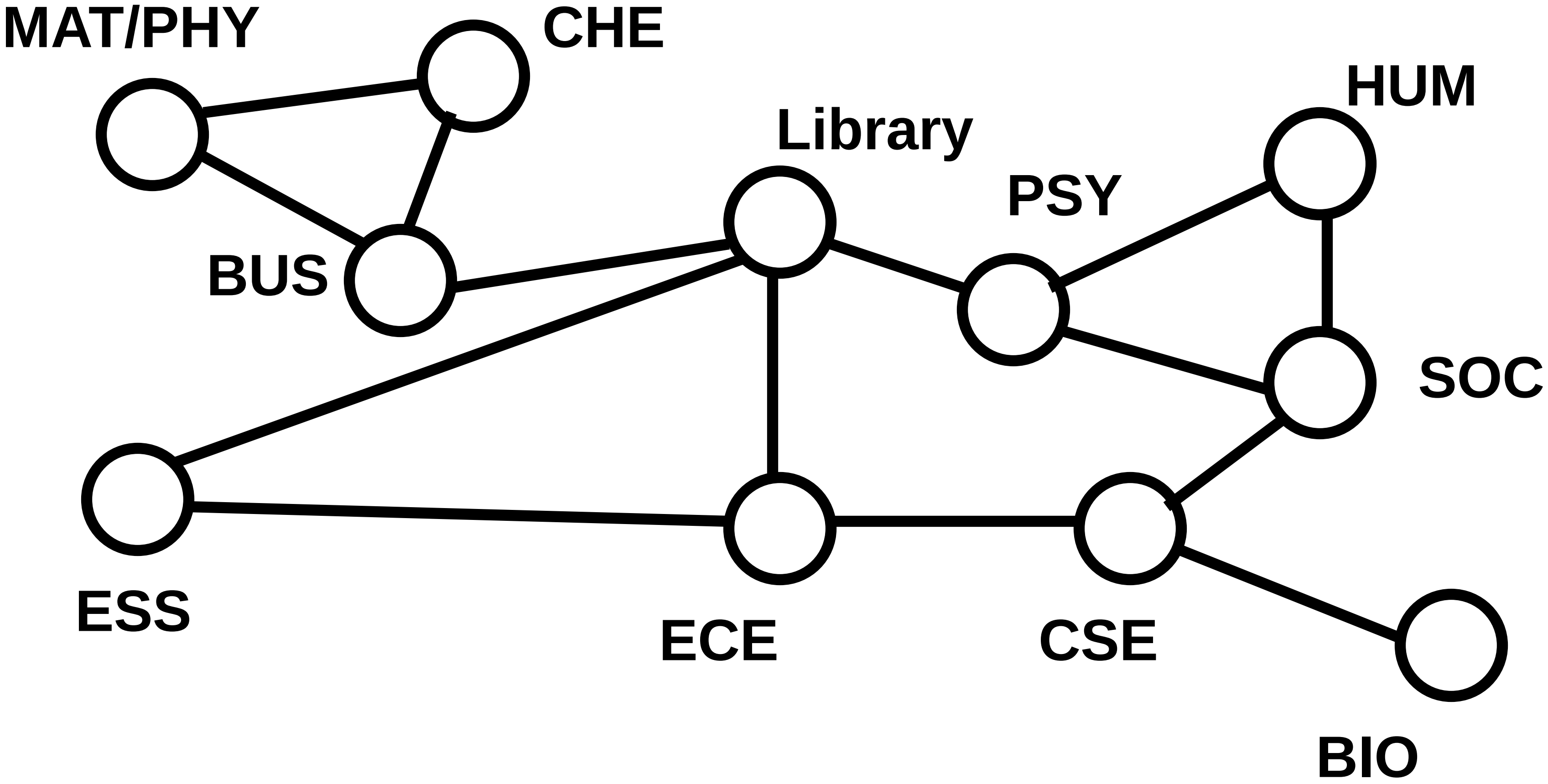
Graphs

Chapter 14

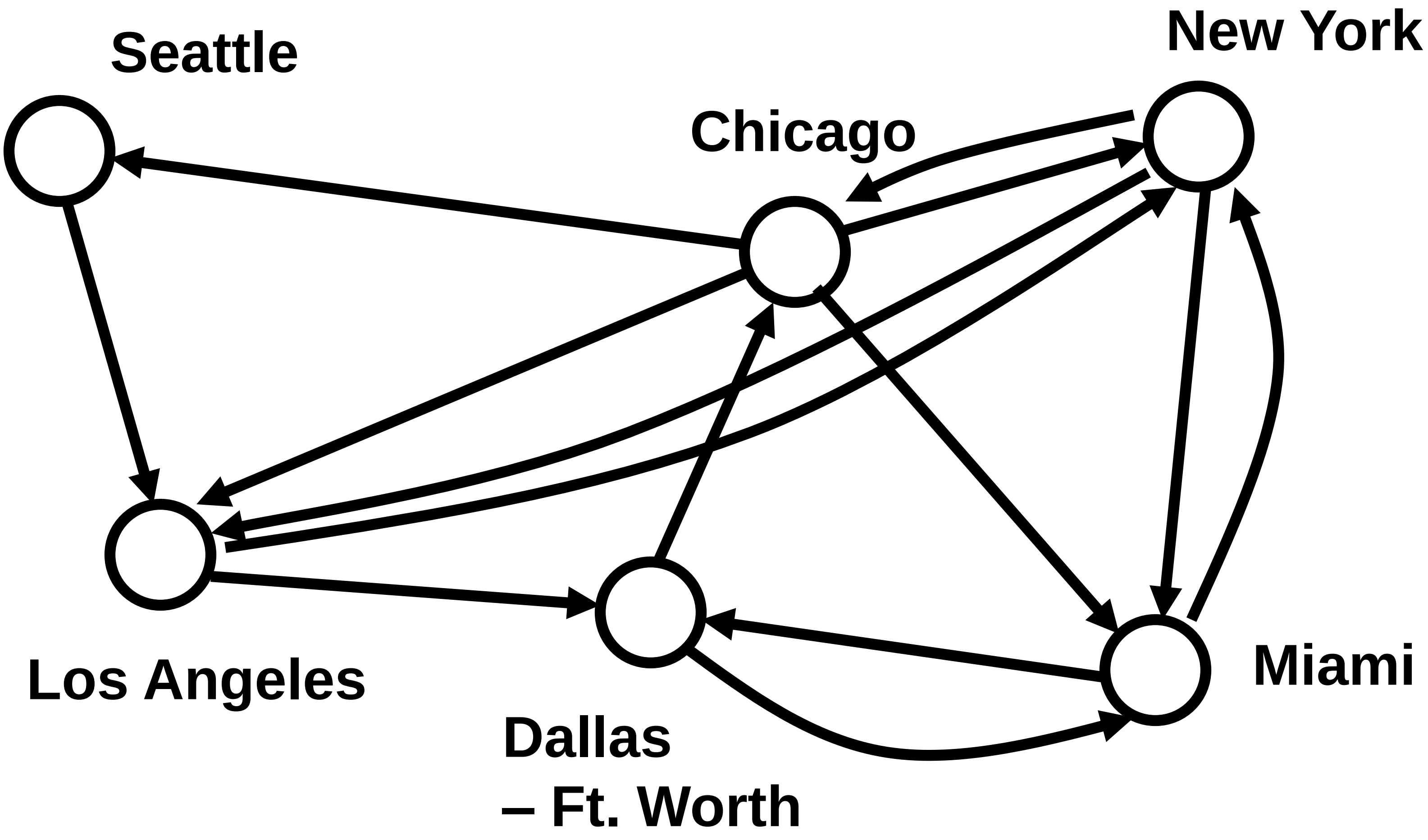
Examples

- Communication Networks



Examples

- Transportation Routes



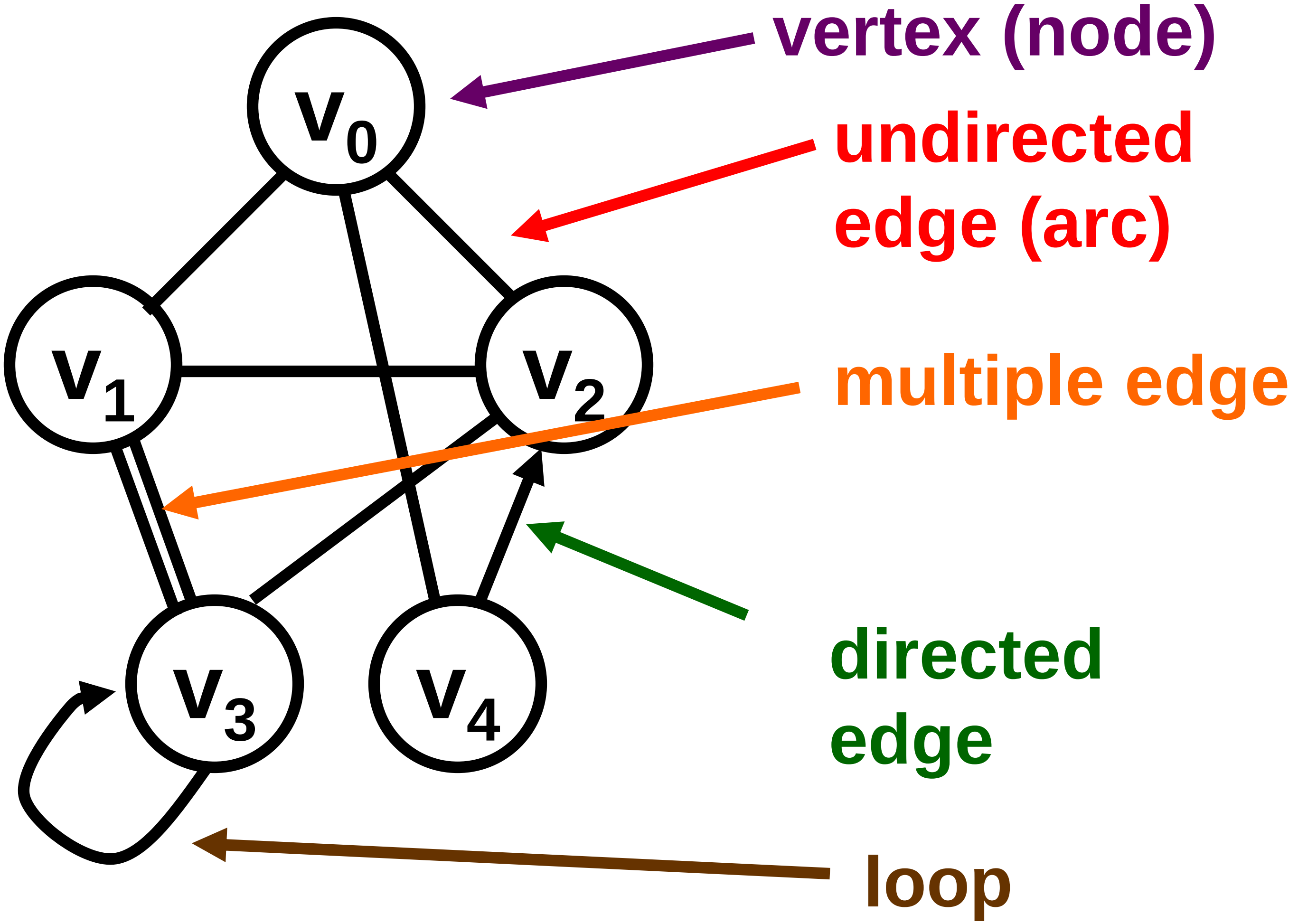
Fundamentals

- 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.

More Fundamentals

- Node v_b is adjacent to node v_a in a graph if there is an edge from v_a to v_b.
- A path 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 cycle is a path that starts and ends at the same vertex (i.e. $p_0 = p_n$).
- The degree of a vertex in an undirected graph is the number of edges that connect to the vertex.

