

Transportation Problem

Another common optimization problem is the transportation problem. In this problem, we're trying to find the best way to ship good from a few warehouses to a lot of stores so that each store meets its demand, but a warehouse doesn't exceed its supply. For now, we assume each warehouse can ship to each store. Additionally, it costs different amounts to ship from each warehouse to each store, so the shipping cost per route must also be taken into account. Our model, then, must minimize costs while ensuring each store has a proper quantity shipped to it without the warehouses running out of goods.

Build the model

We begin by importing the Coopr package and creating an object of the type Model, which we call model.

```
from coopr.pyomo import *  
  
model = Model()
```

The next step is to define our sets that are being acted upon. The warehouses have a supply and the stores have demand, so those two make sense as sets (with supply and demand parameters that rely on them). Each route will have an associated cost, so we could create a set of routes. However, it's more efficient (and easier to read) to make the costs a parameter over two set, the warehouses and the stores, so we will only need to define those two sets.

```
model.warehouses = Set()  
model.stores = Set()
```

As outlined above, we also have several parameters to take into account: supply, demand and costs. We input these similar to how we did it in the diet problem.

```
model.supply = Param(model.warehouses)  
model.demand = Param(model.stores)  
model.costs = Param(model.warehouses, model.stores)
```

Note once again that the costs are a parameter of two dimensions since it takes in two arguments.

We have one final addition to make: the variable to solve for. We're looking for the amount of goods to be sent from each warehouse to each store, so we create a variable over the set of warehouses and the set of stores that determines how much will be shipped from each warehouse to each store. This is slightly different than in the diet problem, where our variable was just over one set, but the difference is minor in the implementation.

```
model.amounts = Var(model.warehouses, model.stores,
    within = NonNegativeReals)
```

We restrict our solutions to be the non-negative reals as otherwise the solution is likely to be illogical and useless—it could involve shipping negative amounts of supplies to gain money, for example.

At this point, we can construct the objective. Remember, the objective is what we’re trying to solve for, which in this case is the minimum total cost. To do that, we take the amount being transported along each route, multiply it with the cost associated with that route, then add all of these values together. Essentially, we take the dot product of the amounts vector and the costs vector, as shown below.

```
def costRule(model):
    return sum(
        model.costs[n,i] * model.amounts[n,i]
        for n in model.warehouses
        for i in model.stores
    )
```

```
model.cost=Objective(rule=costRule)
```

It’s worth looking at what’s happening here a bit more in-depth. The line

```
model.costs[n,i] * model.amounts[n,i]
```

is the dot product outlined above. However, we need to look at every route: starting from every warehouse and ending at every store. Thus, we do two for-loops. What the above syntax does is holds one warehouse constant, then cycles through all the stores, multiplying the costs and amounts for each route. Once it has finished all the stores, it iterates to the next warehouse and cycles through the stores once again. The most important part here, though, is that by doing the double for loop we ensure that every route is accounted for.

Now we must construct our constraints. In this example we only have two, and they are very similar to each other: the minimum demand rule and the maximum supply rule. These constraints will ensure that all the stores be shipped enough material to meet their demand, but no warehouse will exceed its supply. To ensure that we meet our minimum demand, we just need to check that the sum of all the amounts entering a store is greater than or equal to its demand. In code, it looks like this:

```
def minDemandRule(store, model):
    return sum(model.amounts[i, store] for i in model.warehouses)
    >= model.demand[store]
```

```
model.demandConstraint = Constraint(model.stores,
    rule=minDemandRule)
```

By inputting the set “stores” into the constraint, it tells the constraint to run this rule for each element of “stores.” What will happen is the rule will look at each store individually to insure the amount flowing into it exceeds the store’s demands. We construct the supply rule in a similar fashion.

```
def maxSupplyRule(warehouse, model):  
    return sum(model.amounts[warehouse, j] for j in model.stores)  
           <= model.supply[warehouse]  
  
model.supplyConstraint = Constraint(model.warehouses,  
                                   rule=maxSupplyRule)
```

We have now finished constructing our transportation model, which we save as `transportation.py`. (That period is the end of a sentence, not part of the file name). The next step is to create a data file.

