The Travelling Salesman Problem

Alberto Santini

Fall 2024

Many examples in these lecture notes are adapted from popular books:

- Alexander Schrijver (1998). *Theory of linear and integer programming*. Wiley. ISBN: 0-471-98232-6.
- Vašek Chvátal (1983). *Linear Programming*. W.H. Freeman and Company. ISBN: 0-716-71195-8.
- Laurence Wolsey (2020). *Integer Programming*. 2nd Edition. Wiley. ISBN: 978-1-119-60653-6.
- Silvano Martello and Paolo Toth (1990). *Knapsack Problems: algorithms and computer implementations*. Wiley. ISBN: 978-0-471-92420-3.

1 Introduction

Consider the problem of a logistic operator who must deliver parcels to a set of customers. The operator owns a vehicle that starts from a warehouse full of parcels, visits customers in a given order, and finally goes back empty to the warehouse. The objective of the **Travelling Salesman Problem** is to determine the correct order in which to visit customers so as to optimise a certain objective. The objective is usually the cost of operating the vehicle, which, in turn, is assumed to be equivalent to minimising the total distance travelled.

This problem takes the name of Travelling Salesman Problem (**TSP**) because, when it was first introduced, it was described in the following terms. Imagine being a salesman who starts from his hometown, visits a set of other cities, and finally goes back home. In which order should you visit the other cities to minimise the total distance you travel? As you can see, this is just a different wording for the same optimisation problem.

Let us introduce some notation that will help us describe this problem more precisely. We denote with $V = \{1, \dots, n\}$ the set of customers that must be visited and with 0 the vehicle depot. Therefore, the vehicle starts its tour at 0, visits each customer in V once, and goes back to 0. We denote with $V' = \{0\} \cup V$ the set of all relevant locations (the depot's and the customers'). Figure 1 shows an example of a TSP instance with a depot (in yellow) and seven customers (in pink). In this case, we represent the customers as $V = \{1, 2, 3, 4, 5, 6, 7\}$ and the depot as 0. The set of all locations is $V' = \{0, 1, 2, 3, 4, 5, 6, 7\}$.

Remember that a *permutation* of V is a reordering of its elements. Formally, a permutation is a bijective function $\pi : V \to V$ that specifies a new order of the elements of V.[1] We can use such a permutation to specify in which order the vehicle should visit the customers. For example, imagine that $V = \{1, 2, 3, 4\}$ and we want the vehicle to visit the customers in the order $3, 2, 4, 1$.

[1] Remember that π being bijective means that it is defined for all elements of V and that for each $j \in V$, there is exactly one $i \in V$ such that $\pi(i) = j$.

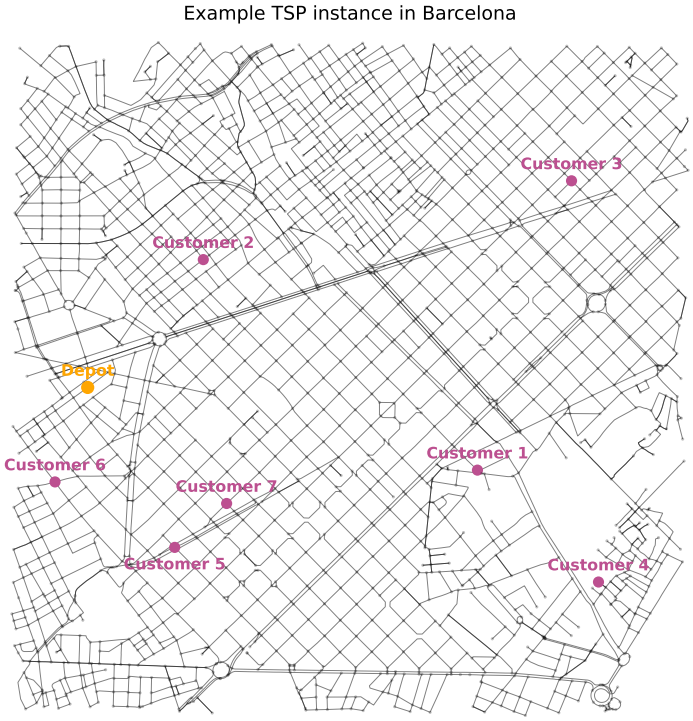


Figure 1: Example of a TSP instance with seven customers ($n = 7$). The depot is yellow, and the customers are pink. In this case, $V = \{1, \ldots, 7\}$ and $V' = \{0, 1, \ldots, 7\}$.

Then, the permutation we are looking for is:

$\pi(1) = 3$	Which reads as "The first customer is 3".
$\pi(2) = 2$	Which reads as "The second customer is 2".
$\pi(3) = 4$	Which reads as "The third customer is 4".
$\pi(4) = 1$	Which reads as "The fourth customer is 1".

