Contents

Formulate FCTP

The mathematical model for this Fixed Charge Transportation Problem (FCTP) we study has the following form:

$$\min \sum_{i \in S} \sum_{j \in T} (c_{ij} x_{ij} + f_{ij} y_{ij})$$

subject to

$$\sum_{j \in T} x_{ij} = a_i \qquad \forall i \in S \quad (1)$$

$$\sum_{i \in S} x_{ij} = b_j \qquad \forall j \in T \quad (2)$$

$$x_{ij} \leq m_{ij} y_{ij} \qquad \forall i \in S, j \in T \quad (3)$$

$$x_{ij} \in \mathbb{Z}_+ \qquad \forall i \in S, j \in T \quad (4)$$

$$y_{ij} \in \{0, 1\} \qquad \forall i \in S, j \in T \quad (5)$$

where:

- S is the set of source nodes that supply goods.
- T is the set of destination nodes that receive goods.
- c_{ij} is the variable cost for transporting a unit of good from source node i to destination node j.
- f_{ij} is the fixed cost incurred for using the transportation link from source node i to destination node j.
- x_{ij} is the number of goods transported from source node i to destination node j.
- y_{ij} is a binary variable that is equal to 1 if the transportation link from source node i to destination node j is used, and 0 otherwise.
- a_i is the supply of goods at source node i.
- b_j is the demand of goods at destination node j.
- m_{ij} is the maximum amount of goods that can be transported from source node i to destination node j if the transportation link is used.

At the same time this FCTP problem can also be written as:

$$\min \hat{\mathbf{c}}^\top \mathbf{x}$$

subject to

$$\mathbf{A}\hat{\mathbf{x}} \leq \mathbf{b}, \quad \hat{\mathbf{x}} \in \mathbb{Z}^n$$

The constraint matrix A and the right hand side vector b are presented below:

x11	x12	x13	x21	x22	x23	x31	x32	x33	y11	y12	y13	y21	y22	y23	y31	y32	y33	≤ =	RHS	Constraint #
1	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	=	1	(1)
0	0	0	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	=	2	(1)
0	0	0	0	0	0	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	=	1	(1)
1	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	=	1	(2)
0	**1**	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	=	2	(2)
0	0	**1**	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	=	1	(2)
1	0	0	0	0	0	0	0	0	**-1**	0	0	0	0	0	0	0	0	<=	0	(3)
0	**1**	0	0	0	0	0	0	0	0	**-1**	0	0	0	0	0	0	0	<=	0	(3)
0	0	**1**	0	0	0	0	0	0	0	0	**-1**	0	0	0	0	0	0	<=	0	(3)
0	0	0	**1**	0	0	0	0	0	0	0	0	**-1**	0	0	0	0	0	<=	0	(3)
0	0	0	0	**1**	0	0	0	0	0	0	0	0	**-2**	0	0	0	0	<=	0	(3)
0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	**-1**	0	0	0	<=	0	(3)
0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	**-1**	0	0	<=	0	(3)
0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	**-1**	0	<=	0	(3)
0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	**-1**	<=	0	(3)

Figure 1: Constraint matrix A and vector b (RHS) for the FCTP.

In this task, we assume that all fixed costs are 10 ($f_{ij} = 10 \, \forall i \in S, j \in T$). Thus the cost vector $\hat{\mathbf{c}}$, following the structure given in constraint matrix A, is given by :

$$\hat{\mathbf{c}} = [2, 1, 2, 1, 2, 1, 2, 1, 2, 10, 10, 10, 10, 10, 10, 10, 10, 10]$$

