Tutorial IV.V - Modelling the Transportation Problem with JuMP

Energy System Optimization with Julia

Introduction

Welcome to this tutorial on the transportation problem using JuMP! As always, don't worry if you're new to optimization - we'll walk through everything step by step using a real-world example.

Imagine you're running a solar panel distribution company. You have several warehouses (suppliers) and need to ship solar panels to various solar farms (customers). Your goal is to minimize the total cost of transportation while meeting all customer demands.

By the end of this tutorial, you'll be able to:

1. Understand what a transportation problem is
2. Set up a transportation problem using JuMP
3. Solve the problem and interpret the results

Let's start by loading the necessary packages:

```
using JuMP, HiGHS
using DataFrames, CSV
```

Section 1 - Understanding the Transportation Problem

The transportation problem involves:

- Suppliers (our warehouses)
- Customers (solar farms)
- Transportation costs between each supplier and customer
- Supply available at each warehouse
- Demand required by each solar farm

Our goal is to decide how many solar panels to ship from each warehouse to each solar farm to minimize total cost.

Let's set up our problem:

- The revenue from each truckload of solar panels is 11000
- The variable costs from each truckload of solar panels is 6300
- The available panels at the supplier are given in the file `available-panels.csv`
- The requested panels at the customer are given in the file `panel-demand.csv`
- The transportation costs between suppliers and customers are given in the file `cost.csv`

> **Note**
>
> We can use the `CSV.read` function to load the data from a CSV file into a DataFrame. If we want to access the directory of the current file, we can again use the convinient `@__DIR__` macro.

```
# Fixed parameters
revenue = 11000  # Revenue per truckload of solar panels
varCosts = 6300  # Variable costs per truckload

# Load data from CSV files
available = CSV.read("$(@__DIR__)/data/available-panels.csv", DataFrame)
requested = CSV.read("$(@__DIR__)/data/panel-demand.csv", DataFrame)
travelCosts = CSV.read("$(@__DIR__)/data/cost.csv", DataFrame)

println("Data loaded successfully!")
```

```
Data loaded successfully!
```

> **Tip**
>
> Make sure, that you donwload the datasets from Github and store them in a folder called `data` in the same directory as the script you are currently working on. You can find the data sets in the GitHub repository for this tutorial. Note, that you don't need to preprocess the data in any way. This tutorial we will focus on the modeling part.

Now, we can check out the data by printing the first few rows of each DataFrame. We can use the `first` function to get the first few rows of a DataFrame.

```
println("Available panels:")
first(available,5)
```

```
Available panels:
```

	supplier	truckloads_available
	String7	Int64
1	a_1	478
2	a_2	371
3	a_3	361
4	a_4	415
5	a_5	354

```
println("Requested panels:")
first(requested,5)
```

```
Requested panels:
```

	solar_farm	truckload_demand
	String7	Int64
1	b_1	9
2	b_2	31
3	b_3	47
4	b_4	46
5	b_5	12

```
println("Travel costs:")
first(travelCosts,5)
```

```
Travel costs:
```

	supplier	solar_farm	costs
	String7	String7	Int64
1	a_1	b_1	8052
2	a_2	b_1	11845
3	a_3	b_1	6103
4	a_4	b_1	11099
5	a_5	b_1	5625

Exercise 1.1 - Understand the Data

Take a moment to look at the data. Can you answer these questions?

1. How many warehouses do we have? Save the number in a variable called `num_warehouses`.
2. How many solar farms are we supplying? Save the number in a variable called `num_solar_farms`.

```
# YOUR ANSWERS BELOW
# Hint: Use the `nrow()` function to count rows
```

```
# Test your understanding
@assert num_warehouses == nrow(available)
@assert num_solar_farms == nrow(requested)

println("Great job! Here are the answers:")
println("Number of warehouses: ", num_warehouses)
println("Number of solar farms: ", num_solar_farms)
```

