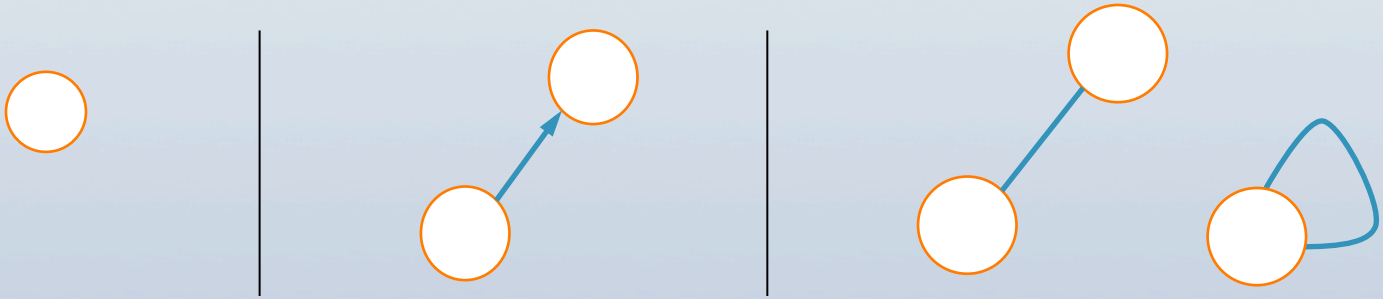
EE 441- CH10

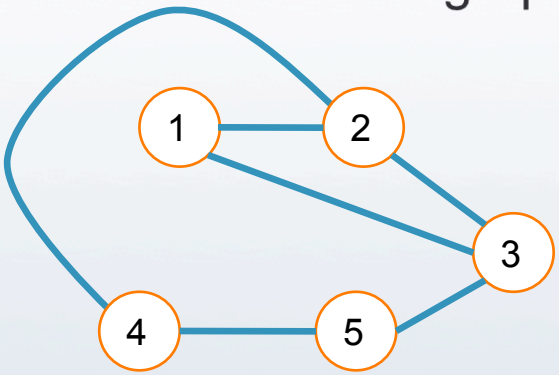
GRAPHS

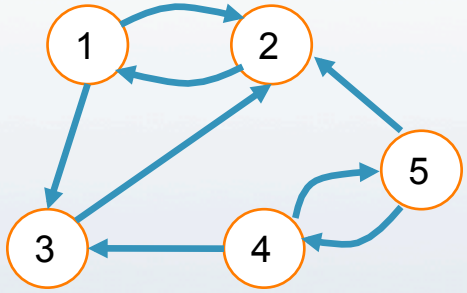
Graphs

- A graph G is an ordered pair G=(V,E) with a set of vertices or nodes (V) and the edges or arcs (E) that connect them.
- E is a binary relation on V, each edge is a tuple < v1,v2 >, where v1,v2 in (V)
- $|E| =< |V|^2$
- The edges indicate how we can move through the graph.
- A **weighted graph** is one where each edge has a cost for traveling between the nodes
- Typical examples for the edges:



- In a directed graph (or «digraph»), edges allow travel in one direction.
- In an undirected graph, edges allow travel in either direction.





The graph G=({1,2,3,4,5},{(1,2), (1,3), (2,3), (2,4), (3,5), (4,5)})

The directed graph G=({1,2,3,4,5}, {(1,2), (1,3), (2,1), (3,2), (4,3), (4,5), (5,2), (5,4)})

Degree of a Vertex

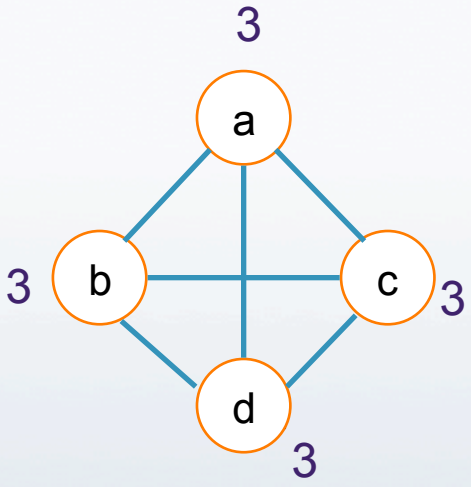
- The **degree** of a vertex is the number of edges incident to that vertex.
- For directed graph,
  - the **in-degree** of a vertex *v* is the number of edges that have *v* as the head.
  - the **out-degree** of a vertex *v* is the number of edges that have *v* as the tail.
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is.

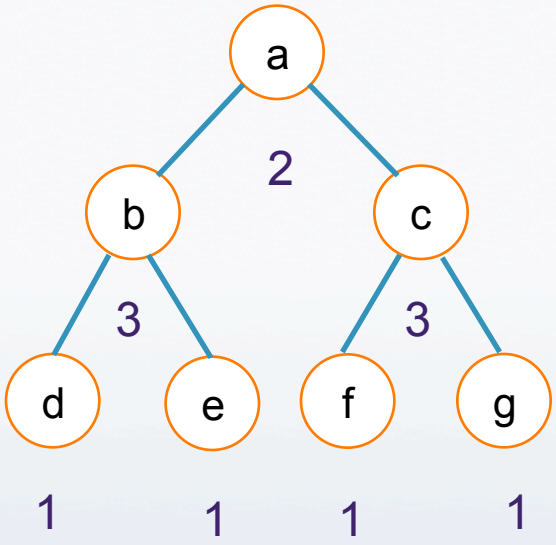
$$e = (\sum_{0}^{n-1} d_i)/2$$

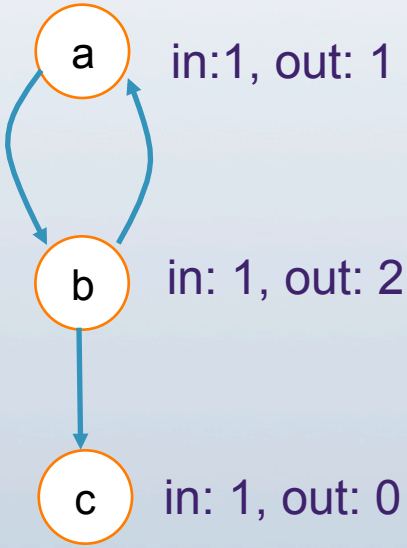
Why? Since adjacent vertices, each count the adjoining edge, it will be counted twice.

