GRAPHS

Chapter 10

Chapter Objectives

- To become familiar with graph terminology and the different types of graphs
- To study a Graph ADT and different implementations of the Graph ADT
- To learn the breadth-first and depth-first search traversal algorithms
- To learn some algorithms involving weighted graphs
- To study some applications of graphs and graph algorithms

Graphs

- Trees are limited in that a data structure can only have one parent
- Graphs overcome this limitation
- Graphs were being studied long before computers were invented
- Graphs algorithms run
  - large communication networks
  - the software that makes the Internet function
  - programs to determine optimal placement of components on a silicon chip
- Graphs describe
  - roads maps
  - airline routes
  - course prerequisites

Graph Terminology

Section 10.1

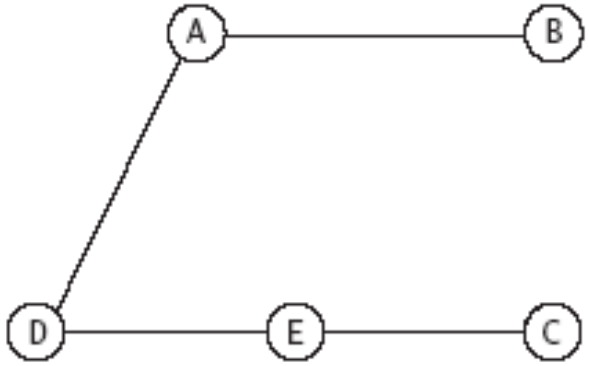
Graph Terminology

- A graph is a data structure that consists of a set of *vertices* (or nodes) and a set of *edges* (relations) between pairs of vertices
- Edges represent paths or connections between vertices
- Both the set of vertices and the set of edges must be finite
- Either set may be empty (if the set of vertices is empty, the set of edges also must be empty)
- We restrict our discussion to simple graphs in which there is at most one edge from a given vertex to another vertex

Visual Representation of Graphs

- Vertices are represented as points or labeled circles and edges are represented as line segments joining the vertices

V = {A, B, C, D, E}
E = {{A, B}, {A, D}, {C, E}, {D, E}}

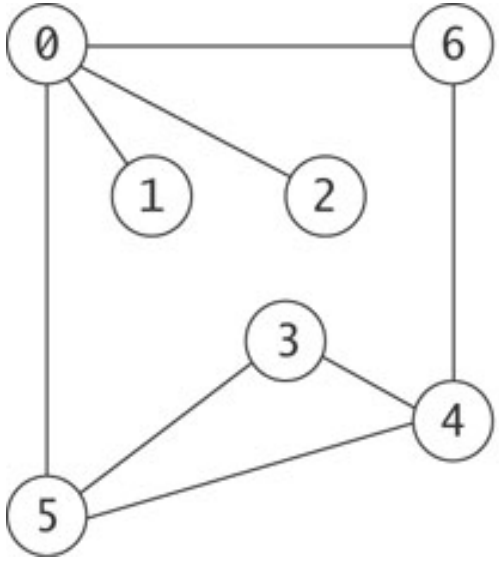


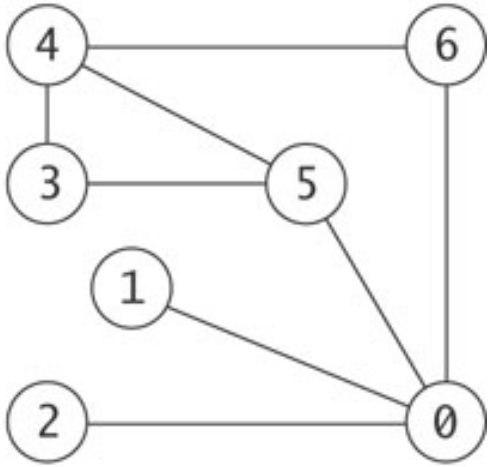
Visual Representation of Graphs (cont.)

- Each edge is represented by the two vertices it connects
- If there is an edge between vertices *x* and *y*, there is a *path* from *x* to *y* and vice versa

Visual Representation of Graphs (cont.)

- The physical layout of the vertices and their labeling is not relevant

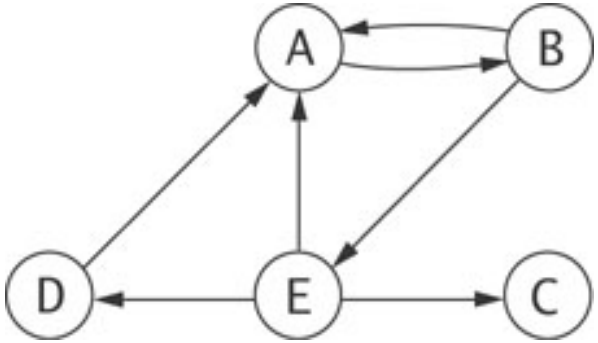




V = {0, 1, 2, 3, 4, 5, 6}
E = {{0, 1}, {0, 2}, {0, 5}, {0, 6}, {3, 5}, {3, 4}, {4, 5}, {4, 6}}

Directed and Undirected Graphs

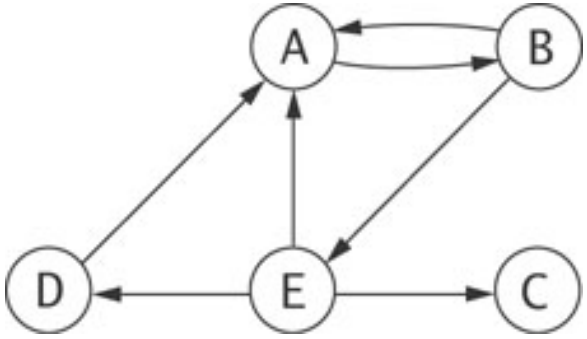
- The edges of a graph are *directed* if the existence of an edge from A to B does not necessarily guarantee that there is a path in both directions



- A graph with directed edges is called a *directed graph* or *digraph*
- A graph with undirected edges is an *undirected graph*, or simply a graph

Directed and Undirected Graphs (cont.)

- A directed edge is like a one-way street; you can travel in only one direction

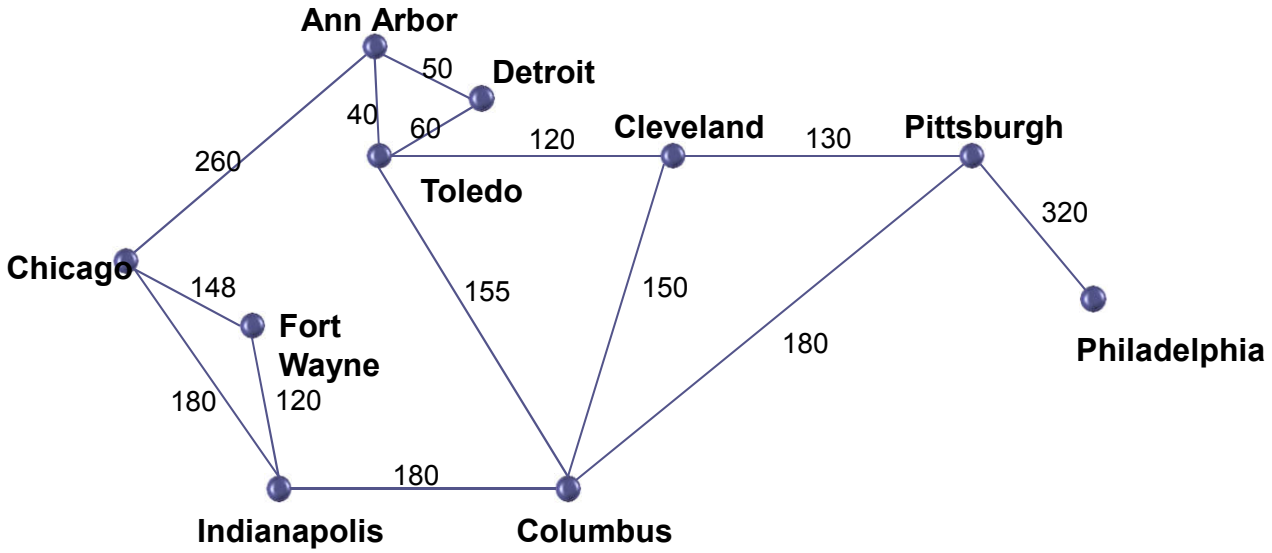


- Directed edges are represented as lines with an arrowhead on one end (undirected edges do not have an arrowhead at either end)
- Directed edges are represented by ordered pairs of vertices (*source*, *destination*)*;* the edges for the digraph on this slide are:

  {(A, B), (B, A), (B, E), (D, A), (E, A), (E, C), (E, D)}

Directed and Undirected Graphs (cont.)

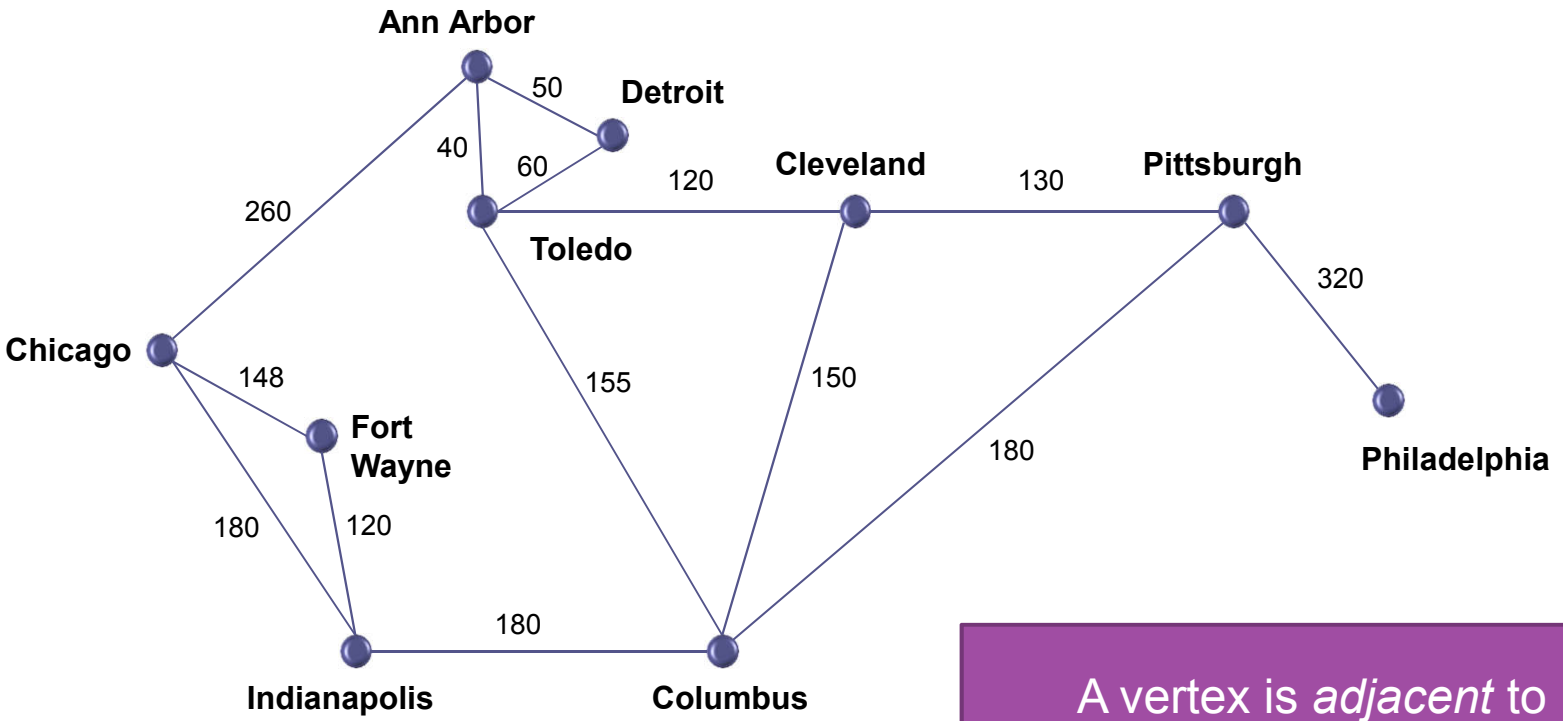
- The edges in a graph may have associted values known as their *weights*
- A graph with weighted edges is known as a *weighted graph*



Paths and Cycles

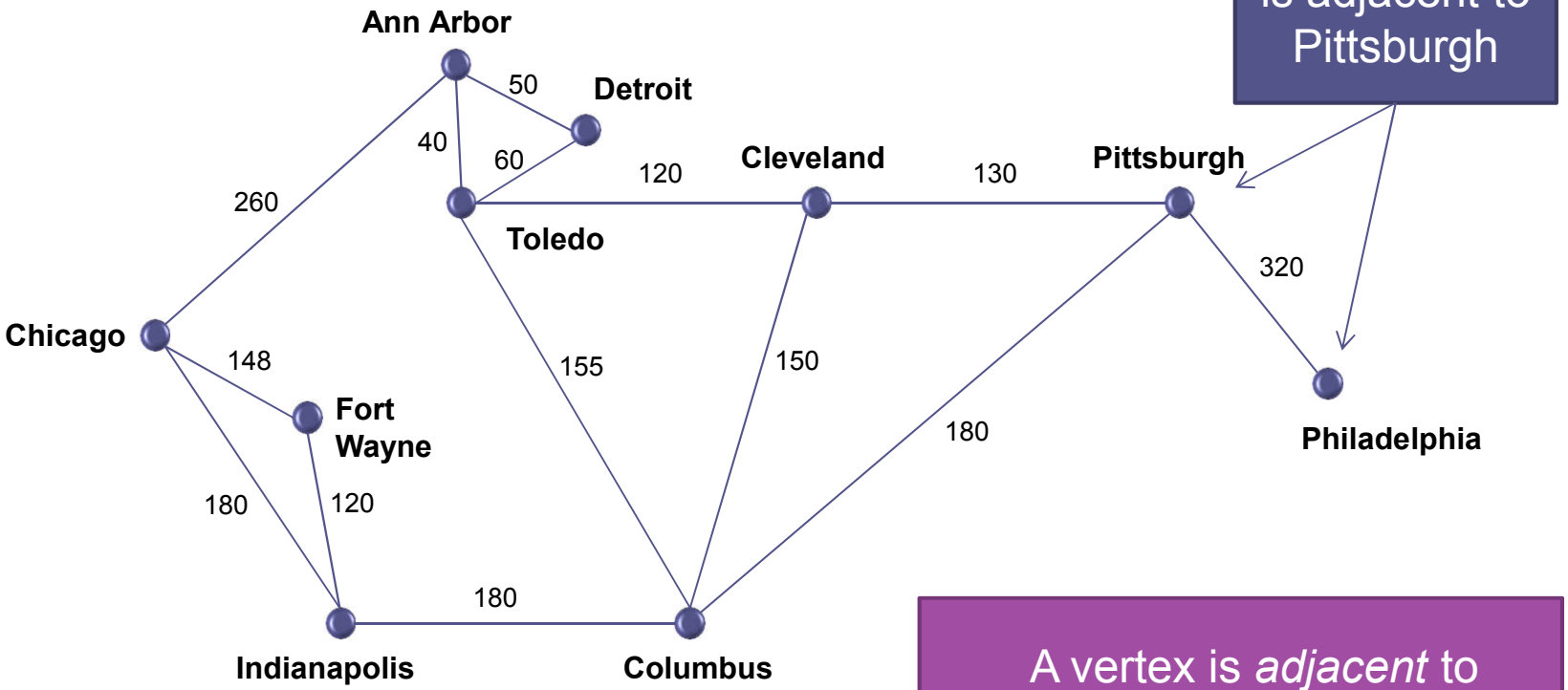
- The following definitions describe pathways between vertices

Paths and Cycles (cont.)



A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

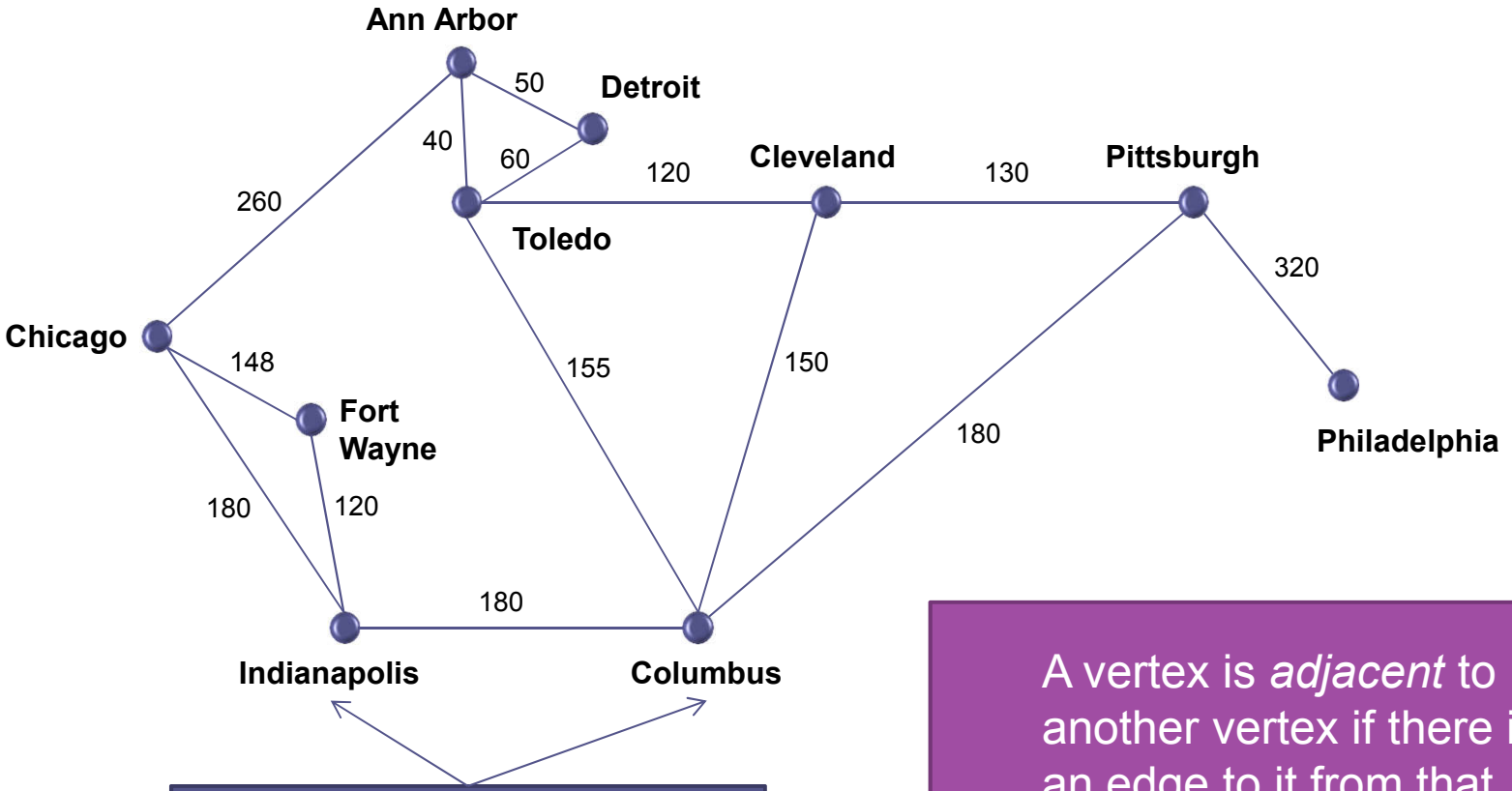
Paths and Cycles (cont.)



Philadelphia is adjacent to Pittsburgh

A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

Paths and Cycles (cont.)

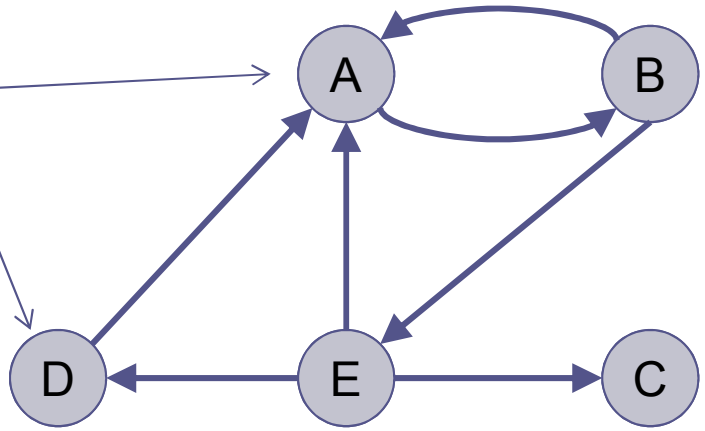


Indianapolis is adjacent to Columbus

A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

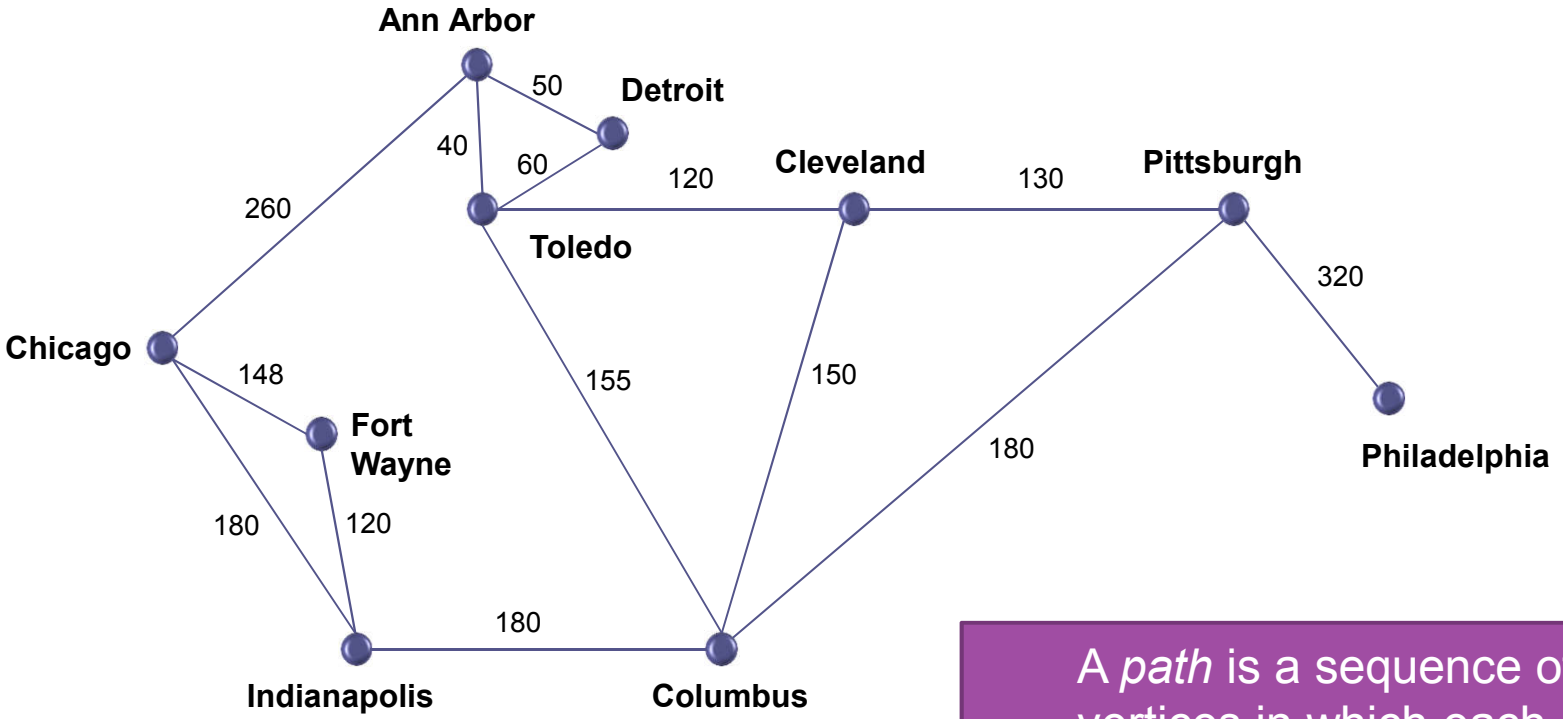
Paths and Cycles (cont.)

A is adjacent to D, but D is NOT adjacent to A



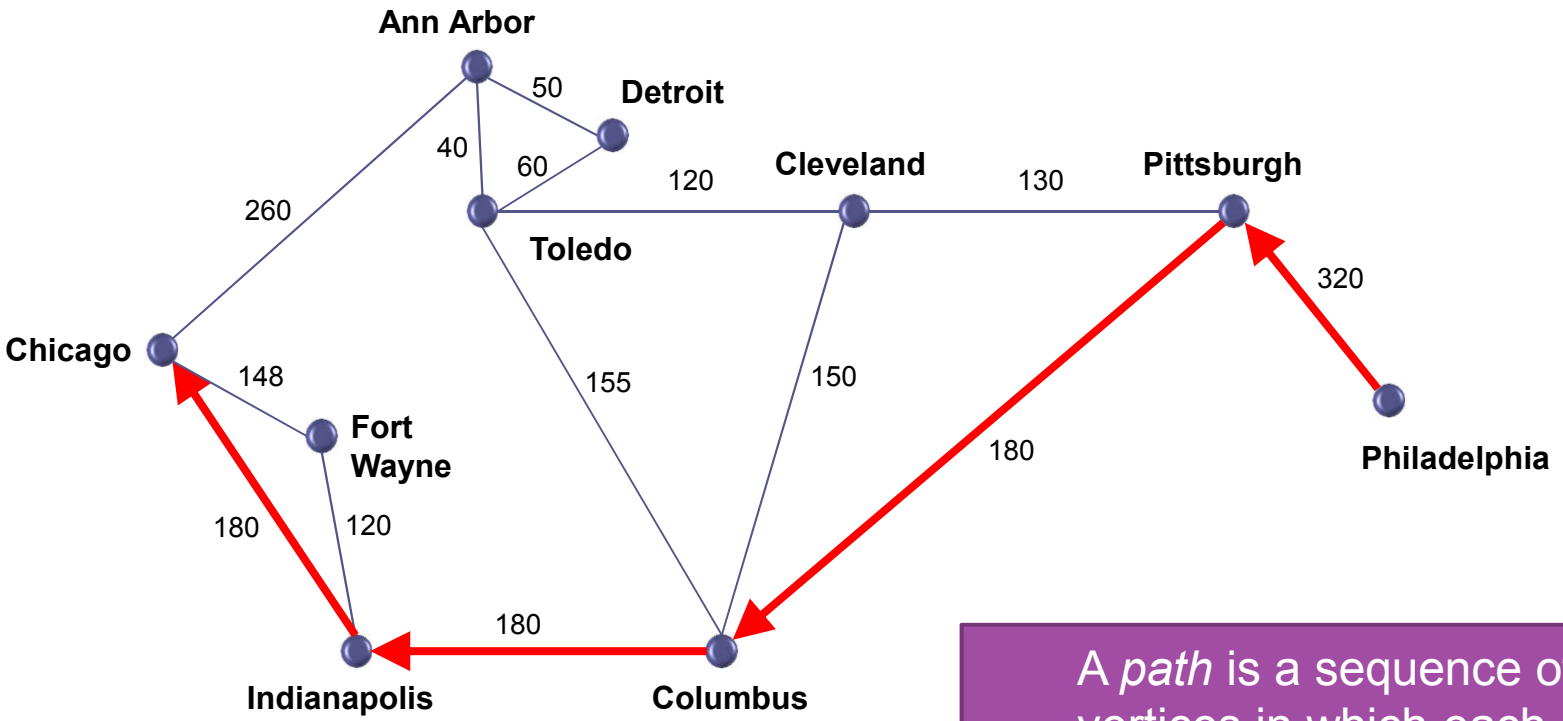
A vertex is *adjacent* to another vertex if there is an edge to it from that other vertex

Paths and Cycles (cont.)



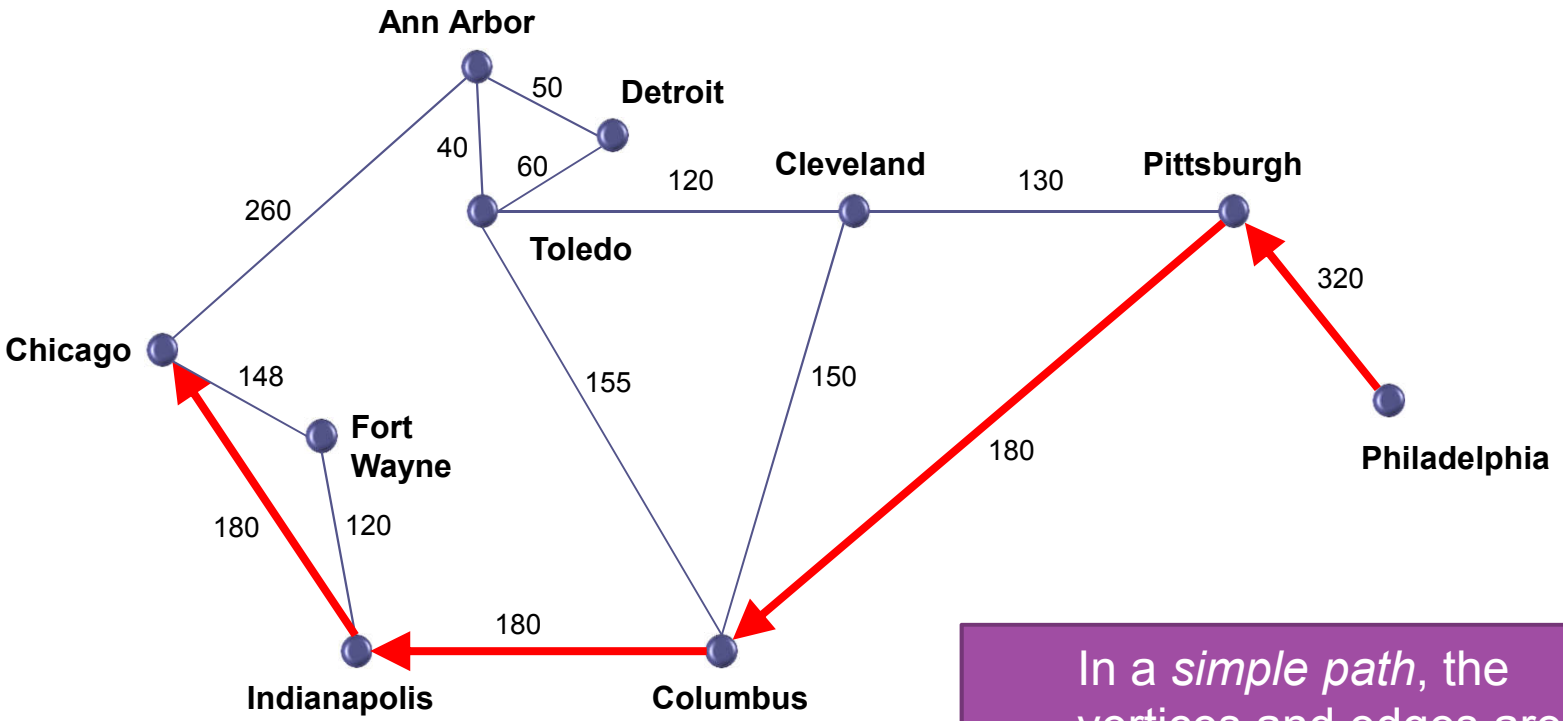
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Paths and Cycles (cont.)



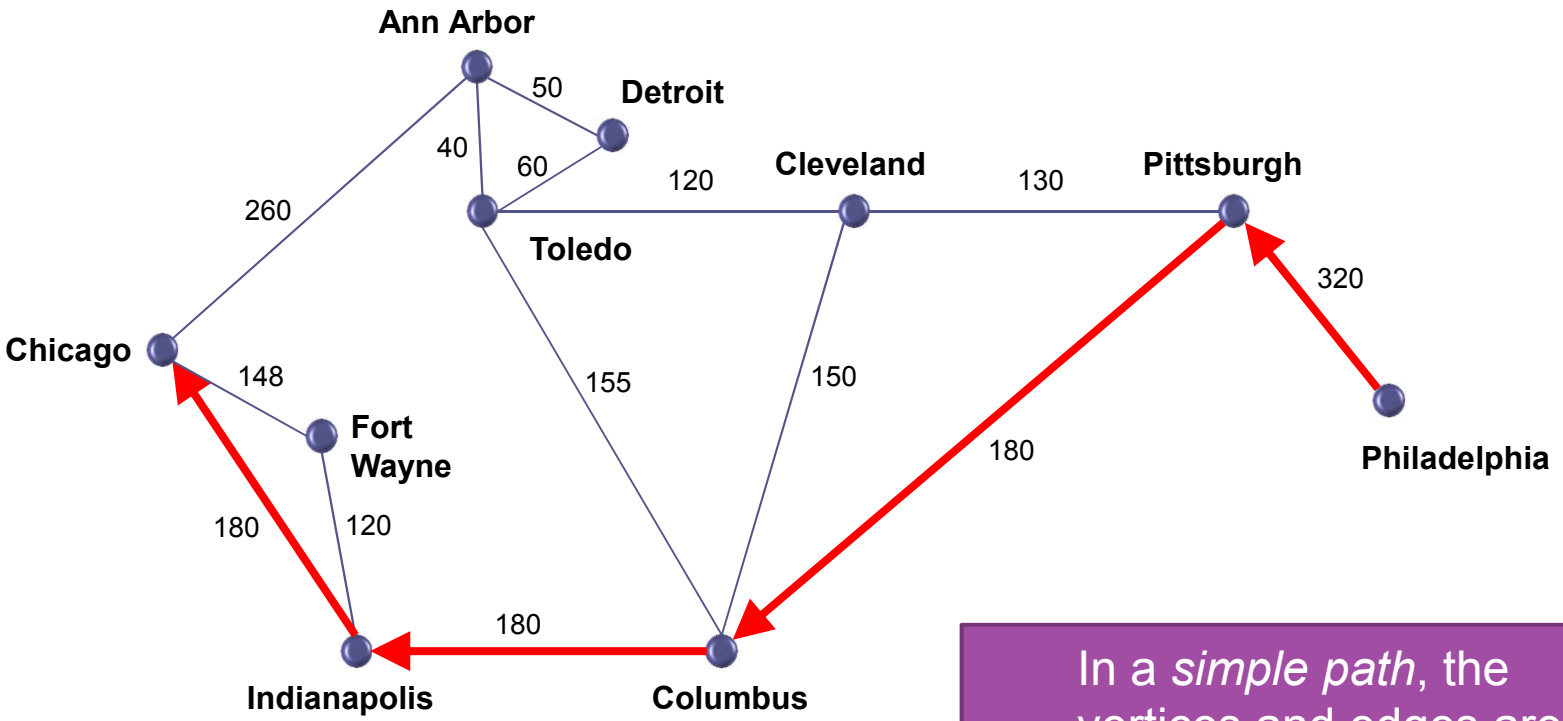
A *path* is a sequence of vertices in which each successive vertex is adjacent to its predecessor

Paths and Cycles (cont.)



In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

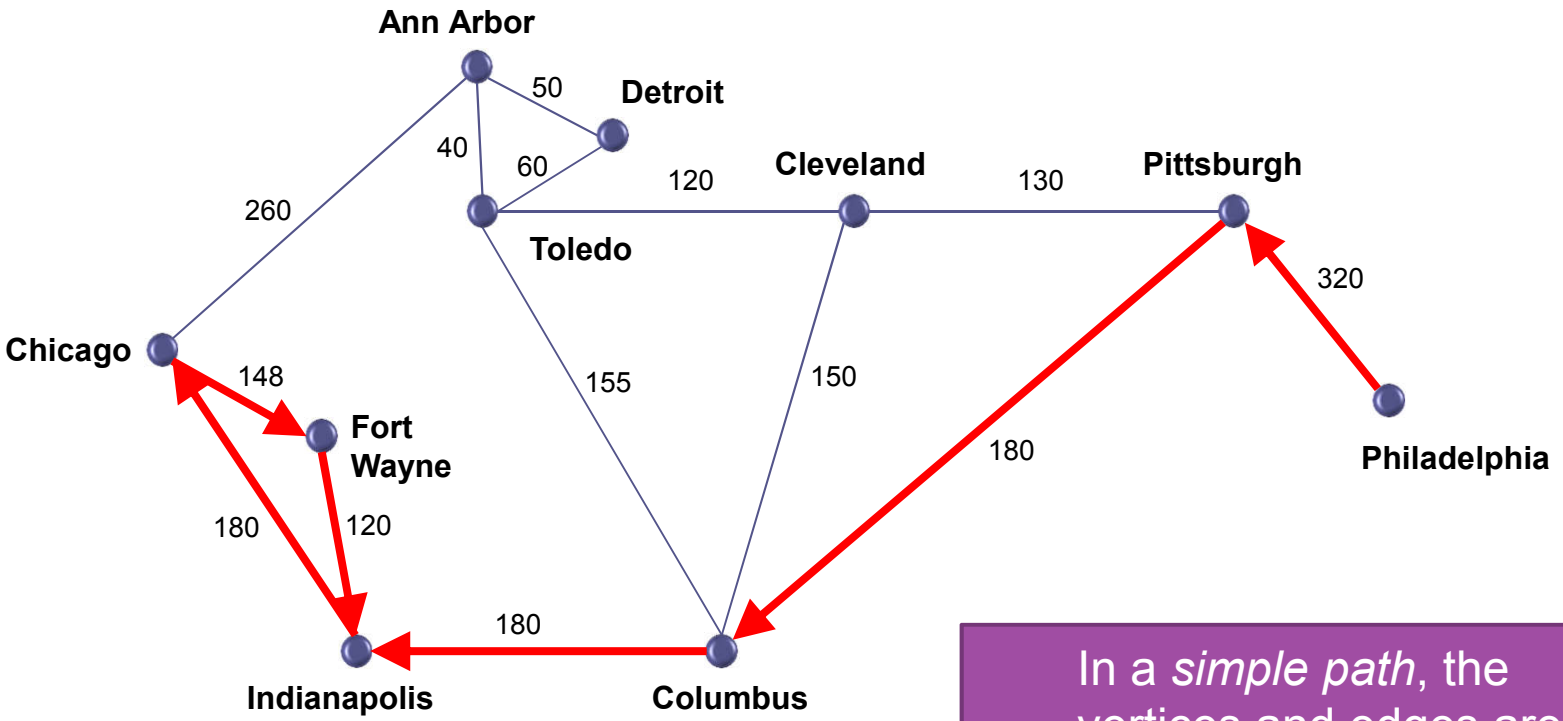
Paths and Cycles (cont.)



This path is a simple path

In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

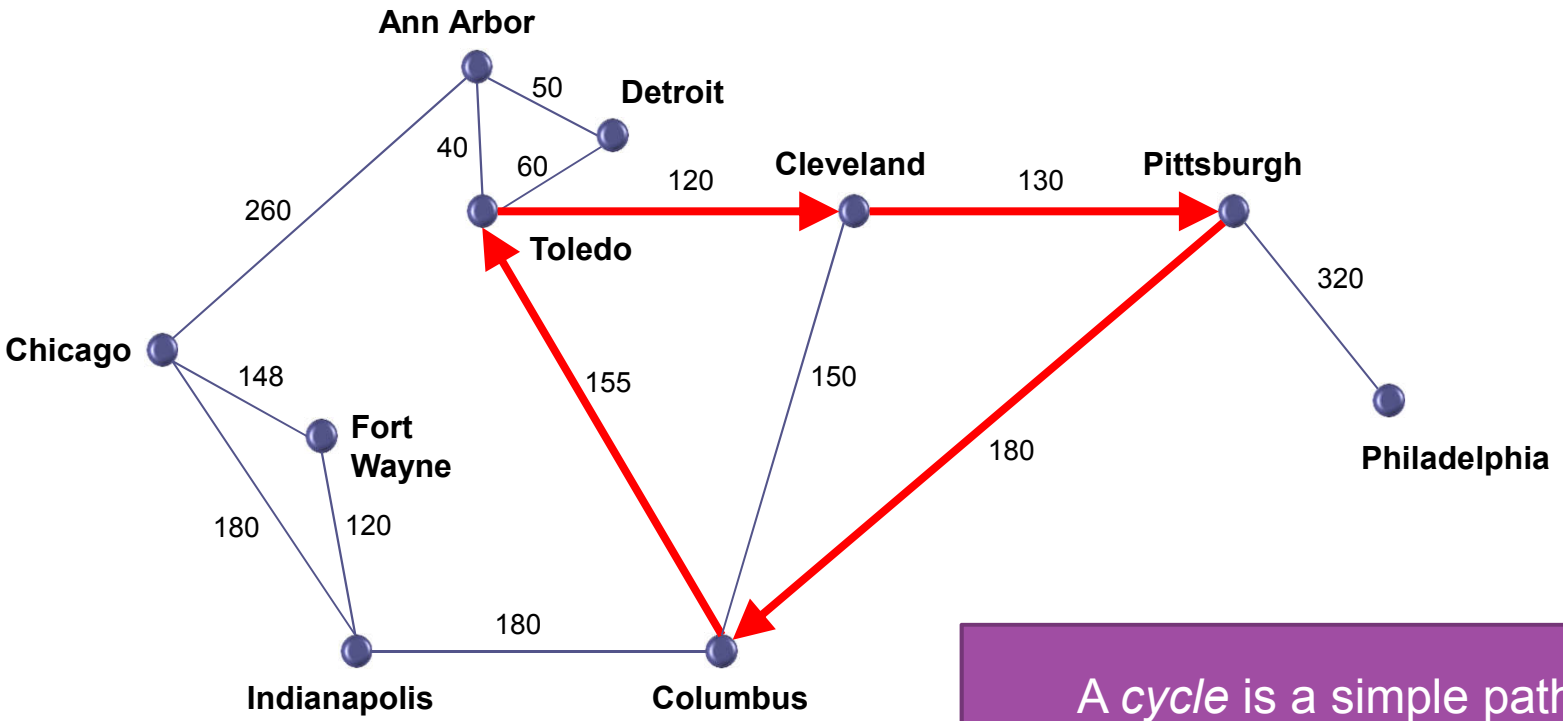
Paths and Cycles (cont.)



This path is NOT a simple path

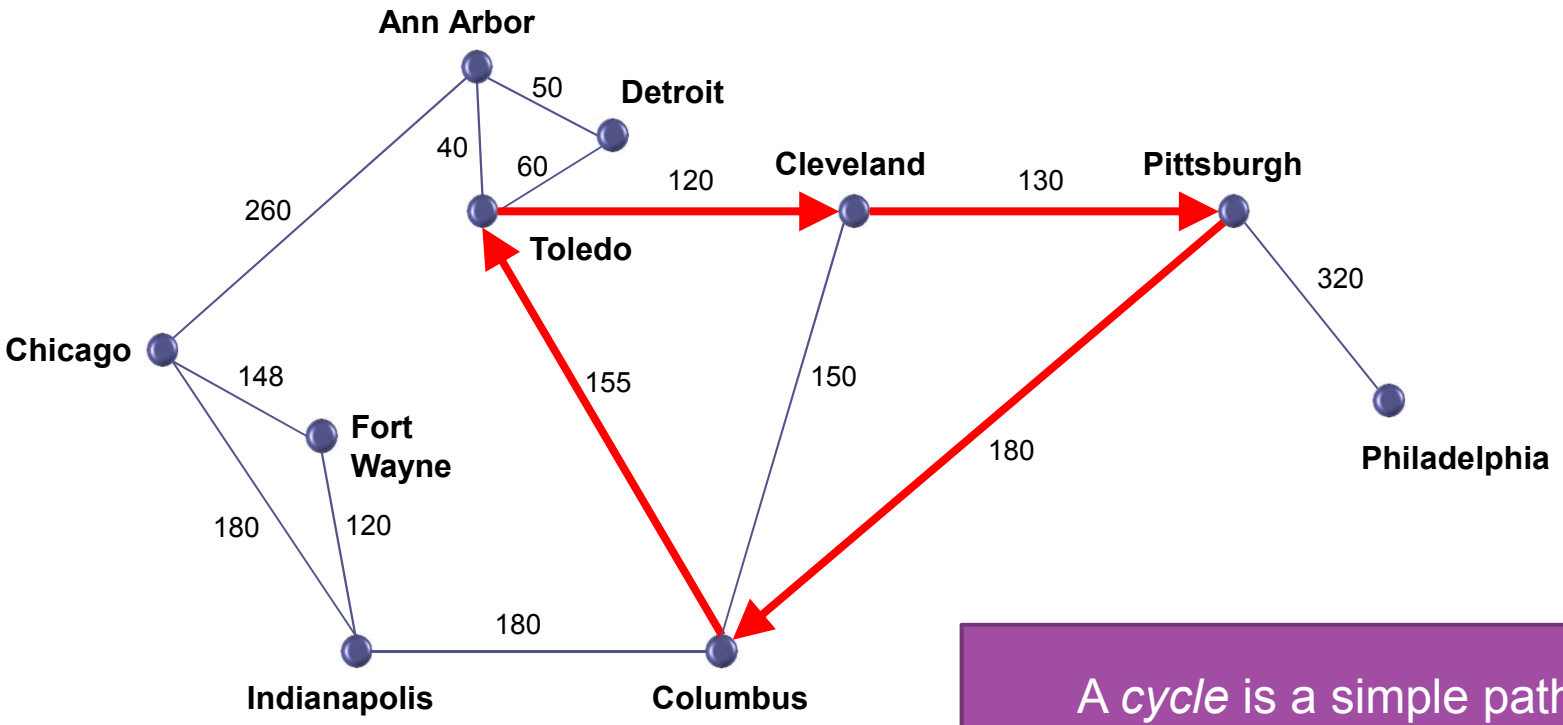
In a *simple path*, the vertices and edges are distinct except that the first and last vertex may be the same

Paths and Cycles (cont.)



A *cycle* is a simple path in which only the first and final vertices are the same

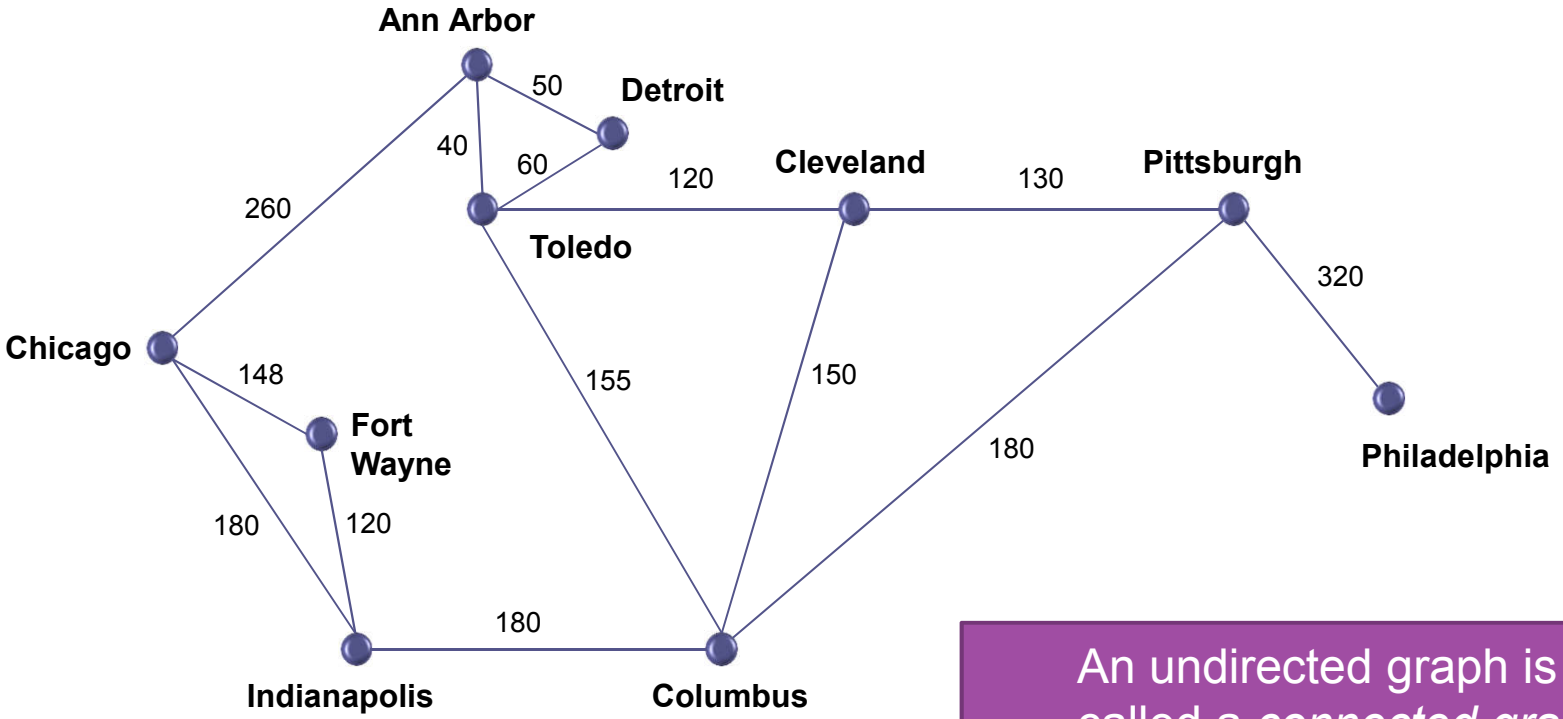
Paths and Cycles (cont.)



In an undirected graph a cycle must contain at least three distinct vertices
Pittsburgh → Columbus → Pittsburgh is not a cycle

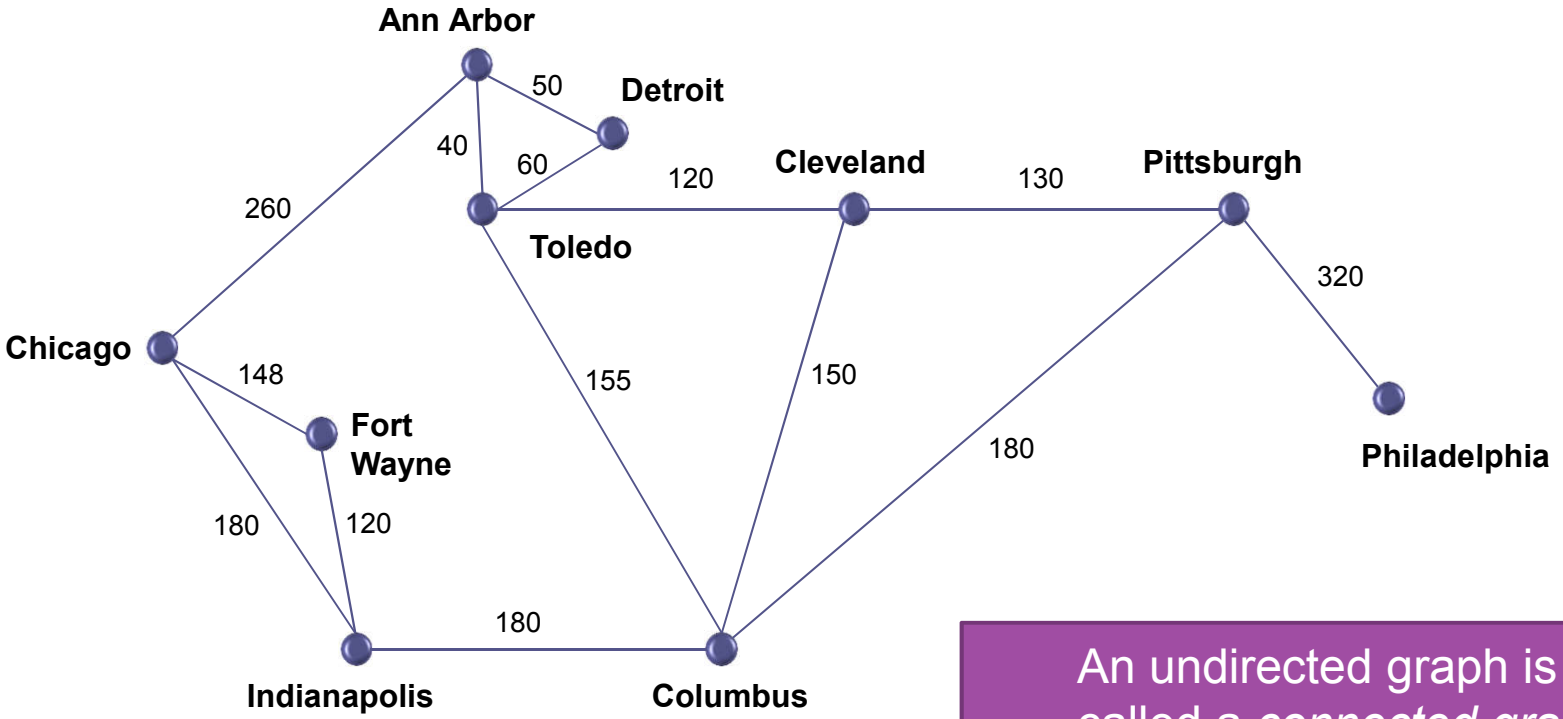
A *cycle* is a simple path in which only the first and final vertices are the same

Paths and Cycles (cont.)



