Tutorial V - Production Planning in Breweries

Applied Optimization with Julia

1. Modelling the CLSP

Load the necessary packages and data

Implement the CLSP from the lecture in Julia. Before we start, let's load the necessary packages and data.

```
using JuMP, HiGHS
using CSV
using DelimitedFiles
using DataFrames
using Plots
using StatsPlots
import Pkg; Pkg.add("PlotlyKaleido")
plotly() # This will create interactive plots later on
```

> **Tip**
>
> If you haven't installed the packages yet, you can do so by running `using Pkg` first and then `Pkg.add("JuMP")`, `Pkg.add("HiGHS")`, `Pkg.add("DataFrames")`, `Pkg.add("Plots")`, and `Pkg.add("StatsPlots")`.

Load the data

Now, let's load the data. The weekly demand in bottles $d_{i,t}$, the available time at the bottling plant in hours a_t, the time required to bottle each beer in hours b_i, and the setup time in hours g_i are provided as CSV files.

```
# Get the directory of the current file
file_directory = "$(@__DIR__)/data"

# Load the data about the available time at the bottling plant
availableTime = CSV.read("$file_directory/availabletime.csv", DataFrame)
println("Number of periods: $(nrow(availableTime))")
println("First 5 rows of available time per period:")
println(availableTime[1:5, :])
```

```
Number of periods: 27
First 5 rows of available time per period:
5×2 DataFrame
```

```
 Row │ period   available_capacity
     │ String7  Int64
─────┼──────────────────────────────
   1 │ week_01                 168
   2 │ week_02                 168
   3 │ week_03                 168
   4 │ week_04                 168
   5 │ week_05                  48
```

```
# Load the data about the bottling time for each beer
bottlingTime = CSV.read("$file_directory/bottlingtime.csv", DataFrame)
println("Number of beers: $(nrow(bottlingTime))")
println("Bottling time per beer:")
println(bottlingTime)
```

```
Number of beers: 6
Bottling time per beer:
6×2 DataFrame
 Row │ beer_type   bottling_time
     │ String15    Float64
─────┼──────────────────────────
   1 │ Pilsener          0.00222
   2 │ Blonde_Ale        0.00111
   3 │ Amber_Ale         0.00139
   4 │ Brown_Ale         0.00222
   5 │ Porter            0.00167
   6 │ Stout             0.00111
```

```
# Load the data about the setup time for each beer
setupTime = CSV.read("$file_directory/setuptime.csv", DataFrame)
println("Setup time per beer:")
println(setupTime)
```

```
Setup time per beer:
6×2 DataFrame
 Row │ beer_type   setup_time
     │ String15    Int64
─────┼──────────────────────
   1 │ Pilsener           10
   2 │ Blonde_Ale         11
   3 │ Amber_Ale           8
   4 │ Brown_Ale           8
   5 │ Porter             11
   6 │ Stout               9
```

```
# Load the data about the weekly demand for each beer
demandCustomers = CSV.read("$file_directory/demand.csv", DataFrame)
println("First 5 rows of demand per beer:")
println(demandCustomers[1:5, :])
```

```
First 5 rows of demand per beer:
5×3 DataFrame
 Row │ beer_type   period   demand
     │ String15    String7  Int64
─────┼──────────────────────────────
   1 │ Pilsener    week_01    3853
   2 │ Blonde_Ale  week_01    8372
   3 │ Amber_Ale   week_01   16822
   4 │ Brown_Ale   week_01   13880
   5 │ Porter      week_01   10642
```

Define the parameters

Consider in your implementation, that each hour of setup is associated with a cost of 1000 Euros, and the inventory holding cost for unsold bottles at the end of each period is 0.1 Euro per bottle. Implement both parameters for the cost of setup and the inventory holding cost in the model. Call them `setupHourCosts` and `warehouseCosts`.

```
# YOUR CODE BELOW
```

Next, you need to prepare the given data for the model. Create a dictionary for the available time, bottling time, and setup time. Call them `dictAvailableTime`, `dictBottlingTime`, and `dictSetupTime`.

