HW10

1. ***Suppose you are given a directed graph G = (V,E), and an array Rwd[1 : n] of non- negative integers, where Rwd[v] is a reward associated with vertex v. You are also given a start vertex s and a destination vertex t. Give an algorithm to calculate the maximum reward on any path from s to t, including paths with cycles, where you collect the reward from each vertex once, however often it is visited. Explain why your solution produces the correct result. Also, justify your running time.***

   > First, we compute the strong components in G, forming the meta-graph H of strong components, which is a dag, like additional question 1. When doing this, we also compute the sum of the reward for each strong component. This step increase $O(n)$ run-time.
   >
   > We use $pathSum[v]$ to store the maximum reward from v to t. Use $Done[v]$ to store if the vertex has been visited.
   >
   > Assume our function name is $FindMaxReward(s,t)$, we can use dynamic programming recursively to calculate the results. Here is the formula.
   >
   > $$pathSum[v] = \begin{cases} max_w\ \{pathSum[w]\} + Rwd(v) & v \neq t \\ Rwd(v) & v = t \end{cases} \quad (1)$$
   >
   > If the vertex v has not been visited, execute $FindMaxReward(v,t)$ and set $Done[v]$ into true, else just use it from the $pathSum[v]$ to reduce the recursive call. This algorithm is based on the fact that we can find t from any path, i.e., any path from s can go to t. If there are paths that cannot find t, we should make the subproblem to 0, which makes sure we don't consider this branch.
   >
   > The initial call is $FindMaxReward(s,t)$. The result is $pathSum[s]$. **This is to calculate the maximum total rewards of paths from s to v, which is $O(|V|+|E|)$ time because each vertex and edge is traversed. Hence, the total is $O(|V|+|E|)$.**

2. *Suppose you are given an undirected graph G = (V, E), and a designated vertex s ∈ V . We define the length of a path in G to be the number of edges on the path. Give a linear time algorithm to determine for each vertex v ∈ V the number of shortest paths from s to v.*

   > We use an array $PathNum[1:n]$ of non-negative integers, where $PathNum[v]$ is the number of paths passing through v. Keep $Expl[u,v]$ for each edge, which stores if (u,v) has been explored. Keep $ShortestLength[v]$ to store the shortest path from s to v.
   >
   > $DeQueue(Q)$ returns the first entry in the queue.
   >
   > Initially, $Expl[u,v] = false$ for all edges (u,v). $PathNum[v] = 0$ for all v.
   > $ShortestLength[v] = 0$ for all v.

```
BFS(Q,s)
EnQueue(Q,s), PathNum[s]<-1, ShortestLength[s]<-1;
While Q is not empty do
  u<-DeQueue(Q)
  for each edge (u,v) do
    if not Expl[u,v] do
      EnQueue(v)
      Expl[u,v]<-true
      if ShortestLength[v]=0 then
        ShortestLength[v]<-ShortestLength[u]+1
        PathNum[v]<-PathNum[v]+PathNum[u]
      else if ShortestLength[v]=ShortestLength[u]+1
        PathNum[v]<-PathNum[v]+PathNum[u]
```

The idea is to add the number of the shortest path to the explored vertex. Due to DFS, the vertex has the shortest path from s when finding it the first time, but it may have more than one shortest path. If we iterate an edge (u, k) to an explored vertex k, if this path has the same length as the shortest path length, we add $PathNum[u]$ to $PathNum[k]$.

We search each vertex once and each edge once and the other operations are $O(1)$. Hence, the total time complexity is $O(|V| + |E|)$

3. *Let G = (V,E) be a directed graph, where all edges have positive lengths, and let s be a designated source vertex. Suppose you want to find shortest paths from s to each vertex v ∈ V and in the event of ties, among these shortest length paths find one with fewest edges. Explain how to modify Dijkstra's algorithm and the path reconstruction, while still achieving the same runtime as Dijkstra's algorithm. Justify your running times and explain why your modifications produce the correct result.*

First, we compute the strong components in G, forming the meta-graph H of strong components, which is a dag. When doing this, we also compute the sum of the reward for each strong component. This step increase $O(n)$ run-time.

