Modelling V: scheduling

Scheduling

We have:

- a set of *resources* we can use
- a set of tasks to be completed

Each task requires some use of the resources.

Goal:

- find a time for each task, and a resource

Example scheduling problems

Manufacturing scheduling – resources are machines

Building construction – resources are peoples and material

Computational workflow – resources are CPUs and memory

Military operations – resources are e.g. planes or people

Maintenance – resources are people

...

Different variants of the scheduling problem

can tasks be interrupted?
are tasks restricted to specific resources?
do tasks have deadlines?
do tasks have release dates?
are tasks grouped into jobs?
do tasks have sequencing constraints?
can tasks operate at the same time?
do tasks take different times on different machines?
do tasks cost more on different machines?
do tasks have sequence dependent set up times?
do tasks consume resources which must be renewed?
do tasks occupy part but not all of a resource?
do tasks occupy multiple resources at a time?
do tasks incur penalties if later than a deadline?
do tasks give value when completed?

How is it done?

There is a large industry out there providing scheduling solutions.

The problem is tackled by

- operations research and management science
- complex algorithms
- AI search
- local search (hill climbers, GAs, ...)
- constraint programming

Complexity?

Most scheduling problem types are NP-hard.

Different variants of the problem have different complexities, and require different solutions.

Many instances are too hard to solve to optimality, and so heuristic and approximate solutions are needed.

Standard practice is to classify the variations, and focus on techniques specific to those variations.

We will start by looking at the *Job-Shop Scheduling* problem.

Job Shop Scheduling

A job is a set of tasks, and each job has one task per resource.

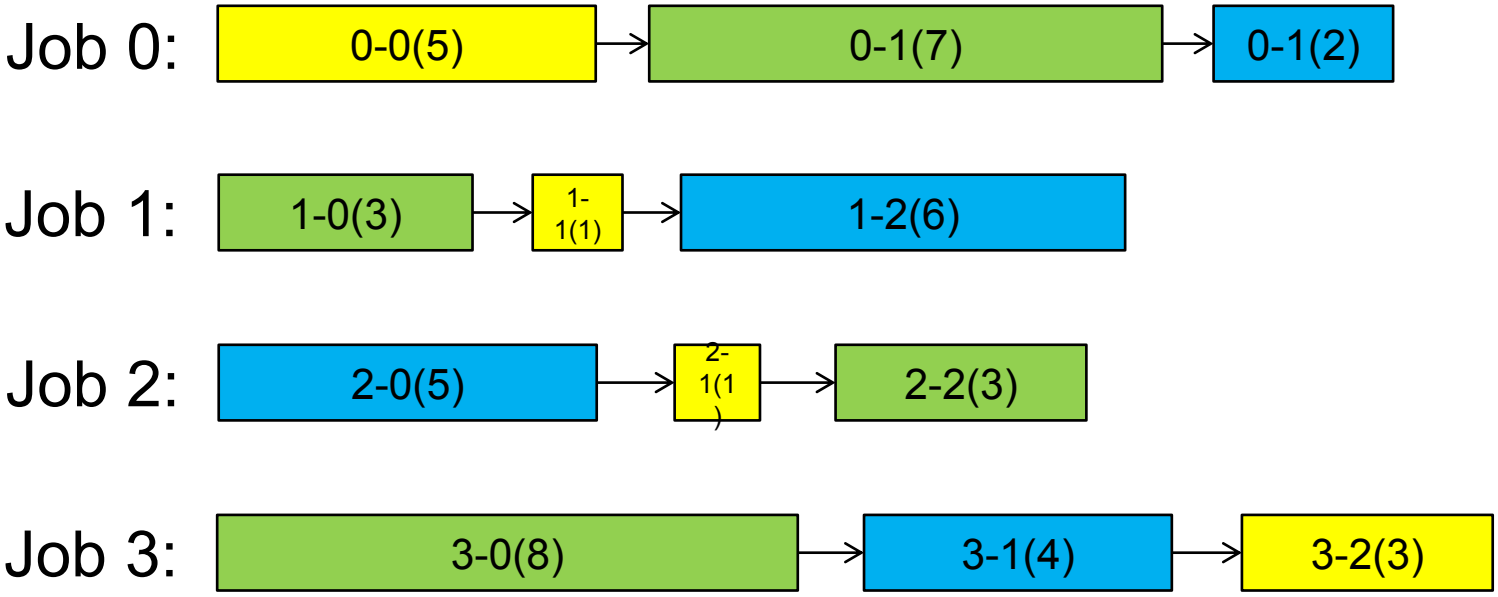
The tasks in a job are totally ordered, and one must finish before the next one starts.

