Combinatorial Optimization Project: The Traveling Salesman Problem

Alberto Parravicini

1 Introduction

The **Travelling Salesman Problem (TSP)** is a combinatorial optimization problem in which a salesman, starting from a home location, has to visit a predefined set of cities and go back to the original one, in a way that each city is visited **just once**.
Moving from one city to another has a fixed *cost* that depends on the cities, and the goal is to find the path that will visit all the cities with *minimum cost*.
Note that given 2 cities, the cost to move between them could be *symmetric* or not. We will focus on the more general case, without symmetry.

There exists various formulations for the **TSP**, but all of them stem from the same root formulation.
In this report, we'll first present 3 different formulations for the **TSP**, some of which are more interpretable, while other are more efficient (in terms of number of *constraints* or *variables*).

These formulation have been implemented in **Julia** by using the **JuMP** package, and have been further optimized to provide the optimal solution as fast as possible. In the second section we'll detail our implementation, while in the third section we'll discuss the execution times and the efficiency of the implementations, with or without additional optimizations.

2 Formulations for the TSP

2.1 *Common notation*

In all the formulations that we are going to present there are a number of common points that can be discussed here.

The set of cities (vertices) V to be visited, and the set of paths (arcs) A between them, can be viewed as a directed graph $G = (V, A)$.

Each arc $a \in A$ goes from a city $u \in V$ to a city $v \in V$, and has a non-negative cost c_a (or c_{uv}) associated to it. The goal is to find an **Hamiltonian cycle**, a path that visits each vertex **exactly once**, of minimum cost.
Note that in the symmetric case, given 2 cities $u,\ v \in V$ the cost from u to v is equal to the cost from v to u, i.e. $c_{uv} = c_{vu}$.
Moreover, we can assume without loss of generality that the graph is **complete**, i.e. there is an arc between each pair of cities. Incomplete graphs can simply be represented as complete graphs in which the missing arcs have an arbitrary large cost, and will never be part of the optimal solution.

Let the vertices be denoted by $\{1, 2, \ldots, n\}$; the costs between each pair of vertices can be represented by the **cost matrix** $C = (c_{ij})_{n \times n}$, where the entry c_{ij} is the cost from vertex i to vertex j.

2.2 *Linear programming formulation*

The simplest formulation of the **TSP** is obtained by *linear programming*. Let $\mathbb{F}$ be the set of Hamiltonian cycles of G, represented as collection of incidence vectors in $\mathbb{R}^{|E|}$, and $P(\mathbb{F})$ its *convex hull*. Then the problem is formulated as:

$$\begin{aligned} \text{Minimize} \quad & \sum_{j=1}^{m} c_j x_j \\ \text{Subject to} \quad & X \in P(\mathbb{F}), \end{aligned}$$

where $X = (x_1,\ x_2,\ \ldots,\ x_m)$ is the set of edges of G.
However there is no known complete linear formulation for $P(\mathbb{F})$, which limits the applicability of this formulation.

2.3 *Integer programming formulation*

A common formulation for the asymmetric TSP (**ATSP**) makes use of binary variables x_{ij}, which are $= 1$ if and only if the path from city i to city j is used. Then, the problem can be expressed as:

$$
\begin{aligned}
\text{Minimize} \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} \\
\text{Subject to} \quad & \sum_{i=1}^{n} x_{ij} = 1,\ j = 1,\ \ldots,\ n, \\
& \sum_{j=1}^{n} x_{ij} = 1,\ i = 1,\ \ldots,\ n, \\
& x_{ij} = 0 \ or \ 1
\end{aligned}
$$

or, more compactly, as:

$$
\begin{aligned}
\text{Minimize} \quad & C \cdot X \\
\text{Subject to} \quad & X \cdot \mathbf{1} = \mathbf{1}, \\
& X^T \cdot \mathbf{1} = \mathbf{1}, \\
& X \in \{0,1\}^{n \times n}
\end{aligned}
$$

with $\mathbf{1}$ being a vector of 1 of size $n \times 1$. The first constraint imposes that each city is left once, and the second constraint that each city is reached once.
Note that this formulation is incomplete, as it allows the presence of **subtours**, i.e. disjoint tours that still respect the previous constraints.
We would like however to have a single tour, and to do so it is necessary to add more constraint, defined as *subtour elimination constraints*.

