Graphs

A graph is a pair \((V, E)\), where

- \(V\) is a set of nodes, called vertices (aka nodes)
- \(E\) is a collection of pairs of vertices, called edges (aka arcs)
- Vertices and edges are positions and store elements

Example:

- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route
EDGE TYPES

- **Directed edge**
  - ordered pair of vertices \((u, v)\)
  - first vertex \(u\) is the origin
  - second vertex \(v\) is the destination
  - e.g., a flight

- **Undirected edge**
  - unordered pair of vertices \((u, v)\)
  - e.g., a flight route

- **Directed graph**
  - all the edges are directed
  - e.g., route network

- **Undirected graph**
  - all the edges are undirected
  - e.g., flight network

- **Mixed graph**: graph that has both directed and undirected edges
APPLICATIONS

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram

© 2014 Goodrich, Tamassia, Goldwasser
**Terminology**

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- **Degree** of a vertex
  - deg(X) = 5; X has degree 5
- Parallel edges (multiple edges)
  - h and i are parallel edges
  - Edges are collections (not sets)
- Self-loop
  - j is a self-loop

**Outgoing edges** of a vertex:
- directed edges whose origin is that vertex.

**Incoming edges** of a vertex:
- directed edges whose destination is that vertex.

**In-degree & out-degree** of a vertex v
- the number of the incoming and outgoing edges of v,
- Denoted indeg(v) and outdeg(v)
**TERMINOLOGY (CONT.)**

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints

- **Simple path**
  - path such that all its vertices and edges are distinct

- **Examples**
  - $P_1=(V,b,X,h,Z)$ is a simple path
  - $P_2=(U,c,W,e,X,g,Y,f,W,d,V)$ is a path that is **not** simple

- Graphs are said to be **simple** if they do not have parallel edges or self-loops
- Most graphs are simple; we will assume that a graph is simple unless otherwise specified
**Cycle**
- circular sequence of alternating vertices and edges
- each edge is preceded and followed by its endpoints

**Simple cycle**
- cycle such that all its vertices and edges are distinct, except for the first and the last

**Examples**
- $C_1=(V,b,X,g,Y,f,W,c,U,a,\ldots)$ is a simple cycle
- $C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,\ldots)$ is a cycle that is **not** simple
TERMINOLOGY (CONT.)

- Given vertices $u$ and $v$ of a (directed) graph $G$,
- $u$ reaches $v$, and that $v$ is reachable from $u$, if $G$ has a (directed) path from $u$ to $v$.
- **reachability**:
  - undirected graph, reachability is symmetric, that is to say, $u$ reaches $v$ if and only if $v$ reaches $u$.
  - directed graph, reachability is asymmetric, it is possible that $u$ reaches $v$ but $v$ does not reach $u$,
A subgraph $S$ of a graph $G$ is a graph such that
+ The vertices of $S$ are a subset of the vertices of $G$
+ The edges of $S$ are a subset of the edges of $G$

A **spanning subgraph** of $G$ is a subgraph that contains all the vertices of $G$.
CONNECTIVITY

- A graph is **connected** if, for any two vertices, there is a path between them.
- A directed graph $G$ is **strongly connected** if for any two vertices $u$ and $v$ of $G$, $u$ reaches $v$ and $v$ reaches $u$.
- A **connected component** of a graph $G$ is a maximal connected subgraph of $G$. 
A (free) tree is an undirected graph $T$ such that
- $T$ is connected
- $T$ has no cycles

This definition of tree is different from the one of a rooted tree

A forest is an undirected graph without cycles

The connected components of a forest are trees
A **spanning tree** of a connected graph is a spanning subgraph that is a tree

A spanning tree is not unique unless the graph is a tree

A **spanning forest** of a graph is a spanning subgraph that is a forest
**PROPERTIES**

Property 1: If $G$ is a graph with $m$ edges and vertex set $V$, then

\[ \sum_{v \in V} \deg(v) = 2m \]

Proof: each edge is counted twice

Property 2: If $G$ is a directed graph with $m$ edges and vertex set $V$, then

\[ \sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v) = m \]

Property 3: Let $G$ be a simple graph with $n$ vertices and $m$ edges. If $G$ is undirected, then

\[ m \leq n(n - 1)/2 \]

Proof: each vertex has degree at most $(n - 1)$

=> A simple graph with $n$ vertices has $O(n^2)$ edges.

