CSCI 136
Data Structures &
Advanced Programming

Trees

Graph Interface

Depth-First Search

Outline

- Recap of Breadth-First Search
- Trees
- The Graph Interface
- Depth-First Search

Reachability and Connectedness

Recall

- A vertex u in G is *reachable* from a vertex v in G if there is a path from v to u
- G is connected if, for *every* vertex v, every *vertex* u is reachable from v

Alternate Definition

- G is connected if, for *some* vertex v, *every* vertex u of G is reachable from v
  - Exercise: Figure out why this is true!

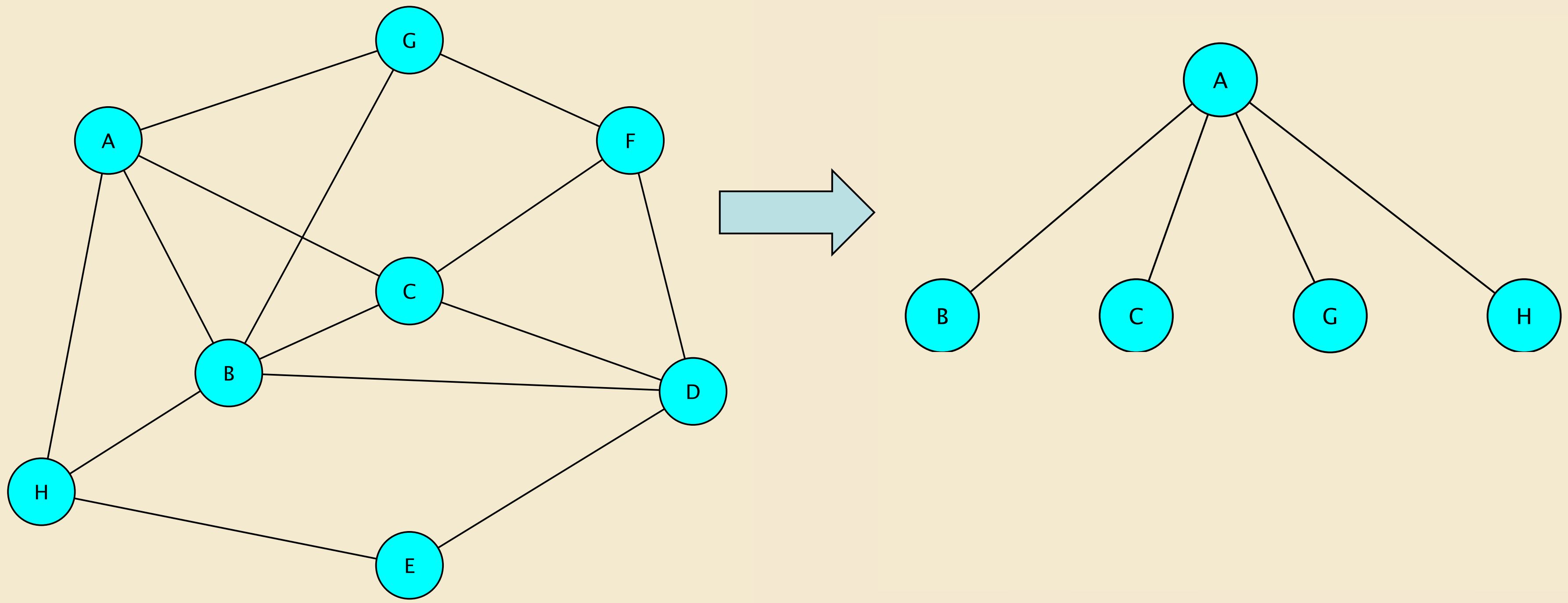
Testing Connectedness : BFS

Recall

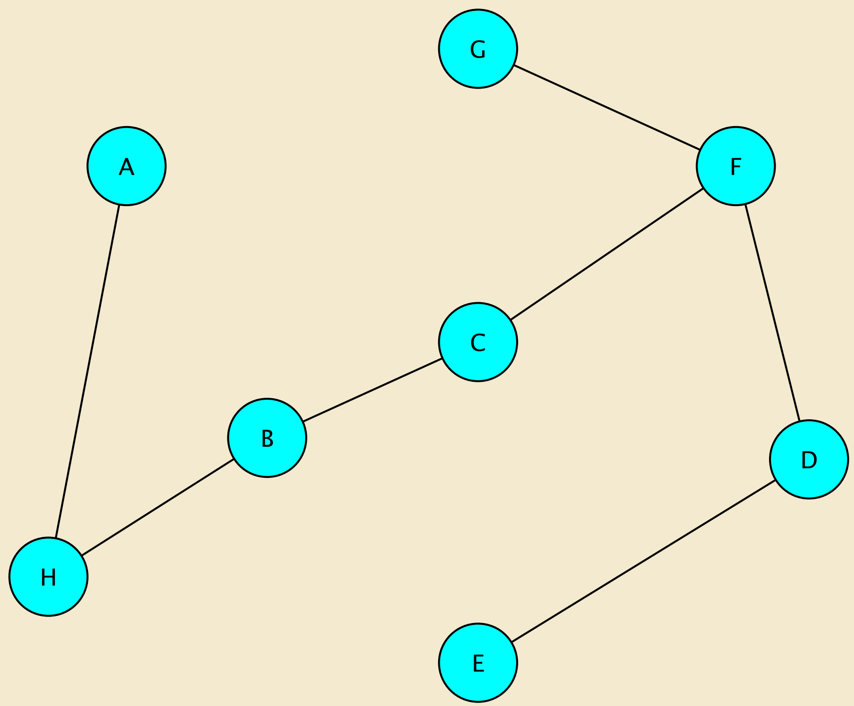
- A simple, queue-driven search (Breadth-First Search) of a graph G, starting at a vertex v, can find all vertices reachable from v
  - See previous presentation
- G is connected if and only if all vertices are reached by the BFS
- BFS can also find shortest paths from v to every other vertex
- These paths form a *tree*

BFS Reflections : Example

Assuming neighbors are visited alphabetically



Trees



Def'n: A graph G=(V,E) is a *tree* if

- G is connected
- G contains no cycles

Note

- Not the same concept as the tree data structure
  - There is no root
  - There is no hierarchical relationship

Trees : Equivalent Definitions

Try these at home

- G = (V, E) is a tree if and only if
  - G is connected
  - For every edge e in E, removing E disconnects G
- G = (V, E) is a tree if and only if
  - G is connected
  - G has exactly one more vertex than edge : |E| = |V| - 1
- G = (V, E) is a tree if and only if
  - For every two vertices u, v in V, there is *exactly* one path between u and v

Implementation with Graph Interface

What are the basic operations we need to describe the BFS method?

- Get a list of the vertices *adjacent* to v
- Mark a vertex as visited
- Add a vertex (to build the BFS tree)
- Add an edge (to build the BFS tree)

Graph Interface

