DSC 40B

Theoretical Foundations II

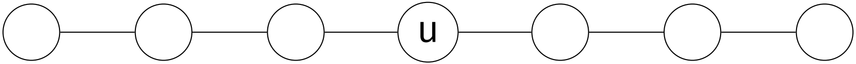
Lecture 13 | Part 1

Depth First Search

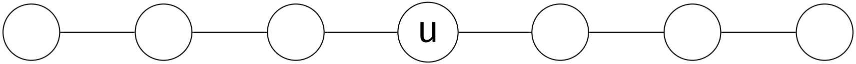
Visiting the Next Node

- Which node do we process next in a search?
- BFS: the **oldest** pending node.
- DFS (today): the **newest** pending node.
  - Naturally recursive.
  - Useful for solving different problems.

Example (BFS)



Example (DFS)

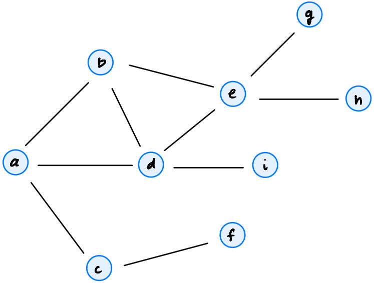


```python
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    # initialize status if it was not passed
    if status is None:
        status = {node: 'undiscovered' for node in graph.nodes}

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

Exercise

Write the nested function calls for a DFS on the graph below.




```python
def dfs(graph, u, status=None):
    """Start a DFS at `u`."""
    ...
    status[u] = 'pending'
    for v in graph.neighbors(u): # explore edge (u, v)
        if status[v] == 'undiscovered':
            dfs(graph, v, status)
    status[u] = 'visited'
```

Differences

- In **BFS**, we "finish" a node u before moving on to the next.
- In **DFS**, we go to many other nodes, but "come back" to u.

Main Idea

We'll see that the nested structure of the **recursive function calls** gives us useful new information about the graph's structure.

Full DFS

- `dfs(u)` will visit all nodes **reachable** from u.
  - But not all nodes may be reachable from u!

- To visit **all** nodes in graph, need **full DFS**.

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


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'
```

Time Complexity

- In a full DFS:
  - `dfs` called on each node exactly once.
  - Like BFS, each edge is explored exactly:
    - once if directed
    - twice if undirected

- Time: $\Theta(V + E)$, **just like BFS**.

DSC 40B

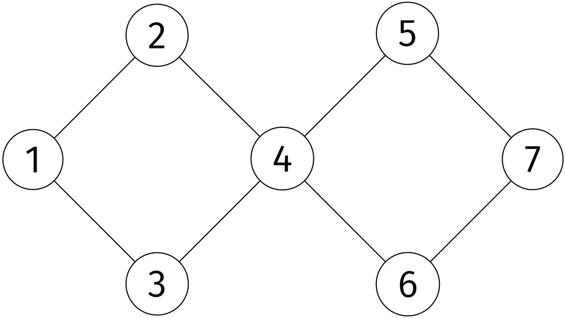
Theoretical Foundations II

Lecture 13 | Part 2

Nesting Properties of DFS

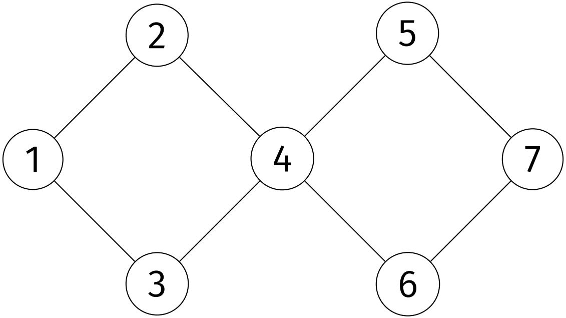
Exercise

True or **False**: if v is reachable from u and v is **undiscovered** when `dfs(u)` is called, then `dfs(v)` must be called during `dfs(u)`.



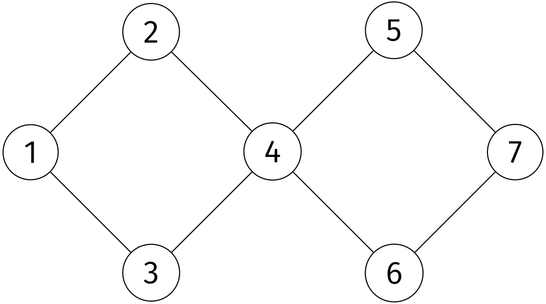
False!

- Suppose `dfs(4)` is the root call.
  - When `dfs(1)` is called, node 5 is undiscovered.
  - But `dfs(5)` is **not** called during `dfs(1)`.



However..

- This intuition is correct if there is a path of **undiscovered** nodes from u to v when `dfs(u)` is called.



Key Property of DFS (Informal)

- If at the time `dfs(u)` is called...
  1. v is **undiscovered**; and
  2. there is a path of **undiscovered** nodes from u to v,
- ...then `dfs(v)` will **start and finish** during the call to `dfs(u)`.

