CSCI 104
Graph Representation and Traversals

Mark Redekopp
David Kempe
Sandra Batista
GRAPH REPRESENTATIONS
Graph Notation

- A **graph** is a collection of vertices (or nodes) and edges that connect vertices

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(a,c)</td>
</tr>
<tr>
<td>b</td>
<td>(a,e)</td>
</tr>
<tr>
<td>c</td>
<td>(b,h)</td>
</tr>
<tr>
<td>d</td>
<td>(b,c)</td>
</tr>
<tr>
<td>e</td>
<td>(c,e)</td>
</tr>
<tr>
<td>f</td>
<td>(c,d)</td>
</tr>
<tr>
<td>g</td>
<td>(c,g)</td>
</tr>
<tr>
<td>h</td>
<td>(d,f)</td>
</tr>
<tr>
<td></td>
<td>(e,f)</td>
</tr>
<tr>
<td></td>
<td>(f,g)</td>
</tr>
<tr>
<td></td>
<td>(g,h)</td>
</tr>
</tbody>
</table>

- Let $|V|$ or $n$ refer to the number of vertices
- Let $|E|$ or $m$ refer to the number of edges

$|V|=n=8$  $|E|=m=11$

A vertex
An edge
Graphs in the Real World

- Social networks
- Computer networks / Internet
- Path planning
- Interaction diagrams
- Bioinformatics
Basic Graph Representation

• Can simply store edges in list/array
  – Unsorted
  – Sorted

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(a,c)</td>
</tr>
<tr>
<td>b</td>
<td>(a,e)</td>
</tr>
<tr>
<td>c</td>
<td>(b,h)</td>
</tr>
<tr>
<td>d</td>
<td>(b,c)</td>
</tr>
<tr>
<td>e</td>
<td>(c,e)</td>
</tr>
<tr>
<td>f</td>
<td>(c,d)</td>
</tr>
<tr>
<td>g</td>
<td>(c,g)</td>
</tr>
<tr>
<td>h</td>
<td>(d,f)</td>
</tr>
<tr>
<td></td>
<td>(e,f)</td>
</tr>
<tr>
<td></td>
<td>(f,g)</td>
</tr>
<tr>
<td></td>
<td>(g,h)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(a,c)</td>
</tr>
<tr>
<td>b</td>
<td>(a,e)</td>
</tr>
<tr>
<td>c</td>
<td>(b,h)</td>
</tr>
<tr>
<td>d</td>
<td>(b,c)</td>
</tr>
<tr>
<td>e</td>
<td>(c,e)</td>
</tr>
<tr>
<td>f</td>
<td>(c,d)</td>
</tr>
<tr>
<td>g</td>
<td>(c,g)</td>
</tr>
<tr>
<td>h</td>
<td>(d,f)</td>
</tr>
</tbody>
</table>

$|V|=n=8$  $|E|=m=11$
Graph ADT

• What operations would you want to perform on a graph?
  • `addVertex()` : Vertex
  • `addEdge(v1, v2)`
  • `getAdjacencies(v1) : List<Vertices>`
    – Returns any vertex with an edge from v1 to itself
  • `removeVertex(v)`
  • `removeEdge(v1, v2)`
  • `edgeExists(v1, v2) : bool`

```cpp
#include<iostream>
using namespace std;

template <typename V, typename E>
class Graph{
    // Perfect for templating the data associated with a vertex and edge as V and E
};
```
More Common Graph Representations

- Graphs are really just a list of lists
  - List of vertices each having their own list of adjacent vertices
- Alternatively, sometimes graphs are also represented with an adjacency matrix
  - Entry at \((i,j) = 1\) if there is an edge between vertex \(i\) and \(j\), 0 otherwise

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c,e</td>
</tr>
<tr>
<td>b</td>
<td>c,h</td>
</tr>
<tr>
<td>c</td>
<td>a,b,d,e,g</td>
</tr>
<tr>
<td>d</td>
<td>c,f</td>
</tr>
<tr>
<td>e</td>
<td>a,c,f</td>
</tr>
<tr>
<td>f</td>
<td>d,e,g</td>
</tr>
<tr>
<td>g</td>
<td>c,f,h</td>
</tr>
<tr>
<td>h</td>
<td>b,g</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adjacency Matrix Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
</tbody>
</table>
Graph Representations

