IE801B: Logistics Systems Optimization
Assignment #6
2024.05.17.

Lecturer: Prof. Changhyun Kwon
Author: Jaewoo Kim
Student ID: 20245084

1 Introduction

This report presents the implementation of the Lagrangian relaxation method for the location problems proposed by Klincewicz and Luss (1986).

No AI tools were utilized for the development and experimental analysis. However, AI assistance was sought for refining the report's language and expression, specifically employing GPT-4.0 for sentence structure and expression enhancement and Grammarly for grammar checks.

Two Julia scripts are submitted:

- **Functions.jl**: Key functions are defined in this script.
- **main.jl**: This script is the main file performing the experiments by importing functions defined in 'Functions.jl.' It solves the given instances and saves the results.

2 Codes summary

The core functions implementing the algorithm are defined in 'Functions.jl.' It consists of two modules: 'Tools' and 'Solvers.' Solvers module includes the exact solver with HiGHS, and the heuristic solver with supporting functions. 'Klincewicz' function is for the heuristic algorithm.

```julia
function Klincewicz(a, b, c, F, w = 0.25, epsilon = 1e-3, max_solve = 200)

N::Int = length(b)
M::Int = length(a)
x_best::Array{Int} = zeros(Int, N, M)
y_best::Array{Int} = zeros(Int, N)
x_dual::Array{Int} = zeros(Int, N, M)
y_dual::Array{Int} = zeros(Int, N)
final_heuristic_flag::Bool = false
flag0::Bool = true
dual_solve_cnt::Int = 0
U::Array{Int} = collect(1:1:N)
capacity_constraint::Array{Float64} = zeros(Float64, N)
Zl_list::Array{Float64} = []
Zu_list::Array{Float64} = []

## Step 0: initialize
# calculate upper bound
Zu, x_best, y_best = add_heuristic(a, b, c, F, N, M)
# calculate lower bound
lambda = zeros(Float64, N)
Zl, x_dual, y_dual = solve_dual(a, b, c, F, N, M, lambda)
```

```julia
push!(Zl_list, Zl)
push!(Zu_list, Zu)
println("Zl: $Zl, Zu: $Zu")
# if relaxed problem is feasible, stop
if is_feasible(a, b, N, x_dual)
    final_heuristic_flag = true
    flag0 = false
    x_best = x_dual
    y_best = y_dual
    obj = sum(x_best.*c) + sum(y_best.*F)
    return obj, x_dual, y_dual
# if not, update lambda
else
    capacity_constraint = reduce(vcat, sum(a'.*x_dual, dims=2)) .- b
    lambda[U] = ...
end

while flag0
    ## Step 1
    #  solve relaxed problem
    Zl_new, x_dual, y_dual = solve_dual(a, b, c, F, N, M, lambda)
    dual_solve_cnt += 1
    # if necessary, update Zl
    if Zl - Zl_new < -EPS
        Zl = Zl_new
        println("Zl: $Zl, Zu: $Zu")
    end
    push!(Zl_list, Zl)

    ## Step 2: if Step 1 obtains an infeasible solution
    if !is_feasible(a, b, N, x_dual)
        if dual_solve_cnt == max_solve
            break
        end
    else
        # initialize counter
        dual_solve_cnt = 0
        # update Zu if necessary
        Zu_new = sum(x_dual.*c) + sum(y_dual.*F)
        if Zu_new - Zu < -EPS
            Zu = Zu_new
            x_best = x_dual
            y_best = y_dual
            final_heuristic_flag = true
            println("Zl: $Zl, Zu: $Zu")
        ## Step 4: if Step 1 obtains a feasible solution
        # if a better soluton already obtained
        else
            push!(Zu_list, Zu)
            break
        end
        # if the gap is tight enough,
        if Zu/Zl <= 1 + epsilon
            push!(Zu_list, Zu)
            break
        end
        # otherwise,unmark multipliers
        U = collect(1:1:N)
    end
```

```
    push!(Zu_list, Zu)

    ## Step 3
    # constraint violation
    capacity_constraint = reduce(vcat, sum(a'.*x_dual, dims=2)) .- b
    # not marked (U) or constraint violated... multipliers to update
    i_to_update = collect(union(Set(U), Set(findall(x -> x > EPS, capacity_constraint))))
    # update multipliers
    lambda_new = copy(lambda)
    lambda_new[i_to_update] = ...
    lambda = lambda_new
    # mark decreased multipliers
    U = collect(setdiff(Set(U), Set(findall(x -> x < -EPS, capacity_constraint))))
end

## Step 5
if final_heuristic_flag
    x_best = final_heuristic(a, b, c, N, M, x_best, y_best)
    y_best = [maximum(x_best[i, :]) for i in 1:1:N]
end

obj = sum(x_best.*c) + sum(y_best.*F)

return obj, x_best, y_best, Zl_list, Zu_list

end
```

