1 Introduction

This report details the development of the two solution methodologies for the Constrained Shortest Path Problem (CSPP). The first methodology is based on a link/node-based Integer Programming (IP) formulation, employing a general IP solver such as Gurobi or HiGHS. The second methodology is the Branch-and-Price algorithm, which is emphasized more.

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:

- **CSPP.jl**: Key functions are defined in this script.
- **main.jl**: This script is the main file performing the experiment by importing functions defined in 'CSPP.jl.' It loads instances, solves them, and saves the results. The results are saved in the 'results' folder as .csv files.

The remainder of this report is organized into three sections. Section 2 introduces the Julia scripts and the methodologies they implement. Section 3 describes the results obtained by the developed codes.

2 Methodologies & codes summary

2.1 Link/node formulation with general IP solver

The Link/node formulation of CSPP is as follows:

$$\text{minimize} \quad \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1}$$

$$\text{subject to} \quad \sum_{j:(1,j)\in A} x_{1j} = 1, \tag{2}$$

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ij} = 0, \quad i \in N \setminus \{1,d\} \tag{3}$$

$$\sum_{j:(j,d)\in A} x_{jd} = 1, \tag{4}$$

$$\sum_{(i,j)\in A} c_{ij}x_{ij} \le T_{\max}, \tag{5}$$

$$x_{ij} \in \{0,1\}, \quad (i,j) \in A \tag{6}$$

where N is the node index set, and A is the arc index set. The decision variable, x_{ij}, indicates whether an arc is used or not. By writing a code throwing the whole formulation to a general IP solver, we can solve CSPP. It is implemented as the function named 'IPSolver' as follows:

```
function IPSolver(data, Tmax, origin, destination)

start_node = Int.(data[:, 1])
end_node = Int.(data[:, 2])
arc_cost = data[:, 3]
arc_time = data[:, 4]
n_nodes = maximum(vcat(start_node, end_node))
n_arcs = length(start_node)

model = Model(HiGHS.Optimizer)

@variable(model, x[i = 1:n_arcs], Bin)

@objective(model, Min, sum(arc_cost .* x))

@constraint(model, flow_origin, sum(x[start_node .== origin]) - sum(x[end_node .== origin]) == 1)
@constraint(model, flow_balance[i = 1:n_nodes; (i != origin) && (i != destination)], sum(x[start_node .== i]) - sum(x[end_node .== i]) == 0)
@constraint(model, flow_destination, sum(x[start_node .== destination]) - sum(x[end_node .== destination]) == -1)

@constraint(model, time, sum(arc_time .* x) <= Tmax)

optimize!(model)


if primal_status(model) == FEASIBLE_POINT
    cost_opt, x_opt = objective_value(model), value.(x)
else
    println("Infeasible instance!")
    cost_opt, x_opt = nothing, nothing
end

path_opt = Set(findall(i->x_opt[i] > 1-EPS, 1:1:n_arcs))

return cost_opt, path_opt

end
```

The inputs are 'data,' 'Tmax,' 'origin,' and 'destination.' The data argument contains the information of the network as $|N| \times 4$ matrix. Each row represents an arc, and each column represents the tail node index, the head node index, associated cost and time, respectively. The optimal cost, 'cost_opt,' and the corresponding path, 'path_opt' are provided as outputs.

