Algorithm Engineering
WS 22/23

Donatien Leray
Cedric Wiese

A Comparative Study of full and lazy constraints ILP-formulation for TSP

Abstract

We implement and compare the practical performance of two different approaches of Integer Linear Programming. Our goal is to find an optimal tour from a list of cities. For this task, two different approaches are available with the help of ILP solvers. Namely the full solving approach, or a lazy variant. Although both approaches lead to an optimized, there are major differences regarding performance.

1 Introduction

The Traveling Salesman Problem (TSP) is a classic optimization problem in computer science and mathematics. It involves finding the shortest route that visits a set of cities exactly once and returns to the starting city. The TSP has numerous real-world applications, including logistics, route planning for delivery trucks, and scheduling for tour groups.

This report will provide an overview of the TSP and its applications, and will present a comprehensive analysis of several heuristics and approximation algorithms that have been developed to solve the problem. The effectiveness and limitations of these algorithms will be evaluated and compared to identify the best approach for solving the TSP.

2 Preliminaries

The TSP is considered to be an NP-hard problem, meaning that finding an optimal solution for large instances of the problem is computationally difficult and time-consuming. As a result, various heuristics and approximation algorithms have been developed to solve the TSP efficiently.

The problem is commonly defined as an undirected graph, where each city is represented by a node and the distance between two cities is represented by an edge. The goal is to find a Hamiltonian cycle, which is a cycle that passes through every node exactly once and returns to the starting node.

2.1 Solving TSP with Integer Linear Programming

Integer Linear Programming (ILP) is a mathematical optimization method that can be used to solve the Traveling Salesman Problem (TSP). In ILP, a problem is modeled as a linear program and solved using linear optimization techniques, with the added constraint that the solution variables must be integers.

To solve the TSP using ILP, the problem is modeled as a linear program where the variables represent the decision of whether to visit a particular city or not. The objective function is to minimize the total distance of the tour, and the constraints ensure that every city is visited exactly once and that the tour starts and ends at the same city.

The ILP model for the TSP can be formulated as follows:

$$\text{Minimize:} \quad \sum_{(i,j)\in E} c_{ij} x_{ij} \tag{1}$$

$$\text{Constraint 1:} \quad \sum_{(i,j)\in E} x_{ij} = 2 \quad \forall i \tag{2}$$

$$\text{Constraint 2:} \quad x_{ij} \in 0, 1 \quad \forall (i,j) \in E \tag{3}$$

$$\text{Constraint 3:} \quad \sum_{j} x_{ij} = 1 \quad \forall i \tag{4}$$

where E is the set of edges connecting the cities, c_{ij} is the cost of traveling from city i to city j, x_{ij} is a binary variable representing the decision to travel from city i to city j, and the constraints ensure that each city is visited exactly once and the tour starts and ends at the same city.

ILP is known for its ability to provide exact solutions to optimization problems, and can be used to solve the TSP for small to medium-sized instances. However, the complexity of the ILP model increases rapidly with the number of cities, making it impractical for large instances of the TSP. In such cases, heuristics or approximation algorithms are more suitable for solving the TSP efficiently.

2.2 Full ILP approach

In the full approach, all subtour constraints are added to the solver before it starts to solve. Subtour constraints are used to eliminate the possibility of visiting cities in a non-tour manner, such as visiting a city more than once or visiting cities in a circular fashion that does not return to the starting city. The full approach involves adding all possible subtour constraints upfront to the ILP model, resulting in a larger and more complex model. However, this approach is guaranteed to find the optimal solution in a finite number of iterations.

2.3 Lazy ILP approach

In the lazy approach, subtour constraints are only added to the solver when subtours have been found. The lazy approach starts with a smaller and simpler ILP model, and as the solver finds subtours, the subtour constraints are added to the model on the fly. This results in a smaller search space, making the optimization process faster. However, there is no guarantee that the optimal solution will be found, as the solver may miss some subtours.

3 Algorithm & Implementation

3.1 Full ILP approach

Algorithm 1 solve_stuff_full

```
1:  procedure SOLVE_STUFF_FULL(edges, l)
2:      result_setup_time ← 0
3:      result_solve_time ← 0
4:      timer ← new Instant
5:      problem ← new ProblemVariables
6:      y ← new Vec of length edges.len()
7:      for i ← 0 to edges.len() − 1 do
8:          y[i] ← problem.add_variable().binary()
9:      end for
10:     objective ← target_func(edges, y)
11:     ccp ← problem.minimize(objective)
12:     for i ← 0 to l − 1 do
13:         ccp.constraint(two_city(y, l, i) = 2)
14:     end for
15:     subtours ← powerset(0 · · · l)
16:     for s ∈ subtours do
17:         if s.len() < 3 or s.len() = l then
18:             continue
19:         end if
20:         constraint ← sub_tour_constraint(s, y, l)
21:         ccp.constraint(constraint ≤ (s.len() − 1))
22:     end for
23:     setup_time ← timer.seconds()
24:     timer ← new Instant
25:     solution ← ccp.solve()
26:     solve_time ← timer.seconds()
27:     tour ← new Vec of length y.len()
28:     for i ← 0 to y.len() − 1 do
29:         tour[i] ← solution.value(y[i])
30:     end for
31:     checked ← check_tour(tour, l)
32:     return (setup_time, solve_time, checked[0])
33: end procedure
```

For the Full ILP approach, we pass our *Solve_Full* function our edges (which contain the lengths of the edges), as well as the number of cities (here: l). The timers are only necessary for measuring the setup time and the solve time, so they will not be explained further.

Within the function, we start off by initiating a new vector for our problem variables. For each edge we add a binary variable to our vector of problem variables.

Next, we create our objective, which uses "target_func()". "target_func()" creates a linear expression, which represents the total length of the tour. The expression builds the sum of all variables (0 or 1) multiplied with the edge length. This objective should be minimised by the solver.

For the constraints, we first add the base constraints, which make sure, that each city is connected to exactly two other cities. We can achieve this by using the "two_city()" function, which, for a specific city, creates a linear expression, adding up all variables which represent an edge to this city. By setting this expression equal to 2, it is guaranteed, that no city is connected to either less or more other cities.

For the subtour constraints, we continue with creating the powerset of all cities. Before adding a subtour for each subset, we check that the size of the subset is longer than 2 (since at least 3 cities are necessary for a circle) and unequal to the number of cities, since we do not want a constraint disallowing all cities to be in one tour.

With the problem variables created and the constraints added, we can now start the solver.

Since the result of the solver is only the result of our variables, we have to create the tour from those variables. Therefore, we create a vector, in which we will push the values of our variables.

