(HW05) The Traveling Salesperson Problem: Graph Implementation Due: Monday, December 3, Noon

Workplace Scenario

Suppose you are working tech. support for Starbucks, and have to tackle the problem of delivery truck routing. Specifically: you need to write a program that will take the location of a supply center and a list of Starbuck locations that are resupplied from that center, and produces a route that will get a truck from the supply center to each of the stores and back while driving the minimum number of miles possible (and hence, presumably, minimizing both fuel and labor costs). It turns out that writing a program to find an optimal route in polynomial time is a *really* difficult. However, if you are not worried about runtime then the algorithm becomes straight-forward. **And really, really slow.** You need to design the best tool you can.

This is an example of the Traveling Salesperson Problem, a problem that shows up (in some variation) in many important applications. If you are going to address it, you probably want to model the problem with a *graph*: each node represents a store location, while edges represent distance. Graphs are one of the most useful general models in Computer Science, and having the correct, efficient, implementation of the Graph ADT (along with basic functionality) is of some importance. In this assignment you will implement the two most commonly used graph data types, and on top of that write a TSP implementation.

Importance: The point to this assignment is in the development of your understanding of *graph representation* and your implementation of a *branch-and-bound* algorithm. The Graph structure is a fundamental data structure that support numerous algorithms, but picking the right implementation is vital to achieving optimal efficiency; understanding the implementations will help you make the correct decisions. The branch-and-bound solution to the TSP is represents a type of algorithmic solution that is applicable to almost any problem you will come across, and sometimes the only fallback for problems known as hard as the TSP (known as NP-Hard problems).

Recommended Reading

Sections 17.1-17.5 of the Sedgewick book for more details on the implementation of graph data.

Steps:

1) Create a new Visual Studio project, and add all code downloaded for this assignment (including Graph.h, Tester.cpp/.h, StarbucksMap.cpp/.h, StarbucksMap.cpp, and GraphAlgs.h).[1] You may want to create a plain vanilla Visual Studio project, instead of using Tinderbox; that way, you can use the Controller.cpp without modification.
2) For the file *MatrixGraph.h* create a file *MatrixGraph.cpp* implements a subclass of the Graph class (see Graph.h) using a AdjacencyMatrix implementation. Make sure that all methods are asymptotically optimal for that representation.

[1] We are using a different version of the Starbucks.csv file, so don't use the one from HW04!

3) For the file *ListGraph.h* create a file *ListGraph.cpp* implements a subclass of the Graph class (see Graph.h) using a AdjacencyList implementation. Make sure that all methods are asymptotically optimal for that representation.
4) Leaving all code relating to the Traveling Salesperson Problem commented out, use the *Controller.cpp* to test your implementations.
5) In files *GraphAlgs.cpp* implement the TSP function defined in *GraphAlgs.h*. Not that the Traveling Salesman Person is a computationally complex problem: it is *very unlikely* that your algorithm will be able to handle very many nodes.
6) Uncomment the test code in *Controller.cpp* to test the correctness and speed of your algorithm.

Quality Measures: (Up to 100 points total)

A) 30 pts – Adjacency List fully working
B) 30 pts – Adjacency Matrix fully working
C) 30 pts – TSP algorithm that returns correct results
D) 10 pts – TSP algorithm that is significantly faster than the naïve brute-force method
E) 10 pts – Use Cinder to create a graph that visualized the performance of your TSP algorithm as a function of N, the number of nodes in the graph.