CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)
Homework 9 Problem 1

Solution: (a) We use a layered construction to keep track of whether Yulie has bought empanadas and whether a gas station has been reached. For the layer where a gas station has not been reached, we need to know which vertices can be reached and which cannot. Therefore, we need to run Dijkstra's algorithm once on G not taking gas stations into account. Following is the algorithm.

(1) Run Dijkstra's algorithm on G starting at s to compute *pred*(v), the predecessor of v in the shortest path from s to v. Let *remainingMiles*(v) be the maximum number of miles that Yulie's car can run when she reaches v. It can be defined as follows:

$$\textit{remainingMiles}(v) = \begin{cases} D & v = s \\ \textit{remainingMiles}(p) - l(pv) & \text{otherwise, } p = \textit{pred}(v) \end{cases}$$

The time complexity of computing *remainingMiles* is $O(1)$ and we can put its computation into each iteration of Dijkstra's algorithm. Since Dijkstra's algorithm explores the shortest path, *remainingMiles*(v) must be the maximum remaining number of miles at v.

(2) Construct a graph $G' = (V', E')$, and define edge lengths $l'(e)$ for E':

- $V' = V \times \{NE, E\} \times \{NG, G\}$, where NE means Yulie has not bought empanadas yet, E means Yulie has bought empanadas, NG means a gas station has not been reached, G means a gas station has been reached.
- For each edge $uv \in E$, we add the following to E' and $l'(e)$. We use a and b to generalize empanadas and gas stations.
  - $(u, a, NG)(v, a, NG)$ if *remainingMiles*$(v) \geq 0$ and $u = \textit{pred}(v)$; $l'(e) = l(uv)$
  - $(u, a, G)(v, a, G)$; $l'(e) = l(uv)$
  - $(u, a, NG)(u, a, G)$ if $u \in Y$; $l'(e) = 0$
  - $(u, NE, b)(u, E, b)$ if $u \in X$; $l'(e) = 0$

(3) Let $s' = (s, NE, NG)$. Run Dijkstra's algorithm on G' starting at s' using $l'(e)$ as edge weights. Let *dist*(v) be the shortest distance from s' to v in G'. Return the minimum of *dist*$((t, E, NG))$ and *dist*$((t, E, G))$.

It is easy to also return a shortest path since Dijkstra's algorithm can keep track of the path. Note that *dist*$(v) = \infty$ if there is not a path from s to v in Dijkstra's algorithm. Therefore the algorithm will return ∞ if there is no way to reach the destination. In the new graph G', $|V'| = 4|V|$ and $|E'| \leq |X| + |Y| + 2|E| \leq |V| + 2|E|$. Since the running time for Dijkstra's algorithm is $O(|E| + |V| \log |V|)$, the running time of this algorithm is $O(|V| + 2|E| + 4|V| \log 4|V|) = O(m + n \log n)$.

(b) Let $k = |Y|$. We want to know the "reach" of each gas station to account for the fact that Yulie can only run R miles after reaching a gas station. Therefore, we run Dijkstra's algorithm k times using each gas station as the start node. The layers are concatenated and form a new graph. Then the problem is reduced to an SSSP problem in the new graph. The algorithm is as follows:

(1) Run Dijkstra's algorithm $k+1$ times on G starting from $s, y_1, y_2, ..., y_k$ where $y_i \in Y$. Let *pred*(y_i, v) be the predecessor of v in the shortest path from y_i to v. Each time Dijkstra's algorithm is run starting from y_i, we compute an array *remainingMiles*(y_i, v) using the same technique as in (a) which represents the remaining miles when Yulie reaches v from y_i. It is defined as follows:

$$\textit{remainingMiles}(y_i, v) = \begin{cases} R & v = y_i \\ \textit{remainingMiles}(y_i, p) - l(pv) & \text{otherwise, } p = \textit{pred}(y_i, v) \end{cases}$$

Let $G_1, G_2, ..., G_k$ be the resulting DAGs representing the shortest paths starting from $y_1, y_2, ..., y_k$. Then for each G_i, we remove all edges $\{uv \mid \textit{remainingMiles}(y_i, v) < 0, u = \textit{pred}(y_i, v)\}$.

