CSOR W4231: Analysis of Algorithms (sec.001) - Problem Set #4

Jiawen Huang (jh4179). Collaborators: A. Turing. - jh4179@columbia.edu

March 29, 2020

Problem 1

We can use BFS for this problem because BFS gives us connectivity condition. We can first delete edge e, which we denote by e(u,v) and then use BFS to check if v is reachable for u without edge e in the graph.
Pseudocode:

```
Cycle(G, e)
  delete edge e(u,v) in G
  BFS(u)
  if v is in the tree then
    return yes
  else
    return no
  end if
```

Correctness:
If there is a cycle which contains edge e(u,v), from u we can at least have two way to get to v. One is directly from e, and the other one is without e, so the algorithm is to check whether the second way exists or not.
Running Time:
The algorithm only needs to do a BFS which takes $O(m+n)$, so the running complexity is $O(m+n)$.
Space:
The space complexity is $O(n)$ which is the same as BFS.

Problem 2

First, we use DFS to divide the graph into different strongly connected components (SCCs). For each SCC, we use BFS to check if it is bipartite. If it is, then it does not have odd-length cycle, otherwise, it has odd-length cycle.

Pseudocode:

OddLengthCycle(G)

```
SCCs=SCC(G)
for SCC in SCCs do
    test bipartiteness
    if SCC is not bipartite then
        return yes
    end if
end for
return no
```

Correctness:

We have proved in the class, if a connected graph G contains an odd-length cycle, then it is not 2-colorable, which means no bipartiteness.

Running Time:

The algorithm needs to take $O(m+n)$ time to find SCCs and do BFS to 2-color different SCCs of graph G which also takes $O(m+n)$ and so the running complexity is $O(m+n)$.

Space:

The space complexity is $O(n)$ to store SCCs and extra variables in BFS and DFS.

Problem 3

a

We can find the cost of a node by find its reachable nodes with minimum price. First, we need to use TopologicalOrder to find the topological order of DAG and for each node its reachable nodes must fall behind it or be itself.
Pseudocode:

```
DAGcost($G, p$)
  Order=TopologicalOrder(G)
  for u in Order(-1) to Order(1) do
    which is in reverse order
    $cost[u] = min\{p_u, p_v\}$ for every $(u, v) \in E$
  end for
```

Correctness:
It's obviously true if when we know the definition of topological order. Topological order of DAG is for every $(u, v) \in E$, u is put ahead of v, so any reachable nodes of u will be put behind u in the topological order of DAG.
Running Time:
The algorithm needs to take $O(m+n)$ time to find topological order and take $O(m+n)$ to fill the values in cost so the running complexity is $O(m + n)$.
Space:
The space complexity is $O(n)$ to store variable Order.

b

For all directed graph, what we need to do is find all the SCCs of G. For each SCC, since the nodes in it are reachable for all the other nodes, so all the cost of nodes in it is at least the minimum price of them. We can see them as a group in this problem.
Pseudocode:

```
DGcost($G, p$)
  SCCs=SCC(G)
  for SCC in SCCs do
    $p_{SCC} = minp_u$ for any $u \in SCC$
  end for
  see each SCC as a node and draw the edges among them based on G to form $G'$
  DAGcost($G'$)
  cost[u]=cost[SCC] for any $u \in SCC$
```

Correctness:

The definition of SCC means for each nodes in SCC it can reach all the others and also all the other nodes can reach it, so obviously they will have the same cost. If we see each SCC as a new node, the problem will be the same as part (a).
Running Time:
The algorithm needs to take O(m+n) time to find SCCs, form G' and run DAGcost. It also needs to take O(n) time to distribute price of each SCC and costs of each node in the last step, so the running complexity is $O(m+n)$.
Space:
The algorithm needs $O(n)$ to store SCCs and p_{SCC} and $O(m+n)$ to G', so the space complexity is $O(m+n)$.

Problem 4

We only to do some minor adjustment to the Dijkstra's algorithm.
Pseudocode:

```
best(G, V, w)
  Initialize(G,s), best
  Q=BuildQueue(V;dist)
  S=empty
  while $S \neq empty$ do
    u=ExtractMin(Q)
    $S = S \cup u$
    for $(u, v) \in E$ do
      Update(u, v)
    end for
  end while
```

```
Update(u, v)
  if $dist[v] > dist[u] + w_{uv}$ then
    $dist[v] = dist[u] + w_{uv}$
    $prev[v] = u$
    $best[v] = best[u] + 1$
  end if
  if $dist[v] = dist[u] + w_{uv}$ then
    if $best[v] > best[u] + 1$ then
      $best[v] = best[u] + 1$
    end if
  end if
```

Correctness:
Correctness of this algorithm is the same as what we proved in Dijkstra's algorithm.
Running Time:
The algorithm needs to take $O(m \log n)$ time, which is the same as Dijkstra's algorithm implemented by priority queue method.
Space:
The space complexity is $O(n)$ to store extra variables.