CS 180 Discussion 1A/1E

Week 3

Haoxin Zheng

10/20/2023

Comments

- Bullet points are recommended in your HW answers
- Time complexity justifications
- “Edges”: undirected? Directed?
- “Graph”: How it is constructed? Array or linked list?

Graph

- Definitions:
  - $G = (V, E)$. V:nodes(vertices), E:edges
  - $n = |V|$, number of nodes
  - $m = |E|$, number of edges

Graph

- Definitions:
  - *Path*: A path is a sequence of nodes connected by edges

Connected

1, 2, 3, 4

1, 2, 3

1, 2, 3, 4, 1

  - *Connect*: Node u and v are connected <=> ∃ a path between u and v

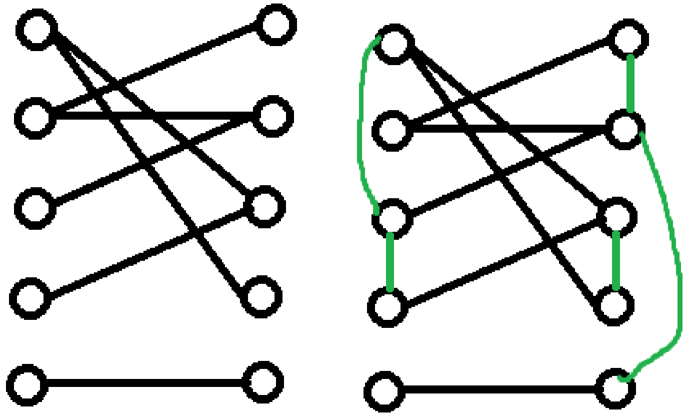
  - *Connected Graph*: All nodes are connected with each other

  - *Cycle:* A sequence of nodes $v_1, v_2\ \ldots v_k\ (k > 2), v_1 = v_k$. No repeat edges, not repeat nodes except $v_1 \& v_k$

Bipartite Graph

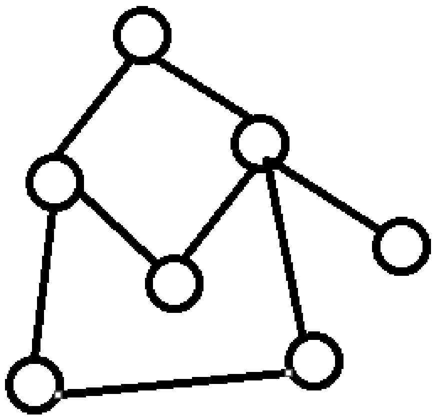
- Definition of Bipartite graph G:
  - G is a Bipartite graph if it is a set of vertices that can be decomposed into two disjoint sets, such that no two vertices within the same set connected by an edge

- Lemma:
  - A graph is bipartite graph <u>*if & only if*</u> there is no odd cycle in the graph
    - Odd cycle: A cycle with odd number of vertices



Bipartite Graph

- Check if the graph is bipartite:
  - Run BFS. Then check whether there exists an edge within the same level.
  - Why:
    - If there is no edge between nodes inside each level, then we can have
      - $V_1 = All\ nodes\ in\ odd\ levels,$
      - $V_2 = All\ nodes\ in\ even\ levels.$
    - If there exists edge between nodes inside each level, then the graph exists at least one odd cycle, cannot be a bipartite graph.



Directed Graph

- Directed Graph: Edge has direction.
  - $(\mathrm{A}, \mathrm{B}) \in E \neq (\mathrm{B}, \mathrm{A}) \in E$
- Strongly Connected (SC) graph:
  - $\forall u, v \in V$, we can find a path to v start from u

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

- How to check if a graph is SC graph?
  - Arbitrarily pick a node S from G.
  - Run DFS/BFS from S on G -> reachable nodes R
  - Run DFS/BFS from S on G(edge direction reversed)-> reachable nodes Q
  - If V==Q==R:
    - We can know $\forall u \in V \rightarrow S$, and $S \rightarrow \forall v \in V$
    - Then we can know $\forall u, v \in V: u \rightarrow v$
    - By definition, the G is a SC graph

Directed Acyclic Graph (DAG) & Topological order

- Directed Acyclic Graph: a directed graph with no cycle.

