

# Graphs

A **graph**  $G = (V, E)$  consists of a set of **vertices**  $V$ , and a set of **edges**  $E$ , such that each edge in  $E$  is a connection between a pair of vertices in  $V$ .

The number of vertices is written  $|V|$ , and the number of edges is written  $|E|$ .

A sequence of vertices  $v_1, v_2, \dots, v_n$  forms a **path** of length  $n - 1$  if there exist edges from  $v_i$  to  $v_{i+1}$  for  $1 \leq i < n$ .

A path is **simple** if all vertices on the path are distinct.

A **cycle** is a path of length 3 or more that connects  $v_i$  to itself.

A cycle is **simple** if the path is simple, except for the first and last vertices being the same.

## Graph Definitions (Cont)

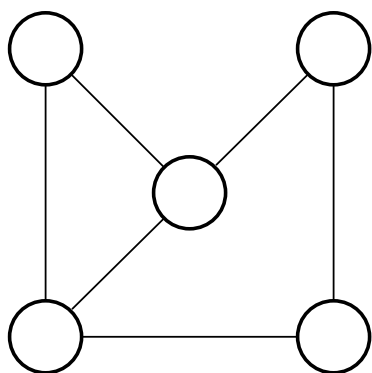
An undirected graph is **connected** if there is at least one path from any vertex to any other.

The maximal connected subgraphs of an undirected graph are called **connected components**.

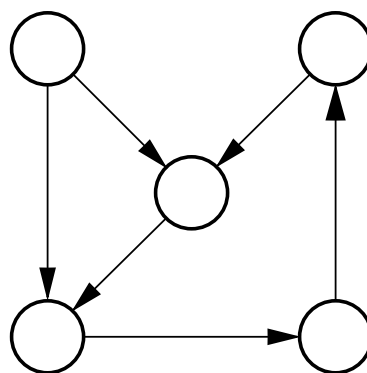
A graph without cycles is **acyclic**.

A directed graph without cycles is a **directed acyclic graph** or DAG.

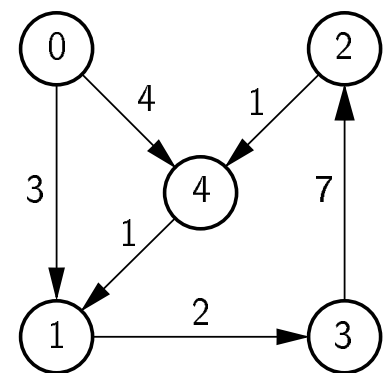
A **free tree** is a connected, undirected graph with no simple cycles. Equivalently, a free tree is connected and has  $|V - 1|$  edges.



(a)

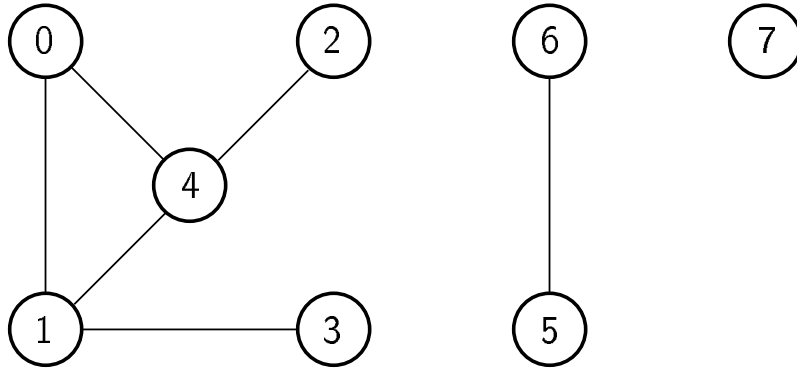


(b)



(c)

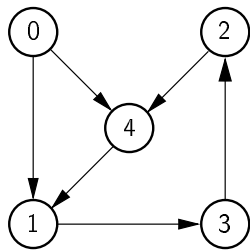
# Connected Components



# Graph Representations

Adjacency Matrix:  $\Theta(|V|^2)$ .

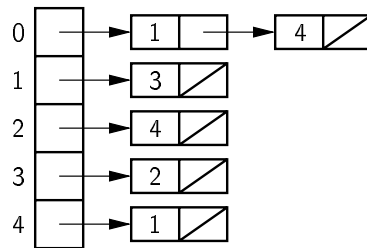
Adjacency List:  $\Theta(|V| + |E|)$ .