The three formulation that are proposed differ exaclty on how these additional constraints are implemented.

2.4 *MTZ constraints*

This first approach to *subtour elimination* was proposed by **Miller, Tucker and Zemlin**.
The idea is to add $n-1$ new unrestricted real variables and $(n-1)^2$ additional constraints of the form:

$$
(n-1)x_{ij} + u_i - u_j \leq (n-2), \quad i,j = 2,3,\ldots,n \tag{1}
$$

If there is more than 1 tour, at least one of them won't pass for node 1. Denote this subtour as $\hat{T}$; if $\hat{T}$ is a single node, the constraint will be violated for $i = j = k$, as $x_{ij} = 1$ and we find $-1 \leq -2$.
If $\hat{T}$ has more than one node, we can add a constraint for each arc in the subtour,

and find another contradiction.
For instance, if a subtour is composed by vertices 2 and 3, we will find:

$$(n-1)x_{23} + u_2 - u_3 \leq (n-2)$$
$$(n-1)x_{32} + u_3 - u_2 \leq (n-2)$$

from which follows $-2 \leq -4$, a contradiction.

2.5 *Claus constraints*

Another way to express the *subtour elimination constraints* is to express them as flow constraints. To each arc ij we can associate a non-negative quantity of flow y_{ij} that is transported through the arc.

Claus presented a representation that uses $(n-1)$ different *commodities*, different types of flows, each denoted by y^k, $k = 2, 3, \ldots, n$. In this representation, subtours are eliminated as follows:

$$\sum_{i=1}^{n} y_{1i}^k = 1, \quad k = 2, 3, \ldots, n \tag{2}$$

$$\sum_{i=1}^{n} y_{i1}^k = 0, \quad k = 2, 3, \ldots, n \tag{3}$$

$$\sum_{i=1}^{n} y_{ik}^k = 1, \quad k = 2, 3, \ldots, n \tag{4}$$

$$\sum_{i=1}^{n} y_{ki}^k = 0, \quad k = 2, 3, \ldots, n \tag{5}$$

$$\sum_{i=1}^{n} y_{ij}^k - \sum_{i=1}^{n} y_{ji}^k = 0, \quad k = 2, 3, \ldots, n;\ j \neq k \tag{6}$$

$$y_{ij}^k \leq x_{ij}, \quad i, j = 1, 2, \ldots, j;\ k = 2, 3, \ldots, n \tag{7}$$

The last set of constraints is called **coupling constraints**, and is used to connect the original variables x to the flow variables y.

2.6 *FCG constraints*

Another formulation that makes use of *flow constraints* was given by **Finke, Claus** and **Gunn**.
In this case we have 2 types of commodities y and z.

$$\sum_{j=1}^{n} y_{1j} - \sum_{j=1}^{n} y_{j1}^{k} = n - 1 \tag{8}$$

$$\sum_{j=1}^{n} y_{ij} - \sum_{j=1}^{n} y_{ji} = -1, \quad i = 2, 3, \ldots, n \tag{9}$$

$$\sum_{j=1}^{n} z_{1j} - \sum_{j=1}^{n} z_{j1} = -(n-1) \tag{10}$$

$$\sum_{j=1}^{n} z_{ij} - \sum_{j=1}^{n} z_{ji} = 1, \quad i = 2, 3, \ldots, n \tag{11}$$

$$\sum_{j=1}^{n} y_{ij} + \sum_{j=1}^{n} z_{ij} = n - 1, \quad i = 2, 3, \ldots, n \tag{12}$$

$$y_{ij} \geq 0, \quad i, j = 1, 2, \ldots, n \tag{13}$$

$$z_{ij} \geq 0, \quad i, j = 1, 2, \ldots, n \tag{14}$$

$$y_{ij} + zij = (n-1)x_{ij} \; for \; all \; (ij) \tag{15}$$

3 Implementation

The formulations presented in the previous section have been implemented using the **JuMP** package for **Julia**. In this section it is described the overall structure of the program, and how to execute it.

