A DEEP DIVE INTO STRATEGIC NETWORK DESIGN PROGRAMMING

OPL CPLEX Edition

Peter Cacioppi & Michael Watson

About the Authors

Michael Watson is a partner at Opex Analytics (http://opexanalytics.com/), an adjunct professor at Northwestern in the Master in Engineering Management and Master in Analytics program, and the co-author of the books *Supply Chain Network Design* (http://networkdesignbook.com/) and *Managerial Analytics* (http://managerialanalytics.com/)

Peter Cacioppi is self-employed and blogs at http://petercacioppi.com/. He is currently doing research to help bring about greater use of bi-objective optimization.

Acknowledgments

We would like to thank Filippo Focacci, CEO of DecisionBrain, for reviewing an early draft and providing feedback.

Also, thanks to Jennifer Hornsby, our editor, for cleaning up the text and making it easier to read.

1. Introduction to the Tutorial and Additional Resources

This book is meant to help you get started writing mathematical models in a commercial modeling language, IBM ILOG CPLEX Studio 12.4. Learning a modeling language gives you the flexibility to create your own industrial-strength models and helps you understand the field of optimization better.

Beginners to modeling languages often find it difficult to get started. Other books or tutorials assume a base level of knowledge of modeling languages or a deep familiarity with some certain optimization problems. Other books on optimization tend to ignore the computer science aspects of writing robust models.

We take a different approach in this book.

This book starts at the very beginning and teaches you how to start from nothing and get models up and running. We stick with one general problem and expand on that model in simple steps. What's more, we discuss how you can create industrial-strength optimization models by borrowing ideas from the field of computer science.

The problem we will focus on is facility location. We wrote this book as a complement to the book *Supply Chain Network Design* (*SCND*). In *SCND*, we build up more and more sophisticated mathematical formulations for a facility location problem. This book follows Chapter 3 to Chapter 10 of *SCND*. You should use *SCND* as an additional reference guide to get more context and explanation of the problem. Although this is optional, it will help you better understand why you are doing what you are doing in CPLEX Studio; you'll learn CPLEX Studio with a problem you are familiar with.

Supply Chain Network Design also works with IBM's ILOG LogicNet Plus XE (available through IBM's academic initiative). LogicNet Plus is a commercial network design tool. As such it includes a nice user interface as well as reporting. The mathematical formulas are in the background and the user directly interacts with them through the input forms. You can complement your learning of CPEX Studio by looking at the same models in LogicNet Plus XE. This also gives you an idea for the types of applications you can build with a solver like CPLEX in the background.

Although, this book discusses network design, this is just one use of CPLEX Studio. With CPLEX Studio, you can model an unlimited number of problems from airline crew scheduling, to financial portfolio optimization, to oil blending, to ocean vessel scheduling, and many more. We think that by learning CPLEX Studio with one particular problem, it will be easier to learn how to build other types of models.

This book is not meant to be comprehensive and cover every keyword, command, and technique. However, it is meant to provide you with a very solid starting point. It takes you from not knowing how CPLEX Studio works to preparing to build industrial strength models. Once you cover this, you will be in a much better position to understand other books and manuals. You will also be in a much better position to bring the best ideas from computer science into your projects.

Resources You Can Use with This Book

The only way to learn to write mathematical models is to do it. You should download and install CPLEX Studio. You should also have a copy of Excel on your machine, as well as Access or another database you can work with.

If you are using *SCND* with this book, you should download and install LogicNet Plus.

CPLEX Studio and LogicNet Plus are available for academic use at no charge through IBM's Academic Initiative. They are also commercially available.

You will hear different terms for CPLEX Studio. The official name of the product that you download and install is IBM ILOG CPLEX Studio. Within this product, you will write your model in the modeling language called OPL (Optimization Programming Language). When you are in the tool, the interface is called the IDE (Integrated Development Environment). The IDE allows you to open windows where you can write models, and you write the models using the language OPL. The IDE also contains utilities for file management, information on the models, reporting, and other useful functions. When you hit the Run button, the model compiles and is sent to an optimization engine for solving. This engine is called CPLEX (just CPLEX). CPLEX does the linear or integer programming calculations. It can actually do more calculations, but that is beyond this tutorial.

Here are some other useful resources.

- Since this tutorial uses an Excel model from the *SCND* book, you should use a more powerful Excel add-in solver, such as the free OpenSolver; see http://optimizationandanalytics.wordpress.com/2012/10/26/opensolver/
- The CPLEX Studio online help is comprehensive, with some handy examples and tutorials: (http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r4/index.jsp?topic=%2Filog.odms.studio.help%2FOptimization_Studio%2Ftopics%2FCOS_home.html). You should read this in parallel to this manual.
- The IBM Academic Initiative has some CPLEX Studio training material: http://www-03.ibm.com/ibm/university/academic/pub/page/ban_ilog_programming

To get the most out of this manual, you should go slowly through the exercises and build the models yourself. Through practice and building models, you will learn how to build optimization models.

2. Creating a New Model

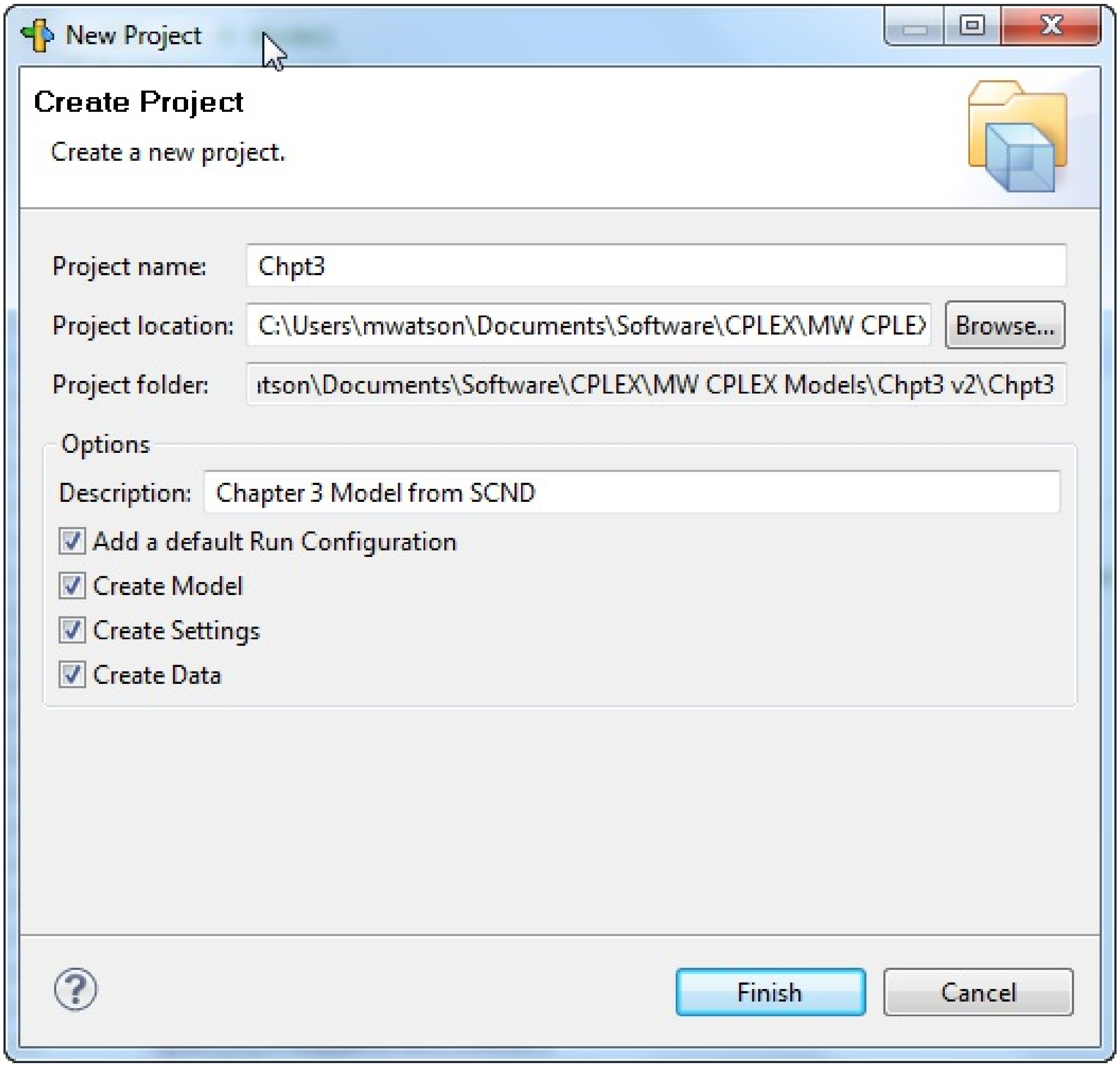
To start, we are going to implement the model from Chapter 3 of *Supply Chain Network Design (SCND)*. This model is also referred to as the P-median problem in other textbooks.

We will build a model that replicates the Excel 9-City Model from the *SCND* book, on pages 53 to 56. You will then be able to make sure your model in CPLEX Studio gives the same results as the Excel model. To do this, we will build the mathematical model seen on pages 48 to 52 of *SCND*.

When you open CPLEX Studio, the interface you see is called the IDE (Integrated Development Environment). The IDE allows you to develop, run, and analyze models.

When building models, one of the first things you will notice is that CPLEX Studio separates the model from the data. We build the model without any reference to the data. Once we have the model built, we can swap different data sets without changing the model. When you build a model with Excel, the data and the model are tied directly to each other. This can make it difficult to change the size of the data files.

To create a new project, go to *File* | *New* | *OPL Project*. You will get a screen like the one in Figure 1. You need to name your project and put it in a folder. Also, we've checked all the options. Don't worry about these yet. We'll explain them later.


New Project

Create Project

Create a new project.

Project name: Chpt3

Project location: C:\Users\mwatson\Documents\Software\CPLEX\MW CPLE

Project folder: tson\Documents\Software\CPLEX\MW CPLEX Models\Chpt3 v2\Chpt3

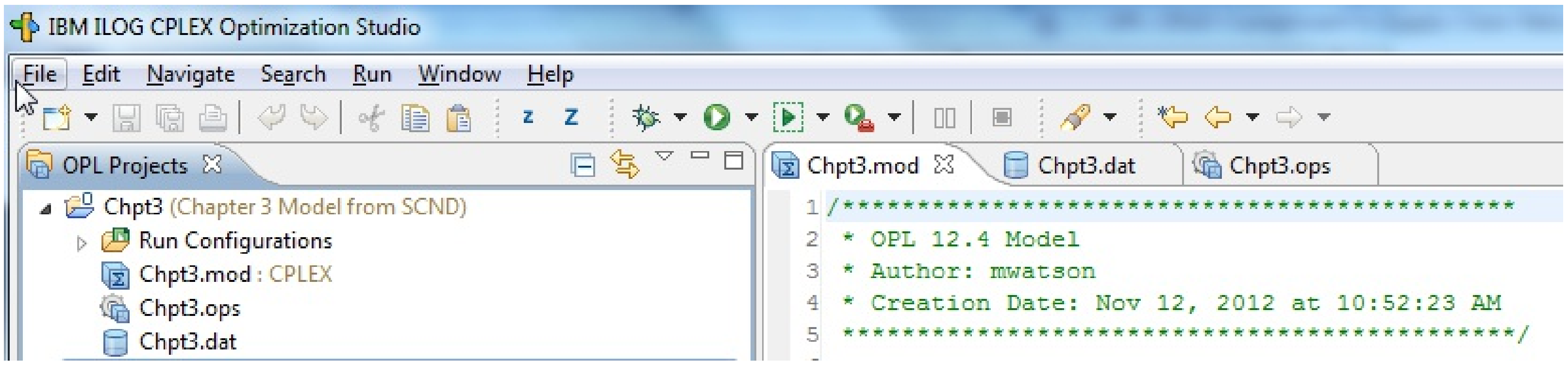
Options

Description: Chapter 3 Model from SCND

- [x] Add a default Run Configuration
- [x] Create Model
- [x] Create Settings
- [x] Create Data

Figure 1: Create Project

Once you click Finish, you'll see that your first project has been created. This is shown in Figure 2. On the left, in the OPL Projects tab, look for your project folder with various files. The file with the **.mod** extension is where you build your model. Don't worry about the others for now. On the right, you will see the workspace with the .mod file opened. CPLEX Studio automatically creates comments at the top of this file.


IBM ILOG CPLEX Optimization Studio

Chpt3 (Chapter 3 Model from SCND)
- Run Configurations
- Chpt3.mod : CPLEX
- Chpt3.ops
- Chpt3.dat

```
/*********************************************
 * OPL 12.4 Model
 * Author: mwatson
 * Creation Date: Nov 12, 2012 at 10:52:23 AM
 *********************************************/
```

Figure 2: Initial Project

The comments start with a /* and end with a */. The asterisks at the start of each of the lines are optional and just make the comments look nicer. It is always a good practice to add comments to help you remember what the model is meant to do.

3. Declaring the Variables

Building a model in OPL is similar to building models in other programming languages. You first declare the variables and then define functions for those variables. OPL is a language for building optimization models, so it's not surprising that it's structured to set up mathematical optimization problems. You can declare sets, input variables, and decision variables. These three different types of variables need to be declared in different ways. You then set up the objective function, followed by the constraints. This follows the same flow as pages 48 to 52 of *SCND*.

For a reference, here is the math formulation of the problem from SCND:

Minimize $$\sum_{i \in I} \sum_{j \in J} dist_{i,j} d_j Y_{i,j}$$

Subject to:

$$\sum_{i \in I} Y_{i,j} = 1; \forall j \in J$$

$$\sum_{i \in I} X_i = P$$

$$Y_{i,j} \le X_i; \forall i \in I, \forall j \in J$$

$$Y_{i,j} \in \{0,1\}; \forall i \in I, \forall j \in J$$

$$X_i \in \{0,1\}; \forall i \in I$$

First we create our sets and input variables. The basic variable types for sets and input variables are **`int`** for integer, **`boolean`** for either a 0 or 1, **`float`** for real numbers, and **`string`** for text.

We create sets for the customers and the warehouses. To create a set, we designate what type of variable it is, put the variable type in curly brackets, { }, name the set, and specify what is in the set. At the end of each statement, OPL is expecting a ; (semicolon). To do this, we type:

```
{string} Warehouses =…; // this is I in the book and called a facility
{string} Customers =…; // this is J in the book
```

The curly brackets tell OPL that this is a set of string values. We are going to use the names of our customers and warehouses as the elements in the set. We've given the sets readable names, Warehouses and Customers, and instead of specifying what is in the set, we used ellipsis points (the three dots). Ellipsis points tell OPL we are going to fill the values for the sets in another file.

You can see that we could have put in all the elements of the set right here in the model. However, this would kill the value of a programming language. We want to separate the model from the data, so changes to data will not affect the model.

Also, note that after the semicolon, we've added some comments to remind us what is in our model.

Next we define the maximum number of facilities. This is P in the *SCND* book. We will give this a name that is easier to read. This is not a set, but an input value, and we define this as:

```
int MaxWarehousesP =…; // this is P in the book on page 49
```

Here we declare this variable as an integer and provide the name of the variable. Since this isn't a set, we don't need to use the curly brackets. Again, we use ellipsis points to indicate that we'll define this somewhere else.

Next we have two input variables that are indexed by the sets. We have the demand for each customer. In the book this is d_j. We also have the distance from warehouse i to j, which is $dist_{i,j}$ in the book. Now you will see the value of declaring the sets. We define these two variables as:

```
float Demand[Customers] =…; // this is d in the book
float Distance[Warehouses][Customers]=…; // this is dist in the book
```

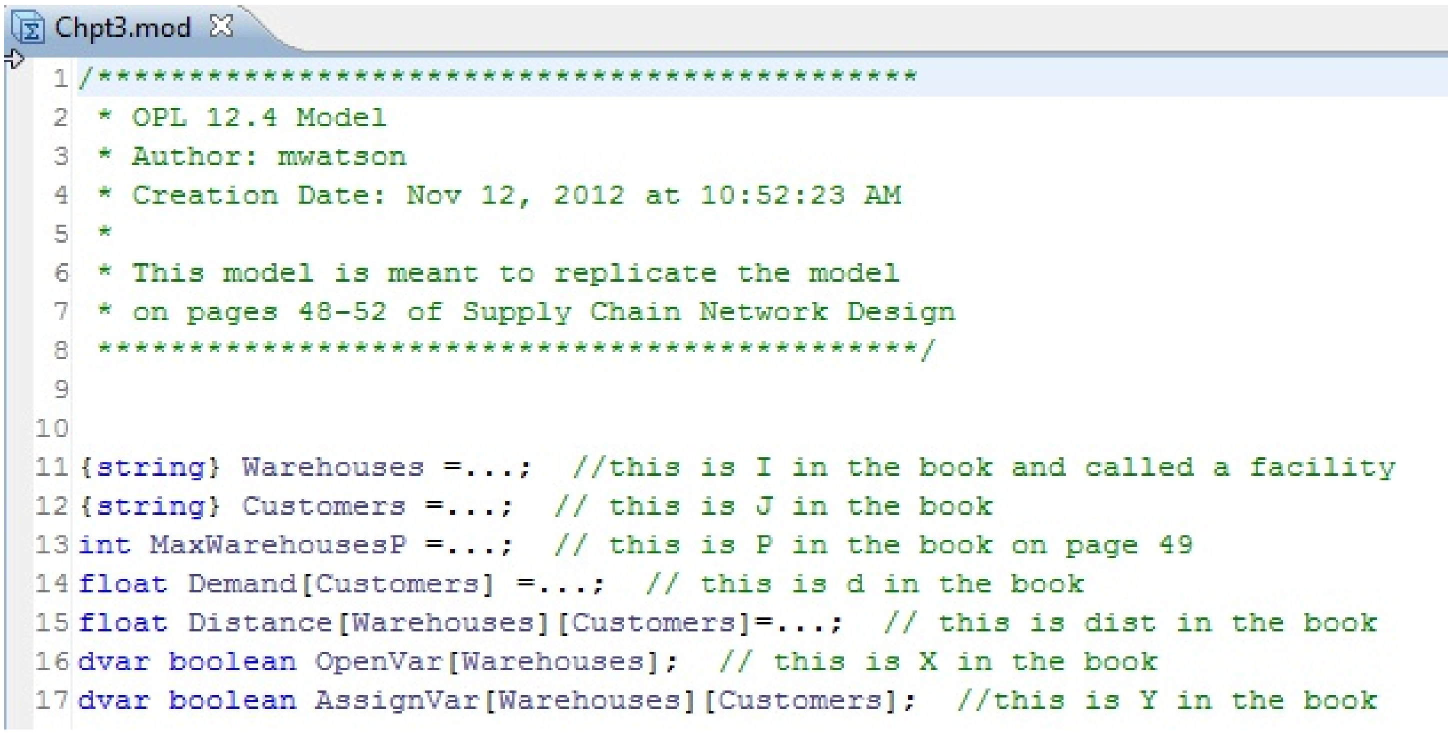
To index a variable name in OPL, you simply put the set name in brackets. If you have more than one index, you can simply add another set of brackets. As pointed out in the book, we will use the variable name *Distance* to actually refer to a family of variables—there is really one variable for every source and destination combination. But you can see that this notation allows us to write this family compactly.

All the input variables have been defined, and we need to define the two decision variables. To do this, we use the OPL designated term **`dvar`** to indicate that this is a decision variable. We do this as follows:

```
dvar boolean OpenVar[Warehouses]; // this is X in the book
dvar boolean AssignVar[Warehouses][Customers]; //this is Y in the book
```

We added the term **`dvar`** to indicate a decision variable, we've set both of these variables to **`boolean`**, and we've named the variables *OpenVar* (and specified that this is indexed over the set of warehouses—we have one decision for each warehouse) and *AssignVar* (and specified that this is indexed over both warehouses and customers). You will notice that we no longer need to specify the values for these variables—that is there is no "=" (equal to) sign after the names. Also, note that we've given these variables readable names and added the term *Var* to the end to designate that they are variables that will not have pre-assigned values. This will help make our code more understandable. *OpenVar* and *AssignVar* are decision variables, so the optimization engine will determine the values for these variables and return them as outputs.

Your workspace for Chpt3.mod should look like Figure 3.



```
/*********************************************
 * OPL 12.4 Model
 * Author: mwatson
 * Creation Date: Nov 12, 2012 at 10:52:23 AM
 *
 * This model is meant to replicate the model
 * on pages 48-52 of Supply Chain Network Design
 *********************************************/


{string} Warehouses =...;  //this is I in the book and called a facility
{string} Customers =...;  // this is J in the book
int MaxWarehousesP =...;  // this is P in the book on page 49
float Demand[Customers] =...;  // this is d in the book
float Distance[Warehouses][Customers]=...;  // this is dist in the book
dvar boolean OpenVar[Warehouses];  // this is X in the book
dvar boolean AssignVar[Warehouses][Customers];  //this is Y in the book
```

Figure 3: Declaration of Variables

4. Setting Up the Objective Function and Constraints

Next we will set up the objective function and constraints for the model. These formulas are highlighted on pages 50 to 52 of *SCND*.

The objective function is executed as a single command. That is, it starts with the keyword of either **`minimize`** or **`maximize`**, includes the formulas, and ends with a semicolon (`;`). Of course, the formulas can be complicated. This is often why you will write the objective function over several lines within the .mod file—to make it readable.

Many optimization problems involve summing a variable over all the elements in a set. In our example, we need to sum over all the customers in the set of *Customers* and all the warehouses in the set of *Warehouses*. OPL handles this with the keyword **`sum`**.

There are many ways to specify the values that OPL will add together with the **sum** function. We are going to use just one such way. Here is how we can set up the objective function.

```
minimize
  sum(w in Warehouses, c in Customers)
  Distance[w][c]*Demand[c]*AssignVar[w][c];
```

Within the parentheses of the **`sum`** function, we specify two new local variables, *w* and *c*. The local variables of *w* and *c*, when defined like this, are only valid for this function—that is, they are only valid until we hit the semicolon. We are specifying that these local variables are **`in`** the sets of Warehouses and Customers respectively. This means that we want to sum over all the warehouses and customers. You can see that we index our *Distance*, *Demand*, and *AssignVar* variables with the *w* and *c* variables. OPL's **`sum`** function uses the local variable to say that we are going to sum over all the values in Warehouse and Customer sets. OPL has you explicitly specify this because you may not want to sum over every element in the set. This syntax gives you flexibility.

Note that there is just a single semicolon at the end of this block of code—we have just a single objective function.

Now, when we model the constraints, we'll have multiple constraints. The constraints start with the key words **`subject to {…}`**. You place all the different constraints within the curly brackets—that is, you only close the curly brackets after you have specified all the constraints. Each constraint or constraint family ends with a semicolon.

We name each of the family of constraints. A family of constraints may be one formula or it may be specified for many (or every) element in a set. That is, just as we compactly wrote a family of constraints as a single line in the book *SCND*, we can do the same in OPL. In OPL, we do this with the keyword **`forall ()`**. Within the parentheses, we specify the individual constraints within the family. Again, this gives us flexibility.

Here are the constraints written in OPL:

```
subject to{

  forall ( c in Customers )
  ctEachCustomersDemandMustBeMet:
    sum( w in Warehouses )
        AssignVar[w][c]==1;

  ctOnlyUse_P_Warehouses:
    sum(w in Warehouses)
        OpenVar[w]==MaxWarehousesP;

  forall (w in Warehouses, c in Customers)
  ctCannotAssignCustomertoWH_UnlessItIsOpen:
  AssignVar[w][c] <= OpenVar[w];
}
```

If the constraint represents a family, it starts with the keyword **`forall`** followed by parentheses and is then followed by the user-defined name of the constraint (which is followed by a colon). Then, you specify the formula for the constraint. Finally, the semicolon tells OPL that the family is done.

In our example, we have three blocks of code for the three constraint equations. Two represent families of constraints and one is a single constraint.

The first block of constraints is a family with a unique constraint for every customer—we see this with the keyword **`forall`** and the local variable c, (*c* **`in`** *Customers*). Then, the constraint is given a name. Here we start the name of the constraint with the letters *ct* and then capitalize the first letter of every word to make it readable: *ctEachCustomersDemandMustBeMet*. After the name, we place a colon (**:**).

We then list the formula for our constraint. We use the same rules for writing the summations as we did in the objective. Unlike the objective, the constraint formula contains some comparison. The comparisons are **<** (less than), **<=** (less than or equal to), **>** (greater than), **>=** (greater than or equal to), and **==** (equal to). Note the two equal signs. In programming, a single equal to (**=**) signifies an assignment—meaning a variable is assigned a value. We use the double equal sign (`==`) to signify that the two sides of the equation must equal each other. Also note that you will typically use either `<=` or `>=` rather than the strict inequalities (`<` or `>`) because numerical rounding issues make the difference somewhat irrelevant.

In the first family of constraints, we have a unique constraint for each customer. We then make sure the sum over all the warehouses for the *AssignVar* variable equals 1. This ensures each customer is served by exactly one warehouse.

The second constraint equation is just a single constraint. This starts with the name of the constraint and then sums up the total number of warehouses (*OpenVar*) to make sure we have exactly the right number picked (*MaxWarehousesP*).

The third family of constraints specifies a unique constraint for every warehouse and customer. The formula then specifies that an assignment cannot be made unless the warehouse is open.

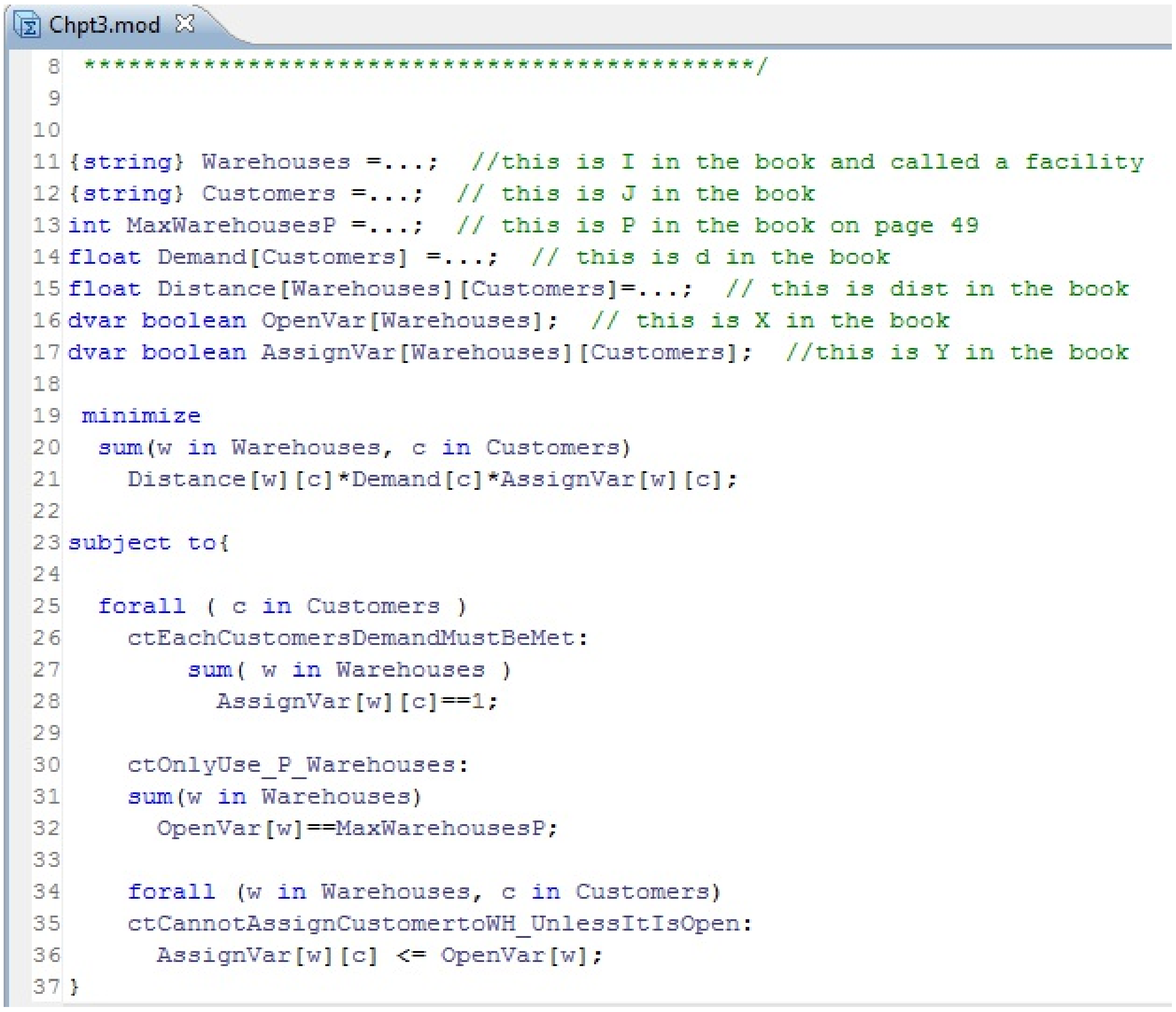
These first three constraint equations correspond, in the same order, as the first three constraints of the formulation at the top of page 52 of *SCND*. You can see that OPL was written to allow you to naturally translate the formal mathematical formulation.

You also will notice that the last two constraints on page 52 of *SCND* are not listed in our set of constraints. These constraints specify that the decision variables must be either 0 or 1 (boolean). We took care of this when we specified the variables.

Once you’ve completed the model, be sure to click on the Save button in the upper left.

5. Full Model and Debugging Capabilities

We now have a complete model file, as shown in Figure 4.

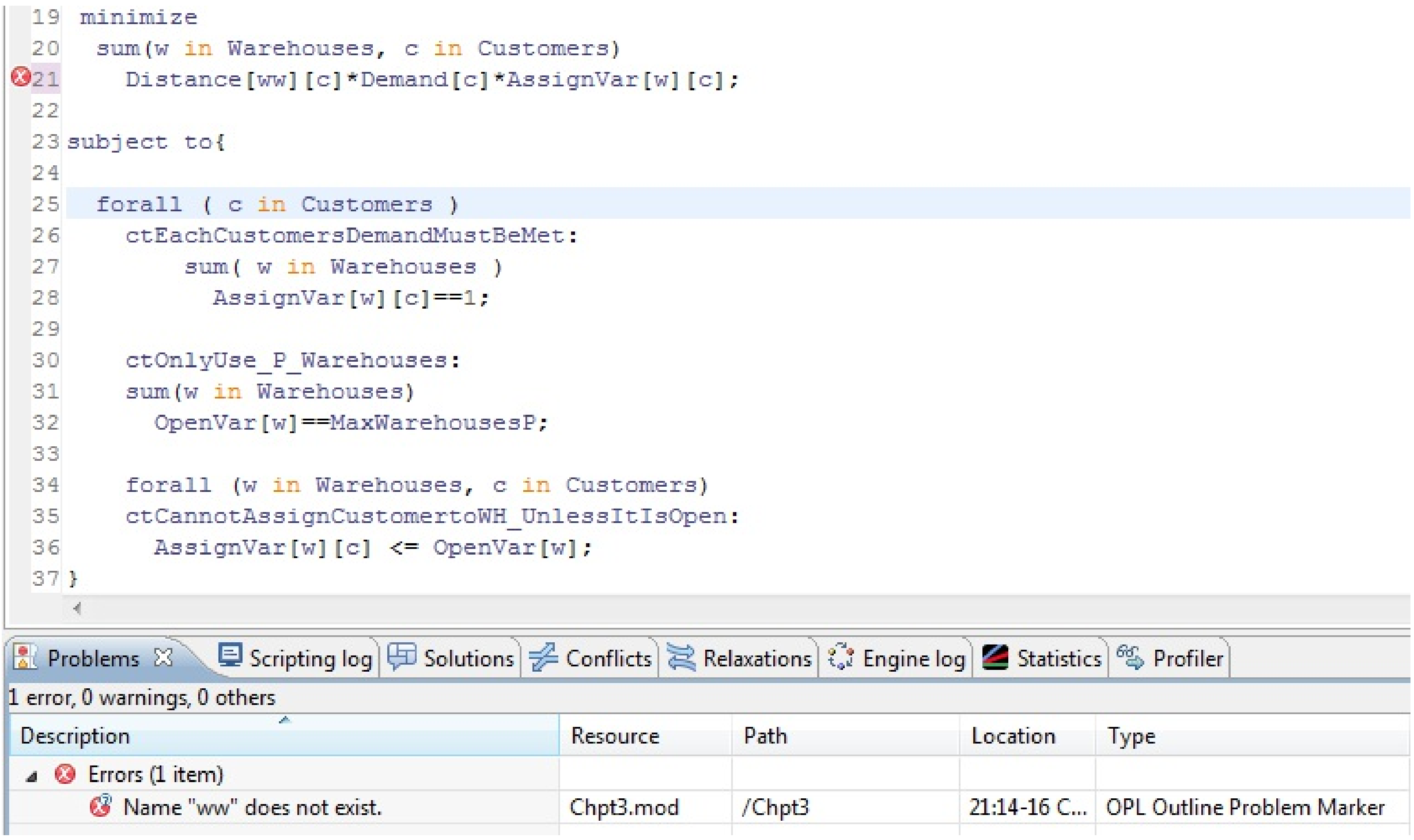


Chpt3.mod

```
 8  *********************************************/
 9
10
11 {string} Warehouses =...;  //this is I in the book and called a facility
12 {string} Customers =...;  // this is J in the book
13 int MaxWarehousesP =...;  // this is P in the book on page 49
14 float Demand[Customers] =...;  // this is d in the book
15 float Distance[Warehouses][Customers]=...;  // this is dist in the book
16 dvar boolean OpenVar[Warehouses];  // this is X in the book
17 dvar boolean AssignVar[Warehouses][Customers];  //this is Y in the book
18
19  minimize
20   sum(w in Warehouses, c in Customers)
21     Distance[w][c]*Demand[c]*AssignVar[w][c];
22
23 subject to{
24
25   forall ( c in Customers )
26     ctEachCustomersDemandMustBeMet:
27         sum( w in Warehouses )
28           AssignVar[w][c]==1;
29
30     ctOnlyUse_P_Warehouses:
31     sum(w in Warehouses)
32       OpenVar[w]==MaxWarehousesP;
33
34     forall (w in Warehouses, c in Customers)
35     ctCannotAssignCustomertoWH_UnlessItIsOpen:
36       AssignVar[w][c] <= OpenVar[w];
37 }
```

