

Constraint Programming 2002: Laboration 2

Introduction: In this lab we will experiment with interval reasoning and bounds consistency for simple arithmetic constraints, and domain propagation for the `serialized` global constraint. You are not allowed to use any CP library to solve this lab.

Laboration 2-1: Bounds Consistency for Linear Sum Constraints

Implement bounds consistency on arithmetic constraints consisting of a linear combination and an inequality using interval reasoning as described in the lectures (and in Stuckey). The constraints you should be able to handle in this lab are on the form

$$c_1x_1 + c_2x_2 + \dots + c_nx_n \text{ CMP } k$$

where x_1, x_2, \dots, x_n are domain variables, **CMP** is one of the symbols in the set $\{<, \leq, =, >, \geq, \neq\}$ and k is a constant. You should be able to handle both integer and floating point variables (that is, you should be able to specify if you want to use integer or floating arithmetics). Note that integer variables give more pruning!

You should also give a set of test examples for the BC algorithm you have implemented. The test cases should include both positive and negative coefficients, and all cases of inequalities.

Laboration 2-2: Bounds Consistency on Nonlinear Constraints

Extend your implementation with bounds consistency propagators for the three constraints

$$x = y \times z$$

$$x = \text{minimum}(y, z)$$

$$x = \text{maximum}(y, z)$$

where x, y and z are domain variables. You must handle the case where any of the domains include 0 correctly (to avoid division by zero). Note that the propagation routine has to be recalled after each pruning step if the domains have changed. As before, your implementation should be able to handle both integer and floating point variables (the user can select either integer or floating point propagation).

Laboration 2-3: Propagation for the `serialized` Constraint

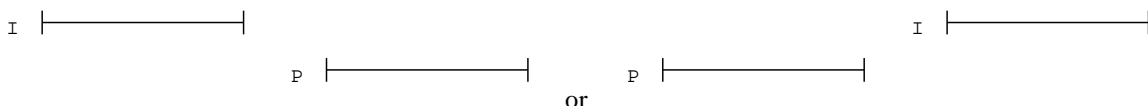
In this lab we will implement a very simple incomplete propagator for the `serialized` constraint. A task is represented by two domain variables, where the first variable represents the start time and the second variable represents the duration of the task. The basic pruning idea is to compute an interval for each task, where the task *must* execute. This interval can then be subtracted from the domains of the other tasks. The interval for each task i with start s_i and duration d_i where i must execute can be calculated as

$$[f, t] = [\bar{s}_i, \underline{s}_i + \underline{d}_i]$$

where \bar{x} and \underline{x} represent the upper and lower bound of the domain for the variable x . If $[f, t]$ is empty, no pruning is possible for this interval.

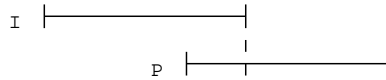
In detail, the following situations can arise when $I = [f, t]$ should be removed from task $t_j = (s_j, d_j)$. We denote the interval of *possible* execution for t_j as $P = [b, e] = [\underline{s}_j, \bar{s}_j + \bar{d}_j]$.

- The intervals I and P do not intersect. This means that the execution of another task in I does not affect the execution of task t_j .



- The interval I partially overlap P , so that P starts after I . In this case we should shrink P from the start (left). This means that we can increase the lower bound on s_j if t is greater than \underline{s}_j :

$$\underline{s}_j \leftarrow \max(\underline{s}_j, t)$$

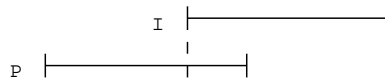


- The interval I partially overlap P , so that P starts before I . In this case we should shrink P from the end (right).
 - Shrink \bar{d}_j if and only if $\underline{s}_j + \bar{d}_j > f$. We can do this by assigning

$$\bar{d}_j \leftarrow \min(\bar{d}_j, f - \underline{s}_j).$$

- Shrink \bar{s}_j if and only if $\bar{s}_j + \underline{d}_j > f$. We can do this by assigning

$$\bar{s}_j \leftarrow \min(\bar{s}_j, f - \underline{d}_j).$$



- The interval I_i overlap I_j intersects in some other way. In this case we either have to remove a part of I_j in the middle, or remove all of I_j . We treat both cases together.

- First we copy I_j into I_j^1 and I_j^2 .
- Process I_j^1 by assuming that we remove I_i from the start of I_j^1 , as in the second case above.
- Process I_j^2 by assuming that we remove I_i from the end of I_j^2 , as in the third case above.
- If I_j^1 is now empty, let $I_j \leftarrow I_j^2$. If I_j^2 is now empty, let $I_j \leftarrow I_j^1$. If neither is empty, we have a disjoint interval, which we have to handle appropriately using our domain representation.



The propagator should be able to prune the domains as above and detect failure. The pruning requires that we can represent holes in the domains of the variables. A possible representation fulfilling this criteria is to represent domains as lists of integer values. Two drawbacks of this approach are that it requires finite domains and that it is wasteful in terms of space and time. A more suitable representation for a domain is to use a list of intervals, which also allow noninteger variables. However, this approach is more difficult to implement. You can choose to implement either of these domain representations. Also, give a set of examples showing that the propagator works as expected.