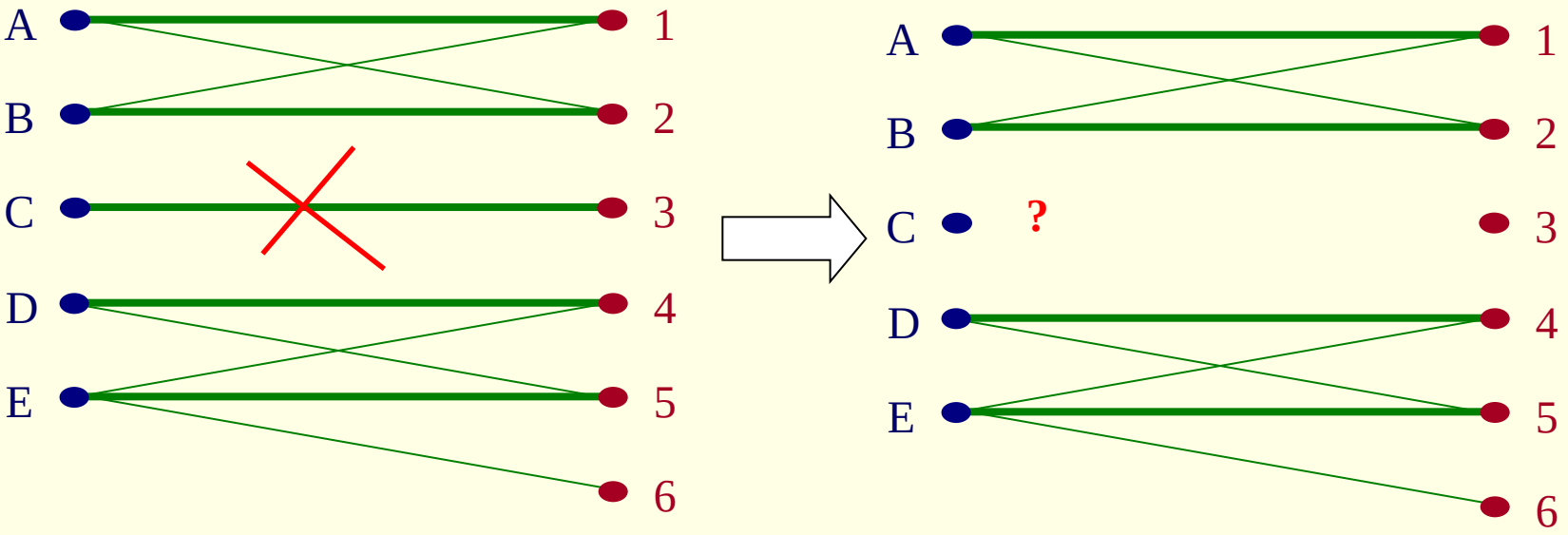
Which immediately sets C=3 and, more generally, filters the initial domains to

A,B :: 1..2, C:: 1,2,3, D:: 2,3,4,5, E:: 3,4,5,6

Global Constraints: all_distinct

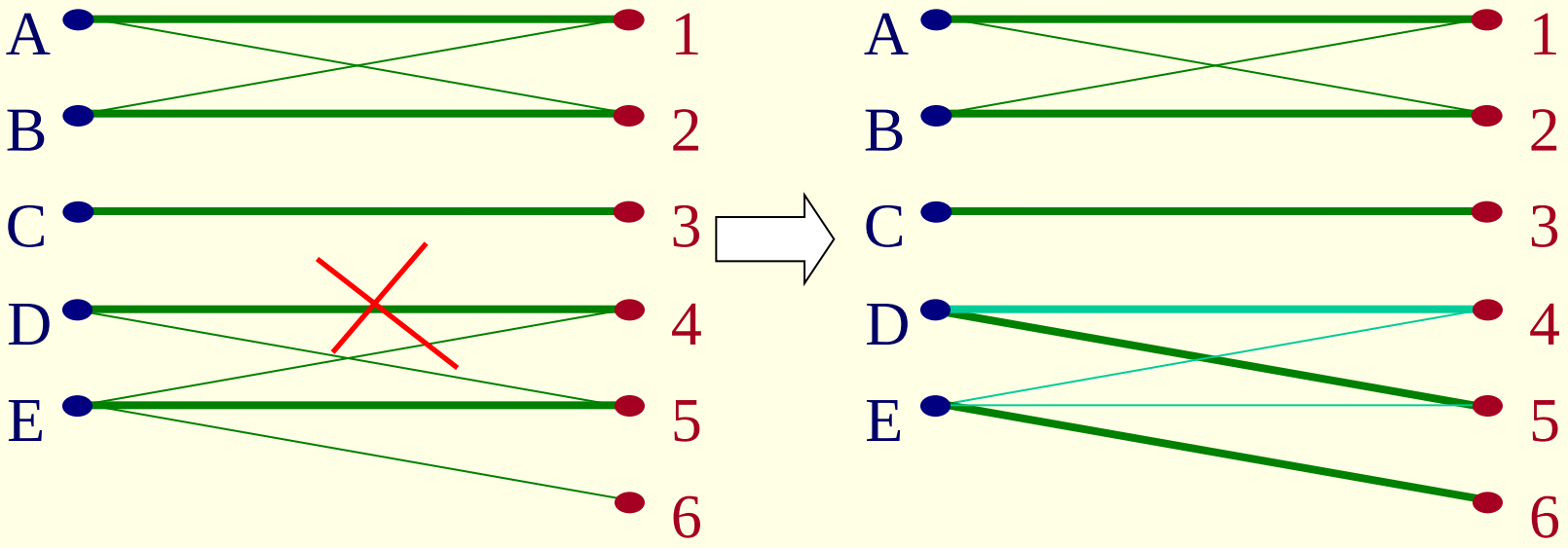
- Upon elimination of some labels (arcs), possibly due to other constraints, the alldifferent constraint propagates such prunings, incrementally. There are 3 situations to consider:

  1. Elimination of a **vital arc** (the only arc connecting a variable node with a value node): The constraint **cannot be satisfied**.



Global Constraints: all_distinct

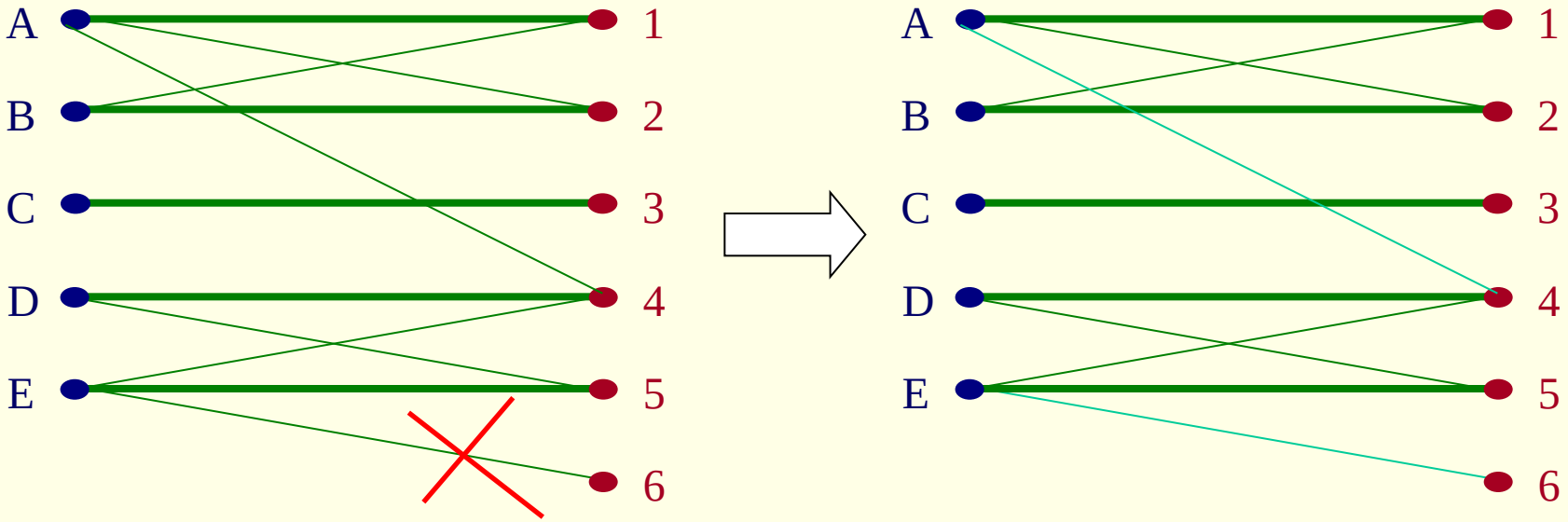
2. Elimination of a **non-vital arc which is a member** of the maximal matching
   - Determine a new maximal matching and restart from there.



A new maximal matching includes arcs **D-5** and **E-6**. In this matching, arc **E-5** does not belong to even alternating paths or alternating cycles.

Global Constraints: all_distinct

3. Elimination of a **non-vital arc which is not a member** of the maximal matching
   - Eliminate the arcs that do not belong any more to an alternating cycle or path.



Arc **A-4** does not belong to the **even alternating path** started in node 6.
D-5 also leaves this path, but it still belongs to an **alternating cycle**.

Global Constraints: all_distinct

Time Complexity:

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 **$O(dn\sqrt{n})$.**
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 **$O(dn+n+D+dn\sqrt{n})$.** Since **$D < dn$**, the total time complexity of the algorithm is dominated by the last term, thus becoming

$$O(dn\sqrt{n}).$$

which is much better than the poor result with a naïf analysis.

Global Constraints: alldifferent

Availability:

1. The all_diff constraint first appeared in the CHIP system (algorithm?).

2. The described implementation is incorporated into the ILOG system, and available as primitive **IlcAllDiff**.

3. This algorithm is also implemented in SICStus, through buit-in constraint **all_distinct/1**.

   Other versions of the constraint, namely **all_different/2**, are also available, possibly using a faster algorithm but with less pruning, where the 2nd argument controls the available pruning options.

4. In ECLiPSe, the global constraint **alldifferent/1** is in **fd_global** library. (*fd:alldifferent* only posts difference constraints for all pairs of variables). **alldifferent(+List, ++Capacity)** is a generalization allowing repeated elements (up to Capacity, of each value)

Global Constraints: Assignment

- The alldifferent constraint is typically applicable to problems where it is intended that different tasks are executed by different agents (or use different resources).

- However, tasks and resources are treated differently. Some are variables and the others the domains of these variables. For example, denoting 4 tasks/resources by variables $\mathbf{T_i}$ / $\mathbf{R_j}$, one would have to chose either one of the specifications below

```
T1 :: 1..3, T2 :: 2..4, T3 :: 1..4, T4 :: 1..3.
```

or

```
R1 :: [1,3,4], R2 :: 1..4, R3 :: 1..4, R4 :: [2,3]
```

- Hence, other constraints could be specified either on tasks or on resources, but not easily involving both types of objects

Global Constraints: Assignment

- Such “unfairness” may be overcome by treating both tasks and resources in a similar way, namely modelling both with distinct variables.

- These variables still have to adhere to the constraint that different tasks are performed in different resources.

