# ICS 340 Programming Project, Deliverable D

***Motivation***

*For finding a good Ham cycle path of a really large graph, it may be better to use random methods to get a good solution in a reasonable time, rather than to try to get the optimal solution by brute force, or even using Dynamic Programming (which is still time O(n2n). For this deliverable, you will use one or more of a variety of randomized methods for finding a low-cost TSP path.*

***Specification:***

A Hamiltonian Cycle of a graph is a path that starts at one node of a graph, and visits every other node exactly once. It’s like a TSP problem except that you don’t go back to the starting city.

Start with your Java program “prog340”, ideally it should be working for Deliverables A and B.

Your goal is to find the best possible Hamiltonian Cycle of a given graph, in which the starting and ending (goal) cities are specified, using an algorithm from Sections 4.7 – 4.9 of the Poole & Mackworth text, or a combination of such algorithms. Write a report in which you discuss your criteria for a “good” algorithm. Print your initial tour, and every path you find that is the better than all previous paths.

Details and suggestions are given below under “Setup” and “Possible Algorithms”.

I also am sharing with you a few papers that might help give you ideas of the vast set of ways that you could implement tabu lists (if you want to), and/or do simulated annealing. Remember, you don’t have to do any specific algorithm, these are just ideas. You could also do a dynamic program as described in Section 4.9.

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

As will always be the case in this class, 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 13 SE and earlier constructs, and
2. Test it on the University systems before submission if you have any doubts about its ability to run on the University Windows.

Submit the Java source code to the open Deliverable C submission folder. You may submit either the source code or a full Eclipse package.

***Setup***

First, I will only give you graphs for which a tour is possible.

For each graph, the algorithms work most easily if you assume that there is an edge between each pair of cities. So when you read the graph, modify it so that if there is no edge from some node “A” to some other Node “B”, you create an “edge” in the graph with some value that is so large that your best tour will certainly not include that “edge”, but small enough so that if you sum all the edges in some random tour, you will not overflow. I suggest fake “edge” lengths in the neighborhood of 100,000 to 1,000,000. (The incomplete graph I will test with will have fewer than 300 cities, and if you have no distances greater than a million, you don’t have to worry about overflowing a Java int.)

***Algorithm***

Start from any city you want, it really doesn’t matter which. I start from the first city in the file because it’s easiest. Create some Ham cycle. Note that if you assign a very long “edge” between each pair of Nodes with no edge between them, as I suggest above, you are guaranteed to have some Ham cycle.

Using the techniques of local search, try to find a better tour. Use one or more of the techniques from Section 4.7 of the Poole & Mackworth book. There are a lot of creative ways to do this, I’ll even point you to some papers that include some of these ideas. Be sure to document what you’re trying to do. If you have questions about whether an idea you have qualifies as a “local search” algorithm, just ask me.

When you’re developing your algorithm, the main question is how to step from one possible path to the next. The key is that you don’t just change a single value to switch between paths, but rather you swap the order of some of the cities, so that in every possible path, each city still appears.

* One option would be to swap the order of two cities and see if this shortens the path. Assume S = Minneapolis and G = Denver.
  + Mpls. 🡪 *Chicago* 🡪 Detroit 🡪 Boston 🡪 *Seattle* 🡪 Miami 🡪 Denver becomes
  + Mpls. 🡪 *Seattle* 🡪 Detroit 🡪 Boston 🡪 *Chicago* 🡪 Miami 🡪 Denver
* You could move a block of cities somewhere else in the path. For instance, you could move Chicago 🡪 Detroit to just before Denver
  + Mpls. 🡪 Chicago 🡪 Detroit 🡪 Boston 🡪 Seattle 🡪 Miami 🡪 Denver becomes
* Mpls. 🡪 Boston 🡪 Seattle 🡪 Miami 🡪 Chicago 🡪 Detroit 🡪 Denver
* You could randomly pick a new path, or randomly swap two cities
* You could use a genetic algorithm of some sort (there’s a paper on using genetic algorithms for the TSP that’s listed).