- A node with in-degree 0 is a root.

Examples







Adjacency

- For undirected graph:
  - Two vertices x, y are **adjacent** if < x, y > is an edge.
- For directed graph:
  - Vertex w is **adjacent** to v iff (v,w) $\in$ E.
    i.e. There is a direct edge from v to w
  - w is successor of v
  - v is predecessor of w

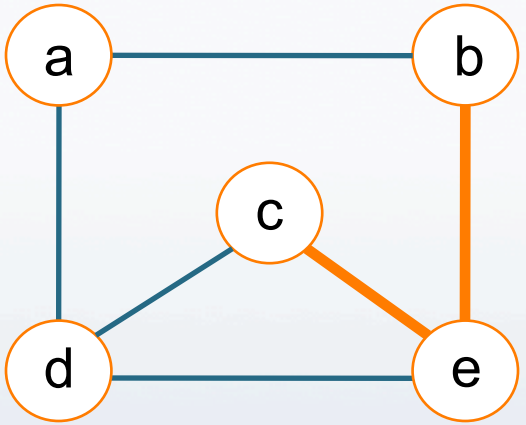
Path

- For undirected graph:
  - **Path:** a sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.
  - i.e. $v_1, v_2, \ldots v_k$ is a path iff $< v_i, v_{i+1} > \in E$ for all $1 \leq i \leq k-1$
- For directed graph:
  - A **directed path** between two vertices is a sequence of directed edges that begins at one vertex and ends at another vertex.
  - i.e. $v_1, v_2, \ldots, v_k$ is a path if $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$
- The length of a path in a graph is the number of edges in the path – Path is of length k.

Cycle

- Path is **simple** if vertices in sequence are distinct, i.e. a **simple path** passes through a vertex only once.

  eg: bec

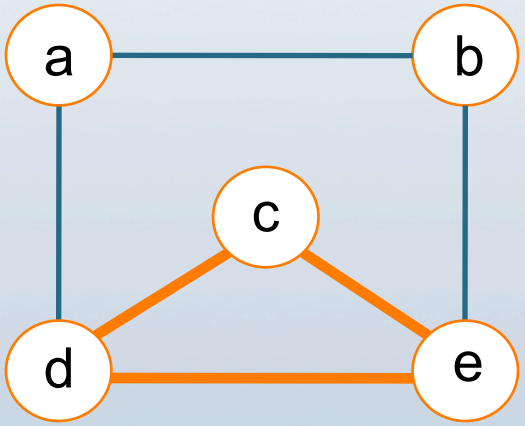


- A **cycle** is a path that begins and ends at the same vertex.

  eg: edce

  eg: dcedabed

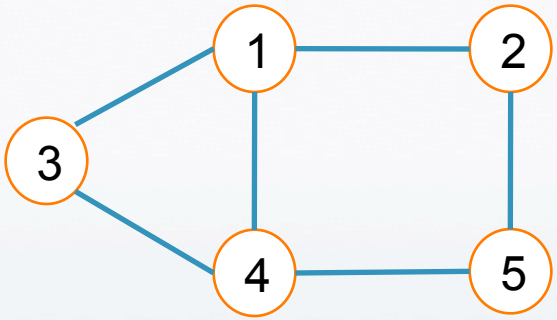
  < v, v > is cycle of length 1



- A **simple cycle** is a cycle that does not pass through any vertex more than once.

  eg: edce

Example – Undirected Graph



A graph G (undirected)

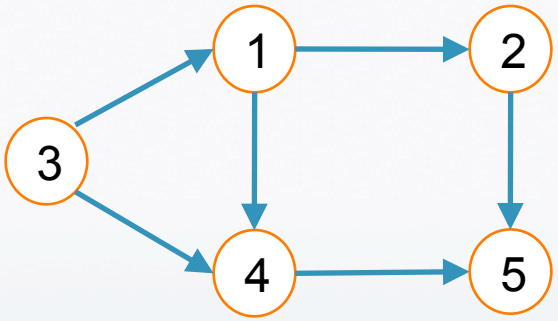
The graph G= (V,E) has 5 vertices and 6 edges:

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

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

- Adjacent:
  1 and 2 are adjacent; 1 is adjacent to 2 and 2 is adjacent to 1
- Path:
  1,2,5 (a simple path), 1,3,4,1,2,5 (a path but not a simple path)
- Cycle:
  1,3,4,1 (a simple cycle), 1,3,4,1,4,1 (cycle, but not simple cycle)

Example - Directed Graph



The graph G= (V,E) has 5 vertices and 6 edges:

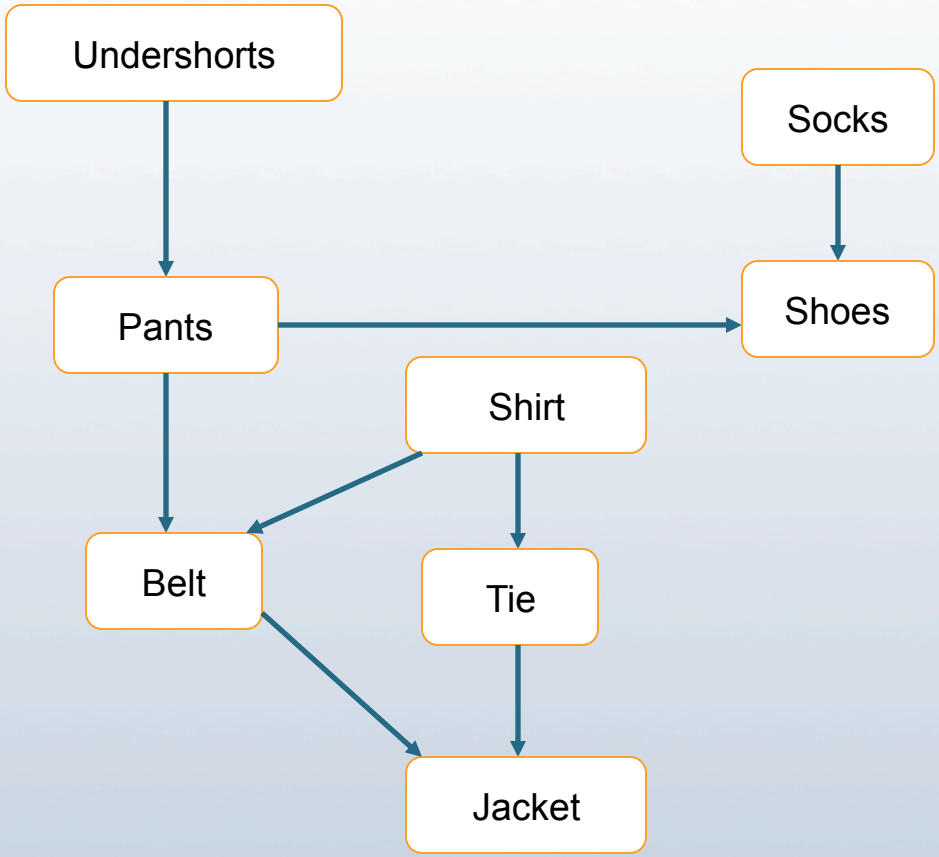
V = {1,2,3,4,5}

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

- *Adjacent:*
  2 is adjacent to 1, but 1 is NOT adjacent to 2
- *Path:*
  1,2,5 (a directed path),
- *Cycle:*
  1,4,3,1 (an undirected cycle, but NOT a directed cycle).

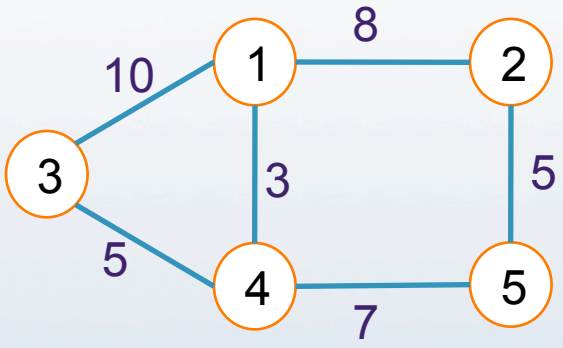
Acyclic Graph

An **acyclic graph** is one that has no cycles.
A Directed Acyclic Graph implies an ordering on events.

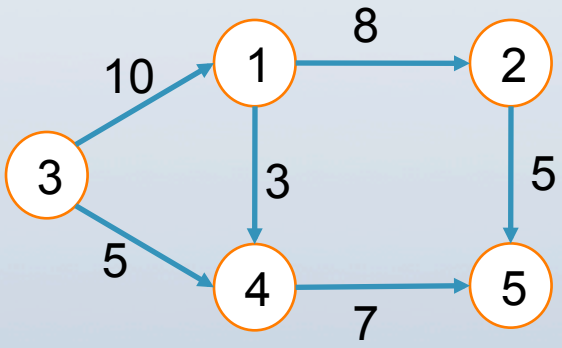


Weighted Graph

- We can label the edges of a graph with numeric values, the graph is called a ***weighted graph***.



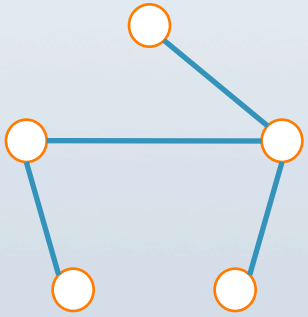
Weighted (Undirected) Graph



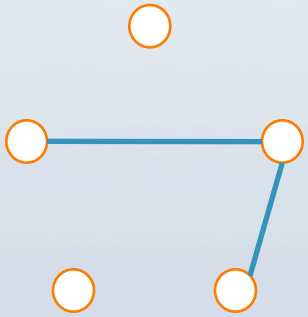
Weighted Directed Graph

Connected Graph

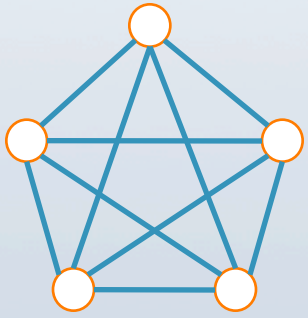
- A **connected graph** has a path between each pair of distinct vertices.
- A directed graph with this property is called **strongly connected**.
  - If a directed graph is not strongly connected, but the underlying graph (without direction to arcs) is connected then the graph is **weakly connected**.



a) connected



b) disconnected



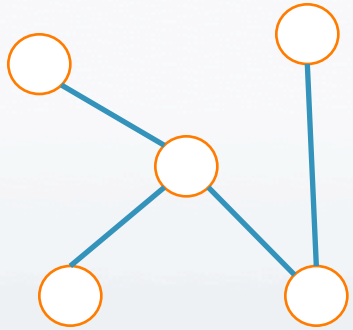
c) complete

More on Connectivity

n = #vertices

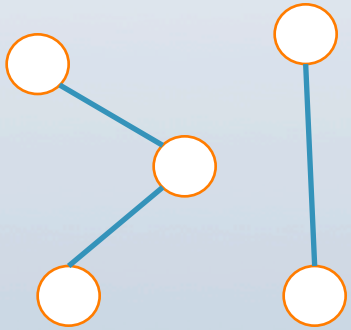
m = #edges

For a tree **m** = **n** - 1



n=5
m=4

If **m** < **n** - 1, G is not connected, else may or may not be connected



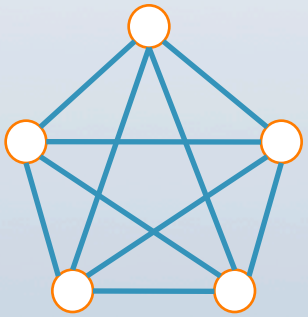
n=5
m=3

More on Complete graph

- Let **n** = #vertices, and **m** = #edges
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges,