- Also, if a task ***j*** is assigned to resource ***i***, then resource ***i*** is assigned to task ***j***. Denoting tasks by T_j and resources by R_i, the following condition must stand for any $i, j \in 1..n$

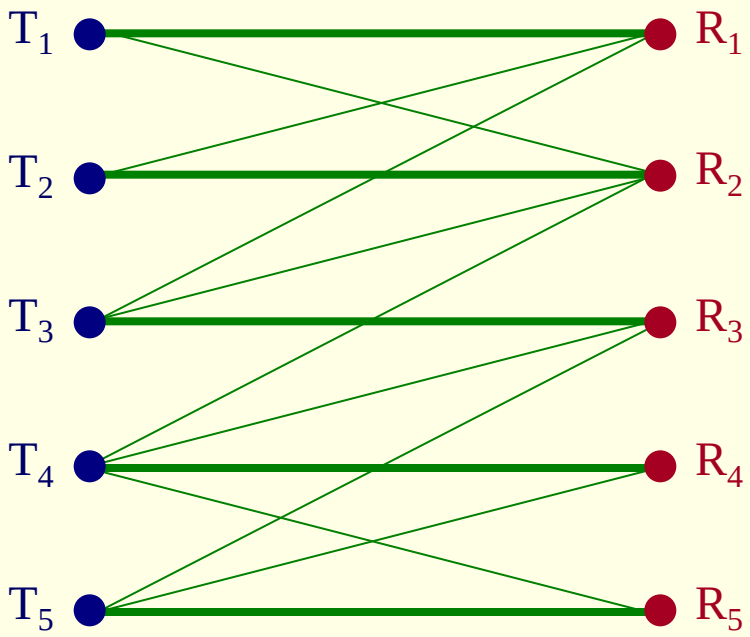
$$R_i = j \Leftrightarrow T_j = i$$

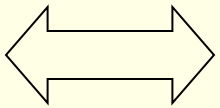
- This is the goal of global constraint **assignment/2**, available in SICStus (**not in ECLiPSe**), that uses the same propagation technique of all_distinct.

Global Constraints: Assignment

Example:

```
T1::1..2, T2::1..2, T3::1..3, T4::2..5, T5:: 3..5,
R1::1..3, R2::1..4, R3::3..5, R4::4..5, R5:: 4..5,
assignment([T1,T2,T3,T4,T5], [R1,R2,R3,R4,R5]).
```





maximal matching

$T_1 = 1$

$T_2 = 2$

$T_3 = 3$

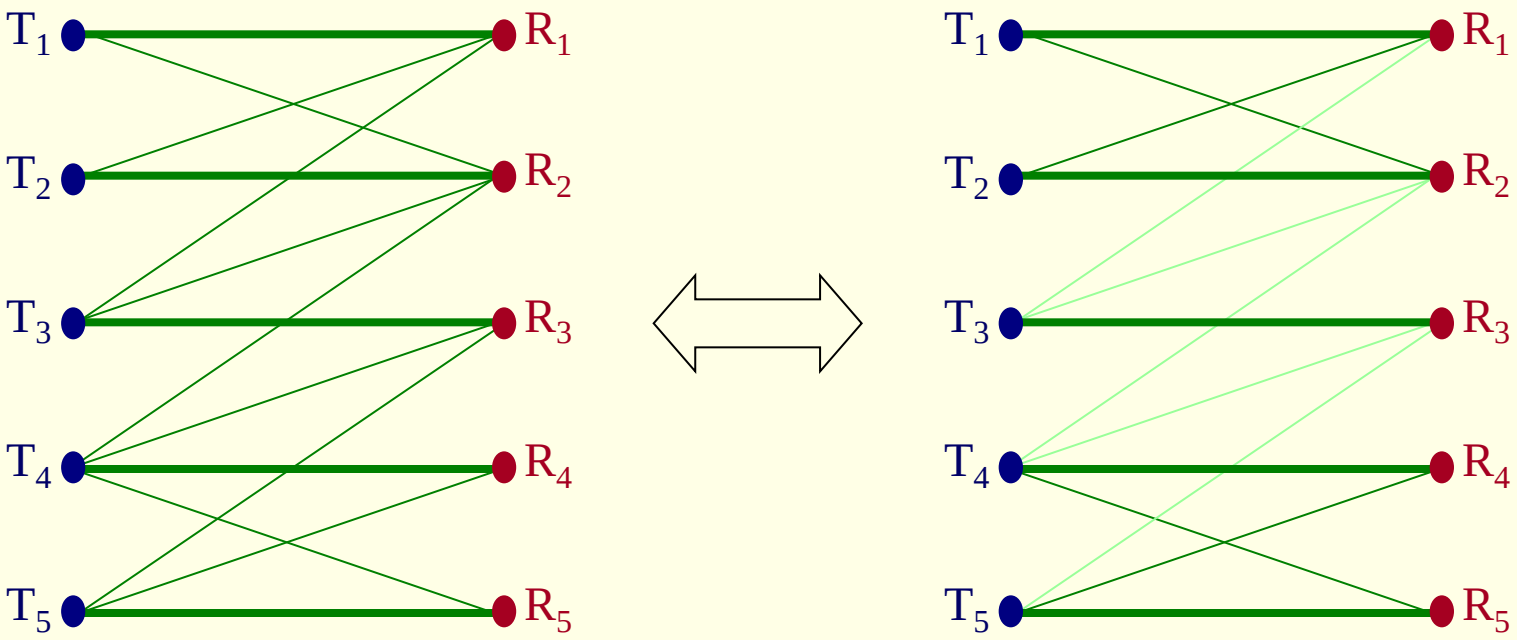
$T_4 = 4$

$T_5 = 5$

Global Constraints: Assignment

Since the assignment constraint imposes a maximal matching in the bipartite graph of tasks T_i and resources R_j, the same filtering techniques of the all_distinct constraint can be used. Hence the initial domains are filtered to

```
T1:1..2, T2::1..2, T3::  3 , T4::4..5, T5:: 4..5.
R1:1..2, R2::1..2, R3::  3 , R4::4..5, R5:: 4..5.
```



Global Constraints: Assignment

An *assignment([R1,...Rn], [T1,...Tn])* constraint can be implemented in ECLiPSe with the same declarative meaning (although not reasoning globally), using **reification**/equivalence:

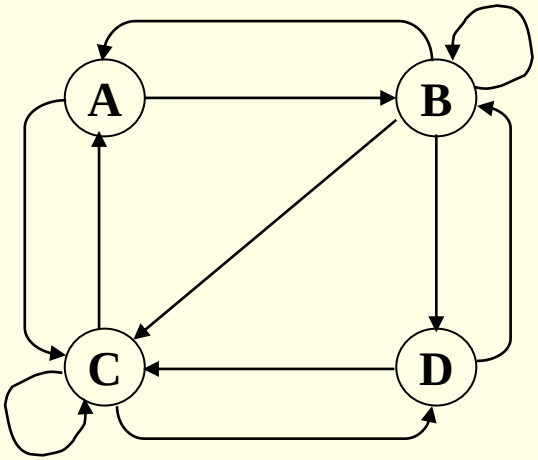
```
 R1 #= 1 #<=> T1 #= 1,
 R1 #= 2 #<=> T2 #= 1,
 R1 #= 3 #<=> T3 #= 1,
...
 R1 #= n #<=> Tn #= 1,
 R2 #= 1 #<=> T1 #= 2,
 R2 #= 2 #<=> T2 #= 2,
...
 R2 #= n #<=> Tn #= 2,
....
 Rn #= 1 #<=> T1 #= n,
...
 Rn #= n #<=> Tn #= n,
```

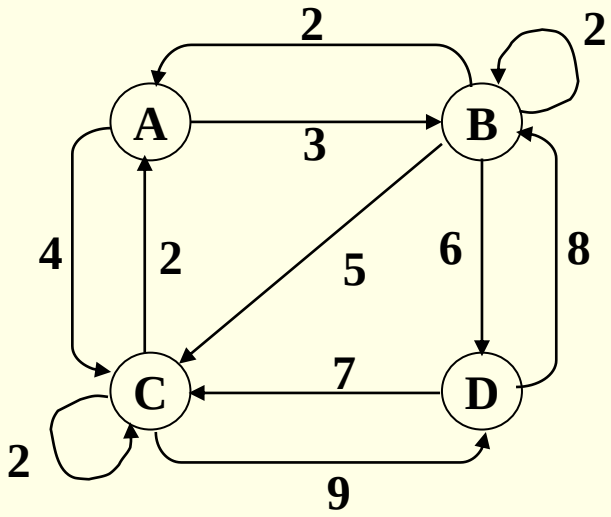
Global Constraints: Circuit

- 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.



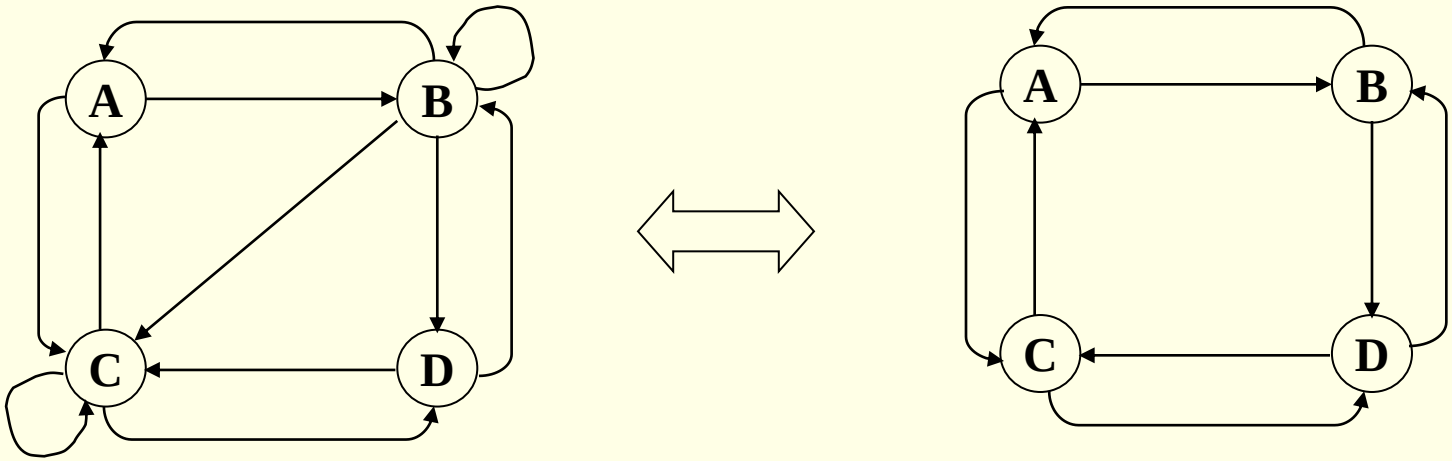


- The arcs may even be labelled by “features” of the precedences, namely transition times.

- This is a situation typical of several problems of the **travelling salesman** type.

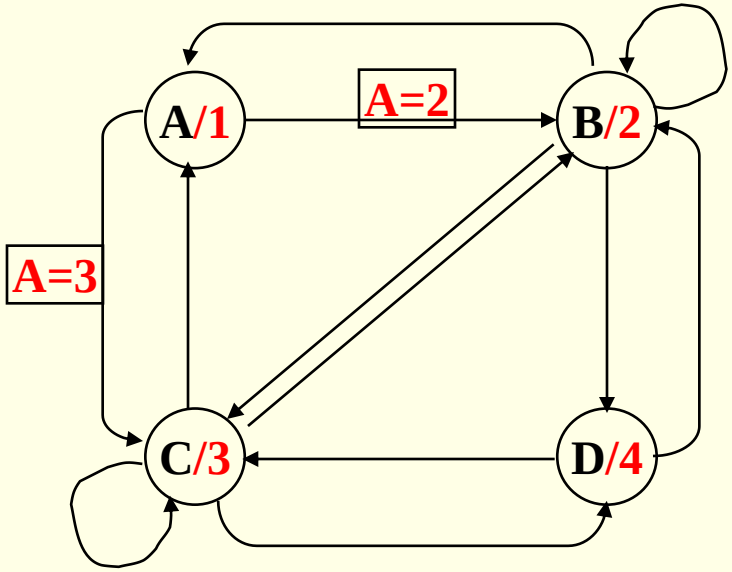
Global Constraints: Circuit

- **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 **A->B->D->C->A** and **A->C->D->B->A**. Certain arcs (e.g. **B->C, B->B**, ...), may not belong to any hamiltonian circuit and can 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/1, available in SICStus.
- This constraint is applicable to a list of domain variables, where the domain of each corresponds to the arcs connecting that variable to other variables, denoted by the order in which they appear in the list.