Generally these are forms of Iterative Best Improvement. 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 (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 path even if it’s longer than the existing path.
* 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. There are also many different ways to define a tabu list. For instance:
  + You could prohibit repeating one of the last *n* paths.
  + You could prohibit any path with an edge of longer than some certain value.
  + You could prohibit any path that has a certain pair of cities next to each other (say, prohibit paths that go Seattle 🡪 Miami).
* You could keep multiple paths, and permute them in parallel (beam search).

Since you don’t know when you have an optimal path 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()).

**Output**

Print a path (to the console and to a file) every time you find a path that is better than the best path you have stored. Print the path by listing the abbreviations of the cities in the order in which they appear (see example below). You should also have print statements, *commented out in the version you hand in*, that would allow me to print every new path that you try, so that I can watch it evolve if I want to.

**Report**

You should write a report that explains what you did. It should indicate the algorithm you chose, and any parameters you used for it. The explanation should be detailed enough that I can read it, look at your code, and follow what you did. You should discuss the goodness of the results you seemed to get.

***Test Files***

The tet files for deliverables B and C are good for these.

**Sample Output**

Courier font is output, *italicized times roman font is commentary on the output.*

Consider file F1:

~ val Ap Ba Ca Du El

Apple S ~ 8 6 5 6

Banana ~ 7 ~ 2 7 6

Cantaloupe ~ 5 3 ~ 8 8

Durian ~ 6 8 7 ~ 8

Elderberry G 7 6 9 9 ~

* Suppose I start with a path from first city to last and back. I travel the cities in order:
  + Ap Ba Ca Du El, for a path distance of 26.
* I try swapping various pairs of cities, according to my algorithm (whatever it is). I try tours like
  + Ap Ba Du Ca El, for a path distance of 30.
  + Ap Ca Ba Du El, for a path distance of 24.
  + Ap Du Ca Ba El, for a path distance of 22. This is the best single swap that I find following my algorithm, so I make this my next proposed path.
* After trying many more iterations, I quit.

Here is what I would print:

Dist = 30: Ap Ba Ca Du El

Dist = 24: Ap Ca Ba Du El

Dist = 22: Ap Du Ca Ba El

Total of 3 paths tried.

**Submit:**

Submit your code as an Eclipse package, or submit all the “.java” source files in a zipped archive. Do not include test files.

**Grading:**

This deliverable is worth 80 points: 45 points for correctness, 5 points for design, 5 points for documentation & style, 15 points for regression testing Deliverables A, B, and C (meaning I’ll run Deliverable A on one file and Deliverable B on another file and Deliverable C on yet a third file, and you get 5 points per correct test), and 10 points for the report.

I will run your program multiple times on multiple files to see how close you come to a correct answer, and how quickly you converge to it.

Note: If you can create a path, evolve it (even randomly) and print out whenever you find a path better than any you have previously encountered, that will be worth at least 25 points for correctness.

**Due Dates:**

The program is due on Saturday, April 17th at 6 PM for full credit in the D2L “Deliverable D” dropbox. For 80% of credit earned, you may (re)submit it by noon Saturday, May 1st (late submission possible until noon Sunday, May 2nd with late penalties). The time of submission is the time that D2L lists the file as submitted.

**Thoughts:**

*The fun thing about this program, and what I think makes it a good program for this class, is that it’s very modular. It has a lot of parts that are relatively independent of each other, and which build on each other. It’s an “easy C” but a “hard A”.*

*Clearly, the first thing to do is to read the graph, modify it to be a complete graph by adding really long edges between each unconnected pair of Nodes, and print out an initial path. This should not be hard. It’s well under 100 lines of code, and not really tough code to write, either. Then write code to generate additional random tours, by randomly starting from scratch (random restart) or by randomly permuting the order of cities in the path (random walk). Print out any path you find that is better than the best one so far. (This is basically section 4.7.2 of the P&M text.)*

*The above is a really simple but poor way to find a good path. But it’ll give you a passing grade on the program. Save it away. I think everybody should be able to do this with a few hours of effort. Personally, I did all this in under an hour. It’s a lot easier than Deliverable B, one you understand what you’re trying to do.*

*Then, if you have time and inclination, do the “real” problem: use Local Search techniques to find better tours systematically. There are a number of techniques, described in section 4.7.3. “Play around” with them until you find some that work well for you. I will basically decide what are the best programs based on how good a path they come up with, and how long they take.*