CS5050 Advanced Algorithms

Spring Semester, 2018

Homework Solution 6

Haitao Wang

1. (a) Using the vertex s as the source vertex, we run the breadth-first-search (BFS) algorithm, after which we check whether every vertex v is visited. If yes, then it is true that there exists a path from s to every vertex of G. Otherwise, it is not true.

   Since BFS runs in $O(m+n)$ time, the total time of the algorithm is $O(m+n)$.

   (b) Let G' to be another graph that is the same as G with the following difference: a directed edge (u,v) is in G' if and only if the reversed edge (v,u) is in G.

   Observe that there is a path from s to v' for all vertices v' of G' if and only if there is a path from v to s, for all vertices v of G. Hence, if we have the adjacency lists of the graph G', then we can simply run the BFS with s as the source vertex and use the same algorithm as above (a) to determine whether there is a path from s to every vertex of G'. In order to obtain the adjacency lists of G', we can use the following algorithm.

   For each vertex u of G, we visit its adjacency list for the graph G, denoted by $Adj(G,u)$. For each vertex v in $Adj(G,u)$, we add u to the adjacency list $Adj(G',v)$ of the vertex v for the graph G'. In this way, after we visit vertices of all adjacency lists of G, the adjacency lists of the graph G' are constructed. The time complexity of the algorithm is $O(n+m)$ because we visit every node of the adjacency lists of G exactly once.

   Hence, the overall time for solving this problem is $O(m+n)$.

   (c) We first prove one direction of the statement: *If G is strongly connected, then there is a path in G from s to every vertex and there is a path from every vertex to s.*

   Consider any vertex v of G. Since G is strongly connected, there must be a path from s to v and a path from v to s. This proves that there is a path in G from s to v and there is a path from v to s for all vertices v of G.

   We then prove the other direction of the statement: *If there is a path in G from s to every vertex and there is a path from every vertex to s, then G is strongly connected.*

   Consider any two vertices u and v. To prove that G is strongly connected, we need to show that G has a path from u to v and a path from v to u. Indeed, since there is a path in G from every vertex to s, there is a path $\pi(u,s)$ from u to s. Since there is a path in G from s to every vertex, there is a path $\pi(s,v)$ from s to v. Hence, the concatenation of the two paths $\pi(u,s)$ and $\pi(s,v)$ is a path from u to v. In the same way, we can show that G has a path from v to u as well.

   This proves the statement.

2. We modify the BFS algorithm. We start the algorithm from s. For each vertex v, during the algorithm we will maintain a value $v.count$, which is the number of different shortest

paths from s to v that have been found so far. After the algorithm, $v.count$ is the number of shortest paths from s to v in the graph G for each vertex v of G, and $t.count$ is the value we are looking for. To compute $v.count$, we modify the BFS algorithm in the following way.

The rationale of the algorithm is based on the following property of the BFS algorithm: For any integer $k \geq 0$, the BFS algorithm visits the vertices whose shortest path lengths from s are $k+1$ after it visists the vertices whose shortest path lengths from s are k.

Our algorithm starts from s. Initially we set $s.count = 1$, meaning that there is a single shortest path from s to itself, and for each other vertex v, we set $v.count = 0$. Recall that BFS algorithm will compute a value $v.d$ for each vertex v, which is the length of the shortest path from s to v. Also, in BFS algorithm we have a for-loop exploring the adjacency list of some vertex u, and for each $v \in adj[u]$, we check whether v has been visited (i.e., if $v.color$ is *blue* or *red*, then v has already been visited and we do nothing). We assume the value $u.count$ has been correctly computed. This is indeed true initially when $u = s$. For each $v \in adj[u]$, if $v.color = white$, then v has not been visited before and we set $v.d = u.d + 1$. Note that at this moment we have found $u.count$ shortest paths from s to v and these paths all contain the edge (u, v). Hence, we set $v.count = u.count$. If $v.color$ is not *white*, then v was visited and $v.d$ has already been computed correctly. However, if $v.d = u.d + 1$, then we have found another $u.count$ new shortest paths from s to v and these paths all contain the edge (u, v), and thus we need to increase $v.count$ by $u.count$.

The psuedocode is given in Algorithm 1. Comparing with the original BFS algorithm, we only add a constant time processing in each for iteration, and therefore, the running time of the algorithm is still $O(m + n)$ time.

