DSC40B: Theoretical Foundations of Data Science II

Lecture 13: *Depth-First Search (DFS)*

Instructor: Yusu Wang

One more property about BFS

- For BFS algorithm, at any moment of the algorithm:
  - Recall the queue stores all the pending nodes (discovered, but not explored)
  - Recall we are inserting nodes into the queue in non-decreasing order their distance from source

  - It is easy to verify that
    - The shortest path distance from the source are non-decreasing in the queue
    - The shortest path distance for nodes in the queue cannot differ more than 1

Prelude

- Previously, Search strategy :
- How to decide which is the next node to explore?
  - BFS (Breadth-first search):
    - choose the ``oldest'' pending node to explore and expand
    - consequently, it explores as wide as possible before goes any ``deeper'' (in terms of distance to the source)
      - i.e, it visits all nodes at distance k to the source before moving to any node at distance $k+1$ from the source.

- Today: Depth-first search (DFS):
  - Choose the ``newest'' pending node to explore
  - Consequently, it will go as deep (farther away from source) as possible during exploration

DFS algorithm and time complexity

Depth-first Search (DFS)

- DFS(G, s)
  - It will perform depth-first search in G starting from a graph node s called the *source* node.
- Idea:
  - All nodes are initialized as undiscovered, other than the source node, which is initialized as pending (i.e, discovered, and to be processed)
  - At each step:
    - take the newest pending node to explore
    - *explore all undiscovered nodes reachable from this node*
    - then mark this node to be visited
  - Repeat till there is no more pending nodes to explore

- To be able to extract the "newest" pending node
  - we need a standard stack data structure to provide FILO (first-in-last-out)

- In algorithm implementation,
  - we use recursive algorithm to achieve this FILO/stack idea implicitly

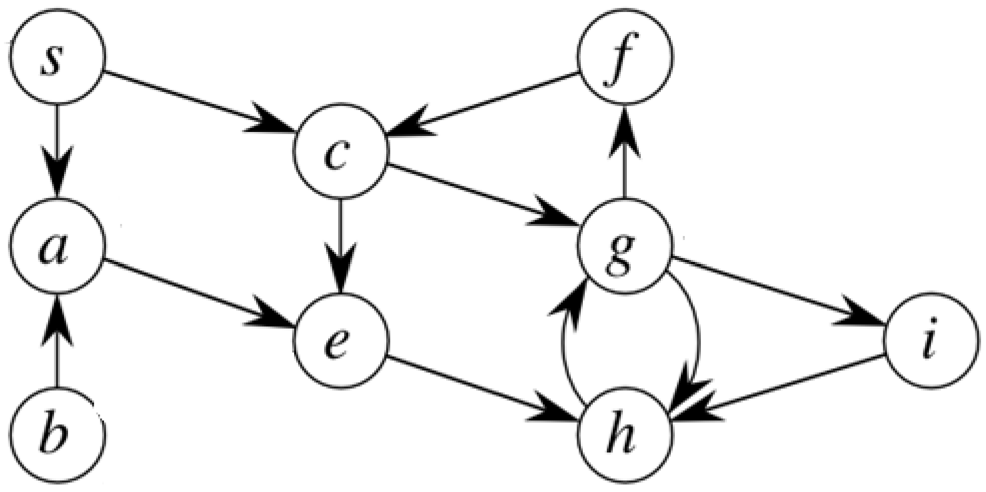
Implementation in Python

```python
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Example

G:



- Call dfs(G, s)

Full DFS

- DFS will visit all nodes reachable from the source node
  - If the input graph is connected, then it will visit all nodes in the same connected component as the source
- To visit all nodes in a graph
  - Needs full-DFS
    - which requires we restart from undiscovered nodes.