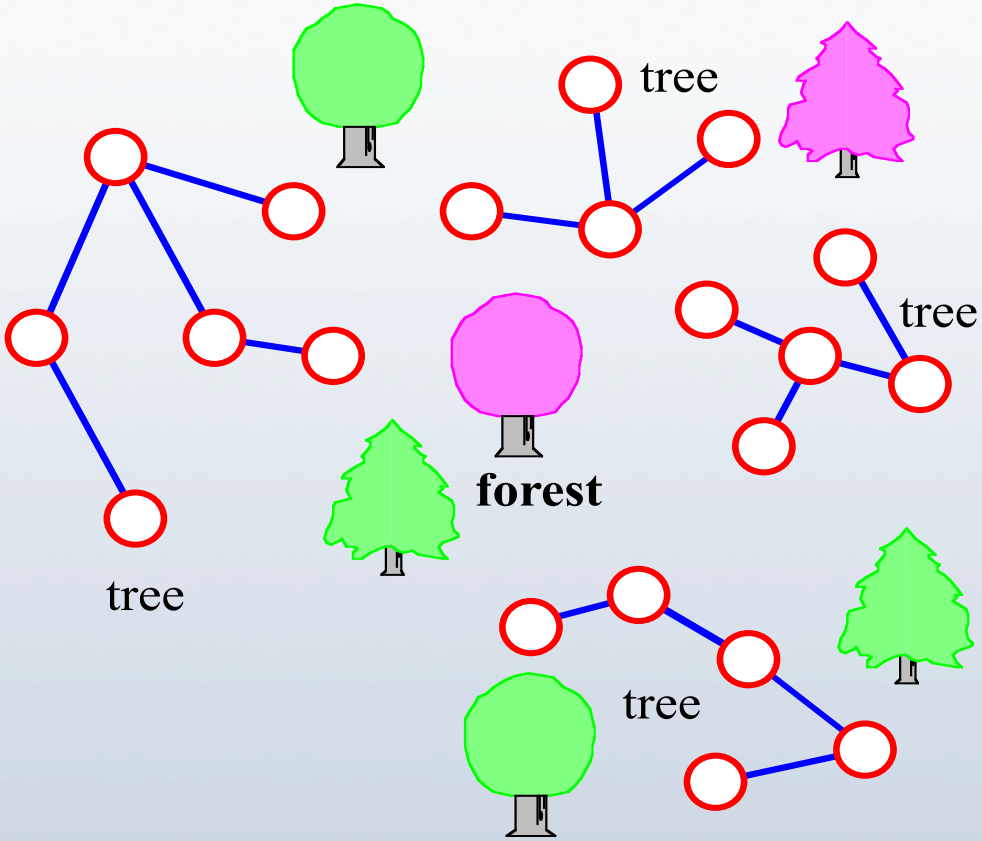
  however, we would have counted each edge twice!

  Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



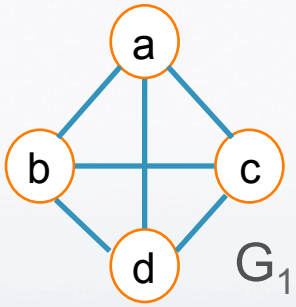
n=5;
m=(5*4)/2=10

- **tree** - connected graph without cycles
- **forest** - collection of trees

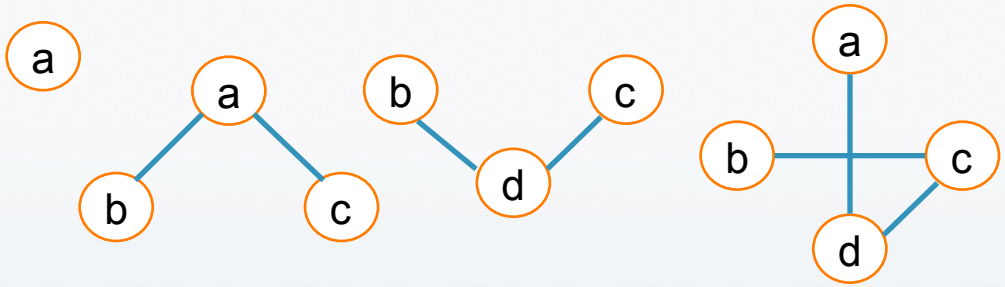


Subgraph

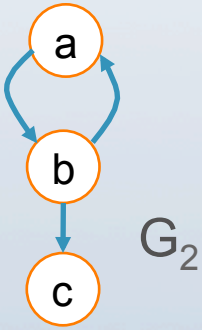
Subset of vertices and edges forming a graph