> **i** Note
>
> Let's understand what's happening with `dictDemand` in the code. It creates a dictionary where:
>
> - Keys are tuples `(beer_type, period)`, e.g., `("IPA", "week_1")`
> - Values are the demand numbers for that beer in that period
>
> You can use this pattern to create dictionaries for:
>
> - `dictAvailableTime`: Maps each period (single key) → available time
> - `dictBottlingTime`: Maps each beer type (single key) → bottling time per bottle
> - `dictSetupTime`: Maps each beer type (single key) → setup time
>
> Notice that these dictionaries have single keys (just the period or just the beer type), not tuple keys like `dictDemand`.

```
# Prepare the data for the model
dictDemand = Dict((row.beer_type,row.period) => row.demand for row in
eachrow(demandCustomers))

# YOUR CODE BELOW
```

```
# Validate your solution
@assert length(dictAvailableTime) == nrow(availableTime) "Available time
dictionary should have same length as input data"
@assert length(dictBottlingTime) == nrow(bottlingTime) "Bottling time
dictionary should have same length as input data"
@assert length(dictSetupTime) == nrow(setupTime) "Setup time dictionary
should have same length as input data"

# Check that all values are positive
@assert all(v -> v > 0, values(dictAvailableTime)) "All available time
values must be positive"
@assert all(v -> v > 0, values(dictBottlingTime)) "All bottling time values
must be positive"
@assert all(v -> v > 0, values(dictSetupTime)) "All setup time values must
be positive"

# Check that dictionaries contain all expected keys
@assert all(p -> haskey(dictAvailableTime, p), availableTime.period)
"Missing periods in available time dictionary"
@assert all(b -> haskey(dictBottlingTime, b), bottlingTime.beer_type)
"Missing beer types in bottling time dictionary"
@assert all(b -> haskey(dictSetupTime, b), setupTime.beer_type) "Missing
beer types in setup time dictionary"
```

Define the model instance

Next, we define the model instance for the CLSP.

```
# Prepare the model instance
lotsizeModel = Model(HiGHS.Optimizer)
set_attribute(lotsizeModel, "presolve", "on")
set_time_limit_sec(lotsizeModel, 60.0)
```

Define the variables

Now, create your variables. Please name them `productBottled` for the binary variable, `productQuantity` for the production quantity and `WarehouseStockPeriodEnd` for the warehouse stock at the end of each period. We will use these names later in the code to plot the results.

Tip

When creating your variables, you'll have to create one variable for each combination of beer type and period. You can use the dictionary keys directly in the variable definition:

```
@variable(model, x[keys(dictBottlingTime), keys(dictAvailableTime)])
```

This creates variables indexed by:

- First index: beer types (from `dictBottlingTime`)
- Second index: periods (from `dictAvailableTime`)

So you'll have variables like `x["IPA", "week_1"]`, `x["IPA", "week_2"]`, etc.

```
# YOUR CODE BELOW
```

```
# Validate your solution
# Check if variables exist in the model
@assert haskey(lotsizeModel.obj_dict, :productBottled) "productBottled
variable not found in model"
@assert haskey(lotsizeModel.obj_dict, :productQuantity) "productQuantity
variable not found in model"
@assert haskey(lotsizeModel.obj_dict, :WarehouseStockPeriodEnd)
"WarehouseStockPeriodEnd variable not found in model"

# Check variable dimensions
@assert length(productBottled) == length(dictBottlingTime) *
length(dictAvailableTime) "Incorrect dimensions for productBottled"
@assert length(productQuantity) == length(dictBottlingTime) *
length(dictAvailableTime) "Incorrect dimensions for productQuantity"
@assert length(WarehouseStockPeriodEnd) == length(dictBottlingTime) *
length(dictAvailableTime) "Incorrect dimensions for
WarehouseStockPeriodEnd"

# Check variable types
@assert all(is_binary, productBottled) "productBottled must be binary
variables"
@assert all(is_integer, productQuantity) == false "productQuantity must be
continuous variables"
@assert all(is_integer, WarehouseStockPeriodEnd) == false
"WarehouseStockPeriodEnd must be continuous variables"
```

Define the objective function

Next, define the objective function.

Tip

The objective function needs to sum costs across all beer types and all periods. With dictionaries, you reference the data like this:

```
@objective(model, Min,
    sum(... for i in keys(dictBottlingTime), t in
keys(dictAvailableTime))
)
```

Inside the sum, you'll need:

1. Setup costs: `setupHourCosts * dictSetupTime[i] * productBottled[i,t]`
   - This uses `dictSetupTime[i]` to get the setup time for beer type `i`
2. Inventory holding costs: `warehouseCosts * WarehouseStockPeriodEnd[i,t]`

Notice how we access dictionary values using `dict[key]` notation!

```
# YOUR CODE BELOW
```

```
# Validate your solution
# Check if the model has an objective
@assert objective_function(lotsizeModel) !== nothing "Model must have an
objective function"

# Check if it's a minimization problem
@assert objective_sense(lotsizeModel) == MOI.MIN_SENSE "Objective should be
minimization"

# Check if the objective function contains both cost components
obj_expr = objective_function(lotsizeModel)
@assert contains(string(obj_expr), "productBottled") "Objective must
include setup costs (productBottled)"
@assert contains(string(obj_expr), "WarehouseStockPeriodEnd") "Objective
must include warehouse costs (WarehouseStockPeriodEnd)"
```

Define the constraints

Now, we need to define all necessary constraints for the model. Start with the demand/inventory balance constraint.