The algorithm is implemented step by step, as the reference paper explains. Note that '*w*,' 'epsilon,' and 'max_solve' are inputs with defaults proposed in the paper. For my experiments, the value of w is tuned to work in my instances.

This function contains other supporting functions: 'add_heuristic,' 'solve_dual,' and 'final_heuristic.' First, the add_heuristic function implements the heuristic, providing the initial guess of the upper bound. It is also written according to the reference paper.

```
function add_heuristic(a, b, c, F, N, M)

facilities = Set(1:1:N)
K = Set([])
Kc = Set(facilities)
Zu = Inf
x_best::Array{Int} = zeros(Int, N, M)
y_best::Array{Int} = zeros(Int, N)
x_curr::Array{Int} = zeros(Int, N, M)
y_curr::Array{Int} = zeros(Int, N)

while true
    ## Step 1
    w = zeros(Float64, N, M)
    R = fill(-Inf, N)
    for i in Kc
        for j in 1:1:M
            w[i, j] = max(minimum(c[collect(K), j].- c[i, j], init=0), 0)
        end
        Omega = sum(w[i, :])
        R[i] = Omega * min(b[i]/sum(a[w[i, :].>0], init=0), 1) - F[i]
    end
```

```julia

        ## Step 2
        i_add = argmax(R)
        push!(K, i_add)
        Kc = setdiff(facilities, K)
        K_list = collect(K)

        ## Step 3
        if (sum(b[K_list]) - sum(a)) < EPS
            continue
        end

        ## Step 4
        if length(K_list) == 1
            order = sortperm([minimum(c[K_list, j]) for j in 1:1:M], rev=true)
        else
            cost_first_second = [sort(c[K_list, j])[1:2] for j in 1:1:M]
            cost_diff = [cost[2] for cost in cost_first_second] .- [cost[1] for cost in cost_first_second]
            order = sortperm(cost_diff, rev=true)
        end

        ## Step 5
        remaining_capacity = b[K_list]
        TC = 0.0
        flag = 0
        x_curr = zeros(Int, N, M)
        for j in order
            tpv = sortperm(c[K_list, j])
            for i in tpv
                if a[j] - remaining_capacity[i] < EPS
                    x_curr[K_list[i], j] = 1
                    remaining_capacity[i] -= a[j]
                    flag = 1
                    break
                end
            end

            if flag == 0
                flag = 2
                break
            end

            flag = 0

        end

        if flag == 2
            continue
        end

        ## Step 6
        y_curr = [maximum(x_curr[i, :]) for i in 1:1:N]
        TC = sum(c.*x_curr) + sum(F.*y_curr)
        if TC - Zu < -EPS
            x_best = x_curr
            y_best = y_curr
            Zu = TC
        else
            break
```

```
    end

    K_ban = K_list[b[K_list] .- remaining_capacity .< EPS]

    if !isempty(K_ban)
        setdiff!(facilities, Set(K_ban))
    end
end

return Zu, x_best, y_best

end
```

solve_dual function solves the Lagrangian relaxed problems that are subproblems for the algorithm. Basically, this function provides a new lower bound. However, when the resultant solution is feasible, that is, satisfies the capacity constraint, then gives an upper bound and terminates the algorithm.

```
function solve_dual(a, b, c, F, N, M, lambda)

model = JuMP.Model(HiGHS.Optimizer)

model = Model(HiGHS.Optimizer)
set_silent(model)

@variable(model, x[i = 1:N, j=1:M], Bin)
@variable(model, y[i = 1:N], Bin)

@objective(model, Min, sum(F.*y) + sum((c .+ lambda .* a') .* x) - sum(lambda.*b))

@constraint(model, single_source[j=1:M], sum(x[:, j]) == 1)
@constraint(model, open_facility[i=1:N, j=1:M], x[i, j] <= y[i])

optimize!(model)

return objective_value(model), round.(Int, value.(x)), round.(Int, value.(y))

end
```