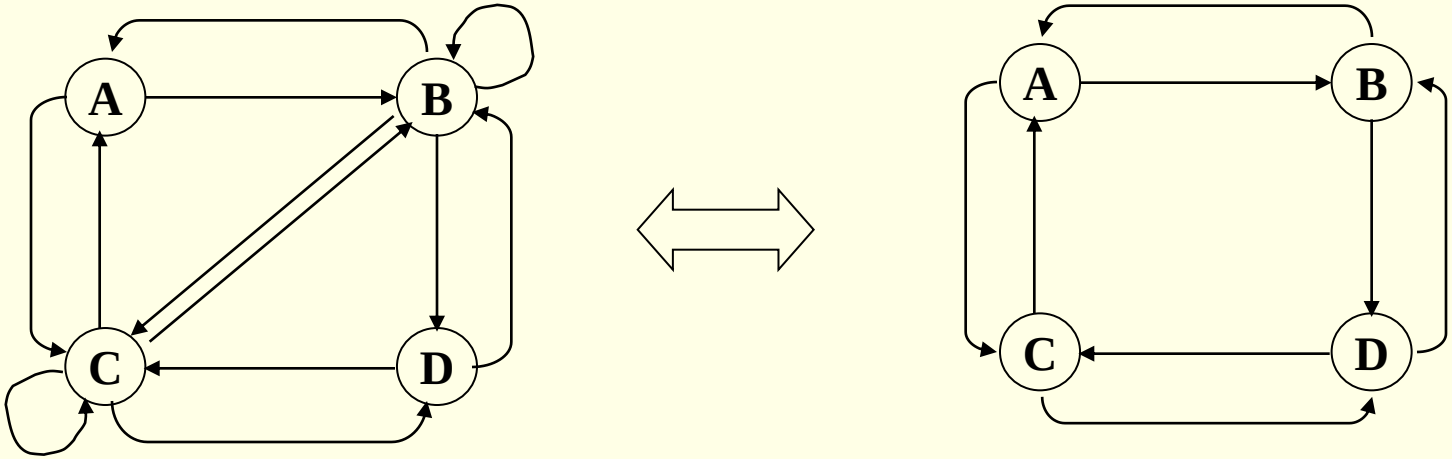
For example:

```
A in 2..3,   B in 1..4,
C in 1..4,   D in 2..3,
circuit([A,B,C,D]).
```

Global Constraints: Circuit

- Global constraint circuit/1 incrementally achieves the pruning of the arcs not in any hamiltonian circuit. For example, posting

```
domain([A,D],2,3), B in 1..4, C in 1..4,
circuit([A,B,C,D]).
```



The following prunning is achieved

```
A in 2..3, B in 1,2,3,4, C in 1,2,3,4, D in 2..3,
```

since the possible solutions are

```
[A,B,C,D] = [2,4,1,3] and [A,B,C,D] = [3,1,4,2]
```

Global Constraints: Element

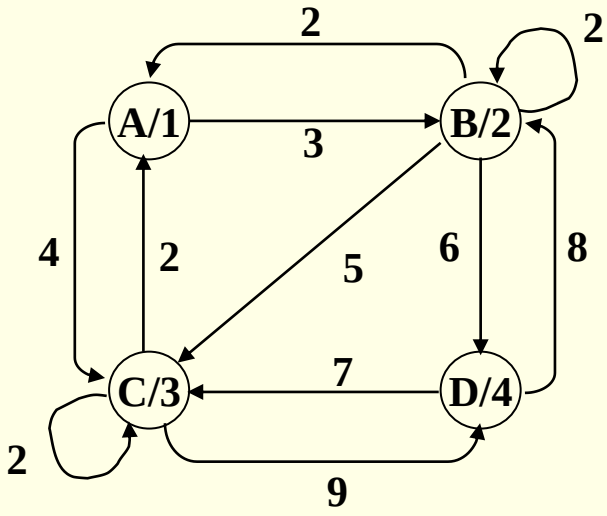
Often a variable not only has its values constrained by the values of other variables, but it is actually defined **conditionally** in function of these values.

For example, the value **X** from the arc that leaves node **A**, depends on the arc chosen:

if **A = 2** then **X = 3,**

if **A = 3** then **X = 4**;

otherwise **X = undefined**



The disjunction implicit in this definition raises, as well known, problems of efficiency to constraint propagation.

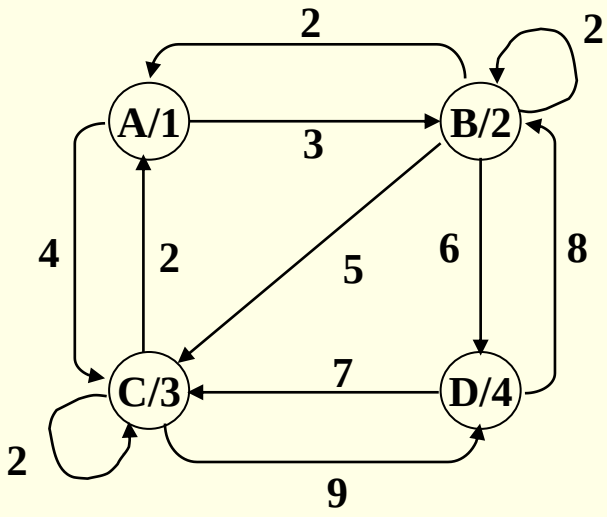
Global Constraints: Element

In fact, the value of **X** may only be known upon labelling of variable **A**. Until then, a naïf handling of this type of conditional constraint would infer very little from it.

if **A = 2** then **X = 3,**

if **A = 3** then **X = 4**;

otherwise **X = undefined**



However, if other problem constraints impose, for example, **X < 4**, an efficient handling of this constraint would impose

not only **X = 3** but also **A = 2**.

Global Constraints: Element

- The efficient handling of this type of disjunctions is the goal of global constraint **element/3**, available in SICStus, ECLiPSe, and CHIP.

$$\textbf{element(X, [}V_1,V_2,...,V_n\textbf{], V)}$$

- In this constraint, **X** is a variable with domain **1..n**, and both **V** and the $\mathbf{V_i s}$ are either finite domain constraints or constants. The semantics of the constraint can be expressed as the equivalence

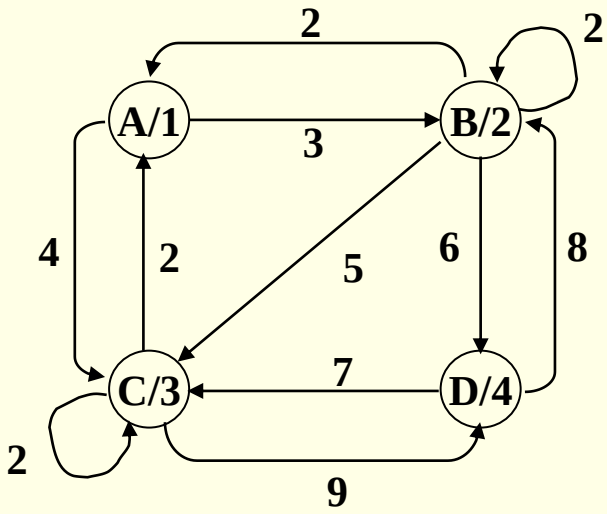
$$X = i \Leftrightarrow V = V_i$$

- From a propagation viewpoint, this constraint imposes arc consistency in **X** and bounds consistency in **V**. It is particularly optimised for when all $\mathbf{V_i s}$ are ground (actually ECLiPSe requires this).

Global Constraints: Element & Circuit

- Global constraints may be used together. In particular, constraints element and circuit may implement the travelling salesman: For some graph, determine an hamiltonian circuit whose length does not exceed Max (say 20).

```
circ([A,B,C,D], Max, Cost):-
   A in 2..3, B in 1..4,
   C in {1}\/{3,4}, D in 2..3,
   circuit([A,B,C,D]),
   element(A,[_,3,4,_],Ca),
   element(B,[2,2,5,6],Cb),
   element(C,[2,_,2,9],Cc),
   element(D,[_,8,7,_],Cd),
   Cost #= Ca+Cb+Cc+Cd,
   Cost #=< Max,
   labeling([],[A,B,C,D]).
```



Global Constraints: Global Cardinality

- Many scheduling and timetabling problems, have quantitative requirements of the type

in these N "slots" M must be of type T

- This type of constraints may be formulated with a cardinality constraint. In some systems, these cardinality constraints are given as built-in, or may be implemented through reified constraints.
- In particular, in ECLiPSe the constraint occurrences/3 may be used to count elements in a list, which replaces some uses of the cardinality constraints (*see ahead).*
- However, cardinality may be further generalized and more efficiently propagated if considered **globally**.

Global Constraints: Global Cardinality

- 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 holiday (h) or stay in reserve (r).

- To model this problem, let us consider a list L, whose variables L_i corresponding to the 7 people available, may take values in domain {m, a, n, h, r} (or {1, 2, 3, 4, 5} in languages like ECLiPSe that require domains to range over integers).

- Both in ECLiPSe and in CHIP this complex constraint may be decomposed in several cardinality constraints.

Global Constraints: Global Cardinality

ECLiPSe:

```
length(L,7), L :: 1..5,
occurrences(1,L,N1), N1 :: [1,2],        % m/1: 1 or 2
occurrences(2,L,N2), N2 :: [1,2],        % a/2: 1 or 2
occurrences(3,L,1),                      % n/3: 1 only
occurrences(4,L,N4), N4 :: 0..2,         % h/4: 0 to 2
occurrences(5,L,N5), N5 :: 0..2          % r/5: 0 to 2
```

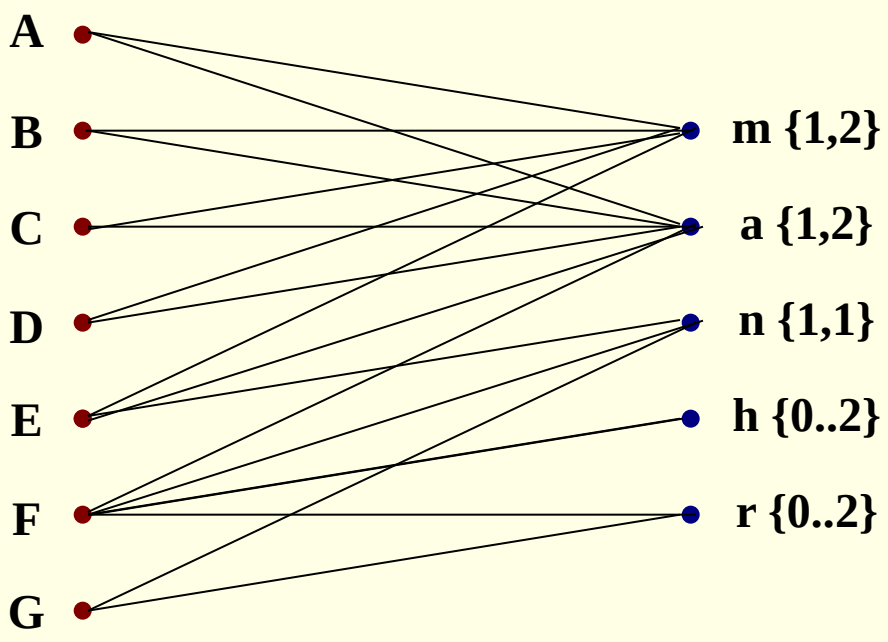
CHIP:

```
among([1,2],L,_,[1]),                    % m/1: 1 or 2
among([1,2],L,_,[2]),                    % a/2: 1 or 2
among(  1  ,L,_,[3]),                    % n/3: 1 only
among([0,2],L,_,[4]),                    % h/4: 0 to 2
among([0,2],L,_,[5]),                    % r/5: 0 to 2
```

