IBM

Model development with IBM ILOG CPLEX Optimization Studio

Version 12 Release 5

Contents

Working environment

Welcome!

Please read this section before starting with the exercises.

It contains details on where to find the documentation and the exercises, and information on the context and architecture of the labs.

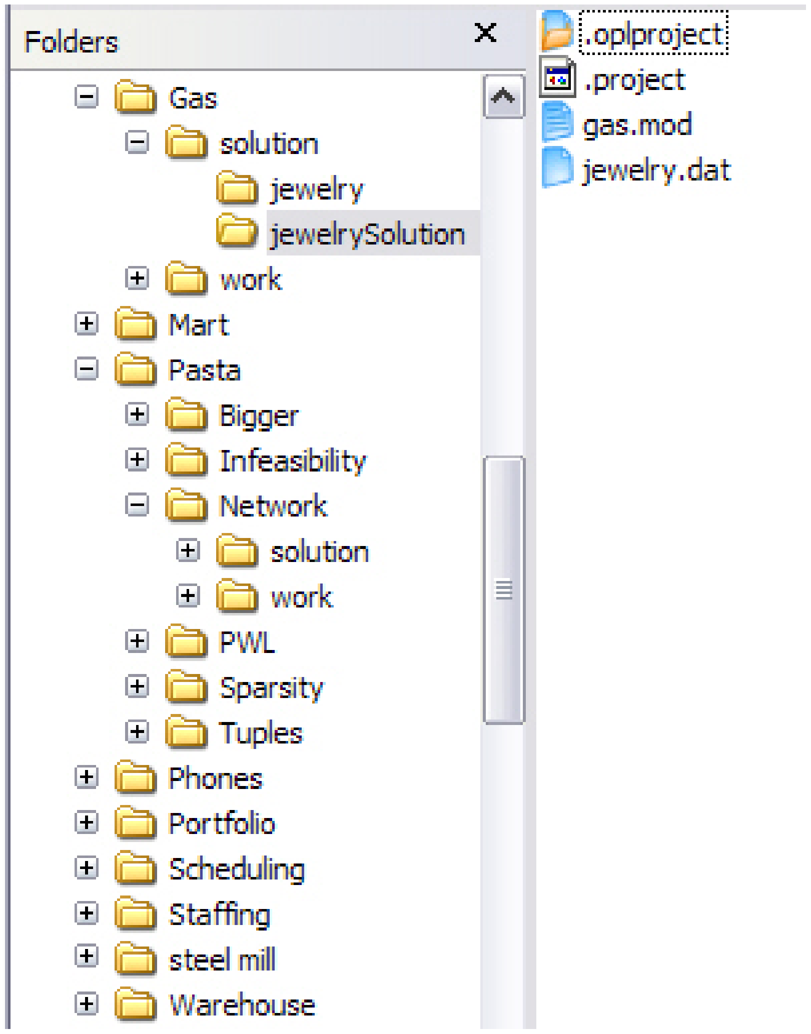
Completing the exercises

Workshop description

- This workshop contains the instructions to complete the exercises.
- For each exercise:
  - The **problem description** explains the context.
  - The **steps** contain instructions to complete the exercise.

  The workshop instructions are repeated in the workbook, at the related workbook content. For some exercises, the steps are distributed across several lessons. It is important to follow the sequence of steps as indicated in the related section of the workbook, and to only complete the step relevant to the current section.

Organization of the exercises



The labs are found in a directory named **`COS125.labs`**,. The exact location of this directory will be given to you by your instructor. The top level directory where training materials are installed is referred to as **`<training_dir>`.** The default location is **`C:\COSTraining`**.

Under **`COS125.labs`,** you can access each exercise separately. Where appropriate, the exercise contains both a **`work`** and a **`solution`** subdirectory. You should start in the **`work`** subdirectory to attempt an exercise, then access the **`solution`** subdirectory to compare your answer with the solution.

Note: Where only one exercise directory is provided, the solution is either already contained in the files, or is given directly in the text of the workshop or training workbook.

The **`C:\COSTraining\CodeSnippets`** directory contains text files with code snippets that you can copy and paste to complete the labs with the same name as the relevant text files. The same code snippets are repeated in this book. Alternatively, you can copy and paste code from the solutions if you get stuck.

Documentation and distribution

Documentation

The IBM ILOG CPLEX Optimization Studio V12.5 documentation set is available:

- On Windows systems, accessible from the Start menu. Click **Start > All Programs > IBM ILOG > CPLEX Optimization Studio V12.5 > Documentation** to access.
- Accessible from within the IDE by selecting **Help > Help Contents** from the main menu.

Distribution

By default, CPLEX Studio V12.5 is installed at **`C:\Program Files\IBM\ILOG\CPLEX_Studio125`**

The instructor may choose a different location at installation time. This top level directory, whether default or custom location, is often referred to as **`<install_dir>`**.

Course prerequisites

- Knowledge of basic algebra.
- Knowledge of the Windows operating systems
- For problems that use CPLEX Optimizer for Mathematical Programming (MP), basic knowledge of MP and modeling concepts
- Basic programming knowledge is helpful with respect to the OPL interfaces, but not required

Chapter 1. Gas production planning

Problem description

In this lab you use the gas production planning model, as described in the workbook, as a starting point to get more familiar with the IBM ILOG CPLEX Optimization Studio IDE.

Exercise folder

`<training_dir>\COS125.labs\Gas\work\gas`

Familiarize yourself with the IDE

Objective

- Familiarize yourself with the CPLEX Studio IDE

Study the project

1. Launch the IDE by clicking its icon on your desktop or in the Windows start menu (**Start > All Programs > IBM ILOG > CPLEX Studio**).
2. Import the project for this problem by selecting **File > Import > Existing OPL 6.x projects** from the main menu. In the **Select root directory** field, navigate to the **`<TrainingDir>\COS125.labs\Gas\work\gas`** directory and select the listed project. Click **Finish**.
3. Expand all the plus signs in the OPL Projects Navigator to see that the project contains two files, **`gas.mod`** (the model file) and **`gas.dat`** (the data file), as well as a default run configuration (labeled **Default (default))**. Both the model and data files have been associated with this run configuration.
4. Double-click the model or data file names to look at the contents.

   Note: Note that in the IDE, keywords are highlighted in blue, comments in green, and string data in purple.
5. Try using contextual help by first selecting **Help > Dynamic Help** from the main menu to open the help window on the right side of the IDE, and then highlighting any keyword in the model file that you'd like information on.
6. Run the project by right-clicking the default run configuration in the OPL Projects Navigator, and selecting **Run this** from the context menu. Examine the result and different output tabs.

Debug an error message

1. Introduce a syntax error in the **`gas.mod`** file by removing the semicolon (;) from the end of one line.
2. Observe the message that appears in the **Problems** Output tab (near the bottom of the screen) and the mark in the margin of the text editor.
3. Try to run the default run configuration (accept the option to save) and see in the **Problems** Output tab that this is not possible before resolving the error.
4. Fix the syntax error, save the model file, and run it.

Reuse the model for jewelry production

Table 1. Data for the jewelry production .dat file

Data Description	Declaration in OPL
Products are: • Rings • Earrings	`Products = { "rings", "earrings" };`
Components are: • Gold • Diamonds	`Components = { "Gold", "Diamonds" };`
Usage for components is: • 3 units of gold and 1 diamond to produce 1 ring • 2 units of gold and 2 diamonds to make 1 set of earrings	`usageFactor = [ [3, 1], [2, 2] ]`
Stock on hand is: • 150 units of gold • 180 diamonds	`stock = [ 150, 180 ]`
Profit for each product is: • ring = 60 • earrings = 40	`profit = [ 60, 40 ]`

Create a new run configuration for jewelry production

In this exercise, you reuse the gas production model for a jewelry production problem. This exercise demonstrates the benefits of separating data and model and CPLEX Studio — an existing model can be used with different data to create a different scenario or application.

1. Create a new run configuration by right-clicking **Run Configurations** in the **Project Navigator** and selecting **New > Run Configuration**. Select the project name as the parent folder, and type `Jewelry production` as the name of the new run configuration.
2. In the **Projects** window, right-click the project name and select **New > Data**.
3. Select the project name as the parent folder, and type `jewelry` for the name of the new data file.
4. Populate the new `jewelry.dat` file with the new data (you can copy the contents of the `gas.dat` file as a starting point).
5. In the OPL Projects Navigator, drag the `gas.mod` and `jewelry.dat` files into the new run configuration.
6. To test whether the new run configuration works, right-click the new run configuration and select **Run this**.
7. Compare your result with the solution at `<TrainingDir>\COS125.labs\Gas\solution\jewelrySolution`

Note: You can create a new OPL project by selecting **File > New > OPL Project**, and filling out the information for the project name, location, and choosing options to create a default run configuration, model file, data file, and settings file. Alternatively, you can create a new OPL project by copying an existing project in the IDE or on the computer file system. If you copy a project directory on the computer file system, you must edit the **`.project`** and **`.oplproject`** files with the new project and file names.

Chapter 2. Telephone production

Problem description

This lab takes you through a basic linear programming problem that demonstrates basics for any OPL model. It also shows how you separate data from the model in the IBM ILOG CPLEX Optimization Studio IDE.

A telephone company processes and sells two kinds of products:
- Desk phones
- Cellular phones

Each type of phone is assembled and painted by the company, which wants to produce at least 100 units of each product.
- A desk phone's processing time is:
  - 12 min. on the assembly machine and
  - 30 min. on the painting machine.
- A cellular phone's processing time is:
  - 24 min. on the assembly machine and
  - 24 min. on the painting machine.
- The assembly machine is available for only 400 hours.
- The painting machine is available for only 490 hours.
- Desk phones return a profit of $12 per unit.
- Cellular phones return a profit of $20 per unit.

The objective is to maximize profit.

Exercise folder

`<training_dir>\COS125.labs\Phones\work`

Note: This directory is empty when you start this lab. You are going to create a project in it.

The text file at **`C:\COSTraining\CodeSnippets\TelephoneProduction.txt`** contains code snippets that you may cut and paste to complete this exercise. Alternatively, you can refer to the solution located at **`<training_dir>\COS125.labs\Phones\solution`**

Write the model

Objective
- Model with OPL

Documentation references

Language > Language Reference Manual > OPL, the modeling language > Data types > Data structures > arrays

Language > Language Reference Manual > OPL, the modeling language > Data types > Basic data types > Floats

Language > Language Quick Reference > OPL keywords > forall

Language > Language Quick Reference > OPL keywords > sum

Write the model

1. Launch the IDE by selecting **Start > All Programs > IBM ILOG > IBM ILOG CPLEX Studio V12.5 > CPLEX Studio IDE**.
2. Create a new project by selecting **File > New > OPL Project**. In the pop-up window, type `phoneswork` for the **Project Name**, and browse to `<training_dir>\COS125.labs\Phones\work` for the **Project Location**.
3. In the **Options**, select all the check-boxes to create a default run configuration with model, data, and option files, and click **Finish**. You will only work with the model file in this step.
4. In the model file, `phoneswork.mod`, declare a set of strings for the product name data elements.
   ```
   {string} Products = {"desk", "cell"};
   ```
5. Declare arrays to specify the other static data elements.

   Tip: All time units must be the same; minutes should be converted into hours.
   ```
   float Atime[Products] = [0.2, 0.4]; //assembly time
   float Ptime[Products] = [0.5, 0.4]; //painting time
   float Aavail = 400; //available time on assembly machine
   float Pavail = 490; //available time on painting machine
   float profit[Products] = [12, 20]; //profit realized from each product
   float minProd[Products] = [100, 100]; //minimum production for each
   product
   ```
6. Declare an array of production decision variables. Use `dvar` to designate a decision variable
   ```
   dvar float+ production[Products];
   ```
7. Declare the expression to maximize

   Tip: use `sum` and `maximize`
8. Declare the constraints

   Tip: use `forall` and `subject to {}`
9. Save the project.

Separate the data from the model

Objective

- Manipulate data initialization

Documentation references

CPLEX Studio IDE > Getting Started with the CPLEX Studio IDE > Getting Started tutorial > Executing a project

Language > Language Reference Manual > OPL, the modeling language > Data sources > Data initialization

Language > Language Reference Manual > OPL, the modeling language > Constraints > Constraint labels > Labeling constraints

Create and fill a .dat file

1. Save a copy of `phoneswork.mod` under the name `phones1.mod` (right-click `phoneswork.mod` and select Copy, then right-click the project name and select Paste. You'll be prompted to enter the name of the new file (`phones1.mod`).

   Tip: Right click the **phonesWork** project and select **refresh** from the context menu, or type `F5`
2. Create a new run configuration (right-click the **phonesWork** project name and select **New > Run Configuration**). Accept the default name `Configuration2` and make sure that you see this (empty) run configuration added to the project in the OPL Projects Navigator.
3. Add the file `phones1.mod` to `Configuration2` by dragging the file to the configuration in the OPL Projects Navigator.
4. Create new data file `phones1.dat` in the project by right-clicking the **phonesWork** project name and selecting **New > Data**.
5. Add file `phones1.dat` to `Configuration2` by dragging the file onto the name of the configuration.
6. Instantiate all the data elements, that are included in `phones1.mod`, in `phones1.dat`.

   Do not include type information from the model file

Note: OPL provides the "..." escape sequence to separate a data declaration from its instantiation. For example, `float Pavail = 490;` becomes `float Pavail = ...;` in the model file. The data type is not given in the data file, for example the data file will contain `Pavail = 490;`, and not `float Pavail = 490;`

Note: It might become difficult to manage a data file if a large number of values are listed. To allow better readability, OPL provides a named instantiation syntax. It is available for arrays or tuples indexed with sets:

- The "#" symbol is used to start and end named array instantiations.
- The symbol ":" separates the name from the value.

Example: `profit = #[desk:12 cell:20]#;`

Update the .mod file

- In the `phones1.mod` file, change the instantiation of data elements so that instance values are taken from the `phones1.dat` file. Use the "`...`" construct.
- Run Configuration2 by right-clicking it and selecting **Run this**.

Note: Note how run configurations can be used to allow different versions of model and data to coexist in a single project.

Compare with an IP solution

Objective

- Compare IP and LP solutions to the telephone production problem.

Set up a new run configuration

1. Open the project `..\COS125.labs\Phones\work\phonesWork.`
2. Select the project in the OPL Projects Navigator. Then select the **File > Copy Files to Project** menu item from the menu bar to copy `..\COS125.labs\phones\solution\phonesSolution\phones2.dat` to `..\COS125.labs\Phones\work\phonesWork`
3. Create a new run configuration (it should be named `Configuration3`).

Populate the run configuration and solve the model

1. Populate `Configuration3` with `phones1.mod` and `phones2.dat.`
2. Examine the data file. In this instance, available time on the painting and assembly machines is not an integer, nor is the profit per each type of phone.
3. Run the model using `Configuration3`. Leave the project open as you are going to return to it in the exercise.

   The model returns a non-integer solution: `production = [299.8 851.93];`

   In reality, you cannot produce 299.8 desk phones and 851.93 cell phones. Production values must be represented by integers.

Compare the IP solution

1. Select the project `phonesWork` in the OPL Projects Navigator. Then Select the **File > Copy Files to Project** menu item from the menu bar to copy.
2. Create a new run configuration (it should be named `Configuration4`).
3. Populate `Configuration4` with the model `phones_MIP.mod` and the data file `phones2.dat.`
4. Examine the model. Note that the `production` decision variable type has been changed from `float` to `int`.
5. Run the model using `Configuration4.`

This time, an integer solution is returned: 299 desk phones and 852 cell phones. Examine these **Output** window tabs:

- **Solutions**
- **Engine Log**
- **Statistics**

In the **Engine Log** you can see that the MIP algorithm has been applied.

The **Statistics** tab includes a progress chart showing how the algorithm progressively converged to the optimal integer solution.

Compare solution times

1. Return to the modified project, and create another run configuration, `Configuration5.`
2. Populate `Configuration5` with `phones_MIP.mod` and `phones1.dat.`
3. Run the model using `Configuration5` and examine the **Profiler** tab. Notice the total time.
4. Now run the model again, but use `Configuration2.`
5. Examine the **Profiler** tab. Notice the difference in time.

Using the data in `phones1.dat` produces an integer solution with both Simplex and MIP algorithms. But the Simplex algorithm is more efficient for solving this problem. While the time difference in this example is relatively insignificant, in very large problems, it can make a big difference. The declaration of **`int`** data will generally force the use of the MIP algorithm. In some cases it may be more efficient to declare **`float`** data, even if the values to be used are integers.

Note: In a real-world situation, for products as small as telephones, with relatively low cost, it is probably more efficient to just round down from a fractional solution than to perform a MIP run. However, if you are manufacturing large, expensive items in quantity, for example, yachts, helicopters or construction cranes, you need an integer solution to avoid wasting resources, time and money.

Chapter 3. Pasta production and delivery

Problem Description

This exercise follows a problem through a series of steps in order to practice the following techniques in IBM ILOG CPLEX Optimization Studio:

- Using tuples
- Creating constraints
- Working with network structures
- Using sparse data structures

To meet the demands of its customers, a company manufactures its products in its own factories (**inside production**) or buys them from other companies (**outside production**).

The level of inside production is subject to some resource constraints: each product consumes a certain amount of each resource. In contrast, outside production is only limited by contractual agreements with the suppliers.

The problem is to determine how much of each product should be produced inside and outside the company while minimizing the overall production cost, meeting the demand, and satisfying the resource and contractual constraints.

You can read a complete description of the problem in the workbook that accompanies this training. See the lesson on **Production Planning and OPL Data Structures**.

