CSE 542 – Advanced Data Structures and Algorithms *Jon Turner*

Lab 2

Due 2/19/2013

General notes for labs. Review and follow the general notes from lab 1.

In this lab you will be evaluating the performance of the shortest augmenting path algorithm for the maximum flow problem, and you will be implementing an optimized version of the algorithm that can often be significantly faster. There are several parts to the lab.

A. In this part, you will be making source code modifications that will be used in later steps to measure the performance of the shortest augmenting path algorithm. In your svn repository, you will find a class called *augPath*, which serves as a base class for all specific variants of the augmenting path algorithm. You will also find a class called *shortPath* that implements the shortest augmenting path algorithm, using *augPath*. Take some time to understand the code in both (the *.h* files are in the include directory, the *.cpp* files are in *graphAlgorithms/maxFlo*).

   Next, add the following five performance counters to *augPath*: *fpCount*, *fpSteps*, *augCount*, *augSteps* and *runtime* (do not change the names). Make them all public. In *augPath.cpp*, you should initialize the first four to zero. Also, in *augPath.cpp*, add code to increment *augCount* every time the *augment* method is executed and *augSteps* for every edge on which the flow is increased. In *shortPath.cpp*, increment *fpCount* whenever the *findpath* method is executed and *fpSteps*, for each iteration of the inner loop. Also in *shortPath.cpp*, include code to measure the total running time of the algorithm and save this in *runtime*.

   You'll find additional instructions in the lab report template.

B. In this part, you will be writing a modified version of the augmenting path algorithm. The standard version adds one augmenting path to the graph after each execution of the *findpath* method. We will be modifying the *augment* method to find multiple augmenting paths during each call. This will reduce the number of augmenting path searches that are required and improve the overall running time. When *augment* is called, the *pEdge* array defines a subtree of the *flograph* in which all edges have positive residual capacity. This subtree includes an augmenting path from the source to the sink, but it also contains information about lots of other paths, and we can often use this to find multiple augmenting paths using the same *pEdge* array.

   Consider the case where the *augment* method, causes just one edge (u,v) to become saturated. When augment is called $pEdge[v]=(u,v)$, so after we add flow to (u,v), the *pEdge* array no longer represents a subtree in which all edges have positive residual capacity. But this is only true because of this one edge. Suppose we could assign a new value to $pEdge[v]$ that would again give us a subtree in which all edges have positive residual capacity. We could then use this to find another augmenting path. And by repeating this process multiple times, we can potentially find several augmenting paths, by making just small changes to the *pEdge* array for each path we find.

To make this work, we must add another array to *augPath*; define $d[u]$ to be the number of edges in the path from u to the source vertex in the subtree defined by the *pEdge* array. For vertices u that are not in the subtree, we let $d[u]=n$. The values of d can be computed by *findpath*; whenever we make $pEdge[v]=(u,v)$, we assign $d[v]=d[u]+1$.

Now, the *augment* method is modified to use a helper method called *reaugment*. *Reaugment* attempts to add flow to a single augmenting path defined by the *pEdge* array. If it succeeds in doing so, it attempts to fixup the *pEdge* array to enable another path to be found on a subsequent call. Specifically, for each edge (u,v) that becomes saturated, *reaugment* attempts to replace $pEdge[v]$ with another edge (w,v), where (w,v) has positive residual capacity and $d[w]=d[u]$. It does this by scanning all edges incident to v until it finds one that satisfies the condition. When *reaugment* is done, it returns the amount of flow added to the path it found or 0 if no flow was added. *Augment* calls *reaugment* repeatedly, until *reaugment* returns 0. *Augment* then returns the total flow added during all calls to *reagument*.

Create classes *faugPath* and *fshortPath* that include the modifications described above (plus the statistics counters you added to *augPath* and *shortPath*). You'll find further instructions in the lab report template.

C. In this part, you will be evaluating the performance of *shortPath* and *fshortPath* on random graphs, as the number of edges increases, while the number of vertices is held constant. You will find detailed instructions in the lab report template.

D. In this part, you will be evaluating the performance of *shortPath* and *fshortPath* on random graphs, as the number of vertices and edges increases together. You will find detailed instructions in the lab report.

E. In this part, you will be evaluating the performance of *shortPath* and *fshortPath* on "hard-case" graphs, for which the running times grow very quickly. You will find detailed instructions in the lab report.