CSE 332: Data Abstractions

Lecture 14: Introduction to Graphs

Ruth Anderson
Autumn 2013
Announcements

• **Homework 4** – due Friday Nov 1st at the BEGINNING of lecture
• **Project 2** – Phase B due Wed Nov 6th 11pm
Today

• Graphs
  – Intro & Definitions
Where We Are

We have learned about the essential ADTs and data structures:

- Regular and Circular Arrays (dynamic sizing)
- Linked Lists
- Stacks, Queues
- Priority Queues, Heaps
- Unbalanced and Balanced Search Trees, B-Trees
- Hash Tables

We have also learned important algorithms

- Tree traversals
- Floyd's buildheap Method
- Sorting algorithms
Where We Are Going

More on algorithms and related problems that require constructing data structures to make the solutions efficient

Topics will include:
- Graphs
- Parallelism
- Concurrency
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
An ADT?

- Can think of graphs as an ADT with operations like `isEdge((v_j, v_k))`
- But it is unclear what the “standard operations” are
- Instead we tend to develop algorithms over graphs and then use data structures that are efficient for those algorithms
- Many important problems can be solved by:
  1. Formulating them in terms of graphs
  2. Applying a standard graph algorithm
- To make the formulation easy and standard, we have a lot of *standard terminology* about graphs
Some graphs

For each, what are the vertices and what are the edges?

- Web pages with links
- Facebook friends
- “Input data” for the Kevin Bacon game
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Family trees
- Course pre-requisites
- ...

Wow: Using the same algorithms for problems across so many domains sounds like “core computer science and engineering”
Undirected Graphs

- In undirected graphs, edges have no specific direction
  - Edges are always “two-way”

- Thus, \((u, v) \in E\) implies \((v, u) \in E\).
  - Only one of these edges needs to be in the set; the other is implicit

- Degree of a vertex: number of edges containing that vertex
  - Put another way: the number of adjacent vertices
Directed Graphs

• In directed graphs (sometimes called digraphs), edges have a direction.

Thus, \((u, v) \in E\) does not imply \((v, u) \in E\).

• Let \((u, v) \in E\) mean \(u \rightarrow v\)
• Call \(u\) the source and \(v\) the destination.

• In-Degree of a vertex: number of in-bound edges, i.e., edges where the vertex is the destination.
• Out-Degree of a vertex: number of out-bound edges, i.e., edges where the vertex is the source.
Self-edges, connectedness

- A **self-edge** a.k.a. a **loop** is an edge of the form \((u, u)\)
  - Depending on the use/algorithm, a graph may have:
    - No self edges
    - Some self edges
    - All self edges (often therefore implicit, but we will be explicit)

- A node can have a degree / in-degree / out-degree of **zero**

- A graph does not have to be **connected** (In an undirected graph, this means we can follow edges from any node to every other node), even if every node has non-zero degree
More notation

For a graph $G = (V, E)$:

- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?

- If $(u, v) \in E$
  - Then $v$ is a neighbor of $u$, i.e., $v$ is adjacent to $u$
  - Order matters for directed edges
    - $u$ is not adjacent to $v$ unless $(v, u) \in E$
More notation

For a graph $G = (V,E)$:

- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum? $0$
  - Maximum for undirected? $|V| \cdot (|V|+1)/2 \in O(|V|^2)$
  - Maximum for directed? $|V|^2 \in O(|V|^2)$
    (assuming self-edges allowed, else subtract $|V|$)

- If $(u,v) \in E$
  - Then $v$ is a neighbor of $u$, i.e., $v$ is adjacent to $u$
  - Order matters for directed edges
    - $u$ is not adjacent to $v$ unless $(v,u) \in E$
Examples again

Which would use directed edges? Which would have self-edges? Which could have 0-degree nodes?

- Web pages with links
- Facebook friends
- “Input data” for the Kevin Bacon game
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Family trees
- Course pre-requisites
- ...
**Weighted graphs**

- In a weighed graph, each edge has a *weight* a.k.a. *cost*
  - Typically numeric (most examples will use ints)
  - *Orthogonal* to whether graph is directed
  - Some graphs allow *negative weights*; many don’t

![Diagram of weighted graph]

- **Clinton** → **Mukilteo**: 20
- **Kingston** → **Edmonds**: 30
- **Bainbridge** → **Seattle**: 35
- **Bremerton** → **Seattle**: 60
Examples

What, if anything, might weights represent for each of these? Do negative weights make sense?

