CSE 431/531: Algorithm Analysis and Design — Fall 2022

Homework 1 Solutions

Instructor: Shi Li — **Deadline: 9/25/2022**

Your Name: ____________________ Your Student ID: ______________

Problems	1	2	3	Total
Max. Score	20	30	30	80
Your Score				

Problem 1. Asymptotic Notations.

(1a) For each pair of functions $f(n)$ and $g(n)$ in the following table, indicate whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ and $f(n) = \Theta(g(n))$ respectively.

$f(n)$	$g(n)$	O	Ω	Θ
$\log_2(n^3)$	$10\log_2(\sqrt{n})$	yes	yes	yes
$5n^2 + n$	$n \log n$	no	yes	no
$10n^2 + n + 10$	n^3	yes	no	no
e^n	2^{2n}	yes	no	no

(1b) Prove $\lceil 10n\sqrt{n}\, \rceil = O(n\sqrt{n})$ using the definition of the O-notation.

For every $n \geq 1$, we have $\lceil 10n\sqrt{n}\, \rceil \leq 10n\sqrt{n}+1 \leq 11n\sqrt{n}$. Therefore, $\lceil 10n\sqrt{n}\, \rceil = O(n\sqrt{n})$.

In the following two problems, we assume every vertex is incident to at least one edge. So we have $n = O(m)$. Then the running time $O(n + m)$ on the slides becomes $O(m)$.

Problem 2: Cycle Detection in (undirected) graphs A cycle in an *undirected* graph $G = (V, E)$ is a sequence of $t \geq 3$ *different* vertices $v_1, v_2, \cdots, v_t$ such that $(v_i, v_{i+1}) \in E$ for every $i = 1, 2, \cdots, t-1$ and $(v_t, v_1) \in E$. Given the linked-list representation of an (undirected) graph $G = (V, E)$, design an $O(m)$-time algorithm to decide if G contains a cycle or not; if it contains a cycle, output one (you only need to output one cycle). To output the cycle, you can just output $v_1, v_2, \cdots, v_t$.

If the correctness of the algorithm is easy to see from your pseudo-code, then there is no need to prove the correctness separately. However, you should briefly mention why the algorithm runs in time $O(m)$.

We use BFS to Find if a cycle exists in an undirected graph. If we find one edge not in the BFS trees, then we found a cycle. The algorithm is called FindCycleUndirected

Algorithm 1 FindCycleUndirected()

1: create arrays *par* and *queue*, mark all vertices in V as “unvisited”
2: **for** every vertex $s \in V$ **do**
3:     **if** s is unvisited **then** FindCycleFrom(s)
4: print(“no cycle exists”)

Algorithm 2 FindCycleFrom(s)

1: $head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$, mark s as “visited”
2: **while** $head \leq tail$ **do**
3:     $v \leftarrow queue[head], head \leftarrow head + 1$
4:     **for** all neighbours u of v **do**
5:         **if** u is “unvisited” **then**
6:             $tail \leftarrow tail + 1, queue[tail] \leftarrow u, par[u] \leftarrow v$
7:             mark u as “visited”
8:         **else if** $u \neq par[v]$ **then** $\triangleright$ Found a Cycle
9:             PrintCycle(u, v) and **exit** the whole algorithm

Algorithm 3 PrintCycle(u, v)

1: create an array *isancestor* with $isancestor[w] = false$ for every $w \in V$
2: $w \leftarrow u$
3: **while** $w \neq s$ **do** $w \leftarrow par[w], isancestor[w] \leftarrow true$
4: $w \leftarrow v$, create an array *path*, $path[1] \leftarrow v, L \leftarrow 1$
5: **while** $isancestor[w] = false$ **do** $w \leftarrow par[w], L \leftarrow L + 1, path[L] \leftarrow w$
6: $x \leftarrow u$
7: **while** $x \neq w$ **do** print(x) $x \leftarrow par[x]$
8: **for** $i \leftarrow L$ downto 1 **do** print($path[i]$)

We elaborate on the procedure PrintCycle(u, v). We need to find the lowest common ancestor w of u and v in the BFS tree. Then the edge (u, v), the path from u to w in the tree, the path from w to v, and the edge (u, v) form a cycle. In Step 1-3, we create an array *isancestor* that indicate if a vertex is a strict ancestor of u or not. In Step 4-5, we start from v and follow the parent array, until we find the first vertex w with $isancestor[w] = true$. This w is the lowest common ancestor of u and v. At the same time, the array *path* will contain the vertices in the path from v to w in the BFS tree. In Step 6-7, we print the vertices in the path from u to w (do not print w). In Step 8, we print the path from w to v; that is, we print the array *path* in the reverse order.

The running time of the algorithm excluding the procedure *PrintCycle* is $O(m)$. The bottleneck comes from Step 4-9 in FindCycleFrom(s). In one iteration of the while loop, the running time of Step 4-9 is $O(d_v)$, where v is the vertex we handle in the iteration, and d_v is its degree in G. Every v is handled at most once during the whole algorithm. So, the overall running time of Step 4-9 is $\sum_{v \in V} O(d_v) = O(m)$.

In PrintCycle(u, v), every step has running time $O(n)$. So, its running time is $O(n)$. Overall the running time is $O(m + n) = O(m)$.

