Tutorial VIII - Police Districting

Applied Optimization with Julia

Introduction

Imagine you're the lead consultant for Hamburg's police force modernization project. The city is facing increasing response times in certain districts, and the Chief of Police has hired you to optimize their district boundaries. Your mission: redesign Hamburg's police districts to ensure rapid response times while meeting several critical operational requirements.

Key Objectives:

- Minimize average response time to incidents
- Ensure every area has adequate police coverage
- Enable efficient backup support between districts
- Account for real-world constraints like traffic patterns

Throughout this tutorial, you'll build this solution step-by-step, starting with a basic model and gradually adding real-world constraints that police departments face daily.

1. Modelling the P-Median Problem

Your first task is to fix a prototype model. It contains 6 critical errors that are causing the model to fail. As the lead consultant, you need to:

1. Identify and fix these errors
2. Document why each fix was necessary in a comment
3. Validate that your solution makes sense

In this task it is not yet necessary to include contiguity and compactness constraints! If the model takes too long on your machine, you can set the relative gap to 10% or decrease the time limit.

To execute the code, several files are provided, which contain the Euclidean distance between the BAs, the driving time between BAs, the number of incidents per BA, and the hexagonal shapes (a `.shp` file and a `.dbf` file - you need both!).

Loading Data and Packages

We start by loading the new `Shapefile` package. No mistakes here!

```
import Pkg
Pkg.add("Shapefile")
```

We then load the required packages. If you have not installed them yet, you can do so by modifying the code chunk above to include the missing packages. Still no mistakes!

```
using JuMP, HiGHS
using DelimitedFiles
using Shapefile
using DataFrames
using Plots
```

We then define the number of departments, load the data into scope and define the number of departments and the weighted driving times. Make sure to use the correct path to the data files - it should be relative to the location notebook file in a folder called `data`. Again, no mistakes!

```
# Define a seed for reproducibility
Random.seed!(42)

# Load the data into scope
file_directory = "$(@__DIR__)/data"
euclidianDistances = readdlm("$file_directory/
euclidianDistances0510.csv",'\t')
drivingTimes = readdlm("$file_directory/drivingTimes0510.csv",'\t')
incidentWeights = vec(readdlm("$file_directory/
incidentWeights0510.csv",'\t'))

# Load the Shapefile for plotting
hexshape = DataFrame(Shapefile.Table("$file_directory/grid0510.shp"))
sort!(hexshape, :id)

# Define the number of departments
p = 10

# Define the weighted driving times
weightedDriving = drivingTimes .* transpose(incidentWeights)
```

Fixing the Mistakes

From here on, the code contains 6 mistakes you need to correct in order to solve the model. Mistakes can be anything so be careful!

```
# MISTAKES BELOW

# Prepare the model instance
pMedianModel = Model(HiGHS.Optimizer)
set_attribute(pMedianModel, "presolve", "on")
set_attribute(model, "time_limit", 120.0)
set_attribute(pMedianModel, "mip_rel_gap", 0.0)

# Define the range of the problem instance
rangeBAs = 1:2
```

```
rangeDepartments = unique(rand(1:size(incidentWeights,1), 100))  # Ensure
unique departments (This line is correct!)

# Define variable
@variable(pMedianModel, X[i = rangeDepartments,j = rangeDepartments], Bin)

# Define objective function
@objective(pMedianModel, Max,
    sum(weightedDriving[i,j]* X[i,j] for i in rangeDepartments, j in
rangeBAs)
    )

# Define the constraints
@constraint(pMedianModel,
    eachAllocated[j=rangeBAs],
    sum(X[i,j] for i in rangeDepartments) == 0
    )

@constraint(pMedianModel,
    pLocations,
    sum(X[i,i] for i in rangeDepartments) == p
    )

@constraint(pMedianModel,
    departmentNecessary[i=rangeDepartments,j=rangeBAs],
    2 * X[i,j] <= X[i,i]
    )

# Start optimization
optimize!(pMedianModel)

# MISTAKES ABOVE
```

We then check the solution. No mistakes here, as all mistakes are in the code in the cell above!

```
# Function to print the model status
function print_model_status(model)
    begin
        println()
    if termination_status(pMedianModel) == OPTIMAL
        println("Great, the solution is optimal.")
        println("The relative gap is $(relative_gap(pMedianModel))")
        println("The solve time (in seconds) is
$(solve_time(pMedianModel))")
    elseif termination_status(pMedianModel) == TIME_LIMIT &&
has_values(pMedianModel)
        println("Solution is suboptimal due to a time limit, but a primal
solution is available")
    else
        error("The model was not solved correctly.")
    end
        println("The objective value is ", objective_value(pMedianModel))
```

```
    end
end
```

```
print_model_status (generic function with 1 method)
```

```
print_model_status(pMedianModel)
@assert termination_status(pMedianModel) == OPTIMAL ||
(termination_status(pMedianModel) == TIME_LIMIT &&
has_values(pMedianModel)) "Unfortunate, the model was not solved correctly.
Have you corrected all mistakes?"
println("Great, the model was solved correctly.")
```