**Notation**

- $n$: number of vertices
- $m$: number of edges
- $\deg(v)$: degree of vertex $v$

**Example**

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Let $G$ be an undirected graph

- If $G$ is connected, then $m \geq n - 1$.
- If $G$ is a tree, then $m = n - 1$.
- If $G$ is a forest, then $m \leq n - 1$. 

© 2014 Goodrich, Tamassia, Goldwasser
A graph is a collection of vertices and edges.

We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.

A Vertex is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code).

- We assume it supports a method, element(), to retrieve the stored element.

An Edge stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.
either undirected or directed

numVertices(): Returns the number of vertices of the graph.
vertices(): Returns an iteration of all the vertices of the graph.
numEdges(): Returns the number of edges of the graph.
edges(): Returns an iteration of all the edges of the graph.
getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
outDegree(v): Returns the number of outgoing edges from vertex v.
inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
insertVertex(x): Creates and returns a new Vertex storing element x.
insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v): Removes vertex v and all its incident edges from the graph.
removeEdge(e): Removes edge e from the graph.
In an **edge list**, we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge \((u,v)\), or the set of all edges incident to a vertex \(v\).

In an **adjacency list**, we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex. This organization allows us to more efficiently find all edges incident to a given vertex.

An **adjacency map** is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows more efficient access to a specific edge \((u,v)\), for example, in \(O(1)\) expected time with hashing.

An **adjacency matrix** provides worst-case \(O(1)\) access to a specific edge \((u,v)\) by maintaining an \(n \times n\) matrix, for a graph with \(n\) vertices. Each slot is dedicated to storing a reference to the edge \((u,v)\) for a particular pair of vertices \(u\) and \(v\); if no such edge exists, the slot will store null.
DATA STRUCTURES FOR GRAPHS: EDGE LIST

- All vertex objects are stored in an unordered list $V$, and all edge objects are stored in an unordered list $E$.

- Components:
  - Vertex object
    - reference to element $v$, to support `getElement()`
    - reference to position in vertex sequence for efficiently removed
  - Edge object
    - reference to element $e$, to support `getElement()`
    - References to the origin vertex object & destination vertex object, to support `endVertices(e)` and `opposite(v, e)`.
    - reference to position in edge sequence sequence for efficiently removed
  - Vertex sequence
    - sequence of vertex objects
  - Edge sequence
    - sequence of edge objects

Space usage is $O(n+m)$
PERFORMANCE OF THE EDGE LIST STRUCTURE

space usage is $\alpha(n+m)$

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>numVertices(), numEdges()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>vertices()</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>edges()</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>getEdge($u$, $v$), outDegree($v$), outgoingEdges($v$)</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>insertVertex($x$), insertEdge($u$, $v$, $x$), removeEdge($e$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>

Exhaustive inspection of all edges needed.

when a vertex $v$ is removed from the graph, all edges incident to $v$ must also be removed
DATA STRUCTURES FOR GRAPHS: ADJACENCY LIST

- Adds extra information to the edge list structure that supports direct access to the incident edges
  - For each vertex \( v \), we maintain a collection \( I(v) \), called *incidence collection* of \( v \)
- Components:
  - Incidence sequence for each vertex
    - sequence of references to edge objects of incident edges
  - Augmented edge objects
    - references to associated positions in incidence sequences of end vertices

adjacency list \( l_{out}(v) \)

positional list to represent \( V \)
PERFORMANCE OF THE ADJACENCY LIST STRUCTURE

assuming that the primary collection \( V \) and \( E \), and all secondary collections \( I(v) \) are implemented with doubly linked lists.