Global Constraints: Global Cardinality

- Nevertheless, the separate, or local, handling of each of these constraints, does not detect all the pruning opportunities for the variables domains. Take the following example:

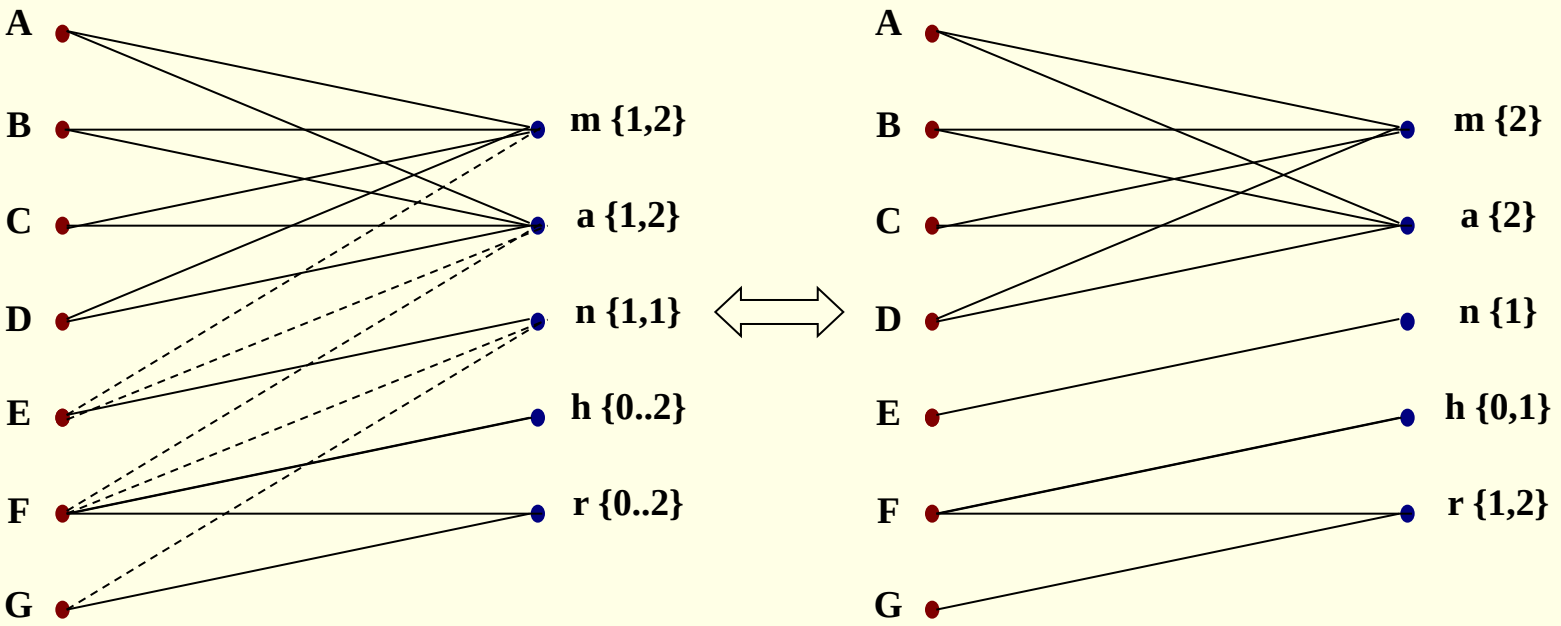
A,B,C,D::{m,a}, E::{m,a,n}, F::{a,n,h,r}, G ::{n,r}



Global Constraints: Global Cardinality

A,B,C,D::{m,a}, E::{m,a,n}, F::{a,n,h,r}, G ::{n,r}

- A, B, C and D may only take values m and a. Since these may only be attributed to 4 people, no one else, namely E or F, may take these values m and a.
- Since E may now only take value n, which must be taken by a single person, no one else (e.g. F or G) may take value n.



Global Constraints: Global Cardinality

- 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.
- A global cardinality constraint **gcc*/4*** (not available in ECLiPSe nor SICStus),
  - constrains a list of **k** variables **$X = [X_1, ..., X_k]$** ,
  - taking values in the domain (with m values) **$V = [v_1, ..., v_m]$**,
  - such that each of the $\mathbf{v_i}$ values must be assigned to between $\mathbf{L_i}$ and $\mathbf{M_i}$ variables.
- Then, m ECLiPSe constraints

```
...
occurrences(v_i,X,Ni), Ni :: L_i..M_i
...
```

  could be replaced by constraint

```
gcc([X_1....,X_k],[v_1,...,v_m],[L_1,...,L_m],[M_1,...,M_m])
```

The Cardinality Operator

- A related constraint is the Cardinality operator
- In ECLiPSe it is present in the fd library as #(?Min, ?Cstrs, ?Max), a meta constraint known in the literature as the cardinality operator. Cstrs is a list of constraint expressions and this operator states that at least Min and at most Max out of them are valid.
- E.g. `#(2, [X#>0, Z#>=0, X+Y#=7, 2*Z-Y#>X], 3)`
- Example implementation (for triples) using reification:

```
#(Min, [C1,C2,C3], Max):-
    B1 isd C1,
    B2 isd C2,
    B3 isd C3,
    occurences(1, [B1,B2,B3], NValid),
    NValid :: Min..Max.
```

Global Constraints for Scheduling

- There are important constraints in a variety of scheduling problems (job-shop, timetabling, etc...).

Some important constraints are

- Precedence : one task executes before the other
- Non-overlapping: Two tasks should not execute at the same time (e.g. they share the same resource).
- Cumulating: The number of tasks that execute at the same time must not exceed a certain number (e.g. the number of resources, such as machines or people, that must be dedicated to one of these tasks).

Precedence

- In general, each task i is modelled by its starting time T_i and its duration D_i, which may both be either finite domain variables or fixed to constant values. Hence, the precedence of task i with respect to task j may be expressed simply as

```
before(Ti, Di, Tj) :-
    Ti + Di #=< Tj.
```

- In practice, such specification of precedence uses bounds consistency.

Non-overlapping

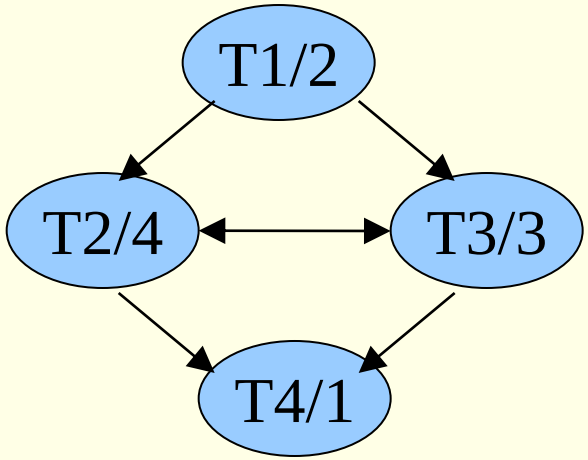
- The non overlapping of tasks is equivalent to the disjunction of two precedence constraints. Either
  - Task i executes before Task j; or
  - Task j executes before Task i.
- Many different possibilities exist to implement this disjunction, namely, by means of:
  1. Alternative clauses;
  2. Least commitment;
  3. Constructive Disjunction;
  4. Specialised global constraints

Non-overlapping

Example

Let us consider a project with the four tasks illustrated in the graph, showing precedences between them, as well as mutual exclusion (↔). The durations are shown in the nodes.

The goal is to schedule the tasks so that T4 ends no later than time 10.




```
project(T):-
   T = [T1,T2,T3,T4], T :: 1..10,
   before(T1, 2, T2),   before(T1, 2, T3),
   before(T2, 4, T4),   before(T3, 3, T4),
   before(T4, 1, 10),
   no_overlap(T2,4, T3,3).
```

Non-overlapping

Alternative clauses

- In a Constraint Logic Programming system, the disjunction of constraints may be implemented with a Logic Programming style (*a la* Prolog):

```
no_overlap(T1, D1, T2,  _):-
   before(T1, D1, T2).
no_overlap(T1,  _, T2, D2):-
   before(T2, D2, T1).
```

This implementation always tries first to schedule task T1 before T2, and this may be either impossible or undesirable in a global context. This greatest commitment will usually show poor efficiency (namely in large and complex problems).

Non-overlapping

Least Commitment

- The opposite least commitment implementation may be made through the cardinality constraint

```
no_overlap(T1,D1,T2,D2):-
   #(1, [T1 + D1 #=< T2, T2 + D2 #=< T1], 1).
```

or directly, with propositional constraints

```
no_overlap(T1,D1,T2,D2):-
   (T1 + D1 #=< T2) #\/ (T2 + D2 #=< T1).
```

or even with reified constraints

```
no_overlap(T1,D1,T2,D2):-
   (T1 + D1 #=< T2) #<=> B1,
   (T2 + D2 #=< T1) #<=> B2,
    B1 + B2 #= 1.
```

- When enumeration starts, if eventually one of the constraints is disentailed, the other is enforced.

Non-overlapping

Constructive Disjunction

- With constructive disjunction, the values that are not part of any solution may be removed, even before a commitment is made regarding which of the tasks is executed first. Its implementation may be done with the appropriate propagation.
- E.g, for **`no_overlap(T1,4, T2,5)`** with **`T1,T2 :: 0..5`**
  Tasks domains can be reduced to

```
T1 :: [0,1,5]

T2 :: [0,4,5]
```

Redundant Constraints

Redundancy

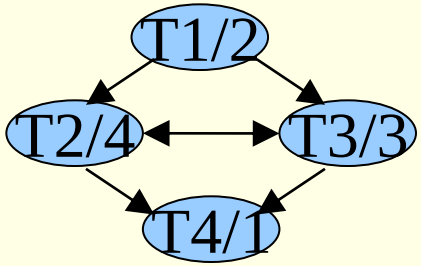
- Often, not even the specification with a global constraint is able to infer all the pruning that should have been made.
- This is of course a common situation, as the constraint solvers are incomplete.
- In many situations it is possible to formulate constraints which can be deduced from the initial ones, i.e. that should not make any difference in the set of results obtained.
- However, if properly thought of, they may provide a precious support to the constraint solver, enabling a degree of pruning that the solver would not be able to make otherwise .

Redundant Constraints

Redundancy

- Hence the name of redundant constraints. Careful use of such constraints may greatly help to increase the efficiency of constraint solving.
- Of course, it is up to the user to understand the working of the solver, and its pitfalls, in order to formulate adequate redundant constraints.
- In our previous example, since tasks 2 and 3 are mutually exclusive, T4 must wait at least the total duration of both, after the first starts.
- Hence, task T4 may never start before

```
min(min(T2),min(T3))+D2+D3
```



Non-overlapping

Results: Redundant constraints / Least Commitment

- Adding the redundant constraints to the formulation of least commitment, the tasks T1 and T4 become well delimited, although as expected, no significant cuts are obtained in tasks T2 and T3.

```
|? T in 1..10, project(T).
  T1 = 1,
  T2 in 3 .. 6,
  T3 in 3 .. 7,
  T4 = 10      ? ;

    no
```

```
|? T in 1..11, project(T).
  T1 in 1 .. 2,
  T2 in 3 .. 7,
  T3 in 3 .. 8,
  T4 in 10 .. 11        ? ;

    no
```

Non-overlapping

Results: Redundant constraints / Constructive Disjunction

- Adding the redundant constraints to the formulation of constructive disjunction, not only T1 and T4 become well delimited, but also T2 and T3 are adequately pruned.

```
|? T in 1..10, project(T).
  T1 = 1,
  T2 in {3}\/{6},
  T3 in {3}\/{7},
  T4 = 10      ? ;

    no
```

```
|? T in 1..11, project(T).
  T1 in 1 .. 2,
  T2 in(3..4) \/ (6..7),
  T3 in(3..4) \/ (7..8),
  T4 in 10 .. 11        ? ;

    no
```

Global Constraints: cumulative

- A general global constraint for tasks with starting times in T and durations in D

`cumulative(T,D,R,L)`

- For a set of tasks Ti, with durations Di and that use an amount Ri of some resource, this constraint guarantees that at no time there are more than L units of the resource being used by the tasks.