Visualizing the Results

The following code then builds and uses a function to plot the results.

```
function visualize_departments(hexshape, X, p)
    # Convert solution matrix to regular Matrix
    allAssignments = Matrix(value.(X))

    # Find assignments where value > 0.5 (accounting for potential
floating-point imprecision)
    assignments = findall(allAssignments .> 0.5)

    # Create copy of hexshape to avoid modifying original
    plot_data = copy(hexshape)

    # Assign departments and initialize colors
    plot_data.department = rangeDepartments[map(x->x[1], assignments)]
    plot_data.color = fill(RGB(0/255, 0/255, 0/255), nrow(plot_data))

    # Get unique department locations and create color mapping
    department_locations = unique(rangeDepartments[map(x->x[1],
assignments)])
    color_palette = cgrad(:Pastel1_9, p, categorical=true)
    color_dict = Dict(department_locations[i] => color_palette[i] for i in
1:p)

    # Color non-department locations
    for hex in eachrow(plot_data)
        if hex.id in department_locations
            hex.color = RGB(0/255, 0/255, 0/255)
        else
            hex.color = color_dict[hex.department]
        end
    end

    # Create and return plot
    return plot(
        plot_data.geometry,
        color=plot_data.color',
```

```
        legend=false,
        axis=false,
        ticks=false,
        size=(800,450)
    )
end

# Plot the results
plot_area = visualize_departments(hexshape, X, p)
```

---

2. Ensuring District Connectivity

The Police Chief has identified a critical flaw in the initial model: some police units would need to drive through other districts to reach parts of their own district! This creates jurisdictional issues and slower response times.

Your challenge:

- Implement contiguity constraints to ensure each district is fully connected
- Compare response times before and after adding these constraints
- Visualize the impact of your changes on the district map

Conditional Constraints

Before we start, we will quickly repeat some basic concepts on constraints and conditions in JuMP. You can add conditions to constraints by using the `;` operator. This is useful if you want to add a constraint only under certain conditions. In the example below, the constraint is only active if the Euclidean distance between two BAs is less than 1.5.

```
@constraint(model,
    conditionalConstraint[
        i=rangeDepartments,
        j=rangeBAs;
        euclidianDistances[i,j] < 1.5
        ],
    X[i,j] == 1
    )
```

Furthermore,we can use conditions within constraints by using the `for` keyword. For example, in the constraint below, the sum is only taken over the BAs that are within 1.5 units of BA i.

```
@constraint(model,
    conditionalConstraint2[i=rangeDepartments],
    sum(X[i,j] for j in rangeBAs if euclidianDistances[i,j] < 1.5) >= 1
    )
```

Extending the Model

Now, we can start to extend the model. Add the contiguity constraint from the lecture to the model.

> ! Important
>
> Take a careful look at the Euclidean distances, as you can use them to determine if two BAs are adjacent to each other. If the distance between two BAs is less than 1.5, then the BAs are adjacent to each other. You can use this information to define the new constraint.

```
# YOUR CODE BELOW
```

Solve the model again, this time with the contiguity constraint.

```
# YOUR CODE BELOW
```

The following code prints the model status and visualizes the districts. If your implementation is correct, the districts should be contiguous and the model should have reached optimality or found a feasible solution before hitting the time limit.

```
print_model_status(pMedianModel)
display(visualize_departments(hexshape, X, p))
```

Compute the Gap

Based on your results, what is the gap between the solution in the previous task and this task? Write a comment answering the question in cell below. You can also use the cell, to compute the gap based on the objective values.

```
#=



=#
```

> Tip
>
> If you computer cannot determine the optimal solution, you can just use the best solutions you found after both runs to compute the gap.

---

3. Emergency Response Time Guarantees

Hamburg's City Council has mandated that high-priority emergencies must receive a response within 20 minutes. Your previous model doesn't guarantee this! Your task: Implement maximum response time constraints.

Extend your model by an additional parameter `max_driving_time = 20` to ensure that no allocation with a driving time $d_{i,j}$ higher than `max_driving_time` minutes from i to j is possible.

```
# YOUR CODE BELOW
```

> Tip
>
> Don't confuse the `weightedDriving` matrix with the `drivingTimes` matrix! The `weightedDriving` matrix contains the weighted driving times, while the `drivingTimes` matrix contains the driving times.

Again, solve the model.

```
# YOUR CODE BELOW
```

The following code prints the model status and visualizes the districts. If your implementation is correct, the districts with previously longer driving times should have shrinked. Furthermore, the model should have reached optimality or found a feasible solution before hitting the time limit.

```
print_model_status(pMedianModel)
display(visualize_departments(hexshape, X, p))
```

---

4. Planning for Peak Demand

During major events or crime waves, districts need backup support from neighboring stations. The Police Union has emphasized this as a critical safety requirement for their officers. Your task: Design constraints ensuring each district has backup support within 20 minutes of the driving time.

```
# YOUR CODE BELOW
```

> Tip
>
> This is a rather complicated task, so don't worry if you cannot find the correct restriction. Try to come up with something or describe your thoughts in comments if you do not find the correct restriction or a way to implement it. If you are not sure, it often helps to start on paper and to draw the districts and the BAs to find a way to implement it.

In case you came up with a solution, solve the model to optimize the layout of the districts.

```
# YOUR CODE BELOW
```

If your implementation is correct, the districts are contiguous and distributed across the city. In addition, the model should have reached optimality or found a feasible solution before hitting the time limit.

```
print_model_status(pMedianModel)
display(visualize_departments(hexshape, X, p))
```

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