5.125 disjoint_tasks

DESCRIPTION LINKS GRAPH

Origin

Derived from `disjoint`.

Constraint

`disjoint_tasks(TASKS1, TASKS2)`

Arguments

TASKS1 : collection(origin−dvar, duration−dvar, end−dvar)
TASKS2 : collection(origin−dvar, duration−dvar, end−dvar)

Restrictions

require_at_least(2, TASKS1, [origin, duration, end])
TASKS1.duration ≥ 0
TASKS1.origin $\leq$ TASKS1.end
require_at_least(2, TASKS2, [origin, duration, end])
TASKS2.duration ≥ 0
TASKS2.origin $\leq$ TASKS2.end

Purpose

Each task of the collection TASKS1 should not overlap any task of the collection TASKS2. Two tasks overlap if they have an intersection that is strictly greater than zero.

Example

$$\left(\begin{array}{l}\left\langle\begin{array}{lll}\texttt{origin}-6 & \texttt{duration}-5 & \texttt{end}-11,\\ \texttt{origin}-8 & \texttt{duration}-2 & \texttt{end}-10\end{array}\right\rangle,\\ \left\langle\begin{array}{lll}\texttt{origin}-2 & \texttt{duration}-2 & \texttt{end}-4,\\ \texttt{origin}-3 & \texttt{duration}-3 & \texttt{end}-6,\\ \texttt{origin}-12 & \texttt{duration}-1 & \texttt{end}-13\end{array}\right\rangle\end{array}\right)$$

Figure 5.279 displays the two groups of tasks (i.e., the tasks of TASKS1 and the tasks of TASKS2). Since no task of the first group overlaps any task of the second group, the `disjoint_tasks` constraint holds.

Typical

$|$TASKS1$| > 1$
TASKS1.duration > 0
$|$TASKS2$| > 1$
TASKS2.duration > 0

Symmetries

- Arguments are permutable w.r.t. permutation (TASKS1, TASKS2).
- Items of TASKS1 are permutable.
- Items of TASKS2 are permutable.
- One and the same constant can be added to the `origin` and `end` attributes of all items of TASKS1 and TASKS2.

Arg. properties

- Contractible wrt. TASKS1.
- Contractible wrt. TASKS2.

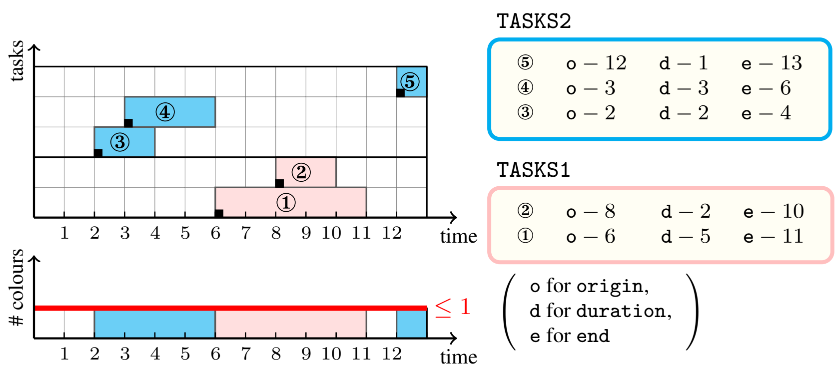

Figure 5.279: The `disjoint_tasks` solution to the **Example** slot with at most one distinct colour in parallel (tasks in `TASKS1` have the pink colour, while tasks in `TASKS2` have the blue colour)

Remark Despite the fact that this is not an uncommon constraint, it cannot be modelled in a compact way with a single `cumulative` constraint. But it can be expressed by using the `coloured_cumulative` constraint: We assign a first colour to the tasks of `TASKS1` as well as a second distinct colour to the tasks of `TASKS2`. Finally we set up a limit of 1 for the maximum number of distinct colours allowed at each time point.

Reformulation The `disjoint_tasks` constraint can be expressed in term of $|\mathtt{TASKS1}| \cdot |\mathtt{TASKS2}|$ reified constraints. For each task $\mathtt{TASKS1}[i]$ ($i \in [1, |\mathtt{TASKS1}|]$) and for each task $\mathtt{TASKS2}[j]$ ($j \in [1, |\mathtt{TASKS2}|]$) we generate a reified constraint of the form $\mathtt{TASKS1}[i].\mathtt{end} \leq \mathtt{TASKS2}[j].\mathtt{origin} \vee \mathtt{TASKS2}[j].\mathtt{end} \leq \mathtt{TASKS1}[i].\mathtt{origin}$. In addition we also state for each task an arithmetic constraint that states that the end of a task is equal to the sum of its origin and its duration.

Systems disjoint in **Choco**.

See also **generalisation:** `coloured_cumulative` *(tasks colours and limit on maximum number of colours in parallel are explicitly given)*.

specialisation: `disjoint` *(*`task` *replaced by* `variable`*)*.

Keywords **constraint type:** scheduling constraint, temporal constraint.

geometry: non-overlapping.

Arc input(s)	TASKS1
Arc generator	$SELF \mapsto$ collection(tasks1)
Arc arity	1
Arc constraint(s)	tasks1.origin + tasks1.duration = tasks1.end
Graph property(ies)	**NARC**= |TASKS1|

Arc input(s)	TASKS2
Arc generator	$SELF \mapsto$ collection(tasks2)
Arc arity	1
Arc constraint(s)	tasks2.origin + tasks2.duration = tasks2.end
Graph property(ies)	**NARC**= |TASKS2|

Arc input(s)	TASKS1 TASKS2
Arc generator	$PRODUCT \mapsto$ collection(tasks1, tasks2)
Arc arity	2
Arc constraint(s)	• tasks1.duration > 0 • tasks2.duration > 0 • tasks1.origin $<$ tasks2.end • tasks2.origin $<$ tasks1.end
Graph property(ies)	**NARC**= 0

Graph model

$PRODUCT$ is used in order to generate the arcs of the graph between all the tasks of the collection TASKS1 and all tasks of the collection TASKS2. The first two graph constraints respectively enforce for each task of TASKS1 and TASKS2 the fact that the end of a task is equal to the sum of its origin and its duration. The arc constraint of the third graph constraint depicts the fact that two tasks overlap. Therefore, since we use the graph property **NARC** $= 0$ the final graph associated with the third graph constraint will be empty and no task of TASKS1 will overlap any task of TASKS2. Figure 5.280 shows the initial graph of the third graph constraint associated with the **Example** slot. Because of the graph property **NARC** $= 0$ the corresponding final graph is empty.

Signature

Since TASKS1 is the maximum number of arcs of the final graph associated with the first graph constraint we can rewrite **NARC** $=$ |TASKS1|. This leads to simplify $\underline{\overline{\mathbf{NARC}}}$ to $\overline{\mathbf{NARC}}$.

We can apply a similar remark for the second graph constraint.

Finally, since 0 is the smallest number of arcs of the final graph we can rewrite **NARC** $=$ 0 to **NARC** ≤ 0. This leads to simplify $\underline{\overline{\mathbf{NARC}}}$ to $\underline{\mathbf{NARC}}$.

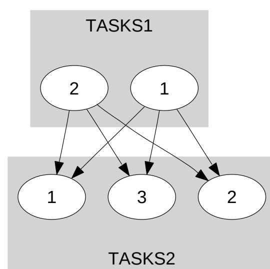

Figure 5.280: Initial graph of the `disjoint_tasks` constraint (the final graph is empty)