The code to construct and populate the constraint matrix A and vectors b and $\hat{\mathbf{c}}$ is given below:

```julia
using LinearAlgebra

# Given supply and demand values for sources (a, b, c) and destinations (d, e, f)
a = [1, 2, 1]  # Supply values for sources a, b, c
b = [1, 2, 1]  # Demand values for destinations d, e, f

# Calculate the minimum of supply and demand for each route (m_ij values)
mij = [min(ai, bj) for ai in a for bj in b]
mij_matrix = reshape(mij, (length(a), length(b)))

# Number of sources and destinations
num_sources = length(a)
num_destinations = length(b)

# Initialize the matrix A with zeros
# It will have a row for each supply constraint, each demand constraint,
# and each route usage constraint, which is num_sources * num_destinations
num_constraints = num_sources + num_destinations + num_sources * num_destinations
num_variables = 2 * num_sources * num_destinations  # For x_ij and y_ij variables
A = zeros(num_constraints, num_variables)

# Populate the supply constraints
for i in 1:num_sources
    A[i, (i - 1) * num_destinations + 1 : i * num_destinations] .= 1
end

# Populate the demand constraintsB
for j in 1:num_destinations
    A[num_sources + j, j:num_destinations:num_sources*num_destinations] .= 1
end

# Populate the route usage constraints x_ij - mij*y_ij <= 0
for i in 1:num_sources
    for j in 1:num_destinations
        constraint_index = num_sources + num_destinations + (i - 1) * num_destinations + j
        x_index = (i - 1) * num_destinations + j
        y_index = num_sources * num_destinations + (i - 1) * num_destinations + j
        A[constraint_index, x_index] = 1  # Coefficient for x_ij
        A[constraint_index, y_index] = -mij_matrix[i, j]  # Coefficient for -m_ij*y_ij
    end
end

# Define the right-hand side (RHS) vector
RHS = [a; b; zeros(num_sources * num_destinations)]

# Display the matrix A and RHS
(A, RHS)
```

Figure 2: Matrix Creation Code

Define Master-Problem and Sub-Problem

Dantzig-Wolfe decomposition is a specific application of column generation applied to certain types of LP problems that can be decomposed into smaller subproblems. The technique involves dividing the original problem into a master problem and one or more subproblems. The master problem finds the optimal combination of solutions from the subproblems, while the subproblems generate potential solutions to be considered by the master problem.

To apply the Dantzig-Wolfe decomposition effectively, it is advantageous to group the columns of the constraint matrix **A** per source. This implies organizing the variables such that $x_{11}, x_{12}, x_{13}, y_{11}, y_{12}, y_{13}$, and so forth, are grouped together. By doing so, we can clearly delineate the sub-problems associated with each source, which are then solved independently within the decomposition framework.

The reordered constraint matrix **A**, illustrating this column grouping, is presented in Figure 3. The coloring in the figure highlights the block angular form of the matrix, with each sub-problem (per source) marked by a distinct color. This block structure is ideal for the Dantzig-Wolfe decomposition as it reveals the presence of |S| independent sub-problems, one for each source, in the master problem.

x11	x12	x13	y11	y12	y13	x21	x22	x23	y21	y22	y23	x31	x32	x33	y31	y32	y33	≤ =	RHS
1	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0	=	1
0	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	=	2
0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	=	1
1	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	=	1
1	0	0	**-1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	<=	0
0	**1**	0	0	**-1**	0	0	0	0	0	0	0	0	0	0	0	0	0	<=	0
0	0	**1**	0	0	**-1**	0	0	0	0	0	0	0	0	0	0	0	0	<=	0
0	0	0	0	0	0	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	=	2
0	0	0	0	0	0	**1**	0	0	**-1**	0	0	0	0	0	0	0	0	<=	0
0	0	0	0	0	0	0	**1**	0	0	**-2**	0	0	0	0	0	0	0	<=	0
0	0	0	0	0	0	0	0	**1**	0	0	**-1**	0	0	0	0	0	0	<=	0
0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	0	0	0	=	1
0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**-1**	0	0	<=	0
0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**-1**	0	<=	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**-1**	<=	0

Figure 3: Reordered constraint matrix **A** showing the block angular form suitable for Dantzig-Wolfe decomposition. Each sub-problem (per source) is indicated by a different color.

The code to reformulate the contraint matrix A and vector b for Dantzig-Wolfe decomposition for each sub-problem is provided below:

```
cols_order = [1, 2, 3, 10, 11, 12, 4, 5, 6, 13, 14, 15, 7, 8, 9, 16, 17, 18]
rows_order = [4, 5, 6, 1, 7, 8, 9, 2, 10, 11, 12, 3, 13, 14 , 15]
# # Create a copy of A to reorder
A_reordered = copy(A)
b_reordered = copy(RHS)

# # Reorder columns
# where each block corresponds to the constraints for one source
A = A_reordered[:, cols_order]
A_new = copy(A)

A = A_new[rows_order,:]
b = b_reordered[rows_order]
```

Figure 4: Necessary Matrix Reordering

Solve Model

Our implementation uses the JuMP package in Julia to model the master and subproblems, and the GLPK solver to solve these problems.