- Let $|V| = n = \text{# of vertices}$ and $|E| = m = \text{# of edges}$
- Adjacency List Representation
  - $O(\_\_\_\_\_\_\_\_\_\_\_)$ memory storage
  - Existence of an edge requires searching adjacency list
- Adjacency Matrix Representation
  - $O(\_\_\_\_\_\_\_\_\_\_)$ storage
  - Existence of an edge requires $O(\_\_\_\_\_\_\_\_\_\_)$ lookup

**Adjacency Lists**

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c,e</td>
</tr>
<tr>
<td>b</td>
<td>c,h</td>
</tr>
<tr>
<td>c</td>
<td>a,b,d,e,g</td>
</tr>
<tr>
<td>d</td>
<td>c,f</td>
</tr>
<tr>
<td>e</td>
<td>a,c,f</td>
</tr>
<tr>
<td>f</td>
<td>d,e,g</td>
</tr>
<tr>
<td>g</td>
<td>c,f,h</td>
</tr>
<tr>
<td>h</td>
<td>b,g</td>
</tr>
</tbody>
</table>

**Adjacency Matrix Representation**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Representations

- Let $|V| = n = \# \text{ of vertices}$ and $|E| = m = \# \text{ of edges}$

- **Adjacency List Representation**
  - $O(|V| + |E|)$ memory storage
  - Existence of an edge requires searching adjacency list
  - Define **degree** to be the number of edges incident on a vertex (deg(a) = 2, deg(c) = 5, etc.)

- **Adjacency Matrix Representation**
  - $O(|V|^2)$ storage
  - Existence of an edge requires $O(1)$ lookup (e.g. matrix[i][j] == 1)

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c, e</td>
</tr>
<tr>
<td>b</td>
<td>c, h</td>
</tr>
<tr>
<td>c</td>
<td>a, b, d, e, g</td>
</tr>
<tr>
<td>d</td>
<td>c, f</td>
</tr>
<tr>
<td>e</td>
<td>a, c, f</td>
</tr>
<tr>
<td>f</td>
<td>d, e, g</td>
</tr>
<tr>
<td>g</td>
<td>c, f, h</td>
</tr>
<tr>
<td>h</td>
<td>b, g</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adjacency Matrix Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
</tbody>
</table>
Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a...
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b

List of Vertices

<table>
<thead>
<tr>
<th></th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c,e</td>
</tr>
<tr>
<td>b</td>
<td>c,h</td>
</tr>
<tr>
<td>c</td>
<td>a,b,d,e,g</td>
</tr>
<tr>
<td>d</td>
<td>c,f</td>
</tr>
<tr>
<td>e</td>
<td>a,c,f</td>
</tr>
<tr>
<td>f</td>
<td>d,e,g</td>
</tr>
<tr>
<td>g</td>
<td>c,f,h</td>
</tr>
<tr>
<td>h</td>
<td>b,g</td>
</tr>
</tbody>
</table>

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a...
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b

**Adjacency List Representation**

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c,e</td>
</tr>
<tr>
<td>b</td>
<td>c,h</td>
</tr>
<tr>
<td>c</td>
<td>a,b,d,e,g</td>
</tr>
<tr>
<td>d</td>
<td>c,f</td>
</tr>
<tr>
<td>e</td>
<td>a,c,f</td>
</tr>
<tr>
<td>f</td>
<td>d,e,g</td>
</tr>
<tr>
<td>g</td>
<td>c,f,h</td>
</tr>
<tr>
<td>h</td>
<td>b,g</td>
</tr>
</tbody>
</table>

**Adjacency Matrix Representation**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Directed vs. Undirected Graphs

• In the previous graphs, edges were **undirected** (meaning edges are 'bidirectional' or 'reflexive')
  – An edge \((u,v)\) implies \((v,u)\)
• In **directed** graphs, links are unidirectional
  – An edge \((u,v)\) does not imply \((v,u)\)
  – For Edge \((u,v)\): the **source** is \(u\), **target** is \(v\)
• For adjacency list form, you may need 2 lists per vertex for both predecessors and successors

