Lab 6: Staff Scheduling

Problem Description

In this workshop you will model a staff scheduling problem using CP Optimizer's specialized scheduling keywords and syntax.

Principles of the problem:

- A telephone company must schedule customer requests for three different types of installations (request types).
- Each request has a requested due date; a due date can be missed, but the objective is to minimize the number of days late.
- The three request types each have a list of tasks that must be completed in order to complete the request.
- There are precedence constraints associated with some of the tasks.
- Each task has a fixed duration and also may require certain fixed quantities of specific types of resources.
- There are specific resource types and fixed numbers of resources of each type available.

Problem data

There are 3 different types of requests:

- **`FirstLineInstall`**
- **`SecondLineInstall`**
- **`ISDNIntall`**

There are 6 task types:

- MakeAppointment
- FlipSwitch
- InteriorSiteCall
- ISDNInteriorSiteCall
- ExteriorSiteCall
- TestLine

There are 5 resource types:

- **`Operator`**
- **`Technician`**
- **`CherryPicker`** (a type of crane)
- **`ISDNPacketMonitor`**
- **`ISDNTechnician`**

The details of the problem, constraining conditions, objective and advice on how to model them is found in the workbook for this training.

Exercise folder

`<trainingDir>\OPL63.labs\Scheduling\Staff\work\sched_staffWork`

Step 1: Declare task interval and precedences

Objective

- Start building a scheduling model using some basic CP Optimizer constructs.

Actions

- Examine data and model files
- Define the tasks and precedences
- Solve the model and examine the output

References

interval
endBeforeStart

Examine data and model files

1. Import the **`sched_staffWork`** project into the OPL Projects navigator (Leave the **Copy projects into workspace** box unchecked) and open the **`step1.mod`** and **`data.dat`** files.

   The **`.mod`** file represents a part of what the finished model will look like. Most of the model is already done.

   Note that there is no objective function. At this point, you have what is called a **satisfiability problem**. Running it will determine values that satisfy the constraints, without the solution necessarily being optimal.

2. Examine closely how the data declarations for the model are formulated.
3. Note especially, the declaration of the set **`demands`**. This creates a set whose members come from the tuple **`Demand`**, which is a tuple of tuples (**`RequestDat`** and **`TaskDat`**). This set is made sparse by filtering it such that only task/request pairs that are found in the tuple set **`recipes`** are included. Effectively, it creates a sparse set of required tasks to be performed in a request and operations (the same tasks associated with a given resource). Only valid combinations are in the set.

   This is a good example of the power of tuple sets to create sparse sets of complex data.

4. The file **`sched_staff.dat`** instantiates the data as outlined in the problem definition. it instantiates
   - **`ResourceTypes`**
   - **`RequestTypes`**
   - **`TaskTypes`**
   - **`resources`**
   - **`requests`**
   - **`tasks`**
   - **`recipes`**
   - **`dependencies`**
   - **`requirements`**

   Discuss how model and data files are related with your instructor and fellow students.

Define the tasks and precedences

You are now ready to start declaring decision variables and constraints. At this point, we will define only the tasks, and the precedence rules that control them.

1. Declare an interval decision value to represent the time required to do each request/operation pair in the set **`demands`**. Name the decision variable **`titasks`**:

```
dvar interval titasks[d in demands] size d.task.ptime;
```

2. An important aspect of the modeling is expressing the precedence constraints on the tasks (demands). These constraints can be expressed using the constraint **`endBeforeStart`**.

   The **`step1.mod`** file already contains the preparatory declarations:

   ```
   forall(d1, d2 in demands, dep in dependencies :
     d1.request == d2.request &&
     dep.taskb == d1.task.type &&
     dep.taska == d2.task.type)
   ```

   Examine these declarations with your instructor to understand clearly what they mean.

