CMPS 101
Fall 2011
Homework Assignment 8 Solutions

1. (1 Point)
   Let $G=(V,E)$ be a weighted directed graph and let $x \in V$. Suppose that after Initialize(G, s) is executed, some sequence of calls to Relax( ) causes $d[x]$ to be set to a finite value. Prove that G contains an s-x path of weight $d[x]$. (Hint: use induction on the number of calls to Relax( ).

   Proof: Let n denote the length of the relaxation sequence. If $n=0$, then the only d-value which is finite after Initialization is that of the source s. Indeed, G does contain an s-s path of weight $d[s]=0$, namely the trivial path. The base case is therefore verified.

   Let $n>0$, and assume for any vertex x, that if $d[x]$ achieves a finite value during a sequence of fewer than n relaxations, then there exists an s-x path in G of weight $d[x]$. Now let $y \in V$ and consider a sequence of n relaxations in which $d[y]$ becomes finite. An edge of the form (x, y) must have been relaxed during this sequence, for some vertex x. On that relaxation step, $d[y]$ was set to $d[x]+w(x, y)$. Since we suppose that this number is finite, $d[x]$ must have been finite before Relax(x, y) was executed. Thus $d[x]$ became finite during a sequence of fewer than n relaxations, and by our induction hypothesis, there must exist an s-x path in G of weight $d[x]$. That path, followed by the edge (x, y), constitutes an s-y path in G of weight $d[x]+w(x, y)=d[y]$. ///

2. (1 Point) 24.1-3 p. 654
   Given a weighted directed graph $G=(V,E)$ with no negative-weight cycles, let m be the maximum over all vertices $x \in V$ of the minimum number of edges in a shortest path from the source $s \in V$ to x. (Here, the shortest path is by weight, not by the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m+1$ passes, even if m is not known in advance.

   Solution:
   We reproduce Bellman-Ford here for reference:

   <u>BellmanFord(G, s)</u>
   1. Initialize(G, s)
   2. for $i=1$ to $|V|-1$
   3.   for each edge $(u,v) \in E$
   4.   Relax(u, v)
   5. for each edge $(u,v) \in E$
   6.   if $d[v]>d[u]+w(u,v)$
   7.   return false
   8. return true

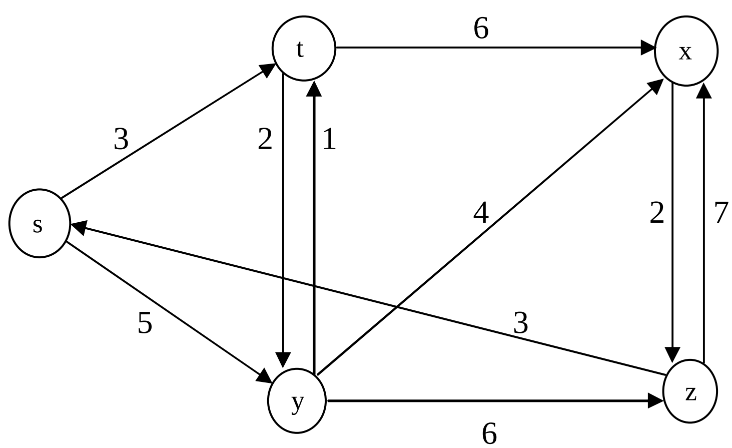
   Note that we cannot simply alter the **for** statement on line 2 to say "for $i=1$ to $m+1$", since the value of m is not known ahead of time. Instead we modify Bellman-Ford so that loop 2-4 terminates as soon as one complete pass over the edge set results in no d-values being changed. Obviously no d-values will be changed by performing any further passes, so if we accept the correctness of Bellman-

Ford (Lemma 24.2 and Theorem 24.4), the d and π values must be correct at that point. It remains only to show that this rule causes loop 2-4 to terminate after $m+1$ passes. To prove this it is sufficient to show that the d-values are correct after exactly m passes. This follows from the path-relaxation property (Lemma 24.15) which says:

> If $p=(v_0,v_1,\ldots,v_k)$ is a shortest path from $s=v_0$ to v_k, and the edges of p are relaxed in the order (v_0,v_1), (v_1,v_2), …., (v_{k-1},v_k), then $d[v_k]=\delta(s,v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.