- If each task uses 1 unit of a resource for which there is only that unit available, we have

`cumulative(T,D,[1,1,...1],1)`

Global Constraints: cumulative

- The global constraint **cumulative/4** allows not only to reason efficiently and globally about the tasks, but also to specify in a compact way this type of constraints, whose decomposition in simpler constraints would be very cumbersome.

- Its semantics is as follows

Let

$\mathbf{a = min_i(T_i)}$;

$\mathbf{b = max_i(T_i+D_i)}$;

$\mathbf{S_{i,k} = R_i}$ if $\mathbf{T_i =< k < T_i+D_i}$ or **0** otherwise.

Then

$$\mathbf{cumulative(T,D,R,L)} \Leftrightarrow \forall_{k \in [a,b]} \sum_i \mathbf{S_{i,k} \leq L}$$

Global Constraints: cumulative

- This global constraint, **cumulative/4**, was initially introduced in the CHIP system (1994) aiming at the efficient execution of a number of problems namely,

    1. Scheduling of disjoint tasks
    2. Scheduling of tasks with resource limitations
    3. Placement problems

Global Constraints: cumulative - Scheduling

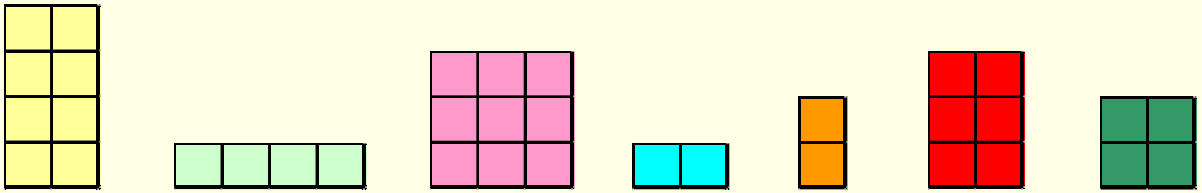
Example:

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

D = [2 , 4 , 3 , 2 , 1 , 2 , 2] ; R = [4 , 1 , 3 , 1 , 2 , 3 , 2]

Find whether the tasks may all be finished in a given due time *Tmax*, assuming there are *Rmax* resources (e.g. Workers) available at all times (*cumulative(T,D,R,Rmax)*, and $max_i(T_i+D_i) \leq Tmax$).

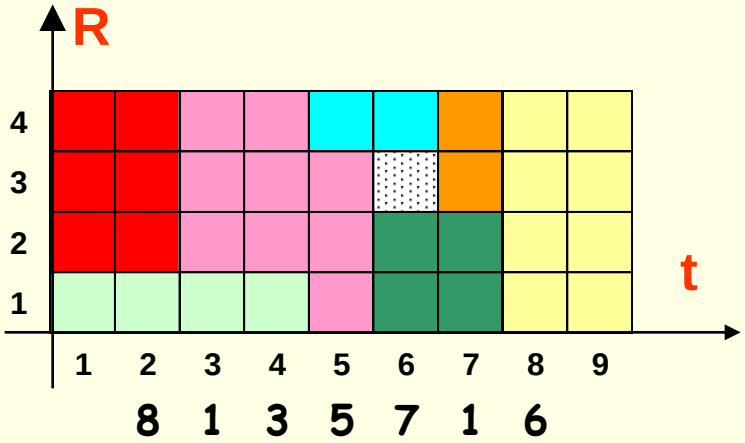
Graphically, the tasks can be viewed as

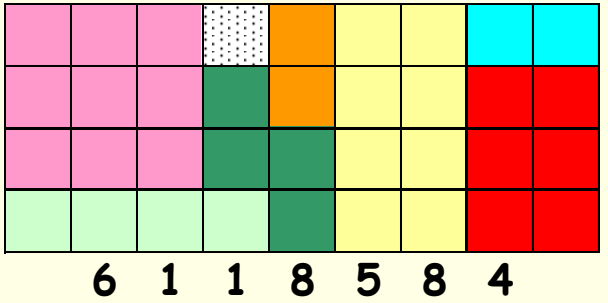


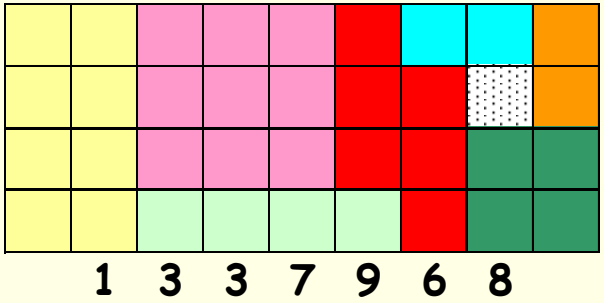
Global Constraints: cumulative - Scheduling

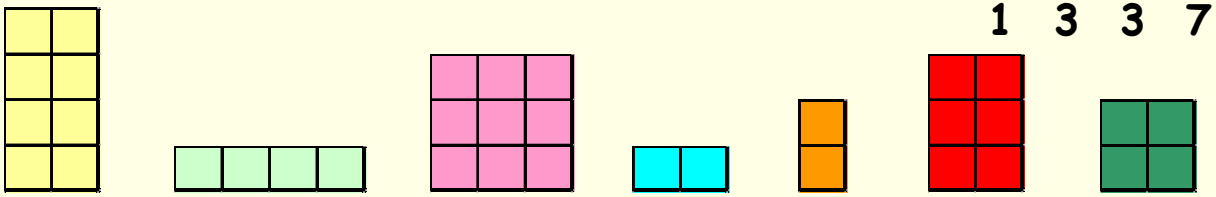
Results

With Tmax = 9 and Rmax = 4 a number of answers are obtained, namely (numbers represent *T*, the list of starting times of the 7 tasks)





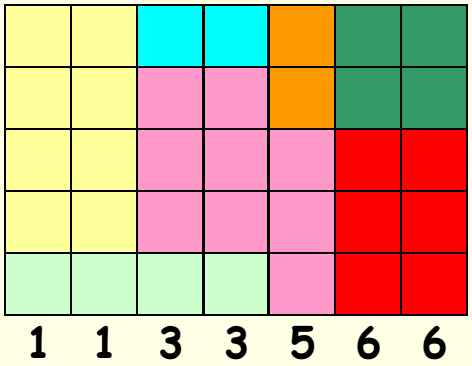


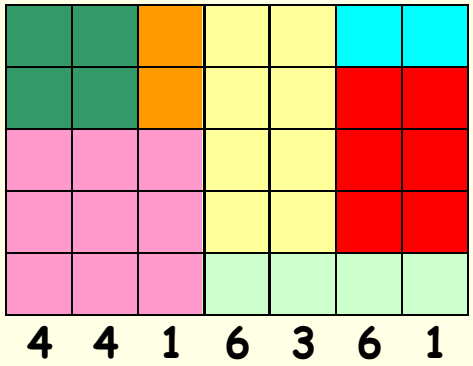


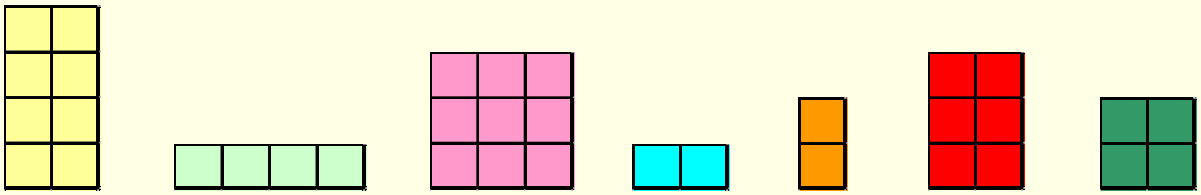
Global Constraints: cumulative - Scheduling

Results

With Tmax = 7 and Rmax = 5 (in this case, no resources may be spared), a number of answers are still obtained, such as



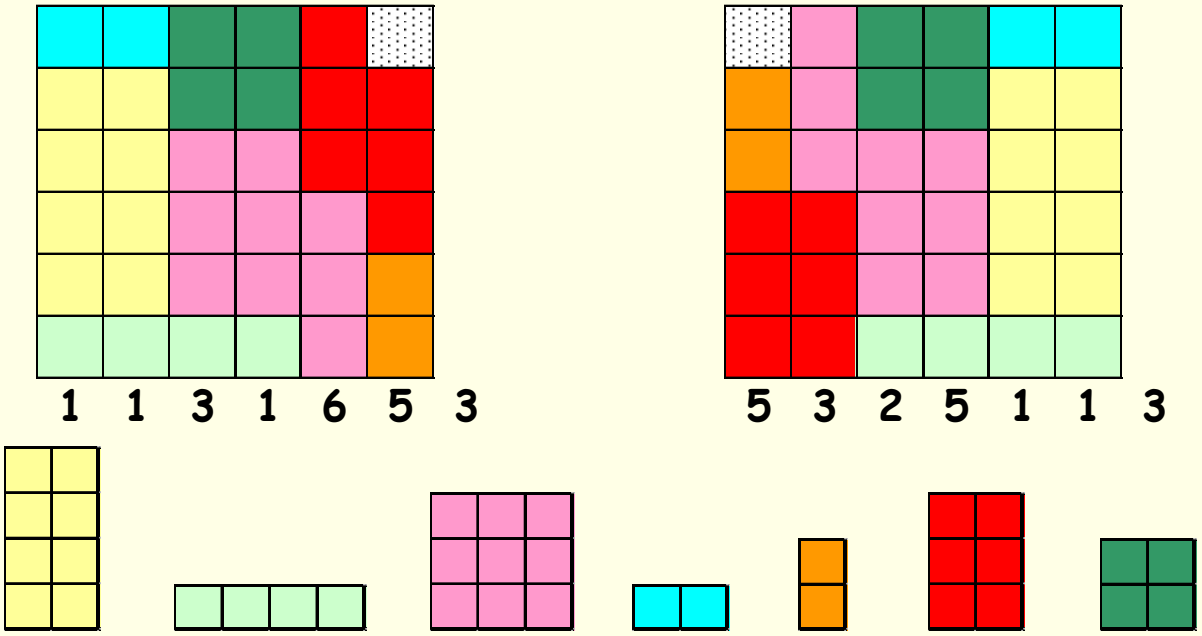




Global Constraints: cumulative - Scheduling

Results

With Tmax = 6 and Rmax = 6 (in this case, one of the 6 workers may rest for an hour), still a number of answers are obtained, namely



Question: What about Tmax = 5 and Rmax = 7 ?