<table>
<thead>
<tr>
<th>List of Vertices</th>
<th>Adjacency Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c,e</td>
</tr>
<tr>
<td>b</td>
<td>c,h</td>
</tr>
<tr>
<td>c</td>
<td>a,b,d,e,g</td>
</tr>
<tr>
<td>d</td>
<td>c,f</td>
</tr>
<tr>
<td>e</td>
<td>a,c,f</td>
</tr>
<tr>
<td>f</td>
<td>d,e,g</td>
</tr>
<tr>
<td>g</td>
<td>c,f,h</td>
</tr>
<tr>
<td>h</td>
<td>b,g</td>
</tr>
</tbody>
</table>

### Adjacency Matrix Representation

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Adjacency Lists

- **Source**
  - a: b, c, d, e, g
  - b: c, h
  - c: a, b, d, e, g
  - d: c, f
  - e: a, c, f
  - f: d, e, g
  - g: c, f, h
  - h: b, g

- **Target**
  - a: c, e
  - b: c, h
  - c: a, b, d, e, g
  - d: c, f
  - e: a, c, f
  - f: d, e, g
  - g: c, f, h
  - h: b, g
Directed vs. Undirected Graphs

- In directed graph with edge \((u,v)\) we define
  - Successor\((u) = v\)
  - Predecessor\((v) = u\)

- Using an adjacency list representation *may* warrant two lists predecessors and successors.
Graph Runtime, $|V| = n$, $|E| = m$

<table>
<thead>
<tr>
<th>Operation vs Implementation for Edges</th>
<th>Add edge</th>
<th>Delete Edge</th>
<th>Test Edge</th>
<th>Enumerate edges for single vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array or Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted array</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adjacency List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adjacency Matrix</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation vs Implementation for Edges</td>
<td>Add edge</td>
<td>Delete Edge</td>
<td>Test Edge</td>
<td>Enumerate edges for single vertex</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>----------</td>
<td>-------------</td>
<td>-----------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Unssorted array or Linked List</td>
<td>$\Theta(1)$</td>
<td>$\Theta(m)$</td>
<td>$\Theta(m)$</td>
<td>$\Theta(m)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$\Theta(m)$</td>
<td>$\Theta(m)$</td>
<td>$\Theta(\log m)$ [if binary search used]</td>
<td>$\Theta(\log m) + \Theta(\text{deg}(v))$ [if binary search used]</td>
</tr>
<tr>
<td>Adjacency List</td>
<td>Time to find List for a given vertex + $\Theta(1)$</td>
<td>Time to find List for a given vertex + $\Theta(\text{deg}(v))$</td>
<td>Time to find List for a given vertex + $\Theta(\text{deg}(v))$</td>
<td>Time to find List for a given vertex + $\Theta(\text{deg}(v))$</td>
</tr>
<tr>
<td>Adjacency Matrix</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(\text{deg}(v))$</td>
</tr>
</tbody>
</table>
Graph Memory Requirements

• For an adjacency list:

• For adjacency matrix:

• We call a graph *sparse* if $|E|$ is $O(n)$

• We call a graph *dense* if $|E|$ is $\Omega(n^2)$

• What representation is better for a sparse graph?

• What representation is better for a dense graph?
BREADTH-FIRST SEARCH
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex that we'll call u
- BFS starts at u (‘a’ in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex

Depth 0: a
Depth 1: c,e
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex

Depth 0: a
Depth 1: c, e
Depth 2: b, d, f, g
Breadth-First Search

- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u (‘a’ in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex

Depth 0: a
Depth 1: c,e
Depth 2: b,d,f,g
Depth 3: h
Developing the Algorithm

• Key idea: Must explore all nearer neighbors before exploring further-away neighbors

• From ‘a’ we find ‘e’ and ‘c’
  – Must explore all vertices at depth i before any vertices at depth i+1
  – What data structure may help us?

Depth 0: a
Depth 1: c,e
Depth 2: b,d,f,g
Depth 3: h
Developing the Algorithm

• Exploring all vertices in the order they are found implies we will explore all vertices at shallower depth before greater depth
  – Keep a first-in / first-out queue (FIFO) of neighbors found
• Put newly found vertices in the back and pull out a vertex from the front to explore next
• We don’t want to put a vertex in the queue more than once...
  – ‘mark’ a vertex the first time we encounter it
  – only allow unmarked vertices to be put in the queue
• May also keep a ‘predecessor’ array: Allows us to find a shortest-path back to the start vertex
Breadth-First Search

Algorithm:

BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9             Q.enqueue(w)
10        pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8          if pred[w] == nil // w not found
9              Q.enqueue(w)
10         pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9            Q.enqueue(w)
10        pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

\[
\text{BFS}(G,u)
\]

1. for each vertex \( v \)
2. \( \text{pred}[v] = \text{nil}, \text{d}[v] = \text{inf} \).
3. \( Q = \text{new Queue} \)
4. \( Q\text{.enqueue}(u), \text{d}[u]=0 \)
5. while \( Q \) is not empty
6. \( v = Q\text{.front}(); Q\text{.dequeue}() \)
7. foreach neighbor, \( w \), of \( v \):
8. if \( \text{pred}[w] == \text{nil} \) // \( w \) not found
9. \( Q\text{.enqueue}(w) \)
10. \( \text{pred}[w] = v, \text{d}[w] = \text{d}[v] + 1 \)
Breadth-First Search

Algorithm:

BFS(G,u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8        if pred[w] == nil // w not found
9           Q.enqueue(w)
10        pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u), d[u]=0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v, d[w] = d[v] + 1
Breadth-First Search

Algorithm:

BFS(G,u)
1 for each vertex v
3 Q = new Queue
4 Q.enqueue(u),  d[u]=0
5 while Q is not empty
6 v = Q.front(); Q.dequeue()
7 foreach neighbor, w, of v:
8 if pred[w] == nil // w not found
9 Q.enqueue(w)
10 pred[w] = v,  d[w] = d[v] + 1

```c
for each vertex v
Q = new Queue
Q.enqueue(u),  d[u]=0
while Q is not empty
v = Q.front(); Q.dequeue()
foreach neighbor, w, of v:
if pred[w] == nil // w not found
Q.enqueue(w)
pred[w] = v,  d[w] = d[v] + 1
```
Breadth-First Search

- Shortest paths can be found by walking predecessor value from any node backward

- Example:
  - Shortest path from a to h
    - Start at h
    - Pred[h] = b (so walk back to b)
    - Pred[b] = c (so walk back to c)
    - Pred[c] = a (so walk back to a)
    - Pred[a] = nil ... no predecessor, Done!!
Breadth-First Search Trees

• BFS (and later DFS) will induce a tree subgraph (i.e. acyclic, one parent each) from the original graph
  – BFS is tree of shortest paths from the source to all other vertices (in connected component)
Correctness

- Define
  - \( \text{dist}(s,v) = \text{correct shortest distance} \)
  - \( d[v] = \text{BFS computed distance} \)
  - \( p[v] = \text{predecessor of } v \)

- Loop invariant
  - What can we say about the nodes in the queue, their \( d[v] \) values, relationship between \( d[v] \) and \( \text{dist}[v] \), etc.?  

```
BFS(G, u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6      v = Q.front(); Q.dequeue()
7      foreach neighbor, w, of v:
8          if pred[w] == nil // w not found
9              Q.enqueue(w)
10             pred[w] = v, d[w] = d[v] + 1
```
Correctness

• Define
  – dist(s,v) = correct shortest distance
  – d[v] = BFS computed distance
  – p[v] = predecessor of v

• Loop invariant
  – All vertices with p[v] != nil (i.e. already in the queue or popped from queue) have d[v] = dist(s,v)
  – The distance of the nodes in the queue are sorted
    • If Q = {v₁, v₂, ..., vᵣ} then d[v₁] <= d[v₂] <= ... <= d[vᵣ]
  – The nodes in the queue are from 2 adjacent layers/levels
    • i.e. d[vₖ] <= d[v₁] + 1
    • Suppose there is a node from a 3rd level (d[v₁] + 2), it must have been found by some, vᵢ, where d[vᵢ] = d[v₁]+1

BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u),  d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9             Q.enqueue(w)
10        pred[w] = v,  d[w] = d[v] + 1
Breadth-First Search

• Analyze the run time of BFS for a graph with n vertices and m edges
  – Find $T(n,m)$

• How many times does loop on line 5 iterate?

• How many times loop on line 7 iterate?