Section 2 - Using dictionaries to store the data

Now, DataFrames are not a very convenient format for our purposes. We have several options now on how to deal with these data sets. As our suppliers and customers are given names, it might be useful to convert the data into dictionaries. Dictionaries are a great data structure that allow us to store key-value pairs, where the keys are unique identifiers and the values are the data associated with those keys. By using dictionaries, we can easily access and modify the data associated with a specific key.

```
available_dict = Dict(
    available.supplier .=> available.truckloads_available
)
requested_dict = Dict(
    requested.solar_farm .=> requested.truckload_demand
)
travelCosts_dict = Dict(
    (row.supplier,row.solar_farm) => row.costs
    for row in eachrow(travelCosts)
)
```

> Tip
>
> You can use the `Dict` function to create a dictionary from two arrays or DataFrames. For example: `Dict(keys .=> values)` will create a dictionary where the keys are the elements of the `keys` array and the values are the elements of the `values` array.

Now, let us check out the dictionaries. We can use the `first` function to get the first few key-value pairs of a dictionary.

```
println("Available panels:")
first(available_dict,5)
```

```
Available panels:
```

```
5-element Vector{Pair{String7, Int64}}:
 "a_13" => 145
 "a_17" => 181
 "a_26" => 405
 "a_90" => 479
 "a_67" => 430
```

```
println("Travel costs:")
first(travelCosts_dict,5)
```

```
Travel costs:
```

```
5-element Vector{Pair{Tuple{String7, String7}, Int64}}:
 ("a_33", "b_450") => 1903
 ("a_99", "b_340") => 1749
 ("a_74", "b_249") => 7016
 ("a_11", "b_278") => 5788
  ("a_40", "b_35") => 11369
```

Remember, we can also access the value associated with a specific key in a dictionary by using the key inside square brackets. For example: `available_dict["a_1"]` will return the value associated with the key `"a_1"`.

```
print("Value associated with supplier 'a_1': ")
available_dict["a_1"]
```

```
Value associated with supplier 'a_1':
```

```
478
```

Our travel costs dictionary is a bit more complex, as it is dictionary with tuples as keys. We can access the value associated with a specific supplier and customer by using two keys inside square brackets. For example: `travelCosts_dict[("a_1","b_1")]` will return the value associated with the keys `("a_1","b_1")`.

```
print("Value associated with supplier 'a_1' and customer 'b_1': ")
travelCosts_dict[("a_1","b_1")]
```

```
Value associated with supplier 'a_1' and customer 'b_1':
```

```
8052
```

We can also extract the keys and values of a dictionary using the `keys` and `values` functions, as shown in the previous tutorial.

```
println("Keys of the travel costs dictionary:")
first(keys(travelCosts_dict),5)
```

```
Keys of the travel costs dictionary:
```

```
5-element Vector{Tuple{String7, String7}}:
 ("a_33", "b_450")
 ("a_99", "b_340")
 ("a_74", "b_249")
 ("a_11", "b_278")
 ("a_40", "b_35")
```

```
println("Values of the travel costs dictionary:")
first(values(travelCosts_dict),5)
```

```
Values of the travel costs dictionary:
```

```
5-element Vector{Int64}:
  1903
  1749
  7016
  5788
 11369
```

Dictionaries make it a lot easier to access the data later on, as we can use the keys to directly access the desired value in our model. This will be useful when we want to define the constraints later on.

Section 3 - The model instance

After the preprocessing and data loading, we now can create the model instance with the HiGHS optimizer.

Exercise 3.1 - Creating the model instance

From the last tutorial, you should know how to do this. Create a model instance called `transport_model` and set the optimizer to HiGHS.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert typeof(transport_model) == JuMP.Model
println("Model instance created successfully!")
```

Section 4 - Defining the model

Define the variables

We can now define the variables of our model. We need to define a variable for each supplier and customer pair. As before, we can use the `@variable` macro to define the variables. The syntax is `@variable(model, varname[index1,index2] >= 0)`, where `model` is the model instance, `varname` is the name of the variable, and `index1` and `index2` are the indices of the variable. We can use vectors as input for the indices, but we could also use the keys of the dictionaries. In the following code block we mixed both options, to show you that it is possible.

