CMSC 341

Graphs

Basic Graph Definitions

A ***graph*** $G = (V,E)$ consists of a finite set of ***vertices***, V, and a finite set of ***edges***, E.

Each edge is a pair (v,w) where $v, w \in V$.

- V and E are sets, so each vertex $v \in V$ is unique, and each edge $e \in E$ is unique.
- Edges are sometimes called ***arcs*** or ***lines***.
- Vertices are sometimes called ***nodes*** or ***points***.

Graph Applications

Graphs can be used to model a wide range of applications including

- Intersections and streets within a city
- Roads/trains/airline routes connecting cities/countries
- Computer networks
- Electronic circuits

Basic Graph Definitions (2)

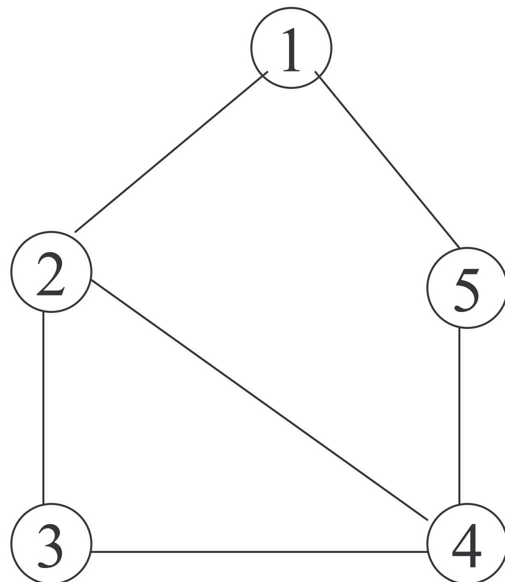
A ***directed graph*** is a graph in which the edges are ordered pairs.
That is, $(u,v) \neq (v,u)$, $u, v \in V$.
Directed graphs are sometimes called ***digraphs***.

An ***undirected graph*** is a graph in which the edges are unordered pairs.
That is, $(u,v) = (v,u)$.

A ***sparse graph*** is one with “few” edges.
That is $|E| = O(|V|)$

A ***dense graph*** is one with “many” edges.
That is $|E| = O(|V|^2)$

Undirected Graph

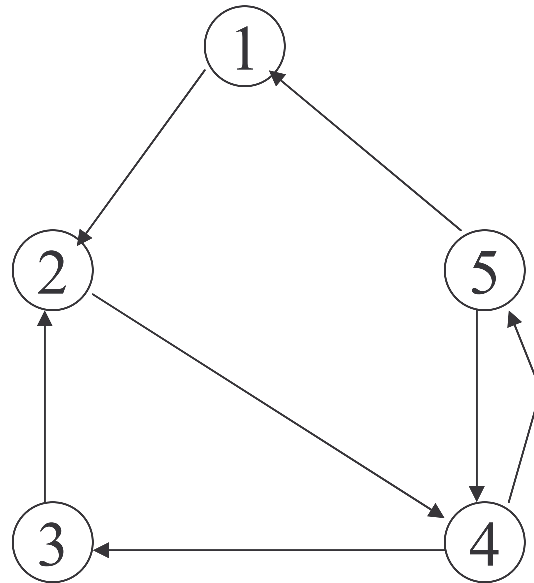


All edges are two-way. Edges are unordered pairs.

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

E = { (1,2), (2, 3), (3, 4), (2, 4), (4, 5), (5, 1) }

Directed Graph

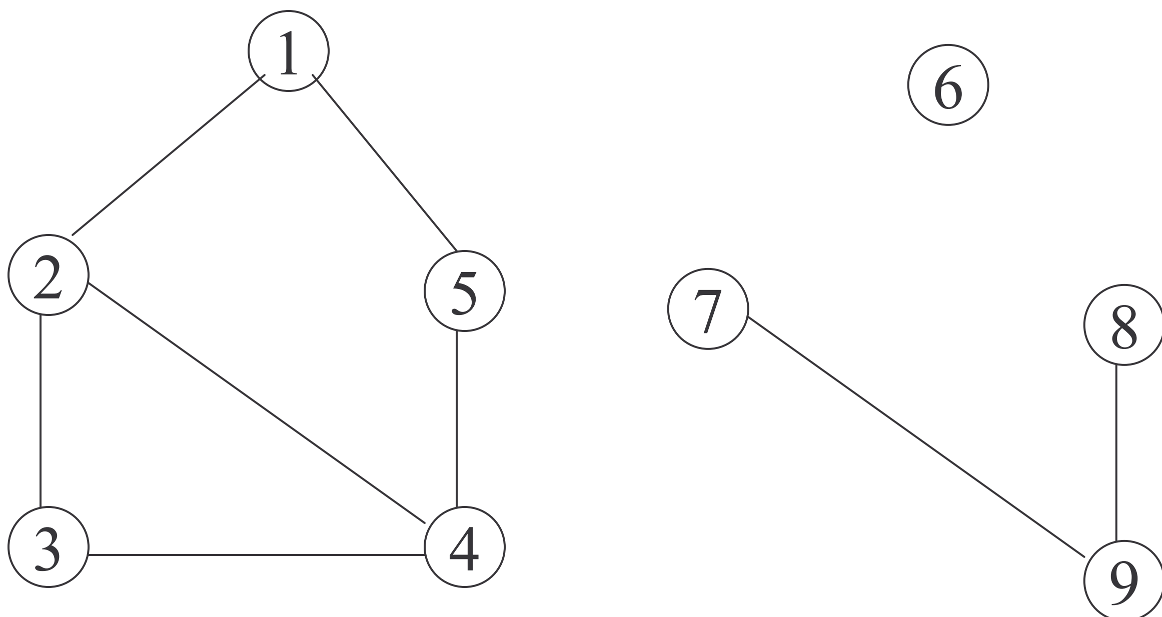


All edges are “one-way” as indicated by the arrows.
Edges are ordered pairs.