Exercise

Suppose while visiting node u, we see that neighbor v is **pending**. True or False: there is a path from v to u.

Start and Finish Times

- Keep a running clock (an integer).
- For each node, record
  - **Start time**: time when marked **pending**
  - **Finish time**: time when marked **visited**
- Increment clock whenever node is marked **pending**/**visited**

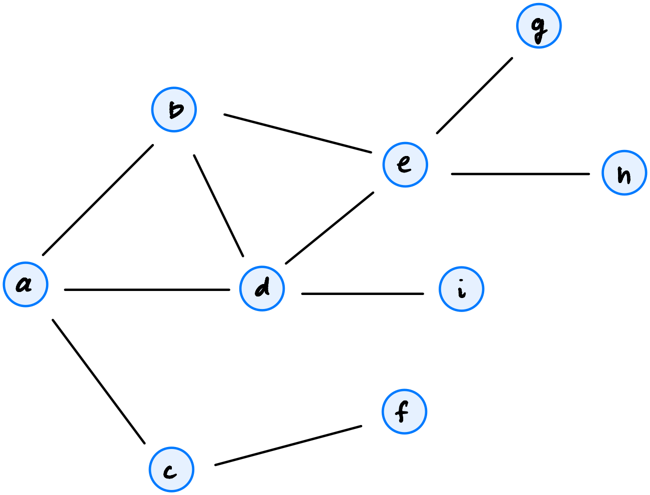
```python
from dataclasses import dataclass

@dataclass
class Times:
    clock: int
    start: dict
    finish: dict

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

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



Key Property of DFS

- Suppose `dfs(u)` is called before `dfs(v)`.
- If when `dfs(u)` is called there is a path of **undiscovered** nodes from u to v, then:

  `start[u]` < `start[v]` < `finish[v]` < `finish[u]`.
- Otherwise:

  `start[u]` < `finish[u]` < `start[v]` < `finish[v]`.

Key Property

- Take any two nodes *u* and *v* ($u \neq v$).
- Assume for simplicity that `start[u]` < `start[v]`.
- Exactly one of these is true:
  - `start[u]` < `start[v]` < `finish[v]` < `finish[u]`
  - `start[u]` < `finish[u]` < `start[v]` < `finish[v]`

DSC 40B

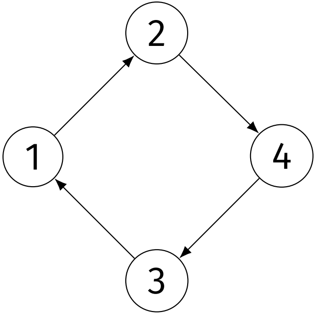
Theoretical Foundations II

Lecture 13 | Part 3

Cycles

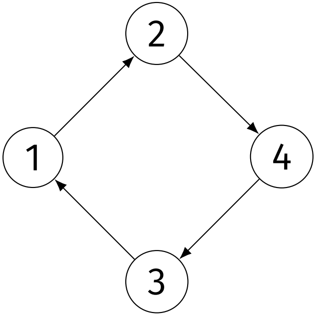
Cycle

- A **cycle** in a directed graph is a path that starts and ends at the same node.



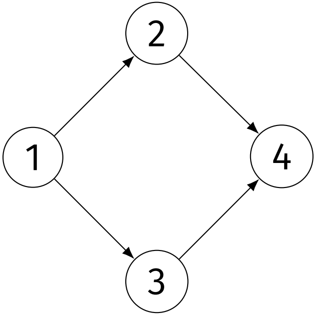
Cycle

- Alternatively: there is a **cycle** if u is reachable from v and v is reachable from u, for some $u \neq v$.



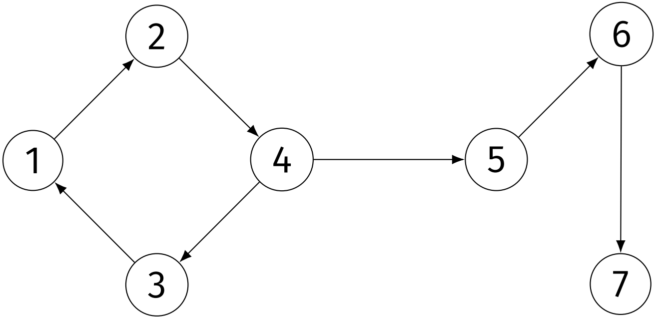
DAG

- A **directed acyclic graph** (DAG) is a directed graph with **no cycles**.



Cyclic Graphs

- A graph is cyclic even if it has only one cycle.
  - It doesn't have to be the whole graph.



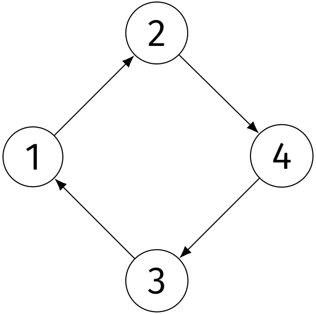
Detecting Cycles

- We check for cycles by looking for **back edges** in a full DFS.

- (u, v) is a **back edge** if while visiting node u, we see that v is **pending**.

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

Example



Theorem
A directed graph has a cycle ***if (and only if)*** *a full DFS finds a back edge.*

Why?

- If a back edge (u, v) is found, then a cycle exists.
  - Suppose v is pending when we visit u.
  - This means that there is a path from v to u.
  - There is also a path from u to v.
  - So there is a cycle.

Why?

- If a cycle exists, then there is a back edge.
  - Suppose there is a cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$.
  - Without loss of generality, assume v_1 is the first node in the cycle that is visited by the full DFS.
