

The Traveling Salesman Problem with Gurobi

Romeo Rizzi, Alice Raffaele

University of Verona

romeo.rizzi@univr.it, alice.raffaele@unitn.it

December 11, 2018

Overview

History

Tackling the TSP

Methods

Example

TSP Formulation

Mathematical Formulation

In-depth analysis: Separation and Optimization

Solving TSP with Gurobi

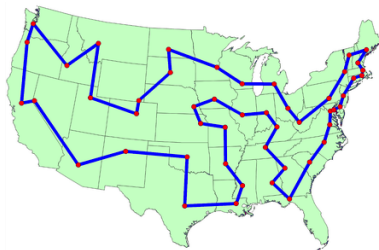
Gurobi Implementation Notes

Our TSP example with Gurobi

Conclusion

Appendix - ATSP

The Traveling Salesman Problem



"It is shown that a certain tour of 49 cities, one in each of the 48 states and Washington DC, has the shortest road distance"
(Dantzig, Fulkerson and Johnson, 1954)

- **Definition:** a salesman must visit n cities exactly once and must return to the original point of departure; the distance (or cost) between cities i and j is known and values c_{ij} . Find the shortest route computing only one minimum length cycle.

- Since every city must be visited, the solution route will be a cyclic permutation of all cities.
- Given n cities, the number of possible solutions are exactly $(n - 1)!/2$, if distances are symmetric → It is not so easy to check all possible solutions, even with a computer, especially for large instances:

No. cities	No. tours	Time
5	12	12 microsecs
8	2520	2.5 millisecs
10	181,440	0.18 secs
12	19,958,400	20 secs
15	87,178,291,200	12.1 hours
18	177,843,714,048,000	5.64 years
20	60,822,550,204,416,000	1927 years

- Karp proved in 1972 that the problem is NP-Hard; this dooms exact methods to take exponential time (only) in the worst case. There are many heuristics that quickly yield feasible solutions of reasonable quality. The metric case allows for approximation algorithms.

Some history

- 1766 - Euler studied the **Knight's tour** problem
- 1856 - Kirkman studied the graph of a **polyhedron**, wondering if there could be a circuit which passes through every vertex only once.
- 1856-1859 - Hamilton developed the **Icosian Game**:



- 1884 - Tait offers an elegant proof of the 4-colors Conjecture essentially assuming that Kirkman's conjecture is true for simple polytopes.

- 1930 - Menger was the first who talked about **Hamiltonian path** and studied the general formulation.
- 1946 - William Tutte's counterexample to Tait's (and Kirkman's) conjecture
- 1949 - The **problem name** *Traveling Salesman Problem* appeared for the first time in Julia Robinson's paper "*On the Hamiltonian Game (A Traveling Salesman Problem)*".
- 1954 - Dantzig, Fulkerson and Johnson formulated the **Symmetric TSP**, the first ILP formulation, and then also the **Asymmetric** version.

How to tackle the TSP

There are essentially three categories of algorithms for ILP:

- **Heuristics algorithms:**
 - Approaches guaranteed to yield some output within the allotted time. The quality of the solutions returned and the computational resources used are the main measures compared in the experimental comparisons in the literature, but also the time to design and implement the heuristic makes the difference in the real world.
- **Approximation algorithms:**
 - Polynomial time algorithms that are guaranteed to return feasible solutions scoring objective function values within proven bounds from the optimum.
- **Exact algorithms:**
 - They always return an optimum feasible solution (unless no feasible solution exists).

Heuristics algorithms

- Several heuristics have been developed during the years, among these:
 - **Constructive algorithms** (e.g., various greedy and GRASP approaches like farthest insertion, random insertion, nearest neighbor): these generate feasible solutions starting from the bare instance. Example of *Nearest neighbor*:
<https://www.youtube.com/watch?v=E85l6euMsd0>
 - **Local Search** (e.g., 2-opt, k-opt, Variable Neighborhood Search, Exponential Neighborhood Search). Example of *2-opt*:
<https://www.youtube.com/watch?v=3zSmjkpvcgw>)
 - **Meta-heuristics** (e.g., Simulated Annealing, Tabu Search, Genetic, Memetic and Ant Colony algorithms, etc.)

Approximation algorithms

- Given a minimum problem, let x^* and x^H be respectively the optimal solution and the solution obtained with the approximation algorithm.
- We can say that $\frac{x^H - x^*}{x^*} \leq \varepsilon \rightarrow x^H \leq (1 + \varepsilon)x^*$.
- ε represents the error.
- In problems like TSP, given an instance I (i.e., all distances between n nodes), poly-time algorithms exists to generate a feasible solution.

Exact algorithms

- In operations research, some main general frameworks to design exact algorithms to NP-hard problems are:
 - **Branch-and-Bound**
 - **Cutting Planes**
 - **Branch-and-Cut**
- Let's define some basic concepts before discussing Branch-and-Bound.