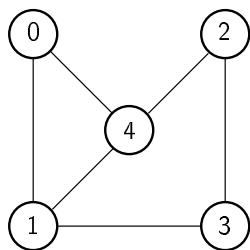
(a)

	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

(b)



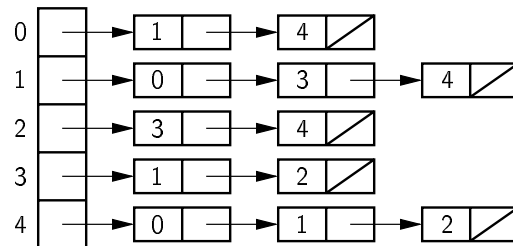
(c)



(a)

	0	1	2	3	4
0		1			1
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

(b)



(c)

# Graph Interface

```
interface Graph {                                // Graph class ADT
    public int n();                               // Number vertices
    public int e();                               // Number of edges
    public Edge first(int v);                     // Get first edge
    public Edge next(Edge w);                     // Get next edge
    public boolean isEdge(Edge w);                // True if edge
    public boolean isEdge(int i, int j);         // True if edge
    public int v1(Edge w);                        // Where from
    public int v2(Edge w);                        // Where to
    public void setEdge(int i, int j, int weight);
    public void setEdge(Edge w, int weight);
    public void delEdge(Edge w);                  // Delete edge w
    public void delEdge(int i, int j);           // Delete (i, j)
    public int weight(int i, int j);              // Return weight
    public int weight(Edge w);                    // Return weight
    public void setMark(int v, int val);         // Set Mark
    public int getMark(int v);                    // Get Mark
} // interface Graph
```

## Implementation: Edge Class

```
interface Edge { // Interface for graph edges
    public int v1(); // Return the vertex it comes from
    public int v2(); // Return the vertex it goes to
} // interface Edge

// Edge class for Adjacency Matrix graph representation
class Edgem implements Edge {
    private int vert1, vert2; // The vertex indices

    public Edgem(int vt1, int vt2)
        { vert1 = vt1; vert2 = vt2; }
    public int v1() { return vert1; }
    public int v2() { return vert2; }
} // class Edgem
```

# Implementation: Adjacency Matrix

```
class Graphm implements Graph { // Adjacency matrix
    private int[][] matrix;      // The edge matrix
    private int numEdge;        // Number of edges
    public int[] Mark;          // The mark array

    public Graphm(int n) {      // Constructor
        Mark = new int[n];
        matrix = new int[n][n];
        numEdge = 0;
    }

    public int n() { return Mark.length; }
    public int e() { return numEdge; }

    public Edge first(int v) { // Get first edge
        for (int i=0; i<Mark.length; i++)
            if (matrix[v][i] != 0)
                return new Edgem(v, i);
        return null; // No edge for this vertex
    }

    public Edge next(Edge w) { // Get next edge
        if (w == null) return null;
        for (int i=w.v2()+1; i<Mark.length; i++)
            if (matrix[w.v1()][i] != 0)
                return new Edgem(w.v1(), i);
        return null; // No next edge;
    }
}
```

## Adjacency Matrix (cont)

```
public boolean isEdge(Edge w) { // True if an edge
    if (w == null) return false;
    else return matrix[w.v1()][w.v2()] != 0;
}
```

```
public boolean isEdge(int i, int j) // True if edge
    { return matrix[i][j] != 0; }
```

```
public int v1(Edge w) {return w.v1();} // Where from
public int v2(Edge w) {return w.v2();} // Where to
```

```
public void setEdge(int i, int j, int wt) {
    Assert.notFalse(wt!=0, "Cannot set weight to 0");
    if (matrix[i][j] != 0) numEdge++;
    matrix[i][j] = wt;
}
```

```
public void setEdge(Edge w, int weight) // Set weight
    { if (w != null) setEdge(w.v1(), w.v2(), weight); }
```

```
public void delEdge(Edge w) { // Delete edge w
    if (w != null)
        if (matrix[w.v1()][w.v2()] != 0)
            { matrix[w.v1()][w.v2()] = 0; numEdge--; }
}
```

```
public void delEdge(int i, int j) { // Delete (i, j)
    if (matrix[i][j] != 0)
        { matrix[i][j] = 0; numEdge--; }
}
```

## Adjacency Matrix (cont 2)

```
public int weight(int i, int j) { // Return weight
    if (matrix[i][j] == 0) return Integer.MAX_VALUE;
    else return matrix[i][j];
}

public int weight(Edge w) { // Return edge weight
    Assert.notNull(w, "Can't take weight of null edge");
    if (matrix[w.v1()][w.v2()] == 0)
        return Integer.MAX_VALUE;
    else return matrix[w.v1()][w.v2()];
}

public void setMark(int v, int val)
    { Mark[v] = val; }

public int getMark(int v) { return Mark[v]; }
} // class Graphm
```