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

E = { (1, 2), (2, 4), (3, 2), (4, 3), (4, 5), (5, 4), (5, 1) }

A Single Graph with Multiple Components



Basic Graph Definitions (3)

Vertex w is ***adjacent to*** vertex v if and only if (v, w) ∈ E.

For undirected graphs, with edge (v, w), and hence also (w, v), w is adjacent to v and v is adjacent to w.

An edge may also have:

- **weight** or **cost** -- an associated value
- **label** -- a unique name

The ***degree*** of a vertex, v, is the number of vertices adjacent to v. Degree is also called *valence*.

Paths in Graphs

A **path** in a graph is a sequence of vertices $w_1, w_2, w_3, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$.

The **length** of a path in a graph is the number of edges on the path. The length of the path from a vertex to itself is 0.

A **simple path** is a path such that all vertices are distinct, except that the first and last may be the same.

A **cycle** in a graph is a path $w_1, w_2, w_3, \ldots, w_n$, $w \in V$ such that:

- there are at least two vertices on the path
- $w_1 = w_n$ (the path starts and ends on the same vertex)
- if any part of the path contains the subpath w_i, w_j, w_i, then each of the edges in the subpath is distinct (i. e., no backtracking along the same edge)

A **simple cycle** is one in which the path is simple.

A directed graph with no cycles is called a **directed acyclic graph**, often abbreviated as DAG

Connectedness in Graphs

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

A directed graph is **strongly connected** if there is a path from every vertex to every other vertex.

A directed graph is **weakly connected** if there would be a path from every vertex to every other vertex, disregarding the direction of the edges.

A ***c*omplete** graph is one in which there is an edge between every pair of vertices.

A **connected component** of a graph is any maximal connected subgraph. Connected components are sometimes simply called **components**.

Disjoint Sets and Graphs

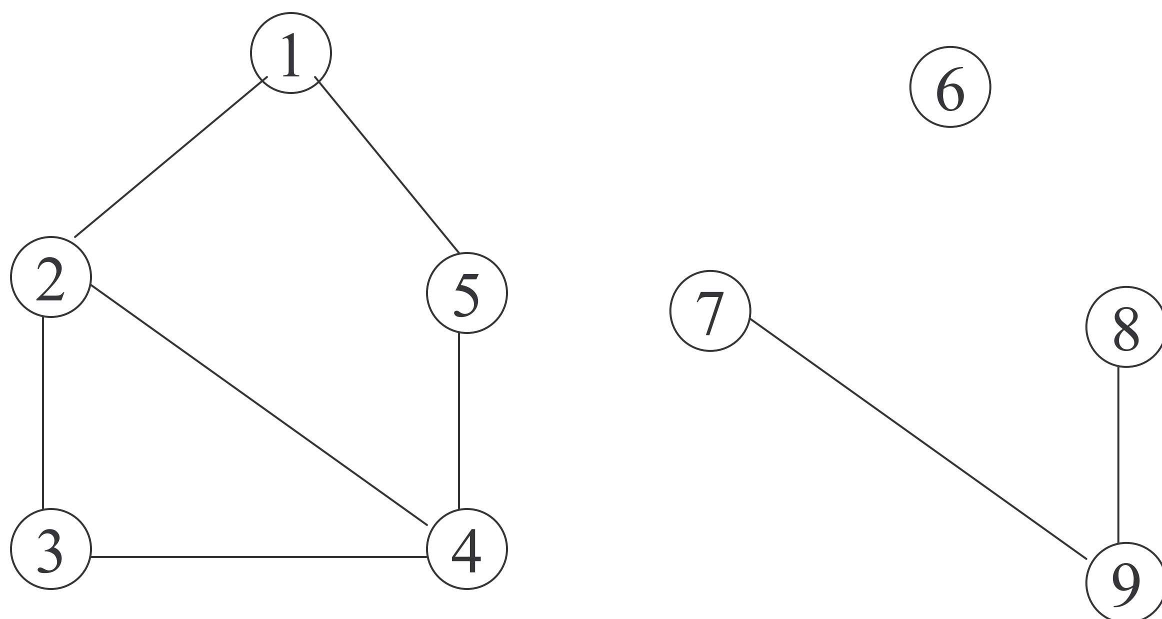
Disjoint sets can be used to determine connected components of an undirected graph.

For each edge, place its two vertices (u and v) in the same set -- i.e. Union(u, v)

When all edges have been examined, the forest of sets will represent the connected components.

Two vertices, x, y, are connected if and only if Find(x) = Find(y)