final_heuristic function is for refining the final solution if there has been any update on the upper bound. This algorithm is much simpler than the initial heuristic since its purpose is to skim the solution roughly and improve if possible.

```
function final_heuristic(a, b, c, N, M, x, y)

K = findall(x -> x > EPS, y)
x_mod::Array{Int} = copy(x)

flag = true
while flag
    flag = false

    cost_first_second = [sort(c[K, j])[1:2] for j in 1:1:M]
    cost_diff = [cost[2] for cost in cost_first_second] .- [cost[1] for cost in cost_first_second]
```

```
    remaining_capacity = b[K] .- [sum(a.*x[i, :]) for i in K]
    order = sortperm(cost_diff, rev=true)
    for j in order
        can_move = K[findall(x -> x < EPS, a[j] .- remaining_capacity)]
        min_move = can_move[argmin(c[can_move, j])]
        if c[min_move, j] - sum(c[:, j] .* x[:, j]) < -EPS
            x_mod[:, j] = zeros(Int, N)
            x_mod[min_move, j] = 1
            flag = true
        end
    end
end

return x_mod

end
```

3 Experiment results

The experiments were performed on the personal computer in my lab. The specifications of the machine, language, and used packages are in Table 1. The specifics of the two instances are saved in the file name 'instances.jld.'

Table 1: Experimental Environment

Resource	Specification
CPU	13th Gen Intel(R) Core(TM) i9-13900K
RAM	128GB
OS	Windows 11
Language	Julia 1.10.3
Used packages	DelimitedFiles (v1.9.1) HiGHS (v1.9.0) JuMP (v1.22.0) JLD (v0.13.5) Plots (v1.40.4) StatsBase (v0.34.3)

3.1 Instance 1

The first instance is generated by the data in the assignment guide. Out of 49 locations, 10 facilities are randomly selected, with the remaining locations designated as customers. The arc lengths between facilities and customers are calculated as per the guide. Additionally, the values for a, b, and F are taken from the fifth, fourth, and sixth columns, respectively, and divided by 100, 100, and 1000. The locations are visualized in Fig. 1, where red dots and blue dots represent the facilities and the customers, respectively.

The solutions from the exact solver and the heuristic solver are illustrated in Fig. 2 and Fig. 3, respectively. The

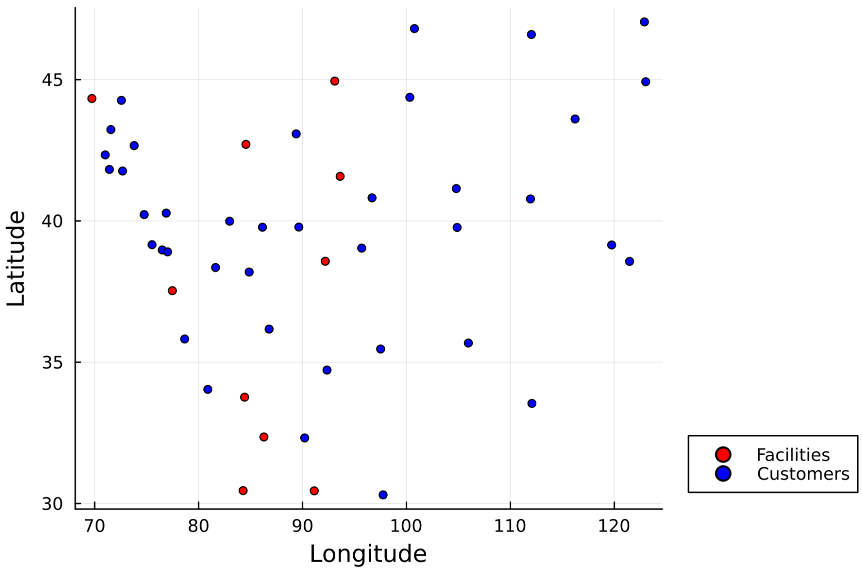

Figure 1: Visualization of Instance 1

heuristic solution is obtained by setting w to be 10^{-4} and other inputs to default. In these figures, yellow stars represent the open facilities, while dark dashes indicate the allocation of customers to these open facilities. One of the two open facilities is the same, and the others are nearby.

