Workforce Scheduling

To load this project, open project loader and search for sched_tasks. Here we add comments to the mod file for model explaining.

// Problem Description : workforce scheduling problem.

//

// A set of resources (tuple ResourceDat) of different types is available to perform some requests.

// Different types of requests are considered (tuple RequestType).

// A given type of request can be decomposed into a set of tasks (tuple Recipe) and

// some temporal dependencies between those tasks (tuple Dependency).

// Each task is associated with a processing time (tuple Task) and a set of resource requirements (tuple Requirement).

// A requirement consists of a task type, a resource type and a quantity

// (number of resources of the specified type to be used for executing this type of task).

//

// The objective is to schedule a set of requests with individual due dates so as to minimize the total tardiness.

//

// Model and Constraints

//

// Each request, task and possible allocation of a task to a resource for a requirement

// is modelled as an interval variable.

// A request spans its tasks.

// Allocations are optional.

```
// Each requirement posts a generalized alternative constraint between
// the task and the set of possible allocations for this requirement.
// The cardinality of this generalized alternative is the number of required resources.
//
// Each resource is a sequence of its non-overlapping allocations.
//
// The objective is to minimize total tardiness.
//
// Redundant cumul.
//
// In this model, a redundant cumul function is used to globally constrain
// the number of resources of a certain type simultaneously used by the tasks.
// This cumul is limited by the number of resources of the resource type.
// These redundant cumuls may help in some models as they enforce a stronger
inference
// in the engine while the whole set of resources for the tasks is still not completely
chosen.
// For more complex problems, e.g resources with several resource types / skills,
// other partitions of the resource set may define efficient redundant cumul.

using CP;
```

```
// Data for resources, requests and tasks
// Each resource as a person has id, type and name. The type can be technician and etc..There are same resources types
tuple ResourceDat {
   key int id;
   string   type;
   string name;
};


// Each request has id, type, due date and name
tuple RequestDat {
   key int id;
   string   type;
   int      duedate;
   string   name;
};
// A request is made of a subet of these tasks. Each task has a real name (type), process time and a tag for easy referencing
tuple TaskDat {
   key int id;
   string   type;
   int      ptime;
```

```
   string   name;
};

{RequestDat}   requests   = ...;
{ResourceDat} resources = ...;
{TaskDat}        tasks        = ...;

// inferred data, get all the resource types as a unique set
{string} resourceTypes = { r.type | r in resources };

// Data for template recipes, dependencies and requirements
// Recipe means for each request, all required tasks are listed as pairs
tuple Recipe {
   string request;
   string task;
};

// This is the dependency constraint between tasks for a request
tuple Dependency {
   string request;
   string taskb;
   string taska;
```

```
  int      delay;
};

// This is for each task, the resource type needed and quantity
tuple Requirement {
  string task;
  string resource;
  int      quantity;
};

{Recipe}         recipes          = ...;
{Dependency}   dependencies = ...;
{Requirement} requirements = ...;

// Set of operations (tasks of a request) and allocations (operation on a possible
resource)
// Inferred data: append data together for easy access
// each request object, append ths task data
tuple Operation {
  RequestDat request;
  TaskDat      task;
};
```

```
// For each request/task (Operation), append the requirement object (resource needed)
and corresponding resource object(resource detail)
tuple Allocation {
  Operation    dmd;
  Requirement req;
  ResourceDat resource;
};
// Now define these inferred data, For conditions, use semicolon
{Operation} operations =
  { <r, t> | r in requests,   m in recipes, t in tasks :
    r.type == m.request && t.type == m.task};

// Each one is a possible allocation for the task of the request
{Allocation} allocs =
  { <o, m, r> | o in operations, m in requirements, r in resources :
    o.task.type == m.task && r.type == m.resource};

// define intervals for each request
dvar interval tirequests[requests];
// define intervals for each operation with process time defined
dvar interval tiops[o in operations] size o.task.ptime;
// each of the operaton is done with optional allocation
```

```
dvar interval tiallocs[allocs] optional;

// define a sequence for each resource such that allocation uses this resource

dvar sequence workers[r in resources] in all(a in allocs: a.resource == r) tiallocs[a];

// calculate the resouece capacity for each resource type

int levels[rt in resourceTypes] = sum(r in resources : r.type == rt) 1;


//define a cumulative function for each resource. This is based on the operation

intervals that use this resource

// Each operation interval links to a specific task defined in the requirment, the

requirment links to the resource needed.

cumulFunction cumuls[rt in resourceTypes] =

  sum(rc in requirements, o in operations : rc.resource == rt && o.task.type == rc.task)

pulse(tiops[o], rc.quantity);

// Objective is minimise lateness

minimize sum(t in requests) maxl(0, endOf(tirequests[t]) - t.duedate);

subject to {

  forall(r in requests) {

    // span for each requiret interval is the minimum starting of all operations and the

max end of all operations

    span(tirequests[r], all(o in operations : o.request == r) tiops[o]);

    forall (o in operations : o.request == r) {

      forall (rc in requirements : rc.task == o.task.type) {
```

```
        // for each operation iterval, and each needed requirement, allocation a resource that meets this resouce demand
        alternative(tiops[o], all(a in allocs : a.req == rc && a.dmd == o) tiallocs[a], rc.quantity);
      }
      forall(tc in dependencies: tc.request == r.type && tc.taskb == o.task.type) {
        // for each operation of the request as taskb in the dependencies, find the the asscoiated task a
        // define this operation must finish before task a
        forall(o2 in operations : o2.request == r && tc.taska == o2.task.type) {
          endBeforeStart(tiops[o], tiops[o2], tc.delay);
        }
      }
    }
  }
  // worker no overlap
  forall(r in resources) {
    noOverlap(workers[r]);
  }
  // resource demand < capacity
  forall(r in resourceTypes: levels[r] > 1) {
    cumuls[r] <= levels[r];
```

```
    }

};
```