Problem 3: Cycle Detection in directed graphs A cycle in a *directed* graph $G = (V, E)$ is a sequence of $t \geq 2$ *different* vertices $v_1, v_2, \cdots, v_t$ such that $(v_i, v_{i+1}) \in E$ for every $i = 1, 2, \cdots, t-1$ and $(v_t, v_1) \in E$. Given the linked-list representation of a directed graph $G = (V, E)$, design an $O(m)$-time algorithm to decide if G contains a cycle or not; if it contains a cycle, output one (you only need to output one cycle). To output the cycle, you can just output $v_1, v_2, \cdots, v_t$.

If the correctness of the algorithm is easy to see from your pseudo-code, then there is no need to prove the correctness separately. However, you should briefly mention why the algorithm runs in time $O(m)$.

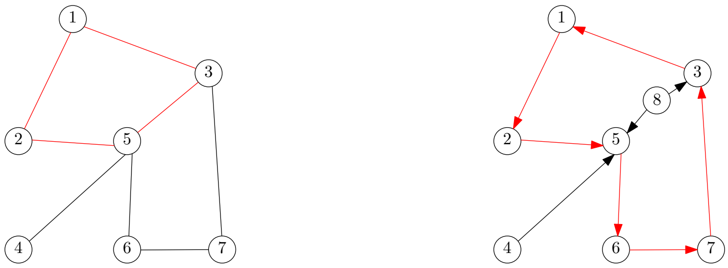

Figure 1: Cycles in undirected and directed graphs are denoted as red edges. (1, 2, 5, 3) is a cycle in the undirected graph. (1, 2, 5, 6, 7, 3) is a cycle in the directed graph. However, (1, 2, 5, 8, 3) is not a cycle in the directed graph.

Remark In a cycle of a directed graph, the directions of the edges have to be consistent. See Figure 1. So, converting a directed graph to a undirected graph and then using algorithm for Problem 2 does not give you a correct algorithm for Problem 3.

We use topological ordering to find if a cycle exists in a directed graph. If we can topologically sort all the n vertices, then there is no directed cycle, otherwise there is a directed cycle.

Algorithm 4 FindCycleDirected()

create an array d with $d[v] = 0$ for every $v \in V$
for every edge $(u, v) \in E$ **do** $d[v] \leftarrow d[v] + 1$
$head \leftarrow 1, tail \leftarrow 0$, **for** every $v \in V$ with $d[v] = 0$ **do**: $tail \leftarrow tail + 1, queue[tail] \leftarrow v$
while $head \leq tail$ **do**
  $v \leftarrow Q[head], head \leftarrow head + 1$
  for every outgoing edge (v, u) of u **do**
    $d[u] \leftarrow d[u] - 1$
    if $d[u] = 0$ **then** $tail \leftarrow tail + 1, queue[tail] \leftarrow u$
if $tail < n$ **then** PrintCycle() **else** print ("no cycle exists")

Algorithm 5 PrintCycle()

1: **for** every $v \in V$ **do if** $d[v] > 0$ **then** break

$\triangleright$ If a vertex has $d[v] = 0$ then v was removed in the topological ordering procedure.

2: create two arrays *trace* and *position*, and let $\mathit{position}[v] \leftarrow 0$ for every $v \in V$.

3: $L \leftarrow 1, \mathit{trace}[1] \leftarrow v, \mathit{where}[v] \leftarrow 1$

4: **while** true **do**

5:     **for** every incoming edge (u, v) of v **do if** $d[u] > 0$ **then** break

6:     **if** $\mathit{position}[u] = 0$ **then**

7:         $L \leftarrow L + 1, \mathit{trace}[L] \leftarrow u, \mathit{position}[u] \leftarrow L, v \leftarrow u$

8:     **else** $\triangleright$ Found a cycle

9:         **for** $i \leftarrow L$ down to $\mathit{position}[u]$ **do** print($\mathit{trace}[i]$)

10:         break

In FindCycleDirected(), we try to topologically sort the vertices of the graph. In the end, if we sort all the n vertices, then the graph does not contain a cycle; otherwise it contains a cycle and we call PrintCycle(). If a vertex v has $d[v] = 0$ after the topological ordering algorithm, then v is sorted and thus removed from the graph.

In PrintCycle(), we find a vertex v with $d[v] > 0$ (Step 1). In the while loop, we find an incoming edge (u, v) of v such that u is not removed (Step 5), let v be u (Step 7) and repeat. We stop and print the cycle if we encountered a same vertex (Step 8-10). In the algorithm, the *trace* array keeps track of the vertices we visited in this procedure, and $\mathit{position}[w]$ indicates the position of w in the *trace* array ($\mathit{position}[w] = 0$ if w is not in the *trace* array.)

If we do not count the running time for PrintCycle(), then then running time of FindCycleDirected() is $O(m)$. Step 6-8 for a vertex v has running time $O(d_v^{\text{out}})$, where $O(d_v^{\text{out}})$ is the out-degree of v. Every vertex v is handled once in the while loop, so overall the running time of FindCycleDirected() is $\sum_{v \in V} O(d_v^{\text{out}}) = O(m)$.

For PrintCycle(), Step 5 has running time $O(d_v^{\text{in}})$ in one iteration of the while loop, where d_v^{in} is the in-degree of v. Each v is handled only once in the while loop, so the running time of the while loop is $\sum_{v \in V} O(d_v^{\text{in}}) = O(m)$.