Introduction
To Graphs

D. Thiebaut
CSC212 — Fall 2014
Formal Definition

- $G = (V, E)$
  - $V$ is the set of *vertices*
  - $E$ is the set of *edges*
Directed Graphs

• $G = (V, E)$

• Each element $e$ of $E$ is an ordered pair $(v_i, v_j)$
Undirected Graphs

- $G = (V, E)$
- For each edge $(v_i, v_j)$ of $E$, there is an edge $(v_j, v_i)$
How Do We Store Graphs?

- Adjacency List
- Adjacency Matrix
- Linked Nodes
- Incidence Matrix
Adjacency List

(a)

(b)

(c)
Adjacency Matrix
Incidence Matrix

Why the different data-structure?
Terminology
Path

cycle

Disconnected

connected
$e$ emanate from $v_i$

out-degree of $v_i$ is 4

$e$ is incident to $V_k$

in-degree of $v_k$ is 3
Java Implementation
Java Review: Iterators
public static void main(String[] args) {
    ArrayList<Integer> array = new ArrayList<Integer>();
    for (int i=0; i<5; i++)
        array.add(i*2 + 1);

    Iterator<Integer> it = array.iterator();

    while (it.hasNext()) {
        int x = it.next();
        System.out.print(x + " ");
    }
}
public static void main(String[] args) {
    ArrayList<Integer> array = new ArrayList<Integer>();
    for (int i = 0; i < 5; i++)
        array.add(i * 2 + 1);

    for (int x : array) {
        System.out.println(x);
    }
}
Java Implementation (one option)
```java
public class Graph1 {
    private boolean[][] adjMat;
    private int noVertices;
    private int noEdges;
    private Set<Integer> vertices;
    private boolean[] visited;
    private int count;

    public Graph1(int V) {
        adjMat = new boolean[V][V]; // allocated to false by default
        noVertices = V;
        noEdges = 0;
        vertices = new TreeSet<Integer>();
    }

    public void addEdge(int v1, int v2) {
        vertices.add(v1);
        vertices.add(v2);
        if (!adjMat[v1][v2])
            noEdges++;
        adjMat[v1][v2] = true;
        adjMat[v2][v1] = true;
    }
}
```
public boolean contains(int v1, int v2) {
    return adjMat[v1][v2];
}

// return list of neighbors of v
public Iterable<Integer> adj(int v) {
    return new AdjMatIterator(v);
}
// support iteration over graph vertices
private class AdjMatIterator implements Iterator<Integer>, Iterable<Integer> {
    int v, w = 0;

    AdjMatIterator(int v) {
        this.v = v;
    }

    public Iterator<Integer> iterator() {
        return this;
    }

    public boolean hasNext() {
        while ( w < noVertices ) {
            if ( adjMat[v][w] ) return true;
            w++;
        }
        return false;
    }

    public Integer next() {
        if ( hasNext() ) return w++;
        return null;
    }
}
/**
 * (0) --- (1) --- (2)
 * |     | \ 
 * |     | \
 * (3) --- (4) (5) --- (6)
 * |
 * (7) --- (8)
 */

public static void main(String[] args) {
    Graph1 G = new Graph1(9);
    G.addEdge( 0, 1 );  G.addEdge( 1, 2 );  G.addEdge( 3, 4 );
    G.addEdge( 5, 6 );  G.addEdge( 7, 8 );  G.addEdge( 0, 3 );
    G.addEdge( 1, 4 );  G.addEdge( 1, 5 );  G.addEdge( 3, 7 );
    G.addEdge( 4, 8 );

    System.out.print( "Vertices adjacent to 1: " );
    for (int v: G.adj(1))
        System.out.print( v + " " );
}

Vertices adjacent to 1: 0 2 4 5
DFS: Depth First Search
Example

1
5
2
4
3
6
0
Example
Example

0 1 2 3 4 5 6

0 1 2 3 4

1, 4, 6

5, 2, 4
Example

0 1 2 3 4 5 6

1, 2, 4

2, 4, 6

1, 2, 6, 3
Example
Example

0 1 2 3 4 5 6

1,4,6
5,2,4
1

\$\emptyset, \varnothing, 0\$

0,4

5,2,6,3
2,4,0

1,2,6,3

\$\emptyset, \emptyset\$

1,2,6,3

6

5

1

3

4
Example

1, 4, 6
5, 2, 4
1, 2, 6, 3
2, 4, 0
3, 6
0, 4
What can we use DFS for?