```
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)
```

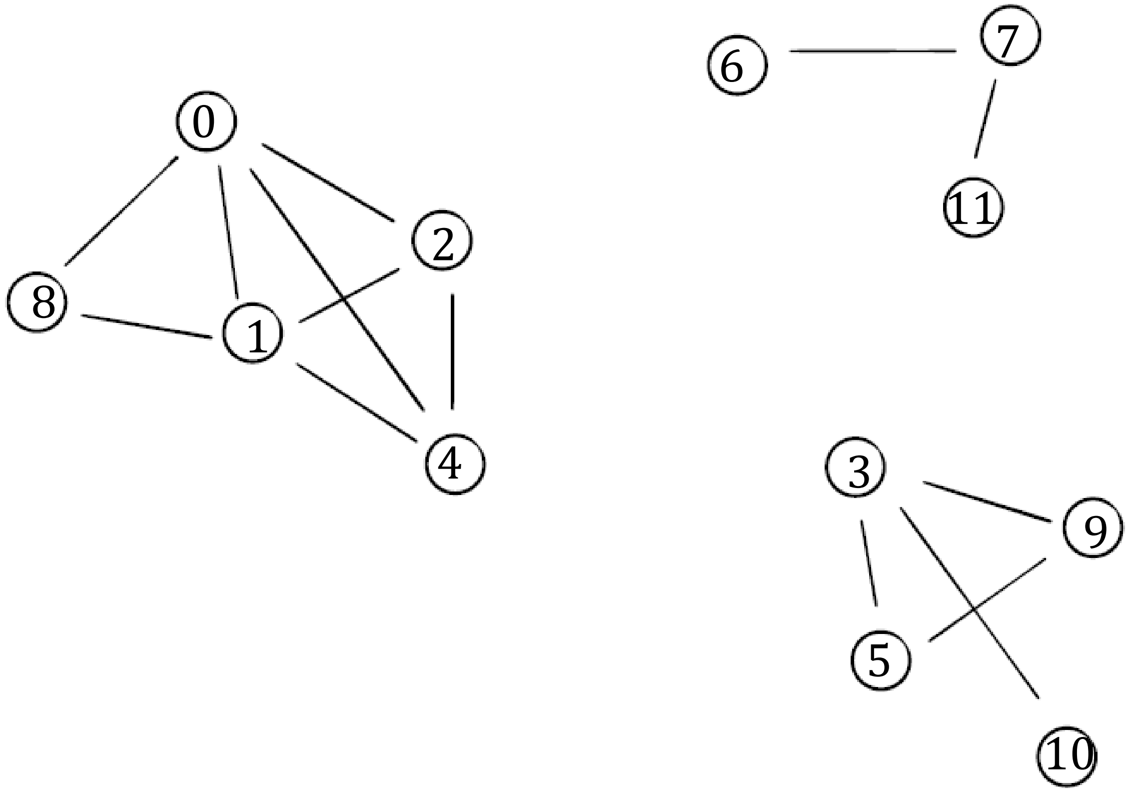
Complete code

```python
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)
```

```python
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Example



Time complexity analysis

- (similar to BFS): for full-DFS
  - Each node will be explored exactly once
    - Each edge will be traversed (visited) twice for undirected graphs
    - Each edge will be traversed (visited) once will for directed graphs
  - Hence total time complexity of full-DFS
    - $\Theta(|V| + |E|)$