```
# Define variable
@variable(
    transport_model,
    X[available.supplier,keys(requested_dict)] >= 0
)
```

Define the objective

Next, we can define the objective of our model. We want to maximize the profit, which is the revenue minus the variable costs and the transportation costs. As before, we can use the `@objective` macro to define the objective. The syntax is `@objective(model, Max, expression)`, where `model` is the model instance, `Max` indicates that we want to maximize the expression, and `expression` is the expression we want to maximize.

```
@objective(transport_model, Max,
    sum((revenue-varCosts-travelCosts_dict[(i,j)]) * X[i,j]
    for i in keys(available_dict), j in keys(requested_dict))
)
```

Define the constraints

We can now define the constraints of our model. We need to ensure that the supply from each supplier is enough to cover the demand of each customer. We can use the `@constraint` macro to define the constraints. The syntax is `@constraint(model, expression)`, where `model` is the model instance and `expression` is the expression we want to constrain.

To illustrate the use of dictionaries, we will again use the keys of the dictionaries to define the constraints in the following code block.

```
@constraint(transport_model,
    restrictAvailable[i in keys(available_dict)],
```

```
    sum(X[i,j] for j in keys(requested_dict)) <= available_dict[i]
)
```

Naturally, we could also use the vectors with the names from the DataFrames to define the constraints or we could also just work with ranges from the beginning, e.g. `1:length(available.supplier)` and `1:length(requested.solar_farm)`. Working with names is often more convenient, though.

```
@constraint(transport_model,
    restrictDemand[j in requested.solar_farm],
    sum(X[i,j] for i in available.supplier) <= requested_dict[j]
)
```

And that's it! We have now defined the model and can start optimizing.

Section 5 - Solving the model

Exercise 5.1 - Start optimization

Start the optimization as usual by calling the `optimize!` function on the model instance.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert termination_status(transport_model) == MOI.OPTIMAL
println("Model optimized successfully!")
```

Now, we can access the values of the variables at the optimal solution. But remember, we defined the variables with the keys of the dictionaries, so we need to convert the result back to a DataFrame. Calling the variable itself will just show the structure of the variable, not the values.

```
first(X,5)
```

Thus, we need to use the `value` function to extract the values from the variable.

```
transport_values = value.(X)
```

The result is a `DenseAxisArray{Float64,2,...}` with index sets. To convert it to a DataFrame, we just need to iterate over the keys dictionaries and store the values in a new DataFrame. As we are not interested in values which are zero, we can skip those.

First, we need to initialize an empty DataFrame with the correct column names.

```
transport_df = DataFrame(
    supplier = [],
    solar_farm = [],
    truckloads = []
)
```

	supplier	solar_farm	truckloads
	Any	Any	Any

Then, we can iterate over the keys of the dictionaries and store the values in the DataFrame if they are greater than zero.

```
for i in keys(available_dict)
    for j in keys(requested_dict)
        if transport_values[i,j] > 0
            push!(transport_df, (
                supplier = i,
                solar_farm = j,
                truckloads = transport_values[i,j]
```

```
                )
            )
        end
    end
end
```

Finally, we can print the first few rows of the transportation plan to check if it looks correct.

```
println("Begining of the transportation plan:")
first(transport_df,5)
```

> **Note**
>
> Although the above code looks rather complicated, it is essentially just iterating over the keys of the dictionaries and storing the values in a new DataFrame. This is a common pattern in optimization, as we often want to convert the result of an optimization problem into a more convenient format for reporting or further processing.

Conclusion

In this tutorial, we have learned how to model and solve the transportation problem using JuMP. We have also learned how to use dictionaries to store and access the data, which will be useful for more complex models. If you have any questions, feel free to ask me via email!

Solutions

You will likely find solutions to most exercises online. However, I strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them in next week's tutorial. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.