Finally, we use "check_tour()", which creates a vector of the city-indices.

3.2 Full ILP approach

For the Lazy approach, we first create two vectors for storning the base constraints and the subtour constraints.

We then create a boolean for tracking our first run, as well as checking whether the solver has found a correct solution.

We enter a while loop which continues until "finished" is true. In the loop, we set up the solver, by creating the problem variables as well as the objective. During the first loop, we also add the base

Algorithm 2 solve_stuff_lazy

```
    procedure SOLVE_STUFF_LAZY(edges, l)
 2:     base_constrs ← Vec :: new()
        subtour_constrs ← Vec :: new()
 4:     finished ← false
        first_run ← true
 6:     sc_counter ← 0
        round_counter ← 0
 8:     tour ← Vec :: new()
        while true do
10:         round_counter+ = 1
            finished = true
12:         y ← new Vec of length edges.len()
            for i ← 0 to edges.len() − 1 do
14:             y[i] ← problem.binary_var()
            end for
16:         objective ← target_func(edges, y)
            ccp ← problem.minimize(objective)
18:         if first_run then
                for i ← 0 to l − 1 do
20:                 ccp.constr(two_city(y, l, i) = 2)
                end for
22:             first_run ← false
            end if
24:         for i ← 0 to subtour_constrs.len do
                ccp.constr(subtour_constr[i].0 ≤
    subtour_constrs[i].1
26:         end for
            sc_counter ← subtour_constrs.len()
28:         solution ← ccp.solve()
            subtour ← Vec :: new()
30:         for i ← 0toy.len() do
                subtour.push(solution.value(y[i])
32:         end for
            checked ← check_tour(subtour, l)
34:         if checked.len() ≥ 2 then
                finished ← false
36:             for cinchecked do
                    constr ← sub_tour_constr(c, y, l)
38:                 subtour_constrs.push((constr, c.len()
    1))
                end for
40:         else
                tour ← checked[0]
42:         end if
        end while
44:     return (sc_counter, round_counter, tour)
    end procedure
```

constraints to our base_constrs vector. Afterwards, we add all of our base constraints and subtour constraints (empty during first run) to the ccp problem.

After solving, we check whether or not subtours have been found. If so, we create a constraint, by taking every set of cities of each subtour, and calculating every possible tour which these cities can create. We then push these new constraints into our subtour_constrs vector. The while loop will continue, until no subtours have been found.

3.3 AE-Principles and techniques

Regarding the AE-principles, we strongly relied on the AE-cycle. We started off by creating an abstract idea of how our code could look like. We then started to implement it step by step, fixing it along the way until it worked as expected. We then tested the run time on various instances, figuring out that we can eliminate all possible subtours of a subset of cities (rather than only the tour which occurred) reduced the running time significantly (close to 93%). Unfortunately, we were not able to keep the solver and only add the subtour constraints, which forced us to always rebuild the solver and pushing all constraints again for every single while loop in the lazy approach.

Regarding libraries, we used the good_lp library, which uses various solvers (from which one can choose). For our tasks, we found that the coin_cbc solver fits our needs the best.

4 Experimental Evaluation

In this section, we conduct a comprehensive evaluation of our algorithms using official TSP datasets based on real-world scenarios. To perform this evaluation, we carry out two separate experiments on the three National TSP datasets of Djibouti, Qatar, and Luxembourg.

In the first experiment, we apply the full ILP formulation on all three TSP datasets. The results of this experiment provide insight into the accuracy and efficiency of our algorithm when using the full ILP formulation.

In the second experiment, we run the lazy constraint version of our algorithm on the same TSP datasets. The results of this experiment will allow us to compare the performance of our algorithm when using the lazy constraint version to the performance when using the full ILP formulation.

In conclusion, these experiments will provide valuable insights into the strengths and weaknesses of our algorithms, helping us to refine and improve our methods for solving the traveling salesman problem in real-world scenarios.

4.1 Data and Hardware

The experiments were performed on a single core of an Intel i7-2600 CPU running at 3.80GHz and equipped with 16GB of RAM. he datasets used were the 3 National TSPs datasets of Djibouti, Qatar and Luxembourg you can find on https://www.math.uwaterloo.ca/tsp/world/countries.html. The TSPs were derived from data contained in the National Imagery and Mapping Agency database of geographic feature names.

To measure the running time of the algorithms, the shell time command was utilized.

```
$ time ./target/release/Exercise05
  'Path_to_tsp_file' 'k' '0' > results.txt
```

the total time solve time, rounds of ILP and subtour elimination constraints. whre measured using

```
$ grep "Time (CPU seconds):" results.txt |
  awk '{sum+=$NF} END
  {print "Sum of Time (CPU seconds):", sum}'
  && tail results.txt -n 10
```

To ensure efficiency, a time limit of 5 minutes was set for each test, and any runs that exceeded this time limit were not recorded.

4.2 Experimental Results

4.3 Full ILP-formulation

The experiment aimed at evaluating the performance of an Integer Linear Programming (ILP) formulation of the Traveling Salesman Problem (TSP). The setup time refers to the time taken to formulate the problem and create a mathematical model for it, while the solving time refers to the time taken to find the optimal solution using an ILP solver.

The results 1 2 3 showed that the setup time for the ILP formulation of the TSP was relatively quick, taking only a few seconds. This was due to the simplicity of the mathematical model and the efficiency of the software used to formulate the problem.

However, the solving time was significantly longer, taking several minutes for larger TSP instances. This was due to the $\mathcal{O}(2^n)$ complexity of the problem and the nature of ILP, which involves finding a solution that satisfies all constraints and optimizes the objective function.

It is noteworthy that the setup time and solving time exhibit an exponential increase in dependecy to k as expected, as evidenced by the data in the 2 4 6.

4.4 Lazy constarints ILP-formulation

The experiment aimed at evaluating the performance of an Integer Linear Programming (ILP) formulation of the Traveling Salesman Problem (TSP) using lazy constraints. The focus was on the number of subtour constraints, solving time, setup time, and number of ILP solving rounds performed.

The results, presented in tables 4 5 6 showed that the use of lazy constraints had a massivly positive impact on the performance of the ILP formulation of the TSP.The setup time for the problem was relatively quick and showed a more linear growth with increasing k, taking only a few milliseconds and becoming nearly negligible for larger values of k. Meanwhile, the solve time appeared to still exhibit an exponential increase, but at a slower rate compared to the full ILP formulation. The solve time was found to be less dependent on k and more heavily influenced by the number of constraints thereby resulting in a less smoothed graph as showed in 8 10 12. The number of subtour constraints displayed a linear growth pattern, but with noticeable irregularities. This variability was largely due to the internal workings of the ILP solver, rather than a direct dependence on k, introducing an element of randomness to the results

wich was reflected by the solving time.

