IE801B: Logistics Systems Optimization **Lecturer:** Prof. Changhyun Kwon
Assignment #2 **Author:** Jaewoo Kim
2024.03.22. **Student ID:** 20245084

1 Introduction

This report presents a comparative analysis of three servers for the vanilla Traveling Salesman Problem (TSP). This report focuses on introducing two primary tasks undertaken for this purpose:

- The development of Julia functions implementing 1) all-at-once solver using a complete formulation containing all the subtour elimination constraints, 2) lazy constraints solver.
- The execution of experiments on different instances with the two developed approaches and Concorde solver.

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:

- **TSP.jl**: Key functions are defined in this script.
- **main.jl**: This script is the main file performing the experiment by importing functions defined in 'TSP.jl.' It generates instances, solves them, and saves the results. The generated instances are saved in the 'instances' folder as .csv files, and the results are saved in the 'results' folder as .csv files. In addition, the visuals are generated as .png and .gif files in the same directory with 'main.jl'

The remainder of this report is organized into four sections. Section 2 offers an introduction to the Julia scripts and the methodologies they implement. Section 3 delves into the experimental design, and presents the results obtained from applying the developed solvers as well as the Concorde solver, facilitating a direct comparison of their performance and efficiency. Section 4 wraps up the report with a conclusion that synthesizes the key findings from the comparative analysis.

2 Methodologies & codes summary

2.1 All-at-once approach

The integer programming formulation for TSP integrating all the subtour eliminations is as follows:

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

$$\text{subject to} \quad \sum_{i \in N} x_{ij} = 1, \quad j \in N \tag{2}$$

$$\sum_{j \in N} x_{ij} = 1, \quad i \in N \tag{3}$$

$$\sum_{i \in N} x_{ii} = 0, \quad i \in N \tag{4}$$

$$t_j \geq t_i + |N| \cdot x_{ij} + |N| - 1, \quad i, j \in N \backslash \{1\} \tag{5}$$

$$x_{ij} \in \{0, 1\}, \quad i, j \in N \tag{6}$$

$$t_i \geq 0, \quad i \in N \backslash \{1\} \tag{7}$$

where N is the index set for sites. Here, the subtour elimination constraints are represented as (5). Such a formulation is called Miller-Tucker-Zemlin (MTZ) formulation. The 'GurobiSolver' function with the above formulation and Gurobi solver is in 'TSP.jl' Julia script as follows:

```
83  function GurobiSolver(C)
84      sitesN = size(C)[1]
85      x_opt = Matrix{Bool}(undef, sitesN, sitesN)
86
87      model = JuMP.Model(Gurobi.Optimizer)
88      set_attribute(model, "OutputFlag", 0)
89
90      @variables(model, begin
91          x[1:1:sitesN, 1:1:sitesN], Bin
92          t[i=2:1:sitesN] >= 0
93      end)
94
95      @constraints(model, begin
96          outflow[i=1:1:sitesN], sum(x[i, :]) == 1
97          inflow[j=1:1:sitesN], sum(x[:, j]) == 1
98          selfloop[i=1:1:sitesN], sum(x[i, i]) == 0
99          subtour[(i, j) = [(i, j) for i in 2:1:sitesN for j in 2:1:sitesN if i!=j]], t[j] - t[i] - sitesN*x[i, j] +
100     end
101     )
102
103     @objective(model, Min, sum(C .* x))
104
105     optimize!(model)
106
107     x_opt = value.([x[i, j] for i in 1:1:sitesN, j in 1:1:sitesN])
108     t_opt = value.([t[i] for i in 2:1:sitesN])
109     opt_tour = sortperm(t_opt).+1; pushfirst!(opt_tour, 1)
110     opt_cost = objective_value(model)
111
112     comp_time = solve_time(model)
113
114     return opt_tour, opt_cost, comp_time
115 end
```

The only input is the matrix C, of which (i, j) component represents the distance between sites i and j. Line 90 to

103 constructs the MTZ formulation. The outputs are optimal tour, its associated cost, and the solution time.