Graph Terminology



vertex (node)

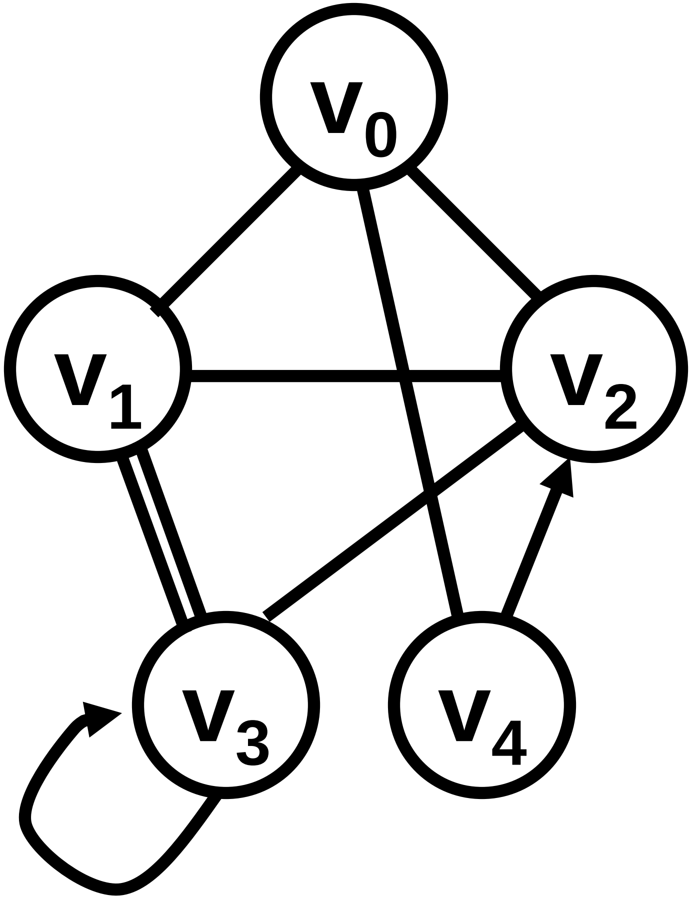
undirected edge (arc)

multiple edge

directed edge

loop

Graph Terminology



paths from v_0 to v_2:

v_0, v_2

v_0, v_1, v_2

v_0, v_4, v_2

v_0, v_1, v_3, v_2

v_0, v_1, v_3, v_3, v_2, etc.

cycles at v_4:

v_4, v_2, v_0, v_4

v_4, v_2, v_1, v_0, v_4, **etc.**

Storing a graph

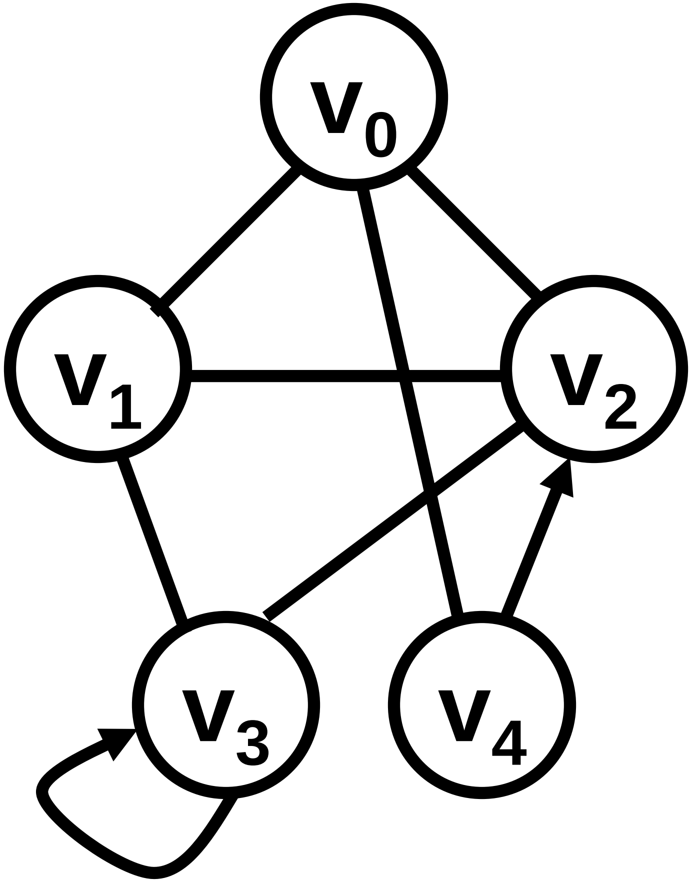
- Use an Adjacency Matrix

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

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

  Is $G_{ik} = G_{ki}$?

Example (no multiple edges)