Lower Bound and Upper Bound

- **Lower Bound (LB):**

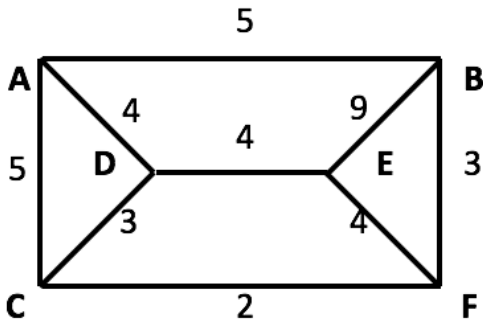
- Given an instance I , a lower bound to $val(x, I)$ is a real b such that $b \leq val(x, I)$ for every feasible solution x to I .
- Bound b_2 is tighter than bound b_1 when $b_1 \leq b_2$.
- A bound b is tight when there exists a feasible solution x such that $val(x, I) = b$.

- **Upper Bound (UB):**

- Given an instance I , an upper bound to $val(x, I)$ is a real B such that $B \geq val(opt(I), I)$.
- Bound B_1 is tighter than bound B_2 when $B_1 \leq B_2$.
- A bound b is tight when $B = val(opt(I), I)$.

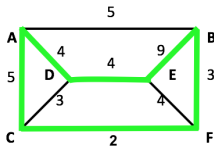
- By a lower bound (upper bound) for a certain problem we often mean a closed formula or a poly-time algorithm that, given I , yields a lower bound (upper bound) for I .
- We will see how lower bounds and upper bounds can both play helpful, especially the tighter they happen to be.
- Heuristics and approximation algorithms provide upper bounds.
- Approximation algorithms also provide lower bounds. Why?

TSP Example

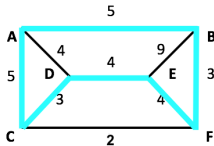


What is a LB for this simple graph? And an UB?

- This is a feasible solution of value 27:



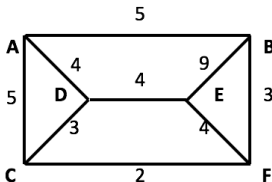
- It certifies that 27 is an upper bound on the minimum cost. Once we know it, we will never be willing to pay more than 27.
- The following solution of cost 24 provides a tighter (more interesting) upper bound:



- Thus for minimization problems, feasible solutions represent upper bounds.

Lower bound for the TSP

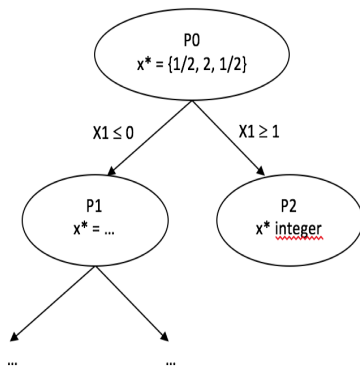
- The cost of any tour is $\frac{1}{2} \sum$ (sum of the costs of the two tour edges adjacent to v).
- The sum of the two tour edges adjacent to a given vertex $v \geq$ sum of the two edges of least cost adjacent to v .
- The cost of any tour is $\geq \frac{1}{2} \sum$ (sum of the costs of the two least cost edges adjacent to v).
- In our example, $LB = \frac{1}{2} [(4+5) + (3+5) + (2+3) + (3+4) + (4+4) + (2+3)] = \frac{1}{2} 42 = 21$.



- Let's introduce an exact method that exploits bounds...

Branch-and-Bound

- It essentially explores all the solutions but, to avoid seeing all of them (like a brute-force algorithm), it implicitly **enumerates** them exploring a tree and **branching** on possible elementary choices.
- It avoids exploring entire subtrees comparing **lower bounds** and **upper bounds**, pruning some branches.

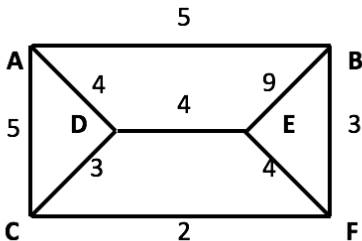


- At any node of the tree, the algorithm makes a finite decision and sets one of the unbound variables.
- A node (sub-problem) is *solved* when:
 - its optimal solution is found;
 - it admits no feasible solution;
 - it has become apparent that all possible solutions to the subproblem are worse than the best one found so far.