What property of the graph can we test with DFS?
Java Implementation
private void recurseDFS(int v) {
    visited[v] = true;

    for (int w : adj(v))
        if (!visited[w])
            recurseDFS(w);
}

public void DFS(int v) {
    visited = new boolean[noVertices];
    recurseDFS(v);
}
Complexity
## Complexity

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Must initialize data structure for each vertex</td>
<td>$O(\ldots)$</td>
</tr>
<tr>
<td>Must follow each edge emanating from each vertex</td>
<td>$O(\ldots)$</td>
</tr>
</tbody>
</table>
Lab: Play with DFS
We stopped here last time...
BFS
Breadth First Search
Observations

• Both DFS and BFS create **spanning trees** with the edges they visit.

• The trees are **rooted** at the first vertex they start with.

• DFS works best as a **recursive** function

• BFS works best with a **queue**
Weighted Graphs
Weighted Graphs: Definitions

- Each edge $e$ has a weight

- The cost of a path through linked vertices is the **sum of the weights** of the edges on that path.
Finding the Shortest Path from a Vertex to the Other Vertices in its Component
Dijkstra’s Algorithm
The Basic Idea
DijkstraAlgorithm( G, start )
   for all vertices v in G
      cost( v ) = \infty;

   cost( start ) = 0;
   unvisited = all vertices in G
   while ( unvisited not empty )
      v = unvisited vertex with lowest cost
      unvisited = unvisited - v
      for all unvisited w adjacent to v
         if cost( w ) > cost( v ) + weight( v, w )
            cost( w ) = cost( v ) + weight( v, w )
            predecessor( w ) = v
Shortest Path from 5 to 3?

Shortest Path from 5 to all?
for all vertices $v$ in $G$
  $\text{cost}(v) = \infty$;

$\text{cost}(\text{start}) = 0$;

$\text{unvisited} = \text{all vertices in } G$

while (unvisited not empty)
  $v = \text{unvisited vertex with lowest cost}$
  mark $v$ as visited
  for all unvisited $w$ adjacent to $v$
    if $\text{cost}(w) > \text{cost}(v) + \text{weight}(v,w)$
      $\text{cost}(w) = \text{cost}(v) + \text{weight}(v,w)$
      $\text{predecessor}(w) = v$
for all vertices \( v \) in \( G \)
\[
\text{cost}( v ) = \infty;
\]
\[
\text{cost}( \text{start} ) = 0;
\]
\[
\text{unvisited} = \text{all vertices in } G
\]
while ( unvisited not empty )
\[
\text{\textbf{v} = unvisited vertex with lowest cost}
\]
\[
\text{mark v as visited}
\]
for all unvisited \( w \) adjacent to \( v \)
\[
\text{if cost}( w ) > \text{cost}(v) + \text{weight}(v,w)
\]
\[
\text{cost}( w ) = \text{cost}(v) + \text{weight}(v,w)
\]
\[
\text{predecessor}( w ) = v
\]
for all vertices \( v \) in \( G \)
\[
cost( v ) = \infty;
\]

\[
cost( \text{start} ) = 0;
\]

unvisited = all vertices in \( G \)

while ( unvisited not empty )
\[
v = \text{unvisited vertex with lowest cost}
\]
mark \( v \) as visited

for all unvisited \( w \) adjacent to \( v \)
if \( cost( w ) > cost( v ) + \text{weight}(v,w) \)
\[
cost( w ) = cost( v ) + \text{weight}(v,w)
\]
predecessor( \( w \) ) = \( v \)
for all vertices $v$ in $G$
$\text{cost}(v) = \infty$;