## Implementation: Edge Class

```
// Edge class for Adjacency List graph representation
class Edgel implements Edge {
    private int vert1, vert2;    // Indices of v1, v2
    private Link itself; // Pointer to node in adj list

    public Edgel(int vt1, int vt2, Link it) //Constructor
        { vert1 = vt1;  vert2 = vt2;  itself = it; }

    public int v1() { return vert1; }
    public int v2() { return vert2; }
    Link theLink() { return itself; } // Access adj list
} // class Edgel
```

# Implementation: Adjacency List

```
class Graph1 implements Graph { // Graph: Adjacency list
    private GraphList[] vertex; // The vertex list
    private int numEdge; // Number of edges
    public int[] Mark; // The mark array

    public Graph1(int n) { // Constructor
        Mark = new int[n];
        vertex = new GraphList[n];
        for (int i=0; i<n;i++) vertex[i] = new GraphList();
        numEdge = 0;
    }

    public Edge first(int v) { // Get first edge
        vertex[v].setFirst();
        if (vertex[v].currValue() == null) return null;
        return new Edgel(v,
            ((int[])vertex[v].currValue())[0],
            vertex[v].currLink());
    }

    public boolean isEdge(Edge e) { // True if an edge
        if (e == null) return false;
        vertex[e.v1()].setCurr(((Edgel)e).theLink());
        if (!vertex[e.v1()].isInList()) return false;
        return ((int[])vertex[e.v1()].currValue())[0]
            == e.v2();
    }

    public int v1(Edge e) { return e.v1(); }
    public int v2(Edge e) { return e.v2(); }
```

## Adjacency List (cont)

```
public boolean isEdge(int i, int j) { // True if edge
    GraphList temp = vertex[i];
    for (temp.setFirst(); ((temp.currValue()!=null) &&
        (((int[])temp.currValue())[0]<j)); temp.next());
    return (temp.currValue() != null) &&
        (((int[])temp.currValue())[0] == j);
}
```

```
public Edge next(Edge e) { // Get next vertex edge
    vertex[e.v1()].setCurr(((Edge1)e).theLink());
    vertex[e.v1()].next();
    if (vertex[e.v1()].currValue()==null) return null;
    return new Edge1(e.v1(),
        ((int[])vertex[e.v1()].currValue())[0],
        vertex[e.v1()].currLink());
}
```

```
public void setEdge(int i, int j, int weight) {
    Assert.notFalse(weight!=0, "Cannot set to 0");
    int[] currEdge = { j, weight };
    if (isEdge(i, j)) vertex[i].setValue(currEdge);
    else // Add new edge to graph
        { vertex[i].insert(currEdge); numEdge++; }
}
```

```
public void setEdge(Edge w, int weight) // Set weight
    { if (w != null) setEdge(w.v1(), w.v2(), weight); }
```

## Adjacency List (cont 2)

```
public int weight(int i, int j) { // Return weight
    if (isEdge(i, j))
        return ((int[])vertex[i].currValue())[1];
    else return Integer.MAX_VALUE;
}

public int weight(Edge e) { // Return edge weight
    if (isEdge(e))
        return ((int[])vertex[e.v1()].currValue())[1];
    else return Integer.MAX_VALUE;
}
} // class Graph1
```

# Graph Traversals

Some applications require visiting every vertex in the graph exactly once.

Application may require that vertices be visited in some special order based on graph topology.

Example: Artificial Intelligence

- Problem domain consists of many “states.”
- Need to get from Start State to Goal State.
- Start and Goal are typically not directly connected.