Given a permutation π, we visit the customers in the order $\pi(1), \pi(2), \ldots, \pi(n)$. Because the vehicle starts and ends at the depot, the complete tour will be:

$$0, \pi(1), \pi(2), \ldots, \pi(n-1), \pi(n), 0.$$

As you can see, each permutation describes exactly one possible tour of the vehicle, and conversely, each possible tour corresponds to a permutation. A possible approach to the TSP, then, is to list all possible permutations, check what the cost (or travel distance) of each corresponding tour is, and select the one with the lowest cost. In other words, we are completely enumerating all possible valid tours that our vehicle can make. This approach seems viable until one starts counting just how many permutations we should check. You might remember that there are $n!$ (n factorial, i.e., $1 \cdot 2 \cdot \ldots \cdot n$) possible permutations of n elements. For the instance shown in Figure 1, where $n = 7$, there are $n! = 5040$ possible permutations and, therefore, possible tours. Although it would be very hard to check all of them by hand, we can imagine that a computer could do it in a fraction of a second. Consider, however, a slightly larger instance. For example, one with 15 customers. In this case, there would be $15! \approx 1.31 \cdot 10^{12}$ possible tours: that's in the order of trillions of tours. A typical delivery van, however, might deliver upward of thirty parcels in a single shift. If we wanted to enumerate all possible tours with $n = 30$, we would

CC BY-NC-SA 4.0

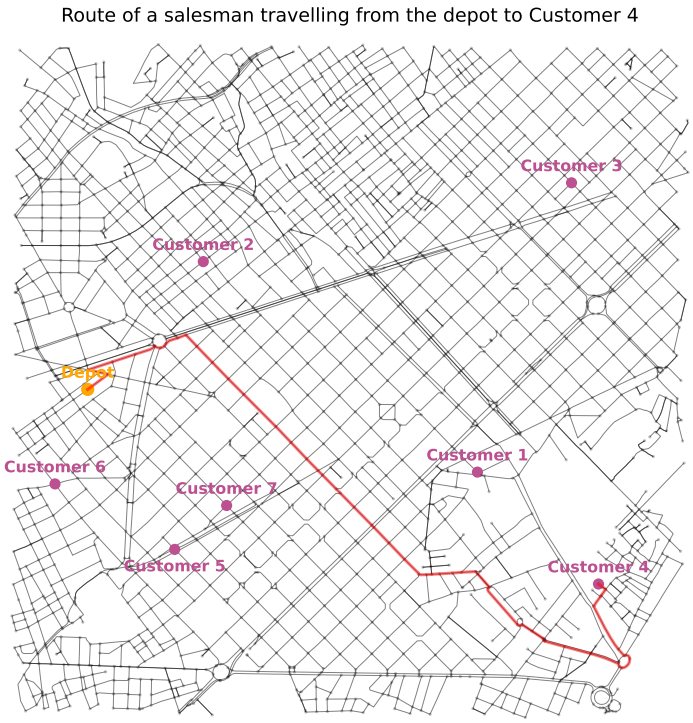


Figure 2: A vehicle following the red route will go directly from the depot (location 0) to Customer 4. The corresponding cost c_{04} will be the length of the red route.

need to consider approximately $2.65 \cdot 10^{32}$ of them. If you had access to one billion computers, and if each computer could check the cost of one billion routes per second, it would still take more than 8 million years to check all of them.

It is clear, then, that a brute-force approach will not lead us anywhere when dealing with real-life-size problems, and we need to call mathematical modelling techniques to the rescue. Before doing so, let us complete the notation by giving a precise meaning to our objective of finding the tour with the lowest cost. To this end, let us denote with $c_{ij} \geq 0$ the cost of travelling directly from location $i \in V'$ to location $j \in V'$ $(i \neq j)$ without intermediate stops. Practically speaking, it is often hard to estimate the monetary cost of travelling between two addresses. In most cases, we will let c_{ij} be the distance of a direct trip from i to j, and we will assume that the shorter the distance, the lower the cost. For example, Figure 2 shows the route taken by a vehicle travelling directly from the depot to Customer 4. We will use the length of the red route as the travel cost c_{04}. Note that, sometimes, travel costs are not symmetric: travelling from i to j might require a longer or shorter trip than travelling from j to i. This difference is especially relevant in urban settings where, due to one-ways and other urban infrastructure, the difference between c_{ij} and c_{ji} can be relevant. The total cost of a tour will be the sum of the costs of each leg composing the tour. For example, if $V = \{1, 2, 3, 4\}$ and the vehicle's tour is $0, 3, 2, 4, 1, 0$ then the total cost is $c_{03} + c_{32} + c_{24} + c_{41} + c_{10}$.

2 Mathematical Formulation

To introduce a mathematical formulation for the TSP, we consider variables $x_{ij} \in \{0, 1\}$ for each pair of locations $i, j \in V'$ with $i \neq j$. A variable x_{ij} will take the value one if and only if, in its

