Constraint Programming

- An overview

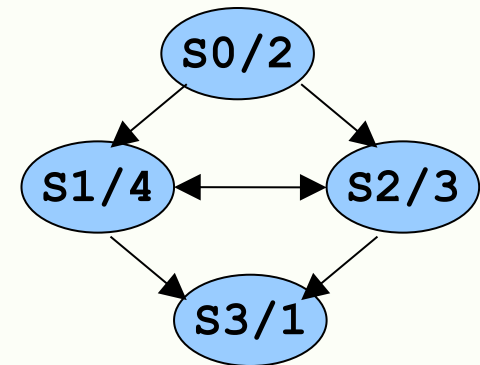
  - Global Constraints in Choco (2)
  - Scheduling Constraints
  - Redundant Constraints
  - Resource Management Constraints

Scheduling Constraints

- We start discussing 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 precedence between them, as well as mutual exclusion (↔). The tasks duration are shown in the nodes.

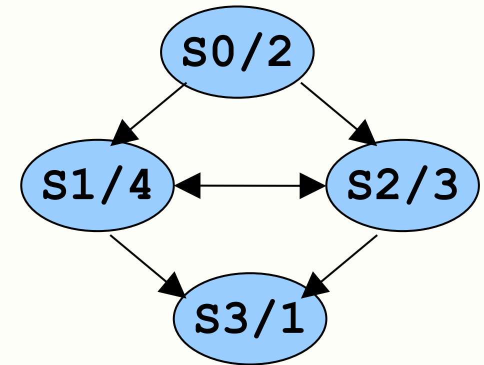


- The problem may be modelled by the following precedence and disjunctive constraints, where
  - s[i] denotes the start time of task i;
  - d[i] denotes the duration of task i;

```
   (s[1] >= s[0]+d[0])
   (s[2] >= s[0]+d[0]);
   (s[3] >= s[1]+d[1]);
   (s[3] >= s[2]+d[2]);
   (s[2] >= s[1]+d[1] || s[1] >= s[2]+d[2]); }
```

Redundant Constraints

```
(s[1] >= s[0]+d[0])
(s[2] >= s[0]+d[0]);
(s[3] >= s[1]+d[1]);
(s[3] >= s[2]+d[2]);
(s[2] >= s[1]+d[1] || s[1] >= s[2]+d[2]);
```



- Because task 0 (that we assume starts at 0) and **both** tasks 1 and 2, must be finished before task 3 starts, and because these tasks have durations 2, 4 and 3 respectively, task 3 may only start at time

  s[3] >= 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

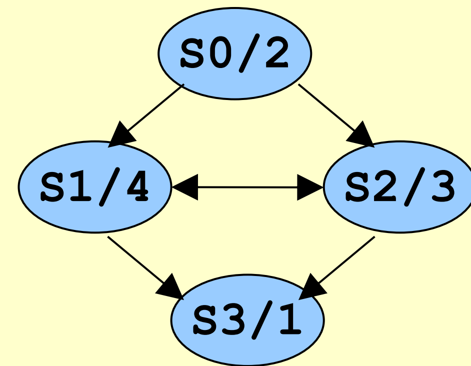
  s[3] >= 0+2+4 = 6 and s[3] >= 0+2+3 = 5

  Assuming all tasks should start until time 9, the domain of s[3] is narrowed to 6..9 rather than fixed to 9.

Cumulative Constraints

- In Choco, the previous example can be encoded as follows:
- First the classes to import, a model is created with its solver, and the instance data is specified.

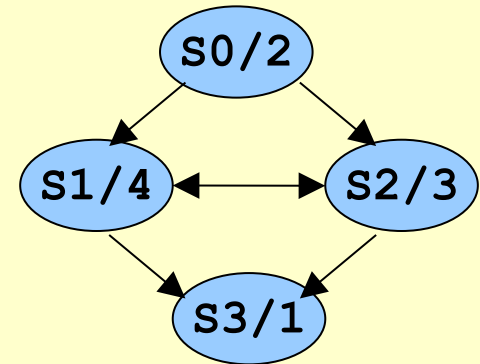
```
package choco;
import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.Solver;

public class tabling {
    public static void main(String[] args) {
        Model md = new Model("table test");
        Solver sv = md.getSolver();
        IntVar [] s = md.intVarArray(”S", 4, 0, 9);
        int [] d = {2,4,3,1};
        int [] h = {1,1,1,1}
        int cap  = 1;
        ....
    }
}
```



Cumulative Constraints

- Then the precedence and non-overlapping constraint are specified.

```
....
public class tabling {
    public static void main(String[] args) {
        ....
        // precedence constraints
        md.arithm(s[1], “>=”, s[0], “+”, d[0]).post();
        md.arithm(s[2], “>=”, s[0], “+”, d[0]).post();
        md.arithm(s[3], “>=”, s[1], “+”, d[1]).post();
        md.arithm(s[3], “>=”, s[2], “+”, d[2]).post();

        // no overlap constraints
        md.or (
            md.arithm(s[2], “>=”, s[1], “+”, d[1]),
            md.arithm(s[1], “>=”, s[2], “+”, d[2])
        ).post();
        ....
    }
}
```