2.2 Branch-and-Price

Implementation of Branch-and-Price, which is our main purpose, is written in the function named 'MyCGSolver.'

```
function MyCGSolver(data, Tmax, origin, destination)

start_node = Int.(data[:, 1])
end_node = Int.(data[:, 2])
arc_cost = data[:, 3]
arc_time = data[:, 4]
n_nodes = maximum(vcat(start_node, end_node))
n_arcs = length(start_node)

## initialize
# containers for algorithms
pred_node = [0]
```

```julia
zero_arcs_node::Vector{Vector{Int}} = [[]]
one_arcs_node::Vector{Vector{Int}} = [[]]
state_node = [0] # 0: unsolved and not pruned out, 1: integer, 2: fractional, 3: infeasible, 4: pruned out
obj_val_node = [0.0]
path_ind_node = [Int[]]
costs_node = [Number[]]
times_node = [Number[]]
paths_node = [Set{Int}[]]

curr_node_ind = 1
best_node_ind = 0
best_path = Int[]
best_obj_val = Inf

# get an initial feasible solution
g = SimpleWeightedDiGraph(n_nodes)
for i in 1:1:n_arcs
    add_edge!(g, start_node[i], end_node[i], arc_cost[i])
end
bf_state = bellman_ford_shortest_paths(g, origin)
new_path = enumerate_paths(bf_state)[destination]
path_arcs = [findfirst(x->(start_node[x]==new_path[i]) && end_node[x]==new_path[i+1], 1:1:n_arcs) for i in 1:1:(length(new_path)-1)]
tpv = sum(data[path_arcs, 3:4], dims=1)
cost, time = tpv[1], tpv[2]

push!(path_ind_node[curr_node_ind], length(path_ind_node[curr_node_ind])+1)
push!(costs_node[curr_node_ind], cost)
push!(times_node[curr_node_ind], time)
push!(paths_node[curr_node_ind], Set(path_arcs))

# algorithm starts
while true
    # solve node by column generation
    lam_val, obj_val, y0_val, feasible = solve_node!(...)

    # update the current node status & best node
    if y0_val > EPS     # 3: infeasible
        state_node[curr_node_ind] = 3
        obj_val_node[curr_node_ind] = Inf
    elseif sum(lam_val .> 1-EPS) == 1     # 1: integer
        state_node[curr_node_ind] = 1
        obj_val_node[curr_node_ind] = obj_val
        if obj_val < best_obj_val # best solution update
            best_node_ind = curr_node_ind
            tpv = findfirst(i->lam_val[i]>1-EPS, path_ind_node[best_node_ind])
            best_path = paths_node[best_node_ind][tpv]
            best_obj_val = obj_val
        end
    elseif sum(lam_val .> 1-EPS) != 1     # fractional
        if best_obj_val - obj_val > EPS  # 2: not pruned out
            state_node[curr_node_ind] = 2
            obj_val_node[curr_node_ind] = obj_val
            ## branch, strategy: the first fractional arc
            if feasible
                # find the first fractional arc
                ...





                # branch
                ...









            end
        else    # 4: pruned out
            state_node[curr_node_ind] = 4
            obj_val_node[curr_node_ind] = obj_val
        end
    end

    # next node to solve
    curr_node_ind = 0
```

```
290     for bInd in eachindex(state_node)
291         if state_node[bInd] == 0
292             curr_node_ind = bInd
293             break
294         end
295     end
296 
297     if curr_node_ind == 0     # if no node to solve, terminate the algorithm
298         break
299     end
300 end
301 
302 # solution history
303 solution_history = (pred_node, one_arcs_node, zero_arcs_node, state_node, obj_val_node)
304 best_solution = (best_node_ind, one_arcs_node[best_node_ind], zero_arcs_node[best_node_ind], best_obj_val, best_path)
305 
306 return solution_history, best_solution
307 
308 end
```

The inputs are the same as the IPSolver function. The overall process consists of 3 parts: 1) initialization (lines 206 to 237), 2) solve node (line 242), and 3) branch (lines 244 to 299).

In the initialization phase, the containers for the algorithm are defined, and an initial path to be included in the Restricted Master Problem (RMP) is obtained to start the algorithm. The containers are as follows:

- pred_node: the predecessor node index
- zero_arcs_node: the arc index that is constrained to be 0, by branching
- one_arcs_node: the arc index that is constrained to be 1, by branching
- state_node: state of nodes (0 - unsolved and not pruned out, 1 - integer solution, 2 - fractional solution, 3 - infeasible, 4 - pruned out)
- obj_val_node: the obtained optimal solution of each node
- path_ind_node: paths index inherited from the predecessor node and obtained from solving pricing sub-problems
- costs_node: costs of paths in path_ind_node
- times_node: times of paths in path_ind_node
- paths_node: selected arcs set of paths in path_ind_node

As the Branch-and-Price algorithm progresses, the containers are updated to keep the information of each node. The initial path is obtained using the Bellman-Ford algorithm, which does not guarantee the feasibility of the path in the CSPP; however, starting from this solution is acceptable.