• Web pages with links
• Facebook friends
• “Input data” for the Kevin Bacon game
• Methods in a program that call each other
• Road maps (e.g., Google maps)
• Airline routes
• Family trees
• Course pre-requisites
• …
Paths and Cycles

- A path is a list of vertices \([v_0, v_1, \ldots, v_n]\) such that \((v_i, v_{i+1}) \in E\) for all \(0 \leq i < n\). Say “a path from \(v_0\) to \(v_n\)”

- A cycle is a path that begins and ends at the same node \((v_0=\equiv v_n)\)

Example path (that also happens to be a cycle):
[Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]
Path Length and Cost

- Path length: Number of edges in a path (also called “unweighted cost”)
- Path cost: Sum of the weights of each edge

Example where:

\[ P = \text{[Seattle, Salt Lake City, Chicago, Dallas, San Francisco]} \]

\[
\text{length}(P) = 4 \\
\text{cost}(P) = 9.5
\]
Simple paths and cycles

• A simple path repeats no vertices, (except the first might be the last):
  [Seattle, Salt Lake City, San Francisco, Dallas]
  [Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

• Recall, a cycle is a path that ends where it begins:
  [Seattle, Salt Lake City, San Francisco, Dallas, Seattle]
  [Seattle, Salt Lake City, Seattle, Dallas, Seattle]

• A simple cycle is a cycle and a simple path:
  [Seattle, Salt Lake City, San Francisco, Dallas, Seattle]
Paths/cycles in directed graphs

Example:

Is there a path from A to D?

Does the graph contain any cycles?
Paths/cycles in directed graphs

Example:

Is there a path from A to D? No

Does the graph contain any cycles? No
**Undirected graph connectivity**

- An undirected graph is **connected** if for all pairs of vertices \( u, v \), there exists a *path* from \( u \) to \( v \)

![Connected graph](image1)

![Disconnected graph](image2)

- An undirected graph is **complete**, a.k.a. **fully connected** if for all pairs of vertices \( u, v \), there exists an *edge* from \( u \) to \( v \)

![Complete graph](image3)

*(plus self edges)*
**Directed graph connectivity**

- A directed graph is **strongly connected** if there is a path from every vertex to every other vertex.

- A directed graph is **weakly connected** if there is a path from every vertex to every other vertex *ignoring direction of edges*.

- A **complete** a.k.a. **fully connected** directed graph has an edge from every vertex to every other vertex *plus self edges*.
Examples

For undirected graphs: connected?
For directed graphs: strongly connected? weakly connected?

- Web pages with links
- Facebook friends
- “Input data” for the Kevin Bacon game
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Family trees
- Course pre-requisites
- ...

10/30/2013
Trees as graphs

When talking about graphs, we say a tree is a graph that is:
- undirected
- acyclic
- connected

So all trees are graphs, but not all graphs are trees

How does this relate to the trees we know and love?...
Rooted Trees

• We are more accustomed to rooted trees where:
  – We identify a unique (“special”) root
  – We think of edges as directed: parent to children

• Given a tree, once you pick a root, you have a unique rooted tree (just drawn differently and with undirected edges)
Rooted Trees (Another example)

• We are more accustomed to rooted trees where:
  – We identify a unique (“special”) root
  – We think of edges as **directed**: parent to children

• Given a tree, once you pick a root, you have a unique rooted tree (just drawn differently and with undirected edges)
Directed acyclic graphs (DAGs)

- A **DAG** is a directed graph with no (directed) cycles
  - Every rooted directed tree is a DAG
    - But not every DAG is a rooted directed tree:

- Every DAG is a directed graph
  - But not every directed graph is a DAG:
Examples

Which of our directed-graph examples do you expect to be a DAG?

• Web pages with links
• “Input data” for the Kevin Bacon game
• Methods in a program that call each other
• Airline routes
• Family trees
• Course pre-requisites
• …
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, \( |E| \) is \( O(|V|^2) \)
- One more fact: If an undirected graph is connected, then \( |E| \geq |V| - 1 \)
- Because \( |E| \) is often much smaller than its maximum size, we do not always approximate as \( |E| \) 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 $M[u][v] == \text{true}$ means there is an edge from $u$ to $v$
## Adjacency Matrix Properties

- **Running time to:**
  - Get a vertex’s out-edges:
  - Get a vertex’s in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:

- **Space requirements:**

- **Best for sparse or dense graphs?**

### Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>B</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
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?

- How can we adapt the representation for weighted graphs?
Adjacency Matrix Properties

• How will the adjacency matrix vary for an undirected graph?
  – Undirected will be symmetric about diagonal axis

• 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:
  - Get all of a vertex’s in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:

- Space requirements:
  - Best for dense or sparse graphs?
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:
  – $O(|V|+|E|)$

• Best for dense or sparse graphs?
  – Best for sparse graphs, so usually just stick with linked lists
Undirected Graphs

Adjacency matrices & adjacency lists both do fine for undirected graphs

• Matrix: Can save roughly ½ the space
  – But may slow down operations in languages with “proper” 2D arrays (not Java, which has only arrays of arrays)
  – How would you “get all neighbors”?
• Lists: Each edge in two lists to support efficient “get all neighbors”

Example:

```
    A   B   C   D
A  F   T   F   F
B  T   F   T   F
C  F   T   F   T
D  F   F   T   F
```
Which is better?

Graphs are often sparse:

- Streets form grids
  - every corner is not connected to every other corner
- Airlines rarely fly to all possible cities
  - or if they do it is to/from a hub rather than directly to/from all small cities to other small cities

Adjacency lists should generally be your default choice

- Slower performance compensated by greater space savings
Next…

Okay, we can represent graphs

Now let’s 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