CS 170 Algorithms
Fall 2013 Satish Rao — HW 5

1. (2 + 6 + 6 + 6 pts.) Even Faster DFS

(a) At each vertex, instead of iterating through all children edges, iterate only through the children edges stored in L. So, we only look at the tree edges stored in L. There are $|V|-1$ edges in L in total, so the $|E|$ part of $O(|V|+|E|)$ in DFS runtime now only takes $O(|V|)$ time. So the total time is $O(|V|+|V|)=O(|V|)$.

The reason this works is because in DFS only the tree edges make a difference - any other edge is ignored. So, it is legal to only consider tree edges in the DFS.

(b) Consider the following graph G where the special vertex is S:

$$\begin{tikzcd} S \arrow[r] \arrow[d] & B \arrow[dl] \arrow[d] \arrow[dr] & \\ A & D \arrow[r] & C \end{tikzcd}$$

Choose L to be the edges from an alphabetical DFS, so $L=\{SA,SB,BC,BD\}$, and the edge e to be added is AD.

After adding the edge we get the following graph:

$$\begin{tikzcd} S \arrow[r] \arrow[d] & B \arrow[dl] \arrow[d] \arrow[dr] & \\ A \arrow[r] & D \arrow[r] & C \end{tikzcd}$$

The algorithm outputs $L=\{SA,SB,BC,BD\}$ or $L=\{SA,SB,BC,AD\}$. Suppose one of these is correct.

Since BA is not a tree edge (in both cases), SA must have been visited before SB.

So, AD is visited before BD, and so AD is a tree edge and $L=\{SA,SB,BC,AD\}$, where we visit SA, AD first.

But then we have to visit DC, and so DC is a tree edge. But DC is not in L. Contradiction.

(c) Since $\texttt{pre}(u) > \texttt{pre}(v)$, v is visited before u in the DFS. Suppose we add the edge (u,v) to G and then run the same DFS (i.e. use the same tie-breaking scheme). Then v will be visited before u, and so by the time we look at the edge (u,v), since v has already been visited the edge is ignored, and so DFS proceeds exactly the same way as before. So, L remains exactly the same.

So, the algorithm is: Add the edge to G and don't update L. This takes $O(1)$ time.

(d) Suppose we add the new vertex and edges to the graph, and run a DFS with the same tie-breaking scheme, except that it now prefers to choose v over every other vertex. Then, the DFS will proceed as usual, and at the first chance it gets it will use an edge (u_i,v). This will be the u_i that was visited first out of $u_1,u_2,\cdots u_d$, which is the u_i with the lowest pre number. The DFS then visits v, sees that there are no outgoing edges, and so it goes back to u_i and continues exactly as before. So, $L' = L \cup \{(u_i,v)\}$, that is it gets one new tree edge, which is (u_i,v). This tree edge should be inserted at the beginning of the list, because it was visited before any other children of u_i.

So, the algorithm is: Add the edge to G normally. Find the u_i with the lowest pre number, and add (u_i, v) to L at the beginning of the list of children edges in L.
It takes $O(d)$ time to find the u_i, and then it takes $O(d_L(v))$ time to add the edge to the beginning of the list. Since the indegree and outdegree of a vertex must always be $O(|V|)$, the total running time is $O(|V|) + O(|V|) = O(|V|)$.

(e) **This part will not be graded.** For each vertex v, store a linked list of the children edges of v in the DFS tree.

2. (2 + 8 pts.) Constructing the Graph for Fast DFS

(a) Consider the following graph:

$$\begin{tikzcd}
A \arrow[d, bend right] \\
B \arrow[u, bend right]
\end{tikzcd}$$

If we add A first, then the edge BA never gets added.
If we add B first, then the edge AB never gets added.

(b) The property is that G is a DAG.

First we prove that if G can be constructed using the algorithm, then G is a DAG.
Suppose the vertices are added to G in the order $v_1, v_2, \cdots v_n$.
Then if we have an edge (v_i, v_j) it must have been added while adding v_j and so we have $i < j$.
Thus, the order $v_1, v_2, \cdots v_n$ is a linearization of the graph, which means that G must be a DAG.