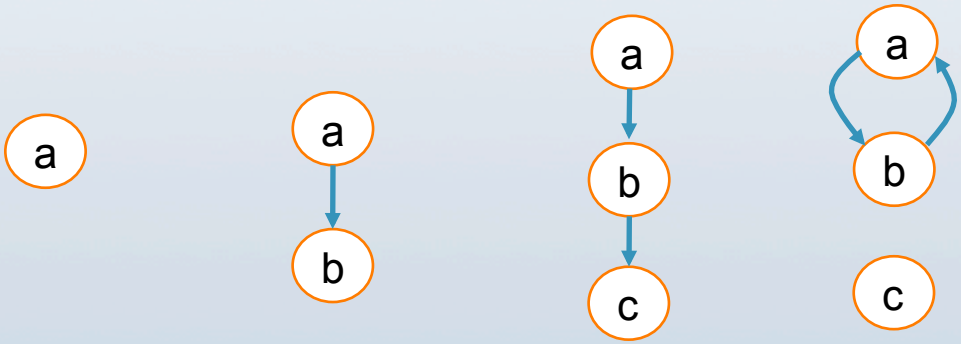
G_1



Some of the subgraphs of G_1



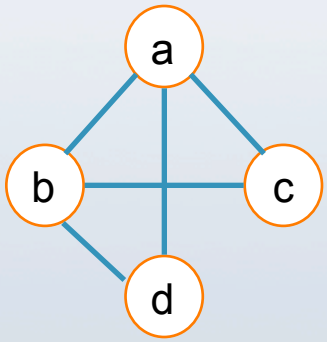
G_2



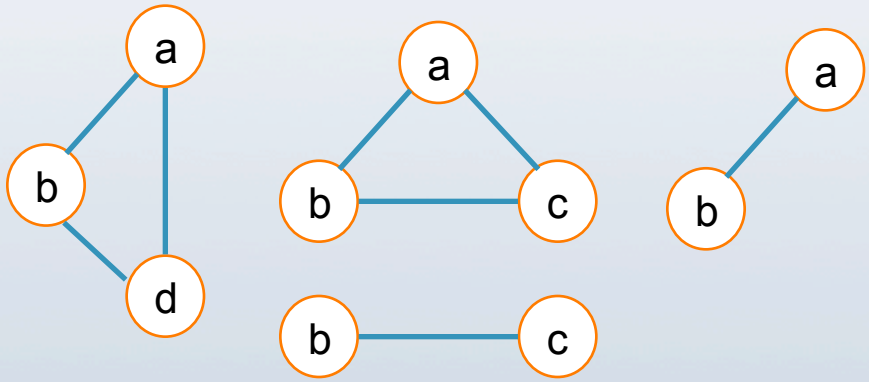
Some of the subgraphs of G_2

Clique

- **Clique:** a complete subset of an undirected graph.
- A subset of vertices of a graph such that every two vertices in the subset are connected by an edge.
- A subset of vertices where all pairs of vertices are adjacent.



G_1



Some cliques of G_1

Graph Class

```
// It is assumed that the number of vertices is fixed when it is constructed,
// but edges can be added and removed.
// It also supports a mark array for traversal algorithms.
class Graph {
private:
   void operator =(const Graph& G) {}      // Protect assignment
   Graph(const Graph&) {}                  // Protect copy constructor
public:
    Graph(int numVert) {}                  // Constructor
   ~Graph(void) {}                         // Destructor
    int n();                               // Return: the number of vertices
    int e();                               // Return: the number of edges
    int first(int v);                      // Return v's first neighbor
    int next(int v, int w) ;               // Return v's next neighbor after w
```

Graph Class (continues)

```
    void setEdge(int v1, int v2, int weight); // Set the weight for an edge
    void delEdge(int v1, int v2);             // Delete an edge  (v1, v2)
    bool isEdge(int v1, int v2);              //Determine if an edge is in the graph
    int weight(int v1, int v2);               // Return an weight of the edge (v1, v2);
    int getMark(int v);                       // Get mark value for a vertex,
    void setMark(int v, int val);             // Set mark value for a vertex,
    void clearMark(void);                     // Set all mark values as unvisited
};
```

Graph Representations

- **Adjacency Matrix:** A two dimensional array
  - appropriate when graph is dense
  - $|E|$ close to $|V|^2$

- **Adjacency Lists:** For each vertex we keep a list of adjacent vertices
  - appropriate when graph is sparse
  - $|E| << |V|^2$

There are some other graph representations such as incidence matrix, incidence list.

Adjacency Matrix

Let G=(V,E) be a graph with n vertices.

The **adjacency matrix** of G is a two-dimensional n by n array, say adj_mat.

If the edge (vi, vj) is in E(G), adj_mat[i][j]=1.

If there is no such edge in E(G), adj_mat[i][j]=0.

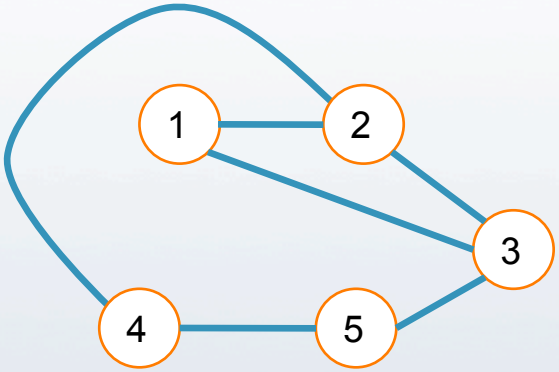
The adjacency matrix for an undirected graph is symmetric. $A=A^T$

The adjacency matrix for a digraph need not be symmetric.

“1” in <j,j> means there is a self-loop on vertex j.

