Lecture 19: Basic Graph Algorithms
Processing Graphs

- Graphs model many scenarios
  - Many problems are presented as graph problems
  - Can then use known general graph algorithms to solve those problems

- Data is inputted as adjacency matrix or, more commonly, an adjacency lists

- To start processing the data, we often need some way to derive structure from this input

- **Breadth First Search** and **Depth First Search** are the most common simple ways of imposing structure.
Breadth First Search

BFS idea. Explore outward from $s$ in all possible directions, adding nodes one “layer” at a time.

BFS.

- $L_0 = \{s\}$.
- $L_1 =$ all neighbors of $L_0$.
- $L_2 =$ all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$.
- $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$.

Def: The distance from $u$ to $v$ is the number of edges on the shortest path from $u$ to $v$.

Theorem. For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.
**BFS Algorithm**

<table>
<thead>
<tr>
<th>Color</th>
<th>indicates status</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>(initial value)</td>
</tr>
<tr>
<td></td>
<td>undiscovered</td>
</tr>
<tr>
<td>gray</td>
<td>discovered, but neighbors</td>
</tr>
<tr>
<td></td>
<td>not fully processed</td>
</tr>
<tr>
<td>black</td>
<td>discovered and neighbors</td>
</tr>
<tr>
<td></td>
<td>fully processed</td>
</tr>
</tbody>
</table>

Every node stores a color, a distance and a parent

Distance (d): the length of shortest path from s to u

Parent (p): u’s predecessor on the shortest path from s to u

Note: Assume, initially, that G is connected (will fix later)
BFS Algorithm Complete

\[ \text{BFS}(G,s) : \]
\[ \text{for each vertex } u \in V - \{s\} \]
\[ \quad \text{u.color} \leftarrow \text{white} \]
\[ \quad \text{u.d} \leftarrow \infty \]
\[ \quad \text{u.p} \leftarrow \text{nil} \]
\[ \text{s.color} \leftarrow \text{gray} \]
\[ \text{s.d} \leftarrow 0 \]

1. initialize an empty queue \( Q \)
2. Enqueue \( Q, s \)
3. while \( Q \neq \emptyset \) do
4. \( u \leftarrow \text{Dequeue}(Q) \)
5. for each \( v \in \text{Adj}[u] \)
6. if \( v.\text{color} = \text{white} \) then
7. \( v.\text{color} \leftarrow \text{gray} \)
8. \( v.d \leftarrow u.d + 1 \)
9. \( v.p \leftarrow u \)
10. Enqueue \( Q, v \)
11. \( u.\text{color} \leftarrow \text{black} \)

Note: Nodes in Queue \( Q \)
- Are ones that have been seen but are unprocessed (gray)

• Algorithm keeps current active nodes in a (FIFO) Queue \( Q \)
• Starts by inserting \( s \) in \( Q \)
• At each step takes node \( u \) off \( Q \)
  • Checks all neighbors \( v \) of \( u \)
  • If \( v \) has not been seen yet
    • Marks \( v \) as seen (gray)
    • Says that distance from \( s \) to \( v \) is \( 1 + \text{dist to } u \)
    • Makes \( u \) the parent of \( v \)
    • inserts \( v \) in queue
• Marks \( u \) as being fully processed
BFS Algorithm Complete

BFS\((G, s)\):
for each vertex \(u \in V - \{s\}\)
\[\begin{align*}
&\text{\(u\).color } \leftarrow \text{ white} \\
&\text{\(u\).d } \leftarrow \infty \\
&\text{\(u\).p } \leftarrow \text{ nil}
\end{align*}\]
\[\begin{align*}
\text{s.color } &\leftarrow \text{ gray} \\
\text{s.d } &\leftarrow 0
\end{align*}\]
initialize an empty queue \(Q\)
Enqueue\((Q, s)\)
while \(Q \neq \emptyset\) do
\[\begin{align*}
&\text{\(u\) } \leftarrow \text{Dequeue}(Q) \\
&\text{\(\text{for each ~} v \in Adj[u]\) if v.color = white then} \\
&\quad \text{\(v\).color } \leftarrow \text{ gray} \\
&\quad \text{\(v\).d } \leftarrow \text{\(u\).d } + 1 \\
&\quad \text{\(v\).p } \leftarrow \text{\(u\)} \\
&\quad \text{Enqueue}(Q, v) \\
&\text{\(u\).color } \leftarrow \text{ black}
\end{align*}\]