The node solution procedure is written as the supporting function named ‘solve_node!’

```
function solve_node!(...)

## select paths satisfying the node conditions
path_ind_rmp = Int[]

if isempty(one_arcs)
    path_ind_rmp = copy(path_ind)
else
```

```julia
    for pInd in path_ind, arc in paths[pInd]
        if (arc in one_arcs) && !(pInd in path_ind_rmp)
            push!(path_ind_rmp, pInd)
        end
    end
end

# delete zero arcs
if !isempty(zero_arcs)
    for pInd in path_ind_rmp, arc in paths[pInd]
        if (arc in zero_arcs) && (pInd in path_ind_rmp)
            deleteat!(path_ind_rmp, findfirst(x->x==pInd, path_ind_rmp))
        end
    end
end

M = 100.0       # Big-M
while true
    # restricted master problem
    rmp = Model(HiGHS.Optimizer)

    @variable(rmp, y0 >= 0)                                       # artificial variable facilitating the solution procedure
    @variable(rmp, lam[path_ind_rmp] >= 0)                              # path variables
    @objective(rmp, Min, M*y0 + sum(costs[path_ind_rmp] .* lam))            # cost minimization
    @constraint(rmp, resource, sum(times[path_ind_rmp] .* lam) <= Tmax)       # resource constraint, 1
    @constraint(rmp, convexity, y0 + sum(lam) == 1)             # convexity constraint, 0
    optimize!(rmp)                                                # solve the relaxaed RMP

    # results
    y0_val = value(y0)
    lam_val = value.(lam)
    obj_val = objective_value(rmp)
    pi0 = dual(convexity)
    pi1 = dual(resource)

    # if infeasible
    if y0_val > EPS
        pi0 = M
    end

    # pricing sub-problem
    psp = Model(HiGHS.Optimizer)
    @variable(psp, x[i = 1:n_arcs], Bin)
    @objective(psp, Min, sum((arc_cost.-arc_time.*pi1) .* x))
    @constraint(psp, flow_origin, sum(x[start_node .== origin]) - sum(x[end_node .== origin]) == 1)
    @constraint(psp, flow_balance[i = 1:n_nodes; (i != origin) && (i != destination)], sum(x[start_node .== i]) - sum(x[end_node .== i]) == 0)
    @constraint(psp, flow_destination, sum(x[start_node .== destination]) - sum(x[end_node .== destination]) == -1)
    if !isempty(one_arcs)
        @constraint(psp, node_ones[i = one_arcs], x[i] == 1)
    end
    if !isempty(zero_arcs)
        @constraint(psp, node_zeros[i = zero_arcs], x[i] == 0)
    end
    @constraint(psp, time, sum(arc_time .* x) <= Tmax)
    optimize!(psp)

    # no feasible path within node
    if primal_status(psp) != FEASIBLE_POINT
        lam_val_full = zeros(Float64, length(path_ind))
        lam_val_full[path_ind_rmp] = lam_val.data
        return lam_val_full, obj_val, y0_val, false
    end

    reduced_cost = objective_value(psp) - pi0

    if y0_val > EPS     # RMP is infeasible
        x_opt = value.(x)
        new_path = Set(findall(i->x_opt[i]>(1-EPS), collect(1:1:n_arcs)))
        if new_path in paths[path_ind_rmp]    # no column to generate
            lam_val_full = zeros(Float64, length(path_ind))
            lam_val_full[path_ind_rmp] = lam_val.data
            return lam_val_full, obj_val, y0_val, true
        elseif reduced_cost < -EPS
            x_opt = value.(x)
            new_path = Set(findall(i->x_opt[i]>(1-EPS), collect(1:1:n_arcs)))
            new_cost = sum(arc_cost[collect(new_path)])
            new_time = sum(arc_time[collect(new_path)])
            # add column to the RMP
            push!(path_ind, length(path_ind)+1)
            push!(costs, new_cost)
            push!(times, new_time)
```

```
161             push!(paths, new_path)
162             push!(path_ind_rmp, length(path_ind))
163         else
164             M *= 10.0
165         end
166     else    # RMP is feasible
167         if reduced_cost < -EPS      # find a new column
168             x_opt = value.(x)
169             new_path = Set(findall(i->x_opt[i]>(1-EPS), collect(1:1:n_arcs)))
170             new_cost = sum(arc_cost[collect(new_path)])
171             new_time = sum(arc_time[collect(new_path)])
172             # add column to the RMP
173             push!(path_ind, length(path_ind)+1)
174             push!(costs, new_cost)
175             push!(times, new_time)
176             push!(paths, new_path)
177             push!(path_ind_rmp, length(path_ind))
178         else        # no column to generate
179             lam_val_full = zeros(Float64, length(path_ind))
180             lam_val_full[path_ind_rmp] = lam_val.data
181             return lam_val_full, obj_val, y0_val, true
182         end
183     end
184 end
185
186 end
```

The inputs of this function are omitted in this report due to its extensive length of codes. In this function, the RMP and pricing sub-problem are solved iteratively. To keep generating columns even if the RMP is infeasible with the current columns, an artificial decision variable 'y0' is introduced in the RMP.

