Algorithm Design and Analysis (Fall 2022) Assignment 2

Xiangyuan Xue (521030910387)

1. Consider the classic Dijkstra algorithm. If the specific shortest path is required, we just maintain an array $prev[v]$ to record the vertex u that explores v. To find out the uniqueness, another array $mark$ is needed. Our algorithm can be divided into three steps:

   (a) Calculate $dist[u]$, $prev[u]$ and $mark[u]$ by Dijkstra algorithm.

   (b) Construct shortest path tree G' based on $prev$ where s is the root.

   (c) Expand $mark$ on G' to find the answers.

   After step 1a, $mark[u]$ reveals, on the premise that all the paths updated are shortest, whether there exists more than one vertex v that $prev[u] = v$. After step 1c, $mark[u]$ simply represents whether there exists more than one shortest path from s to u, namely the answers.

Algorithm 1 Modified Dijkstra

Input: Graph $G = (V, E)$ and start vertex s

Output: Three arrays $dist$, $prev$ and $mark$

$T \leftarrow \{s\}$, $dist[s] \leftarrow 0$
for $(s, u) \in E$ **do**
    $dist[u] \leftarrow w(s, u)$
    $prev[u] \leftarrow s$
while $T \neq E$ **do**
    $u \leftarrow \arg\min_{v \notin T}\{dist[v]\}$
    $T \leftarrow T \cup \{u\}$
    for $(u, v) \in E$ **do**
        if $dist[v] > dist[u] + w(u, v)$ **then**
            $dist[v] \leftarrow dist[u] + w(u, v)$
            $mark[v] \leftarrow 0$, $prev[v] = u$
        else if $dist[v] = dist[u] + w(u, v)$ **then**
            $mark[v] \leftarrow 1$
return $dist$, $prev$, $mark$

Algorithm 2 Shortest Path Uniqueness

Input: Graph $G = (V, E)$ and start vertex s

Output: A Boolean array *mark* as the answers

function EXPAND(u)
    for $(u, v) \in E'$ **do**
        $mark[v] \leftarrow mark[u]\ or\ mark[v]$
        EXPAND(v)
DIJKSTRA(G,s)
Construct shortest path tree $G' = (V', E')$ based on *prev*
EXPAND(s)
return *mark*

To prove its correctness, consider an arbitrary vertex t. By algorithm 1 we have found a shortest path from s to t, denoted by $(s \to v_1 \to \cdots \to v_k \to t)$, which can be constructed with the *prev* array. After step 1b, about the *mark* array, we have following claims:

- **Equivalence**: There exists more than one shortest path from s to t **if and only if** there exists a vertex $u \in \{v_1, v_2, \dots, v_k, t\}$ that is marked, namely $mark[u] = 1$.
- **Transitivity**: If there exists a path from u to v and there exists more than one shortest path from s to u, then there exists more than one shortest path from s to v.

Above properties promises that after step 1c, for any vertex $t \in E$, there exists more than one shortest path from s to t if and only if $mark[t] = 1$.

Consider its running time. Step 1a is essentially same with Dijkstra algorithm. It is $O(|E| + |V| \log |V|)$ with Fibonacci heap optimization. Step 1b and 1c are both linear procedures on the shortest path tree which has no more than $|V|$ vertices and edges. Thus, they are both $O(|V|)$. The total time complexity of this algorithm is $O(|E| + |V| \log |V|)$.

2. In this problem, a critical task is to find out whether a vertex v is reachable from another vertex u. Actually, vertices in the same SCC are reachable to each other. If we shrink the vertices into SCCs, we only need to solve the same task in a DAG. Therefore, our algorithm can be divided into several steps:

   (a) Use DFS algorithm to find all the SCCs in G.

   (b) Construct a DAG $G' = (E', V')$ where SCCs in G are vertices in G' and edges across SCCs in G are edges in G'.

   (c) DFS G' in topological order and calculate the revenue of every $v' \in V'$.

   (d) Output the revenue of every $v \in V$, which is actually the revenue of the SCC that it belongs to.

   Update recursion and more details are described in the following pseudo codes.