Only one task can be active on a machine at any one time.

Given a solution, the finish time of the last task to finish is the *makespan*.

The aim is to minimise the makespan.

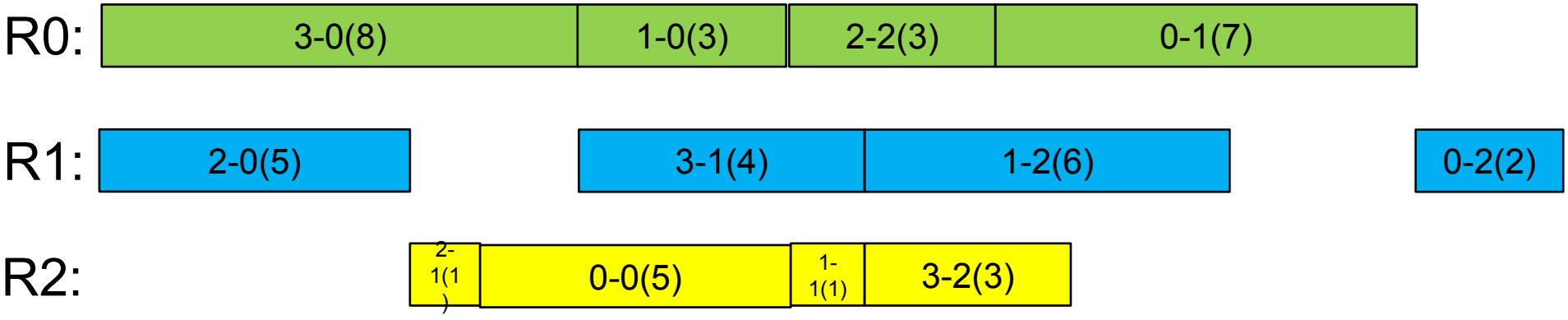
Example



Three resources: Green (0), Blue (1), Yellow (2)
Only one task on a resource at a time.
Each job specifies the order for its tasks.

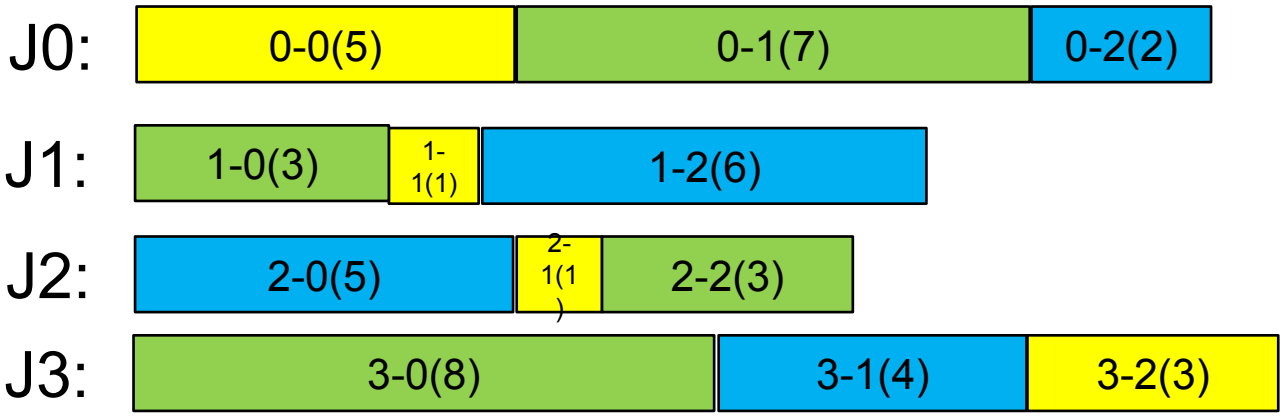
Can we finish all jobs in time ≤ 25?