CC BY-NC-SA 4.0

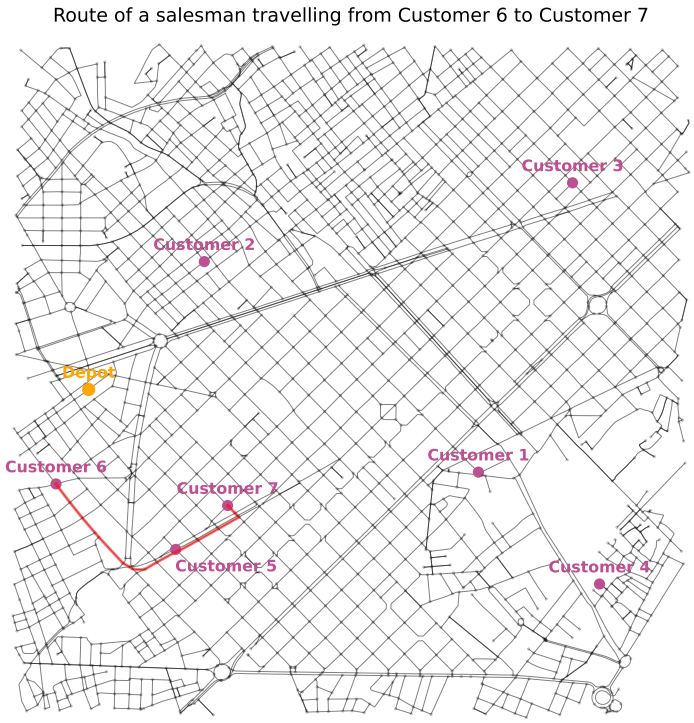


Figure 3: The most direct trip from Customer 6 to Customer 7 passes "through" Customer 5.

tour, the vehicle goes from location i to location j directly (without intermediate stops). For example, if $V = \{1, 2, 3, 4\}$, we can encode the tour $0, 3, 2, 4, 1, 0$ using the x variables as follows:

$$x_{03} = 1 \qquad x_{41} = 1$$
$$x_{32} = 1 \qquad x_{10} = 1$$
$$x_{24} = 1 \qquad \text{all other } x_{ij} = 0.$$

It is important to specify that x_{ij} corresponds to the vehicle travelling from i to j *without intermediate stops*. The reason is that, sometimes, the most direct trip between two customers passes through the location of a third one. For example, Figure 3 shows that going from Customer 6 to Customer 7, one must pass "through" Customer 5, i.e., the shortest path from Customer 6 to 7 makes the vehicle pass just in front of Customer 5's location. In the TSP, it seems obvious that, in such a case, the vehicle should stop and deliver Customer 5's parcel before proceeding to Customer 7. In other variants of the problem, however, we will show that this is not always possible (for example, if customers are at home during specific hours and the vehicle passes in front of Customer 5's house when there is no one to receive the parcel). Therefore, we must use the x variables to distinguish between the two situations, stopping or not at Customer 5's en route between Customer 6 and 7. This is the reason why we add the "without intermediate stops" specification to our interpretation of the x variables. If the truck travels from Customer 6 to 7 without stopping at Customer 5, the corresponding solution will have $x_{67} = 1$ but $x_{65} = x_{57} = 0$. On the other hand, if the truck stops at Customer 5, we will have $x_{65} = x_{57} = 1$ but $x_{67} = 0$.

We will now attempt to write an Integer Programme (IP) that solves the TSP using the x variables. Recall that our solver knows nothing about vehicles, depots and customers; it only sees a list of binary variables and is unaware of the meaning that we attach to them. Therefore,

CC BY-NC-SA 4.0

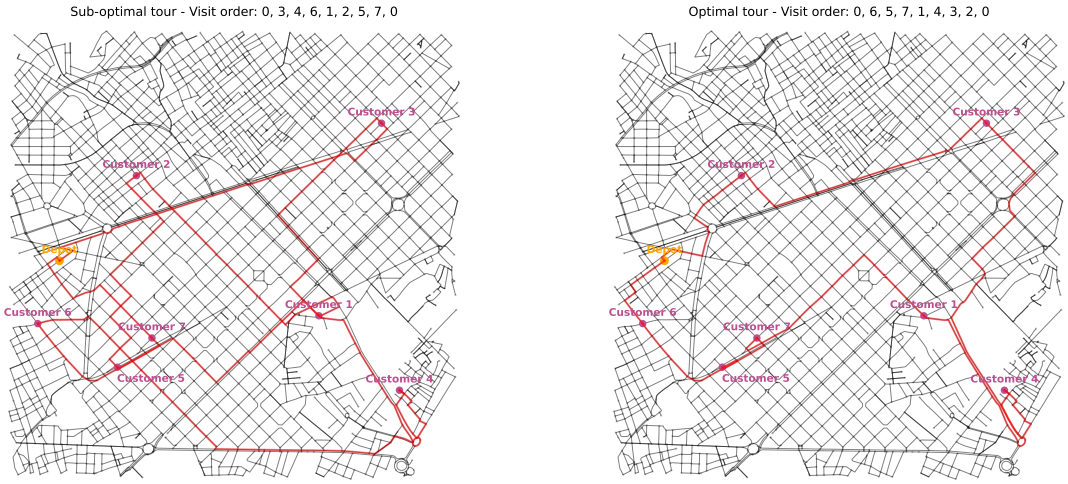


Figure 4: Two possible tours that start and end at the depot, and visit each customer once. The one on the left is sub-optimal, while the one on the right is the optimal one.

