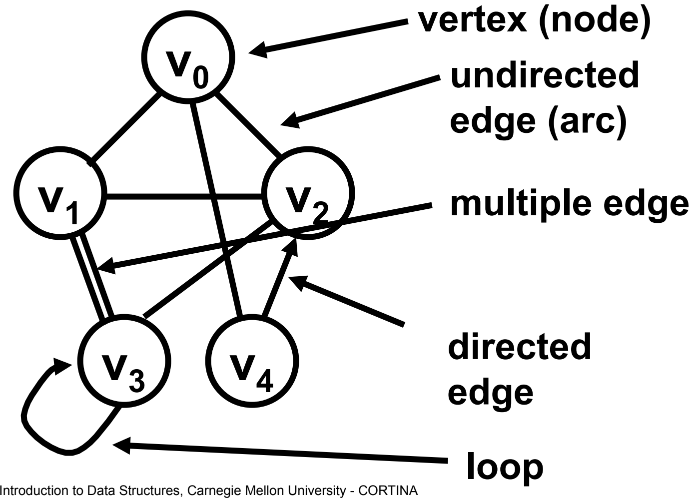
Graphs 9

An Introduction to Graphs

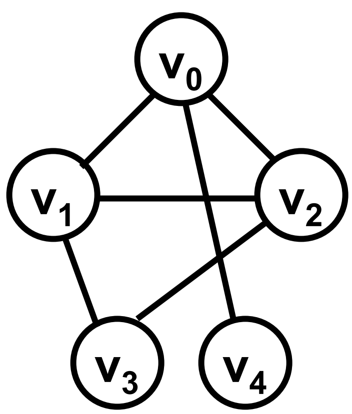
Fundamentals

- A graph G = (V,E) is a set of vertices V and a collection of edges E.
- In an undirected graph, an edge E = (x,y) is said to connect vertex x to vertex y (and vice-versa). Thus, the edges (x,y) and (y,x) are the same edge.
- In a directed graph, an edge E = (x,y) is said to connect vertex x to vertex y (but not vice-versa). Thus, (x,y) and (y,x) are not the same edges.
- A simple graph has no multiple edges between vertices or loops from a vertex to itself.

Graph Terminology



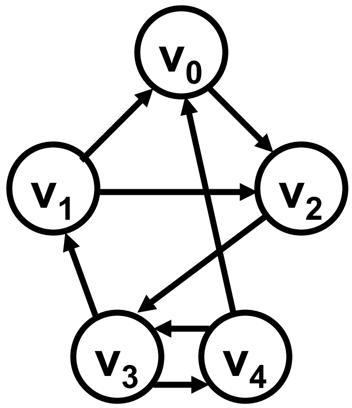
Graph Terminology



$V = \{v_0, v_1, v_2, v_3, v_4\}$

$E = \{ \{v_0, v_1\}, \{v_0, v_2\}, \{v_0, v_4\}, \{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\} \}$

Graph Terminology



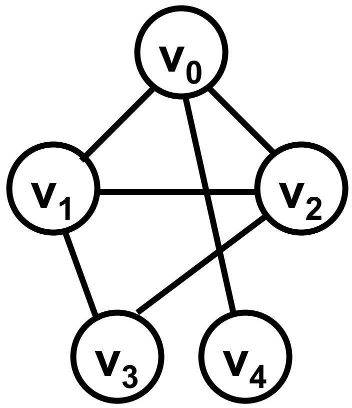
$V = \{v_0, v_1, v_2, v_3, v_4\}$
$|V| = 5$

$E = \{ \{v_1, v_0\}, \{v_0, v_2\}, \{v_4, v_0\}, \{v_1, v_2\}, \{v_3, v_1\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_3\} \}$
$|E| = 8$

More Fundamentals

- Node v_b is <u>adjacent</u> to node v_a in a graph if there is an edge from v_a to v_b.
- A <u>path</u> in a graph is a sequence of vertices p_0, ..., p_n such that each adjacent pair of vertices p_k and p_{k+1} are connected by an edge from p_k to p_{k+1}.
- A <u>cycle</u> is a path that starts and ends at the same vertex (i.e. $p_0 = p_n$).
- The <u>degree</u> of a vertex in an undirected graph is the number of edges that connect to the vertex.