Figure 4: Full Chapter 3 Model

Since it is nearly impossible to write a model without making a mistake, you become familiar with OPL's debugging capability. Just below the workspace where you wrote the model, you see a series of tabs. The Problems tab shows any issues with the model. The model shown in Figure 4 should show no problems. But if you change the first *w* in line 21 to *ww*, you will see an error as shown in Figure 5. It marks the line with the error and provides a description below. In this case, it tells you that no such variable *ww* has been defined.



```
19 minimize
20   sum(w in Warehouses, c in Customers)
21     Distance[ww][c]*Demand[c]*AssignVar[w][c];
22
23 subject to{
24
25   forall ( c in Customers )
26     ctEachCustomersDemandMustBeMet:
27         sum( w in Warehouses )
28           AssignVar[w][c]==1;
29
30     ctOnlyUse_P_Warehouses:
31     sum(w in Warehouses)
32       OpenVar[w]==MaxWarehousesP;
33
34     forall (w in Warehouses, c in Customers)
35     ctCannotAssignCustomertoWH_UnlessItIsOpen:
36       AssignVar[w][c] <= OpenVar[w];
37 }
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
Name "ww" does not exist.	Chpt3.mod	/Chpt3	21:14-16 C...	OPL Outline Problem Marker

Figure 5: Sample Problem Message

Other single errors will lead to multiple problems being shown. For example, if you omit a semicolon (`;`), OPL will go to the next semicolon, and this could cause a cascade of problems. Try removing the semicolon (`;`) at the end of the objective function, line 21. You will then see a series of errors. Also, remove one of the curly brackets from the constraints, misspell the word "Customers" in line 25, and try creating other mistakes.

It is good practice to understand how to debug a model.

You should note that a model showing no syntax errors just means that the model has no OPL errors. You still need to confirm that the model does what you want it to do.

6. Creating Data for the Model—Using Simple Text Data

We have created the model in the .mod file. In that file, we used the ellipsis points (...) when we defined our input variables. This tells the model to go look in the .dat file for the values for these variables.

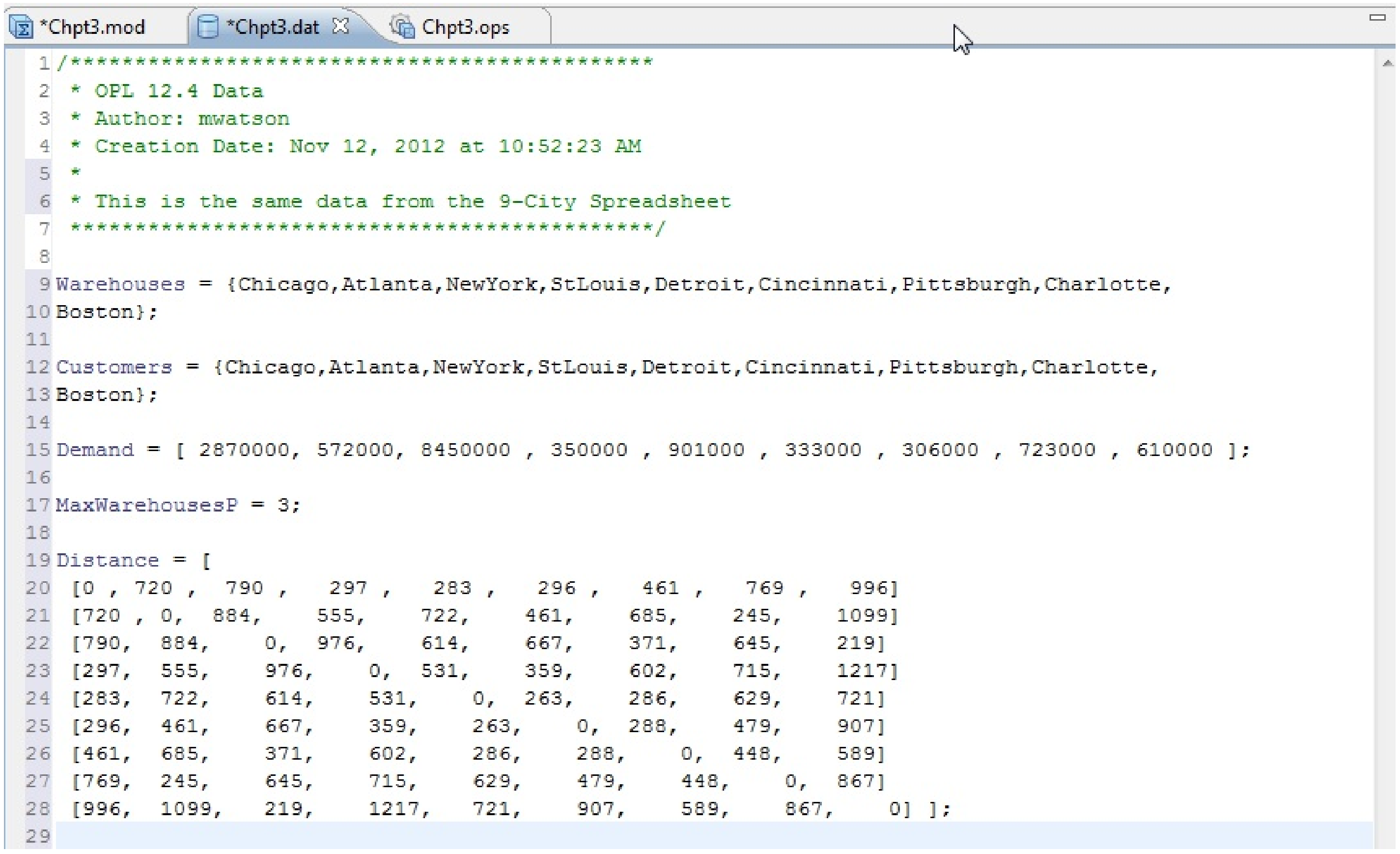
In the .dat file, we can directly put in the value of these variables. Later we'll show how the .dat file can read from other files.

The syntax for entering data directly into the .dat file is to state the variable name, use the equal to symbol (**=**), specify the data (use curly brackets for sets and brackets for arrays), and then close the statement with the semicolon (**;**).

The data is shown in Figure 6. You can see the two sets defined with the curly brackets, the demand array defined with a single set of brackets, the distance array defined with double brackets, and the *MaxWarehousesP* with no brackets.

Also, from a notation point of view, we removed the period from St. Louis and removed the spaces in the city names.

With this data, we now have what we need to run.



```
/**********************************************
 * OPL 12.4 Data
 * Author: mwatson
 * Creation Date: Nov 12, 2012 at 10:52:23 AM
 *
 * This is the same data from the 9-City Spreadsheet
 **********************************************/

Warehouses = {Chicago,Atlanta,NewYork,StLouis,Detroit,Cincinnati,Pittsburgh,Charlotte,
Boston};

Customers = {Chicago,Atlanta,NewYork,StLouis,Detroit,Cincinnati,Pittsburgh,Charlotte,
Boston};

Demand = [ 2870000, 572000, 8450000 , 350000 , 901000 , 333000 , 306000 , 723000 , 610000 ];

MaxWarehousesP = 3;

Distance = [
 [0 , 720 ,  790 ,   297 ,   283 ,   296 ,   461 ,   769 ,    996]
 [720 , 0,  884,    555,    722,    461,    685,    245,    1099]
 [790,  884,    0,  976,    614,    667,    371,    645,    219]
 [297,  555,    976,    0,  531,    359,    602,    715,    1217]
 [283,  722,    614,    531,    0,  263,    286,    629,    721]
 [296,  461,    667,    359,    263,    0,  288,    479,    907]
 [461,  685,    371,    602,    286,    288,    0,  448,    589]
 [769,  245,    645,    715,    629,    479,    448,    0,  867]
 [996,  1099,   219,    1217,   721,    907,    589,    867,    0] ];
```

Figure 6: Text Data File

Once you've entered the data, click on the save button in the upper left.

7. Creating a Run Configuration and Running the Model

To run a model in OPL, you need to set up a run configuration. When we created the model, a run configuration called Configuration1 was already set up for us, as seen in Figure 7.

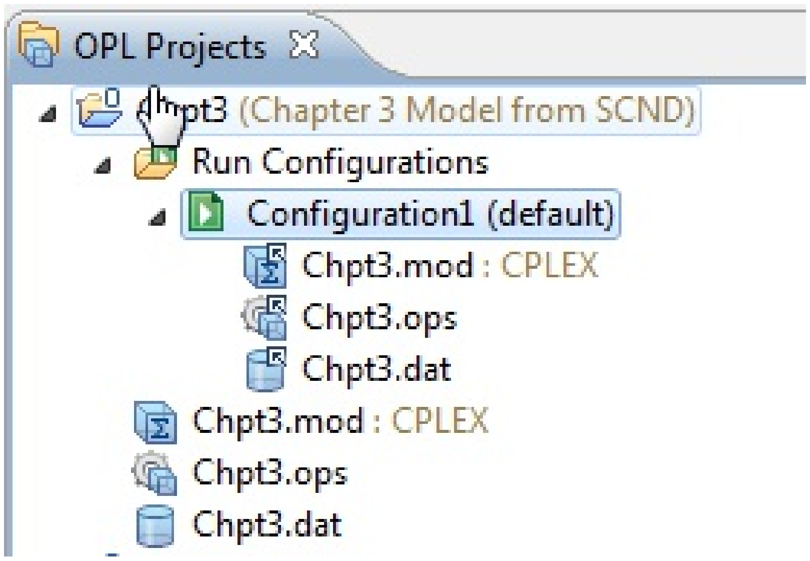


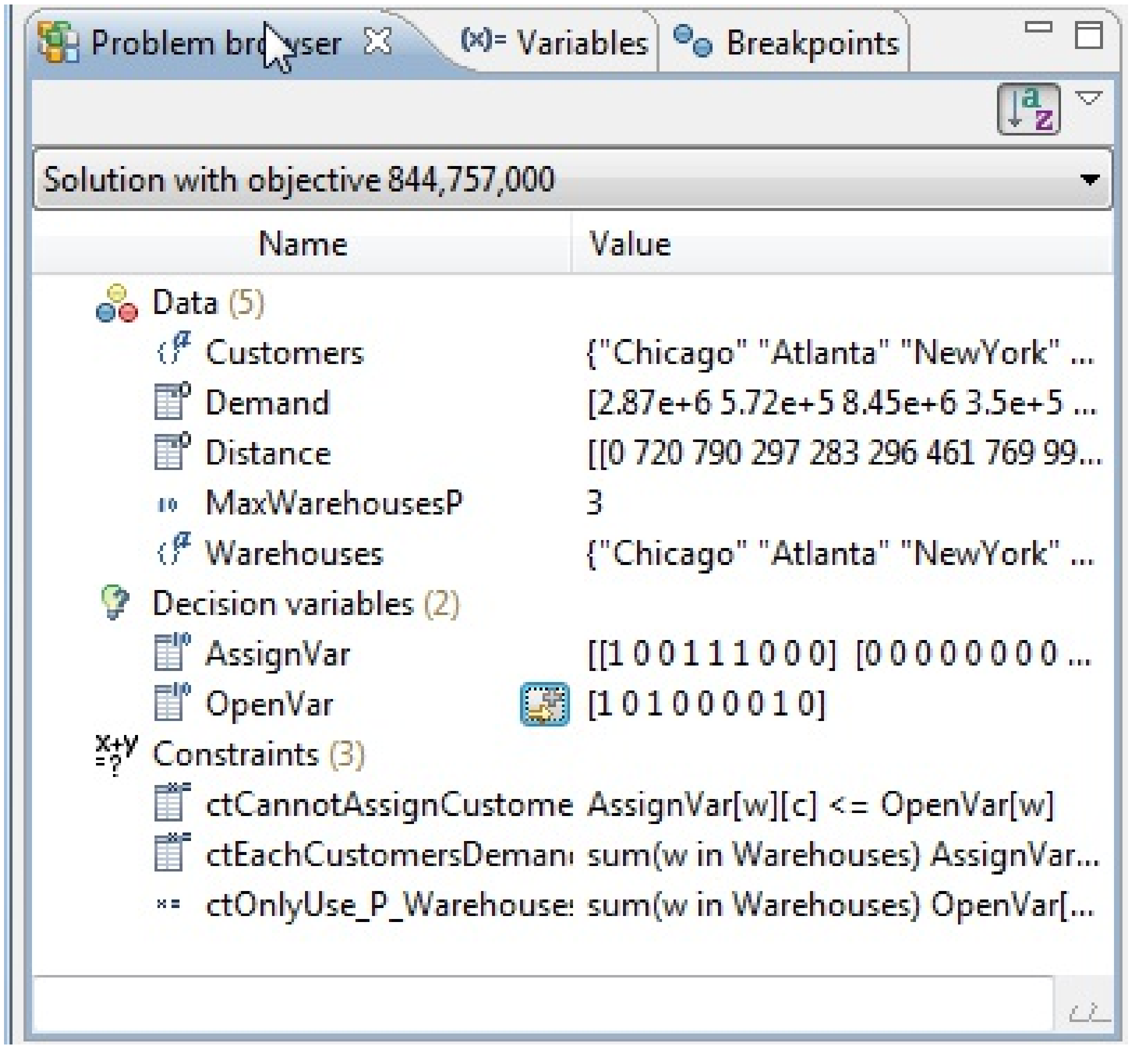
Figure 7: Default Run Configuration

You can see that the run configuration consists of our .mod and .dat files (we can ignore the .ops file for now). The little arrows on the .mod and .dat files indicate that these files just point to the main files. These files are not copies; we are just telling OPL to use the Chpt3.mod and Chpt3.dat files for this run configuration. You will later see how the run configurations give us flexibility. For now, let's see how to run the model.

Once your run configuration has the correct .mod and .dat file, you are ready to run.

To run, right-click on Configuration1 and then click Run This. You can watch the progress at the very bottom and right hand side of the window. This model should solve very fast.

When it is done solving, it populates the Problem Browser window in the bottom left corner, as shown in Figure 8. The value of the objective function is shown in the gray bar at the top. It is showing a value of 844,757,000. (You should confirm that this matches the answer from Excel.)

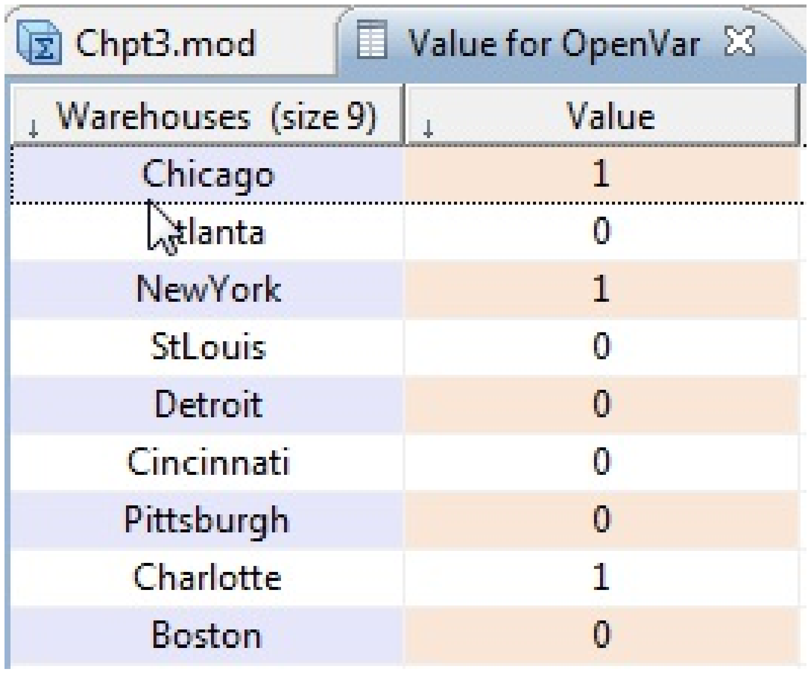


Solution with objective 844,757,000

Name	Value
Data (5)	
Customers	{"Chicago" "Atlanta" "NewYork" ...
Demand	[2.87e+6 5.72e+5 8.45e+6 3.5e+5 ...
Distance	[[0 720 790 297 283 296 461 769 99...
MaxWarehousesP	3
Warehouses	{"Chicago" "Atlanta" "NewYork" ...
Decision variables (2)	
AssignVar	[[1 0 0 1 1 1 0 0 0] [0 0 0 0 0 0 0 0 ...
OpenVar	[1 0 1 0 0 0 0 1 0]
Constraints (3)	
ctCannotAssignCustome	AssignVar[w][c] <= OpenVar[w]
ctEachCustomersDeman	sum(w in Warehouses) AssignVar...
ctOnlyUse_P_Warehouse	sum(w in Warehouses) OpenVar[...

Figure 8: Problem Browser

In the workspace, it shows the values it used for the input data, the results of the decision variables, and information on the constraints. If you highlight the *OpenVar* decision variable, you will see a button with an arrow and a plus sign. Click on that button, and it opens a more readable version of these values in the main worksheet, as seen in Figure 9. You can see that it opened Chicago, NewYork, and Charlotte. You can confirm that the Excel model did the same.



Warehouses (size 9)	Value
Chicago	1
Atlanta	0
NewYork	1
StLouis	0
Detroit	0
Cincinnati	0
Pittsburgh	0
Charlotte	1
Boston	0

Figure 9: Values for the OpenVar Decision Variable

If you click on the other decision variables, you will get more details on those values as well.

8. Running the Model with Data Pulled from Excel

Instead of recreating the data that was in Excel, let's go through an example of just reading directly from Excel. This is just one possible way to do this. We will keep it simple by copying the spreadsheet we were working with into the same directory as the model. In this case, we renamed the sheet MIP9City.xlsx and renamed the main tab 9City.

We first need to create a new .dat file. To do this, right-click on the Chpt3 folder, click New, and select Data. It will ask you to name this file. Let's call it Chpt3Data9CityExcel. Once you do this, you'll see this .dat file in your project directory.

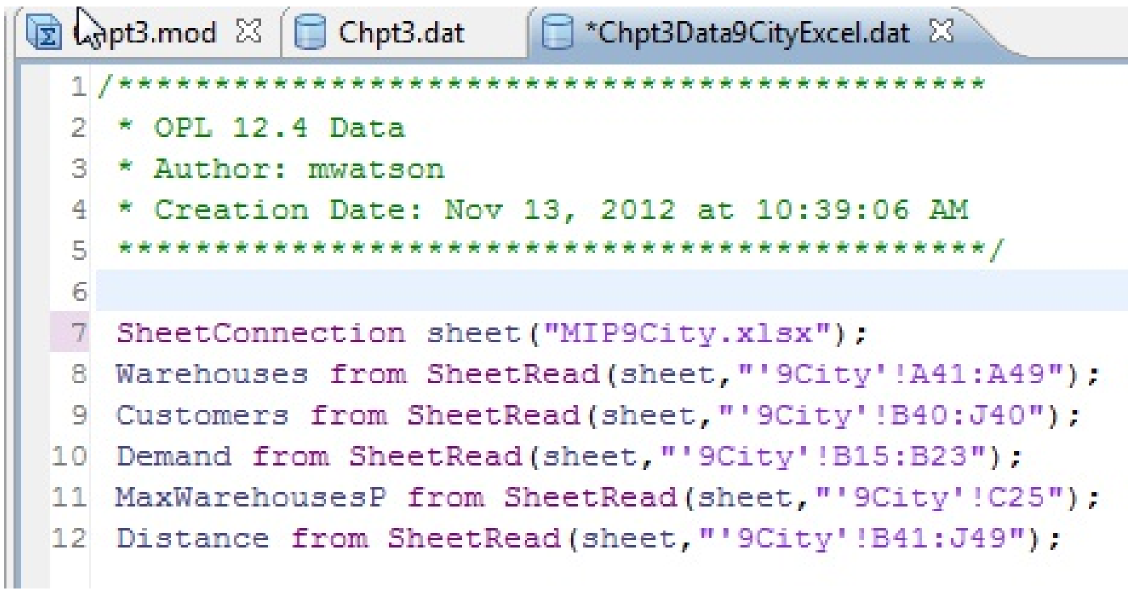
To read from Excel, we first need the **`SheetConnection`** command that connects to the spreadsheet. The **`SheetConnection`** command is followed by a local variable name (*sheet* in this case) and then the name of the Excel file in quotes. We need to end with a semicolon.

```
SheetConnection sheet("MIP9City.xlsb");
```

Once we have connected to the sheet, we then can define where we get the values for our variables. We do this with the **`from SheetRead()`** command. Within the parentheses, we define which Excel file to read with the variable *sheet* in this case, and then in double quotes, define the worksheet name (which is in single quotes if it has spaces) and the cell range. You can see this below.

```
Warehouses from SheetRead(sheet,"'9City'!A41:A49");
Customers from SheetRead(sheet,"'9City'!B40:J40");
Demand from SheetRead(sheet,"'9City'!B15:B23");
MaxWarehousesP from SheetRead(sheet,"'9City'!C25");
Distance from SheetRead(sheet,"'9City'!B41:J49");
```

Our new .dat file is shown in Figure 10.



```
/*********************************************
 * OPL 12.4 Data
 * Author: mwatson
 * Creation Date: Nov 13, 2012 at 10:39:06 AM
 *********************************************/

 SheetConnection sheet("MIP9City.xlsx");
 Warehouses from SheetRead(sheet,"'9City'!A41:A49");
 Customers from SheetRead(sheet,"'9City'!B40:J40");
 Demand from SheetRead(sheet,"'9City'!B15:B23");
 MaxWarehousesP from SheetRead(sheet,"'9City'!C25");
 Distance from SheetRead(sheet,"'9City'!B41:J49");
```

Figure 10: .dat File for Reading from Excel

Now we'll see the flexibility of the run configurations. Right-click on the Chpt3 folder, click New, and select Run Configurations. Name this run configuration 9CityExcel. You will see this new run configuration. Now drag the Chpt3.mod and Chpt3Data9CityExcel.dat files into this new run configuration. You will notice that the main files are still in the same place, but pointers to these files have popped up in the new run configuration, as seen in Figure 11. Now, right-click on 9CityExcel and select Run This. You will get the same answer as before.

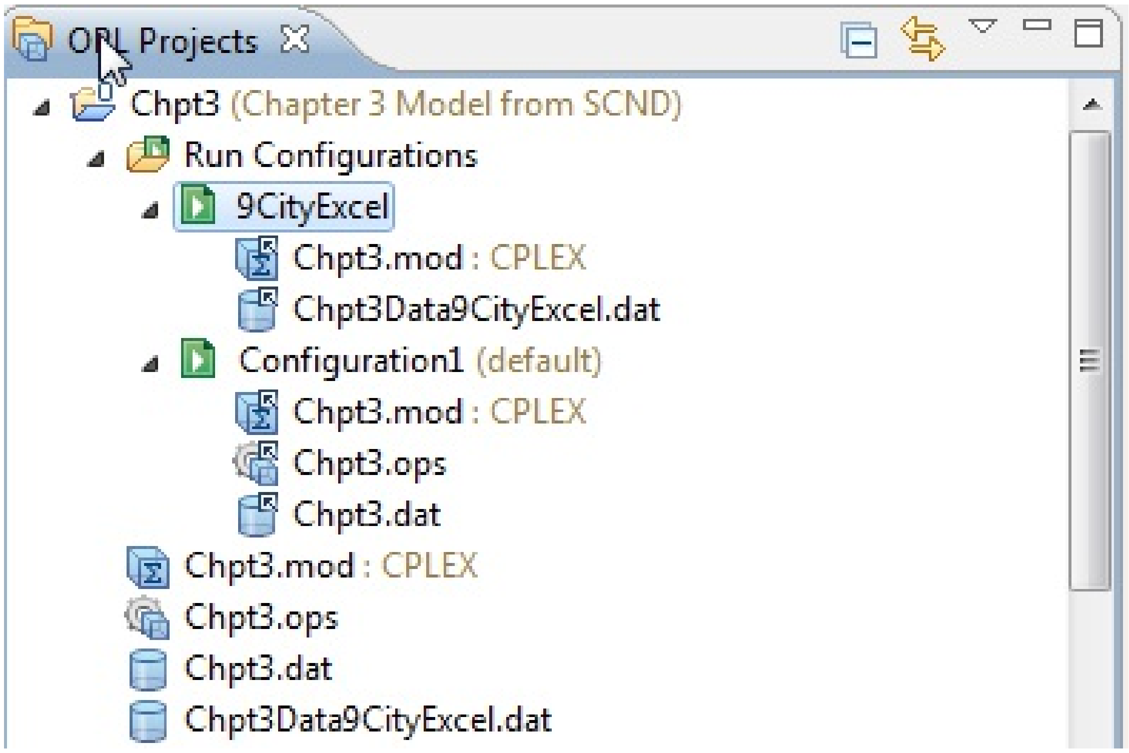


Figure 11: New Run Configuration

Besides seeing how to read from Excel, you can now understand the flexibility of the run configurations. You can create as many as you need and reuse different data sets, leaving original models and data intact. It is possible to create another .dat file that reads from a problem with any number of warehouses and customers without making any changes to your .mod files.

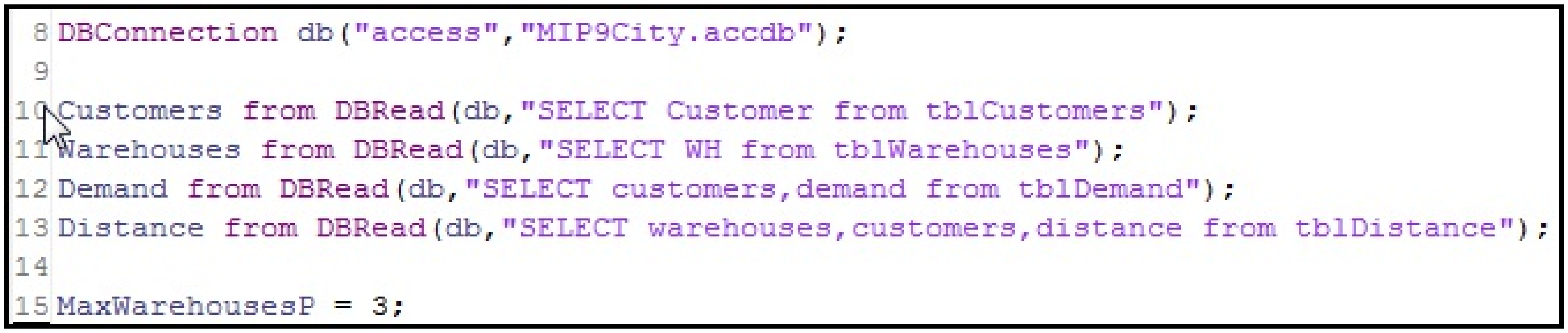
9. Reading from a Database

You can also read data from databases just like we read data from Excel. Likewise, as we'll show in the next section for Excel, you can write back to a database.

Storing data in a database can be much more efficient and flexible than doing it in Excel. For example, with Excel, you can only read two-dimensional tables. As soon as we add another index, we lose our ability to use Excel. (Some convoluted work-arounds do exist, but it is probably not worth the hassle in most cases.) When we read from Excel, we are hard-coding the size of the file we are reading. When reading from a database, we do not have to worry about that ahead of time.

We won't go into much detail on this topic. Rather, our goal in this book is to give you a basic introduction to OPL and a solid base to continue to develop your OPL or math modeling skills. Excel is more accessible, so we will mostly stick to that. We will, however, read from databases in some of the more advanced models at the back of this book, since they require triple-indexed variables.

As a quick introduction, Figure 12 shows you how to read from a database. The command **`DBConnection`** is the keyword that links to a database. We use the term **`from DBRead`** to read from a database, similar to the way read from Excel. Within the parentheses, we specify the database name (*db*, in this case) and use standard SQL commands to pull data. For clarity, when reading from an indexed variable like Demand or Distance, we will name the index fields and follow that up with the field we are reading. So, in the case of Demand, the SQL command tells us to find the correct customer field in the table called tblDemand. Once it finds this, it fills in the demand record with the column in the database called Demand. For Distance, we need to find the right warehouse-customer pair and then read the distance. Finally, to show that you can use the .dat file to read from multiple places, we are directly reading the MaxWarehousesP from the .dat file.



```
DBConnection db("access","MIP9City.accdb");

Customers from DBRead(db,"SELECT Customer from tblCustomers");
Warehouses from DBRead(db,"SELECT WH from tblWarehouses");
Demand from DBRead(db,"SELECT customers,demand from tblDemand");
Distance from DBRead(db,"SELECT warehouses,customers,distance from tblDistance");

MaxWarehousesP = 3;
```

Figure 12: Reading Data from a Database

For more details, explore CPLEX Studio reference manuals.

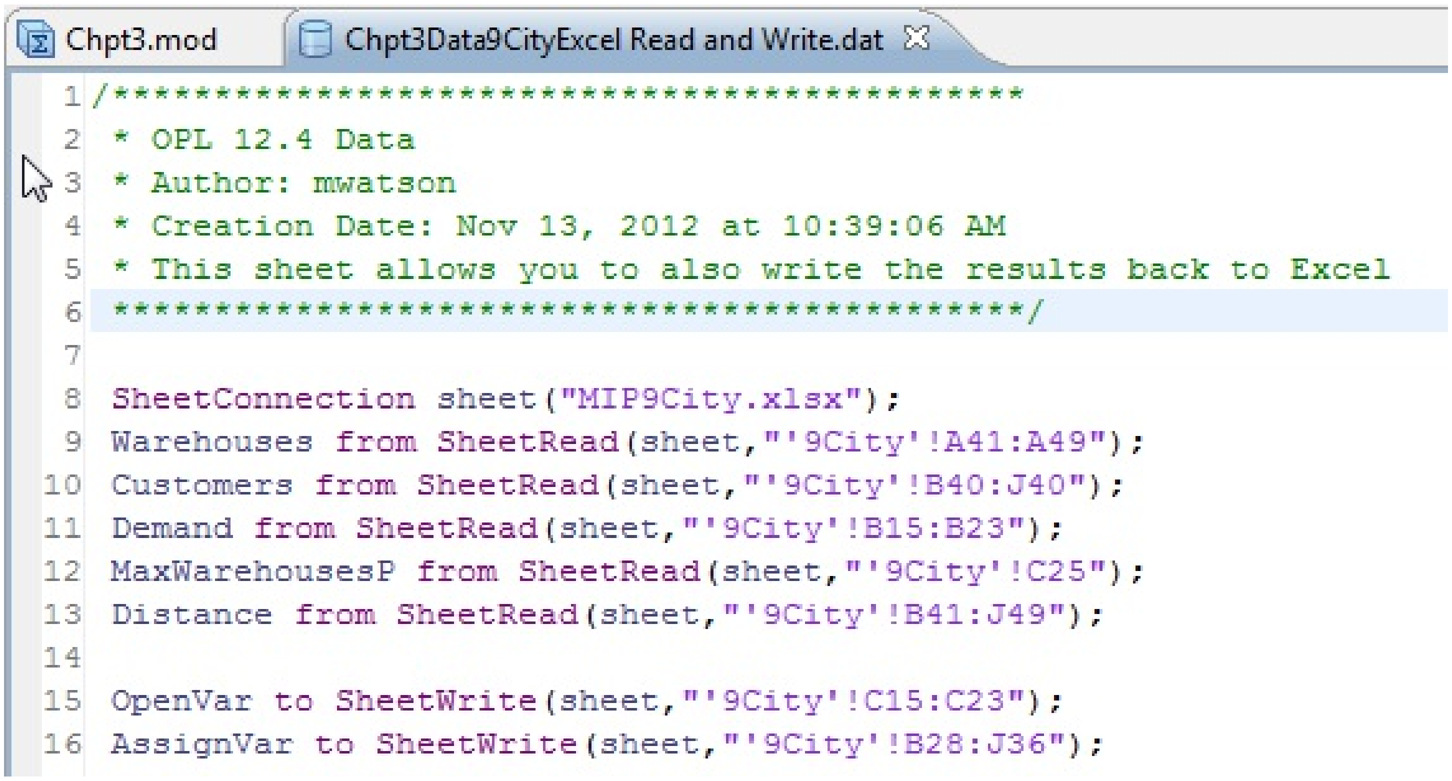
10. Writing the Data Back to Excel

You can also easily write the data back to Excel. In fact, you can embed CPLEX directly into Excel and not have to work within the CPLEX Studio's IDE interface. But for now we'll to continue to write the code within the IDE.

When you run the model, you see the results in the Problem Browser window. This window also showed the values of the decision variables. We've set up our Excel spreadsheet so it can automatically calculate the cost of the solution once the decision variables are entered.[1]

An easy way to write the decision variables back to Excel is to use the command **`to WriteSheet`**.

You can simply add this command to the .dat file. First, you want to copy your original .dat file (the one in Figure 10). The new command is seen in lines 15 and 16 of Figure 13.