Redundant Constraints

- In general, the interaction of many constraints may not be adequately processed by the corresponding propagators separately.
- 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-overlapping 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-overlapping tasks $\mathbf{X_i}$ succeed some task Z the following 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 $\mathbf{s_i}$, **D** be the set of their durations $\mathbf{d_i}$ and **R** the set of the number of resources of a given type required by the tasks, $\mathbf{r_i}$. Denoting by

  $\mathbf{a = min_i(s_i)}$; $\mathbf{b = max_i(s_i+d_i)}$;

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

  then

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

Cumulative Constraints

- In Choco there are many options to specify a cumulative constraint. Possibly, the simplest fom of specifying a cumulative constraint on n tasks, is

```
cumulative(IntVar[] starts, int[] durations, int[] heights, int capacity)
```

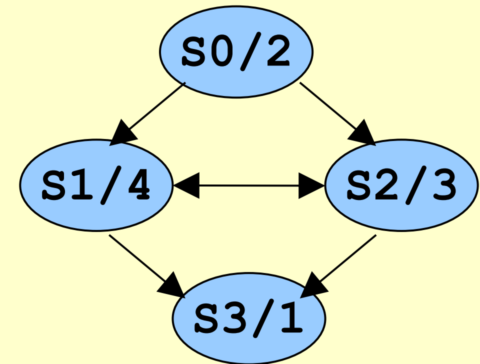
- where
  - **IntVar[] starts:** denote the start of the tasks (a decision array);
  - **int [] duration:** denote the duration of each task (an integer array);
  - **int [] heights:** denotes the resource consumption of each task (an integer array);
  - **int capacity:** total amount of resources available at each time point (an integer).

- This specification creates a cumulative constraint, that enforces that at each point in time, the cumulated height of the set of tasks that overlap that point does not exceed a given limit.

Cumulative Constraints

- In this case, as there are only two non-overlapping tasks (1 and 2, the others do not overlap due to precedence constraints), the non overlapping constraint can be imposed with a cumulative constraint on these variables, as follows:

```
....
public class tabling {
    public static void main(String[] args) {
        ....
        // precedence constraints
        ....
        // no overlap constraints
        //md.or (....).post();
        IntVar [] nover = md.intVarArray(2,1,9);
        nover[0] = s[1];
        nover[1] = s[2];
        md.cumulative(nover, new int [] {d[1],d[2]}, new int [] {h[1],h[2]}, cap);
        ....
    }
}
```

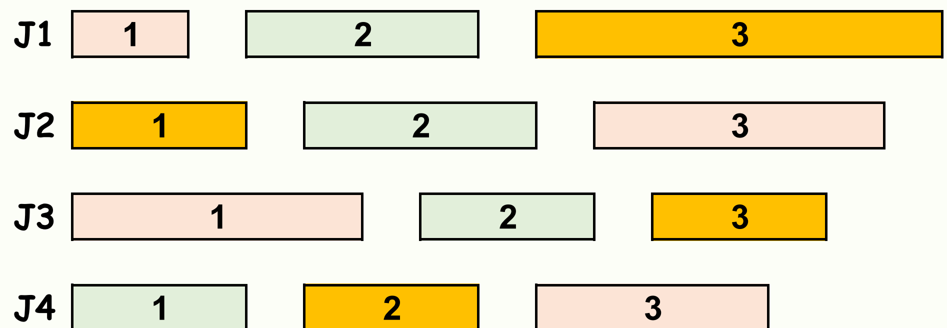


- Of course, in this toy example the cumulative constraint is not worth using. But this is not the case for hard job-shop problems.

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 is the same as the number of tasks in each job.
- A simple instance of the problem (with 3 machines) is given in the table below (with the corresponding graphic representation).

		Tasks Y		
	Z, D	0	1	2
Jobs X	0	0 , 2	1 , 4	2 , 7
	1	2 , 3	1 , 4	0 , 5
	2	0 , 5	1 , 3	2 , 3
	3	1 , 3	2 , 4	0 , 4



Cumulative Constraints: Job Shop

- This instance below (tasks and jobs start at 1, not 0) 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

Cumulative Constraints: Job Shop

- Because the job shop problem is so relevant, Choco provides a class, Tasks, with 3 components, start, duration and end, maintaining the implicit constraint

  start + duration = end

- For this class, a more sophisticated cumulative constraint can be used

  cumulative(tasks, heights, capacity, graphBased, cum)

- where
  - Tasks [] **tasks:** denote the set of non-overlapping tasks;
  - int [] **heights:** resources consumed by each task (an integer array);
  - int **capacity:** total amount of resources available at each time point (an integer).
  - boolean **graphBased**: a parameter indicating how to filter
    - true: applies on subset of overlapping tasks
    - false: applies on all tasks
  - Cumulative.Filter **cum**: a cumulative filter specifying the algorithm to filter
    - possible values are TIME / SWEEP / NRJ