Graph Terminology



paths from v_0 to v_2:

$v_0 \rightarrow v_2$

$v_0 \rightarrow v_1 \rightarrow v_2$

$v_0 \rightarrow v_1 \rightarrow v_3 \rightarrow v_2$

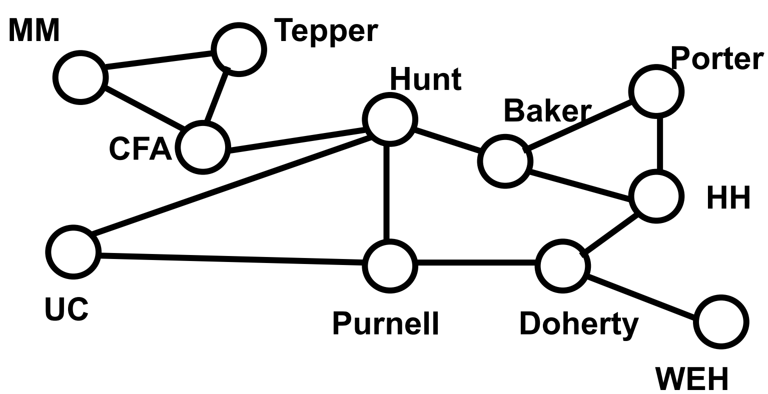
cycles starting at v_3:

$v_3 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$

$v_3 \rightarrow v_2 \rightarrow v_0 \rightarrow v_1 \rightarrow v_3$

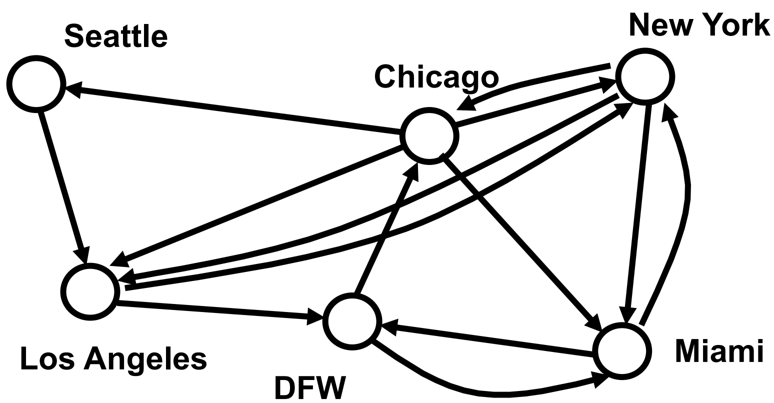
Graph Examples

- Communication Networks



Graph Examples

- Transportation Routes



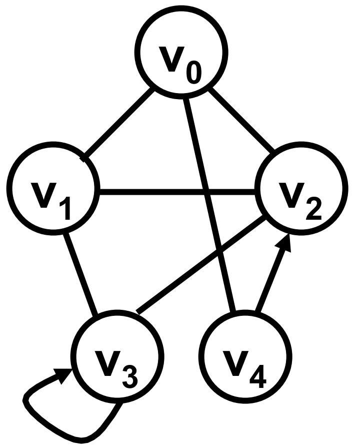
Storing a graph

- Use an Adjacency Matrix

  An adjacency matrix G for an n-node graph is an n x n array of boolean values such that G_{jk} = true if vertex k is adjacent to vertex j; otherwise G_{jk} = false.

  In other words, G_{jk} = true if there is an edge from vertex j to vertex k; otherwise it is false.