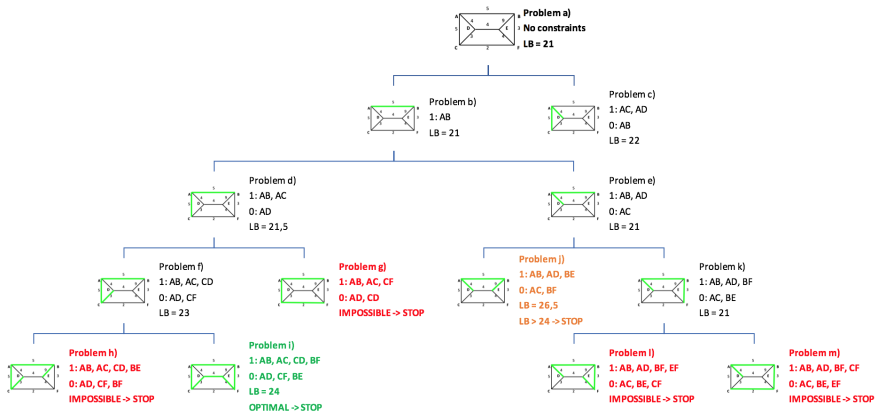
If it is not possible to find its optimal solution, then branch.

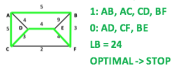
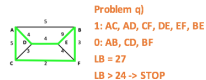
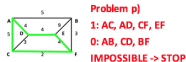
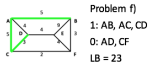
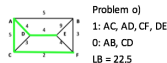
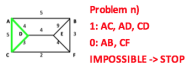
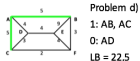
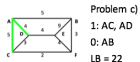
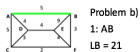
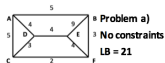
TSP Example with Branch-and-Bound

Let's try to solve this instance:

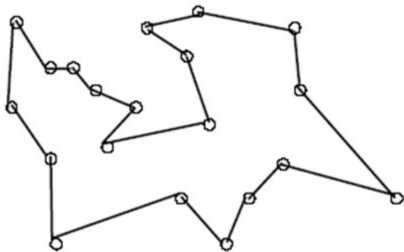


A tree search (Branch-and-Bound)





TSP Mathematical Formulation



- The distance between two vertices is the same in both directions (**undirected graph**).
- Let V be the set of vertices and E the set of edges.
- We introduce a binary variable x_e for every edge $e \in E$:
 $x = 1$ iff the edge e is included in the tour.
- The objective function is to minimize the total cost of edges selected, i.e. traveled by the salesman.

- For every vertex v , precisely two of the edges incident with v are selected.
- For any subset $S \subset V$, let $\delta(S)$ = be the edges with one end-vertex in S .
- We obtain the following initial formulation, where constraints (2), which are called *assignment constraints*, ensure that every vertex is visited exactly twice:

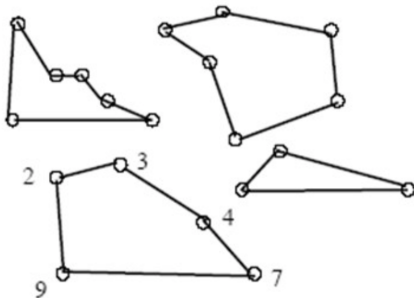
$$\min \quad \sum_{e \in E} c_e x_e \quad (1)$$

$$\text{subject to} \quad \sum_{e \in \delta(v)} x_e = 2, \quad \forall v \in V \quad (2)$$

$$x_e \in \{0, 1\}. \quad (3)$$

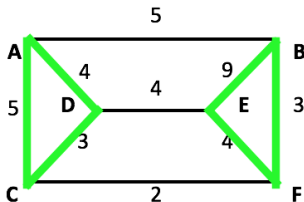
...but it's not enough!

A solution like the following in the picture would be feasible according to the above formulation:



But we do not want our salesman to "teleport" or "jump" from one city to another → Our formulation is hence incomplete.

In fact, in the example before, we could have obtained this:



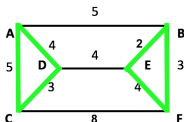
This solution can be used as a lower bound for the problem.
But, in order to formulate the correct model for TSP, we need to add more constraints...

AMPL Example

Consider the same example but change costs of (B,E) from 9 to 2 and (C,F) from 2 to 8:

Model:

```
set V ordered;
set E within {i in V, j in V: ord(i) < ord(j)};
param cost{E};
var X{E} binary;
minimize TourCost: sum {(i,j) in E} cost[i,j] * X[i,j];
subject to VisitAllVertices {i in V}:
sum {(i,j) in E} X[i,j] + sum {(j,i) in E} X[j,i] = 2;
```



Data:

```
set V := A B C D E F;
set E := (A,B) (A,C) (A,D)
         (B,E) (B,F) (C,D)
         (C,F) (D,E) (E,F);
param cost :=
[A,B] 5
[A,C] 5
[A,D] 4
[B,E] 2
[B,F] 3
[C,D] 3
[C,F] 8
[D,E] 4
[E,F] 4;
```

The AMPL solver will find the optimal solution with value 21, choosing edges (A,C), (A,D), (B,E), (B,F), (C,D), (E,F). But it is not what we want...