Cumulative Constraints: Job Shop

- The proposed program to solve the job-shop problem imports two classes in this case, for the tasks and fo the cumulative constraint to be used.

```
package choco;

import org.chocosolver.solver.Model;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.variables.Task;
import org.chocosolver.solver.constraints.nary.cumulative.Cumulative;

public class job_shop {

    public static void main(String[] args) {
        Model md = new Model("(" + 4 +"," + 3 + ") - Job Shop");
        Solver sv = md.getSolver();
        ....
    }
}
```

Cumulative Constraints: Job Shop

- Now the instance is defined (number of jobs, tasks and machines)
- An upper bound for the start times and the make span is defined.
- The instances of the tasks are created with the bounds for the starting times and durations and machines of each task.

	Z, D	Tasks Y 0	1	2
Jobs X	0	0 , 2	1 , 4	2 , 7
	1	2 , 3	1 , 4	0 , 5
	2	0 , 5	1 , 3	2 , 3
	3	1 , 3	2 , 4	0 , 4

```
....
int njobs = 4;
int ntsks = 3;
int nmchs = ntsks;
int maxStrt = 20;
int maxSpan = 30;
int [][] durs = {{2,4,7},{3,4,5},{5,3,3},{3,4,4}};
int [][] mchs = {{0,1,2},{2,1,0},{0,1,2},{1,2,0}};

Task [][] tx = new Task[njobs][ntsks];
for(int i = 0; i < njobs; i++){
    for (int j = 0; j < ntsks; j++){
        tx[i][j] = new Task( md.intVar(1,maxStrt),
                             md.intVar(durs[i][j],durs[i][j]),
                             md.intVar(1, maxSpan));
    }
}
....
```

Cumulative Constraints: Job Shop

- Now the precedence constraints are posted.
- Moreover, the overall makespan of the instance is the maximum of the end times of all the last tasks of each job.
- This makespan is set as the objective to minimise.

```
....
// precedence constraints
for (int i = 0; i < njobs; i++) {
    for (int j = 0; j < ntsks-1; j++) {
        md.arithm(tx[i][j].getEnd(), "<=", tx[i][j+1].getStart()).post();
    }
}

// set objective
IntVar [] lx = md.intVarArray(njobs,1,maxSpan);
IntVar last = md.intVar(1,maxSpan);
for (int i = 0; i < njobs; i++)
    lx[i] = tx[i][ntsks-1].getEnd();
md.max(last, lx).post();
md.setObjective(Model.MINIMIZE, last);
....
```

		Tasks Y		
	Z, D	0	1	2
Jobs X	0	0 , 2	1 , 4	2 , 7
	1	2 , 3	1 , 4	0 , 5
	2	0 , 5	1 , 3	2 , 3
	3	1 , 3	2 , 4	0 , 4

Cumulative Constraints: Job Shop

- A cumulative constraint should be posted for each machine.
- Hence, the tasks executed in each machine are obtained from the specification, and made correspond to the previously defined tasks.

```
....
Task [][] mx = new Task[nmchs][njobs];
for (int k = 0; k < nmchs; k++){
    int p = 0;
    for(int i = 0; i < njobs; i++){
        for (int j = 0; j < ntsks; j++){
            if(mchs[i][j] == k){
                mx[k][p] = tx[i][j] ;
                p = p + 1;
            }
        }
    }
}
....
```

		Tasks Y		
	Z, D	0	1	2
Jobs X	0	0 , 2	1 , 4	2 , 7
	1	2 , 3	1 , 4	0 , 5
	2	0 , 5	1 , 3	2 , 3
	3	1 , 3	2 , 4	0 , 4

Cumulative Constraints: Job Shop

- Finally the cumulative constraint is posted.
- Notice that an instance of the Cumulative.Filter must be created to be used as the last paramenter of the cumulative constraints.
- The solutions are now obtained with decreasing value of the objective value.

```
        ....
        // precedence constraints
        Cumulative.Filter cum = Cumulative.Filter.SWEEP; //TIME, SWEEP, ...
        boolean graphBased = true;
        IntVar one = md.intVar(1,1);
        IntVar [] ones = md.intVarArray(njobs,1,1);
        for (int k = 0; k < nmchs; k++)
            md.cumulative(mx[k], ones, one, graphBased, cum).post();

        while (sv.solve()){
          // show solutions
        }
    }
}
```

		Tasks Y		
	Z, D	0	1	2
Jobs X	0	0 , 2	1 , 4	2 , 7
	1	2 , 3	1 , 4	0 , 5
	2	0 , 5	1 , 3	2 , 3
	3	1 , 3	2 , 4	0 , 4