3. Since the graph G is a DAG, we first do a topological sort on G. For each vertex v, we will compute a value $v.count$, which is the number of different paths from s to v and finally we will simply report $t.count$. Initially we set $s.count = 1$ and set $v.count = 0$ for every other vertex v. Let L be the list of vertices that are topologically sorted. We consider every vertex u in L following their order in L. For each outgoing edge (u, v) of u (i.e., v is in the adjacency list of u), we have found $u.count$ new paths from s to v and these paths all contain the edge (u, v), and thus we increase $v.count$ by $u.count$. The algorithm is done once every vertex of L has been considered.

   The rationale of the algorithm is based on the following observation: For each vertex u, u does not have any path to any vertex v that is in front of u in the sorted list L, and this is due to the topological sort.

   The pseducode is given in Algorithm 2. For the running time, the topological sort takes $O(m + n)$ time. Afterwards, since each edge and each vertex of G have been processed only once, the total running time of the algorithm is $O(m + n)$.

   Remark: For each vertex v in L in front of s, since there is no path from s to v (due to the topological-sort), the value $v.count$ must be zero after the algorithm is finished.

4. We can modify the dynamic programming algorithm studied in class for computing shortest paths in DAGs. Now we define the subproblem as follows. For each $0 \leq l \leq k$ and each vertex v, define $d(v, l)$ as the length of the l-link shortest path from s to v. Therefore, our

Algorithm 1: Modified-BFS(G, s, t)

```
Input: A graph $G = (V, E)$, and two vertices $s$ and $t$.
Output: The number of shortest paths from $s$ to $t$.
for each vertex $u \in V$ do
    $u.color = white$, $u.d = +\infty$, $u.count = 0$;
end
$s.color = blue$, $s.d = 0$, $s.count = 1$;
initialize an empty queue $Q = \emptyset$;
$enqueue(Q, s)$;
while $Q \neq \emptyset$ do
    $u = dequeue(Q)$;
    for each vertex $v \in adj(u)$ do
        if $v.color = white$ then
            $v.color = blue$, $v.d = u.d + 1$, $v.count = u.count$, $enqueue(Q, v)$;
        else
            if $v.d = u.d + 1$ then
                $v.count = v.count + u.count$;
            end
        end
    end
    $u.color = red$;
end
return $t.count$;
```

goal is to compute $d(t, k)$. The dependency relation is the following:

$$d(v, l) = \min_{(u,v)\in G} \{d(u, l-1) + w(u, v)\}.$$

In words, there is an incoming edge (u, v) of v such that an l-link shortest path from s to v is the concatenation of an $(l-1)$-link shortest path from s to u and the edge (u, v).

For the base case, we have $d(s, l) = 0$ for $l = 0, 1, 2, \ldots, k$. Again, we need to compute the topological order as the first step. Algorithm 3 gives the pseudocode. Each iteration of the outmost "for" loop runs in $O(m + n)$ time. Therefore, the total time of the algorithm is $O(k(m + n))$.

Algorithm 2: Paths(G, s, t)

```
Input: A DAG $G = (V, E)$, and two vertices $s$ and $t$.
Output: The number of paths from $s$ to $t$.
1  Topologically sort the vertices of $G$ and let $L$ be the sorted list;
2  for each vertex $u \in L$ do
3      $u.count = 0$;
4  end
5  $s.count = 1$;
6  for each vertex $u \in L$ in order do
7      for each vertex $v \in adj(u)$ do
8          $v.count = v.count + u.count$;
9      end
10 end
11 return $t.count$;
```

Algorithm 3: k-link-shortest-path(G, s, t)

```
Input: A DAG $G = (V, E)$, and two vertices $s$ and $t$.
Output: The length of the $k$-link shortest path from $s$ to $t$.
1  compute the topological order of all vertices of $G$;
2  for $l = 1$ to $k$ do
3      for each vertex $v \in V$ do
4          $d(v, l) = +\infty$;
5      end
6  end
7  for $l = 0$ to $k$ do
8      $d(s, l) = 0$;
9  end
10 for $l = 1$ to $k$ do
11     for each vertex $u$ following the topological order do
12         for each vertex $v \in Adj[u]$ do
13             if $d(v, l) > d(u, l - 1) + w(u, v)$ then
14                 $d(v, l) = d(u, l - 1) + w(u, v)$;
15             end
16         end
17     end
18 end
19 return $d(t, k)$;
```