CSE 373: Data Structures & Algorithms

Introduction to Graphs

Riley Porter
Winter 2017
Announcements

• Midterms done!
  – Wow! Nicely done everyone.
    • Average was ~68/80, which is ~85%
    • Standard Dev: 7 points
    • Not an easy test, you all rocked it! Congrats!
  – Handed back in section on Thursday
  – Scores on Canvas after lecture

• HW4 out tonight -> Graphs
Graphs

• A graph is a formalism for representing relationships among items. One way to write graphs:

• A graph $G = (V, E)$
  – 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

V = \{Han, Leia, Luke\}
E = \{(Luke, Leia),
      (Han, Leia),
      (Leia, Han)\}
Are Graphs An ADT?

• Can think of graphs as an ADT with operations like $\text{isEdge}((v_j, v_k))$, $\text{addVertex}(v_{\text{new}})$, ...

• 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
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, so normalize how you check for it

• 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**
  – 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 (assuming no self loops)
  - Minimum?
  - Maximum for directed? $|V| \times (|V| - 1) \in O(|V|^2)$
  - Maximum for undirected? $(|V| \times (|V| - 1)) / 2 \in O(|V|^2)$

- 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 $(v, u) \in E$

$V = \{A, B, C, D\}$
$E = \{(C, B), (A, B), (B, A), (C, D)\}$
Weighted Graphs

• In a weighed graph, each edge has a weight a.k.a. cost
  – Typically numeric (most examples use ints)
  – Orthogonal to whether graph is directed
  – Some graphs allow negative weights; many do not
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 (\(v_0 == v_n\))

Path: [Seattle, Chicago, Dallas]
Cycle: [Seattle, Salt Lake City, Dallas, San Francisco, Seattle]
Path Length and Cost

• **Path length**: Number of *edges* in a path
• **Path cost**: Sum of *weights* of edges in a path

Example:

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

\[
\begin{align*}
\text{length}(P) &= 5 \\
\text{cost}(P) &= 11.5
\end{align*}
\]
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 and Cycles in Directed Graphs

Example:

Is there a path from A to D?

Does the graph contain any cycles?
Paths and Cycles in Directed Graphs

Example:

Is there a path from A to D?   No

Does the graph contain any cycles?   No
Paths and Cycles in Directed Graphs

Example:

Is there a path from A to D?

Does the graph contain any cycles?
Paths and Cycles in Directed Graphs

Example:

Is there a path from A to D?  Yes

Does the graph contain any cycles?  No
Paths and Cycles in Directed Graphs

Example:

Is there a path from A to D?

Does the graph contain any cycles?
Paths and Cycles in Directed Graphs

Example:

Is there a path from A to D?  Yes

Does the graph contain any cycles? Yes
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](image)

![Disconnected graph](image)

• 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](image)

*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**.
When talking about graphs, we say a **tree** is a graph that is:

- Acyclic (no cycles)
- Connected

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

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

• Given a graph that is a tree, picking a root gives a unique rooted tree
Rooted Trees

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

• Given a graph that is a tree, picking a root gives a unique rooted tree
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

  ![Diagram of a rooted directed tree]

- Not every directed graph is acyclic

  ![Diagram of a cyclic graph]
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 \(O(|V|^2)\)

• Because \(|E|\) is often much smaller than its maximum size, we do not always approximate \(|E|\) as \(O(|V|^2)\)
  – This is a correct upper bound, it just is often not tight
  – If it is tight, i.e., \(|E|\) is \(\Theta(|V|^2)\) we say the graph is dense
  – If \(|E|\) is \(O(|V|)\) we say the graph is sparse
How do we implement this?

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

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

- Assign each vertex/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 \textbf{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

- Better for sparse or dense graphs?
  - Better 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

```
<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>-1</td>
<td>7</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
<td>-1</td>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>4</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>
```
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:
    \( O(|E| + |V|) \) (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:
  - \( O(|V|+|E|) \)

• Better for dense or sparse graphs?
  - Better for sparse graphs
Undirected Graphs

Adjacency matrices & adjacency lists both do fine for undirected graphs

• Matrix: Can save roughly 2x 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:
Applications

- What could we use a graph to represent?

  ALL THE THINGS!
Some Applications as Graphs

For each of the following examples:

- what are the vertices and what are the edges?
- would you use directed edges? Would they have self-edges?
- Are there 0-degree nodes? Is it strongly or weakly connected?
- Does it have weights? Do negative weights make sense?
- Does it have cycles? Is it a DAG?

- Web pages with links
- Facebook friends
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Family trees
- Course pre-requisites
- Political donations to candidates
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