An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

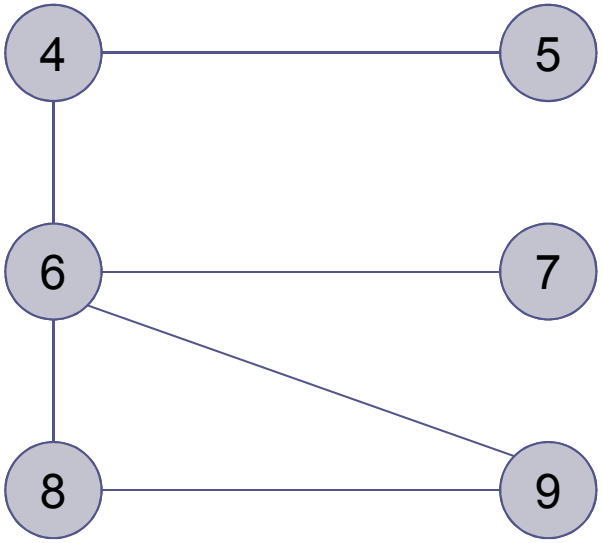
Paths and Cycles (cont.)



This graph is a connected graph

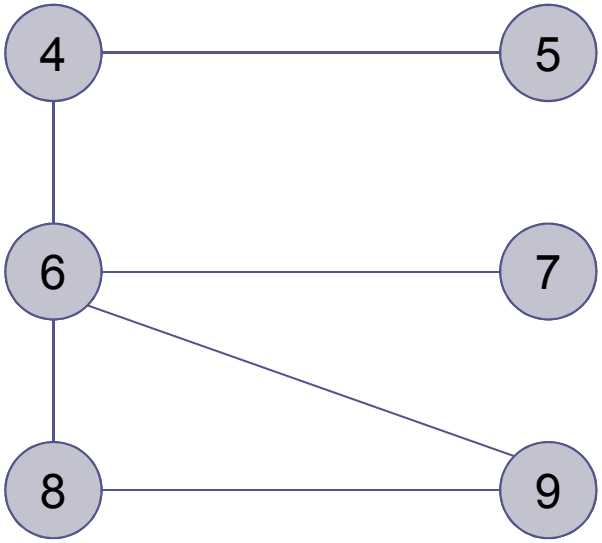
An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

Paths and Cycles (cont.)



An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

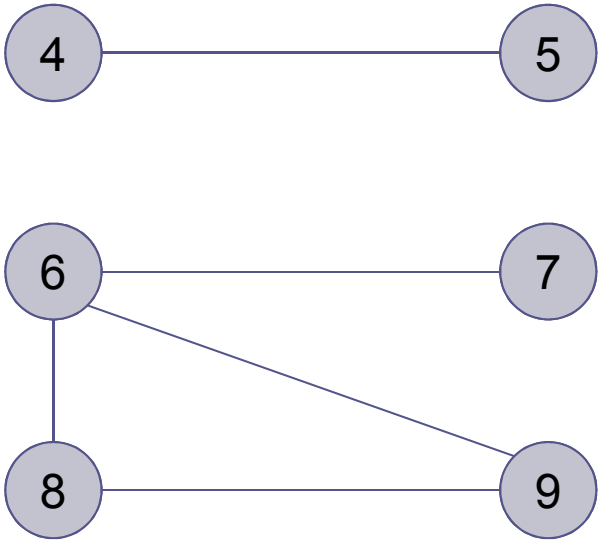
Paths and Cycles (cont.)



This graph is a connected graph

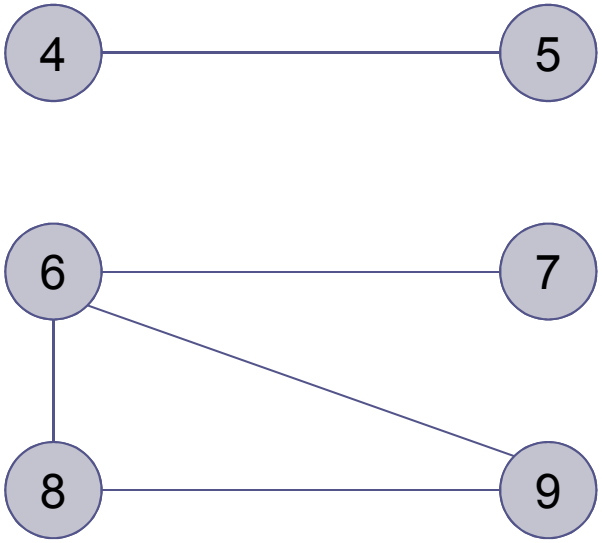
An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

Paths and Cycles (cont.)



An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

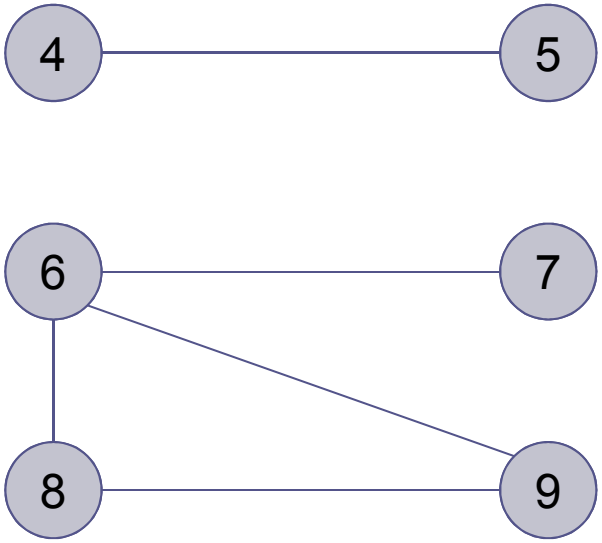
Paths and Cycles (cont.)



This graph is NOT a connected graph

An undirected graph is called a *connected graph* if there is a path from every vertex to every other vertex

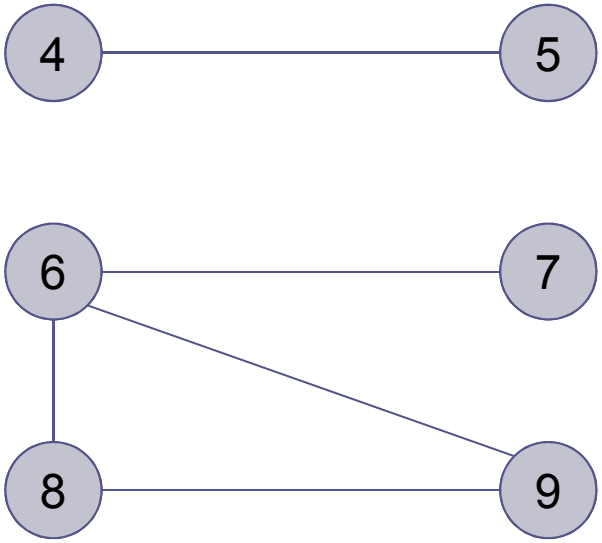
Paths and Cycles (cont.)



If a graph is not connected, it is considered *unconnected*, but still consists of *connected components*

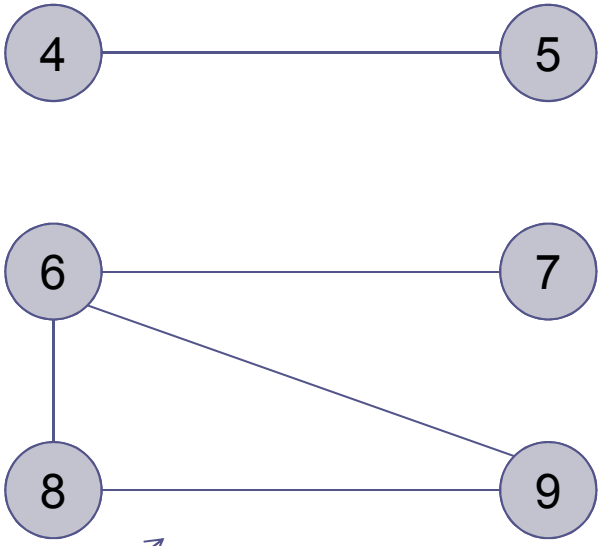
Paths and Cycles (cont.)

{4, 5} are connected components



If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

Paths and Cycles (cont.)

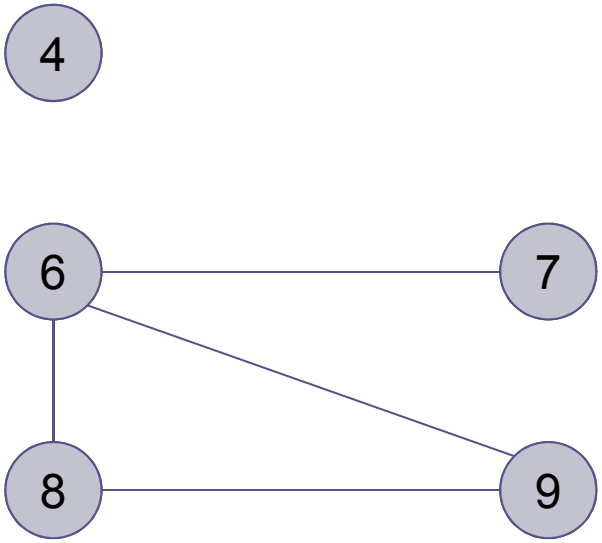


{6, 7, 8, 9} are connected components

If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

Paths and Cycles (cont.)

A single vertex with no edge is also considered a connected component

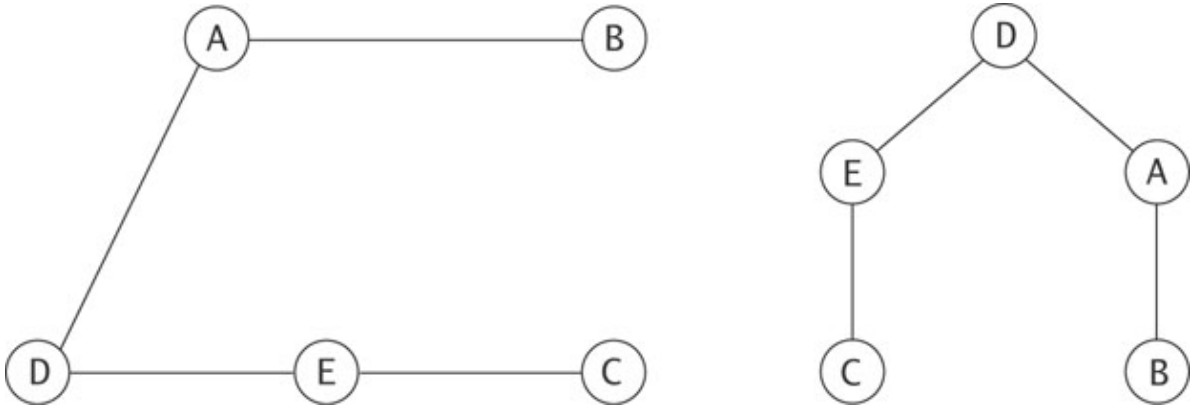


If a graph is not connected, it is considered *unconnected*, but will still consist of *connected components*

Relationship between Graphs and Trees

- A tree is a special case of a graph
- Any graph that is
  - connected
  - contains no cycles

  can be viewed as a tree by making one of the vertices the root



Graph Applications

- Graphs can be used to:
  - determine if one node in a network is connected to all the others
  - map out multiple course prerequisites (a solution exists if the graph is a directed graph with no cycles)
  - find the shortest route from one city to another (least cost or shortest path in a weighted graph)

The Graph ADT and Edge Class

Section 10.2

The Graph ADT and Edge Class

- Java does not provide a Graph ADT
- To make our own, we need
  1. Create a graph with the specified number of vertices
  2. Iterate through all of the vertices in the graph
  3. Iterate through the vertices that are adjacent to a specified vertex
  4. Determine whether an edge exists between two vertices
  5. Determine the weight of an edge between two vertices
  6. Insert an edge into the graph

The Graph AD (cont.)

```
    /** Insert a new edge into the graph.
        @param edge The new edge
    */
    void insert(Edge edge);

    /** Determine whether an edge exists.
        @param source The source vertex
        @param dest The destination vertex
        @return true if there is an edge from source to dest
    */
    boolean isEdge(int source, int dest);

    /** Get the edge between two vertices.
        @param source The source vertex
        @param dest The destination vertex
        @return The Edge between these two vertices
                or an Edge with a weight of
                Double.POSITIVE_INFINITY if there is no edge
    */
    Edge getEdge(int source, int dest);

    /** Return an iterator to the edges connected
        to a given vertex.
        @param source The source vertex
        @return An Iterator<Edge> to the vertices
                connected to source
    */
    Iterator<Edge> edgeIterator(int source);

}
```

Representing Vertices and Edges

- Vertices
  - We can represent the vertices by integers (`int` variable) from 0 up to, but not including, |V|
    - (|V| means the *cardinality* of V, or the number of vertices in set V)

Representing Vertices and Edges (cont.)

- Edges
  - Define the class `Edge` that will contain the
    - source vertex
    - destination vertex
    - weight (unweighted edges use the default value of 1.0)
  - Edges are directed
  - Undirected graphs will have two `Edge` objects for each edge: one in each direction

The Graph ADT and Edge Class (cont.)

Data Field	Attribute
`private int dest`	The destination vertex for an edge.
`private int source`	The source vertex for an edge.
`private double weight`	The weight.
Constructor	**Purpose**
`public Edge(int source, int dest)`	Constructs an `Edge` from `source` to `dest`. Sets the `weight` to 1.0.
`public Edge(int source, int dest, double w)`	Constructs an `Edge` from `source` to `dest`. Sets the `weight` to `w`.
Method	**Behavior**
`public boolean equals(Object o)`	Compares two edges for equality. Edges are equal if their source and destination vertices are the same. The weight is not considered.
`public int getDest()`	Returns the destination vertex.
`public int getSource()`	Returns the source vertex.
`public double getWeight()`	Returns the weight.
`public int hashCode()`	Returns the hash code for an edge. The hash code depends only on the source and destination.
`public String toString()`	Returns a string representation of the edge.

Implementing the Graph ADT

Section 10.3

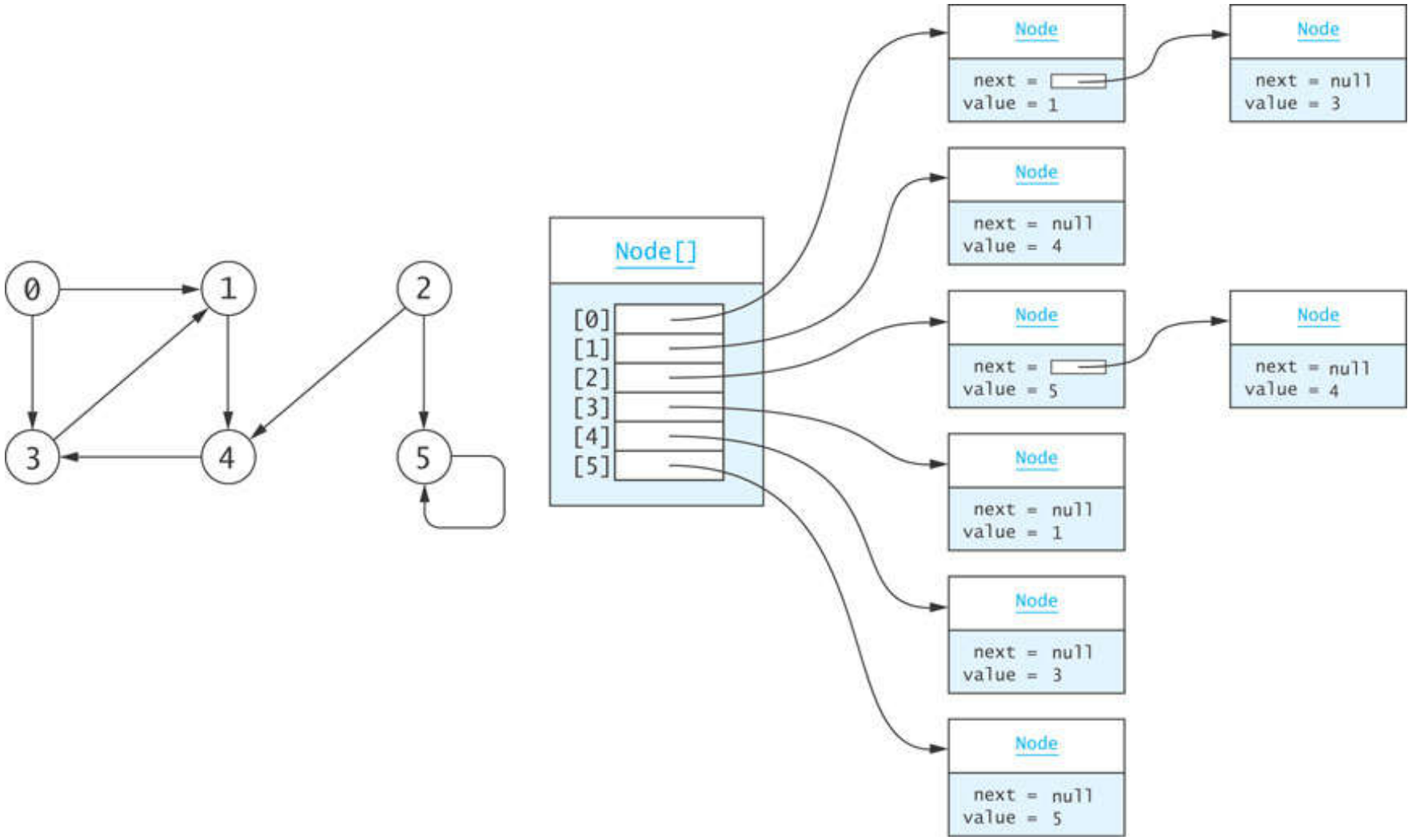
Implementing the Graph ADT

- Many of the original publications of graph algorithms and their implementations did not use an object-oriented approach or even abstract data types
- Two representations of graphs are most common
  - Edges are represented by an array of lists called *adjacency lists*, where each list stores the vertices adjacent to a particular vertex
  - Edges are represented by a two dimensional array, called an *adjacency matrix*, with |V| rows and |V| columns

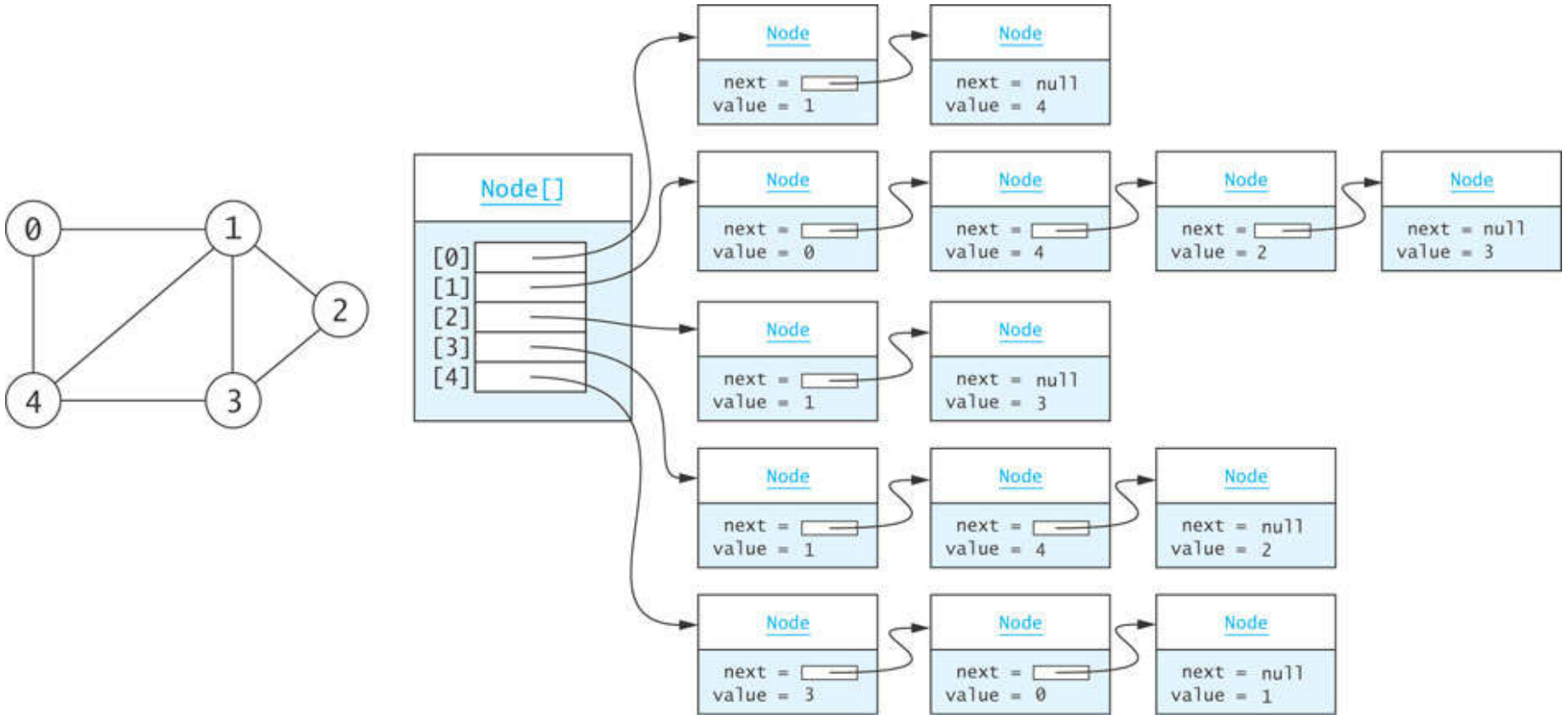
Adjacency List

- An adjacency list representation of a graph uses an array of lists - one list for each vertex
- The vertices are in no particular order

Adjacency List – Directed Graph Example



Adjacency List – Undirected Graph Example



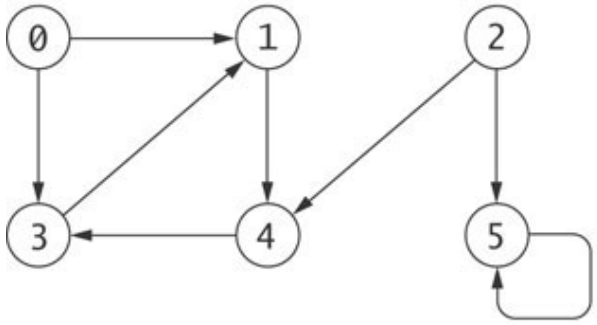
Adjacency Matrix

- Use a two-dimensional array to represent the graph
- For an unweighted graph, the entries can be `boolean` or `int` values:
  - `true` or `1`, edge exists
  - `false` or `0`, no edge
- Integer values have benefits over boolean values for some graph algorithms that use matrix multiplication

Adjacency Matrix (cont.)

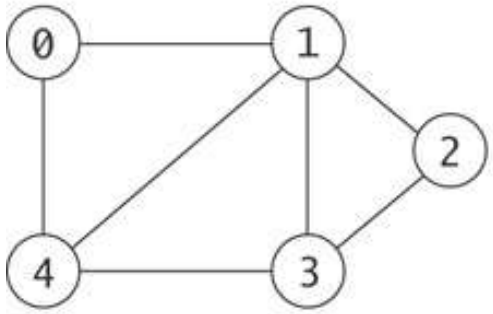
- For a weighted graph, the matrix would contain the weights
  - Since 0 is a valid weight, the Java value `Double.POSITIVE_INFINITY` can represent the absence of an edge
  - An unweighted graph would contain the value 1.0 for the presence of an edge
- In an undirected graph, the matrix is symmetric, so only the lower triangle of the matrix needs to be saved (an example is on next slide)

Adjacency Matrix (cont.)



Column

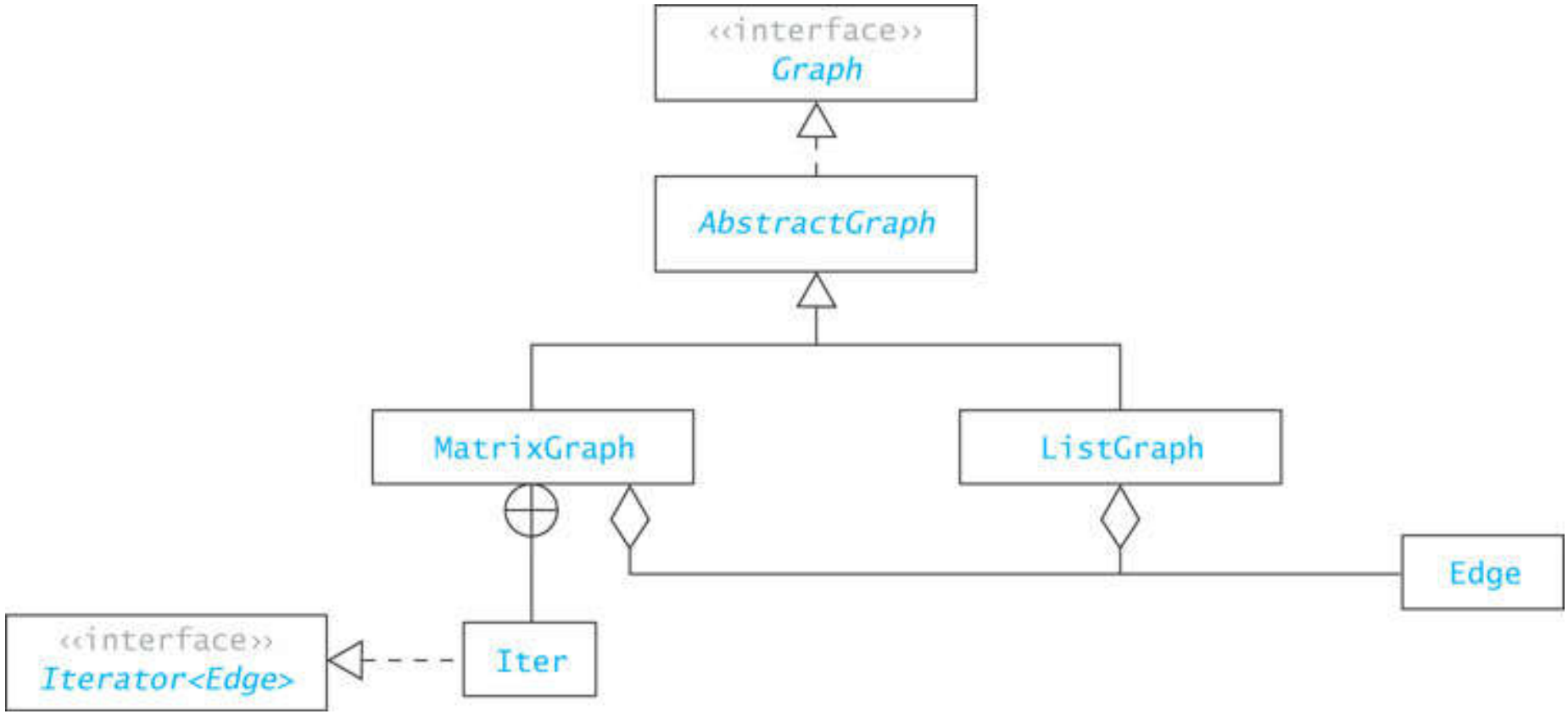
Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		1.0		
[1]					1.0	
[2]					1.0	1.0
[3]		1.0				
[4]				1.0		
[5]						1.0



Column

Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

Overview of the Graph Class Hierarchy



Class `AbstractGraph`

Data Field	Attribute
`private boolean directed`	**`true`** if this is a directed graph.
`private int numV`	The number of vertices.
Constructor	**Purpose**
`public AbstractGraph(int numV, boolean directed)`	Constructs an empty graph with the specified number of vertices and with the specified `directed` flag. If `directed` is **`true`**, this is a directed graph.
Method	**Behavior**
`public int getNumV()`	Gets the number of vertices.
`public boolean isDirected()`	Returns **`true`** if the graph is a directed graph.
`public void loadEdgesFromFile(Scanner scan)`	Loads edges from a data file.
`public static Graph createGraph (Scanner scan, boolean isDirected, String type)`	Factory method to create a graph and load the data from an input file.

Implementation

```
public abstract class AbstractGraph implements Graph {
    // Data Fields
    /** The number of vertices */
    private int numV;
    /** Flag to indicate whether this is a directed graph */
    private boolean directed;

public AbstractGraph(int numV, boolean directed) {
        this.numV = numV;
        this.directed = directed;
    }

@Override
    public int getNumV() {
        return numV;
    }

@Override
    public boolean isDirected() {
        return directed;
    }
```

Implementation

```
public static Graph createGraph(Scanner scan,
            boolean isDirected,
            String type) throws IOException {
        int numV = scan.nextInt();
        scan.nextLine();
        AbstractGraph returnValue = null;
        if (type.equalsIgnoreCase("Matrix")) {
            returnValue = new MatrixGraph(numV, isDirected);
        } else if (type.equalsIgnoreCase("List")) {
            returnValue = new ListGraph(numV, isDirected);
        } else {
            throw new IllegalArgumentException();
        }
        returnValue.loadEdgesFromFile(scan);
        return returnValue;
    }
```

`ListGraph` Class

Data Field	Attribute
`private List<Edge>[] edges`	An array of `Lists` to contain the edges that originate with each vertex.
Constructor	**Purpose**
`public ListGraph(int numV, boolean directed)`	Constructs a graph with the specified number of vertices and directionality.
Method	**Behavior**
`public Iterator<Edge> edgeIterator(int source)`	Returns an iterator to the edges that originate from a given vertex.
`public Edge getEdge(int source, int dest)`	Gets the edge between two vertices.
`public void insert(Edge e)`	Inserts a new edge into the graph.
`public boolean isEdge(int source, int dest)`	Determines whether an edge exists from vertex `source` to `dest`.

Data Fields

```
import java.util.*;

/** A ListGraph is an extension of the AbstractGraph abstract class
    that uses an array of lists to represent the edges.
*/
public class ListGraph extends AbstractGraph {

    // Data Field
    /** An array of Lists to contain the edges that
       originate with each vertex. */
    private List<Edge>[] edges;
. . .
```

Constructor

```
/** Construct a graph with the specified number of
    vertices and directionality.
    @param numV The number of vertices
    @param directed The directionality flag
*/
public ListGraph(int numV, boolean directed) {
    super(numV, directed);
    edges = new List[numV];
    for (int i = 0; i < numV; i++) {
        edges[i] = new LinkedList<Edge>();
    }
}
```

`isEdge` Method

```
/** Determine whether an edge exists.
    @param source The source vertex
    @param dest The destination vertex
    @return true if there is an edge from source to dest
*/
public boolean isEdge(int source, int dest) {
    return edges[source].contains(new Edge(source, dest));
}
```

`insert` Method

```
/** Insert a new edge into the graph.
    @param edge The new edge
*/
public void insert(Edge edge) {
    edges[edge.getSource()].add(edge);
    if (!isDirected()) {
       edges[edge.getDest()].add(new Edge(edge.getDest(),
                                          edge.getSource(),
                                          edge.getWeight()));
    }
}
```

`edgeIterator` Method

```
public Iterator<Edge> edgeIterator(int source) {
    return edges[source].iterator();
}
```

`getEdge` Method

```
/** Get the edge between two vertices. If an
    edge does not exist, an Edge with a weight
    of Double.POSITIVE_INFINITY is returned.
    @param source The source
    @param dest The destination
    @return the edge between these two vertices
*/
public Edge getEdge(int source, int dest) {
    Edge target =
        new Edge(source, dest, Double.POSITIVE_INFINITY);
    for (Edge edge : edges[source]) {
        if (edge.equals(target))
            return edge; // Desired edge found, return it.
    }
    // Assert: All edges for source checked.
    return target; // Desired edge not found.
}
```

`MatrixGraph` Class

- The `MatrixGraph` class extends the `AbstractGraph` class by providing a two-dimensional array for storing edge weights

```
double[][] edges;
```

- Upon creation of a `MatrixGraph` class, the constructor sets the number of rows (vertices)
- It also needs is own iterator class—which is left as a project (Programming Project 1)

Comparing Implementations

- Time efficiency depends on the algorithm and the density of the graph
- The density of a graph is the ratio of |E| to $|V|^2$
  - A *dense graph* is one in which |E| is close to, but less than $|V|^2$
  - A *sparse graph* is one in which |E| is much less than $|V|^2$
- We can assume that |E| is
  - $O(|V|^2)$ for a dense graph
  - O(|V|) for a sparse graph

Comparing Implementations (cont.)

Many graph algorithms are of the form:

1. `for` each vertex *u* in the graph
2. `for` each vertex *v* adjacent to *u*
3. Do something with edge (*u*, *v*)

- For an adjacency list
  - Step 1 is O(|V|)
  - Step 2 is O($|E_u|$)
    - E_u is the number of edges that originate at vertex *u*
- The combination of Steps 1 and 2 represents examining each edge in the graph, giving O(|E|)

Comparing Implementations (cont.)

Many graph algorithms are of the form:

1. `for` each vertex *u* in the graph
2. `for` each vertex *v* adjacent to *u*
3. Do something with edge (*u*, *v*)

- For an adjacency matrix
  - Step 1 is O(|V|)
  - Step 2 is O(|V|)
- The combination of Steps 1 and 2 represents examining each edge in the graph, giving $O(|V^2|)$
- The adjacency list gives better performance in a sparse graph, whereas for a dense graph the performance is the same for both representations

Comparing Implementations (cont.)

Some graph algorithms are of the form:

1. `for` each vertex *u* in some subset of the vertices
2. `for` each vertex *v* in some subset of the vertices
3. if (*u*, *v*) is an edge
4. Do something with edge (*u*, *v*)

- For an adjacency matrix representation,
  - Step 3 tests a matrix value and is O(1)
  - The overall algorithm is $O(|V^2|)$

Comparing Implementations (cont.)

Some graph algorithms are of the form:

1. `for` each vertex *u* in some subset of the vertices
2. `for` each vertex *v* in some subset of the vertices
3. if (*u*, *v*) is an edge
4. Do something with edge (*u*, *v*)

- For an adjacency list representation,
  - Step 3 searches a list and is $O(|E_u|)$
  - So the combination of Steps 2 and 3 is $O(|E|)$
  - The overall algorithm is $O(|V||E|)$

Comparing Implementations (cont.)

Some graph algorithms are of the form:

```
1.  for each vertex u in some subset of the vertices
2.      for each vertex v in some subset of the vertices
3.          if (u, v) is an edge
4.              Do something with edge (u, v)
```

- For a dense graph, the adjacency matrix gives better performance
- For a sparse graph, the performance is the same for both representations

Comparing Implementations (cont.)

- Thus, for time efficiency,
  - if the graph is dense, the adjacency matrix representation is better
  - if the graph is sparse, the adjacency list representation is better
- A sparse graph will lead to a sparse matrix, or one with many POSITIVE_INFINITY entries
- These values are not included in a list representation so they have no effect on the processing time

Comparing Implementations (cont.)

- In an adjacency matrix,
  - storage is allocated for all vertex combinations (or at least half of them)
  - the storage required is proportional to $|V|^2$
  - for a sparse graph, there is a lot of wasted space
- In an adjacency list,
  - each edge is represented by a reference to an `Edge` object which contains data about the source, destination, and weight
  - there is also a reference to the next edge in the list
  - this is four times as much information as is stored in a matrix representation (which stores only the weight)

Comparing Implementations (cont.)

- The break-even point in terms of storage efficiency occurs when approximately 25% of the adjacency matrix is filled with meaningful data

Traversals of Graphs

Section 10.4

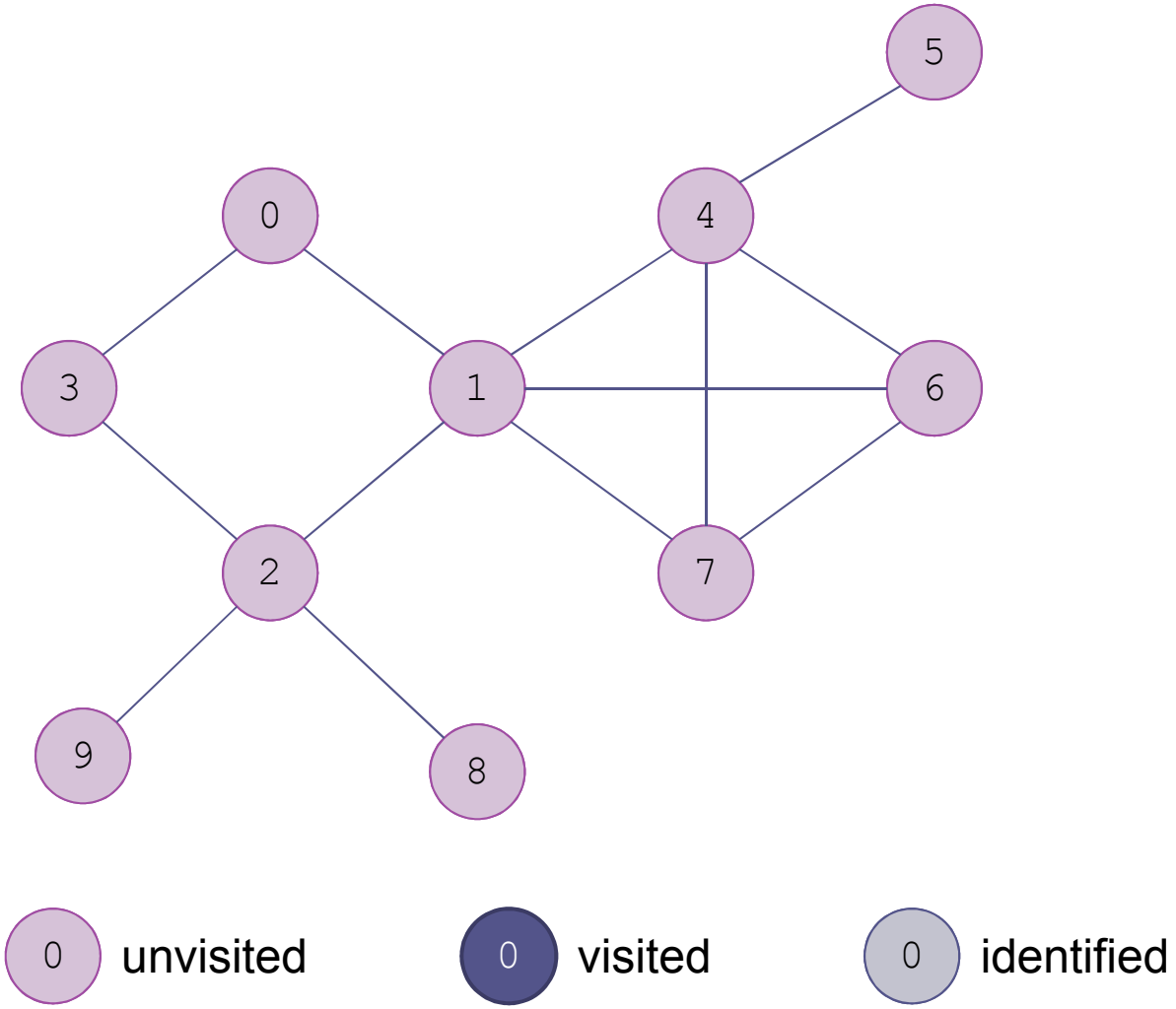
Traversals of Graphs

- Most graph algorithms involve visiting each vertex in a systematic order
- As with trees, there are different ways to do this
- The two most common traversal algorithms are the *breadth-first search* and the *depth-first search*

Breadth-First Search

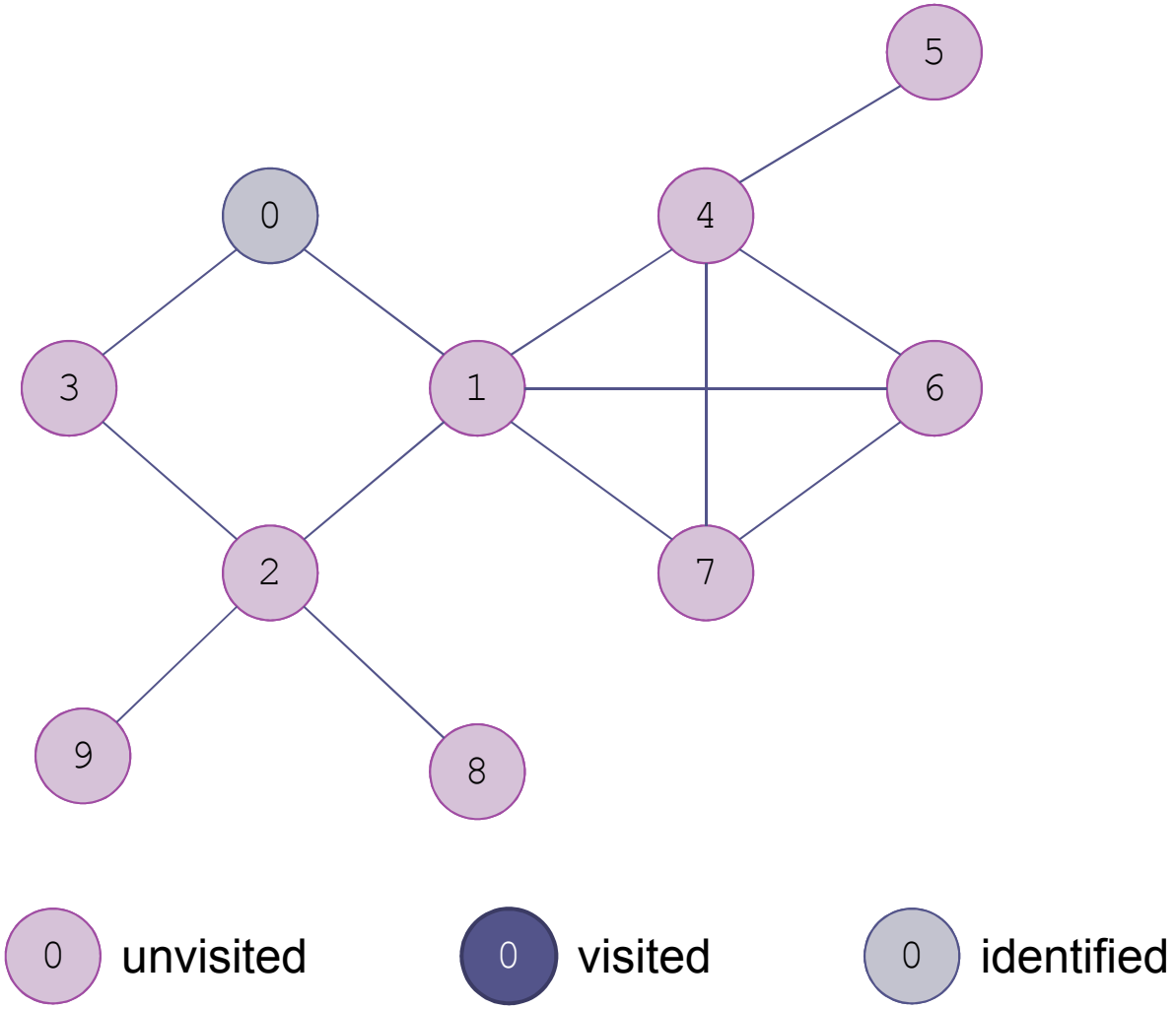
- In a breadth-first search,
  - visit the start node first,
  - then all nodes that are adjacent to it,
  - then all nodes that can be reached by a path from the start node containing two edges,
  - then all nodes that can be reached by a path from the start node containing three edges,
  - and so on
- We must visit all nodes for which the shortest path from the start node is length *k* before we visit any node for which the shortest path from the start node is length *k+1*
- There is no special start vertex– we arbitrarily choose the vertex with label 0

Example of a Breadth-First Search



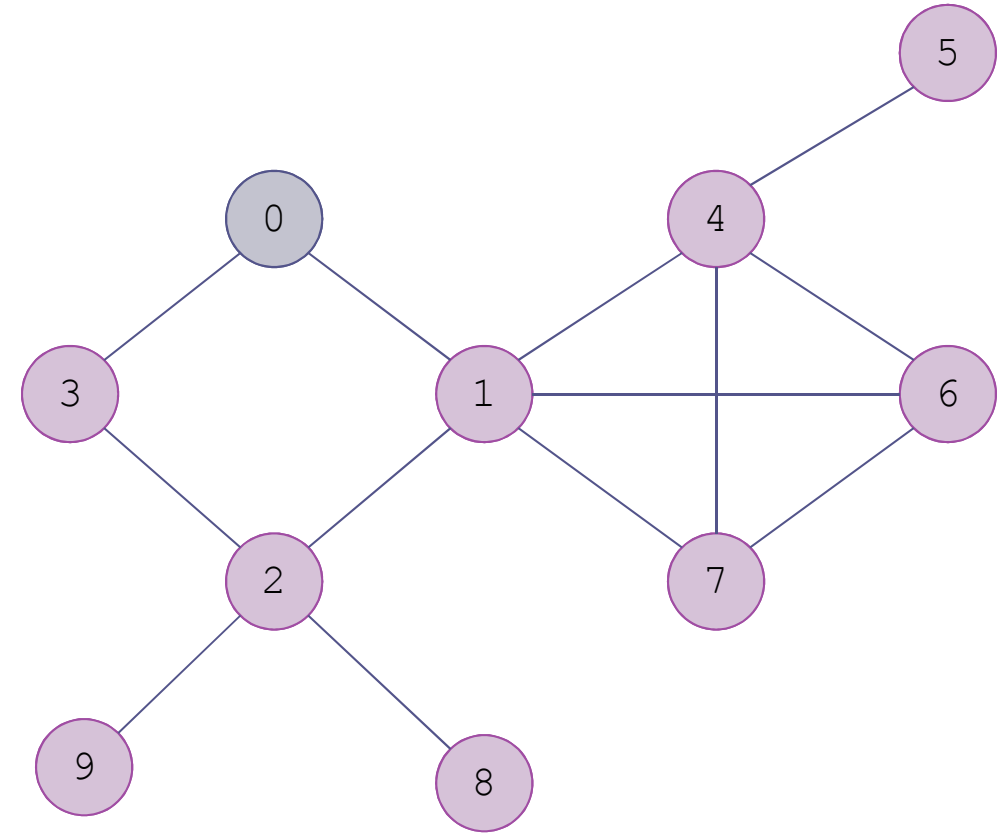
Example of a Breadth-First Search (cont.)

Identify the start node



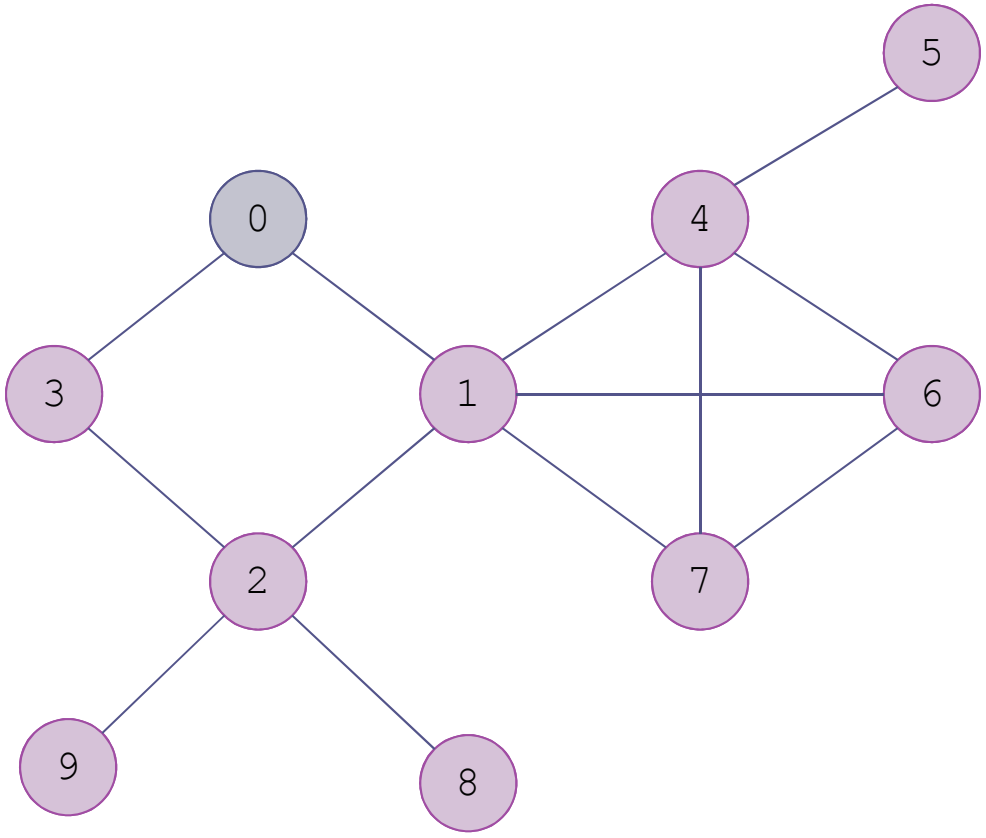
Example of a Breadth-First Search (cont.)

