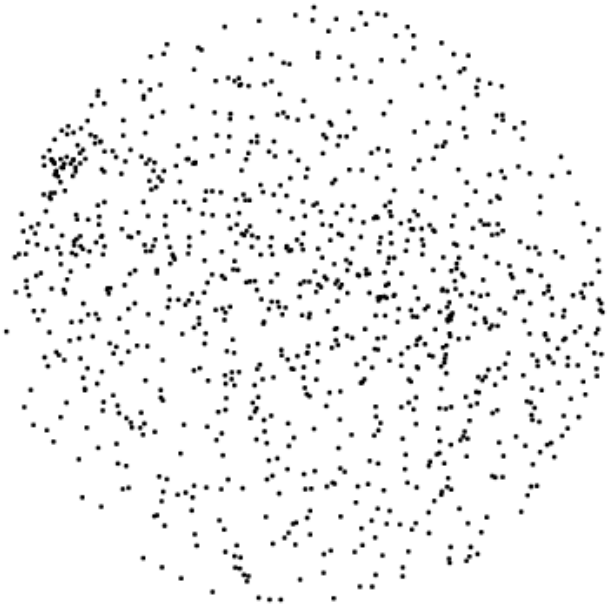
CS Data Structures
Mr. Bredemeier

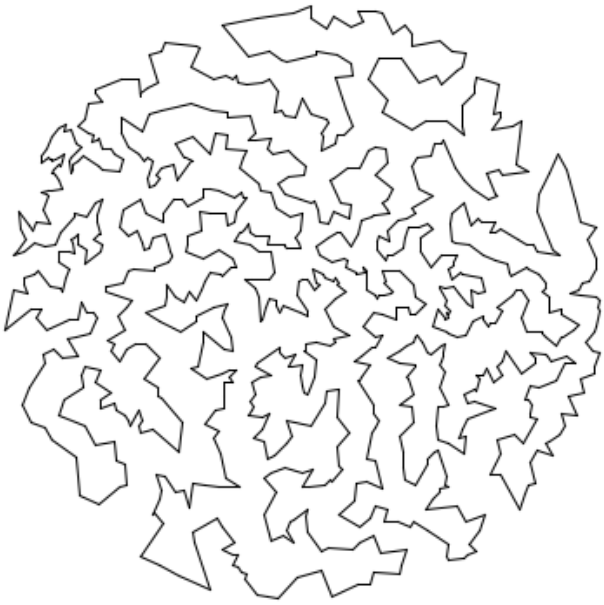
Traveling Salesperson Problem

This assignment was originally developed by Robert Sedgwick and Kevin Wayne at Princeton University (here is the link to the original assignment), and subsequently adapted by Baker Franke at the University of Chicago.

The Traveling Salesperson Problem (TSP)
Given *N* points in the plane, the goal of a traveling salesperson is to visit all of them (and arrive back home) while keeping the total distance traveled as short as possible. Implement two greedy heuristics to find good (but not optimal) solutions to the *traveling salesperson problem* (TSP).



1,000 points



Optimal Tour

Perspective.
The importance of the TSP does not arise from an overwhelming demand of salespeople to minimize their travel distance, but rather from a wealth of other applications such as vehicle routing, circuit board drilling, VLSI design, robot control, X-ray crystallography, machine scheduling, and computational biology. The traveling salesperson problem is a notoriously difficult *combinatorial optimization* problem. In principle, one can enumerate all possible tours and pick the shortest one; in practice, the number of tours is so staggeringly large – for N points there are N! tours - it makes this approach basically useless. An algorithm for finding an optimal tour for large N in any reasonable amount of time is currently an unsolved, and open problem in computer science.

Greedy heuristics.
However, many methods have been studied that seem to work well in practice, even though they are not guaranteed to produce the best possible tour. Such methods are called *heuristics*. A *greedy heuristic* makes the most optimal choice at each stage of the game with the goal of (hopefully) finding the best overall solution.

Your main task is to implement the *nearest neighbor* and *smallest increase* insertion heuristics for building a tour incrementally. 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 heuristic:* 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 increase heuristic:* 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 tour length. (If there is more than one point, insert it after the first such point you discover.)

Your Task – The Big Picture

Your task is to create a **`Tour`** class that represents the sequence of points visited in a TSP tour. The `Tour` class will be a modification of a Linked List where each node in the list holds a point in the `Tour` (an x,y coordinate). A linear traversal of the List will represent the path the salesperson would travel. NOTE: Since the tour is a cycle, you need to make sure that you consider that the first point comes immediately after the last point in the list when performing such calculations as computing the total distance of the tour. Your class will allow for the adding of one `Point` at a time to the tour, which will insert a node into the List in some fashion. The idea of the final program is as follows.