(2) Construct a new graph $G' = (V', E')$:

- $V' = V \times \{N, R_1, R_2, ..., R_k\} \times \{E, NE\}$. E means Yulie has bought empanadas and NE means Yulie has not bought empanadas. N means Yulie has not reached any gas station yet, R_i means the last gas station reached is y_i.
- For each edge $uv \in E$, we add the following to E' and $l'(e)$. We use r to generalize $\{N, R_1, R_2, ..., R_k\}$ and a to generalize $\{E, NE\}$.
  - $(u, N, a)(v, N, a)$ if *remainingMiles*$(s, v) \geq 0$ and $u = \textit{pred}(s, v)$; $l'(e) = l(uv)$
  - $(u, R_i, a)(v, R_i, a)$ if uv is an edge in G_i; $l'(e) = l(uv)$
  - $(u, r, NE)(u, r, E)$ if $u \in X$; $l'(e) = 0$
  - $(u, r, a)(u, R_i, a)$ if $u = y_i$; $l'(e) = 0$

(3) Let $s' = (s, N, NE)$ and run Dijkstra's algorithm on G' starting from s' using $l'(e)$ as edge weights. Return the minimum of the lengths of the shortest paths from s' to (t, r, E) where $r \in \{N, R_1, R_2, ..., R_k\}$.

The algorithm will return ∞ if there is no valid path due to the implementation of Dijkstra's algorithm. The size of the new graph G' is $|V'| = 2(k+1)n$ and $|E'| \leq (k+1)m + n$. We run Dijkstra $k+1$ times on the original graph and once on G', so the overall time complexity is $O(km + kn \log n)$ where $k = |Y|$.

■

CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)
Homework 9 Problem 2

Solution: (a) We begin by computing the meta-graph G^{SCC} of G. The weight of each node $w'(u)$ in G^{SCC} is the sum of all $w(v)$ where $v \in u$ because the eggs at each node can only be collected once. We want to use the algorithm to find the longest path in a DAG (which is mentioned in lecture). In order to convert vertex weights to edge weights, we divide each vertex $v \in G^{\text{SCC}}$ into two halves v_{in} and v_{out} with v_{in} connected to the in-edge and v_{out} connected to the out-edge. The weights of these two edges are 0. We then add an edge $v_{in}v_{out}$ with weight $l(e) = w'(v)$. Following is the algorithm.

(1) Compute meta-graph $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ using DFS. Let $w'(u)$ be the weight of vertex u in V^{SCC}. For each $u \in V^{\text{SCC}}$, compute $w'(u) = \sum_{v \in u} w(v)$.

(2) Construct graph $G' = (V', E')$ defined as follows:
- $V' = V^{\text{SCC}} \times \{\mathit{in}, \mathit{out}\}$
- $E' = \{(u, \mathit{out})(v, \mathit{in}) \mid uv \in E^{\text{SCC}}\} \cup \{(u, \mathit{in})(u, \mathit{out}) \mid u \in V^{\text{SCC}}\}$
- For all edges in the form $e = (u, \mathit{in})(u, \mathit{out})$, edge weight $l(e) = w'(u)$. For the other edges, $l(e) = 0$.

(3) We can see G' is a DAG because splitting vertices cannot create cycles. Let $s' = (s, \mathit{in})$. Run the algorithm mentioned in lecture to find the longest path in G' starting from s'. Let $d(v)$ be the length of the longest path between s' and v. Find the largest $d(v)$ among all $v \in V'$. It is equal to the maximum number of eggs that my friends can collect starting from s.

The size of G' is $|V'| = 2|V^{\text{SCC}}| and |E'| = |E^{\text{SCC}}| + |V^{\text{SCC}}|$. Since the algorithm for finding the longest path in a DAG runs in $O(m+n)$ time and computing meta-graph also runs in $O(m+n)$ time, this algorithm runs in $O(m+n)$ time where $m = |E|$ and $n = |V|$.