Parent pointers:
- Pointing to the node that leads to its discovery
- Parent must be in \(L_{i-1}\)
- Can follow parent pointers to find the actual shortest path
- The pointers form a BFS tree, rooted at \(s\)

Running time:
\[\sum_u (1 + \text{deg}(u)) = \Theta(V + E), \text{ which is } \Theta(E) \text{ if the graph is connected.}\]
Note: BFS finds the shortest path from $s$ to every other node.
Connected Components

Connected component containing \( s \). All nodes reachable from \( s \).

Connected component containing node \( 1 = \{1, 2, 3, 4, 5, 6, 7, 8\} \).

BFS starting from \( s \) finds the connected component containing \( s \).

Repeatedly running BFS from an undiscovered node finds all the connected components.
Modification for Finding Connected Components

**BFS(G):**

for each vertex \( u \in V \) do

- \( u.color \leftarrow \text{white} \)
- \( u.d \leftarrow \infty \)
- \( u.p \leftarrow \text{nil} \)

for each vertex \( u \in V \) do

- if \( u.color = \text{white} \) then

  BFS-Visit\((u)\)

**BFS-Visit\((G,s)\):**

/*Assumes \( s \) is white*/

- \( s.color \leftarrow \text{gray} \)
- \( s.d \leftarrow 0 \)

1. initialize an empty queue \( Q \)
2. Enqueue\((Q,s)\)
3. while \( Q \neq \emptyset \) do
4. \( u \leftarrow \text{Dequeue}(Q) \)
5. for each \( v \in \text{Adj}[u] \)
6. if \( v.color = \text{white} \) then
7. \( v.color \leftarrow \text{gray} \)
8. \( v.d \leftarrow u.d + 1 \)
9. \( v.p \leftarrow u \)
10. Enqueue\((Q,v)\)
11. \( u.color \leftarrow \text{black} \)

The old BFS\((G,s)\) algorithm is renamed BFS-Visit\((G,s)\).

A new upper-level BFS\((G)\) is created.

BFS\((G)\) initializes all vertices to white (unvisited)
It then calls all vertices \( s \), passing them to BFS-visit\((s)\), if \( s \) was not already seen while traversing a previously visited connected component.
**Connected Components**

*Connected component containing* $s$. *All nodes reachable from* $s$. 

BFS-Visit(1) would turn all nodes in leftmost component black.
Connected Components

Connected component containing $s$. All nodes reachable from $s$.

BFS-Visit(1) would turn all nodes in the leftmost component black.

BFS-Visit(2) would turn all nodes in the rightmost component black.
Connected Components

Connected component containing $s$. All nodes reachable from $s$.

BFS-Visit(1) would turn all nodes in leftmost component black.

BFS-Visit(2) would turn all nodes in rightmost component black.
**Connected Components**

*Connected component containing* \( s \). *All nodes reachable from* \( s \).

BFS-Visit(1) would turn all nodes in leftmost component black

BFS-Visit(2) would turn all nodes in rightmost component black

BFS-Visit(i) for \( 3 \leq i \leq 9 \) would do nothing.

BFS-Visit(10) would then turn all nodes in middle component black
Connected Components

Connected component containing $s$. All nodes reachable from $s$.

BFS-Visit(1) would turn all nodes in leftmost component black.

BFS-Visit(2) would turn all nodes in rightmost component black.

BFS-Visit($i$) for $3 \leq i \leq 9$ would do nothing.

BFS-Visit(10) would then turn all nodes in middle component black.
s-t connectivity and shortest path in directed graphs

s-t connectivity (often called reachability for directed graphs). Given two nodes s and t, is there a path from s to t?

- Undirected graph: s can reach t ⇔ t can reach s
- Directed graph: Not necessarily true

s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

- Undirected graph: p is the shortest path from s to t ⇔ p is the shortest path from t to s
- Directed graph: Not necessarily true

BFS on a directed graph. Same as in undirected case

- Ex: Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.
Strong Connectivity in Directed Graphs

Def. Node $u$ and $v$ are mutually reachable if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Definition: vertex $s$ is "strong" in Graph $G$ if,
for every vertex $t$, there is a path from $s$ to $t$ and from $t$ to $s$.