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



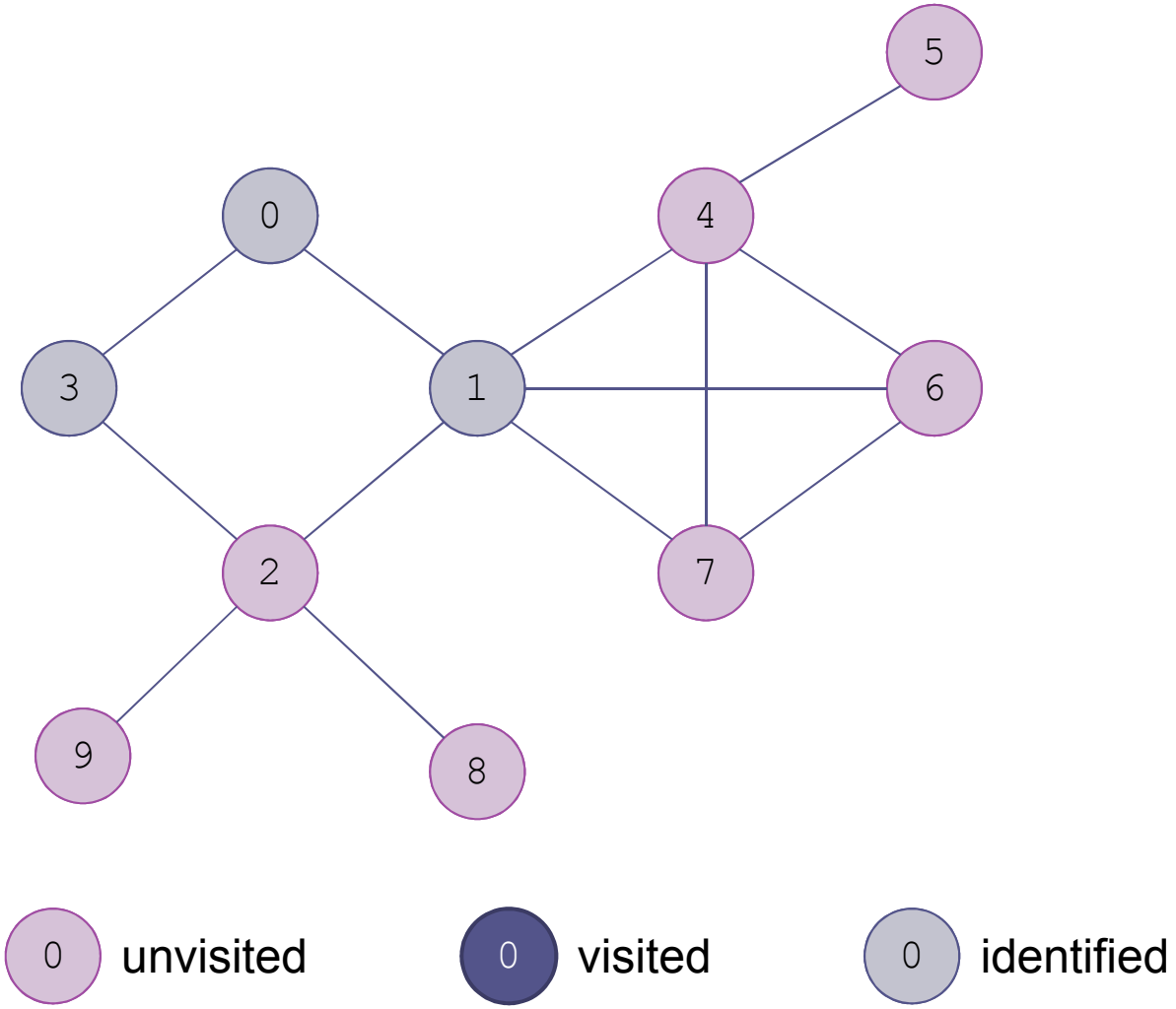
Visit sequence:
0

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

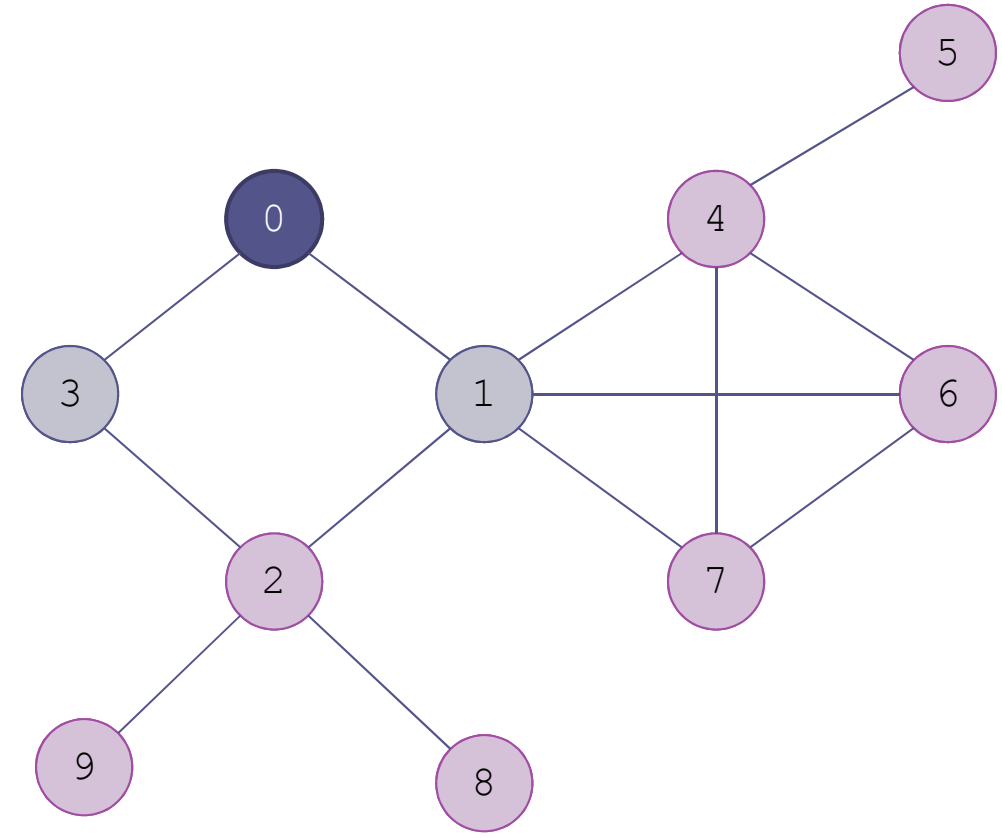


Example of a Breadth-First Search (cont.)

We color the node as visited

Queue:
1, 3

Visit sequence:
0

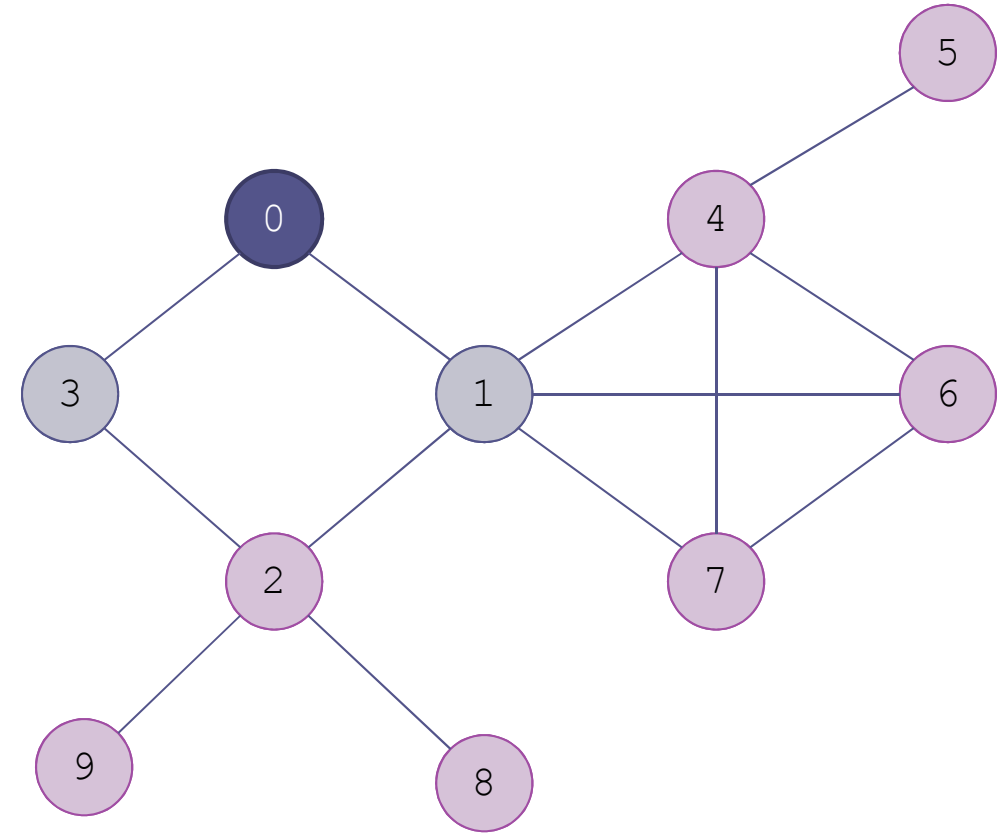


Example of a Breadth-First Search (cont.)

The queue determines which nodes to visit next

Queue:
1, 3

Visit sequence:
0

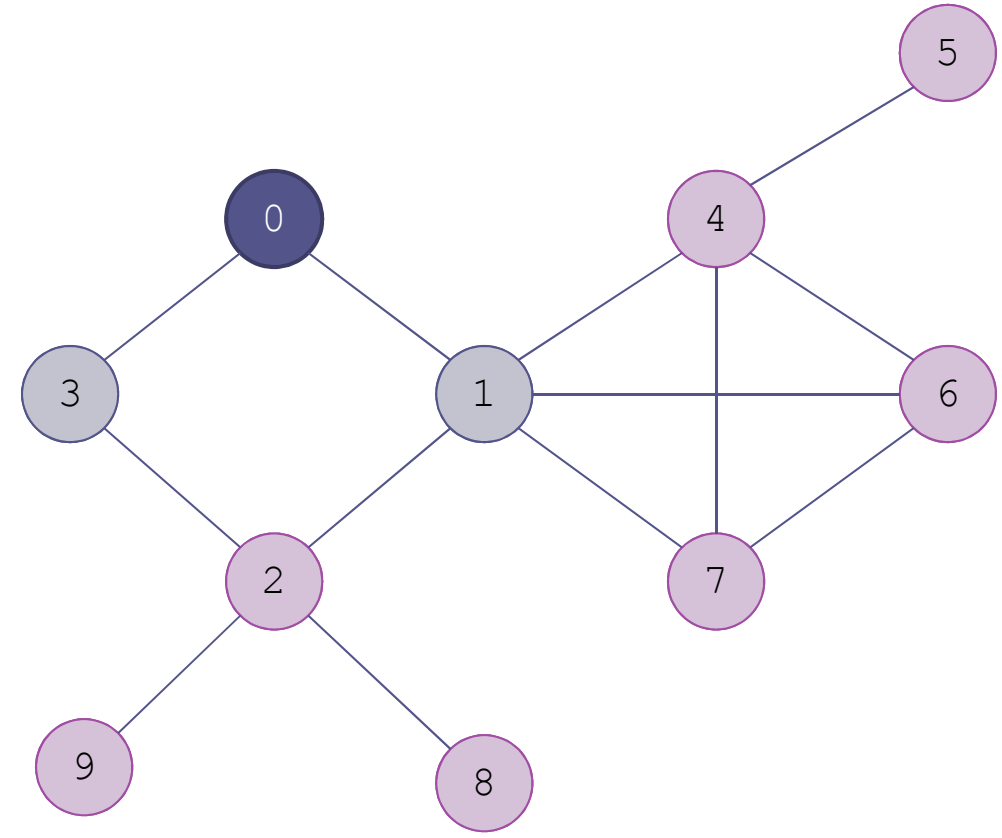


Example of a Breadth-First Search (cont.)

Visit the first node in the queue, 1

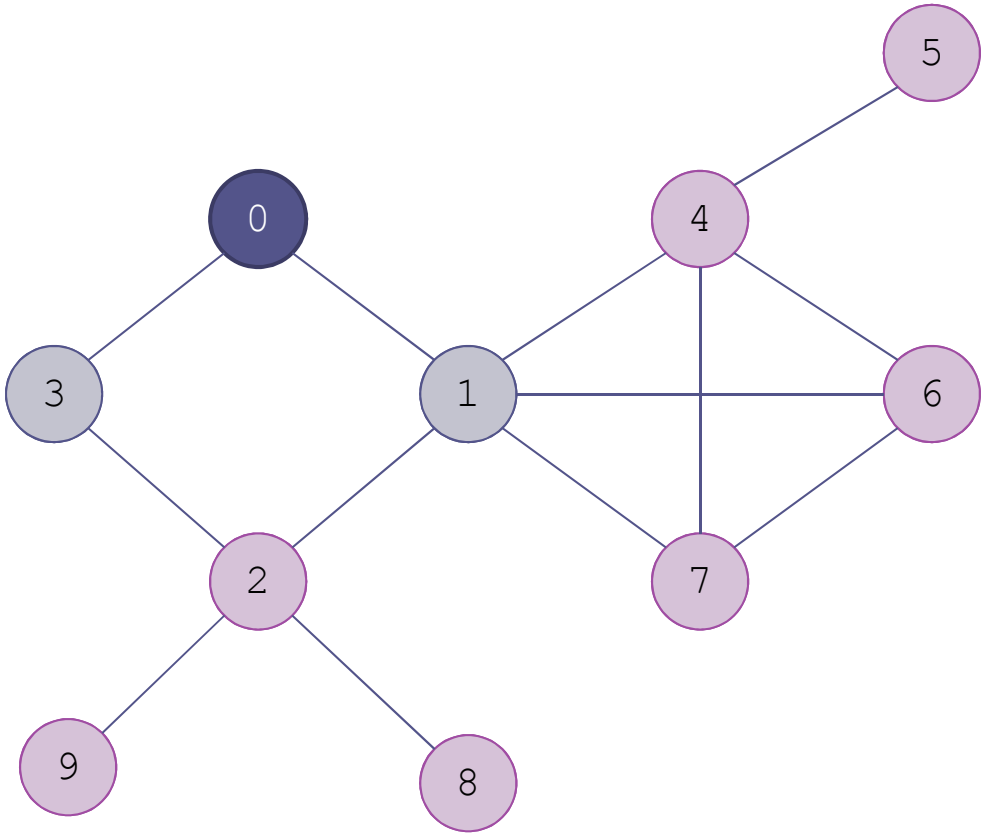
Queue:
1, 3

Visit sequence:
0



Example of a Breadth-First Search (cont.)

Visit the first node in the queue, 1



Queue:
3

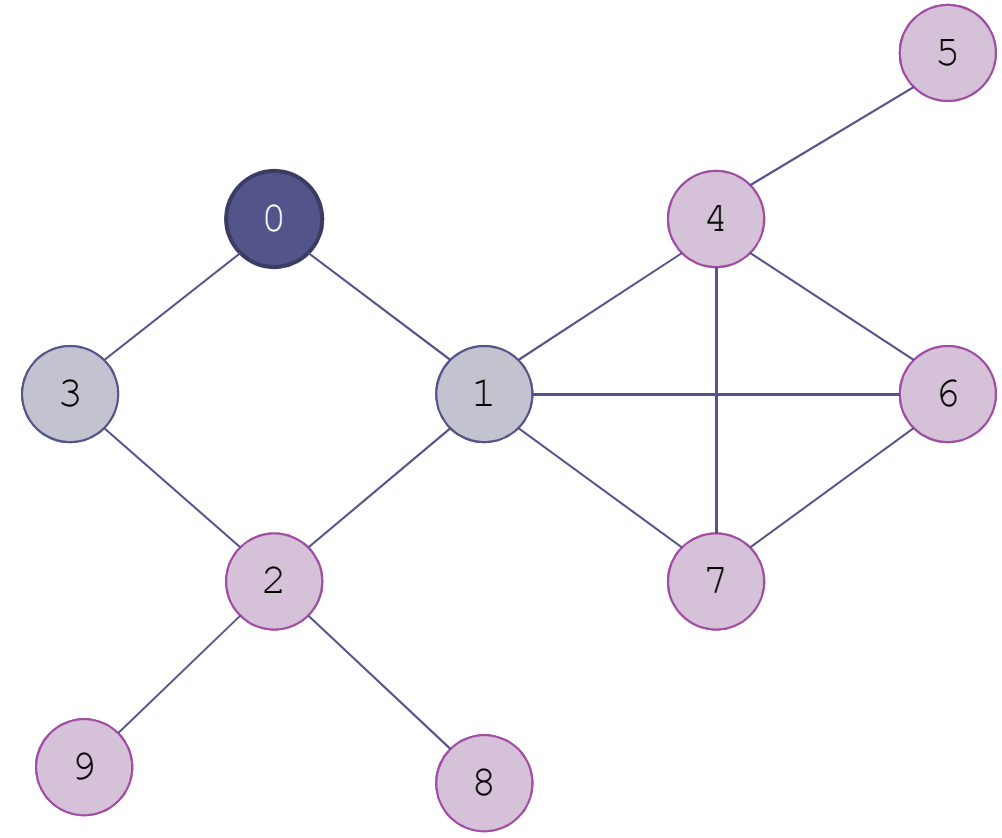
Visit sequence:
0, 1

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

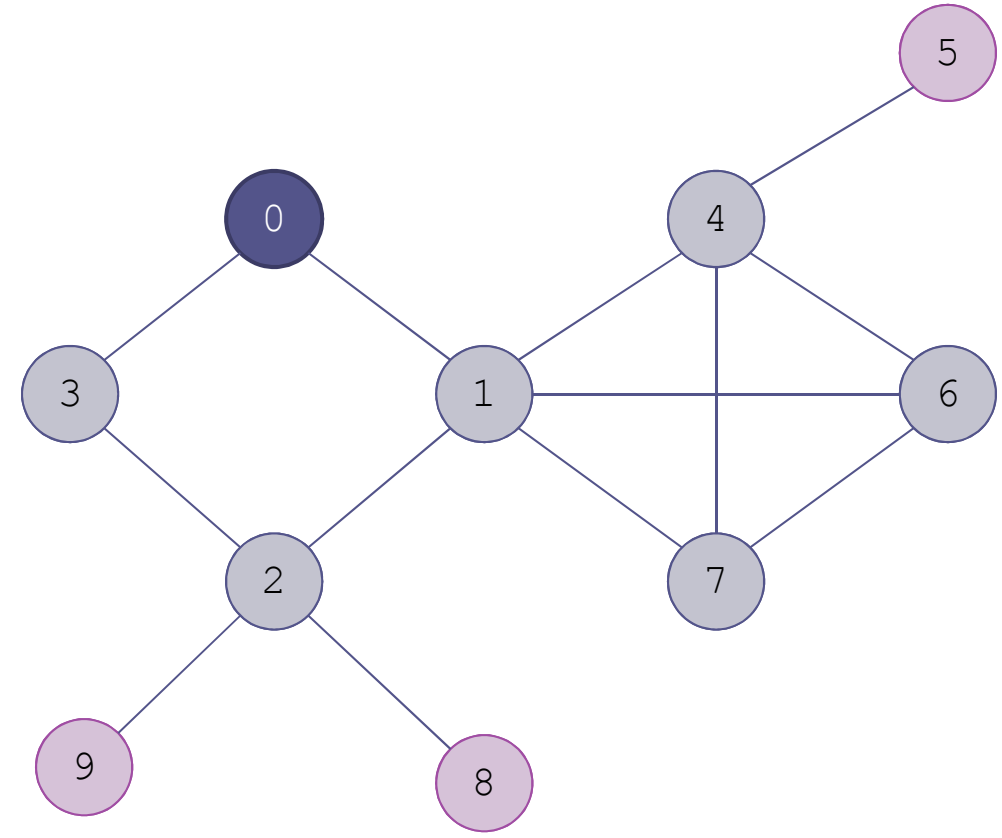


Example of a Breadth-First Search (cont.)

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

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1

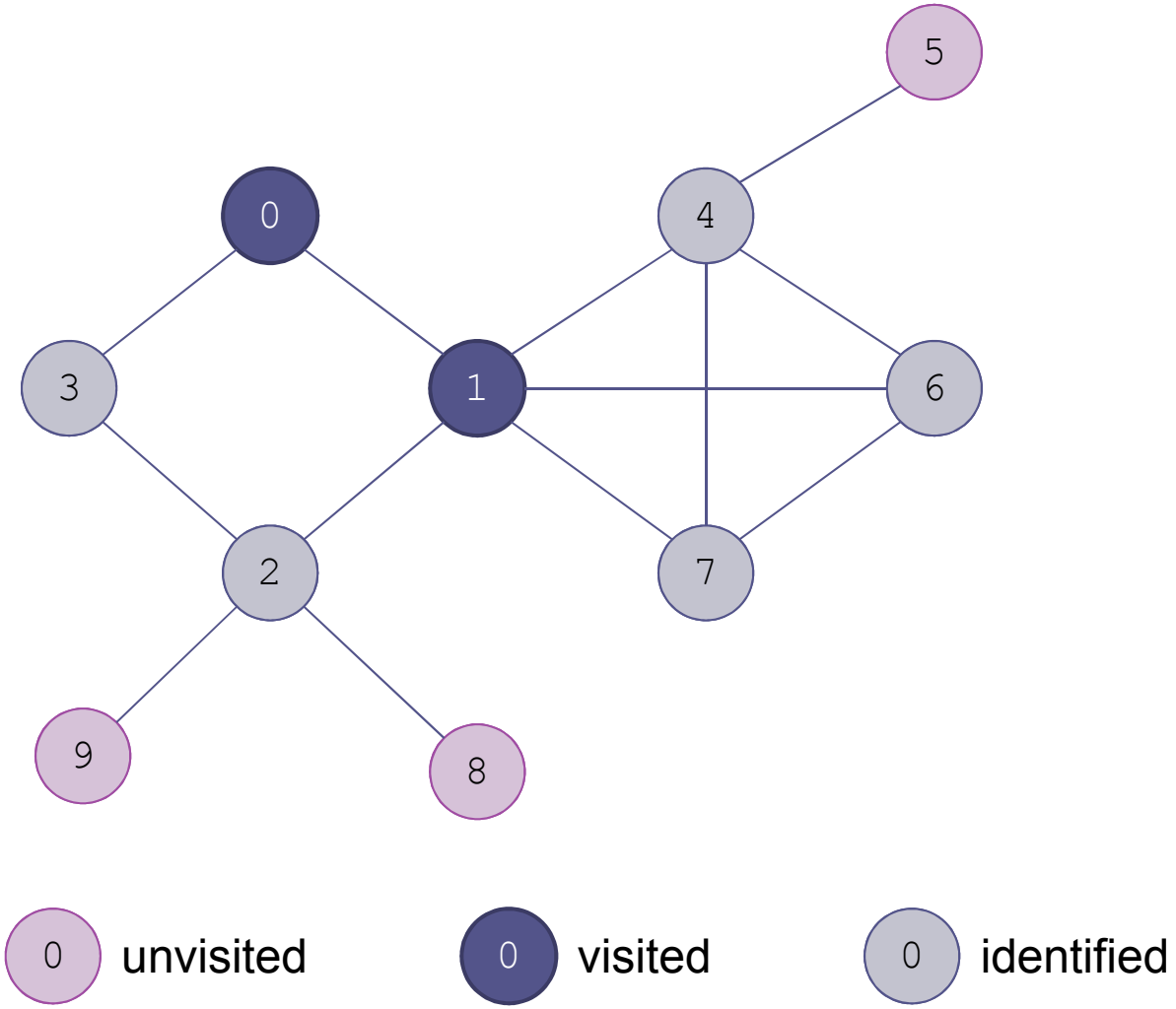


Example of a Breadth-First Search (cont.)

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

Queue:
3, 2, 4, 6, 7

Visit sequence:
0, 1

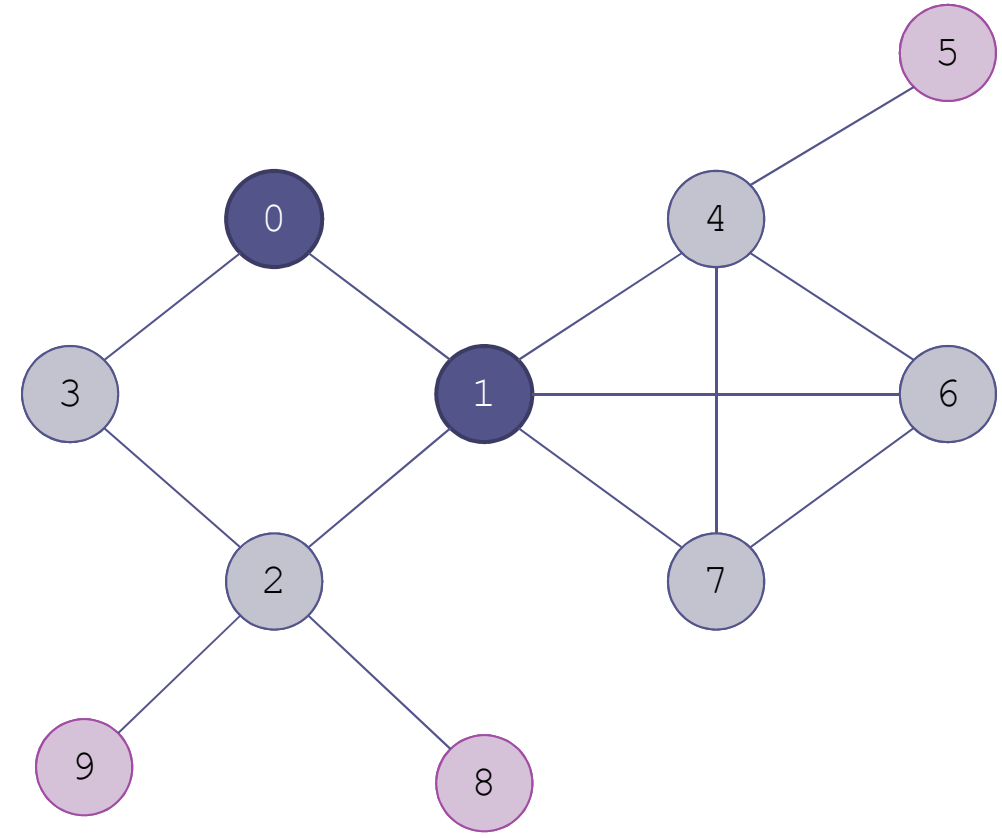


Example of a Breadth-First Search (cont.)

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

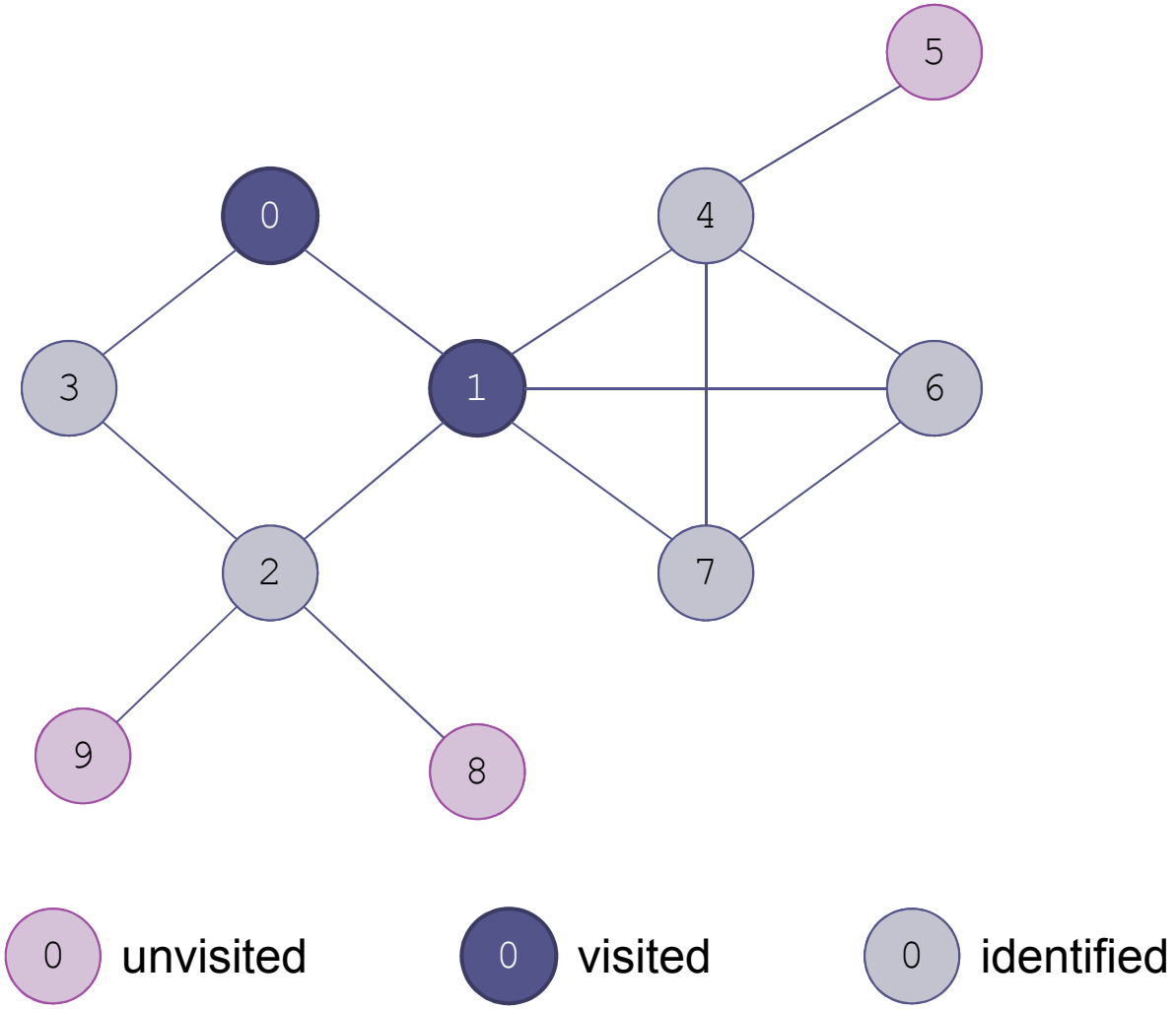


Example of a Breadth-First Search (cont.)

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

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3

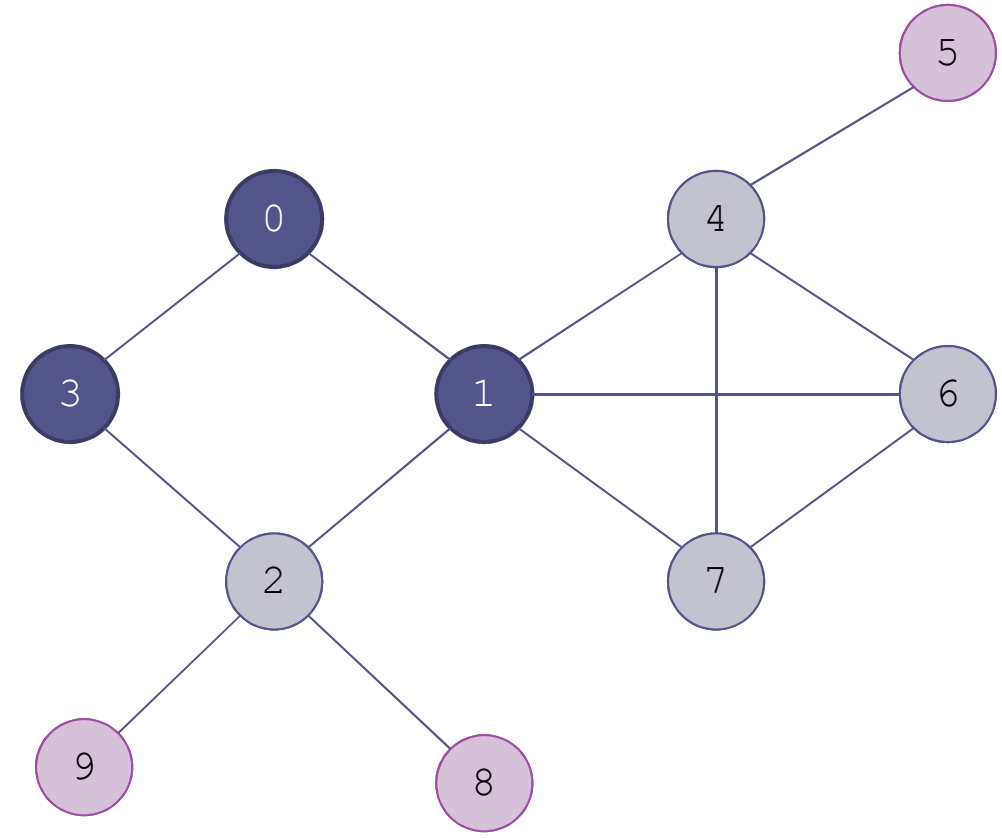


Example of a Breadth-First Search (cont.)

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

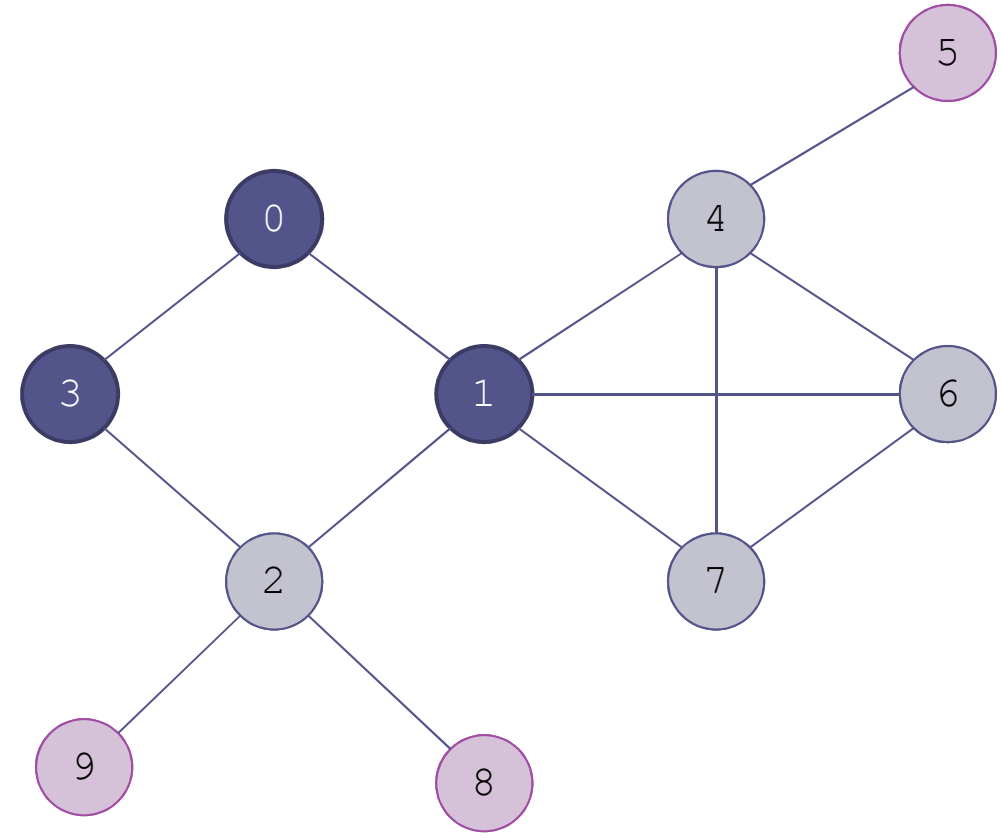


Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue:
2, 4, 6, 7

Visit sequence:
0, 1, 3

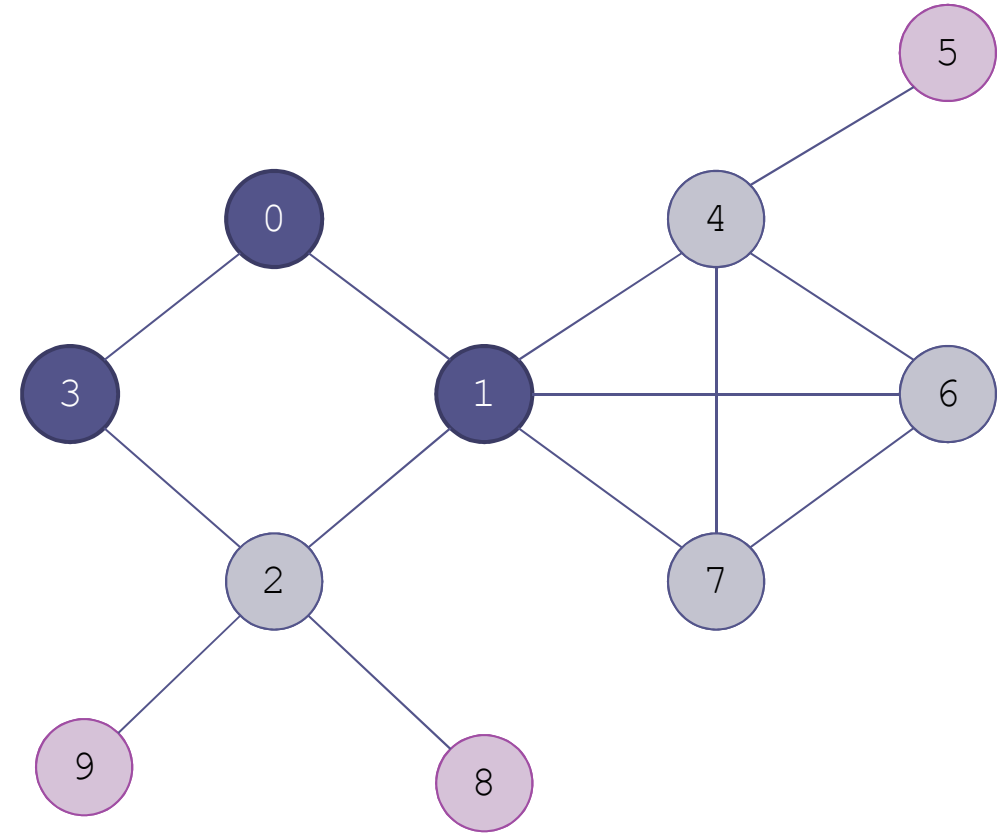


Example of a Breadth-First Search (cont.)

The next node in the queue is 2

Queue:
4, 6, 7

Visit sequence:
0, 1, 3, 2



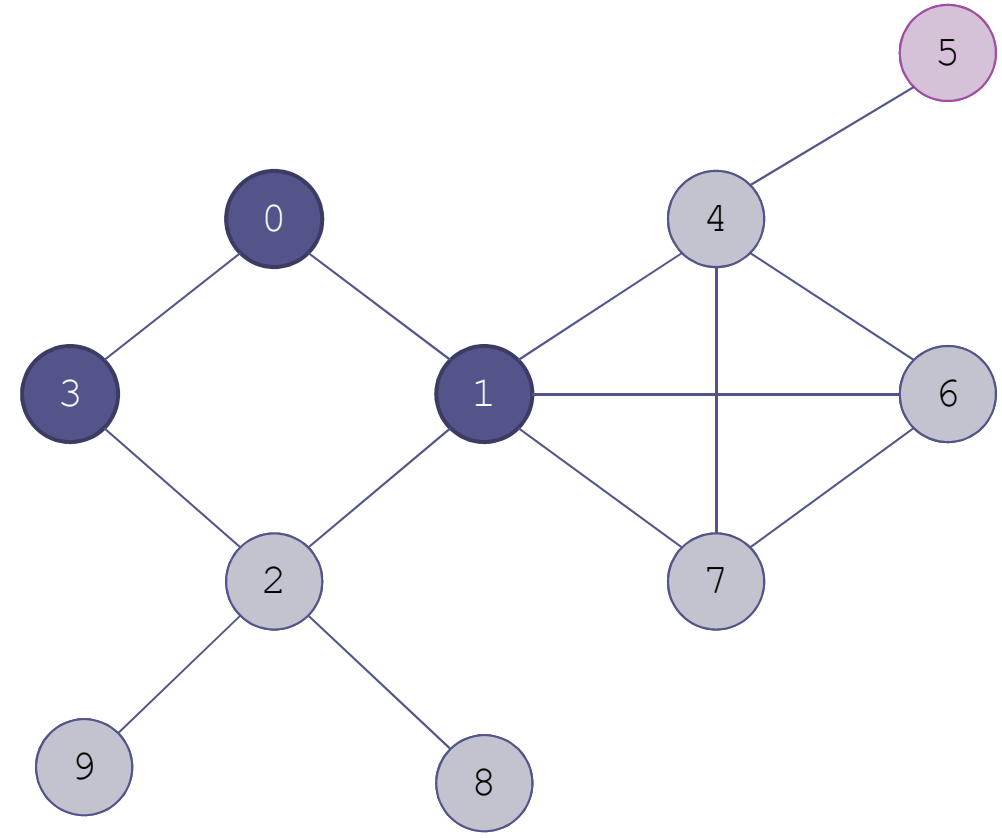
0 unvisited 0 visited 0 identified

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

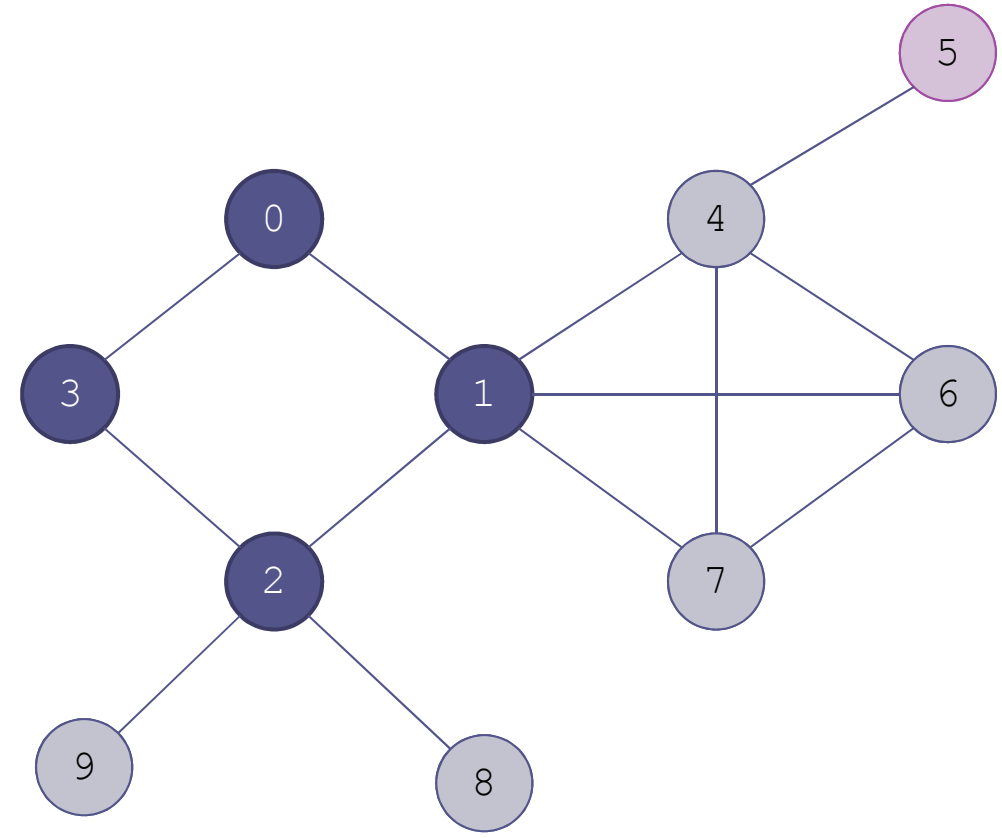


Example of a Breadth-First Search (cont.)