- Suppose we have a set of tasks labeled {1, 2, . . . , *n*} that need to be performed, and there are dependencies among them. For certain pairs *i* and *j*, that *i* must be performed before *j*.

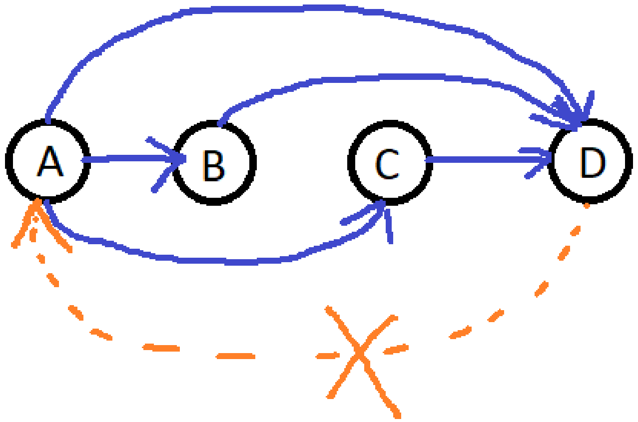
- Similar to directed edges, $v_i \to v_j$

- Given a set of tasks with dependencies, it would be natural to seek a valid order in which the tasks could be performed, so that all dependencies are respected

Topological order

- Specifically, for a directed graph G, we say that a ***topological ordering*** of G is an ordering of its nodes as $v_1, v_2, \dots v_n$ such that for every edge (v_i, v_j) we have $i < j$.

- In other words, all edges point "forward" in the ordering.



- A topological ordering on tasks provides an order in which they can be safely performed. When we come to the task v_j, all the tasks that are required to precede it have already been done.

Topological order

(3.18) *If G has a topological ordering, then G is a DAG.*

(3.20) *If G is a DAG, then G has a topological ordering.*

The inductive proof contains the following algorithm to compute a topological ordering of G.

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

Topological order

(3.18) *If G has a topological ordering, then G is a DAG.*

Proof. Suppose, by way of contradiction, that G has a topological ordering $v_1, v_2, \ldots, v_n$, and also has a cycle C. Let v_i be the lowest-indexed node on C, and let v_j be the node on C just before v_i—thus (v_j, v_i) is an edge. But by our choice of i, we have $j > i$, which contradicts the assumption that $v_1, v_2, \ldots, v_n$ was a topological ordering. ∎

(3.20) *If G is a DAG, then G has a topological ordering.*

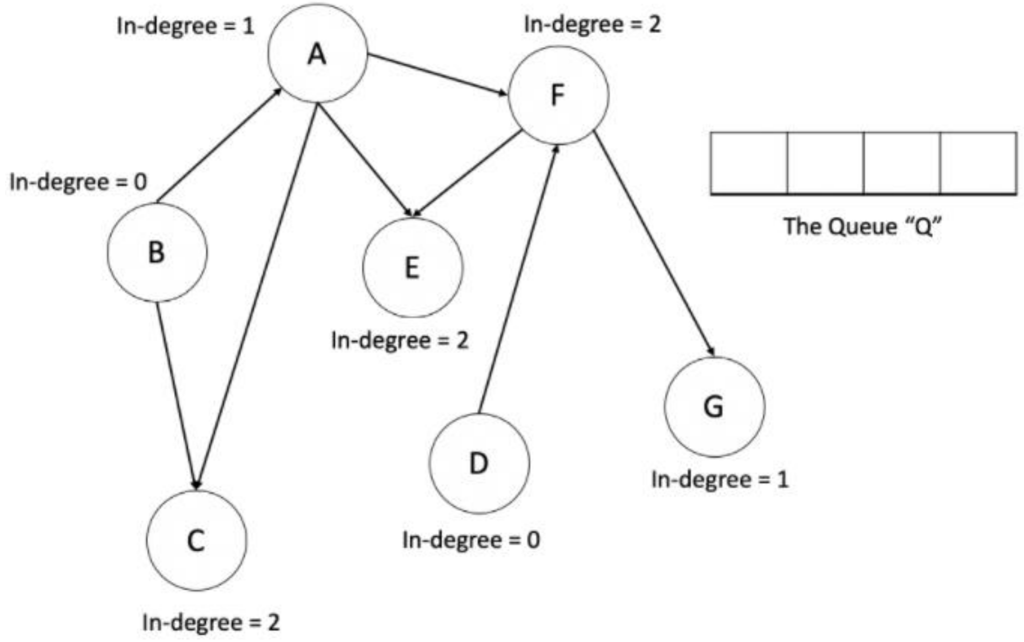
The inductive proof contains the following algorithm to compute a topological ordering of G.