3.1 *Prerequisites*

The following packages are required to run the program:

- **GLPKMathProgInterface**
- **Gurobi** (optional, but recommended)
- **ArgParse**
- **LightGraphs, GraphPlot, Compose, Colors** (optional, but recommended)

3.2 *How to run the programs*

The generic syntax to run the program is

```
julia ./src/main.jl
  --instancename instance_name.atsp
  --randomseed 42
  --solver gurobi
  --constraints claus
  --usehotstart annealing
  --printlevel 2
  --drawgraph true
  --saveresult false
```

Additionally, the flag `-h` or `-help` will display the help screen, with a description of the arguments, and their accepted values.

- `--instancename`, `-i`: path to the instance file to be used. The file is assumed to be located in the 'instances' folder.
- `--randomseed`, `-r`: integer number, used as seed for random number generation by the program. If omitted, the seed is randomized.
- `--solver`, `-s`: name of the solver that will be used; choose between 'gurobi' and 'glpk'. Note that Gurobi requires a license to be used! (default: **Gurobi**)
- `--constraints`, `-c`: Type of *subtour* elimination constraints used in the model; choose between 'mtz', 'fcg', 'claus' (default: **claus**).
- `--usehotstart`: compute the initial solution using the specified heuristic algorithm; choose between 'none', 'random', 'annealing' (default: **none**).

- `--printlevel`, `-p`: Amount of details that will be printed by the solver; it takes an integer value >= 0; higher values will print more details (default: 0).
- `--drawgraph`, `-d`: if true, draws the graph corresponding to the given instance, and highlights the optimal tour;the optimal tour is also drawn on a separate file (default: **false**).
- `--saveresult`, `-v`: if true, the variable values are stored in a file, and the statistics of the computation are appended to the file 'results.csv' (default: **false**).

Example:
```
julia ./src/main.jl -i 1.atsp -r 69 -constraints mtz -d true -p 2
```

This command will solve the instance number 1 using **MTZ** constraints, random seed 69 and no hot start; additionally, a large amount of details will be printed, the plots of the graphs will be saved, but not the results of the optimization. **Gurobi** will be used as solver.

Note: the program should be run from the folder `combinatorial-optimization-ulb`, because Julia interprets relative paths from where the program is ran, and not from the actual location of the program.

Note: the folder `combinatorial-optimization-ulb` contains a **Python** script that automatically runs the program on all the instances, with the specified settings.

3.3 *Overview of the implementation*

This section will focus on the **Julia** implementation of the **TSP** solver. For each file, it is provided a brief description; each file is commented and documented, and more specific details can be found by reading the code.

- `main.jl`

Main access point to the program. It is just a wrapper to the other modules which will read the input arguments, build and solve the model, and save the results.

- `arg_parser.jl`

This module will process the input line arguments, and return the list of options that will be used to build the model and optimize it.

- `plotter.jl`

Utility module that is used to draw the results of the optimization. After finding the optimal solution it will be produced an **SVG** file that contains the entire graph, with the optimal path highlighted, and another **SVG** file which contains just the optimal path, with the costs of the arcs specified.

- `tsp_solver.jl`

This is the hearth of the program. Here are contained the functions to build the model according to the specified parameters (e.g. the type of constraints), and is provided the main function to optimizer the problem.

Note that for each constraint it is provided a reference label to access it (or modify it) as desired, and vectorization is used whenever possible.

Also note that it is possible to specify an adjacency matrix for the graph, so that the computation on incomplete graphs is optimized by not producing useless constraints.
If the graph is complete (the default case), it is produced an adjacency matrix with the main diagonal set to 0, as self-loops are not allowed in the **TSP**.

- `random_pick_heuristic.jl`

This is a very simple heuristic used to find the initial state of the algorithm. A large number of random initial solution is produced, and the best one is kept.
However in practice most optimizers (such as **Gurobi**) provide heuristic algorithms to compute the initial solution, and the results might be better than the ones provided by random sampling.

Note that in heuristic algorithms for the **TSP** is often better to reason about the optimal tour in terms of permutation of vertices, instead than as an adjacency matrix.