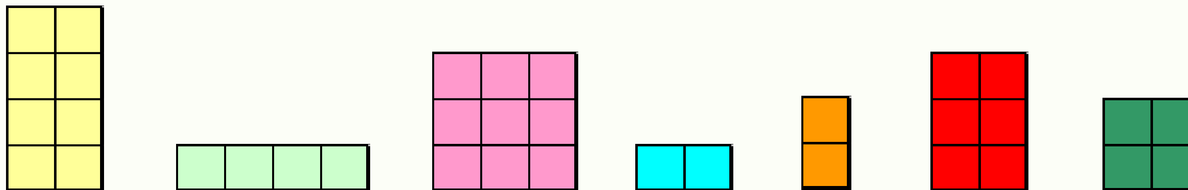
Cumulative Constraints: Job Shop

```
Solving (4,3) - Job Shop (solution 1); with Makespan = 24
--- Jobs ---
 1 to 3;  4 to 8;  17 to 24;
 1 to 4;  11 to 15;  15 to 20;
 3 to 8;  8 to 11;  14 to 17;
 1 to 4;  4 to 8;  8 to 12;
--- Machines ---
 1 to 3;  15 to 20;  3 to 8;
 4 to 8;  11 to 15;  8 to 11;
 17 to 24;  1 to 4;  14 to 17;
Solving (4,3) - Job Shop (solution 2); with Makespan = 20
--- Jobs ---
 1 to 3;  4 to 8;  8 to 15;
 1 to 4;  11 to 15;  15 to 20;
 3 to 8;  8 to 11;  15 to 18;
 1 to 4;  4 to 8;  8 to 12;
--- Machines ---
 1 to 3;  15 to 20;  3 to 8;
 4 to 8;  11 to 15;  8 to 11;
 8 to 15;  1 to 4;  15 to 18;
Solving (4,3) - Job Shop (solution 3); with Makespan = 18
--- Jobs ---
 1 to 3;  4 to 8;  8 to 15;
 1 to 4;  8 to 12;  12 to 17;
 3 to 8;  12 to 15;  15 to 18;
 1 to 4;  4 to 8;  8 to 12;
--- Machines ---
 1 to 3;  12 to 17;  3 to 8;
 4 to 8;  8 to 12;  12 to 15;
 8 to 15;  1 to 4;  15 to 18;

 !!! No (more) solutions after 0.03997849 secs
```

		Tasks Y		
	Z, D	0	1	2
Jobs X	0	0 , 2	1 , 4	2 , 7
	1	2 , 3	1 , 4	0 , 5
	2	0 , 5	1 , 3	2 , 3
	3	1 , 3	2 , 4	0 , 4

Job Scheduling

- In the job shop problem, every task requires the use of a single machine. Because several tasks use the same machine the cumulative constraint effectively enforces a schedule with non-overlapping of the tasks using the same machine.
- However, there are situations where a task does not use all the resources available.
- For example, a company may have several workers that should perform several tasks, that do not require all the works at the same time, as shown in the following example.



- In this case, there are 7 tasks, each with a duration **`d[i]`** (let us assume the time unit to be days), when performed by a number **`r[i]`** of workers.

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

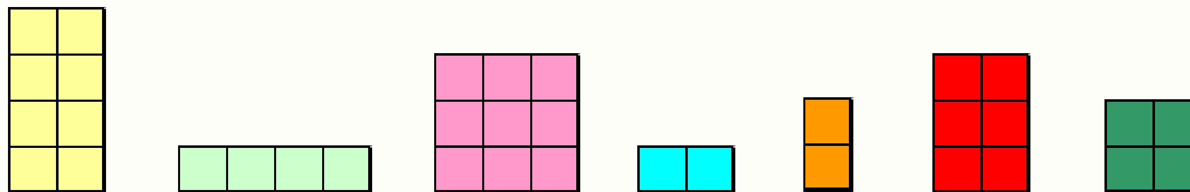
Job Scheduling

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

Graphically, the tasks can be viewed as



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

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

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

- This problem may still be modelled by the cumulative constraint, but now the parameter heights should reflect the different requirement of resources (workers) that each task require.

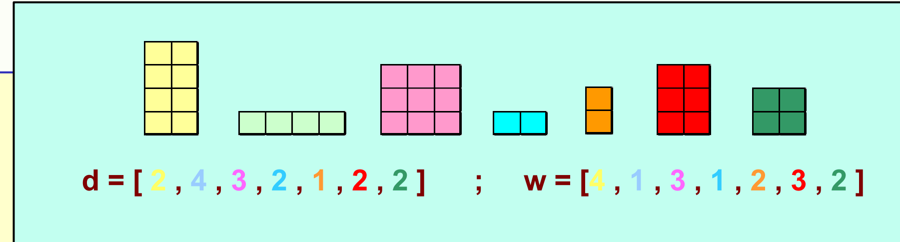
Cumulative Constraints: Job Scheduling

- The code that can be used to solve the problem is shown next.
- First, the import statements and class signatures are defined,
- as well as the model and solver classes
- and problem specific parameters.

```
package choco;

import org.chocosolver.solver.Model;
import org.chocosolver.solver.Solver;
import org.chocosolver.solver.variables.IntVar;
import org.chocosolver.solver.variables.Task;
import org.chocosolver.solver.constraints.nary.cumulative.Cumulative;

public class job_schedule {
    public static void main(String[] args) {
        Model md = new Model("Job Schedule");
        Solver sv = md.getSolver();
        int n = 7;
        boolean flexible = true;  // allow flexible tasks
        .....
    }
}
```

