Topological Sort

Version of September 23, 2016
In a directed graph, we distinguish between edge \((u, v)\) and edge \((v, u)\).
In a directed graph, we distinguish between edge \((u, v)\) and edge \((v, u)\).

- **out-degree** of a vertex is the number of edges leaving it.
- **in-degree** of a vertex is the number of edges entering it.

\[
\sum_{v \in V} \text{out-degree}(v) = \sum_{v \in V} \text{in-degree}(v) = |E|
\]
In a directed graph, we distinguish between edge \((u, v)\) and edge \((v, u)\).

- **out-degree** of a vertex is the number of edges leaving it.
- **in-degree** of a vertex is the number of edges entering it.

\[
\sum_{v \in V} \text{out-degree}(v) = \sum_{v \in V} \text{in-degree}(v) = |E|.
\]
In a directed graph, we distinguish between edge \((u, v)\) and edge \((v, u)\)

- **out-degree** of a vertex is the number of edges leaving it
- **in-degree** of a vertex is the number of edges entering it
- Each edge \((u, v)\) contributes one to the out-degree of \(u\) and one to the in-degree of \(v\)
In a directed graph, we distinguish between edge \((u, v)\) and edge \((v, u)\).

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

\[
\sum_{v \in V} \text{out-degree}(v) = \sum_{v \in V} \text{in-degree}(v) = |E|
\]
Directed graphs are often used to represent order-dependent tasks.

Edge \((u, v)\) denotes that task \(v\) cannot start until task \(u\) is finished.

Clearly, for the system not to hang, the graph must be acyclic. It must be a directed acyclic graph (or DAG).
Directed graphs are often used to represent **order-dependent** tasks.

That is, we cannot start a task before another task finishes.
Directed graphs are often used to represent order-dependent tasks.

- That is, we cannot start a task before another task finishes.

Edge \((u, v)\) denotes that task \(v\) cannot start until task \(u\) is finished.
Directed graphs are often used to represent order-dependent tasks.
- That is, we cannot start a task before another task finishes.

Edge \((u, v)\) denotes that task \(v\) cannot start until task \(u\) is finished.

Clearly, for the system not to hang, the graph must be acyclic.
Directed graphs are often used to represent order-dependent tasks.

- That is, we cannot start a task before another task finishes.

Edge \((u, v)\) denotes that task \(v\) cannot start until task \(u\) is finished.

Clearly, for the system not to hang, the graph must be acyclic.

- It must be a directed acyclic graph (or DAG).
Course dependence chart
09/10

Red: COMP/CSIE Core
Green: COMP/CSIE Required
Purple: CSIE (NW) Required
Blue: CSIE (MC) Required

F and S means offered in Fall and Spring respectively.
Course offering schedule shown here is for reference only; the actual offering schedule may vary slightly from year to year.

Version of September 23, 2016
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.
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.

E.g., order in which classes can be taken.
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.

e.g., order in which classes can be taken
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.

- e.g., order in which classes can be taken

![Graph Diagram]

- Topological ordering may not be unique as there are many “equal” elements!
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.

- e.g., order in which classes can be taken

Topological ordering may not be unique as there are many “equal” elements!

E.G., there are several topological orderings:

- \(0, 6, 1, 4, 3, 2, 5, 7, 8, 9\)
- \(0, 4, 1, 6, 2, 5, 3, 7, 8, 9\)
- ...
Observations

A DAG must contain at least one vertex with in-degree zero (why?)
Topological Sort Algorithm

- Observations
  - A DAG must contain at least one vertex with in-degree zero (why?)

- Algorithm: **Topological Sort**
  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 is a DAG (why?)
Because current graph is always a DAG, algorithm can always output some vertex. So algorithm outputs all vertices.
Suppose order output was 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!
Observations
- A DAG must contain at least one vertex with in-degree zero (why?)

Algorithm: Topological Sort
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
Observations
- A DAG must contain at least one vertex with in-degree zero (why?)

Algorithm: Topological Sort
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 is a DAG (why?)
- Because current graph is always a DAG, algorithm can always output some vertex. So algorithm outputs all vertices.
- Suppose order output was 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;

\[
\text{foreach } u \text{ in } V \text{ do}
\]

\[
\quad \text{if } \text{in-degree}(u) = 0 \text{ then}
\]

\[
\quad \quad \text{// Find all starting vertices}
\]

\[
\quad \quad \text{Enqueue}(Q, u);
\]

\[
\quad \end
\]

\[
\text{end}
\]

\[
\text{while } Q \text{ is not empty do}
\]

\[
\quad u = \text{Dequeue}(Q);
\]

\[
\quad \text{Output } u;
\]

\[
\quad \text{foreach } v \text{ in Adj}(u) \text{ do}
\]

\[
\quad \quad \text{// remove } u\text{'s outgoing edges}
\]

\[
\quad \quad \text{in-degree}(v) = \text{in-degree}(v) - 1;
\]

\[
\quad \quad \text{if } \text{in-degree}(v) = 0 \text{ then}
\]

\[
\quad \quad \quad \text{Enqueue}(Q, v);
\]

\[
\quad \quad \end
\]

\[
\quad \end
\]

\[
\text{end}
\]

\[
\text{end}
\]
Example

\[ Q = \{\} \]

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

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

Output: 0

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

Output: 0, 6
Example

\[ Q = \{4, 3, 2\} \]

Output: 0, 6, 1

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

Output: 0, 6, 1, 4
Example

\[ Q = \{2\} \]

Output: 0, 6, 1, 4, 3

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

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

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

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

\[ Q = \{8\} \]

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

\[ Q = \{9\} \]

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

\[ Q = \{\} \]

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

Done!
We never visit a vertex more than once
Topological Sort: Complexity

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

Question
Can we use DFS to implement topological sort?