CMSC 420: Spring 2021

Programming Assignment 3: Efficient TSP Heuristics

Handed out: Tue, Mar 27. Due: **Tue, May 11, 11pm**. (Submission via Gradescope.)

Overview: In this assignment we will combine our extended AA tree and wrapped kd-tree data structures to implement a data structure for maintaining traveling salesperson (TSP) tours. We are given a discrete set of points P in $\mathbb{R}^2$. Recall from Programming Assignment 0 that a *tour* of P is a cycle that visits all the points of P exactly once.

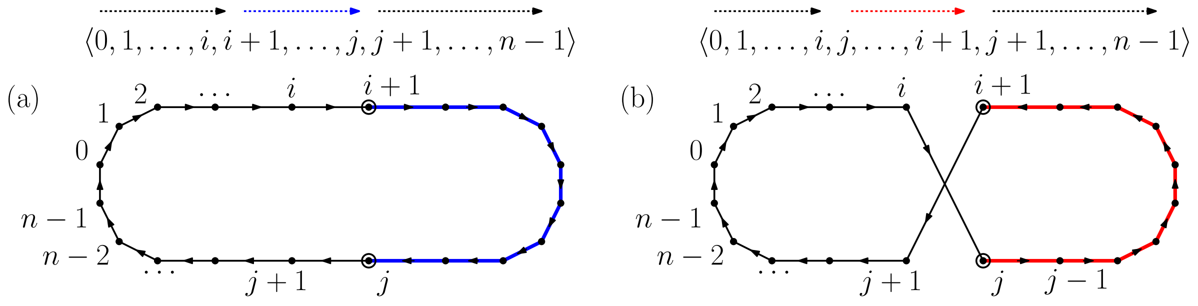
Squared Measure: The (standard) Euclidean TSP problem involves computing the tour over P of the minimum total Euclidean length. Euclidean distances involve square roots, and this results in rather unpredictable round-off errors. We will instead consider a variant of this problem, for the sake of easier testing.

Let T be a tour of P, and let $\langle p_0, p_1, \ldots, p_{n-1}\rangle$ denote the sequence of points along the tour. Define the *squared measure* of the tour, denoted $D^{[2]}(T)$ to be the sum of the squared distances of the edges of the tour, that is,

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

Throughout, indices are taken modulo n, so $p_n = p_0$. (Note that this is different from taking the square of the standard TSP length.) The squared measure has the advantage that if the coordinates P's points are all integers, then $D^{[2]}(T)$ is an integer. This is not generally true for the standard TSP measure.

Modifying Tours: Recall from Programming Assignment 0 that we can represent any tour as a list of points, and we can modify a tour by reversing a sublist. Given any two indices i and j, where $0 \le i < j \le n-1$. We can modify a tour by reversing the sublist from indices $i+1$ through j. (**Note:** We have changed the indexing slightly from Programming Assignment 0. This is a bit cleaner and more consist with established practice .) Let's call this operation reverse(i, j). This has the effect of replacing two edges $(i, i+1)$ and $(j, j+1)$ with the edges (i, j) and $(i+1, j+1)$, and reversing the path from $i+1$ through j (see Fig. 1).



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

Figure 1: 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$(\min(i, j), \max(i, j))$.

Because the reversal only changes two edges, the change in cost is the difference between the squared lengths of the two new edges minus the square lengths of the original edges. Define the change in the measure to be:

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

Using this, we define a few other heuristics for modifying a tour:

2-Opt: Some reversals reduce the overall cost and some do not. Given a pair $0 \leq i, j \leq n-1$, where $i \neq j$, the operation 2-Opt(i, j) that conditionally performs a reversal if the squared measure decreases strictly. In particular, it first checks whether $\Delta(i, j) < 0$, and if so, it performs reverse(i, j). Otherwise, the tour is unchanged.