Nesting properties in DFS

full_DFS

```python
def full_dfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            dfs(graph, node, status)
```

```python
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

“Parent” (Predecessor) information

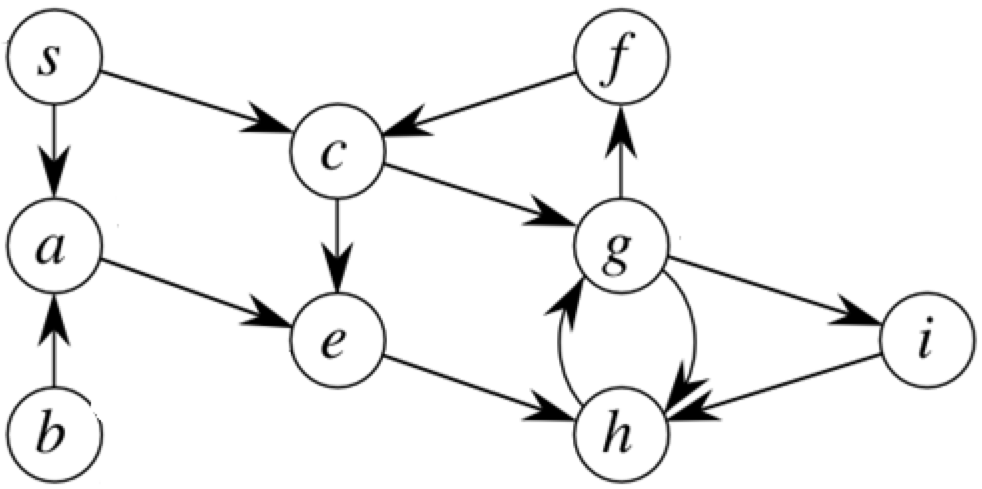
- Similar to BFS, for each node v,
  - its (DFS-)predecessor is the node u where through exploring edge (u, v) the node v was first discovered (status changed to pending).

- Collection of edges of the form (predecessor(v), v) will give a tree
  - called DFS-tree.
  - Predecessor(v) is the parent of v in this DFS-tree.

Observations

- Between marking a node as pending and visited, many other nodes are marked pending or visited.

G:



Start and Finish times

- A node status changed
  - from undiscovered to pending:
    - first time this node is discovered
  - from pending to visited:
    - exploration of this node is finished
- Keep an integer running clock
- For each node:
  - Start time: status changes from undiscovered to pending
  - Finish time: status changes from pending to visited
- Clock increments by 1 whenever some node is marked pending/visiting

full_DFS_times implementation

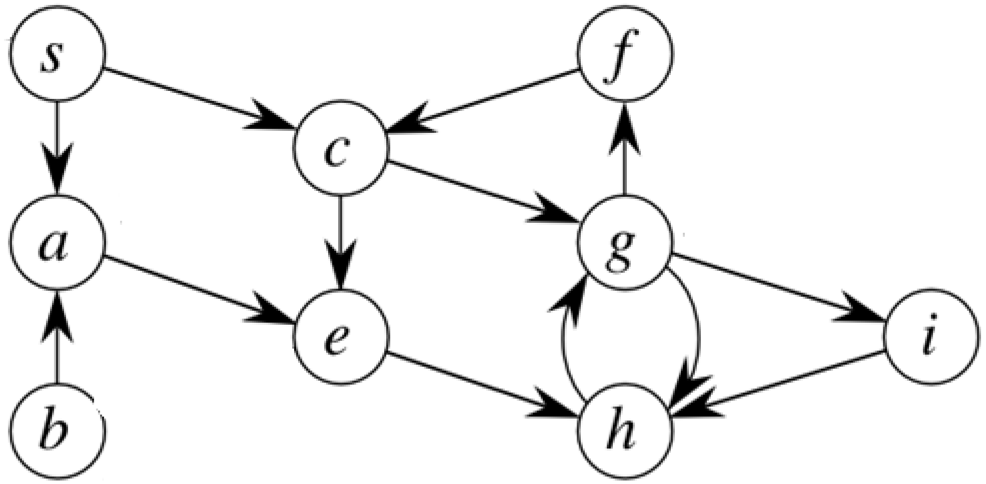
```python
from dataclasses import dataclass
@dataclass
class Times:
    clock: int
    start: dict
    finish: dict
```

```python
def full_dfs_times(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}
    times = Times(clock=0, start={}, finish={})
    for u in graph.nodes:
        if status[u] == 'undiscovered':
            dfs_times(graph, u, status, times)
    return times, predecessor
```

