CSE 373: Data Structures & Algorithms
Lecture 16: Topological Sort / Graph Traversals

Nicki Dell
Spring 2014
Midterm

• This Friday in class
• Closed books, closed notes
• Practice midterms posted online
Graphs

- A graph is a formalism for representing relationships among items
  - Very general definition because very general concept

- A graph is a pair
  \[ G = (V, E) \]
  - A set of vertices, also known as nodes
    \[ V = \{v_1, v_2, \ldots, v_n\} \]
  - A set of edges
    \[ E = \{e_1, e_2, \ldots, e_m\} \]
    - Each edge \( e_i \) is a pair of vertices \( (v_j, v_k) \)
    - An edge “connects” the vertices

- Graphs can be directed or undirected
Density / Sparsity

- Recall: In an undirected graph, $0 \leq |E| < |V|^2$
- Recall: In a directed graph: $0 \leq |E| \leq |V|^2$
- So for any graph, $O(|E|+|V|^2)$ is $O(|V|^2)$
- Another fact: If an undirected graph is connected, then $|V|-1 \leq |E|$
- Because $|E|$ is often much smaller than its maximum size, we do not always approximate $|E|$ as $O(|V|^2)$
  - This is a correct bound, it just is often not tight
  - If it is tight, i.e., $|E|$ is $\Theta(|V|^2)$ we say the graph is dense
    - More sloppily, dense means “lots of edges”
  - If $|E|$ is $O(|V|)$ we say the graph is sparse
    - More sloppily, sparse means “most possible edges missing”
What is the Data Structure?

• So graphs are really useful for lots of data and questions
  – For example, “what’s the lowest-cost path from x to y”

• But we need a data structure that represents graphs

• The “best one” can depend on:
  – Properties of the graph (e.g., dense versus sparse)
  – The common queries (e.g., “is \((u, v)\) an edge?” versus “what are the neighbors of node \(u\)?”)

• So we’ll discuss the two standard graph representations
  – Adjacency Matrix and Adjacency List
  – Different trade-offs, particularly time versus space
Adjacency Matrix

- Assign each node a number from 0 to $|V| - 1$
- A $|V| \times |V|$ matrix (i.e., 2-D array) of Booleans (or 1 vs. 0)
  - If $M$ is the matrix, then $M[u][v]$ being true means there is an edge from $u$ to $v$
Adjacency Matrix Properties

- Running time to:
  - Get a vertex’s out-edges: $O(|V|)$
  - Get a vertex’s in-edges: $O(|V|)$
  - Decide if some edge exists: $O(1)$
  - Insert an edge: $O(1)$
  - Delete an edge: $O(1)$

- Space requirements:
  - $|V|^2$ bits

- Best for sparse or dense graphs?
  - Best for dense graphs
Adjacency Matrix Properties

• How will the adjacency matrix vary for an _undirected graph_?
  – Undirected will be symmetric around the diagonal

• How can we adapt the representation for _weighted graphs_?
  – Instead of a Boolean, store a number in each cell
  – Need some value to represent ‘not an edge’
    • In _some_ situations, 0 or -1 works
Adjacency List

- Assign each node a number from 0 to $|V| - 1$
- An array of length $|V|$ in which each entry stores a list of all adjacent vertices (e.g., linked list)
Adjacency List Properties

• Running time to:
  – Get all of a vertex’s out-edges: \( O(d) \) where \( d \) is out-degree of vertex
  – Get all of a vertex’s in-edges: \( O(|E|) \) (but could keep a second adjacency list for this!)
  – Decide if some edge exists:
    \( O(d) \) where \( d \) is out-degree of source
  – Insert an edge:
    \( O(1) \) (unless you need to check if it’s there)
  – Delete an edge:
    \( O(d) \) where \( d \) is out-degree of source

• Space requirements: \( O(|V|+|E|) \)
  • Good for sparse graphs
Algorithms

Okay, we can represent graphs

Now we’ll implement some useful and non-trivial algorithms

- **Topological sort**: Given a DAG, order all the vertices so that every vertex comes before all of its neighbors

- **Shortest paths**: Find the shortest or lowest-cost path from x to y
  - Related: Determine if there even is such a path
**Topological Sort**

Problem: Given a DAG $G = (V, E)$, output all vertices in an order such that no vertex appears before another vertex that has an edge to it.

