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

Lab 3

Due 3/5/2013

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

In this lab you will be writing a program to solve the *edge coloring problem* in bipartite graphs, using a matching algorithm as a key component. The input to the edge coloring problem is a simple bipartite graph and the output is an assignment of *colors* to the edges, such that each vertex is incident to at most one edge of each color. The objective is to find such an assignment that uses the smallest possible number of colors.

Note that each subset of the edges that share the same color forms a matching. So we can view the problem as one of partitioning the edge set into a minimal number of matchings. Also, note that the number of colors must be at least equal to the maximum vertex degree. For simple bipartite graphs, the number of colors required is always equal to the maximum vertex degree.

Here is a general method for finding an edge coloring of a bipartite graph $G=(V,E)$. We start by making a copy H of G, then repeat the following step as long as there are edges in H.

> *Matching step.* Find a matching M in H that includes an edge incident to every vertex of maximum degree. Let c be a previously unused color, and color all edges in M with color c. Remove all edges in M from H.

To implement the matching step, we need a method to find a matching that includes an edge at every vertex of maximum degree. This can be done using a variant of the augmenting path method for the maximum matching algorithm. In this case, we build a single tree, rooted at a vertex of maximum degree. The tree is constructed in the same way as the trees in the augmenting path method, but we terminate the path search early, if we come to an even vertex that does not have maximum degree. If we "flip" the edges on the path from such a vertex to the root of the tree (that is, matched edges on the path become unmatched and unmatched edges become matched), we get a new matching with the same number of edges, but with one more vertex of max degree incident to a matching edge (the root).

Here's a more detailed description of the method for finding a path to extend the matching. Start by selecting a vertex r of maximum degree. Let $state(r)=even$ and for all other vertices u, let $state(u)=unreached$. For all vertices u, let $p(u)=null$. Now, repeat the following step until we find a path that can be used to extend the matching.

> Let $e=\{v,w\}$ be a previously unexamined edge with v even.
>
> > If w is not *unreached*, ignore e and proceed to the next edge.
> >
> > If w is unreached and unmatched, then the edge e together with the tree path from v to r is an augmenting path, and we terminate the path search.
> >
> > If w is unreached and matched, let $\{w,x\}$ be the matching edge incident to w. Expand the tree by making $p(w)=v$, $p(x)=w$, $state(w)=odd$ and $state(x)=even$. If x is not a maximum

degree vertex, then the tree path from x to r can be flipped to extend the matching, and we terminate the path search.

Note that there are two ways the algorithm can terminate with a path. In both cases, we extend the matching by flipping the edges on this path. Each time we do this, we increase the number of maximum degree vertices incident to matching edges by at least 1.

We claim that for simple bipartite graphs, the algorithm always returns a path. To understand why, note that the basic step described above maintains the following invariant, as long as it does not terminate.

> The number of odd vertices is one less than the number of even vertices and all even vertices have maximum degree.

This is true because every non-terminating step adds one odd and one even vertex, and the algorithm terminates whenever we reach an even vertex that does not have maximum degree. Note that since the even and odd vertices are joined by edges, they are in different subsets of the partition defined by the bipartite graph. Let X be the subset containing the even vertices and Y be the subset containing the odd vertices. Now, suppose there are k even vertices and the maximum degree is Δ. Since no vertex has degree larger than Δ, the number of vertices in Y that are adjacent to an even vertex must be at least k. But only k–1 of these are currently in the tree, so there must be at least one unreached vertex in Y that has an edge connecting it to an even vertex. Consequently, the algorithm will not terminate without returning a path.

The lab is organized into a series of parts which are summarized below. You will find more details in the lab report template.

A. In this part, you will develop a class called *maxdMatch* that finds a matching that includes every vertex of maximum degree, using the algorithm described above. You will find a skeleton of this class in the *graphAlgorithms/match* subdirectory in your svn repository. You will test your implementation against several provided sample graphs. Be sure to include the code required to update the performance statistics. These include the following.

   maxdInit — On completion, this variable should equal the total number of microseconds spent on all initialization within the *maxdMatch* method, before proceeding to compute the matching.

   fpInit — On completion, this variable should equal the total number of microseconds spent on initialization with *findpath* method, summed over all calls to *findpath.* Initialization includes all code proceeding the "main loop" where the algorithm builds the tree, in order to find a path to extend the matching.

   fpLoop — On completion, this variable should equal the number of microseconds spent in the main loop of the *findpath* method, subbed over all calls to *findpath.*

   extend — On completion, this variable should equal the total number of microseconds spent executing the *extend* method.

   total — On completion, this variable should equal the total number of microseconds spent computing the matching.

B. In this part, you will develop a method called *edgeColor* that implements the algorithm described above. You will find a skeleton of this method in the *graphAlgorithms/match*

subdirectory. You will test your implementation against several provided sample graphs. Be sure to include code to update the performance statistics, by accumulating the values computed in each call to *maxdMatch*.

C. In this part, you will be evaluating the performance of *edgeColor* and *maxdMatch* on random graphs. You will find detailed instructions in the lab report template. The results of this analysis will show that a large fraction of the running time is associated with initialization code required in the *findpath* method of the *maxdMatch* algorithm.

D. In this part, you will design a faster version *edgeColor* algorithm, by modifying the implementation of *maxdMatch*. Name the new components *fedgeColor* and *fmaxdMatch.* Use the results from part *C* to guide your decisions about how to speedup *maxdMatch.* Your improved version should reduce the running time by at least a factor of 10. You will repeat measurements from part *C* for the new version, to demonstrate the expected improvement in performance.