MS&E 236 / CS 225: Lecture 4
Pointer networks for the traveling salesman problem, continued

Ellen Vitercik[*]

April 16, 2024

In these notes, we analyze how well the pointer network framework introduced last class performs on the traveling salesman problem (TSP). Before doing so, we will discuss several baselines, which include classic approximation algorithms for TSP.

1 Approximation algorithms for TSP

First, let's remember the notation we discussed last class. Our input is a set $X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$ representing, for example, cities on a map. Given a permutation $\pi : [n] \to [n]$, the length of the tour defined by π is

$$L(\pi \mid X) = \sum_{i=1}^{n-1} \left\| \boldsymbol{x}_{\pi(i)} - \boldsymbol{x}_{\pi(i+1)} \right\|_2 + \left\| \boldsymbol{x}_{\pi(n)} - \boldsymbol{x}_{\pi(1)} \right\|_2 .$$

Our goal is to compute a permutation, or tour, with the shortest length. In other words, our goal is to find a tour whose length is as close to $\mathrm{OPT}(X) = \min_\pi L(\pi \mid X)$ as possible. In this class, we'll see an algorithm that returns a tour π such that $L(\pi \mid X) \leq 2 \cdot \mathrm{OPT}(X)$. This algorithm is called a "2-approximation algorithm." (More generally, if $L(\pi \mid X) \leq \alpha \cdot \mathrm{OPT}(X)$ for some α and any X, then it's called an "α-approximation algorithm.")

1.1 Quick detour: minimum spanning trees

To get to the 2-approximation TSP algorithm, we will first take a quick detour and discuss *minimum spanning trees (MSTs)*. First, we define spanning trees as follows.

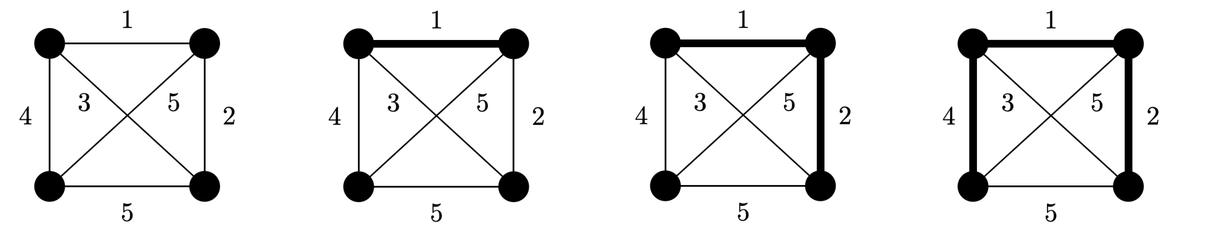
Definition 1.1 (Spanning tree)**.** Given a set of n nodes, a spanning tree is a set of edges $E \subseteq [n] \times [n]$ such that:

1. There are no cycles $i \to j \to k \to \cdots \to i$ with each edge $(i, j), (j, k), \dots$ in E.
2. Each pair of nodes $i, j \in [n]$ is connected using edges in E. In other words, there exists a path $i \to k \to \ell \cdots \to j$ with every edge $(i, k), (k, \ell), \dots$ in E.

[*]These notes are course material and have not undergone formal peer review. Please feel free to send me any typos or comments.

Algorithm 1 Kruskal's MST algorithm (on a complete graph)

```
Input: $X = \{\boldsymbol{x}_1, \dots, \boldsymbol{x}_n\}$
1: Define the initial set of edges $E \leftarrow \emptyset$
2: Define $L$ to be a list of edges between all nodes in $[n] \times [n]$ sorted in non-decreasing order of
   the distance $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$
3: while $E$ is not a spanning tree do
4:     Pop the edge $(i, j)$ with the highest weight from $L$
5:     if $(i, j)$ does not form a cycle in $E$ then
6:         $E \leftarrow E \cup \{(i, j)\}$
Output: MST $E$
```



(a) Initial graph (b) 1st edge added (c) 2nd edge added (d) 3rd edge added

Figure 1: Illustration of Kruskal's algorithm.

The MST is the spanning tree with the minimum total weight

$$\sum_{(i,j) \in E} \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 . \tag{1}$$

Kruskal's algorithm is a simple, classic algorithm for finding MSTs. See Algorithm 1[1] for the pseudo-code and Figure 1 for an illustration.