Each of the edges (v_0,v_1), (v_1,v_2), …., (v_{k-1},v_k) will be relaxed exactly once on each pass over the edge set, so k iterations of loop 2-4 suffice to correctly set the d-value of v_k. But by our definition of m, every vertex v (which is reachable from s) lies at the end of a shortest s-v path containing at most m edges, hence m iterations suffice to correctly set the d-values of all vertices in G. ///

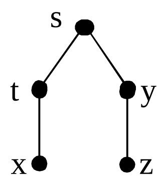
3. (1 Point) 24.3-1 p. 662
Run Dijkstra's algorithm on the directed graph of Figure 24.2 p. 648 (pictured below), first using vertex s as the source and then using vertex z as the source. Show the d and π values and the vertices in set S after each iteration of the **while** loop.



Solution:
With s as source:

	d/π-values after i-th iteration of while					
Vertex	0	1	2	3	4	5
s	0/ nil	0/ nil	0/ nil	0/ nil	0/ nil	0/ nil
t	∞/ nil	3/ s	3/ s	3/ s	3/ s	3/ s
x	∞/ nil	∞/ nil	9/ t	9/ t	9/ t	9/ t
y	∞/ nil	5/ s	5/ s	5/ s	5/ s	5/ s
z	∞/ nil	∞/ nil	∞/ nil	11/ y	11/ y	11/ y
Set S	$\varnothing$	{s}	{s, t}	{s, t, y}	{s, t, y, x}	{s, t, y, x, z}

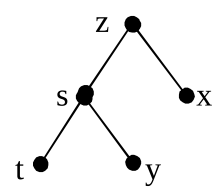
Predecessor subgraph:



With z as source:

	d/π -values after i-th iteration of while					
Vertex	0	1	2	3	4	5
s	∞/ nil	3/ z	3/ z	3/ z	3/ z	3/ z
t	∞/ nil	∞/ nil	6/ s	6/ s	6/ s	6/ s
x	∞/ nil	7/ z	7/ z	7/ z	7/ z	7/ z
y	∞/ nil	∞/ nil	8/ s	8/ s	8/ s	8/ s
z	0/ nil	0/ nil	0/ nil	0/ nil	0/ nil	0/ nil
Set S	$\varnothing$	{z}	{z, s}	{z, s, t}	{z, s, t, x}	{z, s, t, x, y}

Predecessor subgraph:



4. (1 Point) 24.3-6 p. 663
We are given a directed graph $G=(V,E)$ on which each edge $(u,v)\in E$ has an associated value $r(u,v)$, which is a real number in the range $0\le r(u,v)\le 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u,v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Solution:
Let p be a directed x-y path consisting of vertices: $x=v_0, v_1, v_2, \ldots, v_k = y$. Since the probabilities associated with each edge are independent, the probability that no edge along p fails is given by $r(p)=\prod_{i=1}^{k} r(v_{i-1}, v_i) = r(v_0,v_1)\cdot r(v_1,v_2)\cdots r(v_{k-1},v_k)$. The most reliable x-y path which we seek, is the one that maximizes this quantity $r(p)$. Dijkstra's algorithm can be used to find this path by carefully defining an appropriate weight function on edges. Given $(u,v)\in E$, define $w(u,v)=-\log(r(u,v))$, where the log function can have any base greater than 1. Since $0\le r(u,v)\le 1$ we have $-\infty\le \log(r(u,v))\le 0$, and hence $0\le w(u,v)\le\infty$. Edge weights are therefore non-negative (and some may be infinite.) Running Dijkstra's algorithm on the source x will determine an x-y path which minimizes the quantity

$$\begin{aligned}
w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} -\log(r(v_{i-1}, v_i)) \\
&= -\sum_{i=1}^{k} \log(r(v_{i-1}, v_i)) \\
&= -\log\left(\prod_{i=1}^{k} r(v_{i-1}, v_i)\right) \\
&= -\log(r(p)).
\end{aligned}$$

But then p must maximize the quantity $\log(r(p))$, and since log is an increasing function, the path p also maximizes $r(p)$ as required. The following algorithm determines the most reliable directed x-y path in G.

