# The Traveling Salesman

Programming Heuristics

**An Improved Data Structure for TSP**

When thinking about this problem, we care about the *connection* or *link* between cities. Knowing that each city will be represented as a Point that contains geographical information, we can conceive a way to compute the distance between cities. We can do all of that without any regard of an index for each city.

If we are concerned with links or connections between points and have no need for an index, then a *linked list* is a much more natural and clever data structure to utilize than an array!

---

**Algorithms for Approximating TSP**

The TSP is a notoriously difficult combinatorial optimization problem. In principle, you can enumerate all possible tours and pick the shortest one; in practice, the number of tours is so staggeringly large (nearly *N* factorial) that this approach is useless. For large *N*, no one knows of an efficient method that can find the shortest possible tour for any given set of points. However, many methods have been studied that seem to work well in practice, even though they do not guarantee to produce the best possible tour. Such methods are called *heuristics*.

Your main task is to implement the nearest neighbor and smallest increase insertion heuristics for building a tour incrementally, one point at a time. Start with a one-point tour (from the first point back to itself), and iterate the following process until there are no points left:

- **Nearest Neighbor** - Read in the next point and add it to the current tour after the point to which it is closest. (If there is more than one point to which it is closest, insert it after the first such point you discover.)

- **Smallest Insertion** - Read in the next point and add it to the current tour after the point where it results in the least possible increase in the overall tour length. (If there is more than one point, insert it after the first such point you discover.)

**Grading Standards**

**Standards-based Grading (percentage of indicated max points):**
Outstanding – 100%, Excellent – 95%, Acceptable – 87%, Unacceptable – 75%, No

1. I can produce a complete and professional project plan. (20 pts)
2. I can produce an externally correct solution to each of the required methods demonstrated by test case pass rate. (40 pts)
3. I can produce an internally correct solution by providing a complete class header, using quality commenting in my code, using good semantics, cleanly and correctly applying linked lists to all my method solutions, and not using concepts we will be discussing in later units. (20 pts)
4. I can produce a complete and professional project reflection. (20 pts)

**Point Data Type**

You are provided with Point.java. **DO NOT MODIFY THIS CLASS AT ALL!** It adheres to the following API.

```
public class Point {

    // creates the point (x, y)
    public Point(double x, double y)

    //returns the Euclidean distance between this point and that point
    public double distanceTo(Point that)

    // draws this point to standard drawing
    public void draw()

    // draws the line segment between this point and that point
    public void drawTo(Point that)

    // returns a string representation of this point
    public String toString()
}
```

**Tour Data Type**

Create a Tour data type that represents the sequence of points visited in a TSP tour. Represent the tour as a *circularly linked list* of nodes, one for each point in the tour. Each Node contains two references (fields): one to the associated Point and the other to the next Node in the tour. Within Tour.java, define an inner class Node in the standard way:

```
private class Node{

     private Point p;
     private Node next;
}
```

Your Tour data type must implement the following API.

```
public class Tour {

    // creates an empty tour
    public Tour()

    // creates the 4-point tour a->b->c->d->a (for debugging)
    public Tour(Point a, Point b, Point c, Point d)

      // returns the number of points in this tour
    public int size()

    // returns the length of this tour
    public double length()

    // returns a string representation of this tour
    public String toString()

    // draws this tour to standard drawing
    public void draw()

    // inserts p using the nearest neighbor heuristic
    public void insertNearest(Point p)

    // inserts p using the smallest increase heuristic
    public void insertSmallest(Point p)

   //tests this class by directly calling all constructors and methods
    public static void main(String[] args)
}
```

**String representation format**. The toString() returns a string containing the points, one per line, by (implicitly or explicitly) calling the toString() method for each point, starting with the first point in the tour.

**Standard drawing format**. The draw() method draws the tour to standard drawing by calling the drawTo() method for each pair of consecutive points. It must produce no other output.

**Corner cases**
You may assume that no Point argument is null.

**Possible Progress Steps**

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. Study the Point API. In this assignment, you will use the constructor to create a point, the toString() method to print it, the distanceTo() method to compute the distance between two points, and the drawTo() method to draw a line segment connecting two points.

2. Move to Tour.java. Include the standard linked-list data type Node as an inner class. Include one instance variable, say first or home, of type Node that is a reference to the "first/home" node of the circularly linked list. Also, for the sake of efficiency, include an instance variable called size. Both of these instance variables should of course be private.

3. To assist with debugging, define a constructor that takes four points as arguments and constructs a circularly linked list using those four points. First, create four Node objects and assign one point to each. Then, link the nodes to one another in a circular manner.

4. Implement the length() method. It should traverse each Node in the circularly linked list, starting at first/home, and incrementing a variable by accessing two successive points in the tour and call the distanceTo() method from the Point data type. It is important to think about this method carefully, because debugging linked-list code is notoriously difficult and frustrating. With circularly linked lists, the last node in the list points back to the first node, so watch out for infinite loops.

   Test the length() method on the 4-point tour in main—the length should be 1600.0.

   ```
   //print the tour length to standard output
   int length = squareTour.length();
   System.out.println("Tour length = " + length);
   ```

5. Implement the toString() method. It is similar in structure to length(), except that you must create a StringBuilder object (a String that is repeatedly added onto) and append each Point to the StringBuilder object.

   Test the toString() method on the 4-point tour in main.

   ```
   // print the tour to standard output
   System.out.println(squareTour);
   ```

   You should get the following output:

   ```
   (100.0, 100.0)
   (500.0, 100.0)
   (500.0, 500.0)
   (100.0, 500.0)
   ```

6. Implement the draw() method. It is very similar to length(), except that you will need to call the drawTo() method from the Point data type. You will also need to include the statements

```
StdDraw.setXscale(0, 600);
StdDraw.setYscale(0, 600);
```

to resize the x- and y-coordinates of standard drawing.