  - At the moment of `dfs(v_1)`, there is a path of undiscovered nodes between v_1 and v_k.
  - Therefore `dfs(v_k)` will be called during `dfs(v_1)`.
  - During `dfs(v_k)`, we'll see the back edge.

Exercise

Suppose v is reachable from u in a DAG.

True or false: after a full DFS, `finish[v] < finish[u]`.

Claim

- If *v* is reachable from *u* in a DAG, then:

  `finish[v]` < `finish[u]`

DSC 40B

Theoretical Foundations II

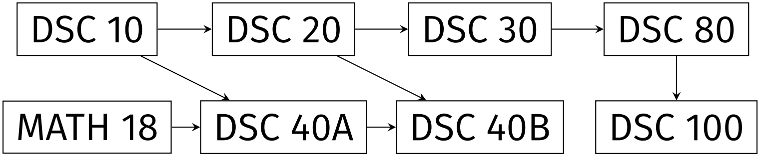
Lecture 13 | Part 4

Topological Sort

Applications of DFS

- Is node v reachable from node u? **DFS**, **BFS**
- Is the graph connected? **DFS**, **BFS**
- How many connected components? **DFS**, **BFS**
- Find the shortest path between u and v. **DFS**, **BFS**
- Does the graph have a cycle? **DFS**, **BFS**

Prerequisite Graphs



Goal: find order in which classes should be taken in order to satisfy the prerequisites of DSC 100.

Note

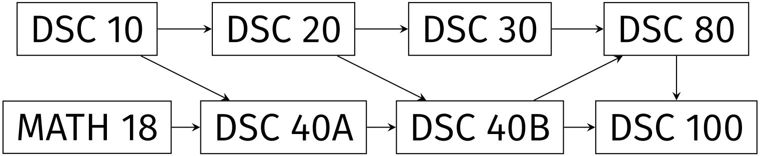
- Prerequisite graphs are[1] DAGs.

[1] Or they should be, at least!

Topological Sorts

- **Given**: a DAG, $G = (V, E)$.
- **Compute**: an ordering of V such that if $(u, v) \in E$, then u comes before v in the ordering
- This is called a **topological sort** of G.

Example

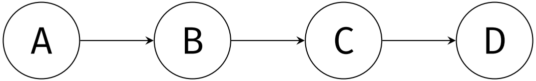


MATH 18, DSC 10, DSC 40A, DSC 20, DSC 40B, DSC 30, DSC 80, DSC 100

Computing a Topological Sort

- How do we compute a topological sort, algorithmically?

- **Observation**: if v is reachable from u, v **must** come **after** u in the topological sort.



Recall

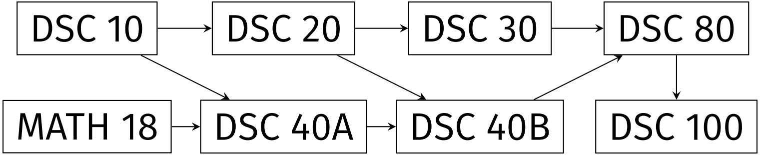
- Take any two nodes u and v ($u \neq v$).
- Assume the graph is a DAG, run DFS.
- If v is reachable from u, then `finish[v] < finish[u]`.

Putting it together...

- **Observation:** If v is reachable from u, then v must come after u in the topological sort.
- **Recall:** If v is reachable from u, then `finish[v] < finish[u]`.

Exercise

Compute start and finish times using DSC 10 as the source.



Idea

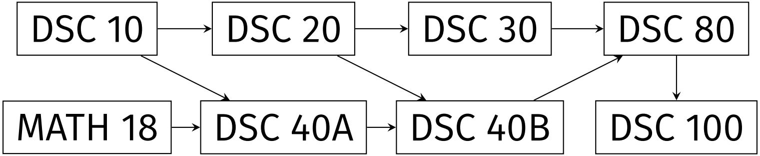
- **Observation:** If *v* is reachable from *u*, then *v* must come after *u* in the topological sort.
- **Recall:** If *v* is reachable from *u*, then `finish[v] < finish[u]`.
- **Therefore:** if `finish[v] < finish[u]`, then *v* must come after *u* in the topological sort.
- **Idea:** sort nodes in **descending** order by finish time.

Algorithm

- To find a topological sort (if it exists):
  - Compute times with Full DFS.
  - Sort in **descending** order by finish time.

- Time complexity:

Example



Note

- There can be many valid topological sorts!