5 Discussion and Conclusion

In conclusion, the comparative study of the Full ILP-formulation and the Lazy Constraints ILP-formulation of the Traveling Salesman Problem (TSP) showed that the use of lazy constraints had a positive impact on the performance of the ILP formulation. The setup time for the Lazy Constraints formulation was quick and nearly negligible for larger values of k, with a more linear growth, while the solve time still had an exponential increase but at a slower rate compared to the Full ILP-formulation. The number of subtour constraints displayed a linear growth pattern, but with noticeable irregularities due to the internal workings of the ILP solver and not a direct dependence on k, bringing an smal element of randomness to the results. The results indicate that the use of lazy constraints can be a promising method to improve the performance of ILP-based approaches to solving the TSP but cant overcome its exponantial nature.

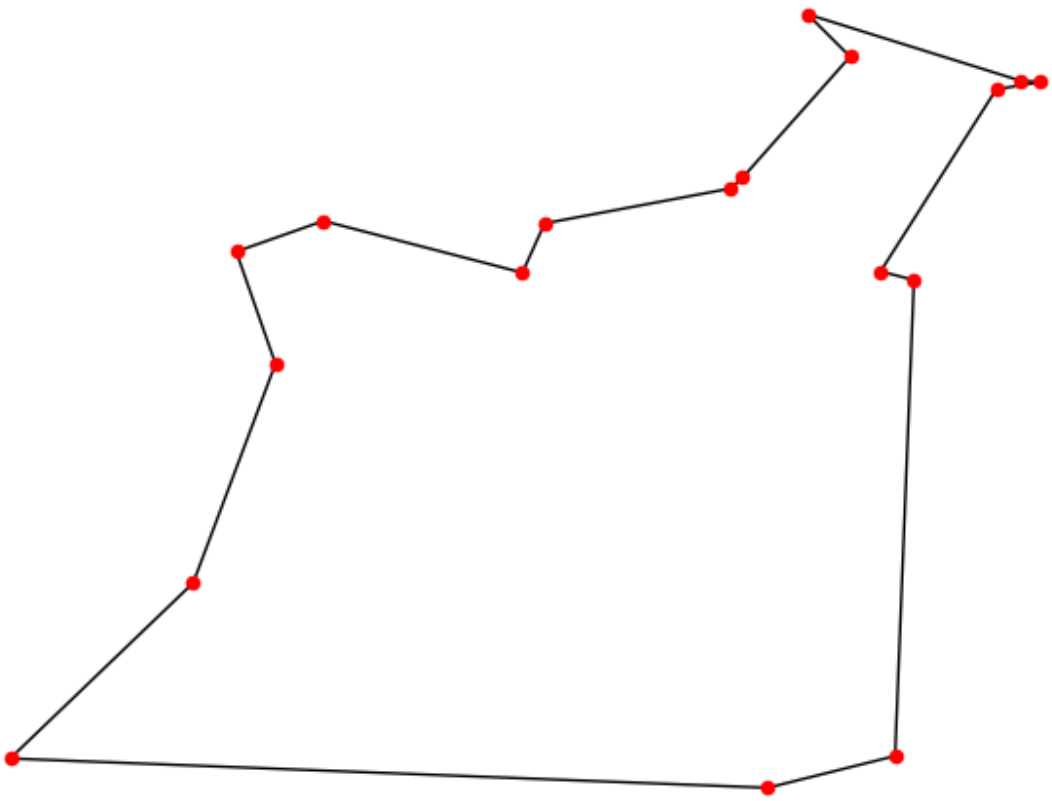

Figure 1: Visualisation of the TSP solution of the 18 first cities from Djibouti found by the full ILP-formulation

k	setup time in ms	solve time in ms
5	0.03	3.49
6	0.08	3.74
7	0.11	4.12
8	0.29	6.84
9	0.68	12.21
10	1.68	26.15
11	4.21	60.34
12	10.28	136.14
13	23.97	324.86
14	58.05	799.84
15	158.07	1,927
16	338.69	4,697
17	782.31	13,455
18	1,838	34,577
19	4,243	84,455

Table 1: the resulting setup time in ms and solve time in ms for given k using full ILP-formulation for Djiboutis TSP

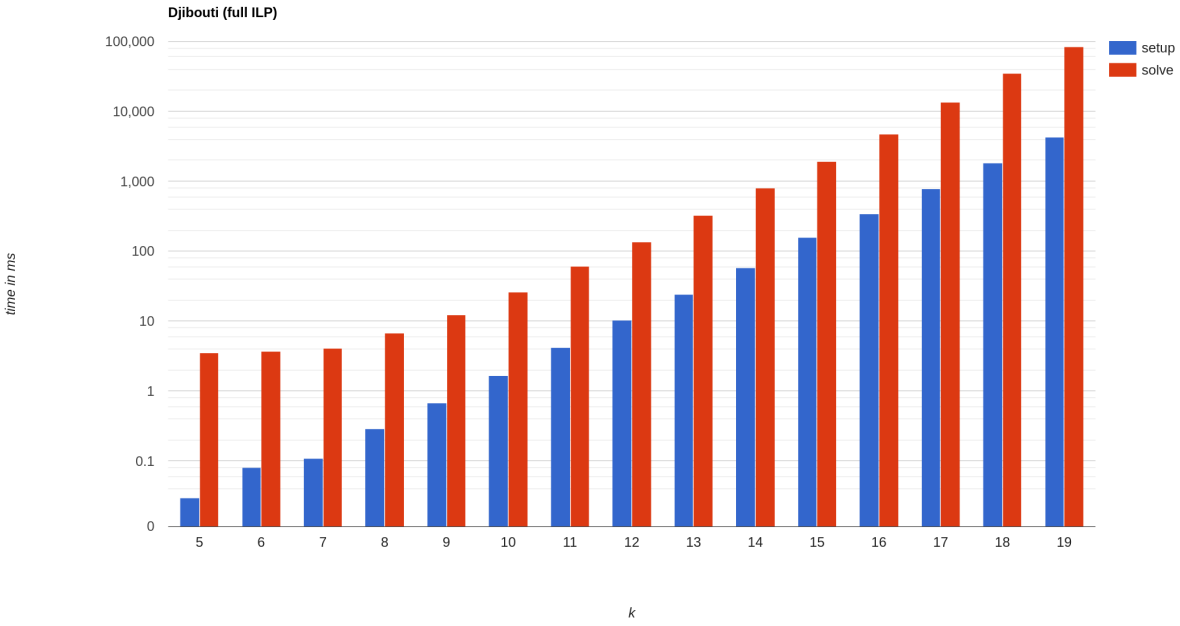

Figure 2: the solving time and setup time set in perspective to each other and k. not the logarithmic scale on the vertical axe. this graphical representation is corresponding to the Data from 1