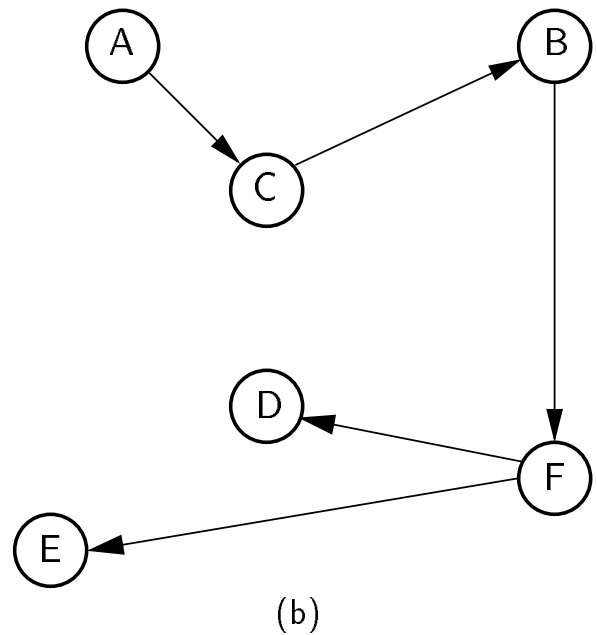
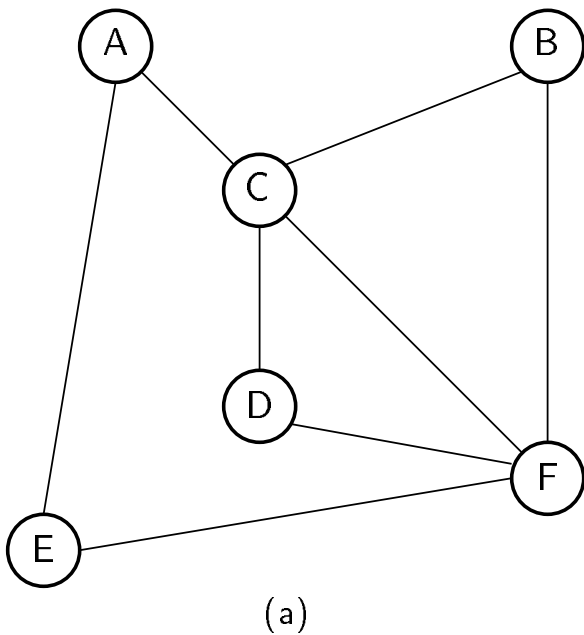
To insure visiting all vertices:

```
void graphTraverse(Graph G) {
    for (v=0; v<G.n(); v++)
        G.setMark(v, UNVISITED); // Initialize mark bits
    for (v=0; v<G.n(); v++)
        if (G.getMark(v) == UNVISITED)
            doTraverse(G, v);
}
```

# Depth First Search

```
static void DFS(Graph G, int v) { // Depth first search
    PreVisit(G, v);           // Take appropriate action
    G.setMark(v, VISITED);
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == UNVISITED)
            DFS(G, G.v2(w));
    PostVisit(G, v);         // Take appropriate action
}
```

Cost:  $\Theta(|V| + |E|)$ .

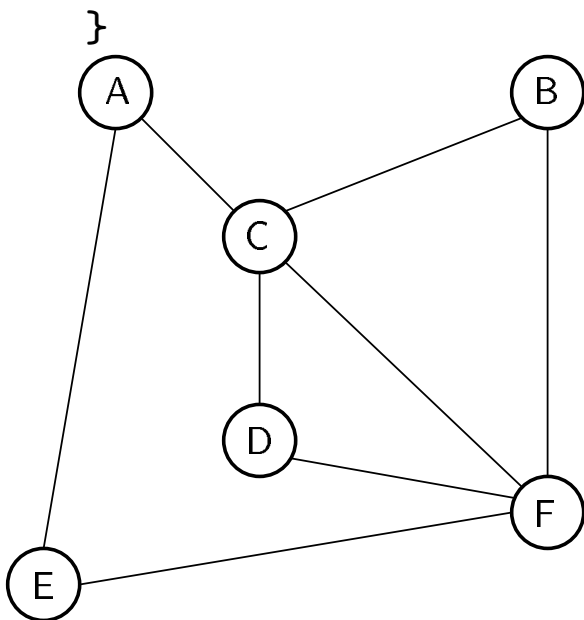


# Breadth First Search

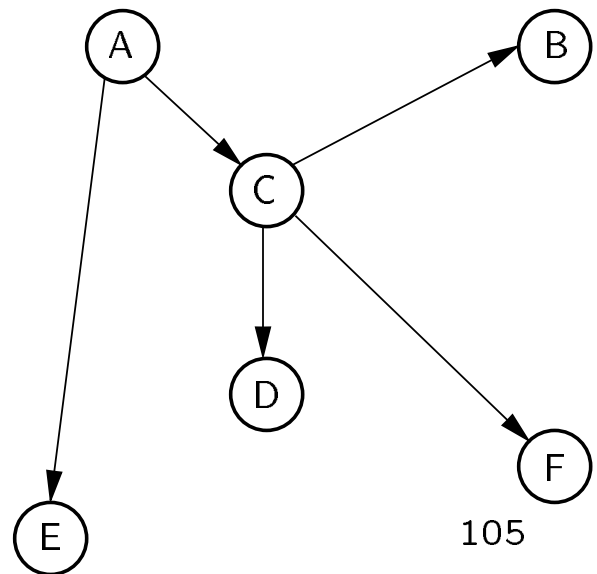
Like DFS, but replace stack with a queue.

