CS161: Design and Analysis of Algorithms **Summer 2004**

Problem Set #6 Solutions

General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.

Throughout this entire problem set, your proofs should be as formal as possible. However, when asked to state an algorithm, you do not need to give pseudocode or prove correctness at the level of loop invariants. Your running times should always be given in terms of $|V|$ and/or $|E|$. If representation of the graph is not specified, you may assume whichever is convenient.

1. **[19 points]** Gabow's Scaling Algorithm for Single-Source Shortest Paths

   (a) **[5 points]** Do problem 24-4(a) on page 616 of CLRS.

   Answer: Since the edge weights are all positive, we can use Dijkstra's algorithm to find the shortest paths from the start vertex to all other vertices. The running time of Dijkstra's algorithm using a binary heap is $O(E \lg V)$. The $\lg V$ term comes from the V calls to extract-min and the E calls to decrease-key.

   Because of the constraints that the edge weights are integers and that the shortest path distances are bounded by $|E|$, we can do better. We can use a method similar to counting sort to maintain the list of vertices. We know that the weights of the path to each vertex will always be an integer between 0 and $|E|$, so we can keep an array SHORTEST of linked lists for each possible value. For Dijkstra's algorithm, we need to implement the INSERT, DECREASE-KEY, and EXTRACT-MIN functions. INSERT is easy. If we want to insert a vertex that is reachable in length i, we simply add it to the beginning of the linked list in SHORTEST[i], which is $O(1)$. To call DECREASE-KEY on a vertex v, we remove v from it's current linked list ($O(1)$), decrease it's key to i, and insert it into SHORTEST[i], for a total time of $O(1)$. To EXTRACT-MIN, we notice that throughout the running of Dijkstra's algorithm, we always extract vertices with shortest paths of non-decreasing value. So, if the previous vertex was extracted at value i, we know that there can be no vertices in the array at values less than i. So, we start at SHORTEST[i] and if it is non-empty, we remove the first element from the linked list. If it is empty, we move up to SHORTEST[$i + 1$] and repeat. The actual extraction takes time $O(1)$. Notice that the scanning for a non-empty list could take time $O(E)$ since there are a total of E lists, but since we never backtrack,

this $O(E)$ scan of the lists is averaged over the $O(V)$ calls to EXTRACT-MIN, for an amoritized cost of $O(E/V)$ each.

Our total running time therefore includes V calls to INSERT ($O(V)$), E calls to DECREASE-KEY ($O(E)$) and V calls to EXTRACT-MIN ($O(V+E)$) for a total running time of $O(V+E)$. Since we know $|E| > |V| - 1$, the $O(E)$ term dominates for a running time of $O(E)$.

(b) **[1 point]** Do problem 24-4(b) on page 616 of CLRS.

Answer: Using part (a), we can show how to compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time. We know that $\delta_1(s, v)$ is computed using the edge weight function w_1, which only uses the first significant bit of the actual edge weights. Therefore, the weights w_1 are always either 0 or 1. The maximum number of edges on a shortest path is $|V| - 1$ since it can have no cycles. Since the maximum edge weight w_1 is 1, the maximum weight of a shortest path is $|V| - 1$. Therefore, we have the condition that for all vertices $v \in V$, we have $\delta_1(s, v) \leq |V| - 1 \leq |E|$ by assumption. From part (a), we know that we can compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time.

(c) **[3 points]** Do problem 24-4(c) on page 616 of CLRS.

Answer: We know that w_i is defined as $\lfloor w(u, v)/2^{k-i} \rfloor$. We will show that either $w_i(u, v) = 2w_{i-1}(u, v)$ or $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Weight w_i is the i most significant bits of w. We can get w_i from w_{i-1} by shifting w_{i-1} to the left by one space and adding the ith significant bit. Shifting to the left is equivalent to multiplying by 2 and the ith significant bit can be either 0 or 1. So, we have that $w_i(u, v) = 2w_{i-1}(u, v)$ if the ith significant bit is a zero or $w_i(u, v) = 2w_{i-1}(u, v)+1$ if the ith significant bit is a one.

