CSE 373: Topological Sort and Minimum Spanning Trees

Michael Lee
Friday, Feb 23, 2018
Topological sort

Design question: suppose we have a bunch of classes with pre-requisites.

MATH 126
CSE142
CSE143
CSE374
CSE373
CSE410
CSE413
CSE417
CSE415
XYZ

Goal: list out classes in a “valid” order
For example: 126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415
Design question: suppose we have a bunch of classes with pre-requisites.

Goal: list out classes in a “valid” order

For example: 126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415
Topological sort

<table>
<thead>
<tr>
<th>Topological sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a <em>directed, acyclic graph</em> (DAG), running <strong>topological sort</strong> on that graph will produce a list of all the vertices in an order such that no vertex appears before another vertex that has an edge to it.</td>
</tr>
</tbody>
</table>
Topological sort

Given a *directed, acyclic graph* (DAG), running **topological sort** on that graph will produce a list of all the vertices in an order such that no vertex appears before another vertex that has an edge to it.

Example applications:

- Any scheduling problem (scheduling courses, scheduling threads)
- Computing order to recompute cells in spreadsheet
- Determining order to compile files using a MAkefile

In general: taking a dependency graph and coming up with order of execution.
Topological sort

Questions

➡️ Can we perform topo-sort on graphs containing cycles?

➡️ Is there always one unique output per graph?
Questions

- Can we perform topo-sort on graphs containing cycles?
  No: how do we decide which node comes first?

- Is there always one unique output per graph?
  No: see example on inked slides
Intuition:

- The only nodes we can start with are also nodes that have in-degree 0
Intuition:

- The only nodes we can start with are also nodes that have in-degree 0
- So, start by adding those to the list
Intuition:

- The only nodes we can start with are also nodes that have in-degree 0
- So, start by adding those to the list
- Is there some way of “repeating” this process?
Topological sort

Setup

▷ Look at each vertex and record its in-degree somewhere
Topological sort

Setup

- Look at each vertex and record its in-degree somewhere

Core loop

- Choose an arbitrary vertex $a$ with in-degree 0
Topological sort

Setup

- Look at each vertex and record its in-degree somewhere

Core loop

- Choose an arbitrary vertex $a$ with in-degree 0
- Output $a$ and conceptually remove it from the graph
Topological sort

Setup

- Look at each vertex and record its in-degree somewhere

Core loop

- Choose an arbitrary vertex $a$ with in-degree 0
- Output $a$ and conceptually remove it from the graph
- For each vertex $b$ adjacent to $a$, decrement the in-degree of $b$
Topological sort

Setup

- Look at each vertex and record its in-degree somewhere

Core loop

- Choose an arbitrary vertex \( a \) with in-degree 0
- Output \( a \) and conceptually remove it from the graph
- For each vertex \( b \) adjacent to \( a \), decrement the in-degree of \( b \)
- Repeat
Topological sort: Example 1

Example again:

Output:
Topological sort: Example 1

Example again:

Output:
Topological sort: Example 1

Example again:

Output: CSE142,
Topological sort: Example 1

Example again:

Output: CSE142,
Example again:

Output: CSE142, MATH126,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143,
Example again:

Output: CSE142, MATH126, CSE143, CSE374,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143, CSE374,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373,
Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373,
Topological sort: Example 1

Example again:

- MATH126: 0
- CSE142: 0
- CSE143: 0
- CSE373: 0
- CSE374: 0
- CSE410: 0
- CSE413: 0
- CSE415: 0
- CSE417: 0
- XYZ: 2

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413,
Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413, CSE410,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413, CSE410,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413, CSE410, XYZ,
Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413, CSE410, XYZ, CSE417,
Topological sort: Example 1

Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413, CSE410, XYZ, CSE417, CSE415
Example again:

Output: CSE142, MATH126, CSE143, CSE374, CSE373, CSE413, CSE410, XYZ, CSE417, CSE415
Now you try. List one possible output:

a, b, g, c, e, h, d, i, f, j, k
Topological sort: Example 2

Now you try. List one possible output:

One possible answer: a, b, g, c, e, h, d, i, f, j, k
Topological sort: Algorithm

Our algorithm so far:

Setup

- Look at each vertex and record its in-degree somewhere

Core loop

- Choose an arbitrary vertex $a$ with in-degree 0
- Output $a$ and conceptually remove it from the graph
- For each vertex $b$ adjacent to $a$, decrement the in-degree of $b$
- Repeat
Topological sort: Algorithm

One possible implementation:

```python
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = getNextVertex(indegrees, visited)
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1

    return output

def getNextVertex(indegrees, visited):
    for (node, num : indegrees):
        if (num == 0 and node not in visited):
            return node
```

Questions:

Worst-case runtime?

$O(|V|^2 + |E|)$

Is this optimal?

Maybe not. Do we really need to look at each node multiple times? Can we somehow get $O(|V| + |E|)$?
Topological sort: Algorithm

One possible implementation:

```python
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = getNextVertex(indegrees, visited)
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1

    return output

def getNextVertex(indegrees, visited):
    for (node, num : indegrees):
        if (num == 0 and node not in visited):
            return node
```

Questions:

Worst-case runtime?