Example (no multiple edges)



destination

source

	0	1	2	3	4
0					
1					
2					
3					
4					

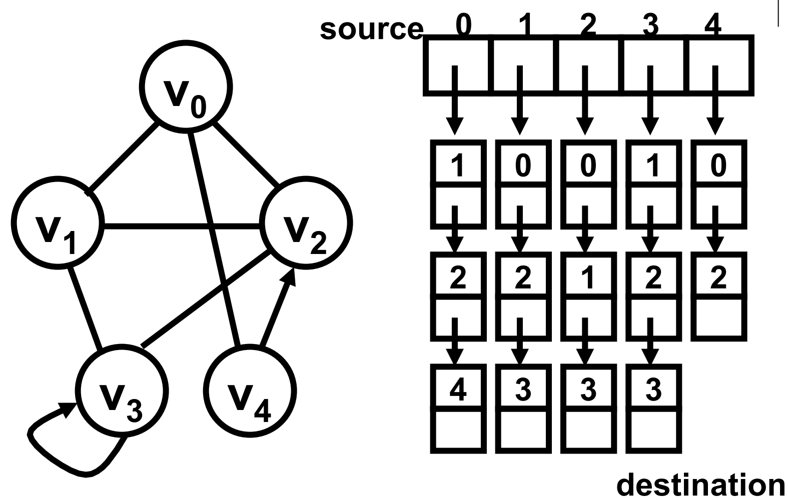
Storing a graph: Another way

- Use an array of edge lists

  An edge list for vertex k is a linked list that stores all nodes that are adjacent to vertex k.

  There is a linked list for every vertex of the graph.

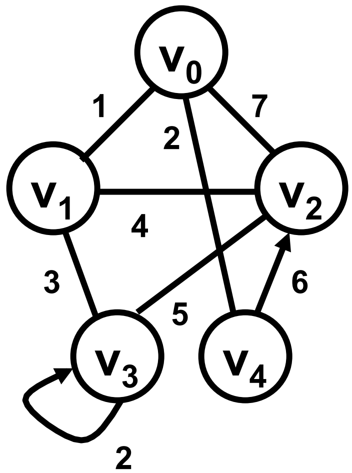
Example (no multiple edges)



Weighted Graphs

- Some graphs have an associated "weight" assigned to each edge.
- Weights: cost, distance, capacity, etc.
- Costs are typical non-negative integer values.
- Possible problems to solve using weighted graphs: shortest path between nodes, minimal spanning tree, etc.

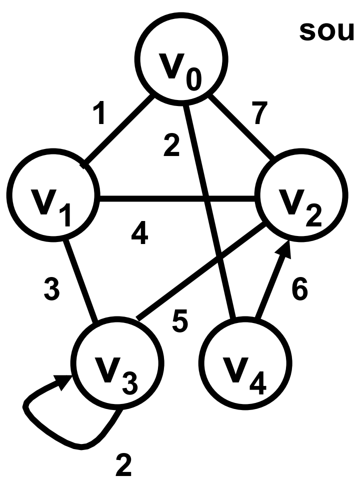
Example (no multiple edges)

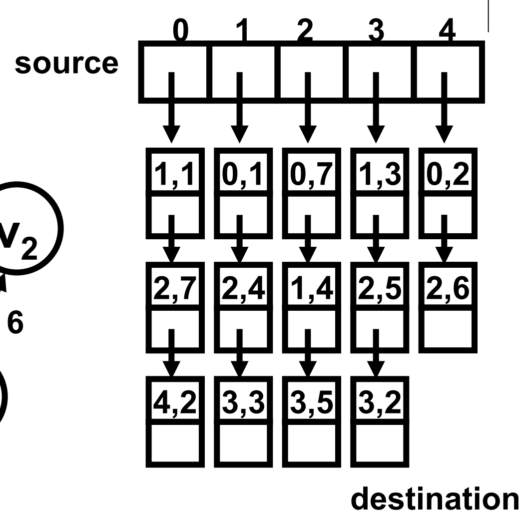


destination

source	0	1	2	3	4
0					
1					
2					
3					
4					

Example (no multiple edges)





