LECTURE 10

Ishaan Lal

April 7, 2025

1 Introduction

We have finally reached the end of the SAT/CP saga of this course. The remaining lectures will be focused on some special topics. Today, we will give our attention to the Traveling Salesperson Problem.

2 The Traveling Salesperson Problem (TSP)

The problem statement for TSP is as follows: Given a *weighted, complete* graph, find a tour (traversal of edges) of minimum total weight that visits every vertex *exactly once* and returns to the starting vertex.

For clarity, our graph can be directed or undirected. Throughout this lecture, we'll mostly be dealing with *complete graphs*, which are graphs that have all edges present.

TSP is equivalent to finding a Hamiltonian cycle in a graph such that the Hamiltonian cycle is of least weight. This problem is a known NP-complete problem. Its applications are widespread from solving problems in routing, to logistics, to even producing microchips.

2.1 Notation

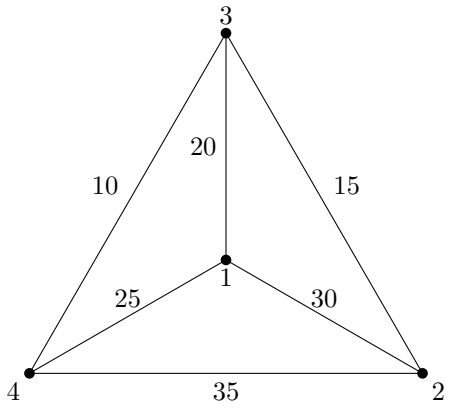
First, a brief note on notation. We'll look at complete directed graphs. A complete graph is a graph such that every pair of vertices have edges between them. "Directed" means that our edges are represented by arrows indicating the direction of travel. Between arbitrary vertices i and j, we will have edges $i \to j$ and $j \to i$. We may use undirected graphs to represent a complete directed graph, where the weight of the undirected edge is equivalent to the weight of the two directional edges in the directed version of the graph.

We notate an edge as (i, j) where $i, j \in V$, and $(i, j) \equiv i \to j$. The weight of edge (i, j) will be notated $w(i, j)$ and we have $w(i, j) \in \mathbb{R}$. In most applications, we deal with $w(i, j) \in \mathbb{R}^+$.

We denote a **tour** as a permutation $v_1, v_2, ..., v_n$ of the vertices, which represents $v_1 \to v_2 \to ... \to v_n \to v_1$

2.2 Example

As a running example, we will consider the following small instance of a TSP problem:



For simplicity, our examples will be drawn as undirected graphs, just as above. You can imagine each edge (i, j) is really just two parallel edges $i \to j$ and $j \to i$ with the same cost/weight.

In the above example, the optimal tour is $1 \to 3 \to 4 \to 2 \to 1$ for a total cost of $30 + 15 + 10 + 25 = 80$

2.3 Attempt 1: Constraint Programming

What happens if we try to solve TSP with CP? As always, we need to define our variables, their values, and the constraints.

The **variables** will be $x_{ij} \in \{0, 1\}$ – boolean variables for each edge, indicating if the edge is part of the TSP tour.

We aim to minimize the total cost:

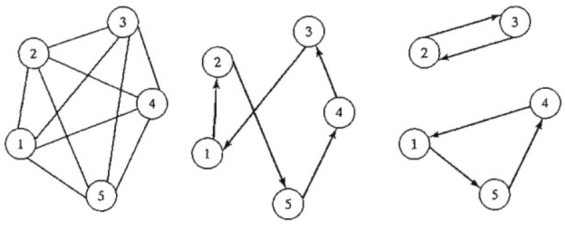
$$\text{COST} = \sum_{(i,j)} w(i, j) \cdot x_{ij}$$

The expression can be understood as follows: for each edge, if it is part of our TSP tour, we must add the weight to our cost. That is, if $x_{ij} = 1$, then add $w(i, j)$. Otherwise, if $x_{ij} = 0$, then we do not need to add the weight.