```
 1 /*********************************************
 2  * OPL 12.4 Data
 3  * Author: mwatson
 4  * Creation Date: Nov 13, 2012 at 10:39:06 AM
 5  * This sheet allows you to also write the results back to Excel
 6  *********************************************/
 7 
 8  SheetConnection sheet("MIP9City.xlsx");
 9  Warehouses from SheetRead(sheet,"'9City'!A41:A49");
10  Customers from SheetRead(sheet,"'9City'!B40:J40");
11  Demand from SheetRead(sheet,"'9City'!B15:B23");
12  MaxWarehousesP from SheetRead(sheet,"'9City'!C25");
13  Distance from SheetRead(sheet,"'9City'!B41:J49");
14 
15  OpenVar to SheetWrite(sheet,"'9City'!C15:C23");
16  AssignVar to SheetWrite(sheet,"'9City'!B28:J36");
```

Figure 13: .dat file for Writing to Excel

You can see that the command **`to WriteSheet`** has the same format as **`from SheetRead`**.

We simply specify the variable we want to output and then in parentheses we specify the Excel File (in this case referenced by the variable "sheet") and the range.

The spreadsheet must be closed so CPLEX Studio can write to it.

We created a new run configuration to test it out. This is shown in Figure 14.

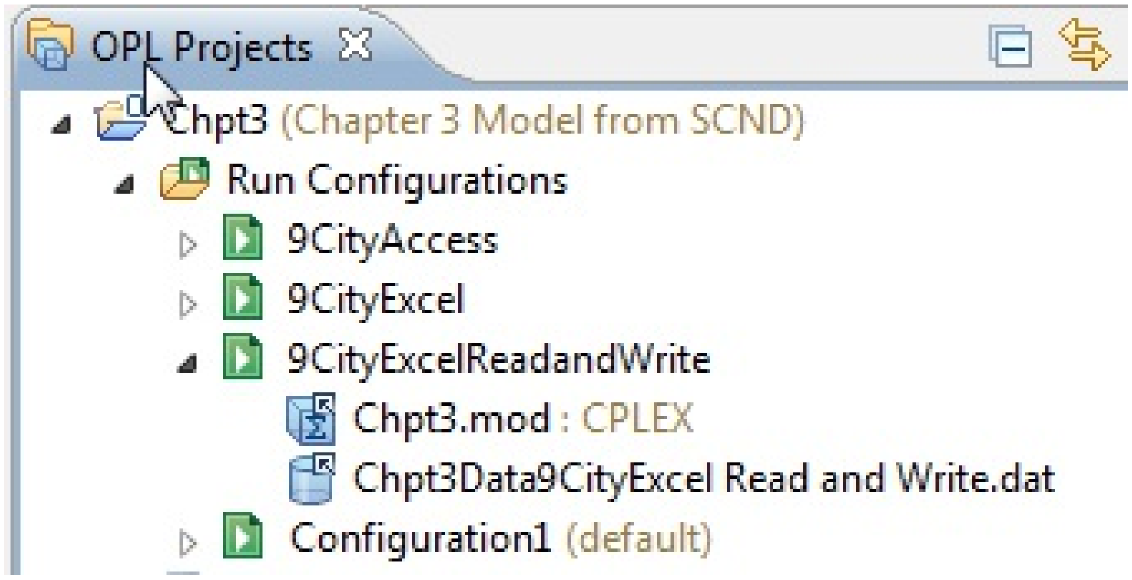


Figure 14: Run Configuration for Read and Write

[1] There are good reasons to double-check the calculations separately from the optimization layer. For more information, please see http://petercacioppi.com

11. Practice Exercise: Creating a Large-Scale Model for Chapter 3 of *SCND*

Up to this point, we have worked with the simple nine-city model. However, let's build a much larger-scale model to see how CPLEX can scale.

To test a larger model, open the spreadsheet that comes with this text called LargeScale3LocationModel.xlsm.

This is a model with 200 demand points with the possibility to locate a warehouse at each point. To keep things simple, we have just put these points on a grid. (You can use this text with IBM ILOG LogicNet Plus to put the points on a map.) Here are the highlights of the spreadsheet:

- The table starting at cell D10 has the following input values:
    - It lists 200 customer ID's in column D.
    - It shows the *X*,*Y* coordinates of the point in columns E and F (the graph is seen to the right of the table).
    - It shows the demand in column G.
- Column H of the table starting at D10 is the decision variable (X_i in Chapter 3 of *SCND*). It is a 1 if we pick this location and a 0 if not.
- Cell H211 shows the total warehouses picked. H212 is the constraint for how many warehouses are allowed. In this case, we have allowed for three warehouses. *Note:* This is a relatively simple spreadsheet and the graph only works for three warehouses. Feel free to expand on the Excel file to make it more flexible.
- Cell I211 is the objective function. Here we are multiplying the demand of each customer by the distance from the warehouse that serves the customer. We had to use an array function in Excel—hence the curly brackets and the need to hit CTRL+SHIFT+ENTER when working in the cell.
- Cell I212 is the distance-weighted average demand.
- Cells I214:I218 are used when we get to Chapter 4 and discuss alternative service levels. In this case, they are set up to report on the demand that is served within 250 miles of a warehouse and to restrict the average and maximum distance.
- Column J allows you to add capacity constraints (which will come in Chapter 5), and column K shows the total demand assigned to each warehouse.
- The graph to the right shows the three warehouses picked and color-codes the customers assigned to each warehouse. Again, the graph is only set up to work for three warehouses.
- Columns to the right of the graph are just used to create the graph.
- The buttons at the top allow you to clear the solution form the graph and to redraw the graph

after you run.

- The first matrix table from E220 to GV419 represents the $Y_{i,j}$ variables showing which customer is assigned to which warehouse. The rows represent warehouses, and the columns represent customers. *Note:*
    - The column to the right of the table shows the total number of customers assigned to each warehouse.
    - The first row below the table (row 420) shows the total warehouses each customer is assigned to. This needs to equal 1.
    - The second row below the table (row 421) shows the distance from the warehouse to the customer.
    - The third row (422) determines whether the assignment is within the high-service demand that we will discuss in Chapter 4.
- The second matrix table from E431 to GV630 is simply the distance from each point to every other point.
- The third matrix table from E635 to GV834 is the link constraint that says we can't use a warehouse unless it is open. Note:
    - The first column corresponds to the our decision variable in cells H11:H210.
    - The next columns just refer to the first column.
    - We needed to set this up this way so we could easily write our constraints in the Excel modeling language. This shows you some of the limitations of Excel. Maybe we were not clever enough, but without a table like this, we would have had to write many unique link constraints using Excel's solver. Doing it this way lets us simplify the Excel model.

As an exercise, you should set up and solve this model with OpenSolver (see Chapter 1 of this book in the section called "Resources You Can Use with This Book" for more information). This will be very similar to your setup of the nine-city model in Excel's solver.

When you do this, you will find that it takes about 20 minutes to solve the model. Most of this time is spent in setting up the variables and model. When building industrial-size models, there are potential bottlenecks in setting up the variable and models, actually running the solver, and creating tables for the output. In industrial applications, you need to watch all three of these areas.

OpenSolver should pick warehouses 72, 78, and 138 for a total distance of 591,087,305 and an average distance of 239.9. The graph of the solution is shown in Figure 15.

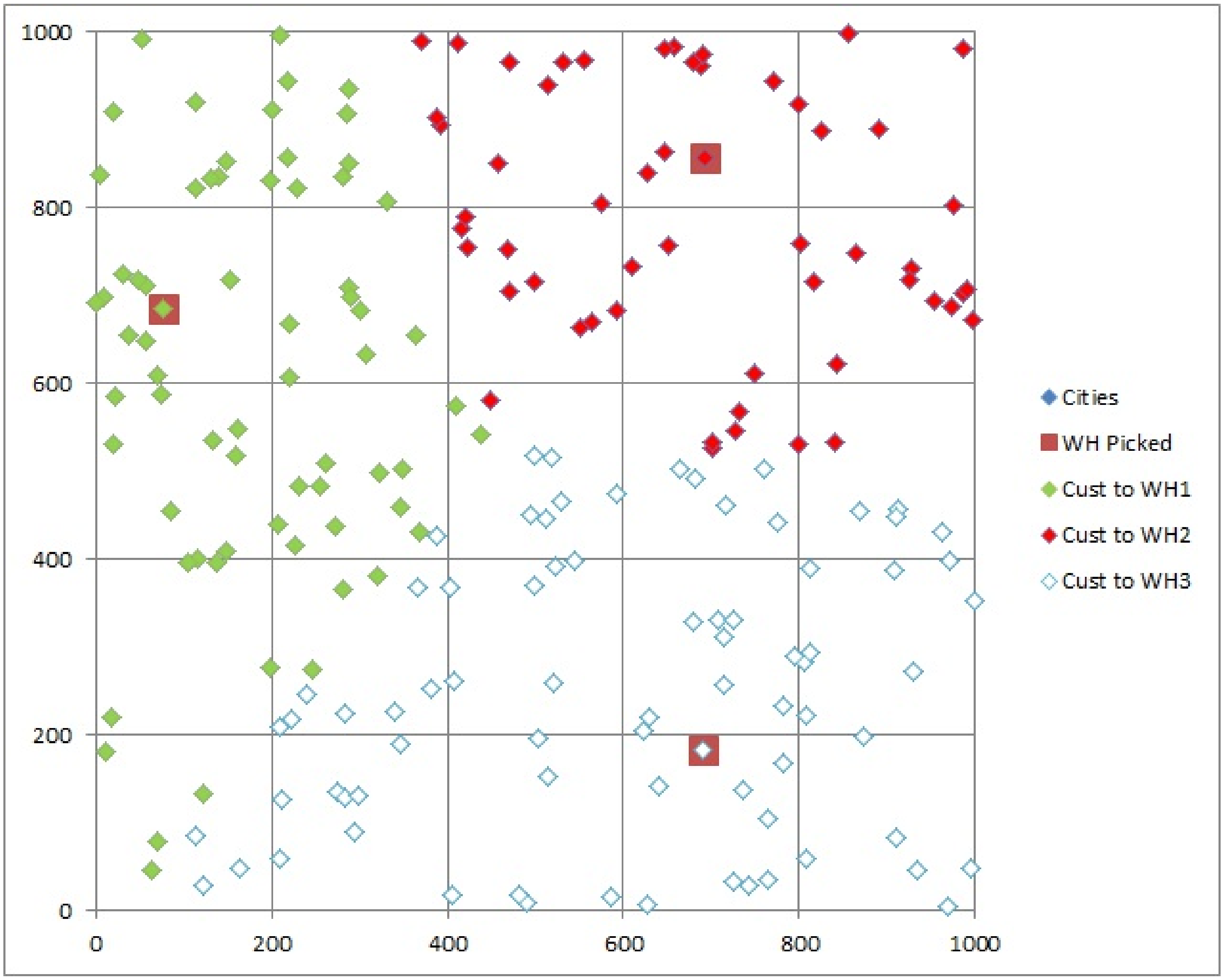


Figure 15: Three-City Large-Scale Model

Now you should run this model in CPLEX. To do this, you need to do the following:

- Create a new .dat file that points to the data in the new spreadsheet. Note that this .dat file is the same one shown in Figure 13, except that you change the spreadsheet name and the cells. Already you can see that the work you did earlier can quickly scale to other models. We did not have to change any of the structure.
- Create a new Run Configuration for this model.
- Add the new .dat and the original .mod file (the one shown in Figure 4). This is the beauty of separating the model from the data. The model is perfectly scalable. No changes are needed.
- Run the model.

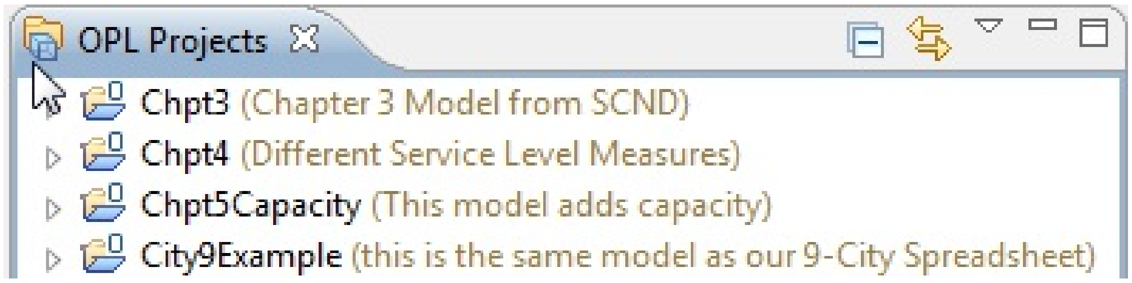
The model should now run in about 20 seconds. You can see the scalability of CPLEX. The data is easy to change, the model is the same, and the run time is fast.

12. Creating a New Project for Building SCND Chapter 4 Models

Moving on to Chapter 4 in *SCND*, we see that we have new definitions of service level and new constraints. We'll cover what you need to know within this book.

To build this model, let's first create a new project in CPLEX. This will give us a new folder and keep this work separate from the models for Chapter 3. To do this, click on File | New | OPL Project. Then you will get the screen seen in Figure 1. This time, don't check any of the boxes at the bottom. We don't need default models or data files.

Once you have created the new OPL project, you will see another folder in your workspace as shown in Figure 16.



Chpt3 (Chapter 3 Model from SCND)
Chpt4 (Different Service Level Measures)
Chpt5Capacity (This model adds capacity)
City9Example (this is the same model as our 9-City Spreadsheet)

Figure 16: View of OPL Projects

Instead of creating new .dat and new .mod files, you should copy these files from your Chapter 3 models. It will be much easier to modify these files than to create new ones.

Finally, you should copy the spreadsheet LargeScale3LocationModel.xlsm from its current location to the new folder you just created for Chapter 4. You do this outside of CPLEX with Windows Explorer. You might want to rename this file in case you want to make changes to it.

13. Changing the Objective Function to Maximize High Service Level

The model introduced in Chapter 4 of *SCND*, shown on pages 71 to 74, introduced a new objective function and two new constraints.

The objective function is set up to maximize the demand that is served within a specified distance from the warehouse. We will call the specified distance the *HighServiceDist*. That is, everything within this distance receives high service levels. For example, if we define 250 miles as our high service level threshold, every customer within this 250 miles receives this service level. You often see this with home delivery service. If you are within a certain distance from the pizza delivery store, you can receive your pizza within 30 minutes.

The constraints restrict the overall average distance (using the parameter *AvgServiceDist*) and the maximum distance to any single customer (using the parameter *MaximumDist*).

We first need to create a variable for the new parameters we introduced in this chapter. We do this in the .mod file and then read them from Excel in the .dat file. You need to modify the .mod file with these three lines:

```
float HighServiceDist =…; // this measures the threshold for high service
float AvgServiceDist =…; // this helps clean up solutions
float MaximumDist =…; // this is the maximum distance we'll allow
```

We can then fill in these values in the .dat file with these three lines:

```
HighServiceDist from SheetRead(sheet,"'City200'!I214");
AvgServiceDist from SheetRead(sheet,"'City200'!I217");
MaximumDist from SheetRead(sheet,"'City200'!I218");
```

The new objective function requires that we do a test to determine whether an assignment is within the *HighServiceDist*. If it is, this assignment helps improve the objective function. Note that the objective function has now changed to **`maximize`**; we want as much demand as possible within the HighServicDist.

We can determine prior to the optimization run whether a particular assignment will meet the criteria. We do this by comparing the distance between two points with the *HighServiceDist*. So, in this sense, this is an input.

However, one of the benefits of a modeling language like OPL is that it allows us to directly specify this condition with the data we have. Just as the book *SCND* uses concise mathematical language for this, we do the same in OPL. Our new objective function becomes:

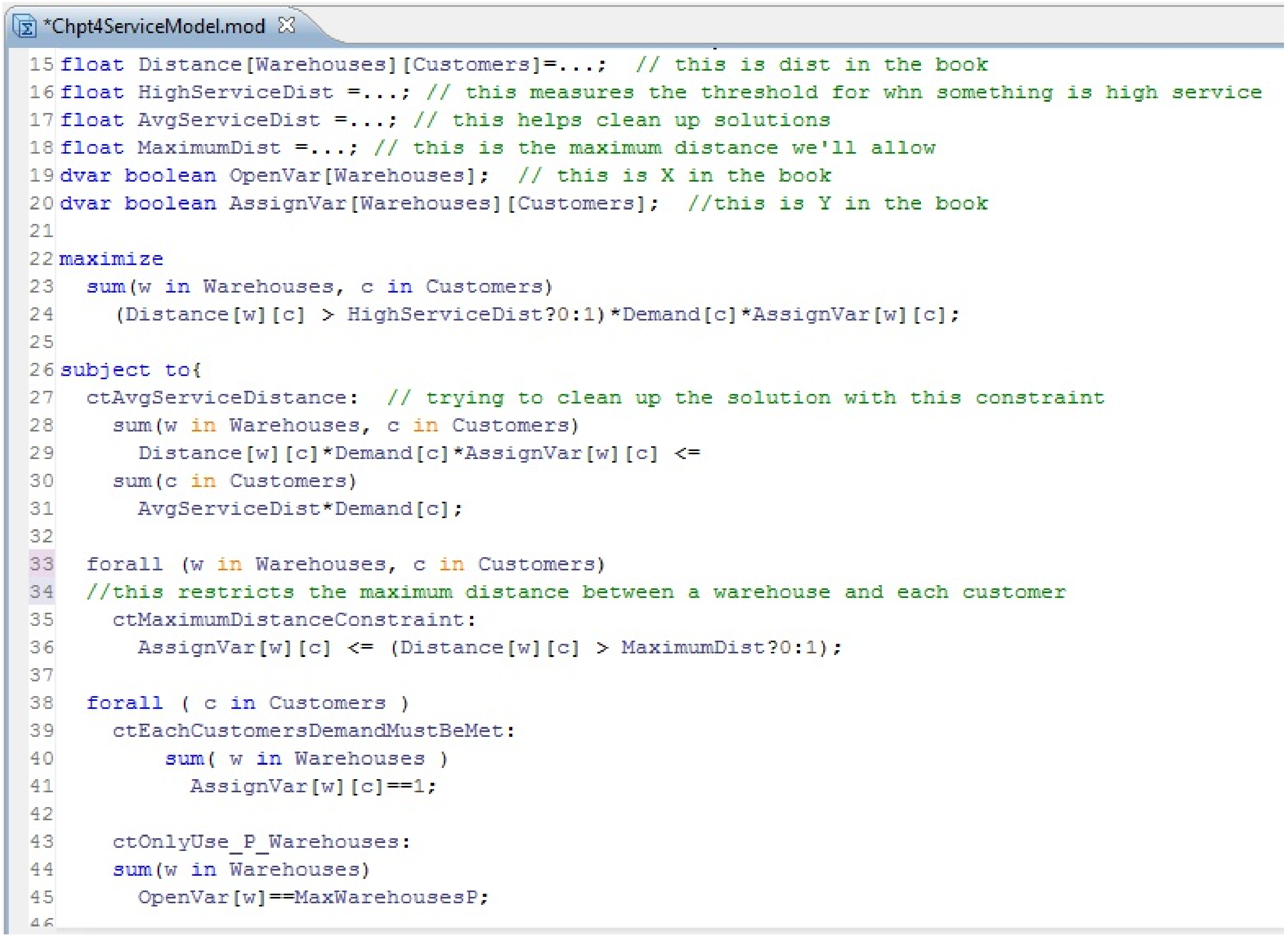
```
maximize
  sum(w in Warehouses, c in Customers)
  (Distance[w][c] > HighServiceDist?0:1)*Demand[c]*AssignVar[w][c];
```

We use the **`maximize`** keyword to start it off. The key difference is that we have added a conditional statement to the objective function. The parenthesis indicates that we have a formula to calculate. Here we are using the standard notation explained on page 71 of *SCND*.(Note that there are other ways to write this test besides the "?0:1" notation we used here.) We are testing whether the distance is greater than the high service threshold. If it is, we return a 0, and if not, we return a 1. This test runs efficiently because it is being performed purely with input data, and not with a decision variable.

The two constraints we need to add to match the first model on page 74, following notation we've used before:

```
ctAvgServiceDistance: // trying to clean up the solution with this
constraint
    sum(w in Warehouses, c in Customers)
        Distance[w][c]*Demand[c]*AssignVar[w][c] <=
    sum(c in Customers)
        AvgServiceDist*Demand[c];

forall (w in Warehouses, c in Customers) //this restricts the maximum
distance between a warehouse and each customer
ctMaximumDistanceConstraint:
AssignVar[w][c] <= (Distance[w][c] > MaximumDist?0:1);
```

Figure 17 shows the main changes to the .mod file.



```
float Distance[Warehouses][Customers]=...;  // this is dist in the book
float HighServiceDist =...; // this measures the threshold for whn something is high service
float AvgServiceDist =...; // this helps clean up solutions
float MaximumDist =...; // this is the maximum distance we'll allow
dvar boolean OpenVar[Warehouses];  // this is X in the book
dvar boolean AssignVar[Warehouses][Customers];  //this is Y in the book

maximize
  sum(w in Warehouses, c in Customers)
    (Distance[w][c] > HighServiceDist?0:1)*Demand[c]*AssignVar[w][c];

subject to{
  ctAvgServiceDistance:  // trying to clean up the solution with this constraint
    sum(w in Warehouses, c in Customers)
      Distance[w][c]*Demand[c]*AssignVar[w][c] <=
    sum(c in Customers)
      AvgServiceDist*Demand[c];

  forall (w in Warehouses, c in Customers)
  //this restricts the maximum distance between a warehouse and each customer
    ctMaximumDistanceConstraint:
      AssignVar[w][c] <= (Distance[w][c] > MaximumDist?0:1);

  forall ( c in Customers )
    ctEachCustomersDemandMustBeMet:
        sum( w in Warehouses )
          AssignVar[w][c]==1;

    ctOnlyUse_P_Warehouses:
    sum(w in Warehouses)
      OpenVar[w]==MaxWarehousesP;
```

Figure 17: Chapter 4 Max Service First Model

You can see the three new input variables, the new objective, and the two new constraints. Otherwise, everything else is the same. We did not have to make many changes to the file.

14. Running the High-Service Models and Cleaning Up the Results

We can now create a new run configuration, add the .mod and .dat files to the configuration, and run.

Your results should show the model picking warehouses 45, 70, and 73. The percentage of customers within 250 miles is 61.2%. This is up from 52.4% in the Chapter 3 model.

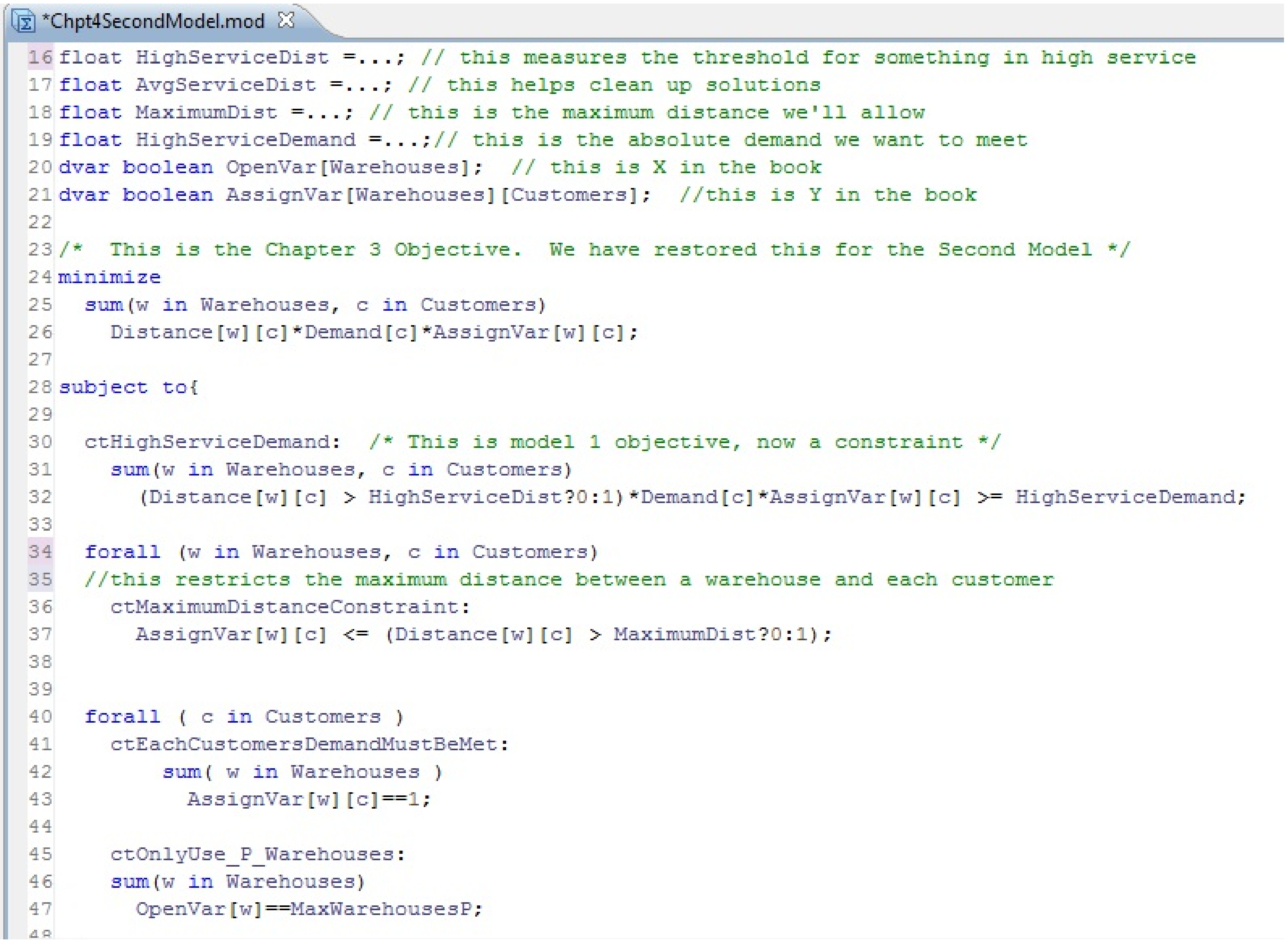
The average service level in this solution is 270.6 miles, up from 239.9. But, remember, we did not optimize for the average service level. Once a point was beyond 250 miles, there was nothing in the model to ensure that the point was assigned to the closet warehouse. Our two new constraints helped make sure the solution was reasonable, but there is a possibility that we could do better than 270.6 and still obtain the 61.2% high service level.

This problem can easily be solved with the "Second Model" on page 74 of *SCND*. In this model, we return to the objective function that minimizes average distance. We also apply a constraint that forces the model to have 61.2% of the demand served within the HighServiceDist. For convenience, we won't use the percentage; we'll use the absolute amount of demand, 1,506,800.

To make this model, you should create a copy of the .mod and .dat files.

Within the .mod file, we need to create a parameter for *HighServiceDemand* to ensure we meet 1,506,800 units of demand within 250 miles.

Then we need to bring back the original objective function and deactivate the max service objective function. We then need to copy the max service objective into the constraints and specify that it must be at least 1,506,800. Figure 18 shows the second model.



```
*Chpt4SecondModel.mod
16 float HighServiceDist =...; // this measures the threshold for something in high service
17 float AvgServiceDist =...; // this helps clean up solutions
18 float MaximumDist =...; // this is the maximum distance we'll allow
19 float HighServiceDemand =...;// this is the absolute demand we want to meet
20 dvar boolean OpenVar[Warehouses];  // this is X in the book
21 dvar boolean AssignVar[Warehouses][Customers];  //this is Y in the book
22
23 /*  This is the Chapter 3 Objective.  We have restored this for the Second Model */
24 minimize
25   sum(w in Warehouses, c in Customers)
26     Distance[w][c]*Demand[c]*AssignVar[w][c];
27
28 subject to{
29
30   ctHighServiceDemand:  /* This is model 1 objective, now a constraint */
31     sum(w in Warehouses, c in Customers)
32       (Distance[w][c] > HighServiceDist?0:1)*Demand[c]*AssignVar[w][c] >= HighServiceDemand;
33
34   forall (w in Warehouses, c in Customers)
35   //this restricts the maximum distance between a warehouse and each customer
36     ctMaximumDistanceConstraint:
37       AssignVar[w][c] <= (Distance[w][c] > MaximumDist?0:1);
38
39
40   forall ( c in Customers )
41     ctEachCustomersDemandMustBeMet:
42         sum( w in Warehouses )
43           AssignVar[w][c]==1;
44
45     ctOnlyUse_P_Warehouses:
46     sum(w in Warehouses)
47       OpenVar[w]==MaxWarehousesP;
48
```

Figure 18: Chapter 4, Second Model

Line 18 shows the new parameter, *HighServiceDemand*. Lines 24 to 26 show the original objective restored. Lines 28 to 30 show the max service level objective is commented out. Lines 34 to 38 show that the average service constraint is not needed. And lines 40 to 42 show the new constraint that will force the model to meet a certain amount of demand.

In the .dat file, we need to add a line to read HighServiceDemand:

```
HighServiceDemand from SheetRead(sheet,"'City200'!M215");
```

You can see we need to update the spreadsheet with a value in cell M215 that is 1,506,800.

We can put this new .mod and .dat file into a new run configuration, and run.

The results come back with a solution that has an average distance of 256.5 miles. In the first model (with an average distance 270.6 miles), the average distance and maximum distance constraints helped keep the solution reasonable. But the second model really allows us to clean up the results and show the best we can do in terms of both the high service level and average distance. You can see in Figure 19 that the graph is cleaner now.

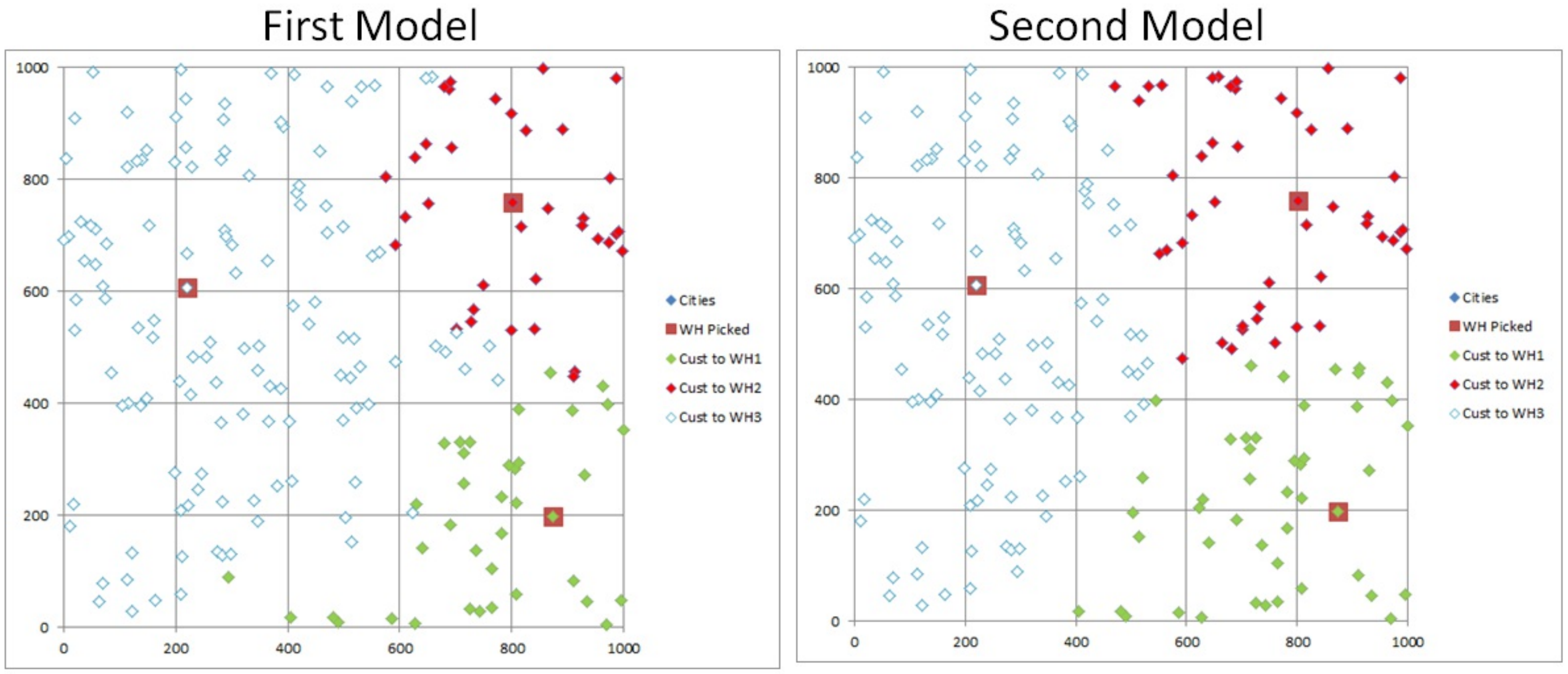


Figure 19: Comparison of First and Second Model

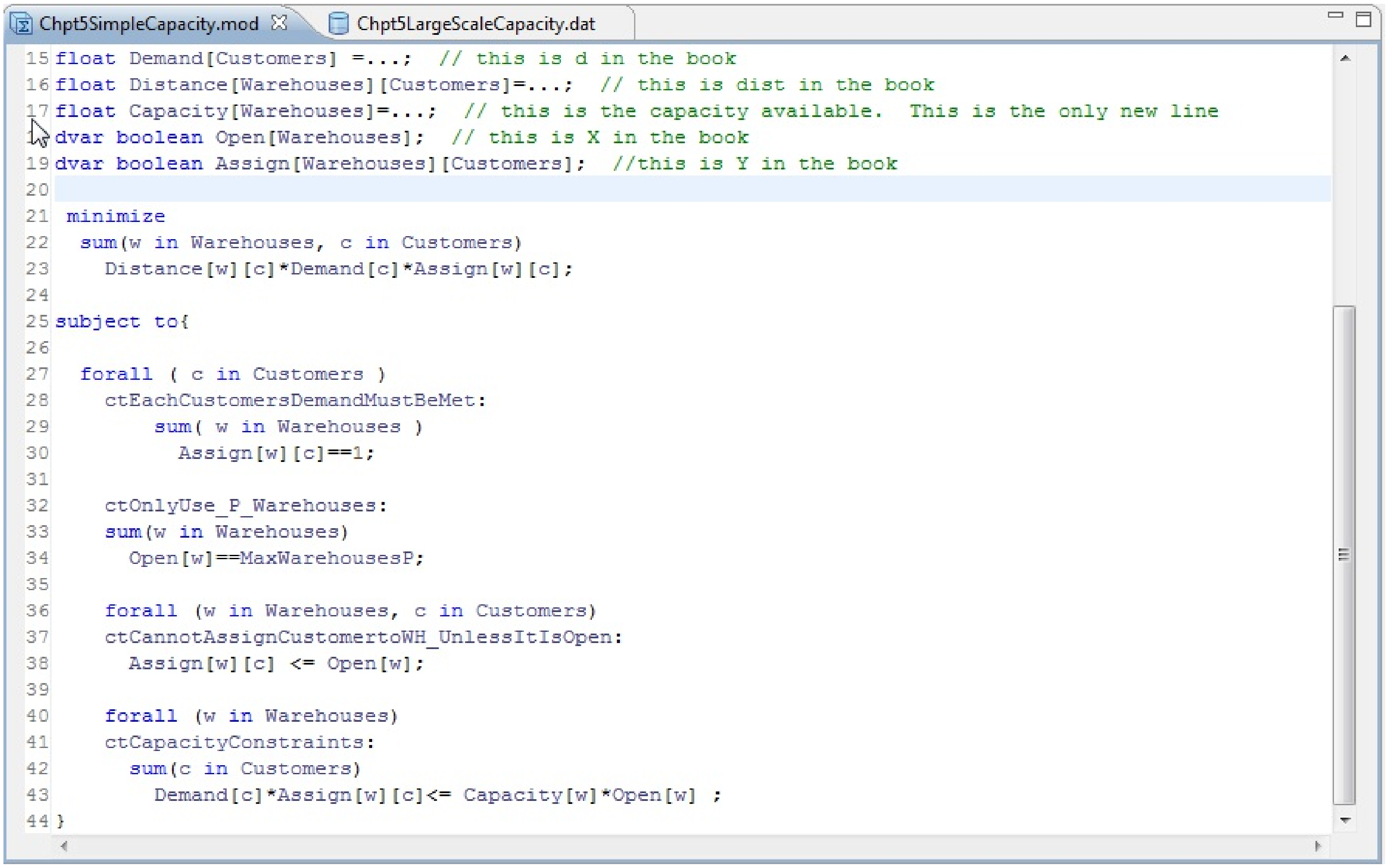
15. Adding Capacity Constraints to the Model

Now we want to add capacity to the model. This follows Chapter 5 of *SCND*. To do this, let's create another OPL Project so we have a folder for this model. An example of this is shown in Figure 16.

We will go back to the model from Chapter 3. Copy the .mod and .dat file from Chapter 3's folder to this folder. Also, copy the spreadsheet from Chapter 3 to the folder that was created on your computer.

We will make one assumption that will simplify this model: let's assume that capacity is measured by total demand and that all demand is equal. The model specified on pages 87 to 89 of *SCND* allows you to enter a parameter, $vol_{i,j}$, that would allow capacity to be measured in different terms (square feet, square meters, cubic meters, pallet positions, and so on) and for different demand to have different capacity requirements. Our assumption will effectively make $vol_{i,j} = Demandj$ for all i and j.

So, we need to make just two changes to the .mod file, as shown in Figure 20.

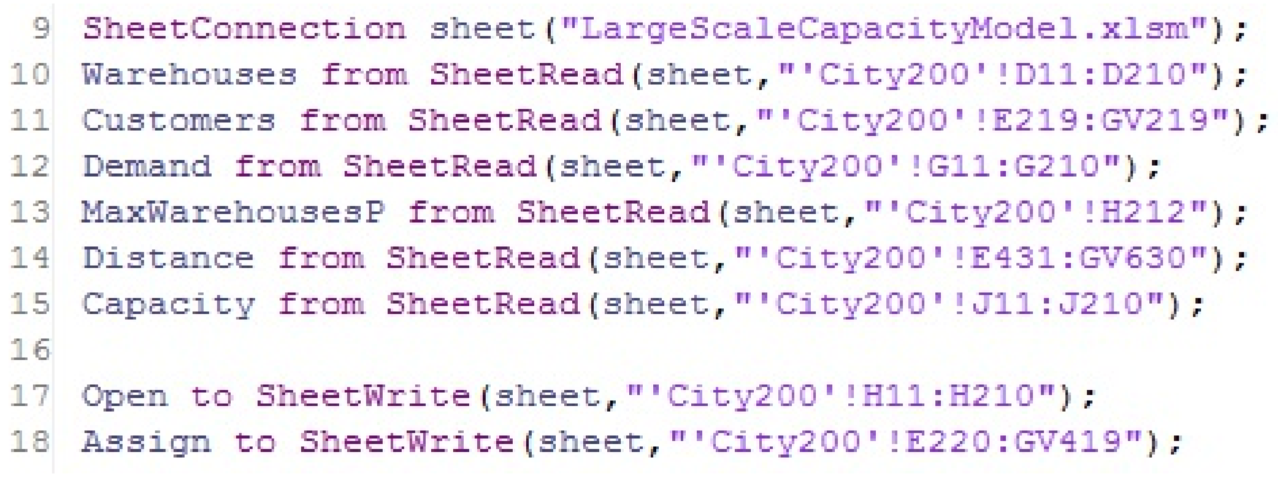