```python
def dfs_times(graph, u, status, predecessor, times):
    times.clock += 1
    times.start[u] = times.clock
    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            predecessor[v] = u
            dfs_times(graph, v, status, times)
    status[u] = 'visited'
    times.clock += 1
    times.finish[u] = times.clock
```

Example

G:



Nesting Property of DFS

- Claim A:
  - Take any two nodes u and v. Assume $\text{start}[u] \le \text{start}[v]$
  - Exactly one of the following two is true:
    - $\text{start}[u] \le \text{start}[v] \le \text{finish}[v] \le \text{finish}[u]$
    - $\text{start}[u] \le \text{finish}[u] \le \text{start}[v] \le \text{finish}[v]$

Nesting Property of DFS

- Take any two nodes u and v where v is reachable from u
  - If while exploring u we reached v, then the exploration of v has to be done first before we finish exploring u
    - That is: start[u] $\leq$ start[v] $\leq$ finish[v] $\leq$ finish[u]

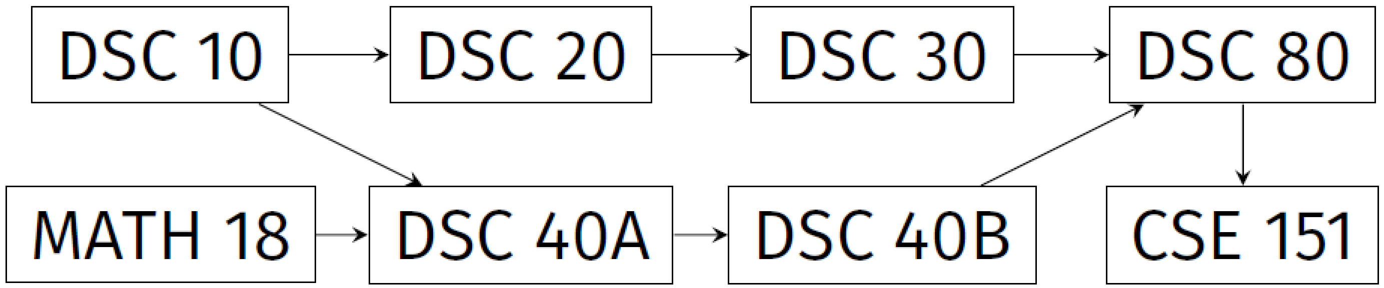
- Claim B: If a graph $G = (V, E)$ does not have cycles, and node v is reachable from u, then finish[v] $\leq$ finish[u]
  - In the case that the graph $G = (V, E)$ is a directed graph, then we mean if G does not have directed cycles.

DAGs and topological sort

Applications of DFS

- Is node v reachable from u?
- Is a undirected graph connected?
- How many connected components are there in a undirected graph?
- Is the input graph a tree?
- Find the shortest path to a source node u?
  - NO !
  - Unlike BFS.

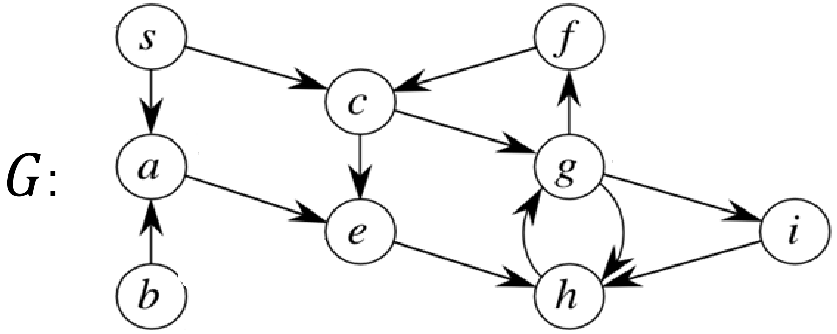
Prerequisite graphs



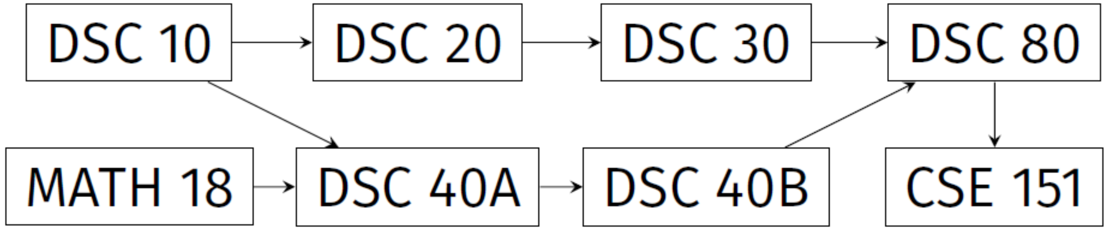
- Note that this graph is a directed graph
  - edge (u, v) means that course u is prerequisite for v
- Goal:
  - Find an ordering so as to take these classes satisfying all prerequisite requirements

Directed Acyclic Graphs (DAGs)

- A directed cycle is a (directed) path from a node to itself.