Data Structures

To effectively implement the Dantzig-Wolfe (DW) reformulation for handling multiple subproblems, it's essential to carefully organize the problem's setup, including the allocation of variables, constraints, and subproblems. Figure 5 illustrates the initialization of these critical data structures.

```
C = [2 1 2 10 10 10 1 2 1 10 10 10 2 1 2 10 10 10]

# Rows of Constraint Matrix A belonging in the Master Problem
masterRows = [1, 2, 3]

# rows of Constraint Matrix A belonging in the sub-problems
subBlocks=[[4],[8],[12]]

# #number of sub-problems
K=length(subBlocks)

# # v[k] is a vector of the variables in subproblem k
V = Vector{Vector{Int64}}(undef,K)

# 3 x-variables and 3 y_variables per problem
V[1] = [1, 2, 3, 4, 5, 6];
V[2] = [7, 8, 9, 10, 11, 12];
V[3] = [13, 14, 15, 16, 17, 18];

A0 = A[masterRows,:]
b0 = b[masterRows,:]

CV = Vector{Array{Float64,2}}(undef,K)
A_V = Vector{Array{Float64,2}}(undef,K)
A0_V = Vector{Array{Float64,2}}(undef,K)
b_sub = Vector{Array{Float64,2}}(undef,K)

for k=1:K
    # submatrix of cost with the columns given by indices V_k
    CV[k] = C[:,V[k]]
    A0_V[k] = A0[:,V[k]]
    A_V[k] = A[[subBlocks[k][1], subBlocks[k][1] + 1,subBlocks[k][1] + 2, subBlocks[k][1] + 3] ,V[k]]
    b_sub[k] = b[subBlocks[k],:]
end
```

Figure 5: Creating the data structures

The code depicted in Figure 5 plays a pivotal role in separating the master problem components, represented by A^0 and b^0, from those of the subproblems. The matrix A^0 and vector b^0 encapsulate the constraints and their right-hand side values that are part of the master problem. The cost vector $\mathbf{C}$ extracts the coefficients related to each subproblem, aligned with the variables identified in $\mathbf{V}$. The matrix A_V isolates the constraints for each subproblem by selecting the appropriate rows from $\mathbf{A}$ based on the subBlocks indices, and filtering the columns according to the variables in $\mathbf{V}[k]$. Similarly, A_V^0 is analogous to A_V, but specifically configured for the master problem's constraints, ensuring that the solutions of subproblems are evaluated within the overall problem context. Lastly, b_{sub} denotes the right-hand side values for each subproblem's constraints, guaranteeing that the solutions satisfy the localized constraints.

Master Problem Definition

In column generation applied to Dantzig-Wolfe decomposition, the master problem often deals with a set of decision variables that represent weights (or multipliers) for columns in the solution space of the subproblems. The objective function and constraints in the master problem are constructed to determine the best combination of these weights to minimize the overall cost while satisfying the original constraints. In our case, the objective is to formulate a Column Generation model utilizing the Dantzig-Wolfe decomposition while considering a relaxed constraint set, where $\hat{X}_*^k$ does not represent the complete set of all extreme points.

$$
\begin{aligned}
\min \quad & \sum_{k \in K} c_{V^k} \hat{X}_*^{\,k} \lambda^k & \\
\text{subject to} \quad & \sum_{k \in K} A^0_{V^k} \hat{X}_*^{\,k} \lambda^k \leq b_0 & \quad (\pi) \\
& \lambda^k = 1 & \quad \forall k \in K(\kappa_k) \\
& \lambda^k \geq 0 & \quad \forall k \in K
\end{aligned}
$$

As depicted in Figure 6, the initial feasible solution is constructed by allocating one unit from Supply A to Demand D (hence, $x_{1,1} = y_{1,1} = 1$) and subsequently fulfilling the remaining demand by sending two units from Supply B to Demand E, and one unit from Supply C to Demand F. This choice respects the maximum permissible transfer capacities, denoted by mij.