target

source

	0	1	2	3	4
0	F	T	T	F	T
1	T	F	T	T	F
2	T	T	F	T	F
3	F	T	T	T	F
4	T	F	T	F	F

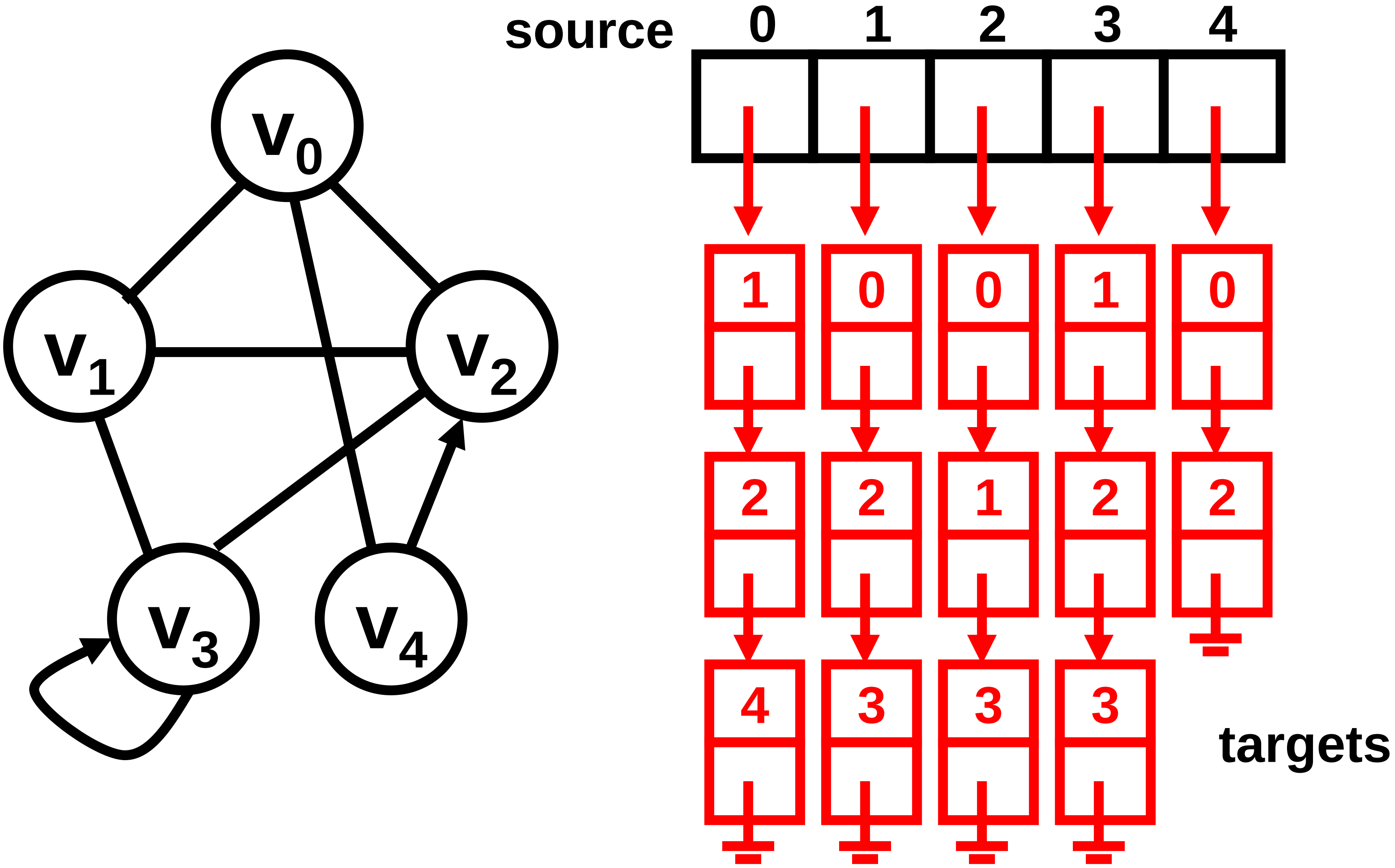
Storing a graph: Another way

- 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.

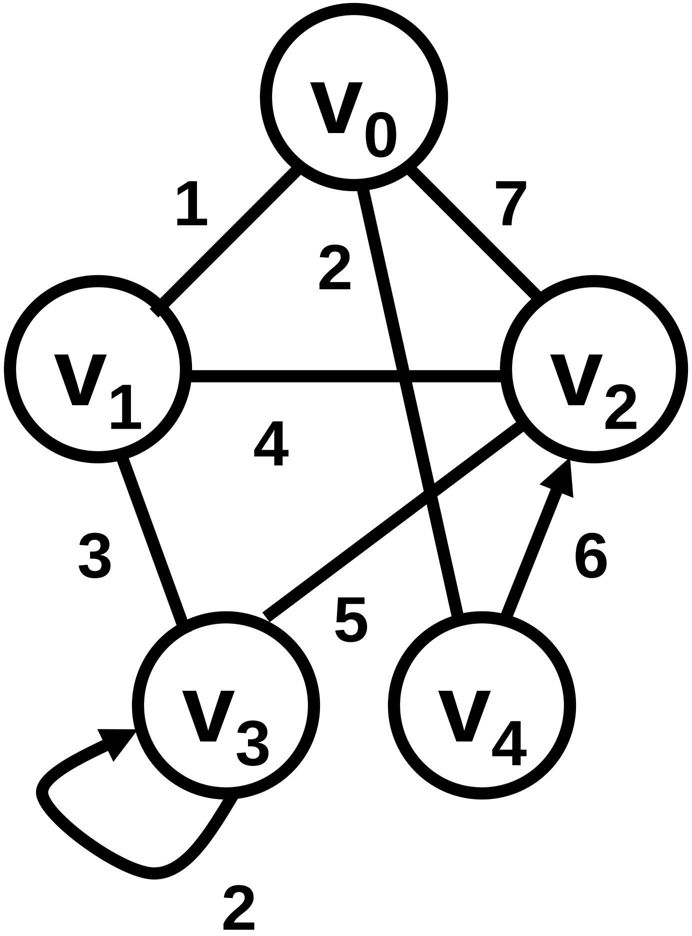
Example Again (no multiple edges)



Weighted Graphs

- Some graphs have an associated “weight” assigned to each edge.
- Weights: cost, distance, capacity, etc.
- Cost 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)

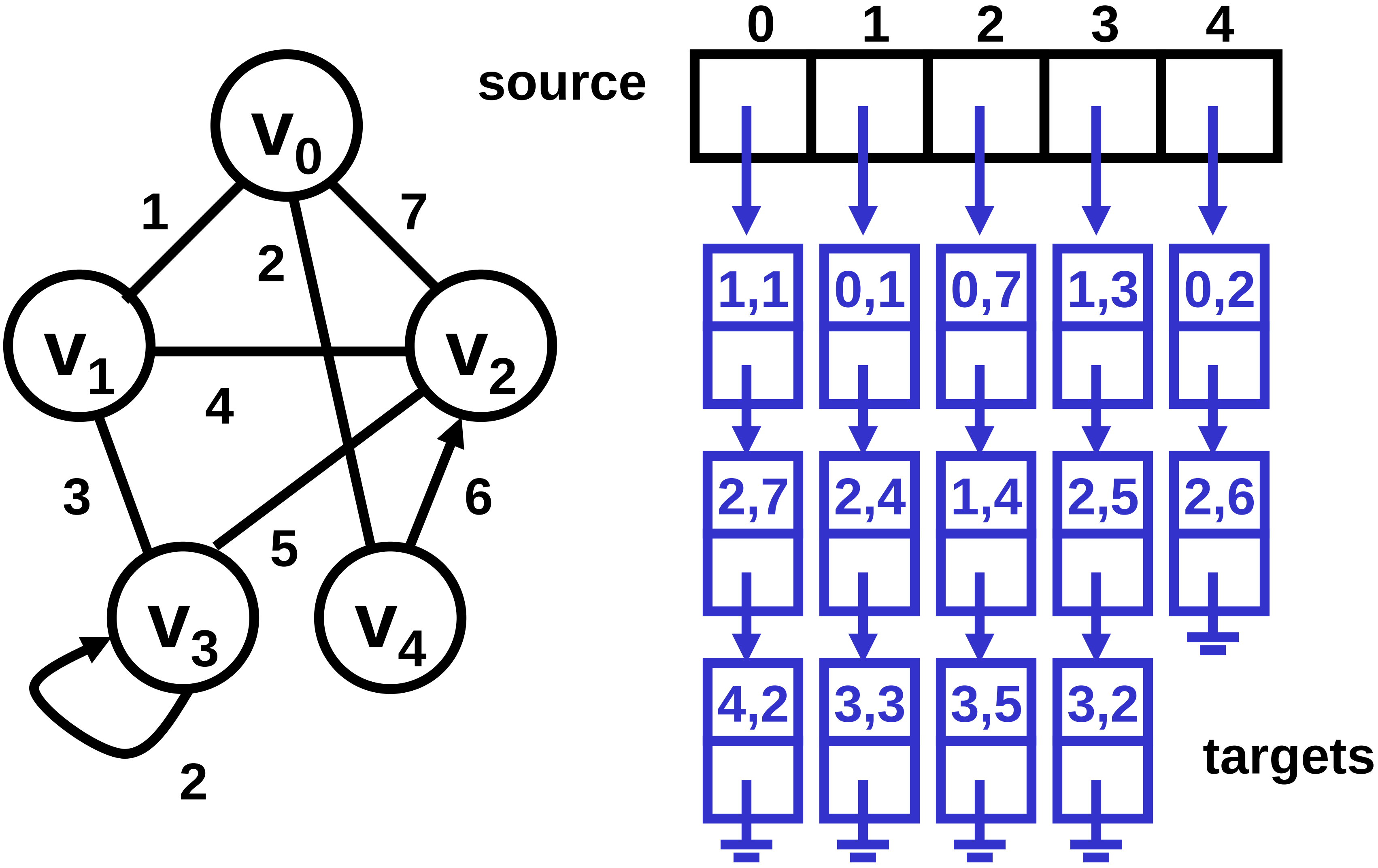


target

source

	0	1	2	3	4
0	-1	1	7	-1	2
1	1	-1	4	3	-1
2	7	4	-1	5	-1
3	-1	3	5	2	-1
4	2	-1	6	-1	-1

Example (no multiple edges)



Which storage method is better?

- Adjacency Matrix
  - Easier to implement
  - Faster to add or remove an edge
  - Faster to check for an edge
- Edge Lists
  - Faster to perform an operation on all nodes adjacent to a node.
  - Uses less memory if graph is sparse.

The Graph ADT (parameters in red)

- constructor: Initializes an empty graph that can store a maximum of n vertices
- size: Returns the maximum number of nodes that the graph can hold.
- addEdge: Adds an edge from vertex source to vertex target.
- isEdge: Returns true if there is an edge from vertex source to vertex target.
- removeEdge: Removes the edge from vertex source to vertex target.
- getLabel: Gets the label for the given vertex.
- setLabel: Sets the label for the given vertex.
- neighbors: Returns an array of the adjacent vertices to the given vertex.