Adjacency Matrix

Example : Undirected Graph

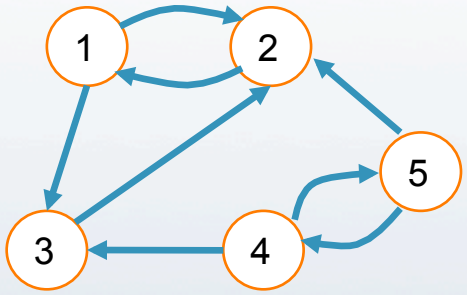


$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

The graph G=({1,2,3,4,5},{{1,2}, {1,3}, {2,3}, {2,4}, {3,5}, {4,5}}

Adjacency Matrix

Example : Directed Graph

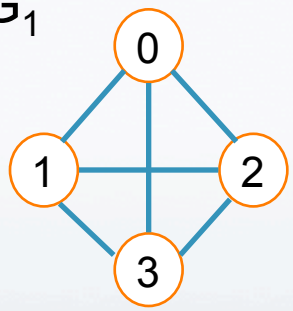


$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

The directed graph G=({1,2,3,4,5}, {(1,2), (1,3), (2,1), (3,2), (4,3), (4,5), (5,2), (5,4)})

Symmetry Examples in Adjacency Matrix

G_1



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

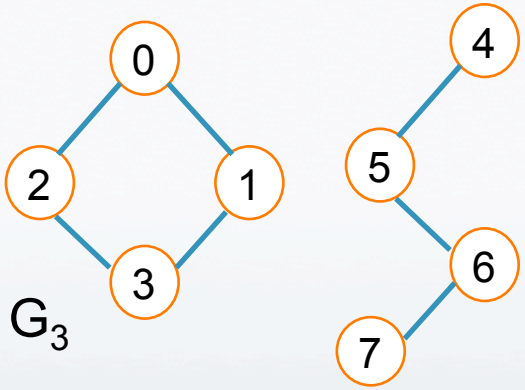
symmetric

G_2

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Not symmetric

G_3



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

symmetric

undirected: $n^2/2$
directed: n^2

```
// Implementation for the adjacency matrix representation
class Graph
{private:
   int numVertex, numEdge;
   int **matrix;
   int *mark;
public:
// Constructor ; Store number of vertices, edges,
// Pointer to adjacency matrix, Pointer to mark array
Graph(int numVert)  {
        int i;
         numVertex = n; numEdge = 0;
         mark = new int[n];       // Initialize mark array
         for (i=0; i<numVertex; i++)
             mark[i] = UNVISITED;
// Make matrix, it is not possible to create 2D array with a single new operation
        matrix = new int*[numVertex];
         for (i=0; i<numVertex; i++)
            matrix[i] = new int[numVertex];
        for (i=0; i< numVertex; i++) // Initialize to 0 weights
             for (int j=0; j<numVertex; j++)
                  matrix[i][j] = 0;
    };
```

```
~Graph() {
 // Destructor, Return dynamically allocated memory
        delete [] mark
         for (int i=0; i<numVertex; i++)
            delete [] matrix[i];
         delete [] matrix; };
 int n() { return numVertex; };
 int e() { return numEdge; };
 // Return first neighbor of v
    int first(int v) {
          for (int i=0; i<numVertex; i++)
              if (matrix[v][i] != 0) return i;
          return numVertex;      // Return n if none
    };
 // Return v's next neighbor after w
     int next(int v, int w) {
           for(int i=w+1; i<numVertex; i++)
                 if (matrix[v][i] != 0)
                       return i;
           return numVertex; };     // Return n if none
```

```
// Set edge (v1, v2) to "wt"
      void setEdge(int v1, int v2, int wt) {
              Assert(wt>0, "Illegal weight value");
              if (matrix[v1][v2] == 0)
                   numEdge++;
              matrix[v1][v2] = wt; };

      void delEdge(int v1, int v2) { // Delete edge (v1, v2)
             if (matrix[v1][v2] != 0)
                  numEdge--;
             matrix[v1][v2] = 0; };

      bool isEdge(int i, int j) // Is (i, j) an edge?
            { return matrix[i][j] != 0; };

      int weight(int v1, int v2) { return matrix[v1][v2]; };
      int getMark(int v) { return mark[v]; };
      void setMark(int v, int val) { mark[v] = val; };
      void clearMark(void)
       { for (i=0; i<numVertex; i++)
              mark[i] = UNVISITED; }
};
```

Merits of Adjacency Matrix

- Storage complexity: $O(|V|^2)$.
- Determining the connection of vertices is easy from the adjacency matrix.
- Edge existence query: O(1).
- The degree of a vertex i is

$$\deg ree(i) = \sum_{j=0}^{n-1} adj\_mat[i][j]$$

- For a digraph (unweighted), the row sum is the out_degree, while the column sum is the in_degree:

$$in\_\deg ree(i) = \sum_{j=0}^{n-1} adj\_mat[j][i] \qquad out\_\deg ree(i) = \sum_{j=0}^{n-1} adj\_mat[i][j]$$

But, ...

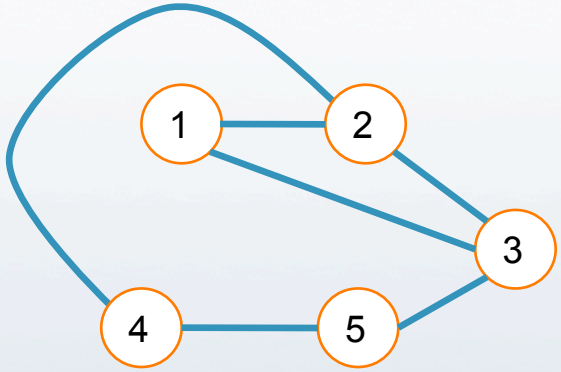
- Many graphs in practical problems are sparse.
- Not many edges --- not all pairs x, y have edge x → y.
- Matrix representation demands too much memory.
- We want to reduce memory footprint.
- Use sparse matrix techniques!