In the current framework, each demand is paired with one extreme point (solution). The model preserves the demand satisfaction constraint from the original master problem, as well as the convexity constraint for the λ variables. It is crucial to observe that for a minimization context, the dual variables are directly adopted from the JuMP model output, as opposed to inverting their signs, which would be the case in a maximization scenario.

Solving this model for the first time provides us with the necessary initial dual variables needed in to generate the column in the subproblem.

```julia
using JuMP, GLPK, LinearAlgebra

include("Task1_Remake.jl")

# X[k] Extreme points for polyhedron k
X = Vector{Array{Int64,2}}(undef,K)

# Initial extreme point
X[1] = [1 0 0 1 0 0]' # send from A to D
X[2] = [0 2 0 0 1 0]' # send from B to E
X[3] = [0 0 1 0 0 1]' # send from C to F

# P[k] number of extreme points for polyhedron k
P = [1,1,1]

master = Model(GLPK.Optimizer)

# Create a vector that contain vectors of decision variables
@variable(master, lambda1[1:P[1]] >= 0 )
@variable(master, lambda2[1:P[2]] >= 0 )
@variable(master, lambda3[1:P[3]] >= 0 )
lambda = [lambda1, lambda2, lambda3]

@objective(master, Min, sum(dot(CV[k]* X[k], lambda[k]) for k=1:K))

@constraint(master, cons, sum(A0_V[k]*X[k]*lambda[k] for k=1:K ) .== b0 )
@constraint(master, convexityCons[k=1:K], sum(lambda[k][j] for j=1:P[k]) == 1)

optimize!(master)

if termination_status(master) == MOI.OPTIMAL
    println("Objective value: ", JuMP.objective_value(master))
    lambdaVal = [JuMP.value.(lambda[k]) for k=1:K]
    println("lambda: ", lambdaVal)
    # REMOVE - from constraint, its a minimization
    println("piVal = ", dual.(cons))
    println("kappa = ", dual.(convexityCons))
else
    println("Optimize was not succesful. Return code: ", termination_status(master))
end
```

Figure 6: Master Problem code

Subproblem Definition

The mathematical model for the subproblem of the master problem is given below:

$$\begin{aligned} \min \quad & c_k^* = c_{V_k} x - \pi A_{V_k} x - \kappa_k \\ \text{subject to} \quad & A_{V_k}^k x \leq b^k \\ & x \in \mathbb{Z}^{n_k} \end{aligned}$$

After solving the master problem (MP), we found the objective function value to be 38, with the dual prices vector π being $[12.0, 7.0, 12.0]$ and the κ_k vector being $[0.0, 0.0, 0.0]$. These values will be utilized in the first iteration of our column generation process.

```
using JuMP, GLPK, LinearAlgebra

include("Task3_MP_remake.jl")

# dual variables from master problem:
# it 1:
piVal = [12.0 7.0 12.0]
kappa = [0.0 0.0 0.0]

for k = 1:K
    sub = Model(GLPK.Optimizer)

    @variable(sub, x[1:length(V[k])]>=0 )
    set_binary(x[4])
    set_binary(x[5])
    set_binary(x[6])
    @objective(sub, Min, dot(CV[k],x) -dot(piVal * A0_V[k], x)- kappa[k])
    @constraint(sub, dot(A_V[k][1,1:end],x) .== b_sub[k][1:1] )
    @constraint(sub, A_V[k][2:end,1:end]*x .<= 0 )

    optimize!(sub)

    if termination_status(sub) == MOI.OPTIMAL
        println("--- Result from sub-problem $k: ---")
        println("Objective value: ", JuMP.objective_value(sub))
        println("x: ", JuMP.value.(x))
    else
        println("Optimize was not succesful. Return code: ", termination_status(sub))
    end
end
```

Figure 7: Sub-problem code

In the depicted subproblem code (Figure 7), our objective is to identify a vector x that minimizes the reduced cost. Introducing this new column, corresponding to an extreme point of the subproblem's feasible region, has the potential to decrease the objective value of the master problem in our minimization task.