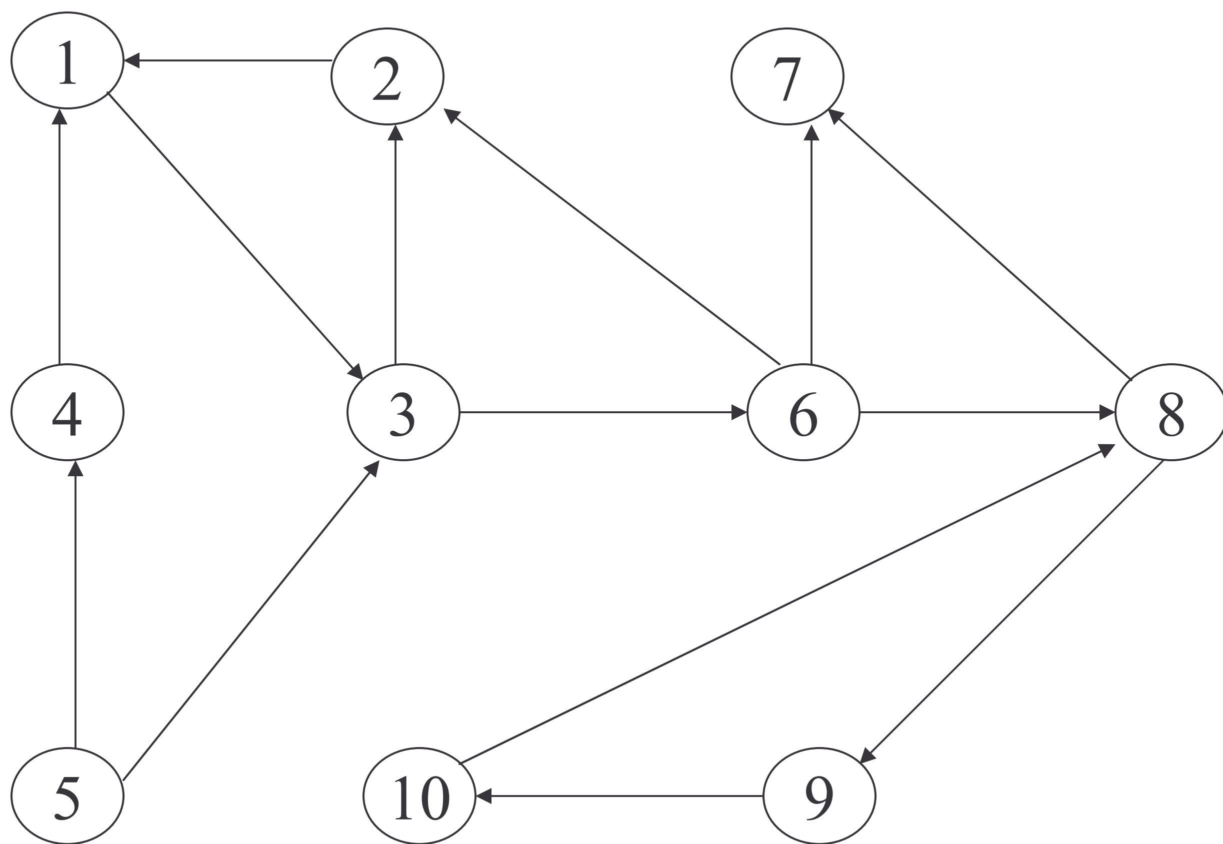
Undirected Graph/Disjoint Set Example



Sets representing connected components

{ 1, 2, 3, 4, 5 }
{ 6 }
{ 7, 8, 9 }

DiGraph / Strongly Connected Components



A Graph ADT

Has some data elements

- Vertices and Edges

Has some operations

- GetDegree(u) -- returns the degree of vertex u
- GetAdjacent(u) -- returns a list of the vertices **adjacent to** vertex u
- IsAdjacentTo (u, v) -- returns TRUE if vertex v is adjacent to vertex u, FALSE otherwise

Has some associated algorithms to be discussed.

Adjacency Matrix Implementation

Uses array of size $|V| \times |V|$ where each entry (i ,j) is boolean

- TRUE if there is an edge from vertex i to vertex j
- FALSE otherwise
- store weights when edges are weighted

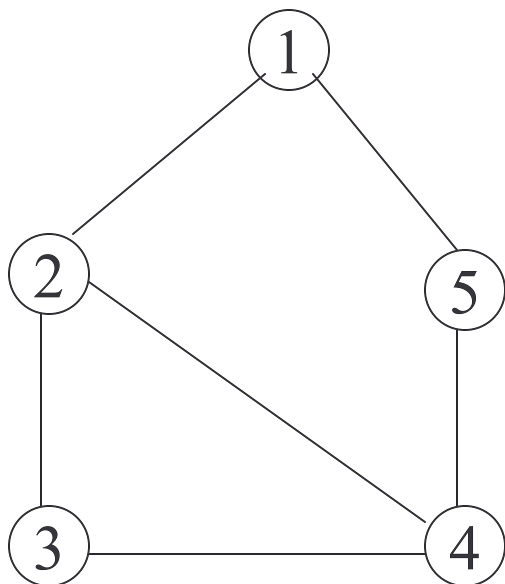
Very simple, but large space requirement = $O(|V|^2)$

Appropriate if the graph is dense.

Otherwise, most of the entries in the table are FALSE.

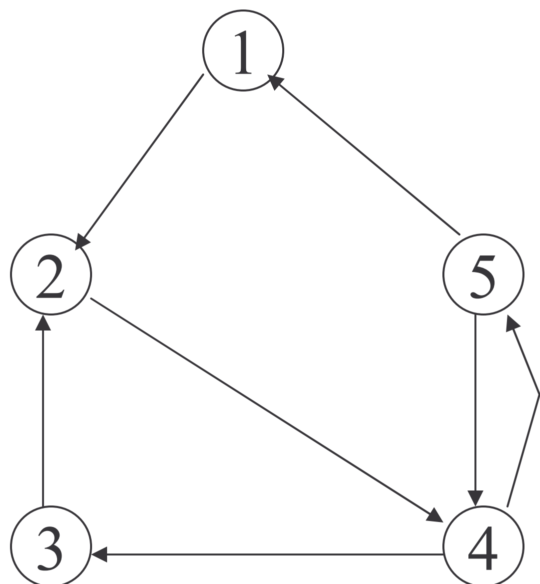
For example, if a graph is used to represent a street map like Manhattan in which most streets run E/W or N/S, each intersection is attached to only 4 streets and $|E| < 4*|V|$. If there are 3000 intersections, the table has 9,000,000 entries of which only 12,000 are TRUE.

