Homework 10, Solution set

1. If G is strongly connected we want to visit every vertex in G, which can be done as for each $v \in V$ there is a path from s to v and a path from v back to s. So it suffices to concatenate all these paths for each $v \in V \setminus \{s, t\}$ and follow them with the path from s to t. So the reward is simply the sum of the rewards for each vertex $v \in V$. Note that we don't have to explicitly build the path.

In general, we begin by computing the meta-graph $H = (W, F)$ induced by G's strong components. Clearly, if our path visits one vertex in a strong component it might as well visit every vertex in the strong component. Consequently, we define the reward for a strong component C to be the sum of the rewards for the vertices in C. We create an array $\mathrm{CtRwd}[1:k]$, where k is the number of strong components and $\mathrm{CtRwd}[i]$ is the reward for the ith component.

Recall that in the second DFS, the DFS explores the strong components one by one. For each strong component, as each vertex is explored, we increment a running total of the rewards of the vertices explored. The final value of this sum will be the reward of that strong component.

Therefore our task reduces to finding a maximum reward path from s's strong component c_s in H to t's strong component c_t in H. This is readily computed by a DFS of H, which uses the following recursive formula for the maximum rewards:

$$\mathrm{MaxRwd}[x] = \max_{(x,y)\in F}\{\mathrm{CtRwd}[x] + \mathrm{MaxRwd}[y]\}.$$

$\mathrm{MaxRwd}[w]$ is initialized to $-\infty$ for all $w \in W \setminus \{c_t\}$ and $\mathrm{MaxRwd}[c_t]$ is initialized to $\mathrm{CtRwd}[w]$. The reason for the $-\infty$ is to ensure that we observe a reward only if a path to t is found.

Pseudo-code for the final DFS follows. The initial call will be to $\mathrm{MtDFS}(H, c_s)$ (short for MetaDFS). The array $\mathrm{MtExpl}[\cdot]$ for the meta-graph plays the same role for H as the array Expl for graph G.

MtDFS(H, x)
MtExpl[x] $\leftarrow$ True;
for each edge (x, y) **do**
  if MtExpl[y] = False **then** MtExpl[y] $\leftarrow$ True; MtDFS(y)
  end if
  MaxRwd[x] = max{MaxRwd[x], CtRwd[x] + MaxRwd[y]}
end for

Our modification of the DFS algorithm for computing strong components will add $O(1)$ processing time for each vertex, thus the running time remains at $O(n + m)$.

The final run of DFS on graph H adds $O(1)$ time for each edge in H, i.e. at most $O(m)$ time overall. Thus it too runs in time $O(n + m)$.

2. Let $\mathrm{dist}(u)$ be the distance to u from s; we will store these values as they are computed in array $\mathrm{Dist}[1:n]$. $\mathrm{NumPth}(v)$, the number of paths to vertex v at distance $d + 1$ from s, is given by $\mathrm{NumPth}(v) = \sum\{\mathrm{NumPth}(u) \mid \mathrm{dist}(u) = d \text{ and } (u, v) \in E\}$. We initialize $\mathrm{NumPth}(v)$ to 0 for all $v \in V \setminus \{s\}$ and $\mathrm{NumPth}(s) = 1$. We then compute $\mathrm{NumPth}(v)$ and $\mathrm{Dist}[v]$ as we perform a BFS from s as follows.

```
PthCntBFS(G, s)
for v = 1 to n do NumPth(v) ← 0; Expl[v] ← False
end for
Q ← φ;
EnQueue(Q, s); Expl[s] ← True; NumPth(s) ← 1; Dist[s] ← 0;
while Q ≠ φ do
    u ← DeQueue(Q)
    for each edge (u, v) do
        if Expl[v] = False then
            EnQueue(Q, v); Expl[v] ← True; Dist[v] ← Dist[u] + 1
        end if
        if Dist[v] = Dist[u] + 1 then
            NumPth[v] ← NumPth[v] + NumPth[u]
        end if
    end for
end while
```

Each iteration of the for loop takes $O(1)$ time as in the unenhanced BFS, so the augmented BFS also runs in time $O(n + m)$.