using \( O(n+m) \) space

<table>
<thead>
<tr>
<th>Method</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>numVertices(), numEdges()</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>vertices()</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>edges()</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>getEdge(u, v)</td>
<td>( O(\min(\text{deg}(u), \text{deg}(v))) )</td>
</tr>
<tr>
<td>outDegree(v), inDegree(v)</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>outgoingEdges(v), incomingEdges(v)</td>
<td>( O(\text{deg}(v)) )</td>
</tr>
<tr>
<td>insertVertex(x), insertEdge(u, v, x)</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>removeEdge(e)</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>( O(\text{deg}(v)) )</td>
</tr>
</tbody>
</table>

search through either \( I(u) \) or \( I(v) \) based on use of \( I(v) \).
DATA STRUCTURES FOR GRAPHS: ADJACENCY MAP

- use a hash-based map to implement \( I(v) \) for each vertex \( v \).
- let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value.
- \( \text{getEdge}(u, v) \) method can be implemented in \( \text{expected } O(1) \) time.

Space usage is \( \Theta(n+m) \)
**DATA STRUCTURES FOR GRAPHS: ADJACENCY MATRIX**

- **adjacency matrix** $A$ allows us to locate an edge between a given pair of vertices in **worst-case** $O(1)$ time.
- Cell $A[i][j]$ holds a reference to the edge $(u,v)$, if it exists, where $u$ is the vertex with index $i$ and $v$ is the vertex with index $j$.
- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non-adjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge.

$\mathcal{O}(n^2)$ space usage

![Graph diagram]

![Adjacency matrix]

© 2014 Goodrich, Tamassia, Goldwasser
adjacency matrix uses $O(n^2)$ space, while all other structures use $O(n+m)$ space
Positional lists to represent each of the primary lists $V$ and $E$

use a hash-based map to represent the secondary incidence map $I(v)$ for each vertex $v$ in $V$

+ each vertex maintains two different map references: outgoing and incoming.
+ Directed graphs: initialized to two distinct map instances, representing $I_{out}(v)$ and $I_{in}(v)$, respectively.
+ Undirected graph: assign both outgoing and incoming as aliases to a single map instance.

For details of the code: please look at the book.
Graph Traversals: Depth-First Search
A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges.

A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.

We will look at two efficient graph traversal algorithms:

- depth-first search (DFS)
- breadth-first search (BFS)
A DFS traversal of a graph $G$
+ Visits all the vertices and edges of $G$
+ Determines whether $G$ is connected
+ Computes the connected components of $G$
+ Computes a spanning forest of $G$

The DFS process naturally identifies what is known as the **depth-first search tree** rooted at a starting vertex $s$.

DFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time

DFS can be further extended to solve other graph problems
+ Find and report a path between two given vertices
+ Find a cycle in the graph

Depth-first search is to graphs what Euler tour is to binary trees
Algorithm DFS($G, u$):

**Input:** A graph $G$ and a vertex $u$ of $G$

**Output:** A collection of vertices reachable from $u$, with their discovery edges

Mark vertex $u$ as visited.

for each of $u$’s outgoing edges, $e = (u, v)$ do

if vertex $v$ has not been visited then

Record edge $e$ as the discovery edge for vertex $v$.

Recursively call DFS($G, v$).
/** Performs depth-first search of Graph g starting at Vertex u. */

public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,  
   Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
    known.add(u); // u has been discovered
    for (Edge<E> e : g.outgoingEdges(u)) { // for every outgoing edge from u
      Vertex<V> v = g.opposite(u, e);
      if (!known.contains(v)) {
        forest.put(v, e); // e is the tree edge that discovered v
        DFS(g, v, known, forest); // recursively explore from v
      }
    }
}
Example of a Depth-First Search

Data Structures Abstraction and Design Using Java, 2nd Edition
by Elliot B. Koffman & Paul A. T. Wolfgang, Wiley, 2010
Example of a Depth-First Search (cont.)

Mark 0 as being visited

Discovery (Visit) order:
0

Finish order:
Example of a Depth-First Search (cont.)

Choose an adjacent vertex that is not being visited.