Undirected Graph / Adjacency Matrix



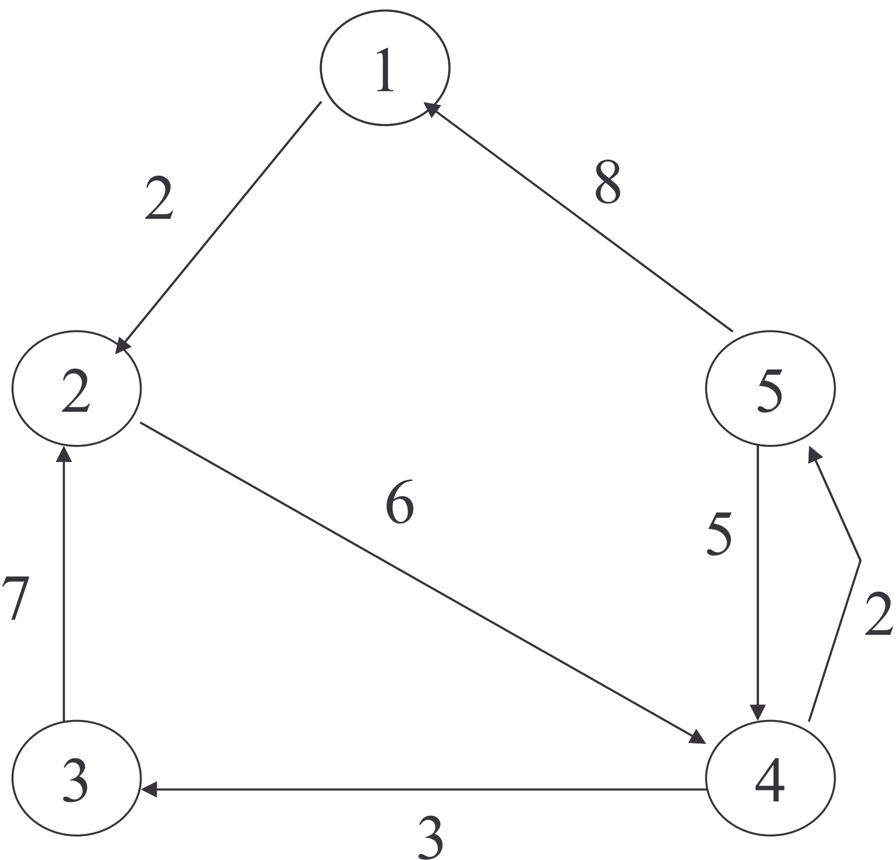
	1	**2**	**3**	**4**	**5**
1	0	1	0	0	1
2	1	0	1	1	0
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	0

Directed Graph / Adjacency Matrix



	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	1
5	1	0	0	1	0

Weighted, Directed Graph / Adjacency Matrix



	1	**2**	**3**	**4**	**5**
1	0	2	0	0	0
2	0	0	0	6	0
3	0	7	0	0	0
4	0	0	3	0	2
5	8	0	0	5	0

Adjacency Matrix Performance

Storage requirement: $O(|V|^2)$

Performance:

GetDegree (u)	
IsAdjacentTo(u, v)	
GetAdjacent(u)	

Adjacency List Implementation

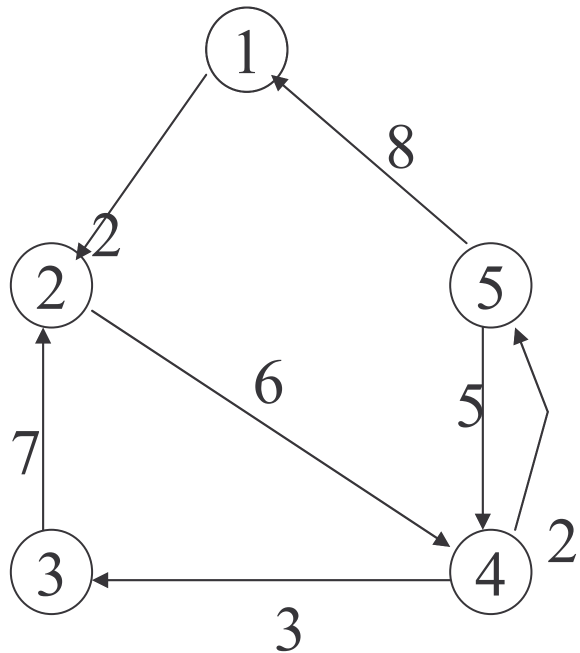
If the graph is sparse, then keeping a list of adjacent vertices for each vertex saves space. Adjacency Lists are the commonly used representation. The lists may be stored in a data structure or in the Vertex object itself.

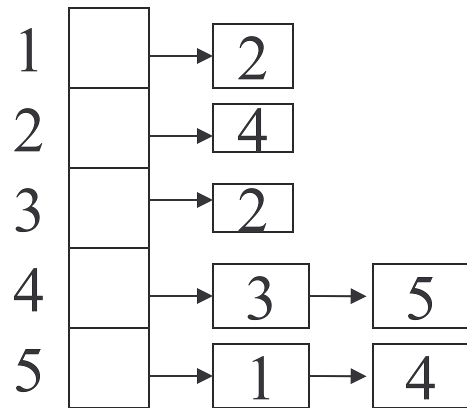
- **Vector of lists**: A vector of lists of vertices. The i-th element of the vector is a list, $L_{i,}$ of the vertices adjacent to v_i.

If the graph is sparse, then the space requirement is $O(|E| + |V|)$, "linear in the size of the graph"

If the graph is dense, then the space requirement is $O(|V|^2)$

Vector of Lists





Adjacency List Performance

Storage requirement:

Performance:

GetDegree(u)	
IsAdjacentTo(u, v)	
GetAdjacent(u)	

Graph Traversals

Like trees, graphs can be traversed breadth-first or depth-first.

