CSE 307, Constraint Logic Programming
TP5: Constraints over finite domains CLP(FD)

Sylvain Soliman

Bachelor of Science of École Polytechnique

1 Reminders

To solve these two problems, you may find useful to look at some of the following operators and predicates in the documentation of SWI-prolog:

arithmetic constraints `#=, #\=, #<, #>, #=<, #>=` as described in class and in the documentation of the `clpfd` library;

`transpose/2` to transpose a matrix represented as a list of lists;

`maplist(Goal, List)` to call `Goal` repeatedly on each element of the list `List`. `maplist/3` and with even more list-based arguments exist, and you can also provide first arguments in `Goal` that will be used each time;

```
?- maplist(plus(1), [1, 2], L).
       L = [2, 3].
```

2 Solving a SUDOKU

We will represent 2D-arrays by lists of lists for simplicity.

1. Write, in the `sudoku.pl` file, a predicate `three_blocks/2` that unifies with its second argument the list of all lists of cells in a 3×3 sub-array of the first argument, that has 3 lines and $3n$ columns.

   For instance:

   ```
   ?- three_blocks([[1,2,3,    4,5,6],
                    [7,8,9,    10,11,12],
                    [13,14,15, 16,17,18]], X).
   X = [[1, 2, 3, 7, 8, 9, 13, 14, 15], [4, 5, 6, 10, 11, 12, 16, 17, 18]]
   ```

2. Now write the `sudoku/1` predicate takes as argument a list of 9 lists of 9 elements (variables or digits) and tries to fill-in the blanks.

   (a) all variables should get a value between 1 and 9. Remember that you can use `V in 1..9` or `[V1, V2] ins 1..9` to enforce that. Note also that `append/2` or `flatten/2` can be used to flatten a list of lists into a (flat) list.

   (b) using the predicates `three_blocks/2` that you have written, and `transpose/2` of `library(clpfd)`, enforce the `all_different` constraint on all rows, all columns and all 3×3 blocks.

   (c) Enforce that you have a solution with `label/1`. Was that necessary in the case of `example1`? of `example2`? What if you use `all_distinct` instead of `all_different`?

3 Job-shop scheduling

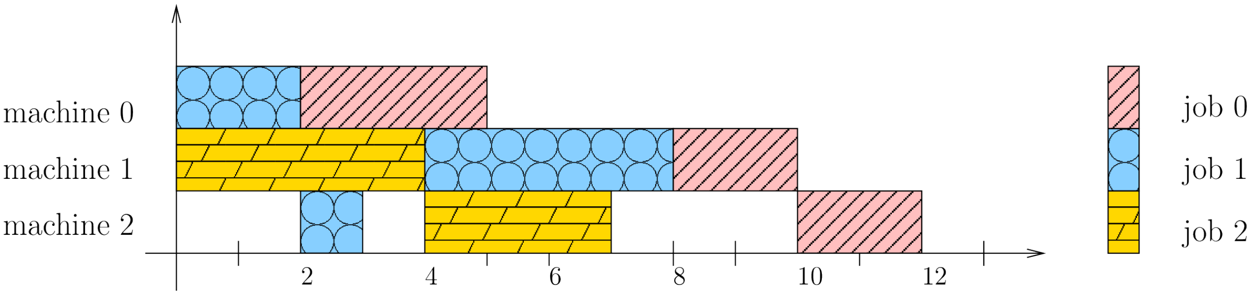
We will now consider the following job-shop scheduling problem:

- job 0 = [(0, 3), (1, 2), (2, 2)]
- job 1 = [(0, 2), (2, 1), (1, 4)]
- job 2 = [(1, 4), (2, 3)]

The problem is defined by a list of jobs, each one constructed from sequential tasks (each has to be finished before the next one starts). Each task also needs a specific machine, and has a specific duration.

Here, tasks are therefore represented by a pair (*machine*, *duration*). The aim is to assign to each of those tasks a starting time such that everything is finished as soon as possible (i.e., minimize the makespan of the problem).

Example (non-minimal) solution:



(source: https://developers.google.com/optimization/scheduling/job_shop)

Most of the structure is already provided in the file `job_shop.pl` At the end, we `label` on the variable `Makespan` first, as a way to minimize it.

1. Write the body of the predicate `precedences/2` that enforces that a sequence of tasks describing a job is indeed done sequentially (the previous one is finished before the next one starts: *start* + *duration* ≤ *next*)

   Do not forget to use the first argument (`Makespan`) to enforce that the last task is finished before the makespan.

2. Write the body of the predicate `non_overlap/2` that enforces that two jobs are non-overlapping (either j_1 is finished before j_2 starts, or the opposite) if they share the same machine.

   One should avoid creating (inefficient) choice-points, and instead use the `#\/` constraint representing a logical OR.