Subtour Elimination Constraints

- TSP asks for an **Hamiltonian circuit**: a circuit passing through each vertex exactly once, without subtours.
- Given $S \subsetneq V$, $S \neq \emptyset$, the cut with shores S and V/S , denoted $\delta(S)$, is the set of those edges with one endpoint in S and the other in V/S .
- We must impose that the solution is **connected**. This means that, for every proper subset of vertices, at least two selected edges have one endpoint in S and the other outside S :

$$\sum_{e \in \delta(S)} x_e \geq 2, \forall S \subset V : 2 \leq |S| \leq |V| - 2$$

- Violated constraints will have the following formulation:

$$\sum_{e \in \delta(S)} x_e^* < 2.$$

- These are called *Subtour Elimination Constraints* (SEC).

The SEC problem

- Introducing this family of constraints, we are adding an **exponential** number of constraints, since we are considering the power set of V (except the empty set, sets composed of single vertices and the set with all vertices).
- This approach is not polynomial for a double reason:
 - we adopted an ILP model;
 - its formulation has an exponential number of variables and constraints.

TSP Formulation

- Putting together the objective function (1) and constraints (2), (3) and (4) , we get to the correct formulation:

$$\min \sum_{e \in E} c_e x_e \quad (1)$$

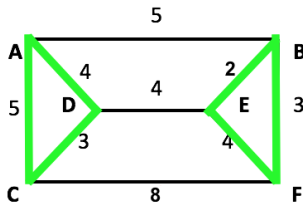
$$\text{subject to} \quad \sum_{e \in \delta(v)} x_e = 2, \quad \forall v \in V \quad (2)$$

$$\sum_{e \in \delta(S)} x_e \geq 2, \quad \forall S \subset V : 2 \leq |S| \leq |V| - 2 \quad (3)$$

$$x_e \in \{0, 1\}. \quad (4)$$

Another approach considering flow constraints

- Let's think about another approach that does not imply adding an exponential number of constraints.
- Consider the wrong solution with subtours that we obtained before with AMPL:



- Suppose we want to send some quantity of flow from vertex A to another vertex:
 - since A is connected to C and D , there is no problem to send it to them;
 - instead, if we want to send the flow to F , we cannot.

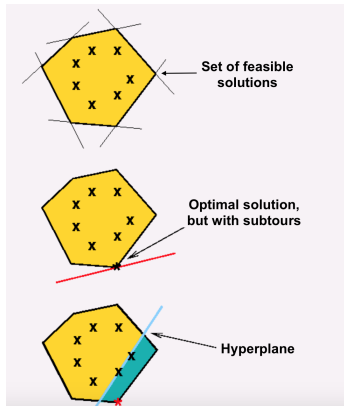
Adding flow constraints

- We introduce two variables $y_{(u,v)}$ and $y_{(v,u)}$ for every edge $e = (u, v) \in E$ and we allow to transmit the flow only through edges selected in the cycle:
 - $y_{(u,v)} \leq x_{(u,v)}$
 - $y_{(v,u)} \leq x_{(u,v)}$.
- We consider the vertices A and F respectively as the source and destination; we want that the outgoing flow from A is equal to 1, whereas the incoming flow is 0:
 - $y(\delta^+(A)) = 1$
 - $y(\delta^-(A)) = 0$.
- We want all other vertices but F to conserve the flow:
 - $y(\delta^-(u)) = y(\delta^+(u)), \forall u \neq A, F$

- In this way, we are forcing the flow to arrive exactly in the only vertex left, i.e. F , connecting A and F .
- This could be repeated for every source vertex and for every destination vertex.
- This, since the number of vertices is n , does not seem exponential at all, but we are adding a polynomial number of constraints and variables.

The Separation problem

- Given an instance of an integer programming problem and a point x , determine whether x is in the convex hull of feasible integral points. Further, if it is not in the convex hull, find a separating hyperplane that cuts off x from the convex hull.
- The algorithm that solves the Separation problem is called **Separation oracle**.



The Separation Oracle



- We need an algorithm that takes the solution as input and either certifies if it is feasible (in this case, it means that it does not contain subtours) or it gives as output the hyperplane corresponding to the cut.

What is separation here?

- Writing the connectivity constraints, we see that it can be seen as a **minimum cut problem**.

Minimum Cut

- Let's consider x^* as our current solution.
- We can use the **support graph** G^* : the graph with $V^* = V$ and $E^* = \{e \in E : x_e^* > 0\}$.
- We give to each $e \in E^*$ the weight x_e^* .
- A violated SEC exists if and only if there is a cut in G^* whose x^* -weight is less than 2.
- We can use an algorithm such as Karger's or we can take advantage of the **Max-flow Min-Cut theorem**.