3. Write the **`endBeforeStart`** constraint.
4. Compare with the solution in **`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\step1.mod`**

Solve the model and examine the output

1. Solve the model by right clicking the **Step1** run configuration and selecting **Run this** from the context menu.
2. Look at the results in the **Solutions** output tab. Can you determine what the displayed values represent?
3. Look at the **Engine log** and **Statistics** output tabs, and note that this model is, for the moment, noted as a “Satisfiability problem.”
4. Close **`Step1.mod`**.

Step 2: Compute the end of a task and define the objective

Actions

- Compute the time needed for each request
- Transform the business objective into the objective function
- Solve the model and examine the output

References

span
all
maxl

Compute the time needed for each request

1. Open **`Step2.mod`** for editing.
2. The requests are modeled as interval decision variables. Write the following declaration in the model:

   ```
   dvar interval tirequests[requests];
   ```

3. Write a **`span`** constraint to link this decision variable to the appropriate **`titasks`** instances .

   Use the **`all`** quantifier to associate the required tasks for each request with the appropriate duration.

4. Check your solution against **`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step2.mod`**.

Transform the business objective into the objective function

The business objective requires the model to minimize the total number of late days (days beyond the due date when requests are actually finished). To do this in the model, you need to write an objective function that minimizes the time for each request that exceeds the due date.

1. Calculate the number of late days for each request:
   - The data element **`requests`** is the set of data that instantiates the tuple **`RequestDat`**. The **`duedate`** is included in this information.
   - The interval **`tirequests`** represents the time needed to perform each request.
   - Subtract the **`duedate`** from the date on which **`tirequests`** ends.

     Use the **`endof`** function to determine the end time of **`tirequests`**.

2. Include a test that discards any negative results (requests that finish early) from the objective function.

   Use the **`maxl`** function to select the greater of:
   - the difference between due date and finish date
   - 0

3. Minimize the sum of all the non-negative subtractions, as calculated for each request.

Check the solution in **`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step2.mod`**.

Solve the model and examine the output

1. Solve the model by right clicking the **Step2** run configuration and selecting **Run this** from the context menu.

2. Look at the **Engine log** and **Statistics** output tabs, and note that the model is now reported as a “Minimization problem,” after the addition of the objective function. Scroll down a little further and notice the number of fails reported.
3. Look at the results in the **Solutions** output tab. You will notice that an objective is now reported, in addition to the values of **`titasks`** and **`tirequests`**.

Step 3: Define the resource constraints

Actions

- Review the needs
- Assign workers to tasks
- “Just enough” constraint
- Check your work and solve the model
- Synchronize simultaneous operations and observe the effects

References

synchronize
optional
sequence

Review the needs

So far, you have defined the following constraints as identified in the business problem, as outlined in the workbook:

- For each demand task, there are exactly the required number of task-resource **`operation`** intervals present.
- Each task precedence is enforced.
- Each **`request`** interval decision variable spans the associated **`demand`** interval decision variables.

You now need to meet the following needs, not yet dealt with in the model:

- Each task-resource **`operation`** interval that is present is synchronized with the associated task's **`demand`** interval.
- There is no overlap in time amongst the present **`operation`** intervals associated with a given resource.
- At any given time, the number of overlapping task-resource **`operation`** intervals for a specific resource type do not exceed the number of available resources for that type.

Assign workers to tasks

It is now time to deal with the question of who does what. We know from the data that there is more than one resource, in some cases, capable of doing a given task. How do we decide who is the best one to send on a particular job?

The key idea in representing a scheduling problem with alternative resources is:

- Model each possible task-resource combination with an *optional* interval decision variable.
- Link these with an interval decision variable that represents the entire task itself (using a **`synchronize`** constraint).

1. Open **`Step3.mod`** for editing.
2. You will see that a new data declaration has been added:

```
tuple Operation {
    Demand      dmd;
    ResourceDat resource;
};
{Operation} opers = {<d, r >| d in demands, m in requirements, r in
resources : d.task.type == m.task && r.type == m.resource};
```

   The members of the tuple set **`opers`** are the set of tasks assigned to a resource.

