CMPSC 465 Data Structures & Algorithms
Fall 2024 Mingfu Shao and Debarati Das

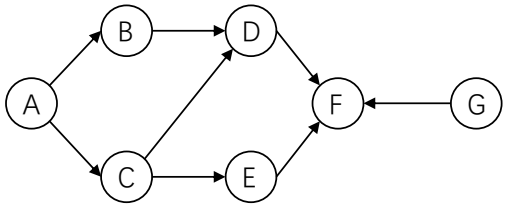
# HW 4

## Due September 27th (Friday), 11:59 pm

Formatting: Each problem should begin on a new page. When submitting in Gradescope, try to assign pages to problems from the rubric as much as you can. Make sure you write all your group members' names. For the full policy on assignments, consult the syllabus.

0. (0 pts.) Acknowledgements. List any resources besides the course material that you consulted in order to solve the assignment problems. If you did not consult anything, write "I did not consult any non-class materials." The assignment will receive a 0 if this question is not answered.

1. (13 pts.) Consider the following directed graph:




   a. (2 pts) What are the sources and sinks of the graph?
   b. (4 pts) Perform DFS with timing on the graph. When there's a choice of vertices, pick the alphabetically first one. Provide the pre and post numbers for each vertex, and the postlist.
   c. (4 pts) How many linearizations does this graph have?
   d. (3 pts) Draw the meta-graph of this graph and list the vertices in each connected component.

   Solution:

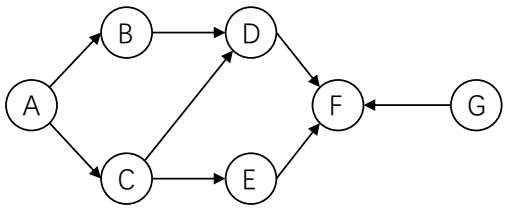
   a. Sources: A, G (no in-edges); Sinks: F (no out-edges)
   b. Here are the pre and post numbers for each vertex, formatted as "vertex: pre-number post-number":
      - A: 1 12
      - B: 2 7
      - D: 3 6
      - F: 4 5
      - C: 8 11
      - E: 9 10
      - G: 13 14

      The postlist consists of vertices in decreasing order of their post numbers, so the postlist is:

- G A C E B D F

c. Instead of listing all possible linearizations, here we try to divide the problem into smaller parts.
   - We found that F is the only sink, meaning no vertices can be placed after F in any linearization. So no matter how ABCDEG are arranged, F must occupy the last position. The number of linearizations can be represented as $n = n_{\text{ACBDEG}} \times 1$, where $n$ is the final answer and $n_{\text{ACBDEG}}$ is the number of linearizations considering only the vertices (and edges among) ACBDEG.
   - We remove F and observe that G has no edges connected to other vertices, meaning G can be placed in any position. If we assume we already have a linearization for ABCDE, then there are 6 valid positions for G. Thus, the total number of linearizations is $n = (n_{\text{ABCDE}} \times 6) \times 1$.
   - We remove FG and find that A is the only source, meaning A must be placed at the beginning of the linearization. We can futher express the answer as $n = ((n_{\text{BCDE}} \times 1) \times 6) \times 1$.
   - Now we consider only BCDE. For BCD, we know B and C must come before D, so there are two valid orderings: BCD and CBD. For E, it must be placed after C. In the first case (BCD), there are 2 possible placements for E, and in the second case (CBD), there are 3 possible placements. Thus, our answer becomes $n = (((2 + 3) \times 1) \times 6) \times 1 = 30$.

d. There are no cycles in the given graph, so each connected component contains only a single vertex. Therefore, the meta-graph of the graph is the graph itself:




2. (10 + 5 (bonus) pts.) A directed graph is said to have the SC-property if for every pair of vertices $(u, v)$ there is atmost one simple path from $u$ to $v$.

   a. (10 pts) Let $G = (V, E)$ be a given directed graph. Given $u \in V$, design an algorithm to decide if there is atmost one simple path from $u$ to every other $v \in V \setminus \{u\}$.

   b. (Bonus question, 2 pts) Analyze the time complexity of your algorithm for part (a). You will get full points if you can show your algorithm runs in $O(|V|)$ time.

   c. (Bonus question, 3 pts) Give an $O(|V|^2)$-time algorithm to check if a given directed graph $G$ has the SC-property.

   Solution:

   a. We can prove that, there exists two paths from $u$ to some other vertex $v$ if and only if in exploring $u$ there exists a moment when examining an edge $(u', v')$, $post[v']$ is already set (i.e., exploring $v'$ has been finished). To detect the existence of such edge, we slightly modify the explore function as the explore1 function, given below. We can use just the visited array to function both visited and the post array: for each vertex $w$, $visited[w] = 0$ means $w$ has not

yet been visited, $visited[w] = 1$ means $w$ has been visited but not yet finished exploring $w$, and $visited[w] = 2$ means $w$ has been finished exploring. Calling explore1($G$, $u$) gives the correct answer. Before calling it, visited array should be initialized to 0 for all vertices.

```
function explore1 (G = (V, E), u ∈ V)
    visited[u] = 1;
    for each v where (u, v) ∈ E
        if (visited[v] = 2): Return False;
        if (visited[v] = 0): b = explore1 (G, v);
        if (b = False): Return False;
    end for;
    visited[u] = 2;
    Return True;
end algorithm;
```

Before proving, we reiterate the concept of "tree edge". In explore when examining edge $(u, v)$ and $visited[v] = 0$ then $(u, v)$ is a tree edge. All tree edges form a tree. (See Figure 1 in Lecture 10; solid edges are tree edges.)