We now want to prove that $2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$. The shortest path $\delta_i(s, v)$ has weight

$$\begin{aligned}
\delta_i(s, v) &= \min \sum_{e \in Path(s,v)} w_i(e) \\
&\geq \min \sum_{e \in Path(s,v)} 2w_{i-1}(e) \\
&= 2 \cdot \min \sum_{e \in Path(s,v)} w_{i-1}(e) \\
&= 2\delta_{i-1}(s, v)
\end{aligned}$$

with the inequality holding because the $w_i(e)$ is greater than or equal to $2w_{i-1}(e)$ since it equals either that or that plus one. The last equality holds because the minumum weight of any path from s to v using the weight function w_{i-1} is equal to the shortest path distance from s to v using w_{i-1}.

Similarly,

$$\begin{aligned}
\delta_i(s,v) &= \min \sum_{e \in Path(s,v)} w_i(e) \\
&\leq \min \sum_{e \in Path(s,v)} (2w_{i-1}(e) + 1) \\
&= \min \left(2 \sum_{e \in Path(s,v)} w_{i-1}(e) + \sum_{e \in Path(s,v)} 1 \right) \\
&\leq \min \left(2 \sum_{e \in Path(s,v)} w_{i-1}(e) + |V| - 1 \right) \\
&= 2\delta_{i-1}(s,v) + |V| - 1
\end{aligned}$$

The first inequality holds because $w_i(e)$ is less than or equal to $2w_{i-1}(e) + 1$ by similar reasoning as above. We then set the minimum weight of any paths from s to v using the weight function w_{i-1} equal to the shortest path as above. The second inequality holds because the minimum path length is bounded by $|V| - 1$ because it can have no cycles.

(d) **[3 points]** Do problem 24-4(d) on page 616 of CLRS.
Answer: We define for $i = 2, 3, \ldots k$ and all $(u, v) \in E$,

$$\hat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v)$$

We wish to prove for all $i = 2, 3, \ldots, k$ and all $u, v \in V$ that $\hat{w}_i$ of edge (u, v) is a nonnegative integer. We can prove this starting from the triangle inequality.

$$\begin{aligned}
\delta_{i-1}(s,v) &\leq \delta_{i-1}(s,u) + w_{i-1}(u,v) \\
2\delta_{i-1}(s,v) &\leq 2\delta_{i-1}(s,u) + 2w_{i-1}(u,v) \\
2\delta_{i-1}(s,v) &\leq 2\delta_{i-1}(s,u) + w_i(u,v) \\
0 &\leq w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v) \\
0 &\leq \hat{w}_i
\end{aligned}$$

Thus, $\hat{w}_i$ is never negative. It is an integer since all the edge weights are always integers and thus all the shortest path distances are always integers.

(e) **[4 points]** Do problem 24-4(e) on page 617 of CLRS.
Answer: We define $\hat{\delta}_i(s, v)$ as the shortest path weight from s to v using $\hat{w}_i$. We want to prove for $i = 2, 3, \ldots k$ and all $v \in V$ that

$$\delta_i(s,v) = \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)$$

and that $\hat{\delta}_i \leq |E|$.

We prove this by expanding the $\hat{\delta}_i(s,v)$ term.

$$\begin{aligned}
\hat{\delta}_i(s,v) &= \min \sum_{e \in Path(s,v)} \hat{w}_i(e) \\
&= \min(\hat{w}_i(s,x_1) + \hat{w}_i(x_1,x_2) + \cdots + \hat{w}_i(x_n,v)) \\
&= \min(w_i(s,x_1) + 2\delta_{i-1}(s,s) - 2\delta_{i-1}(s,x_1)) + \\
&\quad (w_i(x_1,x_2) + 2\delta_{i-1}(s,x_1) - 2\delta_{i-1}(s,x_2)) + \cdots + \\
&\quad (w_i(x_n,v) + 2\delta_{i-1}(s,x_n) - 2\delta_{i-1}(s,v)) \\
&= \min(2\delta_{i-1}(s,s) - 2\delta_{i-1}(s,v) + \sum_{e \in Path(s,v)} w_i(e)) \\
&= -2\delta_{i-1}(s,v) + \min \sum_{e \in Path(s,v)} w_i(e) \\
&= -2\delta_{i-1}(s,v) + \delta_i(s,v) \\
\delta_i(s,v) &= \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)
\end{aligned}$$

We expand $\hat{\delta}_i(s,v)$ in terms of the weights of edges along the path. We see that if we also expand the $\hat{w}_i$ terms, many things cancel. The term $\delta_{i-1}(s,s) = 0$ since the shortest path from a node to itself is zero regardless of the edge weight function. Also, we know that the minimum path length from s to v using the w_i function is $\delta_i(s,v)$.