Max Flow

- $\sum_{e \in \delta(S)} x_e$ is precisely the value of a maximum flow from a node in S to a node in $V \setminus S$.
- We want to determine the max flow that can be sent from a source node $s \in S$ to a terminal node $t \in V \setminus S$.
- We repeat this for each vertex as node s . If we find a flow smaller than 2, we have found a subtour and which constraint is violated, which has to be added to the formulation.
- A famous greedy algorithm that solves Max-Flow problems is **Ford-Fulkerson**, which is polynomial.

Separation = Optimization

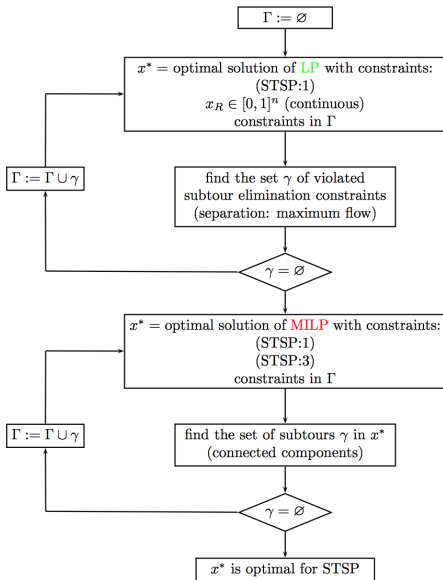
Theorem

Given a linear program, if we can solve the separation problem in polynomial time, then we can solve the optimization problem in polynomial time too, using the Ellipsoid method (1976).

This is exactly what we are doing here, using Max-flow Min-Cut theorem and a polynomial algorithm like Ford-Fulkerson's to solve the separation problem.

Possible procedure

- We can remove the $O(2^n)$ connectivity constraints and add only the ones violated by the current solution → We need to check if the graph has **only one connected component**.
- Possible procedure, applying Branch-and-Bound:
 1. Relax integrality constraints and remove SECs.
 2. At every iteration, solve a linear programming problem and obtain the current solution (and a LB for the integer problem).
 3. If the solution does not contain subtours (i.e., if it is composed of only one connected component):
 - 3.1 If the solution is integer, end;
 - 3.2 Otherwise, branch on a fractional variable.
 4. Else, if the solution contains subtours, solve the **Separation problem**, add the violated constraints to the problem formulation and go back to Step 2.



Solving TSP with Gurobi



GUROBI
OPTIMIZATION

- We can use another easy approach in Gurobi too.
- We let Gurobi solve the problem and, when it finds a new MIP solution, we look for **connected components**:
 - We will compute the shortest (sub)tour of vertices that the solution visits;
 - If the number of vertices in the tour is equal to the total number of vertices in V , the solution is fine; otherwise it means that there are subtours.
- To do so, we need to use **Callbacks** and **Lazy Constraints**.

Callbacks

- Callbacks are functions that can be defined by the user to perform some custom actions automatically in particular cases, for example:
 - During presolve;
 - When a new MIP incumbent solution is found;
 - When printing a log message;
- The callback function must be specified and passed as parameter to the optimize method:

model.optimize(callbackname)

- The callback routines use mainly the **where** argument: it indicates in which state the Gurobi optimizer is (*presolve*, *simplex*, *MIP*, etc.); for each case, develop appropriate code.
- In our case, we want to check subtours everytime a new MIP solution is found: the related value of *where* is 4.

Lazy Constraints

- Using callbacks, during runtime execution we can add two types of custom constraints:
 - **Cutting Planes:**
 - They do not cut off any integer feasible solution, just strengthen the continuous relaxation to speed-up the process.
 - **Lazy Constraints:**
 - They affect only MIP models, cutting off integer solutions that are feasible for the remaining constraint system.
 - They are necessary for the correctness of the model, but added dynamically because usually the general form could be composed of exponential constraints.
 - To use them, the parameter LazyConstraints must be set to 1:
`model.params.LazyConstraints = 1`

Gurobi TSP Example

- You can find a TSP example on Gurobi website (<http://examples.gurobi.com/traveling-salesman-problem/>), that solves the problem considering n random points in the US, minimizing their Eulerian distances:

The screenshot shows the Gurobi website's page for the Traveling Salesman Problem. The header includes the Gurobi logo and navigation links: Home, Interactive Examples, Travelling Salesman Problems, PRODUCTS, DOWNLOADS, RESOURCES, ACADEMIA, SUPPORT, and ABOUT. A search bar and a 'Get Gurobi' button are also present. The main content area features a sidebar with links to Intro, Problem, Model, Implementation, Live Demo, and Try Gurobi for Free. The main heading is 'The Traveling Salesman Problem with integer programming and Gurobi'. Below this, the text explains that the example solves the TSP by constructing a mathematical model in Gurobi's Python interface. A map of the United States is shown with several black dots representing random points and a purple line connecting them in a path. At the bottom right, there are navigation icons for a presentation slide.

