Graph Traversals

CSCI 241, Fall 2018
Lecture 21
Announcements
Goals

• Understand and be able to implement graph traversal/search algorithms:
  – Depth-first search
  – Breadth-first search
Look, a graph!
Look, a graph!
Look, a graph!
Graph Algorithms

• Search
  – Depth-first search
  – Breadth-first search

• Shortest paths
  – Dijkstra's algorithm

• Minimum spanning trees
  – Prim's algorithm
  – Kruskal's algorithm
Representations of Graphs

Adjacency List

1 → 2 → 4
2 → 3
3
4 → 2 → 3

Adjacency Matrix

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 0 & 1 & 0 & 1 \\
2 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 1 & 0 \\
\end{array}
\]
Adjacency Matrix or Adjacency List?

- **Definitions:**
  - $n = \text{number of vertices}$
  - $e = \text{number of edges}$
  - $d(u) = \text{degree of } u = \text{number of edges leaving } u$

- **Adjacency Matrix**
  - Uses space $O(n^2)$
  - Can iterate over all edges in time $O(n^2)$
  - Can answer “Is there an edge from $u$ to $v$?” in $O(1)$ time
  - Better for dense graphs (lots of edges)

- **Adjacency List**
  - Uses space $O(e + n)$
  - Can iterate over all edges in time $O(e + n)$
  - Can answer “Is there an edge from $u$ to $v$?” in $O(d(u))$ time
  - Better for sparse graphs (fewer edges)
Breaking DAG

Which of the following two graphs are DAGs?

**Directed Acyclic Graph**
Breaking DAG

Diagram of a directed acyclic graph (DAG) with nodes connected by arrows. The graph transitions from node 1 to node 3 and then to node 2. A matrix representation of the graph is also shown:

```
  1  2  3
1  0  1  1
2  0  0  0
3  0  1  0
```
Back to Important Things:
Depth-First Search

• Given a graph and one of its nodes $u$
  (say node 1 below)
Depth-First Search

• Given a graph and one of its nodes $u$ (say node 1 below)
• We want to “visit” each node reachable from $u$ (nodes 1, 0, 2, 3, 5)

There are many paths to some nodes.

How do we visit all nodes efficiently, without doing 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
• v is explorable from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.

Suppose all nodes are unvisited.
Depth-First Search

boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is explorable from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.

Suppose all nodes are unvisited.

Nodes exploriable from node 1:
{1, 0, 2, 3, 5}
Depth-First Search

```java
boolean[] visited;

• Node u is visited means: visited[u] is true
• To visit u means to: set visited[u] to true
• v is exploriable from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.
```

Suppose all nodes are unvisited.

Nodes exploriable from node 1: 
\{1, 0, 2, 3, 5\}

Nodes exploriable 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
• v is explorable from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.

Green: visited
Blue: unvisited
boolean[ ] visited;

- Node u is visited means: visited[u] is true
- To visit u means to: set visited[u] to true
- v is **explorable** from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.

**Green**: visited
**Blue**: unvisited

Nodes **explorable** from node 1:
{1, 0, 5}
Depth-First Search

boolean[ ] visited;

• Node u is visited means: visited[u] is true
• To visit u means to: set visited[u] to true
• v is **explorable** from u if there is a path (u, ..., v) in which all nodes of the path are unvisited.

**Green**: visited

**Blue**: unvisited

Nodes **explorable** from node 1:
{1, 0, 5}

Nodes **explorable** from 4: none

Not even 4 itself, because it’s already been visited!
Depth-First Search

/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
}
```

Let u be 1

The nodes explorable from 1 are 1, 0, 2, 3, 5
Depth-First Search

/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */
public static void dfs(int u) {

Let u be 1
The nodes explorable from 1 are 1, 0, 2, 3, 5

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

Let u be 1
The nodes explorable from 1 are 1, 0, 2, 3, 5
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
    visited[u] = true;
}
```

Let u be 1 (visited)
The nodes to be visited are 0, 2, 3, 5
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

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

Have to do DFS on all unvisited neighbors of u!
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}

Suppose the for loop visits neighbors in numerical order. Then \textbf{dfs(1)} visits the nodes in this order: 1 ...
Depth-First Search

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

Suppose the for loop visits neighbors in numerical order. Then \texttt{dfs(1)} visits the nodes in this order: 1, 0 ...
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

Suppose the for loop visits neighbors in numerical order. Then `dfs(1)` visits the nodes in this order: 1, 0, 2 ...
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}

Suppose the for loop visits neighbors in numerical order. Then dfs(1) visits the nodes in this order: 1, 0, 2, 3 ...
/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
**Depth-First Search**

/** Visit all nodes that are explorable from u. Precondition: u is unvisited. */

```java
public static void dfs(int u) {
    visited[u] = true;
    for all edges (u, v) leaving u:
        if v is unvisited then dfs(v);
}
```

Suppose $n$ nodes are explorable along $e$ edges (in total). What is

- Worst-case runtime? $O(n+e)$
- Worst-case space? $O(n)$
Depth-First Search

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

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

That’s all there is to 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.
Depth-First Search in OO fashion