Observation 1: If graph $G$ has a strong vertex $s$ then
EVERY vertex in $G$ is strong

Observation 2: A graph $G$ is strongly connected if and only if every vertex in $G$ is strong
**Strong Connectivity in Directed Graphs**

**Def.** Node $u$ and $v$ are mutually reachable if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

**Def.** A graph is strongly connected if every pair of nodes is mutually reachable.

**Algorithm for checking strong connectivity**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Reverse all edges in $G$, and run BFS from $s$.
- Return true iff all nodes reached in both BFS executions.
Strongly Connected Components

**Strongly-Connected-Components(G):**
create $G^{rev}$ which is $G$ with all edges reversed
while there are nodes left do
  $u \leftarrow$ any node
  run BFS in $G$ starting from $u$
  run BFS in $G^{rev}$ starting from $u$
  $C \leftarrow \{\text{nodes reached in both BFSs}\}$
output $C$ as a strongly connected component
remove $C$ and its edges from $G$ and $G^{rev}$

**Running time:** $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

**Strongly-Connected-Components** $(G)$:
- create $G^{rev}$ which is $G$ with all edges reversed
- while there are nodes left do
  - $u \leftarrow$ any node
  - run BFS in $G$ starting from $u$
  - run BFS in $G^{rev}$ starting from $u$
  - $C \leftarrow \{\text{nodes reached in both BFSs}\}$
  - output $C$ as a strongly connected component
  - remove $C$ and its edges from $G$ and $G^{rev}$

**Running time:** $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components($G$):
create $G^{\text{rev}}$ which is $G$ with all edges reversed
while there are nodes left do
    $u \leftarrow$ any node
    run BFS in $G$ starting from $u$
    run BFS in $G^{\text{rev}}$ starting from $u$
    $C \leftarrow \{ \text{nodes reached in both BFSs} \}$
output $C$ as a strongly connected component
remove $C$ and its edges from $G$ and $G^{\text{rev}}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components($G$):
create $G^{\text{rev}}$ which is $G$ with all edges reversed
while there are nodes left do
    $u \leftarrow$ any node
    run BFS in $G$ starting from $u$
    run BFS in $G^{\text{rev}}$ starting from $u$
    $C \leftarrow \{\text{nodes reached in both BFSs}\}$
output $C$ as a strongly connected component
remove $C$ and its edges from $G$ and $G^{\text{rev}}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components($G$):
create $G^{rev}$ which is $G$ with all edges reversed
while there are nodes left do
  $u \leftarrow$ any node
  run BFS in $G$ starting from $u$
  run BFS in $G^{rev}$ starting from $u$
  $C \leftarrow$ {nodes reached in both BFSs}
output $C$ as a strongly connected component
remove $C$ and its edges from $G$ and $G^{rev}$

Running time: $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

\textbf{Strongly-Connected-Components}(G): \\
create $G^{rev}$ which is $G$ with all edges reversed \\\nwhile there are nodes left do \\\n\hspace{1em} $u \leftarrow$ any node \\\n\hspace{1em} run BFS in $G$ starting from $u$ \\\n\hspace{1em} run BFS in $G^{rev}$ starting from $u$ \\\n\hspace{1em} $C \leftarrow \{\text{nodes reached in both BFSs}\}$ \\\n\hspace{1em} output $C$ as a strongly connected component \\\n\hspace{1em} remove $C$ and its edges from $G$ and $G^{rev}$

\textbf{Running time:} $O(VE)$

See text book for a $\Theta(V + E)$ algorithm (not required)
**Strongly Connected Components**

**Running time:** $O(VE)$

See textbook for a $\Theta(V + E)$ algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components\((G)\):
create \(G^{rev}\) which is \(G\) with all edges reversed
while there are nodes left do
  \(u \leftarrow\) any node
  run BFS in \(G\) starting from \(u\)
  run BFS in \(G^{rev}\) starting from \(u\)
  \(C \leftarrow\) {nodes reached in both BFSs}
output \(C\) as a strongly connected component
remove \(C\) and its edges from \(G\) and \(G^{rev}\)