> Tip
>
> The first period is special, as it does not have a previous period. Furthermore, we are working with strings as variable references, thus we cannot use `t-1` directly as in the lecture. To address this, we could collect and sort all keys and then use their indices to address the previous period. For example, `all_periods[t-1]` would then be the previous period, if we index t just as a range from `2:length(all_periods)`.

```
# Get the first period and all periods
first_period = first(sort(collect(keys(dictAvailableTime))))
all_periods = sort(collect(keys(dictAvailableTime)))
```

With these, we can now define the demand/inventory balance constraint. As this is the first constraint and might be a bit tricky, the solution is already given below.

```
# Inventory balance constraints for periods after first period
@constraint(lotsizeModel,
    demandBalance[i=keys(dictBottlingTime), t=2:length(all_periods)],
    WarehouseStockPeriodEnd[i,all_periods[t-1]] +
productQuantity[i,all_periods[t]] -
WarehouseStockPeriodEnd[i,all_periods[t]] == dictDemand[i,all_periods[t]]
    )
```

Next, we need to ensure that we setup the production for a beer type only if we bottle the type at least once.

```
# YOUR CODE BELOW
```

Last, we need to define the constraint that limits the production quantity to the number of bottles that can be bottled within the available time.

```
# YOUR CODE BELOW
```

Solve the model

Finally, implement the solve statement for your model instance.

```
# YOUR CODE BELOW
```

```
# Validate your solution
@assert 539900 <= objective_value(lotsizeModel) <= 700000 "Objective value
should be between 539,000 and 700,000"
```

Now, unfortunately we cannot assert the value of the objective function perfectly here as we have to abort the computation due to the time limit and everybody is likely getting different results. The solution for the first task will likely be in the range of

600,000 to 700,000. If your model is solved within seconds, your formulation is not correct.

Create the plots

The following code creates production and warehouse plots for you. Use it to verify and visualize your solution in the following tasks.

> **i** Note
>
> The creation of the dataframes and the plots is implemented inside of a function, as we will need to use it multiple times in the following tasks.

```
# Create the production results
function create_production_results()
    # Create a DataFrame to store the results
    productionResults = DataFrame(
        period = String[],
        product = String[],
        productBottled = Bool[],
        productQuantity=Int[],
        WarehouseStockPeriodEnd=Int[]
    )

    # Populate the DataFrame with the results
    for i in keys(dictSetupTime)
        for t in keys(dictAvailableTime)
            push!(
                productionResults,(
                period = t,
                product = i,
                productBottled = value(productBottled[i,t])>0.5 ? true :
false,
                productQuantity = ceil(Int,value(productQuantity[i,t])),
                WarehouseStockPeriodEnd =
ceil(Int,value(WarehouseStockPeriodEnd[i,t])),
                )
            )
        end
    end

    sort!(productionResults,[:period, :product])
    return productionResults
end

# Create the production plot
function create_production_plot(productionResults)
    p = groupedbar(
        productionResults.period,
        productionResults.productQuantity,
        group=productionResults.product,
```

```
        ylabel="Production Quantity (Bottles)",
        xlabel="Period",
        title="Production Schedule by Beer Type",
        size=(1200,600),
        palette = :Set3,
        legend=:outertopright,
        xrotation = 45,
        legendtitle="Beer Type",
        bar_width=0.7,
        grid=false,
        dpi=300
    )
    return p
end

# Create the warehouse stock plot
function create_warehouse_plot(productionResults)
    p = groupedbar(
        productionResults.period,
        productionResults.WarehouseStockPeriodEnd,
        group=productionResults.product,
        ylabel="Warehouse Stock",
        xlabel="Period",
        title="Warehouse Stock",
        size=(1200,600),
        palette = :Set3,
        legend=:outertopright,
        xrotation = 45,
        legendtitle="Beer Type",
        bar_width=0.7,
        grid=false,
        dpi=300
    )
    return p
end
```

The following code creates the production plot.

```
productionResults = create_production_results()
p = create_production_plot(productionResults)
```

The following code creates the warehouse stock plot.

```
productionResults = create_production_results()
p = create_warehouse_plot(productionResults)
```