It's important to emphasize that the subproblem formulation includes two types of constraints: those that represent the transformed supply constraints, and those that dictate route activation. The unique structure of our problem necessitates a combination of continuous variables for routing quantities (the first three components of x) and binary variables for route usage decisions (the last three components of x).

To ensure the model accurately reflects the decision to utilize a route, the variables x_4, x_5, and x_6 must be constrained to binary values, either 0, signifying the route is not used, or 1, indicating the route is active. This binary nature captures the essence of our decision-making process within the network design: whether or not to establish a particular connection, influenced by the costs and capacities involved.

Iterations

The column generation process unfolds over multiple iterations, each refining the solution to the master problem by considering newly generated columns from the subproblems. The details of these iterations are as follows:

First Iteration: Solving the subproblems yields the following results:

- Sub-problem 1 yields an objective value of 0.0 with solution vector $x = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0]$.
- Sub-problem 2 yields an objective value of -2.0, indicating a potential improvement, with solution vector $x = [1.0, 0.0, 1.0, 1.0, 0.0, 1.0]$.
- Sub-problem 3 returns to the objective value of 0.0 with solution vector $x = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0]$.

Given the negative objective value from sub-problem 2, we identify a promising extreme point to be added to the master problem. The updated extreme points, known as the columns generated, are as follows:

```
# Extreme point after first call to the sub
X[1] = [1 0 0 1 0 0]'
X[2] = [0 2 0 0 1 0; 1 0 1 1 0 1]'
X[3] = [0 0 1 0 0 1]'
```

Additionally, the count of extreme points for sub-problem 2 is now two, as indicated by the updated $P = [1, 2, 1]$.

Second Iteration: Upon solving the master problem with the new columns, we achieve an objective value of 38, with dual prices vector π updated to [10.0, 7.0, 12.0] and κ_k to [2.0, 0.0, 0.0]. Solving the subproblems with these new parameters produces the following results:

- Sub-problem 1 presents an improved objective value of -2.0, suggesting an addition to the master problem, with solution vector $x = [0.0, 0.0, 1.0, 0.0, 0.0, 1.0]$.
- Sub-problem 2 maintains an objective value of 0.0 with solution vector $x = [0.0, 2.0, 0.0, 0.0, 1.0, 0.0]$.
- Sub-problem 3 also shows an objective value of 0.0 with solution vector $x = [0.0, 0.0, 1.0, 0.0, 0.0, 1.0]$.

The extreme point from sub-problem 1 is subsequently added to the master problem. The updated extreme points after the second iteration are:

```
# Extreme point after second call to the sub
X[1] = [1 0 0 1 0 0; 0 0 1 0 0 1]'
X[2] = [0 2 0 0 1 0; 1 0 1 1 0 1]'
X[3] = [0 0 1 0 0 1]'
```

The counts of extreme points are now $P = [2, 2, 1]$.

Convergence: The third iteration, after re-solving the master problem, yields an objective value of 38. The dual prices vector π is updated to [11.0, 7.0, 11.0], and the κ_k vector becomes [1.0, 0.0, 1.0]. Subsequent solutions to the subproblems with these values are:

- Sub-problem 1 results in an objective value of 0.0 with solution vector $x = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0]$.
- Sub-problem 2 continues with an objective value of 0.0 and solution vector $x = [0.0, 2.0, 0.0, 0.0, 1.0, 0.0]$.
- Sub-problem 3 repeats an objective value of 0.0, maintaining solution vector $x = [1.0, 0.0, 0.0, 1.0, 0.0, 0.0]$.

With the reduced costs in all subproblems now at 0, no further improvements can be made through the addition of new columns. This indicates convergence of the column generation process.

Consequently, we conclude that the minimum cost achievable for this Fixed Charge Transportation Problem (FCTP) is 38. This optimized solution is reached by a combination of routes as indicated by the extreme points in the master problem, respecting the supply and demand constraints, and minimizing the total transportation cost.

Structure Check

The decomposition proposed in Task 2 does not satisfy the "Identical Sub-problem Assumption" mentioned in the lectures. This assumption would require that each sub-problem be structurally identical. However, in our case, the sub-problems differ due to the dependency of m_{ij} on the specific values of supply a_i and demand b_j, which vary for each source $i \in S$ and destination $j \in T$.