Qatar (full ILP)

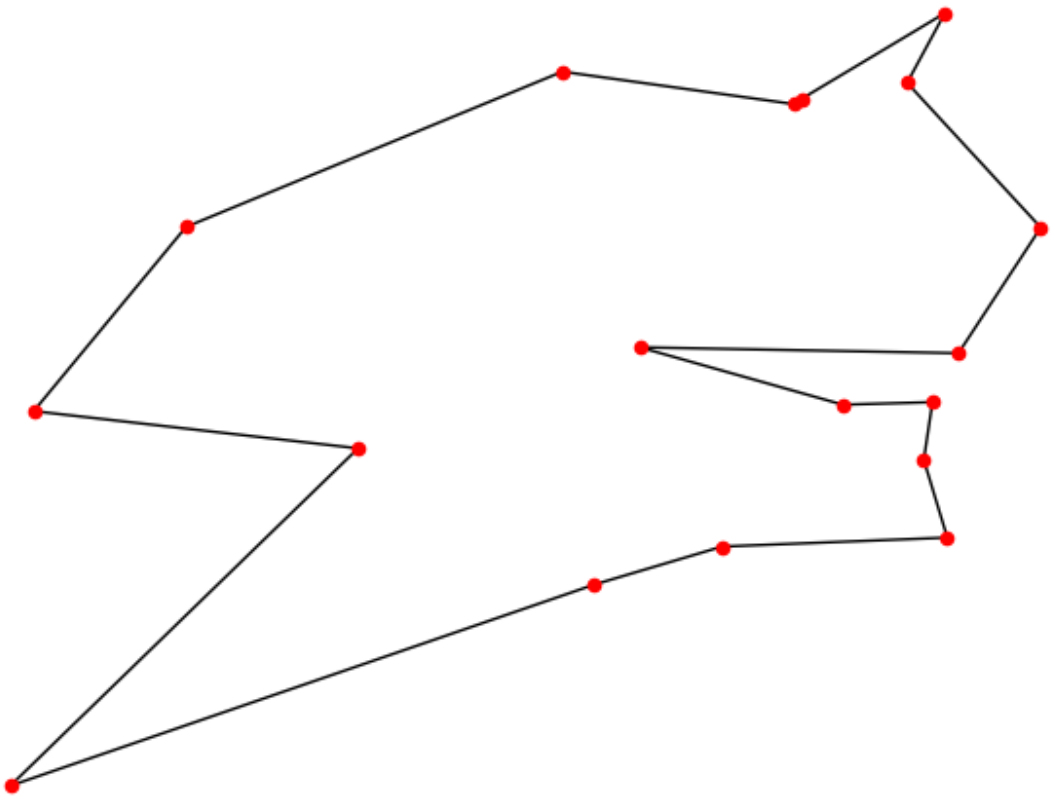

Figure 3: Visualisation of the TSP solution of the 18 first cities from Qatar found by the full ILP-formulation

k	setup time in ms	solve time in ms
5	0.01	6.28
6	0.05	3.57
7	0.12	4.13
8	0.3	6.92
9	0.68	12.66
10	1.5	26.87
11	4.22	55.57
12	10.26	131.93
13	23.54	326
14	57.46	782.84
15	136.97	1,940
16	321.24	4,485
17	780.94	12,861
18	1,838	34,577
19	4,243	84,455

Table 2: the resulting setup time in ms and solve time in ms for given k using full ILP-formulation for Qatar TSP

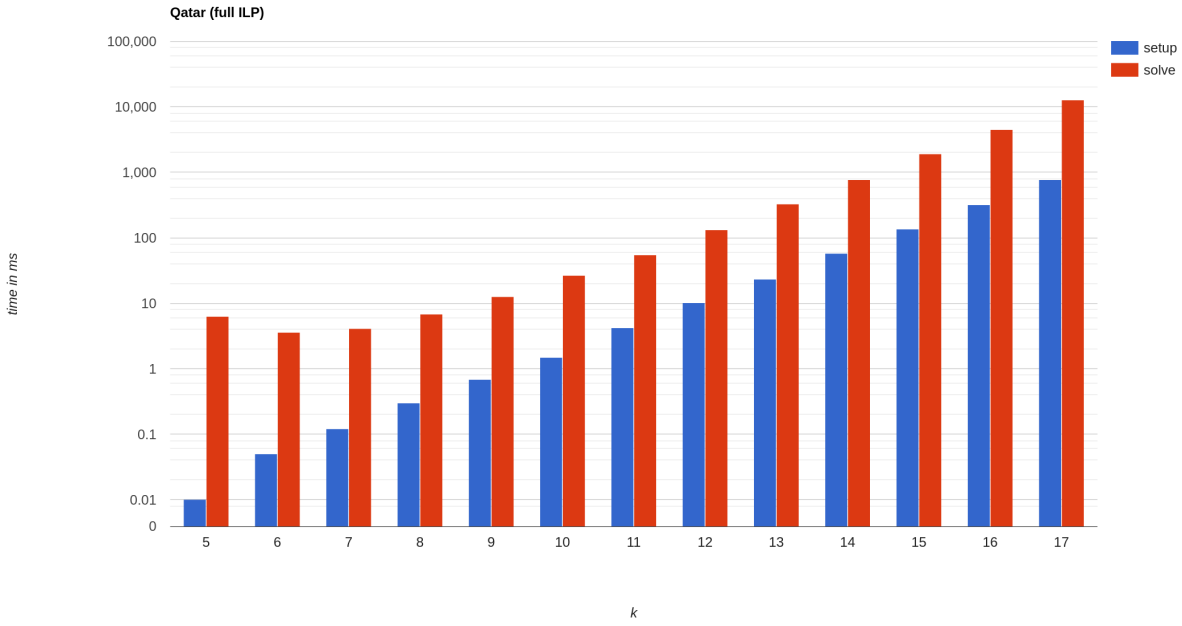

Figure 4: the solving time and setup time set in perspective to each other and k. not the logarithmic scale on the vertical axe. this graphical representation is corresponding to the Data from 2