```
15 float Demand[Customers] =...;  // this is d in the book
16 float Distance[Warehouses][Customers]=...;  // this is dist in the book
17 float Capacity[Warehouses]=...;  // this is the capacity available.  This is the only new line
18 dvar boolean Open[Warehouses];  // this is X in the book
19 dvar boolean Assign[Warehouses][Customers];  //this is Y in the book
20
21  minimize
22   sum(w in Warehouses, c in Customers)
23     Distance[w][c]*Demand[c]*Assign[w][c];
24
25 subject to{
26
27   forall ( c in Customers )
28     ctEachCustomersDemandMustBeMet:
29         sum( w in Warehouses )
30           Assign[w][c]==1;
31
32     ctOnlyUse_P_Warehouses:
33     sum(w in Warehouses)
34       Open[w]==MaxWarehousesP;
35
36     forall (w in Warehouses, c in Customers)
37     ctCannotAssignCustomertoWH_UnlessItIsOpen:
38       Assign[w][c]  <= Open[w];
39
40     forall (w in Warehouses)
41     ctCapacityConstraints:
42       sum(c in Customers)
43         Demand[c]*Assign[w][c]<= Capacity[w]*Open[w]  ;
44 }
```

Figure 20: Chapter 5 Capacity Model

We added line 17 to account for the capacity of the warehouse. Then we added lines 40 to 43 to specify that the demand assigned to the warehouse had to be less than the capacity of the warehouse.

Likewise, we added line 15 to the data file (Figure 21) to read in the capacity data from the spreadsheet.



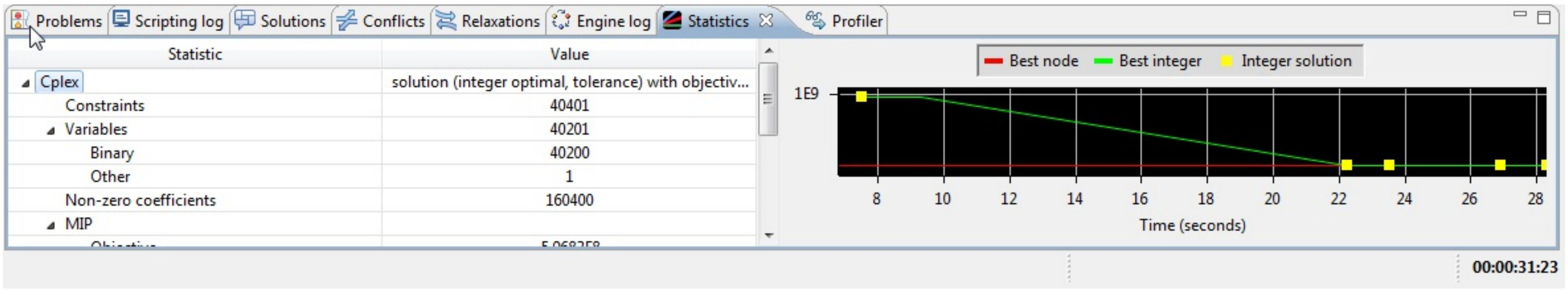
```
 9  SheetConnection sheet("LargeScaleCapacityModel.xlsm");
10  Warehouses from SheetRead(sheet,"'City200'!D11:D210");
11  Customers from SheetRead(sheet,"'City200'!E219:GV219");
12  Demand from SheetRead(sheet,"'City200'!G11:G210");
13  MaxWarehousesP from SheetRead(sheet,"'City200'!H212");
14  Distance from SheetRead(sheet,"'City200'!E431:GV630");
15  Capacity from SheetRead(sheet,"'City200'!J11:J210");
16
17  Open to SheetWrite(sheet,"'City200'!H11:H210");
18  Assign to SheetWrite(sheet,"'City200'!E220:GV419");
```

Figure 21: Data File for the Capacity Model

Before we run, we want to open Excel and set the capacity (cells J11:J210) to 823,400. This will make the warehouses handle almost exactly the same demand. With this constraint, there are 6,300 units of slack—the amount that capacity exceeds demand. This is out of 2,463,900 units of demand.

Create a new run configuration and run. You may notice that the run is a little slower (possibly by 10 to 15 seconds). When you look at the results, the average distance to customers is 242.1 miles, which compares favorably to the 239.9 you get without the capacity constraints. This shows that we had enough demand points and just enough slack to allow the model to find a reasonable solution.

If you reduce the capacity to 821,300, you will have no slack. It turns out that there are enough data points to make this work. The solver takes about 30 seconds. You can watch the solver progress in the window at the bottom of the screen (Figure 22). You can see the run time in the bottom right corner. The tab for Statistics graphically shows the progress of the solver. When the lines converge, you know you are close to finished.

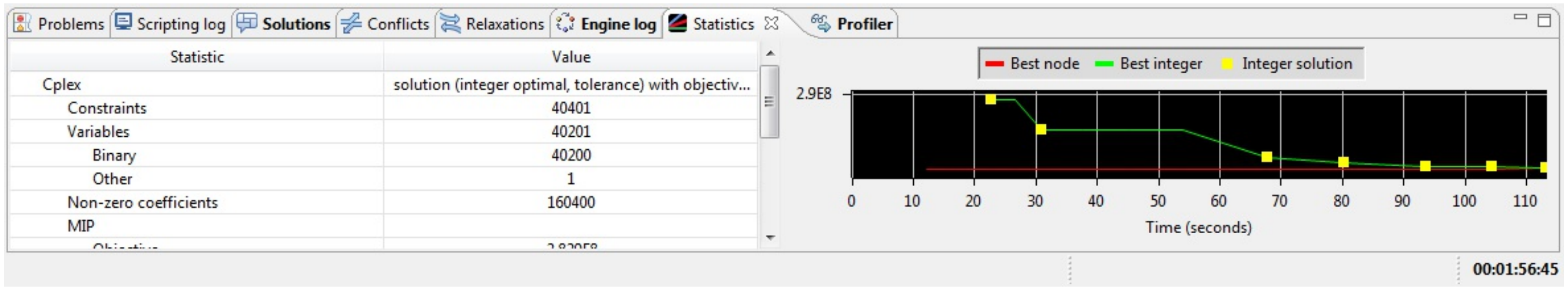


Statistic	Value
Cplex	solution (integer optimal, tolerance) with objectiv...
Constraints	40401
Variables	40201
Binary	40200
Other	1
Non-zero coefficients	160400
MIP	

00:00:31:23

Figure 22: Solver Progress

To see an example of a longer running model, change the demand data to the “Relatively Even Demand” found on the second tab of the spreadsheet, and set the capacity to 407,900. This gives a total of 300 units of slack. The run time is now about two minutes, as shown in Figure 23.



Statistic	Value
Cplex	solution (integer optimal, tolerance) with objectiv...
Constraints	40401
Variables	40201
Binary	40200
Other	1
Non-zero coefficients	160400
MIP	

Figure 23: Solver Progress with New Demand Set

This time it takes much longer for the solver to figure out how to fit all these demand points into the warehouses. You will notice that the lines took much longer to converge.

This should serve as a reminder of the mathematical difficulty of these problems. If we tested more data sets, we would likely find some that could not find a solution or some that took many hours to complete.

16. Bringing Software Engineering Philosophy to Optimization Modeling

Up to this point, we've created a one-to-one CPLEX OPL model of the mathematical formulations you saw in the *SCND* book. We did this to help you strengthen your knowledge of mathematical modeling and build intuition on how the different parts of a math program come together to get an answer.

For your sake, we weren't exactly one-to-one. We did change the *X* and *Y* variable names in the book to *Open* and *Assign*. With limited space in the book and to maintain consistency with other formulations you may have seen, we stuck with *X* and *Y* in *SCND*. But, as we moved to this book and writing OPL code, we expanded the names to make them more readable and understandable.

In the next sections, you'll continue to learn additional OPL skills. More importantly, since industrial-strength mathematical programming requires coding, we are going to apply concepts from the field of computer science to our models. This will allow you to better incorporate optimization into the systems and processes of the firms you work with. The models will be more flexible, easier to support, easier to understand, and easier to maintain. It will increase the models' value to the organization.

17. Using Metric or Helper Variables for Better Models

As you look back and forth from *SCND*'s Chapter 3 and Chapter 4 models (or between the first and second model in Chapter 4), their relative similarity should become apparent. Both models have quite a bit in common. Both have customers that need servicing and warehouses that can be opened to accommodate this goal. Their difference lies primarily in their objective—in Chapter 3, we are trying to minimize the total assignment distance, and in Chapter 4, we are trying to maximize the demand that can be met within the high-service distance restriction.

Besides having a new objective, Chapter 4 also adds a new constraint. Specifically, Chapter 4 requires that the average shipping distance be within a predetermined limit. If you look carefully at the ctAvgServiceDist constraint in Chapter 4, and the objective in Chapter 3, it should be clear that they are functionally identical. Specifically, the left-hand side ctAverageServiceDist is exactly the same as the objective function in Chapter 3. Thus, the Chapter 4 model is the Chapter 3 model with the objective function turned into a constraint, and a new objective function added.

(This is actually understating the difference slightly; Chapter 4 also restricts the maximum shipping distance across a lane. Be patient—we will not neglect this difference when we rewrite our Chapter 4 model.)

This might all become clearer if we recognized that what we really have is one model with multiple goals. The constraints that are common between the two .mod files can be thought of as the "true" constraints or the "mandatory" constraints. They enforce that the solution be "sensible"—that every customer be assigned to a warehouse, that a customer cannot be assigned to an unopened warehouse, and that the correct number of warehouses be opened. Any solution that obeys these constraints can be thought of as feasible, or "legal."

In order to judge which feasible solution is optimal, we have multiple criteria that need to be weighed.

1. What is the average shipping distance? (Or, equivalently, what is the total distance-demand—distance multiplied by demand—of the solution? These two quantities are considered mathematically equivalent, as you can derive one from the other by multiplying or dividing by the total demand.)
2. How much demand is satisfied by high-service shipping lanes? That is, what is the total flow across lanes that are within our threshold high-service distance? For example, how much demand is within 250 miles?
3. What is the longest lane used?

The answer to each of these questions can be thought of as a distinct goal to be minimized or maximized by an optimization engine. None of these goals are explicitly at cross-purposes, because in all instances we are favoring the use of shorter lanes. However, these goals aren't necessarily identical. The solution that achieves the smallest average shipping distance might do so only by forcing a small, low-demand customer to be serviced with an unreasonably long shipping lane. Alternately, it's possible that the solution with the smallest average shipping distance might meet 50% of demand with extremely short shipping lanes, and the other 50% with shipping lanes just slightly too long to qualify as high service.

Although it's possible for these multiple goals to involve difficult trade-offs, for this model, it's more likely that we simply need our optimization model to bear all three goals in mind. We can certainly construct a data set where the only way to meet 90% of our demand with high service is to meet the other 10% with absurdly long shipping lanes. Such a data set would thus mandate making difficult choices between the goal of "maximize high service" and the goals of "minimize average shipping distance" and "avoid long shipping lanes." However, this data set would almost surely be artificial.

What is more likely: from the solutions that achieve 90% high service, we need to select those that also achieve a reasonable average shipping distance and avoid egregiously long lanes. This goal is reflected in the design of the Chapter 4 model. This model maximizes the high-service demand while keeping some reasonable limits on the average shipping distance and the longest lane used. Our goal in this section is to rewrite our Chapter 4 model so that we can easily change which goal is explicitly optimized, and which goals are merely given restrictions.

The approach we shall use is to capture all three of these metrics as variables (we will refer to these as metric variables). You can also think about the metric variables as "helper" variables—they give us some insight into the answer. To properly set the metric variables, we have to introduce additional constraints to ensure that the variables reflect the correct value.

With these metric variables, we can easily edit our model to move between minimizing (or maximizing) one objective and restricting the other two. As a result, the models we are developing in this section will have very simple objective functions. The objective function of the first model consists of a single variable, and that of the second model sums together at most three variables.

If this seems somewhat wasteful to you, then you are correct. These models could be simplified by dispensing with the additional variable/constraint pairs, and instead creating longer and more complicated objective functions (like the ones you saw in the previous sections where we directly translated the mathematical formulations into OPL code). However, such a simplification would not improve the performance of our underlying MIP engine. Every commercial MIP engine comes with a very sophisticated pre-processor that typically removes simple helper variables from the actual model being solved. Even if the pre-processor didn’t perform this task, the underlying engine won’t become confused by our trick, and an additional variable/constraint pair won’t present a noticeable burden to its computational tasks.

In general, good modeling practice doesn’t require you to generate the most algebraically simple model for CPLEX to solve. In fact, the onus tends in the other direction. As professional modelers, you should be most concerned about generating models that are logically correct, human-readable, and flexible enough to alter. These goals promote the creation of models that can be easily debugged, shared amongst modelers, and adjusted to real-world dictates. Often such desires run directly counter to the idea that one should formulate models with the smallest number of variables and constraints possible. Metric variables with the proper constraints can be the most important parts of your model, because they will assist the most error-prone tool being used—the human mind.

That said, the desire for helper variables can be taken too far. If the CPLEX pre-processor removes more than 90% of your variables before launching the true optimization code, you’re probably using too many. Bear in mind that the careful alterations we are making to the Chapter 4 model will not create a model that is harder for CPLEX, but rather, one that is easier for humans.

To begin our rewrite of the Chapter 4 model, we first need to define metric variables for each of our three fitness metrics. Since these variables might be easily confused with parameter constants read from a .dat file, we will name them carefully.

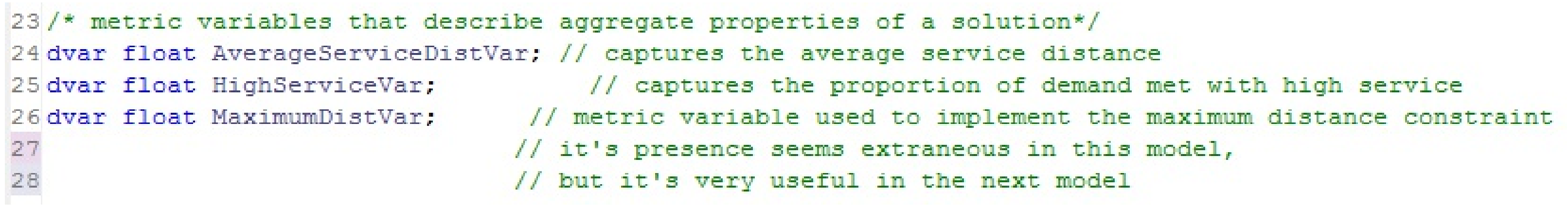
AverageServiceDistanceVar is a continuous variable that will be set to the average shipping distance of the solution. That is, the value of this variable tells us the average shipping distance. Before, we had to calculate this value within the overall logic of our model and our spreadsheet. This is a metric variable, so we are directly calculating it.

HighServiceVar is a continuous variable that will be set to the proportion of demand being met with high service. Again, because this is a metric variable, we are directly calculating it.

MaximumDistVar is a continuous variable that will bound the longest shipping variable being used by our solution. We did not directly discuss this metric before, but it will come in handy.

Each of our three new variables ends with the characters “Var”. This is to make our model more readable; it self-documents these quantities as values that are set by the CPLEX solver engine as part of optimization, and not as data constants read in by a .dat file. In fact, our model in this section will use the “Var” postfix for every variable. This naming convention will help us remember that we are not multiplying any variables together, either in constraints or in the objective function. (It is possible for CPLEX to solve models that multiply variables together. Such models are said to be quadratic. However, CPLEX cannot solve all quadratic models, and the discussion of what quadratic models are appropriate for CPLEX is outside the range of this text.)

Figure 24 shows you how to set up these metric variables in the .mod file.



```
/* metric variables that describe aggregate properties of a solution*/
dvar float AverageServiceDistVar; // captures the average service distance
dvar float HighServiceVar;          // captures the proportion of demand met with high service
dvar float MaximumDistVar;      // metric variable used to implement the maximum distance constraint
                               // it's presence seems extraneous in this model,
                               // but it's very useful in the next model
```

Figure 24: New Metric Variables in the .mod File

Ensuring that the first of these variables is set correctly is straightforward. We can rewrite slightly the ctAvgServiceDistance constraint from Chapter 4.

```
ctAvgServiceDistance:
    sum(w in Warehouses, c in Customers)
    Distance[w][c]*Demand[c]*AssignVar[w][c] ==
    AverageServiceDistVar*sum(c in Customers) Demand[c];
```

Note that this constraint isn't restricting the average service distance, it's setting it. *This is the key idea in setting up the metric or helper variables*—the constraints simply set the value we want. Once we have established that AvgServiceDistVar will reflect the average service distance of any solution, it will be easier to decide whether we want to minimize this quantity or simply restrict it.

Similarly, ensuring that the HighServiceVar variable is set correctly is fairly easy. It employs a very similar constraint.

```
ctHighServiceDistance:
sum(w in Warehouses, c in Customers)
  (Distance[w][c] > HighServiceDist ? 0:1) * Demand[c] * AssignVar[w][c]
  ==
    HighServiceVar * sum(c in Customers) Demand[c] / 100.00;
```

Here the 100.00 represents 100%, and the HighServiceVar represents the percentage of total demand being met with high service. (Bear in mind that HighServiceVar will range between 0 and 100, not 0 and 1, as it is not representing a percentage in a true mathematical sense.)

Finally, we need to be sure that MaximumDistVar is set correctly. This variable is slightly different from the other two, in as much as it restricts the usage of other variables. It needs a family of constraints, as opposed to just one constraint.

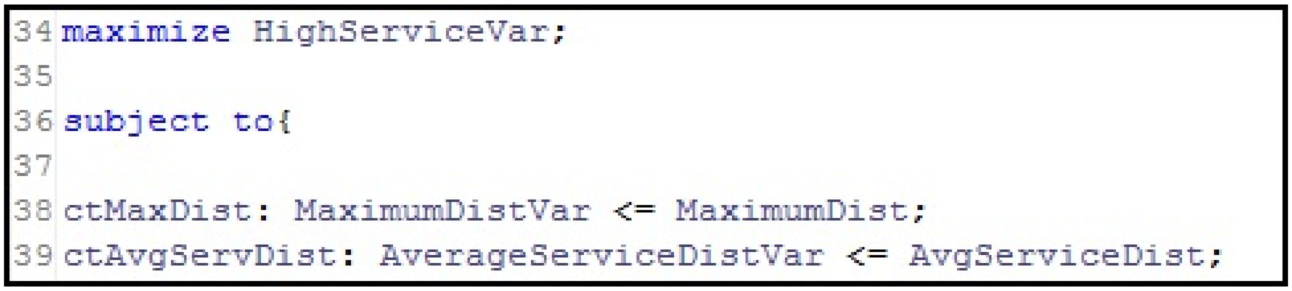
```
forall (w in Warehouses, c in Customers)
ctMaximumDistanceConstraint:
   AssignVar[w][c] * Distance[w][c] <= MaximumDistVar ;
```

Here we exploit the fact that AssignVar[w][c], while logically being interpreted as a boolean, is mathematically represented as a 0-1 variable. If AssignVar[w][c] is given a value of 0 (that is, if w is not assigned to c) then the value of Distance[w][c] irrelevant. When the assignment isn't made, it doesn't matter whether the assignment distance is large or smaller. However, if AssignVar[w][c] is set to a value of 1 (i.e. if w is assigned to c), then we insist that MaximumDistVar must be at least as large as Distance[w][c]. When the assignment is made, the variable tracking the longest assignment cannot be smaller than the length of this assignment.

Note that we cannot always guarantee that MaximumDistVar will return a value that is equal to the distance of the longest shipping variable. However, we can ensure that no shipping lane is used that has a distance larger than MaximumDistVar. As we shall see, this will be sufficient for our purposes.

This last point is subtle, and perhaps requires a bit more clarification. Suppose that there are only four possible shipping lane distances, 0 miles (i.e., sourcing from a co-located warehouse), 50 miles, 100 miles, and 150 miles. Now suppose we formulate our model to maximize the HighServiceVar while restricting MaximumDistVar to be no larger than 125. In this case, any feasible solution will simply avoid using the 150-mile shipping lanes. Suppose CPLEX finds the optimal solution, and this solution uses at least one 100-mile lane. In this case, MaximumDistVar could take on any value between 100 and 125. There is nothing stopping CPLEX from setting MaximumDistVar to 113.113, as such a value will satisfy ctMaximumDistanceConstraint. Of course, CPLEX isn't likely to choose a value quite this arbitrary, but our formulation doesn't preclude such a result. Such an odd state of affairs only occurs if we are not using MaximumDistVar in the objective function. If we are explicitly minimizing MaximumDistVar, and the optimal solution returns with a value for this variable of 100, then we can rest assured that the longest shipping lane used in our solution is precisely 100 miles.

With these constraints as part of our mod file, we can then easily set our model to minimize one goal while restricting the other two. The first three lines of our .mod file (seen in Figure 25) represent a very readable "topic paragraph."



```
maximize HighServiceVar;

subject to{

ctMaxDist: MaximumDistVar <= MaximumDist;
ctAvgServDist: AverageServiceDistVar <= AvgServiceDist;
```

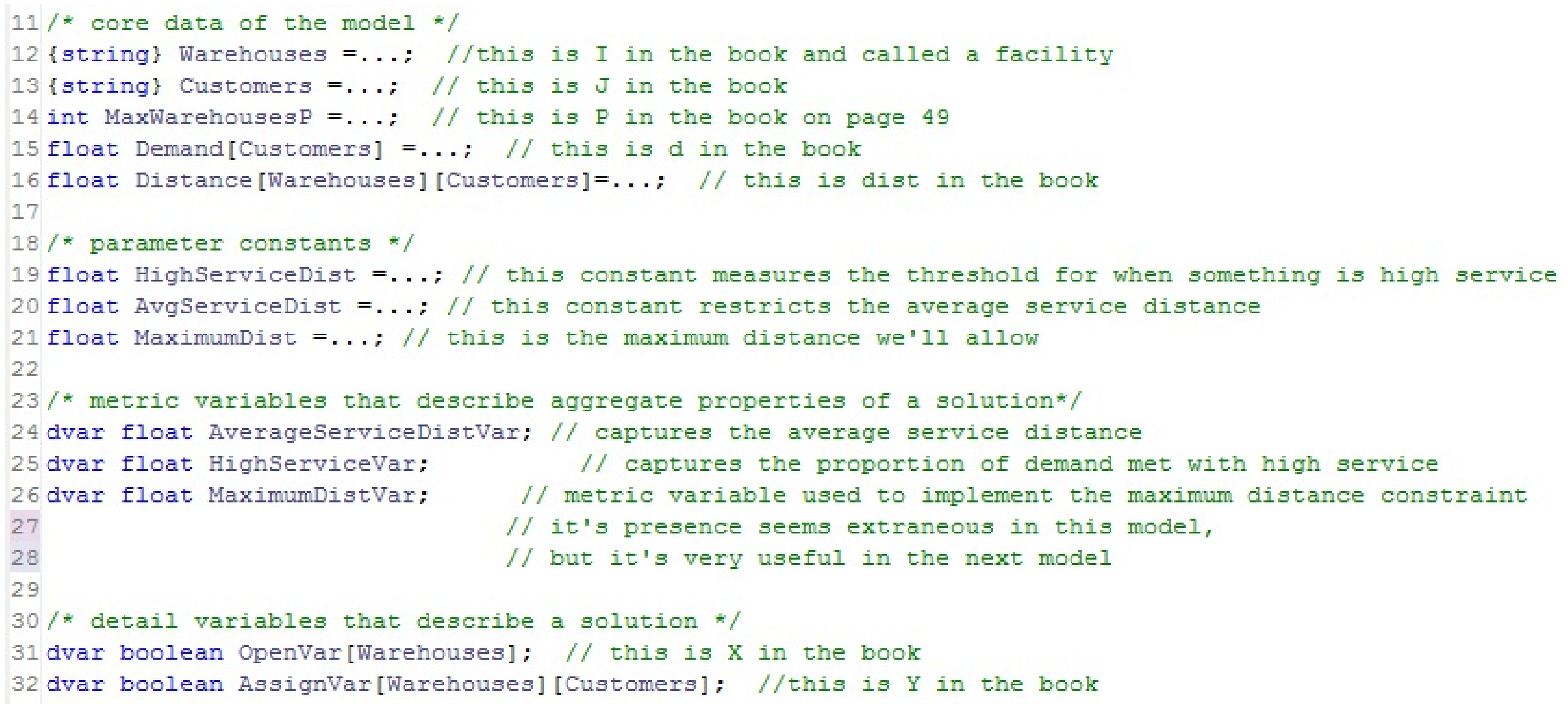
Figure 25: Model Using Metric Variables

It should be clear that this model nicely achieves two goals. First, its first few lines summarize what sort of optimization it will be performing. It will find the solution that achieves high service for the largest possible percentage of customer demand, while ensuring that both the maximum service distance and the average service distance stay below a predefined bound. Moreover, it can easily be updated to alter which goals are being restricted and which are being optimized.

In the next section, we will show the full model, and in the section after that, we will explore how you can easily modify the model and make it more flexible.

18. Full Model with Metric Variables

We have organized the variable section of the .mod file to include the four types of variables we have in the model. This is seen in Figure 26.



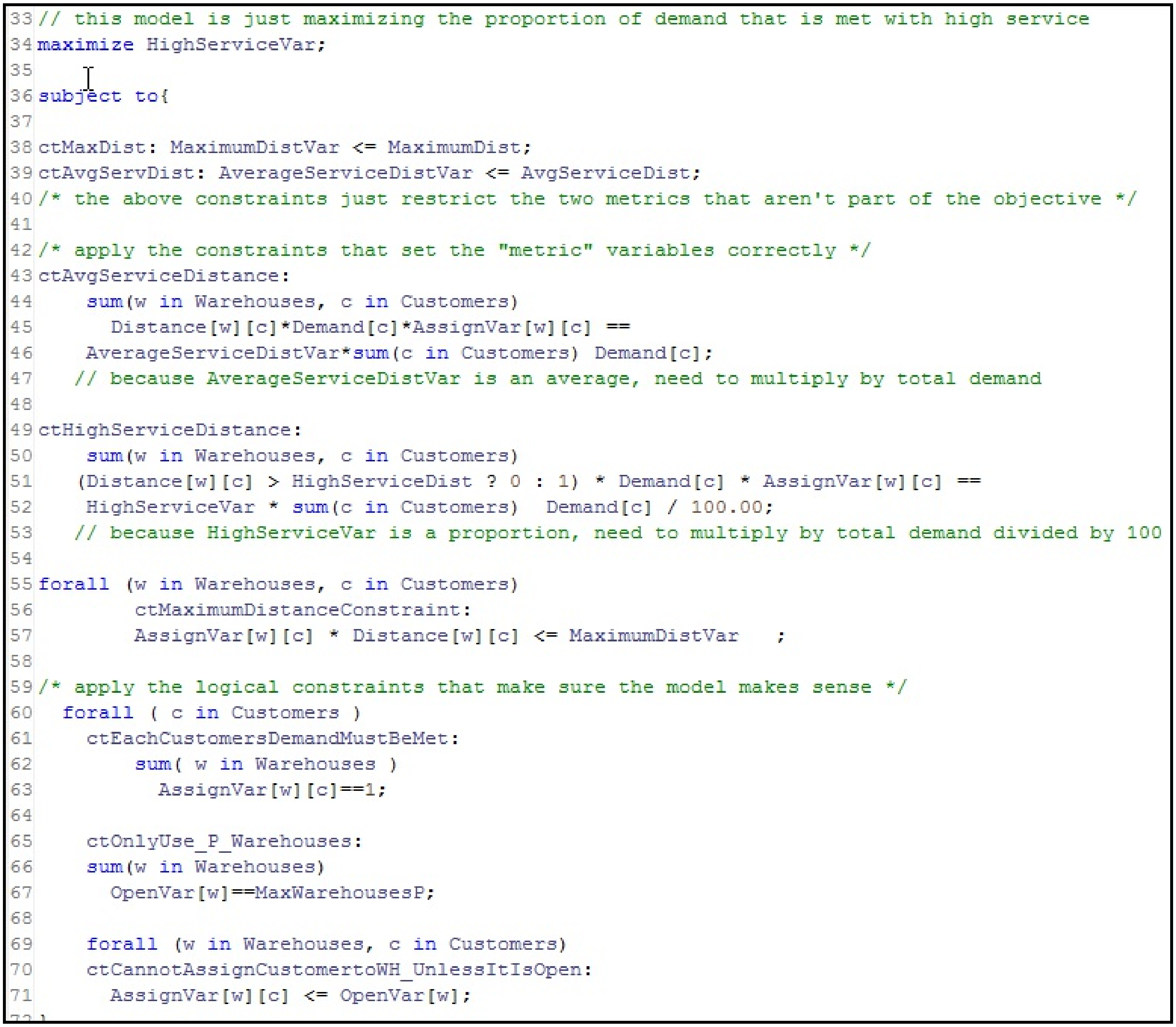
```
11 /* core data of the model */
12 {string} Warehouses =...;  //this is I in the book and called a facility
13 {string} Customers =...;  // this is J in the book
14 int MaxWarehousesP =...;  // this is P in the book on page 49
15 float Demand[Customers] =...;  // this is d in the book
16 float Distance[Warehouses][Customers]=...;  // this is dist in the book
17
18 /* parameter constants */
19 float HighServiceDist =...; // this constant measures the threshold for when something is high service
20 float AvgServiceDist =...; // this constant restricts the average service distance
21 float MaximumDist =...; // this is the maximum distance we'll allow
22
23 /* metric variables that describe aggregate properties of a solution*/
24 dvar float AverageServiceDistVar; // captures the average service distance
25 dvar float HighServiceVar;           // captures the proportion of demand met with high service
26 dvar float MaximumDistVar;       // metric variable used to implement the maximum distance constraint
27                                  // it's presence seems extraneous in this model,
28                                  // but it's very useful in the next model
29
30 /* detail variables that describe a solution */
31 dvar boolean OpenVar[Warehouses];  // this is X in the book
32 dvar boolean AssignVar[Warehouses][Customers];  //this is Y in the book
```

Figure 26: Declaration of Variables

Lines 11 to 16 define the core data elements of the model. We have seen these elements before, just as we've seen the parameter constants in Lines 18 to 21. As part of a better modeling structure, we've separated these from the core model data.

Lines 23 to 26 show our new metric variables. As you can see, we've designated these as decision variables with the keyword **`dvar`**. Lines 29 to 31 show the original decision variables that define the solution.

Figure 27 shows the rest of the .mod file.

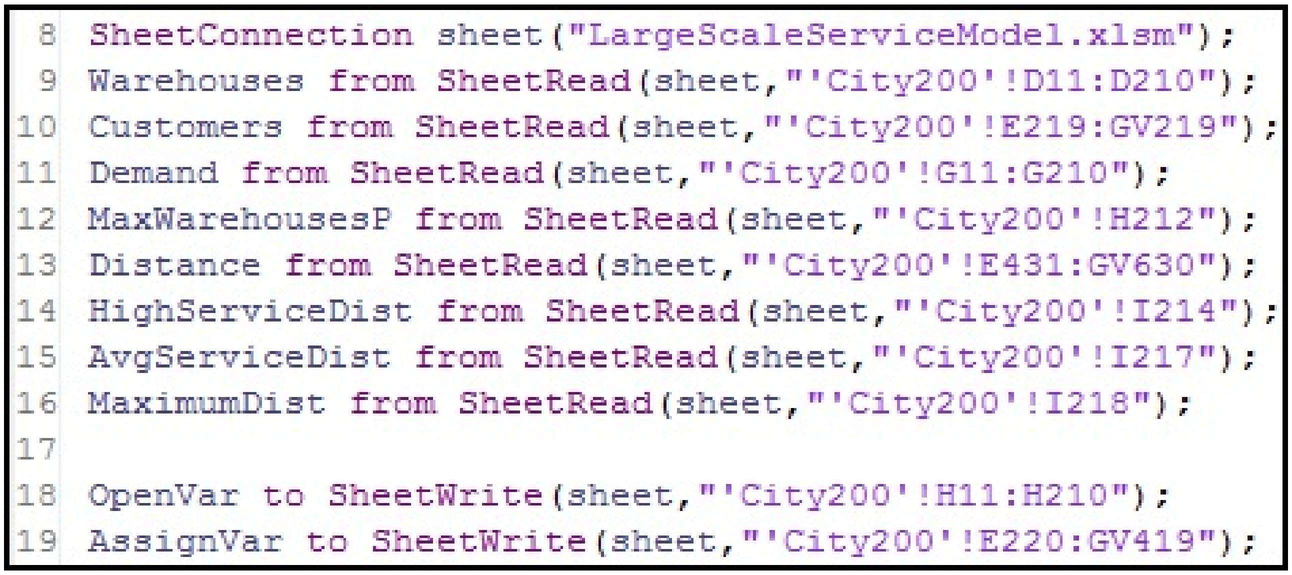


```
33 // this model is just maximizing the proportion of demand that is met with high service
34 maximize HighServiceVar;
35
36 subject to{
37
38 ctMaxDist: MaximumDistVar <= MaximumDist;
39 ctAvgServDist: AverageServiceDistVar <= AvgServiceDist;
40 /* the above constraints just restrict the two metrics that aren't part of the objective */
41
42 /* apply the constraints that set the "metric" variables correctly */
43 ctAvgServiceDistance:
44     sum(w in Warehouses, c in Customers)
45       Distance[w][c]*Demand[c]*AssignVar[w][c] ==
46     AverageServiceDistVar*sum(c in Customers) Demand[c];
47    // because AverageServiceDistVar is an average, need to multiply by total demand
48
49 ctHighServiceDistance:
50     sum(w in Warehouses, c in Customers)
51    (Distance[w][c] > HighServiceDist ? 0 : 1) * Demand[c] * AssignVar[w][c] ==
52     HighServiceVar * sum(c in Customers)  Demand[c] / 100.00;
53    // because HighServiceVar is a proportion, need to multiply by total demand divided by 100
54
55 forall (w in Warehouses, c in Customers)
56         ctMaximumDistanceConstraint:
57         AssignVar[w][c] * Distance[w][c] <= MaximumDistVar   ;
58
59 /* apply the logical constraints that make sure the model makes sense */
60   forall ( c in Customers )
61     ctEachCustomersDemandMustBeMet:
62         sum( w in Warehouses )
63           AssignVar[w][c]==1;
64
65     ctOnlyUse_P_Warehouses:
66     sum(w in Warehouses)
67       OpenVar[w]==MaxWarehousesP;
68
69     forall (w in Warehouses, c in Customers)
70     ctCannotAssignCustomertoWH_UnlessItIsOpen:
71       AssignVar[w][c] <= OpenVar[w];
72 }
```

Figure 27: Full Model with Metric Variables

We have seen lines 33 to 40 in Figure 25. Lines 42 to 57 are the constraints that define the metric variables. And lines 59 to 71 are the same logical constraints we've seen before.

We will reorganize the .dat file in later chapters. For now, we'll use exactly the same .dat file we discussed in Section 1; it's shown in Figure 28.



