DTU42136, Large Scale Optimization using Decomposition

Assignment 1 Improvement

Edward J. Xu (Jie Xu), s181238, DTU Management

March 21th, 2019

1 Introduction

I don't mean to update my report. It's just the small bug in the template that causes some misunderstanding. I found it finally, and I am writing to point it out and hope it's helpful. Besides, there is an improvement in my Benders Decomposition function, which doesn't affect my calculation, but accelerates it. I will write about it in the section 3.

2 Bug in Benders Decomposition Template

The problem I mentioned in page 3 of my final version of the report is:

> There is something wrong in the result. Because the q is actually 600 in the final iteration, the iteration causes the iteration to stop by raising the lower bound by 20. The q doesn't equal to "obj_sub", and it's 600 all the time since about 131-th iteration. Finally, it just return a result with sum equals 1270. while I can't figure out how the result can become 1270 with q being 680 instead of 600.

When the q doesn't equal to "obj_sub", I thought there was something wrong with the calculation of **y**. But it's actually one step missing in the calculation of x, which is to make sure that the q equals to "obj_sub". When two bounds meet, the **y** is the right answer, but the current solution of **x** can be wrong if the q doesn't equal to "obj_sub". So, to make sure the calculation stop only if the two bounds meet and the q equals to "obj_sub", I make the following change in the template and my function (line 91):

```
# while ((boundUp - boundLow > epsilon) && (timesIteration <= timesIterationMax))
while ((!((boundUp - boundLow <= epsilon) && ((result_q == obj_sub)))) &&
    (timesIteration <= timesIterationMax))
```

Then, the returned result is:

```
boundUp: 1270.0, boundLow: 1270.0, difference: 0.0
mat_x: [0   0   0   0   0   0 0   0   0   0
        0   0   0   0   0   0 0   0   0   0
        0   0   0   0   0   0 0   0   0   0
        0   0   0   0   0   0 0   0   0   0
        0   0   0   0   0   0 0   0   0   0
        10  30  0   10  40  0 0   0   0   0
        0   0   0   0   0   0 0   0   0   0
        0   0   0   0   0   0 0   0   15  0
        0   0   0   0   0   0 0   0   0   0
        0   0   20  0   0   0 25  35  0   0]
mat_y: [0 1 0 0 0 0 0 0 0 0
        0 0 0 0 0 1 0 0 0 0
        0 0 0 0 0 0 0 0 0 1
        1 0 0 0 1 0 0 0 0 0
        0 0 0 0 0 1 0 0 0 0
        1 1 0 1 1 0 0 1 0 0
        0 1 1 0 0 0 0 0 0 0
        0 0 0 0 0 0 0 0 1 0
        0 0 0 0 1 0 0 0 0 0
        0 0 1 0 0 1 1 1 1 0]
```

where there is some difference in the result of **x**.

Here is the table showing the objectives from the sub-problem and master problem of each iteration.

```
 ----- --------- ---------- --------- ------- -------- -----------
 Seq   boundUp   boundLow   obj_mas   q       sub/ray  obj_sub/ray
 ----- --------- ---------- --------- ------- -------- -----------
 1     Inf       536.0      536.0     -4.0    ray      1.0
 2     1280.0    633.0      633.0     -7.0    sub      740.0
 3     1280.0    930.0      930.0     300.0   sub      640.0
 4     1270.0    1110.0     1110.0    470.0   sub      640.0
 5     1270.0    1110.0     1110.0    470.0   sub      640.0
 6     1270.0    1140.0     1140.0    470.0   sub      660.0
 7     1270.0    1210.0     1210.0    640.0   sub      600.0
 8     1270.0    1230.0     1230.0    660.0   sub      720.0
 9     1270.0    1240.0     1240.0    640.0   sub      700.0
 10    1270.0    1240.0     1240.0    640.0   sub      680.0
 11    1270.0    1240.0     1240.0    660.0   sub      680.0
 12    1270.0    1250.0     1250.0    640.0   sub      700.0
```

```
  13     1270.0     1270.0      1270.0     600.0   sub        680.0
  14     1270.0     1270.0      1270.0     600.0   sub        600.0
 ______ __________ ___________ __________ _______ __________ ______________
```

where there is only one more iteration (14) compared with that in my previous calculation. This one more iteration (14) is make sure the q equals to "obj_sub" and return me the right **x**. Notice that the times of iterations have been largely reduced, and I will talk about this in section 3.