$O((|V|^2 + |E|))$

Is this optimal? Maybe not. Do we really need to look at each node multiple times? Can we somehow get $O(|V| + |E|)$?
Topological sort: Algorithm

One possible implementation:

```python
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = getNextVertex(indegrees, visited)
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1

    return output

def getNextVertex(indegrees, visited):
    for (node, num : indegrees):
        if (num == 0 and node not in visited):
            return node
```

Questions:
Worst-case runtime?
\( O(|V|^2 + |E|) \)

Is this optimal?
Maybe not. Do we really need to look at each node multiple times? Can we somehow get \( O(|V| + |E|) \)?
Topological sort: Algorithm

One possible implementation:

```python
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = getNextVertex(indegrees, visited)
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1

    return output

def getNextVertex(indegrees, visited):
    for (node, num : indegrees):
        if (num == 0 and node not in visited):
            return node
```

Questions:

Worst-case runtime?
$$O(|V|^2 + |E|)$$

Is this optimal?
Topological sort: Algorithm

One possible implementation:

```python
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = getNextVertex(indegrees, visited)
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1

    return output

def getNextVertex(indegrees, visited):
    for (node, num : indegrees):
        if (num == 0 and node not in visited):
            return node
```

Questions:
Worst-case runtime?
\( O(|V|^2 + |E|) \)

Is this optimal?
Maybe not. Do we really need to look at each node multiple times? Can we somehow get \( O(|V| + |E|) \)?
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()
    compute all indegrees and add to dictionary
    while (we still need to visit vertices):
        current = getNextVertex(indegrees, visited)
        add current to both visited and output
        for (v : current.allNeighbors()):
            indegrees[v] -= 1
    return output

def getNextVertex(indegrees, visited):
    for (node, num : indegrees):
        if (num == 0 and node not in visited):
            return node

How can we improve this?
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()
    compute all indegrees and add to dictionary
    while (we still need to visit vertices):
        current = getNextVertex(indegrees, visited)
        add current to both visited and output
        for (v : current.allNeighbors()):
            indegrees[v] -= 1
    return output

def getNextVertex(indegrees, visited):
    for (node, num : indegrees):
        if (num == 0 and node not in visited):
            return node

How can we improve this?

- Can we get rid of the inner loop somehow?
- Would using different/more data structures help?
- Can we collect additional information somewhere else?
Insight: When we’re updating the indegrees, we already know which nodes now have an indegree of zero!
**Insight:** When we’re updating the indegrees, we already know which nodes now have an indegree of zero!

Why are we discarding and recomputing that info? Let’s just use it!
Insight: When we’re updating the indegrees, we already know which nodes now have an indegree of zero!

Why are we discarding and recomputing that info? Let’s just use it!

```python
def toposort(graph):
    indegrees = new HashMap< Vertex, Integer >()
    visited = new HashSet< Vertex >()
    output = new AnyList< Vertex >()
    stack = new Stack< Vertex >();

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = stack.pop()
        add current to both visited and output

    for (v : current.allNeighbors()):
        indegrees[v] -= 1
        if (indegrees[v] == 0):
            stack.push(v)

    return output
```
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = stack.pop()
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new ArrayList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = stack.pop()
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output

Question: Does this actually work?
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new ArrayList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary

    while (we still need to visit vertices):
        current = stack.pop()
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output

Question: Does this actually work?

Answer: No, there’s a bug! The stack is initially empty, so first pop fails.
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary
    also add all nodes with indegree zero to stack

    while (we still need to visit vertices):
        current = stack.pop()
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary
    also add all nodes with indegree zero to stack

    while (we still need to visit vertices):
        current = stack.pop()
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output

Question: Can we improve this algorithm even more?
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    visited = new HashSet<Vertex>()
    output = new AnyList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary
    also add all nodes with indegree zero to stack

    while (we still need to visit vertices):
        current = stack.pop()
        add current to both visited and output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output

Question: Can we improve this algorithm even more?

Answer: Why do we need the visited set?
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    output = new ArrayList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary
    also add all nodes with indegree zero to stack

    while (we still need to visit vertices):
        current = stack.pop()
        add current to output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    output = new ArrayList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary
    also add all nodes with indegree zero to stack

    while (we still need to visit vertices):
        current = stack.pop()
        add current to output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output

Question: What’s the worst-case runtime now?
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    output = new ArrayList<Vertex>()
    stack = new Stack<Vertex>();

    compute all indegrees and add to dictionary
    also add all nodes with indegree zero to stack

    while (we still need to visit vertices):
        current = stack.pop()
        add current to output

        for (v : current.allNeighbors()):
            indegrees[v] -= 1
            if (indegrees[v] == 0):
                stack.push(v)

    return output

Question: What’s the worst-case runtime now?