- Supports storing a value at each vertex and edge
  - Called a *label*
  - Can be any object
- Supports methods for
  - get vertex/edge value
  - adding/removing vertices/edges
  - searching for vertex/edge labels
  - changing/querying 'visited' state of vertices/edges
  - producing iterators to vertices, neighbors, edges

Graph Interface Methods

- void add(V vtx), V remove(V vtx)
  - Add/remove vertex to/from graph
- void addEdge(V vtx1, V vtx2, E edgeLabel),
  E removeEdge(V vtx1, V vtx2)
  - Add/remove edge between vtx1 and vtx2
- boolean containsEdge(V vtx1, V vtx2)
  - Returns true iff there is an edge between vtx1 and vtx2
- Edge<V,E> getEdge(V vtx1, V vtx2)
  - Returns edge between vtx1 and vtx2
- void clear()
  - Remove all nodes (and edges) from graph

Graph Interface Methods

- boolean visit(V vertexLabel)
  - Mark vertex as "visited" and return *previous* value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as "visited"
- boolean isVisited(V vtx), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V vtx1)
  - Get iterator for all neighbors of vtx1
  - For directed graphs, out-edges only
- Iterator<V> iterator()
  - Get vertex iterator
- void reset()
  - Remove visited flags for all nodes/edges

Edge Class : Partial Description

- Graph *edges* are defined in their own public class
  - `Edge<V,E>(V vtx1, V vtx2,  E label)`
  - Construct a (possibly directed) edge between the two vertices having labels vtx1 and vtx2
- Useful methods:

```
label(), here(), there()
label(), setLabel()
visit(), isVisited()
```

Reachability: Breadth-First Traversal

```
BFS(G, v)        // Do a breadth-first search of G starting at v
// pre: all vertices are marked as unvisited
count ←0;
Create empty queue Q; enqueue v; mark v as visited; count++
While Q isn't empty
        current ←Q.dequeue();
        for each unvisited neighbor u  of current :
                add u to Q; mark u as visited; count++
return count;
```