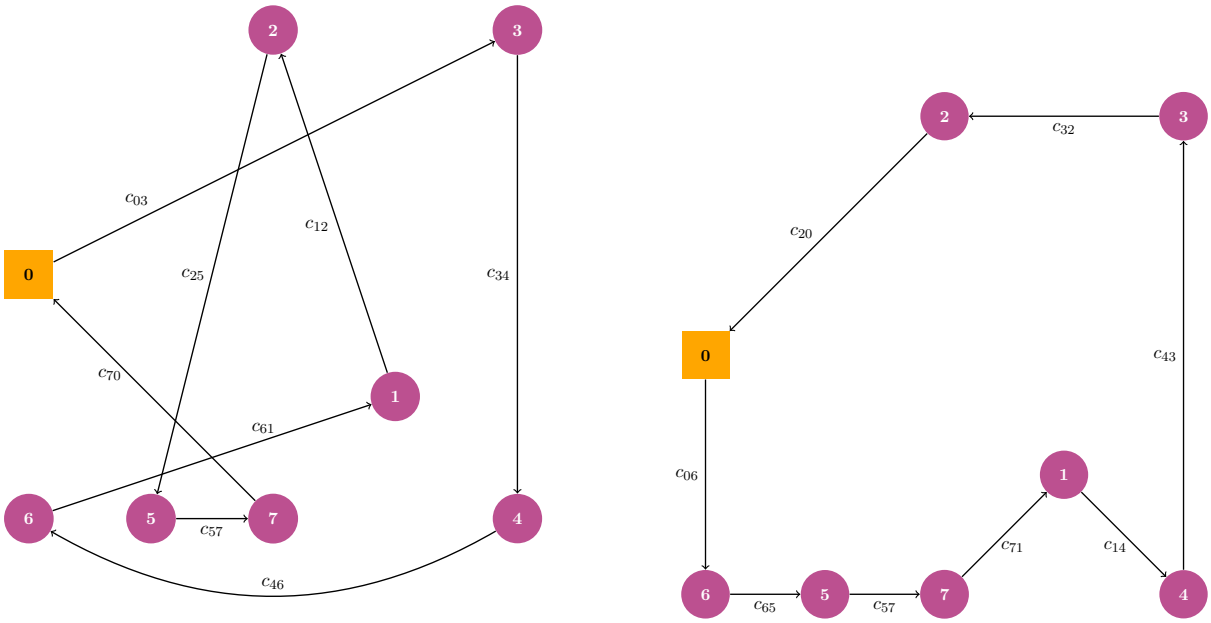


Figure 5: Schematic representation of the solutions depicted in Figure 4.

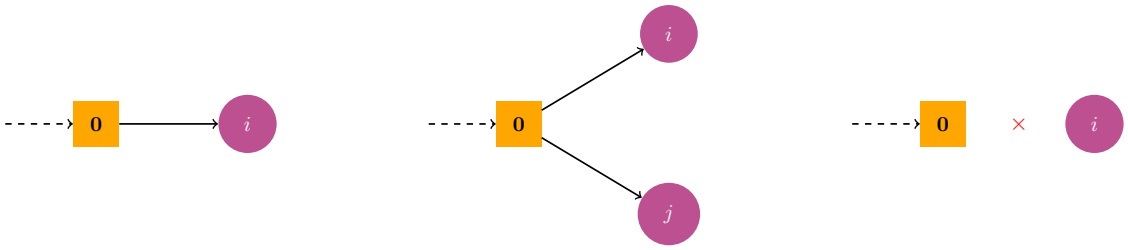
CC BY-NC-SA 4.0

it will be our role to provide the necessary constraints that ensure the optimal solution of our mathematical formulation corresponds to a valid tour. Let us begin with the objective function. Because the total cost of the tour is the sum of the cost of each leg (i.e., of each direct movement from one location to another), we can express its minimisation with the following objective function:

$$\min \quad \sum_{i \in V'} \sum_{\substack{j \in V' \\ j \neq i}} c_{ij} x_{ij}.$$

In the above expression, we pay the cost c_{ij} if we visit location j immediately after location i ($x_{ij} = 1$). If we represent the solutions depicted in Figure 4 in more abstract terms, without using a geographically accurate map, we might come up with the drawings of Figure 5. Each arrow indicates a direct leg from its head to its tail. We only drew arrows corresponding to variables that take the value one in the solution depicted; i.e., an arrow from 6 to 1 means that $x_{61} = 1$ in the solution we are analysing. By writing the cost of each leg next to the corresponding arrow, we see that we can obtain the total tour cost by summing costs c_{ij} for all the location pairs (i, j) selected in the solution ($x_{ij} = 1$).