Discovery (Visit) order:
0

Finish order:
Example of a Depth-First Search (cont.)

Choose an adjacent vertex that is not being visited

Discovery (Visit) order: 0, 1

Finish order:
Example of a Depth-First Search
(cont.)

(Recursively) choose an adjacent vertex that is not being visited

Discovery (Visit) order: 0, 1, 3

Finish order:
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

Discovery (Visit) order:
0, 1, 3

Finish order:
Example of a Depth-First Search (cont.)

(Recursively) choose an adjacent vertex that is not being visited

Discovery (Visit) order: 0, 1, 3, 4

Finish order:

0, 1, 3, 4
Example of a Depth-First Search (cont.)

There are no vertices adjacent to 4 that are not being visited.

Discovery (Visit) order: 0, 1, 3, 4

Finish order:

0, 1, 3, 4
Example of a Depth-First Search (cont.)

Mark 4 as visited

Discovery (Visit) order: 0, 1, 3, 4

Finish order: 4
Example of a Depth-First Search (cont.)

Return from the recursion to 3; all adjacent nodes to 3 are being visited

Finish order: 4
Example of a Depth-First Search (cont.)

Mark 3 as visited

Finish order: 4, 3
Example of a Depth-First Search (cont.)

Return from the recursion to 1

Finish order:
4, 3
Example of a Depth-First Search (cont.)

All vertices adjacent to 1 are being visited

Finish order:
4, 3
Example of a Depth-First Search (cont.)

Mark 1 as visited

Finish order: 4, 3, 1
Example of a Depth-First Search (cont.)

Return from the recursion to 0

Finish order: 4, 3, 1
Example of a Depth-First Search (cont.)

2 is adjacent to 1 and is not being visited

Finish order: 4, 3, 1
Example of a Depth-First Search (cont.)

Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1

2 is adjacent to 1 and is not being visited
Example of a Depth-First Search (cont.)

Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1

5 is adjacent to 2 and is not being visited
Example of a Depth-First Search (cont.)

5 is adjacent to 2 and is not being visited

Discovery (Visit) order:
0, 1, 3, 4, 2, 5

Finish order:
4, 3, 1
**Example of a Depth-First Search**

(cont.)

Discovery (Visit) order: 0, 1, 3, 4, 2, 5

Finish order: 4, 3, 1

6 is adjacent to 5 and is not being visited
Example of a Depth-First Search (cont.)

6 is adjacent to 5 and is not being visited

Discovery (Visit) order: 0, 1, 3, 4, 2, 5, 6

Finish order: 4, 3, 1
Example of a Depth-First Search (cont.)

There are no vertices adjacent to 6 not being visited; mark 6 as visited

Discovery (Visit) order: 0, 1, 3, 4, 2, 5, 6

Finish order: 4, 3, 1
There are no vertices adjacent to 6 not being visited; mark 6 as visited.

Discovery (Visit) order: 0, 1, 3, 4, 2, 5, 6

Finish order: 4, 3, 1, 6
Example of a Depth-First Search (cont.)

Return from the recursion to 5

Finish order: 4, 3, 1, 6
Example of a Depth-First Search (cont.)

Mark 5 as visited

Finish order: 4, 3, 1, 6
Example of a Depth-First Search (cont.)

Mark 5 as visited

Finish order:
4, 3, 1, 6, 5
Example of a Depth-First Search (cont.)

Return from the recursion to 2

Finish order: 4, 3, 1, 6, 5
Example of a Depth-First Search (cont.)

Mark 2 as visited

Finish order:
4, 3, 1, 6, 5
Example of a Depth-First Search (cont.)

Mark 2 as visited

Finish order: 4, 3, 1, 6, 5, 2
Example of a Depth-First Search (cont.)

Return from the recursion to 0

Finish order: 4, 3, 1, 6, 5, 2
Example of a Depth-First Search (cont.)

There are no nodes adjacent to 0 not being visited

Finish order:
4, 3, 1, 6, 5, 2
Example of a Depth-First Search (cont.)