Don't forget there is a missing part in the objective function of the calculation, which is:

$$\text{obj} = \text{result} - \sum_{m,n} \rho_n^m \alpha_n^m \tag{1}$$

So my final result for model one is:

```
obj:  730
mat_x:  [0    0    0    0    0    0 0    0    0    0
         0    0    0    0    0    0 0    0    0    0
         0    0    0    0    0    0 0    0    0    0
         0    0    0    0    0    0 0    0    0    0
         0    0    0    0    0    0 0    0    0    0
         10   30   0    10   40   0 0    0    0    0
         0    0    0    0    0    0 0    0    0    0
         0    0    0    0    0    0 0    0    15   0
         0    0    0    0    0    0 0    0    0    0
         0    0    20   0    0    0 25   35   0    0]
mat_y:  [0 1 0 0 0 0 0 0 0 0
         0 0 0 0 0 1 0 0 0 0
         0 0 0 0 0 0 0 0 0 1
         1 0 0 0 1 0 0 0 0 0
         0 0 0 0 0 1 0 0 0 0
         1 1 0 1 1 0 0 1 0 0
         0 1 1 0 0 0 0 0 0 0
         0 0 0 0 0 0 0 0 1 0
         0 0 0 0 1 0 0 0 0 0
         0 0 1 0 0 1 1 1 1 0]
```

The visualization of the result is the following two figures, which are quite different from fig.2 and fig.3 in my final version of the report.

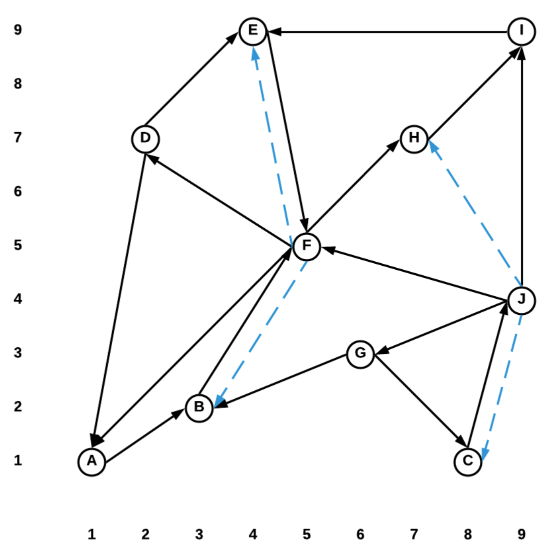

Figure 1. Corrected Existing Gas Pipe Network and Newly Built Pipes

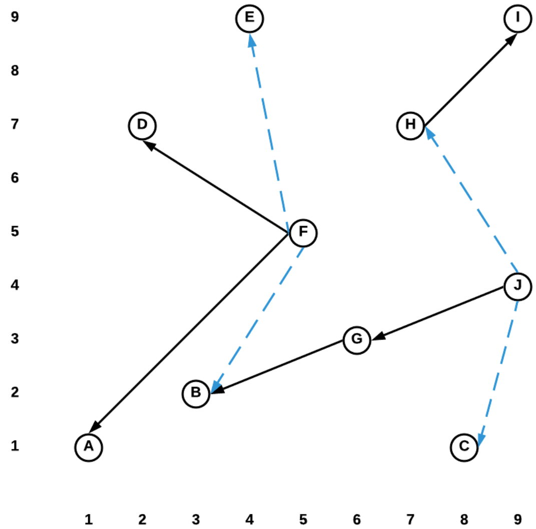

Figure 2. Corrected Operating Pipes in Existing Gas Pipe Network and Newly Built Pipes