Visit the vertex's neighbors before continuing deeper in the tree.

```
static void BFS(Graph G, int start) {
    Queue Q = new AQueue(G.n());           // Use a Queue
    Q.enqueue(new Integer(start));
    G.setMark(start, VISITED);
    while (!Q.isEmpty()) { // Process each vertex on Q
        int v = ((Integer)Q.dequeue()).intValue();
        PreVisit(G, v);           // Take appropriate action
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
            if (G.getMark(G.v2(w)) == UNVISITED) {
                G.setMark(G.v2(w), VISITED);
                Q.enqueue(new Integer(G.v2(w)));
            }
        PostVisit(G, v);         // Take appropriate action
    }
}
```



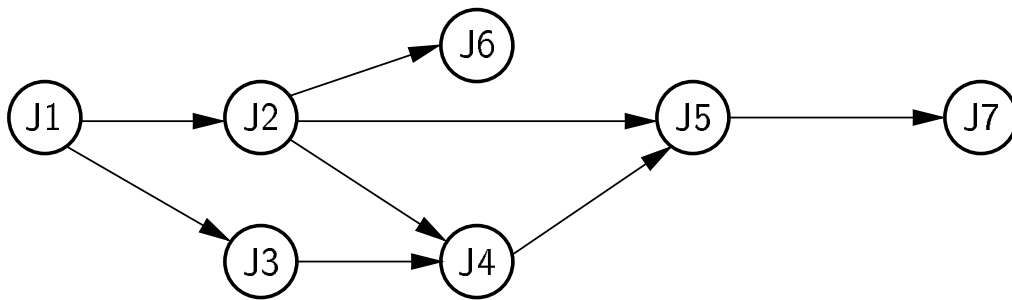
(a)



(b)

# Topological Sort

Problem: Given a set of jobs, courses, etc. with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.



```
static void topsort(Graph G) { // Topo sort: recursive
    for (int i=0; i<G.n(); i++) // Initialize Mark array
        G.setMark(i, UNVISITED);
    for (int i=0; i<G.n(); i++) // Process all vertices
        if (G.getMark(i) == UNVISITED)
            tophelp(G, i); // Call helper function
}
```

```
static void tophelp(Graph G, int v) { // Topsort helper
    G.setMark(v, VISITED);
    for (Edge w = G.first(v); G.isEdge(w); w = G.next(w))
        if (G.getMark(G.v2(w)) == UNVISITED)
            tophelp(G, G.v2(w));
    printout(v); // PostVisit for Vertex v
}
```

# Queue-based Topological Sort

```
static void topsort(Graph G) { // Topo sort: Queue
    Queue Q = new AQueue(G.n());
    int[] Count = new int[G.n()];
    int v;
    for (v=0; v<G.n(); v++) Count[v] = 0; // Initialize
    for (v=0; v<G.n(); v++)           // Process every edge
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
            Count[G.v2(w)]++;           // Add to v2's count
    for (v=0; v<G.n(); v++)           // Initialize Queue
        if (Count[v] == 0)             // Vertex has no prereqs
            Q.enqueue(new Integer(v));
    while (!Q.isEmpty()) {             // Process the vertices
        v = ((Integer)Q.dequeue()).intValue();
        printout(v);                   // PreVisit for Vertex V
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w)) {
            Count[G.v2(w)]--;           // One less prerequisite
            if (Count[G.v2(w)] == 0) // This vertex now free
                Q.enqueue(new Integer(G.v2(w)));
        }
    }
}
```

# Shortest Paths Problems

Input: A graph with weights or costs associated with each edge.

Output: The list of edges forming the shortest path.

Sample problems:

- Find the shortest path between two specified vertices.
- Find the shortest path from vertex  $S$  to all other vertices.
- Find the shortest path between all pairs of vertices.