Requirements:

- Manufacture products from available ingredients
- Meet the customer demand
- Produce yourself (inside) or by a subsidiary (outside) at different costs
- Satisfy resource capacity constraints for the inside production
- Do not exceed the contractual limits for outside production
- Minimize the overall cost

Exercise folder

`<training_dir>\COS125.labs\Pasta\Tuples\work`

The text file at **`C:\COSTraining\CodeSnippets\PastaProductionAndDelivery.txt`** contains some code snippets that you may cut and paste to complete this exercise. Alternatively, you can refer to the solutions located at **`<training_dir>\\ COS125.labs\Pasta`**

Write a basic model

Objectives

- Manipulate and initialize data
- Write constraints.

Finish the `production.mod` file

Launch the IDE if it isn't open yet, and import the project **`productionWork`** into the OPL Projects Navigator by selecting **File > Import > Existing OPL6.x projects**. Browse to **`<training_dir>\COS125.labs\Pasta\Tuples\work`**, select the **`productionWork`** project, and click **Finish**.

Expand all the + signs in the OPL Projects Navigator and click the .mod file to look at its contents:

- The **`consumption`** array is indexed on **`Products`** and **`Resources`**.
- The **`availability`** array is indexed on **`Resources`**.
- **`demand`** and **`cost`** arrays (inside and outside) are indexed on **`Products`**.
- The decision arrays (**`insideProduction`** and **`outsideProduction dvar`** expressions) are indexed on **`Products`**.

You can easily see the correspondence between the arrays as defined in the model, and the contents of the data file:

```
Products =  { "kluski" "capellini" "fettucine" };
Resources = { "flour" "eggs" };

consumption = [ [0.5, 0.2], [0.4, 0.4], [0.3, 0.6] ];
availability = [ 120, 150 ];
demand = [ 100, 200, 300 ];
insideCost = [ 0.6, 0.8, 0.3 ];
outsideCost  = [ 0.8, 0.9, 0.4 ];
maxOutsideProduction = 200;
```

- Complete the following steps to write the model:
  1. Write the objective:

     The sum, for each product, of the **`insideCost`** times the **`insideProduction`** , plus the **`outsideCost`** times the **`outsideProduction`** .
  2. Write the constraint for all resources:

     The sum of the **`consumption`** times the **`insideProduction`** is less than or equal to the **`availability`** of the **`Resource`**.
  3. Write the constraint for all products:

     The **`insideProduction`** plus the **`outsideProduction`** is greater than or equal to the **`demand`**.
- Save and run your model (right-click the configuration name and select **Run this**).
- You can compare your work to the solution found in **`<training_dir>\\ COS125.labs\Pasta\Tuples\solution\productionSolution.`**

Write a model using tuples

Objectives

- Create a new model using tuple structures
- Create a tuple without an internal array

Documentation references

Language > Language Reference Manual > OPL, the modeling language > Data types > Data structures > Tuples

CPLEX Studio IDE > IDE Reference > Tour of the CPLEX Studio Graphical User Interface > Right-click context menu commands

From array to tuple

If you look at the original arrays declared in the previous step, the only one that is not related to the products is `availability`. The availability limitations, indexed on the resources, will remain as an array.

Convert the model to one that uses tuples

1. Create a new project, `productwork`, and in the work directory by selecting **File > New > OPL Project**. Select the check-boxes for the automatic creation of a model file`productwork.mod`, a data file, `product.dat`, and a default run configuration.
2. In the model file, use a tuple to hold all the information about each product. You should declare demand, inside cost, outside cost, and resource consumption as tuple elements.
3. Redefine the objective function and constraints to use the tuple.

   Tip: You should reference individual members of the tuple in the objective function.
4. Change the initialization in the data file: use the original data, but order it to initialize the new tuples.

   Tip: Use named initialization (`#[`) in the `.dat` file for clarity.
5. Run the default configuration to test the project.
6. You can compare your work to the solution found in `<training_dir>\COS125.labs\Pasta\Tuples\solution\productSolution.`

Create a tuple without an internal array

In the solution you just completed, the tuple element `consumption` is an array. If you later need to import data into this tuple element from a spreadsheet or database, this structure cannot be used (spreadsheet and database connections are explained in another lesson with a corresponding workshop). In this step, you discover a way to avoid this problem.

Note: Arrays inside tuples are permitted, but you can use them **only if** you're not getting your data from a spreadsheet or database.

Steps to change the model:

1. Make a copy of the `<training_dir>\\COS125.labs\Pasta\Tuples\work\productwork` project by selecting the project name and typing `<Ctrl>C` followed by `<Ctrl>V` in the OPL Projects Navigator. When the **Copy Project** popup window appears, change the project name to `product2work`.
2. Open the file `product2work.mod` for editing.
3. The array inside the tuple represents the consumption of resources. The first step is to remove the array `consumption` from the tuple `ProductData`.
4. Define a new tuple, named `consumptionData` specifying, for each kind of pasta, how much of each resource is necessary.

   To use this new structure and maintain the same solution, you need the next three steps.
5. Create a set of this tuple, and name it `consumption`.

6. In the data file, remove the initialization of the **`consumption`** tuple element (which you removed in the model file) and reorganize the same data to initialize the new tuple set, also called **`consumption`**, that you just created.
7. Adapt the **`resourceAvailability`** constraint so that it uses the new data structures to express the same constraint as in the previous version of the model.
8. Run your model by right-clicking the run configuration name and selecting **Run this**.
9. Compare your work to the solution found in **`<training_dir>\\COS125.labs\Pasta\Tuples\solution\product2Work.`**

Solve the infeasible model

Objective

- Gain hands-on experience with the two methods in CPLEX Studio to detect infeasibilities and help you resolve them, namely conflict refinement and relaxation.

Actions

- Study the conflicts and suggested relaxations
- Choose a method
- Modify the feasOpt parameters
- Modify the relaxation level

Documentation references

CPLEX Studio IDE > Getting Started with the CPLEX Studio IDE > Getting Started tutorial > Examining a solution to the model > The Output tabs

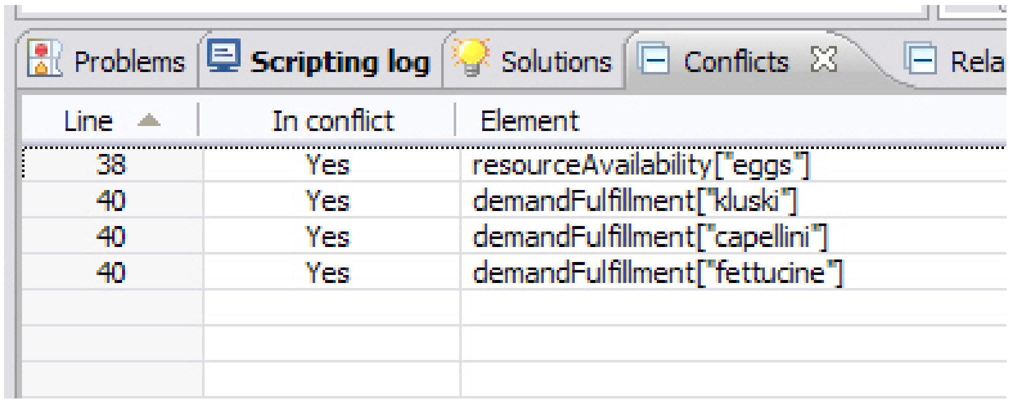
In this step, you use the same model as the one you worked on in the previous step. For now, however, this model has deliberately been made infeasible by changing the value for the upper limit on outsourced production, the value of **`maxOutsideProduction,`** from 200 to 50 in the data file.

Study the conflicts and suggested relaxations

1. Import the project **`<training_dir>\\COS125.labs\Pasta\Tuples\solution\product2Work`** to the OPL Projects Navigator.
2. Run the model and observe the results in these **Output** window tabs:
   - **Engine Log**
   - **Conflicts**
   - **Relaxations**
   - **Solutions**

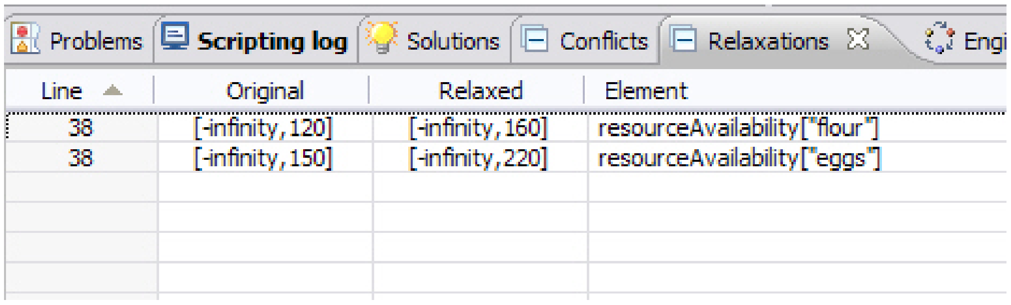
The **Engine Log** tells you **`Implied bounds make row 'resourceAvailability ("flour")' infeasible`** - that is, there is no solution without relaxing at least one constraint.

The **Conflicts** tab tells you that there are conflicts between the **`resourceAvailability`** and **`demandFulfillment`** constraints.



Line	In conflict	Element
38	Yes	resourceAvailability["eggs"]
40	Yes	demandFulfillment["kluski"]
40	Yes	demandFulfillment["capellini"]
40	Yes	demandFulfillment["fettucine"]

The **Relaxations** tab suggests relaxations to the model in **`resourceAvailability.`**



Line	Original	Relaxed	Element
38	[-infinity,120]	[-infinity,160]	resourceAvailability["flour"]
38	[-infinity,150]	[-infinity,220]	resourceAvailability["eggs"]

The proposed relaxation suggests that the supply of eggs be increased from 150 to 220. It also has calculated that if this is done, it will be necessary to increase the flour supply from 120 to 160.

Finally, the **Solutions** tab calculates a "feasible relaxed sum of infeasibilities", that is , a feasible, but not necessarily optimal, solution by minimizing the sum of all required relaxations given in the **Relaxations** tab. In this case, all the outside production is kept to the imposed limit of 50, and the internal capacity is increased to make up the difference in production internally.

Choose a method

Look at the suggested option(s) for solving the problem you have just run in the **Conflicts** and **Relaxations** output tabs.

The problem presents 2 options for removing the infeasibility:

- Either relax the constraint, for example, change **`availability`** to **`160`** for **`flour`**, and **`220`** for **`eggs`***or*
- Remove the conflict by removing either the **`resourceAvailability`** or **`demandFulfillment`** constraint.

Which solution offers the best choice? To answer, it is important to understand the model:

- CPLEX Studio reports the constraints involved in a conflict. You need to determine which constraint is in error. The fact that a conflict exists for one constraint may, in fact, be the result of an error in modeling another constraint.

- Study what is reported in the **Conflicts** and **Relaxations** output tabs, remembering what each constraint and decision variable represents, and apply the solution that makes sense from a practical point of view. Then, complete the following steps:

1. Record the data from the **Solutions** tab for later reference.
2. Try commenting out the `resourceAvailability` constraint and run the problem again. Is the conflict resolved? Is the result meaningful?
3. Restore the `resourceAvailability` constraint and modify the data file as suggested by the **Relaxations** output tab - that is, set

   `availability = [ 160, 220 ];`
4. Run the problem again.
5. Compare the result in the **Solutions** tab to the earlier results you recorded. These solutions are equivalent. Can you think of circumstances in which they would be different?

Note that commenting out the `resourceAvailability` constraint produces a feasible solution in which no outside production takes place, as availability is no longer considered. While the solution is feasible, it is not correct from a real-world perspective, as there will not be enough resources to satisfy the production requirements.

The relaxed solution suggested by CPLEX Studio will not necessarily correspond to the solution obtained after applying the suggested relaxation to the original model. This is because when CPLEX Optimizer attempts to find a relaxed solution, it uses a modified objective function defined by `feasOpt` in order to minimize the constraint violations required for a feasible solution, regardless of the original objective. After the user then "fixes" the infeasibilities by applying the suggested relaxation, the model goes back to using the original objective and therefore may find a different solution.

Modify the feasOpt parameters

You can modify what CPLEX Optimizer takes into account when looking for a relaxation:

- Labeled constraints only
- All decision variables and all labeled constraints

Complete the following steps to see how this works:

1. In the data file, change `availability` to the original values used in steps 1 and 2: `[ 120, 150 ].`
2. In the OPL Projects Navigator, select the settings file, `product2.ops` and click **Feasopt** in the navigation tree.
3. Place your mouse cursor over the document icon next to the **Mode of Feasopt** field to see a description of what this setting does.
4. Select the third item from the dropdown combo-box, **Minimize the number of constraints and bounds requiring relaxation in the first phase only.**

   Note the red exclamation mark to the left of the field label. This indicates that this field is no longer set at its default value. The change can also be noted in the **Outline** area.
5. Run the model, and notice the new suggested relaxation:
   - **Original: `[0,50]`**
   - **Relaxed: `[0,300]`**

- **Element: `outsideProduction["fettucine"]`**

This time, CPLEX Studio suggests that you change the bounds affecting a decision variable, **`outsideProduction,`** which represents the outsourced quantity of **`fettucine.`** In other words, at least for this one product, CPLEX Studio suggests that you change the production limit for the outsourced product (**`maxOutsideProduction`**).

- Look at the relaxed solution suggested in the **Solutions** tab. Does it represent a realistic solution?
- Change the value of **`maxOutsideProduction`** to 300 and run the problem again.
- Compare the solution produced.

Note: You can use IBM ILOG Script to set weights and priorities on constraints during the relaxation process, using the **`IloOplConflictIterator`** and **`IloOplRelaxationIterator`** classes. Refer to **IBM ILOG Script for OPL > Tutorial: Changing Default Behaviors in Flow Control > Setting Preferences on the Search for Conflicts and Relaxations** in the *Language User's Manual* for details.

Modify the relaxation level

1. In the settings (.ops) file, select **Language > General**.
2. In the **Relaxation Level** item, change the option in the drop-down list.
3. Solve the model again, and look at the information displayed in the output tabs.

Factor in a piecewise cost change

Objective

- Redesign the model to take a discontinuous cost change into account.

Action

- Model the cost break

Documentation references

Language > Language User's Manual > The application areas > Applications of linear and integer programming > Piecewise linear programming

Language > Language Reference Manual > OPL, the modeling language > Expressions > Piecewise linear functions

Problem description

Now, imagine that your pasta production facility is getting old, and when you make more than 20 units in a batch, the machine overheats and causes the production cost to go up by a factor of three! This not only changes the inside production cost, it introduces a **breakpoint** where the price takes a sudden, **piecewise** jump.

The problem, then, is to change the model to take the breakpoint and cost change into account.

Model the cost break

1. Import the **`<training_dir>\\COS125.labs\Pasta\PWL\work\productWork`**. project into the OPL Projects Navigator

2. Because the overall cost of inside production now has to be calculated differently from the overall cost of outside production, it is necessary to define them separately. Write two reusable expressions that use the decision variables **`insideProduction`** and **`outsideProduction`** to calculate these values. Name the expressions **`overallInsideCost`** and **`overallOutsideCost`** respectively. **Hint:** use the **`dexpr`** keyword for each of the expressions, and calculate **`overallInsideCost`** using the **`piecewise`** keyword.
3. Rewrite the objective function to minimize the sum of the two overall costs.
4. Run the model and see how the result differs from your original model.
5. Compare your model to the solution found in **`<training_dir>\\COS125.labs\Pasta\PWL\solution\productSolution`**.

Product delivery

Objective

- Describe a network to be modeled for the delivery of the finished product

Problem description

Once the pasta has been produced, it has to be delivered. The pasta company has internal production sites, and an external production partner with its own site.

- Products produced in these diverse locations must be delivered to 7 warehouses of the distributor for onward delivery to customers.
- A road network links the production sites and the warehouses. The company has established routes it uses, with known costs associated.
- The company wants to know how many of each product to ship (i.e. how much to ship from each site) over each existing arc, in order to:
  – meet demand
  – minimize costs

This page describes the problem, gives the data, and lists the steps to complete to arrive at the solution. Read through this page carefully, then go on to the next step, **Add a network to the project**.

Requirements:

- All produced product will be delivered
- Production exactly equals demand
- Only routes which have been approved can be used for shipping
- Any of the products could potentially be shipped on an approved route.
- There is a contractual limit on the amount of outside delivery of any one product to any one warehouse.

Problem data

In addition to the production data you have already entered into the model, you need the following information:

- There are 2 production locations, designated **`L1`** and **`L2.`**
- There are 7 warehouses, designated **`W1 - W7.`**
- The maximum number of an outsourced product to be shipped to any one warehouse is 100.
- Approved arcs are shown in the following table:

		Warehouse						
		W1	W2	W3	W4	W5	W6	W7
Production Site	L1	2	■	2	1	■	■	3
	L2	3	2	■	■	2	1	2
	External	4	■	3	2	4	3	■

Note that because products are now manufactured at multiple sites and shipped to multiple warehouses, most of the model data and decision variables will now be specific to either a production location or a warehouse. For example, product demand is now warehouse-specific and resource availability is now location-specific. You will see in the updated work model, that tuples are used to incorporate the new location-specific format of the data.

Steps to the solution

The steps are summarized below. The next sections include more detail on each step.

