Tutorial IV.III - Constraints in JuMP

Applied Optimization with Julia

Introduction

This Julia script is an interactive tutorial to introduce you to advanced handling of objective functions and constraints in JuMP. You'll learn about defining and managing objective functions and constraints within containers (arrays, dictionaries, matrices), and how to implement conditional constraints effectively.

Follow the instructions, write your code in the designated code blocks, and confirm your understanding with `@assert` statements. Make sure to have the JuMP package installed to follow this tutorial.

Section 1 - Objective Functions with Container Variables

First, we need to load the JuMP and HiGHS packages.

```
using JuMP, HiGHS
```

Then, we need a model first to assign something to it.

```
another_model = Model(HiGHS.Optimizer)
```

```
A JuMP Model
  solver: HiGHS
  objective_sense: FEASIBILITY_SENSE
  num_variables: 0
  num_constraints: 0
  Names registered in the model: none
```

Defining objective functions with variables in containers allows for scalable and dynamic model formulations. First, we need a container with variables for the objective function. For example:

```
@variable(modelName, variableName[1:3] >= 0)
```

Now, we can define an objective function with the container. For example:

```
@objective(modelName, Max, sum(variableName[i] for i in 1:3))
```

Exercise 1.1 - Define an array of variables and an objective function

Scenario: Imagine you're optimizing the production of 8 different products in a factory. Each product has a different profit margin, and you want to maximize total profit.

Define an array of variables and an objective function for `another_model`. The variables should be called `profits` and have a range from `1:8`. It has a lower bound of `0`. The objective should be a Maximization of the sum of all `profits`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert length(profits) == 8 && all(lower_bound(profits[i]) == 0 for i in 1:8)
@assert typeof(objective_function(another_model)) == AffExpr
println("Objective function with container variables defined successfully!")
```

Section 2 - Constraints within Containers

Defining constraints within containers allows for structured and easily manageable models. This is especially important when models become larger! To define a constraint within a container, we can do, for example, the following:

```
@constraint(modelName,
    constraintName[i in 1:3],
    variableName[i] <= 100
)
```

This would create a constraint called `constraintName` for each `i` - thus `1`,`2`, and `3` - where `variableName[1]`, `variableName[2]`, and `variableName[3]` are restricted to be maximally `100`.

Exercise 2.1 - Define constraints called ‘maxProfit’ using an array of variables

Continuing our factory scenario: Each product has a maximum daily production capacity due to machine limitations.

Define constraints called `maxProfit` using an array of variables. The logic: Each profit defined in the previous task should be less than or equal to `12`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert all(is_valid(another_model, maxProfit[i]) for i in 1:8)
println("Constraints within containers defined successfully!")
```

Section 3 - Implementing Conditional Constraints

Conditional constraints are added to the model based on certain conditions, allowing for dynamic and flexible model formulations. To define a constraint within a container under conditions, we can do the following:

```
@constraint(modelName,
    constraintName[i in 1:3; i <= 2],
    variableName[i] <= 50
)
```

This would create a constraint called `constraintName` for each `i` - thus 1,2, and 3 - where `variableName[1]`, `variableName[2]` are restricted to be maximally `50` and `variableName[3]` was not restricted.

Exercise 3.1 - Add a conditional constraint 'smallProfit' to the previous model

Scenario extension: The first 4 products are new and have limited market demand.

Add a conditional constraint `smallProfit` to the previous model. Condition: Only the first 4 variables profit have to be lower than 5.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert all(is_valid(another_model, smallProfit[i]) for i in 1:4)
println("Conditional constraint implemented successfully!")
println("Checking successful implementation.")
optimize!(another_model)
status = termination_status(another_model)
@assert status == MOI.OPTIMAL "Sorry, something didn't work out as the model status is
 ↪  $status"
@assert objective_value(another_model)   68 atol=1e-4 "Although you have an optimal
 ↪  solution,
    the should be 68 not $(objective_value(another_model)). Is the model correct?"
println("Model components validated successfully!")
```

Visualization of Results

Let's visualize our optimal solution:

```
using Plots
# Assuming the model has been solved!!!
optimal_profits = value.(profits)

bar(1:8, optimal_profits,
    title="Optimal Production Levels",
    xlabel="Product",
    ylabel="Profit",
    legend=false)
```

Conclusion

Congratulations! You've completed the tutorial on advanced handling of objective functions and constraints in JuMP. You've learned how to define objective functions and constraints using container variables. Continue to the next file to learn more.