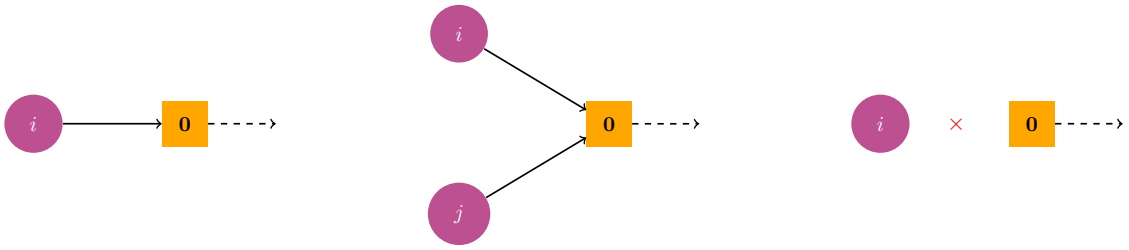
The constraints of the model must ensure that we devise a valid tour. In particular, we must guarantee that all customers are visited (once) and that the tour starts and ends at the depot. To do so, we might reason as follows. When it starts from the depot, the vehicle will go to some other location. Therefore, there will be some location $i \in V$ such that $x_{0i} = 1$, meaning that we travel directly from the depot 0 to i. Furthermore, there can only be one such location because, once the vehicle leaves the depot, it can only go to one location as the "immediate next location": the vehicle cannot split in two and go to two locations at once. In the schematic representation of Figure 5, then, we can only draw one arrow out of the yellow depot but never more than one and never zero. If there were two arrows, the vehicle must somehow have split in two (invalid solution), and if there were zero, the vehicle must not have left the depot (invalid solution).



valid invalid invalid

We can reason the same also about returning to the depot. When the vehicle returns, it must come from exactly one other location and, therefore, there must be exactly one location $i \in V$ such that $x_{i0} = 1$. In the depiction of a valid solution, therefore, we must have exactly one arrow arriving at the depot. If there were two such arrows, the vehicle must have "split" at some point (invalid solution); if there were zero, then the vehicle is not coming back to the depot (invalid solution).

CC BY-NC-SA 4.0



valid invalid invalid

But we can generalise this reasoning to all locations, not just the depot. For any given location (be it the depot or a customer), the vehicle must arrive there, and when it arrives, it can only come directly from exactly one other location. The vehicle must also depart from any location, and when it departs, it can only go directly to one other location. These two requirements translate into the following two constraints:

$$\sum_{\substack{j \in V' \\ j \neq i}} x_{ji} = 1 \qquad \forall i \in V'$$

$$\sum_{\substack{j \in V' \\ j \neq i}} x_{ij} = 1 \qquad \forall i \in V'.$$

Introducing the two above constraints will ensure that each location is visited exactly once. Therefore, it might appear that our model is complete and that we can write it down in full as follows:

$$\min \quad \sum_{i \in V'} \sum_{\substack{j \in V' \\ j \neq i}} c_{ij} x_{ij} \tag{1a}$$

$$\text{subject to} \quad \sum_{\substack{j \in V' \\ j \neq i}} x_{ji} = 1 \qquad \forall i \in V' \tag{1b}$$

$$\sum_{\substack{j \in V' \\ j \neq i}} x_{ij} = 1 \qquad \forall i \in V' \tag{1c}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in V',\ \forall j \in V' \setminus \{i\}. \tag{1d}$$

When we input this model into a solver, and we try to obtain the best tour to visit all customers of the instance in Figure 1, however, we are in for a surprise. The solver says that the following variables should have value one: $x_{06}, x_{65}, x_{57}, x_{70}, x_{12}, x_{23}, x_{34}, x_{41}$. All other variables have a value of zero. If we schematically draw this solution, we obtain Figure 6. This is not a valid TSP solution: a driver that would follow this solution would leave the depot, visit Customers 6, 5 and 7, and go back to the depot. At the same time, according to Figure 6, the driver should somehow teleport the vehicle, e.g., to Customer 2's location and, from there, visit Customers 3, 4, and 1, and go back to Customer 2. From there, the vehicle must teleport back to the depot. This is indeed an impossible solution unless we are in the world of Star Trek. The problem with the solution described in Figure 6 is that it contains **subtours**. In other words, it does not describe a single tour starting and ending at the depot. Rather, it describes multiple tours, only one of which starts involves the depot.

It is important to remark that this solution with subtours perfectly complies with the constraints of model (1a)–(1d) because, for each location i, there is exactly one location j such that the vehicle travels directly from i to j, complying with constraint (1c). Furthermore, for each

CC BY-NC-SA 4.0

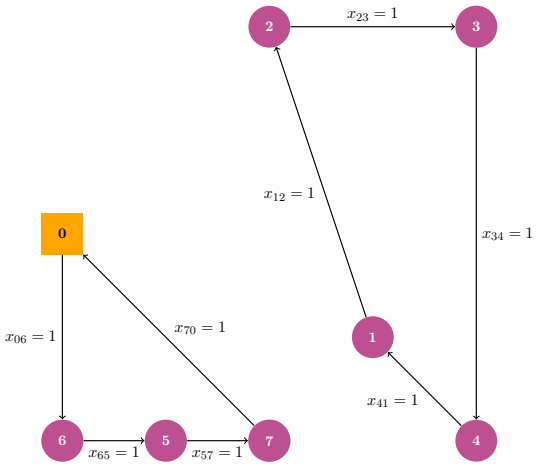


Figure 6: Schematic representation of the optimal solutions of (wrong) model (1a)–(1d).

location i, there is also exactly one location j such that the vehicle travels directly from j to i, complying with constraint (1b).