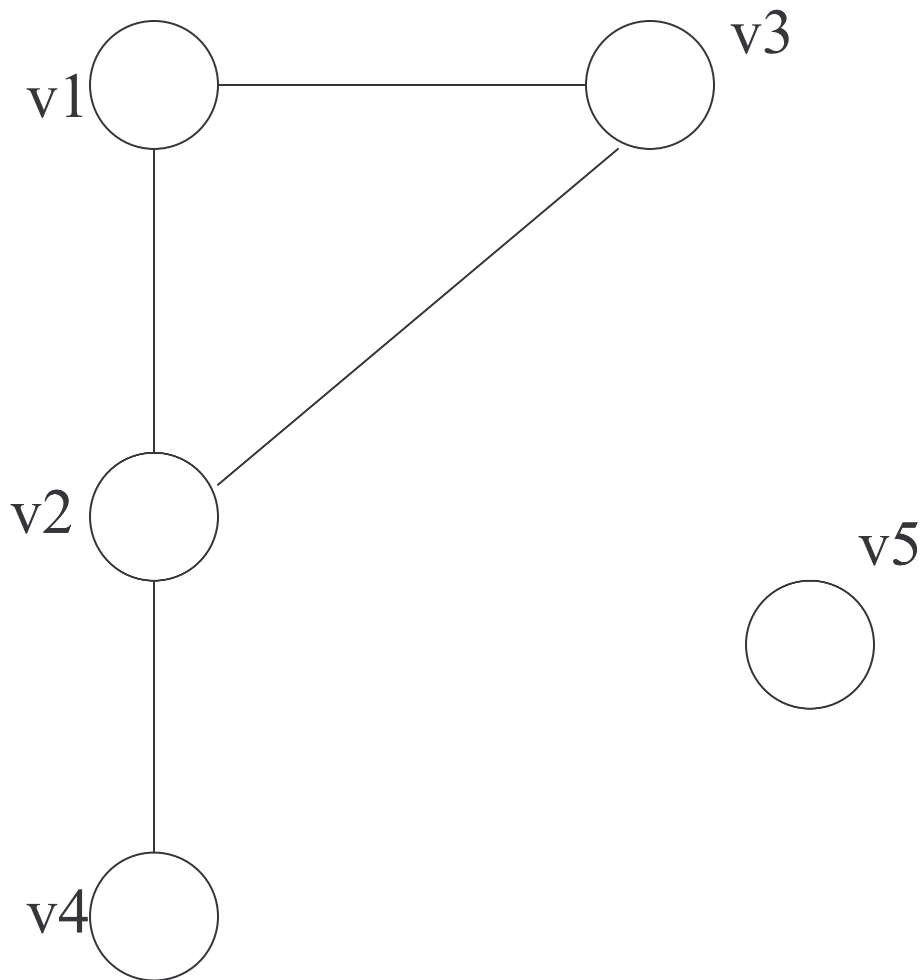
- Use stack (or recursion) for depth-first traversal
- Use queue for breadth-first traversal

Unlike trees, we need to specifically guard against repeating a path from a cycle. Mark each vertex as “visited” when we encounter it and do not consider visited vertices more than once.

Breadth-First Traversal

```
Queue<Vertex> q;
Vertex u, w;

for all v, d[v] = ∞          // mark each vertex unvisited
q.enqueue(startvertex);      // start with any vertex
d[startvertex] = 0;          // mark visited
while ( !q.isEmpty() ) {
   u = q.dequeue( );
   for (each vertex, w, that is adjacent to u)
        if (d[w] == ∞) {     // w not marked as visited
            d[w] = d[u]+1;   // mark visited
            path[w] = u;     // where we came from
            q.enqueue(w);
        }
   }
}
```

Breadth-First Example



Unweighted Shortest Path Problem

Unweighted shortest-path problem: Given as input an unweighted graph, $G = (V, E)$, and a distinguished starting vertex, s, find the shortest unweighted path from s to every other vertex in G.

After running BFS algorithm with s as starting vertex, the length of the shortest path length from s to i is given by d[i]. If $d[i] = \infty$, then there is no path from s to i. The path from s to i is given by traversing path[] backwards from i back to s.