One example output:

126, 142, 143, 374, 373, 417, 410, 413, XYZ, 415
Questions and comments

• Why do we perform topological sorts only on DAGs?
  – Because a cycle means there is no correct answer

• Is there always a unique answer?
  – No, there can be 1 or more answers; depends on the graph

• Do some DAGs have exactly 1 answer?
  – Yes, including all lists

• Terminology: A DAG represents a partial order and a topological sort produces a total order that is consistent with it
Uses

• Figuring out how to graduate

• Computing an order in which to recompute cells in a spreadsheet

• Determining an order to compile files using a Makefile

• In general, taking a dependency graph and finding an order of execution

• …
A First Algorithm for Topological Sort

1. Label (“mark”) each vertex with its in-degree
   - Think “write in a field in the vertex”
   - Could also do this via a data structure (e.g., array) on the side

2. While there are vertices not yet output:
   a) Choose a vertex $v$ with labeled with in-degree of 0
   b) Output $v$ and conceptually remove it from the graph
   c) For each vertex $u$ adjacent to $v$ (i.e. $u$ such that $(v,u)$ in $E$),
      decrement the in-degree of $u$
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed?

In-degree: 0 0 2 1 1 1 1 1 1 3
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed?: x
In-degree: 0 0 2 1 1 1 1 1 1 3 1

Output: 126
Example

Output: 126 142

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed? x x

In-degree: 0 0 2 1 1 1 1 1 1 1 3

1
0
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x
In-degree: 0 0 2 1 1 1 1 1 1 1 3

Output: 126 142 143
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x
In-degree: 0 0 2 1 1 1 1 1 1 1

Output:
126
142
143
374
**Example**

Node: 126 142 143 374 373 410 413 415 417 XYZ

Removed?: x x x x x x

In-degree: 0 0 2 1 1 1 1 1 1 1 1 3

Spring 2014  
CSE373: Data Structures & Algorithms
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x x x x x
In-degree: 0 0 2 1 1 1 1 1 1 1 1 1 1 1 3

Output: 126 142 143 374 373 410 413 415 417 XYZ
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x x x x x x
In-degree: 0 0 2 1 1 1 1 1 1 1 3
            1 0 0 0 0 0 0 0 0 2
            0 1

Output: 126 142 143 374 373 410 417 410
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x x x x x x x
In-degree: 0 0 2 1 1 1 1 1 1 1 1 3

Output: 126 142 143 374 373 410 413 415 417 XYZ
Example

Node: 126 142 143 374 373 410 413 415 417 XYZ
Removed? x x x x x x x x x x x x x
In-degree: 0 0 2 1 1 1 1 1 1 1 1 3

Output:
126
142
143
374
373
410
413
417
XYZ
Example

Output:
126
142
143
374
373
410
413
415
417
XYZ

Node:          126 142  143  374  373  410  413  415  417  XYZ
Removed?   x   x   x   x   x   x   x   x   x   x
In-degree:    0   0   2   1   1   1   1   1   1   3
             1   0   0   0   0   0   0   0   2
             0   0   0   0   0   0   0   0   1
             0   0   0   0   0   0   0   0   0
Notice

- Needed a vertex with in-degree 0 to start
  - Will always have at least 1 because no cycles

- Ties among vertices with in-degrees of 0 can be broken arbitrarily
  - Can be more than one correct answer, by definition, depending on the graph
Running time?

```java
labelEachVertexWithItsInDegree();
for(ctr=0; ctr < numVertices; ctr++){
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```

• What is the worst-case running time?
  – Initialization $O(|V|+|E|)$ (assuming adjacency list)
  – Sum of all find-new-vertex $O(|V|^2)$ (because each $O(|V|)$)
  – Sum of all decrements $O(|E|)$ (assuming adjacency list)
  – So total is $O(|V|^2)$ – not good for a sparse graph!
**Doing better**

The trick is to avoid searching for a zero-degree node every time!
- Keep the “pending” zero-degree nodes in a list, stack, queue, bag, table, or something
- Order we process them affects output but not correctness or efficiency provided add/remove are both \(O(1)\)

Using a queue:

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
   a) \(\mathbf{v} = \text{dequeue}()\)
   b) Output \(\mathbf{v}\) and remove it from the graph
   c) For each vertex \(\mathbf{u}\) adjacent to \(\mathbf{v}\) (i.e. \(\mathbf{u}\) such that \((\mathbf{v}, \mathbf{u}) \in \mathbf{E}\)), decrement the in-degree of \(\mathbf{u}\), if new degree is 0, enqueue it
Running time?

```java
labelAllAndEnqueueZeros();
for(ctr=0; ctr < numVertices; ctr++){
    v = dequeue();
    put v next in output
    for each w adjacent to v {
        w.indegree--;
        if(w.indegree==0)
            enqueue(v);
    }
}
```

- What is the worst-case running time?
  - Initialization: $O(|V| + |E|)$ (assuming adjacency list)
  - Sum of all enqueues and dequeues: $O(|V|)$
  - Sum of all decrements: $O(|E|)$ (assuming adjacency list)
  - So total is $O(|E| + |V|)$ – much better for sparse graph!
Graph Traversals

Next problem: For an arbitrary graph and a starting node $v$, find all nodes reachable from $v$ (i.e., there exists a path from $v$)
- Possibly “do something” for each node
- Examples: print to output, set a field, etc.

- Subsumed problem: Is an undirected graph connected?
- Related but different problem: Is a directed graph strongly connected?
  - Need cycles back to starting node

Basic idea:
- Keep following nodes
- But “mark” nodes after visiting them, so the traversal terminates and processes each reachable node exactly once
Abstract Idea

