Math 482: Linear Programming[1] Mikhail Lavrov

Lecture 35: The Traveling Salesman Problem

May 1, 2020 *University of Illinois at Urbana-Champaign*

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. The salesman's goal is to find the cheapest possible 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.

We are of course usually not interested in solving the problems of actual traveling salesmen. But the same optimization problem occurs in minimizing the costs of other trajectories, including:

- Planning routes in a neighborhood, from mail delivery to garbage collection or a school bus picking up children.
- Part manufacturing, where we might be drilling holes in a circuit board, cutting a sheet of wood or metal with a laser cutter, or constructing an object layer by layer in a 3D printer.

For historical reasons, the TSP formulation is the one people talk about.

It will be convenient for us to assume that we never visit a city more than once in a tour. Depending on the specific problem, this may or may not be a natural constraint to ask for. For example, if the n cities actually correspond to points in the plane, and the cost c_{ij} is proportional to the distance, then the shortest tour will never revisit a city.

If we consider real cities on the U.S. map, with travel times between them as costs, then it is probably easier to go from Champaign to New York via Chicago than to try traveling directly. But we can avoid this problem by defining c_{ij} to be the cost of a cheapest route from city i to city j, even if this ends up revisiting some cities. (Thus, we might define $c_{\text{Champaign,New York}}$ to be the cost of a Champaign–Chicago–New York route.)

[1] This document comes from the Math 482 course webpage: https://faculty.math.illinois.edu/~mlavrov/courses/482-spring-2020.html

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

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

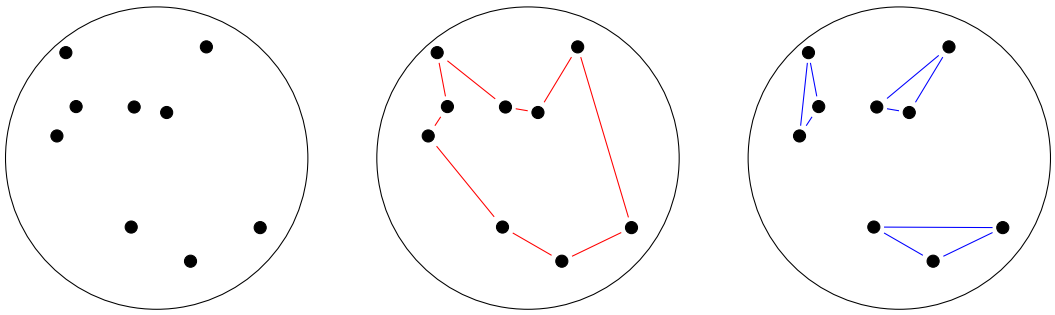
In a tour, we visit each city only once: we enter the city once, and then we leave the city once. (For city 1, we do those in a different order: first we leave city 1, and then we return to it. But this doesn't affect things; in fact, in a tour, it doesn't matter which city is the starting city.) We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

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

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

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. 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 Solution #1: subtour elimination constraints

The first solution to this problem was proposed by Dantzig, Fulkerson, and Johnson in 1954. They add a large number of constraints known as the *subtour elimination constraints*.

The 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 $\{1, 2, \ldots, n\}$ 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 goes from one to the other.

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 $S = \{1, 2, \ldots, n\}$ 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.

4 Solution #2: timing 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. We can call these constraints *timing constraints*, because their goal is to define a variable t_i for each city i representing the time at which city i is visited along the tour. (This terminology is not standard.)

The idea behind the constraint is to write down 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-number" 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 timing 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 these constraints 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 the timing constraints for these three pairs simplify to

$$t_b \geq t_a + 1$$
$$t_c \geq t_b + 1$$
$$t_a \geq t_c + 1$$

There is no solution to these three constraints: by adding them together, we get $t_a + t_b + t_c \geq t_a + t_b + t_c + 3$, or $0 \geq 3$. 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 $t_2, t_3, \ldots, t_n$ don't even need to be integer variables, although there is an optimal solution where they all have integer values.

However, in practice, the LP relaxation of the timing constraints is worse, and so more steps are needed to solve the integer program. We are better off with the subtour elimination constraints, which we can add on the fly.