1.2 2-approximation algorithm for TSP

Algorithm 2 builds upon Kruskal's algorithm to compute a 2-approximate solution to TSP. We next prove that Algorithm 2 is indeed a 2-approximation algorithm.

Theorem 1.2. *Let π be Algorithm 2's output given input s. Then $L(\pi \mid X) \leq 2 \cdot OPT(X)$.*

Proof. Let $\mathsf{weight}(\text{MST})$ be the weight of the MST (as defined in Equation (1)) computed in Step 1 of Algorithm 2. Moreover, let $\mathsf{weight}(\text{walk})$ be the total weight of the walk computed in Step 2 of Algorithm 2. By construction,

$$\mathsf{weight}(\text{walk}) = 2 \cdot \mathsf{weight}(\text{MST}). \tag{2}$$

The algorithm computes π by shortcutting the walk computed in Step 2. Therefore,

$$L(\pi \mid X) \leq \mathsf{weight}(\text{walk}) \tag{3}$$

due to the triangle inequality, which guarantees that for any $i, j, k \in [n]$,

$$\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 + \|\boldsymbol{x}_j - \boldsymbol{x}_k\|_2 \leq \|\boldsymbol{x}_i - \boldsymbol{x}_k\|_2 .$$

[1]This pseudo-code is for Kruskal's algorithm on a complete graph in keeping with the TSP theme of the lecture.

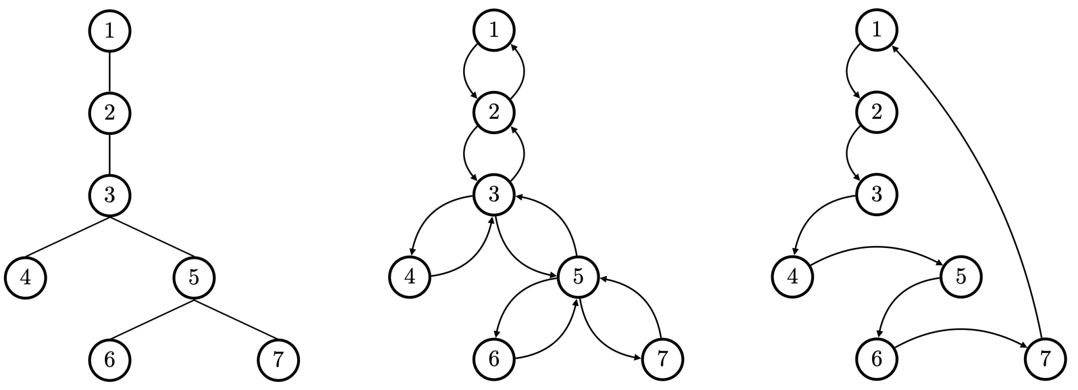

(a) Minimum spanning tree (b) Pre-order walk on MST (c) Shortcut of walk

Figure 2: Illustration of Algorithm 2. This illustrated example is from Section 2.2.2 of the CS 261 notes.

Algorithm 2 TSP 2-approximation algorithm

```
Input: $X = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$
1: Compute an MST of $X$ (Figure 2a).
2: Walk along the MST's edges, visiting each edge exactly twice (Figure 2b). In other words, visit
   the nodes in the order of a depth-first search, which is called a *pre-order walk*.
3: Compute a tour from the walk using shortcuts for nodes that are visited several times. For
   example, in Figure 2b, the walk $1 \to 2 \to 3 \to 4 \to 3 \to 5 \to \cdots$ becomes $1 \to 2 \to 3 \to 4 \to$
   $5 \to \cdots$ because node 3 is visited twice.
Output: The resulting tour
```

Combining Equations (2) and (3), we have that

$$L(\pi \mid X) \leq 2 \cdot \mathsf{weight}(\mathrm{MST}). \tag{4}$$

Next, if we delete one edge from the *optimal* TSP tour, we get a spanning tree T. Since Step 1 of Algorithm 2 computes a *minimum* spanning tree,

$$\mathsf{weight}(\mathrm{MST}) \leq \mathsf{weight}(T) \leq \mathrm{OPT}(X).$$

Combined with Equation (4), this means $L(\pi \mid X) \leq 2 \cdot \mathrm{OPT}(X)$, as claimed. □

The famous Christofides algorithm is a slightly more complicated variation of Algorithm 2, and it is a 1.5-approximation algorithm. See Section 2.2.2 of the CS 261 notes for a description of this algorithm. A bit of history: for 40 years, it was unknown if there was an algorithm with a better approximation bound than 1.5. In 2021, there was a massive breakthrough in theoretical computer science by Karlin et al. [2], who, for some $\epsilon > 10^{-36}$, gave a $1.5 - \epsilon$ approximation algorithm.