Global Constraints: cumulative - Scheduling

- 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 accommodated 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 and resources were constants Kd_i e Kr_i , the durations *Di* and resources *Ri* of flexible tasks must be constrained by

$$\mathbf{Di * Ri \#= Kd_i * Kr_i}$$

Global Constraints: cumulative - Scheduling

- The program below is similar to the previous, but imposes flexibility on tasks with predicate **constrain_tasks/4.** Of course, since both the durations and resources are now variables, labelling must be made in (one of) such variables.

```
plan2(Tmax,Rmax, T, D, R):-
    T  = [T1,T2,T3,T4,T5,T6,T7],
    T :: 1..15,
    D  = [D1,D2,D3,D4,D5,D6,D7],
    Dc = [ 2, 4, 3, 2, 1, 2, 2],
    R  = [R1,R2,R3,R4,R5,R6,R7],
    Rc = [ 4, 1, 3, 1, 2, 3, 2],
    constrain_tasks(D,R,Dc,Rc),
    cumulative(T,D,R,Rmax),
    latest(T,D,Tmax),
    append(T,D,V),
    labeling(V).
```

Global Constraints: cumulative - Scheduling

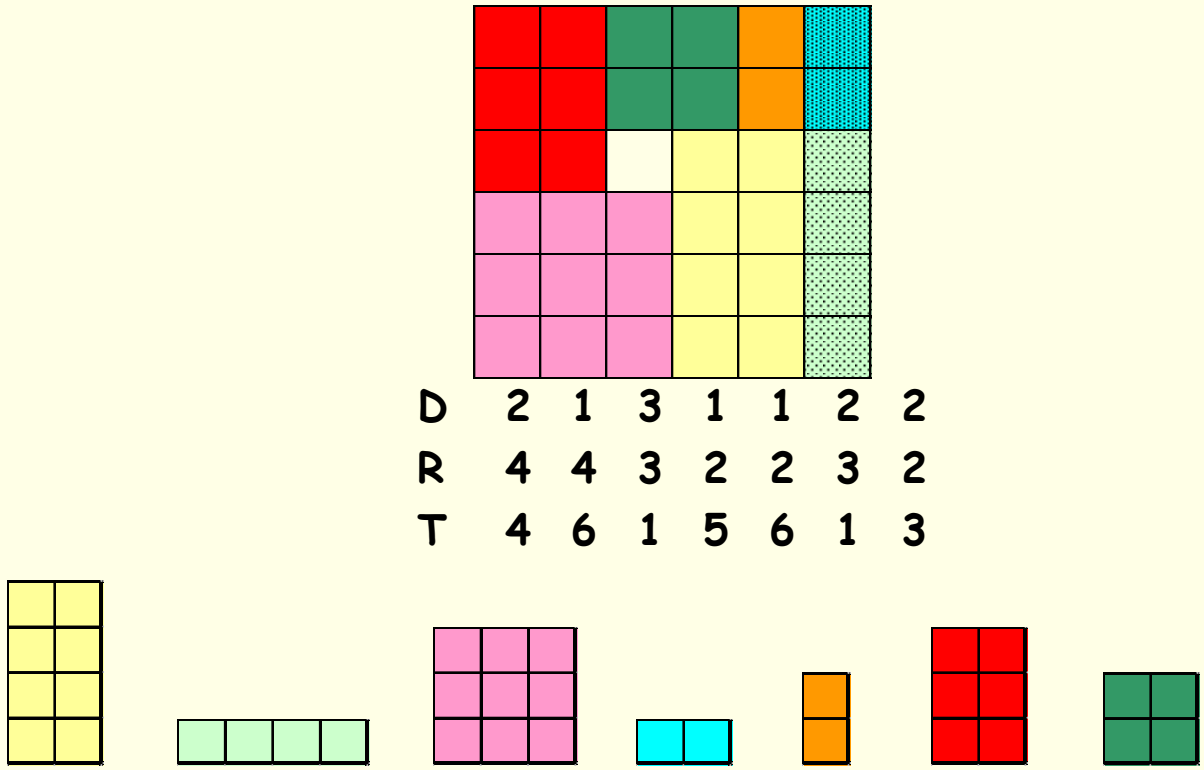
- Predicate **constrain_tasks/4** is implemented as shown below. Variables in D and R are assigned initial domains 1..9 , and for each task, the constraint specifying flexibility is imposed.

```
constrain_tasks([],[],[],[]).
constrain_tasks([D1|Dt1],[R1|Rt1],[D2|Dt2],[R2|Rt2]):-
        [D1,R1] :: 1..9,
        D1 * R1 #= D2 * R2,
        constrain_tasks(Dt1,Rt1,Dt2,Rt2).
```

Global Constraints: cumulative - Scheduling

Results

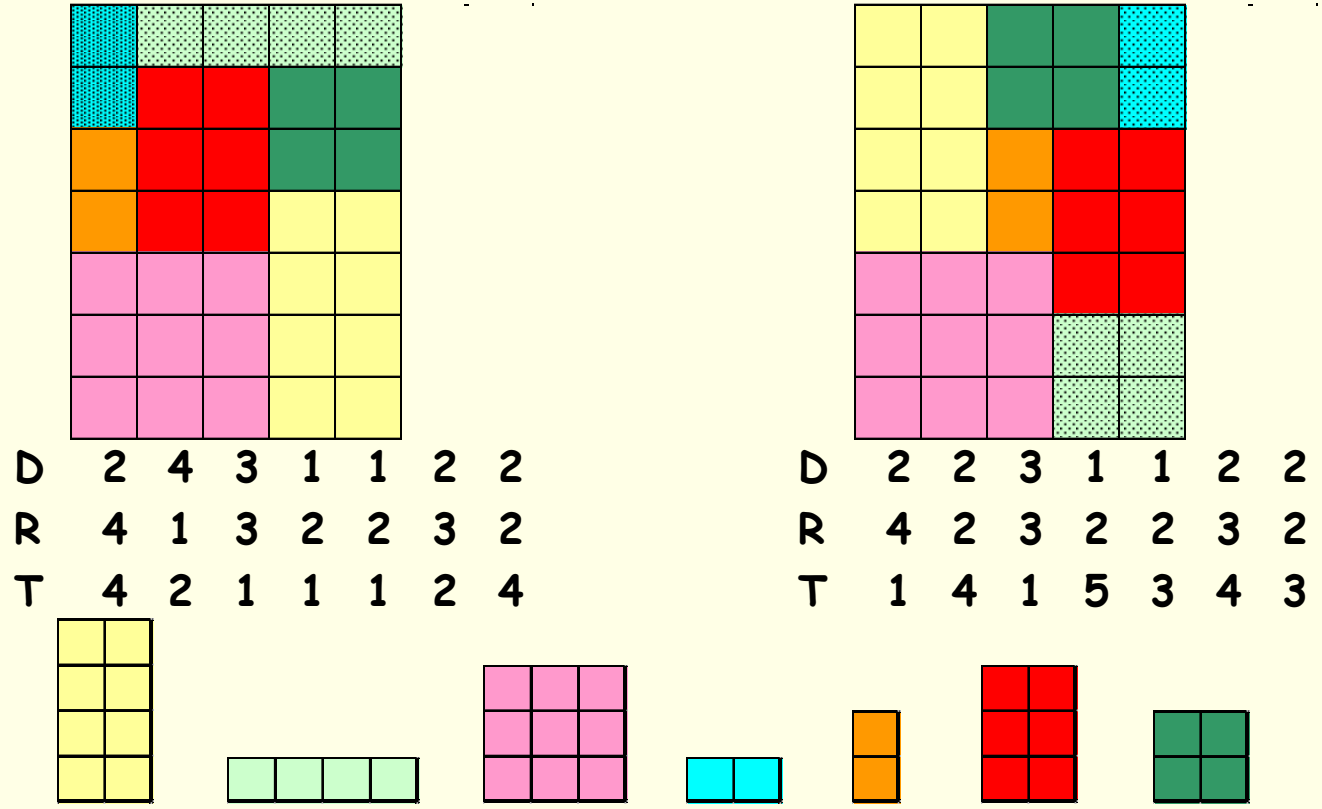
With Tmax = 6 and Rmax = 6 (1 spare hour*worker) new solutions are obtained, such as



Global Constraints: cumulative - Scheduling

Results

With Tmax = 5 and Rmax = 7 (previously impossible) there are now several solutions. (Notice the “deeper” transformation in task 2, from (4*1 $\Rightarrow$ 2*2), in addition to a “rotation”).



Global Constraints: cumulative - Scheduling

ECLiPSe

cumulative/4 is available in ECLiPSe in 3 different libraries: cumulative, edge_finder, and edge_finder3

cumulative:cumulative(+StartTimes, +Durations, +Resources, ++ResourceLimit)
is the weakest

edge_finder:cumulative/4 is quadratic

edge_finder3:cumulative/4 is cubic but computationally heavier

edge_finder and edge_finder3 also provide, for flexible tasks:
cumulative(+StartTimes, +Durations, +Resources, +Areas, ++ResourceLimit)
where even the Areas can be FD variables

Global Constraints: cumulative - Scheduling

SICStus

cumulative(+Tasks)

cumulative(+Tasks, +Options)

Tasks is a list of terms *task(S_i,D_i,E_i,R_i,T_i)*, where:

S_i – Start time; **D_i** – Duration; **E_i** – End Time

R_i – Resource consumption; **T_i** – Task identifier;

(all may be domain variables)

Options may contain {limit(L), precedences(Ps), global(Boolean)}

For *m* machines there's **cumulatives(+Tasks, +Machines [,+Options]):**

T_i is replaced by **M_i** (machine identifier);

R_i may be negative (production);

Machines are terms *machine(M_j, L_j)* (identifier and limit, integers)

Options in {*bound(B), prune(P), generalization(Bool), task_intervals(Bool)*}

See manual…

Global Constraints: cumulative - 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 a machine of a certain type. The number of machines of each type is limited.

Global Constraints: cumulative - Job-Shop

- An instance of the 10*10 job-shop is shown in the following table, where Y denotes the Y-th task of job X, to be executed in machine Z, with duration D

		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

Global Constraints: cumulative - Job-Shop

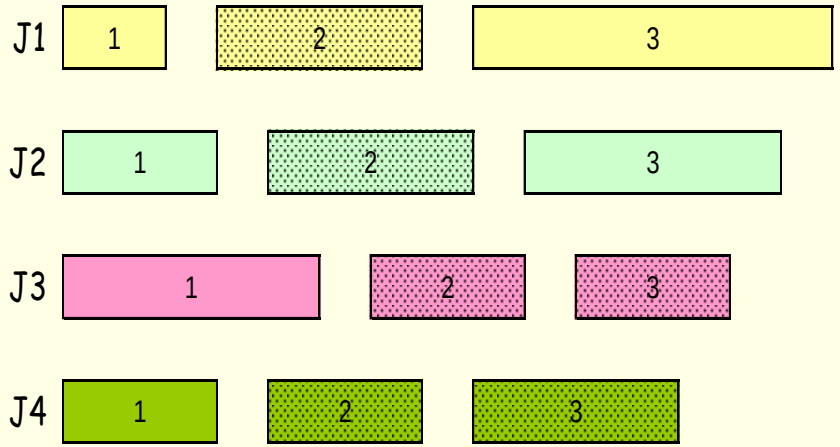
History

- This instance was proposed in the book ‘Industrial Scheduling’ in 1963.
- 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).

Global Constraints: cumulative - Job-Shop

- A simpler instance of the problem is given in the table below (with the corresponding graphic representation).
- Notice that in this instance it is assumed that each task requires one unit of the resource shown, and that there are 2 units of resource 1 and other 2 units of resource 2.