- Add the distribution side of the network to the model - declare:
  - the production locations
  - the warehouses
  - the approved arcs with their shipping costs (unapproved arcs have a shipping cost of 0)
- Declare other data and decision variables:
  - The maximum outside delivery of any product to any warehouse
  - The amount of internally produced product to deliver from each location to each warehouse
  - The amount of externally produced product to deliver to each warehouse
- Modify the objective function so that it minimizes shipping as well as production costs
- Create new constraints that require:
  - Only approved shipping routes can be used (i.e. those that have a non-zero cost assigned to them)
  - For each product, the total production at a particular location is equal to the sum of deliveries across all valid routes originating at that location

Add the network to the project

Objective

- Create the network model

Actions

- Add the distribution side of the network
- Declare data and decision variables
- Modify the objective function
- Create new constraints

Documentation references

Language > Language Reference Manual > OPL, the modeling language > Data types > Data structures > Arrays

Language > Language Reference Manual > OPL, the modeling language > Data sources > Data initialization > Initializing arrays

Exercise Folder

Use your model or the one in **`<training_dir>\\COS125.labs\Pasta\Network\work`**

Add the distribution side of the network

1. Import the **`deliveryWork`** project into the OPL Projects Navigator. It is identical to the **`productSolution`** project that you worked on earlier, with indications of where to add new lines.
2. In the model file, the warehouses and production locations are already declared with the code,

   `{string} Warehouses = ...;`

   `{string} Locations = ...;`
3. Declare the arcs:
   - between internal sources and warehouses
   - between external sources and warehouses

   Hint: for each internal source, declare an array indexed by **`Locations`** and **`Warehouses`** and name this array **`internalArc.`** For each external source, declare an array indexed by **`Warehouses`** and name this array **`externalArc`**.
4. In the data file, the 7 warehouses and 2 internal production locations are already initialized.
5. Initialize the arcs with their cost data using the table in the previous panel. Use named initialization for clarity. Arcs not listed in the table should have values of 0, as shown below.

```
internalArc = #[
       L1 : [ 2 , 0 , 2 , 1 , 0 , 0 , 3]
       L2 : [ 3 , 2 , 0 , 0 , 2 , 1 , 2]
            ]#;
externalArc = [ 4 , 0 , 3 , 2 , 4 , 3 , 0];
```

Declare data and decision variables

Declare the following data:

- Maximum allowable outsourced product to be shipped to any warehouse. Name it **`maxOutsideFlow`**.

Declare 2 new decision variables:

- The amount of each internal product to deliver from each location to each warehouse. Name it **`insideFlow`**
- The amount of each external product to deliver to each warehouse. Name it **`outsideFlow`**

  For the external product, add a limitation that prevents this delivery from exceeding **`maxOutsideFlow.`**

In the data file, initialize **`maxOutsideFlow`** with the values given in the previous "Product delivery" step.

For the **objective function,** the **outsideCost** and **insideCost** data are now represented by the following two tuples:

Modify the objective function

```
tuple outsideCostData {
    key string p;
    float oc;
}
{outsideCostData} outsideCost with p in Products = ...;

tuple insideCostData  {
    key string p;
    key string l;
    float ic;
}
{insideCostData} insideCost with p in Products, l in Locations = ...;
```

The objective function already minimizes production costs. It needs to also include an expression that takes into account the following:

- The different costs associated with each authorized arc (**externalArc** or **internalArc**)
- The total costs for both inside and outside deliveries (**insideFlow[Locations][Warehouses]** and **outsideFlow[Warehouses]**)

What needs to be minimized is the total of the shipping costs for all the arcs to be used. The total shipping cost of the arcs for a given product can be expressed as the number of items to be shipped on an arc multiplied by the cost per arc (per item). Write an expression that defines this for:

- Inside production, indexed by product, location and warehouse
- Outside production, indexed by product and warehouse

In the final objective, the production part of the **objective function** remains to minimize manufacturing costs, except that these costs are now specific to each production location:

```
minimize
  sum(<p,l,ic> in insideCost)ic * insideProduction[p][l]
   + sum(<p,oc> in outsideCost)oc * outsideProduction[p])
```

The extra lines minimize shipping costs:

```
  + sum(p in Products, l in Locations, w in Warehouses)(internalArc[l][w] * insideFlow[p][l][w])
  + sum(p in Products, w in Warehouses)(externalArc[w] * outsideFlow[p][w]);
```

Create new constraints

Constraints limiting the arcs:

The model uses the same array to indicate the cost of an arc, or to indicate that the arc is not authorized or does not exist. A non zero value gives the cost of shipping an item on the arc. A zero value indicates that the arc cannot be used. Write constraints that test for the zero value and prevent such arcs from being assigned. **Hint:** the decision variables **insideFlow** and **outsideFlow** represent the number of items to be shipped on an arc. A 0 value assigned to one of these variables means the arc is unused.

Constraints forcing shipment of all product:

One of the requirements is that production equals demand, and that all product be shipped. Write constraints that require that the number of products produced be equal to the number of products shipped, for both inside locations as well as the external site.

Name these constraints **`insideBalance`** and **`outsideBalance`**.

The solution is found in **`<training_dir>\\COS125.labs\Pasta\Network\solution\deliverySolution`**. Compare your model with it.

Use the network algorithm

The CPLEX Network Optimizer can be used when it is advantageous to do so. This problem is an LP problem with a network structure. To see how the Network Optimizer solves this problem, do the following:

1. Use your model, or the one in **`<training_dir>\\COS125.labs\Pasta\Network\solution\deliveryWork`**. It should already be open in the IDE.
2. Run the model and examine the different tabs in the **Output** window.
3. Double click the settings file, **`delivery.ops`** in the project window to open it.
4. Select **Mathematical Programming > General** and set the **Algorithm for continuous problems** to **Network simplex**.
5. Run the model again, and examine the **Engine Log** output tab. Notice the difference in the information displayed.
6. Examine the other output tabs, and compare them to running the problem with the algorithm set to **Automatic**.

You will note that the solution is identical, and no particular advantage is gained by using the network algorithm compared to the automatically chosen simplex algorithm. You can force use of the network algorithm by using the procedure you just completed.

Make the model sparse

Objective

- Redefine the model to take advantage of sparse data structures.

Actions

- Create the "internalLinkData" and "externalLinkData" tuple
- Revise the objective function
- Clean up the constraints
- Initialize the arcs
- Test the model

Documentation references

Language > Language User’s Manual > Introduction to OPL > Modeling tips > Sparsity

Problem description

The pasta company has met with a certain level of success, and wants to expand. If the network of production and distribution becomes very big, it may take too long to solve the model. This exercise involves exploiting the model and data sparsity to formulate a more efficient model to help deal with this possible expansion.

The data file in the **deliveryWork** project you worked on earlier, shows that the array of arcs contains several zeros:

```
internalArc = #[
        L1 : [ 2 , 0 , 2 , 1 , 0 , 0 , 3]
        L2 : [ 3 , 2 , 0 , 0 , 2 , 1 , 2]
              ]#;
externalArc = [ 4 , 0 , 3 , 2 , 4 , 3 , 0];
```

A sparse version of this model would free up memory used to hold redundant zero values. These arrays can be converted to sparse arrays that only include the truly relevant (non-zero) arcs.

Exercise folder

<training_dir>\\COS125.labs\Pasta\Sparsity\work

Create the "internalLinkData" and “externalLinkData" tuples

1. Import the **SparseDeliveryWork** project, located in the **<training_dir>\\COS125.labs\Pasta\Sparsity\work** directory, into the OPL Projects Navigator.
2. Declare a new tuple named "internalLinkData" containing the following data:
   - **string location;**
   - **string warehouse;**
   - **float cost;**
3. Declare a new tuple named "externalLinkData" containing the following data:
   - **string warehouse;**
   - **float cost;**
4. Declare the arc decision variables (**internalArc** and **externalArc**), making sure that any instance includes the **location**,**warehouse**, or both, structures from the **internalLinkData** and **externalLinkData** tuples.

   Hint: Use the **with** keyword.

Revise the objective function

The objective function now needs to be changed so that it works with the new data structures.

- The first set of expressions in the objective function do not need modification, because they refer to structures that have not changed:
  ```
  sum(<p,l,ic> in insideCost) ic * insideProduction[p][l]
       + sum(<p,oc> in outsideCost)(oc * outsideProduction[p])
  ```
- The next expression depends, in the existing model, on the array structures **internalArc** and **externalArc** as they were originally declared, with zero placeholders where no arc is authorized. You need to change this to take into account the sparse structure you have just declared.

  Hint: The cost summation should now be over the arc sets, as opposed to over the locations and warehouses.

Clean up the constraints

In the current model, the existing constraints **resourceAvailability** and **demandFulfillment** remain unchanged. Modify the other constraints as follows

- Rewrite the constraints **insideBalance** and **outsideBalance** to take the new definition of decision variables **insideFlow** and **outsideFlow** into account.
- The test for zero values in the arcs (two unnamed constraints) is no longer necessary, because the sparse structure you created only initializes valid arcs. These constraints are now handled intrinsically in the data structure itself. Remove them.

Initialize the arcs

Because the arcs are now declared with only their significant elements, they must be initialized in a different manner. By initializing only non-zero arcs, only existing arcs are taken into account by the model, saving memory and calculation time.

- The structures **internalArc** and **externalArc** are declared, by using the **with** keyword, as instances of the **internalLinkData** and **externalLinkData** tuples which contain non-zero values for **cost** . It is unnecessary to test for **cost**, because zero cost means that the arc does not exist. In the data file, initialize these structures with sets of data that define an arc with these three values. For example,

```
internalArc =  {<"L1", "W1", 2>, <"L1", "W3", 2>,...etc.
```

- Continue to declare all the arcs. Use the data table in the **Product delivery** panel.

  Note: This type of data structure takes advantage of both sparse data structures (using tuples) and slicing.

Test the model

1. In the IDE, use your current project, or if you are having difficulty you can start with the solution located at **<training_dir>\\COS125.labs\Pasta\Sparsity\solution\SparseDeliverySolution**.
2. Also import the project located in **<training_dir>\\COS125.labs\Pasta\Network\solution\deliverySolution**.
3. Run the **deliverySolution** project. Examine the following items in the **Output** window:
   - In the **Statistics** tab, note all the information, especially the number of non-zero coefficients and constraints.
   - In the **Profiler** tab, note the total time.
4. Run the **SparseDeliverySolution** project.
5. Note the same information as for the **deliverySolution** project.

The sparse model reduces the number of non-zero coefficients from 168 to 105, thanks to the sparse arrays and slicing. The number of constraints is reduced from 55 to 34. The total solution time is determined in part by your machine, but you may see a difference that can become significant in a very large model with many zero values.

Grow the model

Objective

- Observe how a very large model benefits from sparsity and slicing techniques

Actions

- Examine and run the enlarged model
- Examine and run the sparse version of the model

Documentation references

Language > Language User's Manual > Introduction to OPL > Modeling tips > Sparsity

Problem description

Up to this point, the network model has remained small. In a real business situation, however, the network can quickly become very large, and performance issues can arise when modeling. This lab presents a much larger version of the original network problem (6 production locations and 40 000 warehouses), and shows how using sparse data in the model saves time and memory.

Exercise folder

`<training_dir>\\COS125.labs\Pasta\Bigger`

Examine and run the enlarged model

- Import the project **`bigger`**, located at **`<training_dir>\\COS125.labs\Pasta\Bigger`**, expand all the plus signs in the OPL Projects Navigator, and double click **`bigger.dat`** to load it into the editing area.
- Scroll down the data file until you come to the entries for **`internalArc`** and **`externalArc`**.

  You can see that the model has been artificially enlarged for this workshop to include 6 production locations 40 000 warehouses. Normally, large amounts of data like this will be imported from a database or spreadsheet. This subject is treated in another workshop.
- Run the model. When it finishes, examine the information in the different output tabs. Pay particular attention to the **Profiler** tab.
- Use the **Problem Browser** window to visualize the properties of all the items in the model by category, and to visualize the results more clearly than is shown in the **Solutions** output tab.
- Now examine the model file. Note that, even though it uses the same type of tuple structure as the **`sparseDelivery.mod`** file does, the model runs very slowly. Study the model to see if you know why.

Examine and run the sparse version of the model

- Import the sparse version of the model, namely the**`sparseBigger`** project. Do not close the **`bigger`** project.
- Run the model and compare the statistics in the output tabs, especially the **Profiler** tab, with the results from running **`bigger`**. This version of the model runs much faster.
- Examine the model and data files for differences compared to **`bigger`**.
- Use the **Problem Browser** window to visualize the properties of all the items in the model by category, and to visualize the results more clearly than is shown in the **Solutions** output tab.

Chapter 4. Supermarket Display

Problem Description

In this exercise, you practice modeling and solving a real-life optimization problem in IBM ILOG CPLEX Optimization Studio. A supermarket needs to optimize the display of products on shelves in order to maximize profit. The products can be displayed on different storage shelves at various points in the supermarket. The shelves are limited in volume, and some are refrigerated, others not. Shelved that are not refrigerated are referred to as dry shelves.

The different shelf units have different values for promoting the sales of items, based on the floor plan of the market.

Problem data

Products:
- Pasta
- Tomato Sauce
- Salami
- Rice
- Soya Sauce
- Chicken Wings

Each type of product is characterized by the following attributes:
- The expected unit profit
- The unit volume
- Must be refrigerated or not
- The minimum available quantity to be sold
- The maximum quantity that may be ordered (and hence sold)

The objective is to maximize sales by displaying the products in the right places and the right quantities.

Characteristics of the available shelving:
- A volume capacity
- Is refrigerated? (Yes/No)
- A sales acceleration/efficiency factor (for example, higher for placement at the end of an aisle)

Exercise folder

`<training_dir>\\COS125.labs\Mart\work\martWork`

The text file at `C:\\COSTraining\CodeSnippets\SupermarketDisplay.txt` contains some code snippets that you may cut and paste to complete this exercise. Alternatively, you can refer to the solutions located at `<training_dir>\\COS125.labs\Mart\solution`

Model the input data

Objective

- Use different types of set instantiation to model the input data of this problem

Actions

- Define the products and their attributes
- Define product supply
- Define the shelving and its attributes

Documentation references

Language > Language Reference Manual > OPL, the modeling language > Data types > Data structures > Sets

Language > Language Reference Manual > OPL, the modeling language > Data types > Data structures > Tuples

Language > Language Reference Manual > OPL, the modeling language > Data sources > Data initialization > Initializing sets

Language > Language Reference Manual > OPL, the modeling language > Data sources > Data initialization > Initializing tuples

Define the products and their attributes

In this exercise, for simplicity, the .dat file is used to instantiate the data. You do not need to edit the .dat file, because it has already been completed.

When integrating the optimization engine into a legacy system, the input data may already be available in a format CPLEX Studio can connect to, such as an ascii file, a .csv file, an Excel spreadsheet, a data base, or memory resident objects.

Table 2. **Product attribute matrix**

Product name	Profit margin	Unit volume	Needs refrigeration?
Pasta	1	1	No
Tomato Sauce	1.1	1	No
Salami	1.5	1	Yes
Rice	5	1	No
Soya Sauce	5	1	No
Chicken Wing	1	1	Yes

1. Import the **martWork** project into the IDE, and double-click the .mod file in the OPL Projects Navigator.
2. Model the Product data structure in the .mod file with a tuple:

```
tuple Product
{
 key string name; // Product name
 float margin; // Profit margin
 float unitVolume; // Volume for one unit of product
 int cold; // 1 if refrigeration required , 0 otherwise
};
```

Note: Note that the "Needs refrigeration?" column (yes/no or true/false values) of the matrix is represented as a binary as follows:

- 1 if refrigeration is required
- 0 if no refrigeration is required (for "dry" products)

3. Collect all the members of this tuple together into a set:

```
{Product} products = ...;
```

Define product supply

Table 3. Product supply matrix

Product name	Minimum units available	Maximum possible order
Pasta	100	1000
Tomato Sauce	100	1000
Salami	50	300
Rice	100	1000
Soya Sauce	100	1000
Chicken Wing	40	1000

1. Model the supply data structure with the following tuple:

```
tuple Supply
{
  key string product;
 int minimumStockValue;    // minimim quantity to be ordered for stock/sale

 int maximumStockValue;    // potential maximum extra ordered quantity
}
```

2. Collect all the members of this tuple together into a set:

```
{Supply} supplies = ...;
```

Define the shelving and its attributes

Table 4. Shelving attribute matrix

Name	Maximum Capacity	Sales Accelerator Factor	Refrigerated?
SuperPromoFridge	100	1.5	Yes
StandardShelf	1000	0.5	No
PromoFridge	100	1	Yes
PromoShelf	1000	1	No
StandardFridge	100	0.5	Yes
SuperPromoShelf	1000	1.5	No

1. Model the shelf data structure with the following tuple:

```
tuple Shelf
 {
  key string name; // shelf name
  int volumeCapacity; //maximum capacity
  float  promotionAccelerator; // indicator between 0 and 1.5
  int cold; // 1 for refrigerated, 0 otherwise
 }
```

2. Collect all the members of this tuple together into a set

```
{Shelf} shelves = ...;
```

Model the decision variables

Objective

- Use sparse sets to model decision variables

Actions

- Define compatibilities
- Declare the decision variables

Documentation references

Language > Language Quick Reference > OPL keywords > Union

Language > Language Reference Manual > OPL, the modeling language > Decision types > Expressions of decision variables

Description

To model the decision variables for this problem, consider the following points:

- The unknowns for this problem are:
  - How much of each product to display on each shelf
  - The total quantity of each product to order (the sum over all shelves)
- It is possible to define a decision variable for all products on all shelves – is this a good idea?
- You will define a tuple to create compatible product-shelf pairs, and use this tuple to index an array of decision variables.

Note: It is not a good idea to define decision variables for all products on all shelves, because not all products can be displayed on all shelves. Specifically, only certain products should be displayed on refrigerated shelves. Product-shelf compatibility is determined by whether a product requires refrigeration or not.