GUROBI OPTIMIZATION

Home Interactive Examples Travelling Salesman Problems

Intro
Problem
Model
Implementation
Live Demo
Try Gurobi for Free

The Traveling Salesman Problem

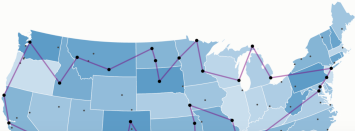
with integer programming and Gurobi

In this example we'll solve the Traveling Salesman Problem.

We'll construct a mathematical model of the problem, implement this model in Gurobi's Python interface, and compute and visualize an optimal solution.

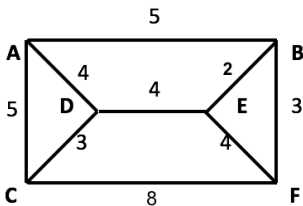
Although your own business may not involve traveling salesmen, the same basic techniques used in this example can be used for many other applications like vehicle routing, circuit design and DNA sequencing.

Click the screenshot to skip directly to the Live Demo!



Our TSP Example with Gurobi

- Let's consider again our graph, with the following edges and costs:



- It is the same instance we tried to solve before with AMPL, where edges (B,E) and (CF) cost respectively 2 and 8.

Sets definition

1. Firstable, import required libraries:

```
9 import math
10 import random
11 from gurobipy import *
```

2. Define the sets of vertices and edges:

```
66 # -----
67 # MODEL DEFINITION
68
69 # Create vertices and edges for our example
70 vertices = [0,1,2,3,4,5] #vertices = ['A', 'B', 'C', 'D', 'E', 'F']
71 # We do not use letters just to avoid issues with indexing and syntax errors
72 edges = {
73     (0,1): 5,
74     (0,2): 5,
75     (0,3): 4,
76     (1,4): 2, #9
77     (1,5): 3,
78     (2,3): 3,
79     (2,5): 8,
80     (3,4): 4,
81     (4,5): 4}
82
83 n = len(vertices)
```

Model creation

3. Create the model and add a variable for each edge.
4. Set the objective function.
5. Add degree constraints.

```
85 # Create the model
86 m = Model()
87
88 # Create the variables
89 vars = {}
90 for i,j in edges.keys():
91     vars[i,j] = m.addVar(obj=edges[i,j], vtype=GRB.BINARY,
92                           name='e[%d,%d]'%(i,j))
93 for i,j in vars.keys():
94     vars[j,i] = vars[i,j] # Edges in both directions
95
96 # To create the model data structure only once, after variables creation
97 m.update()
98
99 # Add the objective function
100 m.setObjective(sum(vars[i,j]*edges[i,j]
101                    for (i,j) in edges.keys()),GRB.MINIMIZE)
102
103 # Add degree-2 constraint
104 for i in vertices:
105     m.addConstr(sum(vars[i,j] for j in vertices
106                     if (i,j) in edges.keys() or (j,i) in edges.keys()) == 2)
```

A useful function: Subtour

6. We implement the method that, given a set of edges as parameter, looks for the shortest subtour:

```
41 # Given a list of edges, this method finds the shortest subtour
42 def subtour(edges):
43     visited = [False]*n
44     cycles = []
45     lengths = []
46     selected = [[] for i in vertices]
47     for x,y in edges:
48         selected[x].append(y)
49         selected[y].append(x) # Edges in both directions
50     while True:
51         current = visited.index(False)
52         thiscycle = [current]
53         while True:
54             visited[current] = True
55             neighbors = [x for x in selected[current] if not visited[x]]
56             if len(neighbors) == 0:
57                 break
58             current = neighbors[0]
59             thiscycle.append(current)
60         cycles.append(thiscycle)
61         lengths.append(len(thiscycle))
62         if sum(lengths) == n:
63             break
64     return cycles[lengths.index(min(lengths))]
```

Note: this code has to be inserted at the beginning, before sets and model definition, so we can call it in the rows below.