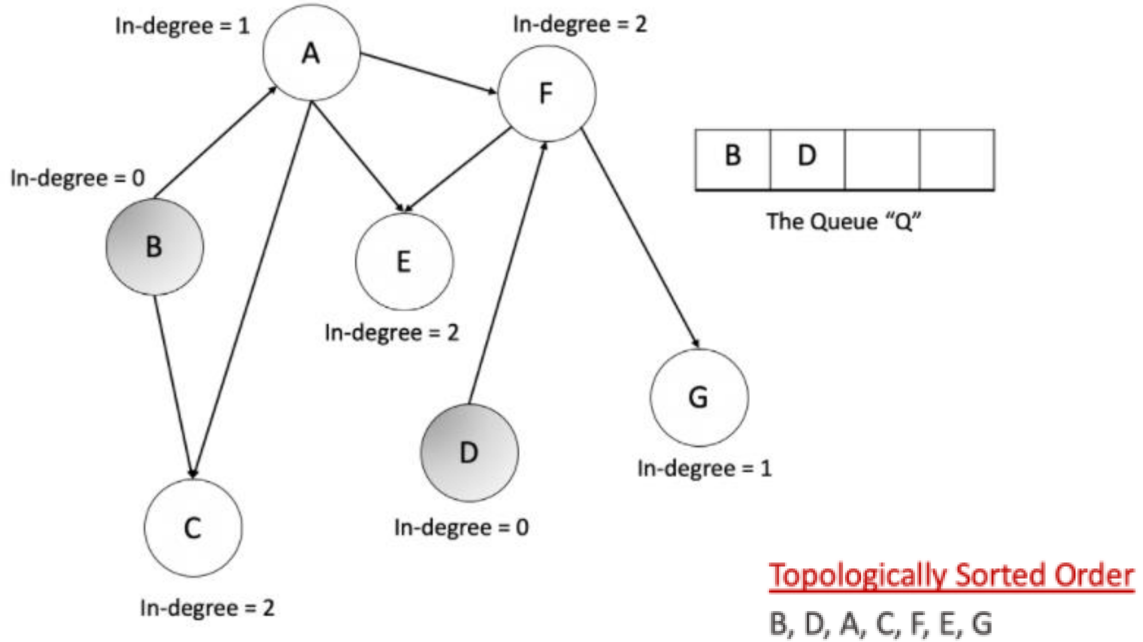
```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

Topological sort

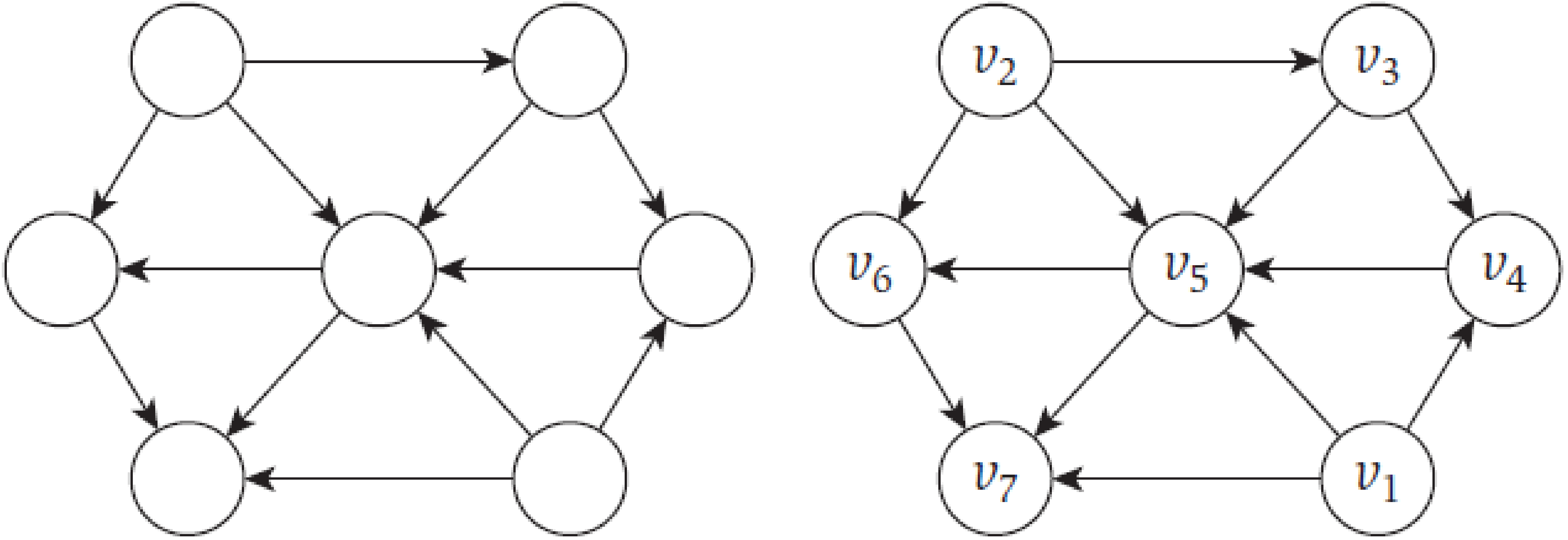
- Precompute indegree[v] $\forall v \in V$
- Q = {v with indegree[v]==0}
- For i=1, 2, … n:
    - u = Q.pop()
    - make u next in T-order
    - $\forall\ v\ such\ that\ (u, v) \in E$:
        - Indegree[v] --
        - If indegree[v] ==0:
            - Q.push(v)

• Time complexity: O(|E|+|V|)



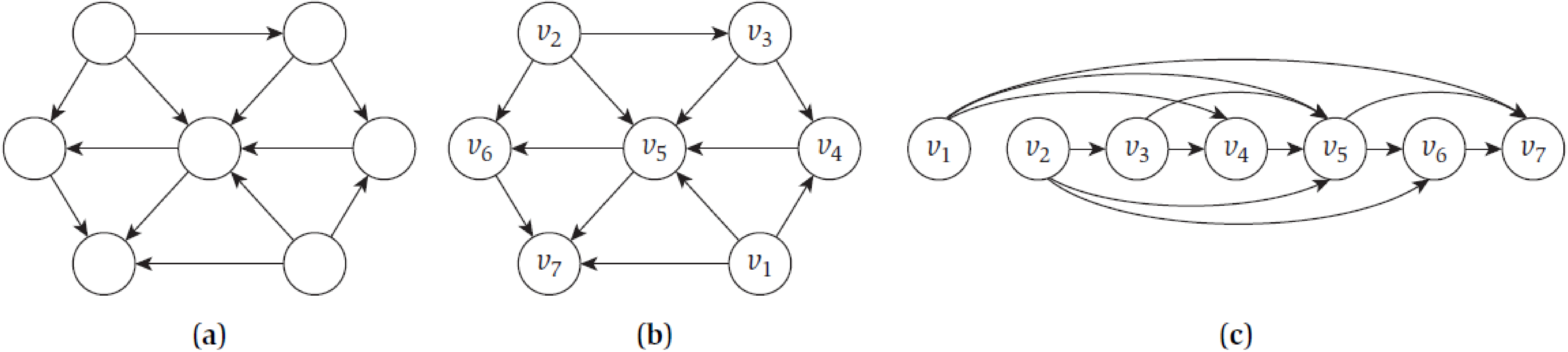


Topological sort



(a) (b)

Topological sort



Topological sort

1. Consider the directed acyclic graph G in Figure 3.10. How many topological orderings does it have?

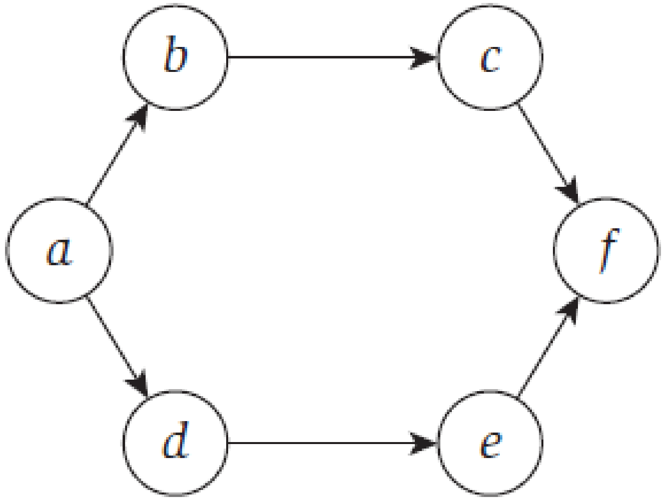


Figure 3.10 How many topological orderings does this graph have?

Exercise

- Given a DAG G. k is the maximum number of edges among all paths. Design an algorithm to Partition the vertices into k+1 groups s.t. For each vertex in the same group, there are no edges

Exercise

- Given a DAG G. k is the maximum number of edges among all paths. Design an algorithm to Partition the vertices into k+1 groups s.t. For each vertex in the same group, there are no edges

- Algorithm:
  - 0) Pre-compute indegree and outdegree of all nodes.
  - 1) Find all nodes with indegree=0 and put them in group 0
  - 2) Remove selected nodes and their outdegree edges.
  - 3) For all remained nodes connected by the nodes removed in the previous step, decrease outdegree by 1 for each edge pointed out by the nodes removed.
  - 4) Back to step 0), with group index $1, 2$ … $k+1$ until all nodes have been removed

- O(|E|+|V|)

210. Course Schedule II

Medium 4971 193 Add to List Share

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [`a_i`, `b_i`]` indicates that you **must** take course b_i first if you want to take course a_i.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have
finished course 0. So the correct course order is [0,1].
```