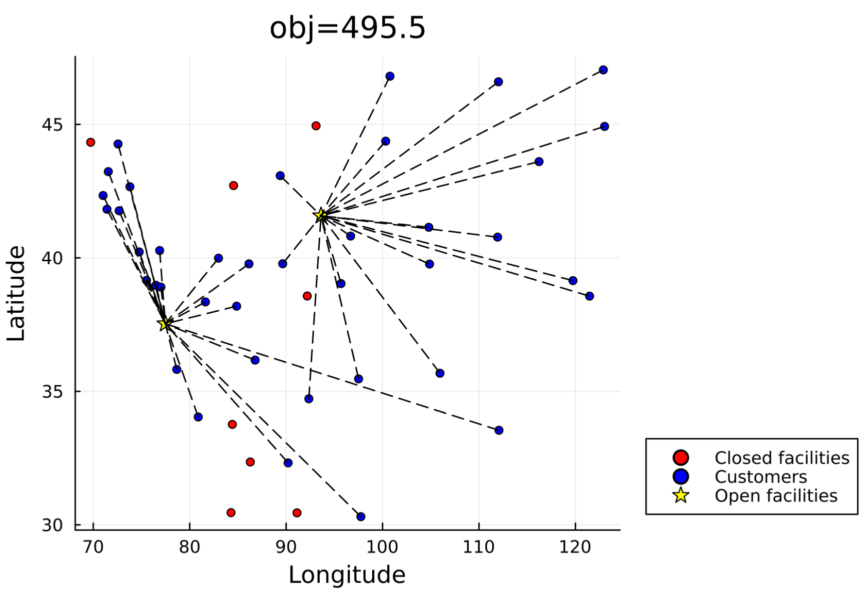

Figure 2: Exact Solution of Instance 1

Fig. 4 illustrates the progression of the lower and upper bounds of the heuristic algorithm. The algorithm attempts to update the lower bound with each iteration. It terminates when the upper bound is updated or if there is no improvement in the lower bound for a specified period. Consequently, the upper bound is updated only once, which seems not that interesting. However, the initial guess and the final solution are seemingly not that bad.