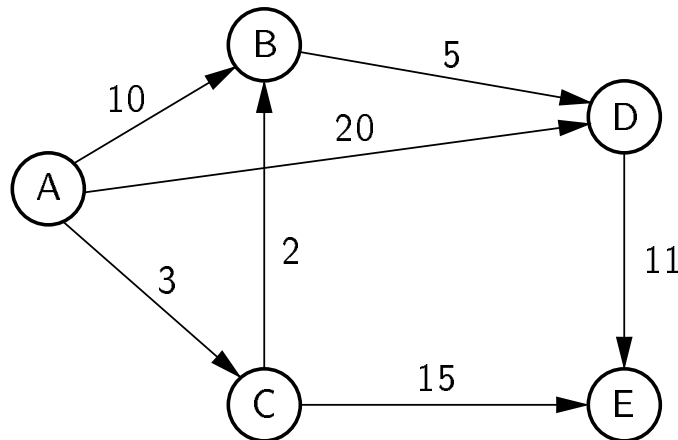
Our algorithms will actually calculate only distances.

# Shortest Paths Definitions

$d(A, B)$  is the shortest distance from vertex A to B.

$w(A, B)$  is the weight of the edge connecting A to B.

- If there is no such edge, then  $w(A, B) = \infty$ .



# Single Source Shortest Paths

Given start vertex  $s$ , find the shortest path from  $s$  to all other vertices.

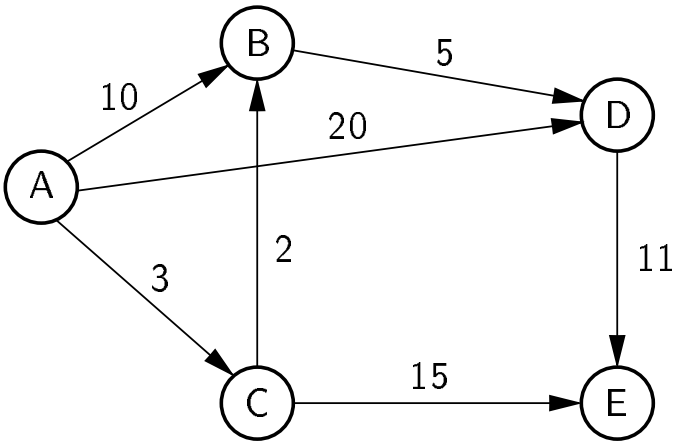
Try 1: Visit all vertices in some order, compute shortest paths for all vertices seen so far, then add the shortest path to next vertex  $x$ .

Problem: Shortest path to a vertex already processed might go through  $x$ .

Solution: Process vertices in order of distance from  $s$ .

# Dijkstra's Algorithm Example

	A	B	C	D	E
Initial	0	$\infty$	$\infty$	$\infty$	$\infty$
Process A	0	10	3	20	$\infty$
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18



## Dijkstra's Algorithm: Array

```
// Compute shortest path distances from s, store in D
static void Dijkstra(Graph G, int s, int[] D) {
    for (int i=0; i<G.n(); i++) // Initialize
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) { // Process vertices
        int v = minVertex(G, D); // Get next-closest vertex
        G.setMark(v, VISITED);
        if (D[v] == Integer.MAX_VALUE) return;
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > (D[v] + G.weight(w)))
                D[G.v2(w)] = D[v] + G.weight(w);
    }
}
```

```
static int minVertex(Graph G, int[] D) {
    int v = 0; // Initialize v to any unvisited vertex;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED) { v = i; break; }
    for (int i=0; i<G.n(); i++) // Find smallest value
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
            v = i;
    return v;
}
```

Approach 1: Scan the table on each pass for closest vertex.

Total cost:  $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ .

# Dijkstra's Algorithm: Priority Queue

```
// Dijkstra's shortest-paths algorithm:
//   priority queue version
static void Dijkstra(Graph G, int s, int[] D) {
    int v; // The current vertex
    DijkElem[] E = new DijkElem[G.e()]; // Heap
    E[0] = new DijkElem(s, 0); // Initialize array
    MinHeap H = new MinHeap(E, 1, G.e()); // Create heap
    for (int i=0; i<G.n(); i++) // Initialize distances
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) { // For each vertex
        do { v = ((DijkElem)H.removemin()).vertex(); }
        while (G.getMark(v) == VISITED);
        G.setMark(v, VISITED);
        if (D[v] == Integer.MAX_VALUE) return;
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > (D[v] + G.weight(w))) {
                D[G.v2(w)] = D[v] + G.weight(w);
                H.insert(new DijkElem(G.v2(w), D[G.v2(w)]));
            }
        }
    }
}
```

Approach 2: Store unprocessed vertices using a min-heap to implement a priority queue ordered by D value. Must update priority queue for each edge.

Total cost:  $\Theta((|V| + |E|) \log |V|)$ .

