Math 3272: Linear Programming[1] Mikhail Lavrov

Lecture 28: The traveling salesman problem

December 1, 2022 *Kennesaw State University*

1 The traveling salesman problem

The "flavor text" of the traveling salesman problem (TSP) is the following. There are n cities, numbered $1, 2, \ldots, n$, with some costs of travel between them. Between two cities i and j, we are given a cost of travel c_{ij} to go from i to j. (We assume that it's possible to travel from any city to any other—maybe you can hire a private jet if you need to—but some costs may be extremely large.)

A salesman starting in city 1 wants to visit all n cities in some order and return to city 1. We call this a **tour** of the n cities—sometimes we call it a **closed tour** to distinguish it from **open tours** which do not have to return to the starting point. The salesman's goal is to find the cheapest possible (closed) tour, adding up the cost of all n legs of the tour. There are $(n-1)!$ orders in which the other cities could be visited, so this is not a problem we can solve by brute force for any reasonable value of n.

These days, we buy everything online, so we are not interested in solving the problems of actual traveling salesmen. On the other hand, an Amazon delivery truck might end up solving the traveling salesman problem if it has to make n deliveries in a neighborhood in the shortest amount of time. Many other route-planning algorithms also require solving the traveling salesman problem, even when literal salesmen are not involved.

There are also many industrial applications in which "cities" and "travel" are more metaphorical. For example, if we are constructing an object layer by layer in a 3D printer, then optimizing the order in which we deposit material is a variant of the traveling salesman problem. We might also be drilling holes in a circuit board, cutting a sheet of wood with a laser cutter, or even manipulating a robot arm to take photos of an object from multiple angles[2].

Even if some of these problems add additional twists to the problem, the starting point is usually one of the two TSP formulations we will look at today.

It will be convenient for us to assume that we never visit a city more than once in a tour. In some formulations, this may require distinguishing between "official" and "unofficial" visits to a city. For example, we can imagine that if we're trying to tour the US by taking airplane flights, we might go from Atlanta to Orlando to Charlotte, and the flight from Orlando to Charlotte might have a layover in Atlanta. In this case, the cost of the Orlando–Charlotte route in our problem would simply be the total cost of the two-leg trip, and we don't even notice the layover in Atlanta when we're finding the optimal tour.

[1]This document comes from the Math 3272 course webpage: https://facultyweb.kennesaw.edu/mlavrov/courses/3272-fall-2022.php

[2]For more details of this unusual application—with pictures!—see the paper where I originally found it: https://doi.org/10.3390/robotics11010016.

On the other hand, we could also consider a problem where "unofficial" visits to a city are not allowed—for example, we are still trying to visit n different cities in the US by airplane and return, but we want our trip to consist of n direct flights. In this case, the cost of going from Orlando to Charlotte might increase if we have to avoid stopping in Atlanta. We will return to this distinction at the end of the lecture.

2 An incomplete formulation

Here is a first attempt at representing the problem with an integer program.

Suppose that to every pair of cities (i, j), we assign an integer variable $x_{ij} \in \{0, 1\}$ which will equal 1 if the tour goes from city i to city j, and 0 otherwise. Then the pair (i, j) constributes $c_{ij}x_{ij}$ to the total cost of a tour: c_{ij}, when $x_{ij} = 1$, and 0, when $x_{ij} = 0$. Therefore, we have an objective function to