Define compatibilities

You should define a decision variable to determine the quantity of each product displayed on each shelf. This should be an array indexed on compatible pairs of products and shelves:

- Cold products can only be displayed on cold (refrigerated) shelves
- Dry products can only be displayed on dry (non-refrigerated) shelves

1. To define a **sparse** set of compatible pairs, declare a tuple
   ```
   tuple ProductShelfCompatibility
   {
    Product product;
    Shelf shelf;
   }
   ```
2. Create two computed sets of instances of **`ProductShelfCompatibility`**:
   - One containing only cold product/shelf pairs named **`coldCompatibilities`**
   - One containing only dry product/shelf pairs named **`dryCompatibilities`**
3. Declare a third set, **`compatibilities`**, that is the aggregate of the two sets you declared in the last step.

   Tip: Use the **`union`** operator.

Next you will use these pairs as indexes for the decision variables. This is a simple way to exploit a sparse matrix.

Declare the decision variables

1. Use the set **`compatibilities`** as an index for an array of decision variables for displayed product quantities over shelves. Call it **`storedQuantities`**:
   ```
   dvar float storedQuantities[compatibilities] in 0..maxint;
   ```
2. Define a decision expression called **`orderedQuantities`** that uses the set **`compatibilities`** to sum up only the quantities on shelves compatible with each product:
   ```
   dexpr float orderedQuantities[ p in products] =   // complete using array initialization
   ```

Write the objective function and constraints

Objectives

- Use sets and decision expressions to define an objective
- Use generic set initialization to define constraints

Actions

- Write the objective function
- Write the constraints

Write the objective function

The objective is to maximize profit by displaying the products in the right places and the right quantities.

In the **`.mod`** file, write an objective function to maximize profit:

1. For each product/shelf compatibility pair, define the potential profit as a function of:
   - Displayed quantity for the shelf (**`storedQuantities`**)
   - Product's profit margin (**`margin`**)
   - The shelf's sales accelerator factor (**`promotionAccelerator`**)

   Tip: Use a decision expression, using the keyword **`dexpr`**. Name it **`profit`**
2. Add the objective to maximize **`profit`**.

Write the constraints

The constraints are:

- A minimum quantity of each product must be displayed.
- The total quantity ordered for each product must not exceed the maximum allowed.
- The total shelf capacity cannot be exceeded.

1. The first constraint specifies that for all products, the sum of the ordered quantities should be greater than or equal to a minimum value:
   ```
   forall( p in products, s in supplies: s.product == p.name){
     MinStockCst: orderedQuantities[p] >= s.minimumStockValue;
   }
   ```
2. The second constraint specifies that for all products, the sum of the ordered quantities should not exceed a maximum value:

```
forall( p in products, s in supplies: s.product == p.name){
  MaxStockCst: orderedQuantities[p] <= s.maximumStockValue;
}
```

3. Write a constraint requiring that for all shelves, the total quantity of all the displayed products multiplied by the product unit volume shall not exceed the total shelf volume capacity.

   Tip: Use the set of **`compatibilities`**.

Two of the constraints are already completed for you.

Display the results

Objective

- Use sets to post-process data for display

Actions

- Create output data structures
- Convert structures into formatted results for display
- Experiment with the values

Documentation references

Language > Language Reference Manual > OPL, the modeling language > Data sources > Data initialization > Initializing sets

Create output data structures

1. In the model, the tuples to be used for the output are already declared:
   - **`StorageResult`**: The quantity of each product allocated to each shelf.
   - **`PurchaseOrderResult`**: The total quantity of each product.
   - **`ShelfUsage`**: The percentage of shelf space in use for each shelf.

     Note: The **`usagePercentage`** member of this tuple is to be calculated during post-processing as part of the set of shelf usage ratios (see the next substep).

   Examine these structures in the model file.

Convert structures into formatted results for display

To display the results in a useful manner, you can create additional data structures for post-processing. Such data structures have already been created for this exercises, namely:

- **`StorageResult`**: The quantity of each product allocated to each shelf.
- **`PurchaseOrderResult`**: The total quantity of each product to display.
- **`ShelfUsage`**: The percentage of shelf space in use for each shelf.

In this step you should use generic set initialization to populate sets of these data structures. See the online help if you need a reminder on how to do this.

1. Use generic set initialization to populate the following sets:
   a. **`POResults`**: Use the product **`name`** and **`orderedQuantites`**.
   b. **`shelfUsages`**: For each shelf, calculate the **`usagePercentage`** by dividing the total quantity stored in the shelf (sum of **`storedQuantities`** for all compatible products) by the shelf capacity.

   c. `storageResults`: Use the product `name`, the shelf `name`, and `storedQuantities`.
2. Compare your work with the solution in `<training_dir>\\COS125.labs\Mart\solution\martSolution`
3. Examine the IBM ILOG Script `execute` block used to write this data to the **Scripting log** output tab. This data could also be exported to an external application such as a database or spreadsheet, for further processing and analysis.

Experiment with the data

In principle, if shelves are not full, the model will change the number of products to be ordered so that all shelf space is used 100%. However, there may be limits of budget or product availability with the result that the maximum permitted order of some products would be exceeded.

Try reducing the value of `maximumStockValue` for some or all products, and see what results you get when solving the model. Compare several different sets of values to see the effect of the changed data.

Examine the displayed results in the **Scripting log** output tab, and the values of data elements and decision variables in the **Problem browser**.

Chapter 5. Steel mill inventory matching

Workshop overview

In this exercise, you will practice using CP Optimizer to solve a steel mill inventory matching problem. The exercise starts with a description of the problem the production manager faces. Then you'll get a chance to derive the CP model from this description and model the problem in OPL. You'll also get a chance to experiment with search phases and alternative formulations to improve the solution speed.

Problem description

The steel mill has an inventory of steel slabs, of a finite number of different capacities (sizes), that are used to manufacture different types of steel coil. During production, some of the steel from the slabs is lost. The production manager has to decide which steel slabs to match with which coil orders in order to minimize the total loss. In optimization terms, the problem can be described as follows:

- **Objective**
  - Minimize the total loss
- **Decision variables**
  - Which slab should be matched with which coil order
  - The capacity of each slab used
- **Constraints**
  - A coil order can be built from at most one slab, although each slab can be used to fill several coil orders.
  - Each type of steel coil requires a specific production process, with each such process encoded by a **color** designated by a number between 1 and 88. A slab can be used for at most two different coil production processes or colors.
  - Each steel coil order has an associated weight, and the total weight of all coil orders matched with a slab must be less than the capacity of that slab.
  - The amount of loss from each slab equals the capacity of the slab minus the total weight of all coil orders assigned to that slab.
  - The total loss is the sum of losses from all slabs.
  - An unlimited quantity of steel slabs of each capacity is available.

Problem data

There are 111 coil orders and 21 different slab sizes, namely 12, 14, 17, 18, 19, 20, 23, 24, 25, 26, 27, 28, 29, 30, 32, 35, 39, 42, 43, and 44. The table below gives the **weight** and **color** data for 5 coil orders. The complete data set can be seen in the data file, which can be found in the exercise folder referred to below.

Table 5. Problem data

Coil order	Weight	Color
1	30	73
2	30	74
3	30	75

Table 5. Problem data (continued)

Coil order	Weight	Color
110	2	6
111	2	4

Even though an unlimited quantity of steel slabs is available, you can use an upper bound of 111 on the total number of steel slabs because of the obvious solution of using one slab per coil order.

Exercise folder

`<training_dir>\\COS125.labs\SteelMill\work`

The solution to this exercise is available in `<training_dir>\\COS125.labs\SteelMill\solution`

Solve the problem using CP Optimizer and OPL

Actions

- Declare the data
- Declare the decision variables
- Define the objective function
- Define the constraints
- Solve the model

Documentation references

Language > Language User's Manual > Introduction to OPL > Language overview > Constraint programming

Declare the data

1. In the IDE, import the project by selecting **File > Import > Existing Projects into Workspace**, and choosing the **SteelMill_work** project from the exercise folder **`<training_dir>\\COS125.labs\Steelmill\work`**. Leave the **Copy projects into workspace** box unchecked.
2. Expand the project to see the contents. For this step, you'll only work with the following:
   - Model file: **`steelmill_work.mod`**
   - Settings file: **`steelmill_work.ops`**
   - Data file: **`steelmill_work.dat`**
   - Run configuration: **`naive model (default)`**
3. Open the model file, **`steelmill_work.mod`**, and see that the following data has been declared:
   - The number of coil orders: **`int nbOrders = ...;`**
   - The weight of each coil order: **`int weight[1..nbOrders] = ...;`**

   Now use similar syntax to declare the following:
   - The integer number of steel slabs available: **`nbSlabs`**
   - The integer number of colors: **`nbColors`**
   - The integer number of distinct slab capacities: **`nbCap`**

- An integer array for the actual capacities associated with each index in **nbCap**: **capacities**
- An integer array for the colors associated with each coil order: **colors**
4. The remaining completed items in the data declaration part of the model file are used later in the model, and are as follows:
   - **maxCap**: This is the greatest available capacity and will be used to define the domain of the decision variables for slab capacity.
   - **caps**: This is the set of available capacities (with the same content as the array **capacities**) and is used in a later substep to define the capacity constraints for each slab.
5. Open the data file, **steelmill_dat.dat**, to see how the data is instantiated.

Declare the decision variables

1. In the model file, **steelmill_work.mod**, specify that you're using CP by adding the line **using CP;** at the top.
2. In the section titled **/* Decision variables */**, the two decision variables have already been declared as follows:
   - **dvar int where[1..nbOrders] in 1..nbSlabs**: This variable is used to determine which slab each coil order is assigned to, and is therefore indexed over all orders with domain of all slab numbers.
   - **dvar int capacity[1..nbSlabs] in 0..maxCap**: This variable is used to determine the capacity of each slab, and is therefore indexed over all slab numbers, with domain of all values between 0 and the maximum available capacity.
3. Declare an integer decision expression called **load** indexed over the slabs. Assign the value of **load** to equal the sum of the weights of all coil orders assigned to a slab.

   Tip: First write the expression to sum the weights of all coil orders. Next, use the **where** variable, together with the logical equals (**==**), to write an expression that evaluates to 1 if an order is assigned to a slab and 0 otherwise. Multiply this expression with the **weight** to add only the weight of orders assigned to the slab.
4. Declare another integer decision expression called **colorAssigned** indexed over the colors and the slabs. Complete the declaration with an expression to assign the value of **colorAssigned** to be 1 if any coil orders assigned to a given slab have a particular color and 0 otherwise.

   Tip: First use the **where** variable and the logical equals (**==**) that you used for the **load** expression to determine whether an order is assigned to a slab (1) or not (0). Next, in the same expression, use the logical OR statement (**or**) to create an "or" over all assigned orders with the particular color. Be sure to check the syntax for the logical OR statement in the online documentation at **Language Quick Reference > OPL keywords**.

Define the objective function

1. In the section titled **Objective function**, write the objective to minimize total loss. The total loss is defined as the sum, over all slabs, of the slab capacity minus the slab load.

Define the constraints

1. In the section titled **Constraints** and within the **subject to** block, there is a **forall** constraint declaration that uses the **caps** set to restrict the domain of the

capacity variable. Because the **capacity** variable has a domain enumerated from a set, its domain cannot be defined during data declaration as follows:

```
dvar int capacity[1..nbSlabs] in caps; // NOT ALLOWED
```

Instead, a continuous domain has to be defined at declaration, and the domain then has to be restricted in the constraint block.

2. Add a constraint within this same **forall** block to state that the slab load must be less than or equal to the slab capacity.
3. Add a constraint called **colorCt** for each slab that states that the number of colors assigned to that slab must be less than or equal to 2.

   Tip: Use the **colorAssigned** decision expression.
4. Your model is complete at this point. If you have any remaining errors, check the solution or check with your instructor before attempting to solve the model.

Solve the model

1. In the **OPL Projects Navigator**, expand the **Run Configurations** for your project. right click **naive model (default)** and select **Run this** from the context menu.
2. Select the **Engine log** output tab to see the solution progress.
3. Let the model run for about a minute and then click the red stop button (you can see the solution time scrolling by periodically on the left side of the log).
4. Scroll to the start of the log and notice that the first solution found (under the **Best** column), is around 1000. Scroll to the end of the log and notice that the best solution found after about a minute is around 30. In the next step you'll get a chance to implement a search phase to improve performance.

Implement a search phase

Actions

- Add a search phase
- Solve the model

Documentation references

Language > Language User's Manual > IBM ILOG Script for OPL > Using IBM ILOG Script in constraint programming > Defining search phases > What is a search phase?

Add a search phase

In this exercise you'll write a search phase to guide CP Optimizer in the selection of decision variables during the solution search. For the steel mill problem, an intuitive search strategy is to first assign orders to slabs (first search on the **where** decision variable), and afterwards assign capacities to each slab (next search on the **capacity** decision variable).

If you are confident that your model is correct, you can continue working with it. Otherwise, you can continue with the **searchPhase_work.mod** file. These instructions assume that you are using the latter file, which contains the model up to this point, together with some instructions on how to do implement the search phase.

1. Scroll down to the section titled **Search phase**. The search phase is written within the **execute** statement.
2. Define the script variable **f** to access the CP search modifier factory.
3. Define a search phase, **phase1**, on the **where** variable using the **searchPhase** method.
4. Define another search phase, **phase2**, on the capacity variable.
5. Use the **setSearchPhase** method to set the search to first use **phase1**, and next **phase2**.
6. If you have any errors, check the solution or check with your instructor before attempting to solve the model.

Solve the model

1. In the **OPL Projects Navigator**, expand the **Run Configurations** for your project. right click **search phase** and select **Run this** from the context menu.
2. Select the **Engine log** output tab to see the solution progress.
3. Let the model run for about a minute and then click the red stop button (you can see the solution time scrolling by periodically on the left side of the log).
4. Scroll to the start of the log and notice that the first solution found (under the **Best** column), is around 20 – much better than the first best solution of 1000 without the search phase. Scroll to the end of the log and notice that the best solution found after about a minute is around 8, again an improvement compared to the solution of 30 without a search phase. In the next step you'll get a chance to try and improve performance by changing the OPL model.

Improve the model

Actions

- Improve the model
- Solve the model

Improve the model

The key to understanding the model improvements in this exercise, is realizing that once the load on a slab is known, its capacity becomes a trivial decision. Specifically, if the load is known, the capacity of that slab will simply be the smallest available capacity just bigger than the load on the slab. In this exercise, you'll take advantage of this knowledge to improve the model by removing the **capacity** decision variable.

1. In the same work project in the IDE, open the **betterModel_work.mod** file.
2. In the **Data declaration** section of the model file, notice that a new line has been added:

   ```
   int loss[c in 0..maxCap] = min(i in 1..nbCap : capacities[i] >= c) capacities[i] - c;
   ```

   This line takes advantage of the fact that the load (and loss) is an integer and there are therefore a finite number of possible values the load on a slab can take, namely all integer values between **0** and **maxCap**. For each such value, the array above defines the loss to be the smallest capacity just larger than the load (defined by using the **min** function), minus the load. The array uses the index **c** and you'll see next how it can be used with the load values instead.
3. In the **Decision variables** section, remove the **capacity** decision variable. All the other variables remain the same.
4. Change your objective function to minimize the **loss** directly instead of using the **capacity** and **load** decision variables.

Tip: Index the **loss** array with the **load** variable.

Note: If you're familiar with MP, you'll notice here one of the major differences between MP modeling and CP modeling: In CP a decision variable can be used to index an array, as is shown here where the **load** variable is used as an index for the **loss** array.

5. In the **Constraints** section, remove the constraints that use the **capacity** decision variable, because they are no longer required when this variable doesn't exist.
6. Finally, in the **Search phase** section, remove **phase2** because you no longer have the **capacity** variable.
7. If you have any errors, check the solution or check with your instructor before attempting to solve the model.

Solve the model

1. In the **OPL Projects Navigator**, expand the **Run Configurations** for your project. right click **Better Model** and select **Run this** from the context mneu.
2. Select the **Engine log** output tab to see the solution progress.
3. See that CP Optimizer finds the optimal match between orders and slabs, resulting in zero loss, in about 0.1 seconds.

   Note: The steel mill problem is a benchmark problem used to benchmark optimization engine performance, and it is worth noting that CP Optimizer is the first constraint programming engine to be able to find an optimal solution to this problem.

Use the pack constraint

Action

- Use the **pack** constraint

Documentation references

Language > Language Quick Reference > OPL functions > pack

Use the pack constraint

This part of the exercise is optional and shows you how to use one of CP Optimizer's more advanced constructs, namely the **pack** constraint. This constraint is generally used to assign items into packs of finite capacity. In this sense, the orders are the items, and the slabs are the packs of finite capacity to which the items are assigned (see the **OPL Help** for further information on the **pack** constraint).

1. In the same project you've been working in, open the **polished_work.mod** file.
2. In the **Data declaration** section, see that a new property has been declared, namely **maxLoad**. This is the sum of the weights of all coil orders.
3. In the **Decision variables** section, see that **load** is no longer a decision expression, but is now an integer decision variable with domain between 0 and **maxLoad**. The reason for this change is that the **pack** constraint does not accept a decision expression as an argument and instead requires a decision variable.
4. In the **Constraints** section, look at the definition of the **pack** constraint and try to understand it by comparing it with the explanation in **OPL Help**.

5. Run the model from the **Polished Model Run Configuration** and see that it also finds the optimal solution of zero loss in around 0.1 seconds.

Chapter 6. Staff Scheduling

Problem Description

In this workshop you will model a staff scheduling problem using IBM ILOG CPLEX 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`
- `ISDNInstall`

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`

Overview of steps

Before you begin to work with the lab files, here is an overview of what you are going to do. Each task will be decomposed step by step to demonstrate the process.