We now prove that, when examining $(u', v')$, if $visited[v'] = 2$ then there must be at least two simple paths from $u$ to $v'$. At that time, $visited[v'] = 2$ means that $v'$ has been visited hence there exists a simple path (path with just tree edges) from $u$ to $v'$ without using edge $(u', v')$. Now we are reaching $v'$ again, using edge $(u', v')$; the path with tree edges from $u$ to $u'$ followed by edge $(u', v')$ is another simple path from $u$ to $v'$.

We prove the other direction by contradiction. Suppose conversely that there exists at least two simple paths from $u$ to $v$ but above moment of checking $(u', v')$ with $visited[v'] = 2$ does not exist. We know that there exists a path $p_1$ from $u$ to $v$ with just tree edges. Now there exists another simple path $p_2$ from $u$ to $v$. Hence $p_2$ must contain some non-tree edge. Let $(u', v')$ be the first non-tree edge on $p_2$. According to the assumption, at this time of examining $(u', v')$, $visisted[v'] \neq 2$. The case that $visited[v'] = 0$ is not possible as otherwise $(u', v')$ becomes a tree edge. Hence, $visisted[v'] = 1$. This implies that exploring $u'$ is within exploring $v'$. This further implies that there exists a path from $v'$ to $u'$ with just tree edges. Since the portion of $p_2$ before $u'$ are all tree edges as we assume $(u', v')$ is the first non-tree edge, $v'$ must be on this portion! This implies that $p_2$ is not simple, a contradiction.

b. Above algorithm runs in $O(|V| + |E|)$ time, the same to the explore function. (The requested $O(|V|)$ time is not feasible.)

c. This part can be solved in $O(|V|^2)$ time. The idea is that, in addition to detect edge $(w, v)$ such that $visted[v] = 2$, there is another condition we can use to decide SC-property for when $visited[v] = 1$ and allow us to halt earlier. When exploring vertex $w$, if there are two (or more) edges $(w, v_1)$ and $(w, v_2)$ such that $visited[v_1] = 1$ and $visited[v_2] = 1$ then we can also immediately conclude that $G$ does not have the SC-property. This is because, $visited[v_1] = 1$ implies that $v_1$ has been visited but not yet finished exploring, so the current exploring $w$ must be within exploring $v_1$; similarly, $visited[v_2] = 1$ means that the exploring $w$ must be also within exploring $v_2$. Hence, the 3 time intervals for $w$, $v_1$, and $v_2$ must be nested. So either exploring $v_1$ is within exploring $v_2$, or vice versa. This results in that either $v_1$ can reach $v_2$ or $v_2$ can reach $v_1$: for the first case we have two paths from $w$ to $v_2$, and for the second case we have two paths from $w$ to $v_1$, both proving that $G$ does not have the SC-property.

The pseudo-code for exploring one vertex, named explore2, is given below. We describe the complete algorithm in the detect-sc-property function. In this function, it is important to reinitialize the visited array after each run.

Note that explore2 runs in $O(|V|)$ time. To see this, consider the total number of edges $(u, v)$ examined. They can be partitioned into 3 classes, according to $visited[v]$ being 0, or 1, or 2. Class-0 edges are tree edges, so there are $O(|V|)$ such edges. In each explore2 clearly at most two class-1 edges are examined per vertex, so the total number of class-1 edges are also $O(|V|)$. The total number of class-2 edges are $\Theta(1)$ since the algorithm will immediately terminates when such edge is met. The entire detect-sc-property function therefore runs in $O(|V|^2)$ time, as it calls explore2 $|V|$ times.

```
function explore2 (G = (V, E), u ∈ V)
    visited[u] = 1;
    cnt = 0;
    for each v where (u, v) ∈ E
        if (visited[v] = 1): cnt = cnt + 1;
        if (cnt ≥ 2): Return False;
        if (visited[v] = 2): Return False;
        if (visited[v] = 0): b = explore2 (G, v);
        if (b = False): Return False;
    end for;
    visited[u] = 2;
    Return True;
end algorithm;
function detect-sc-property (G = (V, E))
    for each u ∈ V
        visited[v] = 0, for all v ∈ V;
        if (explore2 (G, u) == False): return False
    end for;
    Return True;
end algorithm;
```

3. (15 pts.) You are given an undirected graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Your task is to count the total number of simple cycles of length $n$ in the graph. A simple cycle of length $n$ is defined as a cycle that contains exactly $n$ vertices and $n$ edges. Each cycle should only be counted once, regardless of its starting vertex or direction.