$\text{cost}(\text{start}) = 0$;
unvisited = all vertices in $G$
while (unvisited not empty)
    $v =$ unvisited vertex with lowest cost
    mark $v$ as visited
    for all unvisited $w$ adjacent to $v$
        if $\text{cost}(w) > \text{cost}(v) + \text{weight}(v,w)$
            $\text{cost}(w) = \text{cost}(v) + \text{weight}(v,w)$
            predecessor($w$) = $v$
for all vertices \( v \) in \( G \)
\[
\text{cost}(v) = \infty
\]
\[
\text{cost}(\text{start}) = 0;
\]
\( \text{unvisited} = \text{all vertices in } G \)
while ( \text{unvisited} \text{ not empty } )
\[
\text{v} = \text{unvisited vertex with lowest cost}
\]
mark \( v \) as visited
for all unvisited \( w \) adjacent to \( v \)
if \( \text{cost}(w) > \text{cost}(v) + \text{weight}(v, w) \)
\[
\text{cost}(w) = \text{cost}(v) + \text{weight}(v, w)
\]
\[
\text{predecessor}(w) = v
\]
for all vertices $v$ in $G$
\[ \text{cost}(v) = \infty; \]

cost(start) = 0;
unvisited = all vertices in $G$
while ( unvisited not empty )
\[ v = \text{unvisited vertex with lowest cost} \]
mark $v$ as visited
for all unvisited $w$ adjacent to $v$
if cost($w$) > cost($v$) + weight($v$, $w$)
\[ \text{cost}(w) = \text{cost}(v) + \text{weight}(v,w) \]
\[ \text{predecessor}(w) = v \]
for all vertices \( v \) in \( G \)
\[
\text{cost}(v) = \infty;
\]

\[
\text{cost}(\text{start}) = 0;
\]

\( \text{unvisited} = \text{all vertices in } G \)

while ( unvisited not empty )

\( v = \text{unvisited vertex with lowest cost} \)

mark \( v \) as visited

for all unvisited \( w \) adjacent to \( v \)

if \( \text{cost}(w) > \text{cost}(v) + \text{weight}(v,w) \)

\[
\text{cost}(w) = \text{cost}(v) + \text{weight}(v,w)
\]

\[
\text{predecessor}(w) = v
\]
for all vertices $v$ in $G$
\[ \text{cost}(v) = \infty; \]

\[ \text{cost}(\text{start}) = 0; \]
unvisited = all vertices in $G$
while (unvisited not empty)
\[ v = \text{unvisited vertex with lowest cost} \]
mark $v$ as visited
for all unvisited $w$ adjacent to $v$
if cost($w$) > cost($v$) + weight($v$, $w$)
\[ \text{cost}(w) = \text{cost}(v) + \text{weight}(v, w) \]
predecessor($w$) = $v$
for all vertices \( v \) in \( G \)
\[
\text{cost}(v) = \infty;
\]

\[
\text{cost}(\text{start}) = 0;
\]

\( \text{unvisited} = \) all vertices in \( G \)

while ( \text{unvisited} \text{ not empty} )

\[
\text{v} = \text{unvisited vertex with lowest cost}
\]

mark \( v \) as visited

for all unvisited \( w \) adjacent to \( v \)

if \( \text{cost}(w) > \text{cost}(v) + \text{weight}(v,w) \)

\[
\text{cost}(w) = \text{cost}(v) + \text{weight}(v,w)
\]

\[
\text{predecessor}(w) = v
\]
for all vertices \( v \) in \( G \)
\[
\text{cost}( v ) = \infty;
\]

\[
\text{cost}( \text{start} ) = 0;
\]

\( \text{unvisited} = \text{all vertices in } G \)

while ( unvisited not empty )

\( v = \text{unvisited vertex with lowest cost} \)

mark \( v \) as visited

for all unvisited \( w \) adjacent to \( v \)

if \( \text{cost}( w ) > \text{cost}(v) + \text{weight}(v,w) \)

