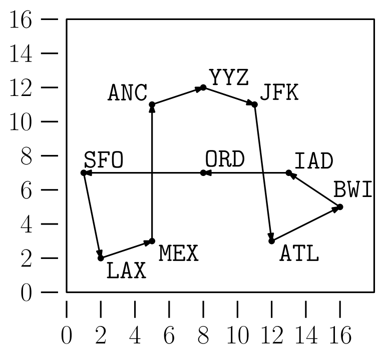
CMSC 420: Lecture X03
Supplemental: TSP Heuristics

Overview: In this lecture, we will discuss the implementation of heuristics for computing traveling salesperson (TSP) tours.

Traveling Salesperson Problem: The TSP problem formally stated is, "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" This is a well-known NP-Hard problem, even when the points (cities) are points in the plane and the distance is the Euclidean distance. To make testing easier, we will consider squared Euclidean distance, to avoid issues caused by round-off errors when computing square roots in the standard Euclidean distance.

Given an n-element point set $P = \{p_0, \ldots, p_{n-1}\}$ in $\mathbb{R}^2$, a *tour* is defined to be a permutation of these points $T = \langle p_{i_0}, p_{i_1}, \ldots, p_{i_{n-1}} \rangle$. To simplify notation, we omit the double subscripts and just write a tour as $T = \langle p_0, p_1, \ldots, p_{n-1} \rangle$, but it is understood that the points of P may appear in any order within this list. A *edge* of the tour is a consecutive pair of elements (p_i, p_{i+1}), for $0 \leq i \leq n-1$. We take indices modulo n, so there is an edge (p_{n-1}, p_0).



```
BWI, IAD, ORD, SFO, LAX, MEX, ANC, YYZ, JFK, ATL
cost = 292
```

Fig. 1: A tour for a set of points.

The *squared measure* of a tour, denoted $D^{[2]}(T)$ is the sum of the square lengths of the edge in the tour

$$D^{[2]}(T) \;=\; \sum_{i=0}^{n-1} \operatorname{dist}^2(p_i, p_{i+1}).$$

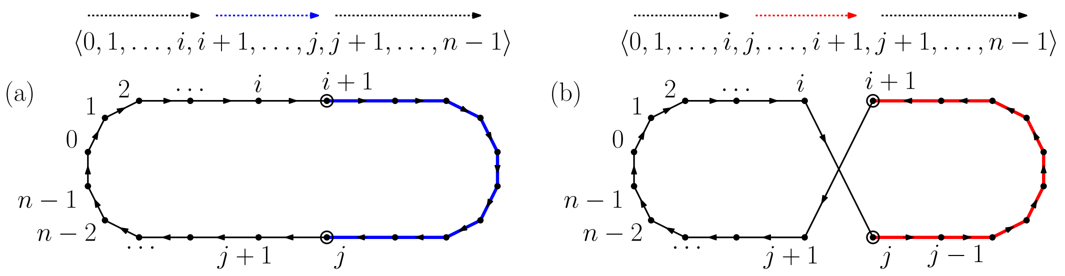
Thus, the TSP problem involves computing the tour (among all $n!$ permutations) that minimizes this measure. There are a variety of algorithms for solving the TSP problem. Exact algorithms are often based upon techniques such as *branch and bound* and *cutting planes*, which avoid brute-force search by focusing on the most promising regions of the solution space. There are well-known approximation algorithms, such as Christofides algorithm (which achieves a $1.5\times$ approximation factor) and $(1+\varepsilon)$ approximation algorithms by Arora and Mitchell, which employ dynamic programming combined with a quadtree-like decomposition.

We will consider simple heuristics based on *local search.* This approach starts with a feasible (but not optimal solution) and attempts to improve it through small local changes. We will focus on perhaps the best-known local search technique, called *2-Opt.*

2-Opt works by swapping two edges out of the tour and swapping two edges in. Bentley observed that most 2-Opts do not lead to improved solution, and he proposed a smart way to select 2-Opts through the use of fixed-radius nearest neighbor searching.

Reversals and 2-Opt: The 2-Opt operation is based on reversing subtours. Given a tour $T = \langle p_0, p_1, \ldots, p_{n-1}\rangle$, and given any two indices i and j, where $0 \leq i < j \leq n-1$. We can modify a tour by reversing the sublist from indices $i+1$ through j. Let's call this operation reverse(i,j). (**Note that this is different from Programming Assignment 1!**)

This has the effect of replacing two edges (p_i, p_{i+1}) and (p_j, p_{j+1}) with the edges (p_i, p_j) and (p_{i+1}, p_{j+1}), and reversing the path from p_{i+1} through p_j (see Fig. 2).