- A directed acyclic graph (DAG) is a directed graph that does not contain any directed cycles



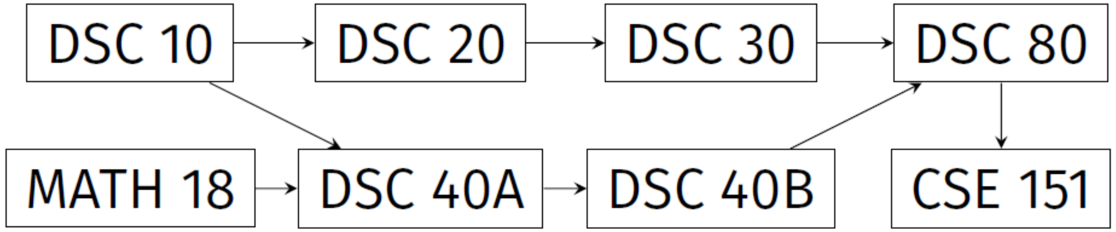
Not a DAG!



A DAG!

Topological Sorts

- Given a DAG $G = (V, E)$, a topological sort of G is an ordering of V such that
  - for any edge $(u, v) \in E$, then u comes before v in this ordering



A topological sort:
DSC10, Math18, DSC20, DSC40A, DSC30, DSC40B, DSC80, DSC151

Another topological sort:
Math18, DSC10, DSC20, DSC30, DSC40A, DSC40B, DSC80, DSC151

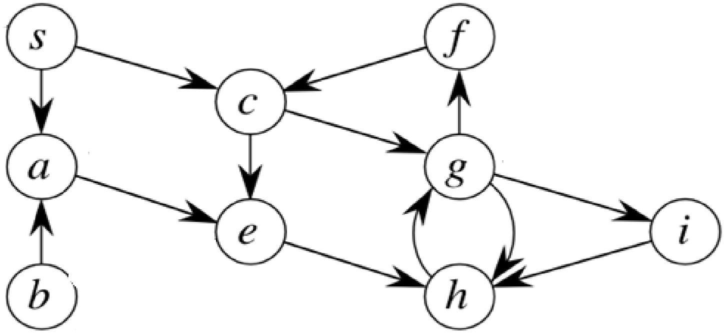
- Topological sorts of the same DAG are not unique.

Why DAG?

- Claim:
  - A directed graph $G = (V, E)$ can admit a topological sort if and only if G is a DAG!

- Why?
  - If there is a cycle, then there is no valid ordering for nodes in that cycle!
  - If it is a DAG, then we will give an algorithm to show we can compute topological sort.

G:



An algorithm to compute topo-sort

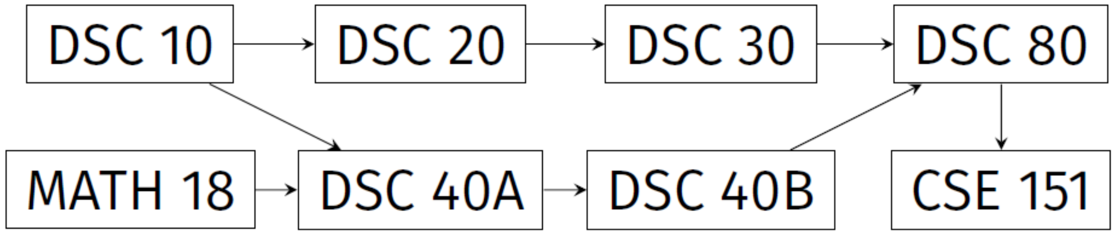
- Recall Claim B: If node v is reachable from u, but u is not-reachable from v
  - then finish[v] $\leq$ finish[u]

- If v is reachable from u, then u should come before v in any topological sort.
- So nodes with later finish-time should come first

- Topo-sort Algorithm:
  - First perform DFS on input graph $G = (V, E)$
  - Output the order in decreasing order of finish-time.

- Time complexity: $\Theta(|V| + |E|)$

Example



Remark:

- There are many other ways to compute a topological sort for a DAG in the same time complexity, without using DFS.

Summary

- DFS:
  - Yet another graph search strategy
  - Similarly to BFS, as a graph search strategy, can help solve many problems, such as checking for connectivity, reachability, finding a path and so on.

  - But has some different properties as BFS
    - BFS: useful for finding shortest path to the source
    - DFS: has nesting properties in terms of start/finish times.
      - An application: Topological sort in DAG

FIN