Comparing Storage Methods

- Adjacency Matrix
  - Faster to add or remove an edge
  - Faster to determine if an edge exists in a graph.
  - Generally better for dense graphs where $|E| = O(|V|^2)$
  - Storage: $O(|V|^2)$
- Edge Lists
  - Faster to perform an operation on all nodes adjacent to a node in a sparse graph.
  - Generally better for sparse graphs where $|E| = O(|V|)$
  - Storage: $O(|V| + |E|)$

Implementing Graphs

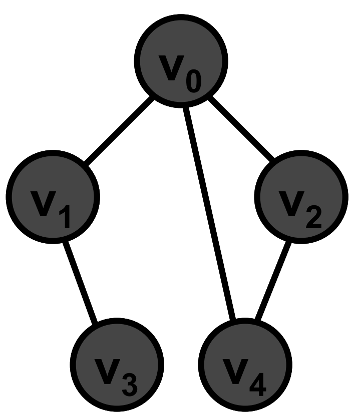
- Vertices will be integers numbered from 0 to |V|-1
- Edge class:

```
private int dest;
private int source;
private double weight;
public Edge(int s, int d)
public Edge(int s, int d, double w)
public int getDest()
public int getSource()
public double getWeight()        etc.
```

Graph Interface

```
public interface Graph {
  int getNumV();
  boolean isDirected();
  void insert(Edge edge);
  boolean isEdge(int source, int dest)
  Edge getEdge(int source, int dest)
  Iterator<Edge> edgeIterator(int source);
}
```


Traversing Graphs: Depth-First Traversal



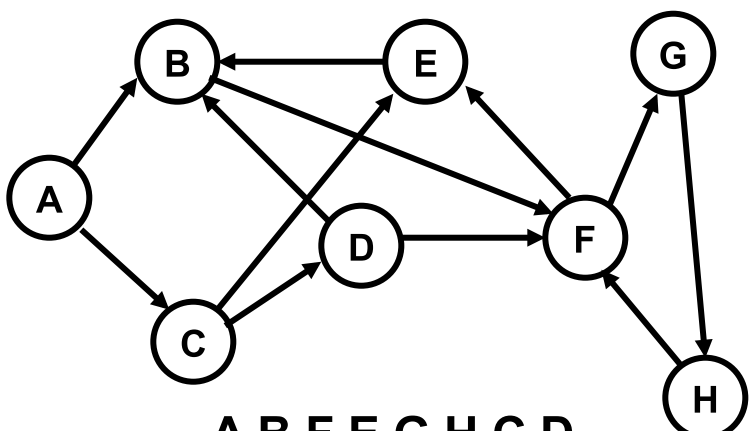
v_0 v_1 v_3 v_2 v_4

Depth-First Traversal

- Pick a starting node.
- Process this node and mark it as visited.
- For each of the neighbors of this node,
  - if the neighbor is unmarked,
    traverse the graph starting at the
    neighbor recursively

Nodes are marked as they are processed to avoid reprocessing these nodes along another path (due to a cycle).

Depth-First Traversal



A B F E G H C D

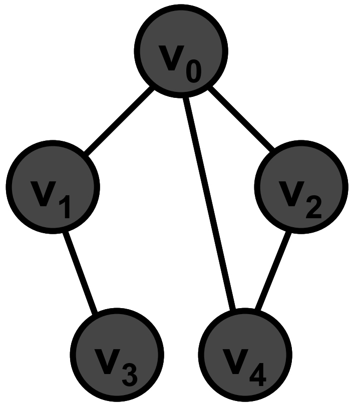
DFT Recursively

```
public static void DFT(Graph g,
  int v, boolean[] visited) {
  visited[v] = true;
  System.out.println(v);
  Iterator<Edge> iter = g.edgeIterator(v);
  while (iter.hasNext()) {
      int neighbor = iter.next().getDest();
      if (!visited[neighbor])
           DFT(g, neighbor, visited);
  }
}
```