2-Opt-NN: There are clearly $O(n^2)$ possible 2-Opts that could be attempted on a tour. If the tour is close to optimum, the vast majority of these operations will not have any effect on the tour. How can we focus attention to 2-Opts that are most likely to reduce the tour's cost? There many different heuristics that could be applied. One idea for identifying 2-Opts that are likely to be effective is to let p_j be the closest point to p_i, assuming that it is closer than the neighbor p_{i+1} it replaces.

This gives rise to an important geometric query called a *fixed-radius nearest neighbor*. We will define this operation in the strict sense. Given a point set P and a query point q and a radius r, the problem is to compute the closest point of $p_j \in P$ to q, assuming that dist$(q, p_j) < r$ (see Fig. 2(a)). If there is no point of P within this distance bound, the query returns `null` (see Fig. 2(b)).

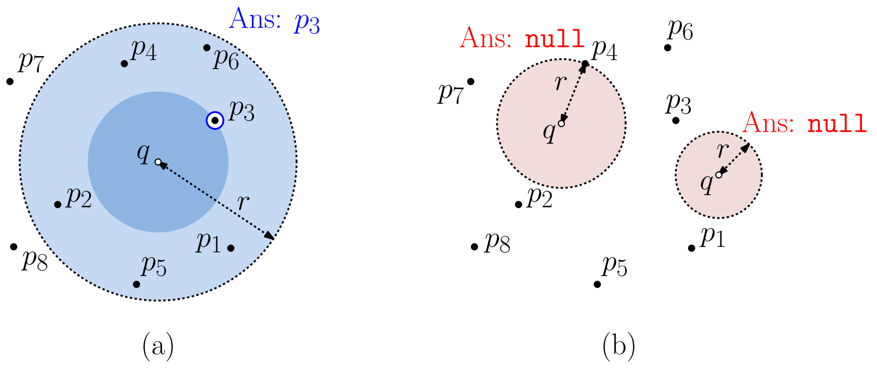


Figure 2: Fixed-radius nearest-neighbor queries.

In a 2-Opt-NN operation, we perform the operation 2-Opt(i, j), where p_j is the closest point to p_i. However, we only want to do this if the point p_j is closer than p_i's current successor p_{i+1} (indices taken modulo n). To find p_j we perform a fixed-radius nearest neighbor query for the query point $q = p_i$ and search radius.[1] $r = \mathrm{dist}(p_i, p_{i+1})$. If we

[1]You might wonder why we need the radius constraint as part of the query. Why not just compute q's nearest neighbor, and then check afterwards whether the distance is smaller than r? The reason is that r is typically quite small, and there may be very few points lying within the query range. Thus, the radius constraint can significantly improve the query's efficiency.

receive a non-null result, p_j, we then perform 2-Opt(i,j). If the result is `null`, the tour is unchanged.

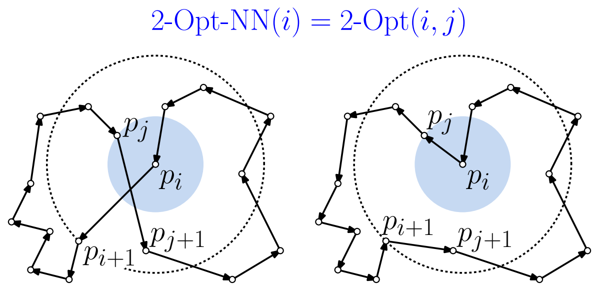


Figure 3: 2-Opt-NN operation.

We need to make one modification to the fixed-radius NN query. Clearly, the closest point p_i is p_i itself (not very useful!). So, the operation 2-Opt-NN(i) is defined formally as follows. First, apply a fixed-radius NN query with $q = p_i$, maximum distance $r = \text{dist}(p_i, p_{i+1})$, and ignoring p_i itself (or equivalently, any points at distance 0). Then perform 2-Opt(i,j).

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.