4 is next

Queue:
6, 7, 8, 9

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



0 unvisited

0 visited

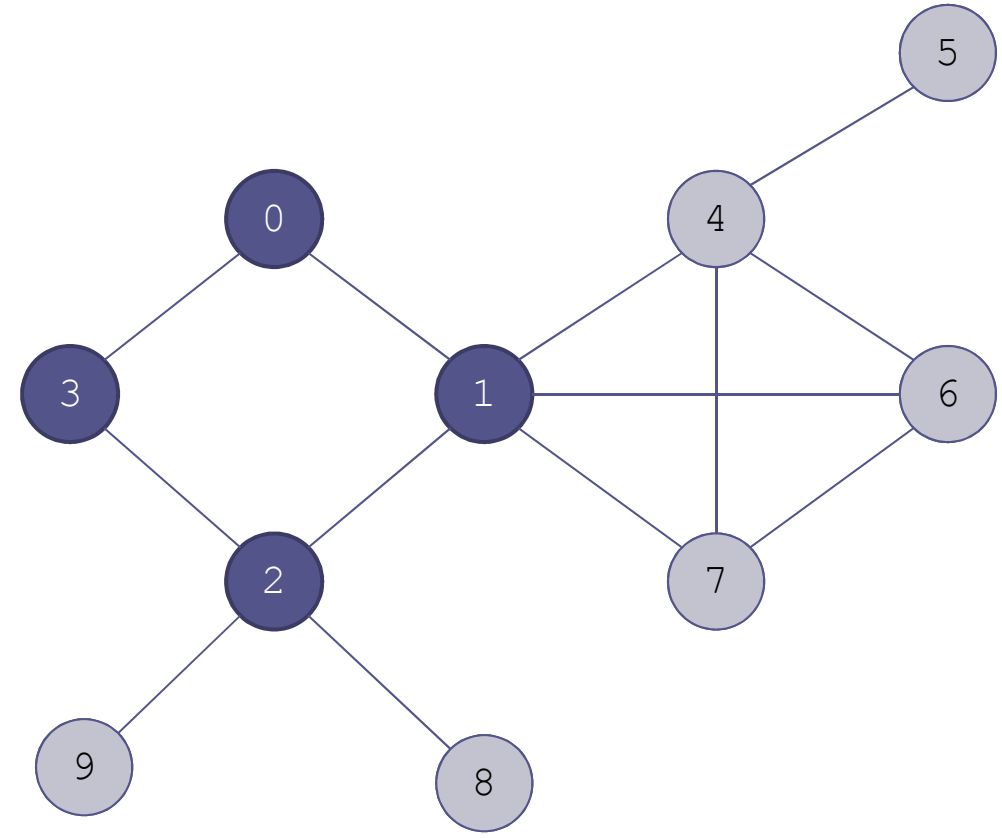
0 identified

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



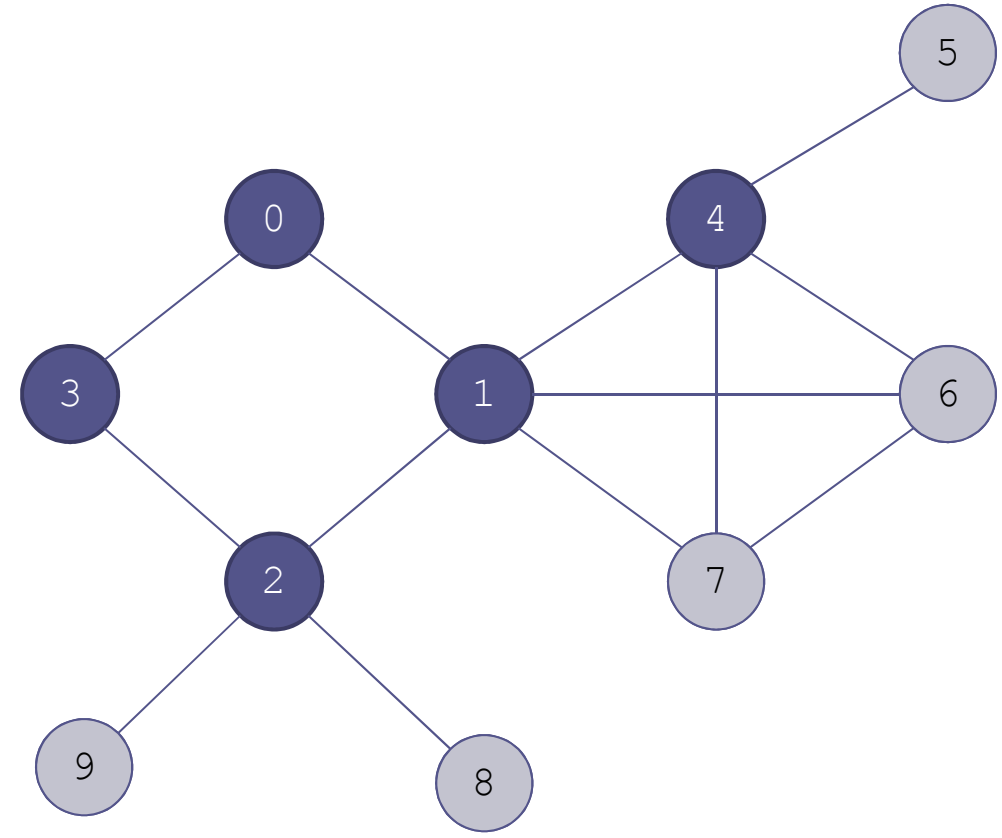
0 unvisited 0 visited 0 identified

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



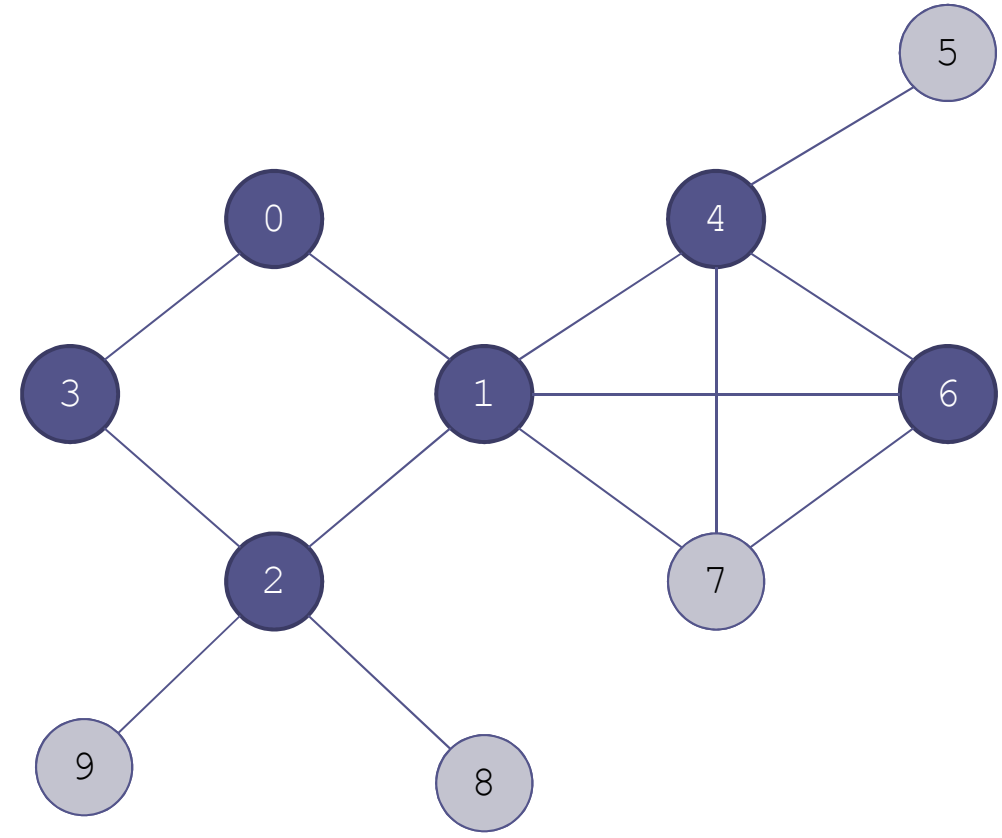
0 unvisited 0 visited 0 identified

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



0 unvisited

0 visited

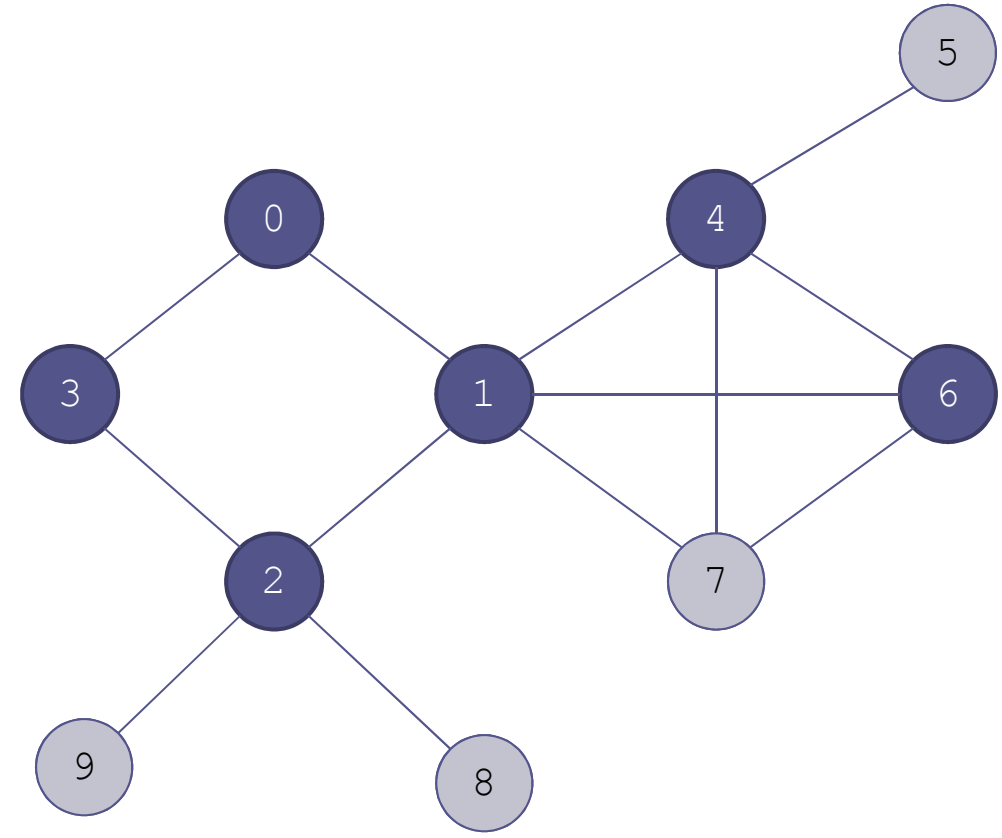
0 identified

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



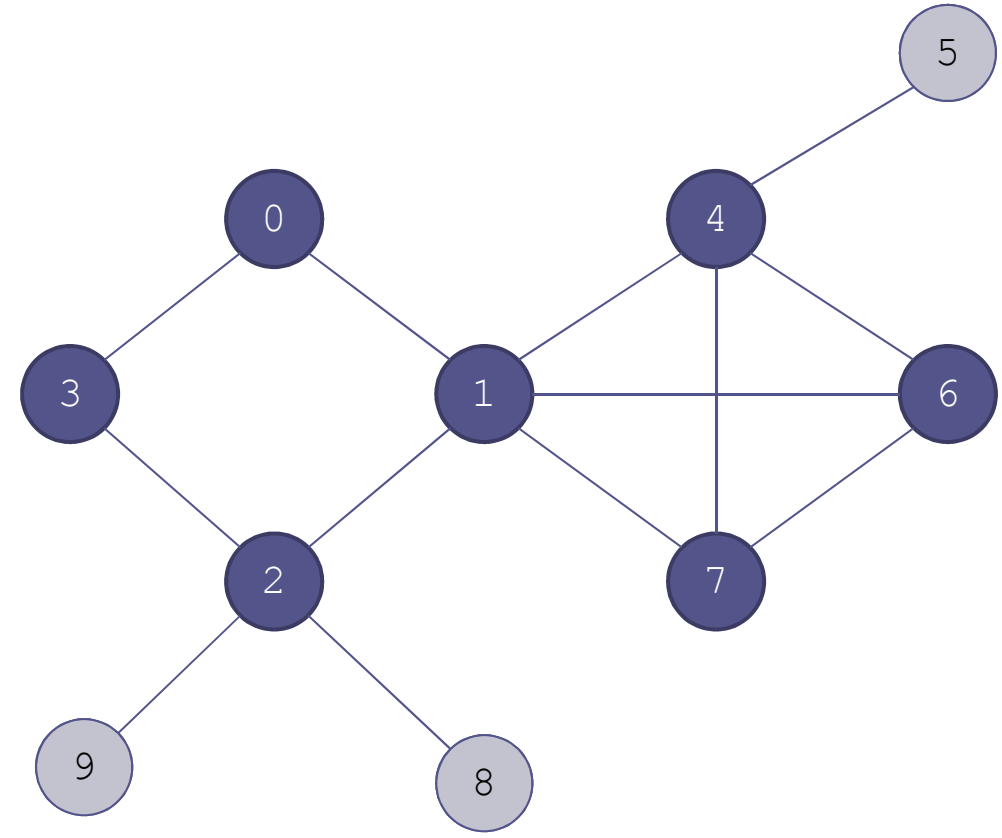
0 unvisited 0 visited 0 identified

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



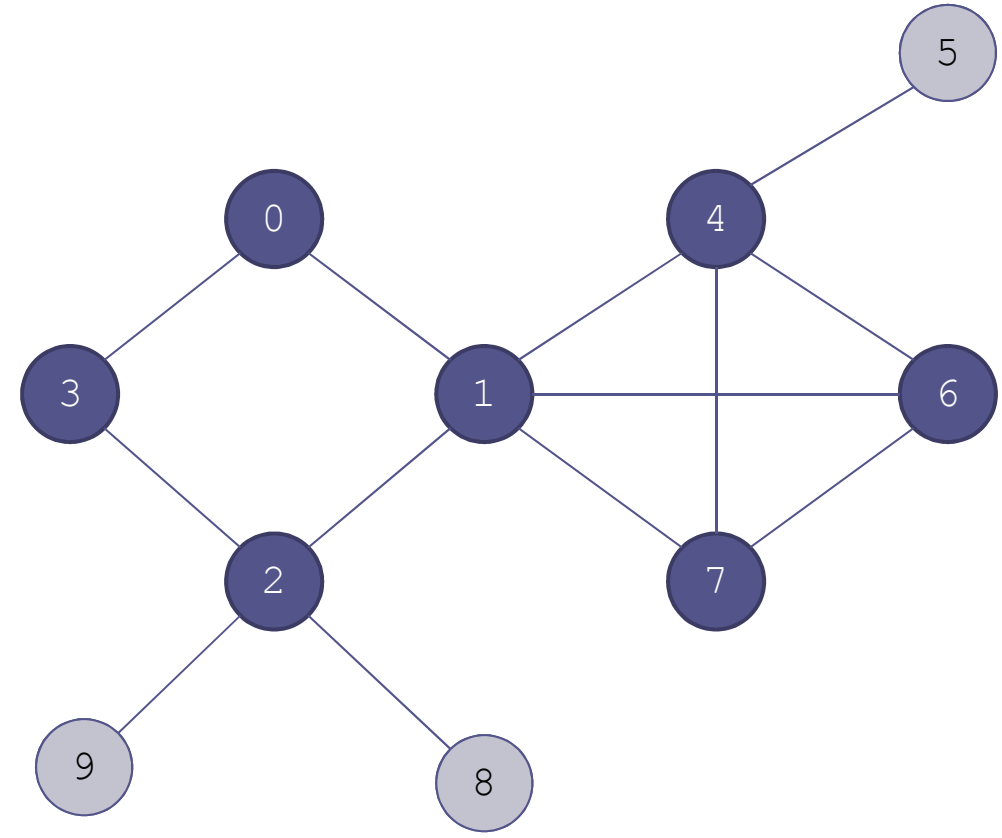
0 unvisited 0 visited 0 identified

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



0 unvisited

0 visited

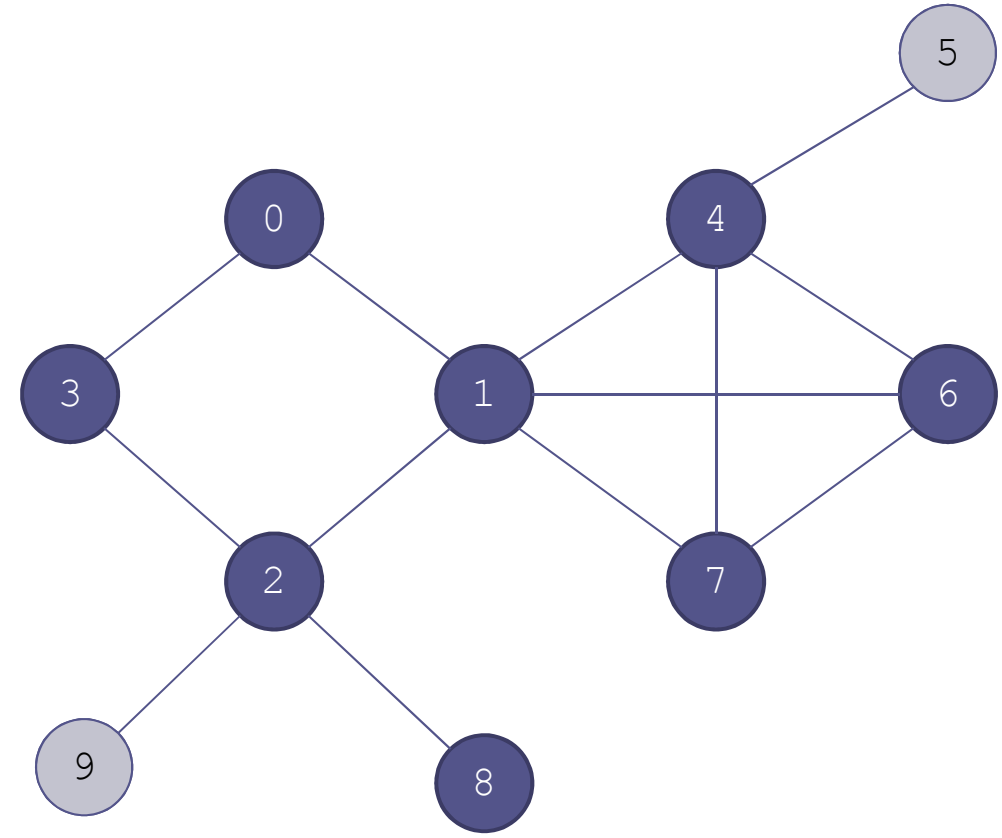
0 identified

Example of a Breadth-First Search (cont.)

8 has no vertices not already visited or identified

Queue:
9, 5

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

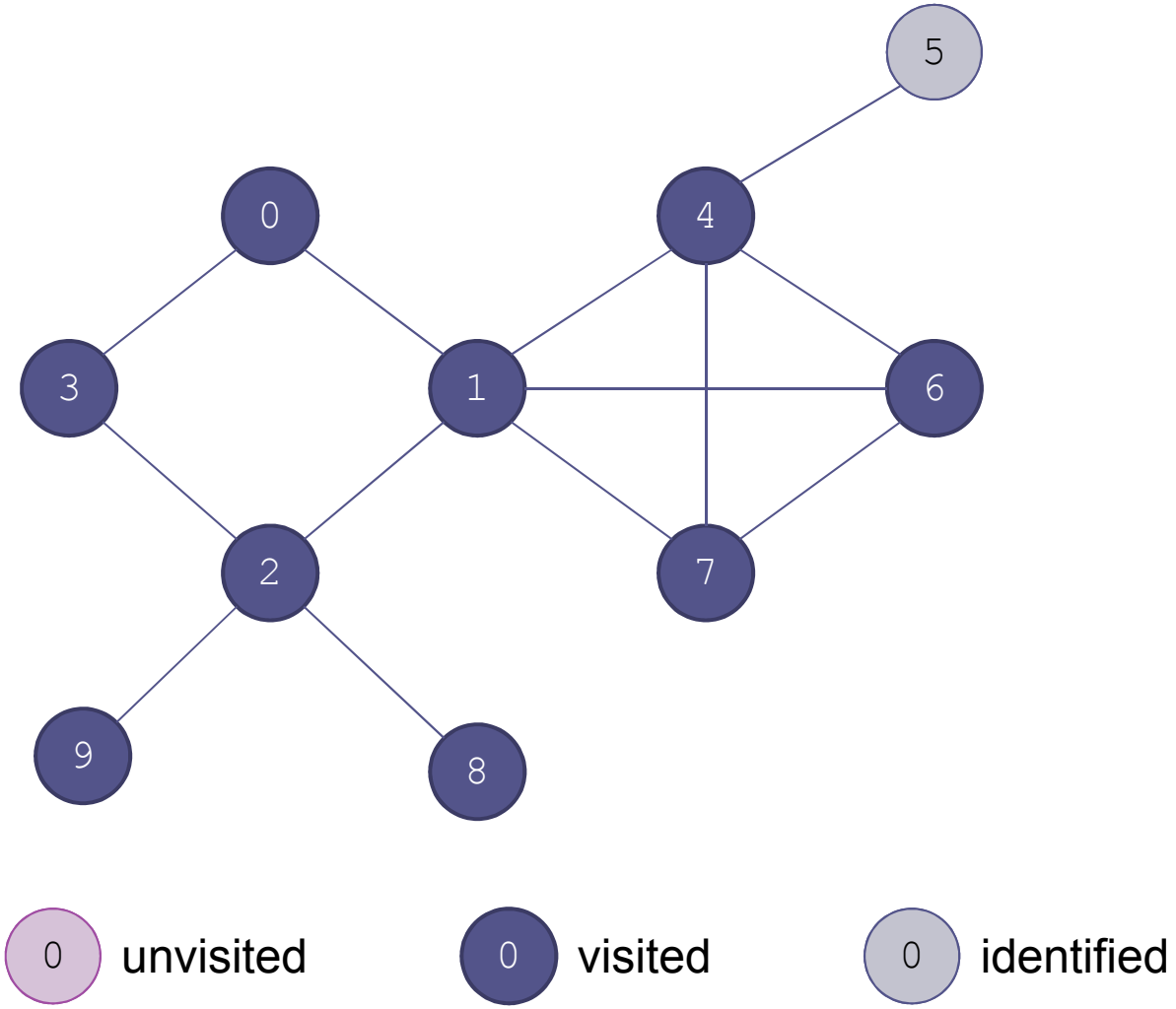


Example of a Breadth-First Search (cont.)

9 has no vertices not already visited or identified

Queue:
5

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

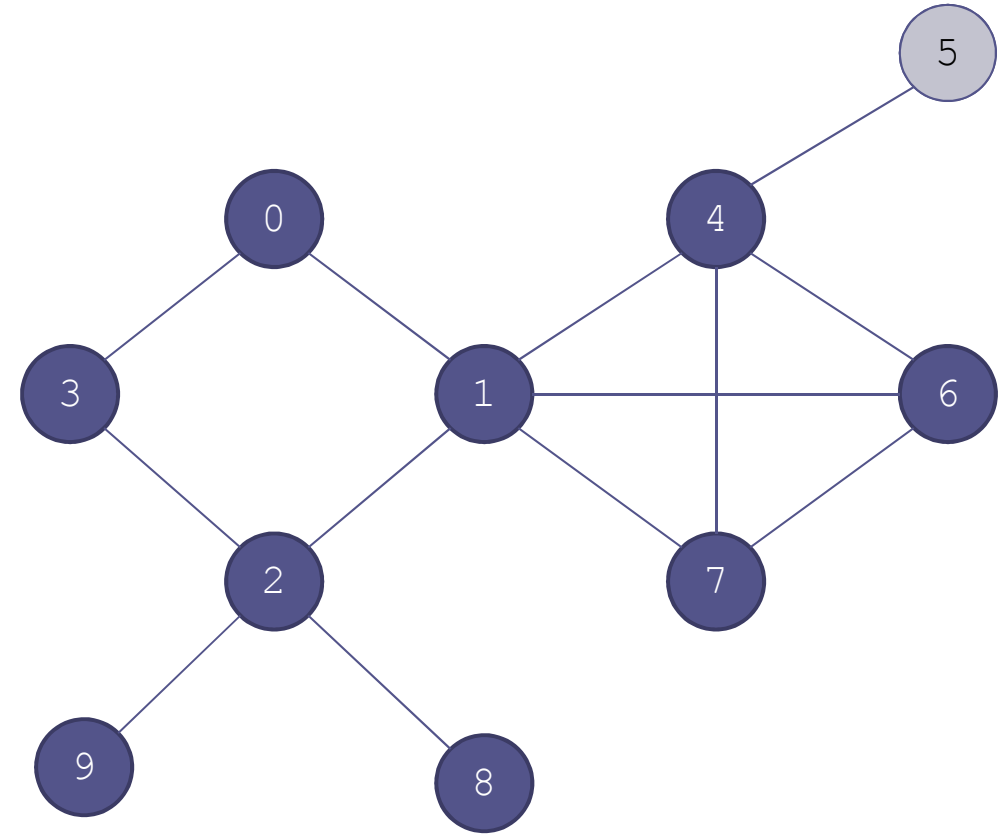


Example of a Breadth-First Search (cont.)

Finally we visit 5

Queue:
5

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



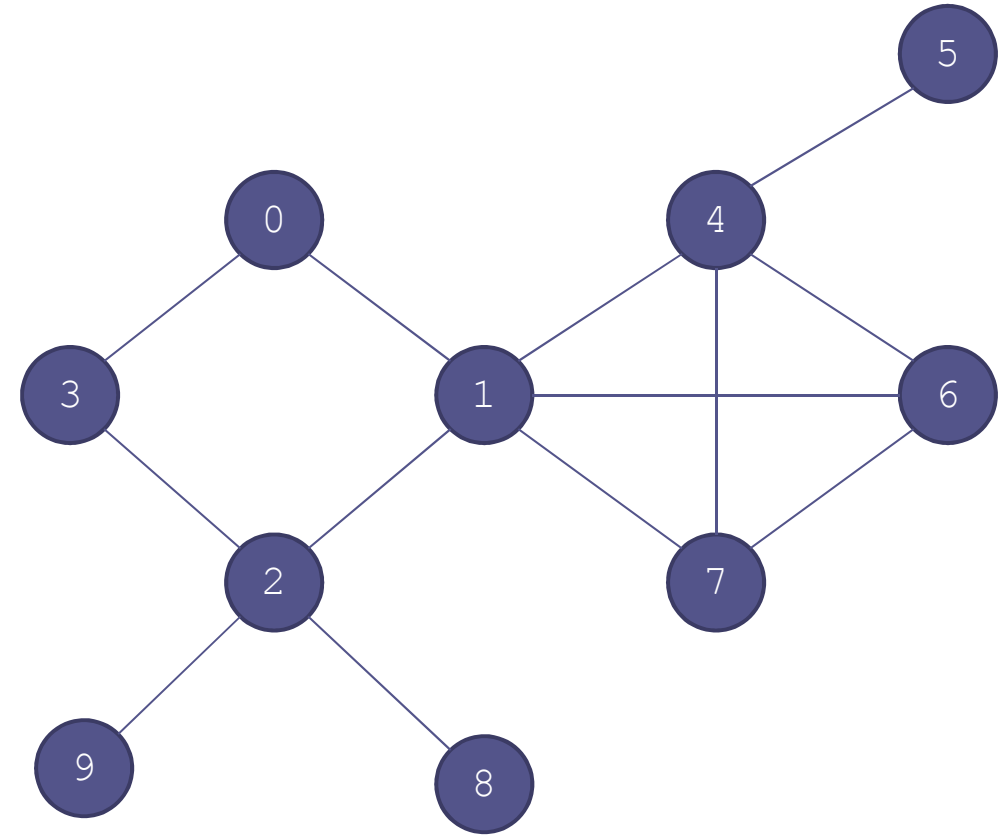
0 unvisited 0 visited 0 identified

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

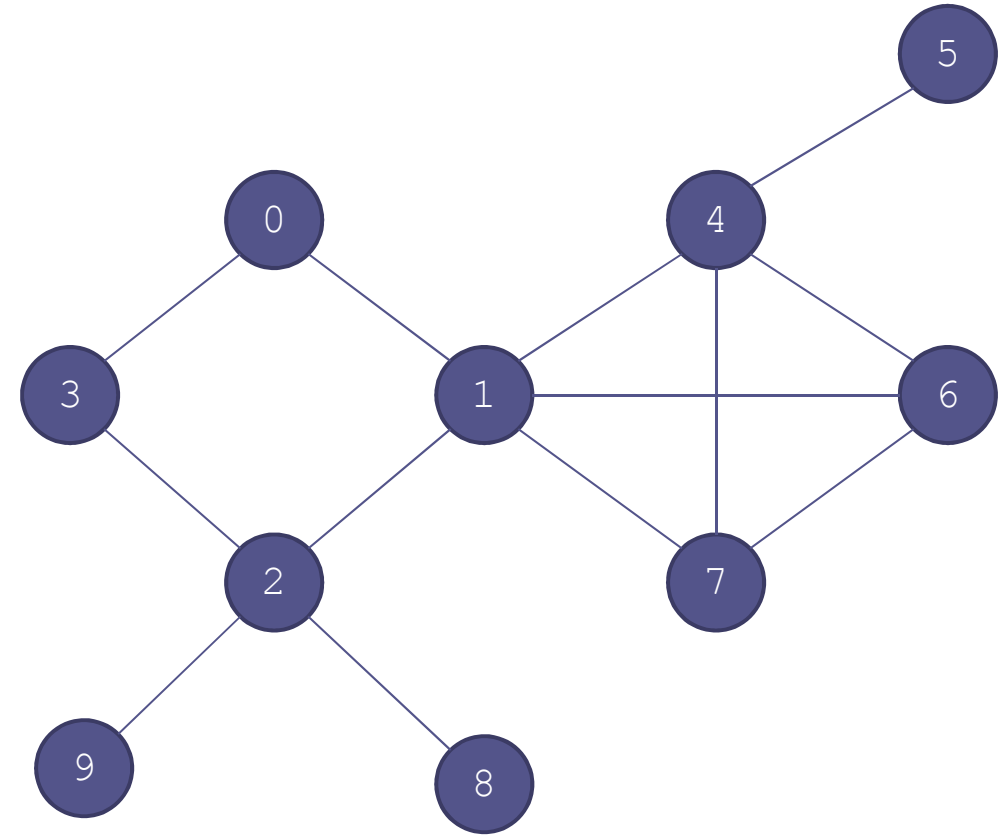


Example of a Breadth-First Search (cont.)

The queue is empty; all vertices have been visited

Queue:
empty

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



0 unvisited 0 visited 0 identified

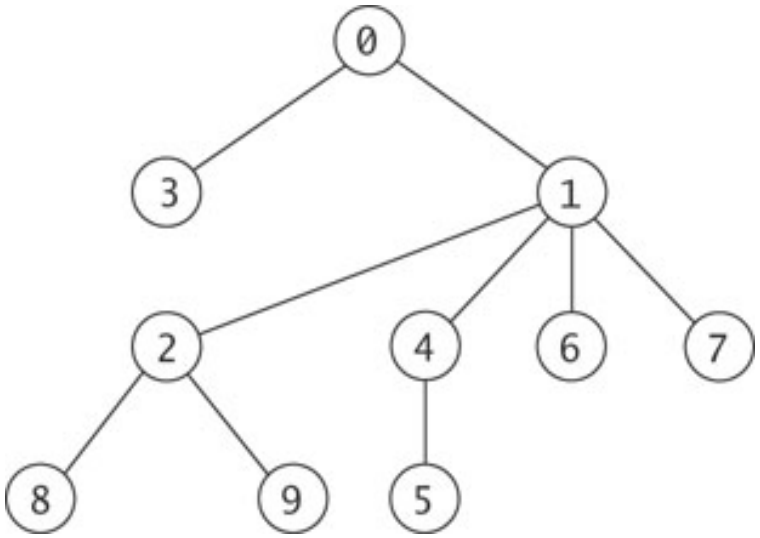
Algorithm for Breadth-First Search

Algorithm for Breadth-First Search

1. Take an arbitrary start vertex, mark it identified (color it light blue), and place it in a queue.
2. while the queue is not empty
3. Take a vertex, u, out of the queue and visit u.
4. for all vertices, v, adjacent to this vertex, u
5. if v has not been identified or visited
6. Mark it identified (color it light blue).
7. Insert vertex v into the queue.
8. We are now finished visiting u (color it dark blue).

Algorithm for Breadth-First Search (cont.)

- We can also build a tree that represents the order in which vertices will be visited in a breadth-first traversal
- The tree has all of the vertices and some of the edges of the original graph



- A path starting at the root to any vertex in the tree is the *shortest path* in the original graph to that vertex (considering all edges to have the same weight)

Algorithm for Breadth-First Search (cont.)

- We can save the information we need to represent the tree by storing the parent of each vertex when we identify it
- We can refine Step 7 of the algorithm to accomplish this:
  - 7.1 Insert vertex *v* into the queue
  - 7.2 Set the parent of *v* to *u*

Performance Analysis of Breadth-First Search

- The loop at Step 2 is performed for each vertex.
- The inner loop at Step 4 is performed for $|E_v|$, the number of edges that originate at that vertex)
- The total number of steps is the sum of the edges that originate at each vertex, which is the total number of edges
- The algorithm is O(|E|)

Implementing Breadth-First Search

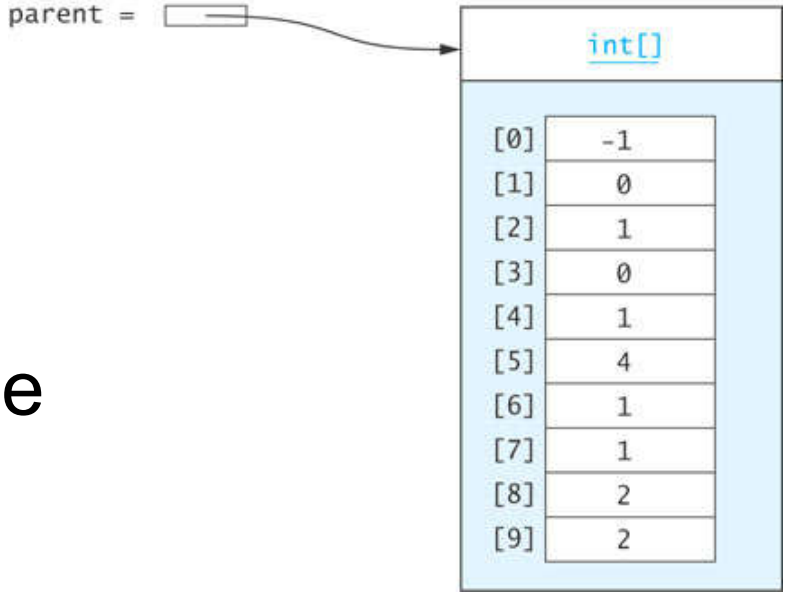
```
public static int[] breadthFirstSearch(Graph graph, int start) {
    Queue<Integer> theQueue = new LinkedList<Integer>();
    // Declare array parent and initialize its elements to -1.
    int[] parent = new int[graph.getNumV()];
    for (int i = 0; i < graph.getNumV(); i++) {
        parent[i] = -1;
    }

    // Declare array identified and
    // initialize its elements to false.
    boolean[] identified = new boolean[graph.getNumV()];
    /* Mark the start vertex as identified and insert it
       into the queue */
    identified[start] = true;
    theQueue.offer(start);

    /* Perform breadth-first search until done */
    while (!theQueue.isEmpty()) {
        /* Take a vertex, current, out of the queue.
         (Begin visiting current). */
        int current = theQueue.remove();
        /* Examine each vertex, neighbor, adjacent to current. */
        Iterator<Edge> itr = graph.edgeIterator(current);
        while (itr.hasNext()) {
            Edge edge = itr.next();
            int neighbor = edge.getDest();
            // If neighbor has not been identified
            if (!identified[neighbor]) {
                // Mark it identified.
                identified[neighbor] = true;
                // Place it into the queue.
                theQueue.offer(neighbor);
                /* Insert the edge (current, neighbor)
                   into the tree. */
                parent[neighbor] = current;
            }
        }
        // Finished visiting current.
    }
    return parent;
}
```

Implementing Breadth-First Search (cont.)

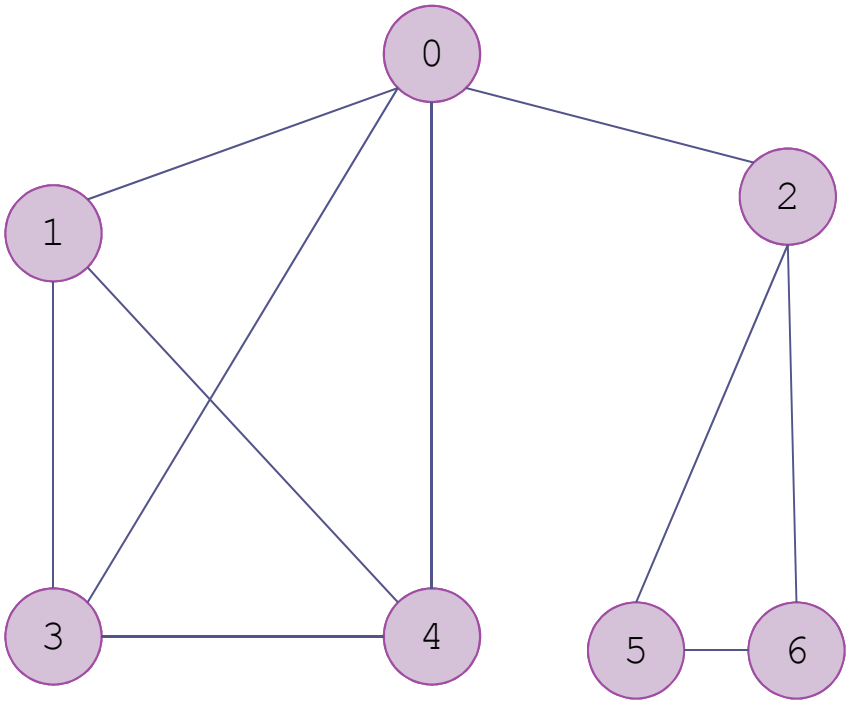
- The method returns array `parent` which can be used to construct the breadth-first search tree
- If we run the search on the graph we just traversed, `parent` will be filled with the values shown on the right



Depth-First Search

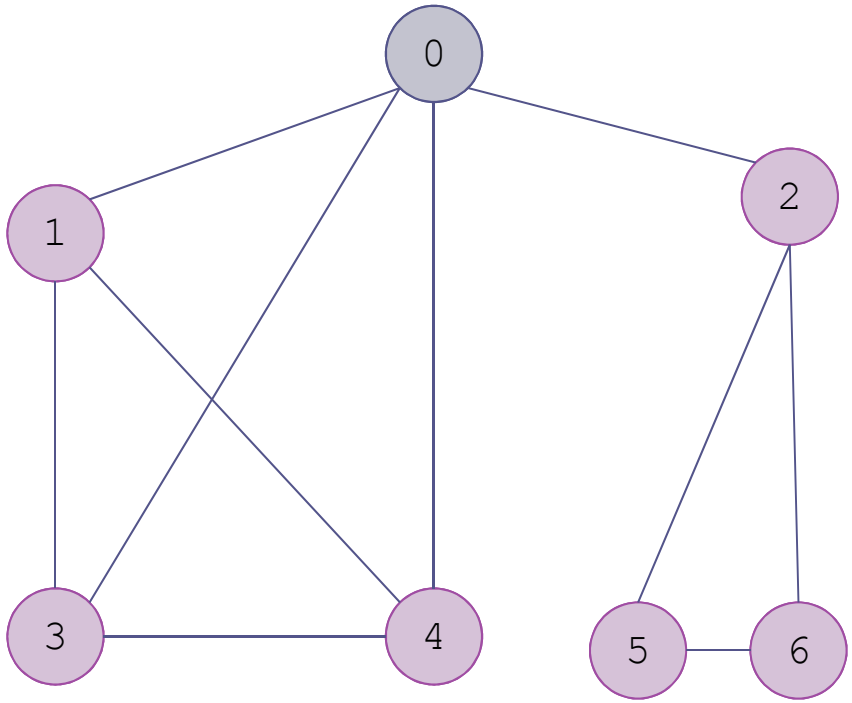
- In a depth-first search,
  - start at a vertex,
  - visit it,
  - choose one adjacent vertex to visit;
  - then, choose a vertex adjacent to that vertex to visit,
  - and so on until you go no further;
  - then back up and see whether a new vertex can be found

Example of a Depth-First Search



Example of a Depth-First Search (cont.)

Mark 0 as being visited

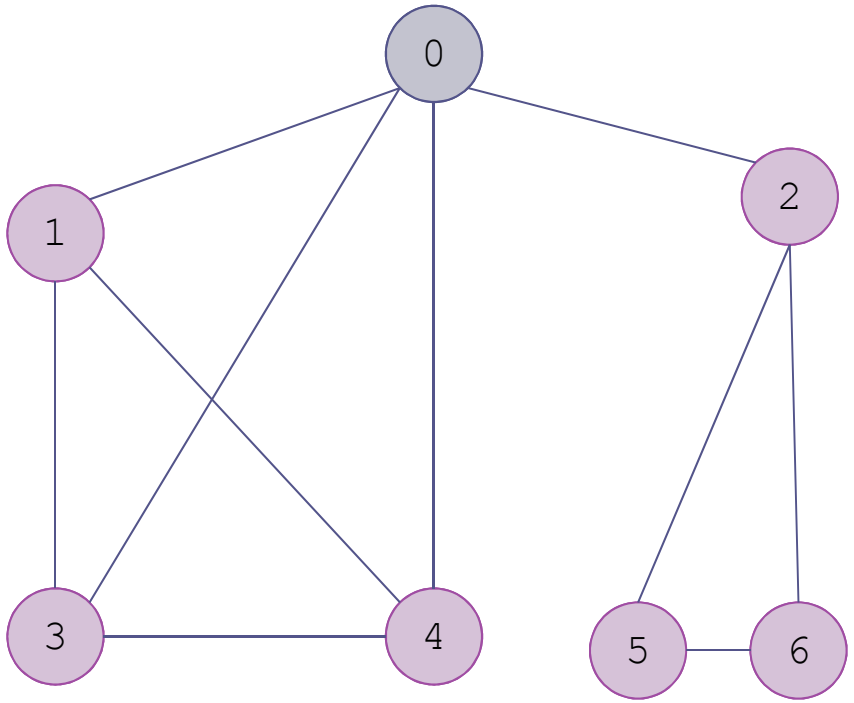


Discovery (Visit) order:
0

Finish order:

Example of a Depth-First Search (cont.)

Choose an adjacent vertex that is not being visited

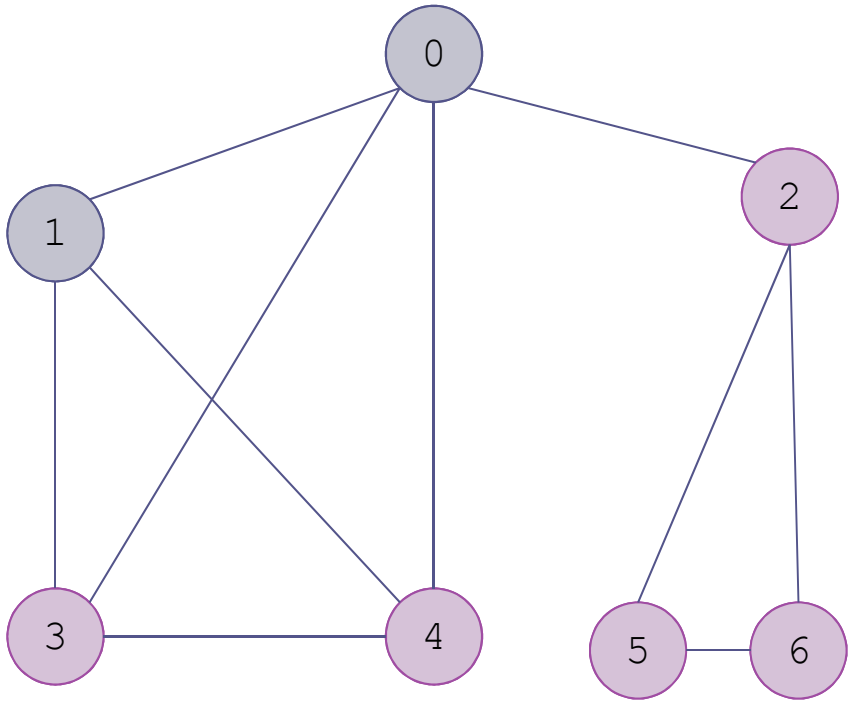


Discovery (Visit) order:
0

Finish order:

Example of a Depth-First Search (cont.)

Choose an adjacent vertex that is not being visited

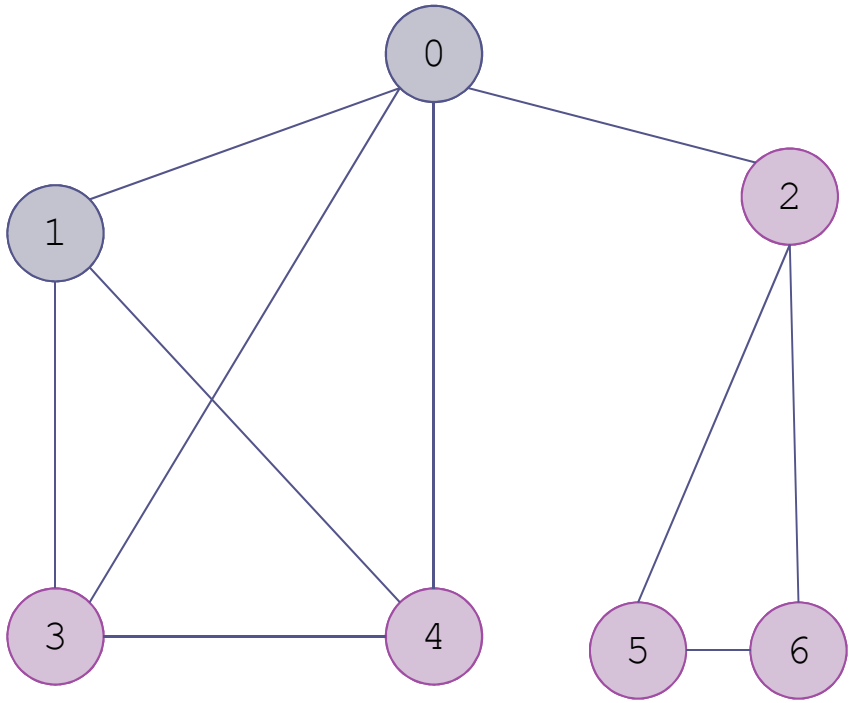


Discovery (Visit) order:
0, 1

Finish order:

Example of a Depth-First Search (cont.)

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

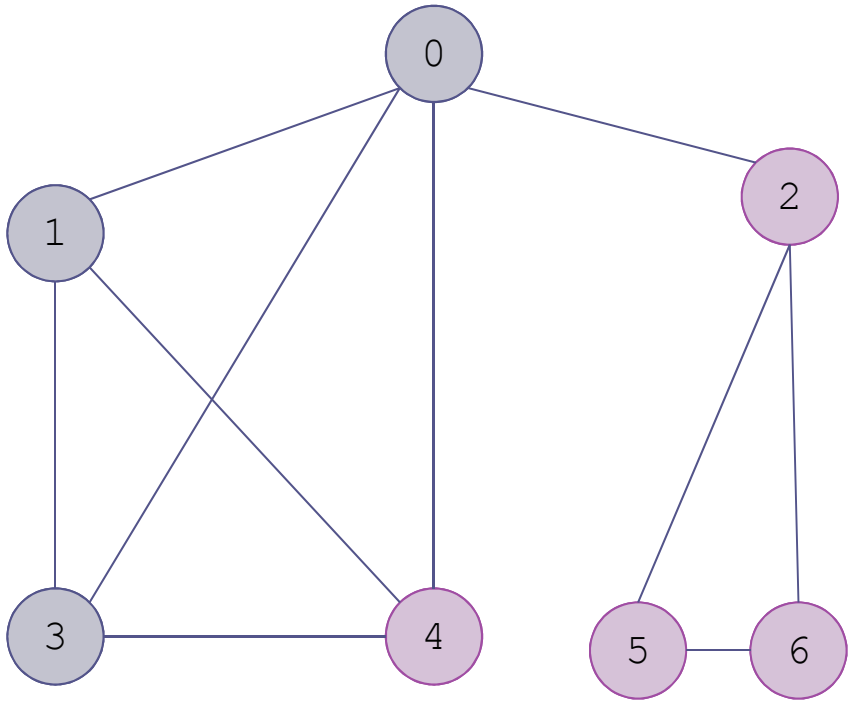


Discovery (Visit) order:
0, 1, 3

Finish order:

Example of a Depth-First Search (cont.)

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

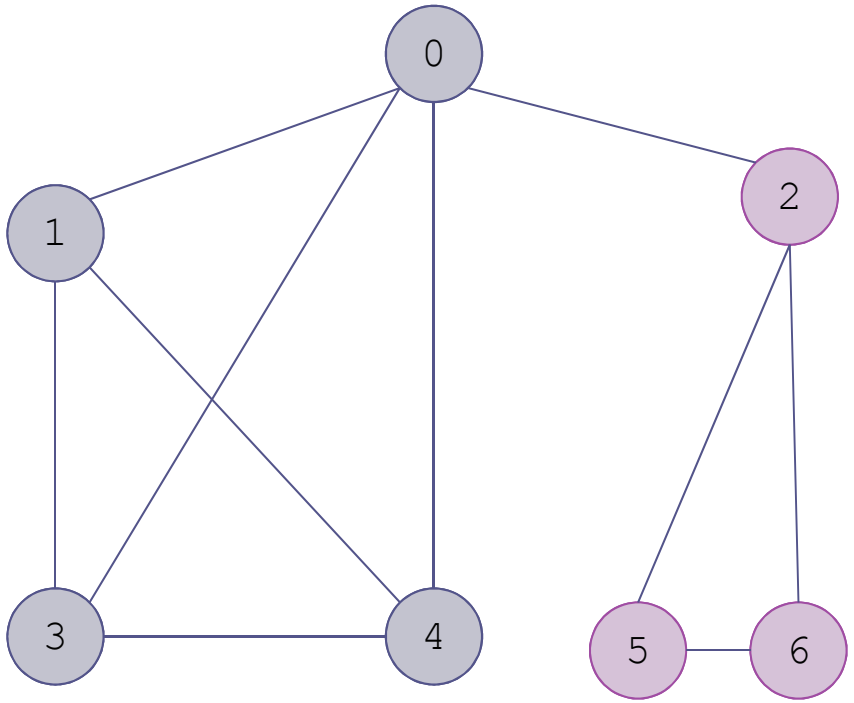


Discovery (Visit) order:
0, 1, 3

Finish order:

Example of a Depth-First Search (cont.)

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

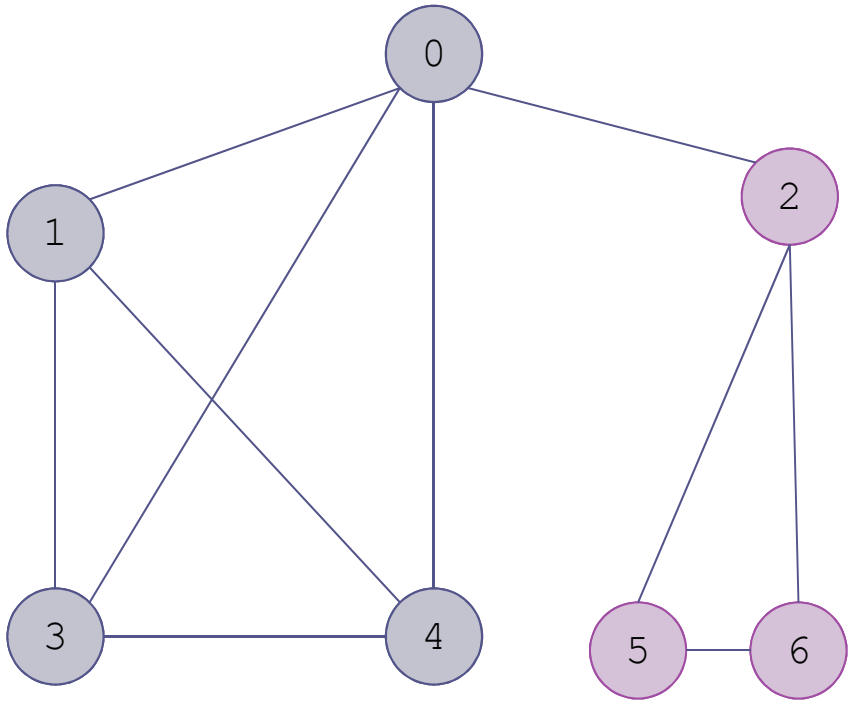


Discovery (Visit) order:
0, 1, 3, 4

Finish order:

Example of a Depth-First Search (cont.)

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

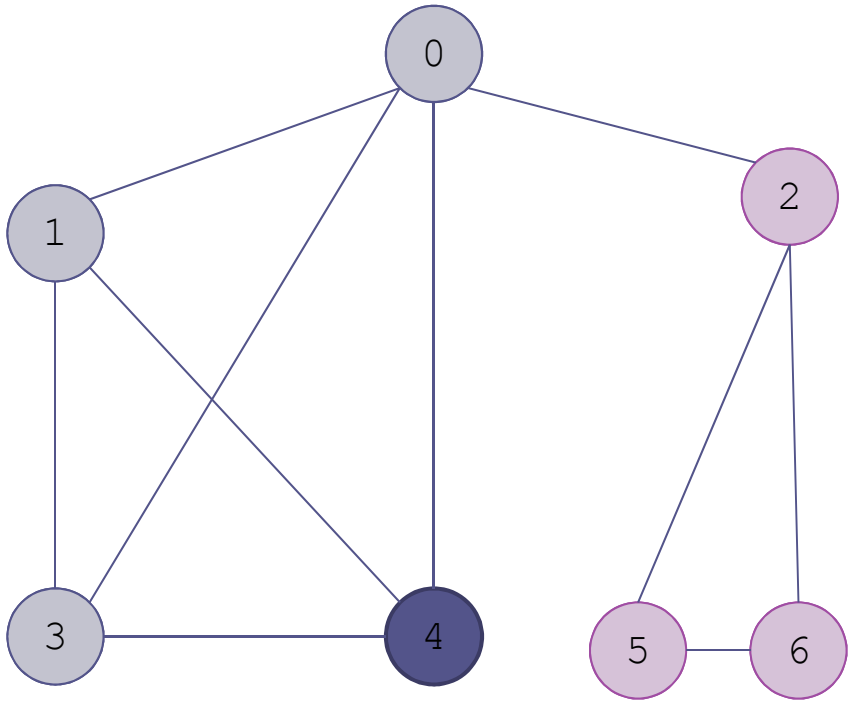


Discovery (Visit) order:
0, 1, 3, 4

Finish order:

Example of a Depth-First Search (cont.)

Mark 4 as visited

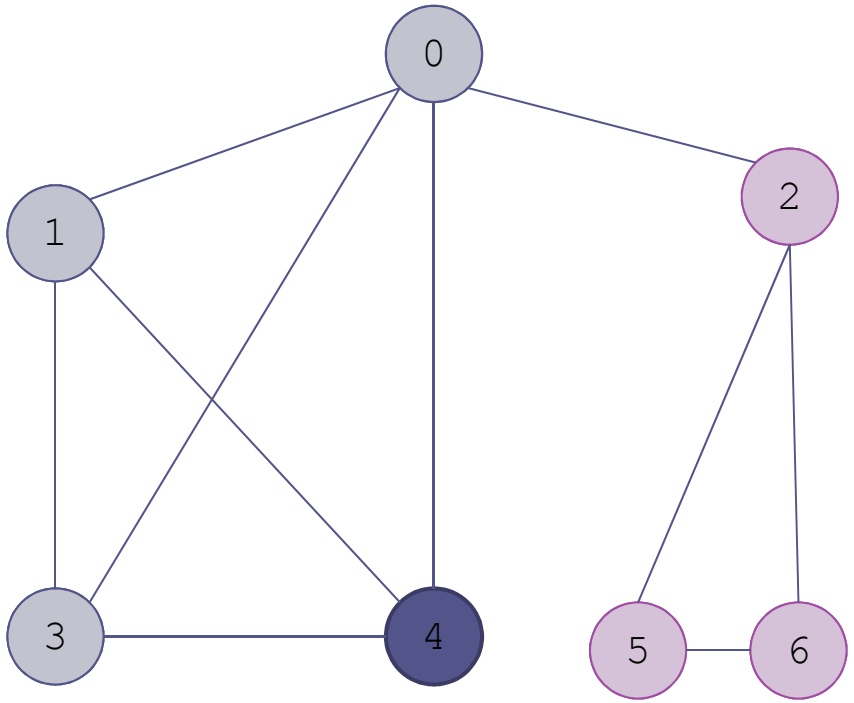


Discovery (Visit) order:
0, 1, 3, 4

Finish order:
4

Example of a Depth-First Search (cont.)

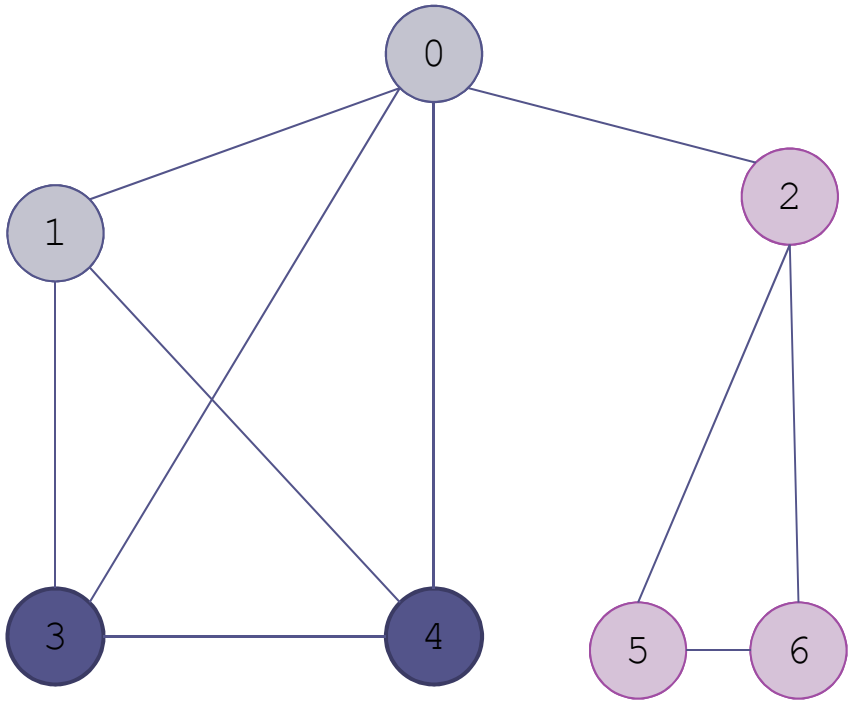
Return from the recursion to 3; all adjacent nodes to 3 are being visited



Finish order:
4

Example of a Depth-First Search (cont.)

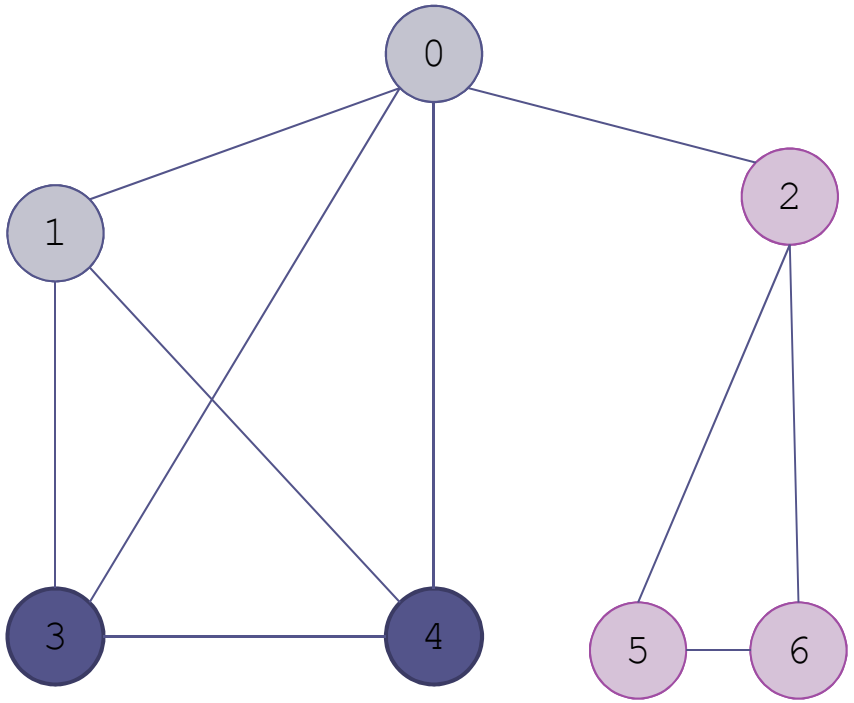
Mark 3 as visited



Finish order:
4, 3

Example of a Depth-First Search (cont.)

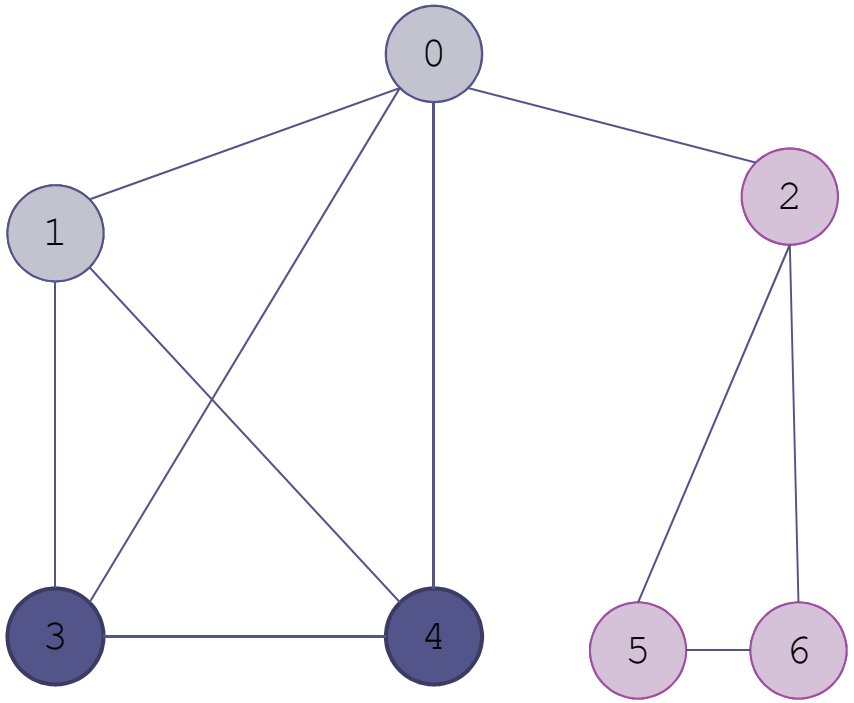
Return from the recursion to 1



Finish order:
4, 3

Example of a Depth-First Search (cont.)