Starting the DFT

```
public static void DFTStart(Graph g,
       int startVertex) {
  int i;
  boolean[] visited
     = new boolean[g.getNumV()];
  for (i=0; i<visited.length; i++) {
      visited[i] = false;
  DFT(g, startVertex, visited);
}
```

Traversing Graphs: Breadth-First Traversal

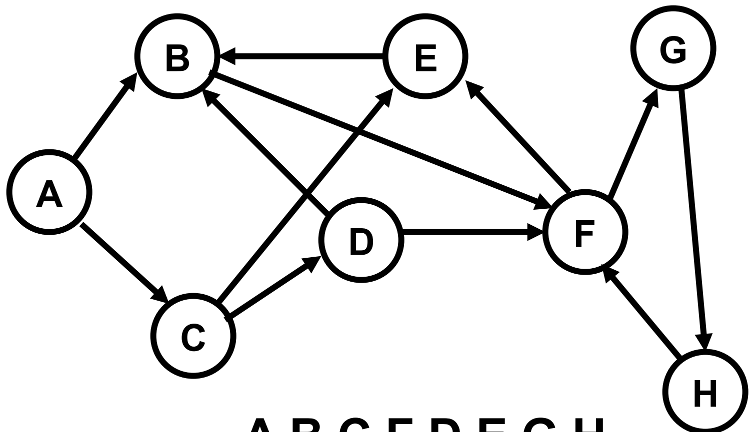


v_0 v_1 v_2 v_4 v_3

Breadth-First Traversal

- Pick a starting node. Mark it as visited and put it in a queue.
- While the queue is not empty:
  - dequeue a node.
  - process that node.
  - for each neighbor that is not marked:
    -mark that neighbor and enqueue it

Breadth-First Traversal



A B C F D E G H

BFT using a queue

```
public static void BFT(Graph g, int v) {
  boolean[] visited
        = new boolean[g.getNumV()];
  Queue<Integer> q = new Queue<Integer>();
  visited[v] = true;
  q.enqueue(v);
  while (!q.isEmpty()) {
   int current = q.dequeue();
   System.out.println(current);
```

(cont'd)

BFT using a queue (cont'd)

```
    Iterator<Edge> iter =
              g.edgeIterator(current);
    while (iter.hasNext()) {
        neighbor = iter.next().getDest();
        if (!visited[neighbor]) {
              visited[neighbor]=true;
              q.enqueue(neighbor);
        }
    }
  } // end while !q.isEmpty()
}
```

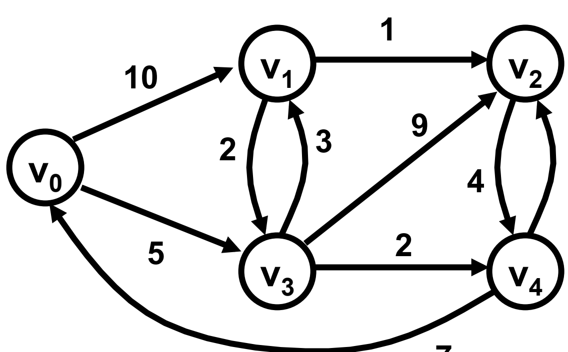

Dijkstra's Shortest Path Algorithm

- This algorithm finds the minimum total weight from a source node to every other node of a graph assuming all edges have non-negative.
  - Developed by Edsger W. Dijkstra, winner of the 1972 ACM Turing Award

- Shortest Path means "least total weight of all the edges on that path"
- weight(u,v) = weight on edge (u,v) or
  infinity if there is no edge from u to v
- This algorithm assumes all weights are positive.