\( \text{cost}( w ) = \text{cost}(v) + \text{weight}(v,w) \)

predecessor( \( w \) ) = \( v \)
for all vertices \( v \) in \( G \)
\[
\text{cost}(v) = \infty;
\]
\[
\text{cost}(\text{start}) = 0;
\]
unvisited = all vertices in \( G \)
while ( unvisited not empty )
  \( v \) = unvisited vertex with lowest cost
  mark \( v \) as visited
  for all unvisited \( w \) adjacent to \( v \)
    if cost( \( w \) ) > cost(\( v \) + weight(\( v \),\( w \))
      cost( \( w \) ) = cost(\( v \) + weight(\( v \),\( w \))
      predecessor( \( w \) ) = \( v \)
for all vertices $v$ in $G$
\[ \text{cost}(v) = \infty; \]

\[ \text{cost}(\text{start}) = 0; \]

\[ \text{unvisited} = \text{all vertices in } G \]

\[ \text{while } (\text{unvisited not empty}) \]

\[ v = \text{unvisited vertex with lowest cost} \]

\[ \text{mark } v \text{ as visited} \]

\[ \text{for all unvisited } w \text{ adjacent to } v \]

\[ \text{if } \text{cost}(w) > \text{cost}(v) + \text{weight}(v,w) \]

\[ \text{cost}(w) = \text{cost}(v) + \text{weight}(v,w) \]

\[ \text{predecessor}(w) = v \]
for all vertices \( v \) in \( G \)
\[
\text{cost}(v) = \infty;
\]
\[
\text{cost}(\text{start}) = 0;
\]
\[
\text{unvisited} = \text{all vertices in } G
\]
while ( \text{unvisited} \text{ not empty} )
\[
\quad v = \text{unvisited vertex with lowest cost}
\]
mark \( v \) as visited
for all unvisited \( w \) adjacent to \( v \)
\[
\quad \text{if cost}(w) > \text{cost}(v) + \text{weight}(v,w)
\]
\[
\quad \text{cost}(w) = \text{cost}(v) + \text{weight}(v,w)
\]
\[
\quad \text{predecessor}(w) = v
\]
for all vertices $v$ in $G$

$$\text{cost}(v) = \infty;$$

cost( start ) = 0;

unvisited = all vertices in $G$

while ( unvisited not empty )

$\vphantom{v}v = \text{unvisited vertex with lowest cost}$

mark $v$ as visited

for all unvisited $w$ adjacent to $v$

if cost( $w$ ) > cost($v$) + weight($v$, $w$)

$$\text{cost}(w) = \text{cost}(v) + \text{weight}(v,w)$$

predecessor( $w$ ) = $v$
How do we implement the collection of all unvisited vertices?

How do we quickly find the vertex with the lowest cost?
Complexity of Dijkstra
DijkstraAlgorithm( G, start )
    for all vertices v in G
        cost( v ) = ∞;
    cost( start ) = 0;
    unvisited = all vertices in G
    while ( unvisited not empty )
        v = unvisited vertex with lowest cost
        unvisited = unvisited - v
        for all unvisited w adjacent to v
            if cost( w ) > cost(v) + weight(v,w)
                cost( w ) = cost(v) + weight(v,w)
                predecessor( w ) = v
DijkstraAlgorithm( G, start )

for all vertices v in G
    cost( v ) = ∞;

    cost( start ) = 0;

unvisited = all vertices in G

while ( unvisited not empty )
    v = unvisited vertex with lowest cost

    unvisited = unvisited - v

    for all unvisited w adjacent to v
        if cost( w ) > cost(v) + weight(v,w)
            cost( w ) = cost(v) + weight(v,w)
            predecessor( w ) = v
DijkstraAlgorithm( G, start )

for all vertices v in G
  cost( v ) = \infty;

cost( start ) = 0;
unvisited = all vertices in G
while ( unvisited not empty )
  v = unvisited vertex with lowest cost
  unvisited = unvisited - v
  for all unvisited w adjacent to v
    if cost( w ) > cost(v) + weight(v,w)
      cost( w ) = cost(v) + weight(v,w)
      predecessor( w ) = v