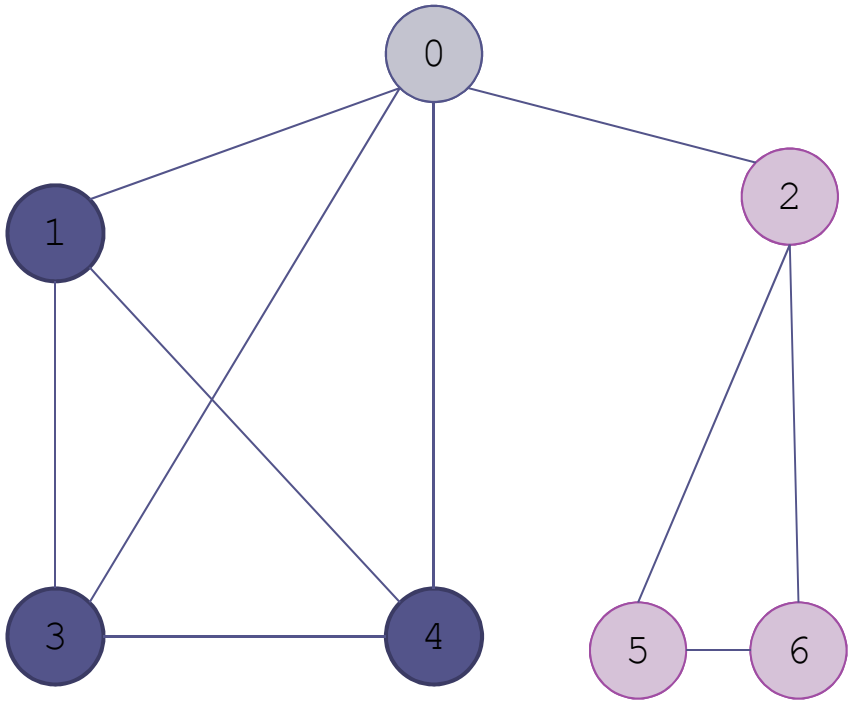
All vertices adjacent to 1 are being visited



Finish order:
4, 3

Example of a Depth-First Search (cont.)

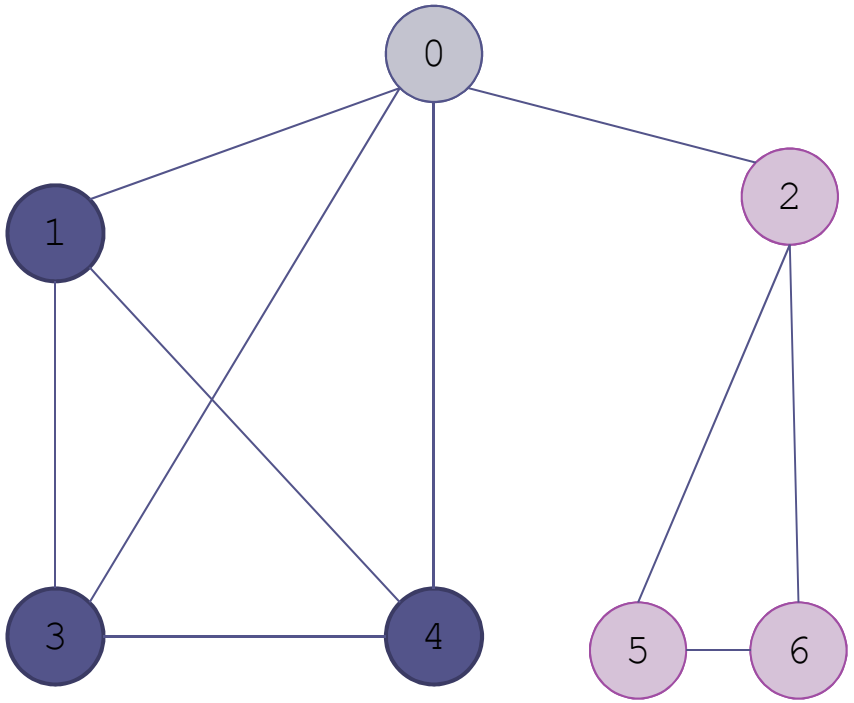
Mark 1 as visited



Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

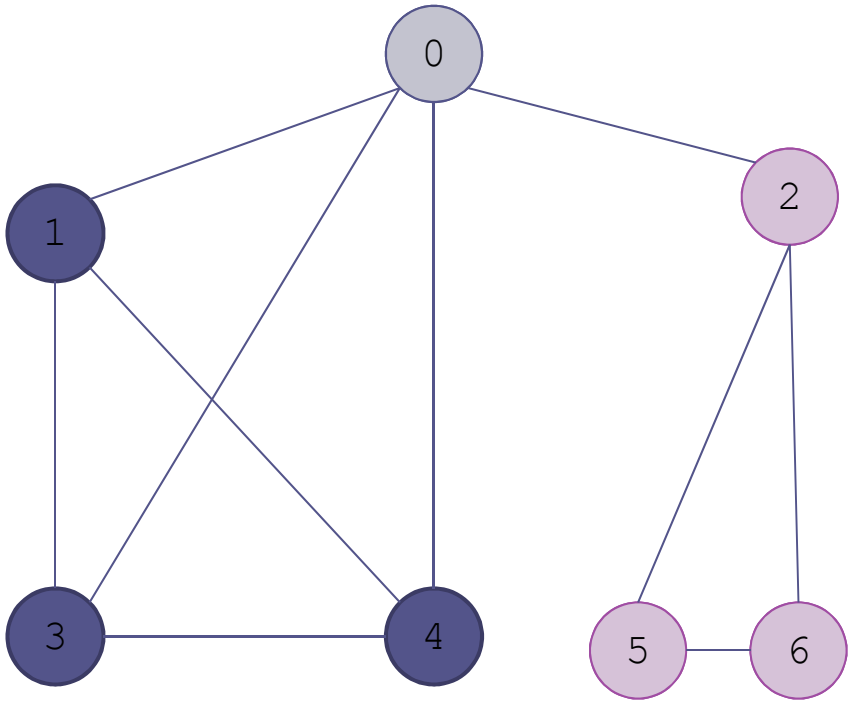
Return from the recursion to 0



Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

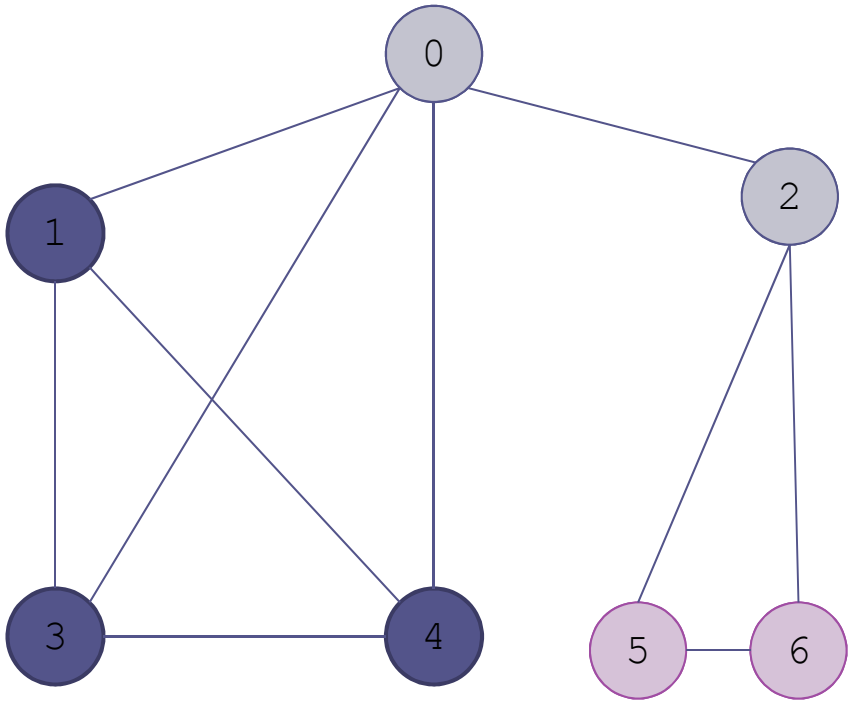
2 is adjacent to 0 and is not being visited



Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

2 is adjacent to 0 and is not being visited

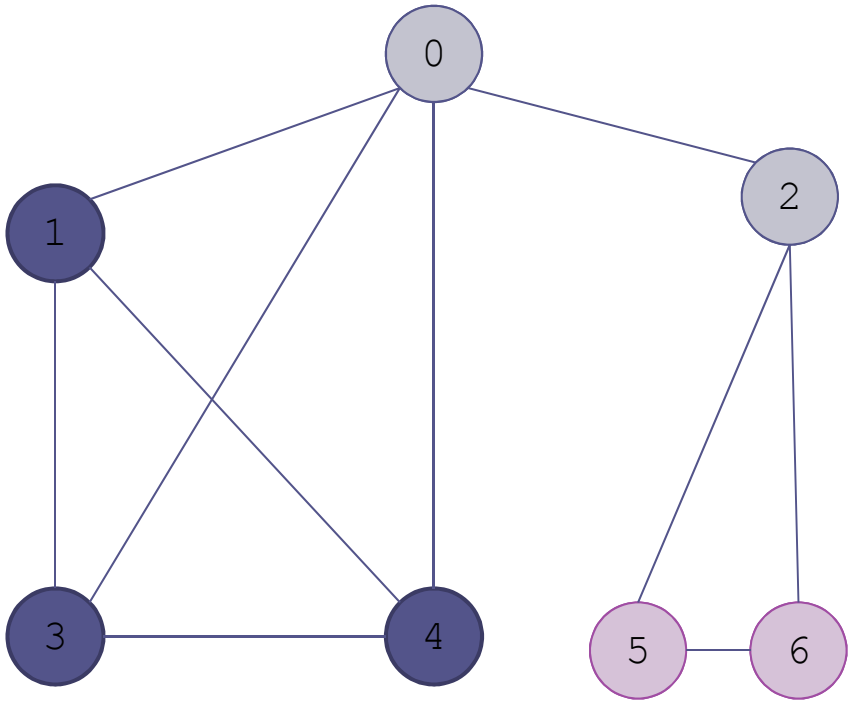


Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

5 is adjacent to 2 and is not being visited

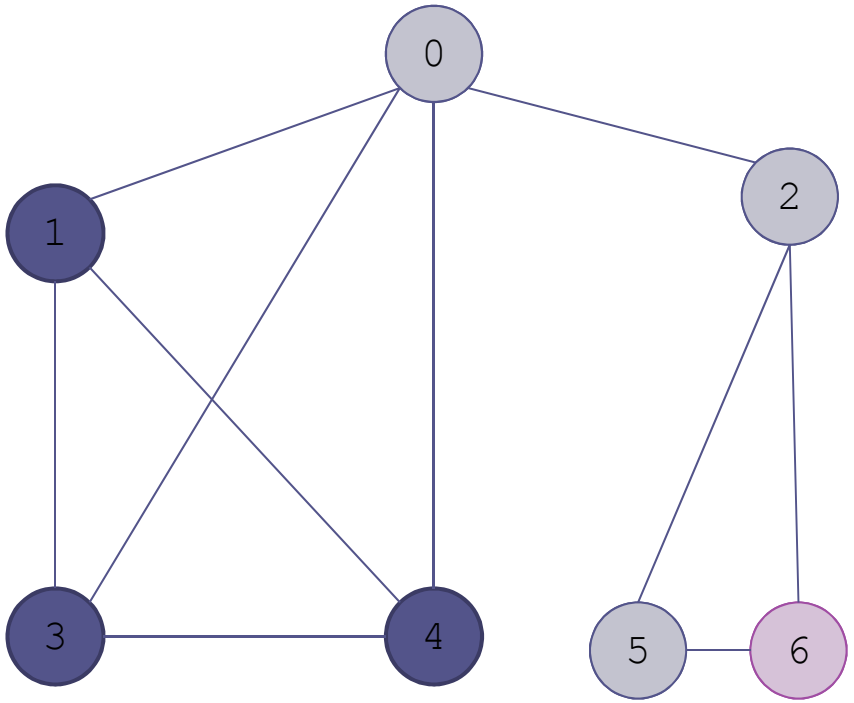


Discovery (Visit) order:
0, 1, 3, 4, 2

Finish order:
4, 3, 1

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

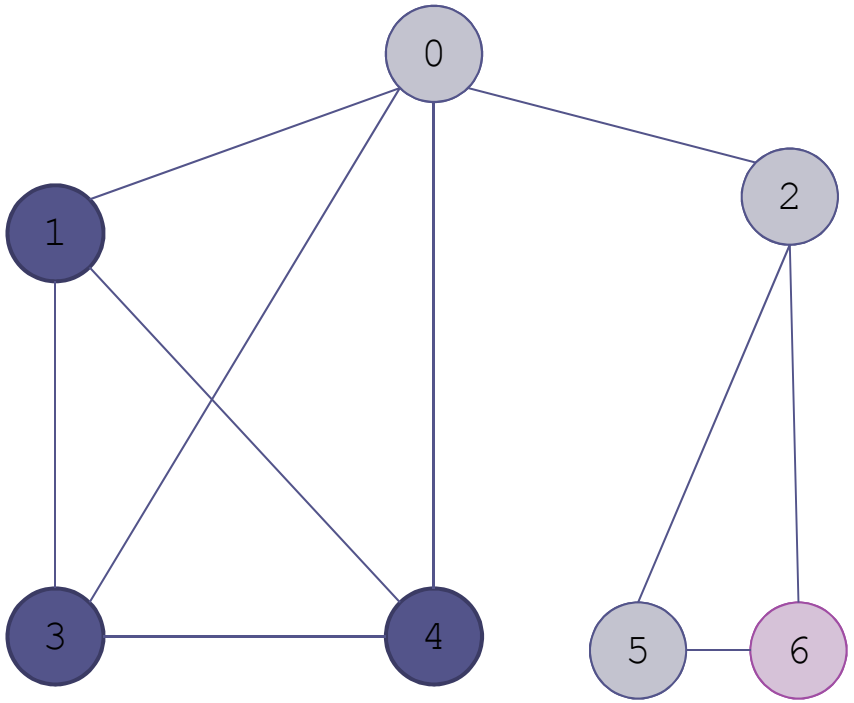
0 unvisited

0 visited

0 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

Finish order:
4, 3, 1

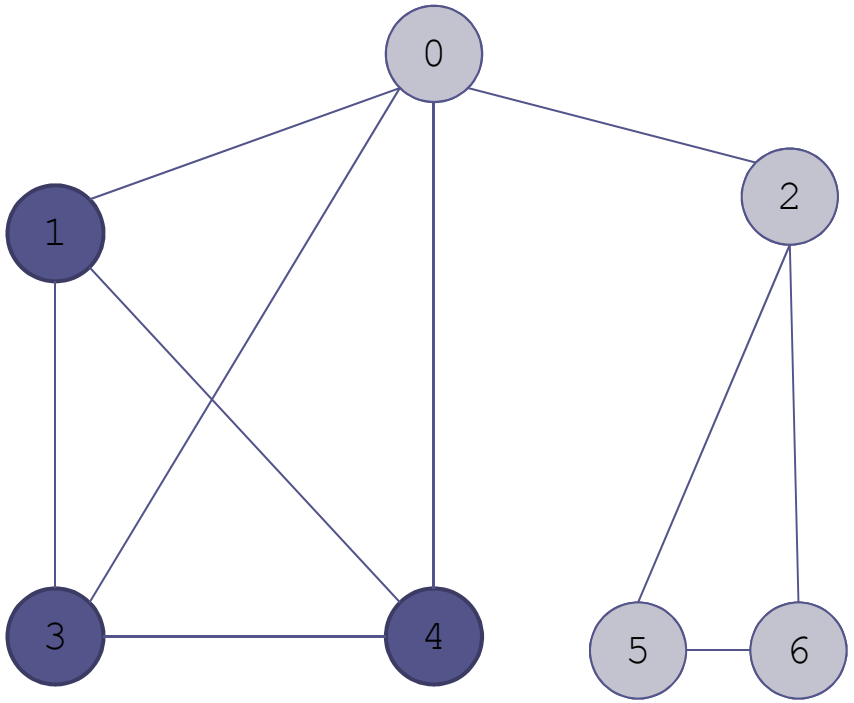
0 unvisited

0 visited

0 being visited

Example of a Depth-First Search (cont.)

6 is adjacent to 5 and is not being visited

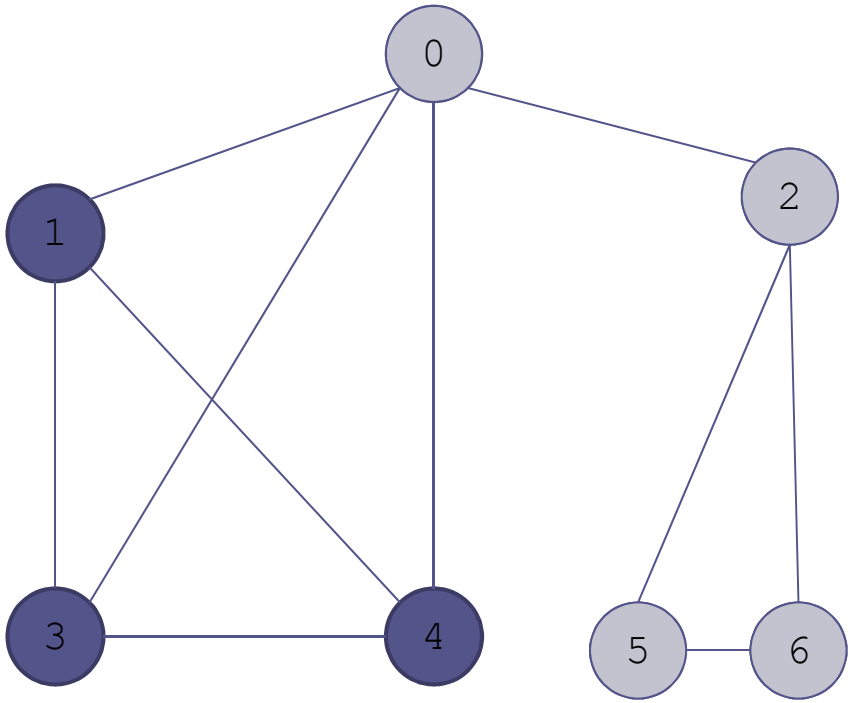


Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

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

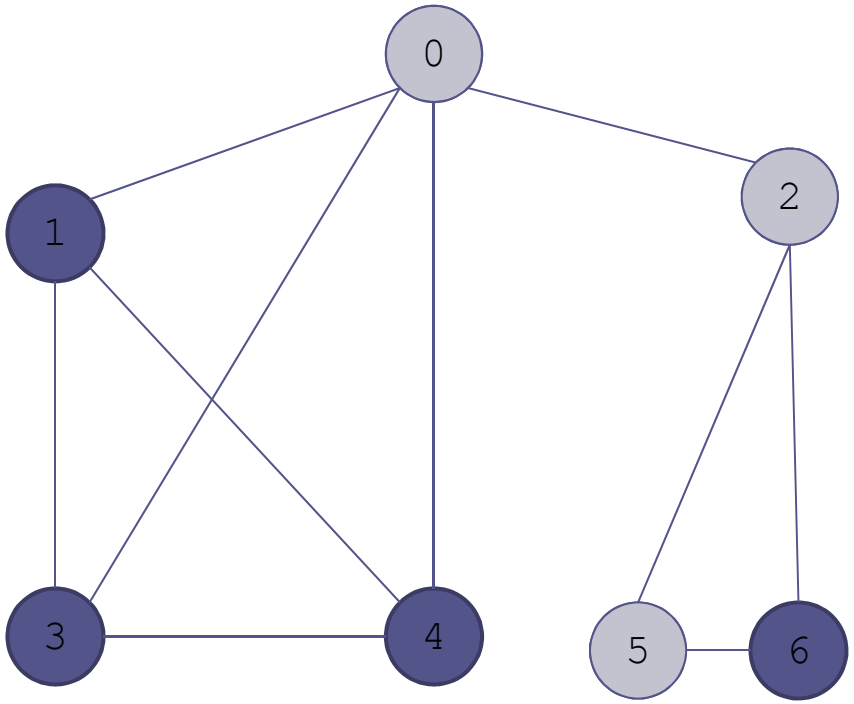


Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1

Example of a Depth-First Search (cont.)

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

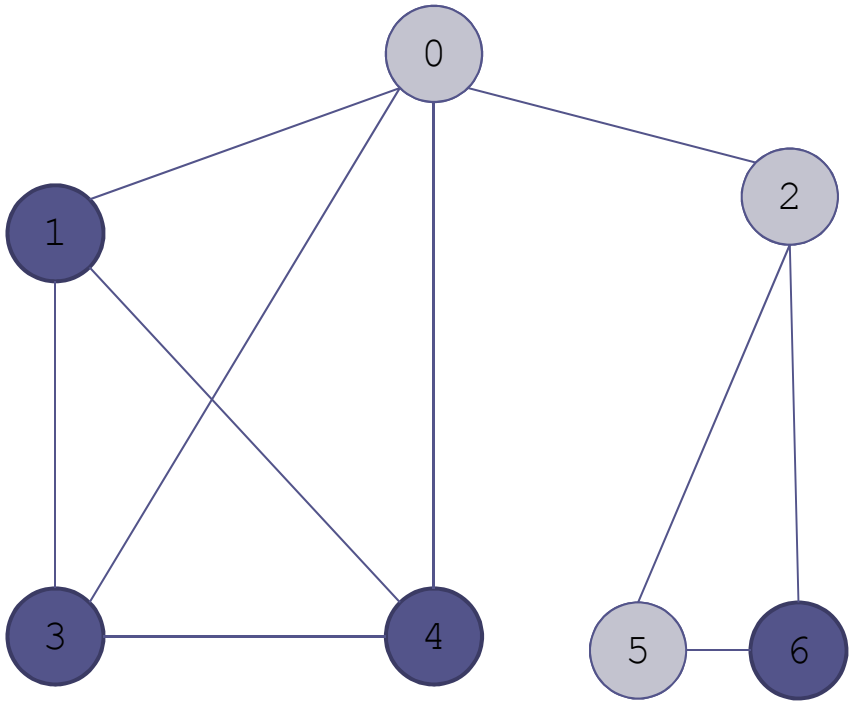


Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1, 6

Example of a Depth-First Search (cont.)

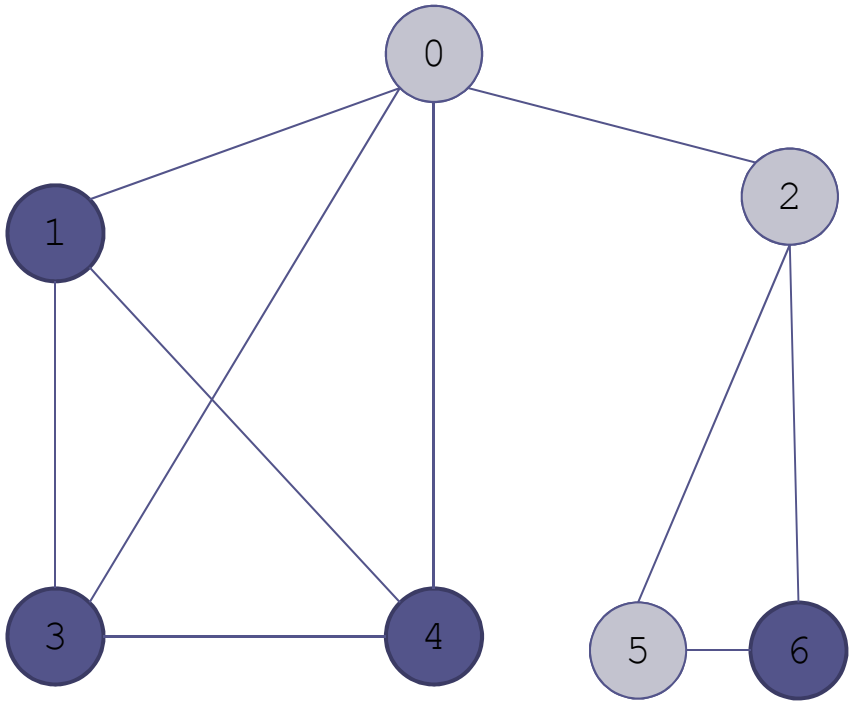
Return from the recursion to 5



Finish order:
4, 3, 1, 6

Example of a Depth-First Search (cont.)

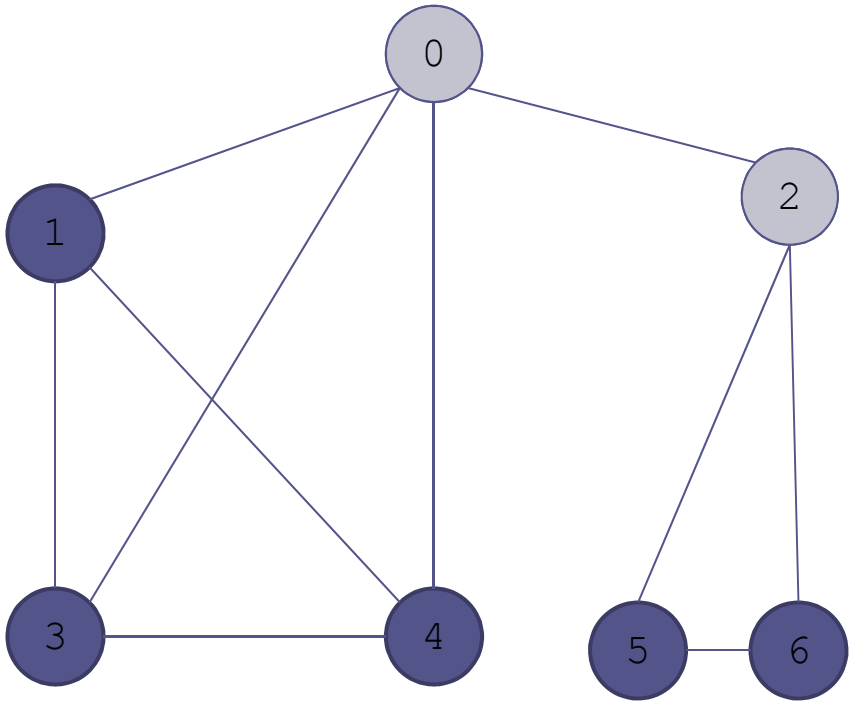
Mark 5 as visited



Finish order:
4, 3, 1, 6

Example of a Depth-First Search (cont.)

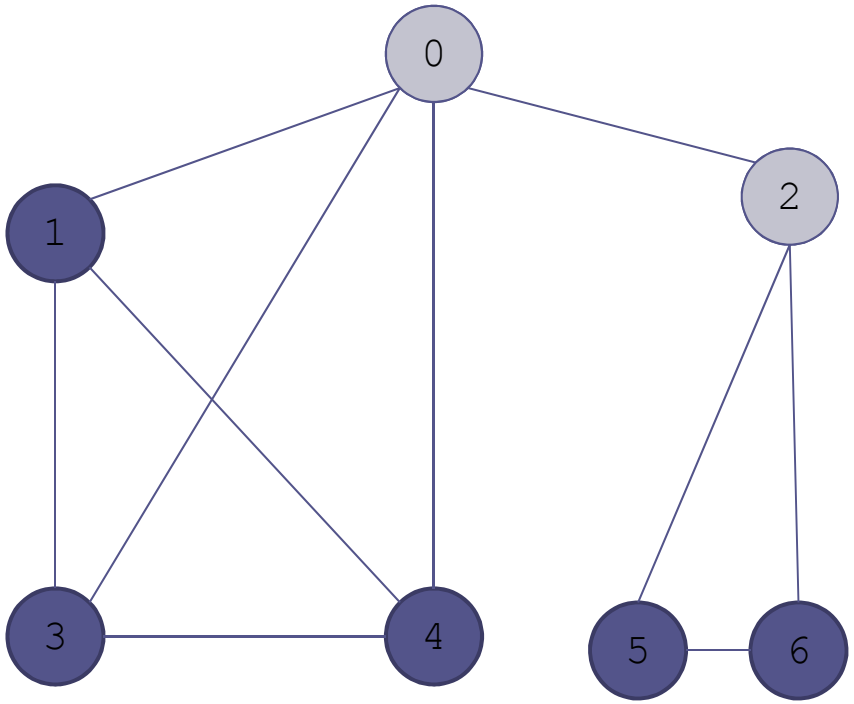
Mark 5 as visited



Finish order:
4, 3, 1, 6, 5

Example of a Depth-First Search (cont.)

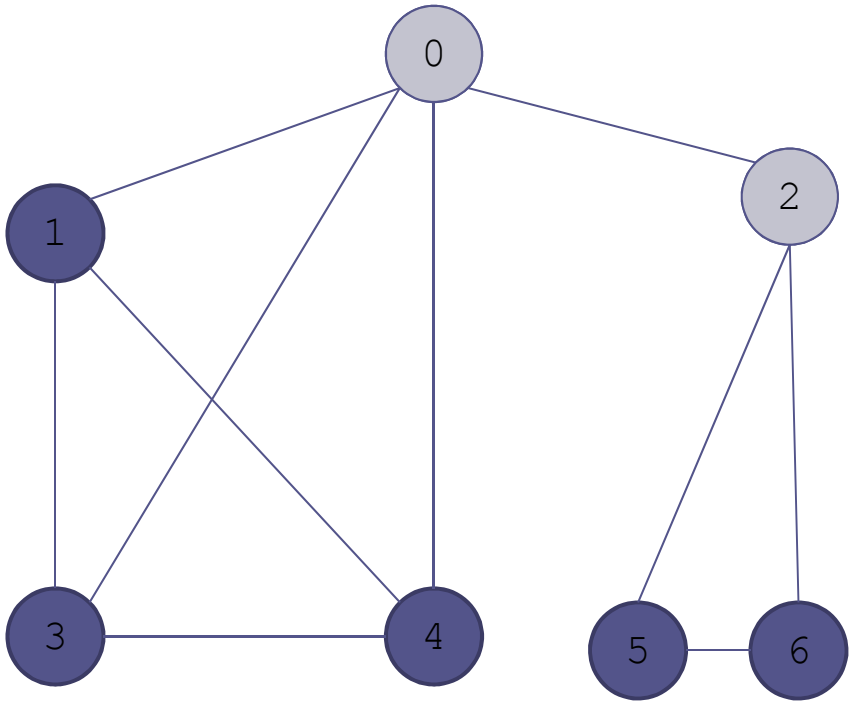
Return from the recursion to 2



Finish order:
4, 3, 1, 6, 5

Example of a Depth-First Search (cont.)

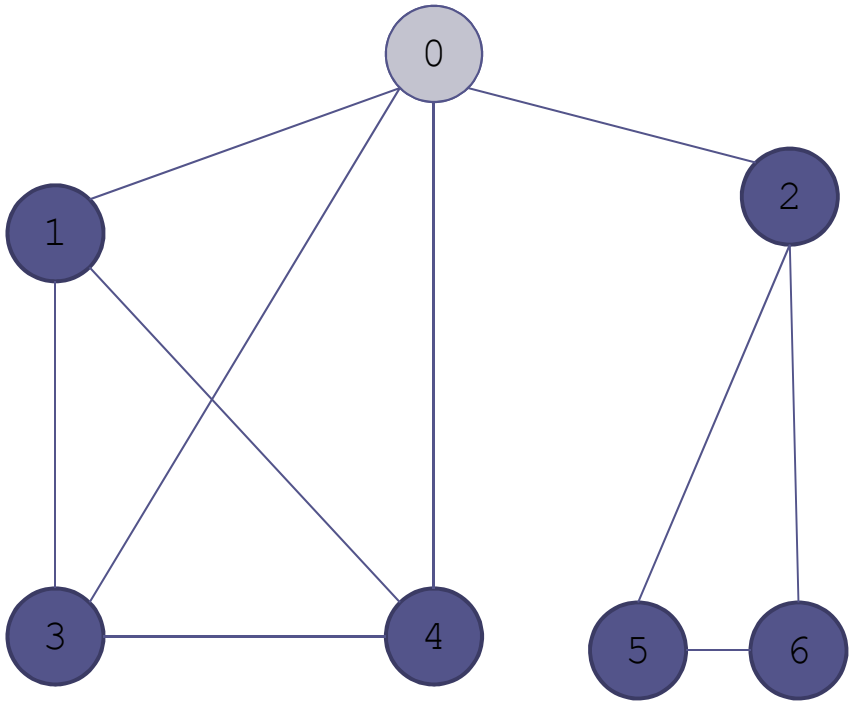
Mark 2 as visited



Finish order:
4, 3, 1, 6, 5

Example of a Depth-First Search (cont.)

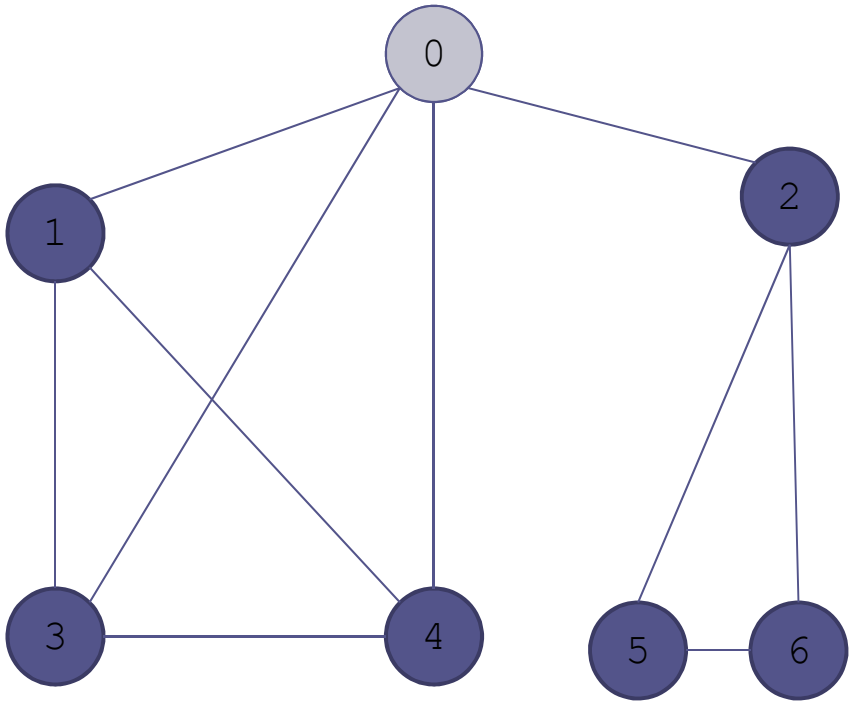
Mark 2 as visited



Finish order:
4, 3, 1, 6, 5, 2

Example of a Depth-First Search (cont.)

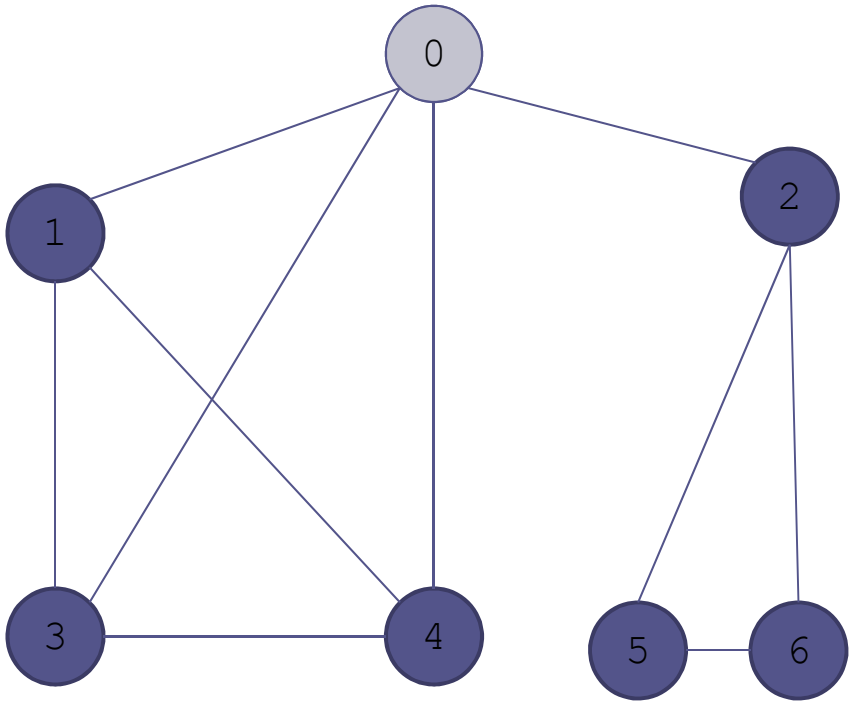
Return from the recursion to 0



Finish order:
4, 3, 1, 6, 5, 2

Example of a Depth-First Search (cont.)

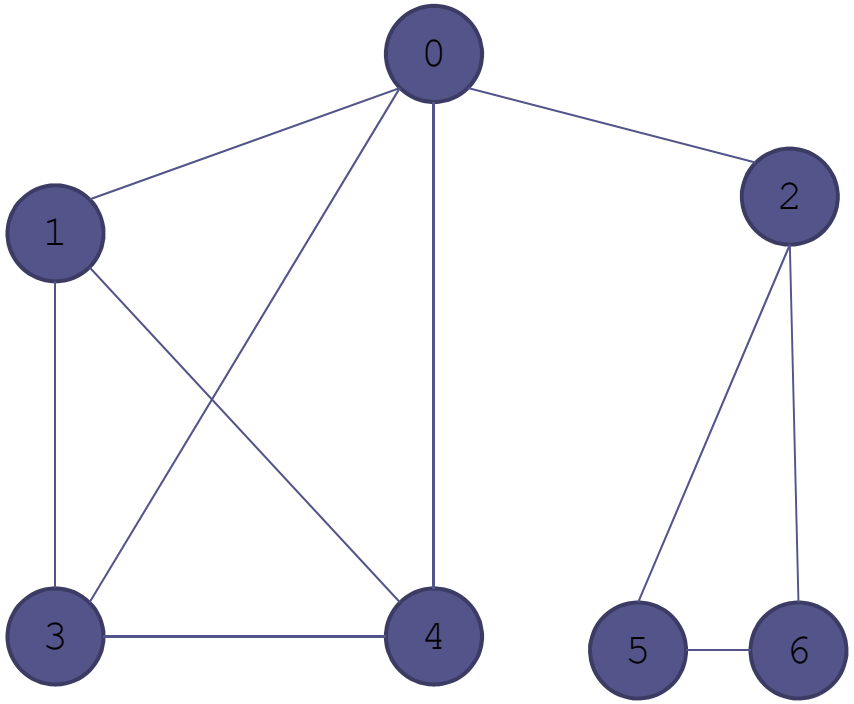
There are no nodes adjacent to 0 not being visited



Finish order:
4, 3, 1, 6, 5, 2

Example of a Depth-First Search (cont.)

Mark 0 as visited



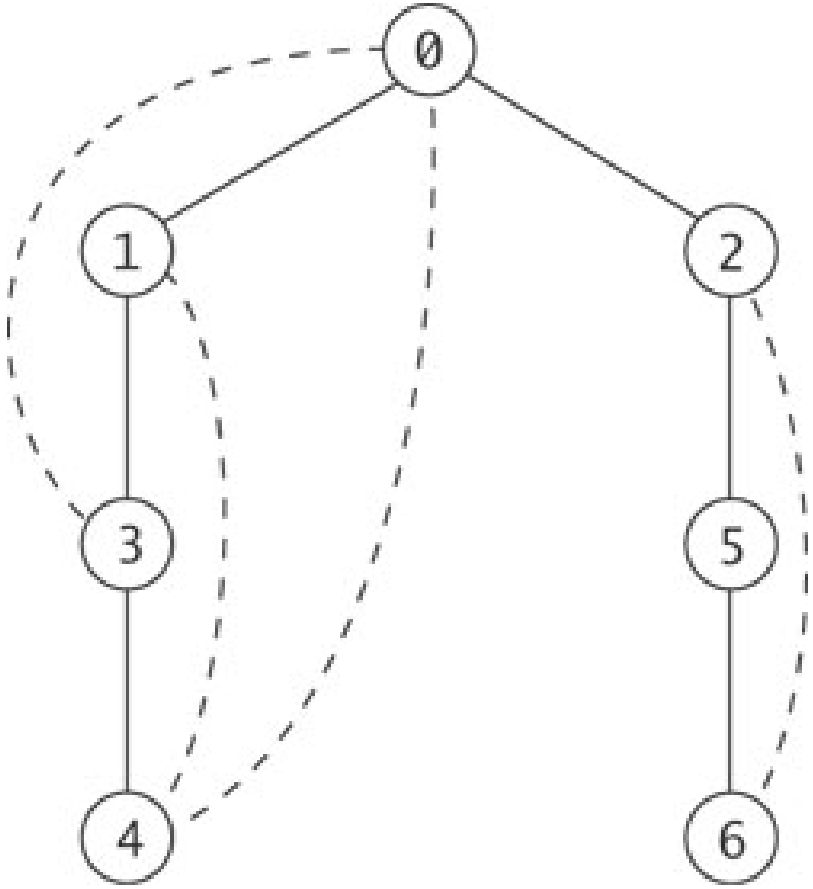
Discovery (Visit) order:
0, 1, 3, 4, 2, 5, 6

Finish order:
4, 3, 1, 6, 5, 2, 0

Search Terms

- The *discovery order* is the order in which the vertices are discovered
  - 0, 1, 3, 4, 2, 5, 6 in this example
- The *finish order* is the order in which the vertices are finished
  - 4, 3, 1, 6, 5, 2, 0 in this example
- *Back edges* connect a vertex with its ancestors in a depth-first search tree

Search Terms (cont.)



Algorithm for Depth-First Search

Algorithm for Depth-First Search

1. Mark the current vertex, *u*, visited (color it light blue), and enter it in the discovery order list
2. for each vertex, *v*, adjacent to the current vertex, *u*
3. if *v* has not been visited
4. Set parent of *v* to *u*.
5. Recursively apply this algorithm starting at *v*.
6. Mark *u* finished (color it dark blue) and enter *u* into the finish order list.

Performance Analysis of Depth-First Search

- The loop at Step 2 is executed $|E_v|$ times
- The recursive call results in this loop being applied to each vertex
- The total number of steps is the sum of the edges that originate at each vertex, which is the total number of edges, |E|
- The algorithm is O(|E|)
- An implicit Step 0 marks all of the vertices as unvisited – O(|V|)
- The total running time of the algorithm is O(|V| + |E|)

Implementing Depth-First Search

- `class DepthFirstSearch` performs a depth-first search on a graph and records the
  - start time
  - finish time
  - start order
  - finish order
- For an unconnected graph or for a directed graph, a depth-first search may not visit each vertex in the graph
- Thus, once the recursive method returns, all vertices need to be examined to see if they have been visited—if not the process repeats on the next unvisited vertex
- Thus, a depth-first search may generate more than one tree
- A collection of unconnected trees is called a *forest*

Implementing Depth-First

Data Field	Attribute
`private int discoverIndex`	The index that indicates the discovery order.
`private int[] discoveryOrder`	The array that contains the vertices in discovery order.
`private int finishIndex`	The index that indicates the finish order.
`private int[] finishOrder`	The array that contains the vertices in finish order.
`private Graph graph`	A reference to the graph being searched.
`private int[] parent`	The array of predecessors in the depth-first search tree.
`private boolean[] visited`	An array of **`boolean`** values to indicate whether or not a vertex has been visited.
Constructor	**Purpose**
`public DepthFirstSearch(Graph graph)`	Constructs the depth-first search of the specified graph selecting the start vertices in ascending vertex order.
`public DepthFirstSearch(Graph graph, int[] order)`	Constructs the depth-first search of the specified graph selecting the start vertices in the specified order. The first vertex visited is `order[0]`.
Method	**Behavior**
`public void depthFirstSearch(int s)`	Recursively searches the graph starting at vertex `s`.
`public int[] getDiscoveryOrder()`	Gets the discovery order.
`public int[] getFinishOrder()`	Gets the finish order.
`public int[] getParent()`	Gets the parents in the depth-first search tree.

Implementing Depth-First Search (cont.)

```
public DepthFirstSearch(Graph graph) {
    this.graph = graph;
    int n = graph.getNumV();
    parent = new int[n];
    visited = new boolean[n];
    discoveryOrder = new int[n];
    finishOrder = new int[n];
    for (int i = 0; i < n; i++) {
        parent[i] = -1;
    }
    for (int i = 0; i < n; i++) {
        if (!visited[i])
            depthFirstSearch(i);
    }
}
```

Implementing Depth-First Search (cont.)

```
public void depthFirstSearch(int current) {
    /* Mark the current vertex visited. */
    visited[current] = true;
    discoveryOrder[discoverIndex++] = current;
    /* Examine each vertex adjacent to the current vertex */
    Iterator<Edge> itr = graph.edgeIterator(current);
    while (itr.hasNext()) {
        int neighbor = itr.next().getDest();
        /* Process a neighbor that has not been visited */
        if (!visited[neighbor]) {
            /* Insert (current, neighbor) into the depth-
               first search tree. */
            parent[neighbor] = current;
            /* Recursively apply the algorithm
               starting at neighbor. */
            depthFirstSearch(neighbor);
        }
    }
    /* Mark current finished. */
    finishOrder[finishIndex++] = current;
}
```

Testing Method DepthFirstSearch

```
/** Main method to test depth-first search method
    pre: args[0] is the name of the input file.
    @param args The command line arguments
*/
public static void main(String[] args) {
    Graph g = null;
    int n = 0;
    try {
          Scanner scan =
              new Scanner(new File(args[0]));
          g = AbstractGraph.createGraph(scan, true, "List");
          n = g.getNumV();
    } catch (IOException ex) {
          ex.printStackTrace();
          System.exit(1); // Error
    }

    // Perform depth-first search.
    DepthFirstSearch dfs = new DepthFirstSearch(g);
    int[] dOrder = dfs.getDiscoveryOrder();
    int[] fOrder = dfs.getFinishOrder();
    System.out.println("Discovery and finish order");
    for (int i = 0; i < n; i++) {
          System.out.println(dOrder[i] + " " + fOrder[i]);
    }
}
```

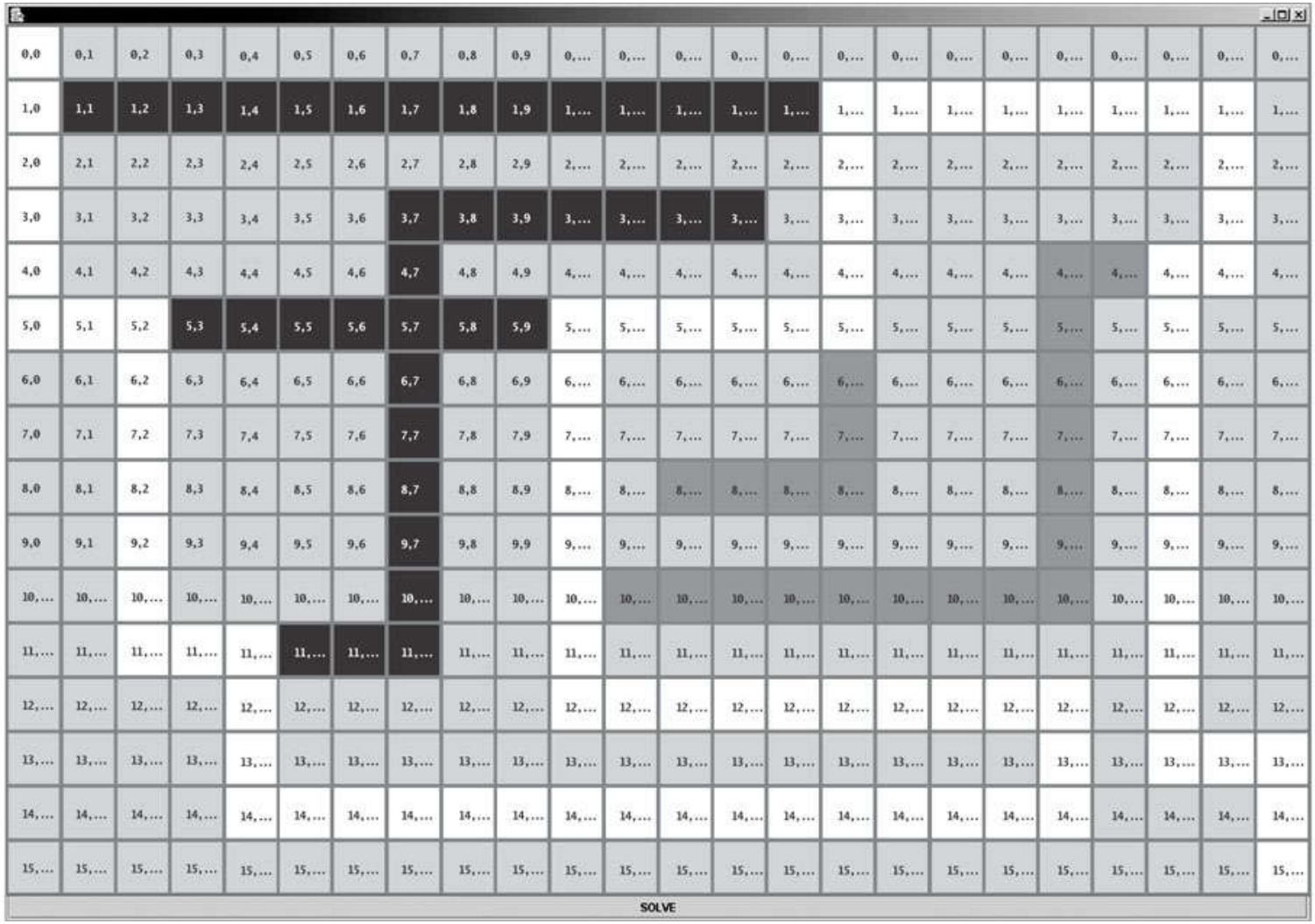
Application of Graph Traversals

Section 10.5

Problem

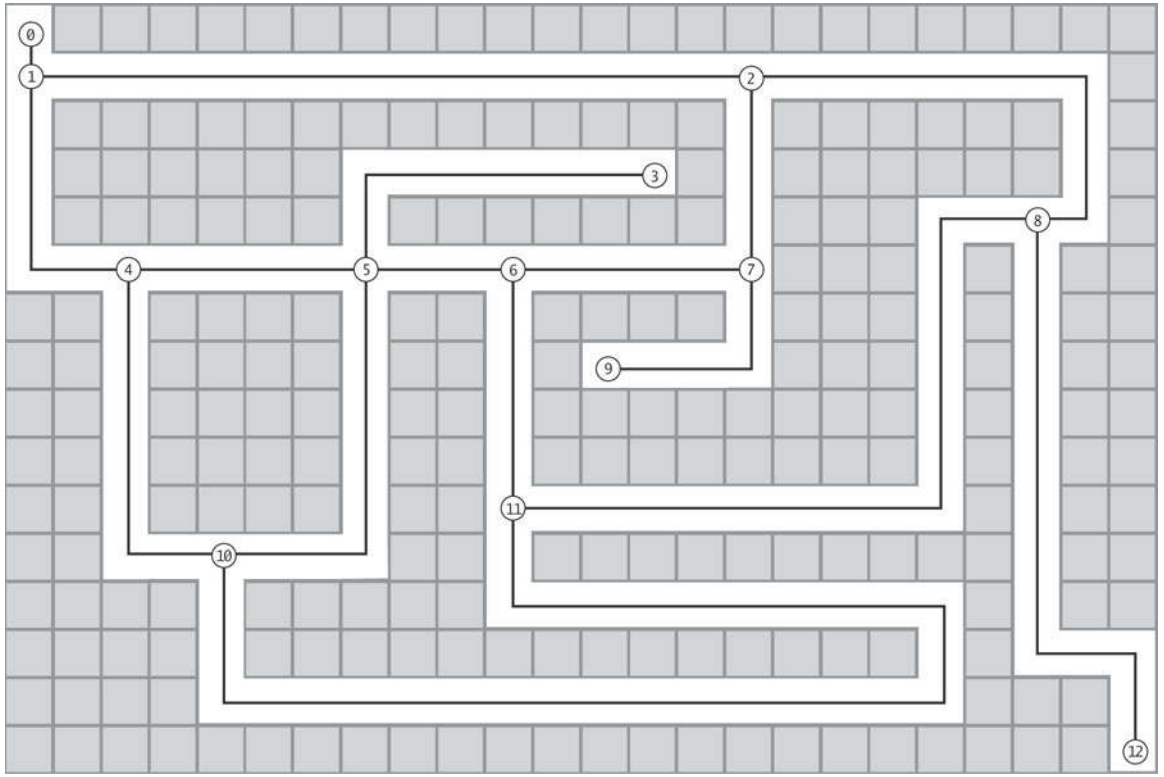
- Design a program that will find the shortest path through a maze

Problem (cont.)