Data Entry

We start by defining out sets, warehouses and stores. The notation for this is simple: just put “set [set name] := [elements of set];”. For example

```
set warehouses := quick brown fox;  
set stores := jumps over the lazy dog;
```

Now we need to define the supply and demand parameters on these two sets. Fortunately, they’re done very similarly to each other with the main difference being which set the parameter is indexed over. Otherwise, the notation is simple: put “param: [name of parameter] :=” on the first line, then an element of the set and its associated value on each subsequent line. Don’t forget to end with a semi-colon.

```
param: supply :=  
quick 3000  
brown 7000  
fox 1000;
```

```
param: demand :=  
jumps 3000  
over 3000  
the 4000  
lazy 500  
dog 1000;
```

Finally, we need to define the costs parameter. To do so, we put “param costs:” on the first line (note that there is no colon after “param”), then construct a matrix in a fashion similar to below.

```

param costs:
    jumps over the lazy dog :=
quick  1      3      2      2      1
brown  4      5      1      1      3
fox    2      3      3      2      1;

```

It's worthwhile mentioning that we put the warehouses, the first set fed into the parameter, as column index while the stores, the second set that went into the parameter, is the row index.

Now that we're done inputting the data, save this file as a .dat file. Once that's done it's time to use CoopR to generate a solution.

Solution

On Linux, through the comand line input "pyomo [filename].py [filename].dat". If using the files from the CoopR page, for example, input "pyomo transportation.py transportation.dat" and let the program run.

Pretty quickly the program will crash saying the model is infeasible. What happened? Look back at the data used, specifically the supply and demand data. In total, we have 11000 supply, but we have 11500 demand—our demand exceeds our supply. Thus, it's impossible for this model to work: we can't supply all of the stores to meet their demand. This means the model is infeasible.

When modeling anything, it's worthwhile taking a moment to look at the constraints and data entered. Sometimes a problem exists that renders the whole model useless. Oftentimes, the errors will be less obvious, but it's still good practice to double check the simple things, such as making sure that there is enough supply to meet demand.

For now, we just change the data slightly to make it work. We add 250 to the supply of both quick and fox, so our new supply parameter will be

```

param: supply :=
quick 3250
brown 7000
fox   1250;

```

A quick check tells us that the total supply is 11500, which is the same as the total demand, so this should be a feasible model. We now run pyomo again which gives us this as an output:

```

# =====
# = Solver Results                                     =
# =====

# -----
#   Problem Information
# -----
Problem:

```

```

- Lower bound: 22500
  Upper bound: inf
  Number of objectives: 1
  Number of constraints: 9
  Number of variables: 16
  Number of nonzeros: 31
  Sense: minimize

# -----
#   Solver Information
# -----
Solver:
- Status: ok
  Termination condition: unknown
  Error rc: 0

# -----
#   Solution Information
# -----
Solution:
- number of solutions: 1
  number of solutions displayed: 1
- Gap: 0.0
  Status: optimal
  Objective:
    f:
      Id: 0
      Value: 22500
  Variable:
    amounts[quick,jumps]:
      Id: 2
      Value: 3000
    amounts[quick,over]:
      Id: 3
      Value: 250
    amounts[brown,lazy]:
      Id: 5
      Value: 500
    amounts[brown,the]:
      Id: 6
      Value: 4000
    amounts[brown,over]:
      Id: 8
      Value: 2500
    amounts[fox,over]:
      Id: 13

```

```
Value: 250
amounts[fox,dog]:
Id: 14
Value: 1000
```

The “Value” under “Objective” tells us the solution to the objective we defined in the model file. In this case, that objective was the minimum cost, so we know that \$22500 is the least amount we can spend to move the items from the warehouses to the stores. We can also see how much we should ship along each route. For example, from the warehouse “quick” to the store “jumps” we should ship 3000 units of the item. Similarly, we only ship 250 units from “quick” to “over.”

At this point, we know how much of each item to ship along each route, and that the total cost will be \$22500. In short, we now know the optimal solution to transporting the goods from the warehouses to the stores.