Max-Reliable(G, x, y, r)
1. for each $(u,v) \in E(G)$
2. $\quad w(u,v) = -\log(r(u,v))$
3. Dijkstra(G, w, x)
4. PrintPath(G, x, y)

///

5. (1 Point) 12.1-5 p. 289

Argue that since sorting n elements takes $\Omega(n\log n)$ time in worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n\log n)$ time in worst case.

Solution:

Let $T(n)$ denote the worst case run time of some comparison-based algorithm that constructs a BST from an arbitrary list of n elements. If we call this algorithm, then follow it with a call to InOrderTreeWalk() on the resulting BST, we obtain a comparison based algorithm that sorts an n element list. Since InOrderTreeWalk() runs in time $\Theta(n)$, this sorting algorithm has worst case run time $T(n)+\Theta(n)$. But any comparison based sorting algorithm has worst case run time $\Omega(n\log n)$, whence $T(n)+\Theta(n)=\Omega(n\log n)$, and therefore $T(n)=\Omega(n\log n)-\Theta(n)=\Omega(n\log n)$. ///

6. (1 Point) 12.2-5 p. 293

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Solution:

Let x be a node in a BST having two children. After printing key[x], a call to InOrderTreeWalk(x) would call itself recursively on the subtree rooted at right[x]. (See pseudo-code on page 288.) The algorithm will call itself on left[right[x]], then on left[left[right[x]]], and will continue to call itself recursively on left children in this subtree until it reaches a node y with no left child, at which point it will print key[y]. This node y is the successor of x, since its key is printed immediately after that of x. Therefore the successor of x has no left child.

Now let z be the predecessor of x. Then x is the successor of z. Assume to get a contradiction, that z has a right child. The above argument with z in place of x, and x in place of y, shows that x has no left child, contradicting that x has two children. Our assumption was therefore false, and hence z has no right child. ///

7. (1 Point) 12.2-6 p. 293
Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

Solution:
Our hypotheses say that $\text{right}[x] = \text{nil}$, that $\text{key}[x] < \text{key}[y]$, and that there is no node z for which $\text{key}[x] < \text{key}[z] < \text{key}[y]$. To show that y is an ancestor of x, first observe that y cannot be a descendent of x. This follows from the fact that all descendents of x lie in it's left subtree (since it has no right subtree), and all such descendents must have keys smaller than key[x], by the BST properties. Assume to get a contradiction that y is a cousin of x. Let z denote the lowest common ancestor of both x and y. Then x and y must lie in *different* subtrees of z. For instance, if they were both in z's left subtree, then left[z] would be a common ancestor of x and y, contradicting our choice of z as their lowest common ancestor. If x were in z's left subtree and y in the right, we would have $\text{key}[x] < \text{key}[z] < \text{key}[y]$. If x were in z's right subtree and y in the left, we would have $\text{key}[y] < \text{key}[z] < \text{key}[x]$. Both inequalities contradict the definition of y as the successor of x. We conclude that y must be an ancestor of x. Note also that since $\text{key}[x] < \text{key}[y]$, x must lie in y's left subtree, and hence x is a descendent of $\text{left}[y]$. In other words, $\text{left}[y]$ is also an ancestor of x (and possibly equal to x).

Now assume, to get a contradiction, that w is an ancestor of x whose left child is also an ancestor of x, and that w is *lower* in the tree than y is. Thus w is itself a descendent of y, and since left[y] is an ancestor of x, it must be that w belongs to y's left subtree. (Note it is possible that $w = \text{left}[y]$.) Therefore $\text{key}[w] < \text{key}[y]$. But since $\text{left}[w]$ is an ancestor of x, we know that x is in the left subtree of w, whence $\text{key}[x] < \text{key}[w]$. Thus $\text{key}[x] < \text{key}[w] < \text{key}[y]$ contradicting yet again that y is the successor of x. Therefore no such node w can exist. This completes the proof. ///