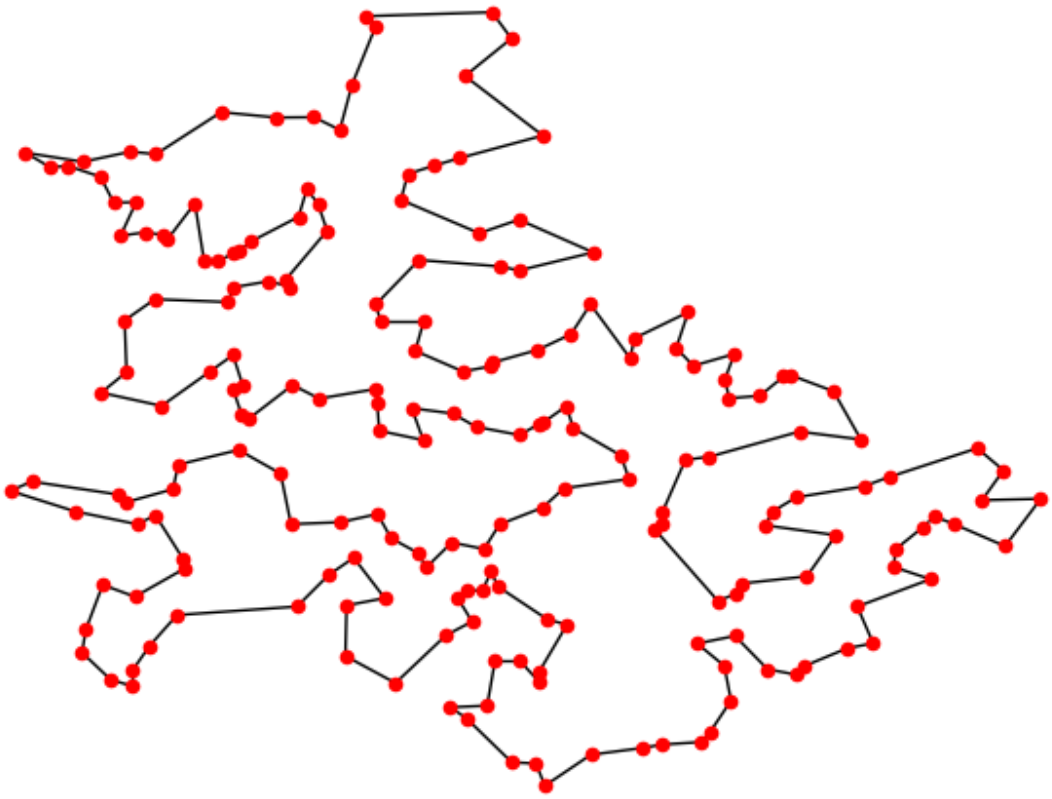

Figure 5: Visualisation of the TSP solution of the 18 first cities from Luxembourg found by the full ILP-formulation

k	setup time in ms	solve time in ms
5	0.02	3.12
6	0.06	4.24
7	0.12	4.22
8	0.25	6.22
9	0.63	12.63
10	2.19	25.58
11	3.91	59.16
12	9.81	138.17
13	24.68	335.89
14	60.03	797.52
15	144.6	1972
16	326.81	4689
17	792.5	13251
18	1876	32117
19	4282	79497

Table 3: the resulting setup time in ms and solve time in ms for given k using full ILP-formulation for Lusembourg TSP

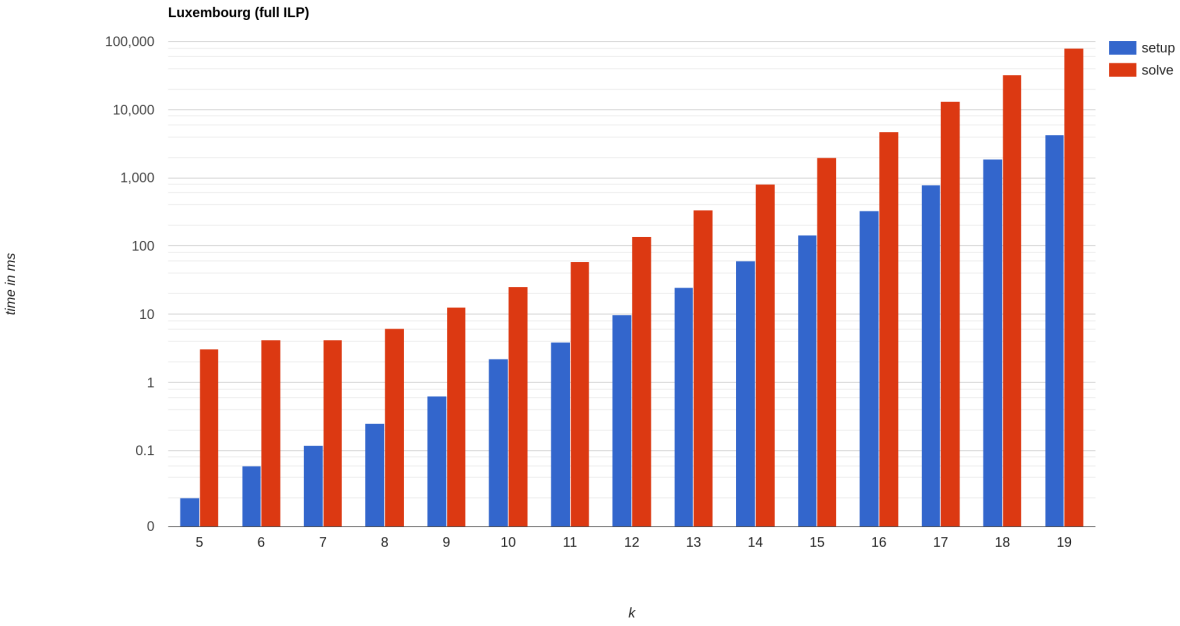

Figure 6: the solving time and setup time set in perspective to each other and k. not the logarithmic scale on the vertical axe. this graphical representation is corresponding to the Data from 3

Djibouti (lazy constraints)