3. We will measure the size of a path as a pair (length, number of edges). Given two paths P and Q, we view P as being a shorter path if either length(P) $<$ length(Q) or length(P) $=$ length(Q) and NumEdges(P) $<$ NumEdges(Q). We run Dijkstra's algorithm with this new way of comparing path sizes.

Pseudo code for the modified Dijkstra's algorithm follows.

```
Dijkstra(G, v)
for v = 1 to n do
    Dist[v] ← maxint; NumEdges[v] ← maxint; Pred[v] ← nil
end for
Dist[s] ← 0; NumEdges[s] ← 0;
EnQueue(Q, V, Dist);
while not Empty(Q) do
    u ← DeleteMin(Q);
    for each edge (u, v) do
        if Dist[v] > Dist[u] + length(u, v) then
            Dist[v] ← Dist[u]+length(u, v); NumEdges[v] ← NumEdges[u]+1; Pred[v] ← u;
            DecreaseKey(Q, v, Dist[v]);
        else
            if Dist[v] = Dist[u] + length(u, v) and NumEdges[u] + 1 < NumEdges[v] then
                NumEdges[v] ← NumEdges[u] + 1; Pred[v] ← u;
                DecreaseKey(Q, v, Dist[v])
            end if
        end if
    end for
end while
```

Each iteration of the for loop still runs in $O(1)$ time, implying that the modified Dijkstra's algorithm runs in $O((n+m)\log n)$ time.

The reconstruction of the tree of shortest paths can be done as for Dijkstra's algorithm, for it consists of the edges $\cup_{v \in V\setminus\{s\} \text{ and } \mathrm{Pred}[v]\neq \mathrm{nil}}(\mathrm{Pred}[v], v)$, and it takes $O(n)$ time to build.

4. For each vertex v, we need to maintain the set $S[v]$ of edges into v that are on a shortest path to v; we store these sets using linked lists. Initially, all these sets are empty. Whenever a new path to v of length equal to the current minimum is found, we add the last edge on this path to $S[v]$, and whenever a new shorter path to v is found, we reset $S[v]$ to consist of the last edge on this path.

Then the graph A of edges on the shortest paths is simply $(V, \cup_{w\in V} S[w])$. To form the graph A, we initialize its adjacency lists to empty, which takes $O(n)$ time, and then traverse the sets $S[v]$, and for each edge $e = (u, v) \in S[w]$, add e to u's adjacency list, which takes $O(1)$ time per edge, or $O(m)$ time in total. This yields a runtime of $O(n + m)$ for building A following the run of the modified Dijkstra's algorithm.

Pseudo code for the modified Dijkstra's algorithm follows.

Dijkstra(G, v)
for $v = 1$ to n **do**
    Dist$[v] \leftarrow$ maxint; $S[v] \leftarrow \phi$
end for
Dist$[s] \leftarrow 0$;
EnQueue(Q, V, Dist);
while not Empty(Q) **do**
    $u \leftarrow$ DeleteMin(Q);
    for each edge (u, v) **do**
        if Dist$[v] >$ Dist$[u]$ + length(u, v) **then**
            Dist$[v] \leftarrow$ Dist$[u]$ + length(u, v);
            DecreaseKey$(Q, v, \mathrm{Dist}[v])$;
            $S[v] \leftarrow \{(u, v)\}$
        else
            if Dist$[v] =$ Dist$[u]$ + length(u, v) **then**
                $S[v] \leftarrow S[v] \cup \{(u, v)\}$
            end if
        end if
    end for
end while

Each iteration of the for loop still runs in $O(1)$ time plus the time for a DecreaseKey operation, implying that the modified Dijkstra's algorithm runs in $O((n + m)\log n)$ time.