$$\text{minimize } \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}.$$

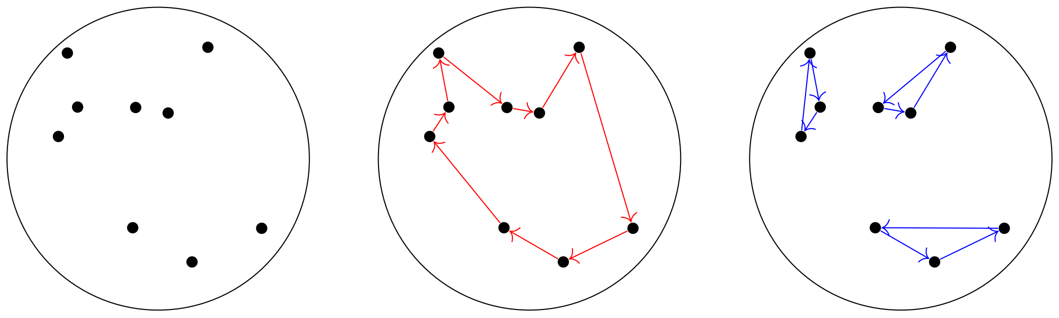
In a tour, we visit each city only once: we enter the city once, and then we leave the city once. (For city 1, we do those in a different order: first we leave city 1, and then we return to it. But this doesn't affect things; in fact, in a tour, it doesn't matter which city is the starting city.) We can represent this requirement by a pair of constraints for each city:

$$\sum_{\substack{1 \le i \le n \\ i \ne j}} x_{ij} = 1 \qquad \text{for each } j = 1, 2, \dots, n \tag{1}$$

$$\sum_{\substack{1 \le k \le n \\ k \ne j}} x_{jk} = 1 \qquad \text{for each } j = 1, 2, \dots, n \tag{2}$$

Equation (1) says that we arrive at city j from exactly one other city. Equation (2) says that we leave city j to go to exactly one other city.

If these constraints were all we needed, we'd be in great shape. (In fact, the constraint matrix so far is totally unimodular, so we wouldn't even need to worry about integer programming techniques. We will not prove this, because it doesn't immediately help us with anything, but it's true.) Unfortunately, there's a problem.



Take a random set of 9 points (as in the first diagram) and let c_{ij} be the distance between the i^{th} point and the j^{th} point. Then the minimum-cost tour between the 9 points, as found by a brute-force search, is shown in the second diagram. Unfortunately, the solution to the integer program with constraints (1) and (2), shown in the third diagram, is not a tour at all!

The optimal solution to the integer program we have so far satisfies the constraint that we must enter each node once and leave it once, and so it looks like a tour "locally". Unfortunately, it is missing the "global" condition that the tour must be connected.

3 Subtour elimination constraints

We solve this problem by adding additional constraints called **subtour elimination constraints** that rule out the disconnected solutions. There are two famous solutions to this problem that take very different approaches, each with its advantages and drawbacks.

3.1 Solution #1: the DFJ constraints

The first solution to this problem was proposed by Dantzig, Fulkerson, and Johnson in 1954. To disambiguate, we will call this set of constraints the DFJ constraints.

The DFJ subtour elimination constraints are the constraints

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1 \qquad \text{for each } S \text{ s.t. } 1 \leq |S| \leq n-1 \tag{3}$$

For every set S of cities, other than the empty set $\varnothing$ and the set $\{1, 2, \dots, n\}$ of all cities, the sum on the left-hand side ranges over all pairs (i, j) such that going from city i to city j leave S. By requiring the sum to be at least 1, we require that the tour will leave the set S at least once.

This is guaranteed to happen for any legitimate tour. Since the tour visits every single city, it must visit a city in S at some point. However, the tour cannot stay in S forever, since there are also cities not in S, so eventually it must take a step that leaves S.

However, the optimal solution to the constraints in (1) and (2) on the previous page violates this condition. We could, for example, take S to be the set of the three points on the bottom. The solution there consists of a "subtour" that just cycles between the three cities in S, and some other thing that happens between the six cities outside S, with no step that leaves S.

With the subtour elimination constraints in play, every integer solution to (1), (2), and (3) is actually a valid tour, and so we can solve the TSP problem using an integer program. A slightly concerning feature of the subtour elimination constraints is that there are $2^n - 2$ of them. That number grows almost as quickly as the number $(n-1)!$ of possible tours, so solving even the linear programming relaxation might not be quicker than solving the TSP problem by brute force.