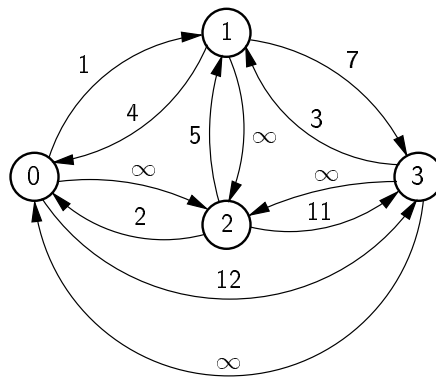
# All Pairs Shortest Paths

For every vertex  $u, v \in V$ , calculate  $d(u, v)$ .

Could run Dijkstra's Algorithm  $V$  times.

Better is Floyd's Algorithm.

Define a k-path from  $u$  to  $v$  to be any path whose intermediate vertices all have indices less than  $k$ .



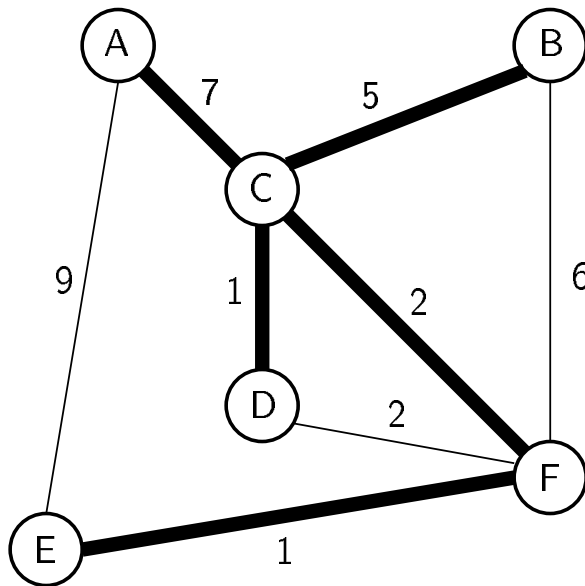
# Floyd's Algorithm

```
// Compute all-pairs shortest paths
static void Floyd(Graph G, int[][] D) {
    for (int i=0; i<G.n(); i++) // Initialize D
        for (int j=0; j<G.n(); j++)
            D[i][j] = G.weight(i, j);
    for (int k=0; k<G.n(); k++) // Compute all k paths
        for (int i=0; i<G.n(); i++)
            for (int j=0; j<G.n(); j++)
                if ((D[i][k] != Integer.MAX_VALUE) &&
                    (D[k][j] != Integer.MAX_VALUE) &&
                    (D[i][j] > (D[i][k] + D[k][j])))
                    D[i][j] = D[i][k] + D[k][j];
}
```

# Minimum Cost Spanning Trees

Minimum Cost Spanning Tree (MST) Problem:

- Input: An undirected, connected graph  $G$ .
- Output: The subgraph of  $G$  that 1) has minimum total cost as measured by summing the values for all of the edges in the subset, and 2) keeps the vertices connected.



# Prim's MST Algorithm

```
// Compute a minimal-cost spanning tree
static void Prim(Graph G, int s, int[] D) {
    int[] V = new int[G.n()]; // V[i] closest to i
    for (int i=0; i<G.n(); i++) // Initialize
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) { // Process vertices
        int v = minVertex(G, D);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v);
        if (D[v] == Integer.MAX_VALUE) return;
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > G.weight(w)) {
                D[G.v2(w)] = G.weight(w);
                V[G.v2(w)] = v;
            }
        }
    }
}
```

This is an example of a greedy algorithm.

# Alternative Prim's Implementation

Like Dijkstra's algorithm, we can implement Prim's algorithm with a priority queue.

```
// Prim's MST algorithm: priority queue version
static void Prim(Graph G, int s, int[] D) {
    int v; // The current vertex
    int[] V = new int[G.n()]; // V[i] closest to i
    DijkElem[] E = new DijkElem[G.e()]; // Heap
    E[0] = new DijkElem(s, 0); // Initialize array
    MinHeap H = new MinHeap(E, 1, G.e()); // Create heap
    for (int i=0; i<G.n(); i++) // Initialize dist array
        D[i] = Integer.MAX_VALUE;
    D[s] = 0;
    for (int i=0; i<G.n(); i++) { // Now, get distances
        do { v = ((DijkElem)H.removemin()).vertex(); }
            while (G.getMark(v) == VISITED);
        G.setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v); // Add MST edge
        if (D[v] == Integer.MAX_VALUE) return;
        for (Edge w=G.first(v); G.isEdge(w); w=G.next(w))
            if (D[G.v2(w)] > G.weight(w)) { // Update D
                D[G.v2(w)] = G.weight(w);
                V[G.v2(w)] = v; // Update who it came from
                H.insert(new DijkElem(G.v2(w), D[G.v2(w)]));
            }
        }
    }
}
```