Reachability: Breadth-First Traversal

BFS(G, v)

create empty queue Q

count ← 0;

enqueue v; mark v as visited

count++

while Q isn't empty

  cur ← Q.dequeue();

  for each unvisited neighbor u of cur

    add u to Q

    mark u as visited

    count++

return count;

```
int BFS(Graph<V,E> g, V src) {
  Queue<V> todo = new
    QueueList<V>();
    int count = 0;
  todo.enqueue(src);
  g.visit(src);
  count++;
  while (!todo.isEmpty()) {
    V node = todo.dequeue();
    Iterator<V> neighbors =
    g.neighbors(node);
    while (neighbors.hasNext()){
      V next = neighbors.next();
      if (!g.isVisited(next)) {
        todo.enqueue(next);
        g.visit(next); count++;
} } }
  return count;
}
```

Breadth-First Traversal

```
int BFS(Graph<V,E> g, V src) {
  Queue<V> todo = new QueueList<V>(); int count = 0;
  g.visit(src); count++;
  todo.enqueue(src);
  while (!todo.isEmpty()) {
    V node = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(node);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next)) {
          g.visit(next); count++;
          todo.enqueue(next);
       }
    }
  }
  return count;
}
```

Breadth-First Traversal of Edges

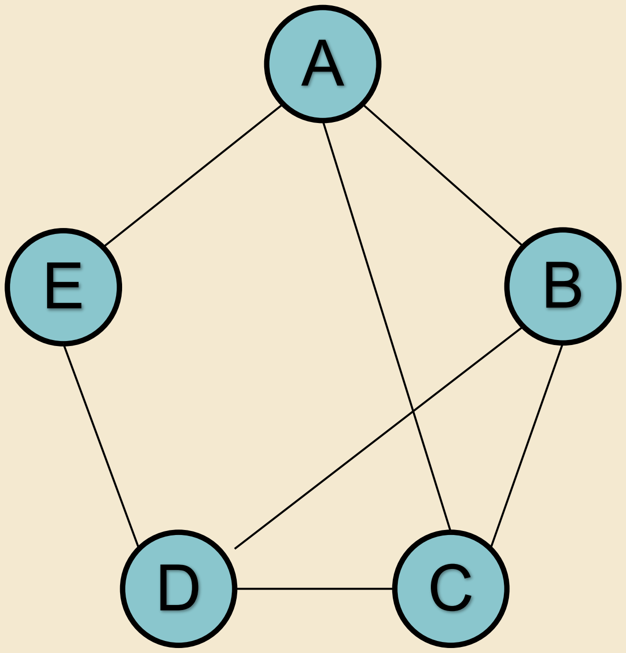
```
int BFS(Graph<V,E> g, V src) {
  Queue<V> todo = new QueueList<V>(); int count = 0;
  g.visit(src); count++;
  todo.enqueue(src);
  while (!todo.isEmpty()) {
    V node = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(node);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisitedEdge(node,next)) g.visitEdge(next,node);
       if (!g.isVisited(next)) {
          g.visit(next); count++;
          todo.enqueue(next);
       }
    }
  }
  return count;
}
```

Creating a Graph

```
Graph<String,Integer> g = new
    GraphListUndirected<String,Integer>();

 g.add("A");
 g.add("B");
 g.add("C");
 g.add("D");
 g.add("E");

 g.addEdge("A","B", 1);
 g.addEdge("A","C", 1);
 g.addEdge("B","C", 1);
 g.addEdge("C","D", 1);
 g.addEdge("D","B", 1);
 g.addEdge("D","E", 1);
 g.addEdge("E","A", 1);
```



Depth-First Search for Graphs

Reachability II : Depth-First Search

Suppose we replace the queue used in the Breadth-First Search algorithm with a stack

It turns out that

- We still visit exactly the vertices reachable from the starting vertex
- The algorithm is equally efficient (Big-O sense)
- The order in which vertices are visited is very different

Reachability II : Depth-First Search

```
DFS(G, v)   // Do a depth-first search of G starting at v
// pre: all vertices are marked as unvisited
count ←0;
Create empty stack S; push v; mark v as visited; count++;
While S isn't empty
    current ←S.pop();
    for each unvisited neighbor u  of current :
        add u to S; mark u as visited; count++
return count;
```