```
SheetConnection sheet("LargeScaleServiceModel.xlsm");
Warehouses from SheetRead(sheet,"'City200'!D11:D210");
Customers from SheetRead(sheet,"'City200'!E219:GV219");
Demand from SheetRead(sheet,"'City200'!G11:G210");
MaxWarehousesP from SheetRead(sheet,"'City200'!H212");
Distance from SheetRead(sheet,"'City200'!E431:GV630");
HighServiceDist from SheetRead(sheet,"'City200'!I214");
AvgServiceDist from SheetRead(sheet,"'City200'!I217");
MaximumDist from SheetRead(sheet,"'City200'!I218");

OpenVar to SheetWrite(sheet,"'City200'!H11:H210");
AssignVar to SheetWrite(sheet,"'City200'!E220:GV419");
```

Figure 28: Original .dat File for Chapter 4 Model

When you place the .mod and .dat files into a new run configuration and run, CPLEX returns an optimal value for the objective function of 61.155 (or 61.155%). You can confirm that this is the same value as when you previously ran this model and that the other aspects of the solution are the same.

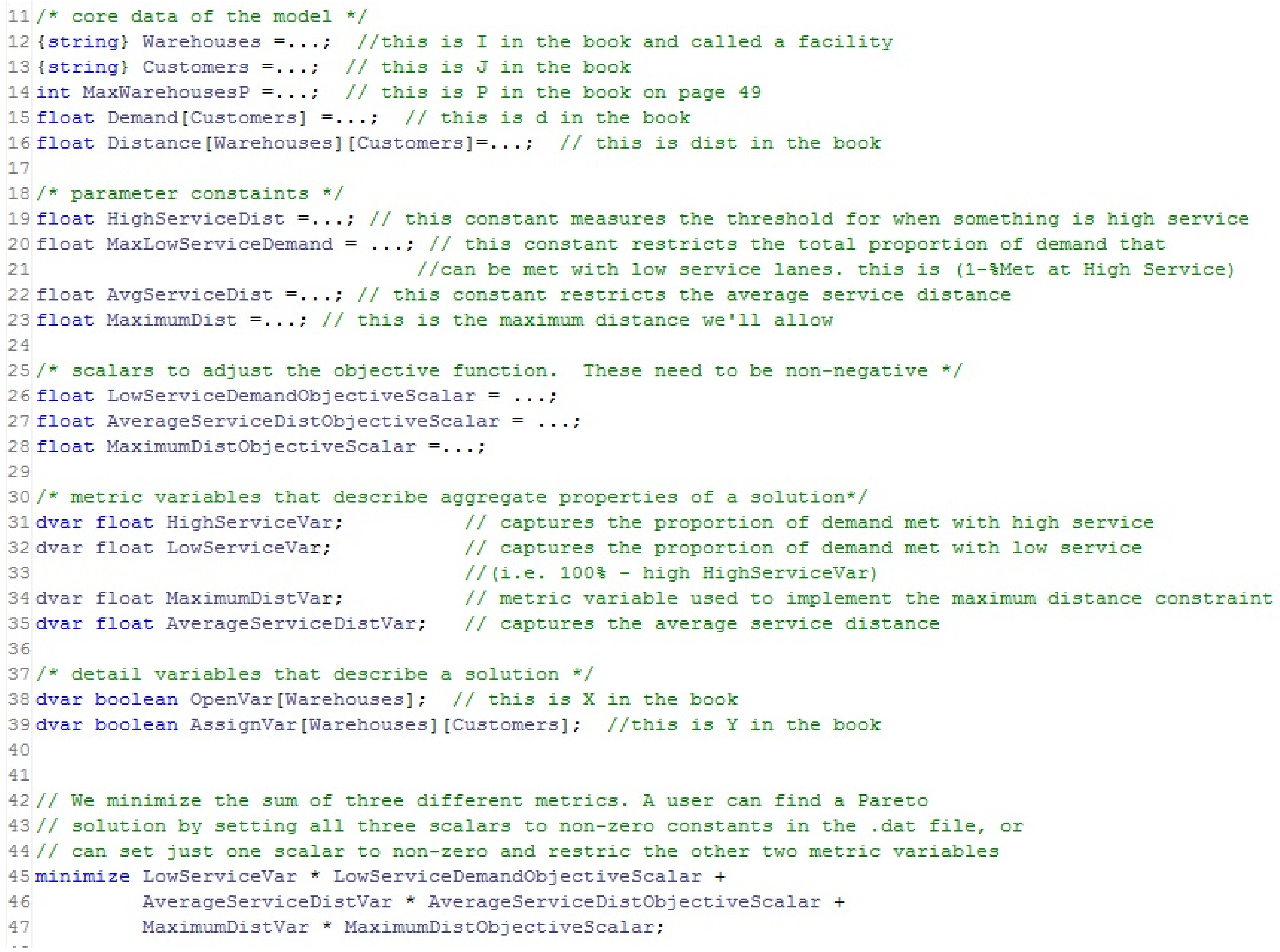
19. Creating Flexible Models

With our metric variables, a user could very easily edit the model in the previous section to instead read:

```
minimize AverageServiceDistVar

subject to {

HighServiceVar >= MinimumHighService

MaximumServiceDistVar <= MaximumDist

… /*remaining body */

}
```

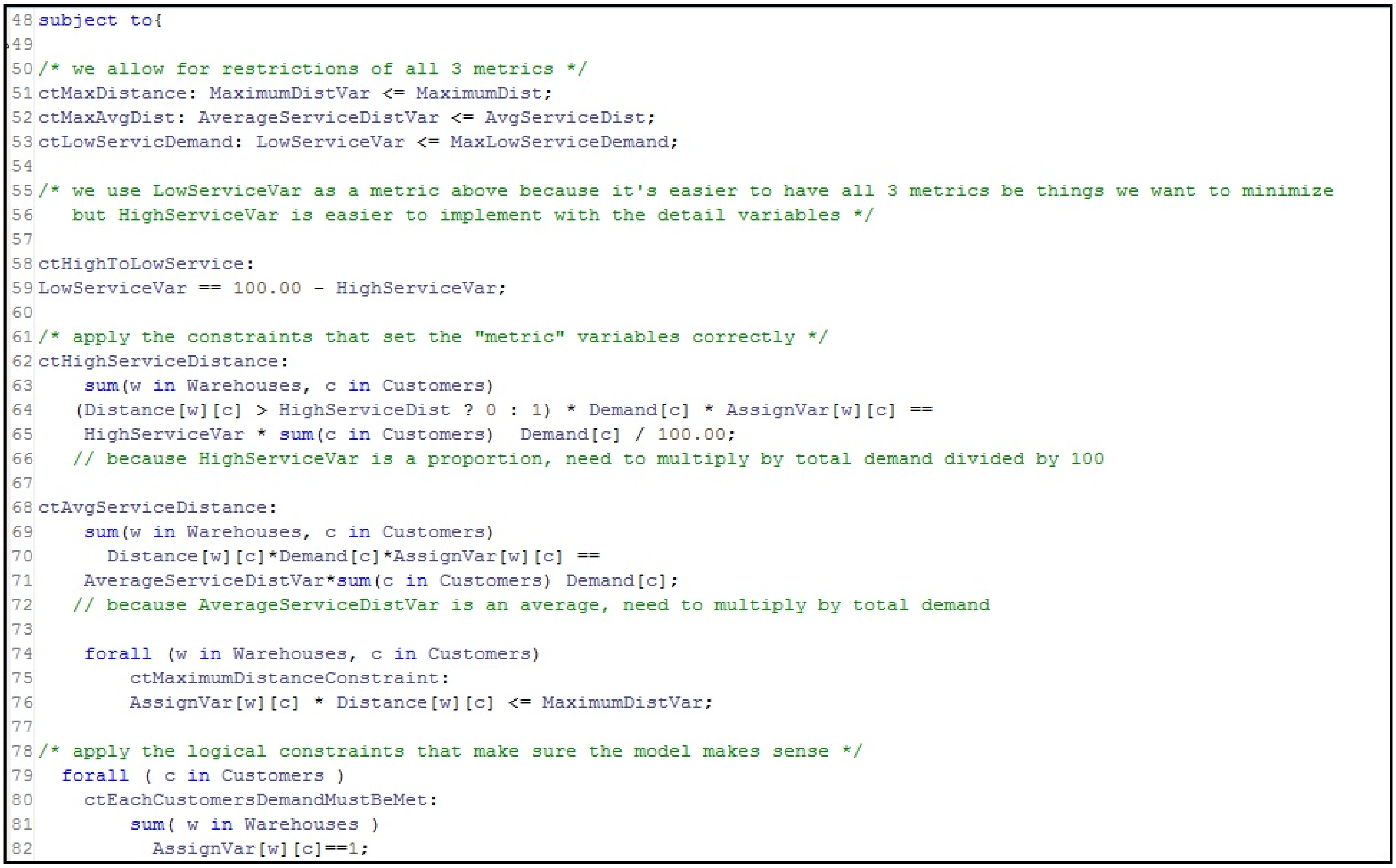
Although this adjustment is fairly easy, it's not necessarily the ideal strategy. For one thing, we now need to introduce an extra constant (here called MinimumHighService) to ensure that our HighServiceVar metric doesn't fall below an acceptable threshold. We also need to change whether the model is minimizing or maximizing. But most importantly, this sort of .mod file editing doesn't really jibe with standard software development practices. A software program generally isn't rebuilt in order to perform a related task. Instead, a programmer builds a piece of software from the ground up to accommodate a suite of associated tasks and lets the user choose from these tasks at run time.

Although this goal might seem impossible for a simple .mod file, it is in fact quite straightforward. The model we present in Figure 29 and Figure 30 has been generalized so it has full flexibility.



```
/* core data of the model */
{string} Warehouses =...;  //this is I in the book and called a facility
{string} Customers =...;  // this is J in the book
int MaxWarehousesP =...;  // this is P in the book on page 49
float Demand[Customers] =...;  // this is d in the book
float Distance[Warehouses][Customers]=...;  // this is dist in the book

/* parameter constaints */
float HighServiceDist =...; // this constant measures the threshold for when something is high service
float MaxLowServiceDemand = ...; // this constant restricts the total proportion of demand that
                                //can be met with low service lanes. this is (1-%Met at High Service)
float AvgServiceDist =...; // this constant restricts the average service distance
float MaximumDist =...; // this is the maximum distance we'll allow

/* scalars to adjust the objective function.  These need to be non-negative */
float LowServiceDemandObjectiveScalar = ...;
float AverageServiceDistObjectiveScalar = ...;
float MaximumDistObjectiveScalar =...;

/* metric variables that describe aggregate properties of a solution*/
dvar float HighServiceVar;          // captures the proportion of demand met with high service
dvar float LowServiceVar;           // captures the proportion of demand met with low service
                                    //(i.e. 100% - high HighServiceVar)
dvar float MaximumDistVar;          // metric variable used to implement the maximum distance constraint
dvar float AverageServiceDistVar;   // captures the average service distance

/* detail variables that describe a solution */
dvar boolean OpenVar[Warehouses];  // this is X in the book
dvar boolean AssignVar[Warehouses][Customers];  //this is Y in the book


// We minimize the sum of three different metrics. A user can find a Pareto
// solution by setting all three scalars to non-zero constants in the .dat file, or
// can set just one scalar to non-zero and restric the other two metric variables
minimize LowServiceVar * LowServiceDemandObjectiveScalar +
         AverageServiceDistVar * AverageServiceDistObjectiveScalar +
         MaximumDistVar * MaximumDistObjectiveScalar;
```

Figure 29: Fully Flexible Model, Part 1 of 2



```
subject to{

/* we allow for restrictions of all 3 metrics */
ctMaxDistance: MaximumDistVar <= MaximumDist;
ctMaxAvgDist: AverageServiceDistVar <= AvgServiceDist;
ctLowServicDemand: LowServiceVar <= MaxLowServiceDemand;

/* we use LowServiceVar as a metric above because it's easier to have all 3 metrics be things we want to minimize
   but HighServiceVar is easier to implement with the detail variables */

ctHighToLowService:
LowServiceVar == 100.00 - HighServiceVar;

/* apply the constraints that set the "metric" variables correctly */
ctHighServiceDistance:
    sum(w in Warehouses, c in Customers)
   (Distance[w][c] > HighServiceDist ? 0 : 1) * Demand[c] * AssignVar[w][c] ==
    HighServiceVar * sum(c in Customers)  Demand[c] / 100.00;
   // because HighServiceVar is a proportion, need to multiply by total demand divided by 100

ctAvgServiceDistance:
    sum(w in Warehouses, c in Customers)
      Distance[w][c]*Demand[c]*AssignVar[w][c] ==
    AverageServiceDistVar*sum(c in Customers) Demand[c];
   // because AverageServiceDistVar is an average, need to multiply by total demand

    forall (w in Warehouses, c in Customers)
        ctMaximumDistanceConstraint:
        AssignVar[w][c] * Distance[w][c] <= MaximumDistVar;

/* apply the logical constraints that make sure the model makes sense */
  forall ( c in Customers )
    ctEachCustomersDemandMustBeMet:
        sum( w in Warehouses )
          AssignVar[w][c]==1;
```

Figure 30: Fully Flexible Model, Part 2 of 2 (model continues, but with exactly the same logical constraints as before)

Some of the model presented here, such as the distance threshold between high and low service, is quite familiar to you. Other data entries require a bit of additional commentary.

- **`MaxLowServiceDemand`** represents the largest percentage of customer demand that can be met with *low* service. In other words, if you set **`MaxLowServiceDemand`** to 10, then the model will only generate solutions that provide high service to 90% or more of customer demand. We define our parameter for low service, and not for high service, so that all of our parameters control “golf score” metrics (i.e., metrics that indicate improvement by growing smaller).
- **`AvgServiceDist`** represents the largest possible average service distance for a solution.
- **`MaximumDist`** represents the longest possible shipping lane that can be used in a feasible solution.

These three parameters all restrict quantities that we want to minimize—the number of customers receiving low service, the average service distance, and the longest shipping lane. Only **`MaxLowServiceDemand`** is new; the others were introduced in previous models.

We also create three very similar parameters to control the optimization goal of the CPLEX model. **`LowServiceDemandObjectiveScalar`**, **`AverageServiceDistObjectiveScalar`** and **`MaximumDistObjectiveScalar`**, are all objective function scalars. The optimization function of our .mod file multiplies each metric variable by its associated scalar and adds the results together. Our .mod file now applies the objective function of

```
minimize LowServiceVar * LowServiceDemandObjectiveScalar +
         AverageServiceDistVar * AverageServiceDistObjectiveScalar +
         MaximumDistVar * MaximumDistObjectiveScalar;
```

Thus, if we set **`LowServiceDemandObjectiveScalar`** to 1 and both **`AverageServiceDistObjectiveScalar`** and **`MaximumDistObjectiveScalar`** to 0 (zero), then the model will minimize the percentage of demand met with low service. (This is functionally equivalent to maximizing the percentage of demand met with high service, as we did with the preceding models). Alternately, we could set **`AverageServiceDistObjectiveScalar`** to 1 (one) while fixing **`MaximumDistObjectiveScalar`** and **`AverageServiceDistObjectiveScalar`** at 0 in order to minimize the average service distance.

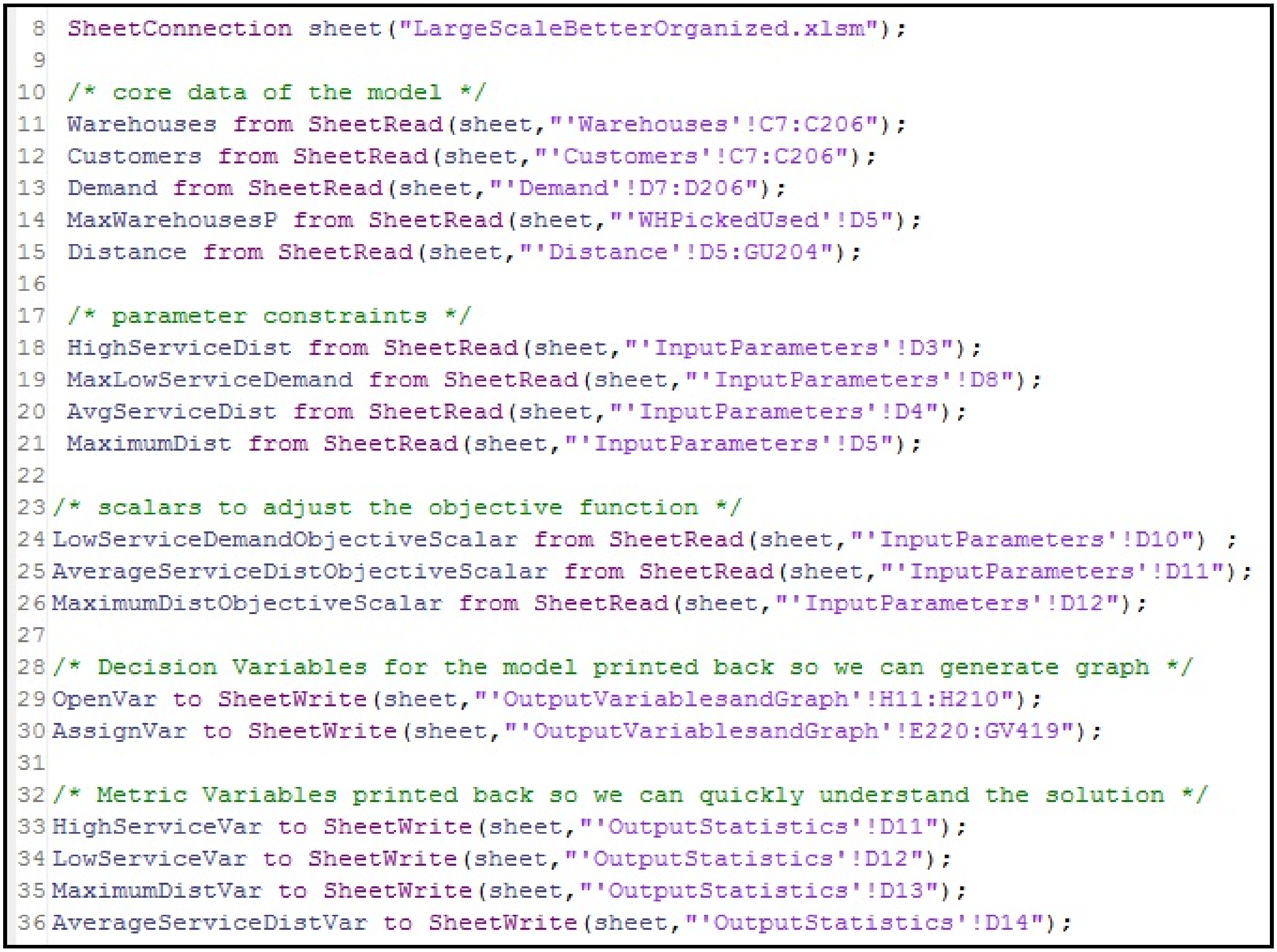
In fact, we could optimize all three objectives at the same time by providing positive values to **`LowServiceDemandObjectiveScalar`**, **`AverageServiceDistObjectiveScalar`**, and **`MaximumDistObjectiveScalar`** in the same data set. Doing so would force CPLEX to generate a "Pareto Optimal" solution, as discussed in Chapter 11 of *SCND*. There might be many Pareto solutions, and setting different positive scalars for each of our scalars would emphasize one Pareto solution over another. That is to say, if **`LowServiceDemandObjectiveScalar`** is 500 and **`AverageServiceDistObjectiveScalar`** = **`MaximumDistObjectiveScalar`** = 1, then the CPLEX engine will provide a greater emphasis on Pareto solutions that maximize high service. Although a more detailed discussion of Pareto Frontier analysis is beyond the scope of this text, it is worth noting that this fairly simple .mod file can provide a useful overview of the multi-objective trade-offs present in a given data set.

To conclude this section, we should emphasize that not all models need to generalize their solution metrics to the extent demonstrated here. This section carries the concept of pre-engineering your model for maximum flexibility to its logical extreme, largely for purposes of exposition. That said, nearly all models benefit from the inclusion of helper variables designed to make the .mod file more readable. Moreover, it a general truism of software development that code that's more readable is almost always more extensible, and thus more appropriate for the unexpected demands of real-world applications. With any luck, this chapter has helped you develop an appreciation for the classic quote "Programs are meant to be read by humans and only incidentally for computers to execute" (Sussman and Abelson, http://amzn.to/WdbURL).

20. Organizing the Data Files

There are many practices we can borrow from software engineering to make the reading of the data more effective as well. Again, this will be driven by the needs of the project. We will touch on a few best practices here, but you should note that there is much more you could do to automate the reading of data, the running of multiple scenarios without intervention, and the reporting.

Figure 31 shows a better organized .dat file.



```
SheetConnection sheet("LargeScaleBetterOrganized.xlsm");

/* core data of the model */
Warehouses from SheetRead(sheet,"'Warehouses'!C7:C206");
Customers from SheetRead(sheet,"'Customers'!C7:C206");
Demand from SheetRead(sheet,"'Demand'!D7:D206");
MaxWarehousesP from SheetRead(sheet,"'WHPickedUsed'!D5");
Distance from SheetRead(sheet,"'Distance'!D5:GU204");

/* parameter constraints */
HighServiceDist from SheetRead(sheet,"'InputParameters'!D3");
MaxLowServiceDemand from SheetRead(sheet,"'InputParameters'!D8");
AvgServiceDist from SheetRead(sheet,"'InputParameters'!D4");
MaximumDist from SheetRead(sheet,"'InputParameters'!D5");

/* scalars to adjust the objective function */
LowServiceDemandObjectiveScalar from SheetRead(sheet,"'InputParameters'!D10") ;
AverageServiceDistObjectiveScalar from SheetRead(sheet,"'InputParameters'!D11");
MaximumDistObjectiveScalar from SheetRead(sheet,"'InputParameters'!D12");

/* Decision Variables for the model printed back so we can generate graph */
OpenVar to SheetWrite(sheet,"'OutputVariablesandGraph'!H11:H210");
AssignVar to SheetWrite(sheet,"'OutputVariablesandGraph'!E220:GV419");

/* Metric Variables printed back so we can quickly understand the solution */
HighServiceVar to SheetWrite(sheet,"'OutputStatistics'!D11");
LowServiceVar to SheetWrite(sheet,"'OutputStatistics'!D12");
MaximumDistVar to SheetWrite(sheet,"'OutputStatistics'!D13");
AverageServiceDistVar to SheetWrite(sheet,"'OutputStatistics'!D14");
```

Figure 31: .dat File for Fully Flexible Model

As you can quickly see, we've set up the model to read in both its metric restrictions and its optimization goals from the Excel data sheet. To help organize the input data, we have created a dedicated sheet for each of the different types of data. You can see this in the different worksheets we call for each line item in the .dat file.

For additional flexibility, we could have read from a database, but most readers will have easier access to Excel, so we've stuck with that.

A technical detail worth noting is that OPL provides the keyword **`infinity`**. Thus, the cleanest way to allow the model to use the longest shipping lane available would be to include the line **`MaximumDist = infinity`** in the .dat file, and not to put a very large number in the **`MaximumDist`** cell of the .xls spreadsheet. So, depending on your needs, it may be easier for your .dat file to read from other sources, but to also include data. You can mix and match where you get data.

You will see that we are also now writing the metric variables to the spreadsheet. That is to say, we are writing two types of results to our Excel file. The true decision variables specify a solution, and the metric variables write out aggregate diagnostic information. Of course, the MaximumDistVar is not perfectly trustworthy as a solution descriptor, as we discussed in the previous section. Nevertheless, it can provide useful reference data and thus we include it on our output spreadsheet.

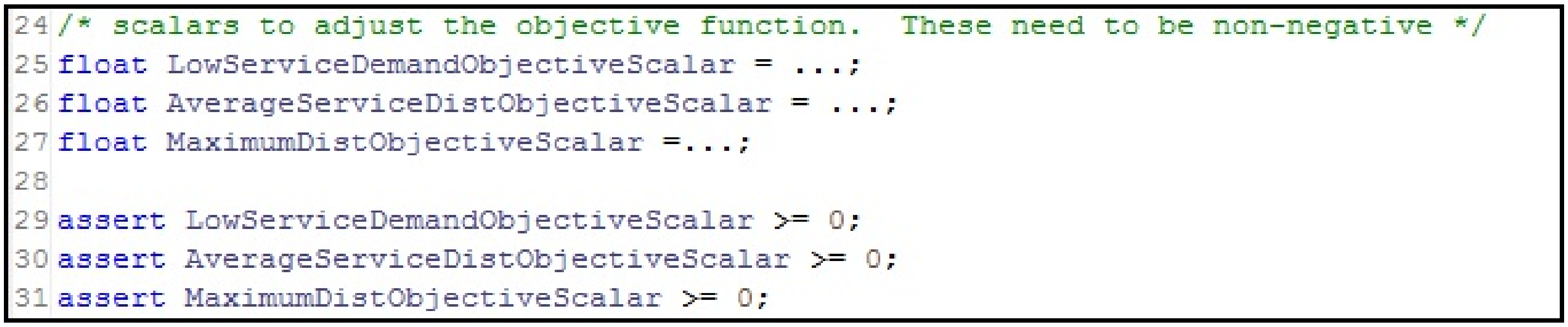
We should note that OPL provides specific language facilities dedicated to the creation of "reporting" variables that aren't strictly part of the optimization. We encourage any reader wishing to become an OPL power user to study the OPL manuals for these more advanced functionalities.

Our .dat file is not fully automated. If you want to test different values for the scalars, you can copy the spreadsheet, rename it, and update the .dat file. In this case, you may want to create multiple run configurations to save the various runs. If you just want the output on the OutputStatistics worksheet, you can run and simply copy the results to a new column, change the scalars, and re-run.

21. Creating More Robust Models

To make our model more robust, we assert that each of our objective scalars is non-negative. Ideally, a user would carefully examine all of our mod file comments and carefully select non-negative scalars for each of our three objective function coefficients. However, we also understand that in the real world, people will play around with models (and really, with any product) without carefully reading the instructions first. Thus, we make our code more robust and self-documenting by insisting that the objective scalars be non-negative.

To do this in OPL, we use the keyword **`assert`**. We modify the model shown in Figure 29 and Figure 30 with three new lines (29 to 31), as shown in Figure 32.

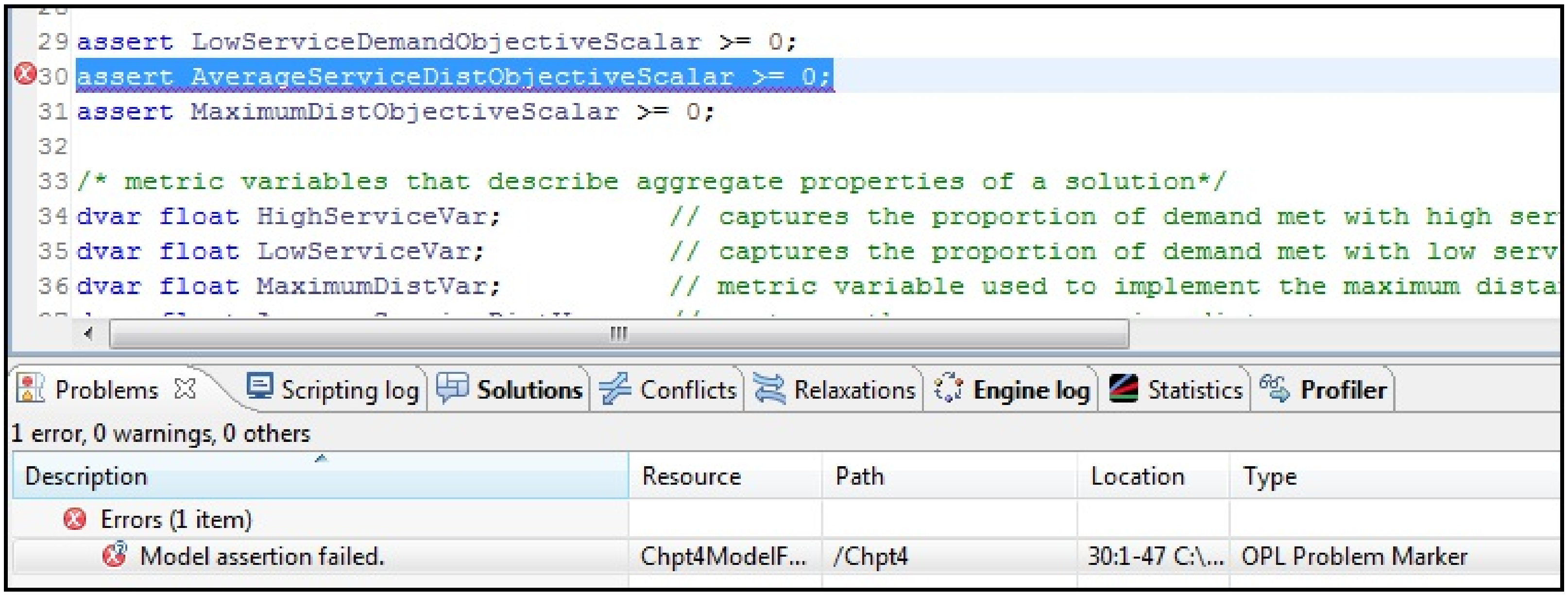


```
24 /* scalars to adjust the objective function.  These need to be non-negative */
25 float LowServiceDemandObjectiveScalar = ...;
26 float AverageServiceDistObjectiveScalar = ...;
27 float MaximumDistObjectiveScalar =...;
28
29 assert LowServiceDemandObjectiveScalar >= 0;
30 assert AverageServiceDistObjectiveScalar >= 0;
31 assert MaximumDistObjectiveScalar >= 0;
```

Figure 32: Adding Asserts into Fully Flexible Model Shown in Figure 29

It is certainly the case that negative objective function coefficients have their place in optimization modeling. (For a quick exercise, rewrite the model in Figure 27 as an equivalent minimization model with a negative coefficient in the objective row). However, any instance of negative objective scalars in Figure 29 and Figure 30 almost surely represents a data entry mistake. (Why? What would the Figure 29 and Figure 30 model do if LowServiceDemandObjectiveScalar were equal to -1 and the other two objective scalars were zero?). Thus, we check for these mistakes with three **`assert`** statements.

If you go to Excel and set the AverageServiceDistObjectiveScalar to -1 and then run, the model will stop quickly and return an error and point to the problem as seen in Figure 33.