Model optimization - Part I

7. We start solving the problem without adding Subtour Elimination Constraints:

```
108 # -----
109 # MODEL OPTIMIZATION
110
111 # Without SECs
112 print '\n-----\n'
113 print 'Optimize model without SECs\n'
114 m._vars = vars
115 m.optimize()
116 # Print optimal solution
117 print '\n-----\n'
118 solution = m.getAttr('x', vars)
119 selected = [(i,j) for i,j in edges.keys() if solution[i,j] > 0.5]
120 print 'Edges in solution: ' + str(selected)
121 print('Optimal tour: %s' % str(subtour(selected)))
122 print('Optimal cost: %g' % m.objVal)
123
```

8. Launching the execution, the console will output this:

Optimize model without SECs

Optimize a model with 6 rows, 9 columns and 18 nonzeros

Variable types: 0 continuous, 9 integer (9 binary)

Coefficient statistics:

Matrix range [1e+00, 1e+00]

Objective range [2e+00, 8e+00]

Bounds range [1e+00, 1e+00]

RHS range [2e+00, 2e+00]

Presolve removed 3 rows and 3 columns

Presolve time: 0.00s

Presolved: 3 rows, 6 columns, 9 nonzeros

Variable types: 0 continuous, 6 integer (6 binary)

Found heuristic solution: objective 26.0000000

Root relaxation: objective 2.100000e+01, 3 iterations, 0.00 seconds

Nodes			Current Node			Objective Bounds			Work	
Expl	Unexpl		Obj	Depth	IntInf	Incumbent	BestBd	Gap	It/Node	Time
*	0	0			0	21.0000000	21.00000	0.00%	-	0s

Explored 0 nodes (3 simplex iterations) in 0.05 seconds

Thread count was 4 (of 4 available processors)

Solution count 2: 21 26

Optimal solution found (tolerance 1.00e-04)

Best objective 2.100000000000e+01, best bound 2.100000000000e+01, gap 0.0000%

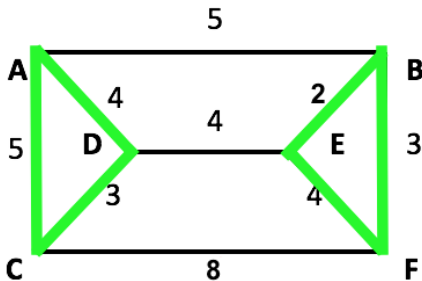
Edges in solution: [(1, 5), (2, 3), (4, 5), (0, 3), (0, 2), (1, 4)]

Optimal tour: [0, 3, 2]

Optimal cost: 21

Note: the optimal subtour visits just three vertices!

9. We do not like the solution we obtained:



Let's apply the approach we discussed before, with lazy constraints...

Model optimization - Part II

10. We define the function which Gurobi will call automatically when it finds a new MIP incumbent solution:

```
16 # Callback: use lazy constraints to eliminate subtours
17 def subtourelim(model, where):
18     if where == GRB.callback.MIPSOL:
19         print '\nNew solution found: checking the presence of subtours...'
20         selected = []
21         # Make a list of edges selected in the solution
22         vals = model.cbGetSolution(model._vars)
23         selected = tuplelist((i,j) for i,j in model._vars.keys()
24                               if vals[i,j] > 0.5)
25
26         # Find the shortest cycle in the selected edge list
27         tour = subtour(selected)
28         if len(tour) < n:
29             # Add a subtour elimination constraint
30             print 'One subtour found: ' + str(tour)
31             expr = 0
32             for i in range(len(tour)):
33                 for j in range(i+1, len(tour)):
34                     expr += model._vars[tour[i], tour[j]]
35             print 'Subtour Elimination Constraint added:'
36             print str(expr) + ' <= ' + str(len(tour)-1)
37             model.cbLazy(expr <= len(tour)-1)
38         else:
39             print 'No subtour found!'
40
```

11. We tell Gurobi to use it during optimization, after setting the `LazyConstraints` parameter to 1 and after bringing the model back to an unsolved state with `reset()`:

```
124 # With Lazy Constraints
125 print '\n-----\n'
126 print 'Adding Lazy Constraints\n'
127 m.reset()
128 m.params.LazyConstraints = 1
129 m.optimize(subtourelim)
130
131 # Print optimal solution
132 print '\n-----\n'
133 solution = m.getAttr('x', vars)
134 selected = [(i,j) for i,j in edges.keys() if solution[i,j] > 0.5]
135 print 'Edges in solution: ' + str(selected)
136 print('Optimal tour: %s' % str(subtour(selected)))
137 print('Optimal cost: %g' % m.objVal)
138
139 # Check if the solution has only one connected component
140 assert len(subtour(selected)) == n
```

If you prefer, you can also define a **printSolution** function.

12. This is the output using Lazy Constraints:

Adding Lazy Constraints

Changed value of parameter LazyConstraints to 1

Prev: 0 Min: 0 Max: 1 Default: 0

Optimize a model with 6 rows, 9 columns and 18 nonzeros

Variable types: 0 continuous, 9 integer (9 binary)

Coefficient statistics:

Matrix range [1e+00, 1e+00]

Objective range [2e+00, 8e+00]

Bounds range [1e+00, 1e+00]

RHS range [2e+00, 2e+00]

Presolve removed 3 rows and 3 columns

Presolve time: 0.00s

Presolved: 3 rows, 6 columns, 9 nonzeros

Variable types: 0 continuous, 6 integer (6 binary)

New solution found: checking the presence of subtours...