A solution to this is to add the constraints in (3) on the fly, one at a time, just as we added the fractional cuts in the previous lecture. Given any integer solution to (1) and (2) that is *not* a tour, we can quickly find a set S for which the corresponding constraint in (3) is violated. For example, we can start at city 1 and follow the path defined by the integer solution (by going from city i to the unique city j such that $x_{ij} = 1$) until we return to city 1. Let S be the set of all cities we visit: then either $S = \{1, 2, \dots, n\}$ and we have a tour, or else the subtour elimination constraint for S

is violated because

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} = 0.$$

As a result, one way to proceed is using a hybrid branch-and-cut method, starting with just (1) and (2) as the constraints. Whenever we find a fractional solution (which can't happen with just those constraints, but might happen if we have added some of the constraints in (3) already), we can branch on one of the fractional variables. Whenever we find an integer solution which doesn't represent a tour, we can find a set S for which the constraint in (3) is violated, and add that constraint to the problem.

3.2 Solution #2: MTZ constraints

Another way to formulate the integer program, discovered by Miller, Tucker, and Zemlin in 1960, avoids the subtour elimination constraints in favor of a more compact formulation. In the MTZ constraints, we add $n-1$ additional variables $t_2, t_3, \dots, t_n$ that, intuitively, represent the time at which a city is visited.

If the times at which we visit the cities are given, then we can eliminate subtours with the condition that, when going from city i to city j, the time t_j at which city j is visited must be later than the time t_i at which city i is visited. Because strict inequalities are not something we allow in linear programs, we will phrase this as the logical implication

$$\text{if } x_{ij} = 1, \text{ then } t_j \geq t_i + 1$$

for every pair (i, j) with $i \neq 1$ and $j \neq 1$. (We leave t_1 undefined and don't include it in these constraints because city 1 is visited twice: at the start of the tour, and at the end.)

We can encode the logical implication with the "big-M" technique:

$$t_j \geq t_i + 1 - M(1 - x_{ij})$$

where M is some large number: when $x_{ij} = 0$, the constraint $t_j \geq t_i + 1 - M$ does nothing, and when $x_{ij} = 1$, we have $t_j \geq t_i + 1$. We can actually choose $M = n$, because the times can be chosen from the range $[0, n-1]$. This gives us the constraints

$$t_i - t_j + 1 \leq n(1 - x_{ij}) \qquad \text{for all } i, j \neq 1 \text{ s.t. } i \neq j \tag{4}$$

In any actual tour, we can satisfy (4) by setting $t_i = 1$ for the first city we visit after city 1, $t_i = 2$ for the second, and so on, with $t_i = n-1$ for the last city (after which we return to city 1).

However, if we have an integer solution to (1) and (2) that is not a tour, it must have a subtour not including city 1. For that subtour, some constraint in (4) must be violated. For example, if the subtour goes from city a to b to c back to a, then we must have $x_{ab} = x_{bc} = x_{ca} = 1$, and constraint (4) for these three pairs simplifies to

$$\begin{aligned} t_a - t_b + 1 &\leq 0 \\ t_b - t_c + 1 &\leq 0 \\ t_c - t_a + 1 &\leq 0 \end{aligned}$$

There is no solution to these three constraints: when we add all three of them together, the variables t_a, t_b, t_c all cancel and we get the false inequality $3 \le 0$. We get a similar contradiction for a longer subtour.

With the equations (1), (2), (4), we only have around n^2 constraints in our (n^2+n)-variable integer program, which is much better than the around 2^n constraints we had earlier. The variables $t_2, t_3, \dots, t_n$ don't even need to be integer variables, although there is an optimal solution where they all have integer values.

It is not necessarily true that the MTZ constraints are better than the DFJ constraints, just because there are fewer of them. In practice, it seems that the DFJ constraints have better performance—and adding the constraints on the fly with a branch-and-cut approach solves the main obstacle to using them. Still, the MTZ constraints have the advantage that they're easier to work with without specialized code: both approaches are useful in the right circumstances.

4 Approximation algorithms

So far in this class, we've talked about ways to solve an integer program exactly. Sometimes, we are not that greedy: we will be happy if we find an integer solution that's pretty good. Even this is not always easy: there is no general-purpose strategy.

