DTU42136, Large Scale Optimization using Decomposition

Assignment 1 Appendix, OptiGas: Optimize Gas Network using Benders Decomposition

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

March 20th, 2019

1 Appendix

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

```julia
# Benders Algorithm for MILP with Sub and Ray Problems
# Version: 4.0
# Author: Edward J. Xu, edxu96@outlook.com
# Date: March 20th, 2019
module BendersMilp_EDXU
    export BendersMilp
    using JuMP
    using GLPKMathProgInterface
    using PrettyTables
    function BendersMilp(; n_x, n_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])
        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)


        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)
            # println("vec_y: $vec_result_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
        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)
            # println("vec_yBar: $vec_yBar")
            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_ray: $(round(obj_ray, digits = 5)).")
                vec_result_y = getvalue(vec_y)
                println("vec_y: $vec_result_y")
            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)).")
                vec_result_y = getvalue(vec_y)
                println("vec_y: $vec_result_y")
            end
            timesIteration += 1
        end  # ----------------------------------------------------------------------------------------------------
        # @constraint(model_mas, q == obj_sub)  # ???
        # solve(model_mas)  # ???
        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", "type_sub", "obj_sub/ray"],
                         compact; alignment=:l)
        end
        # return (vec_result_x, vec_result_y, dict_obj_mas, dict_obj_sub, dict_obj_ray)
        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 20th, 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_a2_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 = zeros(numNodes^2, numNodes^2)
for mn = 1: numNodes^2
    mat_b2_1[mn, mn] = 1
end
# mat_b2_1 = Diagnal(repeat([1], numNodes^2))
mat_b3_1 = zeros(numNodes, numNodes^2)
mat_b_1 = vcat(mat_b1_1, mat_b2_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_b2_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_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_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_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)
```