		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



Global Constraints: cumulative - Job-Shop

This instance of the problem may be easily solved by the following ECLiPSe program:

```
jobs([J1,J2,J3,J4]):-

% definition of the jobs
  J1 = [S111,S122,S131]-[2,4,7]-[1,2,1],
  J2 = [S211,S222,S231]-[3,4,5]-[1,2,1],
  J3 = [S311,S322,S332]-[5,3,3]-[1,2,2],
  J4 = [S411,S422,S432]-[3,3,4]-[1,2,2],

% domain declarations
  [S111,S122,S131] :: 0..15,
  [S211,S222,S231] :: 0..15,
  [S311,S322,S332] :: 0..15,
  [S411,S422,S432] :: 0..15,

% precedence constraints

% resource limitation constraints

% constraints on the end of the jobs

% labelling of the tasks starting times
```

Global Constraints: cumulative - Job-Shop

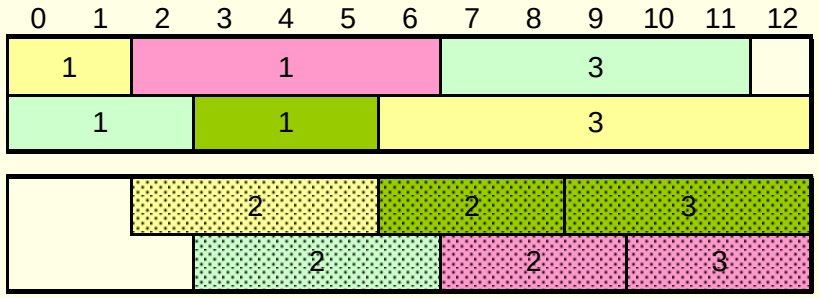
The constraints are as follows:

```
% precedence constraints
    S122#>=S111+2, S131#>=S122+4,
    S222#>=S211+3, S231#>=S222+4,
    S322#>=S311+5, S332#>=S322+3,
    S422#>=S411+3, S432#>=S422+3,
% resource limitation constraints
    cumulative([S111,S131,S211,S231,S311,S411],
               [2,7,3,5,5,3],  [1,1,1,1,1,1],2),
    cumulative([S122,S222,S322,S332,S422,S432],
               [4,4,3,3,3,4],  [1,1,1,1,1,1],2),
% constraints on the end of the jobs
    E #>= S131+7, E #>= S231+5,
    E #>= S332+3, E #>= S432+4,
    E #=< 13,
% labelling of the tasks starting times
    labeling([S111, S122, S131, S211, S222, S231,
              S311, S322, S332, S411, S422, S432]).
```

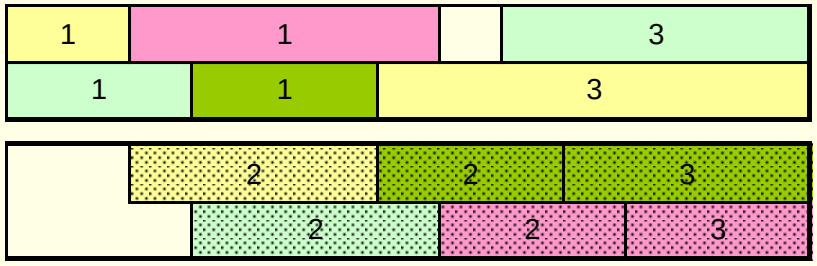
Global Constraints: cumulative - Job-Shop

The possible results, with termination no later than 13, are the following :

```
| ?- jobs(J).
J=[[0,2, 6]-[2,4,7]-[1,2,1],[0,3,7]-[3,4,5]-[1,2,1],
   [2,7,10]-[5,3,3]-[1,2,2],[3,6,9]-[3,3,4]-[1,2,2]]?;
```




```
J=[[0,2, 6]-[2,4,7]-[1,2,1],[0,3,8]-[3,4,5]-[1,2,1],
   [2,7,10]-[5,3,3]-[1,2,2],[3,6,9]-[3,3,4]-[1,2,2]]?;
```




```
no
```

Global Constraints: cumulative - Placement

- 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, where a number of smaller pieces should be taken from large boards:
  - Similar problem in the textile context;
  - 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.

Global Constraints: cumulative - Placement

- 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).

- Hence, the solutions used before should apparently also be used for this new kind of problems, with the above adaptations.

Global Constraints: cumulative - Placement

Example**:**

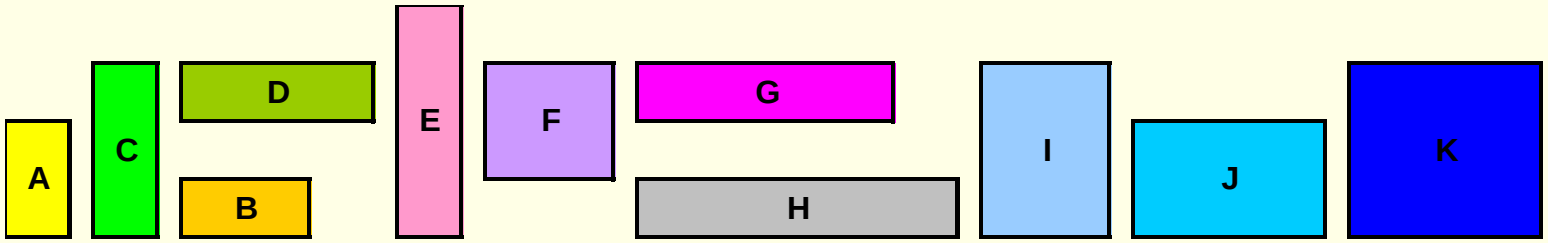
Find the appropriate cuts to be made on a wood board so as to obtain 11 rectangular pieces (A a K).

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



Global Constraints: cumulative - Placement

This problem can be thus specified as below

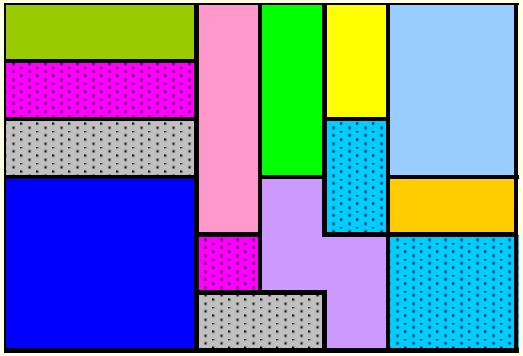
```
  place(Width, Height):-

% rectangles definition
     X = [Ax,Bx,Cx,Dx,Ex,Fx,Gx,Hx,Ix,Jx,Kx],
     W = [ 2, 1, 1, 3, 1, 2, 4, 5, 2, 3, 3],
     H = [ 1, 2, 3, 1, 4, 2, 1, 1, 3, 2, 3],
     X :: 1..Width,

% constraints in X-origins
     maximum(X,W,Width),
     cumulative(X,W,H,Height),

% enumeration of rectangles X-origin
     labeling(X).
```

Global Constraints: cumulative - Placement

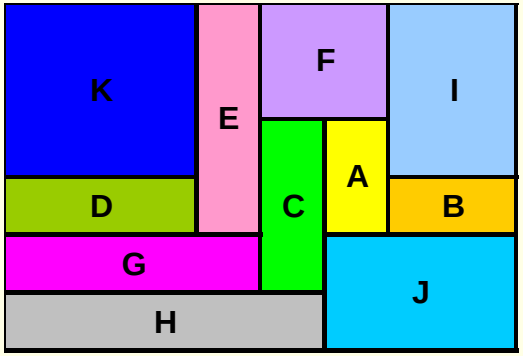
- Unfortunately, the results obtained have not a direct reading. For example, one of the solutions obtained with an 8*6 rectangle is

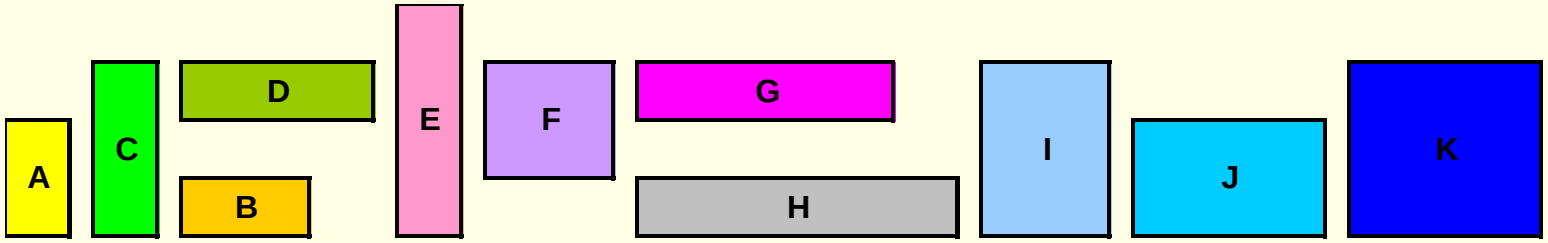
X = [6, 7, 5, 1, 4, 5, 1, 1, 7, 6, 1]

That can be read as (???)



or as





Global Constraints: cumulative - Placement

- 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.

Global Constraints: cumulative - Placement

- Firstly, new Y variables are created

```
place(Width,Height):-
% rectangles definition
  X = [Ax,Bx,Cx,Dx,Ex,Fx,Gx,Hx,Ix,Jx,Kx],
  Y = [Ay,By,Cy,Dy,Ey,Fy,Gy,Hy,Iy,Jy,Ky],
  W = [ 2, 1, 1, 3, 1, 2, 4, 5, 2, 3, 3],
  H = [ 1, 2, 3, 1, 4, 2, 1, 1, 3, 2, 3],
  X :: 1..Width,
  Y :: 1..Height,
% constraints in X- and Y-origins
   ...
% enumeration of rectangles X- and Y origins
  ...
```

Global Constraints: cumulative - Placement

- Secondly, similar constraints (but with a 90º rotation) are imposed on them

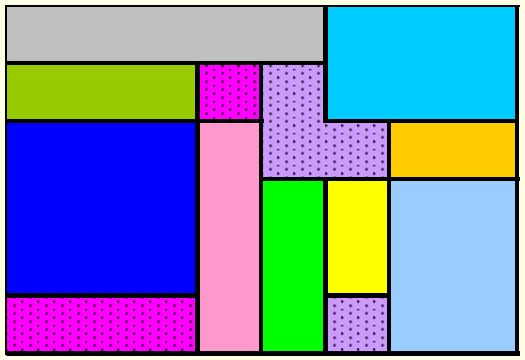
```
 place(Width,Height):-
% rectangles definition
  ...
% constraints in X- and Y-origins
  maximum(X,W,Width),
  cumulative(X,W,H,Height),
  maximum(Y,H,Height),
  cumulative(Y,H,W,Width),
% enumeration of rectangles X- and Y- origins
  labeling(X),
  labeling(Y).
  ...
```

Global Constraints: cumulative - Placement

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

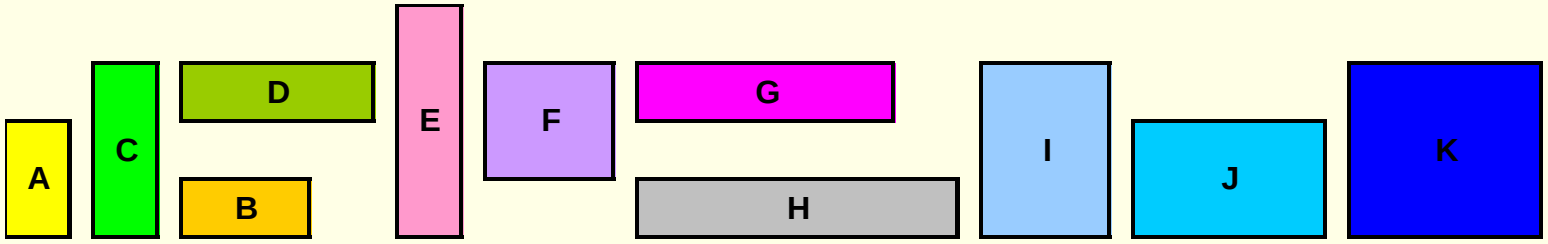
```
(X-Y)   [6-2,7-4,5-1,1-5,4-1,5-4,1-1,1-6,7-1,6-5,1-2]
```

corresponding to



?????