(b) We begin by topologically sorting G^{SCC}. We observe that if uv is an edge in the topological sort, then the length of the longest path starting from u must be larger than the one starting from v. Therefore, for all vertices reachable from u, we only need to calculate once. Following is the algorithm.

```
MaxEggs(G):
  compute meta-graph G^SCC
  do DFS to get a topological sort of G^SCC
  mark all vertices in G^SCC as unvisited
  maxEggs ← 0
  for each v ∈ G^SCC in topological order
      if v is unvisited
          run the algorithm described in (a) using v as the start vertex
          maxEggs ← max (maxEggs, computed maximum number of eggs starting at v)
          do a whatever-first search on G^SCC, mark all vertices reachable from v as visited
```

We can see that although we potentially run the algorithm in (a) multiple times, the parameter of each run is a smaller subgraph of G^{SCC}, and the sum of the sizes of the subgraphs is exactly equal to the size of G^{SCC}. Since the algorithm in (a) runs in $O(m+n)$ time, the algorithm above runs in $O(m+n) + \sum_i O(m_i + n_i)$ time where $\sum_i m_i = m$ and $\sum_i n_i = n$. Therefore, the overall time complexity is still $O(m+n)$.

(c) For this problem, we use G^{SCC} and vertex weights $w'(u) = \sum_{v \in u} w(v)$ for vertex $u \in G^{\text{SCC}}$. Let s be an arbitrary start node. Let *maxEggs*(v, k) be the maximum possible number of eggs collected at at most k vertices in the path from s to v. Consider the following three cases:

- We have collected eggs at less than k locations. Then *maxEggs*$(v, k) =$ *maxEggs*$(v, k - 1)$.
- We have collected eggs at exactly k locations, and we collect eggs at v. Then *maxEggs*$(v, k) = \max_{uv \in E^{\text{SCC}}} ($*maxEggs*$(u, k - 1) + w'(v))$.
- We have collected eggs at exactly k locations, and we do not collect eggs at v. Then *maxEggs*$(v, k) = \max_{uv \in E^{\text{SCC}}} ($*maxEggs*$(u, k))$.

The base case is when $k = 0$ and we cannot get any eggs. Also, *maxEggs*$(s, k) = w'(s)$ for all $k > 0$ because we can only collect eggs at s. Following are the recurrences.

$$\textit{maxEggs}(v,k) = \begin{cases} 0 & k = 0 \\ w'(s) & v = s, k > 0 \\ \max \begin{cases} \textit{maxEggs}(v, k-1) \\ \max_{uv \in E^{\text{SCC}}} (\textit{maxEggs}(u, k-1) + w'(v)) \\ \max_{uv \in E^{\text{SCC}}} (\textit{maxEggs}(u, k)) \end{cases} & \text{otherwise} \end{cases}$$

The idea is similar to (b): if there is an edge uv in the topological sort, then the maximum number of eggs starting from u must be larger than or equal to the maximum number of eggs starting from v, since one can always ignore the eggs in u and start from v. Therefore we only need to consider cases where we start at a vertex with no in-edge.

```
MaxEggsAtMostKLocations(G, k):
compute meta-graph G^SCC = (V^SCC, E^SCC)
n ← |V^SCC|
integer maxEggs[n][k + 1]
initialize maxEggs[v][0] = 0 for v ∈ V^SCC
for i ← 1 to k
    for v ∈ V^SCC in topological order
        if v has no in-edge
            maxEggs[v][i] ← w'(v)
        else
            maxEggs[v][i] ← maxEggs[v][i − 1]
            for each edge uv in E^SCC
                maxEggs[v][i] ← max(maxEggs[v][i], maxEggs[u][i − 1] + w'(v))
                maxEggs[v][i] ← max(maxEggs[v][i], maxEggs[u][i])
return max_{v∈V^SCC} (maxEggs[v][k])
```

To check if a vertex has any in-edge, we can create an $O(n)$ space array initialized with 0. Then we scan all edges uv and set the v^{th} element to 1 in $O(m)$ time. Then checking for in-edge takes $O(1)$ time for each vertex. Since we scan all vertices and all edges k times with constant-time operations during each scan, the time complexity of this algorithm is $O(k(m + n))$.

■