O(V)
DijkstraAlgorithm( G, start )

for all vertices v in G
    cost( v ) = ∞;

cost( start ) = 0;

unvisited = all vertices in G

while ( unvisited not empty )

    v = unvisited vertex with lowest cost

    unvisited = unvisited - v

    for all unvisited w adjacent to v
        if cost( w ) > cost(v) + weight(v,w)
            cost( w ) = cost(v) + weight(v,w)
            predecessor( w ) = v
DijkstraAlgorithm( G, start )

for all vertices v in G
    cost( v ) = ∞;

cost( start ) = 0;

unvisited = all vertices in G

while ( unvisited not empty )
    v = unvisited vertex with lowest cost
    unvisited = unvisited - v
    for all unvisited w adjacent to v
        if cost( w ) > cost(v) + weight(v,w)
            cost( w ) = cost(v) + weight(v,w)
            predecessor( w ) = v
DijkstraAlgorithm( G, start )

for all vertices v in G
  cost( v ) = \infty;

cost( start ) = 0;
unvisited = all vertices in G
while ( unvisited not empty )
  v = unvisited vertex with lowest cost
  unvisited = unvisited - v
  for all unvisited w adjacent to v
    if cost( w ) > cost(v) + \text{weight}(v,w)
      cost( w ) = cost(v) + \text{weight}(v,w)
      predecessor( w ) = v

O(V)

O(V)

?
DijkstraAlgorithm( G, start )

for all vertices v in G
  \text{cost}( v ) = \infty;

\text{cost}( \text{start} ) = 0;

\text{unvisited} = \text{all vertices in G}

while ( \text{unvisited not empty} )
  v = \text{unvisited vertex with lowest cost}
  \text{unvisited} = \text{unvisited} - v
  for all unvisited w adjacent to v
    if \text{cost}( w ) > \text{cost}(v) + \text{weight}(v,w)
      \text{cost}( w ) = \text{cost}(v) + \text{weight}(v,w)
      \text{predecessor}( w ) = v
DijkstraAlgorithm( G, start )

for all vertices v in G

\[ \text{cost}(v) = \infty; \]

\[ \text{cost}(\text{start}) = 0; \]

unvisited = all vertices in G

while (unvisited not empty)

\[ v = \text{unvisited vertex with lowest cost} \]

unvisited = unvisited - v

for all unvisited w adjacent to v

if \[ \text{cost}(w) > \text{cost}(v) + \text{weight}(v,w) \]

\[ \text{cost}(w) = \text{cost}(v) + \text{weight}(v,w) \]

\[ \text{predecessor}(w) = v \]
DijkstraAlgorithm( G, start )

for all vertices v in G
    cost( v ) = ∞;

cost( start ) = 0;

unvisited = all vertices in G

while ( unvisited not empty )
    v = unvisited vertex with lowest cost
    unvisited = unvisited - v
    for all unvisited w adjacent to v
        if cost( w ) > cost(v) + weight(v,w)
            cost( w ) = cost(v) + weight(v,w)
            predecessor( w ) = v

O(V)

O(V)

O(V)

O(V)

O(log(V))
DijkstraAlgorithm\( ( G, \text{start} ) \)

1. for all vertices \( v \) in \( G \)
   
   \[
   \text{cost}( v ) = \infty;
   \]

2. \( \text{cost( start )} = 0; \)

3. \( \text{unvisited } = \text{all vertices in } G \)

4. while ( unvisited not empty )
   
   a. \( v = \text{unvisited vertex with lowest cost} \)
   
   b. \( \text{unvisited } = \text{unvisited } - v \)

5. for all unvisited \( w \) adjacent to \( v \)
   
   a. if \( \text{cost}( w ) > \text{cost}( v ) + \text{weight}( v, w ) \)
      
      \[
      \text{cost}( w ) = \text{cost}( v ) + \text{weight}( v, w )
      \]
      
      \[
      \text{predecessor}( w ) = v
      \]

\( \text{O}(V) \)
DijkstraAlgorithm( G, start )

for all vertices v in G
   cost( v ) = \infty;

cost( start ) = 0;

unvisited = all vertices in G

while ( unvisited not empty )
   v = unvisited vertex with lowest cost
   unvisited = unvisited - v
   for all unvisited w adjacent to v
      if cost( w ) > cost(v) + weight(v,w)
         cost( w ) = cost(v) + weight(v,w)
         predecessor( w ) = v