Analysis

- We can represent the maze on the previous slide as a graph, with a node at each decision point



Analysis (cont.)

- With the maze represented as a graph, we need to find the shortest path from the start point (0) to the end point (12)
- The breadth-first search method will return an array of parents which we can use to find the shortest path to the end point
  - the smallest number of vertices
  - but not necessarily the smallest number of cells

Design

- The program needs the following data structures:
  - an external representation of the maze, consisting of the number of vertices and the edges
  - an object of a class that implements the `Graph` interface
  - an array to hold the predecessors returned from the `breadthFirstSearch` method
  - A stack to reverse the path

Design (cont.)

Algorithm for Shortest Path

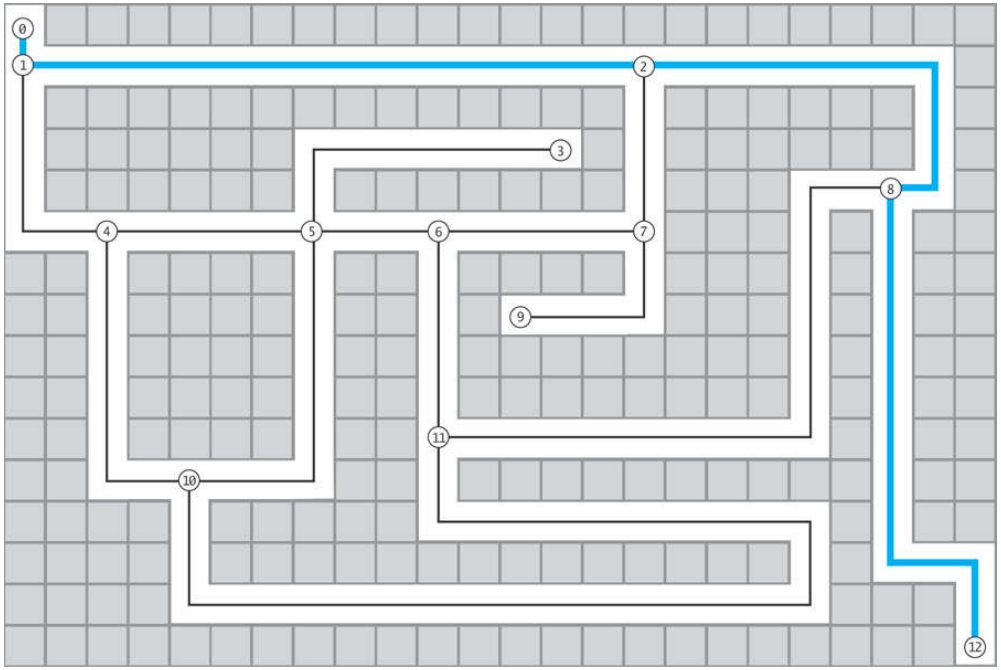
1. Read in the number of vertices and create the graph object.
2. Read in the edges and insert the edges into the graph.
3. Call the **`breadthFirstSearch`** method with this graph and the starting vertex as its argument. The method returns the array parent.
4. Start at v, the end vertex.
5. **`while`** v is not –1
6. Push v onto the stack.
7. Set v to parent[v].
8. **`while`** the stack is not empty
9. Pop a vertex off the stack and output it.

Implementation

- Listing 10.5 (`Maze.java`, pages 575-576)

Testing

- Test the program with a variety of mazes.
- Uses mazes for which the original recursive program (Chapter 5) finds the shortest path and those for which it does not

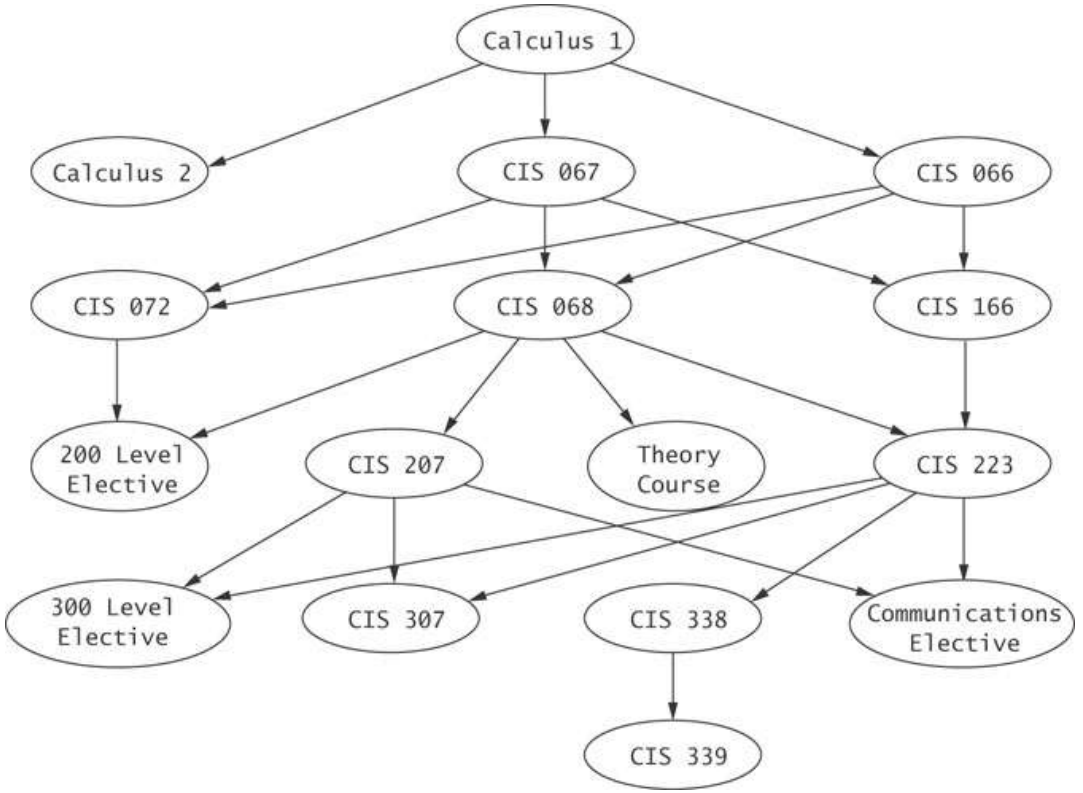


Topological Sort of a Graph

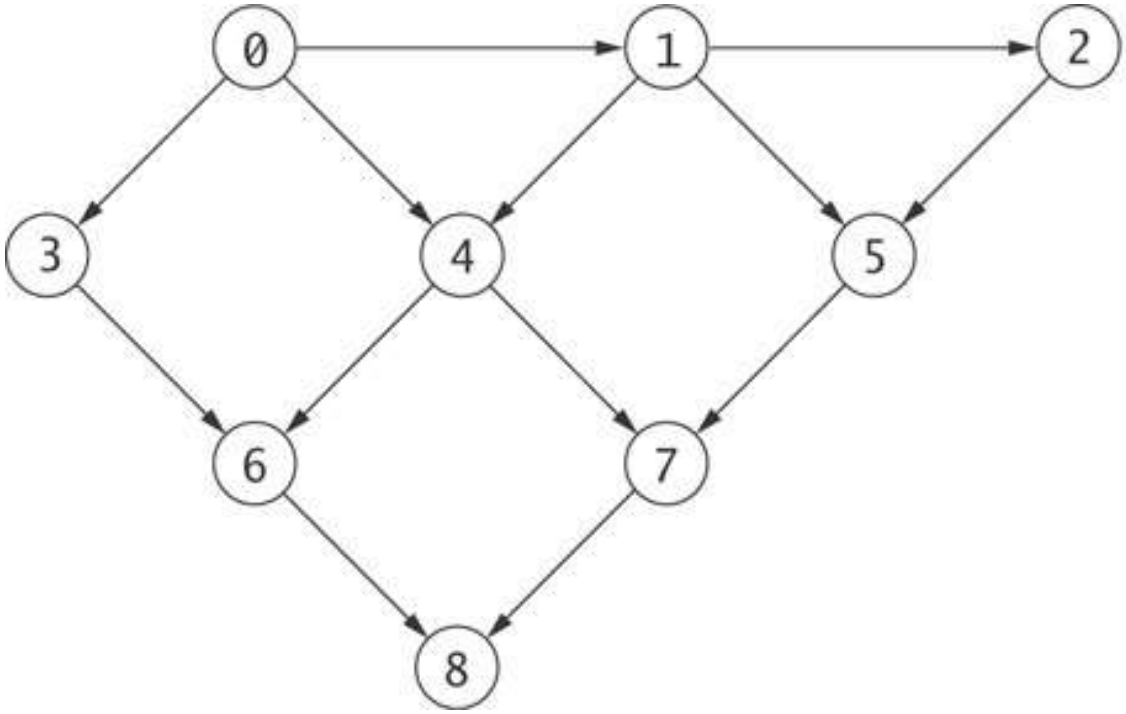
This is an example of a *directed acyclic graph* (*DAG*).

It is a directed graph which contains no cycles.

Once you pass through a vertex, there is no path back to the vertex



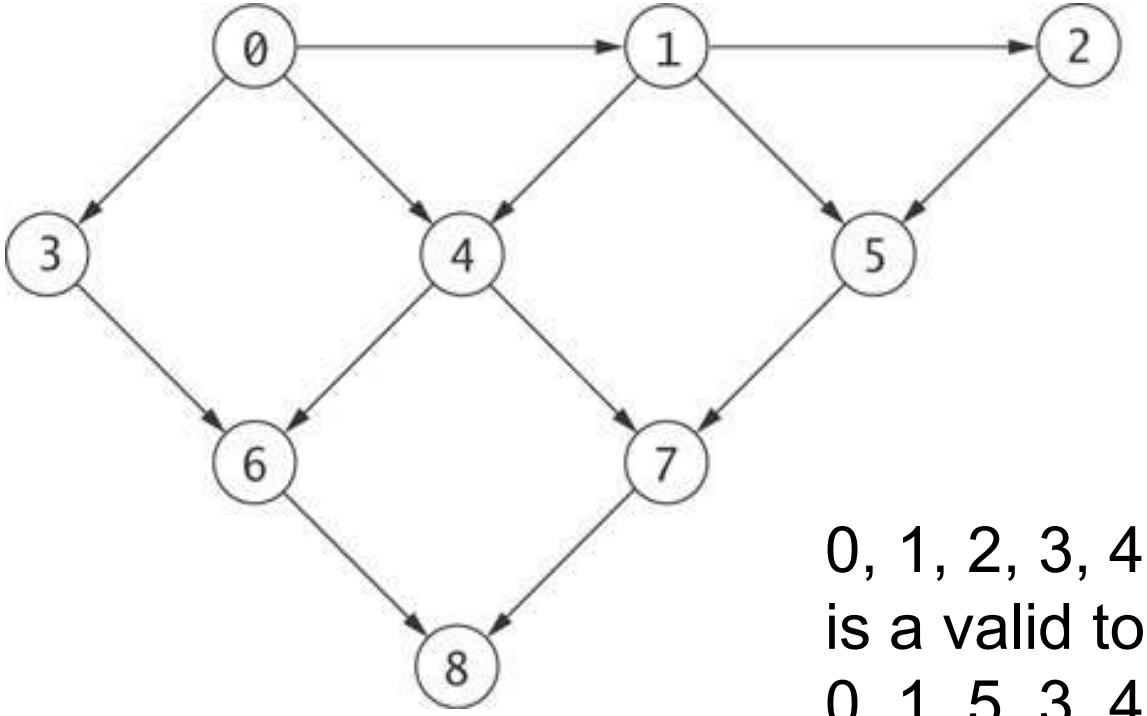
Another Directed Acyclic Graph (DAG)



Topological Sort of a Graph (cont.)

- A *topological sort* of the vertices of a DAG is an ordering of the vertices such that is (*u*, *v*) is an edge, the *u* appears before *v*
- This must be true for all edges
- There may be many valid paths through a DAG and many valid topographical sorts of a DAG

Topological Sort of a Graph (cont.)



0, 1, 2, 3, 4, 5, 6, 7, 8
is a valid topological sort, but
0, 1, 5, 3, 4, 2, 6, 7, 8 is not

Analysis

- If there is an edge from *u* to *v* in a DAG,
  - then if we perform a depth-first search of the graph
  - the finish time of *u* must be after the finish time of *v*
- When we return to *u*, either *v* has not been visited or it has finished
- It is not possible that *v* would be visited but not finished (a loop or cycle would exist)

Design

Algorithm for Topological Sort

1. Read the graph from a data file.
2. Perform a depth-first search of the graph.
3. List the vertices in reverse of their finish order.

Implementation

- Listing 10.6 (`TopologicalSort.java`, page 579)

Testing

- Test the program on several different graphs
- Use sparse graphs and dense graphs
- Make sure each graph you test has no loops or cycles

Algorithms Using Weighted Graphs

Section 10.6

Finding the Shortest Path from a Vertex to All Other Vertices

- The breadth-first search found the shortest path from the start vertex to all other vertices, assuming that the length or weight of each edge was the same
- Dijkstra's algorithm finds the shortest path in a weighted directed graph

Finding the Shortest Path from a Vertex to All Other Vertices (cont.)

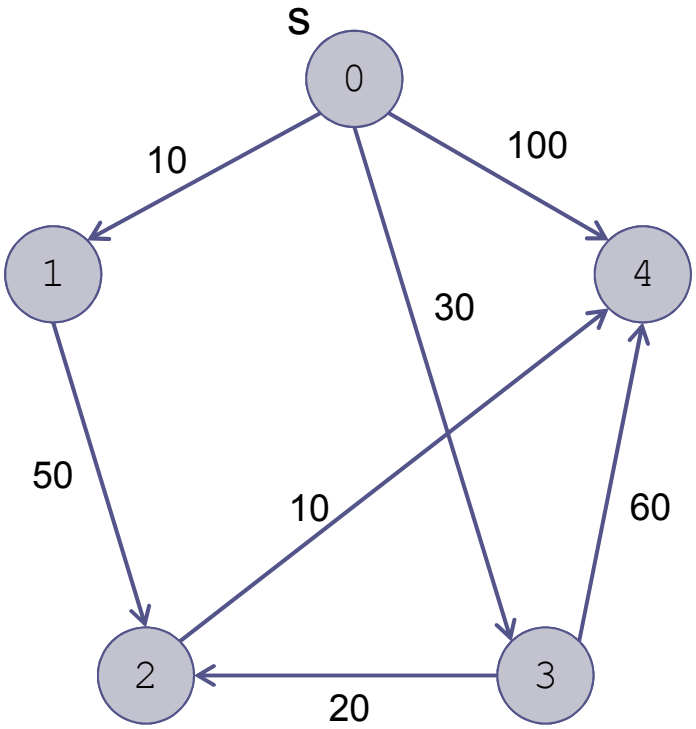
- We need 2 sets and 2 arrays
  - Set *S* will contain the vertices for which we have computed the shortest distance
    - Initialize *S* by placing the starting vertex *s* into it
  - Set *V-S* will contain the vertices we still need to process
    - Initialize *V-S* by placing the remaining vertices into it
  - d[*v*] will contain the shortest distance from *s* to *v*
    - Initially, for each *v* in *V-S*, set d[*v*] to the weight of the edge *w(s,v)* for each vertex *v* adjacent to *s* and to ∞ for the other vertices
  - p[*v*] will contain the predecessor of *v* in the path from *s* to *v*

Dijkstra's Algorithm

$S = \{\ \}$

$V\text{-}S = \{\ \}$

v	d[*v*]	p[*v*]
1		
2		
3		
4		

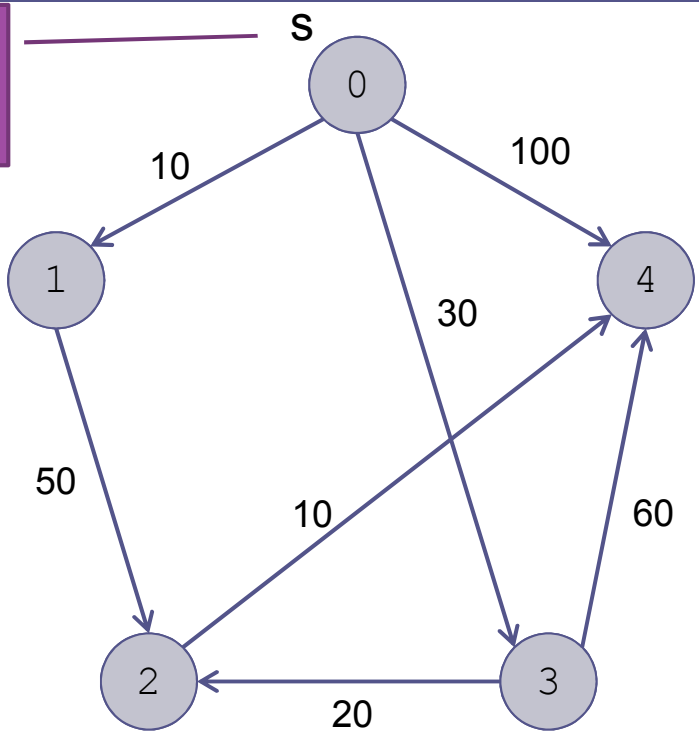


Dijkstra's Algorithm (cont.)

$S = \{\ \}$

$V\text{-}S = \{\ \}$

s is the start vertex



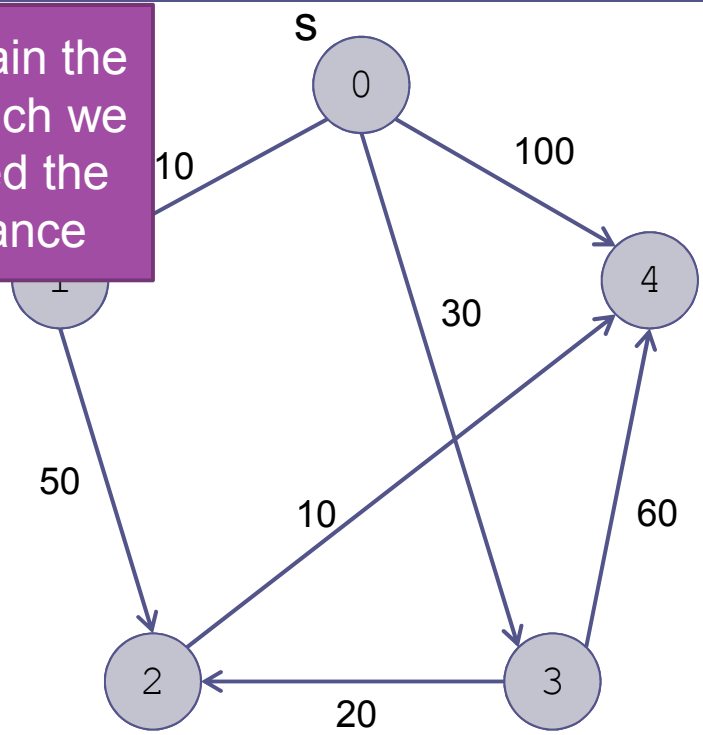
v	d[*v*]	p[*v*]
1		
2		
3		
4		

Dijkstra's Algorithm (cont.)

$S = \{\ \}$

$V\text{-}S = \{\ \}$

Set S will contain the vertices for which we have computed the shortest distance



v	d[v]	p[v]
1		
2		
3		
4		

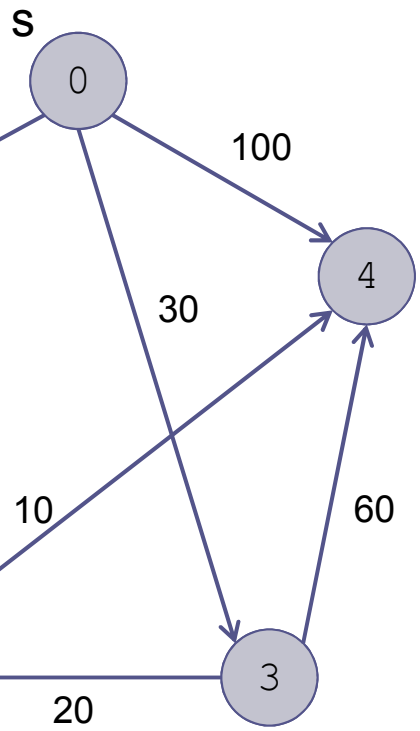
Dijkstra's Algorithm (cont.)

$S = \{\}$

$V\text{-}S = \{\ \}$

Set *V-S* will contain the vertices that we still need to process

v	d[*v*]	p[*v*]
1		
2		
3		
4		



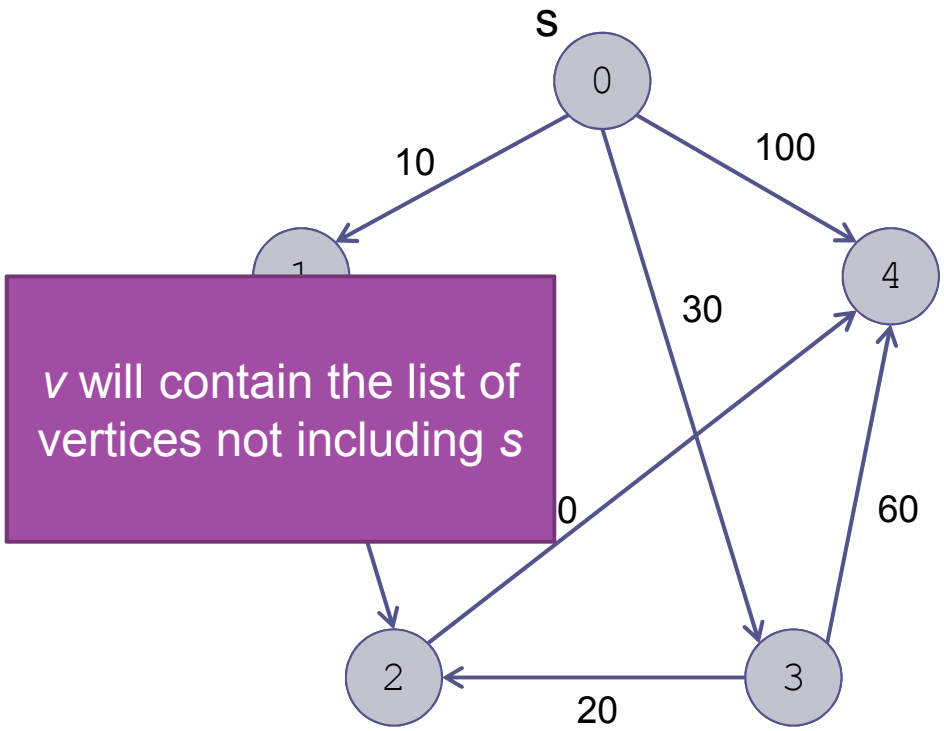
Dijkstra's Algorithm (cont.)

$S = \{\ \}$

$V\text{-}S = \{\ \}$

v	d[*v*]	p[*v*]
1		
2		
3		
4		

v will contain the list of vertices not including *s*



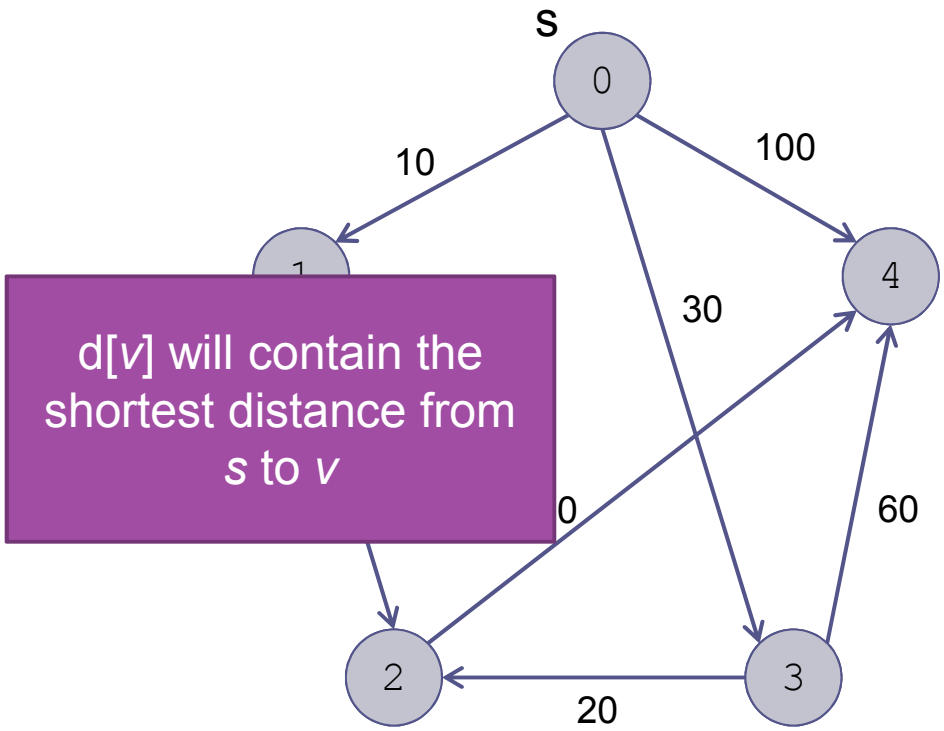
Dijkstra's Algorithm (cont.)

$S = \{\ \}$

$V\text{-}S = \{\ \}$

v	d[v]	p[v]
1		
2		
3		
4		

d[v] will contain the shortest distance from s to v



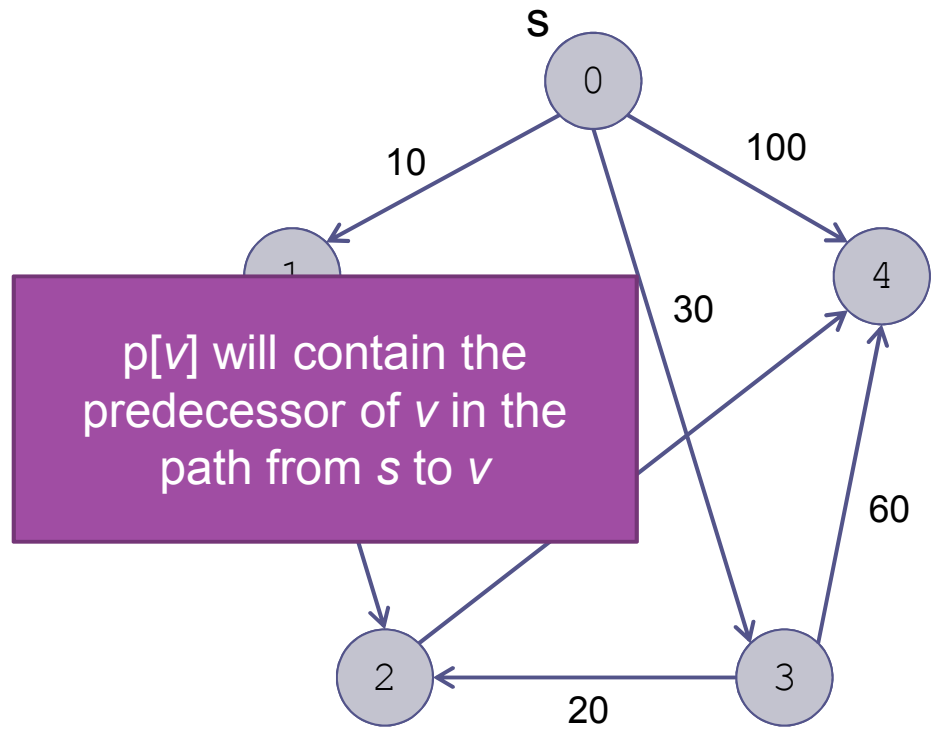
Dijkstra's Algorithm (cont.)

$S = \{\ \}$

$V\text{-}S = \{\ \}$

v	d[v]	p[v]
1		
2		
3		
4		

p[v] will contain the predecessor of v in the path from s to v

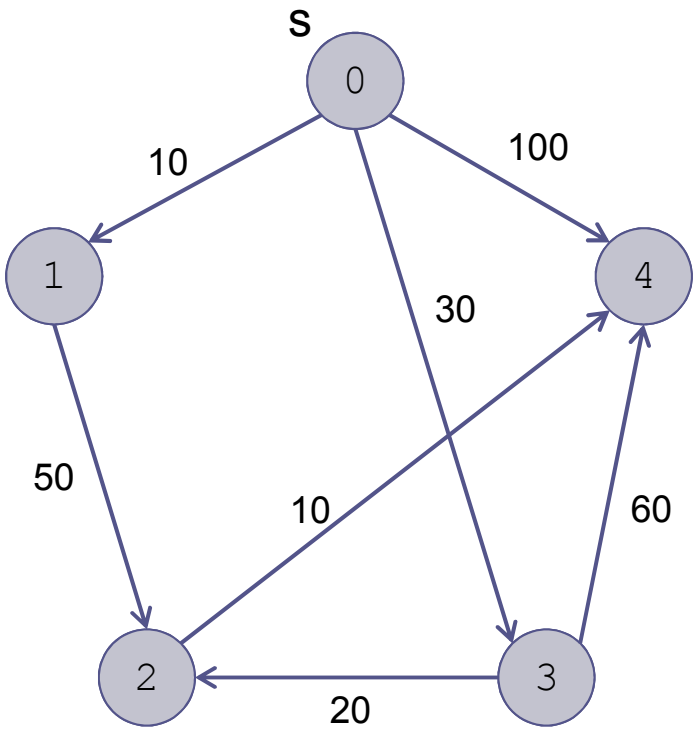


Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

v	d[v]	p[v]
1		
2		
3		
4		



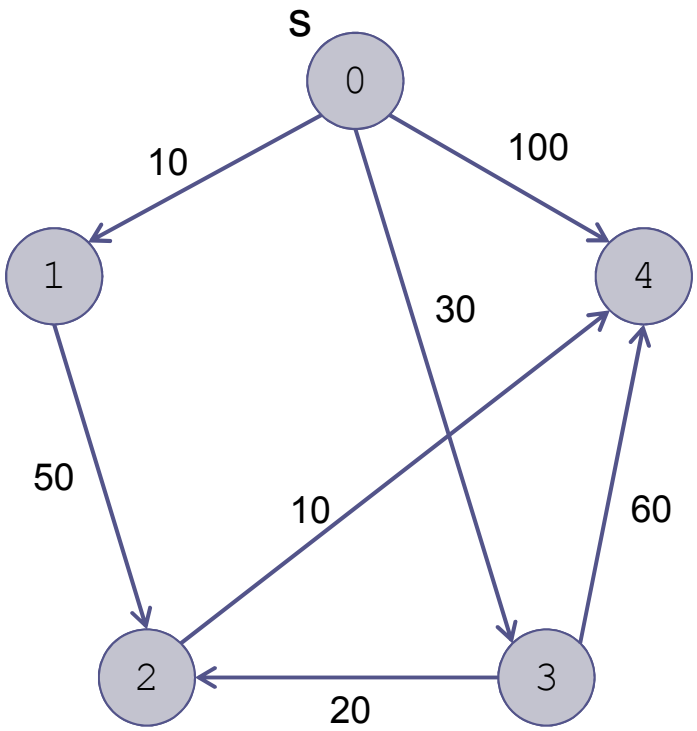
At initialization, the start vertex s is placed in S, and the remaining vertices into V-S

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1		
2		
3		
4		



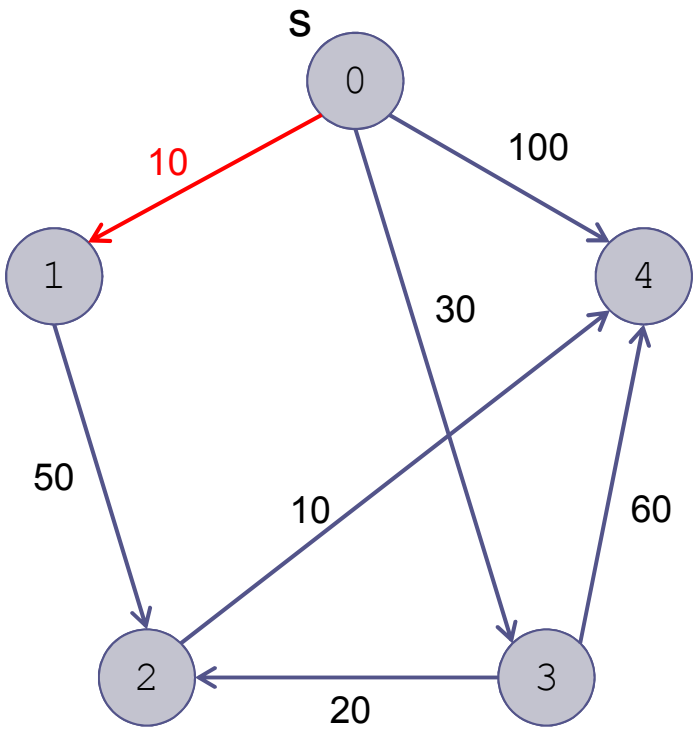
For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	
2		
3		
4		



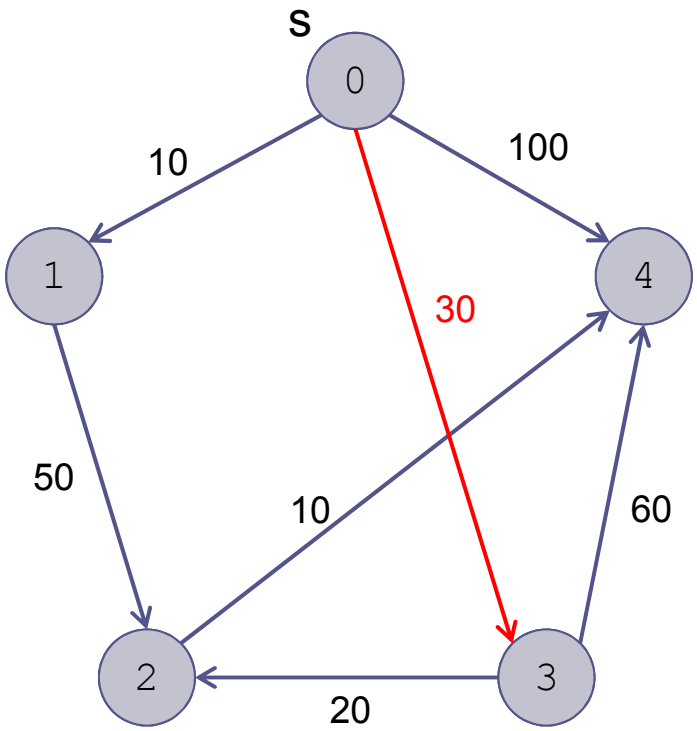
For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	
2		
3	30	
4		



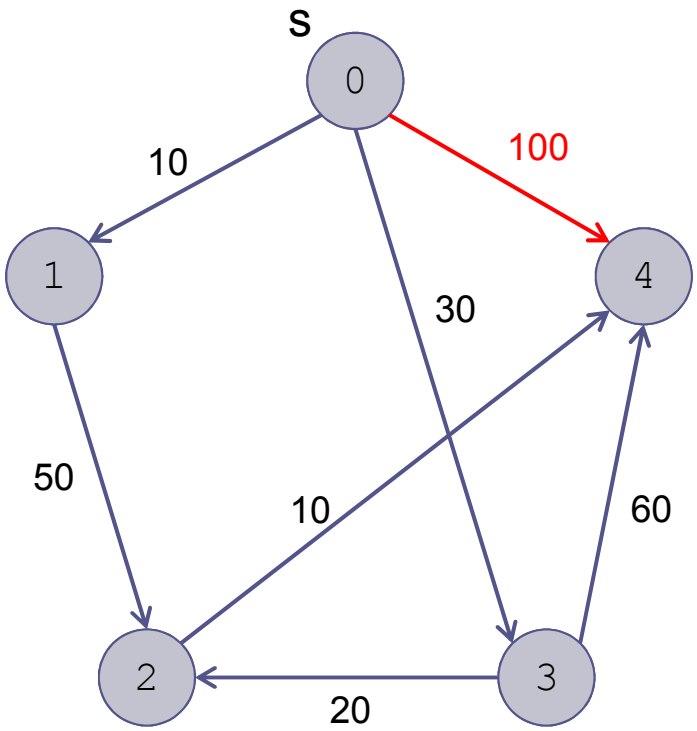
For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	
2		
3	30	
4	100	



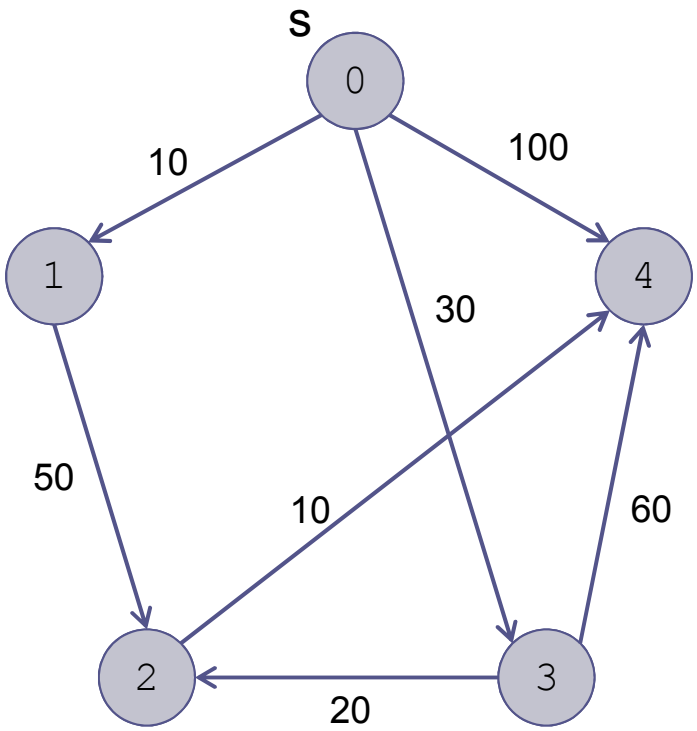
For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	
2		
3	30	
4	100	



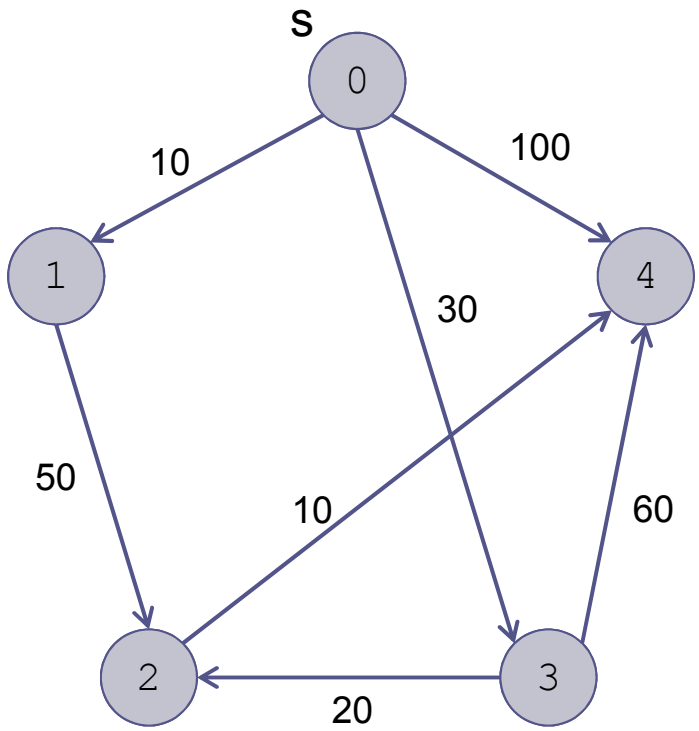
For each *v* not adjacent to *s*, we set d[*v*] equal to ∞

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	
2	∞	
3	30	
4	100	



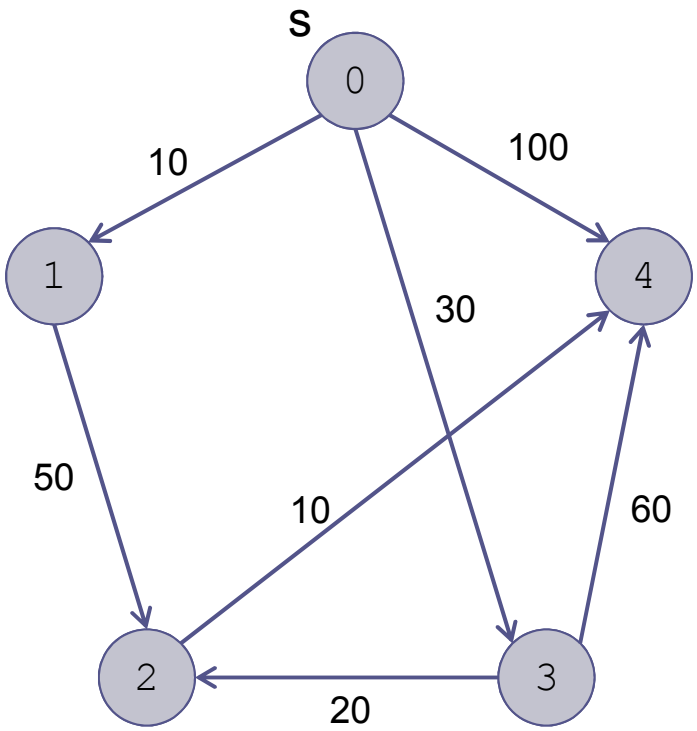
For each *v* not adjacent to *s*, we set d[*v*] equal to ∞

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	
2	∞	
3	30	
4	100	



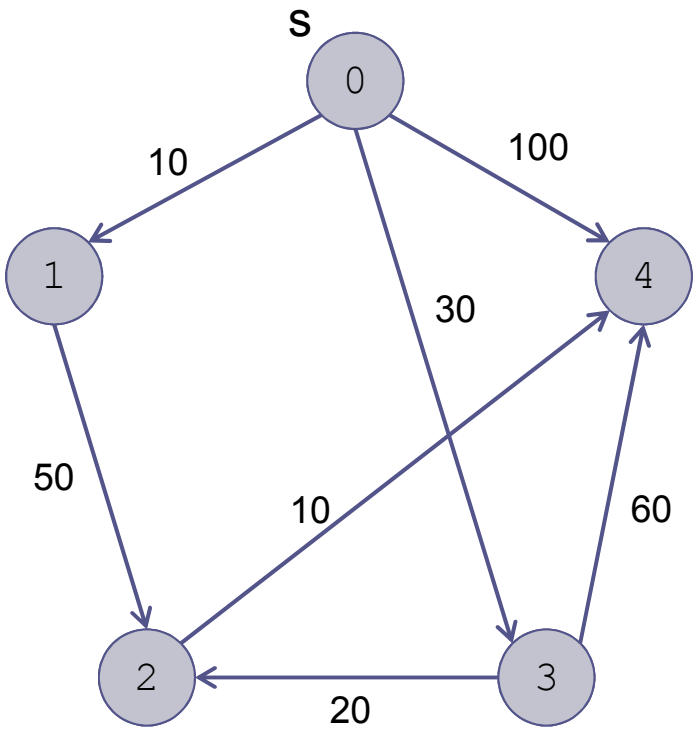
We initialize each p[*v*] to *s*

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	0
2	∞	0
3	30	0
4	100	0



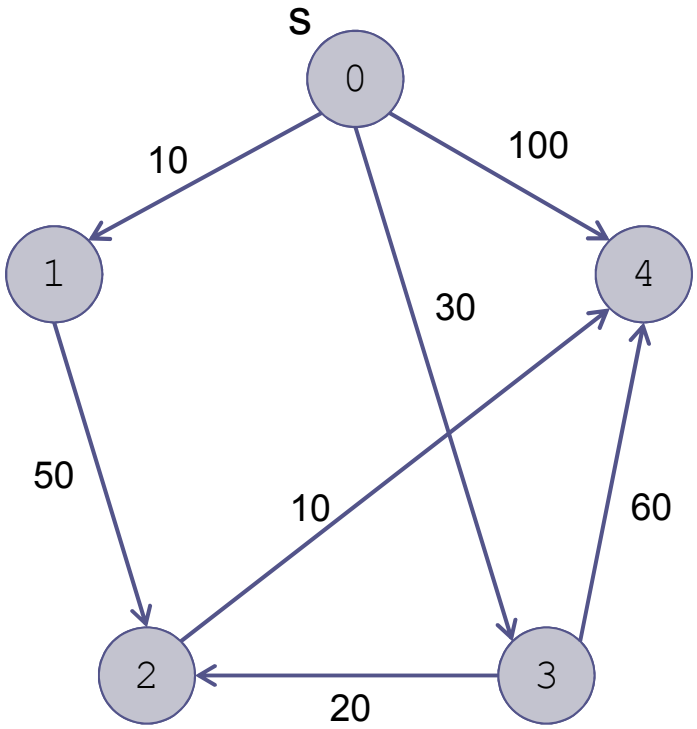
We initialize each p[*v*] to *s*

Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

v	d[*v*]	p[*v*]
1	10	0
2	∞	0
3	30	0
4	100	0



We now find the vertex *u* in *V-S* that has the smallest value of d[*u*]

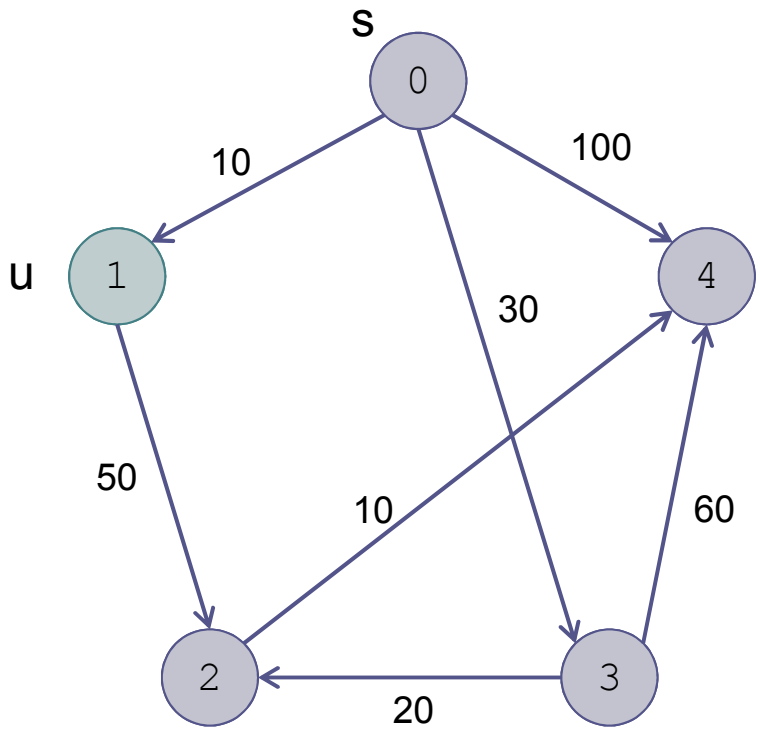
Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0



We now find the vertex u in V-S that has the smallest value of d[u]

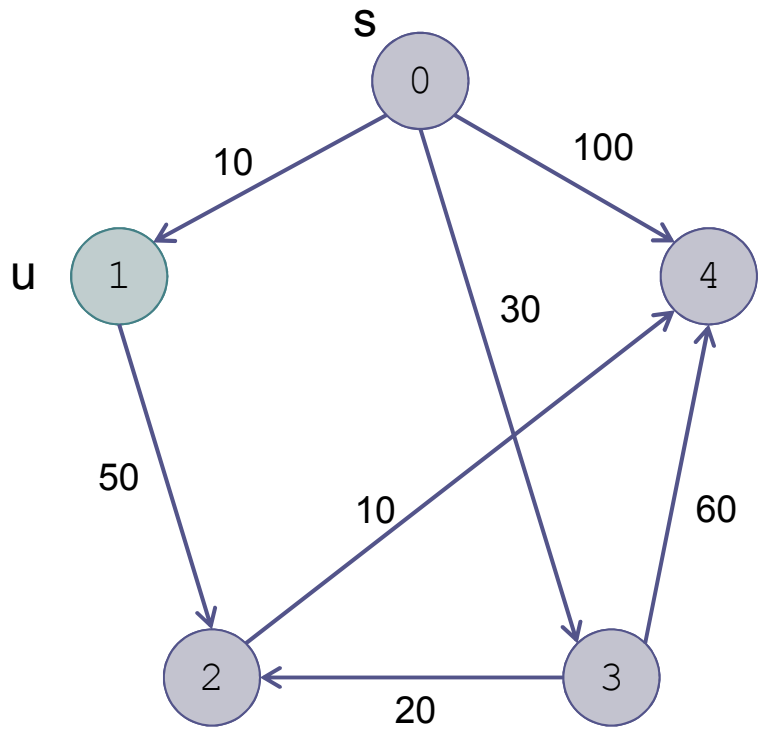
Dijkstra's Algorithm (cont.)

$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

$u = 1$

v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0



Consider the vertices v that are adjacent to u

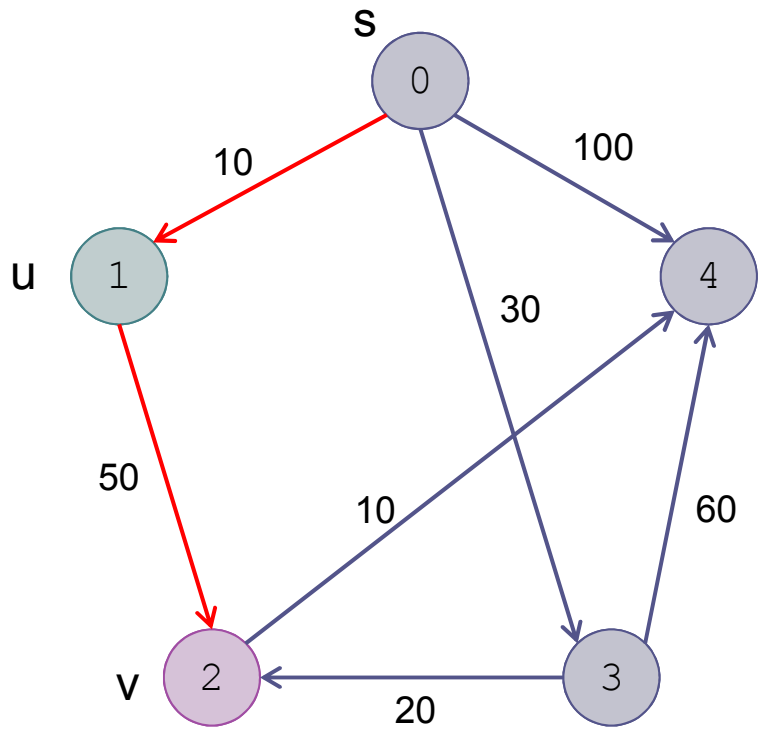
Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0



If the distance from s to u (d[u]) plus the distance from u to v is smaller than d[v] we update d[v] to that value

Dijkstra's Algorithm (cont.)