Test the draw() method on the 4-point tour—you should see a square.

**Note for all text files in the input directory that the first line of text contains the maximum x- and y-coordinates for the text file. You can safely assume that your scales will range from 0 to xMax and 0 to yMax.**

---

Congratulations on reaching this point: writing linked-list code is always a challenge!

7. Implement insertNearest(). To determine which node to insert the point p after, compute the Euclidean distance between each point in the tour and p by traversing the circularly linked list.

   As you proceed, store the node containing the closest point *and* its distance to p. After you have found the closest node, create a node containing p, and insert it *after* the closest node. This involves changing the next field of both the newly created node and the closest node.

   As a check, here is the resulting tour for the 10-point problem which has length 1566.1363. Note that the optimal tour has length 1552.9612 so this rule does not, in general, yield the best tour.

8. After doing the nearest insertion heuristic, you should be able write the insertSmallest() method by yourself, without any hints. The only difference is that you want to insert the point p where it will result in the least possible increase in the total tour length. As a check, the resulting tour length for tsp10.txt is 1655.7462. In this case, the smallest insertion heuristic actually does worse than the nearest insertion heuristic (although this is not typical).

**Input and Testing**

The input format begins with two integers, width and height, followed by pairs of x- and y-coordinates. All x-coordinates will be real numbers between 0 and width; all y-coordinates will be real numbers between 0 and height. For example, tsp1000.txt contains the following data:

800 800
185.0411 457.8824
247.5023 299.4322
701.3532 369.7156
563.2718 442.3282
...
254.9820 302.2548

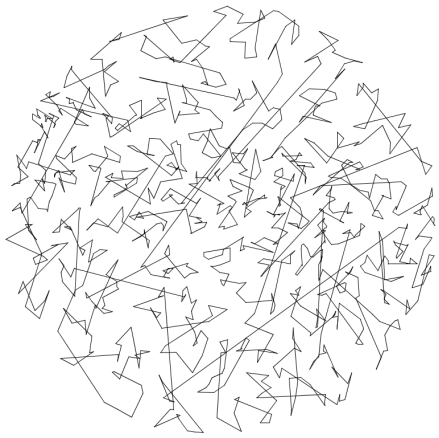
After implementing Tour.java, use the client program NearestInsertion.java, which reads the points from standard input, runs the nearest neighbor heuristic, and prints the resulting tour, its length, and its number of points to standard output. Then it draws the resulting tour to standard drawing. SmallestInsertion.java is analogous but runs the smallest increase heuristic.

Sample outputs using tsp1000.txt:

**Nearest Insertion**

(185.0411, 457.8824)
(198.3921, 464.6812)
(195.8296, 456.6559)
(216.8989, 455.126)
...
(264.57, 410.328)

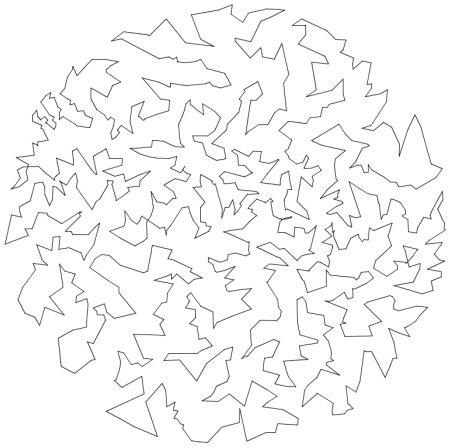
Tour length = 27868.7106
Number of points = 1000



**Smallest Insertion**

(185.0411, 457.8824)
(198.3921, 464.6812)
(195.8296, 456.6559)
(216.8989, 455.126)
...
(264.57, 410.328)

Tour length = 17265.6282
Number of points = 1000



# Beast Mode

Implement a better TSP heuristic. In case you need some inspiration, some ideas are provided below.

---

**Crossing edges**
If two edges in a tour cross, you can decrease the length of the tour by replacing the pair that crosses with a pair that does not cross.

**Farthest insertion**
It is just like the smallest increase insertion heuristic described in the assignment, except that the cities need not be inserted in the same order as the input. Start with a tour consisting of the two cities that are farthest apart. Repeat the following:

- For each city not in the tour, consider the shortest distance from that city to a city already in the tour. Among all cities not in the tour, select the city whose distance is largest.
- Insert that city into the tour in the position where it causes the smallest increases in the tour distance.

You will have to store all of the unused cities in an appropriate data structure, until they get inserted into the tour. If your code takes a long time, your algorithm probably performs approximately $n^3$ steps. If you're careful and clever, this can be improved to $n^2$ steps.

**Node Interchange Local Search**
Run the original greedy heuristic (or any other heuristic). Then repeat the following:

- Choose a pair of cities.
- Swap the two cities if this improves the tour. For example if the original greedy heuristic returns 1–5–6–2–3–4–1, you might consider swapping 5 and 3 to get the tour 1–3–6–2–5–4–1.

Writing a function to swap two nodes in a linked list provides great practice with coding linked lists. Be careful, it can be a little trickier that you might first expect (e.g., make sure your code handles the case when the 2 cities occur consecutively in the original tour).

**Edge exchange Local Search**
Run the original greedy heuristic (or any other heuristic). Then repeat the following:

- Choose a pair of edges in the tour, say 1–2 and 3–4.
- Replace them with 1–3 and 2–4 if this improves the tour.

This requires some care, as you will have to reverse the orientation of the links in the original tour between nodes 3 and 2 so that your data structure remains a circular linked list. After performing this heuristic, there will be no crossing edges in the tour, although it need not be optimal.