1. The task type **FlipSwitch** requires a Technician, and there are two Technicians. For each task of this type, you create three **interval** decision variables. Two of these intervals are optional, meaning they may or may not appear in the solution.
2. To constrain such that when an optional interval is present, it is scheduled at exactly the same time as the task interval, use a **synchronize** constraint.
3. To ensure that the appropriate number of worker intervals are used, write a constraint that requires that the sum of that the sum of present worker intervals is equal to the number of resources required.

See the substeps that follow for more detail on how to complete this exercise.

Note: After solving each step, look at the Gantt Chart solution representation for the **interval** variables to examine the solution and to get a graphical idea of what may possibly still be missing. To see the Gantt Chart, go to the **Problem Browser** and in the **Decision Variables** section hover you mouse next to the name of the **interval** variable to make the **Show data view...** icon visible. The default view when selecting this icon is a tabular view, and you can instead choose the Gantt Chart tab at the bottom of the view to see the Gantt Chart representation of the same data.

Exercise folder

<training_dir>\\COS125.labs\Scheduling\Staff\work\sched_staffWork

The text file at **C:\\COSTraining\CodeSnippets\StaffScheduling.txt** contains some code snippets that you may cut and paste to complete this exercise. Alternatively, you can refer to the solutions located at **<training_dir>\\COS125.labs\Scheduling\Staff\solution**

Declare task interval and precedences

Objectives

- 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

Documentation references

Language > Language Quick Reference > OPL keywords > interval

Language > Language Quick Reference > OPL functions > 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: Note that there is no objective function. At this point, this is 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.

   Note: 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`**

Define the tasks and precedences

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

Details of the intervals for FlipSwitch

- One optional represents **`JohnTec`** being assigned to the task
- The other optional represents **`PierreT`** being assigned to the task.
- The third interval represents the task itself, and is used in other constraints.

Note: In general, if there are multiple resources with identical properties, it is best to model them as a **resource pool**. However, one can imagine that in this example new constraints related to workers' days-off could be added, so here each worker is treated as an individual resource.

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. To model that some tasks in a request must occur before other tasks in the same request, use the precedence constraint **`endBeforeStart`**. While the data for this problem does not require there to be any delay between tasks, you can add a delay to the model to allow for the possibility of a delay.

   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 to understand clearly what they mean — ask your instructor if you need help.

3. Write the **endBeforeStart** constraint.
4. Compare your work with the solution in **<training_dir>\\COS125.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 and compare these with the Gantt Chart representation of the **titasks** variables. 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**.

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

Documentation references

Language > Language Quick Reference > OPL functions > span

Language > Language Quick Reference > OPL keywords > all

Language > Language Quick Reference > OPL functions > 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 .

   Tip: Use the **all** quantifier to associate the required tasks for each request with the appropriate duration.
4. Check your solution against **<training_dir>\\COS125.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 model the objective you need to determine the end time of each request. The request itself can be seen as an interval with a variable length. The request interval must cover, or span, all the intervals associated with the tasks that comprise the request.

You create the objective by finding the difference between the end time and the due date and minimizing it.

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.

     Tip: 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.

   Tip: 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 **<training_dir>\\COS125.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**.
4. Look at the Gantt Chart representations of the two interval variables and verify that the **tirequests** variables span the relevant **titasks** variables, as required by the **span** constraint.

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

Documentation references

Language > Language Quick Reference > OPL functions > synchronize

Language > Language Quick Reference > OPL keywords > optional

Language > Language Quick Reference > OPL keywords > sequence

Review the needs

So far, you have defined the following constraints:

- 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. You know from the data that there is more than one resource, in some cases, capable of doing a given task. For example, the task type **FlipSwitch** requires a technician. There are two technicians, that is, two **alternative resources**, available to do the same task. The model should optimize how each of these is used relative to the objective. How do you decide who is the best one to send on a particular job?

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: 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.

   Tip: 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 **operation**s.

"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 **`<training_dir>\\COS125.labs\Scheduling\Staff\solution\sched_staffSolution\Step3.mod`**

   Important: 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. Also look at the results displayed in the Gantt Charts for the **`interval`** variables.
5. Close **`Step3.mod`**.

Add a surrogate constraint to accelerate search

Actions

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

Documentation references

Language > Language Quick Reference > OPL keywords > 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. 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 **<training_dir>\\COS125.labs\Scheduling\Staff\solution\sched_staffSolution\Step4.mod**

Solve the model and examine the results

1. Solve the model by right clicking the **Step4** run configuration and selecting **Run this** from the context menu.
2. Look at the **Engine log** and **Statistics** output tabs to see a dramatic improvement in the number of fails, thanks to the surrogate constraint.

Chapter 7. House Building Calendar

Problem Description

In this workshop you will model a problem, in IBM ILOG CPLEX Optimization Studio, of scheduling the tasks involved in building multiple houses in such a manner that minimizes the overall completion date of the houses.

Principles of the problem:

- There are five houses to be built.
- There are two workers:
  - Joe
  - Jim
- Each worker can only be assigned to a subset of the total list of tasks (different skills).
- Some tasks must take place before other tasks, and each task has a predefined size.
- Each worker has a calendar detailing the days on which he does not work:
  - Weekends
  - Public holidays
  - Other days off
- On a worker's day off, he does no work on his tasks.
- A worker's tasks may not be scheduled to start or end on a day off.
- Tasks that are in process by the worker are suspended during his days off (for example a task may begin on Friday and finish on Monday with a weekend off in between).

Problem data

Here is the list of house construction tasks, giving the length of each task, the worker assigned to it, and the precedence:

Table 6. House construction tasks

Task	**Size**	**Worker**	**Preceding tasks**
Masonry	35	Joe	—
Carpentry	15	Joe	Masonry
Plumbing	40	Jim	Masonry
Ceiling	15	Jim	Masonry
Roofing	5	Joe	Carpentry
Painting	10	Jim	Ceiling
Windows	5	Jim	Roofing
Facade	10	Joe	Roofing, Plumbing
Garden	5	Joe	Roofing, Plumbing
Moving	5	Jim	Windows, Facade, Garden, Painting

Note: After each solve in this exercise, look at the Gantt Chart solution representation for the **interval** variables to examine the solution and to get a graphical idea of what may still be missing. To see the Gantt Chart, go to the **Problem Browser** and in the **Decision Variables** section hover you mouse next to the name of the **interval** variable to make the **Show data view...** icon visible. The default view when selecting this icon is a tabular view, and you can choose the Gantt Chart tab at the bottom of the view to see the Gantt Chart representation of the same data.

Exercise folder

`<training_dir>\\COS125.labs\Scheduling\Calendar\work\sched_calendarWork`

The text file at **`C:\\COSTraining\CodeSnippets\HouseBuildingCalendar.txt`** contains some code snippets that you may cut and paste to complete this exercise. Alternatively, you can refer to the solutions located at **`<training_dir>\\ COS125.labs\Scheduling\Calendar\solution`**

Declare data and decision variables

Actions

- Define a calendar for each worker
- Declare the decision variable

Documentation references

Language > Language Quick Reference > OPL keywords > intensity

Define a calendar for each worker

1. Import the **`sched_calendarWork`** project into the OPL Projects Navigator (leave the **Copy projects into workspace** box unchecked) and open the **`calendar.mod`** and **`calendar.dat`** files for editing.
2. Examine the first data declarations and their instantiations in the **`.dat`** file:
   - The first two declarations, **`NbHouses`** and **`range Houses`** establish simple declarations of how many houses to build, and a range that is constrained between 1 and that total number.
   - The next two declarations instantiate sets of strings that represent, respectively, the names of the workers and the names of the tasks to perform.
   - The declaration **`int Duration [t in TaskNames] = ...;`** instantiates an array named **`Duration`** indexed over each **`TaskNames`** instance.
   - The declaration **`string Worker [t in TaskNames] = ...;`** instantiates an array named **`Worker`** indexed over each **`TaskNames`** instance.
   - The tuple set **`Precedences`** instantiates task pairings in the tuple **`Precedence`**, where each tuple instance indicates the temporal relationship between two tasks: the task in **`before`** must be completed before the task in **`after`** can begin.
   - The tuple **`Break`** indicates the start date, **`s`**, and end date, **`e`** of a given break period. A list of breaks for each worker is instantiated as the array **`Breaks`**. Each instance of this array is included in a set named **`Break`**.
3. Declare a tuple named **`Step`** with two elements:
   - An integer value, **`v`**, that represents the worker's availability at a given moment (0 for on a break, 100 for fully available to work, 50 for a half-day)

- An integer value, **x**, that represents the date at which the availability changes to this value. Make this element the key for the tuple.

```
tuple Step {
int v;
key int x;
};
```

4. Create a sorted tuple set such that at each point in time where the worker's availability changes, an instance of the tuple set is created. Sort the tuple set by date, and use a **stepfuncion** named **calendar** to create the intensity values to be assigned to each **WorkerName**

   Tip: Use a **stepwise** function:

   ```
   sorted {Step} Steps[w in WorkerNames] =
   { <100, b.s >| b in Breaks[w] } union
   { <0, b.e >| b in Breaks[w] };
   stepFunction Calendar[w in WorkerNames] =
   stepwise (s in Steps[w]) { s.v - >s.x; 100 };
   ```

   Note: When two consecutive steps of the function have the same value, these steps are merged so that the function is always represented with the minimal number of steps.

Declare the decision variable

Once the start dates of the tasks are known, the end dates are also known, because the size of each task and the breaks that determine the length of each task are known. In this step you will see an expression that calculates the start date of each task for each house, using this information.

1. Continue looking at the model file – the following **interval** decision variable is declared:

   ```
   dvar interval itvs[h in Houses, t in TaskNames] size Duration[t] intensity
    Calendar[Worker[t]];
   ```

2. The step function **Calendar** is associated with the interval decision variable by using the keyword **intensity** in order to ensure that the worker's availability is taken into account. Make sure you understand how this is done in the model.

Define the objective function

Actions

- Transform the business objective into the objective function

The business objective requires the model to minimize the total number of days required to build five houses. To do this in the model, you need to write an objective function that minimizes the maximum time needed to build each house and arrive at a minimum final completion date for the overall five-house project.

1. Determine the maximum completion date for each individual house project using the expression **endOf** on the last task in building each house (the **moving** task) and
2. Minimize the maximum of these expressions.

Check the solution in **<training_dir>\\COS125.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod**.

Define constraints

Actions

- Write the precedence constraint
- Write the **noOverlap** constraint
- Write the forbidden start/end period constraint

Documentation references

Language > Language Quick Reference > OPL functions > noOverlap

Language > Language Quick Reference > OPL functions > forbidStart

Language > Language Quick Reference > OPL functions > forbidEnd

Write the precedence constraint

The precedence constraints in this problem are simple **endBeforeStart** constraints with no delay.

1. Write a single constraint that can be applied via the tuple set **Precedences** to each instance of the interval decision variable **itvs**.

   Tip: Use filtering on **(p in Precedences)** to separate out start dates and end dates. Use arrays of the form **[p.before]** and **[p.after]**:
   ```
   forall(p in Precedences)
         endBeforeStart(itvs[h][p.before], itvs[h][p.after]);
   ```
2. Solve the model and check the values of the **itvs** variables in the Gantt Chart view. The Gantt Chart should show the intervals associated with a particular worker overlapping, indicating the need to a **noOverlap** constraint.
3. Check your work against the file **<training_dir>\\COS125.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod**.

Write the noOverlap constraint

1. Write a constraint that says the interval variables associated with a worker are constrained to not overlap in the solution.
2. Solve the model and again check the values of the **itvs** variables in the Gantt Chart view to verify that the intervals for each worker are now longer overlapping. However, it is still possible that a task may start or end on a workers' days off.
3. Check your work against the file **<training_dir>\\COS125.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod**.

Note: You may be surprised by the form of the **noOverlap** constraint in the solution. This form is a shortcut that avoids the need to explicitly define the interval sequence variable when no additional constraints are required on the sequence variable.

Write the forbidden start/end period constrain

1. Write a constraint, using **forbidStart** and **forbidEnd**, that forbids a task to start or end on the associated worker's days off (that is, when **intensity** = 0).
2. Solve the model, and check the Gantt Chart for the **itvs** variables to verify that there are no tasks that start or end during any of the workers' days off.

3. Check your work against the file **<training_dir>\\COS125.labs\Scheduling\Calendar\solution\sched_calendarSolution\calendar.mod**.

Chapter 8. Wood Cutting

Problem Description

In this workshop you will model a problem, in IBM ILOG CPLEX Optimization Studio, of scheduling the tasks involved in a process of cutting different kinds of logs into wood chips, employing state constraints.

Principles of the problem:

- A wood factory machine cuts stands (processed portions of log) into chips.
- Each stand has these characteristics:
  - length
  - diameter
  - species of wood
- The machine can cut a limited number of stands at a time with some restriction on the sum of the diameters that it can accept.
- The truck fleet can handle a limited number of stands at a given time.
- Stands processed simultaneously must all be of the same species.
- Each stand has a fixed delivery date and a processing status of one of::
  - standard
  - rush

  Any delay on a rush stand will cost a penalty

The wood cutting company needs to minimize costs per unit time, and reduce penalty costs resulting from late deliveries of rush stands to a minimum.

Problem data

Here is the list of stands to cut giving the characteristics of each stand, the date delivery is due, and whether the order is a rush order or not.

Table 7. Problem data

Diameter	Species	Length	Due date	Rush?
10	oak	20	991	Yes
10	oak	10	998	No
20	beech	20	210	No
10	pine	30	210	No
30	beech	30	1013	No
30	oak	10	1005	No
10	oak	40	1120	No
10	beech	41	300	Yes
10	beech	42	600	Yes
20	oak	21	1010	No
20	pine	10	918	No

The restrictions on the process are as follows:

- Maximum diameter the machine can process: 60
- Number of trucks in fleet: 10
- Maximum number of stands that can be processed together: 10
- Maximum number of batch periods per day: 10
- Cutting cost per day: 100
- Penalty cost for rush orders delivered late: 100/linear foot
- Species to cut:
  - oak
  - beech
  - pine
- Cutting time for each species:
  - oak: 3
  - beech: 5
  - pine: 7

Note: After each solve in this exercise, look at the Gantt Chart solution representation for the **interval** variables to examine the solution and to get a graphical idea of which constraints may be missing. To see the Gantt Chart, go to the **Problem Browser** and in the **Decision Variables** section hover you mouse next to the name of the **interval** variable to make the **Show data view...** icon visible. The default view when selecting this icon is a tabular view, and you can choose the Gantt Chart tab at the bottom of the view to see the Gantt Chart representation of the same data.

Exercise folder

`<training_dir>\\COS125.labs\Scheduling\Wood\work\sched_woodWork`

Almost all of this model is already done. You are going to examine it, then write a state function and a state constraint to complete it.

Examine the completed parts of the model

Actions

- Import the project
- Modeling the processing of the stands
- Modeling the quantity constraint
- Modeling the diameter constraint
- Modeling the fleet constraint

Import the project

1. Import the **`<training_dir>\\COS125.labs\Scheduling\Wood\work\sched_woodWork`** project into the OPL Projects Navigator. Leave the **Copy projects into workspace** box unchecked.
2. Open the **`sched_wood.mod`** file for editing and examine it.

Modeling the processing of the stands

An interval variable is associated with each of the stands. The size of an interval variable is the product of the length of the stand and the time it takes to cut one unit of the stand's species:

```
dvar interval a[s in stands] size (s.len * cutTime[s.species]);
```

Modeling the quantity constraint

The number of stands being processed at a time can be modeled by a cumulative expression function. Between the start and end of the interval representing the processing of the stand, the cumul function is increased by 1 using the **pulse** function. A constraint that the cumul function never exceeds the stand capacity of the machine is added to the model:

```
cumulFunction standsBeingProcessed = sum (s in stands) pulse(a[s], 1);
```

```
    standsBeingProcessed   <= maxStandsTogether;
```

Modeling the diameter constraint

The total diameter of the stands being processed at a time can be modeled by a cumulative function. Between the start and end of the interval representing the processing of the stand, the cumul function is increased by the diameter using the **pulse** function. A constraint that the cumul function never exceeds the diameter capacity of the machine is added to the model:

```
cumulFunction diameterBeingProcessed = sum (s in stands) pulse(a[s], s.diameter);
```

```
    diameterBeingProcessed <= maxDiameter;
```

Modeling the fleet constraint

The constraint on the number of trucks being used can be placed on the cumul function for the number of stands being processed:

```
cumulFunction trucksBeingUsed = standsBeingProcessed;
```

```
    trucksBeingUsed        <= nbTrucks;
```

Define the one species constraint

Actions

- Declare the state function
- Write an **alwaysEqual** constraint

Documentation references

Language > Language Quick Reference > OPL functions > alwaysEqual

Declare the state function

In this model, the wood cutting company can profit from processing multiple stands at the same time in the same batch, provided that certain constraints are met. One of these is that the cutting machine can only process one species of wood at a time.

- In the model file, declare a state function called **species**.

Write an alwaysEqual constraint

1. Write a constraint that says that the value **species** in each member of the tuple set **stands** is equal when being processed by the cutting machine.

   Tip: Use the **ord** keyword to order the species together, and the scheduling constraint **alwaysEqual** to constrain the state function **species**.

2. Solve the model and check the Gantt Chart for the **a** interval variable.
3. Check your work against the file **<training_dir>\\COS125.labs\Scheduling\Wood\solution\sched_woodSolution\sched_wood.mod**.