3 A Small Improvement in Master Problem in the Function to Accelerate Calculation

Additionally, I make some change in the Benders Decomposition function to accelerate the calculation when there is constraints on $\mathbf{y}$. The original constraints on $\mathbf{y}$ is shown below:

$$\mathbf{A_2} = \mathbf{0}_{100} \tag{2}$$
$$\mathbf{B_2} = \mathbf{I}_{100} \tag{3}$$
$$\mathbf{A_2x} + \mathbf{B_2y} \geq \mathbf{b} \tag{4}$$
$$\mathbf{y} \leq \mathbf{y}_{\text{min}} \tag{5}$$

I change it to:

$$\mathbf{y}_{\text{min}} \leq \mathbf{y} \leq \mathbf{y}_{\text{min}} \tag{6}$$

and the change in code (line 21) is:

```
    @constraint(model_mas, vec_y[1:n_y] .>= vec_min_y)
```

with other changes in the argument of the function.

This change means the available $\mathbf{y}$ values are restricted from the very beginning. All feasible $\mathbf{y}$ leads to possible solution of sub problem. That's why the first model can be solved without ray problem.

This change can tremendously increase the speed when there is a reasonable $\mathbf{y}_{\text{min}}$, because it reduces the chance of unbounded sub-problem, which can be seen in the table of iteration result:

Seq	boundUp	boundLow	obj_mas	q	sub/ray	obj_sub/ray
1	Inf	536.0	536.0	−4.0	ray	1.0
2	1280.0	633.0	633.0	−7.0	sub	740.0
3	1280.0	930.0	930.0	300.0	sub	640.0
4	1270.0	1110.0	1110.0	470.0	sub	640.0
5	1270.0	1110.0	1110.0	470.0	sub	640.0
6	1270.0	1140.0	1140.0	470.0	sub	660.0
7	1270.0	1210.0	1210.0	640.0	sub	600.0
8	1270.0	1230.0	1230.0	660.0	sub	720.0
9	1270.0	1240.0	1240.0	640.0	sub	700.0
10	1270.0	1240.0	1240.0	640.0	sub	680.0
11	1270.0	1240.0	1240.0	660.0	sub	680.0
12	1270.0	1250.0	1250.0	640.0	sub	700.0
13	1270.0	1270.0	1270.0	600.0	sub	680.0
14	1270.0	1270.0	1270.0	600.0	sub	600.0

which reduces 135 times of iteration.

The master problem in Benders Decomposition of MILP then becomes:

$$
\begin{aligned}
\min \quad & \mathbf{f}^T\mathbf{y} + q && (7)\\
\text{s.t.} \quad & \overline{\mathbf{u_j^r}} \cdot (\mathbf{b} - \mathbf{B}\mathbf{y}) \leq 0 \quad \forall j && (8)\\
& \overline{\mathbf{u_i^p}} \cdot (\mathbf{b} - \mathbf{B}\mathbf{y}) \leq q \quad \forall i && (9)\\
& \mathbf{y}_{\min} \leq \mathbf{y} \leq \mathbf{y}_{\min} && (10)\\
& q \in R && (11)
\end{aligned}
$$

So, it actually means we don't use the standard mixed integer linear programming (MILP) format to write the code. But the difference is very small:

$$
\begin{aligned}
\min \quad & \mathbf{c}^T\mathbf{x} + \mathbf{f}^T\mathbf{y} && (12)\\
\text{s.t.} \quad & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} \geq \mathbf{b} && (13)\\
& \mathbf{y}_{\min} \leq \mathbf{y} \leq \mathbf{y}_{\min} && (14)\\
& \mathbf{x} \geq 0 && (15)
\end{aligned}
$$

where the $\mathbf{y}$ is vector of integer variables.