Next we prove that if G is a DAG, it can be constructed using the algorithm.
Since G is a DAG, we can linearize it, giving the order $v_1, v_2, \cdots v_n$.
Suppose we then construct the graph by adding the vertices in linearized order.
By the definition of linearization, any edge (v_i, v_j) must satisfy $i < j$.
So, when adding the vertex v_j, v_i is already in the graph and so it is legal to add (v_i, v_j).
So, every edge is added to G and the algorithm is successful.

3. (10 pts.) Problem 3.22

Consider any graph G. In the metagraph of G, if there are two (or more) source strongly connected components, then there can never be a vertex which can reach both of the source SCCs. So, if G is one-way connected, then there must be only one source SCC, and the special vertex v must be in this source SCC.

Moreover, any vertex u in this source SCC can reach v, and so all vertices in G are also reachable from u. So, if G is one-way connected, then all vertices can be reached from any vertex in the source SCC of G.

This gives the following algorithm - run a DFS on G from any starting node, and find the vertex v with the highest post number (which must be in a source SCC). Then run a DFS from v and check that all vertices are reachable from v. There are 2 DFS's, so this takes $O(|V| + |E|)$ time.

Clearly, if the algorithm returns true, then G is one-way connected (it actually finds the special vertex v). If the algorithm returns false, we know that there is a vertex in a source SCC of G which cannot reach all vertices, and by the preceding discussion this means that G is not one-way connected.

4. (2 + 8 pts.) DFS edge types in BFS

(a) Suppose we have a forward edge (u,v) from vertex u with depth k in the BFS tree. Since (u,v) is a forward edge, v is a non-child descendant of u and so has depth $\geq k+2$.

BFS has the property that the shortest path from the starting vertex s to a vertex v with depth k in the BFS tree has k edges. So, the shortest path to v has at least $k+2$ edges, and the shortest path to u has k edges. However, if we take the shortest path to u and add the edge (u,v), we get a path to v with $k+1$ edges. Contradiction.

(b) In the corresponding algorithm for DFS, we use the pre and post numbers to classify edges. The key insight is that if we remove the non-tree edges from the graph and run DFS again, we will get *exactly the same pre and post numbers*. This is because in normal DFS, any non-tree edge is ignored, and so it has no effect on the pre and post numbers. So, another valid way of classifying edges for DFS is:

1. Run DFS on the graph and construct the DFS tree.
2. Run DFS on the DFS tree to generate pre and post numbers.
3. Classify edges using these pre and post numbers.

Now if we look at steps 2 and 3, we can see that it does not depend at all on the fact that we are using a DFS tree - it would work for any tree. In particular, it would also work for the BFS tree, which gives the following algorithm:

1. Run BFS on the graph and construct the BFS tree.
2. Run DFS on the BFS tree to generate pre and post numbers.
3. Classify edges using these pre and post numbers.

The correctness of this algorithm follows immediately from the correctness of the algorithm for classifying edges in DFS.

Construction of the BFS tree takes $O(|V|+|E|)$ time. Running DFS on the BFS tree takes $O(|V|)$ time (since a tree has $O(|V|)$ edges). Classifying each edge takes $O(|E|)$ time. So, the algorithm takes $O(|V|+|E|)$ time in total.

5. (10 pts.) Modified Problem 4.9

Dijkstra's algorithm will always work on such a graph.

Notice that in any path starting from s, the only edge that can have a negative weight is the first one. (If the i^{th} edge has negative weight with $i \geq 2$, then edge $i-1$ would lead to s which contradicts the assumption that there are no incoming edges to s.)

So, if the shortest path to v is of the form $s \to \cdots \to u \to v$ where $u \neq s$ (so the path has at least 2 edges), then the edge (u,v) has positive weight and so u is closer to s than v. This is exactly the property that was used in the proof of correctness in the textbook (see pg. 123), and the same proof applies here. (See the inductive hypothesis on pg. 124.) The only difference is the base case - instead of choosing $d = 0$ when $R = \{s\}$, we need to choose $d = w < 0$ when $R = \{s, u\}$, where the edge (s, u) is the most negative edge in the graph, so that the edges with negative weights are already taken care of and do not need to be considered in the inductive step.