Example

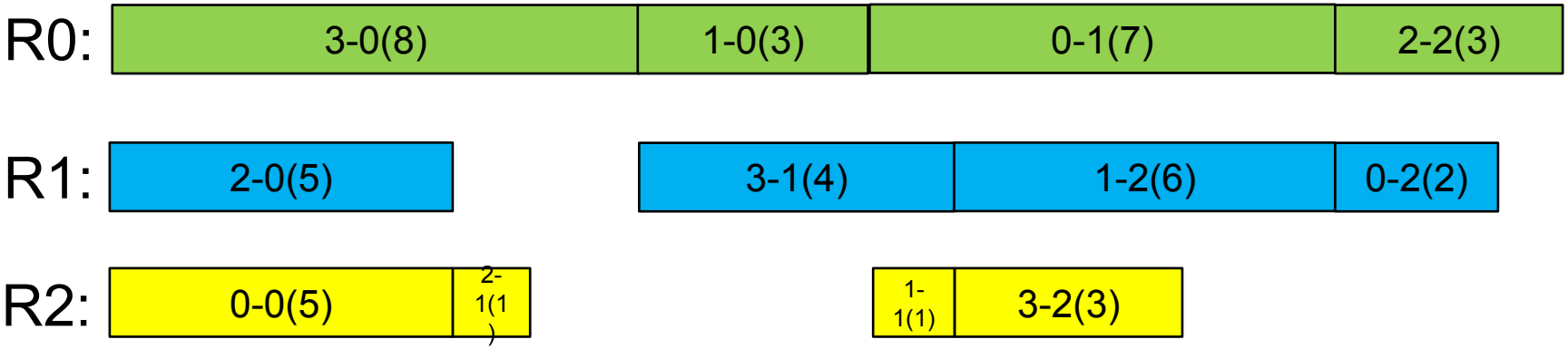


makespan = 23

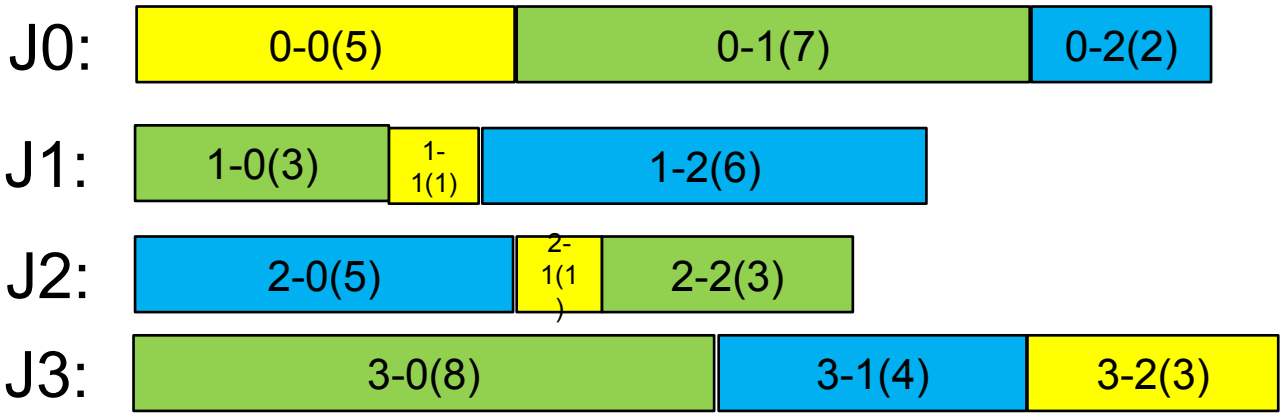
What is the best we could do?