This process has two terminal conditions: First, it terminates if no feasible path satisfies the branching constraints. Specifically, if branching mandates that certain x_{ij} should be kept 0 or 1, and no path satisfies all those conditions, then the node solution process terminates. Second, the process also terminates if there is no column with a negative reduced cost.

Finally, after solving a node, the branch process is initiated. Each node is labeled following the results obtained by the solution process. If the solution is infeasible, integer, or dominated by the current best solution, the node does not branch. However, if the solution is fractional and not dominated by the best solution, the node does branch. To branch out, the first fractional arc in the data that has not been selected to branch is chosen. Two branches are generated, whose selected arc values are 0 and 1, respectively. The details of the codes (lines 263 to 280) can be found in the Julia script.

After the branching process is over, the algorithm finds the next node to solve. If no node remains unsolved, the whole algorithm terminates.

3 Results: test problems

Two given problems in the assignment guide were solved using the IPSolver and the MyCGSolver functions. Both solvers provided the same optimal solution for each problem. Since the Branch-and-Price algorithm is our main target, this section will show its results.

The experiments were performed on my personal computer located in my laboratory, N7-2 3332. The specifications of the computer, language, and used packages are in Table 1.

Table 1: Experimental Environment

Resource	Specification
CPU	13th Gen Intel(R) Core(TM) i9-13900K
RAM	128GB
OS	Windows 11
Language	Julia 1.10.0
Used packages	Graphs (v1.9.0) HiGHS (v1.9.0) JuMP (v1.20.0) JLD (v0.13.5) SimpleWeightedGraphs (v1.4.0)

The results of the first problem are provided as Figure 1. The root node (Node 1) yielded a fractional solution. Following the branching strategy outlined in the previous section, the first fractional arc, $(1,2)$, was chosen for branching. Both Nodes 2 and 3 resulted in integer solutions, eliminating the need for further branching. Node 3 presented the optimal solution with a cost of 13 and the path 1->3->2->4->6

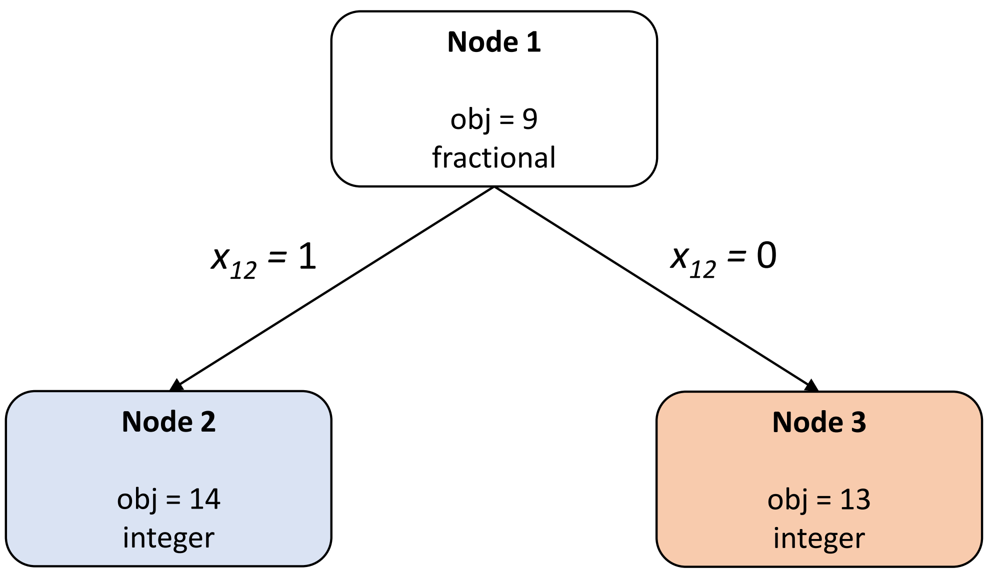

Figure 1: Branch-and-Price Results of Problem 1

The results of the second problem are provided as Figure 2. It gives a more interesting scenario than the first problem. Given that the root node yielded a fractional solution, the first fractional arc $(1,2)$ was chosen to branch. Node 2 resulted in another fractional solution, leading to the creation of Nodes 4 and 5 by respectively setting the decision variable for arc $(2,5)$ to 1 and 0.Subsequently, Node 3 achieved an integer solution. Although Node 4 also produced a fractional solution, it did not undergo further branching as it was dominated by Node 3. Similarly, Node 5 yielded an integer solution and hence did not branch further. Finally, Node 3 emerged as providing the optimal solution, with a cost of 14 and a path of 1->3->6->10.