You will have a class with a `main` method that will:

1. Construct a new empty `Tour` (an empty `LinkedList` of `Points`).
2. Read in a data file of x,y coordinates (pairs of doubles) making a `Point` object of each one and adding each `Point` to the `Tour`.
3. Display the `Tour` in a `DrawingPanel` along with its total distance.

Implementation Details

The starting project provides you with the `TourInterface`, a start for an implementation of the `Tour` which includes an inner `ListNode` class that you should use for maintaining the List. The `DrawingPanel` code is also included.

The major work of the assignment is to implement the methods of the `TourInterface` which are briefly described here:

`Tour Interface` Methods:

`void print()` //prints the entire tour to standard output
`void draw(Graphics G)` //displays the tour using the given graphics context
`double distance()` //return the total distance of the tour
`int size()` //return the number of points in the tour (i.e. nodes in the list).
`void add(Point P)` //appends P to the end of the list.
`void insertNearest(Point P)` // inserts P into the list using nearest neighbor heuristic
`void insertSmallest(Point P)` // inserts P into the list using the smallestDistance heuristic

Getting Started

What follows is a suggested plan of attack for doing this assignment in short pieces. There are several ways to go about it. The following is merely a suggestion.

1. The starting project code has things mostly laid out for you. The `main` method runs but doesn't do anything because the `Tour` class is unimplemented at the moment. But your first step is to poke around the starting code to see what's provided and what's not.

2. You'll want to review the inner `ListNode` class that is a part of `Tour` class. Here we are introducing the concept of a private inner class, which is a separate class within a class. It is very similar in concept to a private helper method, which is particularly useful to the class where it is defined, but to nothing else outside the class. A private inner class is instantiated as a part of the outer class instantiation. In this application, a `ListNode` is only meaningful within the concept of a `Tour` of points, and we can create as many `ListNode` objects as needed to complete a given `Tour`. Because an inner class is a member of the outer class, the outer class has access to the private instance variables of the private inner class. In other words, there is no need for 'getter' and 'setter' methods for the outer class to access or modify the inner class instance variables. In this case, the `Tour` class has direct access to the `ListNode` private instance variables. Note that `ListNode` has different names for its instance variables, namely `data` and `next`.

3. First, declare the instance variables, and complete the constructor of the `Tour` class. Then, write the `size`, `add` and `print` methods of the `Tour` class first. The `size` method just returns the size of the list. The `add` method should work just like the `LinkedStringLog` `add` method which appends a node to the end of the list with the given `Point` as its data value. The `print` method should simply traverse the list and print out each `Point` on its own line. Then run the `main` method in the `TSPClientDriver` class to see that the five sample points are added to the list and then displayed properly. Make sure this is working correctly before you continue any further.

4. Write the `draw` method which accepts the Graphics context to use. The two methods from the Graphics class you will probably want to use are

```
fillOval(int leftEdge, int topEdge, int width, int height)
drawLine(int fromX, int fromY, int toX, int toY)
```

   Thus a call to `g.fillOval(x-2, y-2, 5, 5)` will display a circle with a diameter of five pixels centered at `x`, `y`.

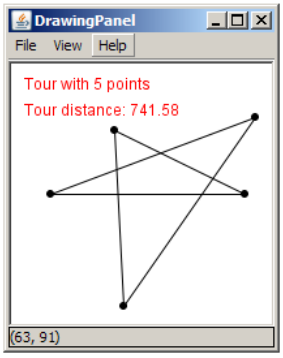
   Start by simply drawing a circle (an oval with same size width and height) on the graphics for each point in the list (we'll worry about connecting the dots later). Reminder: the points have `double` precision, but graphics coordinates have to be `ints`. A `DrawingPanel` has already been constructed in the main method of the `Tour` class, which then calls your `draw` method.

5. Now improve the `draw` method to draw lines between the points. Don't forget to connect the last point to the first. From now on, there is no need to call the `print` method anymore, as the point data will now be displayed in the `DrawingPanel`.

6. Finally, begin to write the primary `Tour` methods that we care about. It's suggested you write these methods in the following order:

`distance()`

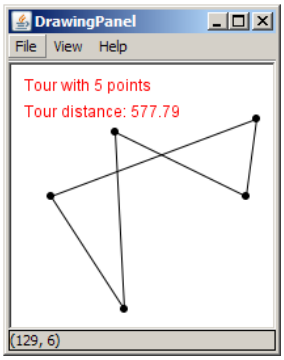
Caclulate the total distance of the `Tour`. NOTE: the `Point` class has a method that calculates the distance between itself and another point. Don't forget to count the distance between the last point and the first.



See if you get the same distance for the 5-point tour as shown in the picture at the right. (741.58).