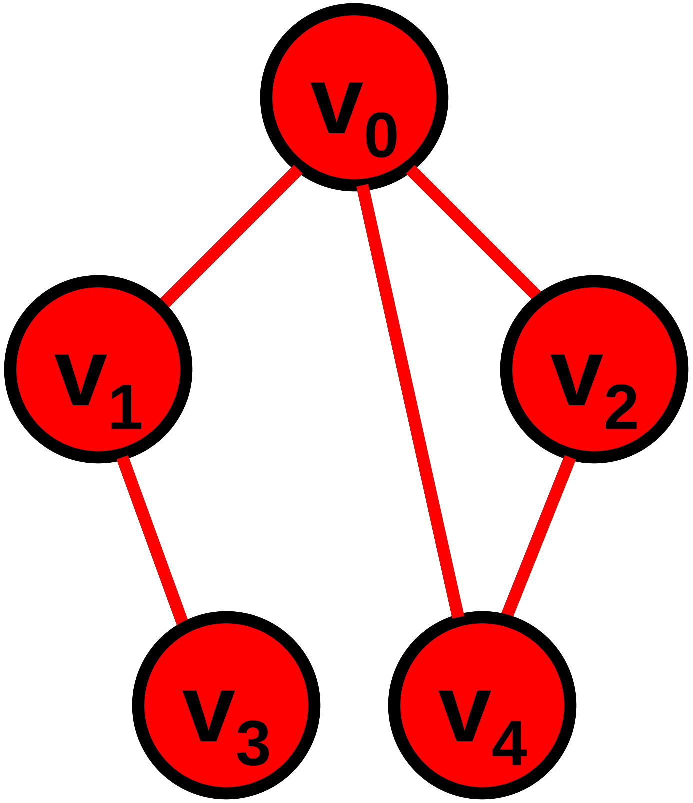
Traversing Graphs: Depth-First Traversal

- 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

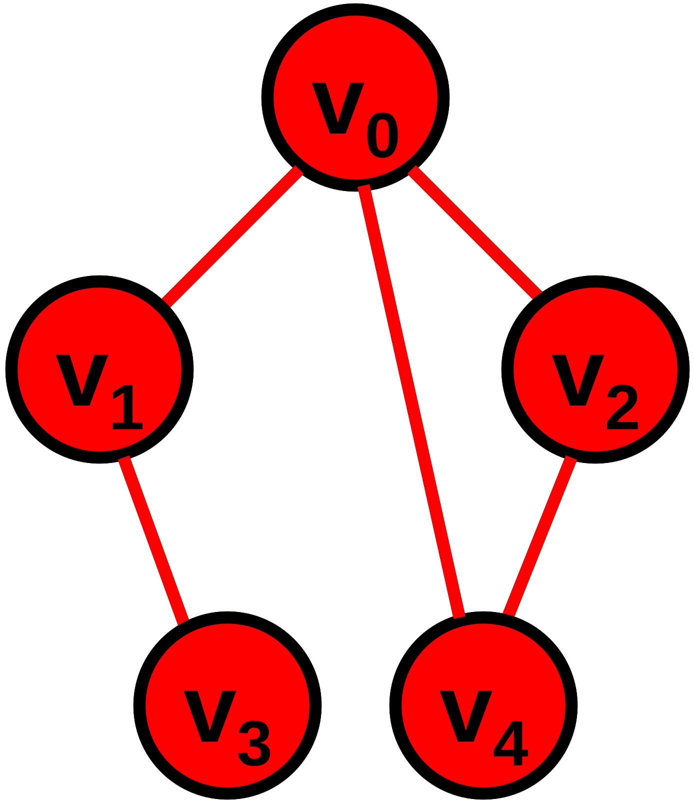
Nodes are marked as they are processed to avoid reprocessing these nodes along another path (due to a cycle).

