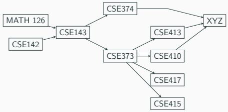
CSE 373: Topological Sort and Minimum Spanning Trees

Michael Lee
Friday, Feb 23, 2018

Topological sort

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



Goal: list out classes in a "valid" order

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

Topological sort

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?
  No: how do we decide which node comes first?
- Is there always one unique output per graph?
  No: see example on inked slides

Topological sort: algorithm

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

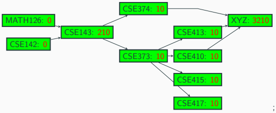
- Look at each vertex and record its in-degree somewhere

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:

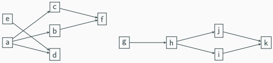


;

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

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:

```
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?

$\mathcal{O}\left(|V|^2 + |E|\right)$

Is this optimal?

Maybe not. Do we really need to look at each node multiple times? Can we somehow get $\mathcal{O}\left(|V| + |E|\right)$?

Topological sort: Algorithm

```
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?

Topological sort: Algorithm 2

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!

```
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
```

Topological sort: Algorithm 2

```
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
```

Question: Does this actually work?

Answer: No, there's a bug! The stack is initially empty, so first pop fails.

Topological sort: Algorithm 2

```
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?

Topological sort: Algorithm 2

```
def toposort(graph):
    indegrees = new HashMap<Vertex, Integer>()
    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 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: $\mathcal{O}(|V| + |E|)$

Minimum spanning trees

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).

Minimum spanning trees

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

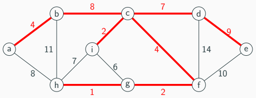
- $V = V'$ (G' is *spanning*)
- There exists a path from any vertex to any other one
- The sum of the edge weights in E' is *minimized.*

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.

Minimum spanning trees: example

An example of an 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?
- 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: properties

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: $|V| - 1$

Minimum spanning trees: algorithm

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.

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

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.