Running time: \(O(VE)\)

See text book for a \(\Theta(V + E)\) algorithm (not required)
Strongly Connected Components

**Strongly-Connected-Components**(\(G\)):
create \(G^{rev}\) which is \(G\) with all edges reversed
while there are nodes left do
  \(u \leftarrow\) any node
  run BFS in \(G\) starting from \(u\)
  run BFS in \(G^{rev}\) starting from \(u\)
  \(C \leftarrow\) {nodes reached in both BFSs}
output \(C\) as a strongly connected component
remove \(C\) and its edges from \(G\) and \(G^{rev}\)

**Running time:** \(O(VE)\)

See text book for a \(\Theta(V + E)\) algorithm (not required)
Strongly Connected Components

Strongly-Connected-Components\((G)\):
create \(G^{rev}\) which is \(G\) with all edges reversed
while there are nodes left do
  \(u \leftarrow\) any node
  run BFS in \(G\) starting from \(u\)
  run BFS in \(G^{rev}\) starting from \(u\)
  \(C \leftarrow\) {nodes reached in both BFSs}
output \(C\) as a strongly connected component
remove \(C\) and its edges from \(G\) and \(G^{rev}\)

Running time: \(O(VE)\)

See text book for a \(\Theta(V + E)\) algorithm (not required)
Depth First Search and DFS Tree

• **Breadth first search** is “Broad”.
  • It builds a wide tree, connecting a node to ALL of the neighbors that have not yet been processed.
  • Once a node starts being processed, it sees ALL of its neighbors before any other node is processed.

• There is another procedure, called **DEPTH first search**.
  • Instead of going broad, it goes **DEEP**
  • It recursively searches deep into the tree.

• When a node $u$ is processed, it looks at each of its neighbors in order.
  • At the time $u$ checks a neighbor $v$, DFS starts processing $v$ (which starts processing its children, which start processing their children, etc.).

• Only after all of $v$’s descendants have been processed does $u$ go on to process its next neighbor.
Depth First Search and DFS Tree

1. Depth First Search
2. DFS Tree
3. DFS Process
4. DFS Search
5. DFS Tree
6. DFS Process
7. DFS Search
8. DFS Tree
9. DFS Process
**DFS Algorithm**

**DFS(G):**

for each vertex \( u \in V \) do
- \( u\.color \leftarrow \text{white} \)
- \( u\.p \leftarrow \text{nil} \)

for each vertex \( u \in V \) do
  if \( u\.color = \text{white} \) then
    DFS-Visit(\( u \))

---

**Colors:**

- **White:** undiscovered
- **Gray:** discovered, but neighbors not fully explored (on recursion stack)
- **Black:** discovered and neighbors fully explored

**Parent pointers:**

- Pointing to the node that leads to its discovery
- The pointers form a tree, rooted at \( s \)

• **DFS(G)** calls the **DFS-visit** search on each vertex \( u \)

• Before **DFS-Visit(\( u \))** returns, all nodes in the connected component containing \( u \) are turned black (will see later)

• So **DFS-Visit** will only be called once for each connected component in \( G \)
**DFS Algorithm**

**DFS(G):**
for each vertex \( u \in V \) do
  \( u.\text{color} \leftarrow \text{white} \)
  \( u.p \leftarrow \text{nil} \)
for each vertex \( u \in V \) do
  if \( u.\text{color} = \text{white} \) then
    \( \text{DFS-Visit}(u) \)

**DFS-Visit(u):**
\( u.\text{color} \leftarrow \text{gray} \)
for each \( v \in \text{Adj}[u] \) do
  if \( v.\text{color} = \text{white} \) then
    \( v.p \leftarrow u \)
    \( \text{DFS-Visit}(v) \)
\( u.\text{color} \leftarrow \text{black} \)

**Running time:** \( \Theta(V + E) \)

**Colors:**
- **White:** undiscovered
- **Gray:** discovered, but neighbors not fully explored (on recursion stack)
- **Black:** discovered and neighbors fully explored