```
assert LowServiceDemandObjectiveScalar >= 0;
assert AverageServiceDistObjectiveScalar >= 0;
assert MaximumDistObjectiveScalar >= 0;

/* metric variables that describe aggregate properties of a solution*/
dvar float HighServiceVar;          // captures the proportion of demand met with high ser
dvar float LowServiceVar;           // captures the proportion of demand met with low serv
dvar float MaximumDistVar;          // metric variable used to implement the maximum dista
```

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
Model assertion failed.	Chpt4ModelF...	/Chpt4	30:1-47 C:\...	OPL Problem Marker

Figure 33: Error When Scalar Set to -1 (negative one)

Assertions are an incredibly valuable tool in software development. They promote “self-aware” software by forcibly alerting users to dangerously unexpected data patterns. The Wikipedia entry on assertions provides a great overview of this topic (http://bit.ly/acSU5W). We have only shown a simple example of an `assert` command. The more your models will be used by others, the more you should use assertions to help test assumptions and trap problems before they get to the solver (where they frustrate the user or are harder to debug).

22. Sparsity and a Simplified Multi-Tier Multi-Product Model

There are two fundamental issues that separate the creation of purely mathematical optimization models and the useful implementation of these theoretical musings. The first is that of development style. Professional developers writing industrial code recognize that software is maintained and extended by people who often work under stress and with poor preparation. As such, your .mod files should use human-readable names and highlight key metrics. These issues were explored in detail when we generalized the concepts of average service distance, maximum service distance, and aggregate service level in Sections 1 to 20. In those sections we marginally increased the computational burden our models impose on the OPL/CPLEX engine, while creating models that are easier to understand and bring deeper analytical power.

In this chapter, we change directions completely. Whereas before we seemed oblivious to the run-time implications of our modeling strategy, here we will fuss over the memory and run-time implications of our actions. In the preceding chapter, our focus was on creating a work-product that sacrificed efficiency for the sake of readability. Here, we sacrifice readability for the sake of efficiency.

It seems odd that the transition from mathematics to programming should require the application of such contradictory goals. One's gut instinct is to pursue one tack or the other, but not both. Although our short-term goals appear to be at cross purposes, we will pursue them in such a way that one doesn't fully cancel out the other. At the end of our two reaches, we will return not to the place we started from, but closer to our destination.

The key here is to recognize the magnitude of our actions. In Sections 1 to 20 we made our models *much* more powerful, while imposing a *slight* computational burden. In this section, we will make our models run *much* faster, while imposing only a *minor* burden on their readability. This is a common theme of software development. If developers were forced to consider the run-time implications of every single line of code, they would become paralyzed by indecision. Instead, developers focus on creating code that makes sense to humans, while pursuing a computational strategy that is reasonably efficient. Once the product is actually functioning, it is then tuned. The tuning process is almost always a hunt for large game. Software often obeys an 80-20 rule, whereby 80% (or more) of the run time is associated with 20% (or less) of the code. Thus, "the second tack" of software development strives to make significant run time improvements by carefully adjusting key areas of the code.

In Mixed Integer Programming (MIP) development, the big game of run-time improvements tends to fall into two general areas—mathematical formulation and data sparsity. Since this book is focused on hands-on development, we'll focus on the latter. Although the largest potential for run time improvements probably lies in the careful placement of a deeply insightful "cut" (or constraint), this area is well addressed by a large body of academic literature. Any class that discusses MIP will introduce you to this concept, and it is touched on briefly in our *SCND* text.

Data sparsity, on the other hand, is almost always taken for granted in purely mathematical MIP formulations. The implication is that the equations will be run on some Platonic computer that will magically skip over obviously irrelevant combinations of source, destination, and product. In fact, this is precisely the area where a professional modeler will most frequently bridge mathematics and programming. If you merely translate equations into OPL, you are running a high risk of creating solutions that pointless chug with vast amounts of utterly irrelevant data.

To illustrate this concept, we first present a simplification of the multi-product model from Chapter 10 of *SCND*.

Minimize $\sum_{l\in L}\sum_{i\in I}\sum_{k\in K} transPW_{l,i,k} Z_{l,i,k} +$

$\sum_{i\in I}\sum_{j\in J}\sum_{k\in K} transWC_{i,j,k} d_{j,k} Y_{i,j,k} +$

$\sum_{i\in I} whFix_i X_i$

Subject to:

$(1) \sum_{i\in I} Y_{i,j,k} = 1; \forall j\in J, \forall k\in K$

$(2) \sum_{i\in I} X_i = P$

$(3) \sum_{j\in J}\sum_{k\in K} vol_{i,j,k} Y_{i,j,k} \le whCap_i X_i; \forall i\in I$

$(4) \sum_{l\in L} Z_{l,i,k} = \sum_{j\in J} d_j Y_{i,j,k} \forall i\in I, \forall k\in K$

$(5) \sum_{i\in I} Z_{l,i,k} \le pCap_{l,k}; \forall l\in L, \forall k\in K$

$(6) Y_{i,j,k} \le X_i; \forall i\in I, \forall j\in J, \forall k\in K$

$(7) Y_{i,j,k} \in \{0,1\}; \forall i\in I, \forall j\in J, \forall k\in K$

$(8) X_i \in \{0,1\}; \forall i\in I$

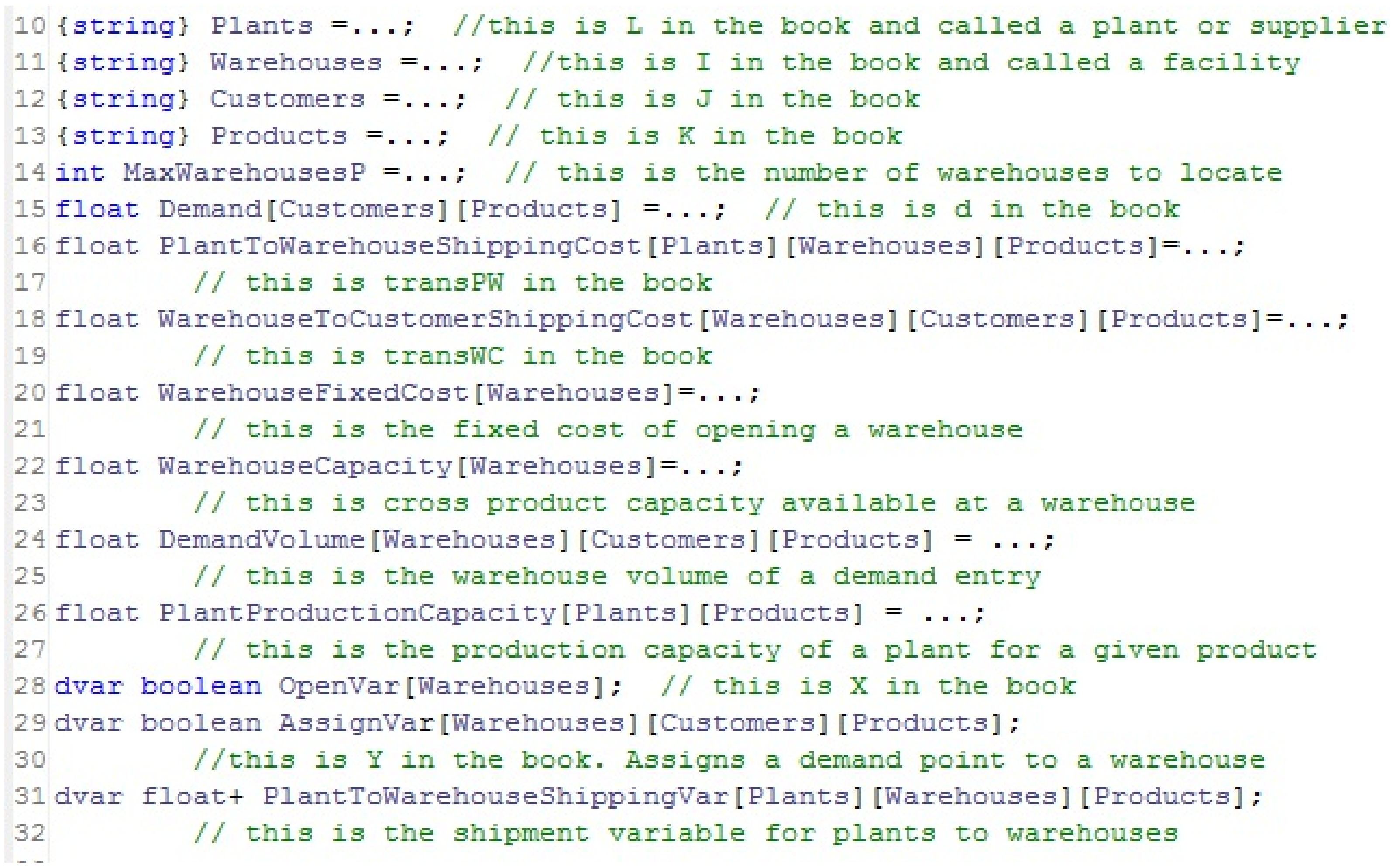
$(9) Z_{l,i,k} \ge 0; \forall l\in L, \forall i\in I, \forall k\in K$

This model is the multi-product model we developed in Chapter 10 of *SCND*, with a few simplifications.

- We assume that each warehouse can be opened only at one size. Thus, the warehouse option index set *W* consists of a single entry, and can be omitted.
- We assume that the warehouse handling fees and plant production costs are zero, and thus remove *whVar* and *pVar*.
- We insist that exactly *P* warehouses be opened, rather than allowing the number of opened warehouses to vary between P_{min} and P_{max}.

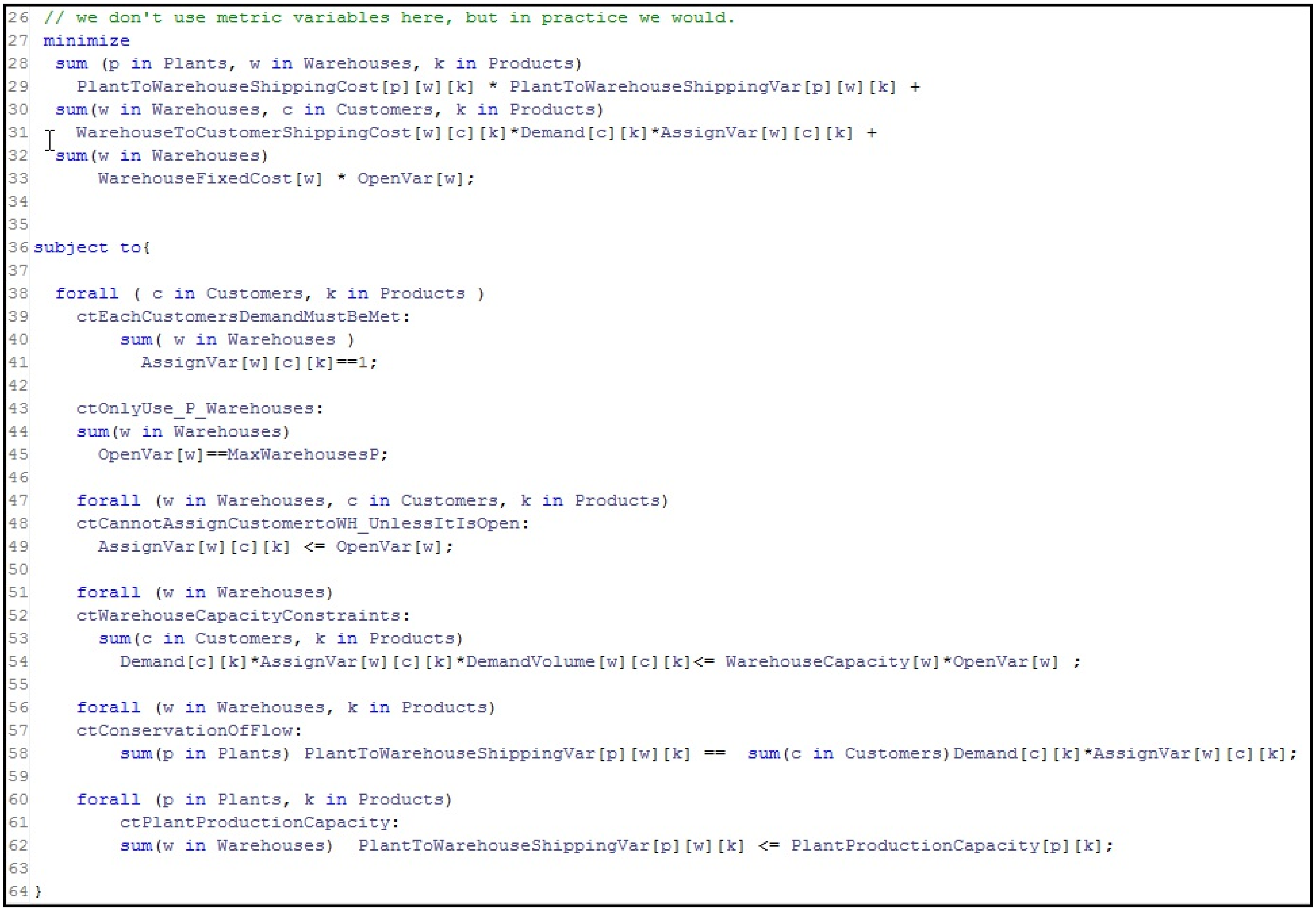
While these assumptions obviously make our model less realistic, they also create a more focused learning environment. We have removed the constants and indices that aren't needed to illustrate the idea of sparsity, thus allowing us to better isolate the elements that require special handling with real-world data.

We present a simple example of this model with SimpleMultiTierMultiProd.mod, shown in Figure 34 and Figure 35. This .mod file makes no allowances for sparsity. There is a variable tracking the assignment of every warehouse to every possible demand point. There is a variable tracking the shipment of every product from every plant to every warehouse. This model is the cleanest translation of mathematics to code—in fact, the code here is almost surely more readable than the math. The only difference to note in this model is that we've defined the decision variable of PlantToWarehouseShippingVar with the keyword **`float+`**. The **+** is a shortcut for defining that the variable must be greater than or equal to zero. (Once you get this model running, you can take away the **+** and see how model is unbounded.) In practice, we would again use metric variables, but in this section, we are just isolating the issue of sparsity.



```
{string} Plants =...;  //this is L in the book and called a plant or supplier
{string} Warehouses =...;  //this is I in the book and called a facility
{string} Customers =...;  // this is J in the book
{string} Products =...;  // this is K in the book
int MaxWarehousesP =...;  // this is the number of warehouses to locate
float Demand[Customers][Products] =...;  // this is d in the book
float PlantToWarehouseShippingCost[Plants][Warehouses][Products]=...;
        // this is transPW in the book
float WarehouseToCustomerShippingCost[Warehouses][Customers][Products]=...;
        // this is transWC in the book
float WarehouseFixedCost[Warehouses]=...;
        // this is the fixed cost of opening a warehouse
float WarehouseCapacity[Warehouses]=...;
        // this is cross product capacity available at a warehouse
float DemandVolume[Warehouses][Customers][Products] = ...;
        // this is the warehouse volume of a demand entry
float PlantProductionCapacity[Plants][Products] = ...;
        // this is the production capacity of a plant for a given product
dvar boolean OpenVar[Warehouses];  // this is X in the book
dvar boolean AssignVar[Warehouses][Customers][Products];
        //this is Y in the book. Assigns a demand point to a warehouse
dvar float+ PlantToWarehouseShippingVar[Plants][Warehouses][Products];
        // this is the shipment variable for plants to warehouses
```

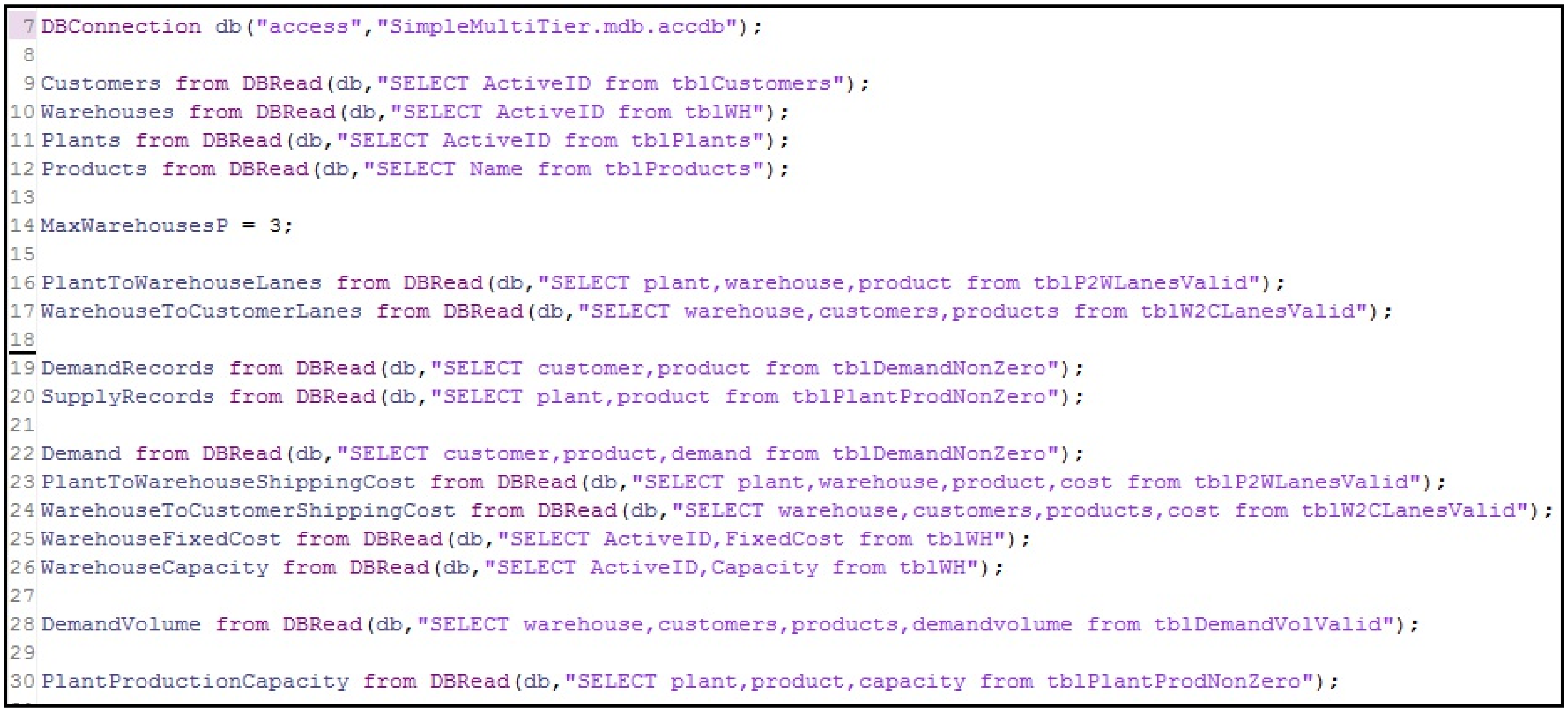
Figure 34: Variable Declarations in SimpleMultiTierMultiProd.mod



```
// we don't use metric variables here, but in practice we would.
minimize
 sum (p in Plants, w in Warehouses, k in Products)
   PlantToWarehouseShippingCost[p][w][k] * PlantToWarehouseShippingVar[p][w][k] +
 sum(w in Warehouses, c in Customers, k in Products)
   WarehouseToCustomerShippingCost[w][c][k]*Demand[c][k]*AssignVar[w][c][k] +
 sum(w in Warehouses)
     WarehouseFixedCost[w] * OpenVar[w];


subject to{

 forall ( c in Customers, k in Products )
   ctEachCustomersDemandMustBeMet:
       sum( w in Warehouses )
         AssignVar[w][c][k]==1;

   ctOnlyUse_P_Warehouses:
   sum(w in Warehouses)
     OpenVar[w]==MaxWarehousesP;

   forall (w in Warehouses, c in Customers, k in Products)
   ctCannotAssignCustomertoWH_UnlessItIsOpen:
     AssignVar[w][c][k] <= OpenVar[w];

   forall (w in Warehouses)
   ctWarehouseCapacityConstraints:
     sum(c in Customers, k in Products)
       Demand[c][k]*AssignVar[w][c][k]*DemandVolume[w][c][k]<= WarehouseCapacity[w]*OpenVar[w] ;

   forall (w in Warehouses, k in Products)
   ctConservationOfFlow:
       sum(p in Plants) PlantToWarehouseShippingVar[p][w][k] ==  sum(c in Customers)Demand[c][k]*AssignVar[w][c][k];

   forall (p in Plants, k in Products)
       ctPlantProductionCapacity:
       sum(w in Warehouses)  PlantToWarehouseShippingVar[p][w][k] <= PlantProductionCapacity[p][k];

}
```

Figure 35: Math Model in SimpleMultiTierMultiProd.mod

For small and moderate data sets, this model is perfectly appropriate, and we include a modest data set with SimpleMultiTier.mdb. This data set has 5 plants, 25 warehouses, 200 customers, and 5 products. We’ve also included it as a database so we can more easily read in the tripled indexed fields. You can see the .dat file in Figure 36.



```
DBConnection db("access","SimpleMultiTier.mdb.accdb");

Customers from DBRead(db,"SELECT ActiveID from tblCustomers");
Warehouses from DBRead(db,"SELECT ActiveID from tblWH");
Plants from DBRead(db,"SELECT ActiveID from tblPlants");
Products from DBRead(db,"SELECT Name from tblProducts");

MaxWarehousesP = 3;

PlantToWarehouseLanes from DBRead(db,"SELECT plant,warehouse,product from tblP2WLanesValid");
WarehouseToCustomerLanes from DBRead(db,"SELECT warehouse,customers,products from tblW2CLanesValid");

DemandRecords from DBRead(db,"SELECT customer,product from tblDemandNonZero");
SupplyRecords from DBRead(db,"SELECT plant,product from tblPlantProdNonZero");

Demand from DBRead(db,"SELECT customer,product,demand from tblDemandNonZero");
PlantToWarehouseShippingCost from DBRead(db,"SELECT plant,warehouse,product,cost from tblP2WLanesValid");
WarehouseToCustomerShippingCost from DBRead(db,"SELECT warehouse,customers,products,cost from tblW2CLanesValid");
WarehouseFixedCost from DBRead(db,"SELECT ActiveID,FixedCost from tblWH");
WarehouseCapacity from DBRead(db,"SELECT ActiveID,Capacity from tblWH");

DemandVolume from DBRead(db,"SELECT warehouse,customers,products,demandvolume from tblDemandVolValid");

PlantProductionCapacity from DBRead(db,"SELECT plant,product,capacity from tblPlantProdNonZero");
```

Figure 36: SimpleMultiTierMultiProd .dat file

However, suppose our high-level data sets have the following cardinalities.

- The customer set *J* has 500 entries, one for each of the 500 largest cities in the US.
- The warehouse set *I* has 100 entries, one for each city where we might realistically locate a warehouse. (Of course, the number of warehouses to locate, *P*, will be far smaller than 100, but the set of potential warehouses is still large.)
- The plant set *L* has 50 entries, as our supply chain draws from a diverse collection of suppliers across North America.
- The product set, *K*, has 1,000 entries. Although this might seem outrageous, 1,000 SKUs is actually a modest entry for a planning model that incorporates a minimum of product aggregation.

Although none of these entries is itself large, the potential number of combinations is huge. For example, we might need to consider 50 * 1,000 = 50,000 product-specific supply points. To insure that the same number of products enter and leave a warehouse, we might need to generate 100 * 1,000 = 100,000 conservation-of-flow constraints. There could be as many as 500 * 1,000 = 500,000 product-specific demand entries. Worst of all are the shipping combinations. We might need to generate 50 * 100 * 1,000 = 5 million plant-to-warehouse shipping variables, and 100 * 500 * 1,000 = 50 million warehouse-to-customer shipping variables. If all of these combinations are fully utilized, the sheer magnitude of our model will be overwhelming.

However, there is a world of difference between a model with large indexing sets and a model that is truly large. A great many of the shipping options can be easily removed through shipping-distance constraints. For each of our 50 plants, only a fraction of the 100 warehouses will be close enough to warrant a direct shipment. Similarly, each warehouse need not consider shipments to all 1,000 customers. We are locating multiple warehouses, so it's unlikely that a warehouse in Los Angeles would service customers in New York.

The opportunity for model reduction is even larger when one considers the localization of products. Any given supplier will almost surely have the capabilities to produce only a fraction of the 1,000 product SKUs. Similarly, a great many of our SKUs likely have regional demand, so a single customer is likely to require only a subset of our 1,000 products.

We incorporate these restrictions into a model with 5 plants, 25 warehouses, and 200 customers by simply not enumerating zero-entry demand or production records. Similarly, we can omit the shipping costs for lanes that are pointlessly long (we didn't in this data set, but could have). We can also omit shipping costs for lanes that would connect a warehouse with either a zero-capacity plant-product combination (that is, a product that doesn't come from that plant) or a zero-demand customer-product combination (a customer that doesn't demand that product). These smaller tables are also seen in the same database, but in other tables. However, the SimpleMultiTierMultiProd.mod data set is unable read to such a restricted data set, because the generic arrays employed by OPL are implicitly complete. Before we alter our SimpleMultiTierMultiProd.mod to read our sparse data set, it is worthwhile to discuss the problems associated with data sparsity independently of data access issues.

Consider a data set that contained all the possible demand point combinations, even if that meant enumerating zero entry demand records. Similarly, consider a data set that enumerated all the shipping combinations, even if that meant using an artificially large shipping cost to enumerate lanes too long to use. (Artificially large objective function coefficients are decidedly *not* good modeling practice—we are generating such a data set to illustrate a point about data sparsity, not numerical rounding.)

As you move from data sets like the ones we provided in the book toward the much larger ones defined immediately above in this section, you will encounter a few problems. The first will likely be memory. The number of shipping variables (even those with large shipping costs) and the number of demand assignment variables (even those with zero demand requirements) will choke off your computer's memory limitations. The first out-of-memory errors will likely occur during pre-processing, as CPLEX requires some memory overhead to perform the processing required to ultimately reduce the model size. As the models become larger, OPL will be unable to pass the model to CPLEX for processing, as even the model formulation will become too large to hold in memory.

The temptation with out-of-memory errors is to simply use larger and larger machines. However, even if running on a capacious 64-bit machine, large and empty data sets can generate run-time problems. The pre-processor will simply require a long time to perform a simple task. Eventually, iterating through all the unused combinations might become a performance problem, although we will probably need at least another layer of data indexing (such as time periods) to discover this particular irritant. This is why data sparsity is a crucial issue separating mathematical formulations from real-world implementations. While an equation modeler can blithely rattle off “for all combinations of source, destination, and product,” a solution developer often must be careful to consider only the data combinations that are truly relevant.

Model SparseMultiTierMultiProd.mod has been rewritten to handle data sparsity in a more intelligent manner. Here we take a minimalist approach, and perform as few adjustments as possible to exploit the sparse data structures provided by OPL. Before we describe these alterations, we first discuss a **`tuple`**, which is a key data structure used by OPL to exploit sparsity.

A tuple is one of the oddly named logical constructions foisted on adult society by computer scientists. Other examples in this category are “memoization” and “big O notation.” Although the term tuple is sometimes used loosely, a tuple is almost always a static, sequenced list. We think of a tuple as being static because a three-element tuple is always a three-element tuple. As a rule, one doesn’t add an element to a three-element tuple, but one can create a new, four-element tuple that extends the data in a given three-element tuple. In some languages, a tuple cannot be altered in any manner whatsoever after it has been created.

In OPL, a tuple is used like a data type such as **`float`** or **`string`**. The multiple elements of the tuple provide flexibility. For example, if you create a two-element tuple called demandRecord with a customer and product as the elements, you can then define a set called DemandRecord that uses the tuple data type demandRecord. (Note that that the capital “D” at the beginning distinguishes this variable from the similarly named tuple.) So, for each entry in the DemandRecord set, we have two elements: a customer and product. You can see the flexibility—whenever we define a demand, we need to know the customer and product. By using a tuple, we are tightly coupling these two elements.

Another way to think about tuples in OPL is to think about the difference between static tuples and dynamic sets and how these entries respond to different data sources. A set (like the {Warehouses} set of strings) will naturally take on different sizes for different input data. A tuple, like demandRecord, will specify its cardinality when it is defined in the .mod file. That means we can adjust the number of warehouses in our optimization model merely by using a different data source. If we change the way a demand record is indexed—for example, adding a time period to the demand record—we will need to create a new .mod file.

Tuples are particularly useful for creating sparse data structures. They are also useful for keeping like data together. In this section, we focus on their use for sparsity. See the Oil sample model in the OPL manuals for an example of tuples used to keep like data together.

Consider the code

```
float plantToWarehouseShippingCost[Plants][Warehouses][Customers]=…;
```

This code specifies that a shipping cost must be provided as input data for every Plant, Warehouse, Customer triplet. Of course, this is exactly the problem, as most triplets aren't required. To address this problem, we create code that will allow us to define just the useful triplets:

```
tuple plantToWarehouseLane

{

    string plant;

    string warehouse;

    string product;

}

{plantToWarehouseLane} PlantToWarehouseLanes = …;
```

First, we define the tuple plantToWarehouseLane. This tuple has three entries, a plant, a warehouse, and a product. These entries are labeled in a legible way. We won't speak of the first entry of a **`plantToWarehouseLane`**; we will instead speak of its **`plant`**. We defined the order of the tuple in the .mod file with the name "plant," so OPL knows how to pull from the correct position.

Once we have defined the **`plantToWarehouseLane`** tuple, we are free to use this data type in the same way we might use a float or a string. Thus, next we can define **`PlantToWarehouseLanes`** as a set of **`plantToWarehouseLanes`**. The curly brackets indicate that we are defining a set, and thus **`{plantToWarehouseLane}`** is a set of **`PlantToWarehouseLanes`**.

(As an aside, you have been using the curly brackets to define sets of data all along. Without the distinction between sets and tuples, we didn't feel the need to emphasize this intuitive concept.)

Although the code snippet

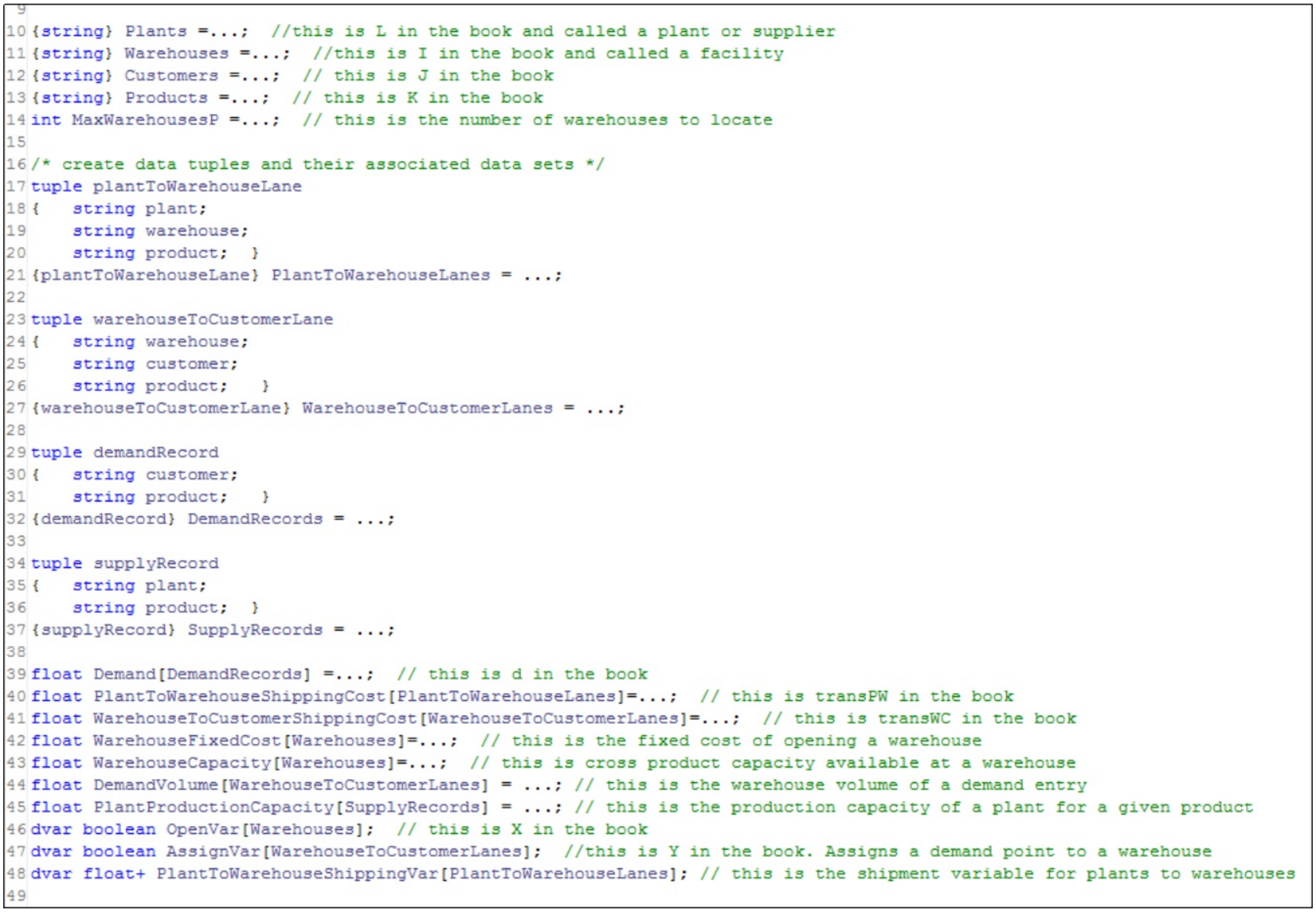
```
{plantToWarehouseLane} PlantToWarehouseLanes = …;
```

might appear as a test of your attentiveness, the similarity between the set name and the tuple name is quite deliberate. It is common programming style for the capitalization of the first letter to carry special significance. Here we only allow tuples to start with lowercase letters, and thus indicate that **plantToWarehouseLane** is more similar to data type definitions like **float** or **boolean** than it is to user-populated data structures like **Demand** or **PlantProductionCapacity**. We hope it won't discourage you to note that in other languages this protocol is exactly reversed. That is to say, C++ programmers often reserve capital starts for class names, and use lowercase starts for variable names. Or, other approaches will start the tuple with a capital T to distinguish it. Applying a consistent standard across all projects is of course futile. However, one can be expected to understand why a standard is applied, and to use it consistently within a project. We are sticking with standard OPL conventions.

Returning to our original goal of exploiting sparsity, we note that our **PlantToWarehouseLanes** set represents the subset of triplets that are appropriate for shipping products from plants to warehouses. We can then use this subset to index our generic arrays of plant-to-warehouse shipping data and variables.

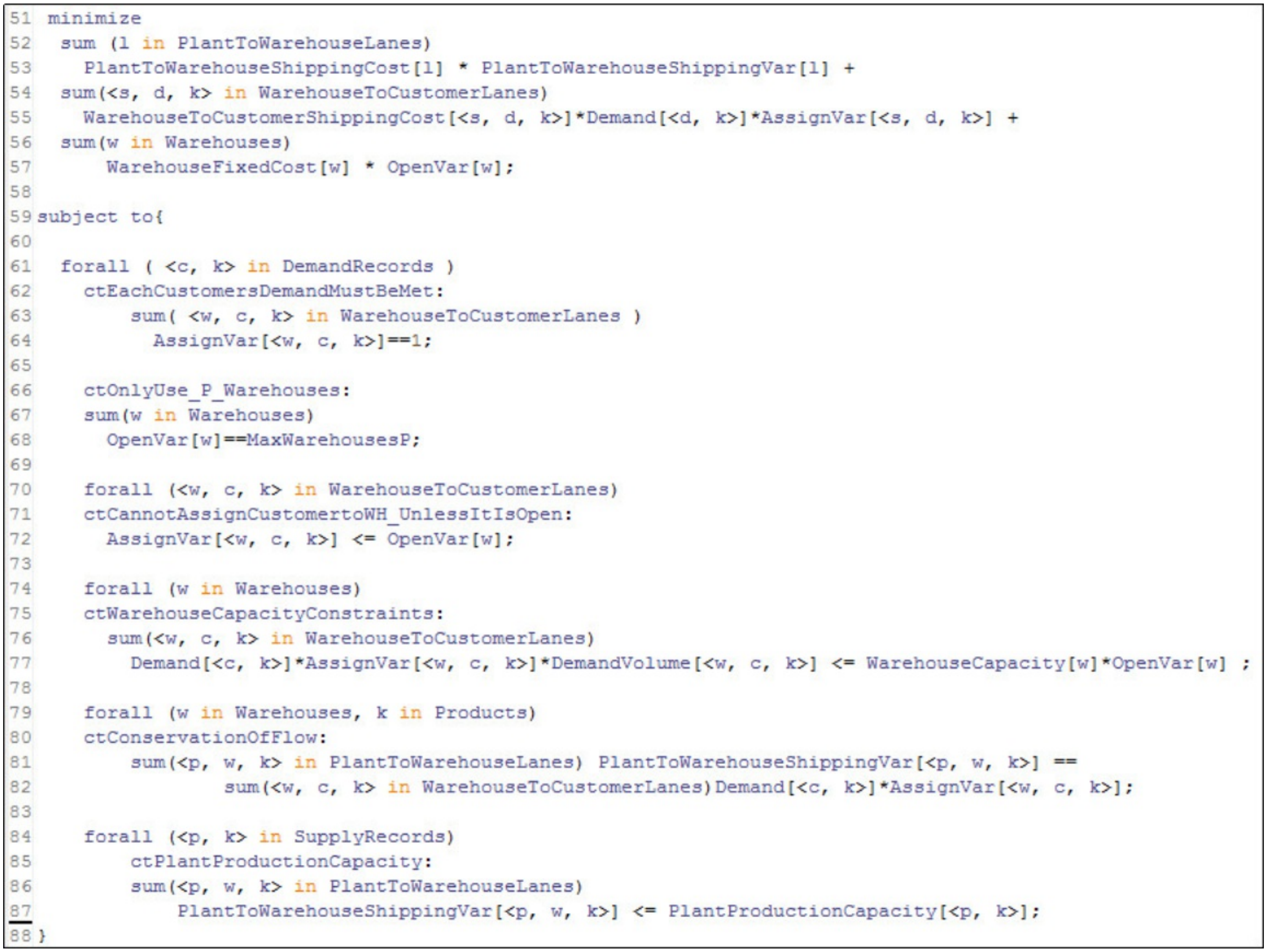
```
float PlantToWarehouseShippingCost[PlantToWarehouseLanes]=…;

dvar float PlantToWarehouseShippingVar[PlantToWarehouseLanes]=…;
```

This pattern is repeated throughout the SparseMultiTierProd.mod file. We create sparse data structures for shipping, production, and demand. This model is shown in Figure 37 and Figure 38.