All 2-Opt: In contrast to 2-Opt-NN, which performs 2-Opt on a judiciously chosen pair, this operation is pure brute force. It iterates through all indices i from 0 to $n-1$, and for all j from $i+1$ to $n-1$, and performs 2-Opt(i,j) for each pair. Thus, in total there are $\binom{n}{2} = O(n^2)$ instances of 2-Opt being performed.

Tour Object: In this assignment, you will implement a data structure, called `Tour`, that will maintain a tour for a set of points. It supports a number of operations, as described below. The data structure will be templated with the point type, which is any class that implements the Java interface `LabeledPoint2D`, as described in the file `LabeledPoint2D.java` from the provided skeleton code. A labeled point is a 2-dimensional point (`Point2D` from the skeleton code) that supports an additional function `getLabel()`. This returns a string associated with the point.

Each tour object will store three principal data elements:

Tour: This is a tour itself, that is, a list (e.g., Java `ArrayList`) containing the points (`LPoint`) of the tour.

Locator: This structure is used for locating the index of an airport in the tour from its code (e.g., "`LAX`"). It is a dictionary (implemented as an `AAXTree`) storing key-value pairs, where the keys are strings and the values are indices (represented as a Java `Integer`).

[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.

The index for a given string gives the index of the corresponding labeled point in the tour. As with Programming Assignment 0, whenever we insert a new point into our tour, we need to record its location, and whenever we move a point to a new index (e.g., through reversal), we need to update its location.

At a minimum, the locator will need to support the operations of `insert`, `find`, `clear`, and a new operation called `replace` (which was not required in Programming Assignment 1). The `replace` operation has the following signature:

```
void replace(Key x, Value v) throws Exception
```

It is given a key `x` (that is, an airport code) and an associated value `v` (that is, an index in the tour). It searches for `x`. If it is not found, it throws an exception with the error message `"Replacement of nonexistent key"`. Otherwise, the value associated with this entry is changed to `v`.

Spatial Index: This is a 2-dimensional spatial index (implemented as a `WKDTree`) storing the points (`LPoint`). At a minimum, it must support the operations `insert`, `find`, `clear`, and a new operation called `fixedRadNN` (which was not required in Programming Assignment 2). The `fixedRadNN` operation has the following signature:

```
LPoint fixedRadNN(Point2D q, double sqRadius)
```

It returns a reference to the fixed-radius nearest-neighbor query to q, where the squared radius of the disk is `sqRadius`. Among the points whose squared distance to q is strictly more than zero and strictly less than `sqRadius`, it returns the closest to q. If there is no such point in the disk, it returns `null`. If there are multiple points at the same distance, your function should return that is lexicographically smallest in terms of its x and y coordinates. (That is, if two points are at each distance to q, prefer the one with the smaller x-coordinate. If both have the same x-coordinate, prefer the one with the smaller y-coordinate.)

Tour Operations: The Tour object should support the following public functions.

`Tour()`: Initializes an empty tour, creating the tour, locator, and spatial index (all empty).

`void append(LPoint pt) throws Exception`: Appends the labeled point `pt` to the end of the tour. If there exists a point with this label, an exception with the error message "`Duplicate label`" is thrown. If there already exists a point with the same coordinates, an exception with the error message "`Duplicate coordinates`" is thrown. Otherwise, the point is added to the tour, its index is added to the locator, and the point is added to the spatial index.

`ArrayList<LPoint> list()`: This returns a Java `ArrayList` containing all the points of the tour in order.

`void clear()`: The clears everything: the tour, the locator, and the spatial index.

`double cost()`: The returns the current squared measure of the tour. For the sake of consistency and accuracy, you should perform all arithmetic operations using `double` variables, and use the `Point2D` function `distanceSq` to compute distances between points.

`void reverse(String label1, String label2) throws Exception`: This begins by locating the indices i and j for the tour points with labels `label1` and `label2`, respectively.