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



Global Constraints: cumulative - Placement

- The non overlapping of the rectangles defined by their X_i and Y_i origins and their widths W_i (X-sizes) and heights H_i (Y sizes) is guaranteed, as long as one of the constraints below is satisfied (for rectangles 1 and 2)

$X_1+W_1 =< X_2$	Rectangle 1 is to the left of 2
$X_2+W_2 =< X_1$	Rectangle 1 is to the right of 2
$Y_1+H_1 =< Y_2$	Rectangle 1 is below 2
$Y_2+H_2 =< Y_1$	Rectangle 1 is above 2

- As explained before, rather than committing to one of these conditions, and change the commitment by backtracking, a better option is to adopt a least commitment approach, for example with some kind of “cardinality” meta-constraint.

Global Constraints: cumulative - Placement

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.
- 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!

Global Constraints: cumulative - Placement

- The program for the placement problem, showing the symmetry on the X and Y axes, is presented below.

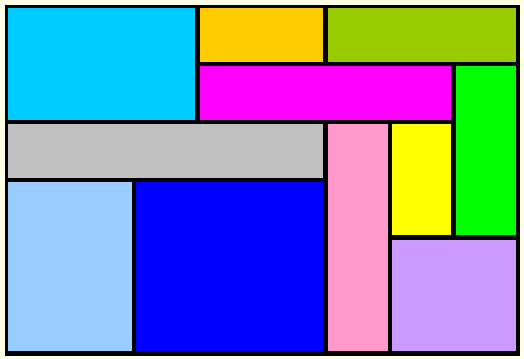
```
place(Width,Height):-
% rectangles definition
   X = [Ax ... Kx],   Y = [Ay ... Ky],
   W :: 1..Width,     Y :: 1..Height,
% constraints on X and Y origins
   maximum(X,W,Width), maximum(Y,H,Height),
% redundant cumulative constraints
   cumulative(Y,H,W,Width), cumulative(X,W,H,Height),
% non overlapping constraints
   none_overlap(X,Y,D,R),
% joint enumeration of X and Y origins
   append(X,Y,Z),
   labeling(Z).
```

Global Constraints: cumulative - Placement

- The results obtained show well the importance of using the redundant cumulative/4 constraints. Testing the program presented with and without these constraints, the following results are obtained

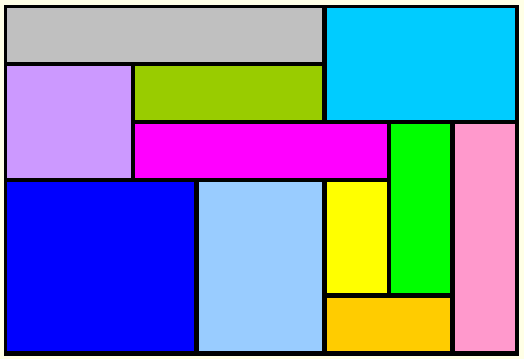
in 16 ms

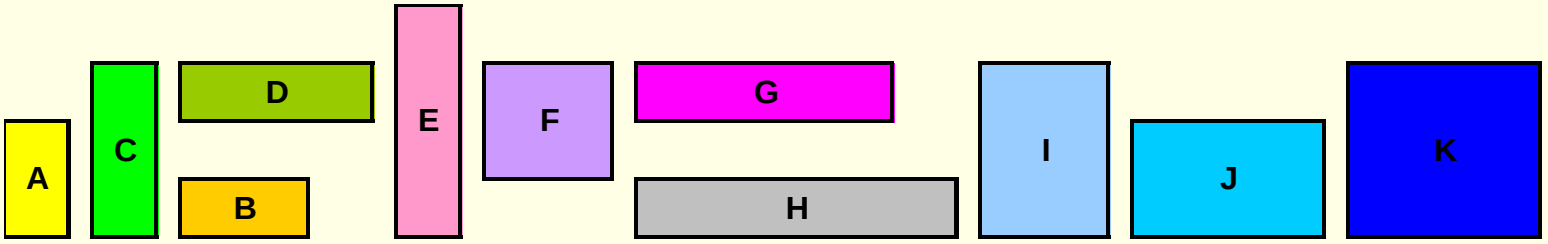
with cumulative



in 5.407 s

without cumulative





Placement w/ Rotation

- An obvious limitation of the program shown is its impossibility to "rotate" the components by 90º for better accommodation in the available space.

- Sometimes, a placement is only possible if these rotations are performed in some of the components.

- The changes to make in the program are small. Given constant dimensions Ac - Bc, it must be guaranteed that they are either interpreted as Width-Height or as its Height-Width .

- Given the fixed dimensions, such flexibility is obtained by simple constructive disjunction, and user controlled by means of parameter Mode (fix/rot) disallowing or allowing rotations, respectively.

Placement w/ Rotation

Example:

```
 ... Wc = [ 6, 6, 5, 5, 4, 4, 3, 3, 2, 2],
     Hc = [ 2, 1, 2, 1, 2, 1, 2, 1, 2, 1],
     domains(Mode,W,H,Wc,Hc),
 ...
=======================

domains(fix, [],[],[],[]).
domains(fix, [W|Wt],[H|Ht],[Wc|Wtc],[Hc|Htc]):-
   W = Wc, H = Hc, domains(fix,Td,Tr,Tdc,Trc).

domains(rot, [],[],[],[]).
domains(rot,[Hd|Td],[Hr|Tr],[Hdc|Tdc],[Hrc|Trc]):-
   [Hd,Hr] :: [Hdc,Hrc],
   Hr * Hd #= Hdc * Hrc,
   domains(rot, Td,Tr,Tdc,Trc).
```

Placement w/ Rotation

- The labeling of these new variables W and H must now be considered in the enumeration. In fact, only one set of these variables requires enumeration, since the other is automatically computed.
- A possible heuristic to be used is to label the most difficult rectangles first. Here we consider that the difficulty of placing the rectangles depends on their largest dimension. Hence we sort them in the merge/3 predicate and label them in this order.
- Only after enumerating the rectangles we enumerate the rotations, for the case they had not been set yet.

```
merge( X, Y, Z),
labeling(Z),
labeling(W),
```

Placement w/ Rotation

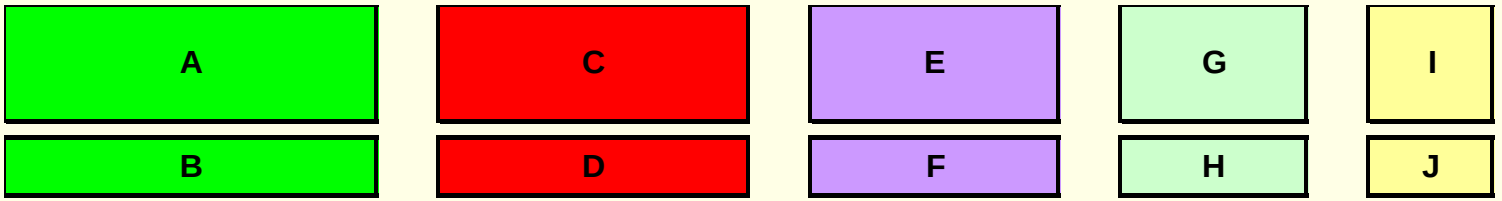
Example:

10 rectangles (A to J) must be cut from a wood board with rectangular shape, with total area of 60, rotations being allowed for better accomodation of the rectangles. The rectangles have the following dimensions (width-W and height-H)

Wc = [6, 6, 5, 5, 4, 4, 3, 3, 2, 2],

Hc = [2, 1, 2, 1, 2, 1, 2, 1, 2, 1],

Graphically,

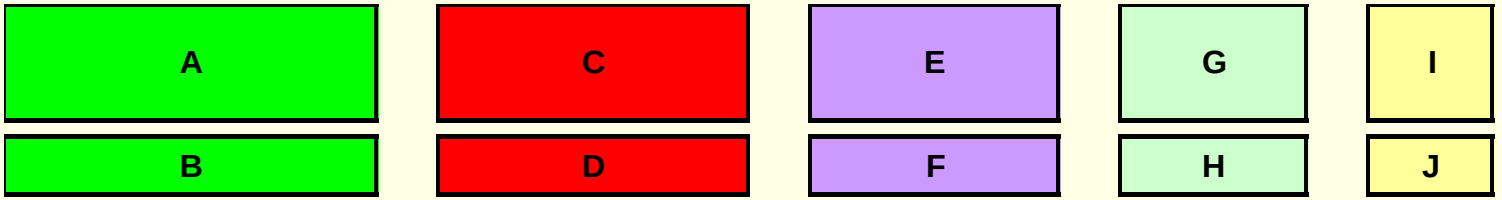


Placement w/ Rotation

Results obtained:

Fix mode :

Width	Height	ms	X-Y
2	30	**532**	no
3	20	542	no
4	15	542	no
5	12	542	no
6	10	552	no
10	6	40	[1-1,1-3,1-4,1-6,7-1,7-3,6-4,6-6,9-4,9-6]
12	5	40	[1-1,1-3,1-4,6-5,7-1,6-4,10-3,7-3,11-1,11-5]
15	4	50	[1-1,1-3,7-1,1-4,12-1,7-3,11-3,6-4,14-3,9-4]
20	3	40	[1-1,1-3,7-1,7-3,12-1,12-3,16-1,16-3,19-1,19-3]
30	2	120	[1-1,7-1,17-1,7-2,22-1,13-1,26-1,12-2,29-1,15-2]

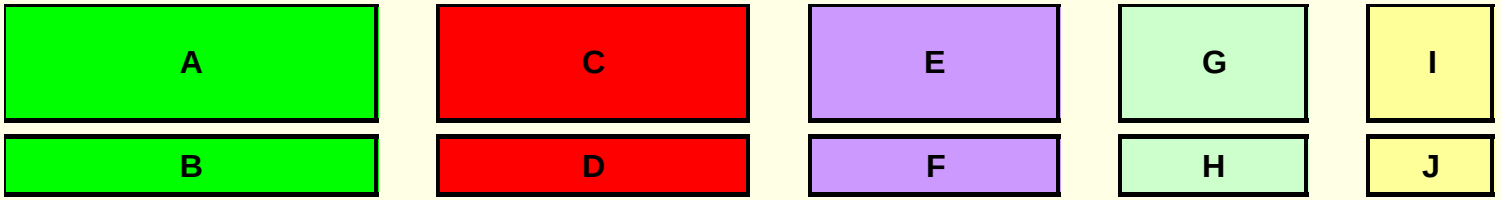


Placement w/ Rotation

Results obtained with rotation (rotated components in red):

Rot mode :

Width	Height	ms	X-Y
2	30	301	[1-1,1-7,1-16,2-7,1-21,2-12,1-25,1-13,1-28,1-30]
3	20	191	[1-1,1-7,1-13,3-13,2-7,3-1,1-18,1-20,2-11,3-5]
4	15	190	[1-1,1-7,2-7,4-5,3-1,4-10,2-14,1-13,2-12,3-5]
5	12	60	[1-1,1-7,2-7,3-1,4-1,2-12,4-7,3-6,4-10,4-5]
6	10	60	[1-1,1-3,1-4,1-6,1-7,3-7,3-8,6-4,5-8,5-10
10	6	60	[1-1,1-3,1-4,1-6,6-4,6-6,7-1,10-4,9-1,9-3]
12	5	70	[1-1,1-3,1-4,6-4,7-1,7-3,11-1,6-5,11-4,9-5]
15	4	250	[1-1,1-3,7-1,1-4,9-3,15-1,12-1,6-4,13-3,7-3]
20	3	50	[1-1,1-3,7-1,7-3,12-1,12-3,16-1,16-3,19-1,19-3]
30	2	261	[1-1,7-1,16-1,7-2,21-1,12-2,25-1,13-1,28-1,30-1]



Placement

Global constraints:

disjoint2(+Rectangles)
disjoint2(+Rectangles, +Options)

are available in SICStus
for non-overlapping rectangles.