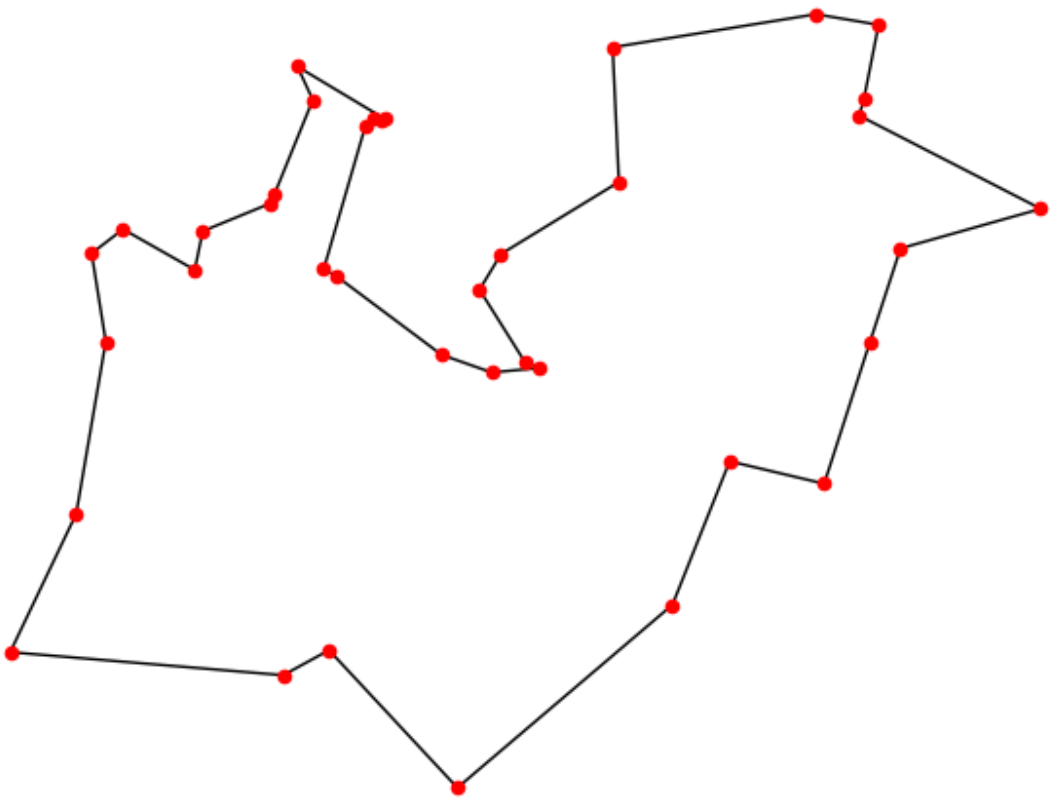

Figure 7: Visualisation of the TSP solution for Djibouti found by the lazy constraints ILP-formulation

k	setup time in ms	solve time in ms	subtour constraints	rounds	relative in %
5	30.28	<10	0	1	0
7	30.37	<10	2	2	0.0156
9	36.48	<10	5	3	0.009
11	39.16	<10	6	5	0.003
13	40.08	<10	5	3	0.6103E-03
15	42.5	<10	5	3	0.1525E-03
17	40.44	<10	7	3	5.3405E-05
19	49.23	<10	13	5	2.4795E-05
21	37.99	50	15	5	7.1525E-06
23	21.06	80	15	4	1.7881E-06
25	10.62	120	16	4	4.7683E-07
27	11.27	90	12	3	8.9406E-08
29	21.36	100	2	2	3.7252E-09
31	26.87	50	6	3	2.7939E-09
33	24.56	140	16	4	1.8626E-09
35	1.27	300	10	4	2.9103E-10
37	25.05	90	10	4	7.2759E-11
38	22.93	120	10	4	3.6379E-11

Table 4: the resulting setup time in ms and solve time in ms, the number of subtour contraints,rounds and latter as an absolute value and also as a percentage of the 2n possible constraints for given k using full ILP-formulation for Djiboutis TSP

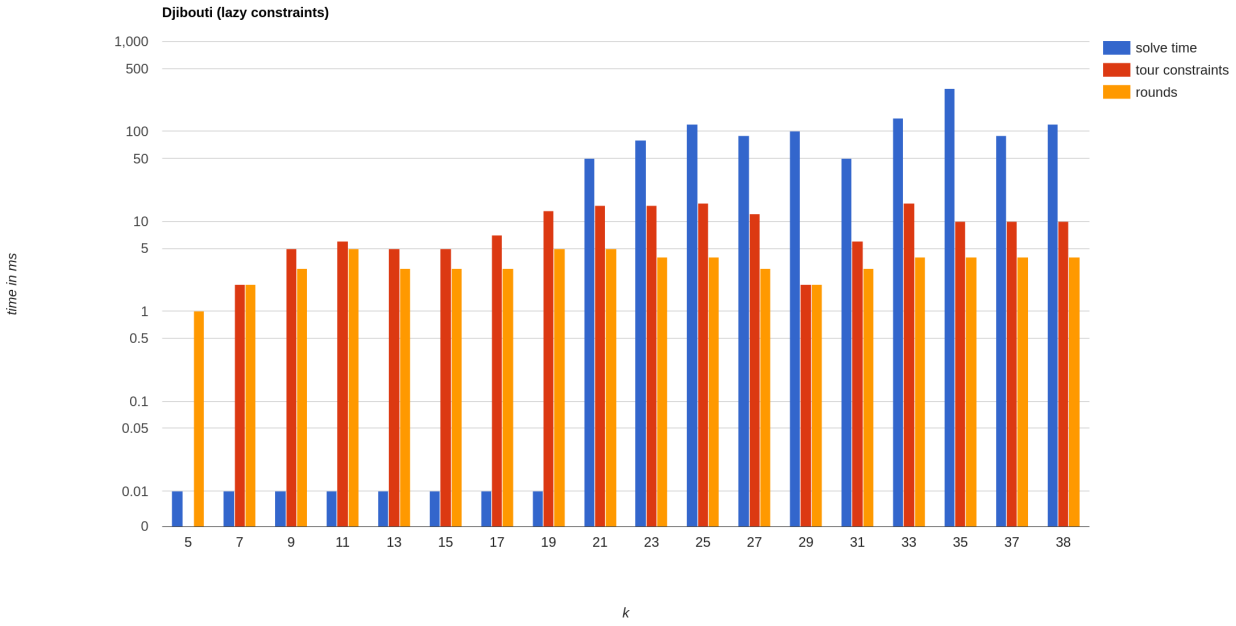

Figure 8: the solving time set in perspective to the number of subtour constarints and rounds for given k. not the logarithmic scale on the vertical axe. this graphical representation is corresponding to the Data from 4