From this, we can easily show that $\hat{\delta}_i \leq |E|$ using the results from part (c).

$$\begin{aligned}
\delta_i(s,v) &= \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v) \\
2\delta_{i-1}(s,v) + |V| - 1 &\geq \hat{\delta}_i(s,v) + 2\delta_{i-1}(s,v) \\
\hat{\delta}_i(s,v) &\leq 2\delta_{i-1}(s,v) + |V| - 1 - 2\delta_{i-1}(s,v) \\
\hat{\delta}_i(s,v) &\leq |V| - 1 \\
\hat{\delta}_i(s,v) &\leq |E|
\end{aligned}$$

(f) **[3 points]** Do problem 24-4(f) on page 617 of CLRS.

Answer: If we are given $\delta_{i-1}(s,v)$, we can compute $\hat{w}_i(u,v)$ using the equation in part (d) in time $O(E)$ since we can easily calculate $w_i(u,v)$ and we need to compute it for each edge. In part (e), we showed that $\hat{\delta}_i(s,v)$ is bounded by $|E|$. So, we can use the results from part (a) to compute the shortest path distances $\hat{\delta}_i(s,v)$ in time $O(E)$. From these results, we can use the equation in part (e) to calculate the shortest path distances $\delta_i(s,v)$ in time $O(V)$, once for each vertex.

Therefore, we can compute $\delta(s,v)$ in time $O(E \lg W)$. We will simply start by calculating $\delta_1(s,v)$ in time $O(E)$ as we showed in part (b). Then, as we just explained, we can calculate each δ_{i+1} from δ_i in time $O(E)$. In total, we need to calculate up till δ_k. Since $k = O(\lg W)$, the running time of this is $O(E \lg W)$.

2. **[12 points]** Transitive Closure of a Dynamic Graph

   (a) **[3 points]** Do problem 25-1(a) on page 641 of CLRS.

   Answer: When we add a new edge to a graph, we need to update the transitive closure. The only new paths we will add are the ones which use the new edge (u, v). These paths will be of the form $a \rightsquigarrow u \rightarrow v \rightsquigarrow b$. We can find all these new paths by looking in the old transitive closure for the vertices $a \in A$ which had paths to u and for vertices $b \in B$ which v had paths to. The new edges in the transitive closure will then be (a, b), where $a \in A$ and $b \in B$. Since the number of vertices in each set A or B is bounded by V, the total number of edges we'll need to add is $O(V^2)$. If we represent the transitive closure graph G^* with an adjacency matrix, we can very easily find the sets A and B. A will be all the vertices which have a one in the column u and B will be all the vertices which have a one in the row v.

   (b) **[1 point]** Do problem 25-1(b) on page 641 of CLRS.

   Answer: Suppose our graph G consists of two disjoint graphs, each of size $V/2$. Assume that each subgraph is complete, that there are edges between each vertex in the subgraph to all other vertices in the same subgraph. If we then add an edge from a vertex in the first subgraph to one in the second subgraph, we will need to add edges in the transitive closure from every vertex in the first subgraph to every vertex in the second subgraph. This is a total of $V/2 \times V/2 = V^2/4$ edges. Regardless of what our algorithm is for calculating the transitive closure, we will always need to add $\Omega(V^2)$ edges, and thus the running time must be $\Omega(V^2)$.

   (c) **[8 points]** Do problem 25-1(c) on page 641 of CLRS.

   Answer: We will give you an outline of the solution to this problem on the final exam. You will then complete it and analyze its correctness and running time.

3. **[24 points]** Assorted Graph Problems

   (a) **[8 points]** Do problem 22.2-7 on page 539 of CLRS. Assume the edges of T are undirected and weighted, and give your running time in terms of $|V|$ (since $|E| = |V| - 1$).

   Answer: First notice that the shortest path distance between two vertices is given by the length of the path between the two vertices, since there is only one path between any pair of vertices in a tree. So, we are basically asking for the longest simple path between any two leaves in a tree.

   We will keep two items of information at each edge $(u, v) \in T$. We will keep $d[u \rightarrow v]$, which represents the longest simple path distance in the tree from u to v to any leaf. Also, we keep $d[v \rightarrow u]$, defined symmetrically.

   First we arbitrarily root the tree at some vertex r by pointing all the edges outward from r. We will compute the downward distances bottom-up, and then the upward distance top-down.