In some (but not all) cases, we can obtain a decent solution by rounding. This happens, for example, with the configuration LP that we used to solve packing problems in Lecture 26. In that problem, solving the LP relaxation might give us a solution with $\frac{5}{2}$ of one configuration, $\frac{5}{2}$ of another configuration, and $\frac{5}{3}$ of a third configuration. Rounding these values up could give us an integer solution with 3 of the first configuration, 3 of the second, and 2 of the third. This is not the best integer solution, but it is decent: we are guaranteed to exceed the optimal solution by at most the number of configurations!

The traveling salesman problem is not a case where we can get anywhere by rounding. A fractional solution might end up "leaving" city 1 by setting $x_{12} = x_{13} = \frac{1}{2}$. If we round these values up to integers, we obtain a solution that is supposed to go from city 1 to both city 2 *and* to city 3 at the same time: this is nonsense!

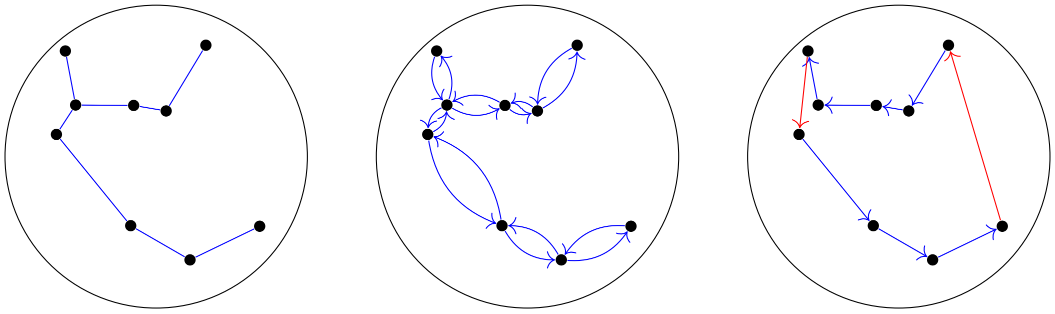
We will see a decent approximation algorithm in the case of the *metric* traveling salesman problem: where costs are symmetric ($c_{ij} = c_{ji}$) and satisfy the triangle inequality $c_{ij} + c_{jk} \ge c_{ik}$. This is true of distances in the plane, for example.

The trick here is that a related problem turns out to be much easier to solve. Suppose that we want to find the cheapest set of connections that join all the cities together, even if we cannot visit them in order. This is always guaranteed to be *at least as cheap* as the cheapest closed tour. Moreover, this is a problem that can be solved greedily: if we repeatedly take the cheapest connection that does not create a subtour until we reach $n-1$ connections (for n cities), then the result will be optimal for this simplified problem.

Given such a set of connections, we can find a closed tour, with some redundancies in it, that uses each connection twice: once in each direction. This is where we lose optimality: we are now at twice the cost of the simplified problem, which could be as bad as twice the cost of the optimal traveling salesman tour (but no worse).

Finally, assuming that the triangle inequality $c_{ij} + c_{jk} \geq c_{ik}$ holds, we can simplify our solution to a standard closed tour that does not revisit any cities. The way to do this is simple: every time you'd be coming back to a city where you've already been, just skip ahead to the next new city! This can only reduce the cost of the tour.

An example of this is shown below: first the cheapest set of connections that join together all the cities, then the tour that uses each of those connections twice, then the simplified closed tour that does not revisit cities. (In the second diagram, the curved arcs are just there to help distinguish the two times we use a connection; the distances we use for costs are still measured along a straight line. In the last diagram, the two times we "skip ahead" are drawn in red.)



In this case, though we did not find the optimal solution, we got much closer than the factor-2 guarantee. The optimal solution (shown on a previous page) has total length about 4.88733; the solution found by our approximation algorithm has total length about 5.01606.

A fancier version of this approximation algorithm, called the Christofides algorithm, does even better: its cost is at most 1.5 times the cost of an optimal tour. This is not always what we want, but it can be better than nothing if our integer programs are too big to solve directly!