Mark 0 as visited

Discovery (Visit) order: 0, 1, 3, 4, 2, 5, 6, 0

Finish order: 4, 3, 1, 6, 5, 2, 0
PROPERTIES OF DFS

Property 1

\( DFS(G, v) \) visits all the vertices and edges in the connected component of \( v \).

Property 2

The discovery edges labeled by \( DFS(G, v) \) form a spanning tree of the connected component of \( v \).
ANALYSIS OF DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED (Finished)
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
GRAPH TRAVERSALS: BREADTH-FIRST SEARCH
BREADTH-FIRST SEARCH

- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G

- BFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time

- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one
The algorithm uses a mechanism for setting and getting “labels” of vertices and edges.

Algorithm **BFS(G)**

**Input** graph G  
**Output** labeling of the edges and partition of the vertices of G

for all \( u \in G.vertices() \)  
setLabel\( (u, \text{UNEXPLORED}) \)

for all \( e \in G.edges() \)  
setLabel\( (e, \text{UNEXPLORED}) \)

for all \( v \in G.vertices() \)  
if getLabel\( (v) = \text{UNEXPLORED} \)  
BFS\( (G, v) \)

Algorithm **BFS(G, s)**

\( L_0 \leftarrow \) new empty sequence  
\( L_0.addLast(s) \)
setLabel\( (s, \text{VISITED}) \)

i \( \leftarrow 0 \)

while \( \neg L_i.isEmpty() \)

\( L_{i+1} \leftarrow \) new empty sequence  
for all \( v \in L_i.elements() \)  
for all \( e \in G.incidentEdges(v) \)  
if getLabel\( (e) = \text{UNEXPLORED} \)

w \( \leftarrow \) opposite\( (v,e) \)
if getLabel\( (w) = \text{UNEXPLORED} \)
setLabel\( (e, \text{DISCOVERY}) \)
setLabel\( (w, \text{VISITED}) \)
\( L_{i+1}.addLast(w) \)

else
setLabel\( (e, \text{CROSS}) \)

i \( \leftarrow i + 1 \)
```
/** Performs breadth-first search of Graph g starting at Vertex u. */
public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
    Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
    PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
    known.add(s);
    level.addLast(s); // first level includes only s
    while (!level.isEmpty()) {
        PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
        for (Vertex<V> u : level)
            for (Edge<E> e : g.outgoingEdges(u)) {
                Vertex<V> v = g.opposite(u, e);
                if (!known.contains(v)) {
                    known.add(v);
                    forest.put(v, e); // e is the tree edge that discovered v
                    nextLevel.addLast(v); // v will be further considered in next pass
                }
            }
        level = nextLevel; // relabel 'next' level to become the current
    }
}
```
Example of a Breadth-First Search
Example of a Breadth-First Search
(cont.)

Identify the start node

0 unvisited  0 visited  0 identified
Example of a Breadth-First Search (cont.)

While visiting it, we can identify its adjacent nodes.
Example of a Breadth-First Search (cont.)

We identify its adjacent nodes and add them to a queue of identified nodes.

Visit sequence:
0
Example of a Breadth-First Search (cont.)

We identify its adjacent nodes and add them to a queue of identified nodes

Queue: 1, 3

Visit sequence: 0
We color the node as visited.

Queue: 1, 3

Visit sequence: 0

Example of a Breadth-First Search (cont.)
Example of a Breadth-First Search (cont.)

Queue:
1, 3

Visit sequence:
0

The queue determines which nodes to visit next.

Visit sequence:
0

Queue:
1, 3
Visit the first node in the queue, 1

Queue: 1, 3

Visit sequence: 0

Example of a Breadth-First Search (cont.)
Visit the first node in the queue, 1

Queue:
3

Visit sequence:
0, 1
Example of a Breadth-First Search (cont.)

Select all its adjacent nodes that have not been visited or identified

Queue:
3

Visit sequence:
0, 1
Example of a Breadth-First Search (cont.)

Select all its adjacent nodes that have not been visited or identified

Queue: 3, 2, 4, 6, 7

Visit sequence: 0, 1

0 VISITED
0 IDENTIFIED
0 UNVISITED
Example of a Breadth-First Search (cont.)

Now that we are done with 1, we color it as visited.