Moreover, there is small change in the following transformation equations, which is the remove of $\mathbf{A_2}$, $\mathbf{B_2}$ and $\mathbf{b_2}$.

$$\mathbf{x} = \left[x_1^1, x_2^1, \ldots, x_{10}^1, x_1^2, x_2^2, \ldots, x_n^m, x_{10}^{10}\right]^T \tag{16}$$

$$\mathbf{y} = \left[y_1^1, y_2^1, \ldots, y_{10}^1, y_1^2, y_2^2, \ldots, y_n^m, y_{10}^{10}\right]^T \tag{17}$$

$$\mathbf{c} = \left[\beta_1^1, \beta_2^1, \ldots, \beta_{10}^1, \beta_1^2, \beta_2^2, \ldots, \beta_n^m, \beta_{10}^{10}\right]^T \tag{18}$$

$$\mathbf{f} = \left[\rho_1^1, \rho_2^1, \ldots, \rho_{10}^1, \rho_1^2, \rho_2^2, \ldots, \rho_n^m, \rho_{10}^{10}\right]^T \tag{19}$$

$$\mathbf{Y} = \{0, 1\} \tag{20}$$

$$\mathbf{A_1} = -\mathbf{I}_{100} \tag{21}$$

$$\mathbf{A_3} = \begin{bmatrix} 0 & -1 & -1 & \cdots & -1, & 1 & 0 & 0 & \cdots & 0, & 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \cdots & 0, & -1 & 0 & -1 & \cdots & -1, & 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & \cdots & 0, & 0 & 0 & 1 & \cdots & 0, & -1 & -1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{22}$$

$$\mathbf{B_1} = 170 \times \mathbf{I}_{100} \tag{23}$$

$$\mathbf{B_3} = \mathbf{0}_{100} \tag{24}$$

$$\mathbf{b} = \left[\underbrace{0, 0, \ldots, 0}_{100}, \underbrace{(\kappa_1 - \omega_1), (\kappa_2 - \omega_2), \ldots, (\kappa_m - \omega_m), \ldots, (\kappa_{10} - \omega_{10})}_{10}\right]^T \tag{25}$$

which is the same as the transforming equations in the second model.