\text{O}(V) \quad \text{O}(V) \quad \text{O}(V)
DijkstraAlgorithm( G, start )

for all vertices v in G
    cost( v ) = ∞;

cost( start ) = 0;

unvisited = all vertices in G

while ( unvisited not empty )
    v = unvisited vertex with lowest cost
    unvisited = unvisited - v
    for all unvisited w adjacent to v
        if cost( w ) > cost(v) + weight(v,w)
            cost( w ) = cost(v) + weight(v,w)
            predecessor( w ) = v

O(V)
O(V)
O(V)
O(E + V.log(V))
<table>
<thead>
<tr>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Dijskra’s Shortest Path Algorithm: $O(\ldots)$</td>
</tr>
</tbody>
</table>
Dijkstra will fail...

- If graph contains edges with negative weights
Dijkstra in Java
class Vertex implements Comparable<Vertex> {
    public final int Id;
    public Edge[] adj;
    public double minDistance = Double.POSITIVE_INFINITY;
    public Vertex predecessor;

    public Vertex( int i ) { Id = i; }
    public String toString() { return Id +" "; }
    public int compareTo(Vertex other) {
        return Double.compare(minDistance, other.minDistance);
    }
}
class Edge {
    public final Vertex source;
    public final Vertex dest;
    public final double weight;

    public Edge(Vertex argSource, Vertex argTarget, double argWeight) {
        source = argSource;
        dest = argTarget;
        weight = argWeight;
    }
}
public class Graph2 {
    Vertex[] vertices;

    Graph2() {
        vertices = null;
    }

    public static void dijkstraPaths(Vertex source) {
        source.minDistance = 0.;
        PriorityQueue<Vertex> vertexQueue = new PriorityQueue<Vertex>();
        vertexQueue.add(source);

        while (!vertexQueue.isEmpty()) {
            Vertex u = vertexQueue.poll();

            // Visit each edge emanating from u
            for (Edge e : u.adj) {
                Vertex v = e.dest;
                double weight = e.weight;
                double distanceThroughU = u.minDistance + weight;
                if (distanceThroughU < v.minDistance) {
                    vertexQueue.remove(v);
                    v.minDistance = distanceThroughU;
                    v.predecessor = u;
                    vertexQueue.add(v);
                }
            }
        }
    }
}

All-Pair Shortest Paths
Warshall, Floyd & Ingerman’s Algorithm
WFIAAlgorithm( matrix weight)
  for i = 0 to V-1 // V= # vertices
    for j = 0 to V-1
      for k = 0 to V-1
        if weight[j][k] > weight[j][i] + weight[i][k]
          weight[j][k] = weight[j][i] + weight[i][k]
### Complexity

- **Warshall, Floyd, Ingerman’s Algorithm:** $O(\ldots)$
Topological Sort

Which course should I take first?
Basic Idea

topologicalSort( digraph )
  for v in V:
    find a vertex v with empty adjacency list
    number( v ) = i++
    remove v and all edges incident to v from consideration
Algorithm (1)

topologicalSort( digraph ) {
    for all v in V:
        num(v) = 0;
        TSnum(v) = 0;
    i = 0;
    j = 0;
    while there is v s.t. num(v)==0
        TS( v );
    return list of v sorted by TSnum;
}
Algorithm (2)

```c
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
    if num(u) == 0
        TS(u);
    else if TSNum(u) == 0
        error("Cycle detected!")
    TSNum(v) = j++;
}
```
Algorithm (2)