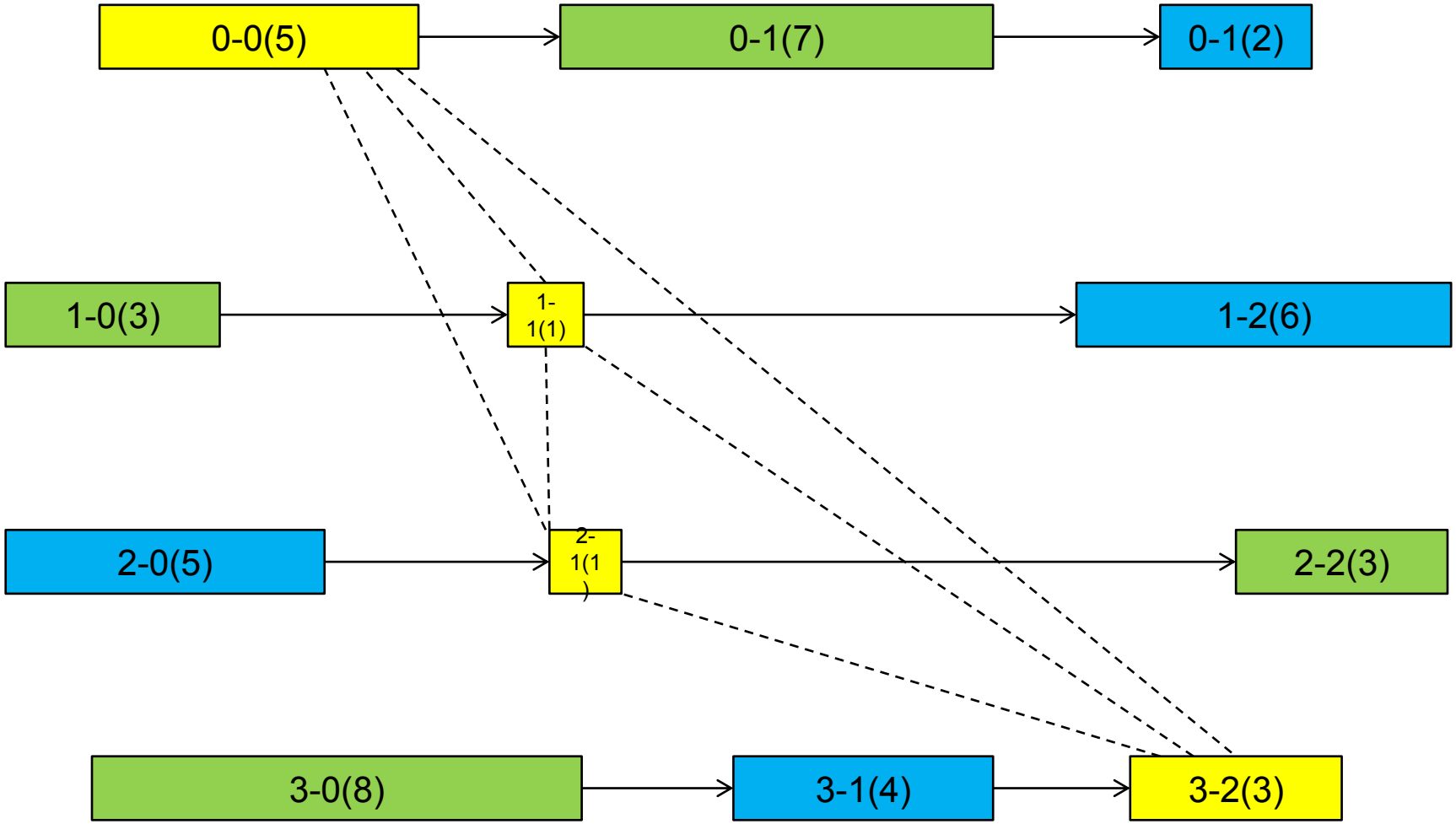
Example



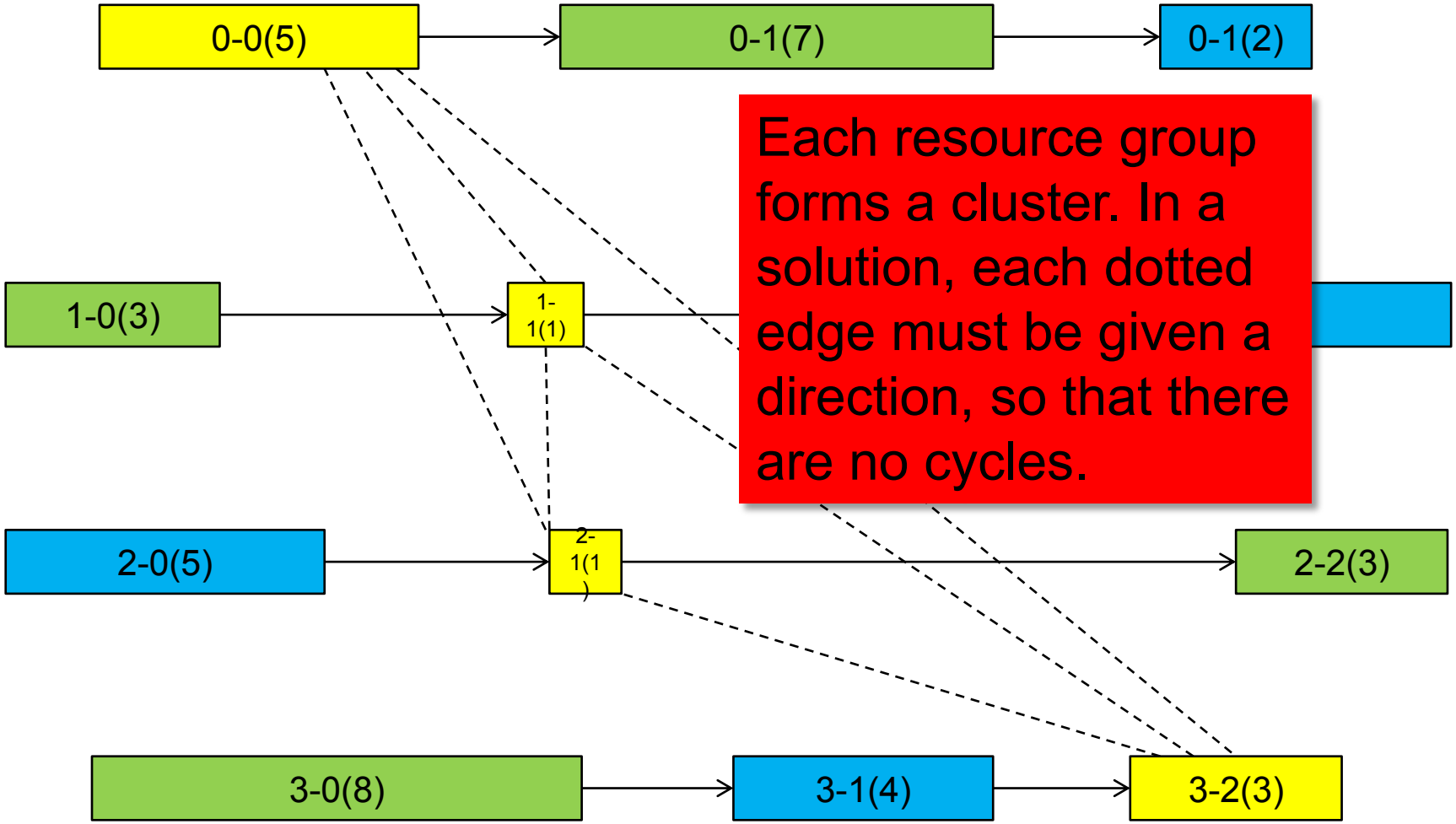
makespan = 21



'Disjunctive' Graph



'Disjunctive' Graph



Simple Constraint Model

Each task is a variable, with domain the set of possible start times. Each task has a known fixed duration.