No subtour found!

Found heuristic solution: objective 26.000000

Root relaxation: objective 2.100000e+01, 3 iterations, 0.00 seconds

New solution found: checking the presence of subtours...

One subtour found: [0, 3, 2]

Subtour Elimination Constraint added:

<gurobi.LinExpr: e[0,3] + e[0,2] + e[2,3]> <= 2

When Gurobi finds the solution with value 21, it uses the callback and finds a subtour composed of vertices 0, 2 and 3
→ It adds to the current formulation $e_{0,3} + e_{0,2} + e_{2,3} \leq 2$.

13. Here is the right optimal solution:

New solution found: checking the presence of subtours...
No subtour found!

Nodes			Current Node			Objective Bounds			Work	
Expl	Unexpl		Obj	Depth	IntInf	Incumbent	BestBd	Gap	It/Node	Time
*	0	0			0	24.0000000	24.00000	0.00%	-	0s

Cutting planes:
Lazy constraints: 1

Explored 0 nodes (4 simplex iterations) in 0.07 seconds
Thread count was 4 (of 4 available processors)

Solution count 2: 24 26

Optimal solution found (tolerance 1.00e-04)
Best objective 2.400000000000e+01, best bound 2.400000000000e+01, gap 0.0000%

Edges in solution: [(0, 1), (1, 5), (2, 3), (4, 5), (3, 4), (0, 2)]
Optimal tour: [0, 1, 5, 4, 3, 2]
Optimal cost: 24

Note: now the optimal tour visits all vertices.

Conclusion

- On Gurobi website you can find a lot of examples implemented, also a TSP with random points.
- We defined the TSP, using one easy formulation and one more complete and beautiful but, unfortunately, with an exponential number of constraints.
- We saw several methods to tackle a ILP problem (heuristics, approximation and exact algorithms), according to the purpose.
- We solved an easy instance of TSP using Branch-and-Bound and then exploiting Gurobi and Lazy Constraints.
- In Appendix you can find the description of Asymmetric TSP.

References



University of Waterloo, *TSP*

<http://www.math.uwaterloo.ca/tsp/>



R.Mansini, *Algoritmi di Ottimizzazione*, University of Brescia

<https://www.unibs.it/ugov/degreecourse/65037>



Wikipedia, *Traveling Salesman Problem*

https://en.wikipedia.org/wiki/Travelling_salesman_problem



Gilbert Laporte, *A Short History of the Traveling Salesman Problem*

<http://neumann.hec.ca/chairedistributique/common/laporte-short.pdf>



Springer, *Encyclopedia of Optimization*

<http://www.springer.com/it/book/9780387747583>



Gurobi Optimization and Official Documentation

<http://www.gurobi.com>

<http://www.gurobi.com/documentation/>



Gurobi Quick Start Guide

https://www.gurobi.com/documentation/7.5/quickstart_linux.pdf



Gurobi Python Implementation of TSP with random points

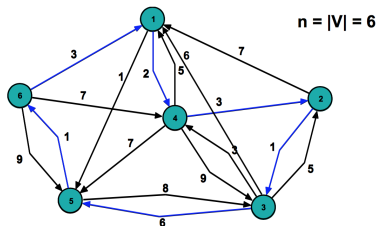
http://www.gurobi.com/documentation/7.5/examples/tsp_py.html



Operations Research Group, *Asymmetric TSP*, University of Bologna

http://www.or.deis.unibo.it/algottm/files/8_ATSP.pdf

Appendix - The Asymmetric TSP



- There can be cases where going from vertex i to vertex j does not cost the same then going from vertex j to vertex i or maybe it would not be possible to go through both directions.
- We need to use a set of arcs A , instead of the set of edges E (**directed graph**): this variant is called **asymmetric TSP**.
- Let V be the set of vertices.
- We introduce a binary variable $x_{i,j}$ for every arc $(i,j) \in A$: if $x_{i,j} = 1$, then arc (i,j) appears on the tour.

ATSP Formulation

- Keeping in mind what we said about symmetric TSP, we can formulate the asymmetric TSP as follows:

$$\min \sum_{(i,j) \in A} c_{i,j} x_{i,j} \quad (1)$$

$$\text{subject to } \sum_{j \neq i} x_{i,j} = 1, \forall i \in V \quad (2)$$

$$\sum_{i \neq j} x_{i,j} = 1, \forall j \in V \quad (3)$$

$$\sum_{i \in S, j \in S} x_{i,j} \leq |S| - 1, \forall S \subset V, S \neq \emptyset \quad (4)$$

$$x_{i,j} \in \{0, 1\}. \quad (5)$$

- With constraints (2) and (3), we are distinguishing arcs that enter in a certain vertex and arcs that exit from it.