TS( v ) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}
\[ TS(v) \{ \]
\[ \quad \text{num}(v) = i++; \]
\[ \quad \text{for all } u \text{ adjacent to } v \]
\[ \quad \quad \text{if } \text{num}(u) == 0 \]
\[ \quad \quad \quad TS(u); \]
\[ \quad \quad \text{else if } \text{TSNum}(u) == 0 \]
\[ \quad \quad \quad \text{error}("Cycle detected!") \]
\[ \quad \text{TSNum}(v) = j++; \]
\[ \} \]
\[ \text{TS}(v) \{ \]
\[ \quad \text{num}(v) = i++; \]
\[ \quad \text{for all } u \text{ adjacent to } v \]
\[ \quad \quad \text{if } \text{num}(u) == 0 \]
\[ \quad \quad \quad \text{TS}(u); \]
\[ \quad \quad \else \text{if } \text{TSNum}(u) == 0 \]
\[ \quad \quad \quad \text{error(“Cycle detected!”)} \]
\[ \quad \quad \text{TSNum}(v) = j++; \]
\[ \} \]
\[ TS(v) \] {
    num(v) = i++;
    for all u adjacent to v
    if num(u) == 0
        TS(u);
    else if TSNum(u) == 0
        error("Cycle detected!")
    TSNum(v) = j++;
}
TS(v) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS(u);
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}
\[ TS( v ) \begin{cases} 
    \text{num}(v) = i++; \\
    \text{for all } u \text{ adjacent to } v \\
    \quad \text{if } \text{num}(u) == 0 \\
    \quad \quad TS( u ); \\
    \quad \text{else if } \text{TSNum}(u) == 0 \\
    \quad \quad \text{error(“Cycle detected!”)} \\
    \text{TSNum}(v) = j++; 
\end{cases} \]
\[
\text{TS}(v) \{
\text{num}(v) = i++; \\
\text{for all } u \text{ adjacent to } v \\
\quad \text{if } \text{num}(u) == 0 \\
\quad \quad \text{TS}(u); \\
\quad \text{else if } \text{TSNum}(u) == 0 \\
\quad \quad \text{error(“Cycle detected!”)} \\
\text{TSNum}(v) = j++; \\
\}
\]

```
num       = 1 4 2 3 5 0 6
TSnum     = 0 0 0 0 0 0 0 1
```
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
    if num(u) == 0
        TS( u );
    else if TSNum(u) == 0
        error("Cycle detected!")
    TSNum(v) = j++;
}
\[ \text{TS}( v ) \{ \]
\begin{align*}
\text{num}(v) & = i++; \\
\text{for all } u \text{ adjacent to } v & \\
\quad \text{if } \text{num}(u) == 0 & \\
\quad \quad \text{TS}( u ); & \\
\quad \text{else if } \text{TSNum}(u) == 0 & \\
\space & \quad \quad \text{error(“Cycle detected!”)} \\
\quad \quad \text{TSNum}(v) & = j++; \\
\}
\end{align*}

\text{a} \quad \text{c} \quad \text{d} \\
\text{b} \quad \text{e} \\
\text{f} \quad \text{g}

\begin{array}{cccccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} \\
\text{num} & 1 & 4 & 2 & 3 & 5 & 0 & 6 \\
\text{TSnum} & 0 & 3 & 0 & 0 & 0 & 2 & 0 & 1
\end{array}
TS( v ) {
    num(v) = i++;  
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;  
}
TS( v ) {
    num(v) = i++;  
    for all u adjacent to v  
        if num(u) == 0 
            TS( u );  
        else if TSNum(u) == 0  
            error("Cycle detected!")
    TSNum(v) = j++;  
}

diagram with nodes and edges labeled

num 1 4 2 3 5 7 6
TSnum 0 3 0 0 2 4 1
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}
\[ \text{TS}(v) \{ \]
\[ \quad \text{num}(v) = i++; \]
\[ \quad \text{for all } u \text{ adjacent to } v \]
\[ \quad \quad \text{if } \text{num}(u) == 0 \]
\[ \quad \quad \quad \text{TS}(u); \]
\[ \quad \quad \text{else if } \text{TSNum}(u) == 0 \]
\[ \quad \quad \quad \text{\textcolor{red}{error}}(\text{"Cycle detected!"}) \]
\[ \quad \text{TSNum}(v) = j++; \]
\[ \} \]
```c
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}
```
TS( v ) {
    num(v) = i++;
    for all u adjacent to v
        if num(u) == 0
            TS( u );
        else if TSNum(u) == 0
            error("Cycle detected!")
    TSNum(v) = j++;
}