2 Performance of pointer networks for TSP

Now that we have an idea of the theoretical baselines, we will compare them against the policy that pointer networks learn. In the last class, we saw that pointer networks use LSTMs

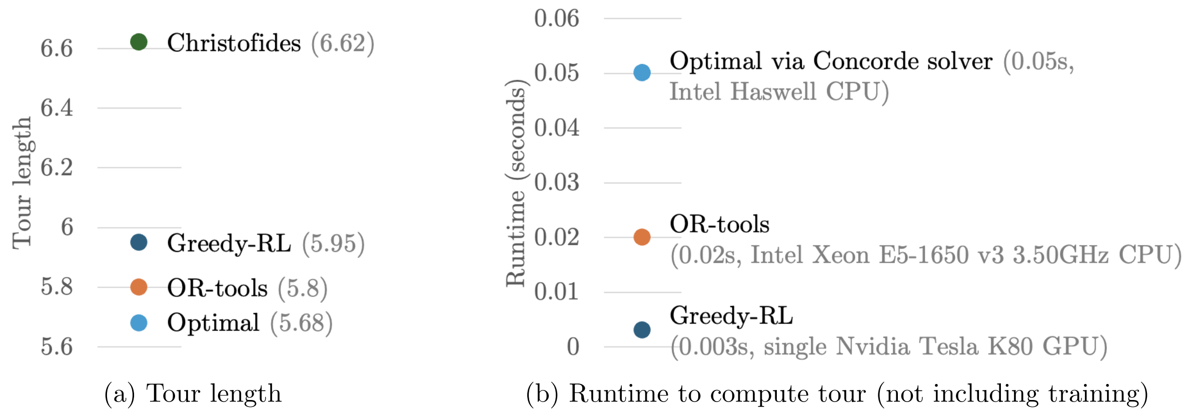


Figure 3: Performance of Greedy-RL when trained and tested on graphs with 50 nodes.

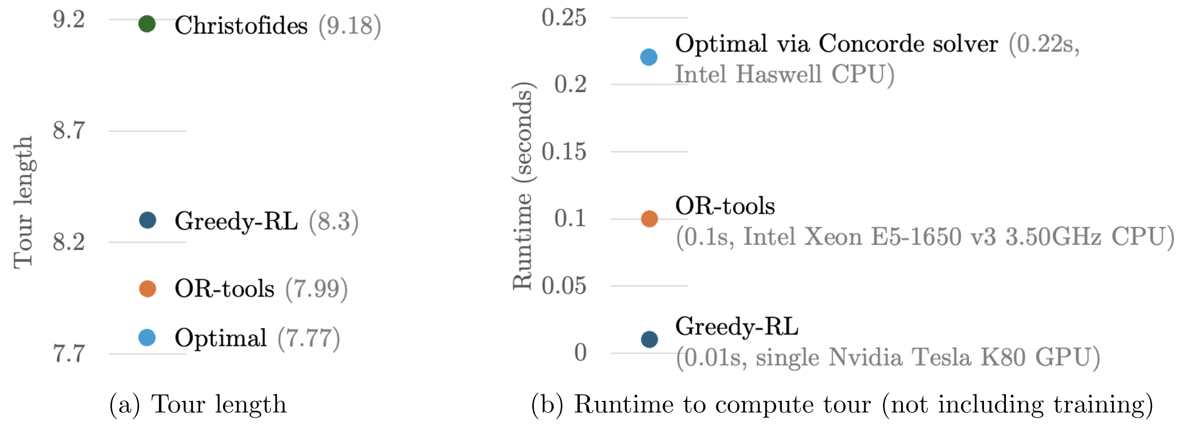


Figure 4: Performance of Greedy-RL when trained and tested on graphs with 100 nodes.

to compute $\mathbb{P}[\pi(i) = j \mid \pi(< i), X]$ for all $j \in [n]$. We will analyze two approaches to using the pointer network:

Greedy-RL: For the i^{th} cit in the tour, this approach chooses the city j that maximizes $\mathbb{P}[\pi(i) = j \mid \pi(< i), X]$. This approach is very fast to run, but the tour will likely not be very good because this approach isn't actually sampling from the distribution that the PN defines.