Cumulative Constraints: Job Scheduling

- Now, the dimensions of the tasks are defined,
- and converted into decision variables.




```
.....
int maxT = 5; // 9, 7, 5, 4
int maxR = 7; // 4, 5, 7, 9

int [] days = {2 , 4 , 3 , 2 , 1 , 2 , 2};
int [] wrks = {4 , 1 , 3 , 1 , 2 , 3 , 2};

IntVar [] d = md.intVarArray(n, 1, 100);
IntVar [] w = md.intVarArray(n, 1, 100);
if (flexible){
    for(int i = 0; i < n; i++)
        md.arithm(d[i], "*", w[i], "=", days[i]*wrks[i]).post();
} else {
    for(int i = 0; i < n; i++){
        md.arithm(d[i], "=", days[i]).post();
        md.arithm(w[i], "=", wrks[i]).post();
    }
}
.....
```

Cumulative Constraints: Job Scheduling

- Finally the tasks are defined,
- the makeSpan is also set to be the maximum end of all the tasks, and
- the cumulative constraint is posted, with the parameters seen before.

```
.....
Task [] tx = new Task[n];
for(int i = 0; i < n; i++){
    tx[i] = new Task(md.intVar(1,maxT+1),d[i], md.intVar(1,maxT+1));
}
// set makeSpan
IntVar [] lx = md.intVarArray(n,1,maxT+1);
IntVar makeSpan = md.intVar(1,maxT);
for (int i = 0; i < n; i++)
    md.arithm(lx[i], "=", tx[i].getEnd(), "-", 1).post();
    md.max(makeSpan, lx).post();
}
IntVar maxH = md.intVar(maxR,maxR);
Cumulative.Filter cum = Cumulative.Filter.SWEEP; //TIME, SWEEP, ...
boolean graphBased = true;
md.cumulative(tx, w, maxH, graphBased, cum).post();
.....
```

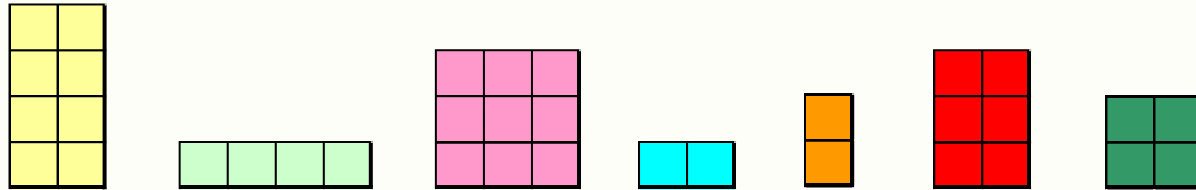
Cumulative Constraints: Job Scheduling

- Finally the objective is set (i.e. minimise the make span – not used here)
- And the solutions (if any) are displayed).

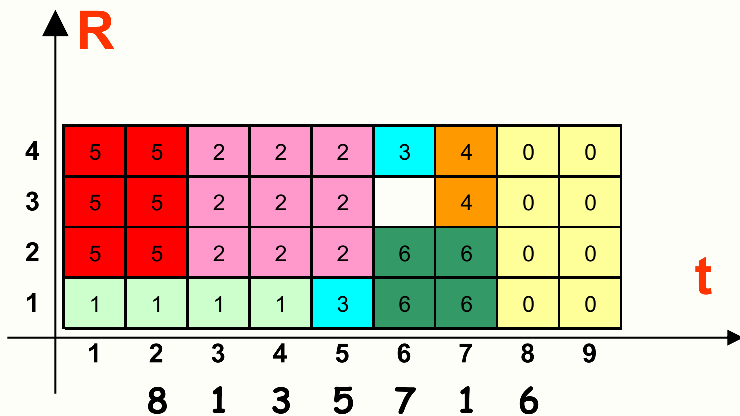
```
    .....

    // set objective
    // md.setObjective(Model.MINIMIZE, makeSpan);

    while (sv.solve()){
        System.out.print("Solving " + sv.getModelName() + " (solution " +
                                  sv.getSolutionCount() + ");");
        System.out.println(" with Makespan = " + makeSpan.getValue());
        System.out.println("--- Start and End Times ---");
        for(int i = 0; i< n; i++){
            System.out.print(" " + tx[i].getStart().getValue() + " to " +
                          (tx[i].getEnd().getValue()-1)+"; ");
        }
        System.out.println();
    }
    System.out.println("\n !!! No (more) solutions after " + sv.getTimeCount() +
                                                         " secs");
    }
}
```