One constraint to the problem is that each vertex is visited exactly once. To formalize this, consider an arbitrary vertex i. For it to be visited exactly once, there must be exactly one edge that "enters" it, and exactly one edge that "exits" it from the tour. We write this as:

$$\sum_{j \neq i} x_{ij} = 1 \qquad \sum_{j \neq i} x_{ji} = 1 \qquad \forall\, 1 \leq i \leq n$$

If we were to run our Constraint Program as we have defined it, it would not work. This CP formulation allows "subtours" rather than forcing one contiguous tour. To visualize this, in the diagram below, the left graph is the starting configuration; the middle path is the idealized tour; and the right graph is a possible tour that could be returned by our CP – it abides to the constraint of each vertex being visited exactly once!



One way to remedy this issue is as follows: enforce that for each *possible subtour* of vertices S, make sure that we take less than $|S|$ edges between them.

Written as a constraint:

$$\sum_{i \neq j \in S} x_{ij} < |S|, \quad \forall S \subset V, \quad |S| > 1$$

This brings to light a new problem: *there are exponentially many subtours!* There are ways to fix this such as adding constraints lazily, but in general, CP is not state-of-the-art for TSP (yet).

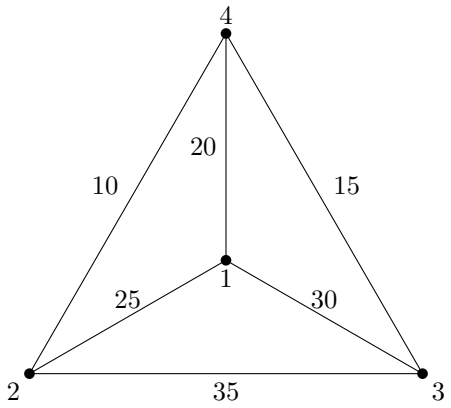
To reconcile these issues, we will leverage a key observation: TSP is an approximation-friendly problem. That is, in practice, a "good enough" solution is usually good enough! So our goal transforms into finding an approximate solution to TSP. We will **design efficient heuristics that give an empirically cheap tour (perhaps not the cheap*est*)**.

3 Nearest Neighbor

The first technique we will look at is called "Nearest Neighbor" (NN), and is described as follows with respect to TSP: Start at any vertex u. Then pick the nearest unseen out-neighbor v of u and add it to the end of the tour. Then, repeat starting from v, and continue until all vertices are added.

This is a greedy approach to the problem, whose pros are that it is simple, intuitive, and relatively efficient. The main con is that it *is* a greedy algorithm, so it can easily miss shortcut paths.

Recall our example graph:



Performing Nearest-Neighbor on the above would be simulated as follows:

- Start from vertex 1. Its nearest neighbor is vertex 4, so our path is now $1 \to 4$.
- From vertex 4, its nearest neighbor not yet on the path is 2, making our path $1 \to 4 \to 2$.

- From vertex 2, its nearest neighbor not yet on the path is 3, making our path $1 \to 4 \to 2 \to 3$
- At this point, we have visited all vertices, so we must return to 1 from 3.
- Our tour is thus $1 \to 4 \to 2 \to 3 \to 1$, for a total cost of 95.

4 Nearest Insertion

Let's take a step to improve upon our Nearest Neighbor technique. We'll maintain the greedy nature, but now we will allow the tour to be edited at an intermediary point, instead of just tacking on a stop at the end of the tour. The Nearest-Insertion (NI) heuristic is described as follows:

1. Start the tour T at any vertex
2. Pick the nearest unseen out-neighbor v of **any** vertex that has been included in the tour.
3. Insert it into the tour $T = t_1, ..., t_k$ so that the total distance is minimized. That is, find i such that $w(t_i, v) + w(v, t_{i+1}) - w(t_i, t_{i+1})$ is minimized
4. Repeat until all vertices are added to the tour.

We simulate NI on the same toy example:

- We'll choose to start our tour at vertex 1. Our closed tour is currently $1 \to 1$
- The nearest unseen out-neighbor is 4, which is 20 away. We cannot insert 4 between anything, so we just tack it on to the end. Our closed tour is now $1 \to 4 \to 1$
- The nearest unseen out-neighbor of 1 or 4 is 2, which is 10 away from 4. Given that our current tour is $1 \to 4 \to 1$
  - If we insert 2 after 1 (towards the tour $1 \to 2 \to 4 \to 1$), we would remove the edge $1 \to 4$, and add edges $1 \to 2$ and $2 \to 4$. Thus, the net change in total weight would be $w(1,2)+w(2,4)-w(1,4) = 25 + 10 - 20 = 15$.
  - If we insert 2 after 4 (towards the tour $\to 4 \to 2 \to 1$), we would remove the edge $4 \to 1$, and add edges $4 \to 2$ and $2 \to 1$. Thus, the net change in total weight would be $w(2,4)+w(1,2)-w(1,4) = 10 + 25 - 20 = 15$
  - Both cases are the same, so we will break the tie arbitrarily and insert 2 after 1. So our current tour is now $1 \to 2 \to 4 \to 1$
- The nearest unseen out-neighbor of 1, 2, or 4 is 3. Recall, our current tour is $1 \to 2 \to 4 \to 1$, giving us 3 possible locations to insert 3.
  - If we insert 3 after 1, then the net change in weight is $w(1,3)+w(3,2)-w(1,2) = 30+35-25 = 40$
  - If we insert 3 after 2, then the net change in weight is $w(2,3)+w(3,4)-w(2,4) = 35+15-10 = 40$
  - If we insert 3 after 4, then the net change in weight is $w(4,3)+w(3,1)-w(1,4) = 15+30-20 = 25$
  - The net weight is minimized when we insert after 4, so our new tour is $1 \to 2 \to 4 \to 3 \to 1$
- NI finishes, and we obtain tour $1 \to 2 \to 4 \to 3 \to 1$ which has weight $25 + 10 + 15 + 30 = 80$

5 Farthest Insertion

Let's see what happens when we modify the Nearest Insertion heuristic slightly. The only change we will make is in step 2, instead of choosing the *nearest* unseen out-neighbor, we will choose the **farthest** unseen out-neighbor (out-neighbor with highest edge weight). This will give us something called the Farthest Insertion heuristic.

It may seem like this idea is pointless – why would we be seeking vertices that have high weight when we are trying to minimize weight? For this, we need to see the flaw of the NN and NI approaches. These were greedy with respect to low-weight edges, but as a result, it may leave high-weight edges towards the end of the procedure. Now, with FI, we will be greedy with respect to high-weight edges, and ideally, as a result, it will leave low-weight edges later on. We are addressing the "outlying" vertices sooner.

Again, we will do a simulation, this time with FI:

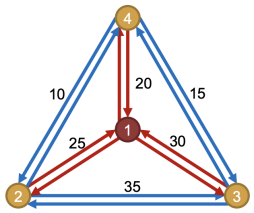
- We'll choose to start our tour at vertex 1. Our closed tour is currently $1 \to 1$
- The **furthest** unseen out-neighbor is 3 with weight 30. We add it to the tour in the only way we can, giving us a tour of $1 \to 3 \to 1$.
- The **furthest** unseen out-neighbor of vertices 1 and 3 is 2, since $w(2,3) = 35$. We now check for insertions, aiming to *minimize* the net weight added (just like NI!)
  - If we insert 2 after 1 (towards the tour $1 \to 2 \to 3 \to 1$), we would remove the edge $1 \to 3$ and replace it with $1 \to 2$ and $2 \to 3$. Thus, the net change in total weight would be $w(1,2)+w(2,3)-w(1,3) = 25+35-30 = 30$
  - If we insert 2 after 3 (towards the tour $1 \to 3 \to 2 \to 1$), the net change in weight would be $w(3,2)+w(2,1)-w(3,1) = 35+25-30 = 30$
  - The net change is tied between the two cases. We'll choose to insert it after 1, making our tour $1 \to 2 \to 3 \to 1$
- The **furthest** out-neighbor $1, 2, 3$ is 4. Recall our current tour is $1 \to 2 \to 3 \to 1$. We have three possible insertion locations:
  - If we insert 4 after 1, the net change in weight is $w(1,4)+w(4,2)-w(1,2) = 20+10-25 = 5$
  - If we insert 4 after 2, the net change in weight is $w(2,4)+w(4,3)-w(2,3) = 10+15-35 = -10$
  - If we insert 4 after 3, the net change in weight is $w(3,4)+w(4,1)-w(3,1) = 15+20-30 = 5$
  - The second option minimizes the net change, so we insert 4 after 2.
- FI terminates, and our tour is $1 \to 2 \to 4 \to 3 \to 1$ which yields weight $25+10+15+30 = 80$

6 A Small Note On Heuristics

The point of these heuristics were to build some techniques that moved away from some of the flaws of the naive greedy NN approach. Primarily, unlike NN, with NI and FI, we can modify the partial tour. As a drawback, NI and FI are more expensive than NN, but they tend to work better in practice. It turns out the FI works well in practice, but NI doesn't work so well.