Calculate the setup and inventory costs

Next, we calculate the setup and inventory costs for each period and store them in a DataFrame. This should also work for you, if you followed the previous name instructions.

```
# Calculate costs per period
function create_cost_results()
    costResults = DataFrame(
        period = String[],
        setup_costs = Float64[],
        inventory_costs = Float64[]
    )

    for t in sort(collect(keys(dictAvailableTime)))
        # Calculate setup costs for this period
        period_setup_costs = sum(
            setupHourCosts * dictSetupTime[i] * value(productBottled[i,t])
            for i in keys(dictBottlingTime)
        )

        # Calculate inventory costs for this period
        period_inventory_costs = sum(
            warehouseCosts * value(WarehouseStockPeriodEnd[i,t])
            for i in keys(dictBottlingTime)
        )

        push!(costResults, (
            period = t,
            setup_costs = period_setup_costs,
            inventory_costs = period_inventory_costs
        ))
    end

    # Stack the cost columns
    stacked_costs = stack(costResults, [:setup_costs, :inventory_costs],
                          variable_name="Cost_Type", value_name="Cost")
    return stacked_costs
end

# Create the cost plot
function create_cost_plot(stacked_costs)
    p = groupedbar(
        stacked_costs.period,
        stacked_costs.Cost,
        group=stacked_costs.Cost_Type,
        ylabel="Costs (€)",
        xlabel="Period",
        title="Setup and Inventory Costs per Period",
        size=(1200,600),
        palette=:Set2,
        legend=:outertopright,
        xrotation=45,
        legendtitle="Cost Type",
        bar_width=0.7,
        grid=false,
        dpi=300
    )
    return p
end
```

The following code calls the setup and inventory costs plot.

```
stacked_costs = create_cost_results()
p = create_cost_plot(stacked_costs)
```

---

2. Initial Warehouse Stock

The model currently sets the initial warehouse stock levels without any restrictions. Modify your model to incorporate an initial stock for all types of beer of zero at the beginning of the initial planning period.

To solve this task, you can simply extend the previous model by these additional constraints in the cell below. Afterwards, you can re-run the optimization.

```
# YOUR CODE BELOW
```

```
# Validate your solution
@assert 659900 <= objective_value(lotsizeModel) <= 760000 "Objective value
should be between 659,9000 and 760,000"
```

The objective value should now be higher, as the solution space is smaller than before and the initial stock is zero for all beer types. You can check the plots for the production and warehouse stock to verify this.

```
productionResults = create_production_results()
p = create_production_plot(productionResults)
```

```
productionResults = create_production_results()
p = create_warehouse_plot(productionResults)
```

```
stacked_costs = create_cost_results()
p = create_cost_plot(stacked_costs)
```

---

3. Scheduled Repair

Unfortunately, the bottling plant has to undergo maintenance in periods `"week_10"` and `"week_11"`. Extend your model to prevent any production in those two periods.

Again, to solve this task, you can simply extend the previous model by these additional constraints in the cell below. Afterwards, you can re-run the optimization.

```
# YOUR CODE BELOW
```

```
# Validate your solution
@assert 661400 <= objective_value(lotsizeModel) <= 800000 "Objective value
should be between 661,400 and 800,000"
```

Again, the objective value should be higher, because the solution space is smaller. You can check the plots for the production and warehouse stock to verify whether the production is zero in the maintenance periods.

```
productionResults = create_production_results()
p = create_production_plot(productionResults)
```

```
productionResults = create_production_results()
p = create_warehouse_plot(productionResults)
```

```
stacked_costs = create_cost_results()
p = create_cost_plot(stacked_costs)
```

---

4. Production Schedule Analysis

Analyze the production schedule outlined in section 2 of this tutorial. Is the workload distributed evenly across all time periods? Provide a rationale for your assessment.

Please answer in the following cell. Note, that `#=` and `=#` are a comment delimiter for multiline comments. You can write whatever you want between them and the code will not be executed.

```
# YOUR REASONING BELOW
#=



=#
```

Based on the production data from the final period, calculate the ending inventory levels for each type of beer. Discuss any significant findings. Compute the ending inventory levels for each type of beer in the following cell. You can name the DataFrame however you want.

```
# YOUR CODE BELOW
```

---

5. Biannual Bottling Strategy

Reflecting on a scenario where the company schedules its bottling operations biannually using the current method: identify and discuss potential pitfalls of this strategy.

Offer at least one actionable suggestion for enhancing the efficiency or effectiveness of the production planning process.

Your answer goes here.

```
# YOUR ANSWER BELOW
#=



=#
```

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.

Bibliography