An alternative method explored is the use of lazy constraints. This approach initially omits subtour elimination constraints (equation (5)), building the formulation based on equations (1), (2), (3), (4), (6). Instead, adding subtour elimination constraint (8), which appears in the Dantzig-Fulkerson-Johnson (DFJ) formulation, whenever meet a subtour consisting of sites index set $S \subset N$.

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1, \quad i, j \in S \tag{8}$$

The implementation of the lazy constraints approach is encapsulated within the 'GurobiSolverWithLazyConstraints' function is in 'TSP.jl' Julia script as follows:

```
function GurobiSolverWithLazyConstraints(C)
    sitesN = size(C)[1]

    model = JuMP.Model(Gurobi.Optimizer)
    set_attribute(model, "OutputFlag", 0)

    @variable(model, x[1:1:sitesN, 1:1:sitesN], Bin)

    @constraints(model, begin
        outflow[i=1:1:sitesN], sum(x[i, :]) == 1
        inflow[j=1:1:sitesN], sum(x[:, j]) == 1
        selfloop[i=1:1:sitesN], sum(x[i, i]) == 0
    end
    )

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

    function find_subtour(x_val, s=1)
        subtour = [s]
        while true
            _, next_site = findmax(x_val[subtour[end], :])
            if next_site == s
                break
            else
                push!(subtour, next_site)
            end
        end
        return subtour
    end

    function call_back_function(cb_data)
        status = callback_node_status(cb_data, model)

        if status != MOI.CALLBACK_NODE_STATUS_INTEGER
            return
        end

        x_val = (y->callback_value(cb_data, y)).(x).data

        subtour = find_subtour(x_val)
```

```
157             S = length(subtour)
158             if S > sitesN-1
159                 return
160             end
161
162             ex = AffExpr()
163             for i in subtour, j in subtour
164                 add_to_expression!(ex, 1, x[i, j])
165             end
166
167             con = @build_constraint(ex <= S-1)
168             MOI.submit(model, MOI.LazyConstraint(cb_data), con)
169
170             return
171         end
172
173         MOI.set(model, MOI.LazyConstraintCallback(), call_back_function)
174
175         optimize!(model)
176
177         x_opt = value.([x[i, j] for i in 1:1:sitesN, j in 1:1:sitesN])
178         opt_tour = find_subtour(x_opt)
179         opt_cost = objective_value(model)
180         comp_time = solve_time(model)
181
182         return opt_tour, opt_cost, comp_time
183     end
```

Lines 147 to 173, the callback function is defined and integrated into the solver. Remark that in this specific implementation, subtour elimination constraints exclusively for the subtour that include site 1 are added, each time a subtour is identified.

The inputs and outputs structure are same with that of the 'GurobiSolver' introduced earlier.

3 Experiments design and results

Two distinct experiments were designed to evaluate the performance of the three algorithms discussed. Experiment 1 was designed to analyze the performance trends of each algorithm as the problem size escalates. It involves applying all three algorithms to a series of randomly generated instances, with the number of sites ranging from 5 to 40. For each site number, one instance was generated and tested. In addition, Experiment 2 focuses on the comparison between the all-at-once approach and the lazy constraints approach. It involves testing 100 instances that all share the same problem size.

For both experiments, the problem instances were constructed based on sites located within a 1 x 1 area, with the x and y coordinates of each site generated uniformly at random.

The experiments were performed in 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
28 Used packages	Random, Base, Statistics, LinearAlgebra StatsBase (v0.33.21) DataStructures (v0.18.18) DelimitedFiles (v1.9.1) ProgressBars (v1.5.1) Plots (v1.40.2) JuMP (v1.20.0) Gurobi (v1.2.2) Concorde (v0.1.3)

3.1 Experiment 1: increasing the number of sites

Experiment 1 increases the number of sites from 5 to 40, applying each of the three approaches to solve the problem at every increment. The results are given in Table 2, and its graphical representation is in Figure 1.