BFS($G,u$)
1. for each vertex $v$
2. $\text{pred}[v] = \text{nil}$, $d[v] = \text{inf}$.
3. $Q$ = new Queue
4. $Q$.enqueue($u$), $d[u]=0$
5. while $Q$ is not empty
6. $v = Q$.front(); $Q$.dequeue()
7. foreach neighbor, $w$, of $v$:
8. if $\text{pred}[w] == \text{nil}$ // $w$ not found
9. $Q$.enqueue($w$)
10. $\text{pred}[w] = v$, $d[w] = d[v] + 1$
Breadth-First Search

• Analyze the run time of BFS for a graph with n vertices and m edges
  – Find T(n)
• How many times does loop on line 5 iterate?
  – N times (one iteration per vertex)
• How many times loop on line 7 iterate?
  – For each vertex, v, the loop executes \( \deg(v) \) times
  – \( = \sum_{v \in V} \theta[1 + \deg(v)] \)
  – \( = \theta(\sum_v 1) + \theta(\sum_v \deg(v)) \)
  – \( = \Theta(n) + \Theta(m) \)
• Total = \( \Theta(n+m) \)

```plaintext
BFS(G,u)
1  for each vertex v
3  Q = new Queue
4  Q.enqueue(u), d[u]=0
5  while Q is not empty
6      v = Q.front(); Q.dequeue()
7      foreach neighbor, w, of v:
8          if pred[w] == nil // w not found
9              pred[w] = v, d[w] = d[v] + 1
```
DFS Algorithm

- DFS visits and completes all children before completing (and going on to a sibling)
- Process:
  - Visit a node
  - Mark as visited (started)
  - For each visited neighbor, visit it and perform DFS on all of their children
  - Only then, mark as finished
- Let’s trace recursive DFS!
- If cycles in the graph, mark nodes so we know to stop examining them:
  - White = unvisited,
  - Gray = visited but not finished
  - Black = finished

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6    DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4    DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
Depth First-Search

DFS-All (G)
1 for each vertex u
2 $u.color = WHITE$
3 $finish_list = empty_list$
4 for each vertex u do
5 if $u.color == WHITE$ then
6 DFS-Visit (G, u, $finish_list$)
7 return $finish_list$

DFS-Visit (G, u,l)
1 $u.color = GRAY$
2 for each vertex v in Adj(u) do
3 if $v.color = WHITE$ then
4 DFS-Visit (G, v)
5 $u.color = BLACK$
6 l.append(u)
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)
Depth First-Search

DFS-All (G)
1   for each vertex u
2      u.color = WHITE
3   finish_list = empty_list
4   for each vertex u do
5      if u.color == WHITE then
6         DFS-Visit (G, u, finish_list)
7   return finish_list

DFS-Visit (G, u, l)
1   u.color = GRAY
2   for each vertex v in Adj(u) do
3      if v.color = WHITE then
4         DFS-Visit (G, v)
5   u.color = BLACK
6   l.append(u)

Finish_list:

DFS-Visit(G,h,l):
DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):
**Depth First-Search**

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

Finish_list:
- h

DFS-Visit(G,f,l):
- Finish_list: h

DFS-Visit(G,d,l):
- Finish_list: h

DFS-Visit(G,a,l):
- Finish_list: h
Depth First-Search

DFS-All (G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5    if u.color == WHITE then
6      DFS-Visit (G, u, finish_list)
7  return finish_list

DFS-Visit (G, u,l)
1  u.color = GRAY
2  for each vertex v in Adj(u) do
3    if v.color = WHITE then
4      DFS-Visit (G, v)
5    u.color = BLACK
6  l.append(u)

Finish_list:
  h

DFS-Visit(G,g,l):
DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):
Depth First-Search

**DFS-All (G)**
1. for each vertex $u$
2. $u$.color = WHITE
3. finish_list = empty_list
4. for each vertex $u$ do
5. if $u$.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. $u$.color = GRAY
2. for each vertex $v$ in Adj($u$) do
3. if $v$.color = WHITE then
4. DFS-Visit (G, v)
5. $u$.color = BLACK
6. l.append($u$)

Finish_list:
- h, g

```
DFS-Visit(G,g,l):
DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):
```
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6.   DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3.   if v.color = WHITE then
4.     DFS-Visit (G, v)
5.   u.color = BLACK
6.   l.append(u)

Finish_list: h, g, f

DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h,
- g,
- f,
- d

DFS-Visit(G,d,l):
- d,
- f,
- g,
- h

DFS-Visit(G,a,l):
- a,
- d,
- f,
- g,
- h
Depth First-Search

**DFS-All (G)**
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

**DFS-Visit (G, u, l)**
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)

Finish_list:
- h, g, f, d

DFS-Visit(G,a,l):
- a, b, c, d, e, f, g, h

Finish_list:
- h, g, f, d
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h
- g
- f
- d