If a solver produces an absurd solution when solving the model we provided it, make no mistake: it is not the solver that is wrong; it is our model. In particular, our model cannot ensure that the optimal solution does not contain subtours. There are various possible ways to amend the model and solve this shortcoming. Here, we will discuss one particular way that involves introducing an exponential number of inequalities, i.e., a group of inequalities whose number grows as an exponential function of the size of V'. These constraints are known as the "**DFJ** constraints" from the names of the three people who first proposed them: Dantzig, Fulkerson and Jhonson. They are also known as **SEC**s, i.e., Subtour Elimination Constraints. They read as follows:

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1 \quad \forall S \subsetneq V',\ S \neq \emptyset. \tag{2}$$

Inequalities (2) state that, given a proper subset of V' (i.e., a subset which is not empty and is not the entire V'), the solution must include at least one crossing arc. A crossing arc is an arc whose origin is in S and whose destination is not.

Remark 1. The following is an alternative way of formulating SECs (2).

$$\sum_{i \in S} \sum_{j \in S \setminus \{i\}} x_{ij} \leq |S| - 1 \quad \forall S \subsetneq V',\ S \neq \emptyset. \tag{3}$$

This constraint states that all proper subsets S of V' must contain strictly less than $|S|$ internal arcs. An arc is called internal if its origin and destination are vertices of S.

A complete TSP formulation reads as follows:

$$\min \quad \sum_{i \in V'} \sum_{\substack{j \in V' \\ j \neq i}} c_{ij} x_{ij} \tag{4a}$$

$$\text{subject to} \quad \sum_{\substack{j \in V' \\ j \neq i}} x_{ji} = 1 \qquad \forall i \in V' \tag{4b}$$

$$\sum_{\substack{j \in V' \\ j \neq i}} x_{ij} = 1 \qquad \forall i \in V' \tag{4c}$$

CC BY-NC-SA 4.0

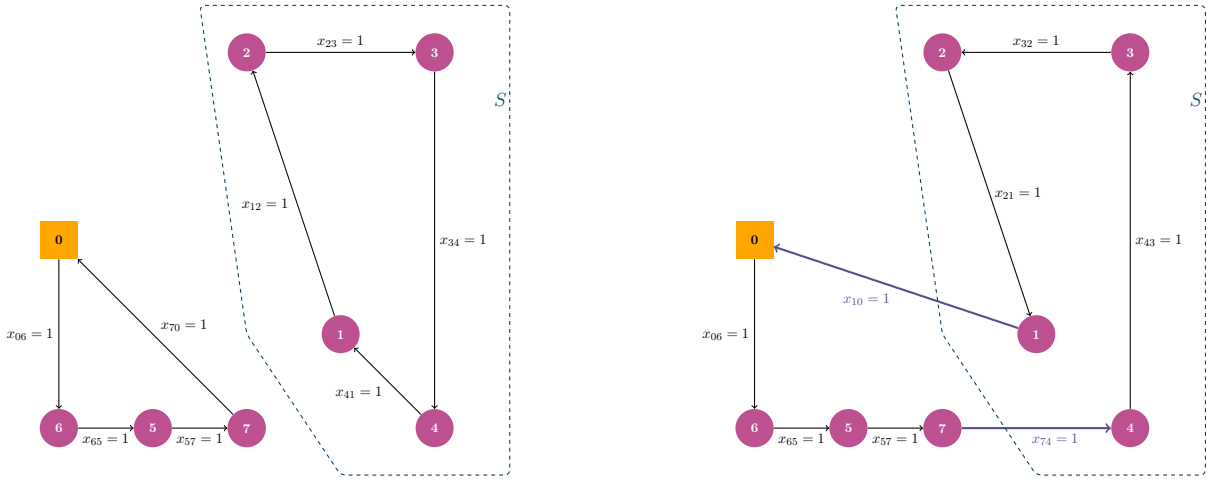


Figure 7: A solution with subtours violates at least one subtour elimination constraint (left). A solution without subtours does not violate any (right). Thick arcs in the right figure represent crossing arcs for set S.

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1 \qquad \forall S \subsetneq V',\ S \neq \emptyset \tag{4d}$$

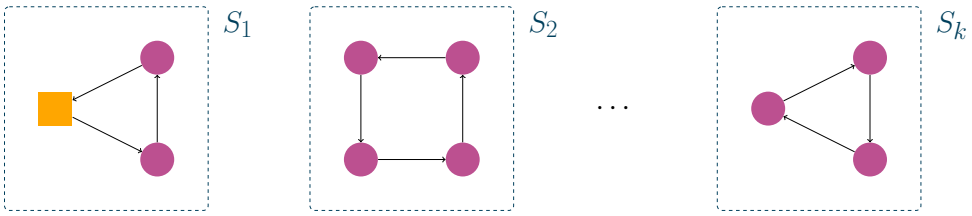
$$x_{ij} \in \{0, 1\} \qquad \forall i \in V',\ \forall j \in V' \setminus \{i\}. \tag{4e}$$

We remark that the SECs (4d) grow exponentially with the number of vertices $|V'|$. For example, a TSP with 101 locations (a depot and 100 customers) would have roughly $1.26 \cdot 10^30$ SECs. Such a large number is unmanageable, and we could not input these many constraints into a computer. At the same time, formulation (4a)–(4e) is known for being a rather strong TSP formulation, i.e., one whose continuous relaxation's optimal objective is not far from the integer optimal objective. Can we get the best of both worlds? A strong formulation that is practically solvable on hardware? To achieve this aim, we must introduce a new type of algorithm known as **branch-and-cut** (B&C).