For instance, in the second source, we observe a value of -2 in the constraint matrix where all the other sources have a value of -1. This indicates that the minimum of supply and demand, denoted as $m_{ij} = \min\{a_i, b_j\}$, differs among sources, leading to non-identical sub-problems. If m_{ij} had been the same across all sources, such as -1, then we could consider the sub-problems identical.

Branch-and-Price

In Task 5, we address an expanded Fixed Charge Transportation Problem (FCTP) featuring 5 sources and 5 sinks. The branch-and-price algorithm, a fusion of branch-and-bound and Dantzig-Wolfe decomposition, is deployed to navigate through this complex instance.

Data Preparation

A similar procedure to task 2 is followed to structure the constraint matrix A and vector b, which underpin our optimization efforts. Figures 8 and 9 depict the data and its preparation.

```
using LinearAlgebra

# include data
# cost per unit sent from Source i to  Terminal j
c = [
        1.5  5.8  2.3  1.0  6.2
        8.9  4.9  9.5  8.2  6.3
        8.2  3.4  8.7  7.9  4.7
        5.0  8.9  5.7  3.0  9.7
        4.2  2.2  4.5  4.5  2.9
    ]

# Supply for sources
supply = [
    2
    4
    5
    8
    10]

# Demand
demand = [3
    9
    3
    6
    8]

# fixed cost of connecting customer from Source i to Terminal j
f = [
    11.0  12.0  14.0  20.0  14.0
    17.0  15.0  15.0  12.0  13.0
    11.0  16.0  17.0  14.0  10.0
    11.0  13.0  17.0  18.0  11.0
    11.0  12.0  15.0  18.0  11.0]
```

Figure 8: Data for Task 5

```
m = [min(ai, bj) for ai in supply, bj in demand]  # Calculate mij as min{ai, bj}

# Initialize the matrix A with zeros
# For 5 sources and 5 terminals, we have 5*5=25 xij and yij variables each
# Additionally, there are 5 supply constraints, 5 demand constraints, and 25 xij-yij constraints
# So, the matrix A will have 5+5+25 rows and 25+25 columns for xij and yij
A = zeros(5 + 5 + 25, 25 + 25)

# Populate the matrix for supply constraints
for i in 1:5
    A[i, (i-1)*5+1:i*5] .= 1  # Set 1 for xij variables corresponding to supply ai
end

# Populate the matrix for demand constraints
for j in 1:5
    A[5+j, j:5:end] .= 1  # Set 1 for xij variables corresponding to demand bj
end

# Populate the matrix for xij - mij*yij <= 0 constraints
for i in 1:5, j in 1:5
    row_index = 10 + (i-1)*5 + j  # Starting from row 10, for each xij-yij constraint
    A[row_index, (i-1)*5 + j] = 1            # Set 1 for xij
    A[row_index, 25 + (i-1)*5 + j] = -m[i, j]  # Set -mij for yij
end

# Since RHS is not part of the matrix A, we create it separately
RHS = [supply; demand; zeros(25)]  # RHS for supply, demand, and xij-yij constraints

A, RHS
#*****************************************************************************

cols_order = [1, 2, 3, 4, 5, 26, 27, 28, 29, 30,
              6, 7, 8, 9, 10, 31, 32, 33, 34, 35,
              11, 12, 13, 14, 15, 36, 37, 38, 39, 40,
              16, 17, 18, 19, 20, 41, 42, 43, 44, 45,
              21, 22, 23, 24, 25, 46, 47, 48, 49, 50]

# # Create a copy of A to reorder
A_reordered = copy(A)
b_reordered = copy(RHS)

# # Reorder columns
# where each block corresponds to the constraints for one source
A = A_reordered[:, cols_order]
b = b_reordered

# Concatenate each row into a column vector
cost_vector = reshape(c', 25, 1)
fixed_vector = reshape(f',25,1)

c_titled = vcat(cost_vector,fixed_vector)
c_temp = copy(c_titled)
c_titled = c_temp[cols_order]'
```

Figure 9: Data Preparation

Branch-and-Price Algorithm

In Figure 10, we delve into the branch-and-price framework, utilizing the ColGen-Generic.jl module to facilitate the decomposition aspect of the algorithm's architecture. This figure captures the definition of the integer programming (IP) model for the Fixed Charge Transportation Problem (FCTP), inclusive of supply, demand, and route constraints. It features the integer variable x[i,j] that denotes the quantity to be transported and the binary variable y[i,j] indicating whether a route is operational.