Depth-First Traversal



$v_0 \; v_1 \; v_3 \; v_2 \; v_4$

Traversing Graphs: Depth-First Traversal

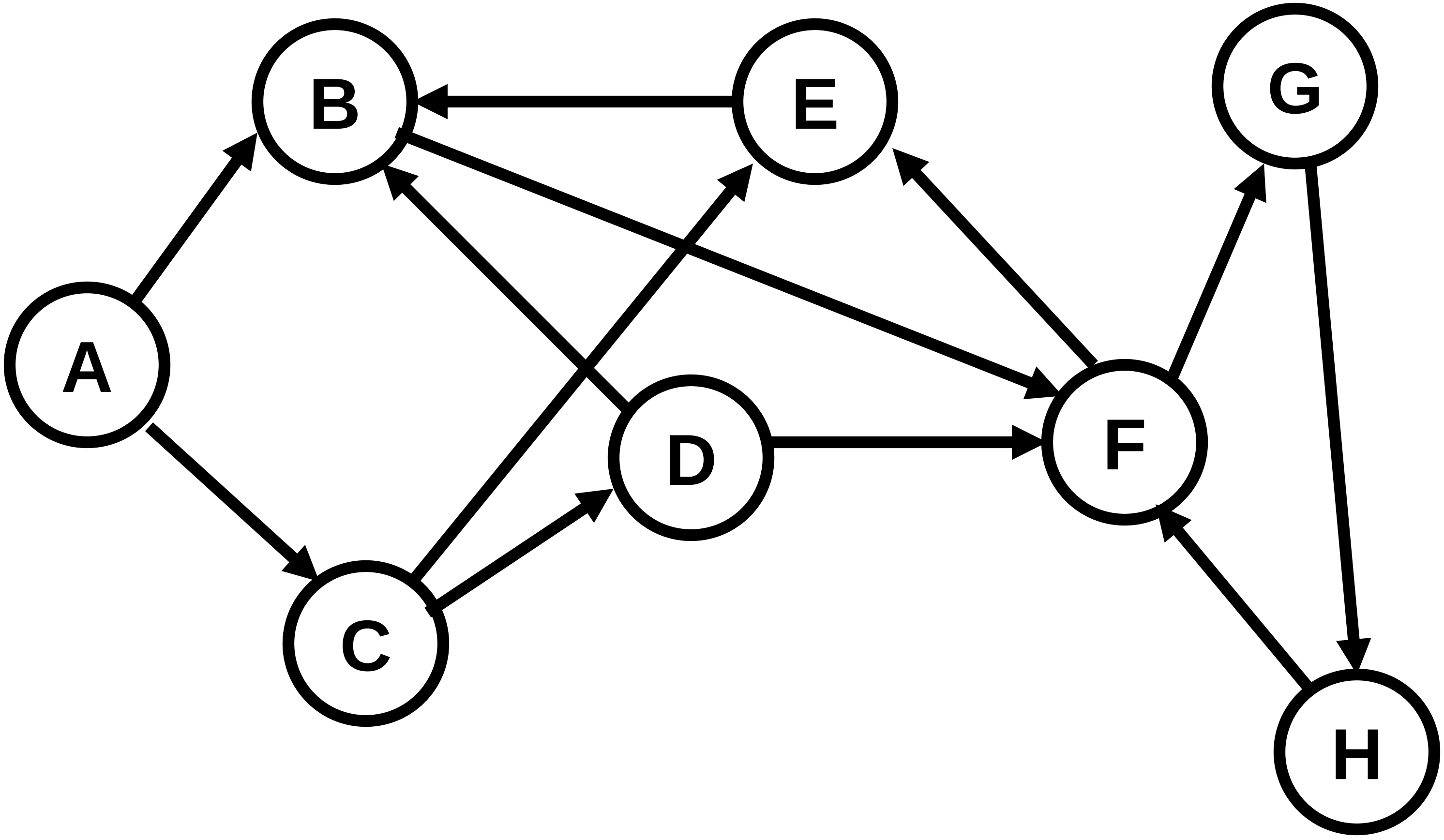


V_0 V_1 V_3 V_2 V_4

V_0 V_2 V_4 V_1 V_3

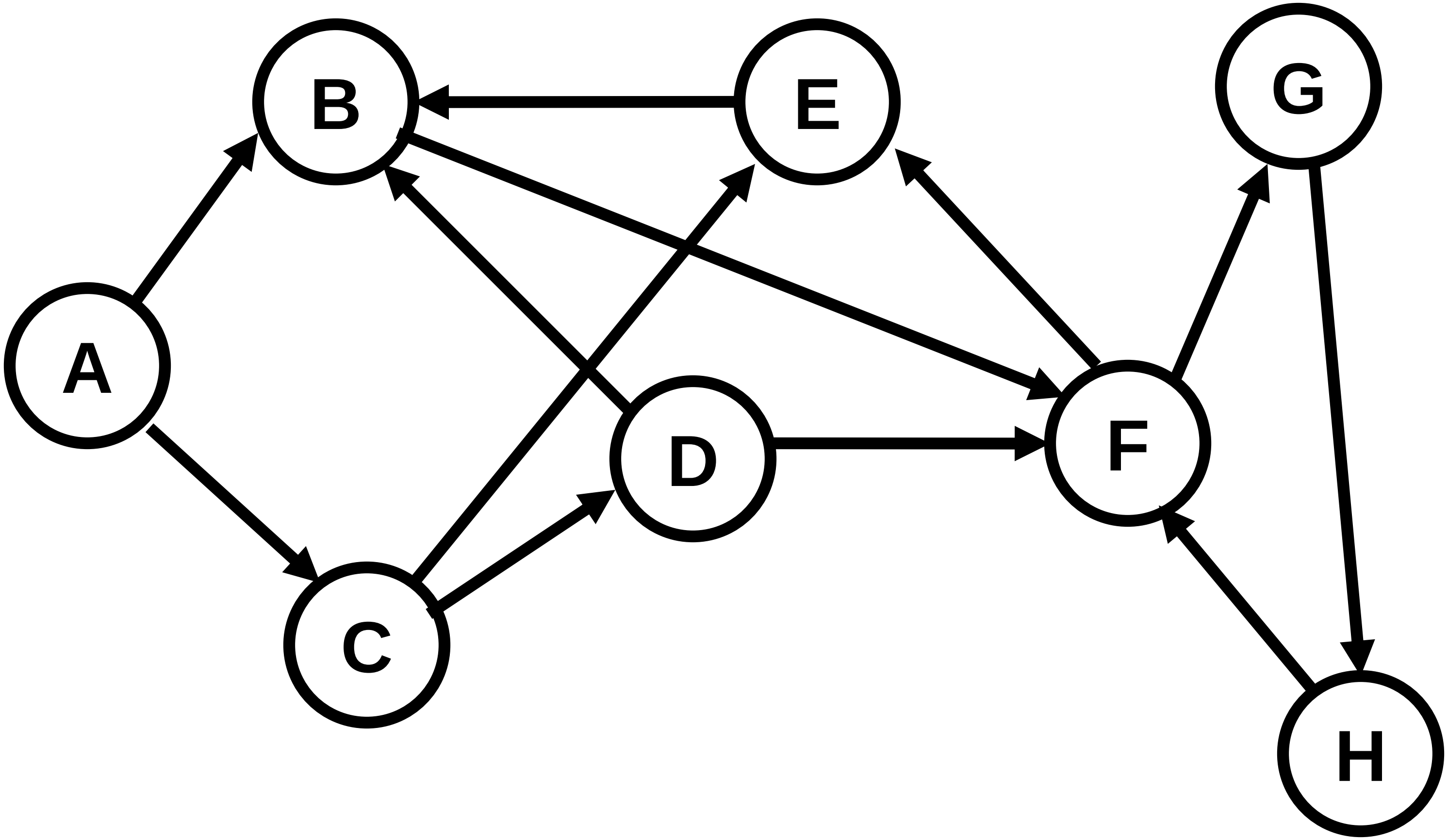
V_0 V_4 V_2 V_1 V_3

Depth-First Traversal



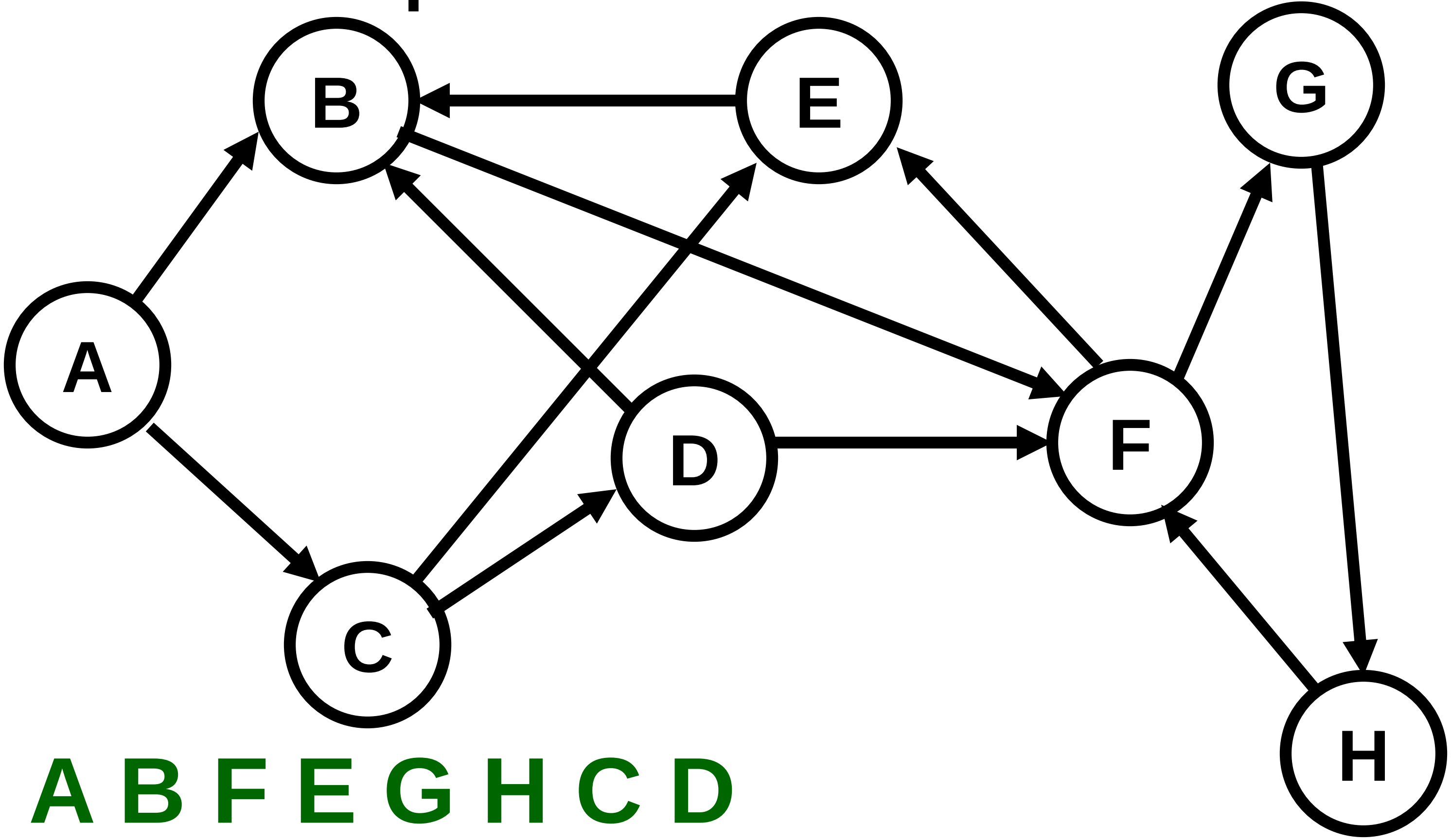
A B F E G H C D

Depth-First Traversal



A B F E G H C D

Depth-First Traversal



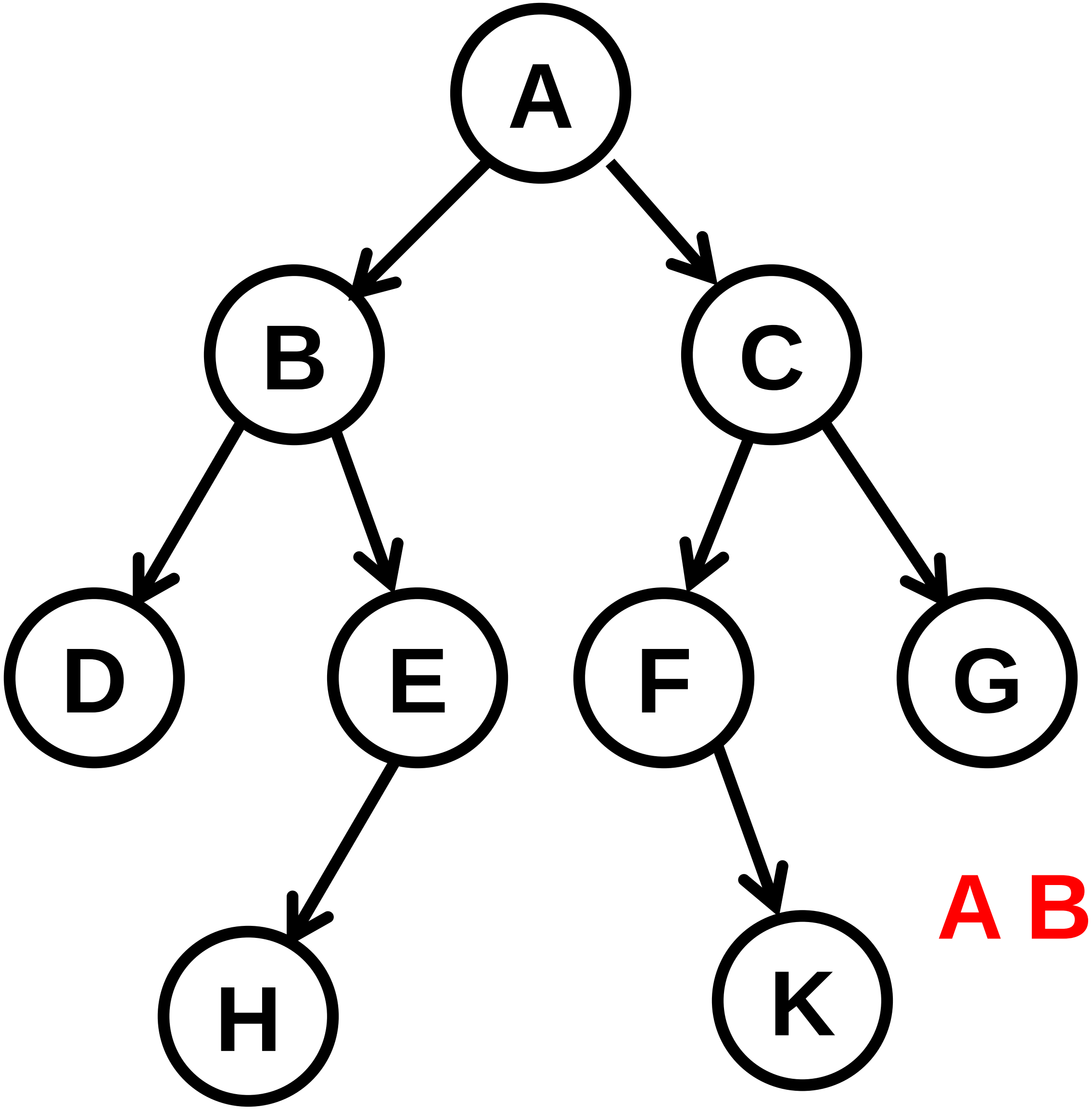
A B F E G H C D

A C D F G H E B

A C D B F E G H

A C E B F G H D

Depth-First Traversal



A B D E H C F K G

```
public static void DFT(Graph g,
  int v, boolean[] marked) {
  int[] connections = g.neighbors(v);
  int i;
  int nextNeighbor;
  marked[v] = true;
  System.out.println(g.getLabel(v));
  for (i=0; i<connections.length; i++) {
     nextNeighbor = connections[i];
     if (!marked[nextNeighbor])
          DFT(g, nextNeighbor, marked);
  }
}
```

Is DFT tail recursive?