3 A B&C algorithm with separation on integer solutions

The idea behind the B&C algorithm is that we have more SECs than we can handle, so we should start solving the TSP via branch-and-bound (B&B) *without* SECs. In other words, we start solving the incomplete formulation (1a)–(1d).

During the exploration of the B&B tree, we will encounter integer solutions when the optimum of the continuous relaxation of (1a)–(1d) plus the branching constraints is integer at the current node. Because we have removed the SECs, these solutions will likely contain subtours. Let $S_1, S_2, \ldots, S_k$ be the subtours of a given integer solution.



If $k = 1$, i.e., the solution does not contain any subtour, then it is a valid TSP solution; we

CC BY-NC-SA 4.0

need not do anything about it, and we keep exploring the B&B tree. Otherwise, we add to the formulation k SECs, one for each subtour:

$$\sum_{i \in S_1} \sum_{j \in V' \setminus S_1} x_{ij} \geq 1$$
$$\sum_{i \in S_2} \sum_{j \in V' \setminus S_2} x_{ij} \geq 1$$
$$\vdots$$
$$\sum_{i \in S_k} \sum_{j \in V' \setminus S_k} x_{ij} \geq 1.$$

We discard the solution with the subtours and solve the continuous relaxation again, this time with the newly added SECs. These new constraints will remain part of the formulation until the end of the B&B tree exploration, i.e., we keep them even when exploring other tree nodes. The new SECs will prevent the formulation from producing the same solution with subtours again. Therefore, when solving the continuous relaxation again, we either get a fractional solution, an integer solution with no subtours, or an integer solution with *different* subtours. In this latter case, we repeat the procedure and add even more SECs. Eventually, we will close the current B&B node and explore other nodes.

We will repeat this procedure at all nodes when encountering an integer solution. When we close all the nodes, the best integer solution encountered is the optimal solution. Because we have discarded all solutions with subtours, the best solution must be feasible for TSP (i.e., it cannot contain subtours).

Remark that finding the subtours in an integer solution can be done by simple inspection. Start from vertex 0 and follow the tour by inspecting the x variables' values in the solution. Because of constraints (4c), there will be exactly one vertex i such that $x_{0i} = 1$ in any integer solution. Therefore, i follows 0 in the tour. We can repeat this process to find the unique vertex j that follows i (i.e., $x_{ij} = 1$), etc. Finally, the tour completes when we return to the depot 0. If the tour does not visit all vertices, it forms a subtour S. In that case, we take any other vertex $i' \in V' \setminus S$ and repeat the procedure to find the subtour including i'. We continue until all vertices are accounted for. At the end of this procedure, we get the list of subtours $S_1, \ldots, S_k$.

4 A B&C algorithm with separation on integer and fractional solutions

The procedure described in Section 3 is correct and complete, i.e., it returns an optimal integer feasible solution. In fact, because no infeasible integer solution is ever accepted, the final solution returned at the end of the tree's exploration is necessarily feasible. However, intermediate fractional solutions are allowed to be infeasible. These intermediate solutions are the (fractional) optimal solutions of the continuous relaxation of the problem at non-leaf nodes. They can be infeasible in the sense that violate some SEC (4d) which was not added to the model.

For fractional solutions, the SECs do not have an obvious interpretation in terms of subtours. Still, a fractional solution can satisfy them if the left-hand side is ≥ 1 or violate them if the left-hand side is < 1. When we solve the continuous relaxation during the B&B tree exploration, we are "relaxing the formulation twice": first, we relax the integrality constraints and, second, we relax all the SECs that are not explicitly added to the model. For this reason, the dual

CC BY-NC-SA 4.0

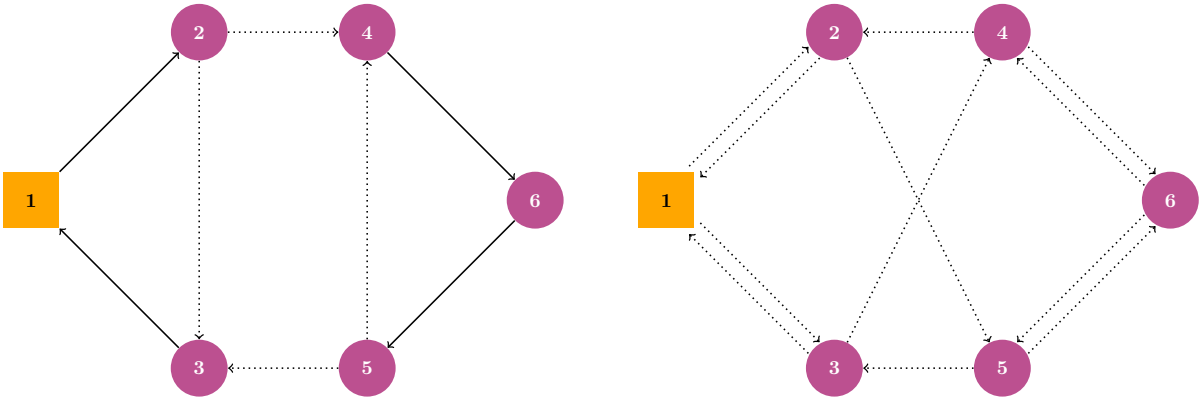


Figure 8: Two fractional solutions in which solid arrows represent arcs chosen with $x = 1$ and dotted arcs represent arcs chosen with $x = \frac{1}{2}$. The solution in the left figure violates a SEC (for example, for $S = \{4, 5, 6\}$). The solution on the right does not violate any SEC.