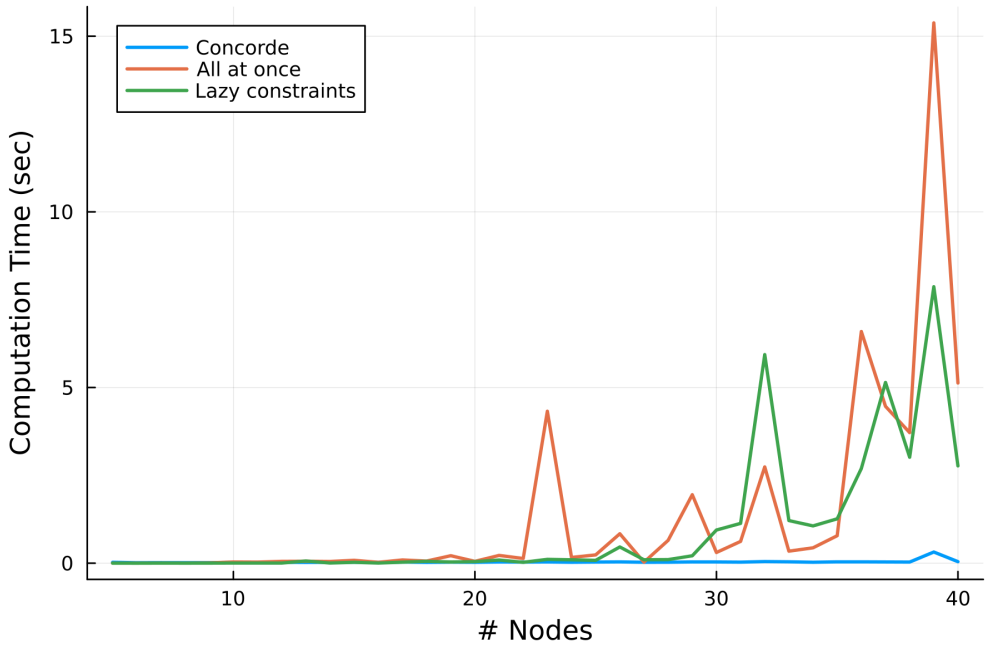

Figure 1: Experiment 1 Results Plot

While the relative performance of the solvers varies with each problem instance, there are notable general observations from the experiments. Despite increasing problem size, the Concorde solver shows slight increases in time

Table 2: Time Consumption of Each Solver in Experiment 1

	Time Consumption (sec)		
# of Sites, $\|N\|$	Concorde	All-at-Once	Lazy Constraints
5	0.026	0.003	0.001
6	0.014	0.003	< 0.001
7	0.016	0.005	0.002
8	0.015	0.003	0.001
9	0.015	0.01	0.002
10	0.023	0.034	0.004
11	0.022	0.031	0.004
12	0.026	0.052	0.004
13	0.022	0.055	0.062
14	0.025	0.049	0.005
15	0.029	0.082	0.025
16	0.023	0.025	0.004
17	0.032	0.092	0.029
18	0.023	0.06	0.054
19	0.029	0.213	0.034
20	0.025	0.052	0.044
21	0.034	0.221	0.09
22	0.031	0.134	0.025
23	0.035	4.325	0.109
24	0.026	0.164	0.096
25	0.032	0.236	0.081
26	0.037	0.838	0.462
27	0.026	0.028	0.097
28	0.027	0.65	0.102
29	0.036	1.949	0.211
30	0.035	0.304	0.944
31	0.03	0.62	1.133
32	0.046	2.74	5.937
33	0.039	0.342	1.213
34	0.027	0.438	1.061
35	0.038	0.782	1.263
36	0.038	6.594	2.695
37	0.035	4.465	5.144
38	0.032	3.716	3.016
39	0.318	15.379	7.869
40	0.043	5.129	2.769