```
{string} Plants =...;  //this is L in the book and called a plant or supplier
{string} Warehouses =...;  //this is I in the book and called a facility
{string} Customers =...;  // this is J in the book
{string} Products =...;  // this is K in the book
int MaxWarehousesP =...;  // this is the number of warehouses to locate

/* create data tuples and their associated data sets */
tuple plantToWarehouseLane
{   string plant;
    string warehouse;
    string product;  }
{plantToWarehouseLane} PlantToWarehouseLanes = ...;

tuple warehouseToCustomerLane
{   string warehouse;
    string customer;
    string product;   }
{warehouseToCustomerLane} WarehouseToCustomerLanes = ...;

tuple demandRecord
{   string customer;
    string product;   }
{demandRecord} DemandRecords = ...;

tuple supplyRecord
{   string plant;
    string product;  }
{supplyRecord} SupplyRecords = ...;

float Demand[DemandRecords] =...;  // this is d in the book
float PlantToWarehouseShippingCost[PlantToWarehouseLanes]=...;  // this is transPW in the book
float WarehouseToCustomerShippingCost[WarehouseToCustomerLanes]=...;  // this is transWC in the book
float WarehouseFixedCost[Warehouses]=...;  // this is the fixed cost of opening a warehouse
float WarehouseCapacity[Warehouses]=...;  // this is cross product capacity available at a warehouse
float DemandVolume[WarehouseToCustomerLanes] = ...; // this is the warehouse volume of a demand entry
float PlantProductionCapacity[SupplyRecords] = ...; // this is the production capacity of a plant for a given product
dvar boolean OpenVar[Warehouses];  // this is X in the book
dvar boolean AssignVar[WarehouseToCustomerLanes];  //this is Y in the book. Assigns a demand point to a warehouse
dvar float+ PlantToWarehouseShippingVar[PlantToWarehouseLanes]; // this is the shipment variable for plants to warehouses
```

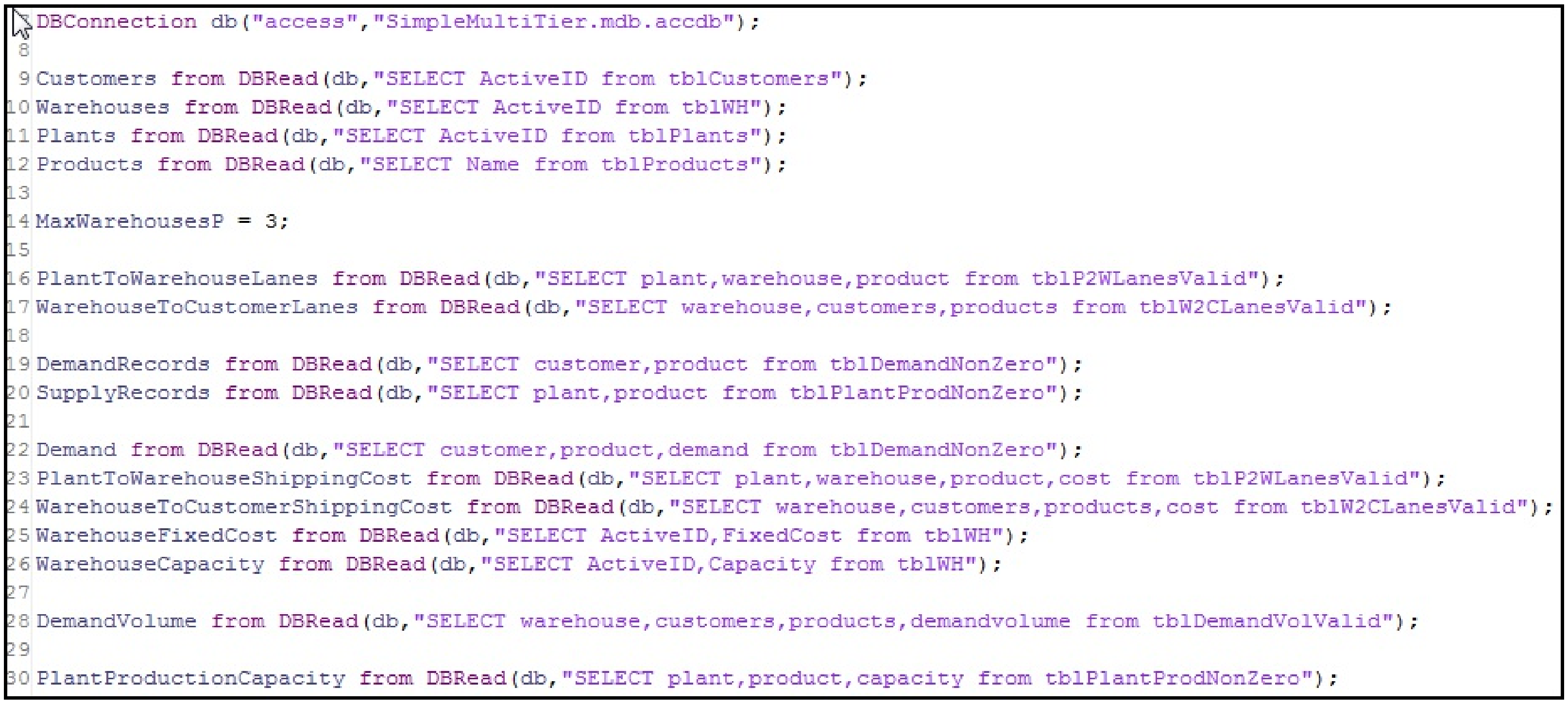
Figure 37: Variable Declarations in SparseMultiTierProd.mod



```
 minimize
  sum (l in PlantToWarehouseLanes)
    PlantToWarehouseShippingCost[l] * PlantToWarehouseShippingVar[l] +
  sum(<s, d, k> in WarehouseToCustomerLanes)
    WarehouseToCustomerShippingCost[<s, d, k>]*Demand[<d, k>]*AssignVar[<s, d, k>] +
  sum(w in Warehouses)
      WarehouseFixedCost[w] * OpenVar[w];

subject to{

  forall ( <c, k> in DemandRecords )
    ctEachCustomersDemandMustBeMet:
        sum( <w, c, k> in WarehouseToCustomerLanes )
          AssignVar[<w, c, k>]==1;

    ctOnlyUse_P_Warehouses:
    sum(w in Warehouses)
      OpenVar[w]==MaxWarehousesP;

    forall (<w, c, k> in WarehouseToCustomerLanes)
    ctCannotAssignCustomertoWH_UnlessItIsOpen:
      AssignVar[<w, c, k>] <= OpenVar[w];

    forall (w in Warehouses)
    ctWarehouseCapacityConstraints:
      sum(<w, c, k> in WarehouseToCustomerLanes)
        Demand[<c, k>]*AssignVar[<w, c, k>]*DemandVolume[<w, c, k>] <= WarehouseCapacity[w]*OpenVar[w] ;

    forall (w in Warehouses, k in Products)
    ctConservationOfFlow:
        sum(<p, w, k> in PlantToWarehouseLanes) PlantToWarehouseShippingVar[<p, w, k>] ==
                sum(<w, c, k> in WarehouseToCustomerLanes)Demand[<c, k>]*AssignVar[<w, c, k>];

    forall (<p, k> in SupplyRecords)
        ctPlantProductionCapacity:
        sum(<p, w, k> in PlantToWarehouseLanes)
            PlantToWarehouseShippingVar[<p, w, k>] <= PlantProductionCapacity[<p, k>];
}
```

Figure 38: Math Model for SparseMultiTierProd.mod

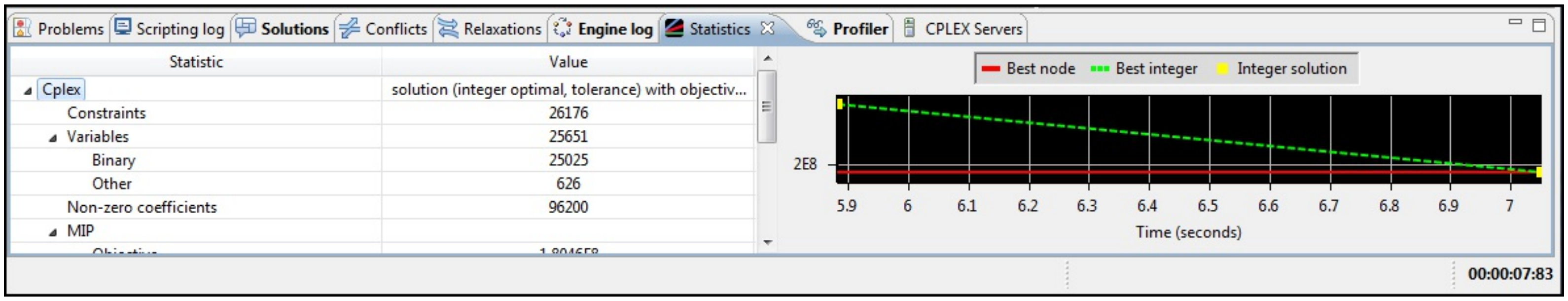
Figure 39 shows the new .dat file to read in the new model with tuples. You should note that we are calling the same database, but different tables. We had to pre-process data to create tables with only the valid entries; we removed the zero records and removed lanes that logically cannot be used. Remember, the underlying problem is exactly the same, mathematically, but the model is now sparse.



```
DBConnection db("access","SimpleMultiTier.mdb.accdb");

Customers from DBRead(db,"SELECT ActiveID from tblCustomers");
Warehouses from DBRead(db,"SELECT ActiveID from tblWH");
Plants from DBRead(db,"SELECT ActiveID from tblPlants");
Products from DBRead(db,"SELECT Name from tblProducts");

MaxWarehousesP = 3;

PlantToWarehouseLanes from DBRead(db,"SELECT plant,warehouse,product from tblP2WLanesValid");
WarehouseToCustomerLanes from DBRead(db,"SELECT warehouse,customers,products from tblW2CLanesValid");

DemandRecords from DBRead(db,"SELECT customer,product from tblDemandNonZero");
SupplyRecords from DBRead(db,"SELECT plant,product from tblPlantProdNonZero");

Demand from DBRead(db,"SELECT customer,product,demand from tblDemandNonZero");
PlantToWarehouseShippingCost from DBRead(db,"SELECT plant,warehouse,product,cost from tblP2WLanesValid");
WarehouseToCustomerShippingCost from DBRead(db,"SELECT warehouse,customers,products,cost from tblW2CLanesValid");
WarehouseFixedCost from DBRead(db,"SELECT ActiveID,FixedCost from tblWH");
WarehouseCapacity from DBRead(db,"SELECT ActiveID,Capacity from tblWH");

DemandVolume from DBRead(db,"SELECT warehouse,customers,products,demandvolume from tblDemandVolValid");

PlantProductionCapacity from DBRead(db,"SELECT plant,product,capacity from tblPlantProdNonZero");
```

Figure 39: Data File to Read the Tuples

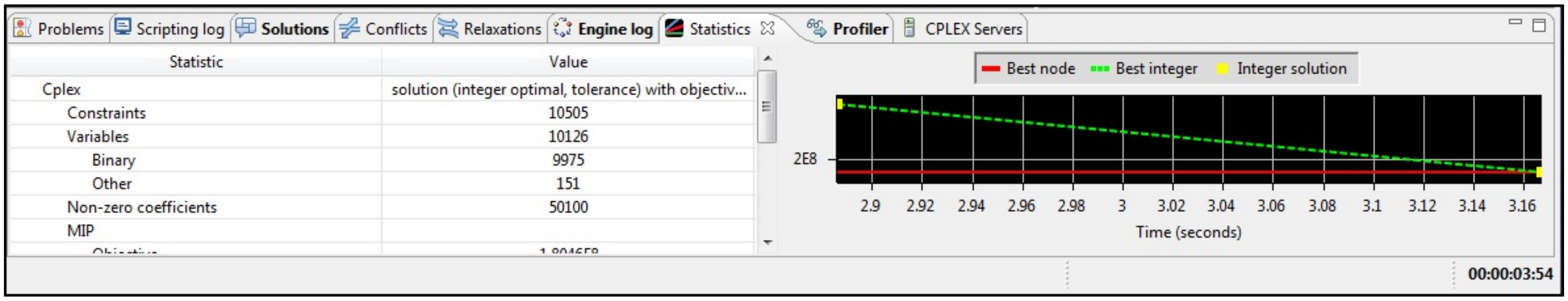
We can run our two models. We will get the same answer (possibly with very trivial rounding). You can see the results in Figure 40 and Figure 41.



Statistic	Value
Cplex	solution (integer optimal, tolerance) with objectiv...
Constraints	26176
Variables	25651
Binary	25025
Other	626
Non-zero coefficients	96200
MIP	

00:00:07:83

Figure 40: Run Results for the Original Model



Statistic	Value
Cplex	solution (integer optimal, tolerance) with objectiv...
Constraints	10505
Variables	10126
Binary	9975
Other	151
Non-zero coefficients	50100
MIP	

00:00:03:54

Figure 41: Run Results Removing Sparsity

You can see the run time in the bottom right corner of these figures. The original model, with the overhead of sparse data (lots of records with a zero value), ran in 7.8 seconds, while the dense formulation (using tuples to remove useless records) ran in 3.5 seconds. You can also see how the number of constraints shrank from 26,176 to 10,505, and the variables decreased from 25,651 to 10,126.

Although both of these models run fine, this shows the order of magnitude of possible improvements. In industrial applications, a run that is doubly improved in run time and the size of the model can make the difference between a model that runs and one that doesn't.

Although SparseMultiTierProd.mod is a big improvement over SimpleMultiTierMultiProd.mod, it still might run into unnecessary bottlenecks. Take this code, for example.

```
forall (w in Warehouses, k in Products)

ctConservationOfFlow:

   sum(<p, w, k> in PlantToWarehouseLanes)
PlantToWarehouseShippingVar[<p, w, k>] == sum(<w, c, k> in
WarehouseToCustomerLanes)Demand[<c, k>]*AssignVar[<w, c, k>];
```

The code stands out as retaining the simplistic "for all pairs" mentality that is characteristic of nonindustrial code. For the sample data that we provide, this isn't a real problem, because the number of warehouse-product pairs that need to be searched isn't terribly vast. However, it is useful to see how this block of code can be made more efficient.

The first step to make this block of code more efficient is to recognize that not all warehouse-product pairs demand a conservation-of-flow constraint. We need such a constraint only if this combination has at least one lane inbound or outbound. (Note that we refrained from saying "has at least one lane inbound *and* outbound." If the Cleveland warehouse can ship cola outbound but cannot receive cola from a supplier, then we need to create a one-sided conservation-of-flow constraint. Otherwise, our model will empower Cleveland to create cola out of thin air.)

So how can we create only those conservation-of-flow equations that are needed? The easiest technique would be to recognize which warehouse-product combinations have an inbound or outbound lane. This can be accomplished quite efficiently via a generic array of sets.

```
{string} WarehouseInbound[w in Warehouses] =

   { k | <p, w, k> in PlantToWarehouseShippingLanes };

{string} WarehouseOutbound[w in Warehouses] =

   { k | <w, c, k> in WarehouseToCustomerShippingLanes };
```

This code might seem a little intimidating, but it's simply combining ideas with which you are already familiar. You've seen both sets and arrays. Here we present an array of sets (two of them, actually). **`WarehouseInbound`** will, given a warehouse, refer to the products for which this warehouse has an inbound lane. Similarly, **`WarehouseOutbound`** will provide quick access to all the possible outbound products for a given warehouse.

We can now iterate over our conservation-of-flow constraints in a far smarter manner.

```
ctConservationOfFlow:

forall (w in Warehouses,

    k in (WarehouseInbound[w] union WarehouseOutbound[w]))

    sum(<p, w, k> in PlantToWarehouseLanes)
    PlantToWarehouseShippingVar[<p, w, k>] ==

    sum(<w, c, k> in WarehouseToCustomerLanes)
    Demand[<c,k>]*AssignVar[<w, c, k>];
```

Clearly, this forall loop is much more efficient. Given a warehouse *w*, we will only consider those products for which there is an inbound or outbound lane. Given the current set of modeled constraints, perhaps we could simply consider only the outbound products. However, it's easy to imagine some future adjustment of this model that includes a minimum shipment requirement inbound to a given warehouse, and thus it's safer to ensure that products neither disappear nor appear at a warehouse.

To summarize, we present a simplified development pattern that can be used to address sparsity concerns in OPL models.

- Identify data tables that are likely to be both large and sparse. Here, sparsity is simply defined as any table that can be "incomplete" without any data being missing. For this model, this means shipping data (where a missing lane means no shipments are possible), demand data (where a missing record implies zero demand), and supply data (where a missing record implies zero supply).
- Create a tuple that matches with the identifier fields of your sparse table. That is to say, a shipping record is identified by its source, destination, and product. A demand record is identified by a customer and product. A supply record is identified by a plant and product.
- Create a set of these tuples to be read from the input data. In essence, we are reading the sparse data table twice—first to read the identifiers, and second, to read the data.
- Use this set of tuples to index any input data or decision variables that mirror the sparse table.

These steps represent a reasonable first pass that is often sufficient to create industrial models from mathematical formulations. If your model persists in bottlenecking outside of the core CPLEX solve process, then the next turn down the rabbit hole likely plays out roughly as follows.

- Identify the nested for-loops that iterate over all combinations (typically in the creation and population of constraints).
- Populate "arrays of sets" to serve as simple subset references. That is to say, `WarehouseInbound[w]` can be used in lieu of `Products` whenever one needs the set of products that are inbound from a given warehouse.
- Reference the appropriate subset instead of the entire indexing set in the inner for-loop(s).

It's very important to emphasize that even this second round of improvements might not be enough to fully industrialize your model. Indeed, OPL is quite a rich language, and there are many other tricks that can be employed to create efficient model-building routines. However, it is worth remembering, "Which way you ought to go depends on where you want to get to." If your goal is to create reasonably efficient code that nevertheless retains a strong resemblance to the purity of mathematical equations, then you will likely work within an "ersatz math" subset of OPL's abilities. If the resulting reasonably efficient OPL code is still dangerously slow, then perhaps achieving a strong resemblance to mathematical purity is simply inappropriate for your final work product. In such an event, an entirely different programming language should perhaps be considered (Python, or perhaps Java or C#, are reasonable fourth-generation languages). The history of software development includes many projects that were rapidly prototyped with one language before ultimately being ported to another. OPL is clearly without peer as a dedicated MIP-rendering language, and it can certainly do far more than mere rapid prototyping, but you shouldn't think of it as the extent of your model-building toolset.

23. Model Run Time and the Optimization Gap

The sample models we have presented in this text have all run reasonably well. This should not give you confidence that all models like this run fast. In fact, we picked data sets and created the formulations so that the run time of the models would not interfere with learning.

But in industrial settings you will likely encounter run-time problems. There are three general reasons your model runs too slowly:

1. The model is very large, and you are pushing the limits of computers and math programming. There is nothing you can do except to develop a new strategy to arrive at an answer; such as breaking the problem into sub-problems.
2. You have not formulated the model in the best possible way. Our section on sparsity addresses some of this. But you can also write the equations in a way that creates long run times. Only practice and studying math programming will help here—this is beyond the scope of our book.
3. You are unnecessarily waiting for the model to converge. You can fix this problem.

In this section we'll focus on the third problem.

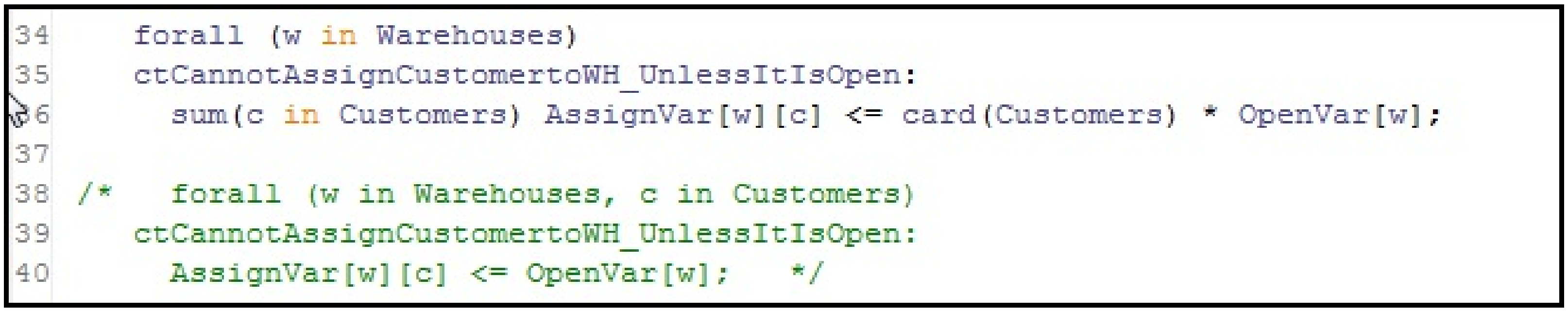
When Mixed Integer Programming problems are solved, the engine works on finding better and better feasible solutions in parallel to solving a series of relaxed versions of the problem. The engine remembers the best feasible solution, and the cost of this solution is the "best known" result to our model. It also remembers the cost of the most accurate relaxation of our model, and this is known as the "best possible" result. (Technically, the "most accurate" relaxation of the original model is usually a series of relaxed models that are collectively exhaustive, but this is an implementation detail that doesn't require a deep understanding on your part.) The difference between the Best Known and Best Possible values is typically expressed as a percentage and called the optimization gap. When these two values converge, you have the proven optimal solution.

When we are solving minimization problems, the "best known" result is sometimes called the upper bound, and the "best possible" result is sometimes called the lower bound. Because this book is (almost) always concerned with minimization models, and given that any maximization model can be easily rewritten as a minimization model, we will use this terminology interchangeably in the remainder of this chapter.

If you think about this carefully, if the upper bound and lower bound are very close, say within 0.01%, you can consider your work done. In the vast majority of cases, there is no need to get the gap down to 0%.

In practice, however, it turns out that most of our instincts lead us to set the gap to 0%. We have seen this in many different projects, but if you want to successfully implement these solutions in practice, you have to break this instinct. First of all, there is a great chance that your data is not accurate to within 0%. If your data is accurate to within 5%, there is no reason to solve to a 0% gap. Secondly, in practice, the amount of time to get from a 50% to a 5% gap is usually much shorter than from 5% to 1%, which, in turn, is shorter than getting from 1% to 0.5%. And after achieving 0.5%, the engine may take hours to go to 0%. Why waste all that time? Third, and finally, a high gap may be caused by a bad lower bound, so the upper bound can still be a very good solution.

To illustrate this, we are going to take the large-scale model we created in Section 1 that was based on the model shown in Figure 4 and artificially make it take longer to run. To make it run longer, you just need to change the constraint family that allows a facility to serve a customer only if it is open. You can see the new constraint in Figure 42. This exercise will also show you the value of writing good constraints.



```
    forall (w in Warehouses)
    ctCannotAssignCustomertoWH_UnlessItIsOpen:
      sum(c in Customers) AssignVar[w][c] <= card(Customers) * OpenVar[w];

/*    forall (w in Warehouses, c in Customers)
    ctCannotAssignCustomertoWH_UnlessItIsOpen:
      AssignVar[w][c] <= OpenVar[w];    */
```

Figure 42: Rewriting a Constraint to Make it Run Slower

The original constraint is shown commented out. The new constraint uses the function **`card`,** which is the cardinality of the set (the size of the set).

If you set up a new run configuration and run this model, you will see that it is noticeably slower. We gave up after about 3 or 4 minutes. During that time, we received a memory warning message from our anti-virus software, not OPL, and the fan kicked on to cool down the processor. You can watch the performance with the Statistics window (Figure 43) that graphically shows the upper and lower bounds converging. And you can look at the Engine Log (Figure 44) and see the best solution in the fourth column from the right, the best lower bound in the third column from the right, and the current optimization gap in the last column on the far left. You may have to do a hard stop (the red button at the top) when you’re tired of watching it run.

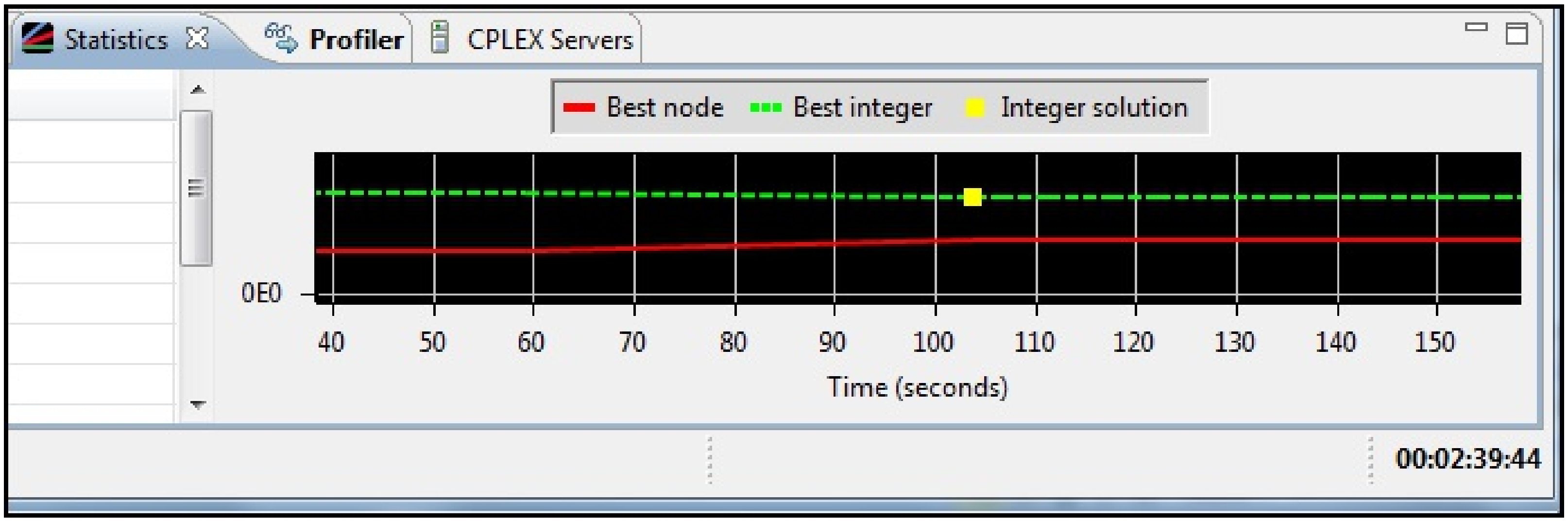
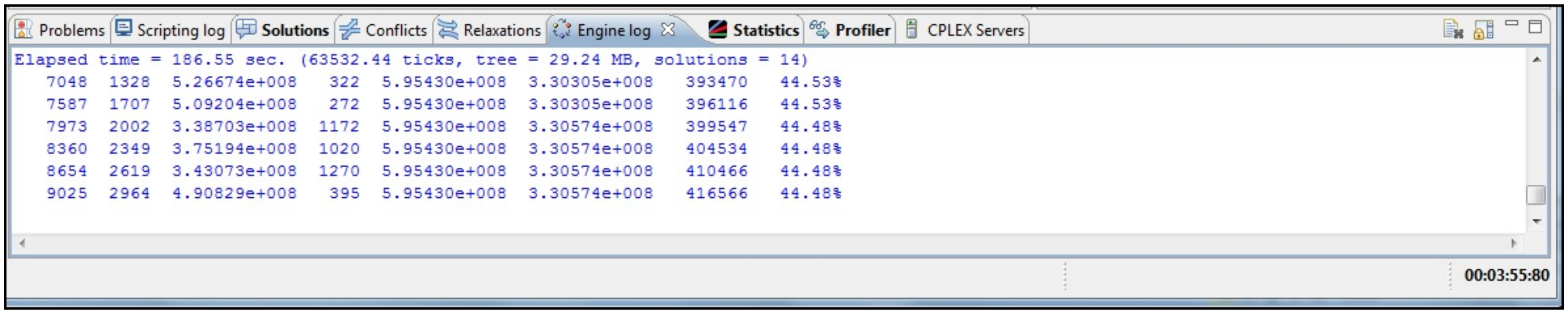


Figure 43: Statistics Window Showing Upper and Lower Bound