Cumulative Constraints

Some Results:



- With maxR = 4 (4 resource units available) and imposing that all tasks finish no later than time 9 (**note**: task.getEnd = 9+1) a number of answers are obtained, (allowing one of the 6 workers to rest for one hour) namely

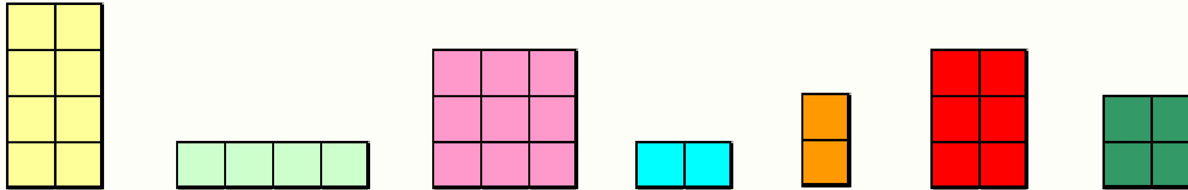


2	2	2	5	5		4	0	0
2	2	2	5	5	6	4	0	0
2	2	2	5	5	6	6	0	0
1	1	1	1	3	3	6	0	0
	8	1	1	5	7	4	6	

2	2	2	5	5		4	0	0
2	2	2	5	5	6	4	0	0
2	2	2	5	5	6	6	0	0
1	1	1	1	3	3	6	0	0
	8	1	1	5	7	4	6	

Cumulative Constraints

Some Results:



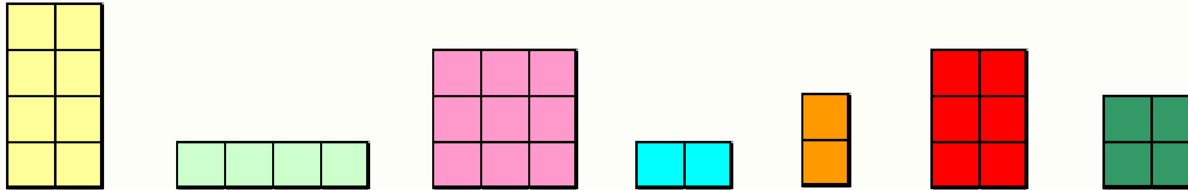
- With maxR = 5 (5 resource units available) and imposing that all tasks finish no later than time 7 (**note**: task.getEnd = 7+1) a number of answers are also obtained, (this time not allowing any workers to rest) namely

1	1	1	1	6	6	4
0	0	3	3	6	6	4
0	0	2	2	2	5	5
0	0	2	2	2	5	5
0	0	2	2	2	5	5
1	**1**	**3**	**3**	**7**	**6**	**5**

1	1	1	1	6	6	4
5	5	0	0	6	6	4
5	5	0	0	2	2	2
5	5	0	0	2	2	2
3	3	0	0	2	2	2
3	**1**	**5**	**1**	**7**	**1**	**5**

Cumulative Constraints

Some Results:



- In general, we may want to trade time for resources, e.g. instead of fitting the whole tasks is a rectangle of MaxT x MaxR , we may want to fit the tasks in a rectangle of MaxT x MaxR.

- We found solutions for maxT = 7 and maxR = 5, namely

1	1	1	1	6	6	4
0	0	3	3	6	6	4
0	0	2	2	2	5	5
0	0	2	2	2	5	5
0	0	2	2	2	5	5
1	**1**	**3**	**3**	**7**	**6**	**5**

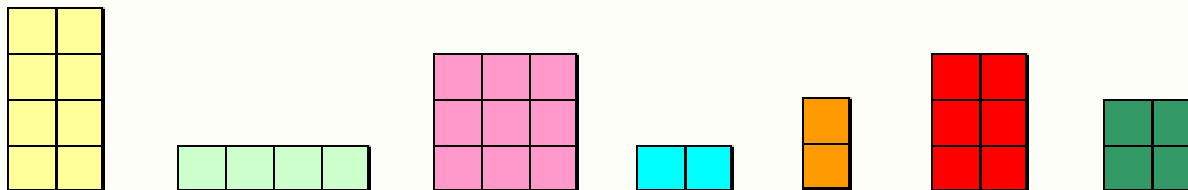
1	1	1	1	6	6	4
5	5	0	0	6	6	4
5	5	0	0	2	2	2
5	5	0	0	2	2	2
3	3	0	0	2	2	2
3	**1**	**5**	**1**	**7**	**1**	**5**

- However, **there are no solutions** with maxT = 7 and maxR = 5 !

Job Scheduling

- Sometimes, the tasks are flexible, in that the duration of the task depends on the number of workers used (for example a task requiring 4 workers-days might be performed by
  - 1 worker in 4 days;
  - 2 workers in 2 days; or
  - 4 workers in 1 day; or even
  - 3 workers in 1 day and 1 worker in another day.

