ICS 340 Programming Project, Deliverable C (50 pts)

Specification:

Start with your Java program "prog340" which implements Deliverables A and B.

This assignment is based on the definition of the *Traveling Salesperson Problem* (the TSP): Given a set of cities, you want to find the shortest route that visits every city and ends up back at the original starting city. For the purposes of this problem, every city will be directly reachable from every other city (think flying from city to city).

Your goal is to use a non-genetic local search algorithm or algorithms from Chapter 4 of Poole & Mackworth to find the shortest paths that you can. Note that for any reasonably sized problem, you will not be able to try every possibility, and so you won't ever <u>know</u> when you have the shortest possible path.

You should have two types of output: Summary Mode and Verbose Mode.

Verbose Mode should print to a file every path you try until your program terminates. Summary mode, which prints to the console, should print a path only when that path is better than any path that you have found previously. In all cases, print the single-letter <u>mnemonic</u> of the city (Node), not its full name.

Suggestions

You can start from any city you want, it really doesn't matter which. For example, you can start from the first city in the file, since it is the easiest. Create <u>some</u> tour. You can do this randomly, or you can use a greedy algorithm of some sort, or you can just visit the cities in the order they appear in the file, whatever you prefer. Your choice probably won't affect the goodness of your final tour, but it might affect how long it takes you to get to a good solution.

Selecting and using the techniques of local search, try to find a better tour. Be sure to document what you're trying to do.

The rest of this section assumes you want to create your own algorithm. Here are some thoughts on what is probably the main question: how to step from one possible tour to the next.

- One option would be two swap the order of two cities and see if this shortens the tour

    o

    o   Mpls. → *Seattle* → Detroit → Boston → *Chicago* → Miami → Denver → Mpls.

- Or you could pick one city in the tour, and try removing it and inserting it between every pair of cities to see which gives the shortest tour.

- There are other forms of Iterative Best Improvement, too.

- When deciding what cities to swap or what city to shuffle, you could use either a 1-stage or 2-stage choice algorithm.

With any sort of iterative best improvement, there are various techniques you could use if you wish. One of the big differences among peoples' algorithms will be their choices to use or not use these techniques, and the algorithm they use to decide when to use them (for example, how often to do a random step or restart, temperatures for simulated annealing, etc.)

- You could choose to use Simulated Annealing to decide whether to keep a given tour even if it's longer than the existing tour.

- You could inject randomness by sometimes making random permutations (random walk).

- You could decide after a certain number of iterations that you should just start over from scratch with a random order of cities (random restart).

- You could choose to keep a tabu list or not, and if you do, you could choose its size.

- You could keep multiple tours, and permute them in parallel (beam search).

Since you don't actually know when you have an optimal tour for any reasonably sized problem, you need to decide on a stopping criterion. Some possible suggestions:

- Stop after some number of steps (where the number of steps might be some function of the number n of cities).

- Stop after you have gone some number k of steps without improving on your best answer.

- Stop after the program has run for some amount of real-time (measured by something like System.time.millis()).

Report

You should write a brief report that explains what you did. It should indicate the algorithm you chose, what techniques you selected and any parameters you used for it. You should discuss how well your algorithm does based on the results you seemed to get.

Don't Break Existing Functionality

Running Deliverable A and Deliverable B on the test files should still work.

Administrative Details

The "prog340" handout describes the format of the input file for this and all program deliverables.

As always, the program must be written in Java and must run on the University Windows computer systems. To ensure this I strongly recommend that you:

1. Use only Oracle Java SE, and

2. Test it on the University systems before submission if you have any doubts about its ability to run on the University Windows.

3. As before, try to minimize disruption to the existing codebase.

Output:

Here is sample output for one graph *C5.txt*. First, here is the file.

```
~                 Val   C    H    K    M    P
Cleveland          C   ~ 1114  700  630  360
Houston            H 1114    ~  644 1056 1341
KansasCity         K  700  644    ~  413 1039
Minneapolis        M  630 1056  413    ~  985
Philadelphia       P  360 1341 1039  985    ~
```

Suppose that you do the following:

- Start out by traveling the cities in this order: Cleveland, Kansas City, Houston, Minneapolis, Philadelphia, back to Cleveland.

- Switch the positions of Houston and Kansas City

- Switch the positions of Kansas City and Philadelphia

- Switch the positions of Kansas City and Minneapolis

- Terminate

The *verbose* output would look like this.

```
CKHMPC 3745 (random restart)
CHKMPC 3516 (swap)
CPHMKC 3870 (swap)
CPHKMC 3388 (swap)

Shortest path found was CPHKMC with distance 3388 after 4 steps.
```

The *summary* output would look like this, only printing the improved tours.

```
CKHMPC 3745
CHKMPC 3516
CPHKMC 3388

Shortest path found was CPHKMC with distance 3388 after 4 steps.
```

The verbose output for graph <name>.txt should be written to file <name>V.txt. The summary output should be written to the console.

Submit:

Submit a full Eclipse package to the Deliverable C submission folder.

Test Files:

There are four test files that will be used for grading: CD5.txt, CD7.txt, CD10.txt, and CD15.txt. I have also included a larger file, CD49.txt, for your own testing purposes.

Grading:

40 points for passing the tests "correctly" – you get good answers using a reasonable algorithm in a reasonable amount of time. Yes, there is some subjectivity to this. There are four test files at 10 points each; 2 points each for design and documentation; 6 points for the report.

Due Dates:

The program is due on April 27th by 11:59 pm in the D2L "Deliverable C" dropbox. The time of submission is the time that D2L lists the file as submitted.