`insertNearest(Point p)`

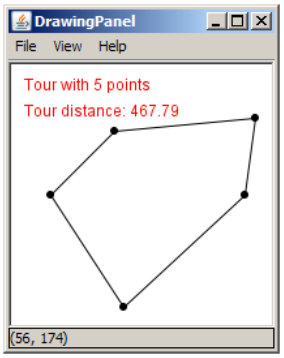
This method will change how you add points to the list. Follow the nearest neighbor heuristic – adding this `Point` into the list immediately after the point it is closest to. After writing `insertNearest`, change the code your `main` method to add points to the tour using `insertNearest()` rather than `add()`. Note: since adding one point using this method will take O(N) time, then adding all points in the list will take $O(N^2)$ time.



See if you get the same distance for the 5-point tour as shown in the picture at the right. (577.79).

`insertSmallest(Point p)`

Write the `insertSmallest` method which should follow the smallest increase heuristic – add a point into the list at the location that causes the total distance of the Tour to increase the least.

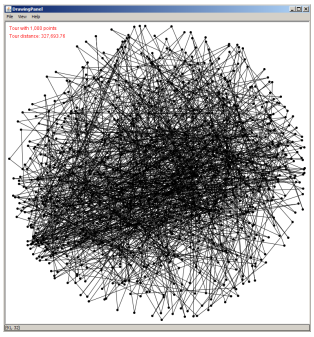


Careful: When you run `insertSmallest` on a very large data set, say something with more than 10,000 points, it can take a while. If it takes a REALLY long time, you've probably got a $O(N^3)$ algorithm. If you're careful/clever you can get this down to $O(N^2)$. Think about it.

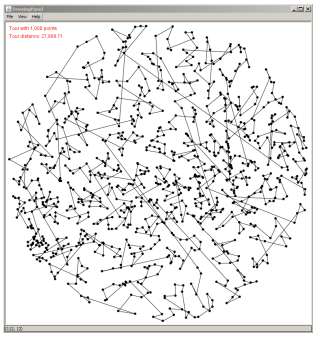
See if you get the same distance for the 5-point tour as shown in the picture at the right. (467.79).

Testing on larger data sets

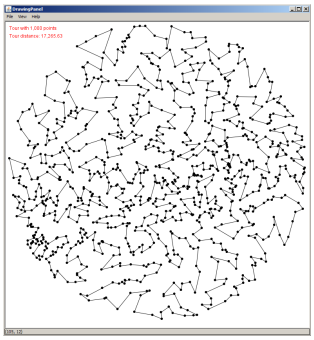
The starting project contains several different data sets (look for the .txt files). By just changing the filename of the file being read in in the main method you can test on larger data sets to see how fast or slow the various heuristics work. Here's what the tsp1000.txt file looks like using the various methods:



add()
dist: 327,693.76



insertNearest()
dist: 27,868.71



insertSmallest()
dist: 17,265.63

Requirements

- All of the `Tour` methods implemented correctly, and each method has *abundant comments* that fully describe your code.
- Complete tour is displayed in the `DrawingPanel`.
- Calls to `insertNearest` or `insertSmallest` should be right next to each other with one commented out. (I will run it to see one, and then un/comment the other and run it again).
- The README file (click on the white lined box in the BlueJ window to open it) contains both you and your partner's name, date, and a complete and thorough description of how both of your algorithms work. State the overall complexity (using big-oh analysis) of each algorithm in your descriptions.
- Your `insertSmallest` algorithm should process the tsp1000.txt data file in a reasonable amount of time (5-10 seconds). In order to receive full credit, it needs to be an $O(N^2)$ algorithm, which runs in less than 1 second (approximately).
- If you want me to look at something in particular, like a challenge (see below) please mention it in the README.

Turning it in

- Name your project file directory Traveling Salesperson.
- Zip up the entire project file directory by control-clicking on the directory name and then select Compress "Traveling Salesperson" from the menu.
- Submit the zipped directory using the Canvas assignment page. I only need one submission for both you and your partner (pick one). In other words, only one of you needs to turn this in for both of you to get credit for it (of course both of your names will be listed in the README file).

Extra Challenges

Here are a series of challenges in no particular order. The fall under two general categories: improving the user interface/graphics, or improving the algorithm. You may do as many or as few as you like.

Animate the path finding

Use a `java.util.Timer` to add points to a Tour at some interval, re-drawing the current state of the Tour after each point is added. Since you have to add the points one at a time when a Timer event is fired, you might want to read the data file into a simple `LinkedList<Point>`. Then every time the timer fires, pull a point from the list and add it to the tour.

Below are some ideas for finding a better TSP tour. None of the methods guarantees to find an optimal tour, but they often lead to a good tour in practice.

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:

- Among all cities not in the tour, choose the one that is *farthest* from any city already in the tour.
- Insert it 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, you 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 in 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.