- `simulated_annealing_heuristic.jl`

A more complex heuristic used to provide a good initial solution to the optimizer. The idea is to start from an initial solution and iteratively modify it (for instance, by swapping the order of visit of 2 random cities) in order to build a better solution.
After modifying the current solution, the result will be accepted with a probability that depends on how better it is compared to the previous one, and on how many iterations the algorithm has done already (more iterations imply lower probability).
This algorithm provides better results than *random sampling*, but it's often slower than leaving the heuristic initialization to the solver.

4 Results analysis

In this section the implementation is tested against 8 instances of various size (2 of the 10 instances weren't used in this test as the time to solve them was much higher than the others), in order to see how the algorithm behaves when using the different parameters that can be specified. For each instance, 10 repetitions have been performed.
First, the 3 types of constraints are tested, without using any custom heuristic initialization.
Then, the best of the 3 constraint types is tested *with* and *without* additional optimizations, to see if there are significant improvements over the basic implementation.

All the tests were done by using **Gurobi**, with the default parameters.

To have comparable results across the tests on a single problem instance, the **seed** used by the *random number generator* was kept fixed across the tests on the same instance.

The tests were performed on the following machine:

- Computer: Microsoft Surface Pro 4
- CPU: Intel Core i5-6300U at 2.4 GHz (clocked at 2.95 Ghz)
- RAM: 4 GB at 1867Mhz

4.1 *Tests on the different formulations*

In this section it is tested the execution time of the program when using the 3 different *subtour elimination constraints* types that have been presented before.
Each formulation has a different number of variables and constraints, and in some cases the additional variables can be *real*, *binary* or *integer*.
As such, we expect to find that some formulations will be slower, and that the differences could become larger, or smaller, depending on the instance size. The program was run against 8 instances of various size, and the results are reported below.

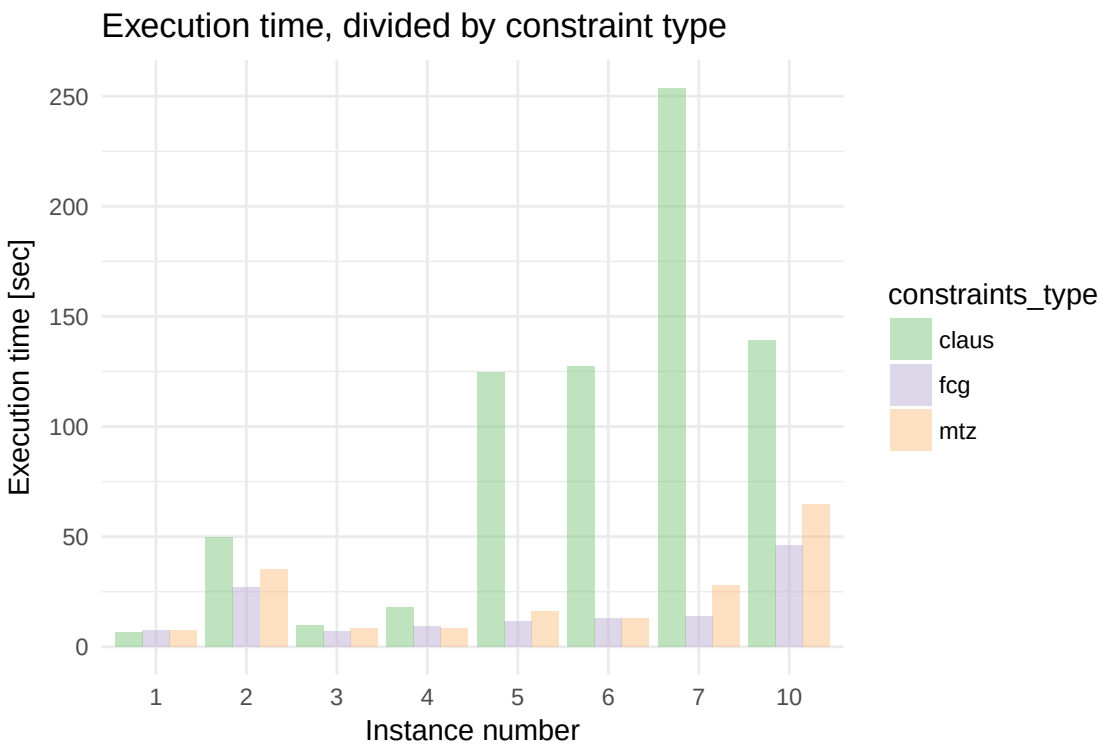

Figure 1: *Bar plots of the execution times for the instances, divided by constraint type.*

