Exercises corresponding to CP Programming

Exercise #1:

Let us consider a multistage, multiproduct batch plant, having several units in parallel at each stage. A series of "n" jobs must be scheduled in such plant. Each one has a ready-time rd_j, as well as a due-date d_j. The batch size of each job is known in advance.

Let **Task[j,t]** be the processing task associated to job j at stage s. The following variables are related to **Task[j,t]**:
Task[j,t].start, Task[j,t].end, Task[j,t].duration

Let **Us** be the set of equipment units available at stage **s**. The processing time (**ProcTime[j,u]**) of the batch associated to job j in each of the available units **u** that belongs to **Us** is known in advance.

It is well known that the performance of a CP formulation improves when new relevant constraints are added. Provide analytical expressions for the following additional constraints to be included in the model:

(i) Lower bound (earliest start time) associated to each **Task[j,t].start** variable
(ii) Upper bound (latest completion time) associated to each **Task[j,t].end** variable

Exercise #2:

Let us consider the same case study that was described in the previous exercise. The basic CP model resorts to the constraint :

```
forall(st in Stages)
        forall(j in Jobs)
               Task[j,st] requires s
```

in order to indicate that each processing task should be assigned to a piece of equipment.

Since equipment units are declared as unary resources, they can execute at most one operation at a given time. So, job overlapping will not exist at any unit.

The construct **activityHasSelectedResource(Task[j,St],s,tool[unit])** can be used in extra speeding-up constraints that reinforce the previous ideas and may facilitate domain reduction and constraint propagation procedures.

Use this special construct (that evaluates to true or "1" when it holds, or false ("0") otherwise). Propose expressions representing the following additional constraints:

(i) A given **Task[j,st]** must be assigned to just one processing unit.
(ii) If any two activities **Task[j,st]** and **Task[j´,st]**, with **j ≠ j´**, are assigned to the same unit, a disjunctive constraint holds between them.

Exercise #3:

Let us consider the same case study that was described in Exercise # 1. Let us assume that the following data definitions are included in the model:

```
struct forbidden{
   Jobs job;
   Equipment unit;
   };

{forbidden} forbiddenJU=...;
```

The first one declares the existence of a structure, representing forbidden job-unit assignments. The second one defines a particular instance of such structure. The data corresponding to this second element may be:

```
forbiddenJU={<j1,e1>};
```

Thus, it indicates that job #1 cannot be processed in equipment unit e1.

Generate an additional constraint that forbids the assignment of jobs to their associated prohibited units.

Exercise #4:

Let us consider the same case study that was described in Exercise # 1. Let us assume that the following data definitions are included in the model:

```
struct topology{
   Equipment beforeUnit;
   Equipment afterUnit;
   };

{topology} PlantTopology=...;
```

The first one declares the existence of a structure, representing pairs of equipment units that are indeed connected. The second one defines a particular instance of such structure. The data corresponding to this second element may be:

```
PlantTopology ={<e1,e3>, <e1,e4>, ….};
```

Thus, it indicates that equipment unit e1 is connected to e3 and e4, and so on and so forth.

Generate a constraint that will guarantees that any two consecutive activities associated to a given job are assigned to two units, if and only if they are connected among themselves.

Exercise #5:

Though it is redundant, propose the set of data and constraints that will assure that any two consecutive activities, associated to a given job, **will not be** assigned to two units if these units are not connected among themselves. Note: Define a data structure representing non-connected units.