**Parent pointers:**
- Pointing to the node that leads to its discovery
- The pointers form a tree, rooted at \( s \)

We can add starting and finishing time for each \( u \):
- **Starting time** when \( u.\text{color} \leftarrow \text{gray} \)
- **Finishing time** when \( u.\text{color} \leftarrow \text{black} \)
Adjacency Lists:
a: b, i, k, n, h
b: a, f, d, e, i
c: f
d: b
e: b, i
f: c, b
g: h
h: a, g
i: e, b, k, a
j: n, m
k: i, a
l: n
m: j, n
n: a, j, m, l

The starting and finishing times are useful for some applications (to be discussed later)
The bold edges form the DFS tree.
The rest of the edges (light red) point to ancestors in the tree, and are called back-edges.
Back edges are also useful for some applications.
Application: Cycle Detection

Problem: Given an undirected graph $G = (V, E)$, check if it contains a cycle.

Idea:
- A tree (connected and acyclic) contains exactly $V - 1$ edges.
- If it has fewer edges, it cannot be connected.
- If it has more edges, it must contain a cycle.

Algorithm:
- Run BFS/DFS to find all the connected components of $G$.
- For each connected component, count the number of edges.
- If # edges \(\geq\) # vertices, return “cycle detected”.

Running time: $\Theta(V + E)$

Q: What if we also want to find a cycle (any is OK) if it exists?
Tree edges, back edges, and cross edges

After running BFS or DFS on an undirected graph, all edges can be classified into one of 3 types:

- **Tree edges**: traversed by the BFS/DFS.
- **Back edges**: connecting a node with one of its ancestors in the BFS/DFS-tree (other than its parent).
- **Cross edges**: connecting two nodes with no ancestor/descendent relationship.

**Theorem**: In a DFS on an undirected graph, there are no cross edges.

**Pf**: Consider any edge \((u, v)\) in \(G\).
- Without loss of generality, assume \(u\) is discovered before \(v\).
- Then \(v\) is discovered while \(u\) is gray (why?).
- Hence \(v\) is in the DFS subtree rooted at \(u\).
  - If \(v.p = u\), then \((u, v)\) is a tree edge.
  - If \(v.p \neq u\), then \((u, v)\) is a back edge.

**Theorem**: In a BFS on an undirected graph, there are no back edges.

(Not proven)
**DFS for cycle detection**

**Idea:** Run DFS on each connected component of $G$.
- If $(u, v)$ is a back edge.
  - $v$ is an ancestor (but not parent) of $u$ in the DFS trees.
  - There is thus a path from $v$ to $u$ in the DFS-tree and
  - $v$ to $u$ plus back edge $(u, v)$ creates a cycle.
- If no back edge exists then it only contains (DFS) tree edges
  - the graph is a forest, and hence is acyclic.

- In DFS starting at $a$, $(i,b)$ was first back edge found
  - $b$ was ancestor (not parent) of $i$ in tree
  - tree contains path $(b\rightarrow e\rightarrow i)$ from $b$ to $i$
  - this path plus edge $(i,b)$ is the cycle $b\rightarrow e\rightarrow i\rightarrow b$
DFS for cycle detection

\textbf{CycleDetection}(G):
\begin{algorithmic}
\FOR {each vertex \textit{u} $\in V$}
\STATE \textit{u}.\textit{color} $\leftarrow$ white
\STATE \textit{u}.\textit{p} $\leftarrow$ nil
\ENDFOR
\FOR {each vertex \textit{u} $\in V$}
\IF {\textit{u}.\textit{color} = white}
\STATE \textbf{DFS-Visit}(\textit{u})
\ENDIF
\ENDFOR
\STATE return "No cycle"
\end{algorithmic}

\textbf{DFS-Visit}(\textit{u}):
\begin{algorithmic}
\STATE \textit{u}.\textit{color} $\leftarrow$ gray
\FOR {each \textit{v} $\in$ \textit{Adj}[\textit{u}]}
\IF {\textit{v}.\textit{color} = white}
\STATE \textit{v}.\textit{p} $\leftarrow$ \textit{u}
\STATE \textbf{DFS-Visit}(\textit{v})
\ENDIF
\ELSE IF {\textit{v} $\neq$ \textit{u}.\textit{p}}
\STATE //back edge (\textit{u},\textit{v})
\STATE output "Cycle found:"
\STATE \WHILE {\textit{u} $\neq$ \textit{v}}
\STATE output \textit{u}
\STATE \textit{u} $\leftarrow$ \textit{u}.\textit{p}
\STATE output \textit{v}
\STATE return
\STATE \textit{u}.\textit{color} $\leftarrow$ black
\ENDIF
\end{algorithmic}