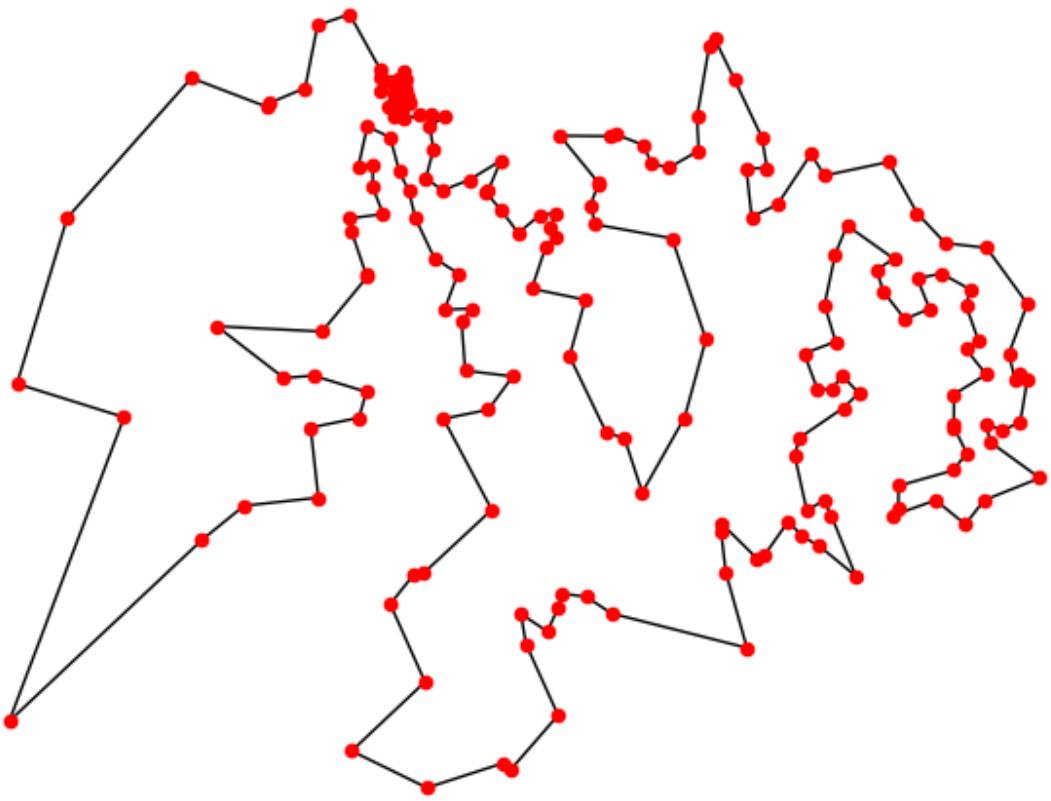

Figure 9: Visualisation of the TSP solution for Qatar found by the lazy constraints ILP-formulation

k	setup time in ms	solve time in ms	subtour constraints	rounds	relative in %
10	30.68	<10	3	2	0.003
20	23.38	60	5	3	4.7683E-06
30	10.16	230	5	3	4.6566E-09
40	41.42	240	5	3	4.5474E-12
50	16.06	440	5	3	4.4408E-15
60	32.76	490	18	4	1.5612E-17
70	20	1,440	22	5	1.8634E-20
80	50	2,060	26	5	2.1506E-23
90	80	2,580	34	6	2.7464E-26
100	110	4,360	38	6	2.9976E-29
110	80	4,690	34	6	2.6192E-32
120	120	5,660	28	5	2.1064E-35
130	100	6,170	35	5	2.5713E-38
140	180	10,240	43	7	3.0850E-41
150	180	9,540	34	5	2.3822E-44
160	180	9,900	38	5	2.6000E-47
170	190	11,200	41	5	2.7395E-50
180	1,880	159,460	69	14	4.5024E-53
190	1,670	124,490	74	15	4.7155E-56
194	1,190	84,350	72	13	2.8675E-57

Table 5: the resulting setup time in ms and solve time in ms, the number of subtour contraints,rounds and latter as an absolute value and also as a percentage of the 2n possible constraints for given k using full ILP-formulation for Qatar TSP

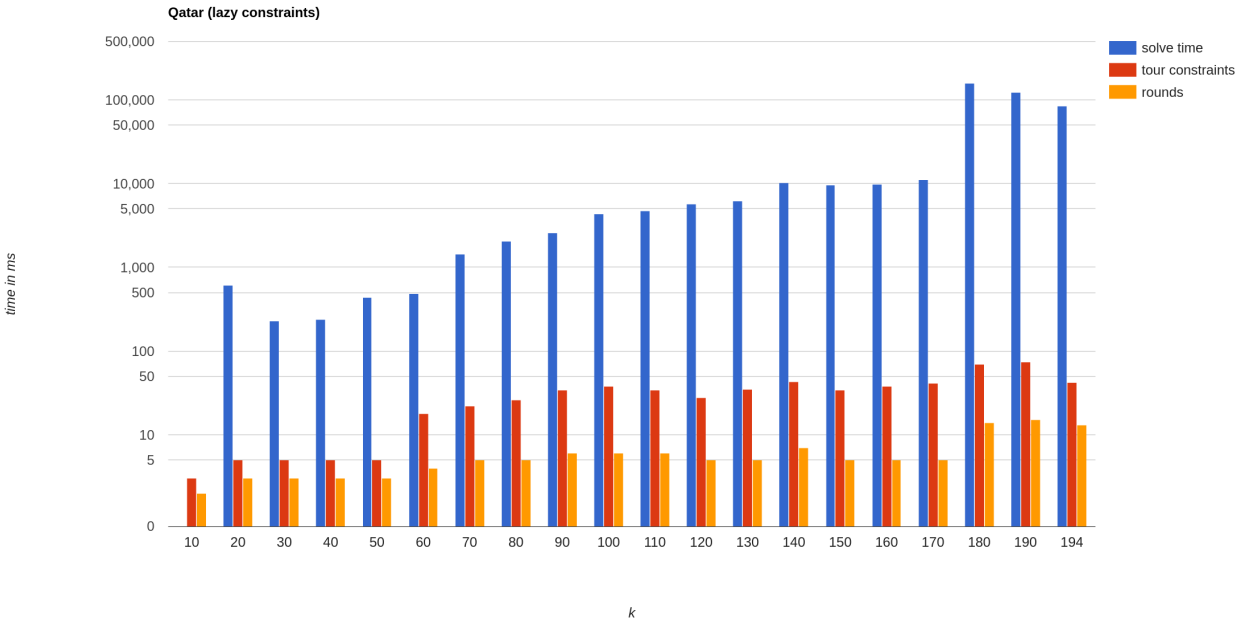

Figure 10: the solving time set in perspective to the number of subtour constarints and rounds for given k. not the logarithmic scale on the vertical axe. this graphical representation is corresponding to the Data from 5

Luxembourg (lazy constraints)