Reachability II : Depth-First Search

DFS(G, v)

create empty stack S

count ← *0;*

push v onto S; mark v as visited

count++

while S isn't empty

  cur ← *S.pop();*

  for each unvisited neighbor u of cur

    push u onto S

    mark u as visited

    count++

return count;

```
int DFS(Graph<V,E> g, V src) {
  Stack<V> todo = new
    StackList<V>();
    int count = 0;
  todo.push(src);
  g.visit(src);
  count++;
  while (!todo.isEmpty()) {
    V node = todo.pop();
    Iterator<V> neighbors =
    g.neighbors(node);
    while (neighbors.hasNext()){
      V next = neighbors.next();
      if (!g.isVisited(next)) {
        todo.push(next);
        g.visit(next); count++;
  } } }
  return count;
}
```

Reachability II : Depth-First Search

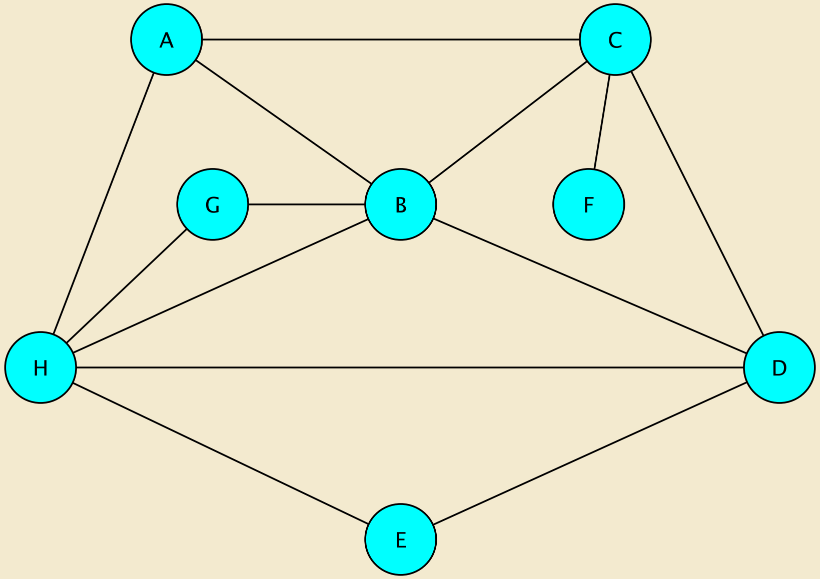
```
public static <V,E> int DFS(Graph<V,E> g, V src) {
   Stack<V> todo = new StackList<V>(); int count = 0;
   g.visit(src); count++;
   todo.push(src);
   while (!todo.isEmpty()) {
     V node = todo.pop();
     Iterator<V> neighbors = g.neighbors(node);
     while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next)) {
         g.visit(next); count++;
         todo.push(next);
       }
     }
   }
   return count;
 }
```

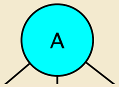
DFS Reflections

- The DFS algorithm traces out a tree different from that produced by BFS
  - It still consists of the edges connecting a visited vertex to (as yet) unvisited neighbors
- It is called a DFS *tree of G with root v* (or *from v*)
- Vertices are processed in (a variant of) pre-order w.r.t. the tree
- By manipulating the stack differently, we could produce a post-order version of DFS
- And perhaps write DFS recursively….

DFS : Example

Assuming neighbors are stacked in *reverse* order





Reachability III : Recursive DFS

```
// Before first call to DFS, set all vertices to unvisited
//Then call DFS(G,v)
DFS(G, v)
        Mark v as visited; count = 1;
        for each unvisited neighbor u of v:
                count += DFS(G,u);
        return count;
```

Reachability III : Recursive DFS

DFS(G, v)

Mark v as visited

count = 1;

for each unvisited neighbor u of v:

count += DFS(G,u);

return count;

```
public static <V,E> int
DFS(Graph<V,E> g, V src) {

 g.visit(src);

 int count = 1;

 Iterator<V> neighbors =
     g.neighbors(src);

 while (neighbors.hasNext()) {
  V next = neighbors.next();

  if (!g.isVisited(next))
   count+= DFS(g, next);
  }

 return count;
}
```

Reachability III : Recursive DFS

```
public static <V,E> int DFS(Graph<V,E> g, V src) {
   g.visit(src);
   int count = 1;
   Iterator<V> neighbors = g.neighbors(src);
   while (neighbors.hasNext()) {
     V next = neighbors.next();
     if (!g.isVisited(next)) count+= DFS(g, next);
   }
   return count;
 }
```

Summary & Observations

- Two different methods to traverse a connected component of a graph
  - Breadth-First Search
    - Explores short paths from v before long paths
  - Depth-First Search
    - Explores longest paths possible
- Graph Interface
  - Allows writing of Graph algorithms based on local structure of graph
    - Independent of implementation of graph structure
- Coming up: Directed Graphs & Implementations!