$$t.end = t.start + t.duration$$

Constraints:

- any two tasks requiring the same resource must not overlap

$$t.end \leq s.start \text{ OR } s.end \leq t.start$$

- any task in a job must finish before the next one starts

$$t1.end \leq t2.start$$

- for all tasks z which are the last ones in their job

$$z.end \leq makespan$$

Elements of the Choco model

Choco provides a Task class, whose objects maintain a start time variable, a duration variable, and an end time variable.

For us, in the JSSP, the duration will be fixed.

Task(IntVar s, IntVar d, IntVar e)
Container representing a task:
It ensures that: start + duration = end

Maintain different representations of the tasks for ease of use:

- all tasks on a job
- all tasks using a resource
- all tasks in the problem

```
//a 2d array for the Task variables [job][task]
Task[][] jobtask = new Task[numJobs][numRes];

//a 2d array for the tasks assigned to each machine [res][job]
Task[][] restask = new Task[numRes][numJobs];

//an array of all the Task variable start times
IntVar[] allTasks = new IntVar[numJobs*numRes];
```

```
//for each resource, create "no overlap" constraints
//between the tasks
//these constraints are not needed if we are using
//cumulative
for (int res = 0; res < numRes; res++) {
  for (int task1 = 0; task1 < numJobs-1; task1++) {
    for (int task2 = task1+1; task2 < numJobs; task2++) {
      solver.post(LCF.or(
          ICF.arithm(restask[res][task1].getEnd(),
                     "<=",
                     restask[res][task2].getStart()),
          ICF.arithm(restask[res][task2].getEnd(),
                     "<=",
                     restask[res][task1].getStart())));
    }
  }
}
```

```
//for each job, create precedence constraints between its
//tasks
for (int j = 0; j < numJobs; j++) {
  for (int before = 0; before < numRes-1; before++) {
    solver.post(ICF.arithm(jobtask[j][before].getEnd(),
                           "<=",
                           jobtask[j][before+1].getStart()));
  }
  //and post that the last task finishes before or on the
  //makespan
  solver.post(ICF.arithm(jobtask[j][numRes-1].getEnd(),
                         "<=", makespan));
}
```

```
//for each resource, create a cumulative constraint,
//for the relevant tasks, all with height 1, for
//capacity 1

for (int res = 0; res < numRes; res++) {
  solver.post(ICF.cumulative(restask[res],
                             heights[res],
                             capacity[res]));
}
```

Global constraint: cumulative

•**cumulative**
public static Constraint cumulative(Task[] TASKS, IntVar[] HEIGHTS,
IntVar CAPACITY, boolean INCREMENTAL)

Cumulative constraint: 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.

Parameters:
TASKS - TASK objects containing start, duration and end variables
HEIGHTS - integer variables representing the resource consumption of each task
CAPACITY - integer variable representing the resource capacity
INCREMENTAL - specifies if an incremental propagation should be applied
Returns:
a cumulative constraint

A note on the problems

Some of the sample problems are very hard.

FT10 was unsolved (i.e. the optimal solution was not known) for many years.

There are more sophisticated methods for handling scheduling constraints.

Constraint Programming offers an advantage over other techniques in that it is relatively easy to add new arbitrary constraints and cost functions for tasks that would stop other approaches from working.

A better model ?

Rather than assign a start time to each task, we could just specify the total ordering on the tasks on each resource.

If we have n jobs and m resources, then each resource will have n tasks to do.

Any permutation of tasks can be in a solution

There are n! permutations of n tasks.

Therefore there are $n!^m$ possible solutions.

but for now we will use the other model ...

Next lecture ...

cumulative scheduling