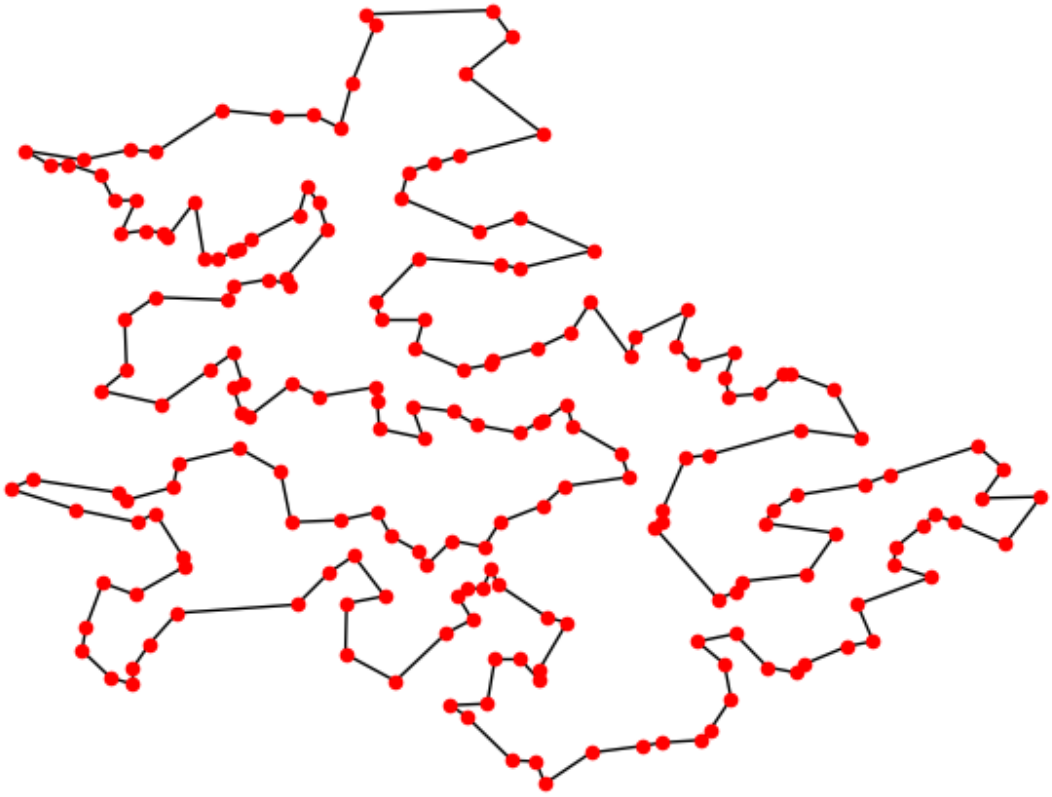

Figure 11: Visualisation of the TSP solution for the first 200 cities from Luxembourg found by the lazy constraints ILP-formulation

k	setup time in ms	solve time in ms	subtour constraints	rounds	relative in %
10	25.51	10	0	1	0
20	34.66	20	5	2	4.7683-06
30	22.82	100	10	3	9.3132E-09
40	42.09	340	19	5	1.7280E-11
50	50.13	810	24	6	2.1316E-14
60	110	2,880	39	11	3.3827E-17
70	60	2,590	33	4	2.7952E-20
80	50	4,900	34	10	2.8124E-23
90	130	5,260	39	9	3.1503E-26
100	80	6,410	46	9	3.6287E-29
110	90	8,330	43	8	3.3125E-32
120	100	6,580	43	6	3.2349E-35
130	140	7,060	43	6	3.1591E-38
140	250	16,980	47	6	3.3720E-41
150	540	47,980	62	10	4.3440E-44
160	2,690	250,380	91	13	6.2264E-47
170	2,230	220,520	92	12	6.1473E-50
180	N/A	over 5 min	N/A	N/A	N/A
190	1,020	47,640	75	9	4.7792E-56
200	2,760	146,070	84	10	5.2273E-59

Table 6: the resulting setup time in ms and solve time in ms, the number of subtour contraints,rounds and latter as an absolute value and also as a percentage of the 2n possible constraints for given k using full ILP-formulation for Luxembourg TSP

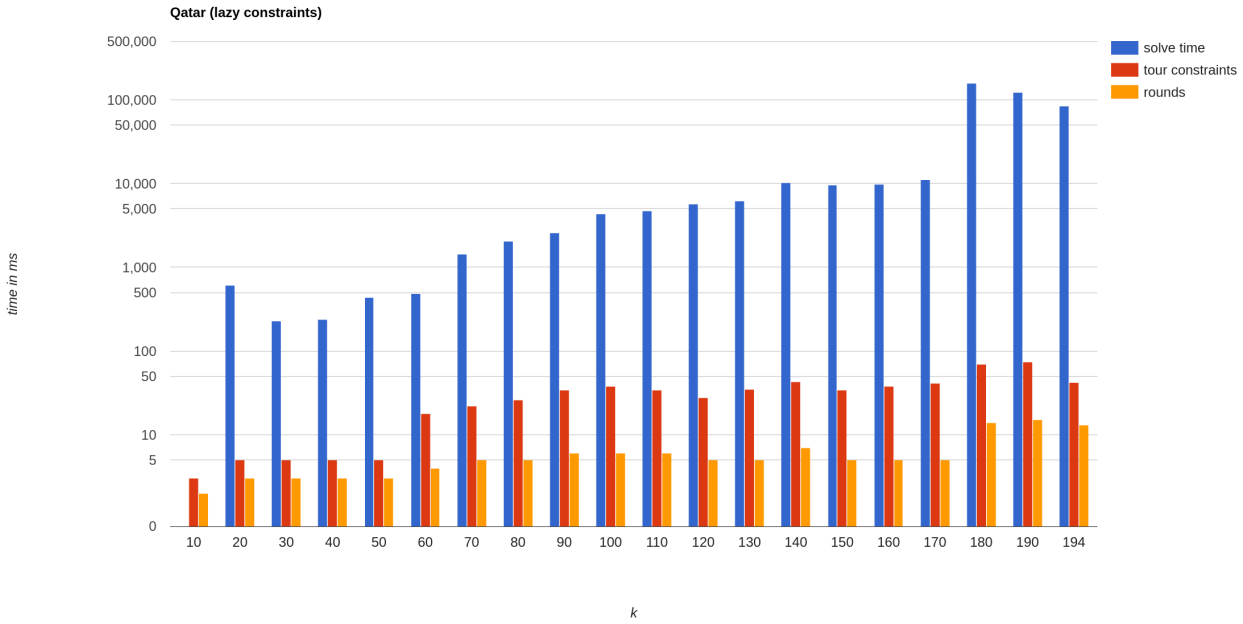

Figure 12: the solving time set in perspective to the number of subtour constarints and rounds for given k. not the logarithmic scale on the vertical axe. this graphical representation is corresponding to the Data from 6