If either label is not found in the locator, an exception with the error message "`Label not found`" is thrown. If $i == j$ (or equivalently, if the labels are equal), an exception with the error message "`Duplicate label`" is thrown. Otherwise, the operation reverse(i, j) is performed on the tour. (It may be that $i < j$ or $j < i$. Your function should work correctly in either case.)

`boolean twoOpt(String label1, String label2) throws Exception:` This is the same as `reverse` above, but after checking the validity of the arguments, instead of reverse, the operation 2-Opt(i, j) is performed on the tour. That is, we check whether $\Delta(i, j) < 0$ (note that the inequality is strict), and if so, we perform reverse(i, j). If the reversal is performed, the operation is said to be *effective*. If the operation is effective, we return `true`, and otherwise we return `false`.

`LPoint twoOptNN(String label) throws Exception:` This first locates the index i for the tour point with label `label`. If this label is not found in the locator, an exception with the error message "`Label not found`" is thrown. Otherwise, the operation 2-Opt-NN(i) is performed on the tour. That is, we invoke `fixedRadiusNN(q,rsq)` where $q = p_i$ and $rsq = \text{dist}^2(p_i, p_{i+1})$ (where p_{i+1} is the point immediately following p_i in the tour). If it returns `null`, then we return `null`. Otherwise, let p_j denote the result. We invoke 2-Opt(i, j). If it is effective, then we return a reference to the point p_j. Otherwise, we return `null`.

`int allTwoOpt():` This performs the operation all-2-Opt() on the tour. (For consistency in testing, this should be done exactly as described for all i from 0 to $n - 1$ and all j from $i + 1$ to $n - 1$, performing 2-Opt(i, j).) Among the $\binom{n}{2}$ 2-Opts performed, return the number that were effective.

Hint on Helpers: In order to implement the above functions, you may define whatever local helper functions you like. The functions above take labels as inputs, but it is more natural to work with tour indices. We would recommend that for each of the above label-based functions, you have a local index-based function to perform the actual operation. For example, the helper for `reverse` might be called `void reverseHelper(int index1, int index2)`, where `index1` and `index2` are the indices in the tour for the respective labels. The advantage of doing this is that your other helper functions can easily invoke one another.

Doubles not Floats: For some of the larger test cases we are planning to use, the number of digits in the tour costs will be too large to store in a single float variable. For the sake of testing, we have converted all the instances of `float` in the supporting classes (e.g. `Airport.java`, `Point2D.java`, `Rectangle2D.java`) to be of type `double`. You may need to make a similar change in your `WKDTree.java` to keep the compiler from complaining.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. We will also provide canonical versions of our `AAXTree` and `WKDTree` implementations. (Note that you will still need to add the new functions `replace` and `fixedRadiusNN`.)

```
package cmsc420_s21;

public class Tour<LPoint extends LabeledPoint2D> {
    public Tour() { /* you fill these in */ }
```

```
        public void append(LPoint pt) throws Exception { /* ... */ }
        public ArrayList<LPoint> list() { /* ... */ }
        // ... and so on
    }
```

Efficiency requirements: The new `AAXTree` operation `replace` should run in $O(\log n)$ time. The new `WKDTree` operation `fixedRadNN` should be reasonably efficient, in the sense that the code should check each node's wrapper. If the wrapper for some node does not overlap the query disk or is farther away than the closest point seen so far, it should not recursively visit a node's children.

Testing/Grading: As before, we will be using Gradescope's autograder and JUnit for testing and grading your submissions. You can just drag your files `AAXTree.java`, `WKDTree.java`, and `Tour.java` into the Gradescope upload window. You may include other files, but note that the files given in the skeleton code (e.g., `Point2D.java`, `Rectangle2D.java`, and so on) will be overwritten by the autograder. So, there is nothing to be gained by modifying these files.

As always, we will provide some sample test data and expected results along with the skeleton code. Note that some portion (up to 20%) of the final grade will be based on hidden tests.