Dijkstra's Shortest Path Algorithm

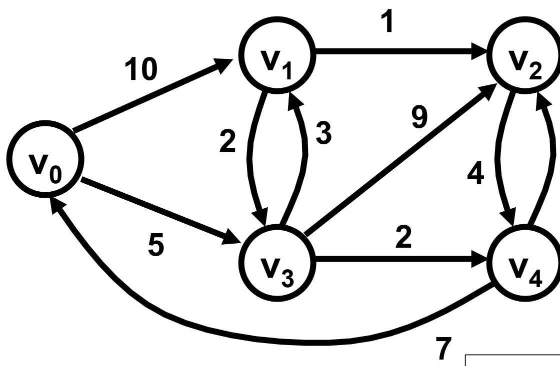
```
for each vertex v in V do distance[v] = infinity
distance[source] = 0
S = { }
for i = 1 to (|V| – 1)
   next = index of min distance of all vertices in V - S
   S = S ∪ next
   for each vertex v in V – S that is neighbor of next
      if (distance[next] + weight(next,v) < distance[v])
        distance[v] = distance[next] + weight(next,v)
```

	0	1	2	3	4
distance	**0**	**10**	**∞**	**5**	**∞**

S = {0}
V – S = {1,2,3,4}
next = 0

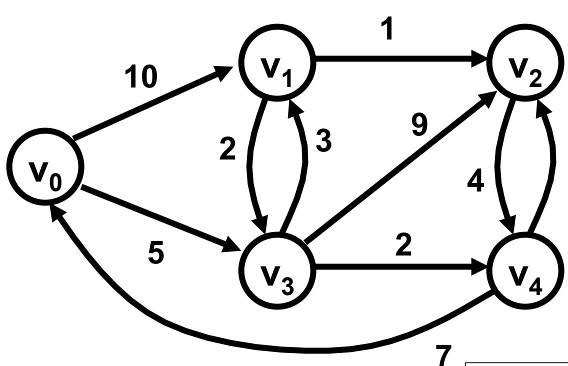
distance[0] + weight(0,1) = 10   distance[1] = ∞
distance[0] + weight(0,3) = 5   distance[3] = ∞



	0	1	2	3	4
distance	0	8	14	5	7

S = {0,3}
V – S = {1,2,4}
next = 3

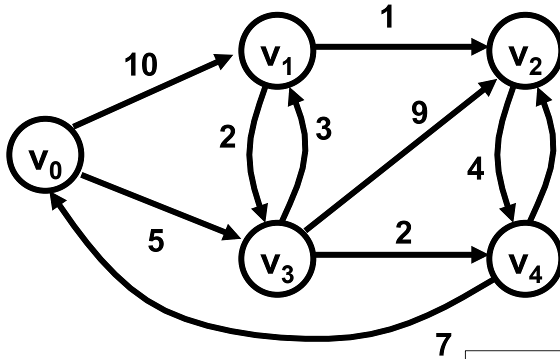
distance[3] + weight(3,1) = 8   distance[1] = 10
distance[3] + weight(3,2) = 14   distance[2] = ∞
distance[3] + weight(3,4) = 7   distance[4] = ∞



	0	1	2	3	4
distance	0	8	13	5	7

S = {0,3,4}
V – S = {1,2}
next = 4

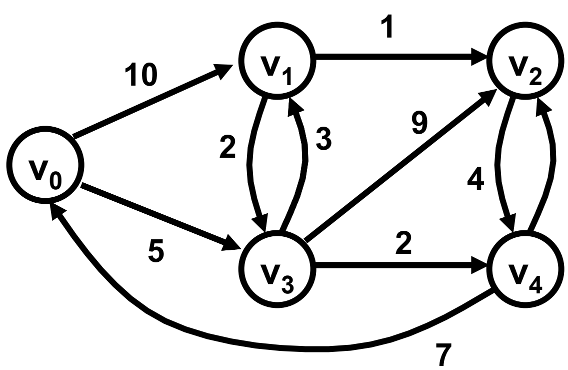
distance[4] + weight(4,2) = 13   distance[2] = 14



S = {0,1,3,4}
V – S = {2}
next = 1

	0	1	2	3	4
distance	0	8	9	5	7

distance[1] + weight(1,2) = 9 distance[2] = 13



S = {0,1,3,4}
V – S = {2}

	0	1	2	3	4
distance	0	8	9	5	7

See your textbook for an implementation in Java.