CSC 495 – Final Programming Challenge — Due Tuesday, April 24

Challenge Problem: Chop Up The Graph

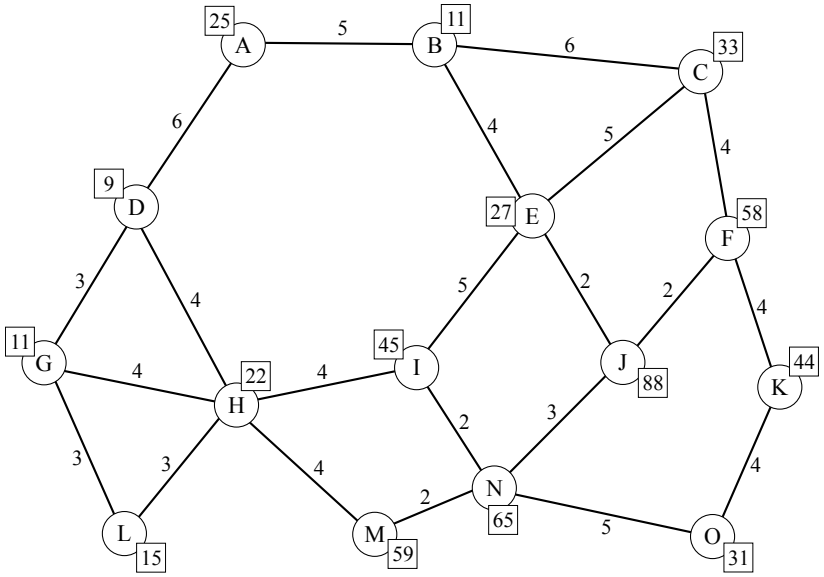
This is your final challenge problem for CSC 495. Unlike earlier problems, there is not a simple algorithm that will find the correct answer. You will have to combine several strategies that have come up repeatedly in this class, including efficient data structures, efficient searching, and possibly others. Note that the underlying problem is NP-hard, so don't spend a lot of time trying to come up with an optimal solution — instead, think about heuristics and refining solutions.

Background

Consider the following problem: You have a map of a city, with road segments labeled by the amount of time it takes a fire truck to traverse them. Your goal is to place fire stations so that every point in the city can be reached in less than 8 minutes from some fire station. You can model this as an edge-weighted graph, with edges representing road segments and weights representing times. Your goal is to mark a subset of the vertices (for fire stations) so that every vertex has a weighted distance of at most 8 to a marked vertex. Our goal here is to pick as few vertices as possible while satisfying the time constraint (you can't mark all vertices!).

You can also consider a variation in which all calls don't have to be answered within the given bound, but a certain fraction does — for example, say 90% of the calls must be answered within 5 minutes. To answer this, we need to know the distribution of calls — some neighborhoods might have a higher call rate than others, due to older houses, more densely packed neighborhoods, etc. To model this, add weights to the vertices of the model in the previous paragraph. The number on a vertex is the number of calls from that location in some unit of time (say in a year).

Consider the following example — in this picture, the letters inside the vertices are the names of the locations (in general these will be strings), the numbers in boxes represent the rate of calls to that location, and the edge weights represent the time to traverse that segment.



On the given graph, if we want to be able to respond to all calls within 8 minutes, we can do this with stations at vertex D and vertex F (at first it looks like you can't get to vertex B in less than 10 minutes, but the path is to go F→J→E→B (distance 8). However, with just two stations, many of the distances are almost at the limit of 8 minutes. If instead we say that we would like to respond to 90% of the calls within 5 minutes, then a solution is to place three stations at vertices D, F, and N (there is no two-station solution). Location B is still out of luck (at distance 8) but since B has a relatively low call rate of 11, and M and N have relatively high call rates (59 and 65, respectively) this is overall a good solution.

The Problem to Solve

You are to write a program that reads in a description of a problem as described above, and finds the best solution possible within a given time bound. Note that when the problem is small, as in the pictured problem, you can do a brute force search and find the optimum solution. However, your program should be able to handle much larger inputs: 500 to 5,000 vertices. In that case, brute force doesn't stand a chance of finishing during your lifetime (in fact, brute force wouldn't finish before the sun goes supernova, destroying the earth and all things on it — fire stations aren't going to help then). When determining which solutions are "best," the following rules apply: a solution meeting the constraints with fewer stations will always be better than a solution with more stations; among multiple solutions with the same number of stations, a solution is better if it has a smaller worst-case distance for response time (disregarding call frequency).