4 Appendix: Updated Code

Function file "BendersMilp_EDXU" to calculate mixed integer linear problem using Benders Decomposition:

```julia
# Benders Algorithm for MILP with Sub and Ray Problems
# Version: 5.0
# Author: Edward J. Xu, edxu96@outlook.com
# Date: March 21th, 2019
module BendersMilp_EDXU
    export BendersMilp
    using JuMP
    using GLPKMathProgInterface
    using PrettyTables
    function BendersMilp(; n_x, n_y, vec_min_y, vec_max_y, vec_c, vec_f, vec_b, mat_a, mat_b, epsilon, timesIterationMax)
        println("————————————————————————————————————————————————————————————————\n",
                "———————————————————————— 1/4. Begin Optimization ————————————————————————\n",
                "————————————————————————————————————————————————————————————————\n")
        # Define Master problem
        n_constraint = length(mat_a[:, 1])
```

```julia
    model_mas = Model(solver = GLPKSolverMIP())
    @variable(model_mas, q)
    @variable(model_mas, vec_y[1: n_y] >= 0, Int)
    @objective(model_mas, Min, (transpose(vec_f) * vec_y + q)[1])
    @constraint(model_mas, vec_y[1: n_y] .<= vec_max_y)
    @constraint(model_mas, vec_y[1: n_y] .>= vec_min_y)


    function solve_master(vec_uBar, opt_cut::Bool)
        if opt_cut
            @constraint(model_mas, (transpose(vec_uBar) * (vec_b - mat_b * vec_y))[1] <= q)
        else  # Add feasible cut Constraints
            @constraint(model_mas, (transpose(vec_uBar) * (vec_b - mat_b * vec_y))[1] <= 0)
        end
        @constraint(model_mas, (transpose(vec_uBar) * (vec_b - mat_b * vec_y))[1] <= q)
        solve(model_mas)
        vec_result_y = getvalue(vec_y)
        return getobjectivevalue(model_mas)
    end


    function solve_sub(vec_uBar, vec_yBar, n_constraint, vec_b, mat_b, mat_a, vec_c)
        model_sub = Model(solver = GLPKSolverLP())
        @variable(model_sub, vec_u[1: n_constraint] >= 0)
        @objective(model_sub, Max, (transpose(vec_b - mat_b * vec_yBar) * vec_u)[1])
        constraintsForDual = @constraint(model_sub, transpose(mat_a) * vec_u .<= vec_c)
        solution_sub = solve(model_sub)
        print("——————————————— Sub Problem ———————————————\n")  # , model_sub)
        vec_uBar = getvalue(vec_u)
        if solution_sub == :Optimal
            vec_result_x = zeros(length(vec_c))
            vec_result_x = getdual(constraintsForDual)
            return (true, getobjectivevalue(model_sub), vec_uBar, vec_result_x)
        end
        if solution_sub == :Unbounded
            print("Not solved to optimality because feasible set is unbounded.\n")
            return (false, getobjectivevalue(model_sub), vec_uBar, repeat([NaN], length(vec_c)))
        end
    end


    function solve_ray(vec_uBar, vec_yBar, n_constraint, vec_b, mat_b, mat_a)
        # model_ray = Model(solver = GurobiSolver())
        model_ray = Model(solver = GLPKSolverLP())
        @variable(model_ray, vec_u[1: n_constraint] >= 0)
        @objective(model_ray, Max, 1)
        @constraint(model_ray, (transpose(vec_b - mat_b * vec_yBar) * vec_u)[1] == 1)
        @constraint(model_ray, transpose(mat_a) * vec_u .<= 0)
        solve(model_ray)
        print("——————————————— Ray Problem ———————————————\n")  # , model_ray)
        vec_uBar = getvalue(vec_u)
        obj_ray = getobjectivevalue(model_ray)
        return (obj_ray, vec_uBar)
    end


    # Begin Calculation ———————————————————————————————————————————————————————
    let
        boundUp = Inf
        boundLow = - Inf
        epsilon = 0
        # initial value of master variables
        vec_uBar = zeros(n_constraint, 1)
        vec_yBar = zeros(n_y, 1)
        vec_result_x = length(n_x)
        dict_obj_mas = Dict()
        dict_q = Dict()
        dict_obj_sub = Dict()
        dict_obj_ray = Dict()
        dict_boundUp = Dict()
        dict_boundLow = Dict()
        obj_sub = 0
        timesIteration = 1
        # Must make sure "result_q == obj_sub" in the final iteration
        # while ((boundUp - boundLow > epsilon) && (timesIteration <= timesIterationMax))  !!!
        while ((!((boundUp - boundLow <= epsilon) && ((result_q == obj_sub)))) &&
            (timesIteration <= timesIterationMax))
            (bool_solutionSubModel, obj_sub, vec_uBar, vec_result_x) = solve_sub(vec_uBar, vec_yBar, n_constraint,
                                                                                  vec_b, mat_b, mat_a, vec_c)
            if bool_solutionSubModel
                boundUp = min(boundUp, obj_sub + (transpose(vec_f) * vec_yBar)[1])
            else
                (obj_ray, vec_uBar) = solve_ray(vec_uBar, vec_yBar, n_constraint, vec_b, mat_b, mat_a)
```

```
                end
