

IPv4 INTERNET TOPOLOGY MAP  
AS-LEVEL INTERNET GRAPH

GRAPHS

Lecture 18  
CS2110 - Fall 2014

### Graph Algorithms

- Search
  - depth-first search
  - breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Minimum spanning trees
  - Prim's algorithm
  - Kruskal's algorithm

### Depth-First Search

Given a graph and one of its nodes  $u$  (say node 1 below).

We want to "visit" each node reachable from  $u$  ONCE (nodes 1, 0, 2, 3, 5).

There are many paths to some nodes.

How to visit nodes only once, efficiently, and not do extra work?

### Depth-First Search

```
boolean[] visited;
```

node  $u$  is visited means: `visited[u]` is true  
To visit  $u$  means to: `set visited[u] to true`

Node  $v$  is REACHABLE from node  $u$  if there is a path  $(u, \dots, v)$  in which all nodes of the path are unvisited.

Suppose all nodes are unvisited.

Nodes REACHABLE from node 1: 1, 0, 2, 3, 5

Nodes REACHABLE from 4: 4, 5, 6.

### Depth-First Search

```
boolean[] visited;
```

node  $u$  is visited means: `visited[u]` is true  
To visit  $u$  means to: `set visited[u] to true`

Node  $u$  is REACHABLE from node  $v$  if there is a path  $(u, \dots, v)$  in which all nodes of the path are unvisited.

Red nodes: visited.  
Blue nodes: unvisited

Nodes REACHABLE from 1: 1, 0, 5

Nodes REACHABLE from 4: none

### Depth-First Search

```
/** Node u is unvisited. Visit all nodes that are REACHABLE from u. */
public static void dfs(int u) {
}
```

Let  $u$  be 1  
The nodes that are REACHABLE from node 1 are 1, 0, 2, 3, 5

### Depth-First Search

7

```

/** Node u is unvisited. Visit all nodes
that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
}
    
```

Let u be 1  
The nodes that are REACHABLE from node 1 are 1, 0, 2, 3, 5

### Depth-First Search

8

```

/** Node u is unvisited. Visit all nodes
that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v) leaving u:
        if v is unvisited then dfs(v);
}
    
```

Let u be 1  
The nodes to be visited are 0, 2, 3, 5

Have to do dfs on all unvisited neighbors of u

### Depth-First Search

9

```

/** Node u is unvisited. Visit all nodes
that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v) leaving u:
        if v is unvisited then dfs(v);
}
    
```

Let u be 1  
Nodes to be visited are: 0, 2, 3, 5

Suppose the loop visits neighbors in numerical order. Then dfs(1) visits the nodes in this order: 1, 0, 2, 3, 5  
**Depth First!**

### Depth-First Search

10

```

/** Node u is unvisited. Visit all nodes
that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v) leaving u:
        if v is unvisited then dfs(v);
}
    
```

Notes:

Suppose n nodes are REACHABLE along e edges (in total). What is Worst-case execution? Worst-case space?

### Depth-First Search

11

```

/** Node u is unvisited. Visit all nodes
that are REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v) leaving u:
        if v is unvisited then dfs(v);
}
    
```

That's all there is to the basic dfs. You may have to change it to fit a particular situation.

If you don't have this spec and you do something different, it's probably wrong.

**Example:** Use different way (other than array visited) to know whether a node has been visited

**Example:** We really haven't said what data structures are used to implement the graph.

### Depth-First Search in an OO fashion

12

```

public class Node {
    boolean visited;
    List<Node> neighbors;

    /** This node is unvisited. Visit all nodes
    REACHABLE from this node */
    public void dfs() {
        visited= true;
        for (Node n: neighbors) {
            if (!n.visited) n.dfs();
        }
    }
}
    
```

Each node of the graph is an Object of class Node

No need for a parameter. The object is the node

### Depth-First Search written iteratively

```

13
/** Node u is unvisited. Visit all node REACHABLE from u. */
public static void dfs(int u) {
    Stack s= (u); // Not Java
    // inv: all nodes that have to be visited are
    // REACHABLE from some node in s
    while ( s is not empty ) {
        u= s.pop(); // Remove top stack node, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
    
```

### Depth-First Search written iteratively

```

14
/** u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v):
                s.push(v);
        }
    }
}
    
```

1  
stack s

### Depth-First Search written iteratively

```

15
/** u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v):
                s.push(v);
        }
    }
}
    
```

Call dfs(1)      Iteration 0

0  
2  
5  
stack s

### Depth-First Search written iteratively

```

16
/** u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Stack s= (u);
    while (s is not empty) {
        u= s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v):
                s.push(v);
        }
    }
}
    
```

Call dfs(1)      Iteration 1

Yes, 5 is put on stack twice,  
once for each edge to it. It  
will be visited only once.

Using a  
stack causes  
depth-first  
search      3  
6  
2  
5  
stack s

### Breadth-First Search

```

17
/** Node u is unvisited. Visit all node REACHABLE from u. */
public static void dfs(int u) {
    Queue q= (u); // Not Java
    // inv: all nodes that have to be visited are
    // REACHABLE from some node in q
    while ( q is not empty ) {
        u= q.removeFirst(); // Remove first node, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
    
```

### Breadth-First Search

```

18
/** u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    Queue q= (u);
    while (q is not empty) {
        u= q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v):
                append(v);
        }
    }
}
    
```

1  
Queue q

### Breadth-First Search

19

```

/** u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
  Queue q= (u);
  while (q is not empty) {
    u= q.popFirst();
    if (u has not been visited) {
      visit u;
      for each edge (u, v):
        append(v);
    }
  }
}

```

Call dfs(1)      Iteration 0

0 2 5  
Queue q

### Breadth-First Search

20

```

/** u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
  Queue q= (u);
  while (q is not empty) {
    u= q.popFirst();
    if (u has not been visited) {
      visit u;
      for each edge (u, v):
        append(v);
    }
  }
}

```

Call dfs(1)      Iteration 1

0 2 5 5  
Queue q

### Breadth-First Search

21

```

/** u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
  Queue q= (u);
  while (q is not empty) {
    u= q.popFirst();
    if (u has not been visited) {
      visit u;
      for each edge (u, v):
        append(v);
    }
  }
}

```

Call dfs(1)      Iteration 2

Breadth first:

- (1) Node u
- (2) All nodes 1 edge from u
- (3) All nodes 2 edges from u
- (4) All nodes 3 edges from u
- ...

2 5 3 5  
Queue q