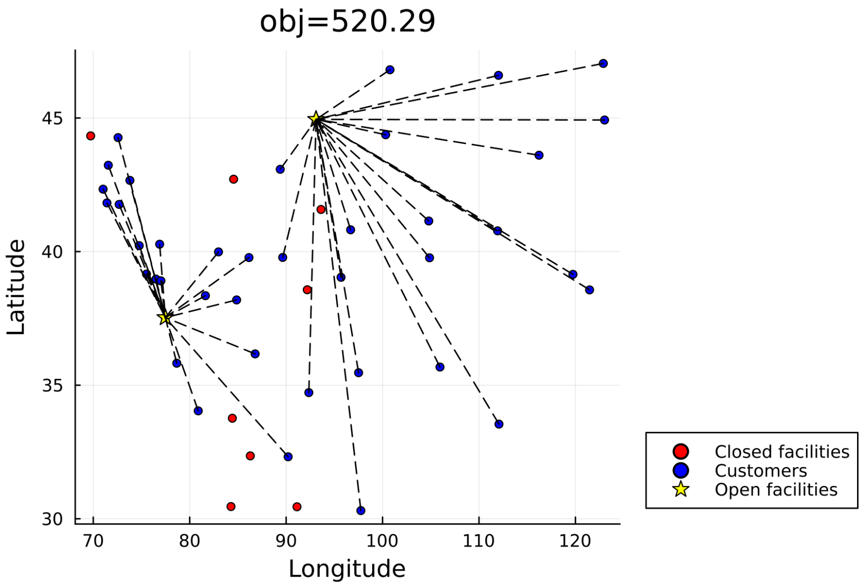

Figure 3: Heuristic Solution of Instance 1

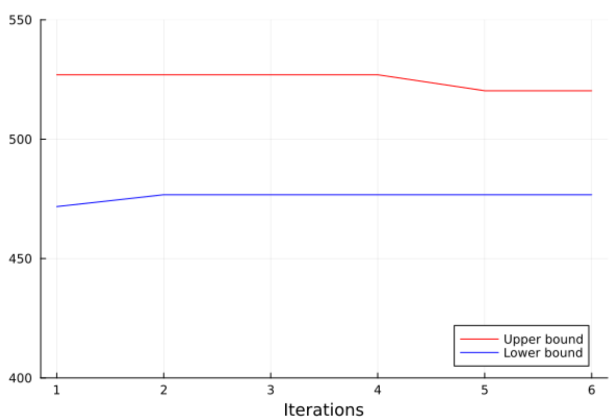

Figure 4: # Iterations vs. Bounds Plot of Instance 1

3.2 Instance 2

Instance 2 is randomly generated, with the locations of 75 facilities and 25 customers distributed within a 1 x 1 box. The arc lengths are calculated as ten times the Euclidean distance. The coefficients a, b, and F are randomly generated within the possible values specified in Table. 2. Fig. 5 provides the visualization. This instance has more facilities with lower capacities than instance 1. Consequently, finding an optimal combination and allocating each customer to a facility is more challenging, which is expected to result in a more interesting solution (from my intuition).

Similar to instance 1, Figs. 6 and 7 represent the solutions obtained from the exact solver and the heuristic solver, respectively. In this case, w is also set to 10^{-4} for the heuristic solver. Each solution uses a different number of facilities, but they share one facility in common. Additionally, the gap between the solutions appears to be greater than in instance 1.

Table 2: Possible Values of Coefficients

Coefficient	Possible Values
a	$\{1,2,\ldots,10\}$
b	$\{20,30,40,50\}$
F	$\{50,100,150\}$