The model is outlined as follows, with 'ub[i,j]' designating the computed upper bounds, which effectively cap the quantity transferable from each source i to each sink j, reflective of m_{ij} within the problem's context:

```
@variable(FCTPModel, 0 <= x[i=1:m, j=1:n] <= ub[i,j], Int)
```

The final part of the code snippet exemplifies the allocation of each block. The goal is to embody both the source and the route constraints, defining the structural parameters of each subproblem's variable set within the optimization problem. This structuring is instrumental for the subsequent application of the Dantzig-Wolfe decomposition, setting the stage for a refined branch-and-price algorithm execution.

```julia
include("ColGenGeneric.jl")
using Gurobi, JuMP
using .DW_ColGen

function FCTP_MIP(c,f,mij,supply,demand)
    # 5x5 set of variables
    (m,n) = size(mij)

    myModel = Model(Gurobi.Optimizer)
    @variable(myModel, 0 <= x[i=1:m,j=1:n] <= mij[i,j], Int)
    @variable(myModel, y[1:m,1:n], Bin)

    @objective(myModel, Min, sum(c[i,j]*x[i,j] + f[i,j]*y[i,j] for i=1:m for j=1:n))

    # Demand constraints
    @constraint(myModel, [j=1:n], sum(x[i,j] for i=1:m) == demand[j])

    # Supply constraints
    @constraint(myModel, SupplyCons[i=1:m], sum(x[i,j] for j=1:n) == supply[i])
    # Route constraints
    @constraint(myModel, routeCons[i=1:m, j=1:n], x[i,j] <= mij[i,j]*y[i,j])

    # Define blocks (Each block becomes a sub-problem)
    # we have two constraints in the sub
    blocks = Vector{Vector{ConstraintRef}}(undef, m)
    for i in 1:m
        blocks[i] = Vector{ConstraintRef}()  # Initialize as an empty vector
        push!(blocks[i], SupplyCons[i])
        for j in 1:n
            push!(blocks[i],routeCons[i,j])
        end
    end
    return myModel, blocks
end
```

Figure 10: Branch-and-Price call function

Lastly, the main function is called, streamlining the application of Dantzig-Wolfe column generation. Underneath the hood, the files from week 4, DecompMatrix.jl and JumpModelToMatrix.jl are leveraged, transforming the JuMP model into a matrix form suitable for decomposition.

```julia
function main()
    # Builds matrix A from data
    include("Task5_Build_matrix_A.jl")
    FCTP_Model, blocks = FCTP_MIP(c,f,m,supply,demand)
    DW_ColGen.DWColGenEasy(FCTP_Model, blocks)
end

main()
```

Figure 11: Branch-and-Price call function

DW results

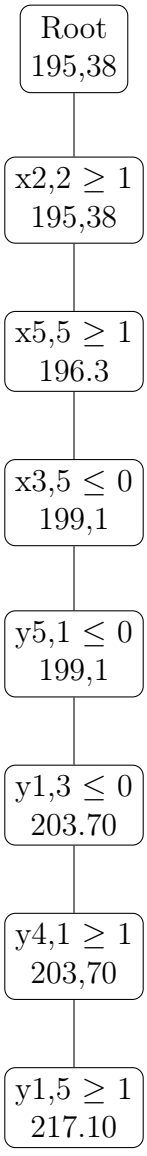
The initial results from the DW relaxation provide a lower bound of 195.38. The solution's variable values are listed below:

- $x_{1,1} = 1.0000000000000002$
- $y_{1,1} = 0.5000000000000001$
- $x_{1,3} = 0.9999999999999998$
- $y_{1,3} = 0.4999999999999999$
- $x_{2,2} = 3.9999999999999996$
- $y_{2,2} = 0.9999999999999999$
- $x_{3,5} = 5.0$
- $y_{3,5} = 1.0$
- $x_{4,1} = 2.0$
- $y_{4,1} = 1.0$
- $x_{4,4} = 6.0$
- $y_{4,4} = 1.0$
- $x_{5,2} = 5.0$
- $x_{5,3} = 2.0$