$\langle 0, 1, \ldots, i, i+1, \ldots, j, j+1, \ldots, n-1\rangle$
(a)
0
1
2
$\ldots$
i
$i+1$
$n-1$
$n-2$
$\ldots$
$j+1$
j
$\langle 0, 1, \ldots, i, j, \ldots, i+1, j+1, \ldots, n-1\rangle$
(b)
0
1
2
$\ldots$
i
$i+1$
$n-1$
$n-2$
$\ldots$
$j+1$
j
$j-1$

Fig. 2: The operation reverse(i,j).

Note that this operation is not defined when $i = j$, but we can generalize it to any pair $i \neq j$ by performing reverse$(\min(i,j), \max(i,j))$. Note also that the operation has no effect on the tour when p_i and p_j are adjacent in the tour (since it has the effect of reversing a subtour consisting of a single point). A couple of examples are shown in Fig. 3.

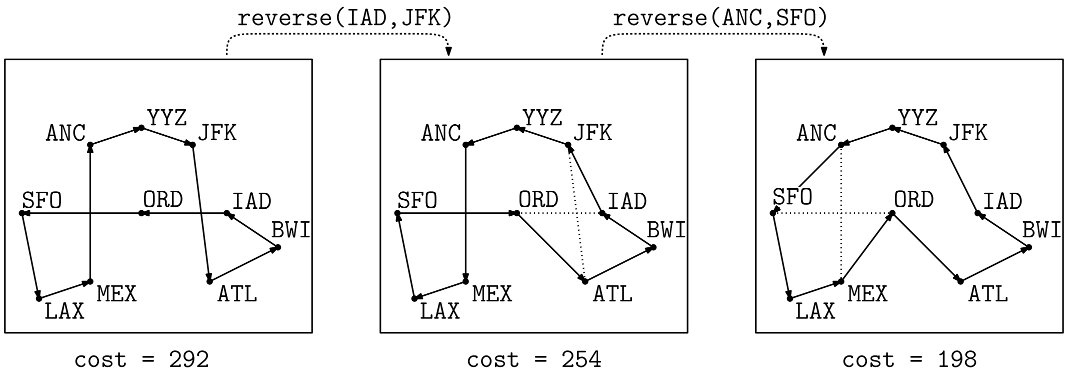


Fig. 3: Examples of reversals.

We are only interested in reversals that (strictly) reduce the cost of the tour. Define the change in the measure to be:

$$\Delta(i,j) = (\text{dist}^2(p_i, p_j) + \text{dist}^2(p_{i+1}, p_{j+1})) - (\text{dist}^2(p_i, p_{i+1}) + \text{dist}^2(p_j, p_{j+1}))$$

For $0 \leq i, j \leq n-1$ $(i \neq j)$, we define 2-Opt(i,j) as follows. If $\Delta(i,j) < 0$, perform reverse(i,j), and otherwise the tour is unchanged. If the reversal is performed, we say that the operation is *effective*.

Limiting 2-Opts: There are $O(n^2)$ possible 2-Opts that could be attempted on a tour. If the tour is close to optimal, the vast majority of these operations will not have any effect on the tour. Bentley (whom you might remember invented the kd-tree) proposed a simple way to filter out 2-Opts that are clearly ineffective. Consider an point p_i in the tour. In order for the operation 2-Opt(i, j) to be effective, one of the newly added edges, say (p_i, p_j), should be shorter than at least one of the two edges are being replaced. Using this observation, he proposed that the other point p_j should be closer to p_i than its successor p_{i+1}. In other words, p_j lies within a ball of radius dist(p_i, p_{i+1}) centered at p_i. Such a point is called a *fixed-radius near neighbor.*

In his TSP heuristic, Bentley tried 2-Opts with all such points. We will implement a more limited variant of Bentley's proposal by computing just the closest point to p_i (excluding p_i itself) that lies within the given distance.

Let's define the *fixed-radius nearest neighbor query* (or *FRNN query*) as follows for a point set P. Given a query point q and a radius r, the problem is to compute the closest point $p_j \in P$ to q, assuming that $0 < \text{dist}(q, p_j) < r$ (see Fig. 4(a)). (The reason for demanding that the distance be strictly larger than zero is that we don't want to return the same point that made the query.) If there is no point of P within this distance range, the query returns `null` (see Fig. 4(b)).