- All these situations can be modelled in Choco.



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

Cumulative Constraints

- In some applications, tasks are flexible, in the sense that time may be traded for resources.
- Flexible tasks may be more easily accommodated within the resources (and time) available, namely to solve instances of the problem that cannot be solved with fixed tasks.
- Scheduling of this type of tasks may be specified very similarly to what was done before.
- However, if before we could make the durations **d** and workers **w** used by each task i to be constants

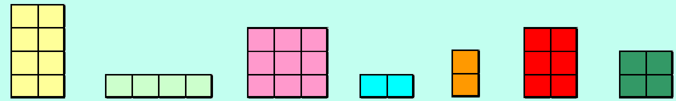
```
d[i] * w[i] = days[i] * wrks[i]
d[i] * w[i] = days[i] * wrks[i]
```

now they should simply be constrained to

```
d[i] * w[i] = days[i] * wrks[i]
```

- Cf. code provided, that controls this flexibility with a Boolean variable flexible.

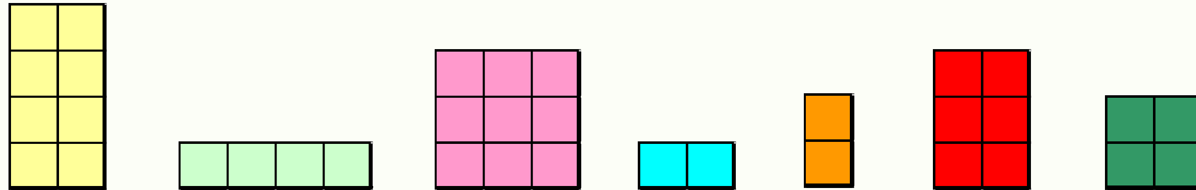
Cumulative Constraints: Job Scheduling




```
.....
int maxT = 5; // 9, 7, 5, 4
int maxR = 7; // 4, 5, 7, 9

int [] days = {2 , 4 , 3 , 2 , 1 , 2 , 2};
int [] wrks = {4 , 1 , 3 , 1 , 2 , 3 , 2};

IntVar [] d = md.intVarArray(n, 1, 100);
IntVar [] w = md.intVarArray(n, 1, 100);
if (flexible){
    for(int i = 0; i < n; i++)
        md.arithm(d[i], "*", w[i], "=", days[i]*wrks[i]).post();
} else {
    for(int i = 0; i < n; i++){
        md.arithm(d[i], "=", days[i]).post();
        md.arithm(w[i], "=", wrks[i]).post();
    }
}
.....
```

Cumulative Constraints

Some Results:



- With maxT = 7 and maxR = 5 (previously impossible) there are now several solutions. But notice, in the solution below, the “deeper” transformation in several tasks.

- Some tasks, simply change the number of workers for the number of days.
  - This is the case of tasks 4 and 6
- Other tasks may be more deeply changed, in that the very “shape” of the task may be changed.
  - This is the case of task 1, that changes its shape from 1x4 to 2x2.

2	2	2	5	5
2	2	2	5	5
2	2	2	5	5
3	3	0	0	6
4	4	0	0	6
1	1	0	0	6
1	1	0	0	6

s	3	1	1	1	1	4	5
d	2	2	3	2	2	2	1
w	4	2	3	1	1	3	4

Tiling Problems

- Several applications of great (economic) importance require the satisfaction of **tiling** constraints, i.e. place a number of components in a given space, without overlaps.
- Some of these applications include:
  - Wood boards: a number of smaller pieces should be cut from large boards:
  - Containers: Placement of boxes 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).

Tiling Problems

Example:

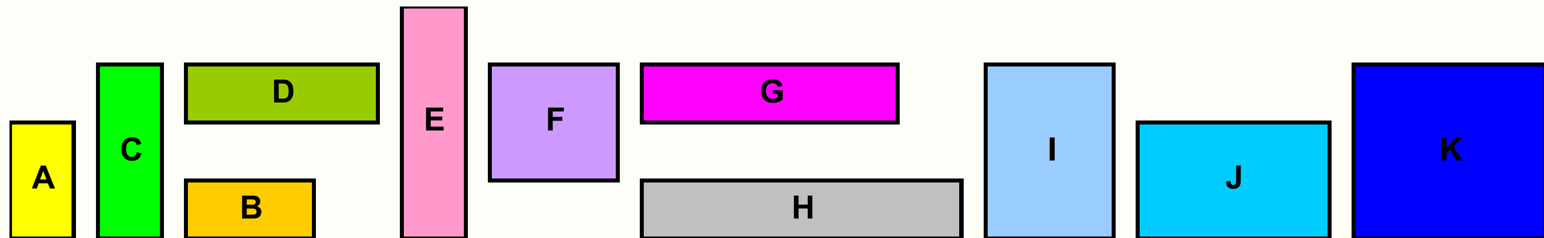
Find the appropriate cuts to be made on a wood board of dimensions W * H so as to obtain 11 rectangular pieces (A a K).