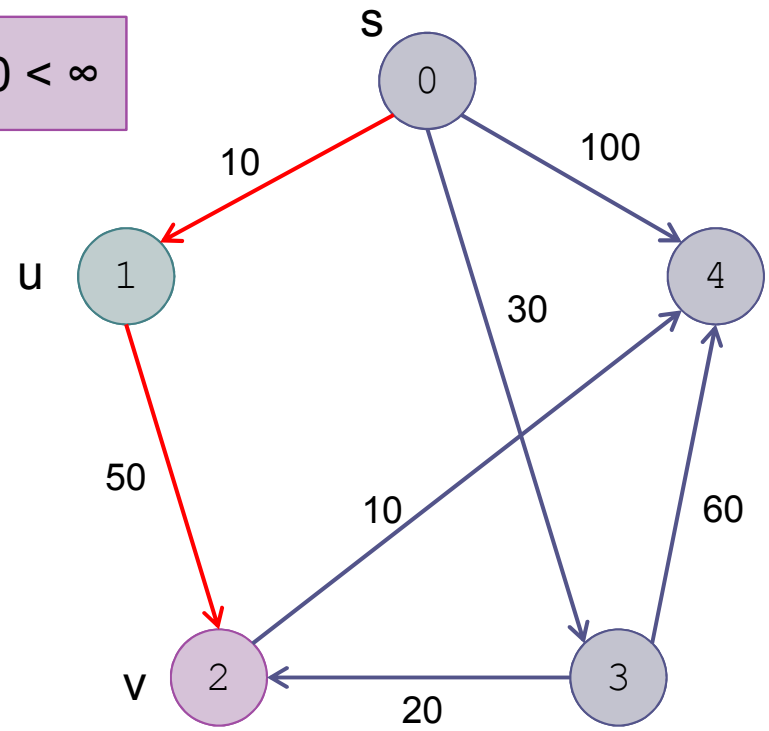
$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

$u = 1$

v	d[*v*]	p[*v*]
1	10	0
2	∞	0
3	30	0
4	100	0

60 < ∞



If the distance from *s* to *u* (d[*u*]) plus the distance from u to v is smaller than d[*v*] we update d[*v*] to that value

Dijkstra's Algorithm (cont.)

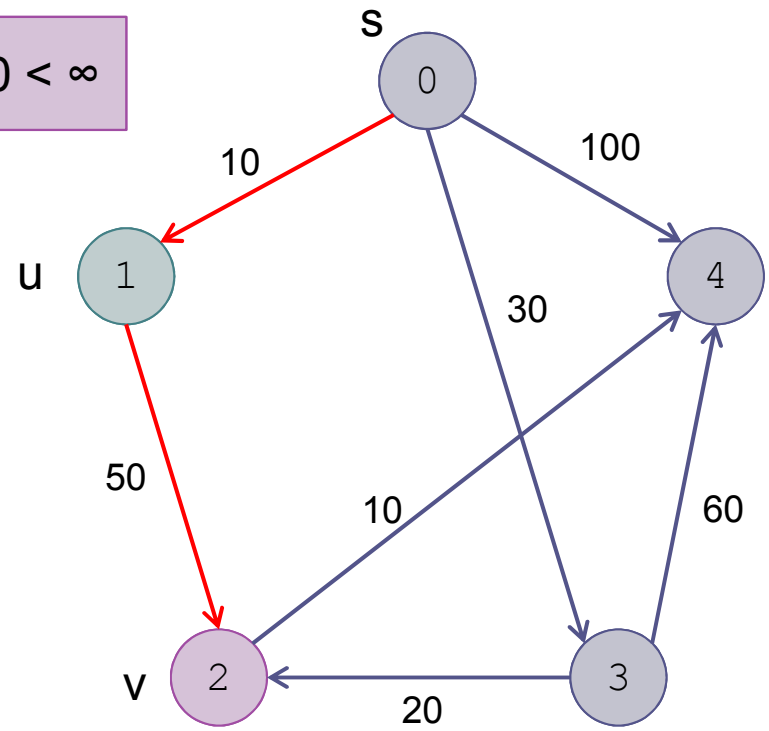
$S = \{ 0 \}$

$V\text{-}S = \{ 1, 2, 3, 4 \}$

$u = 1$

v	d[*v*]	p[*v*]
1	10	0
2	60	0
3	30	0
4	100	0

60 < ∞



If the distance from *s* to *u* (d[*u*]) plus the distance from u to v is smaller than d[*v*] we update d[*v*] to that value

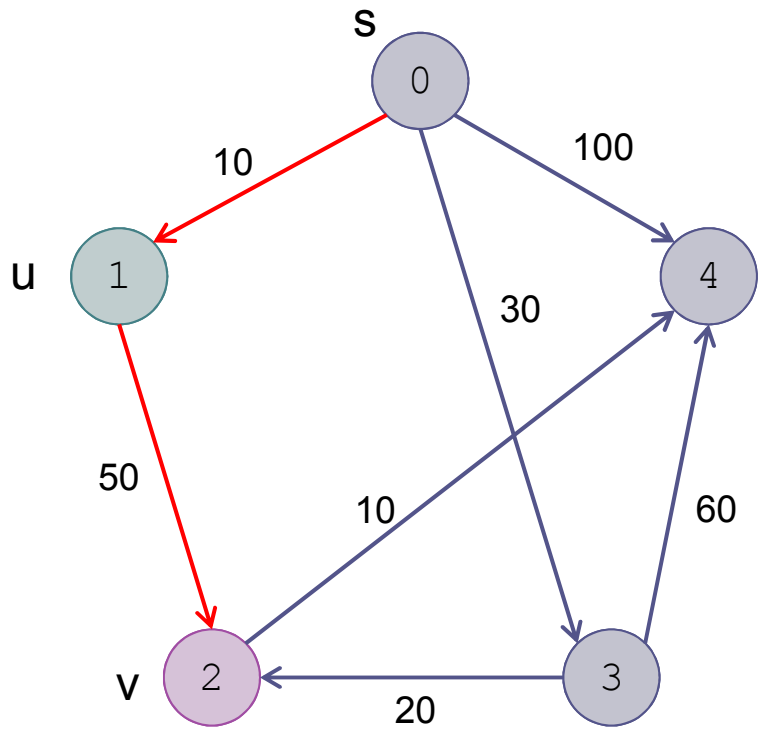
Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0



and set p[v] to u

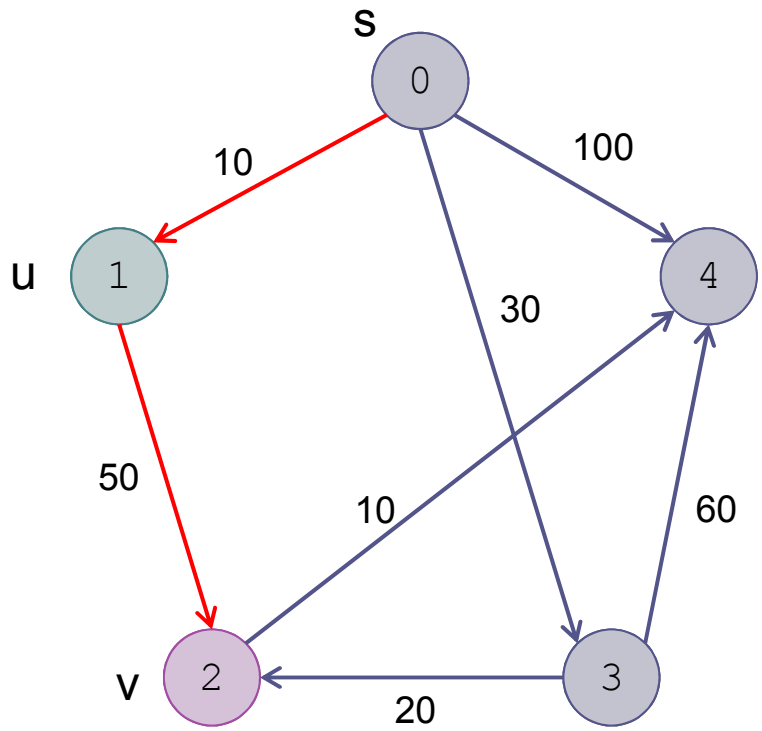
Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0



Remove u from V-S and place it in S

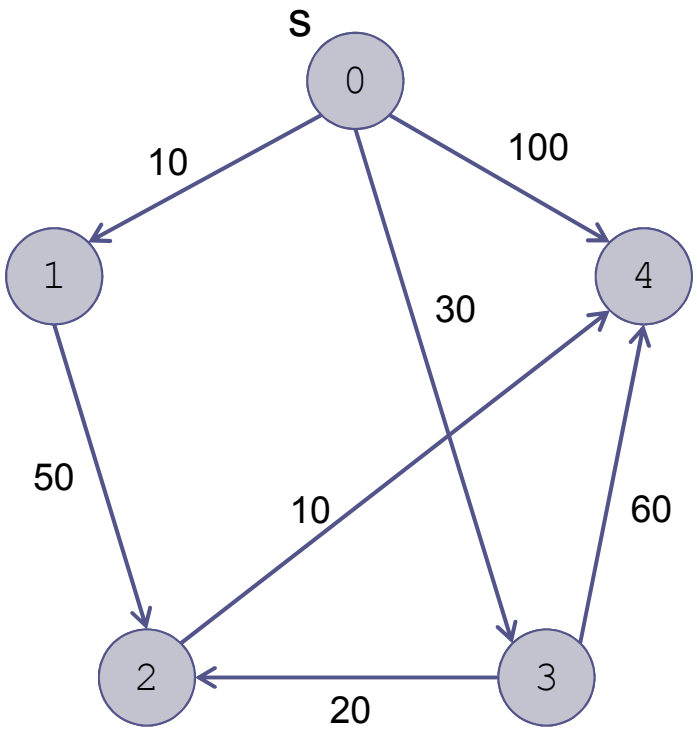
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0



Repeat until V-S is empty

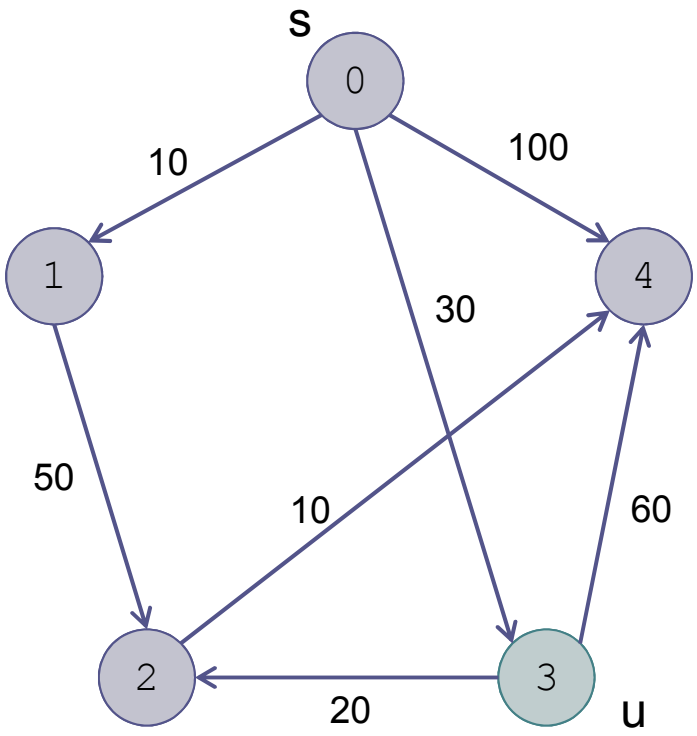
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0



The smallest d[v] in V-S is vertex 3

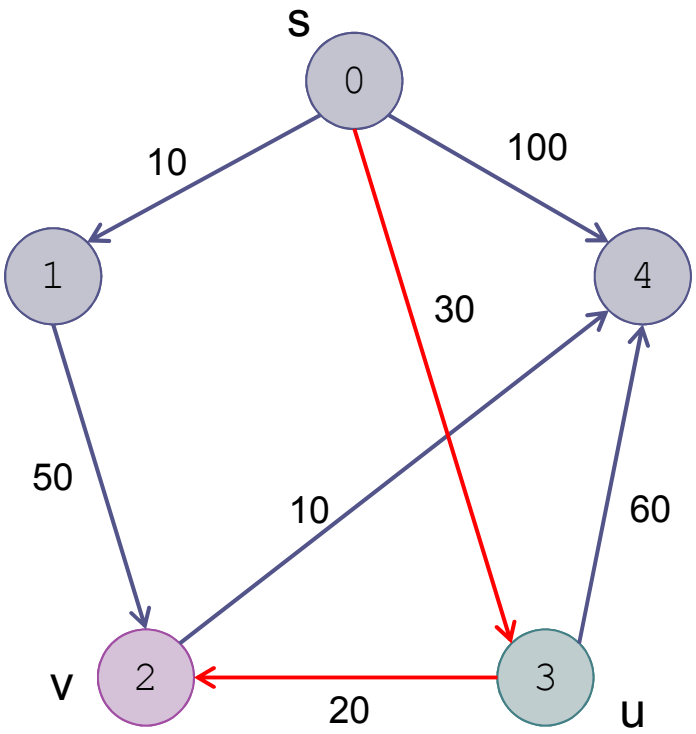
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0



The distance from s to u plus the distance from u to v is 50

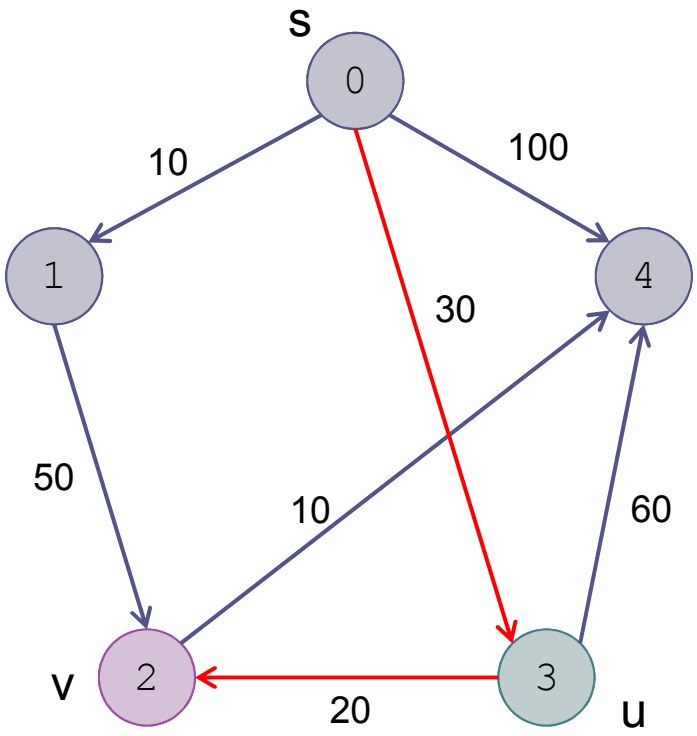
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0



50 < d[2] (which is 60)

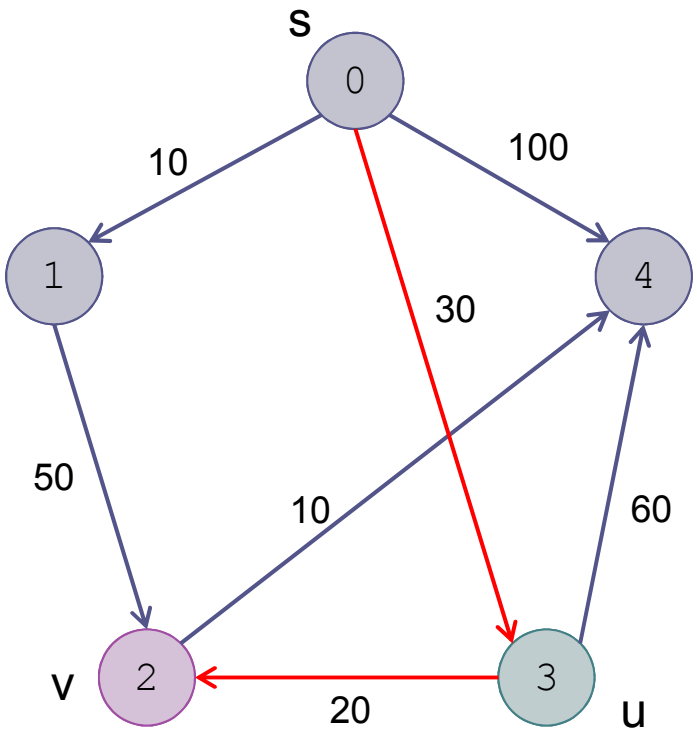
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	100	0



Set d[2] to 50 and p[2] to u

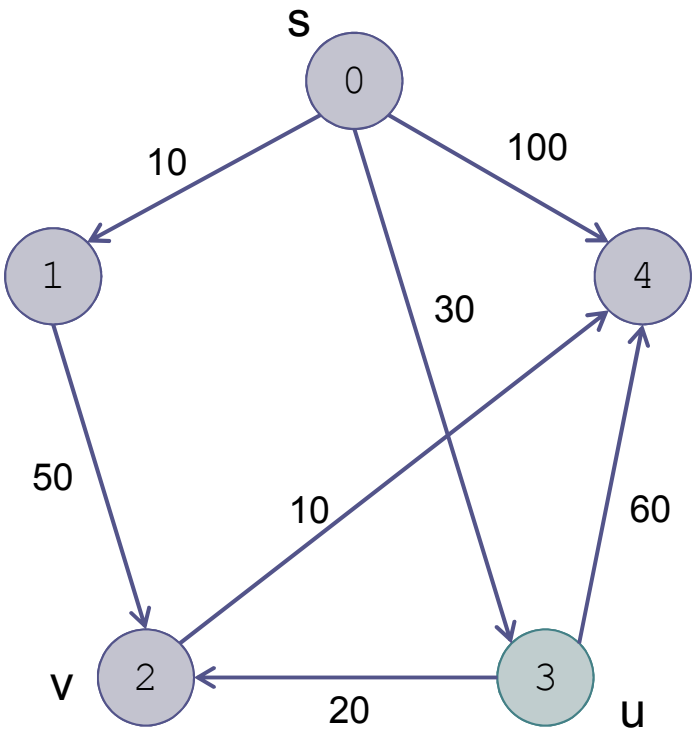
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	100	0



Continue to the next adjacent vertex

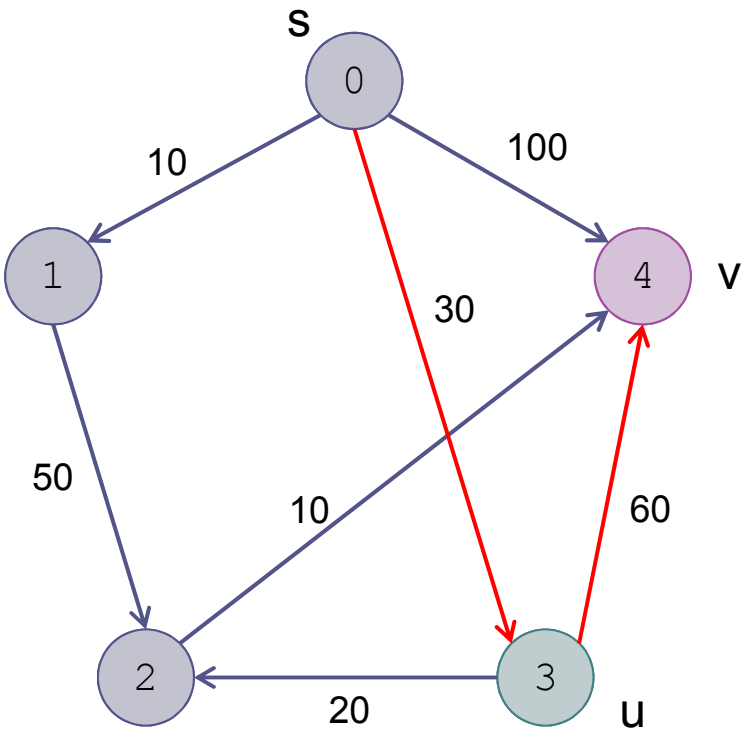
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	100	0



Continue to the next adjacent vertex

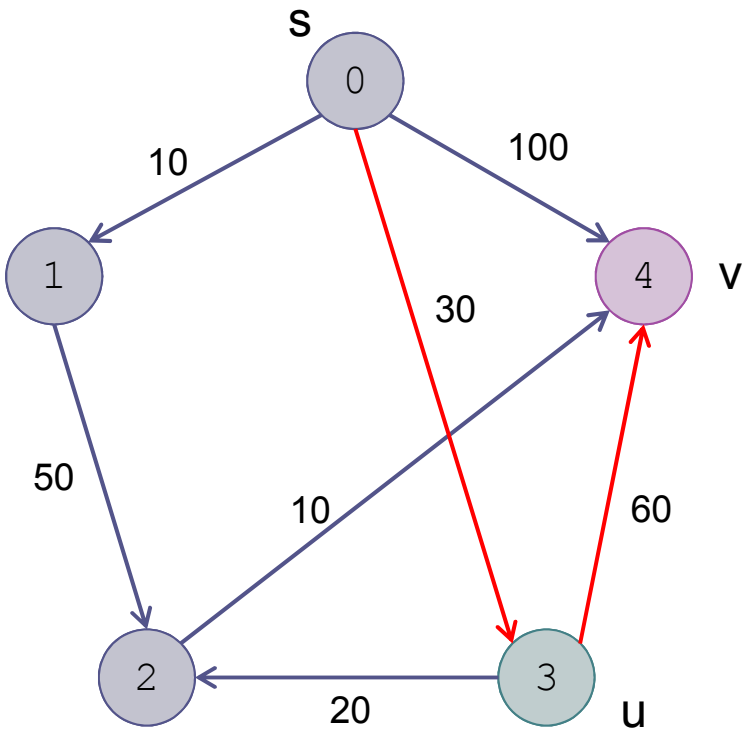
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	100	0



90 < 100

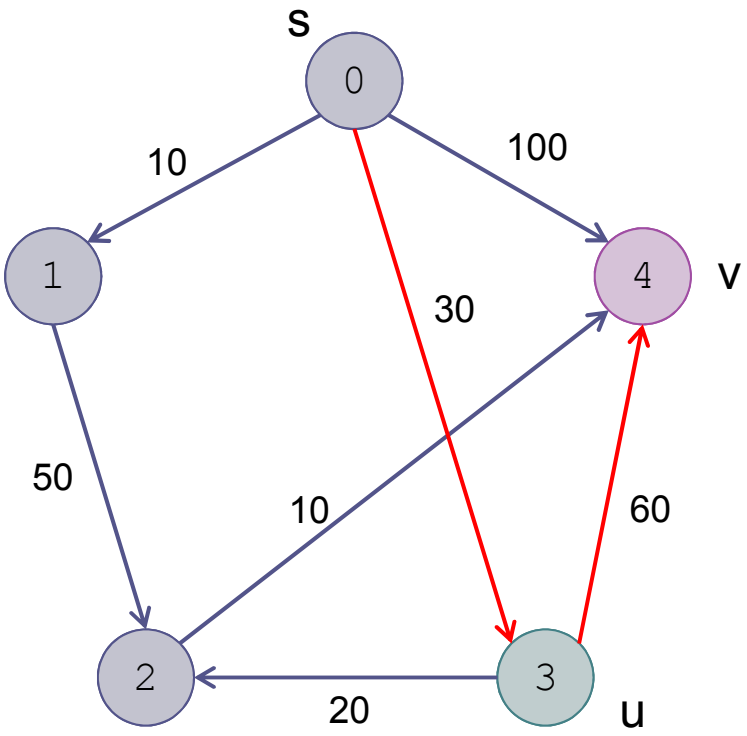
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



90 < 100

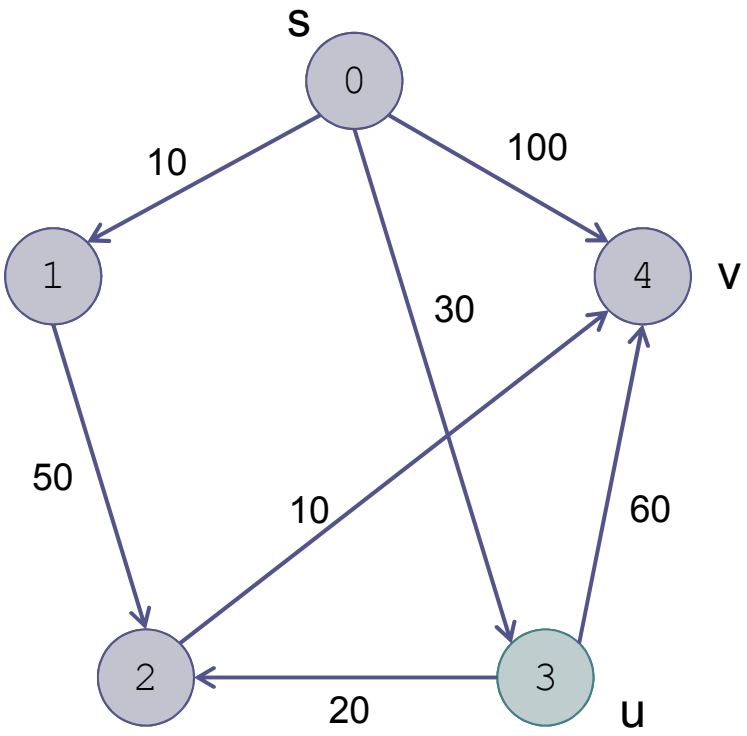
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



Move u from V-S to S

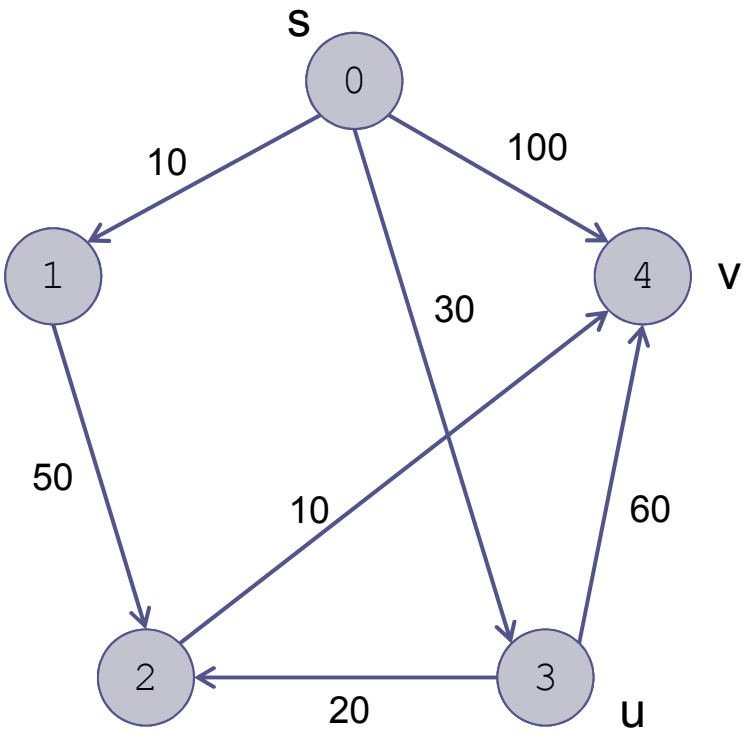
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-S = { 2, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



Move u from V-S to S

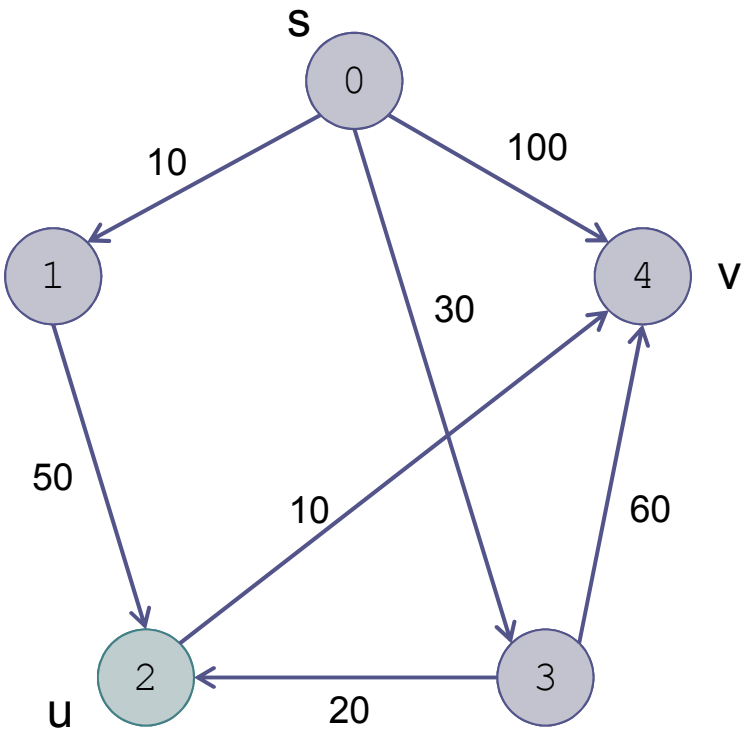
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-S = { 2, 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



Select vertex 2 from V-S

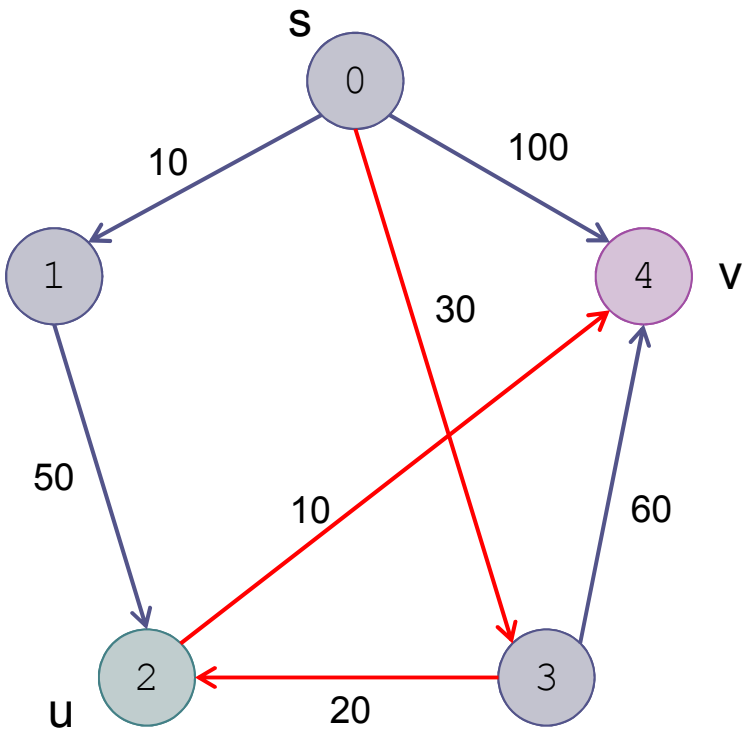
Dijkstra's Algorithm (cont.)

$S = \{ 0, 1, 3 \}$

$V\text{-}S = \{ 2, 4 \}$

$u = 2$

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



d[2] + w(2,4) = 50 + 10 = 60

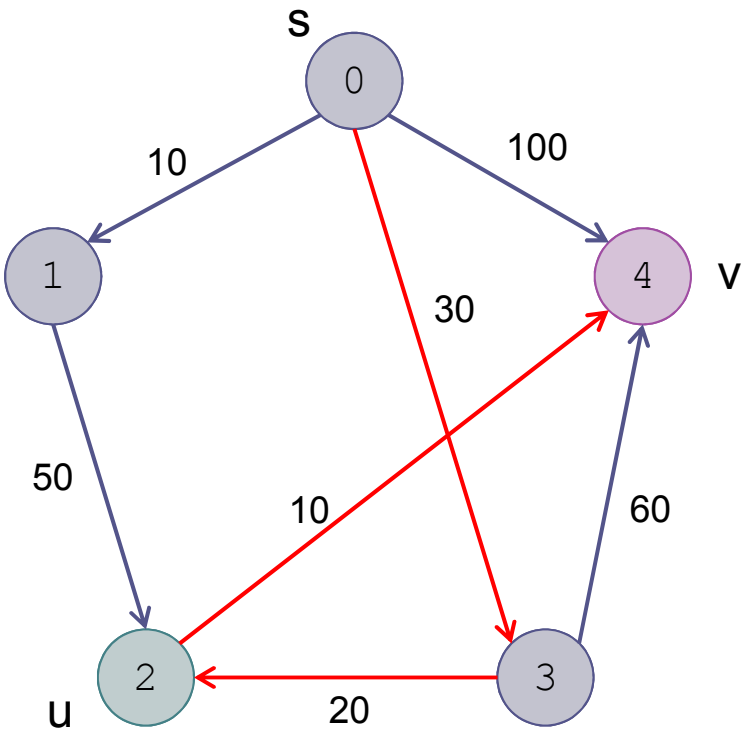
Dijkstra's Algorithm (cont.)

$S = \{ 0, 1, 3 \}$

$V\text{-}S = \{ 2, 4 \}$

$u = 2$

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	90	3



60 < 90 (d[4])

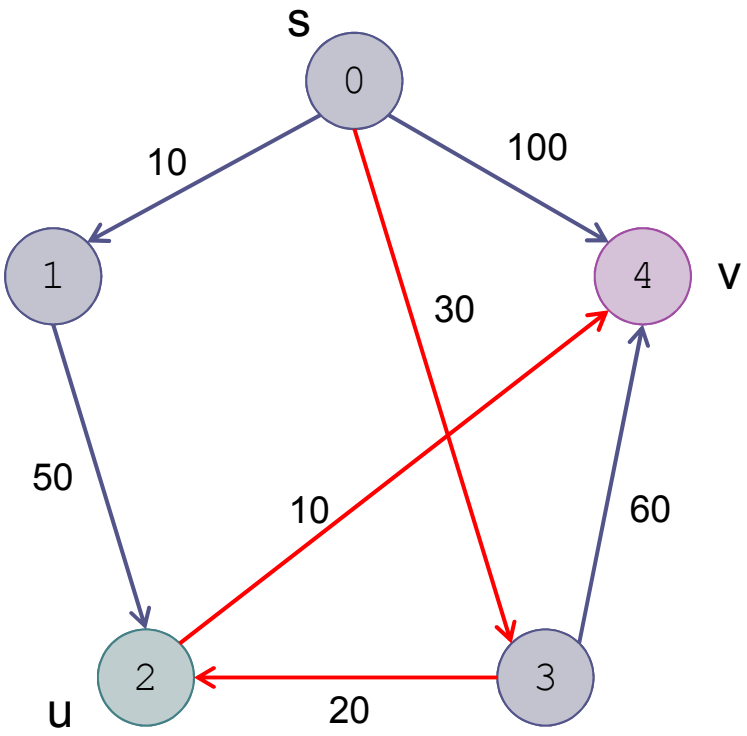
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-S = { 2, 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



update d[4] to 60 and p[4] to 2

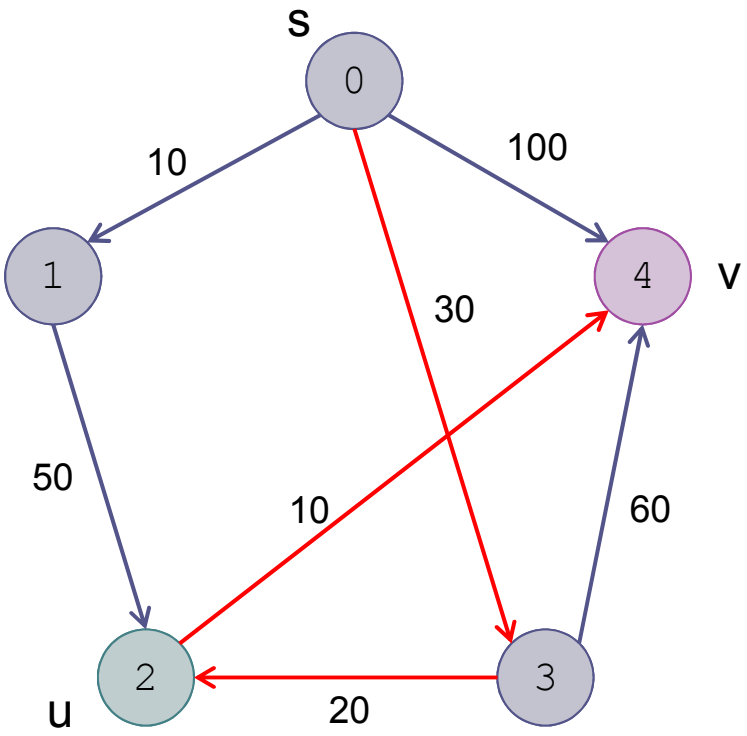
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-S = { 2, 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



update d[4] to 60 and p[4] to 2

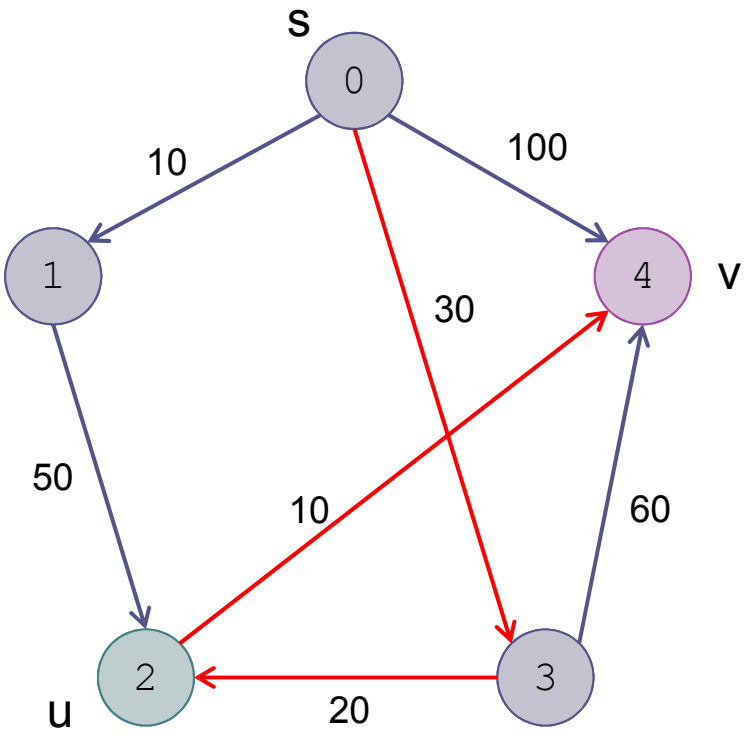
Dijkstra's Algorithm (cont.)

$S = \{ 0, 1, 3 \}$

$V\text{-}S = \{ 2, 4 \}$

$u = 2$

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



Remove 2 from *V-S*

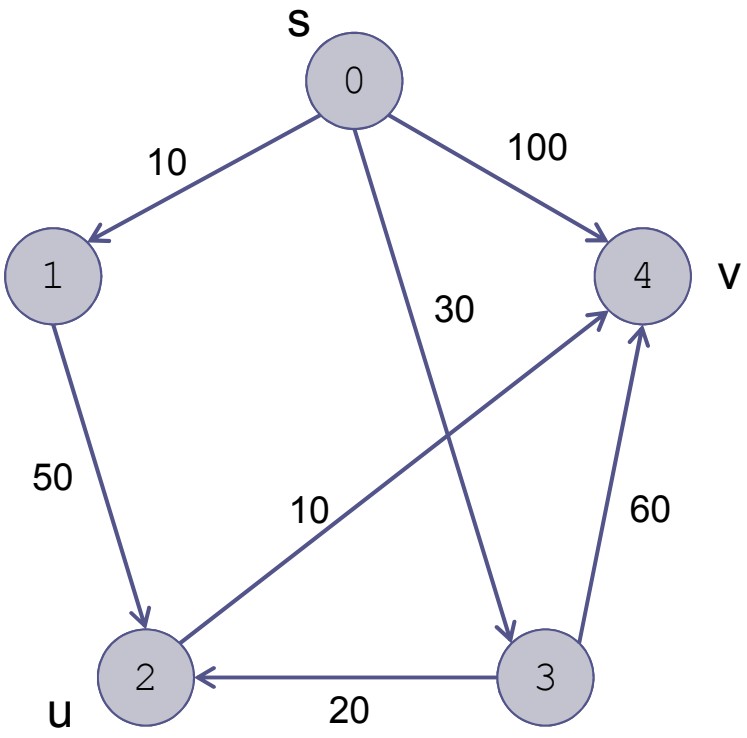
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-S = { 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



Remove 2 from V-S

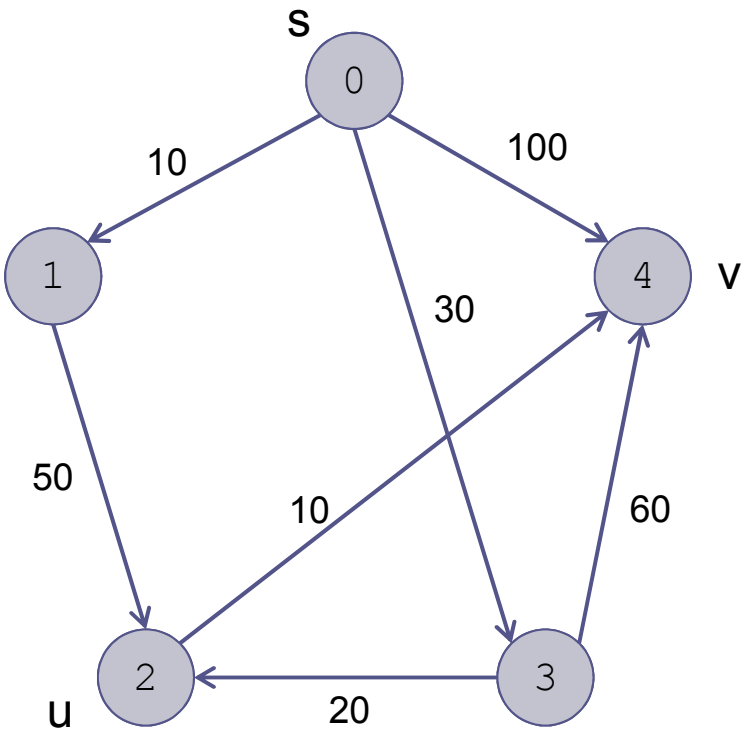
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-S = { 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



The final vertex in V-S is 4

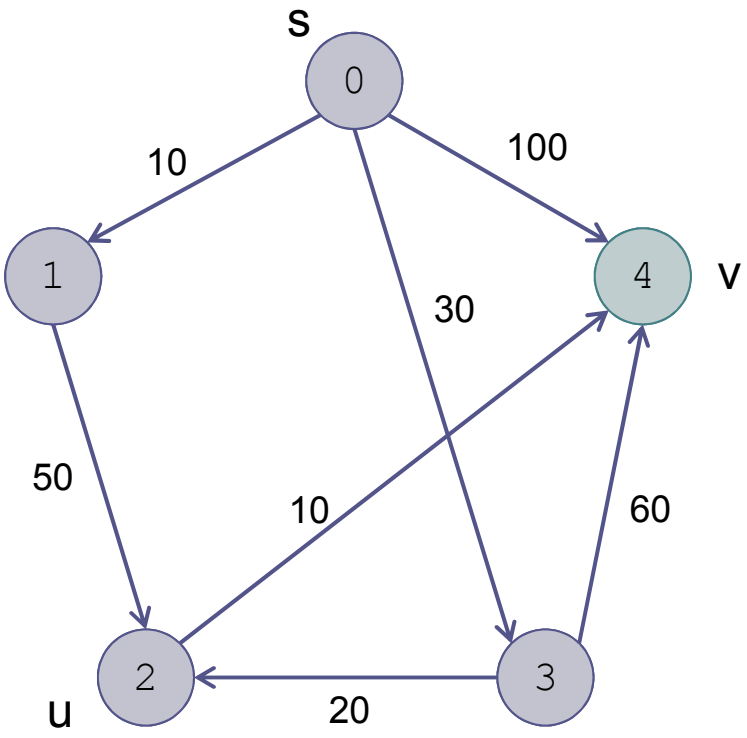
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-S = { 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



The final vertex in V-S is 4

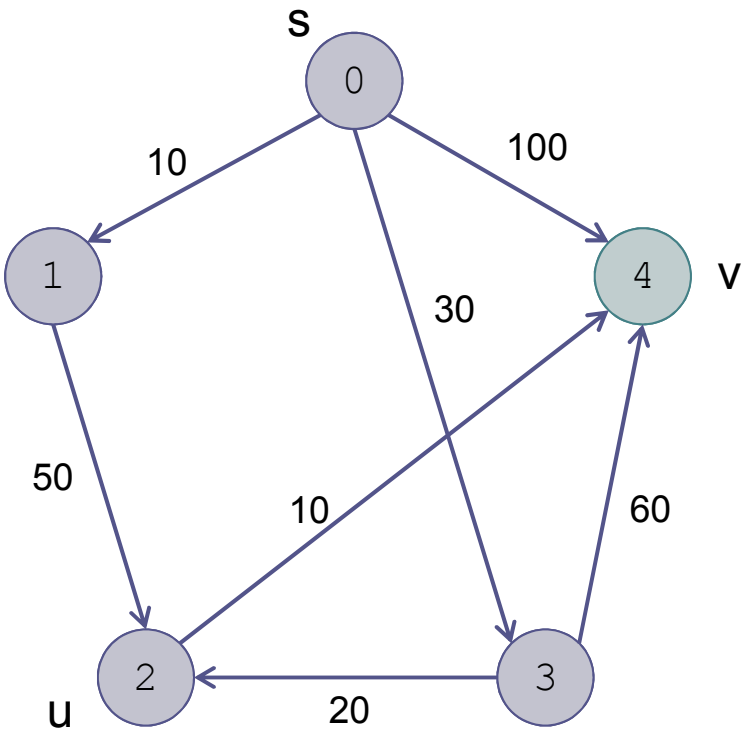
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-S = { 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



4 has no adjacent vertices; we move 4 into S

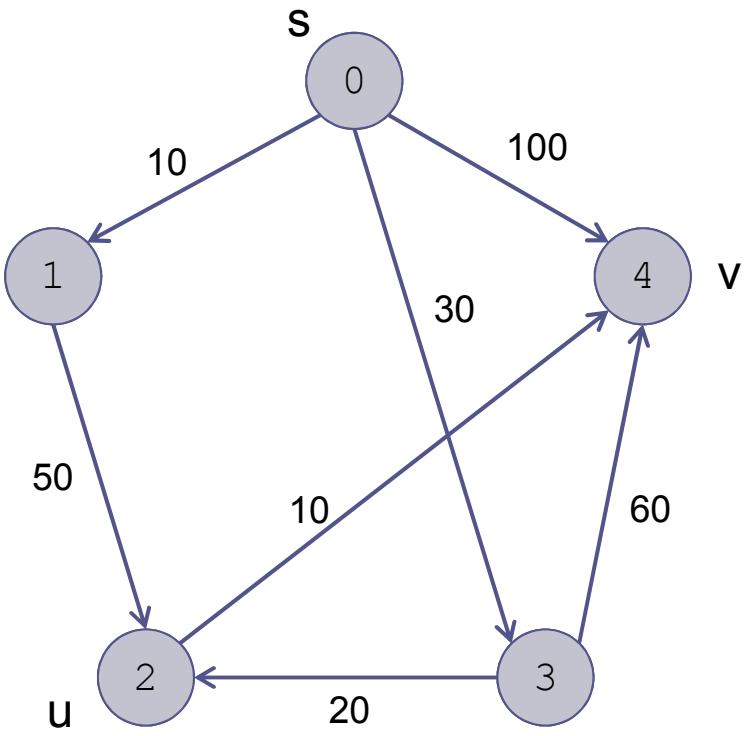
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2, 4 }

V-S = { }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



4 has no adjacent vertices; we move 4 into S

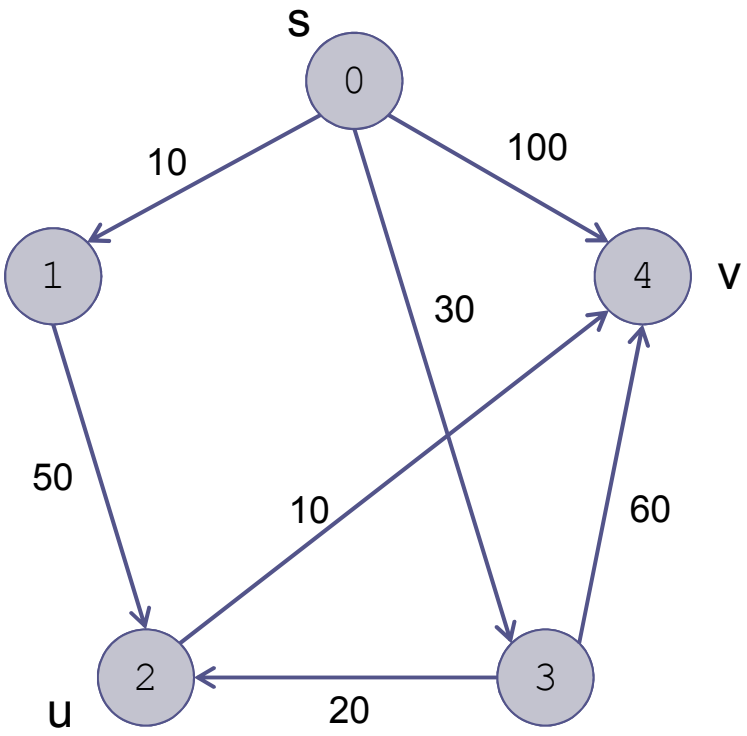
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2, 4 }

V-S = { }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



We are finished

Dijkstra's Algorithm (cont.)

Dijkstra's Algorithm

```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.  for all v in V–S
3.          Set p[v] to s.
4.          if there is an edge (s, v)
5.                  Set d[v] to w(s, v).
            else
6.                  Set d[v] to ∞.
7.  while V–S is not empty
8.          for all u in V–S, find the smallest d[u].
9.          Remove u from V–S and add u to S.
10.         for all v adjacent to u in V–S
11.                 if d[u] + w(u, v) is less than d[v].
12.                         Set d[v] to d[u] + w(u, v).
13.                         Set p[v] to u.
```

Analysis of Dijkstra's Algorithm

- Step 1 requires |V| steps
- The loop at Step 2 is executed |V|-1 times
- The loop at Step 7 also is executed |V|-1 times
- Steps 8 and 9 search the vertices in *V-S*, which decreases each time through the loop
  - $O(|V|^2)$
- Dijkstra's Algorithm is $O(|V|^2)$
- We look at possible improvements to this for sparse graphs in algorithm in the next section

Implementatic

```
public static void dijkstrasAlgorithm(Graph graph,
                                       int start,
                                       int[] pred,
                                       double[] dist) {
    int numV = graph.getNumV();
    HashSet<Integer> vMinusS = new HashSet<Integer>(numV);
    // Initialize V-S.
    for (int i = 0; i < numV; i++) {
        if (i != start) {
            vMinusS.add(i);
        }
    }
    // Initialize pred and dist.
    for (int v : vMinusS) {
        pred[v] = start;
        dist[v] = graph.getEdge(start, v).getWeight();
    }
    // Main loop
    while (vMinusS.size() != 0) {
        // Find the value u in V-S with the smallest dist[u].
        double minDist = Double.POSITIVE_INFINITY;
        int u = -1;
        for (int v : vMinusS) {
            if (dist[v] < minDist) {
                minDist = dist[v];
                u = v;
            }
        }
        // Remove u from vMinusS.
        vMinusS.remove(u);
        // Update the distances.
        for (int v : vMinusS) {
            if (graph.isEdge(u, v)) {
                double weight = graph.getEdge(u, v).getWeight();
                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pred[v] = u;
                }
            }
        }
    }
}
```

Implementation (cont.)