\textbf{Running time:} $\Theta(V)$
\begin{itemize}
\item Only traverse DFS-tree edges, until the first non-tree edge is found
\item At most $V-1$ tree edges
\end{itemize}
**Directed Graph**

A directed graph distinguishes between edge \((u, v)\) and edge \((v, u)\). Directed graphs are often used to represent order-dependent tasks.

- **out-degree** of vertex \(v\) is the number of edges leaving \(v\)
- **in-degree** of vertex \(v\) is the number of edges entering \(v\)
- Each edge \((u, v)\) contributes one to the out-degree of \(u\) and one to the in-degree of \(v\), so

\[
\sum_{v \in V} \text{out-degree}(v) = \sum_{v \in V} \text{in-degree}(v) = |E|
\]
Topological Sort

- **Directed Acyclic Graph (DAG):** Directed graph with no cycles.

- **A Topological ordering** of a graph is a linear ordering of the vertices of a DAG such that if \((u, v)\) is in the graph, \(u\) appears before \(v\) in the linear ordering.

- Topological ordering may not be unique

- The graph above has many topological orderings
  - 0, 6, 1, 4, 3, 2, 5, 7, 8, 9
  - 0, 4, 1, 6, 2, 5, 3, 7, 8, 9
  - ...
Topological Sort Algorithm

• Observations
  • A DAG must contain at least one vertex with in-degree zero

• Algorithm: Topological Sort (TS)
  1. Output a vertex $u$ with in-degree zero in current graph.
  2. Remove $u$ and all edges $(u, v)$ from current graph.
  3. If graph is not empty, goto step 1.

• Correctness
  • At every stage, current graph remains a DAG (why?)
  • Because current graph is always a DAG, TS can always output some vertex. So algorithm outputs all vertices.
  • Suppose output order is not a topological order.
    => Then there is some edge $(u, v)$ such that $v$ appears before $u$ in the order. This is impossible, though, because $v$ can not be output until edge $(u, v)$ is removed!
Topological Sort Algorithm

Topological Sort(G)

Initialize Q to be an empty queue;

foreach u in V do
    If in−degree(u) = 0 then
        // Find all starting vertices
        Enqueue(Q, u);
    end
end

while Q is not empty do
    u = Dequeue(Q);
    Output u;

    foreach v in Adj(u) do
        // remove u’s outgoing edges
        in−degree(v) = in−degree(v) − 1
        if in−degree(v) = 0 then
            Enqueue(Q, v);
        end
    end
end
Example

\[ Q = \{ \} \]
Example

\[ Q = \{0\} \]
Example

\[ Q = \{0\} \]
Example

\[ Q = \{6,1,4\} \]

Output: 0
Example

\[ Q = \{1,4,3\} \]

Output: 0,6
Example

$Q = \{4, 3, 2\}$

Output: 0, 6, 1
Example

\[ Q = \{3, 2\} \]

Output: 0, 6, 1, 4
Example

\[ Q = \{2\} \]
Output: 0,6,1,4,3
Example

\[ Q = \{7,5\} \]

Output: 0,6,1,4,3,2
Example

\[ Q = \{5,8\} \]

Output: 0,6,1,4,3,2,7
Example

\[
Q = \{8\}
\]

Output: 0,6,1,4,3,2,7,5
Example

\[ Q = \{9\} \]

Output: 0, 6, 1, 4, 3, 2, 7, 5, 8
Example

\[ Q = \{\} \]

Output: 0, 6, 1, 4, 3, 2, 7, 5, 8, 9

Done!
Topological Sort: Complexity

- We never visit a vertex more than once
- For each vertex, we examine all outgoing edges
  - $\sum_{v \in V} \text{out-degree}(v) = E$
- Therefore, the running time is $O(V + E)$