Example 2:

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have
finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished
course 0.
So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].
```

Example 3:

```
Input: numCourses = 1, prerequisites = []
Output: [0]
```

```python
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        # Create a prerequisite dict. (containing courses (nodes) that need to be taken (visited)
        # before we can visit the key.
        preq = {i:set() for i in range(numCourses)}
        # Create a graph for adjacency and traversing.
        graph = collections.defaultdict(set)
        for i,j in prerequisites:
            # Preqs store requirments as their given.
            preq[i].add(j)
            # Graph stores nodes and neighbors.
            graph[j].add(i)

        q = collections.deque([])
        # We need to find a starting location, aka courses that have no prereqs.
        for k, v in preq.items():
            if len(v) == 0:
                q.append(k)
        # Keep track of which courses have been taken.
        taken = []
        while q:
            course = q.popleft()
            taken.append(course)
            # If we have visited the numCourses we're done.
            if len(taken) == numCourses:
                return taken
            # For neighboring courses.
            for cor in graph[course]:
                # If the course we've just taken was a prereq for the next course, remove it from its prereqs
                preq[cor].remove(course)
                # If we've taken all of the preqs for the new course, we'll visit it.
                if not preq[cor]:
                    q.append(cor)
        # If we didn't hit numCourses in our search we know we can't take all of the courses.
        return []
```

https://leetcode.com/problems/course-schedule-ii/solutions/762346/python-bfs-beats-98-with-detailed-explanation-and-comments/