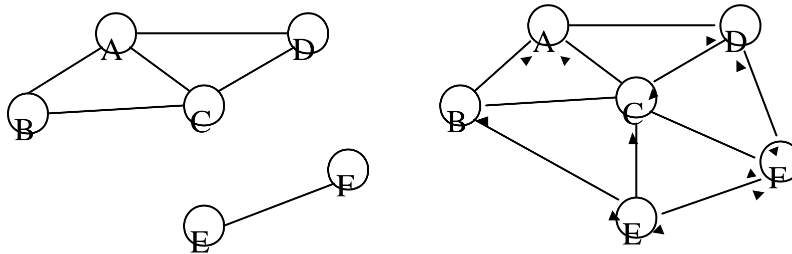
# Graphs

- A *graph* consists
  - Of a collection of points sometimes called vertices
  - Some pairs of points are connected by a line segment sometimes called an edge
- Edges
  - May have a direction associated with them in which case the graph is called a *directed graph* (*digraph*)
  - A graph that contains edges that do not have a direction associated with them is called an *undirected graph*



# Graphs




Graphs are usually drawn using points for vertices and lines for Edges, but a graph is defined independent of its representation

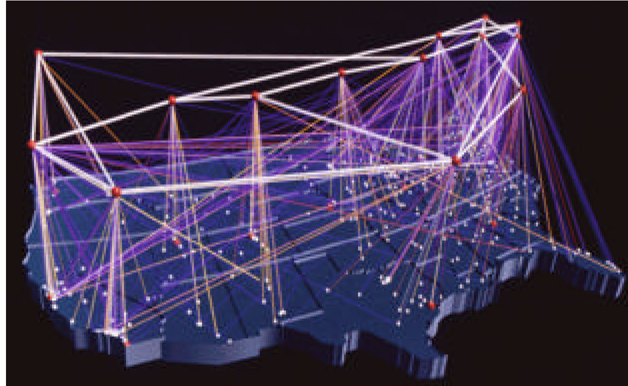
# Weights

- In addition to being directed or undirected, edges can be weighted or un-weighted.
  - A weighted edge has a value associated with it
  - The weight often measures the cost of using the edge to go from one node to another
- A vertex may also have data associated with it.



# What Are They Good For?

## What Are They Good For?





## Terminology

- Two different vertices, *x* and *y*, in a graph are said to be *adjacent* if an edge connects *x* to *y*
- A *path* is a sequence of vertices in which each vertex is adjacent to the next one
  - The *length* of a path is the number of edges in the path
  - A *simple path* is a path in which no vertex is repeated
  - A *cycle* is a path of length greater than one that begins and ends at the same vertex
    - A graph with no cycles is called a *tree*

## Terminology

- The *degree* of a vertex $x$ is the number of edges $e$ in which $x$ is one of the endpoints of edge $e$
- The *neighbors* of a vertex $v$, are the vertices that are directly connected to $v$



## Formal Definition

- A graph $G = ( V, E )$, consists of a set of vertices, $V$, along with a set of edges, $E$, where the edges in $E$ are formed from pairs of distinct vertices in $V$.
- In an undirected graph, each edge $e = \{ v_1, v_2 \}$ is an unordered pair of distinct vertices, which connects the two vertices $v_1$ and $v_2$, without prescribing a direction from $v_1$ to $v_2$ or from $v_2$ to $v_1$.
- In a directed graph, each edge $e = \{ v_1, v_2 \}$ is an ordered pair of vertices, which connects the pair of vertices $v_1$ and $v_2$, in the direction from $v_1$ to $v_2$. In this case we say $v_1$ is the origin of the edge $e = \{ v_1, v_2 \}$ and $v_2$ is the end of the edge $e$.

# Connectivity

- Two vertices in a graph $G$ are said to be *connected* if there is a path from the first to the second in $G$
  - If $x \in V$ and $y \in V$, where $x \neq y$, then $x$ and $y$ are *connected* if there exists a path, $p = v_1, v_2, \ldots, v_n$, in $G$, such that $x = v_1$ and $y = v_n$
- In the graph $G$, a *connected component* is a subset, $S$, of the vertices $V$ that are all connected to one another
  - $S$ is a *connected component* of $G$ if, for any two distinct vertices, $x \in S$ and $y \in S$, $x$ is connected to $y$
- A graph is *connected* if there is a path from every node to every other node in the graph
  - A graph that is not connected is made up of connected components



# Connectivity

- A digraph is said to be strongly connected if for every pair of nodes, $n_i$ and $n_j$, there exists a path from node $n_i$ to node $n_j$.
- An undirected graph is said to be connected if for every pair of nodes, $n_i$ and $n_j$, there is a path connecting the two nodes

# Edges

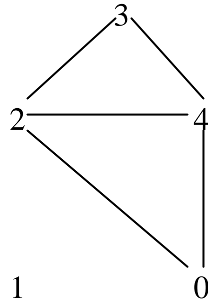
- If we denote the number of vertices in a graph by $V$ and the number of edges by $E$, note that $E$ can range anywhere from 0 to $\frac{1}{2}V(V-1)$
  - Graphs with all edges present are called *complete* graphs
  - Graphs with relatively few edges (say less than $V \log V$) are called *sparse* graphs
  - Graphs with relatively few of the possible edges missing are called *dense*



# DiGraph.java

```
import java.util.*;

public interface DiGraph {

    // Methods to build the graph

    public void addVertex( Object key, Object data );
    public void addEdge( Object fromKey, Object toKey, Object data ) throws NoSuchVertexException;

    // Operations on edges

    public boolean isEdge( Object fromKey, Object toKey ) throws NoSuchVertexException;
    public Object getEdgeData( Object fromKey, Object toKey ) throws NoSuchVertexException;

    // Operations on vertices

    public boolean isVertex( Object key );
    public Object getVertexData( Object key ) throws NoSuchVertexException;
    public int numVertices();
    public int degree( Object key );
    public Collection neighborData( Object key ) throws NoSuchVertexException;
    public Collection neighborKeys( Object key ) throws NoSuchVertexException;

    // Utility

    public Collection vertexData();
    public Collection vertexKeys();
    public Collection edgeData();
    public void clear();

} // DiGraph
```