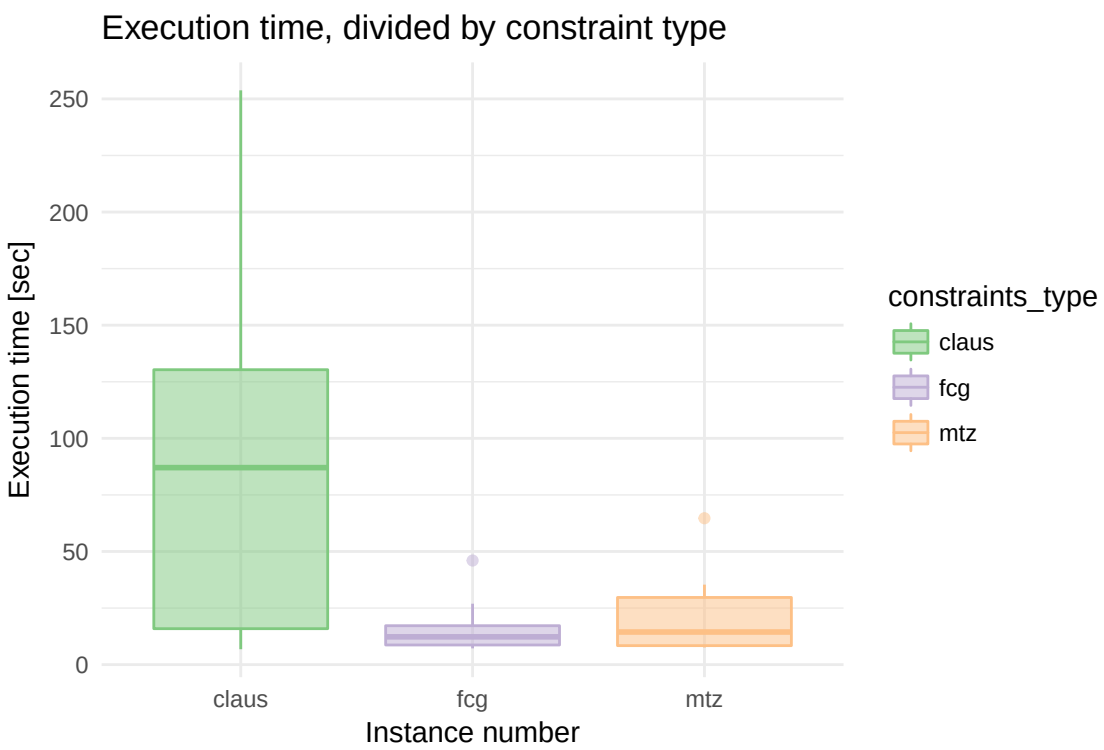

Figure 2: *Box plots of the execution times for the constraint types.*

From the execution times we can see that **FCG** amd **MTZ** constraints are usually faster, and provide more consistent results.
Claus constraints are sometimes as fast as the others, but on few occasions they result much slower.
Their higher variance in execution time is also confirmed by the box-plots.