Sampling-RL: This approach samples many tours from the distribution that the PN defines, i.e.,

$$\mathbb{P}[\pi \mid X] = \prod_{i=1}^{n} \mathbb{P}[\pi(i) \mid \pi(< i), X],$$

and returns the tour with the shortest total length. Under this approach, the resulting tour will improve the more samples we take, but it is relatively slow to run.

To generate the inputs X, we will simply sample each point $\boldsymbol{x}_i$ uniformly from $[0, 1]^2$.

Figures 3, 4, and 5 show a subset of the results from the paper by Bello et al. [1][2]. In all figures, numbers are reported on average over a test set of 1000 instances. Figure 3

[2]We include the compute resources in the figures since that landscape has changed so much since 2017.

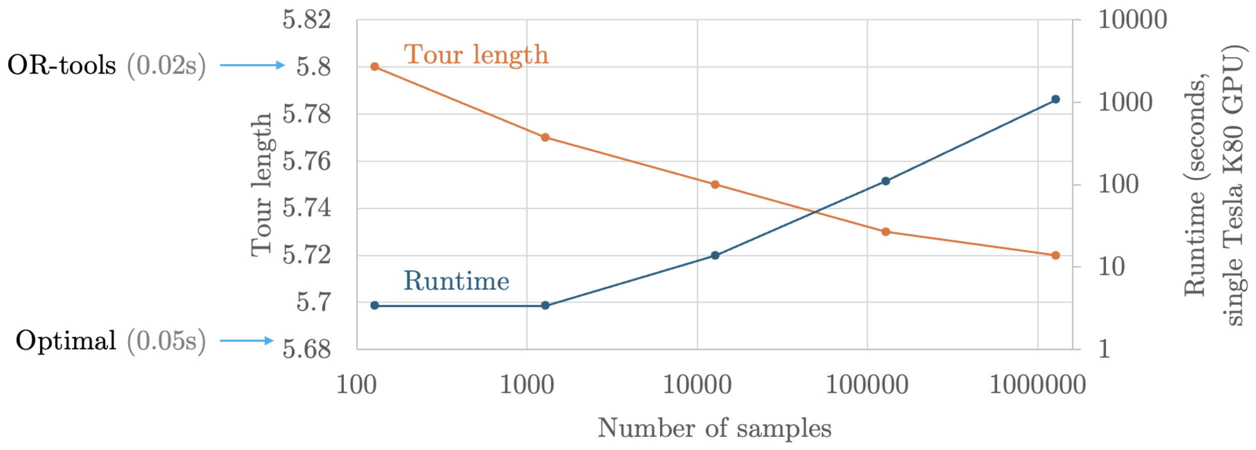


Figure 5: Performance of Sampling-RL when trained and tested on instances with 50 nodes as a function of the number of samples.

shows the performance of Greedy-RL when the PN is trained and tested on instances with 50 points, compared against the Christofides algorithm, a Google library for operations research problems called OR-tools, and the optimal tour, which is computed using the Concorde solver. The takeaway is that the tours that Greedy-RL produces have lengths that are competitive with specialized heuristics (namely, OR-tools), but Greedy-RL is significantly faster to run. We can glean a similar takeaway from Figure 4, where the PN is trained and tested on instances with 100 points.

Figure 5 shows the performance of Sampling-RL when trained and tested on instances with 50 nodes as a function of the number of samples. We can see that the tour length decreases as the number of samples grows, but the runtime is extremely large and just not worth it, given that Concorde takes about 0.05 seconds on average to find an optimal tour.

References

[1] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *Workshop track of the International Conference on Learning Representations (ICLR)*, 2017.

[2] Anna R Karlin, Nathan Klein, and Shayan Oveis Gharan. A (slightly) improved approximation algorithm for metric TSP. In *Proceedings of the Annual Symposium on Theory of Computing (STOC)*, 2021.