## Adjacency Matrix

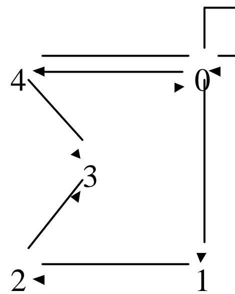



| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | F | F | T | F | T |
| **1** | F | F | F | F | F |
| **2** | T | F | F | T | T |
| **3** | F | F | T | F | T |
| **4** | T | F | T | T | F |

The adjacency matrix representation is satisfactory only if the graphs to be represented are dense (most of the array will be fa lse)

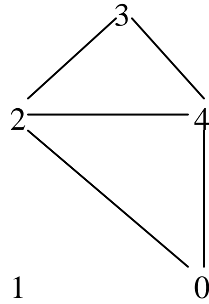


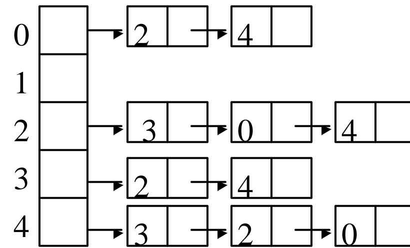
## Adjacency Matrix




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | T | T | F | F | T |
| **1** | F | F | T | F | F |
| **2** | F | F | F | T | F |
| **3** | F | F | F | F | F |
| **4** | T | F | F | T | F |

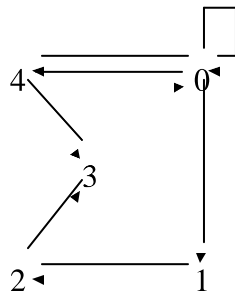
# Adjacency List

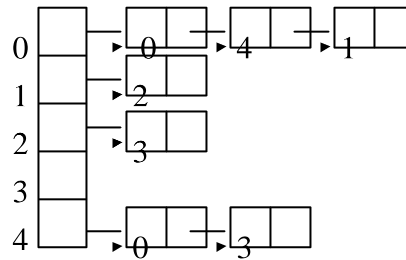





# Adjacency List



## Depth-First Search

- Several questions arise when processing a graph
  - Is the graph connected?
  - If not, what are the connected components?
  - Does the graph have a cycle?
  - …
- These problems can be solved using a technique called a *depth-first search* (DFS)
  - A way in which to *visit* every node in a systematic fashion



## DFS Pseudo-code

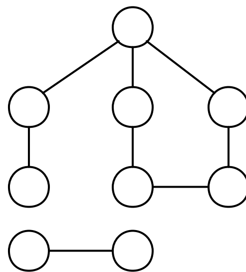
- Associate with each vertex an `Integer` value
  - A value of zero indicates that the vertex has not been visited
  - A non-zero value indicates that the node has been visited
- The *pseudo-code*
  - Build the graph and initialize the values associated with each vertex to zero
  - Get a collection that contains the keys of all the vertices in the graph
  - Set an integer variable, named component, to 1
  - Iterate over the keys
    - Get the data associated with the vertex identified by the current key
    - If the *visit* value is 0 → visit( vertex, component)
    - Increment component by 1

# DFS Visit

- Visit( vertex *v*, Integer *component* )
  - Change the visit value associated with *v* to *component*
  - Get a collection that contains the keys of the neighbors of *v*
  - Iterate over the collection
    - Get the data associated with the current key
    - If the vertex has not been visited → visit( *vertex*, *component*)



# DFS



## DFS Visit – Non Recursive

- Visit( Stack *s*, vertex *v*, Integer *component* )
  - Push *v* onto the stack *s*
  - While the *s* is not empty
    - Pop the stack and make the vertex the current vertex
    - Change the visit value associated with the current vertex to *component*
    - Get a collection that contains the keys of the neighbors of the current vertex
    - Iterate over the collection
      - If the current key has not been visited and the visit value != -1
        - » Change the current visit value to –1
        - » Push onto the stack



## Breadth-First Search (BFS)

- What if we changed the stack in the non-recursive DFS-visit to a queue?
- Visit( Queue *q*, vertex *v*, Integer *component* )
  - Enqueue *v* into the queue *q*
  - While the *q* is not empty
    - Dequeue and make the vertex the current vertex
    - Change the visit value associated with the current vertex to *component*
    - Get a collection that contains the keys of the neighbors of the current vertex
    - Iterate over the collection
      - If the current key has not been visited and the visit value != -1
        - » Change the current visit value to –1
        - » Enqueue the current key

## Dijkstra's Shortest Path Algorithm

- Each node is labeled with its distance from the source node along the best path
- Initially no paths are known, so the values are infinity
- The algorithm starts at the source node and explores possible paths, one hop at a time
- When a label is marked permanent, its label will not change



## The Algorithm

- Make the source node permanent; the source node is the first working node
- Examine each non-permanent node adjacent to the working node
  - if it is not labeled, label with the distance from the source and the name of the working node
  - if it is labeled, see if the cost computed using the working node is better than the cost in the label; if so change the label to reflect the better path

# The Algorithm

- Find the non-permanent node with the smallest label, and make it permanent
- If all the nodes are marked permanent, the algorithm terminates
- Otherwise, the node just made permanent becomes the working node
- When the algorithm is complete, the path is found (in reverse) by reading the labels from the destination node back to the source



# Example