Examine the objective function

Actions

- Transform the business objective into the objective function

Transform the business objective into the objective function

The objective requires the model to minimize the sum of two calculations.

The first is the product of the maximum cutting time per stand and the cost per time unit.

- In the model, the maximum cutting time per stand is defined by a decision expression using the **dexpr** OPL keyword:

```
dexpr int makespan =
   max (s in stands) endOf(a[s]);
```

- The first part of the objective function calculates the product of **makespan** and the cost per time unit:

```
minimize makespan * (costPerDay / nbPeriodsPerDay)
```

The second quantity to be minimized is the product of the length of rushed stands that are late and the cost per unit of length for being late.

- To calculate the length (in feet, in this case) of stands identified as "rush" orders that are to be delivered late, use another decision expression, named **lateFeet**:

```
dexpr float lateFeet =
   sum (s in stands : s.rush == 1) s.len * (endOf(a[s])  >s.dueDate);
```

- You can now complete the objective function by adding the calculation of cost of late rushed footage. The entire objective function is:

```
minimize makespan * (costPerDay / nbPeriodsPerDay)
  + costPerLateFoot * lateFeet;
```

Spend some time examining the solution in **<training_dir>\\COS125.labs\Scheduling\Wood\solution\sched_woodSolution\sched_wood.mod**.

Pay special attention to how the state constraint interacts with the objective.

When you're done, solve the model and again look at the Gantt Chart for the **a** variable to see how the change in the objective impacts the schedule.

Chapter 9. Warehouse location

Problem Description

In this exercise you will model a warehouse location problem, using integer programming and Boolean decision variables, in IBM ILOG CPLEX Optimization Studio.

Principles of the problem:

- A company is considering a number of locations for building warehouses to supply its existing stores.
- Each possible warehouse has a fixed cost associated with opening the warehouse, as well as a unique cost associated with assigning a store to a warehouse.
- Each warehouse has a maximum capacity specifying how many stores it can support.
- Each store can be supplied by only one warehouse.
- The decisions to be made are whether to open each warehouse or not, and which stores to assign to each open warehouse, while minimizing the total cost, which is the sum of the fixed opening costs and cost of assigning each store to each warehouse.

Data:

- There are 5 warehouses and 10 stores.
- The fixed costs for the warehouses are all identical and equal to 30.
- The following table shows the 5 locations with their respective capacities and costs of assigning each store to each warehouse.

Table 8. Problem data

	Bonn	**Bordeaux**	**London**	**Paris**	**Rome**
capacity	1	4	2	1	3
store1	20	24	11	25	30
store2	28	27	82	83	74
store3	74	97	71	96	70
store4	2	55	73	69	61
store5	46	96	59	83	4
store6	42	22	29	67	59
store7	1	5	73	59	56
store8	10	73	13	43	96
store9	93	35	63	85	46
store10	47	65	55	71	95

Exercise folder

`<training_dir>\\COS125.labs\Warehouse\work`

The text file at **`C:\\COSTraining\CodeSnippets\WarehouseLocation.txt`** contains some code snippets that you may cut and paste to complete this exercise. Alternatively, you can refer to the solution located at **`<training_dir>\\ COS125.labs\Warehouse\solution`**

Model the problem using IP

Objective

- Use IP principles and Boolean decision variables to create a model optimizing warehouse allocation

Action

- Define constraints

Documentation references

Language > Language User's Manual > Introduction to OPL > Language overview > Mathematical programming > Integer programming

Language > Language User's Manual > The application areas > Applications of linear and integer programming > Mixed integer linear programming

Language > Language Reference Manual > OPL, the modeling language > Expressions > Boolean expressions

Use Boolean decision variables

- Import the **`warehouseWork`** project and examine the model file.
- The key idea in representing a warehouse-location problem as an integer program is to use a Boolean (1–0 or true/false) decision variable for each (warehouse, store) pair to represent whether a warehouse supplies a store:

  `dvar boolean Supply[Stores][Warehouses];`

  In other words, **`Supply[s][w]`** is **`1`** if warehouse **`w`** supplies store **`s`** and zero otherwise.
- In addition, the model associates a decision variable with each warehouse to indicate whether the warehouse is open:

  `dvar boolean Open[Warehouses];`

Define the objective function

The objective function

```
minimize
   sum(w in Warehouses) Fixedcost * Open[w] +
   sum(w in Warehouses, s in Stores) Supplycost[s][w] * Supply[s][w];
```

expresses the goal that the model minimizes the fixed cost of the selected (i.e. open) warehouses and the supply costs of the stores.

Define constraints

The constraints state that:

- Each store must be supplied by a warehouse
- Each store can be supplied only by an open warehouse
- No warehouse can deliver more stores than its allowed capacity

A warehouse can supply a store only when it is open. This constraint can be expressed by inequalities of the form:

```
forall(w in Warehouses, s in Stores)
    Supply[s][w] <= Open[w];
```

This ensures that when warehouse **`w`** is not open, it does not supply store **`s`**. This follows from the fact that **`open[w] == 0`** implies **`supply[w][s] == 0`**.

As an alternative, you can write:

```
forall(w in Warehouses)
    sum(s in Stores) Supply[s][w] <= Open[w]*Capacity[w];
```

This formulation implies that a closed warehouse has no capacity.

- Look at the model carefully and discuss with your instructor and fellow trainees how the model is constructed.
- Write the remaining constraints.

Define instance data

- The file **`warehouse.dat`** defines the instance data as shown in the table in the problem definition.
- It declares the warehouses and the stores, the fixed cost of the warehouses, and the supply cost of a store for each warehouse.
- Run the model and examine the results.

Chapter 10. Portfolio optimization

Problem Description

This is a quadratic programming exercise for IBM ILOG CPLEX Optimization Studio.

Principles of the problem:

- In order to mitigate risk while ensuring a reasonable level of return, investors purchase a variety of securities and combine these into an investment portfolio. Any given security has an expected return and an associated level of risk (or variance). There is also a tendency for securities to **covary**, i.e. to change together with some classes of securities (positive covariance), and in the opposite direction of other classes of securities (negative covariance).
- To optimize a portfolio in terms of risk and return, an investor will evaluate the following:
  - Sum of expected returns of the securities
  - Total variances of the securities
  - Covariances of the securities
- A portfolio that contains a large number of positively covariant securities is more risky (and potentially more rewarding) than one that contains a mix of positively and negatively covariant securities.

What to model:

- Choose securities for the portfolio to improve its return and decrease its volatility.

  Tip: As the securities covary with one another, selecting the right mix of stocks can change or even reduce the volatility of the portfolio with the same expected return.
- At a given expected rate of return, there is one portfolio which has the lowest risk.
- The problem is to write a model that finds the mix of securities that provides the lowest risk for a pre-selected rate of return.

A quadratic function

- If you plot each lowest-risk portfolio for each expected rate of return, you will observe that the result is a convex graph, called the efficient frontier.
- The risk-return characteristics of a portfolio change in a non-linear fashion, and so, **quadratic expressions** are needed to model them.

Exercise folder

`<training_dir>\\COS125.labs\Portfolio\work`

The text file at **`C:\\COSTraining\CodeSnippets\Portfolio.txt`** contains some code snippets that you may cut and paste to complete this exercise. Alternatively, you can refer to the solution located at **`<training_dir>\\COS125.labs\Portfolio\solution`**

.

Write objective and constraints

Objectives

- Use QP principles to create a portfolio optimization model

Actions

- Model the problem using QP
- Define objective and constraints
- Use a logical constraint to limit diversity

Documentation references

Language > Language User's Manual > The application areas > Quadratic programming

Model the problem using QP

- Import the **`portfolioWork`** project and examine the **`portfolio.mod`** file.
- In this model, most of the problem is already defined. It is up to you to write the objective and constraints.
- The data elements are:

```
{string} Investments = ...;
float Return[Investments] = ...;
float Covariance[Investments][Investments] = ...;
float Wealth = ...;
float goalReturn = ...;
range float FloatRange = 0.0..Wealth;
```

- The decision variables are:

```
dvar float  allocation[Investments] in FloatRange;
```

Define objective and constraints

- The objective is to minimize the portfolio risk (variance).

  The covariance of returns of stocks **`i`** and **`j`** is defined as the allocation of the portfolio to stock **`i`** times the allocation of the portfolio to stock **`j`** times the covariance between **`i`** and **`j`**.
- Write the constraints
  - Allocate All Wealth
  - Meet Total Return Minimum

    Return is defined as the sum of the allocation of the portfolio to stock **`i`** times the expected return of stock **`i`**.
- Run your solution and debug it if necessary.

Use a logical constraint to limit diversity

If you run the solution file **`<training_dir>\\COS125.labs\Portfolio\solution\portfolioSolution`**, you will note that every possible security has an allocation assigned to it. In many cases, this type of diversity of investment is desirable, but some investors' objectives may include a limit on the number of different stocks in their portfolio.

To do this, it might seem logical to simply use a linear constraint: count the number of **`allocations`** (using the **`card`** function) and ensure that this number is always greater than or equal to a data element set as the diversity limit (call it **maxSecurities**). However, **card** only works over sets, and **allocations** is a decision variable, so another way has to be found, using a logical constraint.

1. Copy the **portfolioSolution** project and paste it as**<training_dir>\\ COS125.labs\Portfolio\work\portfolioLimitedWork**

   Note: You must do this from inside the OPL Projects Navigator. Do not attempt to copy and paste the project directory in the file system, because you will not be able to import the copy unless you also edit the .project and .oplproject files to change the project name, as well as (optionally) the .mod and .dat filenames.
2. Add an integer data declaration to create the data element **maxSecurities.** This can be initialized internally in the model or from the **.dat** file. Set its value to 5.
3. Define a logical constraint that says that the number of non-zero **allocations** must be equal to or less than **maxSecurities.**

   Tip: To set the maximum for non-zero applications equal to **maxSecurities,** use a formula that tests for the number of allocations set to 0, and constrains the model to limit that number to be at least equal to or greater than the difference between the total number of securities under consideration, and **maxSecurities** (strict inequalities are not permitted in OPL).
4. Run the model and debug it if necessary.

A different logical constraint: investment percentage

Investors' objectives can be very different, so the model needs to be flexible enough to be reused with a variety of investor needs. Simply by changing the logical constraint, for example, you can adjust the model to require that any investment represent, at minimum, 6% of the overall portfolio value.

1. Copy the **portfolioLimitedWork** project and paste it as**<training_dir>\\ COS125.labs\Portfolio\work\portfolioLimited2Work**

   Note: You must do this from inside the OPL Projects Navigator. Do not attempt to copy and paste the project directory in the file system, because you will not be able to import the copy unless you also edit the .project and .oplproject files to change the project name, as well as (optionally) the .mod and .dat filenames.
2. Replace the integer data declaration ,**maxSecurities**, with a float value called **minAllocation**. Initialize this value from the **.dat** file as 0.06.
3. Replace the logical constraint **maxStock** with one called **minInvestment**. This time, it must require that any investment, at minimum, must be greater than or equal to the value of **minAllocation**.
4. Run the model and debug it if necessary.
5. Compare the results with the results from the **portfolioSolution** project.
6. How many allocations are there? Try changing the value of **minAllocation** and running the problem again. Compare the different results

Chapter 11. Staffing Problem

Objective of the exercise

This exercise reviews many of the concepts for creating and solving optimization models in IBM ILOG CPLEX Optimization Studio. It applies several MP and OPL concepts, including:

- the OPL concepts of sets, tuples and sparse data structures
- integer programming
- constraint relaxation
- IBM ILOG Script
- multi-table queries to get the required data from a database
- OPL interfaces

Problem description

- A foreman needs to hire workers to complete a job. As the customer has already paid for the work, there is no time deadline to meet and the foreman does not want to hire more people than necessary.
- Each worker has a certain number of days of availability, as well as an associated skill group. Each skill group contains a collection of skills that can be used to complete the job.
- The job demands a given number of hours of each skill.
- The total amount of time spent among all workers on any given skill should exceed the time the job demands for that skill.
- The total amount of time spent among all workers with the same skill group should not exceed the total availability associated with workers in that skill group.

The foreman wants to know which workers to hire in order to complete the project using the minimum number of workers.

Database description

- This lab uses a DB2 database. Similar actions are possible with other supported databases, such as Microsoft Access or Oracle.
- The name of the DB2 database is `SKILLS`. To access this database:
  1. Click the green database icon in the lower-right corner of the screen
  2. Select DB2 Control Center...
  3. In the pop-up, select the Advanced View
  4. In the Object View, expand **All Databases > SKILLS**
  5. Right-click SKILLS and select Connect...
  6. Connect with User ID = `OPLuser` and password = `OPL1234`. Note that this is case-sensitive.
  7. Once you've connected to the SKILLS database, you can look at the contents of the following tables by double-clicking the table name:
     - SKILLS: List of skills, with fields `skill_id` and `skill_name`
     - SKILL_AVAILABILITY: Availability of each worker, with associated skill group, with fields `name`, `skill_group_id`, and `availability`

  - SKILL_DEMAND: Demand for each skill, with fields `skill_id` and `demand`
  - SKILL_GROUPS: List of skill groups, with fields `skill_group_id` and `skill_group_name`
  - SKILL_GROUP_SKILLS: List of skills for each skill group, with fields `skill_group_id` and `skill_id`
  - SKILL_AVAILABILITY_INF: Infeasible availability data to be used in one of the exercises
  - NEW_HIRES: The list of new hires. The solution will be exported to this table.
- The remaining tables in the SKILLS database were generated by default when the database was created. You do not need to look at these tables.

Summary of steps

This lab takes you through a series of steps to review some old material and introduce some new material. These steps are:

1. Steps to the database solution: Review defining OPL data structures and constraints, and practice using both table loading and existing sets to populate OPL data structures.
2. Constraint relaxation: Review interpreting the **Conflicts** and **Relaxations** output tabs in the case of infeasible solutions.
3. LP relaxation script: Practice using IBM ILOG Script to write a simple solution heuristic.
4. Model access script: Practice using IBM ILOG Script to solve a variation of the original model.
5. LP relaxation API: Practice using OPL interfaces to duplicate the LP relaxation step.
6. Model access API: Practice using OPL interfaces to duplicate the Model access script step.

Exercise folder

`<training_dir>\COS125.labs\Staffing\...`

Steps to the database solution

Objectives

- Practice defining OPL data structures and constraints
- Practice using table loading to populate OPL data structures
- Practice using OPL sets to populate OPL data structures

Actions

- Declare and instantiate `worker` and `skillGroup` pairs and worker's availability
- Declare and instantiate the skills list and the demand for each skill
- Data: get relation of skills to workers
- Define decision variables and objective
- Define constraints
- Post-processing for result output

Documentation references

Language > Language Reference Manual > OPL, the modeling language > Data types > Data structures > Tuples

Language > Language Quick Reference > OPL keywords > key

CPLEX Studio IDE > IDE Tutorials > Working with external data

Language > Language Reference Manual > OPL, the modeling language > Data sources > Database initialization > Reading from a database

Language > Language Reference Manual > OPL, the modeling language > Data sources > Data initialization > Initializing sets

Language > Language User's Manual > IBM ILOG Script for OPL > Introduction to scripting > Preprocessing and postprocessing

Exercise folder

`<training_dir>\COS125.labs\Staffing\Database_model\work`

You will update both the **`.mod`** and **`.dat`** files of the **`staffingWork`** project.

Important: This lab uses a DB2 database. However, at the end of the .dat files you can find commented examples of connection strings to be used for Microsoft Access or ODBC datasources.

Note: Make sure that you are connected to the SKILLS database (as described at the start of this workshop) before starting with the exercises.

Launch the IDE and import the **`staffingWork`** project.

Declare and instantiate worker and skillGroup pairs and worker's availability

Each worker belongs to a single **`skillGroup`** that represents a subset of all the skills of all the workers. In human resources terms, this would be the set of competencies for which a worker is qualified. An individual worker can share his/her available time performing any of the associated **`skillGroup`**'s skills, but cannot work in any skill outside the **`skillGroup`**.

In the model file:

1. Look at the declaration (already done) of the tuple called **`WorkerSkillGroupPair`** that it contains the information for the **`skillGroupName`** and the **`workerName`** . Keys are used here for data integrity. Note that if no keys are declared, the default setting is to assume that all tuple elements are keys.
2. Declare a set **`workerSkillGroupPairs`** of type **`WorkerSkillGroupPair`**.

   Note: What you are doing here is creating a set, **`workerSkillGroupPairs`**, which has as its members, instances of the tuple **`WorkerSkillGroupPair`**. The tuple name is usually declared in the singular (**`WorkerSkillGroupPair`**), and the tuple set that collects it usually has the same name as the tuple data type but in the plural form (**`workerSkillGroupPairs`**). Also, note that usually, the data name starts with lower case while the data type starts with upper case (e.g. **`tuple Pair {...};`** and **`{Pair} pairs = ...;`**).
3. Declare an array **`workerAvailability`** of type **`float`** indexed over **`workerSkillGroupPairs`**

Note: This array is a sparse array because it is indexed over the skill group name combination, `WorkerSkillGroupPairs` as opposed to being indexed individually over `skillGroupName` and `workerName`.

In the data file:

1. Populate `workerSkillGroupPairs` and `workeravailability` from the database using table loading. Use the following query:

   ```
   select sa.name,sg.skill_group_name, sa.availability from skill_groups
   sg, skill_availability sa where sg.skill_group_id = sa.skill_group_id
   ```

   Note: Note that the index of the availability array and the elements of the array are created simultaneously from the database thanks to table loading.
2. Use `workerSkillGroupPairs` to populate other data structures
3. Create the set of `namesOfWorkers`

Declare and instantiate the skills list and the demand for each skill

Demand is skill dependent. You therefore have to design several different structures to manipulate different aspects of the relationship between a worker and its associated skill set.