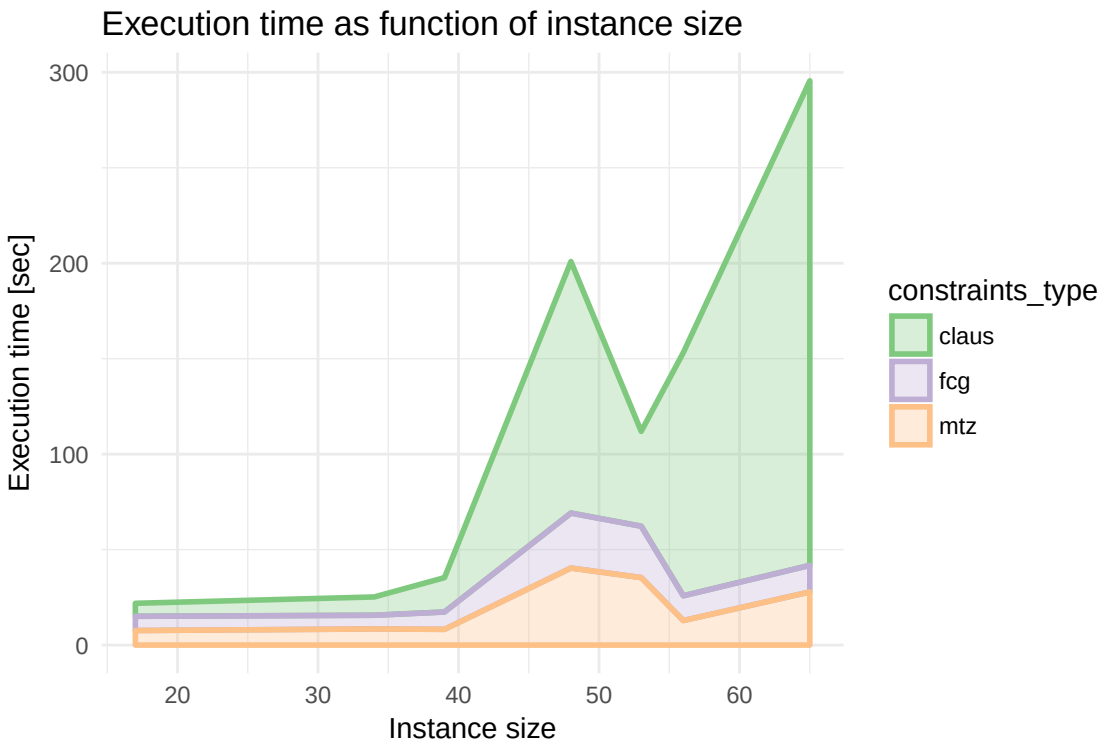

Figure 3: *Execution time as function of instance size, divided by constraint type.*

Given the limited number of instances it would be premature to draw definitive conclusions regarding the relation between instance size and execution time. Still, it seems clear that for smaller instances the 3 types of constraints perform almost equivalently, while **Claus** becomes much slower (and variant) as the instances grow in size.

Type	Min	Mean	Median	Max	Std. Dev.
Claus	6.67	92.88	89.66	258.38	87.70
FCG	7.18	16.90	12.28	46.00	13.33
MTZ	7.62	21.80	13.27	64.60	19.80

Table 1: Summary statistics of the **execution time** of the various constraint types.

It's evident that **Claus** is overall the worst choice, while the other 2 seems quite similar. Testing with good statistical significance whether one is actually superior

would require more instances, or more repetitions, though.
In any case, **FCG** seems sightly faster. In the next section, we'll test whether using further optimizations can increase its speed even more.

4.2 *Additional improvements*

In this section it will be tested if the addition of an heuristic algorithm at the start of the optimization process can speed up the overall optimization, by providing an initial solution close to the global optimum.

The tests were done on the same 8 instances used before, again wirth 10 repetitions. In this case, only the **MTZ** constraint type was considered, being the most efficient of the 3.
Here, we test if using **no** custom initial heuristic (and leaving the initial heuristic to the solver) is better than **random sampling** (where a large number of random states are tested, and the best is kept) and **simulated annealing**.

Figure 4: *Execution time as function of the initial heuristic.*

Type	Min	Mean	Median	Max	Std. Dev.
None	7.62	21.87	13.29	64.60	19.84
Random	7.41	16.61	12.68	35.28	10.81
Annealing	8.44	24.68	22.23	43.44	13.78

Table 2: Summary statistics of the **execution time** of the various initial heuristics.

We can notice how using **simulated annealing** gives an overall slower execution time. This is because the algorithm itself is quite slow, and the benefits given by the better initial solution are compensated by the longer time required to find it. Using different sets of parameters seems to improve sightly the performances, but not to a significant amount.
On the other hand, using **random sampling** seems to give small improvements on the execution time, which is quite surprising, given the simplicity of the heuristic.