- For an adjacency list representation, modify the code:

```
// Update the distances.
Iterator<Edge> edgeIter = graph.edgeIterator(u);
while (edgeIter.hasNext()) {
    Edge edge = edgeIter.next();
    int v = edge.getDest();
    if (vMinusS.contains(new Integer(v));
        double weight = edge.getWeight();
        if (dist[u] + weight < dist[v]) {
            dist[v] = dist[u] + weight;
            pred[v] = u;
        }
    }
}
```

Minimum Spanning Trees

- A *spanning tree* is a subset of the edges of a graph such that there is only one edge between each vertex, and all of the vertices are connected
- The *cost of a spanning tree* is the sum of the weights of the edges
- We want to find the *minimum spanning tree* or the spanning tree with the smallest cost
- The solution formulated by R.C. Prim and is very similar to Dijkstra's algorithm

Prim's Algorithm

- The vertices are divided into two sets:
  - *S,* the set of vertices in the spanning tree
  - *V*-*S*, the remaining vertices
- As in Dijkstra's algorithm, we maintain two arrays,
  - d[*v*] contains the length of the shortest edge from a vertex in *S* to the vertex *v* that is in *V*-*S*
  - p[*v*] contains the source vertex for that edge
- The only difference between algorithms is the contents of d[*v*]; in Prim's algorithm, d[*v*] contains only the length of the final edge

Prim's Algorithm (cont.)

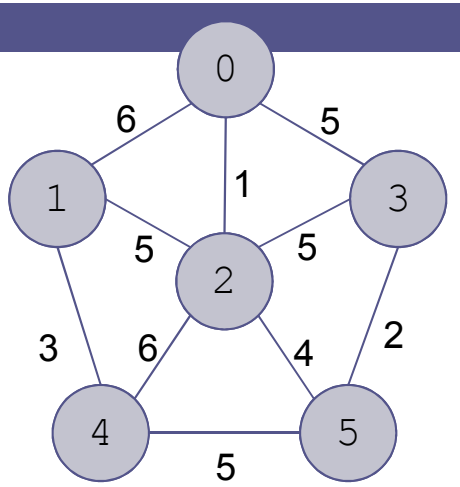
Prim's Algorithm for Finding the Minimum Spanning Tree

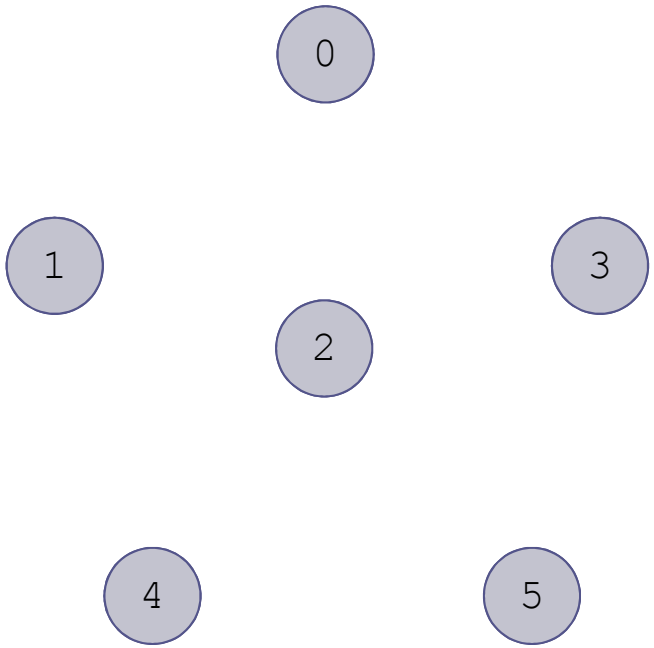
```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.  for all v in V–S
3.          Set p[v] to s.
4.          if there is an edge (s, v)
5.                  Set d[v] to w(s, v).
            else
6.                  Set d[v] to ∞.
7.  while V–S is not empty
8.          for all u in V–S, find the smallest d[u].
9.          Remove u from V–S and add it to S.
10.         Insert the edge (u, p[u]) into the spanning tree.
11.         for all v in V–S
12.                 if w(u, v) < d[v]
13.                         Set d[v] to w(u, v).
14.                         Set p[v] to u.
```

Prim's Algorithm Example

S = { 0 }

V-S = { 1, 2, 3, 4, 5 }

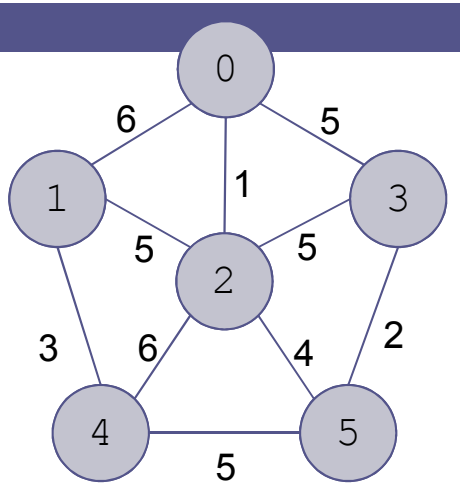




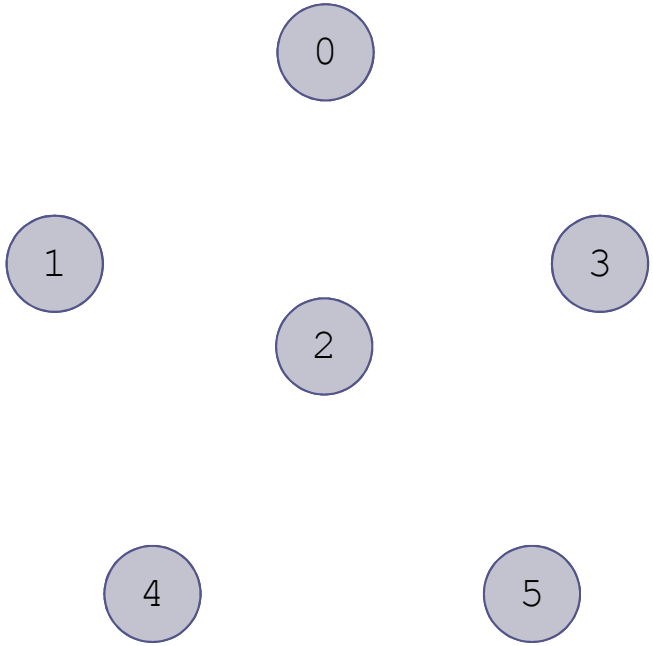
Prim's Algorithm Example (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4, 5 }



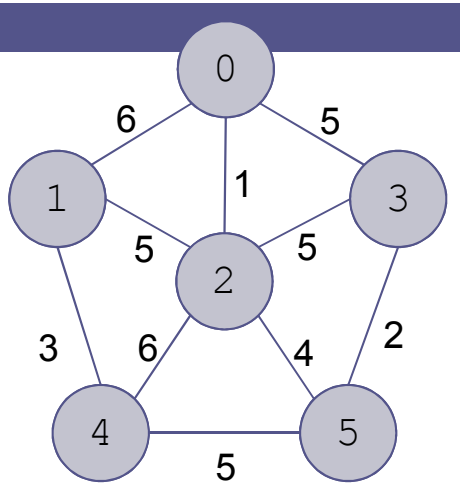
The smallest edge from u to v where u is in S and v is in V-S is the edge (0,2)



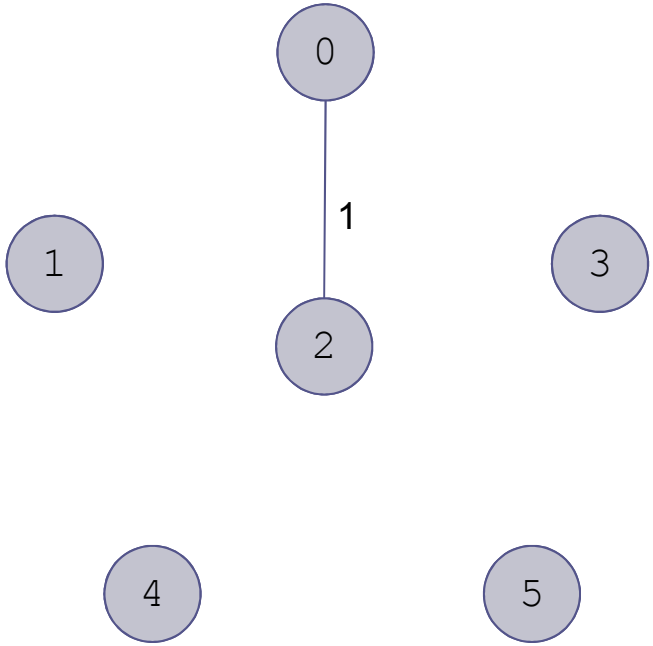
Prim's Algorithm Example (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4, 5 }



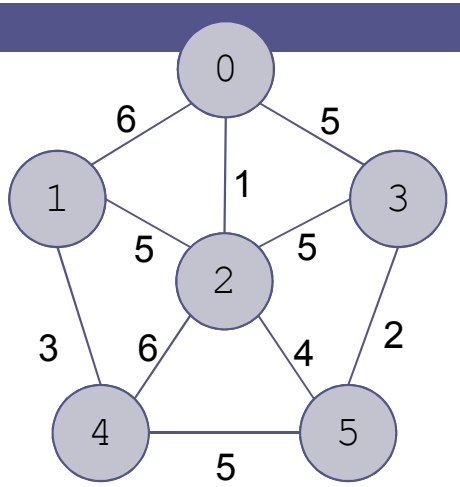
Add this edge to the spanning tree



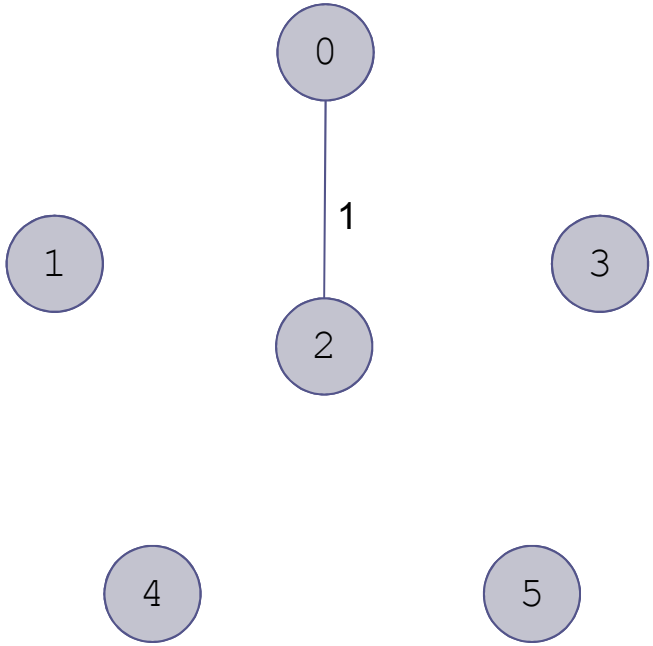
Prim's Algorithm Example (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4, 5 }



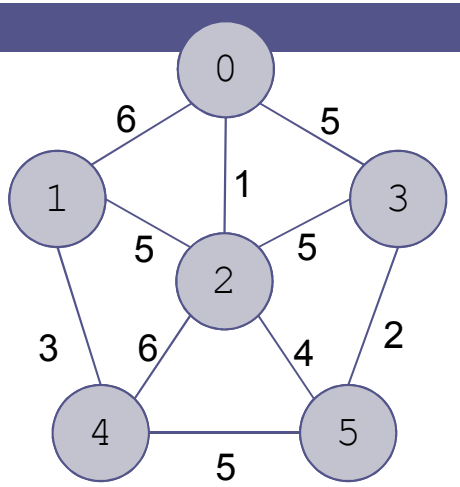
and move 2 to S



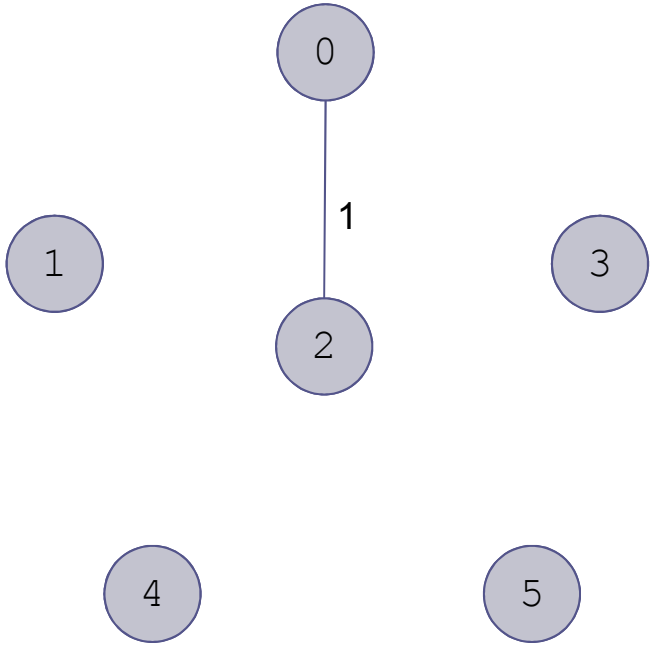
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



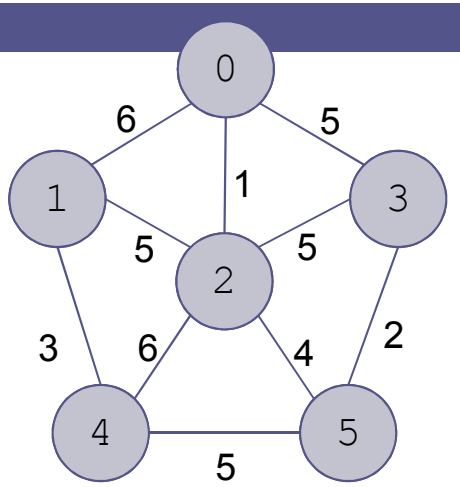
and move 2 to S



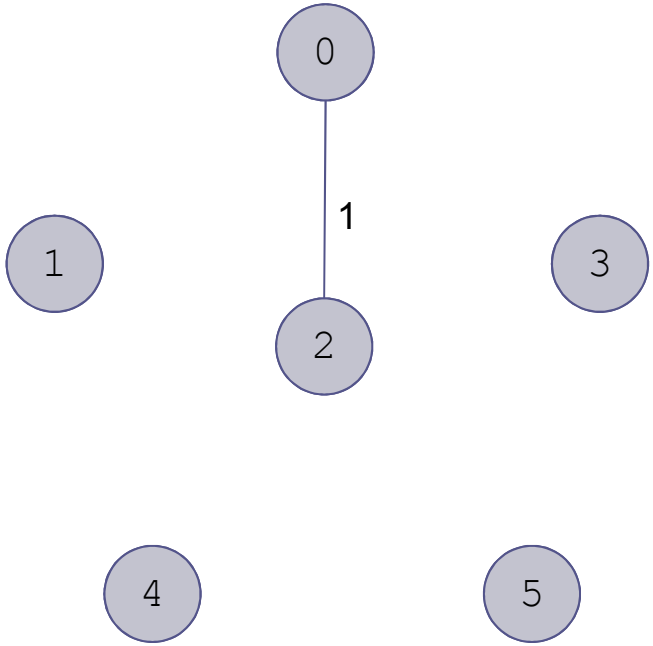
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



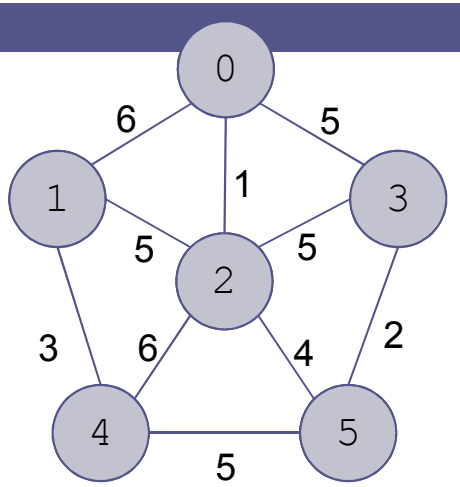
Consider all edges (u, v) where u is in S and v is in V-S (there are 8 possible edges)



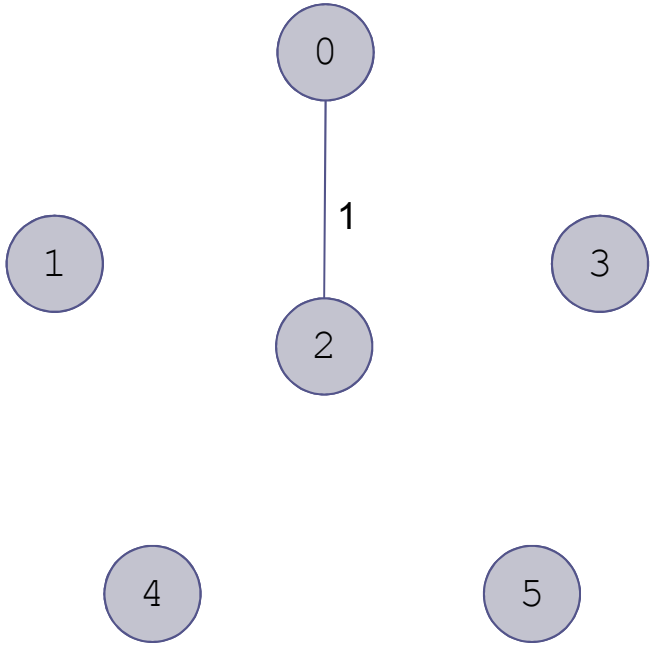
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



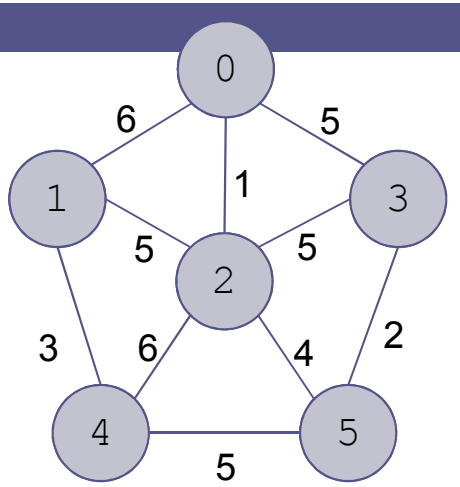
The smallest is (2, 5)



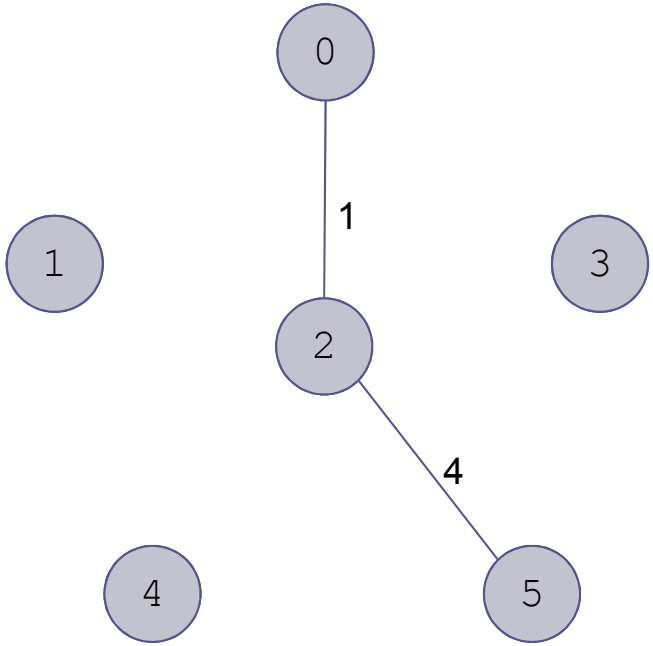
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



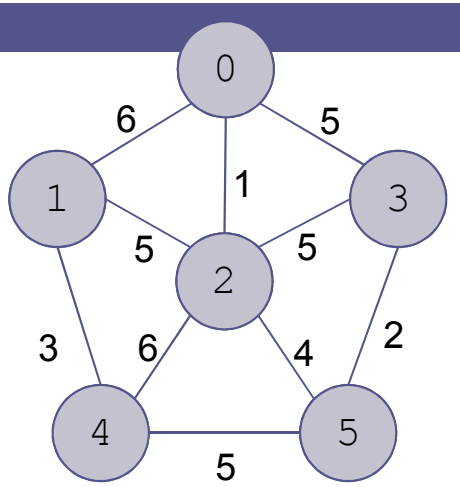
Add (2,5) to the spanning tree



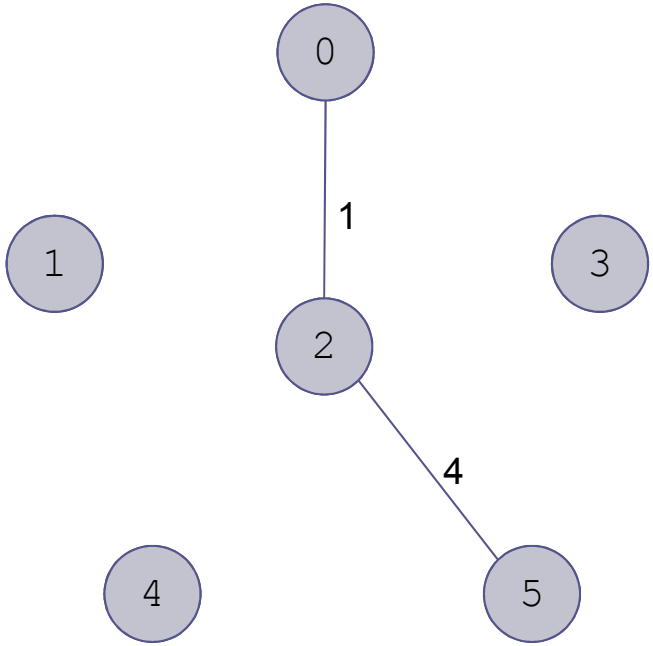
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



Move 5 from V-S to S

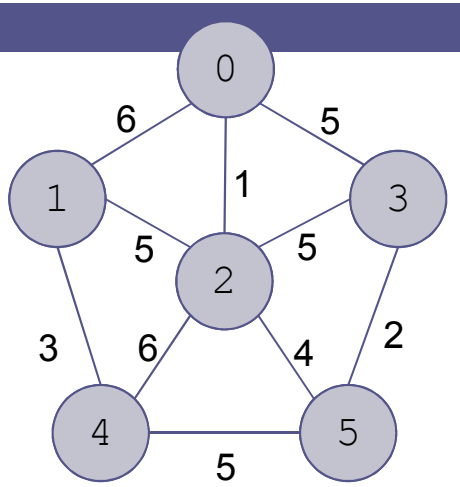


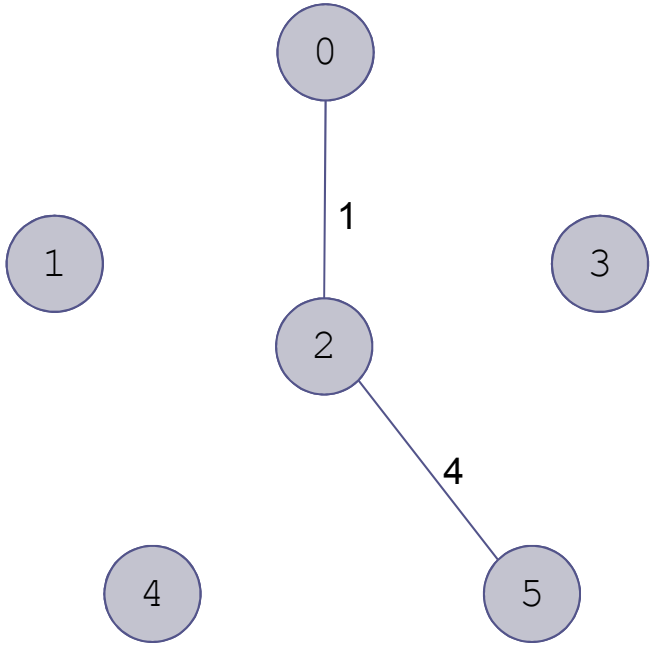
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }

Move 5 from V-S to S

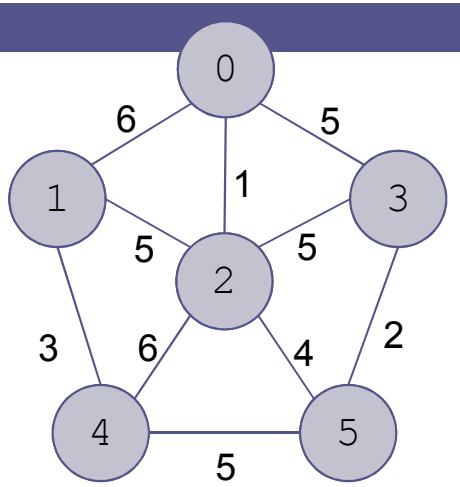




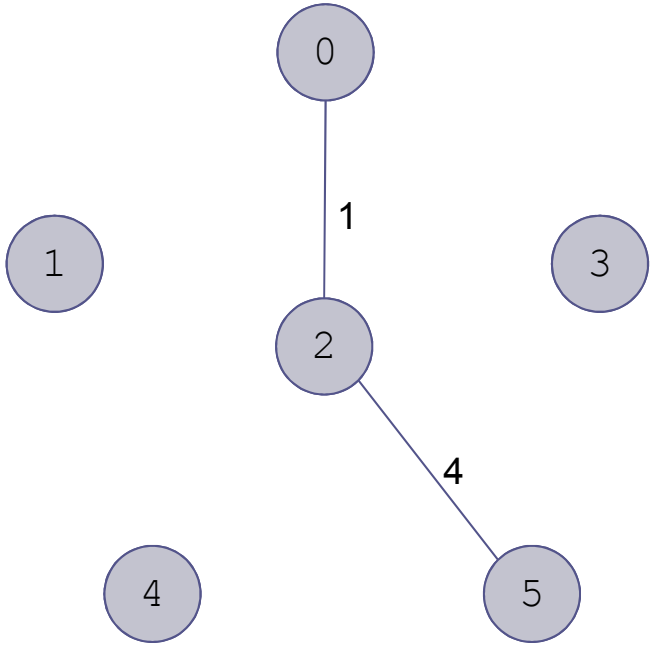
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }



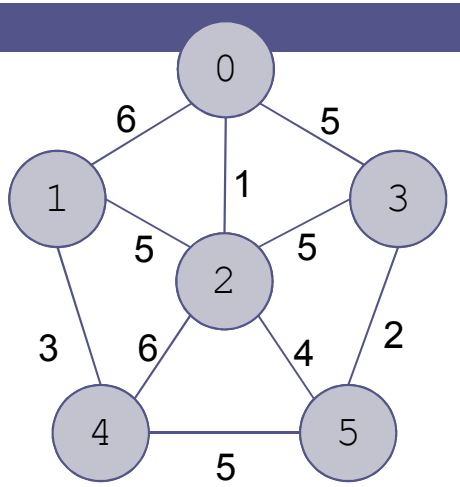
Find the smallest edge (u, v) where u is in S and v is in V-S



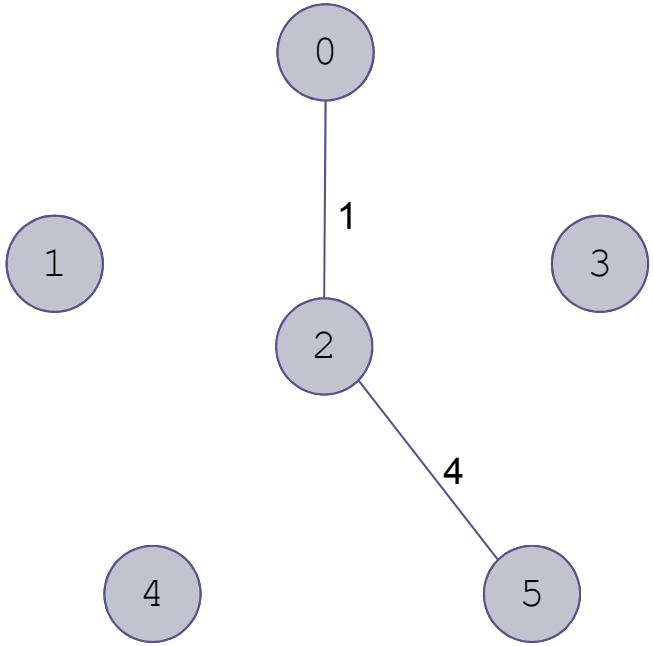
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }



The smallest edge is (5, 3)

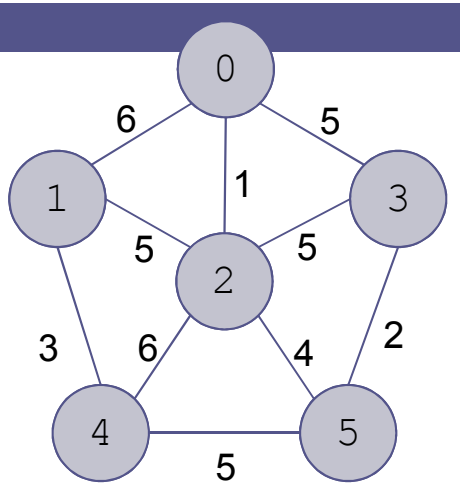


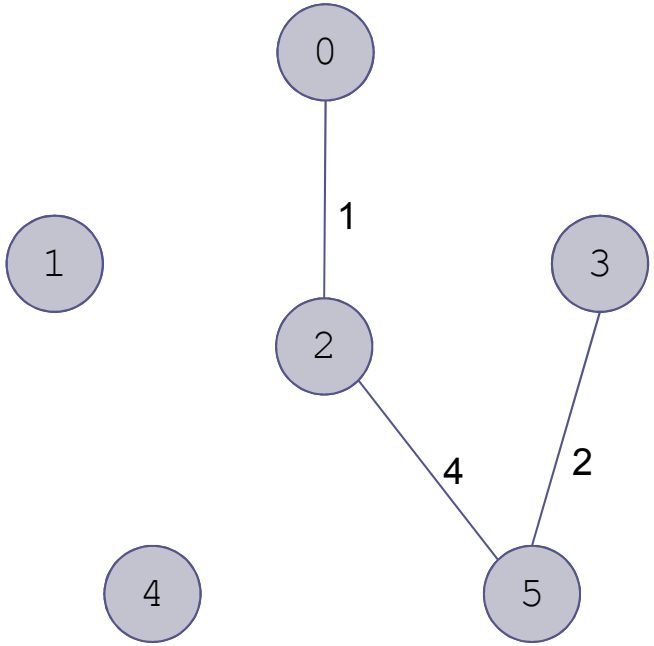
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }

The smallest edge is (5, 3)



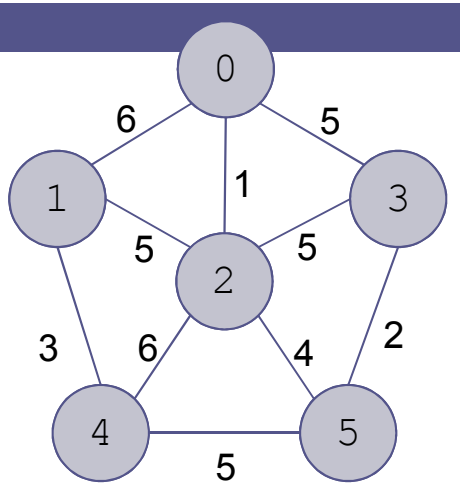


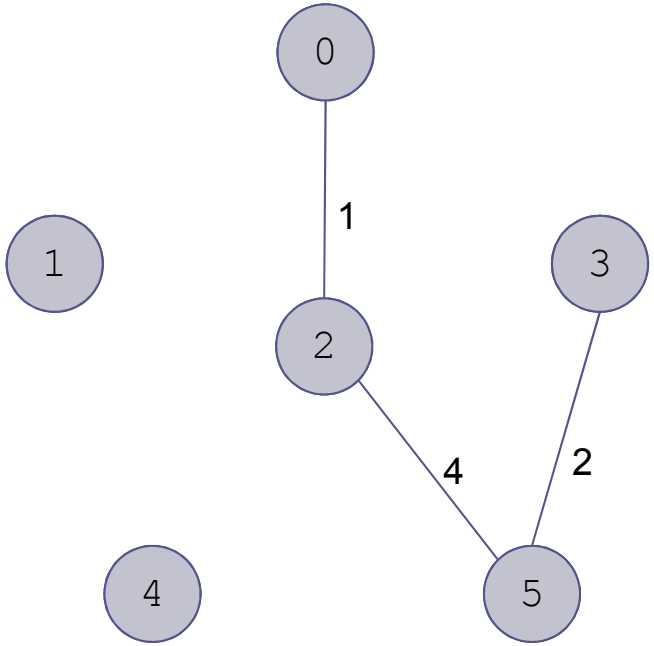
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }

Move 3 to S



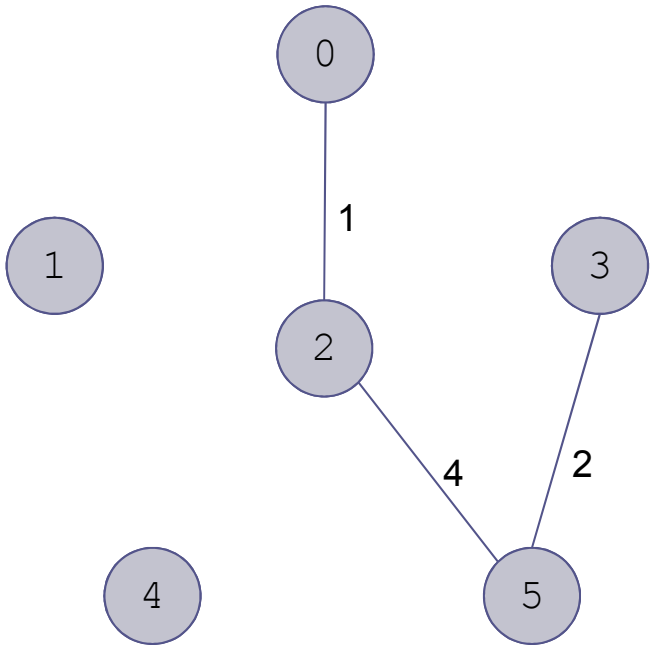


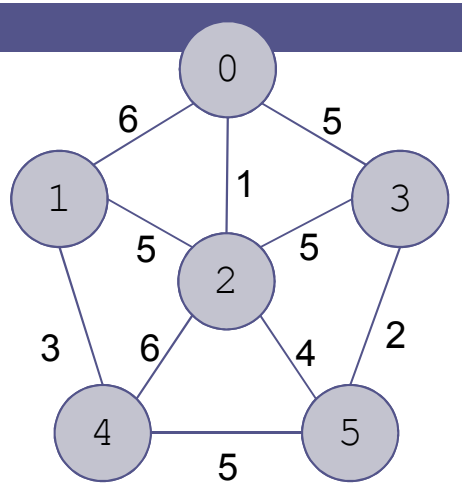
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }

Move 3 to S



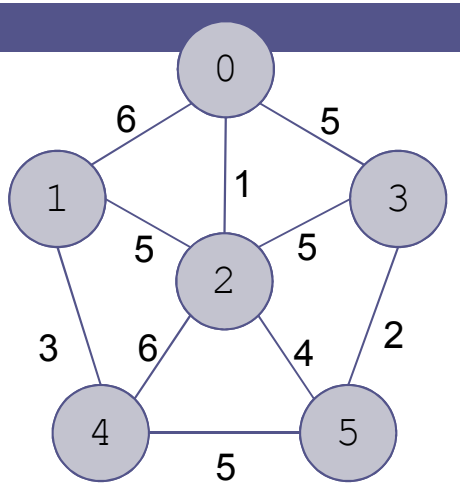


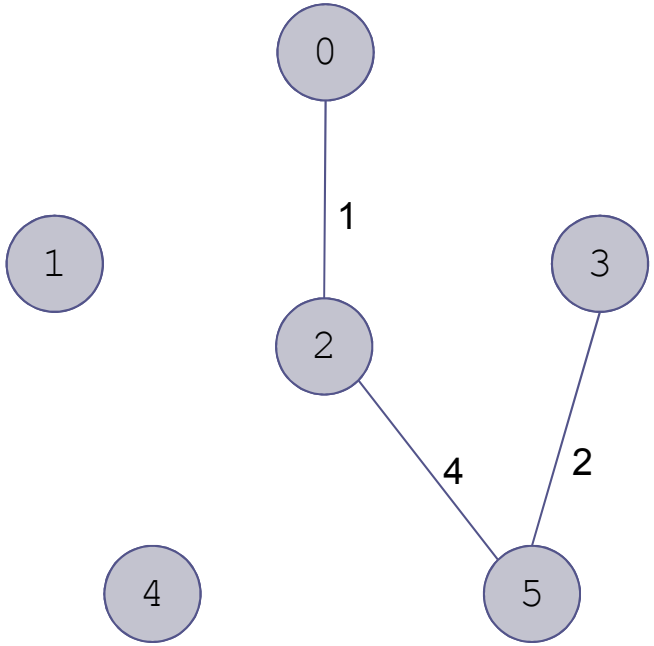
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }

The next smallest edge is (2, 1)



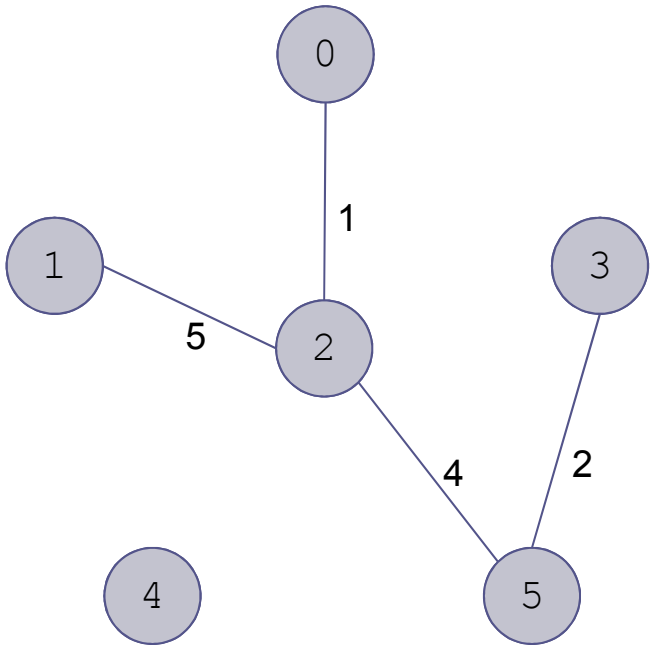


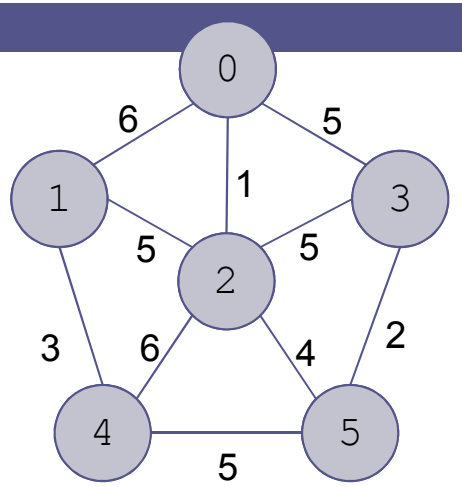
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }

The next smallest edge is (2, 1)



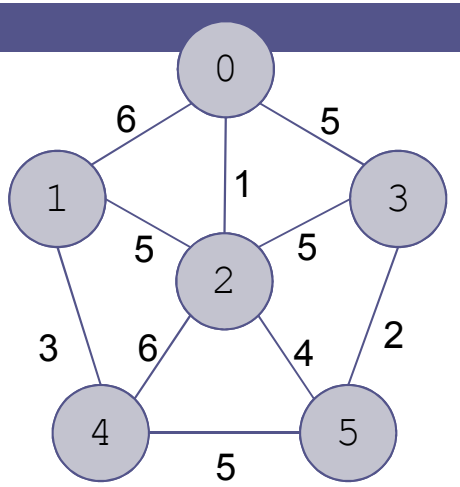


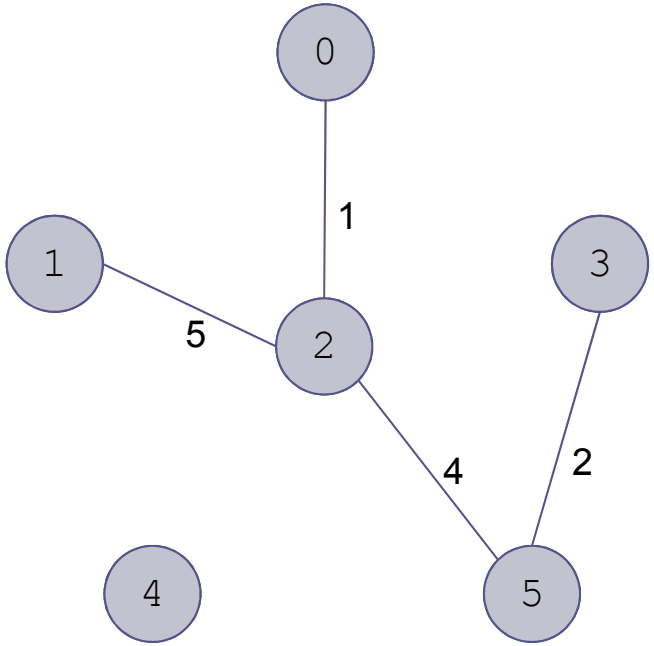
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }

Move 1 to S



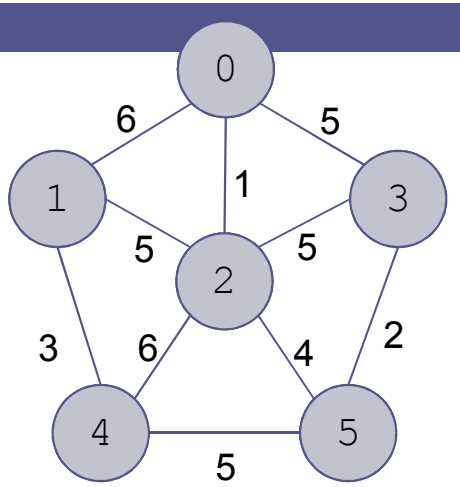


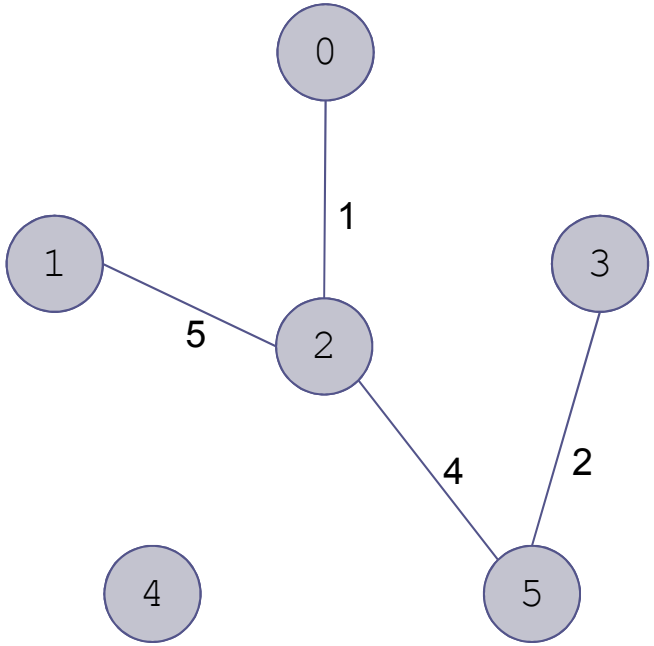
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3, 1 }

V-S = { 4 }

Move 1 to S

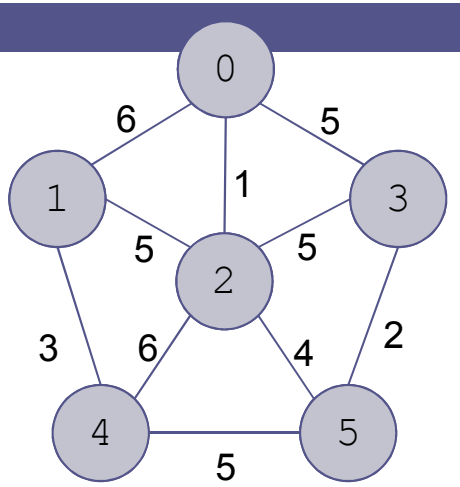




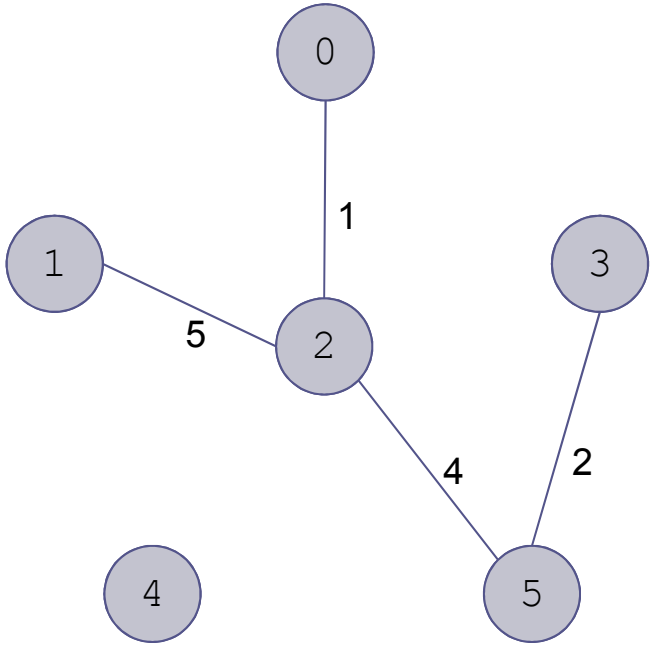
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3, 1 }

V-S = { 4 }



The smallest edge to 4 is (1, 4)

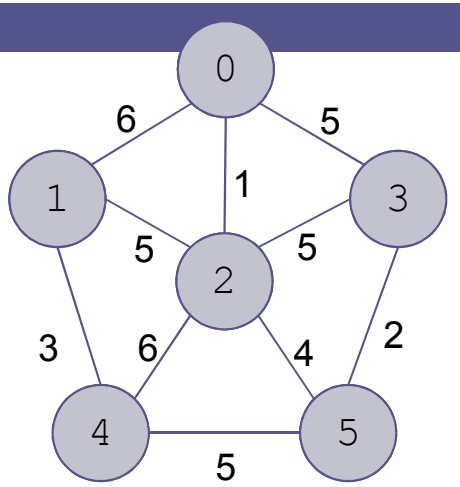


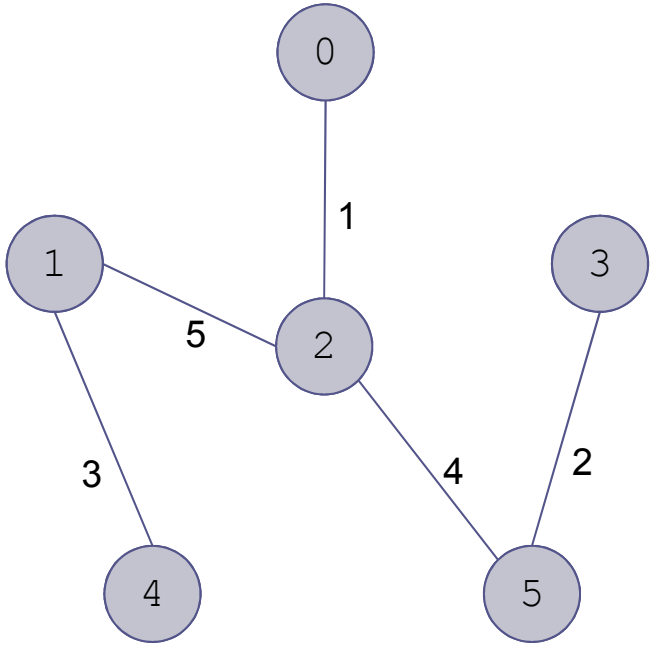
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3, 1 }

V-S = { 4 }

The smallest edge to 4 is (1, 4)

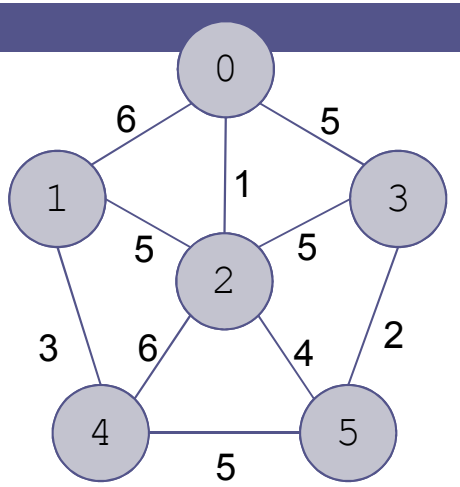




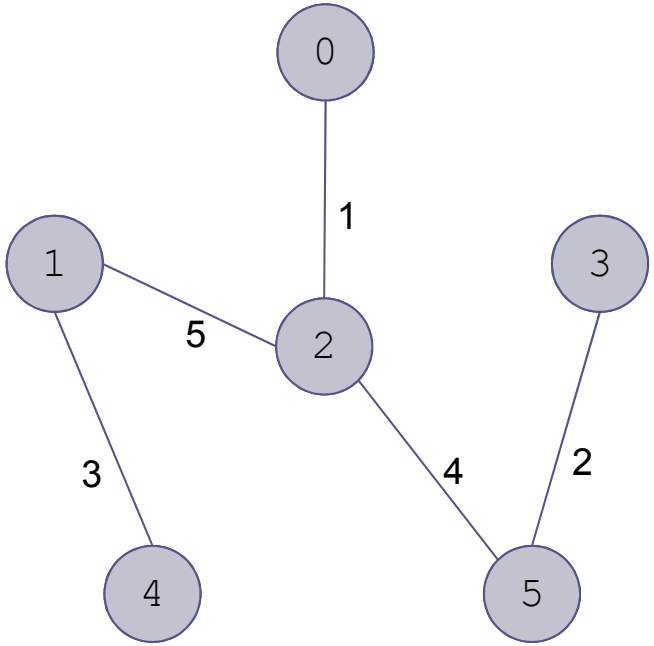
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3, 1, 4 }

V-S = { }



The spanning tree is complete



Analysis of Prim's Algorithm

- Step 8
  - is O(|V|)
  - and is within a loop at Step 7, so it is executed O(|V|) times
  - for a total time of $O(|V|^2)$
- Step 11
  - is $O(|E_u|)$,
  - Is executed for all vertices for a total of O(|E|)
- The overall cost is $O(|V|^2)$ ($|V|^2$ is greater than |E|)
- Using a priority queue can reduce Step 8 to O(log *n*) where *n* is the size of the priority queue
  - The algorithm becomes O(|E| log |V|)
- For a dense graph (where|E| is approximately $|V|^2$), this is not an improvement; for a sparse graph, it is an improvement
- Researchers have developed an improved priority queue implementation that gives O(|E| + |V| log |V|) or better performance

Implementatio

```
public static ArrayList<Edge> primsAlgorithm(Graph graph,
                                             int start) {
    ArrayList<Edge> result = new ArrayList<Edge>();
    int numV = graph.getNumV();
    // Use a HashSet to represent V-S.
    Set<Integer> vMinusS = new HashSet<Integer>(numV);
    // Declare the priority queue.
    Queue<Edge> pQ =
        new PriorityQueue<Edge>(numV, new CompareEdges());
    // Initialize V-S.
    for (int i = 0; i < numV; i++) {
        if (i != start) {
            vMinusS.add(i);
        }
    }
    int current = start;
    // Main loop
    while (vMinusS.size() != 0) {
        // Update priority queue.
        Iterator<Edge> iter = graph.edgeIterator(current);
        while (iter.hasNext()) {
            Edge edge = iter.next();
            int dest = edge.getDest();
            if (vMinusS.contains(dest)) {
                pQ.add(edge);
            }
        }
        // Find the shortest edge whose source is in S and
        // destination is in V-S.
        int dest = -1;
        Edge edge = null;
        do {
            edge = pQ.remove();
            dest = edge.getDest();
        } while(!vMinusS.contains(dest));
        // Take dest out of vMinusS.
        vMinusS.remove(dest);
        // Add edge to result.
        result.add(edge);
        // Make this the current vertex.
        current = dest;
    }
    return result;
}
```