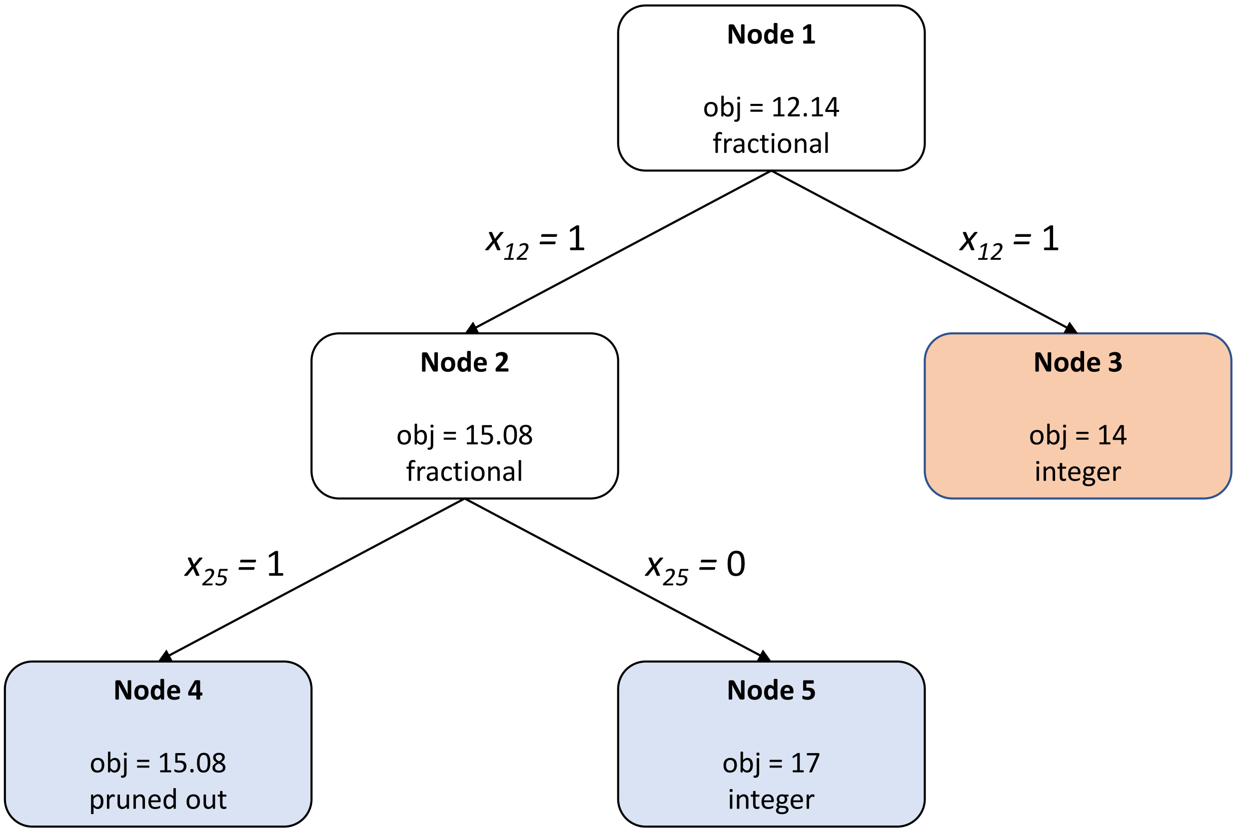

Figure 2: Branch-and-Price Results of Problem 2

4 Conclusions

To summarize,

- Julia functions for solving CSPP were developed in two ways: 1) link/node-based formulation with a general IP solver, 2) Branch-and-Price algorithm
- The solution procedure of Branch-and-Price algorithm was visualized

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Pearson.

Desrosiers, J. and L ubbecke, M. E. (2005). *A primer in column generation. In Column generation, pages 132*. Springer.

Handler, G. Y. and Zang, I. (1980). *A dual algorithm for the constrained shortest path problem. Networks*, 10(4):293309.