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]

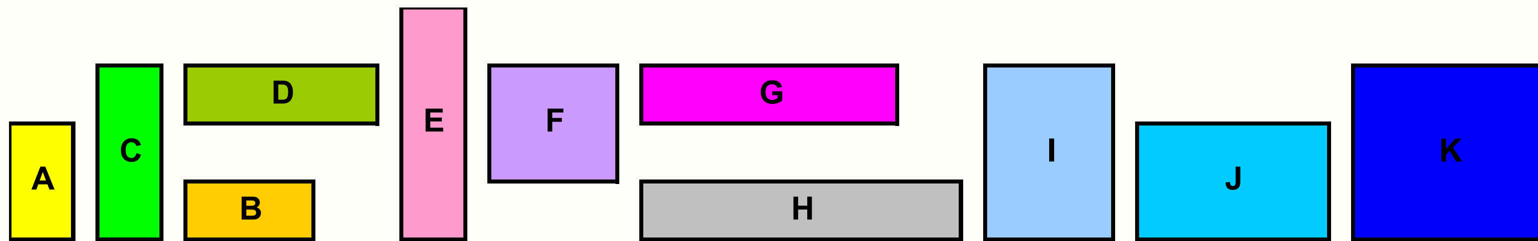
h = [2, 1, 3, 1, 4, 2, 1, 1, 3, 2, 3]

Graphically



and a cumulative constraint can be used as before, adapting durations to widths and resources to heights.

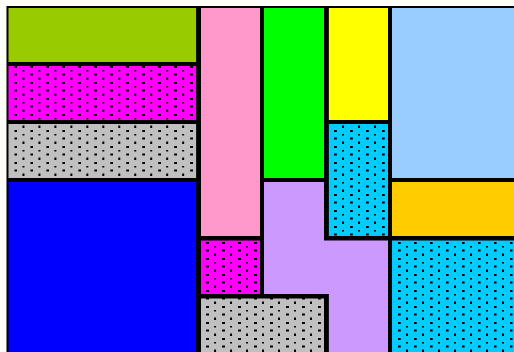
Tiling Problems

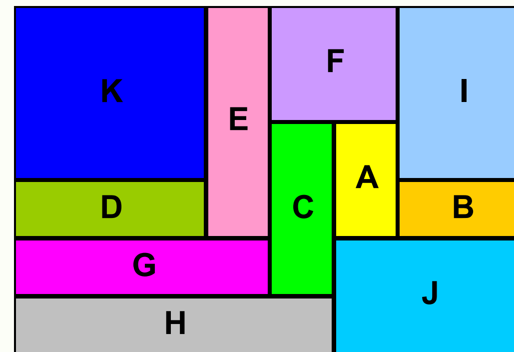


- Unfortunately, the results obtained might not have a direct reading. For example, one of the solutions obtained with an 8*6 rectangle is

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

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





Tiling Problems

- In fact, what we want here is that the rectangles do not overlap!
- 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 committing 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, as in

```
md.or(
   md.arithm(x[i],"+", w[i], "<=", x[j]),
   md.arithm(x[j],"+", w[j], "<=", x[i]),
   md.arithm(y[i],"+", h[i], "<=", h[j]),
   md.arithm(y[j],"+", h[j], "<=", h[i]),
.post();
```

Tiling Problems

- Rather than using the disjunction constraint, a specialised global constraint **diffN** achieves exactly the same purpose (i.e. the semantics is the same). Thus, instead of using the disjunctive constraint over all rectangles

```
for (int i = 0; i < n; i++){
   md.or(
      md.arithm(x[i],”+", w[i], ”<=", x[j]),
      md.arithm(x[j],”+", w[j], ”<=", x[i]),
      md.arithm(y[i],”+", h[i], ”<=", h[j]),
      md.arithm(y[j],”+", h[j], ”<=", h[i]),
   .post();
```

the global constraint diffN can be used as

```
md.diffN(x,y,w,h,true)
```

to achieve the same goal

Rediundant Constraints

- In fact, the global constraint used the disjunction constructively, and so may be (slightly) more efficient in this case.
- For example if a situation is detected where

  $$x < 5 \text{ or } x > 10$$

  then the global constraint infers that the values in the range 5..10 can be safely removed from the domain of x.
- More importantly, although not strictly necessary, the cumulative constraints can still be used, not only in the X dimension as before, but also in the Y dimension
  - `md.cumulative(x, w, h, maxW) and`
  - `md.cumulative(y, h, w, maxY) and`
- In fact, we do not need to impose these constraints, since the last parameter of the diffN constraint, has exactly this purpose if set to true.

```
md.diffN(x,y,w,h,true)
```

- In “hard” tiling problems, the efficiency is quite significant (solutions are found with speed ups of several orders of magnitude).

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, ...**) 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

- Choco also provides a number of these 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

as well as many others that can be checked in the IIntConstraintFactory.pdf.