Announcements

- Catie will be teaching on Friday and possibly Monday
- Homework 4 due on Monday
Graphs

- A graph $G = (V, E)$
  - represents relationships among items
  - can be directed or undirected
- For any graph, complexity is $O(|E| + |V|)$ is $O(|V|^2)$
  - undirected graph, $0 \leq |E| < |V|^2$
  - directed graph, $0 \leq |E| \leq |V|^2$
  - Can be sparse
    - $|E|$ is $O(|V|)$
  - Can be dense
    - $|E|$ is $\Theta(|V|^2)$

$V = \{\text{Han, Leia, Luke}\}$
$E = \{(\text{Luke, Leia}),\ (\text{Han, Leia}),\ (\text{Leia, Han})\}$
What is the Data Structure?

• 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 \(u\)?”)

• 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!)
  - 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, output all vertices in an order so that no vertex appears before another vertex that points 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
   – Could "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 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

Output:

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

Removed?

In-degree: 0 0 2 1 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 1 3

Output: 126
Example

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

Output:
126
142
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 3
          1 0 0
          0

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 3
           1 0 0
           0

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

Output: 126 142 143 374 373
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 3
          0 0 0 0 0 0 0 0 0 0 2
          0

Output: 126 142 143 374 373 417 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 3
           1 0 0 0 0 0 0 0 0 2
           0 1

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

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

Output:  126 142 143 374 373 410 413 415 417  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 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 0 0 0 0 0 0 0 0 1
          0 0 0 0 0 0 0 0 0 0
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
In-degree: 0 0 2 1 1 1 1 1 1 3

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

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!

```java
labelEachVertexWithItsInDegree();
for(ctr=0; ctr < numVertices; ctr++){
  v = findNewVertexOfDegreeZero();
  put v next in output
  for each w adjacent to v
    w.indegree--;
}
```
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) $v = \text{enqueue}()$
   b) Output $v$ and 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 $u$, 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)
  - Total: $O(|E| + |V|)$ – much better for sparse graph!
Graph Traversals

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

Can we use this to answer:
- Is an undirected graph connected?
- Is a directed graph strongly connected?

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

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
  - stack “depth-first search” “DFS”
  - queue “breadth-first search” “BFS”

- DFS and BFS
  - 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
Example: Depth First Search

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

```java
DFS(Node start) {
    mark and process start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```
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
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
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
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
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
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
Example: Another Depth First Search

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

```java
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
    }
}
```
Example: Breadth First Search

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

```java
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
Example: Breadth First Search

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

```java
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
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
Example: Breadth First Search

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

```java
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
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
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
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 finds shortest paths
  - Better for “what is the shortest path from \( x \) to \( y \)”

- But depth-first can use less space in finding a path

- 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?

• How to do it:
  – Instead of just “marking” a node, store the previous node along the path
  – 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
Shortest Path using BFS

What is shortest path from Seattle to Tyler?
- Remember marked nodes are not re-enqueued
- May not be unique
Single source shortest paths

- Found the minimum path length from $v$ to $u$ in $O(|E|+|V|)$ using BFS.
- Actually, can find the minimum path length from $v$ to every node.
  - Still $O(|E|+|V|)$
- 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
Can we use BFS?

Why BFS won’t work: Lowest cost path may not have the fewest edges

We will assume there are no negative weights

- **Problem** is *ill-defined* if there are negative-cost *cycles*
- **Our algorithm** is *wrong* if *edges* can be negative
  - There are other, slower (but not terrible) algorithms
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”
  - Will use a priority queue
- An example of a greedy algorithm
  - A series of steps
  - At each one the locally optimal choice is made