In the model file:

1. Declare the skills using a set structure
2. Declare an array of the skill's demand indexed by each skill name

In the data file:

1. Populate the skill set and the demand array from the database using table loading. Use the following query:

   ```
   select sa.name, s.skill_name from skill_availability sa, skills s,
   skill_group_skills sgs where sa.skill_group_id = sgs.skill_group_id and
   sgs.skill_id = s.skill_id
   ```

Data: get relation of skills to workers

Each worker provides several skills, which are included in the pool of skills of his skill group.

In the model file:

1. The model file already contains the declaration of a tuple `WorkerSkillPair` that contains a worker and one of the worker's associated skills.
2. Declare a set `workerSkillPairs` of type `WorkerSkillPair` that is read from the data base.

   Note: What you are doing here is creating a set, `workerSkillPairs`, which has as its members, instances of `WorkerSkillPair`.
3. Use the set `workerSkillPairs` to create an array `workerSkillsList`, indexed over `namesOfWorkers`, that lists the skills belonging to each worker.

   Note: Here, you are creating a two-dimensional array where one dimension is the name of each worker and the other dimension contains every instance of the set `workerSkillPairs` that contains the list of skills of each worker.
4. Use the set `workerSkillPairs` to create an array `skillsWorkerList`, indexed over skills, that lists the workers capable of each skill.

Note: Here, you are doing the same thing, but this time listing each skill, then referencing all workers who have that associated skill.

Note that using a tuple structure for **`workerSkills`** contributes to the sparsity of the model, because only the relevant worker/skill combinations are listed, as opposed to all possible combinations of workers and skills.

In the data file:

1. Populate **`workerSkillPairs`** from the database by using the following query:

   `select sa.name, s.skill_name from skill_availability sa, skills s, skill_group_skills sgs where sa.skill_group_id = sgs.skill_group_id and sgs.skill_id = s.skill_id`

Define decision variables and objective

1. Define a Boolean decision variable **`hireWorker[namesOfWorkers]`** to indicate whether each worker is hired or not.
2. Define a float decision variable **`workerSkillTime[workerSkillPairs]`** to indicate how much time each worker spends on each skill.
3. Define the objective to minimize the number of workers hired.

Define constraints

1. Define a constraint **`ctAvailability`** that ensures that each individual worker's availability limit is met as follows:

```
          forall (w in namesOfWorkers)
            ctAvailability : sum(s in workerSkillsList[w])
workerSkillTime[<w,s>] <= workerAvailability[<w>,<workerSkillGroup[w]>] * hireWorker[w];
```

2. Define a constraint **`ctMeetDemand`** to ensure that the amount of time spent by all workers with a particular skill is at least as great as the demand for that skill.

Post-processing for result output

1. Create a list, **`hiredWorkers`**, of the names of the workers to be hired.
2. Note the use of an **`execute`** script block to write the list to the **Scripting log** output tab.
3. Use **`DBExecute`** and **`DBUpdate`** statements to first clear the table of new hires and then populate it with the solution to your model.

Solution

1. Check the solution of both the **`.mod`** and the **`.dat`** files in the **`<training_dir>\COS125.labs\Staffing\Database_model\Solution`** directory.
2. Run your model and make any necessary changes if it doesn't correspond to the solution.
3. Make a note of the solution.

Constraint relaxation

Objective

- Review interpreting relaxations made by CPLEX.

Action

- Run model and make correction

Documentation references

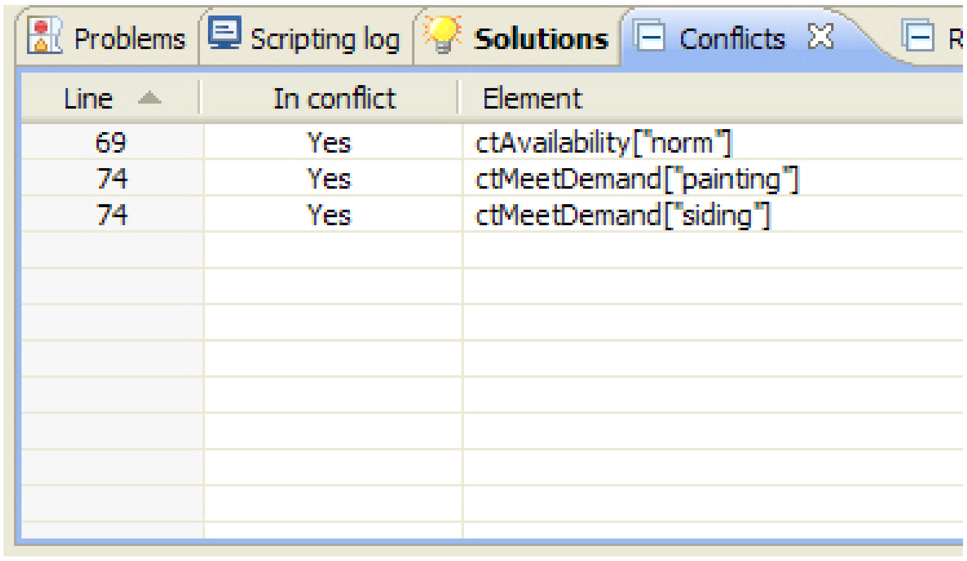
CPLEX Studio IDE > IDE Tutorials > Relaxing infeasible models > How relaxation and conflict search works > Relaxations

Language and Interfaces Examples > OPL examples that use CPLEX Optimizer for Mathematical Programming > Relaxation of infeasible models

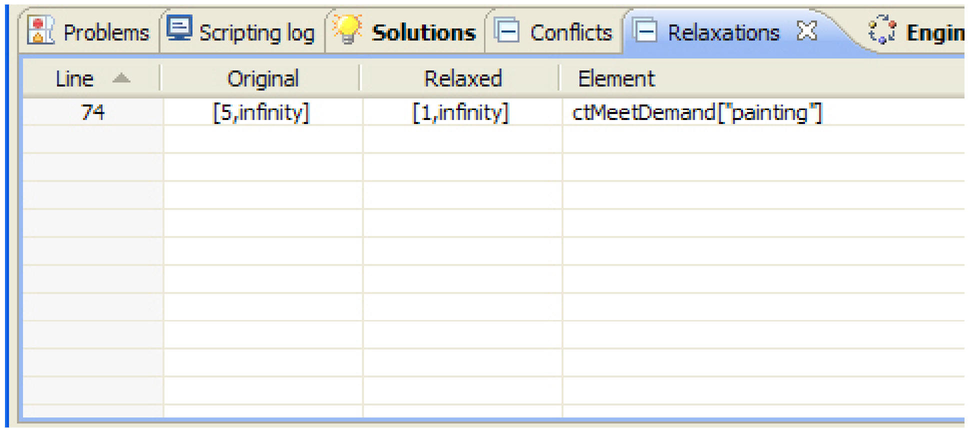
This step reviews some of the material on how to interpret the **Conflicts** and **Relaxations** output tabs in the case of infeasible models. The step is observation-oriented: you'll have very little to do except understanding what is happening.

Run the model and make corrections

1. Import the **`<training_dir>\COS125.labs\Staffing\Constraint_Relaxation\ relaxStaffingSolution`** project into the OPL Projects Navigator. Leave the **Copy projects into workspace** box unchecked.
2. See in the .dat file that the availability data now comes from the SKILL_AVAILABILITY_INF table. This table contains data that leads to an infeasible model. You will practice using the information in the **Conflicts** and **Relaxations** tabs in the IDE to find the cause of the infeasibility.
3. Run the model.
4. Look at the **Solutions** output tab and see that CPLEX Studio proposes a relaxed solution because the model is now infeasible.
5. Study the **Conflicts** and **Relaxations** output tabs to determine the cause of the infeasibility:



Line	In conflict	Element
69	Yes	ctAvailability["norm"]
74	Yes	ctMeetDemand["painting"]
74	Yes	ctMeetDemand["siding"]



Line	Original	Relaxed	Element
74	[5,infinity]	[1,infinity]	ctMeetDemand["painting"]

This relaxation suggests that the demand for painting should be reduced to 1 day, however this is not a reasonable change to make to the data.

6. Notice in the **Conflicts** tab that the constraint **`ctMeetDemand`** is in conflict with Norm's availability limit. Norm's availability in the SKILL_AVAILABILITY_INF table is 9 compared to his original availability of 13 in the SKILL_AVAILABILITY table.
7. Change Norm's availability to 13 in the SKILL_AVAILABILITY_INF table, and click **Commit** and then **Close**.
8. Run the model again and compare your solution to the original. Notice that the feasible solution is slightly different (Emma is now hired instead of Eduardo) because you are now using the more correct form of the availability constraint.

LP relaxation script

Objectives

- Use IBM ILOG Script to write a simple solution heuristic

Actions

- A heuristic solution approach
- Steps to implement the heuristic

Documentation references

Language > ILOG Script Reference Manual > IloOplModel

Exercise folder

`<training_dir>\COS125.labs\Staffing\Scripted_models\LP_Relaxation\work`

A heuristic solution approach

For some complex MIP models, it may take prohibitively long to find even a first feasible solution, not even to mention an optimal solution. In such cases, it could be very useful to use a heuristic approach in order to find a partial MIP solution, and then to feed that solution as an advanced starting point to the original model. CPLEX's default settings are such that if an advanced start is available in memory, it will be used in a subsequent solve and you therefore do not need to do anything special to ensure that this advanced information will be used, except for solving the models in sequence using IBM ILOG Script (or an API).

In this exercise, you will practice implementing such a heuristic for the staffing problem. The steps of the heuristic you'll be implementing are as follows:

1. Solve the LP relaxation of the original MIP model.
2. Fix all Boolean decision variables that have a relaxed value above 0.6 to 1.
3. Solve the MIP with the fixed subset of Boolean decision variables. Note that the model is still guaranteed to be feasible, because only a subset of workers is forced to be hired and there are no forced exclusions. The MIP now has a reduced set of decision variables, because a subset of them has been fixed. This would be especially useful for larger models where the size of the MIP could be significantly reduced. The solution to this model is not necessarily optimal, but you're not interested in optimality at this point seeing that you're only using the solution as a starting point for the original model.
4. Use the solution from Step 3 as an advanced starting point for solving the original MIP.

Steps to implement the heuristic

1. Import the `staffing2Work` project in the `<training_dir>\COS125.labs\Staffing\Scripted_models\LP_Relaxation\work` directory. Leave the **Copy projects into workspace** box unchecked. You'll write your heuristic inside the `main` block. Some script statements to print intermediate values to the **Scripting log** output tab are already included. To see the **Scripting log** output tab, select **Window > Show view > Scripting log** from the main menu bar.
2. Generate the original model using the `thisOplModel.generate()` command, relax the model using the `thisOplModel.convertAllIntVars()` command, and solve the relaxation using the `cplex.solve()` command
3. Test each Boolean decision variable in the array `hireWorker[namesOfWorkers]` for values greater than or equal to 0.6, and fix the lower bounds on any such Booleans to 1 using the `thisOplModel.hireWorker[w].LB = 1` statement.
4. Undo the relaxation to the model using the `thisOplModel.unconvertAllIntVars()` command. Solve the model using the `cplex.solve()` command.
5. Use the previous solution as an advanced start to the original MIP. To do this, reset all lower bounds to 0 and solve the MIP.
6. Check your script with the solution in the `<training_dir>\\COS125.labs\Staffing\Scripted_models\LP_Relaxation\solution` directory, and make any necessary changes.
7. Run your model and check the **Scripting log** output tab to follow the sequence of steps.

Note: IBM ILOG Script is especially useful for writing heuristics such as the one your just implemented. Without script, you would've had to write separate models, solve each of the models individually and fix variable values manually in between solves. IBM ILOG Script automates all these steps in the `main` script block, without changing anything in the original OPL model definition.

Model access script

Objectives

- Practice using flow control and model access to solve a variation of the original model.

Actions

- Problem description

- Steps to complete the model access script

Documentation references

Language > ILOG Script Reference Manual > IloOplModel

Language > Language Reference Manual > IBM ILOG Script for OPL > Language structure > Syntax

Exercise folders

`<training_dir>\COS125.labs\Staffing\Scripted_models\Model_Access\work`

Note: You can find useful examples related to model access and flow control in the example directory installed with the product,**`<\COShome>\examples\opl`,** where **`<\COShome>`** is the top level directory where CPLEX Studio is installed. Specifically, **`mulprod`** and **`cutstock`** projects, in the **`main`** run configurations, contain some examples for changing decision variables and accessing model information using IBM ILOG Script.

Problem description

1. The foreman hopes to hire fewer people by getting one of the workers to work a few extra hours.
2. Determine which person should have their availability raised to reduce the number of workers by one.
3. Determine the minimum raise in availability required to reduce the minimum number of workers by one.
4. Use a script to solve this problem and do not edit the original model definition.

Steps to complete the model access script

1. Import the project **`Staffing3Work`** in the work folder. Leave the **Copy projects into workspace** box unchecked. Study the part of the script that has been completed and note that a name, **`hiring`**, has been assigned to **`thisOplModel`**.
2. Under the comment that reads **`// define script variables worker, slackVal and lowest`**, define the following script variables: **`worker`** to denote the worker with least slack in their availability, **`slackVal`** to denote the slack value of each **`ctAvailability`** constraint, and **`lowest`** to denote the lowest slack value among all **`slackVal`** with initial value of **`Infinity`**.
3. Fill in the **`for (var w in hiring.namesOfWorkers)`** loop: To determine which worker should have their availability raised, iterate through the list of workers, **`namesOfWorkers`**, and determine which worker's availability constraint, **`ctAvailability`**, has the least slack. With each iteration, first find the value of the slack, **`slackVal`**, using the expression **`thisOplModel.ctAvailability[w].slack`**. Next test whether **`slackVal`** is less than the lowest slack found thus far, **`lowest`**. If it is, assign **`slackVal`** to **`lowest`**. Break out of the loop if a **`slackVal`** equal to zero is reached, because all slack values are greater than or equal to zero (you may see some very small negative numbers due to precision errors – in this case test whether **`slackVal`** is less than or equal to a small number such as 0.0001, instead of testing whether the slack is equal to zero).
4. Define and assign a value to a script variable representing the **`skillGroup`** corresponding to the worker with the lowest slack.

   Tip: Use the **`workerSkillGroup[worker]`** array.

5. Define and assign values to a script variable `availabilityIndex` representing the element in `workerSkillGroupPairs` corresponding to the `skillGroup` and the `worker`.

   Tip: Use the `find` keyword.
6. Use a `while` loop to iteratively add 1 unit to the worker's `endingAvailability` and then solve the model. The `while` loop should stop as soon as the objective value is 1 less than the original objective. The bulk of this loop has been completed. You need to insert the termination criteria, a statement to increase the `endingAvailability`, and a statement to set the worker's availability to the new `endingAvailability` value.
7. Include an extra termination criteria for the `while` loop, namely that the `endingAvailability` should not be more then 20 units greater than the `initialAvailability`, to cover the possibility that increasing this particular person's availability will not be of any help. Add an error message to be written to the Scripting log in case this condition is reached.
8. Check you script against the solution provided in the `<training_dir>\COS125.labs\Staffing\Scripted_models\Model_Access\solution` directory. Make any necessary changes.
9. Run your model and check the output in the **Scripting log** output tab.

LP relaxation API

Objective

- Get familiar with using the OPL interfaces.

Action

- LP relaxation steps

Documentation references

Language > ILOG Script Reference Manual > IloOplModel

Exercise folders

`COS125.labs\Staffing\API\LP_Relaxation\LP_Relaxation_Solution`

This lab uses Java code in an Eclipse project. Alternatively, you can find the .net solution at:

`COS125.labs\Staffing\API\LP_Relaxation\dotnet`

Note that if you were to use Microsoft Visual Studio for the .net solution (which is not used in this lab, but you may do this in your own time), you'd have to add references to the appropriate .dll as follows:
1. Open the `.sln` file
2. Go to the **Solution Explorer**
3. Right click the project or the **References** folder
4. Check that the OPL dll (`oplall.dll`) is in the references, and if not add it by selecting **Add References**

You can view the `mulprod` example in the `<\COShome>\examples\...` folder to see similar code as some of the code used for this lab.

LP relaxation steps

In this lab, you will duplicate the LP relaxation script step by using the API language code. In this version, you will first solve the original model, then the relaxed model, and then revert back to the original model.

The Eclipse project has already been set up and you will only work with the source code.

1. Launch Eclipse from the shortcut on the desktop and accept the default workspace.
2. Select **File > Import > General > Existing projects into workspace** and click **Next**.
3. In the field named Select root directory, browse to **`COS125.labs\Staffing\API\LP_Relaxation\LP_Relaxation_work`**. Select the **`LP_Relaxation_work`** project, and click **Finish**.
4. In the **Project Explorer**, expand the plus signs and double-click the **`LPRelax.java`** file to access the Java source code.
5. Take some time to study the code that has been partially completed. Notice how the existing **`staffing2.mod`** and **`staffing2.dat`** files are incorporated into the Java code.
6. Fill in the missing code where indicated with the comment "//FILL IN":
   - Use the **`convertAllIntVars()`** method to convert the model into the relaxed model.
   - Use the **`unconvertAllIntVars()`** method to convert the model back to a MIP.
   - Call **`postProcess()`** where indicated and note the effect thereof when solving the model. This demonstrates how tasks that are more easily programmed in the .mod file can be written in the post-processing script and called in the API code.
7. To run your code, right-click **`build.xml`** in the **Package Explorer** and select **Run As > 1 Ant Build**.
8. If you're having problems, check the solution in the **`LP_Relaxation_Solution`** project (import this the same way as the **LP_Relaxation_Work** project), and make any necessary changes.