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

We can also use a stack in this implementation to avoid using recursion.

Traversing Graphs: Breadth-First Traversal

- Pick a starting node. Mark it as visited and put it in a queue.
- While the queue is not empty:

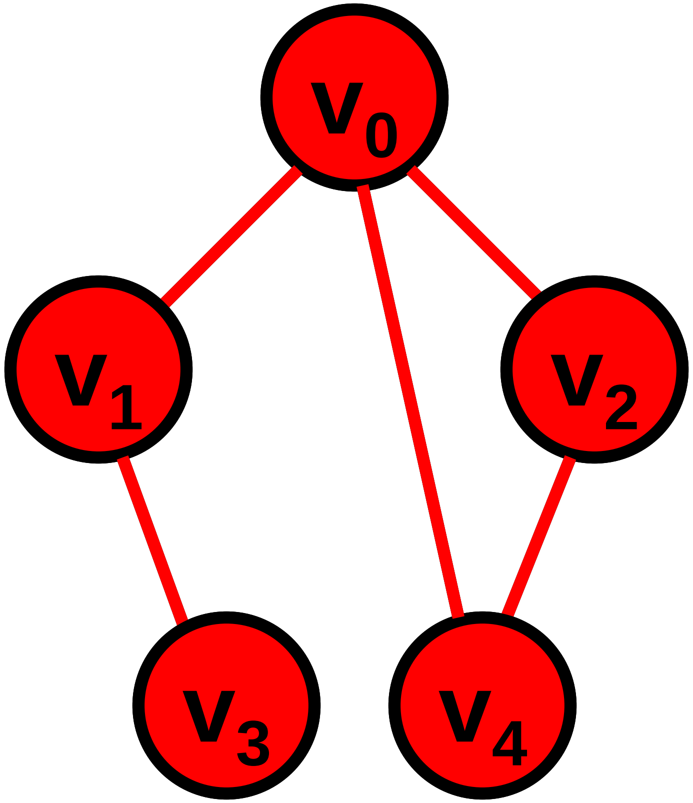
  - dequeue a node.

  - process that node.

  - for each neighbor that is not marked:

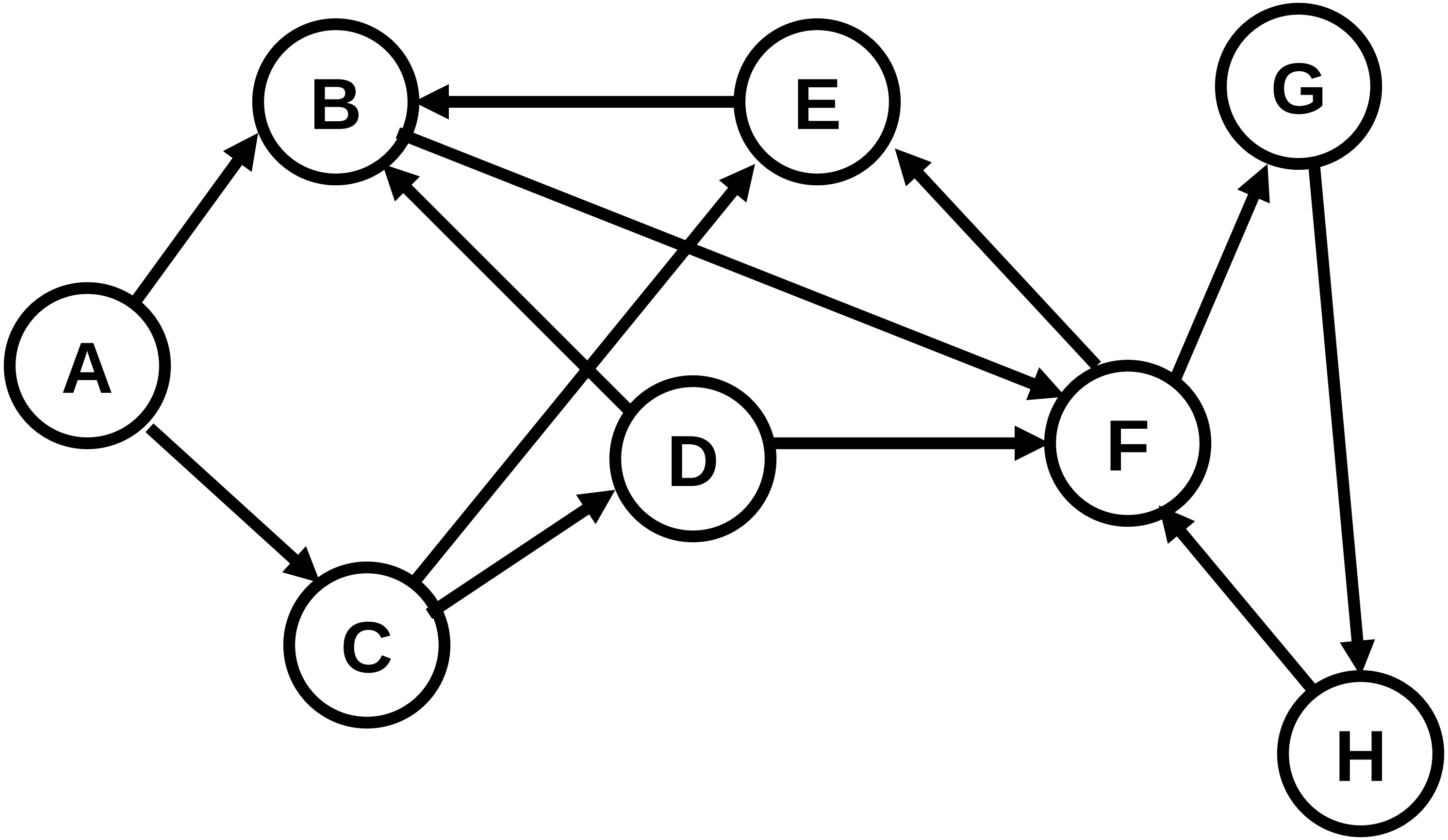
    -mark that neighbor and enqueue it

Breadth-First Traversal



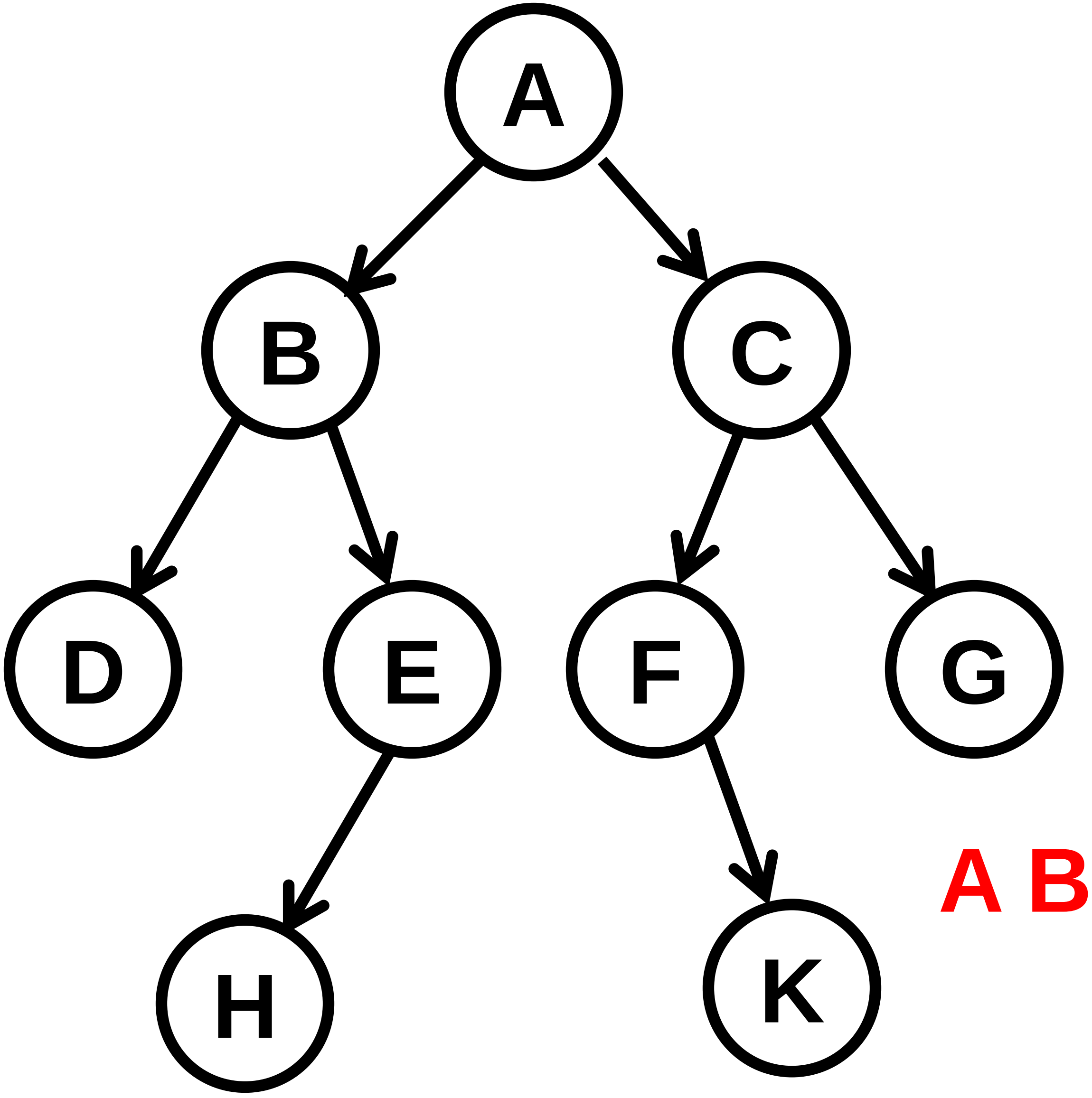
v_0 v_1 v_2 v_4 v_3

Breadth-First Traversal



A B C F D E G H

Breadth-First Traversal



A B C D E F G H K

```
public static void BFT(Graph g, int v) {
  boolean[] marked
         = new boolean[g.size()];
  int[] connections;
  int i;
  int vertex, nextNeighbor;
  IntQueue q = new IntQueue();
  marked[v] = true;
  q.enqueue(v);
  while (!q.isEmpty()) {
     vertex = q.dequeue();
     System.out.println
          (g.getLabel(vertex));
```

```
    connections = g.neighbors(vertex);
    for (i=0; i<connections.length; i++)
    {
        nextNeighbor = connections[i];
        if (!marked[nextNeighbor]) {
            marked[nextNeighbor]=true;
            q.enqueue(nextNeighbor);
        }
    }
  } // end while loop
}
```

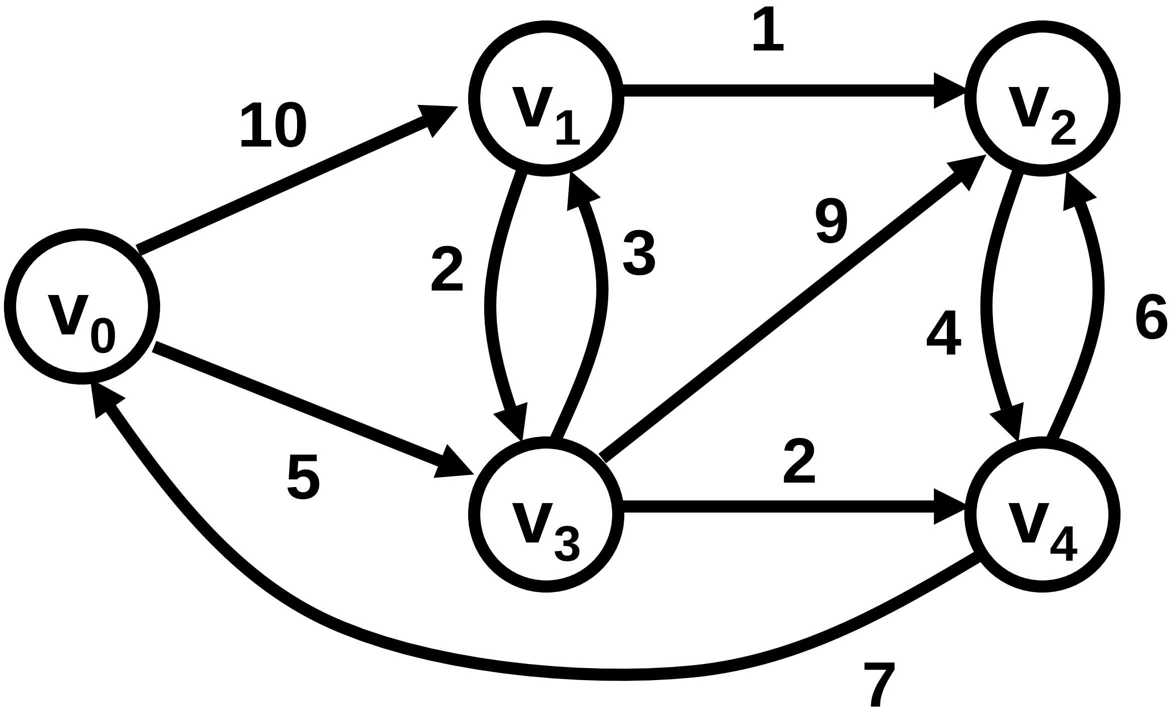
Dijkstra’s Shortest Path Algorithm

(optional)

- This algorithm finds the minimum total weight from a source node to every other node of a graph assuming all edges have non-negative.
- 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