- $x_{5,5} = 2.9999999999999982$
- $y_{5,2} = 0.6190476190476191$
- $y_{5,3} = 0.6666666666666667$
- $y_{5,5} = 0.714285714285714$

Exploring the Branch-and-Bound Tree

The exploration initiates at the root node and spans eight additional nodes, adhering to a Depth First Search strategy and branching based on the largest variable values.



The resulting decision variables from this exploration are as follows:

- $x_{1,1} = 1.0$
- $y_{1,1} = 1.0$
- $x_{1,5} = 1.0$
- $y_{1,5} = 1.0$
- $x_{2,2} = 4.0$
- $y_{2,2} = 1.0$
- $x_{3,2} = 5.0$
- $y_{3,2} = 1.0$
- $x_{4,1} = 2.0$
- $y_{4,1} = 1.0$
- $x_{4,4} = 6.0$
- $y_{4,4} = 1.0$
- $x_{5,3} = 3.0$
- $x_{5,5} = 7.0$
- $y_{5,3} = 1.0$
- $y_{5,5} = 1.0$

Our approach to branching within the `FCTP_MIP` function necessitated iterative updates for each new branching rule. Figure 12 below illustrates the total adjustments to the function.

Bounds and Conclusions

During our analysis on the branching, we successfully uncovered an integer solution valued at 217.0. Regrettably, a lapse in our record-keeping has led to a present non-integer objective value of 217.10, despite all variables retaining their integer status. This discrepancy is likely attributable to rounding errors. We remain confident, however, in the existence of an integer upper bound at 217. Our forthcoming efforts will be directed towards pinpointing a solution that adheres to the constraints delineated by the original Dantzig-Wolfe lower bound and this newly recognized integer upper bound.

```
function FCTP_MIP(c,f,mij,supply,demand)
    # 5x5 set of variables
    (m,n) = size(mij)

    myModel = Model(Gurobi.Optimizer)
    @variable(myModel, 0 <= x[i=1:m,j=1:n] <= mij[i,j], Int)
    @variable(myModel, y[1:m,1:n], Bin)

    @objective(myModel, Min, sum(c[i,j]*x[i,j] + f[i,j]*y[i,j] for i=1:m for j=1:n))

    # Demand constraints
    @constraint(myModel, [j=1:n], sum(x[i,j] for i=1:m) == demand[j])
    # Supply constraints
    @constraint(myModel, SupplyCons[i=1:m], sum(x[i,j] for j=1:n) == supply[i])
    # Route constraints
    @constraint(myModel, routeCons[i=1:m, j=1:n], x[i,j] <= mij[i,j]*y[i,j])

    @constraint(myModel, branch1, x[2,2] >= 1)
    @constraint(myModel, branch2, x[5,5] >= 1)
    @constraint(myModel, branch3, x[3,5] <= 0)
    @constraint(myModel, branch4, y[5,1] <= 0)
    @constraint(myModel, branch5, y[1,3] <= 0)
    @constraint(myModel, branch6, y[4,1] >= 1)
    @constraint(myModel, branch7, y[1,5] >= 1)
    @constraint(myModel, branch8, y[1,1] >= 1)

    blocks = Vector{Vector{ConstraintRef}}(undef, m)
    for i in 1:m
        blocks[i] = Vector{ConstraintRef}()  # Initialize as an empty vector
        push!(blocks[i], SupplyCons[i])
        for j in 1:n
            push!(blocks[i],routeCons[i,j])
        end
    end
    push!(blocks[2], branch1)
    push!(blocks[5], branch2)
    push!(blocks[3], branch3)
    push!(blocks[5], branch4)
    push!(blocks[1], branch5)
    push!(blocks[4], branch6)
    push!(blocks[1], branch7)
    push!(blocks[1], branch8)
    return myModel, blocks
end
```

Figure 12: Branch rules in FCTP$_M$ *IP function*

List of Figures