3. There is also a new decision variable associated with this tuple set that calculates the time required for each **`operation`**:

```
dvar interval tiopers[opers] optional;
```

Note that this variable is optional. If one of the optional interval variables is present in a solution, this indicates that the resource associated with it is assigned to the associated task.

Remember that in this model a task is called a **demand**, and a task-resource pair is called an **operation**.

4. Declare a **sequence** decision variable named **workers**, associated with each resource.

   Use **all** to connect each **resource** used in an **operation** to its related **tiopers** duration:

```
dvar sequence workers[r in resources] in all(o in opers : o.resource ==
 r) tiopers[o];
```

5. Constrain this decision variable using a **noOverlap** constraint to indicate the order in which a resource performs its **operations**.

"Just enough" constraint

Another constraint states that for each demand task, there are exactly the required number of task-resource operation intervals present ("just enough" to do the job – not more or less). The presence of an optional interval can be determined using the **presenceOf** constraint:

```
forall(d in demands, rc in requirements : rc.task == d.task.type) {
 sum (o in opers : o.dmd == d && o.resource.type == rc.resource)
presenceOf(tiopers[o]) == rc.quantity;
```

- Write this into the model file.

Check your work and solve the model

1. Compare your results with the contents of **<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step3.mod**

   Do not yet copy the synchronization constraint into your **work** copy. First you are going to solve the model and observe the results.

2. Solve the model by right clicking the **Step3** run configuration and selecting **Run this** from the context menu.
3. Look at the **Engine log** and **Statistics** output tabs, and note that the number of variables and constraints treated in the model has increased slightly.
4. The results in the **Solutions** output tab show values for four decision values now, as well as the solution.

Synchronize simultaneous operations and observe the effects

1. Declare a constraint that synchronizes each task-resource **operation** interval that is present with the associated task's **demand** interval:

```
forall (r in requests, d in demands : d.request == r)
      synchronize(titasks[d], all(o in opers : o.dmd == d) tiopers[o]);
```

2. Solve the model by right clicking the **Step3** run configuration and selecting **Run this** from the context menu.
3. Look at the **Engine log** and **Statistics** output tabs. The number of variables and constraints treated in the model has increased significantly, as has the number of fails.
4. Look at the results in the **Solutions** output tab, and note, especially how values for **workers** have changed from the previous solve.
5. Close **Step3.mod**.

Step 4: Add a surrogate constraint to accelerate search

Actions

- Declare the cumulative function
- Constrain the cumulative function
- Solve the model and examine the results

Reference

cumulFunction

Declare the cumulative function

1. Open **`Step4.mod`** for editing.
2. To model the surrogate constraint on resource usage, a cumulative function expression is created for each resource type. Each **`cumulFunction`** is modified by a **`pulse`** function for each demand. The amount of the **`pulse`** changes the level of the **`cumulFunction`** by the number of resources of the given type required by the demand:

```
cumulFunction cumuls[r in ResourceTypes] =
  sum (rc in requirements, d in demands : rc.resource == r && d.task.type
 == rc.task) pulse(titasks[d], rc.quantity);
```

Constrain the cumulative function

1. You will see that a new intermediate data declaration exists:

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

   This is used to test for the presence of a given resource in a resource type.

2. Write a constraint that requires, when a resource is present in a resource type, that the value of the **`cumulFunction`** must not exceed the value of **`levels`**.
3. Compare your results with the contents of **`<trainingDir>\OPL63.labs\Scheduling\Staff\solution\sched_staffSolution\Step4.mod`**

Solve the model and examine the results

- Solve the model by right clicking the **Step4** run configuration and selecting **Run this** from the context menu.
- If you look at the **Engine log** and **Statistics** output tabs, you will note a dramatic improvement in the number of fails, thanks to the surrogate constraint.