   Input: an undirected graph $G = (V, E)$, represented as an adjacency list; an integer $n$, which represents the desired length of the cycle.

   Output: return the total number of simple cycles of length $n$ in graph $G$.

   Design an algorithm for above problem and analyze its time complexity. (Hints: consider a DFS-based algorithm; ensure that each cycle is counted only once, despite the undirected nature of the graph; optimize the DFS to avoid redundant searches and prune unnecessary paths that do not lead to valid cycles.)

Solution

The high-level idea is to use DFS to search all simple cycles of length $n$. One of the key points is to avoid counting redundant cycles. For instance, the sequence $1 \to 2 \to 4 \to 3 \to 1$ forms a cycle, which can also be represented as $2 \to 4 \to 3 \to 1 \to 2$, or $4 \to 3 \to 1 \to 2 \to 4$. To ensure that permutations of the same cycle does not get counted, the algorithm only considers cycles where the starting index has the smallest vertex. For above example, only $1 \to 2 \to 4 \to 3 \to 1$ gets counted. Furthermore, due to the undirected nature of the input graph, the same simple cycle will still be counted twice. For example, $1 \to 2 \to 4 \to 3 \to 1$ is the same cycle with $1 \to 3 \to 4 \to 2 \to 1$. Hence, at the end, we will need to divide the counts by 2 to just count unique cycles.

For each vertex $v_i$, we count the number of simple cycles of length $n$ in which $v_i$ is the vertex with the smallest index. We use a function similar to explore to perform such counting/searching. To ensure that only simple cycles are counted, we will need a visited array to keep track of vertices in the partially built cycle. We will also maintain a depth $d$, to keep track of the number of vertices already included in the partial cycle (i.e., number of visited vertices). Specifically, we define recursive function explore-cycles($v_i$, $v_j$, $d$, visited, $n$, $G$) that returns the number of simple cycles of length $n$ in graph $G$ where the smallest vertex is $v_i$, the first $d$ vertices are already determined and stored in "visited" array, and the last vertex of the partial cycle is $v_j$.

function explore-cycles($v_i$, $v_j$, $d$, visited, $n$, $G = (V, E)$)
- count = 0;
- If $d = n$: (## the partial cycle already contains $n$ vertices)
  - If $(v_j, v_i) \in E$: return 1; (## cycle found)
  - Else: return 0;
- End if
- For each edge $(v_j, v_k) \in E$:
  - If $k \leq i$: continue; (## $v_i$ must be the smallest)
  - If visited[$k$] = 1: continue; (## $v_k$ already included in the partial cycle)
  - visited[$k$] = 1; (## include $v_k$ in the partial cycle)
  - count = count + explore-cycles($v_i$, $v_k$, $d + 1$, visited, $n$, $G$); (## recursively counts)
  - visited[$k$] = 0; (## excludes $v_k$ to consider the alternative)
- Return count.

End

The algorithm that counts simple cycles in $G$ using above procedure is given below.

Algorithm DFS-counts-cycles($n$, $G = (V, E)$)
- init cycle-count = 0.
- For each vertex $(v_1, v_2, \cdots, v_n) \in V$:
  - init a visited array of size $|V| = n$, with all 0s;
  - visited[$i$] = 1; (## include $v_i$ in the partial cycle)
  - cycle-count = cycle-count + explore-cycles($v_i$, $v_i$, 1, visited, $n$, $G$).
- End
- Return cycle-count / 2; (## since cycles are counted twice in undirected graphs).

End

Time Complexity: This algorithm enumerates all simple cycles of length $n$ in a given graph, and

hence runs in $O(|V|^n)$ time. (Note: if $n$ is not a constant, this algorithm does not run in polynomial-time since $|V|^n$ is an exponential function. It is the "track-back" step, i.e., resets visited[$k$] to 0, in the explore-cycles function that makes the algorithm exponential. This problem demonstrates that use of DFS to do exhaustive search.)

4. (5 bonus pts.) In Lecture 11 we showed that in order to obtain a special order to run DFS to identify all connected components in a directed graph $G$, we need to build the reverse graph $G_R$ of $G$ and then run DFS-with-timing on $G_R$ to get the resulting postlist (termed $polistlist_R$), which is the special order. You might wonder if running DFS on $G$ to obtain the postlist followed by reversing it would also result in a correct special order. Show that this approach does not work by giving an example. You need to draw a directed graph $G$ (the example), show all connected components of $G$, show the postlist after running DFS-with-timing on $G$, show the visited array after runing DFS following the reverse of the postlist, and verify that it does not give the correct connected components.

   Solution: See the example below. This graph contains two connected components: the first one is $\{A, B, C\}$ and the second one is $\{D\}$. The running of DFS-with-timing on this graph is given in the second figure, where the $[pre, post]$ values are labeled next to the vertices; so the postlist is $(A, C, D, B)$. The reverse of this list is therefore is: $(B, D, C, A)$. Running DFS following this list will give this visited array: $[1, 1, 1, 1]$, since $B$ can reach all vertices. This is obviously incorrect.