Algorithm 3 Vertex Revenue

Input: Graph $G = (V, E)$ and reward $\{r_v | v \in V\}$

Output: An array *revenue* as the answers

function $\textsc{Update}(u)$
  $visited[u] \leftarrow 1$
  for $(u, v) \in E'$ **do**
    if $visited[v] = 0$ **then**
      $\textsc{Update}(v)$
      $revenue'[u] \leftarrow \max(revenue'[u], revenue'[v])$
Find all the SCCs $S_1, S_2, \ldots, S_k \subset G$ by DFS algorithm
Shrink vertices into a DAG $G' = (E', V')$
for $v \in V'$ **do**
  $revenue'[v] \leftarrow \max_{u \in S_v}\{r_u\}$
Sort V' in topological order
for $v \in V'$ **do**
  if $visited[v] = 0$ **then**
    $\textsc{Update}(v)$
for $v \in V$ **do**
  $revenue[v] \leftarrow revenue'[S_v]$
return *revenue*

Note that this algorithm has following properties:

- For any vertex u and v, v is reachable from u in G **if and only if** either u and v are in the same SCC S or S_v is reachable from S_u in G'.
- The revenue of every vertex u is updated after every vertex v that is reachable from u is updated, namely by topological order.

These properties promise that every vertex's revenue is correctly calculated.

Consider its running time. Step 2a can be done in $O(|V| + |E|)$ time by DFS algorithm. The DAG constructed contains no more than $|V|$ vertices and $|E|$ edges, so step 2b is also $O(|V| + |E|)$. Topological sort in step 2c requires another $O(|V| + |E|)$. Note that the update procedure visits each vertex and edge only once, so it is also $O(|V| + |E|)$. Output procedure in 2d is obviously $O(|V|)$. Therefore, the total time complexity is linearly $O(|V| + |E|)$.

3. This problem is essentially the same as 2. Similarly, we describe the steps as follow:

   (a) Use DFS algorithm to find all the SCCs in G.

   (b) Construct a DAG $G' = (E', V')$ where SCCs in G are vertices in G' and edges across SCCs in G are edges in G'.

   (c) Traverse G' from the SCC of s by DFS (topological order first) and determine the existence of the path.

Note that if a path from s to t containing every vertex $v \in V$ exists, following equivalent conditions should be satisfied:

- The DAG G' is connected.
- The destination SCC S_t is reachable from the source SCC S_s.
- The path from S_s to S_t contains all the SCCs.

We can also verify these conditions by DFS, which is shown as follow.

Algorithm 4 Traverse Path

Input: Graph $G = (V, E)$, source vertex s and destination vertex t

Output: A Boolean value for whether a path exists

function TRAVERSE(u, d)
    $visited[u] \leftarrow 1$, $depth[u] \leftarrow d + 1$
    for $(u, v) \in E'$ **do**
        if $visited[v] = 0$ **then**
            TRAVERSE(v, $d + 1$)

Find all the SCCs $S_1, S_2, \ldots, S_k \subset G$ by DFS algorithm
Shrink vertices into a DAG $G' = (E', V')$
Sort V' in topological order
TRAVERSE(S_s, 1)
return $depth[S_t] = |V'|$

Just DFS from S_s and record the depth of every vertex $v \in V'$. Note that the vertex with smaller topological order is preferentially choosed. We can state that $depth[S_t] = |V'|$ **if and only if** the DAG G' forms a chain which begins with S_s and ends with S_t, so the equivalent conditions are all satisfied. This promises the correctness of this algorithm.

Consider its running time. Step 3a and 3b can be done in $O(|V| + |E|)$ time, which has been discussed in 2. The traverse procedure visits each vertex and edge in G' only once, so it is also $O(|V| + |E|)$. Therefore, the total time complexity is $O(|V| + |E|)$.

4. (a) **Lemma** In a fully reachable graph $G = (V, E)$, let $M(G) = V - H(G) - T(G)$ denote the set of intermediate vertex. For any intermediate vertex $v \in M(G)$, there exists a head vertex $h \in H(G)$ and a vertex $t \in T(G)$ such that v is reachable from h and t is reachable from v.

Proof Since v is an intermediate vertex, it must have incoming edges. Choose any of them and trace back to the previous vertex recursively. Since G is acyclic and $|V|$ is finite, we will finally reach a head vertex $h \in H(G)$. Similarly, a tail vertex $t \in T(G)$ can be reached. They satisfy the properties described above.

Proposition To make a fully reachable graph $G = (V, E)$ strongly connected, the minimum number of edges to be added is $k = \max(|H(G)|, |T(G)|)$.

Proof We will prove this proposition from two aspects.

- Adding exactly k edges can make G strongly connected.

Suppose $H(G) = \{h_0, h_1, \dots, h_{n-1}\}$, $T(G) = \{t_0, t_1, \dots, t_{m-1}\}$. Added edges can be constructed as $e_i = (t_p, h_q)$, where $p = i \bmod n$ and $q = i \bmod m$.
Since every tail vertex $t \in T(G)$ can be reached by every head vertex $h \in H(G)$ and every head vertex $h \in H(G)$ can be reached by some tail vertex $t_i \in T(G)$, the induced subgraph G' by $H(G)$ and $T(G)$ is strongly connected.
By lemma, for every intermediate vertex $v \in M(G)$, G' can be reached from v and v can be reached from G', so the strongly connected subgraph G' can be expanded on $M(G)$. Since $V = H(G) \cup T(G) \cup M(G)$ contains all the vertices, the whole graph G is strongly connected.