DFS-Visit(G,a,l):

DFS-Visit(G,c,l):
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h,
- g,
- f,
- d

DFS-Visit(G,e,l):
- b
- d
- f
- g
- h

DFS-Visit(G,c,l):
- b
- d
- f
- g
- h

DFS-Visit(G,a,l):
- b
- d
- f
- g
- h
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h, g, f, d, e

DFS-Visit(G,e,l):
DFS-Visit(G,c,l):
DFS-Visit(G,a,l):
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list: h, g, f, d, e, c

DFS-Visit(G,c,l):
DFS-Visit(G,a,l):
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list: h, g, f, d, e, c, a

DFS-Visit(G,a,l):
Depth First-Search

DFS-All (G)
1 for each vertex u
2 u.color = WHITE
3 finish_list = empty_list
4 for each vertex u do
5 if u.color == WHITE then
6 DFS-Visit (G, u, finish_list)
7 return finish_list

DFS-Visit (G, u, l)
1 u.color = GRAY
2 for each vertex v in Adj(u) do
3 if v.color = WHITE then
4 DFS-Visit (G, v)
5 u.color = BLACK
6 l.append(u)

Finish_list:
- h, g, f, d, e, c, a

DFS-Visit(G,b,l):
- May iterate through many complete vertices before finding b to launch a new search from
Depth First-Search

DFS-All (G)
1 for each vertex u
2 \( u.color = \text{WHITE} \)
3 \( \text{finish\_list} = \text{empty\_list} \)
4 for each vertex u do
5 \( \text{if } u.color == \text{WHITE} \) then
6 \( \text{DFS-Visit} (G, u, \text{finish\_list}) \)
7 return \( \text{finish\_list} \)

DFS-Visit (G, u, l)
1 \( u.color = \text{GRAY} \)
2 for each vertex v in \( \text{Adj}(u) \) do
3 \( \text{if } v.color = \text{WHITE} \) then
4 \( \text{DFS-Visit} (G, v) \)
5 \( u.color = \text{BLACK} \)
6 \( l.append(u) \)

Finish\_list:
- h
- g
- f
- d
- e
- c
- a
- b

DFS-Visit(G,b,l):
- u
Depth First-Search

DFS-All (G)
1. for each vertex u
2. u.color = WHITE
3. finish_list = empty_list
4. for each vertex u do
5. if u.color == WHITE then
6. DFS-Visit (G, u, finish_list)
7. return finish_list

Finish_list: h, g, f, d, e, c, a, b

DFS-Visit (G, u, l)
1. u.color = GRAY
2. for each vertex v in Adj(u) do
3. if v.color = WHITE then
4. DFS-Visit (G, v)
5. u.color = BLACK
6. l.append(u)
ITERATIVE VERSION
Depth First-Search

DFS (G,s)
1 for each vertex u
2 u.color = WHITE
3 st = new Stack
4 st.push_back(s)
5 while st not empty
6 u = st.back()
7 if u.color == WHITE then
8 u.color = GRAY
9 foreach vertex v in Adj(u) do
10 if v.color == WHITE
11 st.push_back(v)
12 else if u.color != WHITE
13 u.color = BLACK
14 st.pop_back()
BFS vs. DFS Algorithm

• BFS and DFS are more similar than you think
  – Do we use a FIFO/Queue (BFS) or LIFO/Stack (DFS) to store vertices as we find them

BFS-Visit (G, start_node)
1 for each vertex u
2 u.color = WHITE
3 u.pred = nil
4 bfsq = new Queue
5 bfsq.push_back(start_node)
6 while bfsq not empty
7 u = bfsq.pop_front()
8 if u.color == WHITE
9 u.color = GRAY
10 foreach vertex v in Adj(u) do
11 bfsq.push_back(v)

DFS-Visit (G, start_node)
1 for each vertex u
2 u.color = WHITE
3 u.pred = nil
4 st = new Stack
5 st.push_back(start_node)
6 while st not empty
7 u = st.top(); st.pop()
8 if u.color == WHITE
9 u.color = GRAY
10 foreach vertex v in Adj(u) do
11 st.push_back(v)