consumption. Moreover, it shows relatively stable performance. An interesting aspect of the Concorde solver is its requirement that distances between sites be integers. While this requirement might initially appear restrictive, it aligns well with the TSP with arbitrary arc lengths, where rational numbers can closely approximate the real numbers. A real number's approximation by a rational number, followed by transforming a problem with rational coefficients into one with integer coefficients (by multiplying by the least common multiple of the denominators), is a feasible process for TSP instances. It seems that Concorde solver leverages such a trick to solve the problem in an extremely short time.

In comparison, the experimental data reveal a trend that the time consumption noticeably increases as the problem

size increases in the case of the lazy constraints approach and all-at-once approach. It seems that the lazy constraints approach often surpasses the all-at-once approach regarding time consumption. This could be attributed to the strategic addition of subtour elimination constraints only when a subtour is detected, rather than incorporating all possible constraints from the beginning. The method likely benefits from a reduced computational load and a more targeted exploration of the solution space, facilitating quicker convergence to optimal solutions.

Nonetheless, evaluating the solvers based on a single instance per problem size may not fully capture the performance gaps, especially between the lazy constraints and all-at-once approaches. Experiment 2 was designed to address this gap and provide a more detailed comparison of these strategies.

+) Figure 2 which presents a sequence of solutions showcasing the detection and resolution of subtours in the lazy constraints approach while solving the instance with 10 sites. It moves in Adobe PDF Reader, but I'm not sure in other pdf viewer. It is also submitted as 'animation_tsp10.gif'

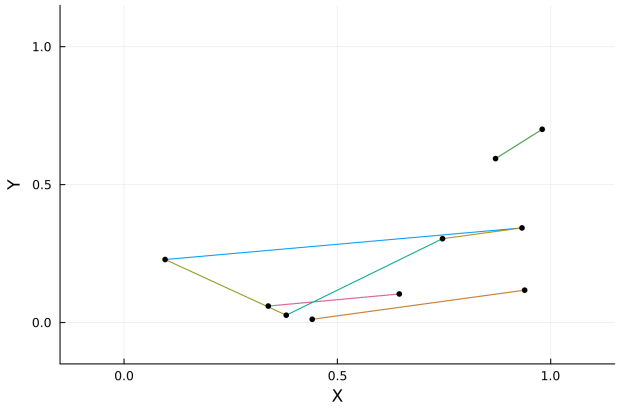

Figure 2: Lazy Constraints Solution Procedure Visualization ($|N| = 10$)

3.2 Experiment 2: multiple instances with the same number of sites

In Experiment 2, the three algorithms solve 100 instances of TSP with 30 sites. The mean and standard deviation of the time consumption are given in Table 3. The result tells that Concorde totally surpasses other, and lazy constraints approach surpasses all-at-once approach.

Table 3: Mean and Standard Deviation of Time Consumption of Each Solver in Experiment 2

Solver	Mean (sec)	Standard Deviation (sec)
Concorde	0.036	0.012
All-at-Once	15.12	84.82
Lazy Constraints	0.64	0.58

4 Conclusions

To summarize the findings,

- Lazy constraints approach is much better than all-at-once approach
- Concorde solver is way better than lazy constraints and all-at-once solvers with Gurobi

References

https://www.gurobi.com/documentation/11.0/examples/tsp_py.html

https://jump.dev/JuMP.jl/stable/tutorials/algorithms/tsp_lazy_constraints/#Lazy-constraint-method

https://opensourc.es/blog/mip-tsp/

https://orinanobworld.blogspot.com/2012/08/user-cuts-versus-lazy-constraints.html

https://gist.github.com/chkwon/d10d0dd3adbae8c145d680403ea1af18