                obj_mas = solve_master(vec_uBar, bool_solutionSubModel)
                vec_yBar = getvalue(vec_y)
                boundLow = max(boundLow, obj_mas)
                dict_boundUp[timesIteration] = boundUp
                dict_boundLow[timesIteration] = boundLow
                if bool_solutionSubModel
                    dict_obj_mas[timesIteration] = obj_mas
                    dict_obj_sub[timesIteration] = obj_sub
                    result_q = getvalue(q)
                    dict_q[timesIteration] = result_q
                    println("-------------------- Result in $(timesIteration)-th Iteration with Sub ",
                            "----------------------\n", "boundUp: $(round(boundUp, digits = 5)), ",
                            "boundLow: $(round(boundLow, digits = 5)), obj_mas: $(round(obj_mas, digits = 5)), ",
                            "q: $result_q, obj_sub: $(round(obj_sub, digits = 5)).")
                else
                    dict_obj_mas[timesIteration] = obj_mas
                    dict_obj_ray[timesIteration] = obj_ray
                    result_q = getvalue(q)
                    dict_q[timesIteration] = result_q
                    println("-------------------- Result in $(timesIteration)-th Iteration with Ray ",
                            "----------------------\n", "boundUp: $(round(boundUp, digits = 5)), ",
                            "boundLow: $(round(boundLow, digits = 5)), obj_mas: $(round(obj_mas, digits = 5)), ",
                            "q: $result_q, obj_ray: $(round(obj_ray, digits = 5)).")
                end
                timesIteration += 1
            end  # ------------------------------------------------------------------------------------------------
            println("obj_mas: $(getobjectivevalue(model_mas))")
            println("------------------------------ Master Problem ------------------------------\n")
            println(model_mas)
            println("--------------------------------------------------------------------------------\n",
                    "-------------------------------- 2/4. Result --------------------------------\n",
                    "--------------------------------------------------------------------------------")
            println("boundUp: $(round(boundUp, digits = 5)), boundLow: $(round(boundLow, digits = 5)), ",
                    "difference: $(round(boundUp - boundLow, digits = 5))")
            println("vec_x: $vec_result_x")
            vec_result_y = getvalue(vec_y)
            result_q = getvalue(q)
            println("vec_y: $vec_result_y")
            println("result_q: $result_q")
            println("--------------------------------------------------------------------------------\n",
                    "-------------------------- 3/4. Iteration Result --------------------------\n",
                    "--------------------------------------------------------------------------------")
            # Initialize
            seq_timesIteration = collect(1: (timesIteration - 1))
            vec_boundUp = zeros(timesIteration - 1)
            vec_boundLow = zeros(timesIteration - 1)
            vec_obj_subRay = zeros(timesIteration - 1)
            vec_obj_mas = zeros(timesIteration - 1)
            vec_q = zeros(timesIteration - 1)
            vec_type = repeat(["ray"], (timesIteration - 1))
            #
            for i = 1: (timesIteration - 1)
                vec_obj_mas[i] = round(dict_obj_mas[i], digits = 5)
                vec_boundUp[i] = round(dict_boundUp[i], digits = 5)
                vec_boundLow[i] = round(dict_boundLow[i], digits = 5)
                vec_q[i] = round(dict_q[i], digits = 5)
                if haskey(dict_obj_sub, i)
                    vec_type[i] = "sub"
                    vec_obj_subRay[i] = round(dict_obj_sub[i], digits = 5)
                else
                    vec_obj_subRay[i] = round(dict_obj_ray[i], digits = 5)
                end
            end
            table_iterationResult = hcat(seq_timesIteration, vec_boundUp, vec_boundLow,
                                         vec_obj_mas, vec_q, vec_type, vec_obj_subRay)
            pretty_table(table_iterationResult,
                         ["Seq", "boundUp", "boundLow", "obj_mas", "q", "sub/ray", "obj_sub/ray"],
                         compact; alignment=:l)
        end
        println("--------------------------------------------------------------------------------\n",
                "---------------------------- 4/4. Nominal Ending ----------------------------\n",
                "--------------------------------------------------------------------------------\n")
    end
end
```