Queue: 3, 2, 4, 6, 7

Visit sequence: 0, 1
Example of a Breadth-First Search (cont.)

and then visit the next node in the queue, 3 (which was identified in the first selection)

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1

0 unvisited  0 visited  0 identified
Example of a Breadth-First Search (cont.)

Queue: 2, 4, 6, 7

Visit sequence: 0, 1, 3

and then visit the next node in the queue, 3 (which was identified in the first selection)
3 has two adjacent vertices. 0 has already been visited and 2 has already been identified. We are done with 3

Queue: 2, 4, 6, 7

Visit sequence: 0, 1, 3
Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3
Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue: 4, 6, 7

Visit sequence: 0, 1, 3, 2
Example of a Breadth-First Search (cont.)

8 and 9 are the only adjacent vertices not already visited or identified

Queue: 4, 6, 7, 8, 9

Visit sequence: 0, 1, 3, 2
Example of a Breadth-First Search (cont.)

Queue: 6, 7, 8, 9

Visit sequence: 0, 1, 3, 2, 4

4 is next
Example of a Breadth-First Search (cont.)

5 is the only vertex not already visited or identified

Queue: 6, 7, 8, 9, 5

Visit sequence: 0, 1, 3, 2, 4
Example of a Breadth-First Search (cont.)

6 has no vertices not already visited or identified

Queue: 7, 8, 9, 5

Visit sequence: 0, 1, 3, 2, 4, 6
Example of a Breadth-First Search (cont.)

Queue:
7, 8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6

6 has no vertices not already visited or identified
Example of a Breadth-First Search (cont.)

7 has no vertices not already visited or identified

Queue:
8, 9, 5

Visit sequence:
0, 1, 3, 2, 4, 6, 7
Example of a Breadth-First Search (cont.)

7 has no vertices not already visited or identified

Queue: 8, 9, 5

Visit sequence: 0, 1, 3, 2, 4, 6, 7
Example of a Breadth-First Search (cont.)

We go back to the vertices of 2 and visit them.

Queue: 8, 9, 5

Visit sequence: 0, 1, 3, 2, 4, 6, 7
Example of a Breadth-First Search (cont.)

Queue: 9, 5
Visit sequence: 0, 1, 3, 2, 4, 6, 7, 8

8 has no vertices not already visited or identified
Example of a Breadth-First Search (cont.)

9 has no vertices not already visited or identified

Queue:
5

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9
Finally we visit 5

Queue:
5

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9
Example of a Breadth-First Search (cont.)

which has no vertices not already visited or identified

Queue:
empty

Visit sequence:
0, 1, 3, 2, 4, 6, 7, 8, 9, 5
Example of a Breadth-First Search (cont.)

The queue is empty; all vertices have been visited.

Queue: empty

Visit sequence: 0, 1, 3, 2, 4, 6, 7, 8, 9, 5
PROPERTIES

Notation
\( G_s \): connected component of \( s \)

Property 1
\( BFS(G, s) \) visits all the vertices and edges of \( G_s \)

Property 2
The discovery edges labeled by \( BFS(G, s) \) form a spanning tree \( T_s \) of \( G_s \)

Property 3
For each vertex \( v \) in \( L_i \)
  + The path of \( T_s \) from \( s \) to \( v \) has \( i \) edges
  + Every path from \( s \) to \( v \) in \( G_s \) has at least \( i \) edges
ANALYSIS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - Once as UNEXPLORED
  - Once as VISITED
- Each edge is labeled twice
  - Once as UNEXPLORED
  - Once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence $L_i$
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
Using the template method pattern, we can specialize the BFS traversal of a graph $G$ to solve the following problems in $O(n + m)$ time:

- Compute the connected components of $G$
- Compute a spanning forest of $G$
- Find a simple cycle in $G$, or report that $G$ is a forest
- Given two vertices of $G$, find a path in $G$ between them with the minimum number of edges, or report that no such path exists