While there are many possible approaches to this problem, your program must be written so that it finishes in under 30 minutes. So, for example, a reasonable approach is to search through potential solutions in some sensible order, looking for the best one that you can find. You should keep track of the time used by your program (similar to the simulations in Problem Set 10) and output the best solution you have found when you approach the 30 minute limit. Note that it is very important that your program finish and output a solution within 30 minutes: if it does not produce a solution, then it can not be graded as correct. You probably want to test this with smaller time limits (like 1 minute). If you have any difficulties with this part of the code see me to work them out — you should be thinking about solving the problem, and shouldn't have a program fail because the timing code isn't working.

Managing resources and keeping track of information in your program gets challenging as the inputs get larger. To get an "A", you must do this well enough to handle large inputs (up to 5,000 vertices) — if you can only handle smaller numbers of vertices, your program will still be graded but the maximum grade you can receive will vary as follows:

Small Problems Only: If your program produces a valid solution only for small inputs (30 or fewer vertices), then the maximum grade it can receive is an 80.

Medium Problems Only: If your program produces a valid solution only for medium-sized inputs (500 or fewer vertices), then the maximum grade it can receive is a 90.

Challenge Competition

Programs will be tested with data sets of two sizes, medium (500 vertices) and large (5000 vertices), and allowed to run for 30 minutes. The solutions will be checked for correctness (they must give a feasible solution), and then ranked according to quality of solution. 20 extra ninja points will be distributed for each input size, distributed according to the quality of the solutions produced.

Other Notes and Thoughts

This problem contains many of the components of some common real-world resource allocation problems, and small changes can result in interesting variations. For example, what if we aren't picking designated locations, but instead are just partitioning the graph into regions that have a bounded diameter (length of the longest path within the region)? This is what you would get if you were defining police patrol areas rather than fire station locations — a patrol car is mobile, so could be at any location in its region, and should be able to respond to a call from anywhere else in the region within a reasonable time. For another variant, consider simplifying the original problem so that all edge distances are one (which actually doesn't make it that much easier from a computational standpoint) — this reflects the problem of deciding where to put wired base stations in an ad hoc network (bounding the number of hops a packet has to take before it reaches a base station). Other variations take into account the dynamic nature of the underlying data: as a municipality grows, where should we build new stations to cover the increased area, with the understanding that you can't just pick up all the existing fire stations and move them to new optimal locations? Many of these problems and variants touch on research questions that we don't have good answers to currently — it's a rich and deep area to explore!

Input and Output

The first line of the input will contain two integers: the distance limit (requests/calls must be within this distance of a marked vertex to be "good") and the percentage of requests that must be good in a feasible solution. This is followed by a line containing the number of vertices, followed by the vertices, each on a line containing the vertex name (a unique string with no spaces) and a weight. This is followed by a line containing the number of edges, followed by a description of each edge (the two endpoints and a weight). The output should consist of a line giving the number of marked vertices in your solution and the worst-case distance to a vertex, followed by the locations of the stations (follow the format shown in the samples).

The samples on the following page show a representation of the graph drawn on the first page. The first sample output shows a request in which 100% of the requests must be within distance 8 of a marked vertex, and the second output (input not shown, but first line changed to "`5 90`") shows a request in which 90% of the calls must be within distance 5 of a marked vertex.

Sample Input 1

```
8 100
15
A 25
B 11
C 33
D 9
E 27
F 58
G 11
H 22
I 45
J 88
K 44
L 15
M 59
N 65
O 31
22
A B 5
A D 6
B C 6
B E 4
C E 5
C F 4
D G 3
D H 4
E I 5
E J 2
F J 2
F K 4
G H 4
G L 3
H I 4
H L 3
H M 4
I N 2
K O 4
M N 2
N J 3
N O 5
```

Sample Output 1 (100% distances ≤ 8)

```
2 stations, worst distance:  8
D
F
```

Sample Output 2 (90% distances ≤ 5)

```
3 stations, worst distance:  8
A
F
H
```