File "OptiGas_DTU42136_EDXU" to calculate the assignment:

```
# OptiGas: Optimize Gas Network using Benders Algorithm
# Edward J. Xu, edxu96@outlook.com
# March 21th, 2019
push!(LOAD_PATH, "$(homedir())/Desktop/OptiGas, DTU42136")
cd("$(homedir())/Desktop/OptiGas, DTU42136")
```

```
using BendersMilp_EDXU
using LinearAlgebra
# ------------------------------------------------------------------------
# 1. Parameters
vec_nameNodes = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]
numNodes = length(vec_nameNodes)
vec_xNode = [1    3    8    2    4    5    6    7    9    9]
vec_yNode = [1    2    1    7    9    5    3    7    9    4]
mat_arcTwoNodes = [0    1    0    0    0    0    0    0    0    0;
                   0    0    0    0    0    1    0    0    0    0;
                   0    0    0    0    0    0    0    0    0    1;
                   1    0    0    0    1    0    0    0    0    0;
                   0    0    0    0    0    1    0    0    0    0;
                   1    0    0    1    0    0    0    1    0    0;
                   0    1    1    0    0    0    0    0    0    0;
                   0    0    0    0    0    0    0    0    1    0;
                   0    0    0    0    1    0    0    0    0    0;
                   0    0    0    0    0    1    1    0    1    0]
vec_netEject =  [0     0    0     0     0    90     0     0    0    80 ]
vec_netInject = [10    30   20    10    40   0     25    20   15    0 ]
mat_distance = zeros(Float64, numNodes, numNodes)
for m = 1: numNodes
    for n = 1: numNodes
        mat_distance[m, n] = floor(sqrt( (vec_xNode[m] - vec_xNode[n]) * (vec_xNode[m] - vec_xNode[n]) +
                                         (vec_yNode[m] - vec_yNode[n]) * (vec_yNode[m] - vec_yNode[n]) ) )
    end
end
mat_fixedCost = zeros(Float64, numNodes, numNodes)
for m = 1: numNodes
    for n = 1: numNodes
        mat_fixedCost[m, n] = 10 * mat_distance[m, n]
    end
end
# ------------------------------------------------------------------------------------------------------------
# Question 4
# Fomulation of Matrix A
mat_a1_1 = zeros(numNodes^2, numNodes^2)
for mn = 1: numNodes^2
    mat_a1_1[mn, mn] = -1
end
# mat_a2_1 = zeros(numNodes^2, numNodes^2)
mat_a3_1 = zeros(numNodes, numNodes^2)
seq_zeroTenNinety = collect(0:10:90)
for m = 1: numNodes
    mat_a3_1[m, (10 * m - 9): (10 * m)] = repeat([-1], 10)  # sent out
    mat_a3_1[m, (seq_zeroTenNinety + repeat([m],10))] = repeat([1], 10)  # sent in
    mat_a3_1[:, (11 * m - 10)] = repeat([0], 10)  # self-sending doesn't count
end
mat_a_1 = vcat(mat_a1_1, mat_a3_1)
# Fomulation of Matrix B
mat_b1_1 = zeros(numNodes^2, numNodes^2)
for mn = 1: numNodes^2
    mat_b1_1[mn, mn] = 170
end
# mat_b2_1 = Diagnal(repeat([1], numNodes^2))
mat_b3_1 = zeros(numNodes, numNodes^2)
mat_b_1 = vcat(mat_b1_1, mat_b3_1)
# Fomulation of Vector b
vec_b1_1 = zeros(numNodes^2)
# vec_b2_1 = zeros(numNodes^2)
# for m = 1: numNodes
#     for n = 1: numNodes
#         vec_b2_1[10 * (m - 1) + n] = mat_arcTwoNodes[m, n]
#     end
# end
vec_b3_1 = zeros(numNodes)