Adjacency List

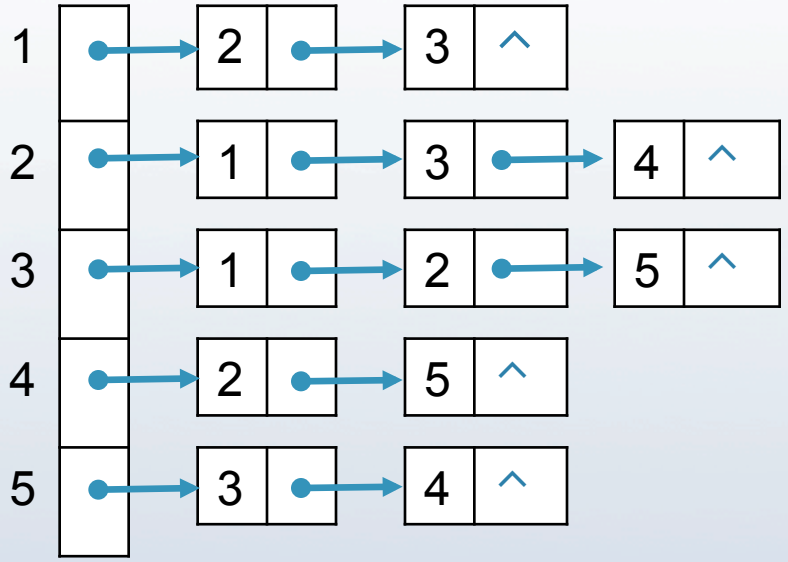
- A list of pointers, one for each node of the graph.
- These pointers are the start of a linked list of nodes that can be reached by one edge of the graph.
- Adj [u] is list of all vertices adjacent to u.
- List does not have to be sorted.
- For a weighted graph, this list would also include the weight for each edge.
- Undirected graphs: Each edge is represented twice.

- Notice that
  Adjacency matrix is better if the graph is dense (too many edges)
  Adjacency list is better if the graph is sparse (few edges)

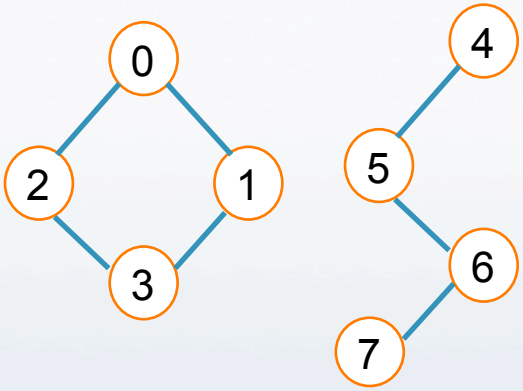
Adjacency List Example



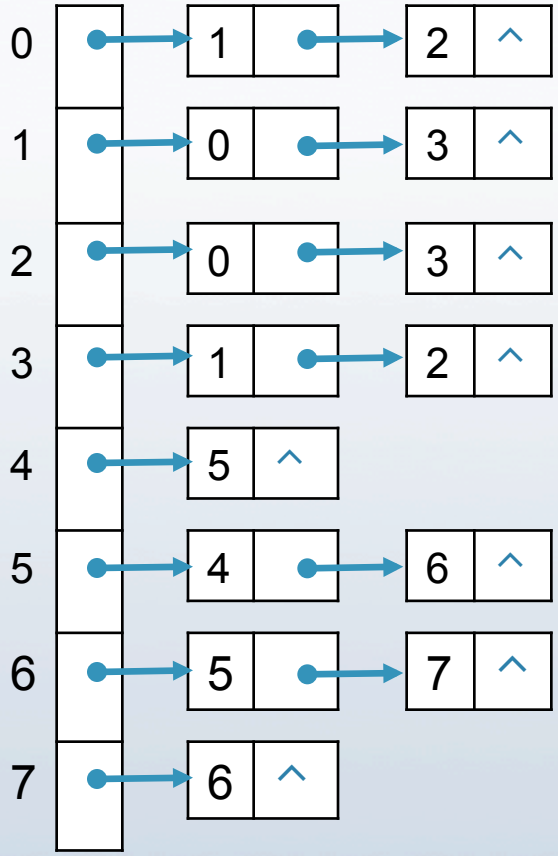
The graph
G=({1,2,3,4,5}, {{1,2}, {1,3}, {2,3}, {2,4}, {3,5}, {4,5}}



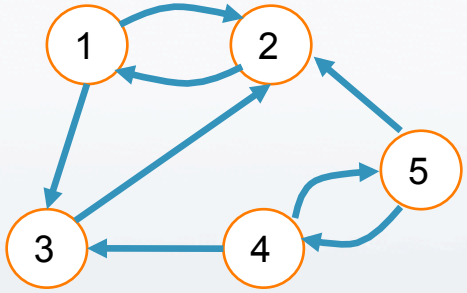
Adjacency List Example



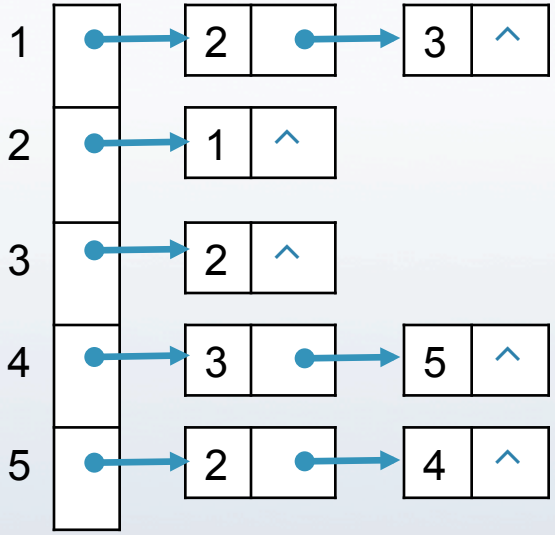
G_3



Adjacency List Example



The directed graph G=({1,2,3,4,5}, {(1,2), (1,3), (2,1), (3,2), (4,3), (4,5), (5,2), (5,4)})



Adjacency list features

- Storage Complexity:
  O(|V| + |E|)
  In undirected graph: O(|V|+2*|E|) = O(|V|+|E|)

- Edge query check:
  O(|V|) in worst case

- **degree of a vertex** in an undirected graph:
  # of nodes in adjacency list

- **# of edges** in a graph:
  determined in O(|V|+|E|)

- **out-degree** of a vertex j in a directed graph:
  # of nodes in its adjacency list
  Length of Adj[j] O(|V|) calculation