7 Savings Heuristic

Let us see one more heuristic called the Savings Heuristic. For this, it will be a bit more helpful to view the undirected edges in our graph visual as two directed edges. For now, ignore the colored edges.



The Savings Heuristic is described as follows:

1. Pick any vertex x to be the "central vertex"
2. Start with $n-1$ subtours of the form $x \to v \to x$ for all $v \in V - x$
3. For each $(i,j) \in E$ with $i, j \neq x$, compute its **savings** $s(i,j)$:

$$s(i,j) = w(i,x) + w(x,j) - w(i,j)$$

4. Sort all edges in decreasing order of savings
5. Repeat the following until only one tour remains:
   - Let (i,j) be the next edge in sorted order
   - If edges (i,x) and (x,j) are in our subtours, **and** i and j are in different tours, then replace (i,x) and (x,j) by (i,j)

This heuristic kind of has the vibes of FI, where we are considering "bad" edges first. It differs in the process of building up the tours. Let's perform the simulation:

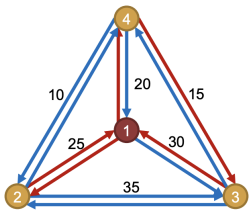
We will choose vertex 1 to be the central vertex, and so we have 3 subtours to start, those being: $1 \to 2 \to 1$, $1 \to 3 \to 1$, and $1 \to 4 \to 1$. These subtours are highlighted in red in the above diagram.

Next, we compute the savings of the remaining edges:

$$\begin{aligned}
s(2,3) &= w(2,1) + w(1,3) - w(2,3) = 25 + 30 - 35 = 20 \\
s(3,2) &= w(3,1) + w(1,2) - w(3,2) = 30 + 25 - 35 = 20 \\
s(2,4) &= w(2,1) + w(1,4) - w(2,4) = 25 + 20 - 10 = 35 \\
s(4,2) &= w(4,1) + w(1,2) - w(4,2) = 20 + 25 - 10 = 35 \\
s(3,4) &= w(3,1) + w(1,4) - w(3,4) = 30 + 20 - 15 = 35 \\
s(4,3) &= w(4,1) + w(1,3) - w(4,3) = 20 + 30 - 15 = 35
\end{aligned}$$

By sheer luck (/s) our edges happen to be sorted in order of savings. We will process these in **decreasing** order of savings. So the first edge we will consider is $(4,3)$.

For the edge $(4, 3)$ we check two things: Are $(4, 1)$ and $(1, 3)$ in our subtours? Are 4 and 3 in different subtours? The answer to both of these questions is yes, so we proceed to replace $(4, 1)$ and $(1, 3)$ with edge $(4, 3)$:

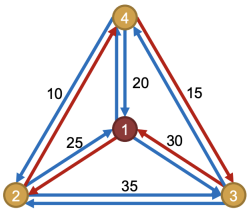


We now have two subtours being $1 \to 2 \to 1$ and $1 \to 4 \to 3 \to 1$.

We now proceed with the next edge: $(3, 4)$, and ask the same two questions. This time, the answer to "Are 3 and 4 in different subtours" is `No`, so we skip over this edge.

The next edge is $(4, 2)$. When we ask the question: "Are $(4, 1)$ and $(1, 2)$ in our subtours", we answer `No`, so this edge also gets skipped.

Next, we process $(2, 4)$, which passes both questions. We then replace edges $2 \to 1$ and $1 \to 4$ with the edge $2 \to 4$:



We are left with one tour, and so the algorithm terminates. The tour produced has weight 80.

7.1 A Comparison

We have seen NN, NI, FI, and now Savings. These heuristics have been tested extensively in practice. If we were to compare the four with respect to each other, the general conclusion is described as following:

	Worse Solution	Better Solution
Worse Runtime	NI	FI
Better Runtime	NN	Savings