for n = 1: numNodes
    vec_b3_1[n] = vec_netInject[n] - vec_netEject[n]
end
vec_b_1 = vcat(vec_b1_1, vec_b3_1)
vec_b_1 = hcat(vec_b_1)
#
vec_max_y_1 = repeat([1], numNodes^2)
vec_max_y_1 = hcat(vec_max_y_1)
vec_min_y_1 = zeros(numNodes^2)
for m = 1: numNodes
    for n = 1: numNodes
        vec_min_y_1[10 * (m - 1) + n] = mat_arcTwoNodes[m, n]
    end
end
#
vec_c_1 = zeros(numNodes^2)
for m = 1: numNodes
```

```
    for n = 1: numNodes
        vec_c_1[10 * (m - 1) + n] = mat_distance[m, n]
    end
end
vec_c_1 = hcat(vec_c_1)
#
vec_f_1 = zeros(numNodes^2)
for m = 1: numNodes
    for n = 1: numNodes
        vec_f_1[10 * (m - 1) + n] = mat_fixedCost[m, n]
    end
end
vec_f_1 = hcat(vec_f_1)
#
BendersMilp(n_x = numNodes^2,
            n_y = numNodes^2,
            vec_min_y = vec_min_y_1,
            vec_max_y = vec_max_y_1,
            vec_c = vec_c_1,
            vec_f = vec_f_1,
            vec_b = vec_b_1,
            mat_a = mat_a_1,
            mat_b = mat_b_1,
            epsilon = 0.0001,
            timesIterationMax = 1000)
# 1280 - sum(mat_arcTwoNodes .* mat_fixedCost)
# ------------------------------------------------------------------------------------------------------------------------
# Question 5
# Fomulation of Matrix A
mat_a1_2 = zeros(numNodes^2, numNodes^2)
for mn = 1: numNodes^2
    mat_a1_2[mn, mn] = -1
end
mat_a3_2 = zeros(numNodes, numNodes^2)
seq_zeroTenNinety = collect(0:10:90)
for m = 1: numNodes
    mat_a3_2[m, (10 * m - 9): (10 * m)] = repeat([-1], 10)  # sent out
    mat_a3_2[m, (seq_zeroTenNinety + repeat([m],10))] = repeat([1], 10)  # sent in
    mat_a3_2[:, (11 * m - 10)] = repeat([0], 10)  # self-sending doesn't count
end
mat_a_2 = vcat(mat_a1_2, mat_a3_2)
# Fomulation of Matrix B
mat_b1_2 = zeros(numNodes^2, numNodes^2)
for mn = 1: numNodes^2
    mat_b1_2[mn, mn] = 170
end
mat_b3_2 = zeros(numNodes, numNodes^2)
mat_b_2 = vcat(mat_b1_2, mat_b3_2)
# Fomulation of Vector b
vec_b1_2 = zeros(numNodes^2)
vec_b3_2 = zeros(numNodes)
for n = 1: numNodes
    vec_b3_2[n] = vec_netInject[n] - vec_netEject[n]
end
vec_b_2 = vcat(vec_b1_2, vec_b3_2)
vec_b_2 = hcat(vec_b_2)
#
vec_max_y_2 = repeat([1], numNodes^2)
vec_max_y_2 = hcat(vec_max_y_2)
#
vec_c_2 = zeros(numNodes^2)
for m = 1: numNodes
    for n = 1: numNodes
        vec_c_2[10 * (m - 1) + n] = mat_distance[m, n]
    end
end
vec_c_2 = hcat(vec_c_2)
#
vec_f_2 = zeros(numNodes^2)
for m = 1: numNodes
    for n = 1: numNodes
        vec_f_2[10 * (m - 1) + n] = mat_fixedCost[m, n]
    end
end
vec_f_2 = hcat(vec_f_2)
#
BendersMilp(n_x = numNodes^2,
            n_y = numNodes^2,
            vec_min_y = repeat([0], numNodes^2),
            vec_max_y = vec_max_y_2,
            vec_c = vec_c_2,
            vec_f = vec_f_2,
            vec_b = vec_b_2,
```

```
            mat_a = mat_a_2,
            mat_b = mat_b_2,
            epsilon = 0.0001,
            timesIterationMax = 1000)
# obj - sum(mat_arcTwoNodes .* mat_fixedCost)
```