5.108 disjunctive

DESCRIPTION LINKS GRAPH

Origin [82]

Constraint disjunctive(TASKS)

Synonym one_machine.

Argument TASKS : collection(origin−dvar, duration−dvar)

Restrictions required(TASKS, [origin, duration])
TASKS.duration ≥ 0

Purpose All the tasks of the collection TASKS that have a duration strictly greater than 0 should not overlap.

Example

$$\left\langle \begin{array}{ll} \texttt{origin} - 1 & \texttt{duration} - 3, \\ \texttt{origin} - 2 & \texttt{duration} - 0, \\ \texttt{origin} - 7 & \texttt{duration} - 2, \\ \texttt{origin} - 4 & \texttt{duration} - 1 \end{array} \right\rangle$$

Figure 5.220 shows the tasks with non-zero duration of the example. Since these tasks do not overlap the disjunctive constraint holds.

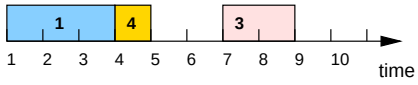

Figure 5.220: Tasks with non-zero duration

Typical |TASKS| > 1
TASKS.duration ≥ 1

Symmetries

- Items of TASKS are permutable.
- TASKS.duration can be decreased to any value ≥ 0.
- One and the same constant can be added to the origin attribute of all items of TASKS.

Remark Some systems like Ilog CP Optimizer also imposes that zero duration tasks do not overlap non-zero duration tasks.

A soft version of this constraint, under the hypothesis that all durations are fixed, was presented by P. Baptiste *et al.* in [17]. In this context the goal was to perform as many tasks as possible within their respective due-dates.

When all tasks have the same (fixed) duration the `disjunctive` constraint can be reformulated as an `all_min_dist` constraint for which a filtering algorithm achieving bound-consistency is available [10].

Within the context of linear programming [190, page 386] provides several relaxations of the `disjunctive` constraint.

Some solvers use in a pre-processing phase, while stating precedence and cumulative constraints, an algorithm for automatically extracting large cliques [79] from a set of tasks that should not pairwise overlap (i.e., two tasks t_i and t_j can not overlap either, because t_i ends before the start of t_j, either because the sum of resource consumption of t_i and t_j exceeds the capacity of a cumulative resource that both tasks use) in order to state `disjunctive` constraints.

Algorithm

We have four main families of methods for handling the `disjunctive` constraint:

- Methods based on the compulsory part [223] of the tasks (also called time-tabling methods). These methods determine the time slots which for sure are occupied by a given task, an propagate back this information to the attributes of each task (i.e., the origin and the duration). Because of their simplicity, these methods have been originally used for handling the `disjunctive` constraint. Even if they propagate less than the other methods they can in practice handle a large number of tasks. To our best knowledge no efficient incremental algorithm devoted to this problem was published up to now (i.e., september 2006).
- Methods based on constructive disjunction. The idea is to try out each alternative of a disjunction (e.g., given two tasks t_1 and t_2 that should not overlap, we successively assume that t_1 finishes before t_2, and that t_2 finishes before t_1) and to remove values that were pruned in both alternatives.
- Methods based on *edge-finding*. Given a set of tasks $\mathcal{T}$, edge-finding determines that some task must, can, or cannot execute first or last in $\mathcal{T}$. Efficient edge-finding algorithms for handling the `disjunctive` constraint were originally described in [83, 84] and more recently in [393, 273].
- Methods that, for any task t, consider the maximal number of tasks that can end up before the start of task t as well as the maximal number of tasks that can start after the end of task t [402].

All these methods are usually used for adjusting the minimum and maximum values of the variables of the `disjunctive` constraint. However some systems use these methods for pruning the full domain of the variables. Finally, *Jackson priority rule* [199] provides a necessary condition [84] for the `disjunctive` constraint. Given a set of tasks $\mathcal{T}$, it consists to progressively schedule all tasks of $\mathcal{T}$ in the following way:

- It assigns to the first possible time point (i.e., the earliest start of all tasks of $\mathcal{T}$) the available task with minimal latest end. In this context, available means a task for which the earliest start is less than or equal to the considered time point.
- It continues by considering the next time point until all the tasks are completely scheduled.

Systems

`disjunctive` in **Choco**, `unary` in **Gecode**.

See also

common keyword: `calendar`, `disj` *(scheduling constraint)*.

generalisation: `cumulative` *(*`task` `heights` *and* `resource` `limit` *are not necessarily all equal to* 1*)*, `diffn` *(*`task` *of* `heigth` 1 *replaced by orthotope)*.

specialisation: `all_min_dist` *(*`line` `segment` *replaced by* `line` `segment`*, of same length)*, `alldifferent` *(*`task` *replaced by* `variable`*)*.

Keywords

characteristic of a constraint: core.

complexity: sequencing with release times and deadlines.

constraint type: scheduling constraint, resource constraint, decomposition.

filtering: compulsory part, constructive disjunction, Phi-tree.

modelling: disjunction, sequence dependent set-up, zero-duration task.

modelling exercises: sequence dependent set-up.

problems: maximum clique.

Arc input(s)	TASKS
Arc generator	$CLIQUE(<) \mapsto$ collection(tasks1, tasks2)
Arc arity	2
Arc constraint(s)	$\bigvee \left(\begin{array}{l} \texttt{tasks1.duration} = 0, \\ \texttt{tasks2.duration} = 0, \\ \texttt{tasks1.origin} + \texttt{tasks1.duration} \leq \texttt{tasks2.origin}, \\ \texttt{tasks2.origin} + \texttt{tasks2.duration} \leq \texttt{tasks1.origin} \end{array} \right)$
Graph property(ies)	**NARC**$= \lvert\texttt{TASKS}\rvert * (\lvert\texttt{TASKS}\rvert - 1)/2$

Graph model We generate a *clique* with a non-overlapping constraint between each pair of distinct tasks and state that the number of arcs of the final graph should be equal to the number of arcs of the initial graph.

Parts (A) and (B) of Figure 5.221 respectively show the initial and final graph associated with the **Example** slot. The `disjunctive` constraint holds since all the arcs of the initial graph belong to the final graph: all the non-overlapping constraints holds.

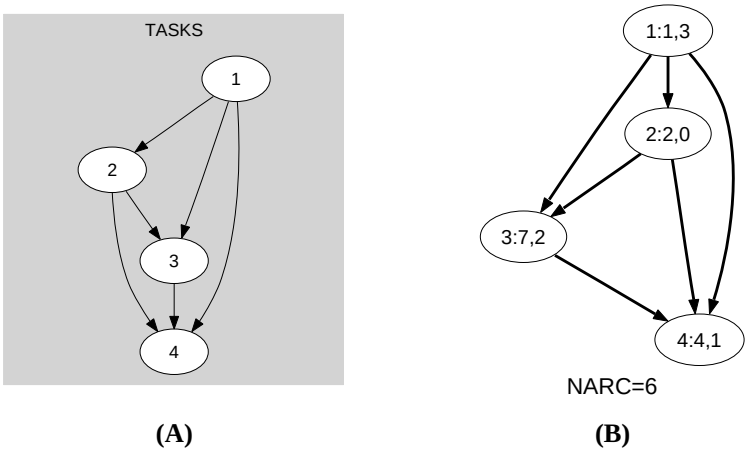

Figure 5.221: Initial and final graph of the `disjunctive` constraint