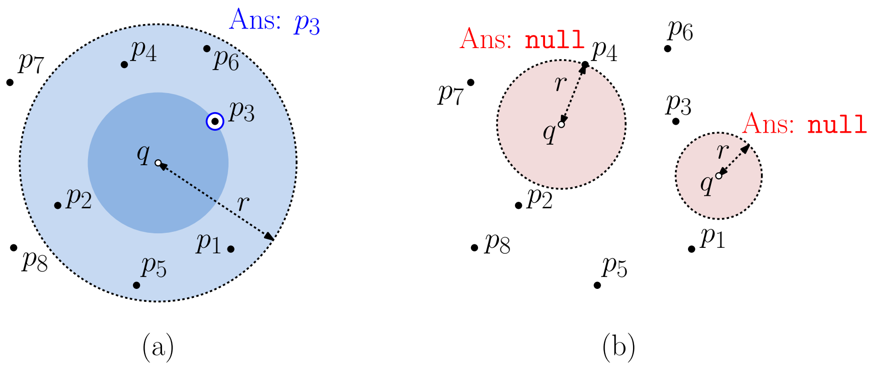


Fig. 4: Fixed-radius nearest-neighbor queries.

Given $0 \leq i \leq n - 1$, we define the operation 2-Opt-NN(i) as follows. First, we invoke a fixed-radius nearest neighbor query with the center point p_i and radius dist(p_i, p_{i+1}). (Recall that indices are taken modulo n.) If the query returns `null`, we do nothing. Otherwise, letting p_j denote the result of the query, we perform the operation 2-Opt(i, j). Note that this operation may be ineffective (depending on the other distances involved). If the point p_j exists and the 2-Opt is effective, then p_j is returned. Otherwise, the operation returns `null`.

An example is shown in Fig. 5. We first find the closest point to p_i lying within the ball whose radius is the distance to p_{i+1}. The point p_j is the closest. We then invoke 2-Opt(i, j), which reverses the subpath from p_{i+1} to p_j. In this case, the square measure decreases, so the operation is effective.

Implementing FRNN Queries: Let's next consider how to efficiently implement fixed-radius nearest neighbor queries using the `WKDTree` data structure? We define a recursive helper function

```
LPoint fixedRadNN(Point2D center, double sqRadius, LPoint best)
```

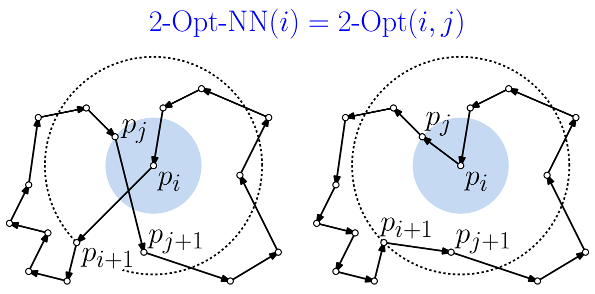


Fig. 5: 2-Opt-NN operation.

where `center` is the center of the disk, `sqRadius` is its squared radius, and `best` is the best point we have seen so far. At the top level, we invoke this function on the root of the tree with `best` set to `null`.

If we arrive at an internal node, we compute the squared distance of the wrapper to `center`. If it is greater than or equal to `sqRadius`, then the node cannot provide viable point, and we return the current value of `best`. Otherwise, we compare the squared distance from the wrapper to `center` against the squared distance from `best` to `center` (assuming `best` is not `null`). If the wrapper distance is strictly greater than the best distance, then again we can return `best`. Otherwise, we invoke the helper function recursively on each of the node's children, and update `best` accordingly (e.g., `best = left.fixedRadNN(..., best)`).[1] Finally, we return `best`.

If we arrive at an external node, we compute the squared distance of the point stored in this node to the `center`. If this squared distance is:

- Equal to zero, or
- Greater than or equal to `sqRadius`, or
- Greater than the squared distance from the query point to `best` (assuming `best` is non-null)

then there is no change and we return `best`. Otherwise, we return the point in this node as the new `best`. (But see the next paragraph in case the squared distances are equal.)

What if there are multiple candidates for the nearest neighbor? For the sake of consistency, let's agree that the point to be selected is the one that is lexicographically smallest with respect to its coordinates.[2] That is, among all nearest neighbors, its x-coordinate should be the smallest, and among all that have the same x-coordinate, the y-coordinate should be the smallest. (This could happen for example for the point set in Fig. 1. The points `MEX`, `IAD`, `ANC`, `YYZ`, `JFK` are all equidistant from `ORD`. Among these points, `MEX` has the lowest coordinates lexicographically, and so it would be chosen had we invoked a fixed-radius nearest neighbor search on `ORD`.

[1]As a further optimization, we could invoke the helper function first on the child that is closer to the `center`, since that child is more likely to provide a closer point. (We did not implement this in our version, and so you do not need to do so either.

[2]We do not expect many test cases to check for this condition, so in your first pass, you might ignore this issue. The input file `test05-input.txt` has an instance where there are multiple candidates for the nearest neighbor.