- Adding less than k edges, G is not strongly connected.
  Without the loss of generality, we can suppose $n \geq m$. By pigeonhole principle, there exists at least one head vertex $h \in H(G)$ which is not covered by added edges. Since it has no incoming edges, it is not reachable from any other vertex $v \in V \backslash \{t\}$. Therefore, G is not strongly connected.

Therefore, the minimum number of edges to be added is $k = \max(|H(G)|, |T(G)|)$.

(b) **Proof** Since G is not fully reachable, there exists $h \in H(G)$ and $t \in T(G)$ such that t is not reachable from h. Let the new edge $e = (t, h)$ and we have following statement:

- e is an incoming edge for h and an outgoing edge for t, which makes h no longer a head vertex and t no longer a tail vertex. Since none of other vertices is affected, for $G' = (V, E \cup \{e\})$, we have $|H(G')| = |H(G)| - 1$ and $|T(G')| = |T(G)| - 1$.
- Suppose e brings a cycle for G'. Since the cycle contains $e = (t, h)$, there exists a path from h to t in original graph G, which is contradictory to the premise. So it is promised that G' has no cycle and is still a DAG.

Therefore, such new edge $e = (t, h)$ can always be found.

(c) **Theorem** To make $G = (V, E)$ strongly connected, the minimum number of edges to be added is $k = \max(|H(G)|, |T(G)|)$.

Proof According to the conclusion in 4b, we keep adding edges until G' is fully reachable. Suppose d edges are added in this procedure. Then we have $|H(G')| = |H(G)| - d$ and $|T(G')| = |T(G)| - d$. According to the conclusion in 4a, to make G' strongly connected, at least $\max(|H(G')|, |T(G')|)$ edges need to be added. So the exact number of edges to be added into G is:

$$k = d + \max(|H(G)| - d, |T(G)| - d) = \max(|H(G)|, |T(G)|)$$

The proof of the proposition in 4a also explains that adding $k' < k$ edges can not make G strongly connected. Therefore, the minimum number of edges to be added is exactly $k = \max(|H(G)|, |T(G)|)$.

Algorithm 5 Minimum Edge Adding for DAG

Input: A DAG $G = (V, E)$

Output: The minimum number of edges to be added

$H(G) \leftarrow$ the set of head vertex in G

$T(G) \leftarrow$ the set of tail vertex in G

return $\max(|H(G)|, |T(G)|)$

To find $H(G)$ and $T(G)$, we only need to traverse G and calculate every vertex's in-degree and out-degree. Those vertices whose in-degree is 0 forms $H(G)$, and those vertices whose out-degree is 0 forms $T(G)$. This is obviously a linear procedure, so the total time complexity is $O(|V| + |E|)$.

(d) Based on previous conclusions, we describe the steps as follow:

i. Use DFS algorithm to find all the SCCs in G.

ii. Construct a DAG $G' = (E', V')$ where SCCs in G are vertices in G' and edges across SCCs in G are edges in G'.

iii. Traverse G' to find $H(G'), T(G')$ and output the answer.

Algorithm 6 Minimum Edge Adding for Directed Graph

Input: A directed graph $G = (V, E)$

Output: The minimum number of edges to be added

Find all the SCCs $S_1, S_2, \ldots, S_k \subset G$ by DFS algorithm

Shrink vertices into a DAG $G' = (E', V')$

$H(G') \leftarrow$ the set of head vertex in G'

$T(G') \leftarrow$ the set of tail vertex in G'

return $\max(|H(G')|, |T(G')|)$

Since vertices in the same SCC are strongly connected, they can be viewed as the same vertex. So the minimum number of edges to be added for G is exactly equal to that for G', which promises the correctness.

Consider the running time of this algorithm. Step 4(d)i and 4(d)ii can be done in $O(|V| + |E|)$ time, which has been discussed in 2. According to 4c and the fact that $|V'| < |V|, |E'| < |E|$, step 4(d)iii is $O(|V'| + |E'|) = O(|V| + |E|)$. Therefore, the total time complexity is $O(|V| + |E|)$.

5. (a) It takes me about 6 hours to finish this assignment.

(b) I prefer a 3/5 score for its difficulty.

(c) I have no collaborators, but appreciate some instruction from Yuhao Zhang.