COMPUTE-DOWN(x)
1 **if** IsLeaf(x)
2 **return**
3 **for** $y \in children(x)$
4 COMPUTE-DOWN(y)
5 $d[x \to y] \leftarrow w(x, y) + \max_{z \in children(y)} d[y \to z]$

After calling COMPUTE-DOWN on the root of the tree r, every edge will have its downward field set. Notice that because of the recursion, the information is actually first computed at the leaves and then propagated upward.

The top-down pass is somewhat tricker, because now we must integrate information from different branches as we descend down the tree. There are two differences: first, the information is now computed at the top first, and at the bottom at the end; second, updating $d[y \to x]$ where x is the parent of y now depends not only on $d[x \to p(x)]$ but also on $d[x \to z]$, where z are other children of x. However, these values were computed on the downward pass; therefore, as long as we compute $d[x \to p(x)]$ before $d[y \to x]$, we have all the information we need.

COMPUTE-UP(x)
1 $p \leftarrow parent(x)$
2 **for** $y \in children(x)$
3 $d[y \to x] \leftarrow w(x, y) + \max_{z \in children(x) \cup \{p\}, z \neq y} d[x \to z]$
4 COMPUTE-UP(y)

Thus, to calculate the diameter, we root the tree arbitrarily at a vertex r, call COMPUTE-DOWN(r) and then COMPUTE-UP(r). The largest value $d[u \to v]$ computed during these procedures is indeed the diameter of the tree.

The first procedure clearly runs in $O(V)$, since every edge $u \to v$ is examined at most twice - once when we $d[u \to v]$, and once when we set $d[p(u) \to u]$.

The second procedure as it is written runs in worst-case time $O(V^2)$, since if we have a node with $\Theta(V)$ chilren, we will be looking through all the downward distance values V times for each maximization step in line 3. However, we notice that the only useful values from that step are the two largest downward $x \to z$ values, since at most one such z can be the y we are currently looking at. Thus, we precompute at each node x the two largest $x \to z$ values, and then use the largest unless $z = y$. Thus, each edge is considered at most a constant number of times, and the running time is $O(V)$ as well.

(b) **[8 points]** Do problem 22-3(b) on page 559 of CLRS. You may assume the result of 22-3(a) without proof.

Answer: From part (a), we know that if there is an Euler tour in the graph G, then the in-degree equals the out-degree for every vertex v. Because of this fact, we know that if we pick a path with unique edges, any time we reach a vertex (use one of the in-degree edges), there must still be a way to leave the vertex (use one of the out-degree edges) except for the first vertex in the path.

Our algorithm therefore is to pick a random starting vertex v. We pick an outgoing

edge from v, (v, u), at random. We move to vertex u and delete edge (v, u) from the graph and repeat. If we get to a vertex with no outgoing edges and it is not equal to v, we report that no Euler tour exists. If the vertex with no outgoing edges is v, we have a cycle, which we represent as $v \to u \to \cdots \to v$.

The problem is that we may not have visited all the edges in the graph yet. This may only occur if there are vertices in our cycle which have outgoing edges that we didn't traverse. Otherwise, our graph would not be connected. If we can efficiently pick one of these vertices u and start a cycle from that point, we know that the new cycle from u will end at u by the same reasoning as above. Therefore, we can connect the two cycles $\{\ldots a \to u \to b \ldots\}$ and $\{u \to c \cdots \to d \to u\}$ by making one big cycle $\{\ldots a \to u \to c \cdots \to d \to u \to b \ldots\}$. This is a valid cycle since we removed all the edges in the first cycle before computing the second cycle, so we still have the property that each edge is used at most once. We assume that when we start the new cycle from u, we keep a pointer to the position of u in the old cycle so that we know where to break the old cycle in constant time.

If we repeat this process until there are no more edges in the graph, we will have an Euler tour. The only problem is how to pick the vertex u that still has outgoing edges from the current cycle. Our original vertex was v. For the first cycle created, we simply walk along the cycle from v and let u equal the first vertex which still has outgoing edges. Then, after adding the next cycle, we start walking from u along the newly merged cycles and so on. Once we have traversed the path from $v \rightsquigarrow u$, we know that none of the vertices in that section can ever have outgoing edges, so we don't need to consider them the second time. We will need to walk over the entire Euler tour at some point in the algorithm, so the total running time of finding the vertex u for all the cycles is $O(E)$. Also, the cost of actually finding the cycles traverses each edge exactly once, so it is also $O(E)$. Therefore, the total running time of the algorithm is $O(E)$.