(lower) bound that we obtain from the objective value of the optimal solution of the continuous relaxation is weaker than it would be if we enumerated all the SECs a priori. In the context of the B&B algorithm, this fact is parcitularly noxious because a loose dual bound allow less pruning compared to a tighter bound. Therefore, we can expect an improvement on the performance of our algorithm if we could also ensure that optimal fractional solutions of the continuous relaxation satisfy with the SECs. To this end, we require a mechanism that efficiently detects if a given fractional solution violates one or more SECs.

In the integer solution case, a simple inspection of the solution is enough to determine if it violates a SEC. This is not the case for fractional solutions. Figure 8 shows two fractional solutions of a small TSP instance. In the figure, an arrow represents an arc (i, j) selected in the solution. The corresponding variable has value $x^*_{ij} = 1$ if the arc is solid and $x^*_{ij} = \frac{1}{2}$ if it is dotted. Both depicted solutions respect all in-degree and out-degree constraints (4b) and (4c). Moreover, the solution on the right does not violate any SEC while the left solution does. For example, taking $S = \{4, 5, 6\}$, we have that, in the left figure:

$$\sum_{i \in \{4,5,6\}} \sum_{j \in \{1,2,3\}} x^*_{ij} = \frac{1}{2} < 1.$$

In the rest of this section, we build a **separation routine**, i.e., a procedure that takes a fractional solution as its input and returns a set that the solution violates. Given a solution x^*, the first step is to build a "support graph" $G* = (V, A^*)$ in which we only keep the edges associated with positive variables x^*:

$$A^* = \big\{(i, j) \in A \ : \ x_{ij} > 0\big\}.$$

Next, we define a family of min-cut problems on graph G^*. To this end, we must provide the arc costs c_{ij} for each $(i, j) \in A^*$, a source vertex s and a sink vertex t. In our procedure, we use the values of the x^* variables as the cost, i.e., $c_{ij} = x^*_{ij}$ for each $(i, j) \in A^*$. We use the depot $0 \in V'$ as the source node, i.e., $s = 0$. Finally, we take any vertex $t \in V$ (that is, any vertex other than the depot) as the sink node.

Our central claim is the following. If the min-cut problem has cost < 1, then x^* violates at least one SEC. Let (S, T) be the s-t partition associated with an optimal solution of the min-cut

CC BY-NC-SA 4.0

problem. If the min-cut optimal objective value is strictly smaller than one, by definition

$$\sum_{(i,j)\in A_{S,T}} x^*_{ij} < 1.$$

Because $T = V' \setminus S$, we can write the above relation as

$$\sum_{i\in S} \sum_{\substack{j\in V'\setminus S \\ \text{s.t. } (i,j)\in A^*}} x^*_{ij} < 1,$$

and, because $x^*_{ij} = 0$ when $(i,j) \notin A^*$, we have

$$\sum_{i\in S} \sum_{j\in V'\setminus S} x^*_{ij} < 1. \tag{5}$$

Condition (5) clearly states that the SEC (4d) associated with set S is violated.

When x^* violates a SEC, we do not know which sink t would give rise to a min-cut problem that "uncovers" the violated SEC. Therefore, in our separation procedure, we will iterate through the possible values of t until we find one for which the min-cut optimal solution has cost < 1. In this case, we add the corresponding violated SEC. Conversely, if we try all possible values of $t \in V$ and none of the corresponding min-cut problems has cost < 1, we are sure that x^* does not violate any SEC.

Furthermore, we will exploit the max-flow/min-cut duality, and instead of solving the min-cut problems directly, we will solve their dual max-flow problems. Because of strong duality, the min-cut optimal cost and the max-flow optimal flow are equal, and thus, the cost is < 1 if and only if the flow is < 1. We solve the max-flow problem because there is a richer literature on efficient algorithms for the max-flow than for the min-cut.

A concern about the previous approach is that, while the optimal objectives of max-flow and min-cut coincide due to strong duality, to separate a violated SEC, we need an s-t partition, and solving the max-flow problem does not produce one. However, it can be proven that it is easy to retrieve the s-t cut corresponding to an optimal min-cut solution, given an optimal max-flow solution. Virtually all mainstream Python libraries that solve the max-flow problem provide a routine to convert a max-flow into a min-cut solution.

The notebooks available on the website provide a complete implementation of the B&C algorithms presented in Section 3 and Section 4.

CC BY-NC-SA 4.0