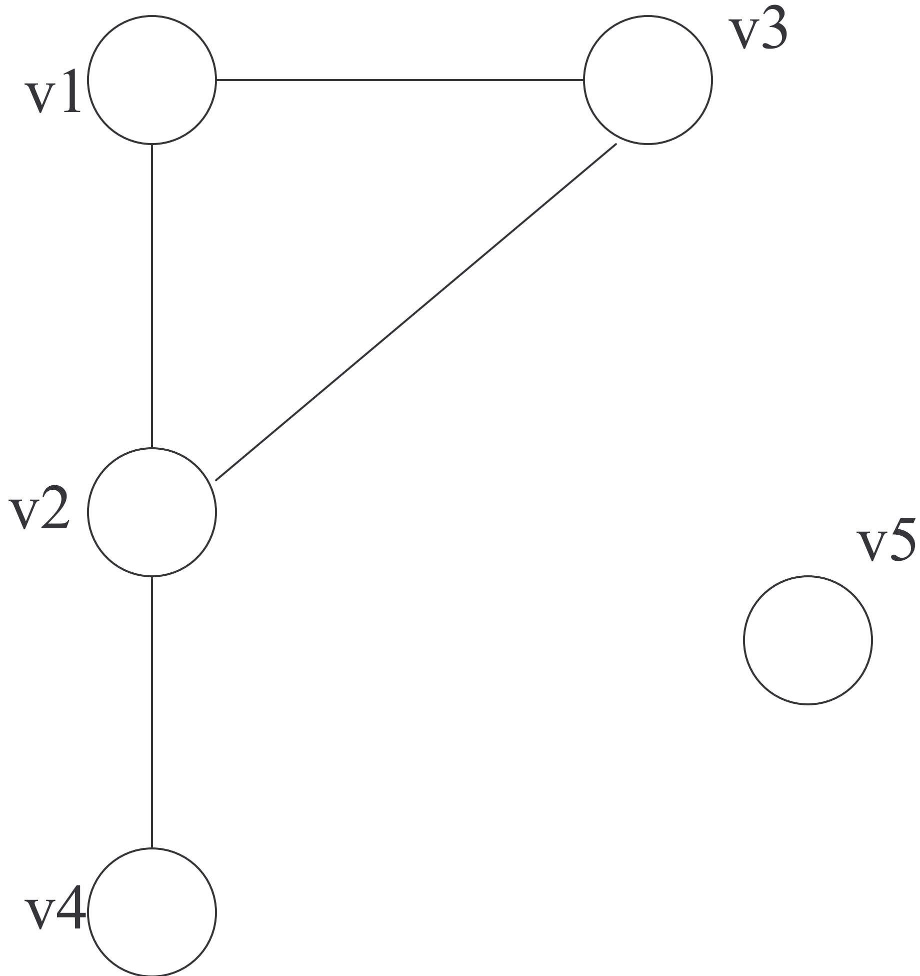
Recursive Depth First Traversal

```
dfs(Graph G) {
   for (each v ∈ V)
        dfs(v)
}

dfs(Vertex v)
{
   if (v is not marked as visited)
   {
        MarkVisited(v);
        for(each vertex, w, that is  adjacent to v)
              if ( w is not marked as visited )
                    dfs(w)
   }
}
```

DFS with explicit stack

```
dfs( Graph G )
{
   Stack<Vertex> s;
   Vertex u, w;

   s.push(startvertex);
   MarkVisited(startvertex);
   while ( !s.isEmpty() ) {
        u = s.Pop();
        for (each vertex, w, that is adjacent to u)
                if (w is not marked as visited) {
                        MarkVisited(w);
                        s.push(w);
        }
   }
}
```

DFS Example



Traversal Performance

What is the performance of DF and BF traversal?

Each vertex appears in the stack or queue exactly once in the worst case. Therefore, the traversals are at least $O(|V|)$. However, at each vertex, we must find the adjacent vertices. Therefore, df- and bf-traversal performance depends on the performance of the `GetAdjacent` operation.

GetAdjacent

Method 1: Look at every vertex (except u), asking "are you adjacent to u?"

```
List<Vertex> L;
for (each vertex v, except u)
  if (IsAdjacentTo(u,v))
   L.push_back(v);
```

Assuming O(1) performance for `push_back` and `IsAdjacentTo`, then `GetAdjacent` has $O(|V|)$ performance and traversal performance is $O(|V^2|)$;

GetAdjacent (2)

Method 2: Look only at the edges which impinge on u. Therefore, at each vertex, the number of vertices to be looked at is D(u), the degree of the vertex

This approach is O(D(u)). The traversal performance is

$$O\left(\sum_{i=1}^{|V|} D(v_i)\right) = O(|E|)$$

since `GetAdjacent` is done O(|V|) times.

However, in a disconnected graph, we must still look at every vertex, so the performance is O(|V| + |E|).

Number of Edges

Theorem: The number of edges in an undirected graph $G = (V,E)$ is $O(|V|^2)$

Proof: Suppose G is fully connected. Let $p = |V|$.

Then we have the following situation:

vertex	connected to
1	2,3,4,5,…, p
2	1,3,4,5,…, p
…	
p	1,2,3,4,…,p-1

– There are $p(p-1)/2 = O(|V|^2)$ edges.

So $O(|E|) = O(|V|^2)$.

Weighted Shortest Path Problem

Single-source shortest-path problem:

Given as input a weighted graph, $G = (V, E)$, and a distinguished starting vertex, s, find the shortest weighted path from s to every other vertex in G.

Use Dijkstra's algorithm

- keep tentative distance for each vertex giving shortest path length using vertices visited so far
- Record vertex visited before this vertex (to allow printing of path)
- at each step choose the vertex with smallest distance among the unvisited vertices (greedy algorithm)