Model access API

Objectives

- Use flow control and model access to increase worker availability and reduce the number of workers required

Actions

- Steps to completing the lab

Documentation references

Language > ILOG Script Reference Manual > IloOplModel

Language > ILOG Script Reference Manual > IloCplex

Language > ILOG Script Reference Manual > IloTupleSet

Language > ILOG Script Reference Manual > IloOplDataElements

Exercise folders

`\COS125.labs\Staffing\API\Model_Access\Model_Access_Work`

This lab uses Java code in an Eclipse project. Alternatively, you can find the .net solution at:

`\COS125.labs\Staffing\API\Model_Access\dotnet\work`

Examples that use similar code as used for this lab, are **`cutstock\cutstock_change.mod`** and **`mulprod\mulprod_main.mod`** in the **`<\COShome>\examples\opl`** folder.

Steps to completing the lab

In this lab, you will duplicate the Model Access script step by using the API language code.

The Eclipse project has already been set up and you will only work with the source code.

1. Launch Eclipse from the shortcut on the desktop and accept the default workspace.
2. Select **File > Import > General > Existing projects into workspace** and click **Next**.
3. In the field named Select root directory, browse to **`\COS125.labs\Staffing\API\Model_Access\Model_Access_Work`**. Select the **Model_Access_Work** project, and click **Finish**.
4. In the **Project Explorer**, expand the plus signs and double-click the **`AvChange.java`** file to access the Java source code.
5. Take some time to study the code that has been partially completed.
6. Fill in the missing code where indicated with the comment **`//FILL IN`**:
   - Link the code to the existing **`staffing3.mod`** and **`staffing3.dat`** files in the **Model_Access_Work** folder
   - Add code to solve the model and store the objective value.
   - Open **`staffing3.mod`** file in the IDE by double-clicking it and scroll down to the post-processing script. Note that the post-processing is used to print the solution, and also to write the slack values for the **`ctAvailability`** constraints to an array. You will access the slacks in the API using this array. In the Java code, call **`postProcess`** after solving the model.
   - Write the **`slacks`** from the array in the OPL model to an **`IloNumMap`** called **`workerSlackMap`** by using the **`getElement`** method. Note the various class names used to access OPL data types throughout the code.
   - In order to find the tuple corresponding to the skill group and worker with the least slack, get the contents of the **`skillGroupNames`** tuple set using the **`getElement`** method and iterate through the set until you've found the **`index`** that corresponds to the relevant skill group and worker.
   - Create a tuple from this **`index`** by using the **`makeTuple`** method, and find the **`initialAvailability`** associated with that tuple from the **`workerAvailabilityMap`**.
   - In the **`while`** loop, create new **`IloOplModel`** and **`IloCplex`** instances (see the top of the code for examples).

7. To run your code, right-click **`build.xml`** in the **Package Explorer** and select **Run As > 1 Ant Build**.
8. If you are having problems, check the solution in the **`LP_Relaxation_Solution`** project (import this the same way as the **LP_Relaxation_Work** project), and make any necessary changes in your work.

Chapter 12. OPL interfaces tutorial

Tutorial description

- This tutorial examines **8 of the 10 main extension classes** covered during the training for IBM ILOG CPLEX Optimization Studio.
- The original tutorial can be found in the documentation, in the Interfaces User's Manual. Links will lead you to the appropriate pages of the documentation. This workshop gives extra explanations on some classes and their instantiation.
- The tutorial is mostly based on the **`mulprod`** example, a basic multiperiod production case which corresponds to the following description:
  - Large linear-programming problems are often obtained from simpler ones by generalizing them along one or more dimensions
  - A typical extension of production-planning problems is to consider several production periods and to include inventories in the model. The multi-period production planning model generalizes the inside/outside production model. The main generalization is to consider the demand for the products over several periods and to allow the company to produce more than the demand in a given period. There is an inventory COSt associated with storing the additional production

Look at the mulprod example

Objective

- Global view of the code

Get familiar with the example

Select one of the following, depending on your language preference (C++, Java, C# or VB):

- **`<\COSHome>\examples\cpp\src\mulprod.cpp`**
- **`<\COSHome>\examples\java\mulprod\src\mulprod\Mulprod.java`**
- **`<\COSHome>\examples\dotnet\x86_.net2005_8.0\CSharp\Mulprod`**
- **`<\COSHome>\examples\dotnet\x86_.net2005_8.0\VisualBasic\Mulprod`**

In this tutorial, you will get familiar with some of the methods in the OPL API. You will not be doing any coding, and you can look at the code with a text editor.

Follow the tutorial steps

Objective

- Overview of extension classes in context

Action

- Complete the tutorial

Documentation references

Interfaces > Interfaces User's Manual > Tutorial > Creating an OPL model

Interfaces > Interfaces User's Manual > Tutorial > Specifying a data source

Interfaces > Interfaces User's Manual > Tutorial > Generating the concert model

Interfaces > Interfaces User's Manual > Tutorial > Solving the model

Interfaces > Interfaces User's Manual > Tutorial > Accessing the solution

Complete the tutorial

1. In the CPLEX Studio documentation, go to **Interfaces > Interfaces User's Manual > Tutorial**, and complete all the tutorial steps.

Important methods

Actions

- `IloEnv` and `OplFactory`
- `OplModel`

Documentation references

Language > Language User's Manual > IBM ILOG Script for OPL > Tutorial: Flow control and column generation

`IloEnv` and `OplFactory`

`IloOplFactory.end()` in Java and .Net; For C++ use **`IloEnv::end()`**.

- Deletes the OPL environment and releases the memory used by ALL the objects created by the factory
- Most of the time, you won't need to use the **`end()`** method of other classes, unless you write something memory intensive like an iterative task
- The **`end`** method is disabled by default in the IDE and can be enabled by setting **`mainEndEnabled`** to true.

  Note: Use caution in applying this setting. When it is enabled, you must ensure that memory is properly managed by your script. Faulty memory management, such as attempting to use an object after it has been deleted, may result in crashes.

`IloOplModel`

You should pay specific attention to the following methods that you're likely to use more often (refer to the documentation to learn more about these):

- **`addDataSource (IloOplDataSource source)`**
- **`convertAllIntVars()`** and **`unconvertAllIntVars()`** for LP relaxation
- **`end()`** (in Java; use **`IloEnv::end()`** in C++)
- **`generate()`**
- **`main()`**
- **`makeDataElements()`**

Chapter 13. Concurrent processing

Exercise objective

After completing this exercise, you should be able to implement a multi-threaded solution for high-volume processing, by using the Java OPL API.

Problem description

- The problem involves portfolio optimization, where the goal is to choose the best combination of assets to invest in.
- The objective involves a trade-off between maximizing return on investment and minimizing risk, captured by a factor, θ:
  - ρ = 0: risk is not important
  - ρ = 1: risk and return are of equal importance
  - ρ > 1: minimizing risk is more important than maximizing return
- The model considers N possible values of ρ, between 0 and 1, to find the optimal trade-off.
- Each value of N leads to an independent sub-model which can be solved on a separate thread.
- Once all sub-models are solved, a "tangency portfolio" is calculated, representing the best investment choice.

In this exercise, you will complete a procedure to distribute the N sub-models on multiple threads, and gather the results of all N solves to calculate the tangency portfolio. You do not need to understand the detailed mathematics behind the portfolio optimization problem, or the formulation of the OPL model, in order to complete this exercise. However, if you are interested in reading more, see the documentation where this problem is described in detail at **IDE and OPL > Optimization Programming Language (OPL) > Language User's Manual > Performance and memory usage > Multi-threading > A typical problem to solve**.

Summary of steps

The complete OPL .mod and .dat files are given – you only need to implement the multi-threading part in Java. The high-level steps are:

1. Study the partially completed code
2. Complete the code to create an solve a single problem instance
3. Complete the code to distribute N problem instances over multiple threads, and gather the results to calculate the tangency portfolio
4. Run the multi-threaded solution

Exercise folder

`<training_dir>\COS125.labs\ConcurrentProcessing\ConcurrentProcessing_Work`

Study the existing code

Objectives

- Open the work project in Eclipse
- Study the Java code structure

Documentation references

IDE and OPL > Optimization Programming Language (OPL) > Language User's Manual > Performance and memory usage > Multi-threading > A typical problem to solve

Open the work project in Eclipse

1. Launch Eclipse by double-clicking the shortcut named "Eclipse" on the desktop (accept the default workspace).
2. Select **File > Import > General > Existing Projects into Workspace**.
3. In the **Import** window that opens, in the **Select root directory** text field, browse to **`<training_dir>\COS125.labs\ConcurrentProcessing\ConcurrentProcessing_Work`** and click OK to import the **ConcurrentProcessing_Work** project. Click **Finish**.
4. Expand the project to see the partially completed source code:
   - **ConcurrentProcessing_Work > src > ConcurrentProcessing > ConcurrentProcessing.java**
   - **ConcurrentProcessing_Work > src > ConcurrentProcessing > PortfolioProblem.java**
5. Take a look at the contents of the OPL **`portfolio.mod`** file, by right-clicking the filename in the **Package Explorer** and choosing **Open with > Text editor**. This OPL model represents one instance of the portfolio problem, that is, a model for a given value of ρ. In this exercise, you will create multiple instances of this model (for various values of ρ) to be solved on multiple threads. You do not need to edit this file during this exercise.
6. Take a look at the contents of the OPL **`portfolio.dat`** file, by right-clicking the filename in the **Package Explorer** and choosing **Open with > Text editor**. This .dat file includes data for one instance of the portfolio problem. Notice that no value for ρ (**`rho`**) is given, even though this data item is declared in the .mod file. This is because the values for ρ will be generated in the Java code. You do not need to edit this file during this exercise.

Study the code structure for the Java implementation

Look at the partially completed Java code. The implementation consists of two main classes:

- **`PortfolioProblem.java`**: This code represents an instance of the portfolio problem, with the following contents:
  - An inner class **`Request`**, which represents the value of **`rho`** for the problem instance (already completed).
  - A method **`solve`**, used to create and solve the problem instance. You will complete this method in a subsequent step.
  - An inner class **`Result`**, which represents the solution (result data) to the problem instance (already completed).
- **`ConcurrentProcessing.java`**: This code represents the multi-threaded solution procedure. The procedure creates N instances of **`PortfolioProblem`**, submits them to multiple threads, and processes the results to arrive at the tangency portfolio. You will complete part of this class in subsequent steps.

Complete the code to create and solve a problem instance

Objective

- Create the code to create and solve a problem instance

Complete the solve method

Before implementing the multi-threaded solution procedure, you should complete the code to create and solve a problem instance. The **solve** method of the **PortfolioProblem.java** class is used for this purpose.

1. Go to **PortfolioProblem.java**, and scroll to the partially completed **solve** method. Notice that it takes a **Request** (in effect, a value for **rho**) as an argument.
2. Complete the first line to get the value of **rho** from the request:
   ```
   double rho = request.getRho();
   ```
3. Before the **try** block, add code to create the OPL factory for the instance (each instance will have its own OPL factory, to be distributed to a thread):
   ```
   // The OPL factory will create and handle all the needed OPL objects.
        IloOplFactory oplF = new IloOplFactory();
   ```
4. At the start of the **try** block, add code to create the model and data source from the **portfolio.mod** and **portfolio.dat** files:
   ```
   // Create the OPL model source based on the .mod file.
          IloOplModelSource source = oplF.createOplModelSource(DATADIR + "/portfolio.mod");
          // Create the OPL data source based on the .dat file.
          IloOplDataSource dataSource = oplF.createOplDataSource(DATADIR + "/portfolio.dat");
   ```
5. The next few lines of code have already been completed, and are used to create the error handler and default settings, to create the OPL model definition by linking the source and the settings, to create the CPLEX Optimizer instance that will solve the model, and to create the OPL model from the model definition and CPLEX Optimizer instance:
   ```
    // Create an error handler.
          IloOplErrorHandler handler = oplF.createOplErrorHandler();
          // Create the default settings.
          IloOplSettings settings = oplF.createOplSettings(handler);
          // Create the OPL model definition by linking the source and the settings.
          IloOplModelDefinition def = oplF.createOplModelDefinition(source, settings);
          // Gets the algorithm.
          IloCplex cplex = oplF.createCplex();
          // Create the OPL model from the OPL defition and the algorithm.
          IloOplModel opl = oplF.createOplModel(def, cplex);
   ```
6. Next, complete the code to add the value of **rho** to the data, and add the data sources to the OPL instance:
   ```
   // Create the missing OPL data elements (rho).
          IloOplDataElements dataElements = oplF.createOplDataElements();
          dataElements.addElement(dataElements.makeElement("Rho", rho));
          // Add the data sources to the OPL model.
          opl.addDataSource(dataElements); // the value of rho
          opl.addDataSource(dataSource); // all other data in the .dat file
   ```
7. The next few lines of code are already completed, and are used to generate and solve the model on one thread, and to get the solution (**totalReturn** and **totalVariance**):
   ```
   opl.generate(); // generate the model

           cplex.setOut(null);
          // Here, CPLEX is forced to use only 1 thread to get only the OPL
          // multithreading efficiency without interference from CPLEX.
          cplex.setParam(IntParam.Threads, 1);
   ```

```
        if (!cplex.solve()) {
          throw new IloException("solve failed");
        }

         double totalReturn = opl.getElement("TotalReturn").asNum();
         double totalVariance = opl.getElement("TotalVariance").asNum();
```

8. Complete the code to create the **Result** object to return:

```
// Create the result handler to return.
       Result result = new Result(totalReturn, totalVariance, rho);
       return result;
```

You've now completed the code to solve an instance of the portfolio problem on a single thread. The next step is to implement the multi-threaded solution process.

Complete the code to implement multi-threaded processing

Objectives

- Study the main method
- Create requests
- Submit the requests to multiple threads
- Gather and process the results

Study the main method

1. Look at the **main** method, which is already completed. This method performs four solutions procedures, all making use of the **calculate** method:
   - Solve all N (**samples** in the code) sub-models on 1 thread to find a warm start.
   - Solve all N sub-models on 1 thread to find a reference value for the solution time of N sub-models in sequence.
   - Solve the N sub-models distributed over a specified number of threads (4 by default), and calculate the efficiency compared to 1 thread.
   - Solve the N sub-models distributed over the maximum available number of threads, and calculate the efficiency compared to 1 thread. These four procedures are included for future reference, but for this exercise you only need to be concerned with the **calculate** method.
2. The **calculate** method is used to create requests, submit them to the multiple threads, and process the results. You will complete this method in subsequent steps.

Complete the code to create requests

1. In the **calculate** method, complete the first line to create the list of requests

```
// Create the request to sumbit to the pseudo OPL server.
       List<requests> requests = makeRequests(samples);
```

2. The **makeRequests** method takes the number of sub-models (**samples**) as an argument, and returns a list of **requests**. Scroll down to complete the **makeRequests** method by adding the line of code to create a **Request**, to be added to the list of **requests**

```
// Create a request
       tasks.add(new Request(rho));
```

Complete the code to submit the requests to multiple threads

1. Now that the list of **requests** is available, complete the line of code to submit the **requests** in the **calculate** method:

```
 long t1 = System.currentTimeMillis();
       List<results> results = submitRequests(requests, threads);
 long t2 = System.currentTimeMillis();
```

`t1` and **`t2`** are used to measure the complete solution time.

2. The **`submitRequests`** method takes the list of **`requests`** and the number of **`threads`** as arguments, and returns the list of **`results`**. Scroll down and complete the **`submitRequests`** method to transform the list of requests to a list of **`Callables`** (**`tasks`**), create a thread pool, invoke the **`tasks`** for solution on the multiple threads, and get the list of **`results`** to return:

```
// Transformation of the request criteria (rho values) into real requests.
List<Callable<Result>> tasks = getCallables(requests);

// create a thread pool
ExecutorService service = Executors.newFixedThreadPool(threads);

// invoke tasks for solution on multiple threads
List<Future<Result>> result = service.invokeAll(tasks);

service.shutdownNow();

// results to return
List<Result> ret = getResults(result);
return ret;
```

3. Complete the **`getCallables`** method. This method takes the list of **`requests`** as an argument (in effect, values of **`rho`**) and transforms it into a list of portfolio problem instances (**`tasks`**) to solve:

```
tasks.add(new Callable<Result>() {
         public Result call() throws Exception {
           return (new PortfolioProblem()).solve(m);
         }
       });
```

Complete the code to gather and post-process results

1. The **`getResults`** method is already completed in the code. This method takes the **`result`** list, from the multiple threads, as an argument and returns a list, **`ret`**, of **`Result`** objects. It is called from within the **`submitRequests`** method, which in turn returns the list of **`results`** to the **`calculate`** method.
2. In the calculate method, complete the code to process the results:

```
// process the results
processResults(results, rfr);
```

3. The **`processResults`** method is already completed in the code. It takes the **`results`** and a calculation parameter (**`rfr`**) as arguments, and outputs the tangency portfolio. The details of this method come from the problem description available in the documentation (see the documentation reference at the start of this workshop).

Run the multi-threaded solution

Objectives

- Run the completed code

Run the multi-threaded solution process

1. Once you've completed the code, right-click **`build.xml`** in the **Project Explorer** and select **Run As > 1 Ant Build**.
2. The training virtual machine image has only one processor, and this code is therefore run on one thread. However, it should give you the idea of what would happen when this is run on multiple threads.

3. Refer to the solution in the **`ConcurrentProcessing_Solution`** folder if you are having any problems.

Printed in USA