```java
traverseGraph(Node start) {
    Set pending = emptySet()
    pending.add(start)
    mark start as visited
    while (pending is not empty) {
        next = pending.remove()
        for each node u adjacent to next
            if (u is not marked) {
                mark u
                pending.add(u)
            }
    }
}
```
Running Time and Options

- Assuming `add` and `remove` are $O(1)$, entire traversal is $O(|E|)$
  - Use an adjacency list representation

- The order we traverse depends entirely on `add` and `remove`
  - Popular choice: a stack “depth-first graph search” “DFS”
  - Popular choice: a queue “breadth-first graph search” “BFS”

- DFS and BFS are “big ideas” in computer science
  - Depth: recursively explore one part before going back to the other parts not yet explored
  - Breadth: explore areas closer to the start node first
Example: Depth First Search

- A tree is a graph and DFS and BFS are particularly easy to “see”

DFS(Node start) {
  mark and process start
  for each node u adjacent to start
    if u is not marked
      DFS(u)
}

- A B D E C F G H
- Exactly what we called a “pre-order traversal” for trees
  - The marking is because we support arbitrary graphs and we want to process each node exactly once
Example: Another Depth First Search

- A tree is a graph and DFS and BFS are particularly easy to “see”

```
DFS2(Node start) {
    initialize stack s and push start
    mark start as visited
    while(s is not empty) {
        next = s.pop() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
    }
}
```

- A different but perfectly fine traversal

A C F H G B E D
Example: Breadth First Search

- A tree is a graph and DFS and BFS are particularly easy to “see”

BFS(Node start) {
    initialize queue q and enqueue start
    mark start as visited
    while(q is not empty) {
        next = q.dequeue() // and “process”
        for each node u adjacent to next
            if(u is not marked)
                mark u and enqueue onto q
    }
}

- A B C D E F G H
- A “level-order” traversal
Comparison

• Breadth-first always finds shortest paths, i.e., “optimal solutions”
  – Better for “what is the shortest path from \( x \) to \( y \)”

• But depth-first can use less space in finding a path
  – If longest path in the graph is \( p \) and highest out-degree is \( d \)
    then DFS stack never has more than \( d \times p \) elements
  – But a queue for BFS may hold \( O(|V|) \) nodes

• A third approach:
  – Iterative deepening (IDFS):
    • Try DFS but disallow recursion more than \( K \) levels deep
    • If that fails, increment \( K \) and start the entire search over
  – Like BFS, finds shortest paths. Like DFS, less space.
Saving the Path

- Our graph traversals can answer the reachability question:
  - “Is there a path from node x to node y?”

- But what if we want to actually output the path?
  - Like getting driving directions rather than just knowing it’s possible to get there!

- How to do it:
  - Instead of just “marking” a node, store the previous node along the path (when processing u causes us to add v to the search, set v.path field to be u)
  - When you reach the goal, follow path fields back to where you started (and then reverse the answer)
  - If just wanted path length, could put the integer distance at each node instead
Example using BFS

What is a path from Seattle to Tyler

- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique
Single source shortest paths

• Done: BFS to find the minimum path length from \( v \) to \( u \) in \( O(|E|+|V|) \)

• Actually, can find the minimum path length from \( v \) to every node
  – Still \( O(|E|+|V|) \)
  – No faster way for a “distinguished” destination in the worst-case

• Now: Weighted graphs

  Given a weighted graph and node \( v \),
  find the minimum-cost path from \( v \) to every node

• As before, asymptotically no harder than for one destination
Applications

- Driving directions
- Cheap flight itineraries
- Network routing
- Critical paths in project management
Why BFS won’t work: Shortest path may not have the fewest edges
  – Annoying when this happens with costs of flights

We will assume there are no negative weights

- **Problem** is *ill-defined* if there are negative-cost *cycles*
- **Today’s algorithm** is *wrong* if edges can be negative
  – There are other, slower (but not terrible) algorithms
Dijkstra’s Algorithm

- Named after its inventor Edsger Dijkstra (1930-2002)
  - Truly one of the “founders” of computer science; this is just one of his many contributions
  - Many people have a favorite Dijkstra story, even if they never met him

Computer science is no more about computers than astronomy is about telescopes.

(Edsger Dijkstra)
Dijkstra’s Algorithm

• The idea: reminiscent of BFS, but adapted to handle weights
  – Grow the set of nodes whose shortest distance has been computed
  – Nodes not in the set will have a “best distance so far”
  – A priority queue will turn out to be useful for efficiency
• An example of a greedy algorithm
  – A series of steps
  – At each one the locally optimal choice is made