Dijkstra's Algorithm

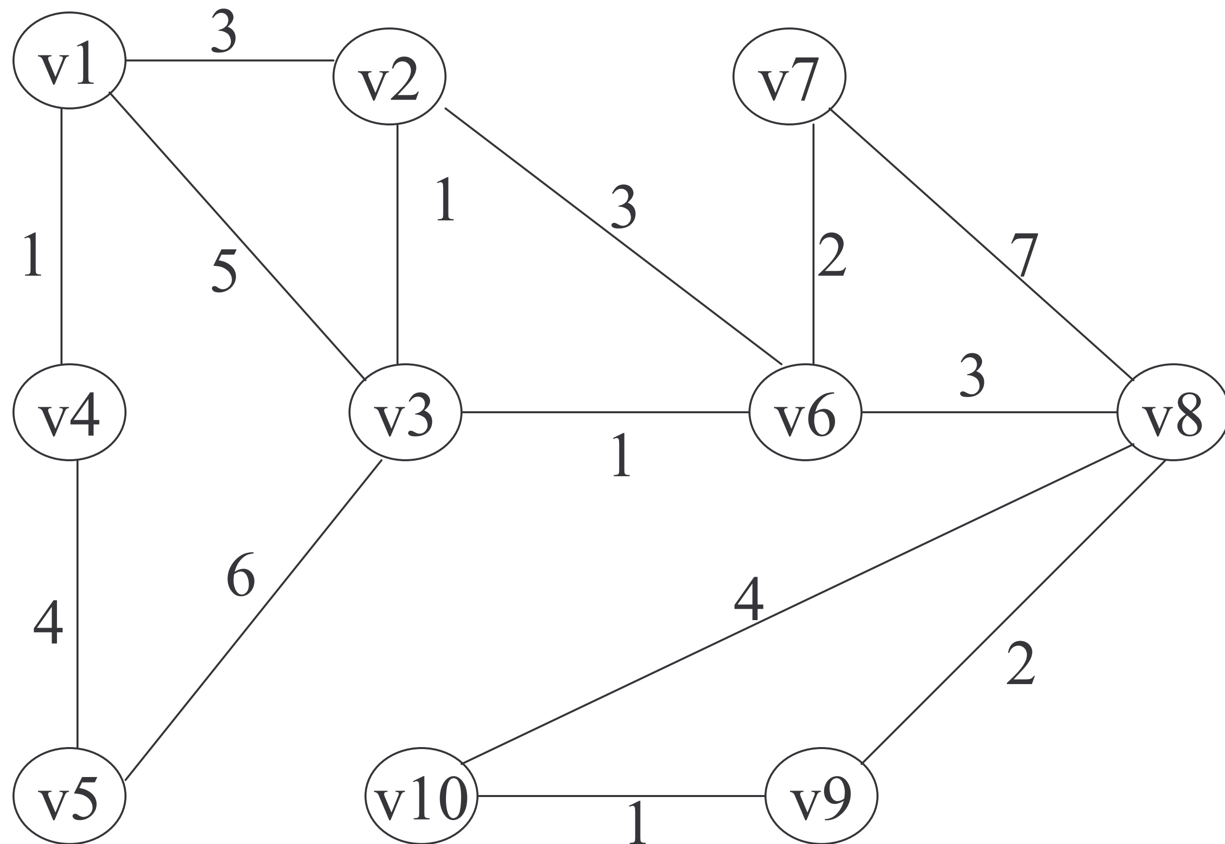
The pseudo code for Dijkstra's algorithm assumes the following structure for a Vertex object

```
struct Vertex
{
  List      adjacent;  // adjacency list
  bool      known;     // specific for Dijkstra
  int       distance;
  Vertex    path;      // our predecessor
};
```

Dijkstra's Algorithm

```
Vertex v, w;

For all vertices, v
  v.distance = INFINITY; v.known = FALSE; v.path = NONE;

start.distance = 0;
while there are unknown vertices
  v = unknown vertex with smallest distance
  v.known = TRUE;
  for each w adjacent to v
    if (!w.known)
      if (v.distance + weight(v, w)  < w.distance) {
        decrease( w.distance to v.distance + weight(v, w))
        w.path = v;
      }
```

Dijkstra Example



Correctness of Dijkstra's Algorithm

The algorithm is correct because of a property of shortest paths:

If $P_k = v_1, v_2, \ldots, v_j, v_k$, is a shortest path from v_1 to v_k, then $P_j = v_1, v_2, \ldots, v_j$, must be a shortest path from v_1 to v_j, otherwise P_k would not be as short as possible since P_k extends P_j by just one edge (from v_j to v_k)

Also, P_j must be shorter than P_k (assuming that all edges have positive weights). So the algorithm must have found P_j on an earlier iteration than when it found P_k.

i.e. Shortest paths can be found by extending earlier known shortest paths by single edges, which is what the algorithm does.

Running Time of Dijkstra's Algorithm

The running time depends on how the vertices are manipulated.

The main 'while' loop runs $O(|V|)$ time (once per vertex)

Finding the "unknown vertex with smallest distance" (inside the while loop) can be a simple linear scan of the vertices and so is also $O(|V|)$. With this method the total running time is $O(|V|^2)$. This is acceptable (and perhaps optimal) is the graph is dense ($|E| = O(|V|^2)$) since it runs in linear time on the number of edges.

If the graph is sparse, ($|E| = O(|V|)$), we can use a priority queue to select the unknown vertex, using the deleteMin operation ($O(|\lg V|)$). We must also decrease the path lengths of some unknown vertices, which is also $O(\lg|V|)$. The deleteMin operation is performed for every vertex, and the decrese path length is performed for every edge, so the running time is $O(|E| \lg|V| + |V|\lg|V|) = O(|E| \lg|V|)$.

Dijkstra and Negative Edges

Note in the previous discussion, we made the assumption that all edges have positive weight. If any edge has a negative weight, then Dijkstra's algorithm fails. Why is this so?

Suppose a vertex, u, is marked as "known". This means that the shortest path from the starting vertex, s, to u has been found.

However, it's possible that there is negatively weighted edge from an unknown vertex, v, back to u. In that case, taking the path from s to v to u is actually shorter than the path from s to u without going through v.

Other algorithms exist that handle edges with negative weights for weighted shortest-path problem.

Directed Acyclic Graphs

A ***directed acyclic graph*** is a directed graph with no cycles.

A ***strict partial order*** R on a set S is a binary relation such that

- for all $a \in S$, aRa is false (irreflexive property)
- for all $a,b,c \in S$, if aRb and bRc then aRc is true (transitive property)

To represent a partial order with a DAG:

- represent each member of S as a vertex
- for each pair of vertices (a,b), insert an edge from a to b if and only if aRb

More Definitions

Vertex i is a **predecessor** of vertex j if and only if there is a path from i to j.

Vertex i is an **immediate predecessor** if vertex j if and only if (i, j) is an edge in the graph.