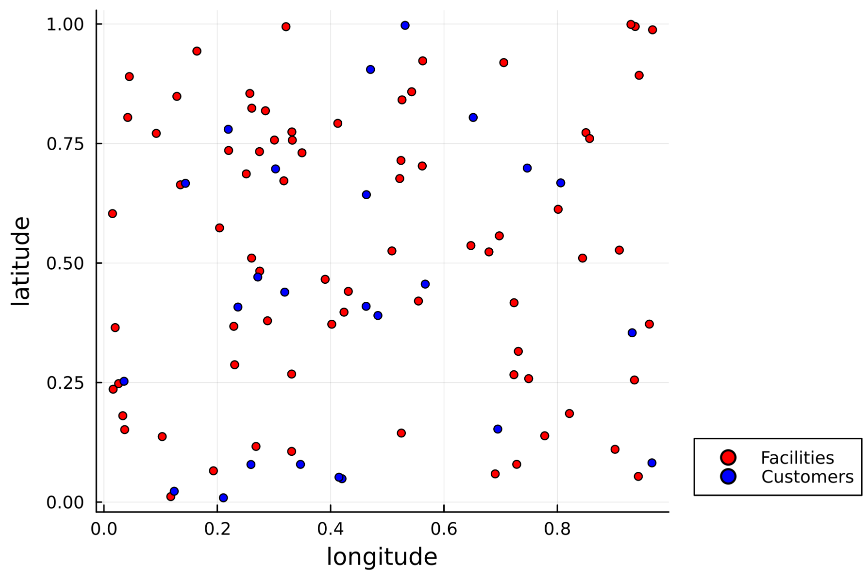

Figure 5: Visualization of Instance 2

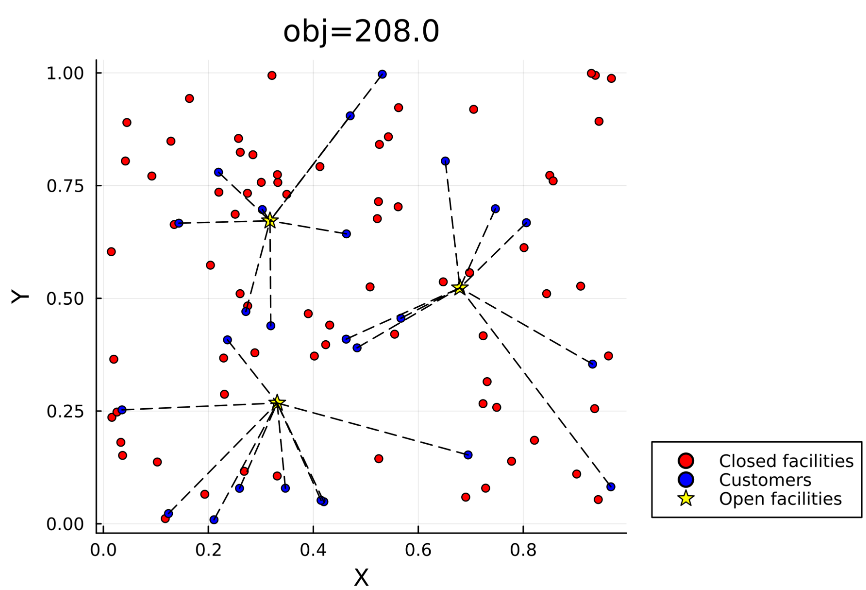

Figure 6: Exact Solution of Instance 2

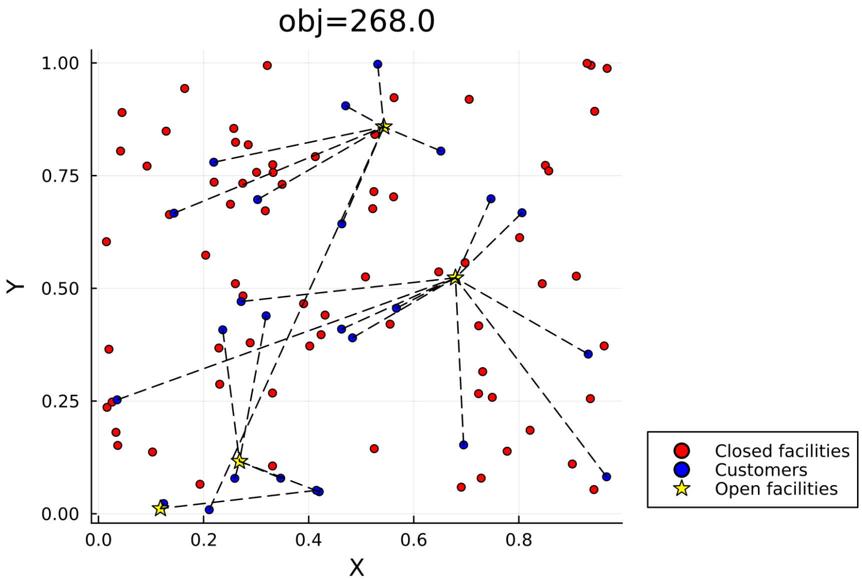

Figure 7: Heuristic Solution of Instance 2

Fig. 8 indicates that the upper bound was not updated until the algorithm terminated. As a result, the final solution is the same as the initial guess obtained from the add heuristic.

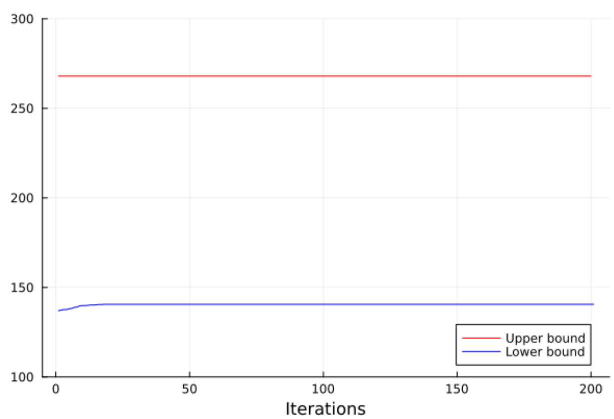

Figure 8: # Iterations vs. Bounds Plot of Instance 2

References

Klincewicz, J. G. and Luss, H. (1986). A Lagrangian relaxation heuristic for capacitated facility location with single-source constraints. *Journal of the Operational Research Society*, 37(5):495500.