```
Elapsed time = 186.55 sec. (63532.44 ticks, tree = 29.24 MB, solutions = 14)
   7048  1328  5.26674e+008   322  5.95430e+008  3.30305e+008   393470   44.53%
   7587  1707  5.09204e+008   272  5.95430e+008  3.30305e+008   396116   44.53%
   7973  2002  3.38703e+008  1172  5.95430e+008  3.30574e+008   399547   44.48%
   8360  2349  3.75194e+008  1020  5.95430e+008  3.30574e+008   404534   44.48%
   8654  2619  3.43073e+008  1270  5.95430e+008  3.30574e+008   410466   44.48%
   9025  2964  4.90829e+008   395  5.95430e+008  3.30574e+008   416566   44.48%
```

Figure 44: Engine Log Showing Solutions and Optimization Gap

In this case, if you look at the best upper bound, it has found a solution with a cost of about $595 million. If you remember this model from before, the optimal solution was about $591 million. Although the gap seems high, we have found a good solution.

Why Is One Model Faster Than Another?

The run-time issue between model A and model B is dramatic as seen in Figure 43. To many people (including people with real technical expertise), it might seem a bit surprising. The explanation for this dramatic change in performance between two seemingly interchangeable models lies in the nuances of the CPLEX engine. While the scope of our text here doesn't allow for a deep investigation of the technical details of such issues, it is worthwhile to provide a layman's survey of the topic.

At their core, MIP engines apply an algorithm with a hope-for-the-best philosophy on integral variables. The engine will initially solve the model as if there were no integrality requirements at all. If this result happens to set all the assignVar and openVar variables to either 0 or 1, then an optimal result has been found. Otherwise, the algorithm will identify at least one assignVar or openVar instance that is "cheating" (i.e., taking on a value that is between 0 and 1) and enforce the integrality rules for that one variable. The algorithm then repeats the process until it discovers a solution that doesn't cheat at all and whose cost is provably optimal or nearly optimal.

This is a very broad summary of the branch and bound algorithm, and is accurate only as a schematic, bird's-eye view of the CPLEX engine. However, it does point to a key implementation detail. The integrality rules are difficult to force onto the optimization process, so CPLEX applies them as sparingly as possible. In the best possible world, the model is such that there really is no incentive for integral variables to "cheat" and take on nonintegral values. (There is a rigorous mathematical check for this property, called unimodularity). More broadly your model performance tends to improve if it can be reformulated so as to minimize the allure of "integrality cheating," even if it cannot be eliminated entirely.

This will all be clearer with an example. Consider a model where we are trying to select the two cheapest warehouses to service four customers. Suppose that the warehouse locations perfectly overlap the customer locations; let the same four locations serve as both potential warehouse sites and as customer demand points, and assume we need to select the two locations that will serve all four demand points with the lowest possible total distance.

Let's see what the optimal results are if "integrality cheating" is allowed for each of our two models. For model A, it should be fairly obvious what the optimal solution will be under such a lax regime. We will simply assign each customer to their local warehouse, and "cheat" by setting openVar = 0.5 for each warehouse. Because of the way ctCannotAssignCustomerToWH_UnlessItIsOpen scales the assignVar, a warehouse open variable only needs to be 0.25 or larger to allow at least one associated assignVar to be 1. You should be able to see for yourself how "cheating" is well rewarded in this case.

Now consider what happens when "cheating" is allowed with Model B. Can we simply assign each customer to its closest warehouse? If we try this, the ctCannotAssignCustomerToWH_UnlessItIsOpen (which is now dedicated to each assignVar) will force the associated warehouse to open completely. If the warehouse is "half opened," then the nearby demand point can be, at most, "half serviced." Since we have a separate constraint that forces exactly two warehouses to be opened, this means that the solution of assigning each demand point to the nearest potential warehouse isn't legal even if "integrality cheating" is allowed.

This is equivalent to saying that integrality cheating is well rewarded in model A, but hardly rewarded at all in model B. In fact, it would be most accurate to say that integrality cheating isn't rewarded at all in model B. We could remove the integrality restrictions for all the assignVar and openVar variables, and it wouldn't change the total cost of the optimal solution for any data set. (It might change the solution that was found, particularly if non-default parameter settings were used, but the cost of the returned solution would remain the same). Thus, model B does exhibit the unimodularity property associated with the utter absence of an "incentive to cheat." Such models tend to enjoy reasonably quick run times, as is the case with our example data here.

However, if we wanted to, we could generate more sophisticated models that still experience a significant improvement when “force a warehouse to be turned on when it is assigned to a demand point” is handled via dedicated constraints (as it is in Model B). In other words, we don’t need an all-or-nothing approach to managing the impact of integrality on our models. As a rule of thumb, any rewrite of our constraints that reduces the allure of non-integrality will tend to help MIP performance. As is the case here, sometimes these rewrites result in a significant increase in the total number of constraints. This is a trade-off well worth making.

Setting the Optimization Gap

In OPL, if you don’t specify the stopping gap, it will use 0.01% as the default; the developers of OPL realize that 0% is numerically difficult and many times not useful.

To control the gap—and you will want to do this for industrial projects—you need to create a settings file (a .ops file) and place it in the run configuration. You can see how to create this setting file in Figure 45.

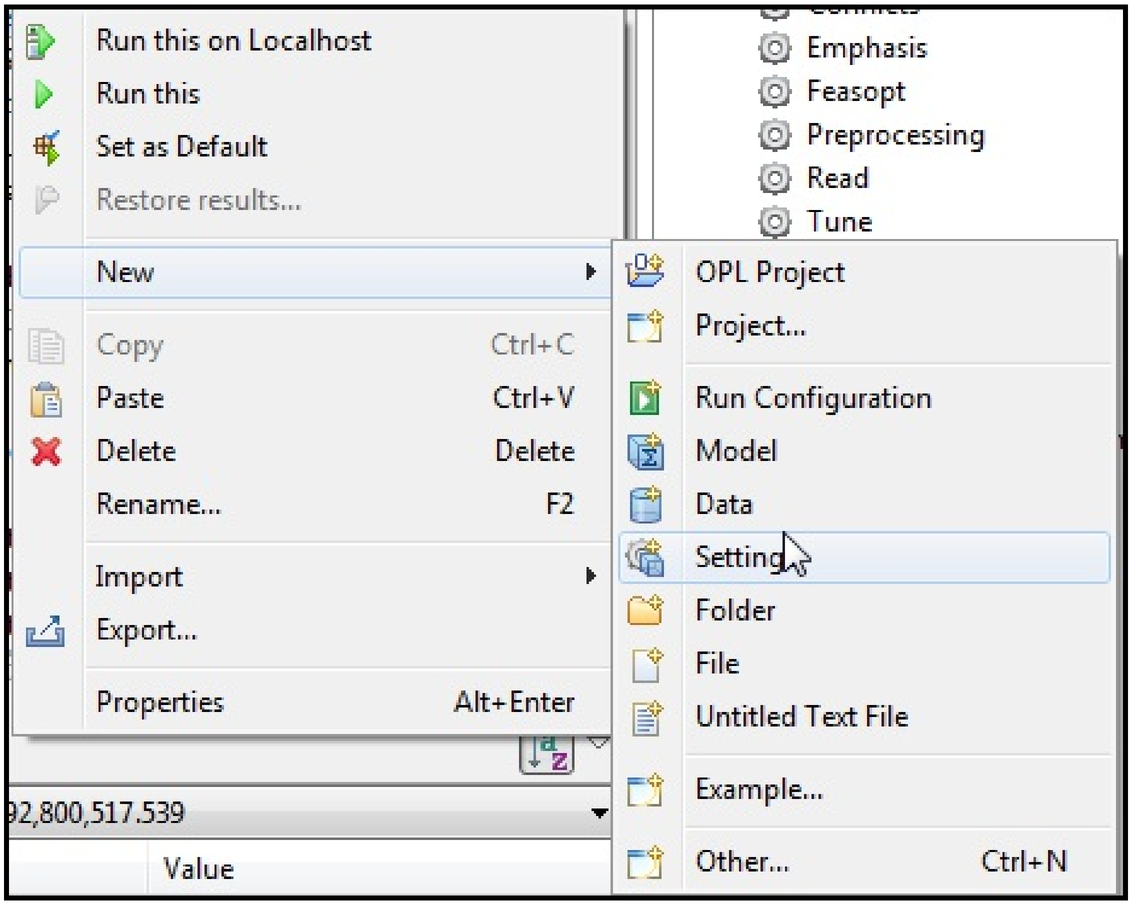
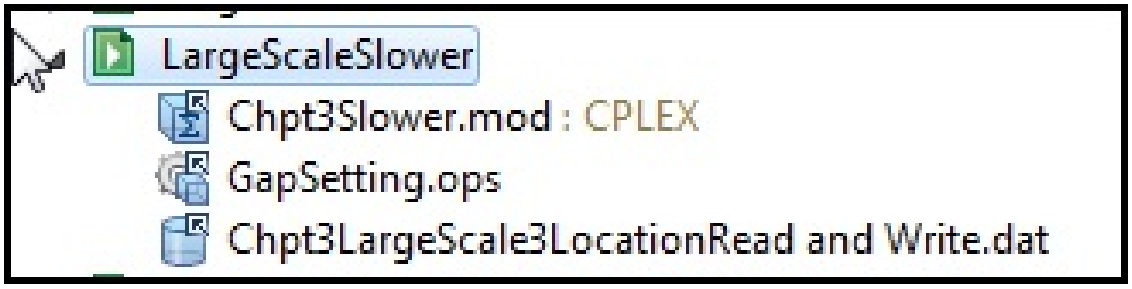


Figure 45: Creating a Settings File

Once you create the settings file, you can place it in the run configuration.


LargeScaleSlower

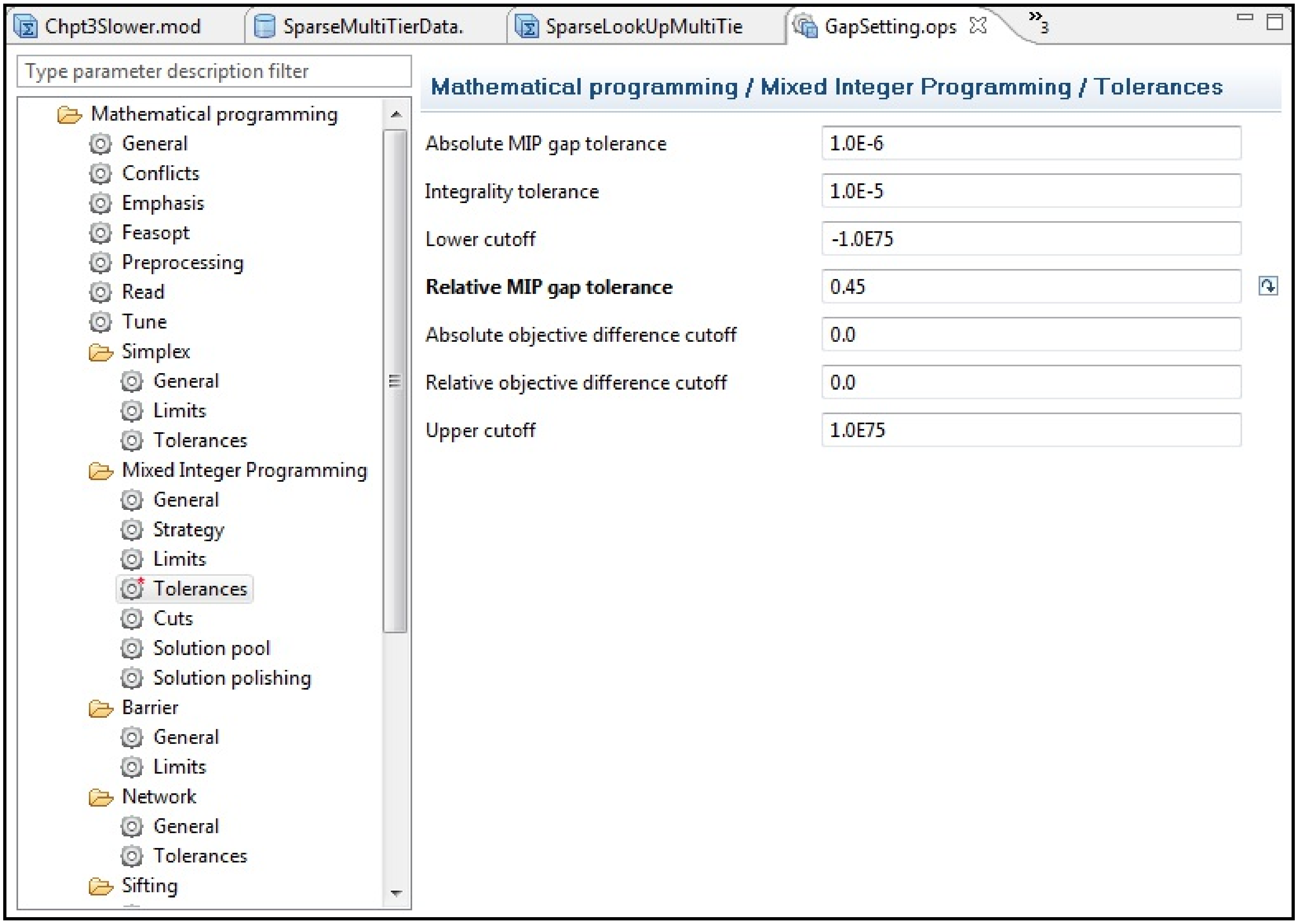
Chpt3Slower.mod : CPLEX

GapSetting.ops

Chpt3LargeScale3LocationRead and Write.dat

Figure 46: Run Configuration with the Settings File

To change the gap, go into the settings file, go to the Mixed Integer Programming folder, and select Tolerances. You can see this in Figure 47. You want to change the "Relative MIP gap tolerance" from 1E-4 (0.01%) to .45 (45%). Then run the model again. This time, the model will automatically stop when it hits the gap and will return an answer.



Mathematical programming / Mixed Integer Programming / Tolerances

Absolute MIP gap tolerance	1.0E-6
Integrality tolerance	1.0E-5
Lower cutoff	-1.0E75
Relative MIP gap tolerance	0.45
Absolute objective difference cutoff	0.0
Relative objective difference cutoff	0.0
Upper cutoff	1.0E75

Figure 47: Changing the Optimization Gap

In the next section, we'll refer to model results as either consistent or inconsistent. Our use of these terms relates directly to upper and lower bound results. Suppose you have two different results, *(upperBound1, lowerBound1)* and, *(upperBound2, lowerBound2)*. We will call these two results consistent if and only if they satisfy both *lowerBound2* <= *upperBound1* and *lowerBound1* <= *upperBound2*.

For example, consider the two different results we demonstrate above. Our original, carefully designed model in Figure 4 quickly returned a result where *upperBound1* = *lowerBound1* = $591 million. Our deliberately crippled model in Figure 44 returned a result where *upperBound2* = $595 million and *lowerBound2* = $330 million. Although these results are not the same, they are consistent. Our first model is obviously more efficient, but, given the results here, we have no reason to believe that our second model will converge on a fundamentally different outcome.

Suppose, for the sake of argument, that the second model terminated with *upperBound2* = $595 million and *lowerBound2* = $594 million. Clearly, the second model must be doing something profoundly different from the first. Whereas the first model has found a solution that costs = $591 million, our second model has deduced that there must not be a solution that improves upon $594 million. Since both models here are supposed to be equivalent, and they are running on the exact same data set, an inconsistent result would indicate a bug. In such an event, we would have to dig more deeply to find the exact problem.

In a situation like that described above, running two different models on the same data set and generating inconsistent results would be a cause for concern. However, we discuss below how the testing of your model leads you to actively seek out inconsistent results. For example, if your model has been newly extended to include plant-to-warehouse transportation costs, you would expect that increasing the plant-to-warehouse shipping costs would cause the overall cost of the optimal solution to increase. Suppose your problems are so computationally challenging that they terminate with an optimization gap of 0.5% or higher on any reasonably complex data set. Assume you run your model on two different data sets. The first, with all zeros for plant-to-warehouse shipping costs, yields *upperBound1 = $100 million and lowerBound1 = $98 million. The second, with meaningful, non-zero plant-to-warehouse shipping costs, yields upperBound2 = $101 million and lowerBound2 = $99.5 million.* Without the verification of an inconsistent result, you have no way of knowing whether this upper bound increase reflects a meaningful change or merely a different computational path being taken on a pair of essentially equivalent MIP problems.

24. Mathematical Programming and Writing Code

Writing code can be a bit like shaving or brushing your teeth—there is more than one way to do it, but it's often assumed to be too simple to discuss. It sometimes seems as if this is particularly true of linear and mixed integer programming.

Although digressions into semantics can border on the pedantic, in this case it is worth pondering the use of the word programming. Computer programming and linear/mixed-integer programming historically invoke a different meaning for the same word. Computer programming describes the art and practice of software engineering. It is concerned with how humans create IT work products that can perform in a manner consistent with other engineering disciplines. That is to say, a study of computer programming will by definition concern itself with best-practice ideas for the creation of reliable, reusable, and maintainable software products. Although not purely a study of people, computer programming will by definition include a concern with (although perhaps not an emphasis on) human-friendly strategies and outcomes.

By contrast, the "program" in linear programming refers to the actual data encoded in the solution vector. Linear programming (which was extended to the more powerful mixed-integer programming) was developed in response to the large-scale logistics problems that were part of the American war effort in the 1940s. The "program" here is the plan being generated, and "programming" refers to our algorithms' ability to perform the sort of scheduling tasks that are roughly equivalent to Julie McCoy's cruise-director responsibilities on *The Love Boat*.

With this etymology, it is perhaps not surprising that textbooks on mixed-integer and linear programming almost never discuss how to actually perform low-bug-count software development. By the literal use of the word, those are not the programs they are looking for. It is partly to fill this void that this particular text was written.

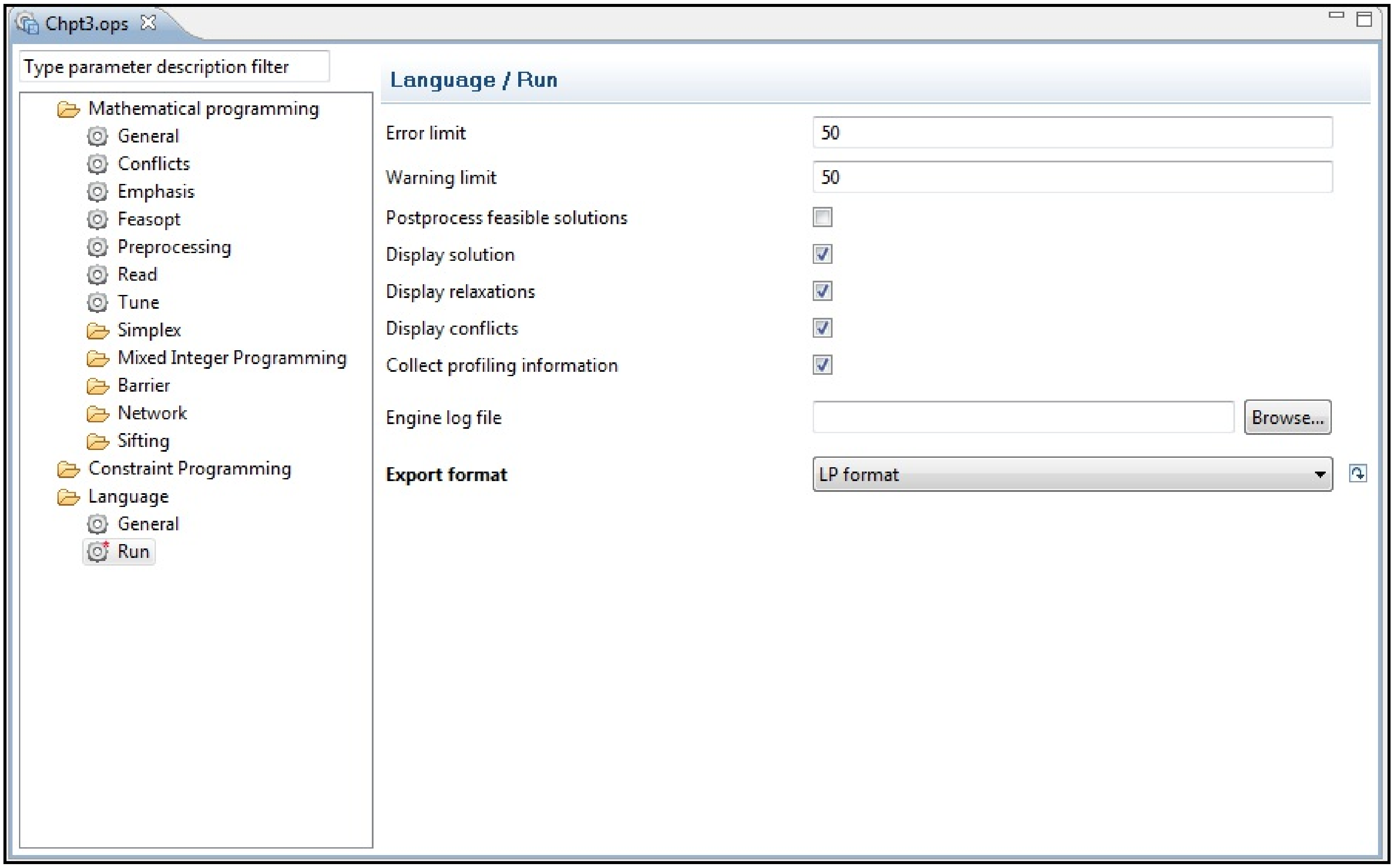
Go Slow to Go Fast

A general rule of thumb for software development is *go slow to go fast*. Your task isn't done when your code is written, or even when it is running. Your task is done when your code has been debugged. Of course, one can argue that a piece of software is never fully debugged, and in some sense that is particularly true for MIPs. (For example, some users like to pretend they are optimizing Warren Buffet's budget and minimize 50-cent purchases for a total budget measuring in the tens of billions. It is almost impossible for your OPL code to be completely bomb-proof with respect to these sort of small-to-large numerical rounding challenges.) However, there is a reasonable standard of debugging that can be achieved. For any model of decent complexity, this task is harder than it looks.

The insight in going slow to go fast is that the amount of time spent coping with a bug is proportional to the lifetime of the bug. The earlier a bug is discovered in the development process, the fewer man-hours will be spent coping with its aftermath. Thus, a workflow that might seem overly cautious or mildly paranoid will often prove itself to be fastest over the long haul.

To this end, it is important to carefully study the "two programs" you are developing. The first is the OPL program itself, and the second is the mixed-integer-programming model that is created by your OPL program. Of course, the whole point of your OPL program is to enable the creation of a family of MIP models. The actual industrial application of your model will likely involve the creation of MIP models that are too large to be human readable. However, you shouldn't eschew the utility of actually studying the MIP models themselves, even if this requires the careful creation of artificial data sets.

Thus, when we consider how to proceed after your .mod and .dat files appear solid, the natural next step is to carefully examine the mixed-integer programs that result from small data sets. You can find the option to export the .lp file in the .ops file as seen in Figure 48. The resulting file can be found in the directory of your project and opened as a text file. This file (named .lp by convention regardless of whether your model has integer variables or not) represents the actual MIP model that will be solved by CPLEX.



Language / Run

Error limit	50
Warning limit	50
Postprocess feasible solutions	☐
Display solution	✓
Display relaxations	✓
Display conflicts	✓
Collect profiling information	✓
Engine log file	
Export format	LP format

Figure 48: OPS File for Parameters

To facilitate human readability, the .lp file might round some numbers. A purely binary .sav can also be generated, and this (along with the parameters represented in the .prm file) is the preferred format for discussing problem models with CPLEX tech support.

For smaller data sets, the .lp file makes for informative reading. This is particularly true for models that use complex techniques to exploit data sparsity. The more elaborate the machinations you employ to efficiently populate your model, the more important it is to test this logic independently of the actual optimization goal. Feel free to create small data sets with human readable data patterns that can be easily validated by studying an .lp file (for example, products named “a1, a2, a3, b1, b2, b3” and customers named “a, b,” with the demand table populated appropriately).

Another good programming maxim is, “You haven’t read your code until you’ve walked through your code.” The act of watching a debugger step line by line through your code while watching the variables change is almost always time well spent. A reasonable analog for OPL development could be this: you haven’t read your model until you’ve read some .lp files. Remember, this sort of precautionary approach doesn’t need to catch bugs every single time to prove its worth. Given the mayhem an escaped bug can wreak, a forced review protocol likely needs to catch bugs on as few as one pass out of ten to be worth doing.

Create More Than Code

It’s a natural misconception that computer programmers primarily write code. Bricklayers lay bricks, lumberjacks fell trees, and programmers write programs. Seems reasonable, right?

In fact, if you are tasked to write an OPL program, your work product should consist of more than .mod and .dat files. In particular, we recommend that you think of MIP development as a three-legged stool, with each leg equally necessary:

1. Model-building code
2. Validation code
3. Test bed of validated data

The tendency is naturally to think of the first leg as the work product, and the other two as optional. This is not a professional way to view software development. Although most of this book is concerned with the first leg (which is, arguably, the most complex task), we feel compelled to emphasize the importance of the other two.

Validation Code

One of the themes of computer science is that validation is easier than construction. Indeed, this concept extends beyond the realm of computer science—the management practice of spot-checking an employee’s work enables a single supervisor to effectively oversee a large staff.

With optimization technology, it is almost always worth your while to validate your results. That is not to say that we think the core CPLEX engine is not trustworthy. However, the “write a program to make a program” development process is certainly challenging, and, as with .lp file review, the time spent developing validation code is almost surely a good investment.

A useful mental exercise is to think of the entire optimization solution as a black box that reads input data from a data store (likely a database) and pushes output data back into this same data store. A useful programming task is to simply validate the output data relative to the input data.

In general, the validation process will likely consist of answering two questions. Does the output data represent a feasible solution? If so, does this solution have an objective function result consistent with what has been reported? These two computational tasks—feasibility and cost—will ideally be performed in a language that is very close to the data store itself.

In the case of our data sets, the good news is that both of these questions can be easily answered with a set of Access database queries. The bad news is that readers unfamiliar with databases might not appreciate the introduction of yet another technology learning curve. Bear in mind that this isn’t a database book, and we aren’t going to dive into a discussion of “left outer join” syntax. (However, there is no shortage of wonderful first database books. Our favorite is http://amzn.to/UZuiAO). We are implementing our validation code as database queries because this is the cleanest and most elegant technique for our purposes. Hopefully a high level tour of our Access queries will demonstrate the concept even to database neophytes. (Note that you don’t have to use Access, but to the general reader, this should be more accessible than other programming languages to do these checks. But, you can certainly do these checks in other languages, including OPL’s Java Script.)

First, we review the result tables.

- tblRsltCustomerAssignments is a primary result table. It stores the warehouse to customer assignments. Note that we deliberately engineered this table so that no more than one warehouse can be assigned to a given customer.
- tblRsltOpenWarehouses is also a primary result table. It stores the list of warehouses opened by the solution.
- tblRsltTotalSKUDist stores total SKU distance of the solution, as reported by CPLEX. Although this result data isn’t technically required, we include it so as to cross-reference the CPLEX-based calculation with the actual solution data in our database.

We can now review the analysis queries:

- qryAssignmentsPerCustomer captures the number of assignments per customer. Those experienced with database queries will notice that we use an outer join to ensure that all customers are included in this query, regardless of whether they are given an assignment.
- qryBadAssignmentsBadReference captures warehouse-to-customer assignments for which no distances have been specified. The assumption here is that a missing distance record should indicate an invalid warehouse-to-customer assignment possibility. (The modeling of such an assumption requires an OPL model capable of exploiting sparsity, but this black-box detail isn't the concern of the validation code. The latter is designed independently of the former.) This query also throws in some more pedestrian integrity references, by ensuring that the customer and warehouse references are valid. Given enough programming experience, you will agree with Andy Grove that "only the paranoid survive."
- qryBadAssignmentsUnopenedWarehouse captures warehouse-to-customer assignments that are associated with unopened warehouses.
- qryCustomersBadlyAssigned ties all of these issues together into one grand query. If this query is non-empty, then you immediately know there is a problem with your result tables.
- qryAssignmentSKUDistance calculates the SKU-distance of each assignment in the result tables.
- qrtRsltTotalSkuDistWrapped is a simple little query to handle the case where tblRsltTotalSKUDist isn't populated.
- qryTotalSKUDistance computes the total SKU-distance from the result tables, and combines it with the expected result that is reported by the CPLEX engine and stored in tblRsltTotalSKUDist. It also computes some utility columns that help the next query.
- qryBadTotalSKUDistance flags bad data in qryTotalSKUDistance. Here we look for one of three conditions:
    1. CPLEX thinks there is a zero-cost solution and our queries compute a solution that is larger than a "small number" tolerance.
    2. Our queries compute a zero total-SKU solution but CPLEX records a "bigger than tiny" cost result.
    3. The proportional difference between our computed result and the CPLEX reported result is larger than our "tiny number" tolerance.

We recognize that we are running on numerically imperfect machines in qryBadTotalSKUDistance. Users who approach optimization from a purely theoretical background sometimes fail to recognize that they live in a fallen world. Readers looking for a reality check are advised to open Windows Calculator, twice take the square root of 7, copy the result into a fresh launch of Calculator, and twice square the contents. MIP engines tackle much harder problems.

So we have boiled down our validation code to two queries. If either qryBadTotalSKUDistance or qryCustomersBadlyAssigned is non-empty, then we know our result tables are problematic. Of course, finding non-empty results in these queries is a necessary but not sufficient precondition for recognizing a debugged solution. However, when combined with a rich test suite, the validation logic can be extremely important to a professional development process.

Test Suite Development

The third leg of our “more than just OPL code” stool is probably the most puzzling. Your goal is to create a debugged OPL model. To support this goal, you need to create a suite of data sets with known solution characteristics, which can then be used to demonstrate the accuracy of your OPL model.

There seems to be an obvious chicken-and-egg problem. How can you create the solutions for your test suite, if not with the OPL model itself? And if the OPL model creates the solutions for each data set, then how can the test suite possibly be useful in finding bugs in the OPL model?

This is where you have to think a bit less like a mathematician and a bit more like a gambler. (Of course, there are those who would argue this should not be a big switch.) The concerns raised above are real, and there is no logic trick that can fully remove them. We must always live with the possibility that our OPL model contains an unknown bug, and our test suite is failing to make that bug apparent. However, we can make this condition rare. Rigorous and archived testing will help us achieve this goal.

There are a few techniques that tend to be useful here. Some of them are based on tried-and-true software development methods. Others are more particular to the nature of optimization. We will work from the general to the specific, first describing the application of software development principles to optimization, and then discussing tricks that are specific to optimization testing.

Build Small and Bootstrap

“Build small” seems like an obvious place to start with your test suite. The act of creating your modeling code tends to encourage the creation of very small data sets. It is a natural human instinct to want to drive around the block with your new car, and this instinct is useful here. A professional modeler who submits a work-product without creating simple data sets that are reasonably amenable to human inspection is not worthy of her title.

However, building small is often not quite as easy as it looks. After all, a complex data model has complex data requirements. Look at the three-echelon model in Chapter 9 (page 165) of *SCND*. This model has so many different components that the task of creating an easy-to-inspect data model is not at all trivial. Even a data set with a handful of plants, warehouses, customers, and products is bound to be complex.

Here is where we benefit from the experience of other software engineers. It's natural to conceive of your OPL development process as consisting of a series of steps.

- Design the model (i.e., write equations on paper or in LaTeX).
- Write the OPL code.
- Test the OPL code.

This workflow roughly follows the waterfall model of software development. It's a natural way to think of the building process: design, construct, and inspect.

However, software construction is unlike physical media, and software developers have often found that a rigid three-step process is not effective. In particular, the problem most apparent here is the challenge of validating something that is both complex and untested. This would all be so much simpler if only one of those clauses applied—for instance, if our model wasn't complex or if it wasn't wholly untested.

In fact, we are under no obligation to take the both-complex-and-untested challenge. Consider the sequence of models in *SCND*. We began with models that were in fact quite simple, and thus relatively easy to test. Each new model extended the functionality of the previous model, so it could use the previous model's test suite (perhaps with some simple data conversions). Thus, a user who is building model X can immediately validate that it behaves in a manner consistent with model X-1. If this consistency fails, then you will have a good starting point for debugging. In this case, try to hone in on the smallest, simplest model that gives inconsistent results in X-1 and X. If this consistency succeeds (i.e., if the entire X-1 test suite gives consistent results when converted and applied to X), then model X is no longer wholly untested.

This sort of bootstrap development is in fact quite common in the world of software. (Indeed, the idea of leveraging simple programs to create complex programs is a theme of computer science, and is arguably a generalization of our preceding theme, that validation is easier than construction.) A common technique in this form of iterative development is called "refactoring." Refactoring can be thought of as creating a stepping stone between step X-1 and step X. When we refactor model X-1, we don't yet add the functionality of model X. Instead, we reorganize the code of X-1 to make the subsequent feature additions easier.

You have already seen an example of refactoring. Consider Sections 1 and 18 of this text, where we discuss model generalization. In this section, we began with a model that was optimizing service level and restricting maximum and average service distance, and we ended with a model capable of creating Pareto optimal solutions of all three objectives. However, we didn't make this jump immediately. Instead, we rewrote our original model to use metric variables while retaining the exact same functionality. This "metric variable" version of the model probably seemed a bit overwrought at first, but it was a useful stepping stone in the creation of the fully generalized model.

"Push" Your Models

We can use bootstrapping to avoid the false challenge of "both complex and untested." We will be testing either very simple models or complex models whose behavior has been validated against a subset of their functionality. While helpful, this bootstrap protocol doesn't remove all our testing challenges. We still must consider new functionality—either features that are introduced by the current iteration or the functionality of a relatively simple model that is completely untested.

There are two approaches to testing new functionality in an optimization model. The first is to use tiny data changes whose effects can be easily validated. Although such models are useful and should be included in your test suite, this approach is limited in its scope. Eventually, you will need to play with data sets whose behavior is difficult to predict.

The mental image we like to use when creating these complex data sets is that of pushing. We can't force a complex model to behave in a fully anticipated manner, but we can push it to go in the correct general direction. We can then inspect the resulting model to ensure that this movement was consistent with our expectations.

An example should make this clear. Imagine we're introducing plants and plant-to-warehouse shipping into a model that previously allowed goods to "appear" at a warehouse. When we converted our test suite to accommodate this new data element, we introduced a variety of zero-cost plants with infinite capacity. We have now validated that our new model generates consistent results with this test suite, and we wish to include models that have plant-production costs. Some of our models are quite tiny, and we can eyeball the correct behavior of a not-free plant production quite easily.

We will now consider pushing one of our larger, more complex models. To do this, we wish to increase the plant production (or equivalently, plant-to-warehouse shipping) costs so as to achieve two goals:

1. The current solution should no longer be an optimal result.
2. The new model should be, on whole, more expensive than the original model.

These two changes aren't really dictated by a specific mathematical theory. However, when combined with the validation code of cost and feasibility, they represent a meaningful way to sanitycheck our new functionality. Although both of these changes are fairly intuitive, there is enough optimization-specific complexity here to warrant further examination.

Suppose our original model opens a warehouse in Cleveland. We adjust our (originally zero) plant-production and shipping costs to make inbound shipments to Cleveland more expensive than other warehouses. When we solve the model with these data changes, Cleveland is no longer opened, and our validation code is correct. Mission accomplished? Not really. Essentially, we want to make sure that our data adjustment is real, and is read and interpreted by our optimization code.

The first step is to ensure that our original, Cleveland-opened solution is no longer optimal. To do this, we need to validate that if this solution is forced upon the adjusted data, it is more expensive than the solution created by our OPL model. Luckily, this check is simple if we've implemented our validation code. Assuming our validation code is implemented as database queries, as in the example above, we can simply examine the "total cost" query result both before and after running our optimization model on the new data set. If this value goes down, then we know that our original solution is suboptimal in the altered model.

Our second step is to ensure that our new data set is, in aggregate, more expensive than our original data set. If our model can simply find a different $10 million solution to each data set, then we can't be completely confident that the optimization model is truly seeing a more-expensive world. Perhaps there is some bug in our .dat file that causes CPLEX to see two MIPs that are fundamentally the same but have different rounding errors, and CPLEX is responding by finding two different solutions that cost the same. Remember, we have only shown that the opening-Cleveland solution is more expensive in our data set, not that it is more expensive in the optimization model.

To this end, we can ensure that the top-line optimization results are inconsistent. We want the "expensive inbound to Cleveland" data set to have a "best possible" result that is more expensive than the "best known" result of the original model. For example, if the original dataset opens Cleveland, and the Cleveland solution costs $10 million (both reported by CPLEX and validated in our Total Cost query), then we want our "expensive inbound to Cleveland" data set to have a "best possible" optimization result that is larger than $10 million. Personally, I look for the new "best possible" result to be at least 0.1% larger, or else we might be looking at a different numerical-rounding result. (To read about the difference between "best known" and "best possible," see Section 1.)

Note that to create this result, we might need to perform some clever data adjustment, to include some revisionist history; this testing step might work better if the original model included particularly cheap warehouse-to-customer shipping costs outbound from Cleveland. Then, when we introduce expensive costs inbound to Cleveland, CPLEX will pick the lesser of two evils and accept a more expensive warehouse-to-customer shipping total to avoid the expense of bringing goods into the Cleveland warehouse. Of course, we are under no restriction that our original data set actually be pre-existing; it need only be a data set that could have been pre-existing. Bootstrap development is our ally here, as we can create whatever X-1 data set is most useful to our model X testing.

Finally, note that for large-scale models and projects, you will want to test programmatically. And the models in your test suite will likely include models that do not quickly (or ever) converge to a 0% optimization gap. In those models, you will want to pull out the “best known” and “best possible” values programmatically to confirm that your models are performing correctly. For the purposes of this text, you can manually inspect these values in the solver log file, as shown in Figure 44.

25. Conclusion

This book was meant to lay a strong foundation so you can build better mathematical models. We went over the basics of building a model and extended this to show the usefulness of bringing lessons from software development to mathematical programming. This is powerful combination that you should keep in mind as you learn more about mathematical programming.

26. Glossary of OPL Commands and Language

This glossary provides a quick review of the key OPL key words. You should see the IBM ILOG CPLEX Studio help files and tutorials for much more information and other ways to accomplish tasks.

File types: .mod designates the file where you write the model, .dat designates the file where you put the data, and .ops contains the CPLEXrameters (this is optional and if you leave it out, CPLEX uses its default parameters).

Comments: Comments across multiple lines start and end with **/*** and ***/** (forward slash asterisk and asterisk forward slash). Comments at the end of the line that do not spill over to the next line start with // (two forward slashes).

Variable types: **`int`** for integer, **`boolean`** for either a 0 or 1, **`float`** for real numbers, and **string** for text.

Declaring a decision variable: Use the keyword **`dvar`** to declare that a variable is a decision variable.

Designating a set: Use the curly brackets, { }, to designate a set. You use these brackets when you declare a set in the .mod file and when you specify the elements of the set in the .dat file.

Designating an array: Use the square brackets, [ ], to designate an array. You use these brackets when you declare a set in the .mod file and when you specify the elements of the set in the .dat file.

Specifying that data for model resides outside the model: The ellipsis points, … (the three dots), are used when you declare an input variable in the .mod file but will assign it values in another location.

Ending an OPL programming line: Use the semicolon (;) to end a command in OPL.

Setting up the objective function: The objective function starts with a keyword minimize or maximize, is followed by the function, and ends with a semicolon.

Setting up constraints: The constraints start with the keywords subject to. All the family of constraints are contained with curly brackets { }. Within the curly brackets, you define each family of constraints with its own name, followed by a colon (:), the function, and ending with a semicolon. If a family of constraints contains many different individual constraints, you use the keyword forall at the beginning of the constraint family. Within the constraint function, you are comparing the right- and left-hand side of the equations with either <, <=, >, >=, or == (double = are used to specify that the two sides of the equation should be equal to each other).

Connecting to Excel: To connect to a spreadsheet, use the term SheetConnection. To read from the spreadsheet, use the command **`from SheetRead.`** To write back to Excel, use the command **`to WriteSheet`**.

If-Then processing of data in the objective or constraints: For input data, you can set restrictions on what data is used. You put the conditions in parenthesis. You have seen this most often with the summation. For example, when you sum over all the warehouses, we write **(w in Warehouses)**. We gave an example of an If-Then written in the following format: **(***Test Condition* **?** *What to do if True* **:** *What to do if False***)**. You write your test condition, follow by a question mark, and separate what to do if true and false by a colon. The model from this guide used: `(Distance[w][c] > MaximumDist?0:1)`.

Validating data: For testing data and ensuring it is clean, you can use the **`assert`** command.

Counting the size of a set: To count the size of a set, you can use the keyword **`card`**.