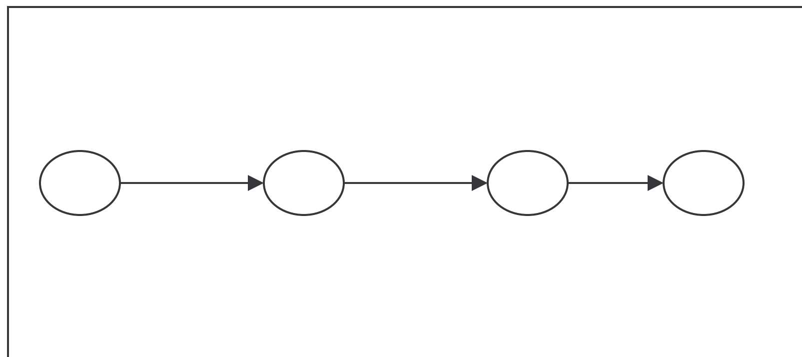
Vertex j is a **successor** of vertex i if and only if there is a path from i to j.

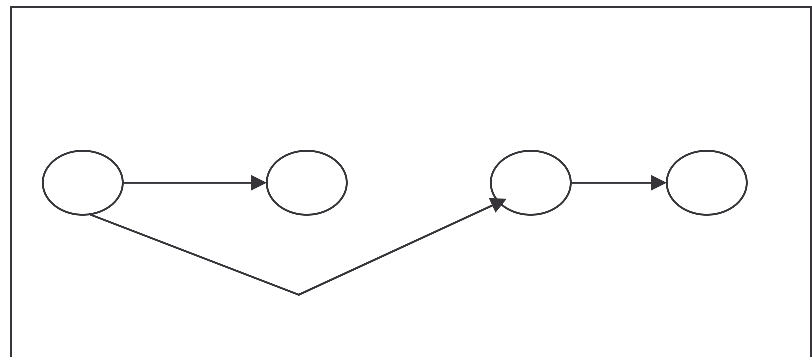
Vertex j is an **immediate predecessor** if vertex i if and only if (i, j) is an edge in the graph.

The **indegree** of a vertex, v, is the number of edges (u, v). I.e. the number of edges that come "into" v.

Topological Ordering

A topological ordering of the vertices of a DAG G = (V,E) is a linear ordering such that, for vertices i, j $\in$ V, if i is a predecessor of j, then i precedes j in the linear order. I.e. if there is a path from v_i to v_j, then v_i comes before v_j in the linear order

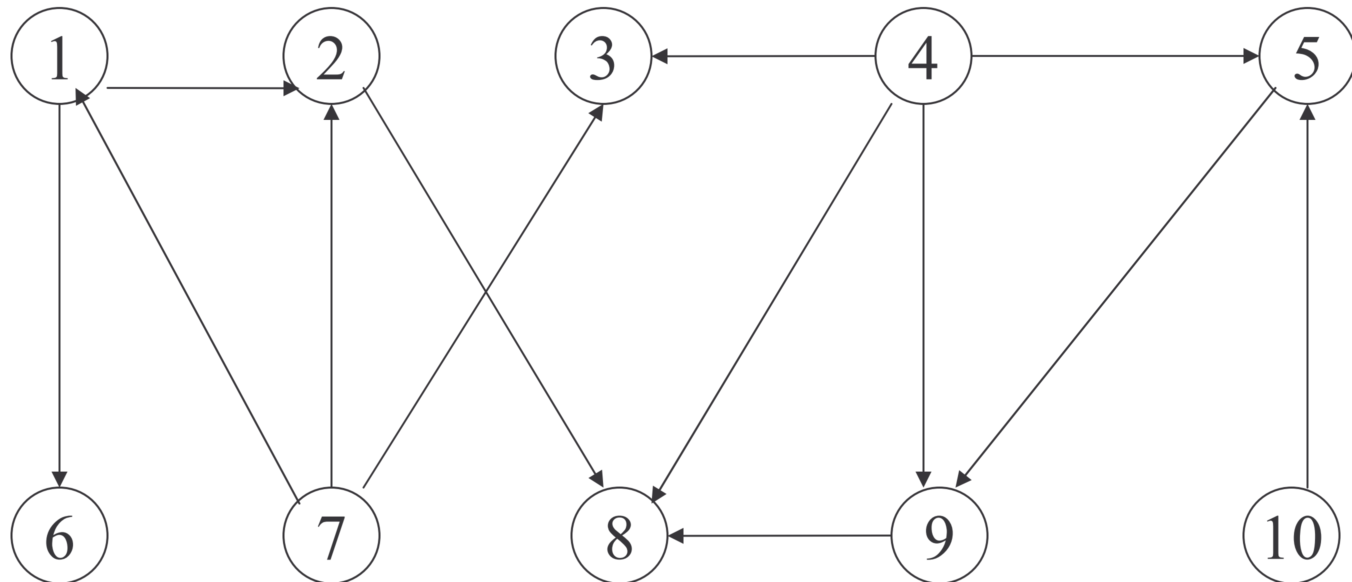




Topological Sort

```
void TopSort(Graph G) {
   unsigned int counter = 0 ;
   Queue<Vertex> q;
   Vertex indegree[|V|];
   for each Vertex v {
        v.indegree = GetInDegree(v);
        if (v.indegree == 0) q.enqueue(v);
   }
   while (!q.isEmpty()) {
        v = q.dequeue();
        Put v on the topological ordering;
        counter++;
        for (each vertex, w, adjacent to v {
               if (--w.indegree == 0)
                      q.enqueue(w);
        }
   }
   if (counter != G.NumVertices())
        declare an error -- G has a cycle
}
```

TopSort Example



Running Time of TopSort

1. At most, each vertex is enqueued just once, so there are $O(|V|)$ constant time queue operations.
2. The body of the for loop is executed at most once per edges = $O(|E|)$
3. The initialization is proportional to the size of the graph if adjacency lists are used = $O(|E| + |V|)$

The total running time is therefore $O(|E| + |V|)$