Keep $Alarm[v]$ for each vertex when the next nearest distance equals $Alarm[v]$, we will process the vertex v.

Use $EdgeLength[v]$ to record the edges that the path from s to v has.

Use $Path[v]$ to record the shortest length from s to v. This is represented by a string, like "s->v_1->v_2"

Initially, $Alarm[v] = MaxInt$, $Path[v] =$ "" for all $v \in V$.

```
MDJA(Q,s)
Alarm[s]<-0, Path[s]<-"s"
EnQueue all v ∈ V on a PriorityQueue Q with key Alarm[v]
While Q is not empty do
  u<-DeleteMin(Q)
  for each edge (u,v) do
if Alarm[v] > Alarm[u] + length(u,v) then
  Alarm[v]<-Alarm[u]+length(u,v)
  Path[v]<-Path[u]+"->"+v
  EdgeLength[v] = EdgeLength[u]+1
else if Alarm[v] = Alarm[u]+length(u,v) && EdgeLength[v] > EdgeLength[u]+1 then
  Path[v]<-Path[u]+"->"+v
  EdgeLength[v] = EdgeLength[u]+1
```

Here we can use Dijkstra's algorithm directly because all of the edges are positive lengths. We can always make sure the length of the path is getting smaller.

We can use an array $PreVertex[v]$ to store the previous vertex in the shortest path instead of using string, then we can reconstruct the path in reverse order like in question 4.

Hence, as we only increase $O(1)$ run-time at each if statement and the deletion of the queue costs $O(\log n)$, the total time complexity will be $O((|V| + |E|) \log |V|)$. The shortest path of v is $Path[v]$

4. *Suppose you are given a directed graph G = (V, E), where all edges have positive lengths, and you are also give a designated vertex s ∈ V. It may be that there is more than one shortest path from s to any given v ∈ V .*

   By modifying Dijkstra's algorithm and the path recovery procedure, give an algorithm that constructs the directed acyclic graph of shortest paths from s to every other vertex. The modified Dijkstra's algorithm should still run in O((n + m) log n) time and the modified path recovery procedure should run in O(n + m) time. Justify your running times and explain why your modifications produce the correct result.

   As we said in question 3, as all edges have positive lengths, so we can use Dijkstra's algorithm directly.

   Use $MinEdge[v]$ to record the vertices in the shortest path pointing to v by LinkedList. For example, there are two shortest paths passing through v, which are u and k. Then, we should do $MinEdge[v].append(v)$ and $MinEdge[v].append(k)$

   Initially, $Alarm[v] = MaxInt$ for all $v \in V$.

```
DAGSP(Q,s)
Alarm[s]<-0,
EnQueue all v ∈ V on a PriorityQueue Q with key Alarm[v]
While Q is not empty do
  u<-DeleteMin(Q)
  for each edge (u,v) do
if Alarm[v] > Alarm[u] + length(u,v) do
  MinEdge[v].clear().append(u)
 Alarm[v]<-Alarm[u]+length(u,v)
else if Alarm[v] = Alarm[u] + length(u,v) do
  MinEdge[v].append(u)
 Alarm[v]<-Alarm[u]+length(u,v)
```

Here, this only increase $O(1)$ operation in Dijkstra's algorithm, so the time complexity is still $O((n+m)\log n)$.

As we store the vertices in the shortest path pointing to v in $MinEdge$, we should recover the path from the target vertex v to s. If we use an adjacency matrix $AM[u,k]$ to store the DAG, then we iterate the MinEdge from v, assuming we get k, then let $AM[k,v]=1$. If the DAG $H=(V,E)$, then we will iterate $c*max(|V|,|E|)$ times. Hence, the total cost will be $O(n+m)$.