Dijkstra’s Shortest Path Algorithm

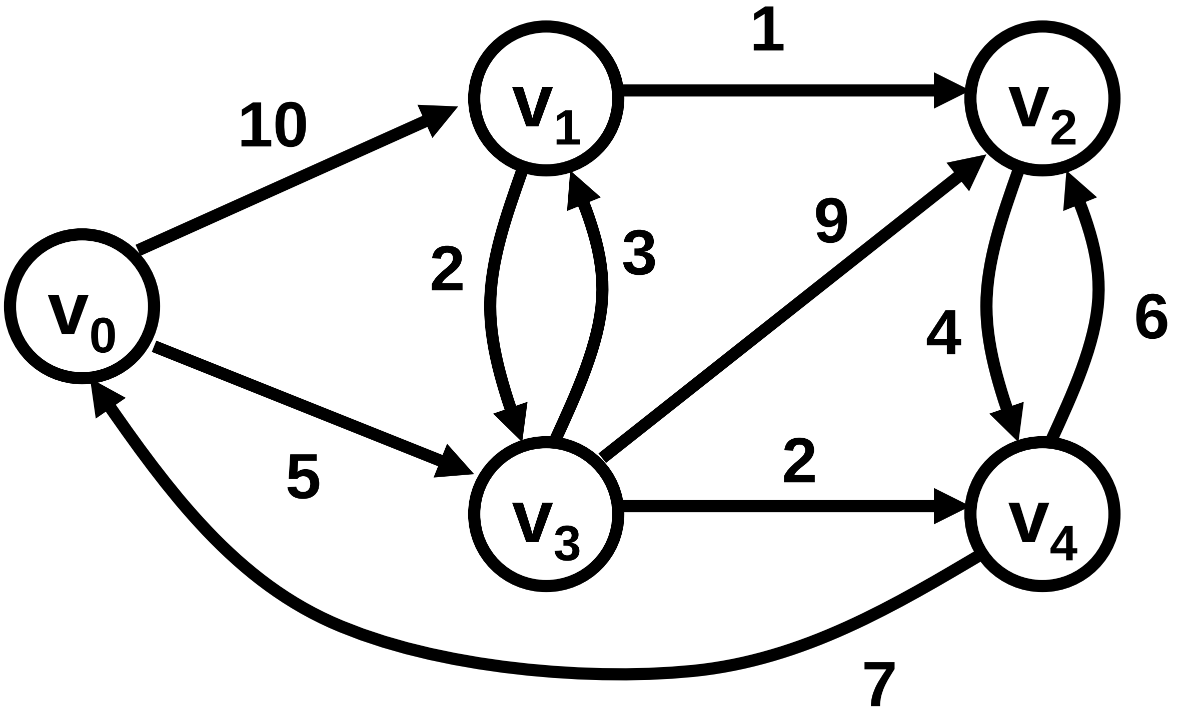
```
for each vertex v in V do distance[v] = infinity
distance[source] = 0
S = { }
for i = 1 to (number of vertices – 1)
   next = index of min distance of all vertices in V - S
   S = S ∪ next
   for each vertex v in V – S that is neighbor of next
      if (distance[next] + weight(next,v) < distance[v])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



	0	1	2	3	4
distance	0	∞	∞	∞	∞

S = {}

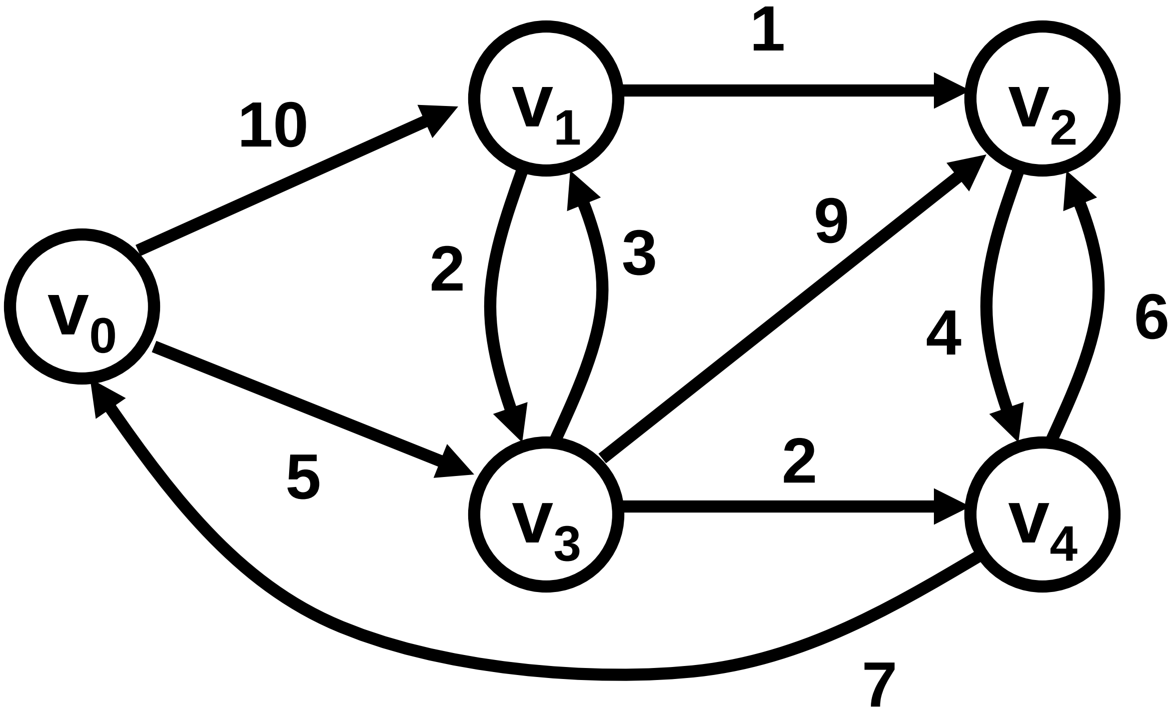
V – S = {0,1,2,3,4}



	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

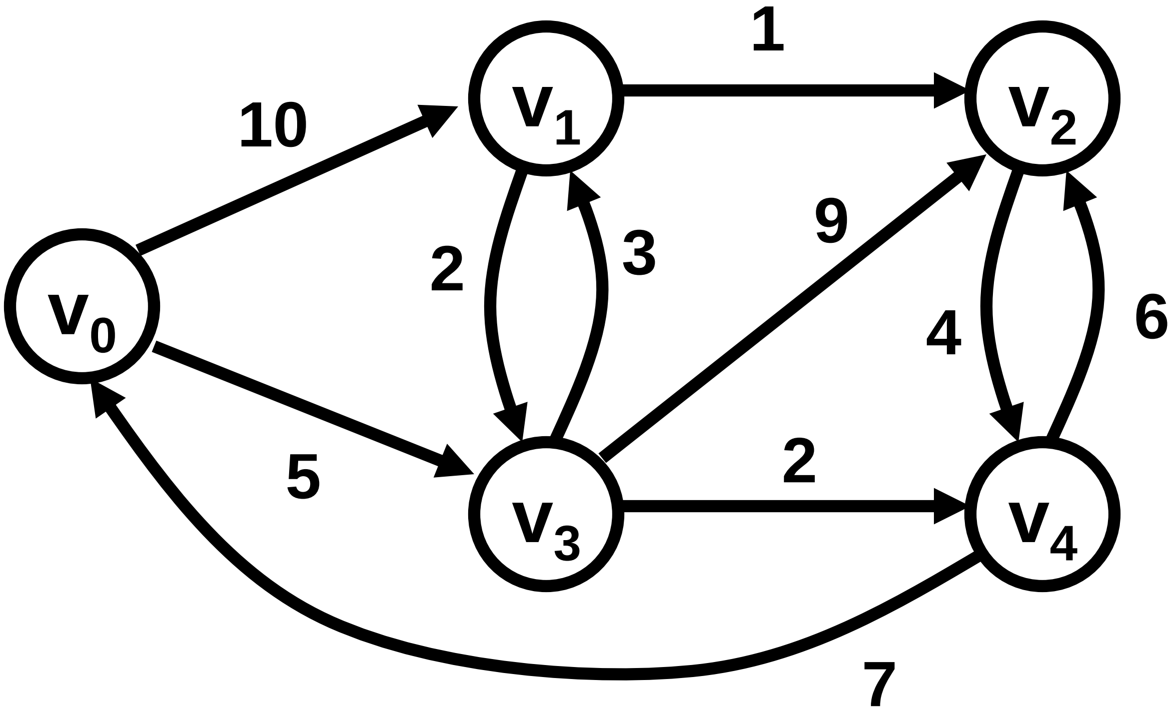
distance[3] + weight(3,2) = 14

distance[3] + weight(3,4) = 7

distance[1] = 10

distance[2] = ∞

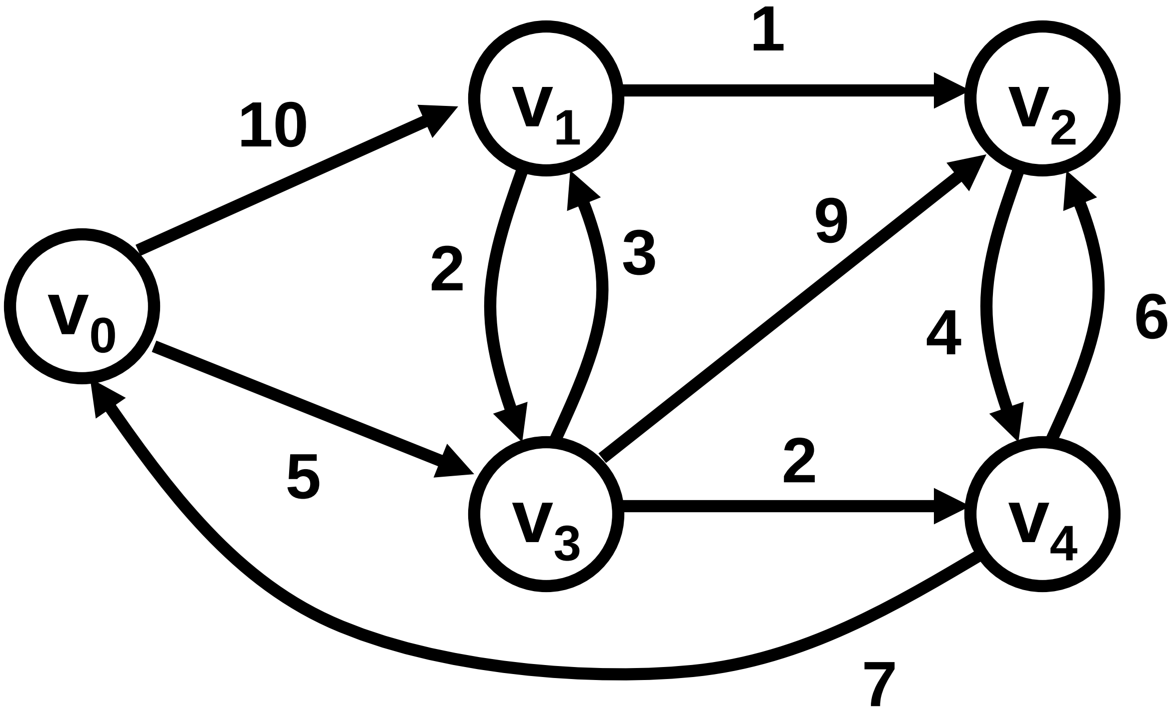
distance[4] = ∞



	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



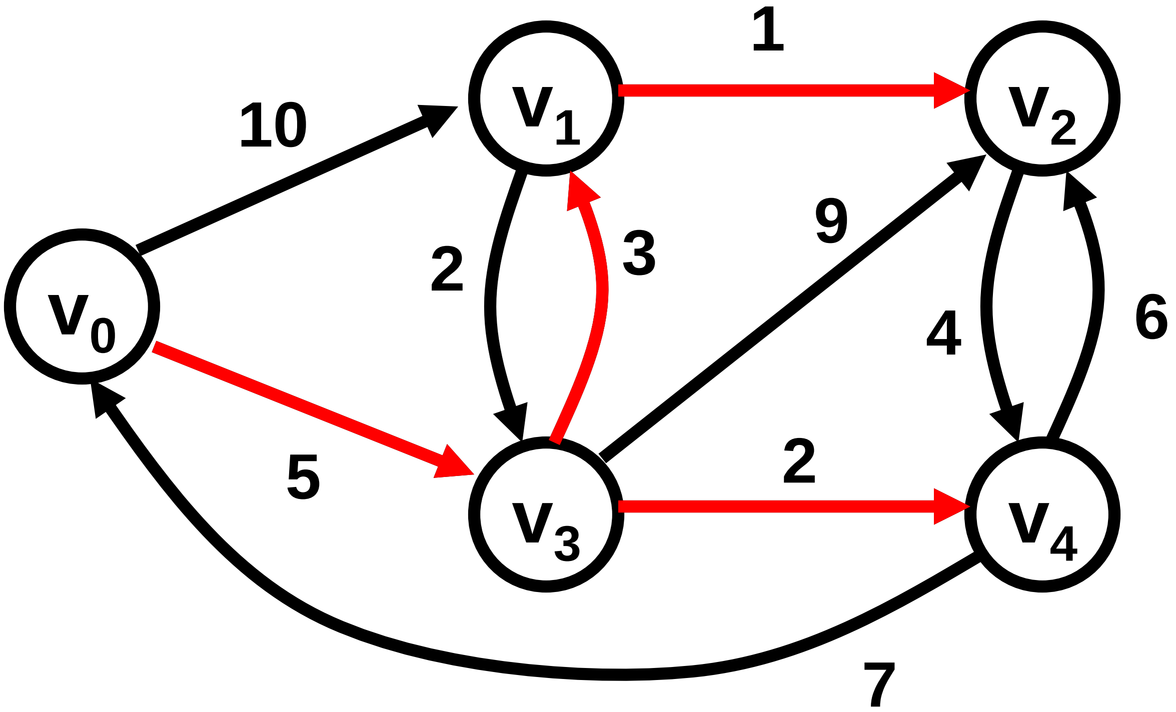
	0	1	2	3	4
distance	0	8	9	5	7

S = {0,1,3,4}

V – S = {2}

next = 1

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



	0	1	2	3	4
distance	0	8	9	5	7

$S = \{0,1,3,4\}$

$V - S = \{2\}$