Answer: \( O(|V| + |E|) \)
And now, for something completely different...
Minimum spanning trees

Punchline: a MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

<table>
<thead>
<tr>
<th>Minimum spanning trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a connected, undirected graph $G = (V, E)$, a <strong>minimum spanning tree</strong> is a subgraph $G' = (V', E')$ such that...</td>
</tr>
</tbody>
</table>
Minimum spanning trees

Punchline: a MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

Minimum spanning trees

Given a connected, undirected graph \( G = (V, E) \), a **minimum spanning tree** is a subgraph \( G' = (V', E') \) such that...

- \( V = V' \) (\( G' \) is spanning)
Minimum spanning trees

Punchline: a MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

<table>
<thead>
<tr>
<th>Minimum spanning trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a connected, undirected graph $G = (V, E)$, a <strong>minimum spanning tree</strong> is a subgraph $G' = (V', E')$ such that...</td>
</tr>
<tr>
<td>▶ $V = V'$ ($G'$ is spanning)</td>
</tr>
<tr>
<td>▶ There exists a path from any vertex to any other one</td>
</tr>
</tbody>
</table>
Minimum spanning trees

Punchline: a MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

Minimum spanning trees

Given a connected, undirected graph $G = (V, E)$, a **minimum spanning tree** is a subgraph $G' = (V', E')$ such that...

- $V = V'$ (G' is spanning)
- There exists a path from any vertex to any other one
- The sum of the edge weights in $E'$ is minimized.
Punchline: a MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

**Minimum spanning trees**

Given a connected, undirected graph $G = (V, E)$, a **minimum spanning tree** is a subgraph $G' = (V', E')$ such that...

- $V = V'$ (*$G'$ is spanning*)
- There exists a path from any vertex to any other one
- The sum of the edge weights in $E'$ is *minimized*.

In order for a graph to have a MST, the graph must...

- ...be connected – there is a path from a vertex to any other vertex. (Note: this means $|V| \leq |E|$).
- ...be undirected.
An example of a minimum spanning tree (MST):
Minimum spanning trees: Applications

Example questions:

- We want to connect phone lines to houses, but laying down cable is expensive. How can we minimize the amount of wire we must install?

Other applications:
- Implement efficient multiple constant multiplication
- Minimizing number of packets transmitted across a network
- Machine learning (e.g. real-time face verification)
- Graphics (e.g. image segmentation)
Minimum spanning trees: Applications

Example questions:

- We want to connect phone lines to houses, but laying down cable is expensive. How can we minimize the amount of wire we must install?
- We have items on a circuit we want to be “electrically equivalent”. How can we connect them together using a minimum amount of wire?
Minimum spanning trees: Applications

Example questions:

- We want to connect phone lines to houses, but laying down cable is expensive. How can we minimize the amount of wire we must install?
- We have items on a circuit we want to be “electrically equivalent”. How can we connect them together using a minimum amount of wire?

Other applications:

- Implement efficient multiple constant multiplication
- Minimizing number of packets transmitted across a network
- Machine learning (e.g. real-time face verification)
- Graphics (e.g. image segmentation)
Minimum spanning trees: Applications

Example questions:

▶ We want to connect phone lines to houses, but laying down cable is expensive. How can we minimize the amount of wire we must install?
▶ We have items on a circuit we want to be “electrically equivalent”. How can we connect them together using a minimum amount of wire?

Other applications:

▶ Implement efficient multiple constant multiplication
▶ Minimizing number of packets transmitted across a network
▶ Machine learning (e.g. real-time face verification)
▶ Graphics (e.g. image segmentation)
Some questions...

- Can a valid MST contain a cycle?

- If we take a valid MST and remove an edge, is it still an MST?

- If we take a valid MST and add an edge, is it still an MST?

- If there are $V$ vertices, how many edges are contained in the minimum spanning tree?
Some questions...

- Can a valid MST contain a cycle?
  Answer: no. If there’s a cycle, we can always remove one edge to break the cycle while still leaving all nodes connected.

- If we take a valid MST and remove an edge, is it still an MST?
  Answer: No. If we’re already using the fewest edges possible, removing an edge would make the nodes no longer connected.

- If we take a valid MST and add an edge, is it still an MST?
  Answer: No. Since all the edges are already connected, this would introduce a cycle.

- If there are $V$ vertices, how many edges are contained in the minimum spanning tree?
  Answer: $|V| - 1$
Design question: how would you implement an algorithm to find the MST of some graph, assuming the edges all have the same weight?
Design question: how would you implement an algorithm to find the MST of some graph, assuming the edges all have the same weight?

One idea: run DFS, and keep all the edges that don’t connect back to an already-visited vertex.

Another idea: iterate through the edges, and add an edge as long as it doesn’t introduce a cycle.
Next time:

How do we account for edge weights?

- **Prim’s algorithm:** Traverse through graph, and add nodes
Minimum spanning tree: coming up next

Next time:

How do we account for edge weights?

- **Prim’s algorithm**: Traverse through graph, and add nodes
- **Kruskal’s algorithm**: Iterate through edges, and add edges
Next time:

How do we account for edge weights?

- **Prim’s algorithm**: Traverse through graph, and add nodes
- **Kruskal’s algorithm**: Iterate through edges, and add edges

In both cases, we avoid adding nodes/edges that introduce a cycle, and need to figure out how to pick the “best” node or edge.