# Proof of Prim's MST Algorithm

**Theorem 14.1** *Prim's algorithm produces a minimum cost spanning tree.*

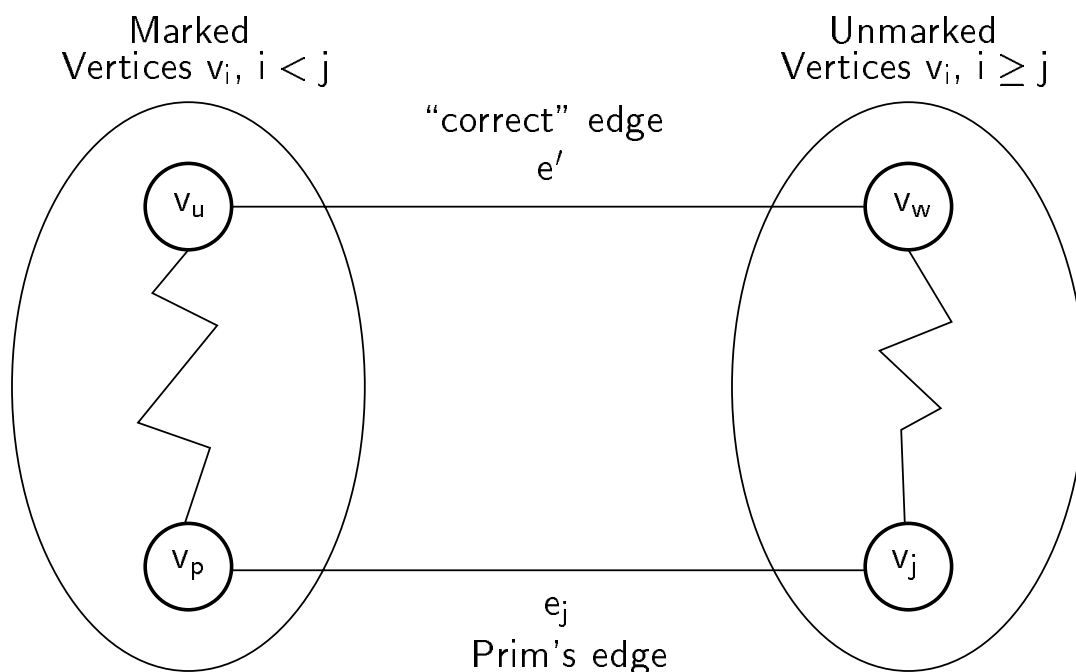
Proof by contradiction:

Order vertices by how they are added to the MST by Prim's algorithm.  $v_1, v_2, \dots, v_n$ .

Let edge  $e_i$  connect  $(v_x, v_{i+1})$ ,  $x < i$ .

Let  $e_j$  be the lowest numbered (first) edge added by the algorithm such that the set of edges selected so far *cannot* be extended to form an MST for  $G$ .

Let  $V_1 = (v_1, \dots, v_j)$ . Let  $V_2 = (v_{j+1}, \dots, v_n)$ .



# Kruskal's MST Algorithm

```
static void Kruskal(Graph G) { // Kruskal's MST alg
    GenTree A = new GenTree(G.n());
    KruskalElem[] E = new KruskalElem[G.e()];
    int edgecnt = 0; // Count of edges

    for (int i=0; i<G.n(); i++) // Put edges on array
        for (Edge w=G.first(i); G.isEdge(w); w=G.next(w))
            E[edgecnt++] = new KruskalElem(G, w);
    MinHeap H = new MinHeap(E, edgecnt, edgecnt);
    int numMST = G.n(); // Initially n equivs
    for (int i=0; numMST>1; i++) { // Combine equivs
        KruskalElem temp = (KruskalElem)H.removemin();
        Edge w = temp.edge();
        int v = G.v1(w); int u = G.v2(w);
        if (A.differ(v, u)) { // If different equivs
            A.UNION(v, u); // combine equiv classes
            AddEdgetoMST(G.v1(w), G.v2(w)); // Add MST edge
            numMST--; // One less MST
        }
    }
}
```

How do we compute function  $MSTof(v)$ ?

Solution: Use Parent Pointer representation to merge equivalence classes.

# Kruskal's Algorithm Example

Time dominated by cost for initial edge sort.

Total cost:  $\Theta(|V| + |E| \log |E|)$ .