(c) **[3 points]** Suppose we have a single-source shortest path algorithm A that runs in time $f(|V|, |E|)$ when all edge weights are nonnegative. Give an algorithm to solve problem 24.3-4 on page 600 of CLRS in time $f(|V|, |E|) + O(E)$ by using A as a subroutine.

Answer: The reliability of a path from u to v is the product of the reliabilities of each edge in the path. This is because the probability that the channel does not fail is the probability that all individual edges do not fail. Since the failures or successes of the individual edges are independent, we simply multiply to get the total probability of success.

But, our shortest path algorithm finds the shortest path by summing the edge weights. We need some transformation of our reliabilities so that we can use the shortest path algorithm given. One function we have seen that converts a product of terms to a sum of terms is the log function.

If we define a new weight function $w(u, v) = -\lg(r(u, v))$, then all the edge weights are positive since the log of a positive number less than 1 is negative. As the reliability decreases, the value of the weight of the edge will increase. Since we are trying to find the most reliable path, this will correspond to the shortest

weight path in the graph with the new edge weights.

We can show this more formally, where RPATH represents the reliability of a path.

$$\begin{aligned} RPATH(s,t) &= \prod_{e \in Path(s,t)} r(e) \\ \log RPATH(s,t) &= \sum_{e \in Path(s,t)} \log r(e) \\ -\log RPATH(s,t) &= \sum_{e \in Path(s,t)} -\log r(e) \end{aligned}$$

Since the log function is monotonic as long as the base is greater than 1, if we find the minimum path with edge weights w, we will also find the most reliable path.

The time for the conversion to the weight function w is constant for each edge, so the total time is $O(E)$. We then call the single-source shortest path algorithm which runs in time $f(|V|, |E|)$ and returns our most reliable path. Therefore, the total running time is $f(|V|, |E|) + O(E)$.

(d) **[5 points]** Do problem 25.2-9 on page 635 of CLRS.

Answer: If we have an algorithm to compute the transitive closure of a directed acyclic graph, we can use it to calculate the transitive closure of any directed graph. We first decompose the directed graph G into its strongly connected component graph G_{SCC}. This runs in time $O(V + E)$ and results in a graph with at most V vertices and E edges. We keep track of which set of vertices in G map to each vertex in G_{SCC}. Call this $g(v)$ which returns a stronly connected component in G_{SCC}. We then call the transitive closure algorithm on the graph G_{SCC}. This runs in time $O(f(|V|, |E|))$. All that is left is to convert back from the strongly connected component graph's transitive closure to G's transitive closure. There is an edge (u, v) in the transitive closure of G if and only if there is an edge $(g(u), g(v))$ in the graph G_{SCC}. The forward direction is easy to see. If there is a path from u to v in G, then there must be a path between the component that u belongs to and the one that v belongs to in G_{SCC}. The reverse direction is equally easy. If there is a path between u's strongly connected component and v's strongly connected component in G_{SCC}, then there is a path between some vertex a in $g(u)$ and some vertex b in $g(v)$ in G. But, since u and a are in the same SCC, there is a path from $u \rightsquigarrow a$ and since v and b are in the same SCC, there is a path from $v \rightsquigarrow b$. Therefore, there is a path $u \rightsquigarrow a \rightsquigarrow b \rightsquigarrow v$ in G.

To actually calculate the transitive closure of G this way would require time V^2 since we'd need to look at all pairs. Instead, we will look at the transitive closure of G_{SCC}. For each edge (a, b) in the transitive closure of G_{SCC}, we assume that we know the set of vertices in G that map to a and b, call these sets $g^{-1}(a)$ and $g^{-1}(b)$ respectively. We will add the edge (x, y) for all vertices $x \in g^{-1}(a)$ and

$y \in g^{-1}(b)$. This will take time equal to the number of edges in the transitive closure of G, $O(E^*)$.

Therefore the total running time for the algorithm is the time to build the strongly connected components ($O(V+E)$), the time to run the transitive closure algorithm on G_{SCC} ($O(f(|V|, |E|))$ and the time to convert back to the transitive closure of G ($O(E^*)$). Since we know that $E < E^*$, the total time is $O(f(|V|, |E|)+V+E^*)$.