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

    /** Visit all nodes that are explorable * from u. Precondition: u is unvisited */
    public void dfs() {
        visited = true;
        for (Node n: neighbors) {
            if (!n.visited) n.dfs();
        }
    }
}
```

Each node of the graph is an object of type Node

No need for a parameter. The object is the node.
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void dfs(int nodeID) {
    Stack s = (nodeID);  // Not Java!
    // inv: all nodes that have to be visited are
    // explorable 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);
        }
    }
}
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void dfs(int u) {
    Stack s = new Stack();
    s.push(u);
    while (!s.isEmpty()) {
        int u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)

Stack s
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
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) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  
Iteration 0

Stack s
**Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void dfs(int u) {
    Stack s = new Stack(u);
    while (!s.isEmpty()) {
        u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)  
Iteration 0

Stack s
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void dfs(int u) {
    Stack s = new Stack();
    while (!s.isEmpty()) {
        int u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call `dfs(1)`

Iteration 0

Stack `s`
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
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) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  
Iteration 0

Stack s
0
2
5
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void dfs(int u) {
    Stack s = new Stack(u);
    while (!s.isEmpty()) {
        int u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)  

Iteration 1  

Stack s
/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    s.push(u);
    while (!s.isEmpty()) {
        u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  

Iteration 1  

Stack s
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    s.push(u);
    while (!s.isEmpty()) {
        int curr = s.pop();
        if (!visited[curr]) {
            visit(curr);
            for (int next : edges.get(curr)) {
                s.push(next);
            }
        }
    }
}

Call dfs(1)  Iteration 1

```
1 2 3 4 5 6
```

Stack s

2
5
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
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) leaving u:
                s.push(v);
        }
    }
}
```
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
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) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  Iteration 2

Stack s
/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    s.push(u);
    while (!s.isEmpty()) {
        u = s.pop();
        if (!visited(u)) {
            visit(u);
            for (each edge (u, v) leaving u:
                s.push(v);
            }
        }
    }
}
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    s.push(u);
    while (s is not empty) {
        u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  Iteration 2

Yes, 5 is put on the stack twice, once for each edge to it. It will be visited only once.
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
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) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)  

Iteration 3  

Stack s

```
3
5
5
```
Depth-First Search written iteratively

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

Call dfs(1)  
Iteration 3

Stack s
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = new Stack();
    s.push(u);
    while (s is not empty) {
        u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  Iteration 3

Stack s

Call dfs(1)
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */
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) leaving u:
                s.push(v);
        }
    }
}

Call dfs(1)  Iteration 3

Stack s

Call dfs(1)  Iteration 3

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

Call dfs(1)  
Iteration 4

Stack s
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void dfs(int u) {
    Stack s = new Stack();
    while (!s.isEmpty()) {
        int u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)  Iteration 4

Stack s

1 -> 2 -> 3
1 -> 0
0 -> 5
5 -> 6
Depth-First Search written iteratively

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

Call dfs(1)  Iteration 4

1 2 3

5 4

0 5 6

Stack s
**Depth-First Search written iteratively**

/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
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) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)

Iteration 5

Stack s
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
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) leaving u:
                s.push(v);
        }
    }
}
```

Call dfs(1)  
Iteration 6

Stack s

Diagram of a graph with nodes labeled 0 to 6, and edges connecting them. The stack `s` is shown with nodes 1, 2, and 3 in order, illustrating the progress of the algorithm.
Depth-First Search written iteratively

/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void dfs(int u) {
    Stack s = new Stack();
    while (!s.isEmpty()) {
        int u = s.pop();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                s.push(v);
        }
    }
}
```

```
That’s DFS!

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s= (u);  // Not Java!
    // inv: all nodes that have to be visited are
    // explorable 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);
        }
    }
}

Want to see a magic trick?
Depth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void dfs(int u) {
    Stack s = (u);  // Not Java!
    // inv: all nodes that have to be visited are
    // explorable 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);
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void bfs(int u) {
    Queue q = (u); // Not Java!
    // inv: all nodes that have to be visited are
    // explorable from some node in s
    while ( q is not empty ) {
        u = q.popFirst(); // Remove first node in queue, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v); // Add to end of queue
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void bfs(int u) {
    Queue q = (u); // Not Java!
    // inv: all nodes that have to be visited are
    // explorable from some node in s
    while (q is not empty) {
        u = q.popFirst(); // Remove first node in queue, put in u
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v); // Add to end of queue
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void bfs(int u) {
    Queue q = new Queue(u);
    while q is not empty) {
        u = q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1)

```
1

Queue q
```
/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1)  Iteration 0

```
1
Queue q
```
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)     Iteration 0

Queue q
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)  
Iteration 0

Queue q
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)  Iteration 0

0 2
Queue q
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)   Iteration 1

0 2
Queue q
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)  Iteration 1

1

0

2

3

4

5

6

7

Queue q
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

```java
public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}
```

Call bfs(1)  
Iteration 1  

Queue q
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)  Iteration 1

queue q

1 2 3 4 5 6 7

2 7
/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q = new Queue(u);
    while (q is not empty) {
        u = q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */

public static void bfs(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) leaving u:
                q.append(v);
        }
    }
}

Call bfs(1)  Iteration 2

1  0  2  3  4  5  6  7

Queue q
Breadth-First Search

/** Visit all nodes explorable from u. Pre: u is unvisited. */
public static void bfs(int u) {
    Queue q = new Queue(u);
    while q is not empty) {
        u = q.popFirst();
        if (u has not been visited) {
            visit u;
            for each edge (u, v) leaving u:
                q.append(v);
        }
    }
}
Breadth-First Search

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

Call bfs(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
...

7 3 5
Queue q
Some working code for DFS

- [https://codeboard.io/projects/97448](https://codeboard.io/projects/97448)
- Sample graph constructed by the code:

  - Suggested exercises:
    - Run DFS by hand
    - Run BFS by hand
    - Code BFS
Questions to Ponder

• BFS(root) on a tree corresponds to which tree traversal?

• Write out the order nodes are visited in this undirected graph, when calling:
  – BFS(5)
  – DFS(5)
  – DFS(0)
  (if there are ties, visit the lower # first)