- **in-degree** of a vertex j in a directed graph:
  traverse the whole data structure
  Check all Adj[ ] lists O(|V|+|E|)

Graph Traversals

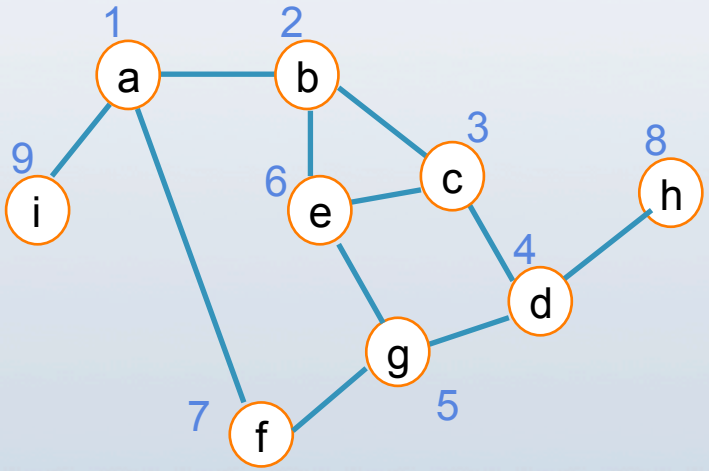
- A *graph-traversal* algorithm starts from a vertex v, visits all of the vertices that can be reachable from the vertex v.
- A graph-traversal algorithm visits all vertices if and only if the graph is connected.
- A connected component is the subset of vertices visited during a traversal algorithm that begins at a given vertex.
- We look at two graph-traversal algorithms:
  1. Depth-First Traversal
  2. Breadth-First Traversal

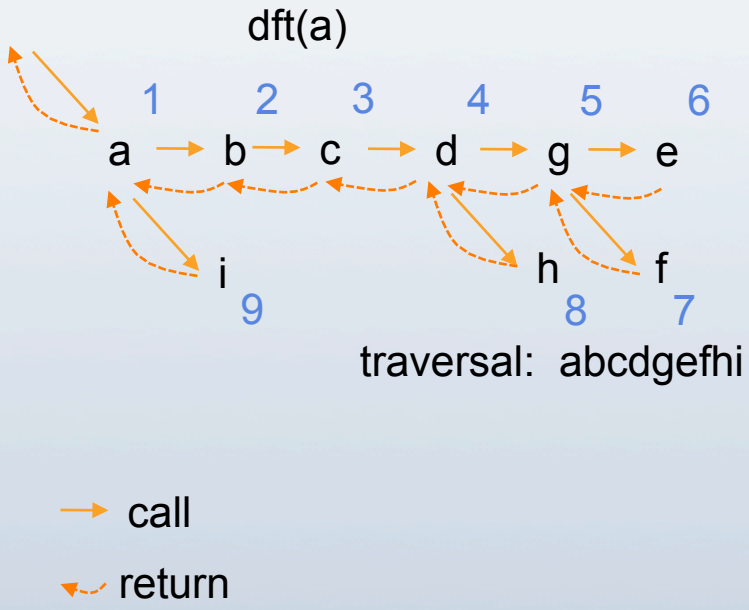
Depth-First Traversal (DFT)

- For a given vertex v, the **depth-first traversal** (also known as the **depth-first search, DFS)** algorithm proceeds along a path from v as deeply into the graph as possible until we reach a dead end.
- We then back up until we reach a node with an edge to an unvisited node.
- We take this edge and again follow it until we reach a dead end.
- This process continues until we back up to the starting node and it has no edges to unvisited nodes.
- The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
- We may visit the vertices adjacent to v in sorted order.

Recursive Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version
   Mark v as visited;
   for (each unvisited vertex u adjacent to  v)
     dft(u)
}
```

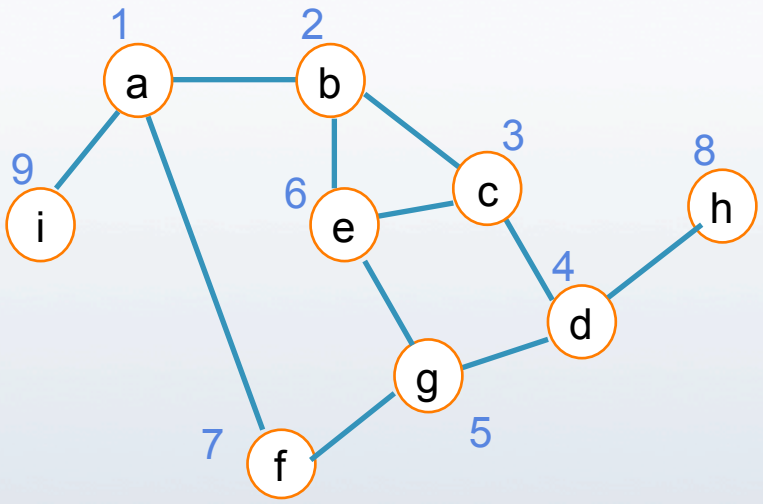




Iterative Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
   // Traverses a graph beginning at vertex v
   // by using depth-first strategy: Iterative Version
   s.createStack();
   // push v into the stack and mark it
   s.push(v);
   Mark v as visited;
   while (!s.isEmpty()) {
      if (no unvisited vertices are adjacent
           to the vertex on the top of stack)
         s.pop();  // backtrack
      else {
         Select an unvisited vertex u adjacent
         to the vertex on the top of the stack;
         s.push(u);
         Mark u as visited;
      }
   }
}
```

Trace of Iterative DFT



starting from vertex a

visit order	node visited	push	Stack (bottom to top)	pop	top
1	a	a	a		a
2	b	b	ab		b
3	c	c	abc		c
4	d	d	abcd		d
5	g	g	abcdg		g
6	e	e	abcdge		e
	(backtrack)		abcdg	e	g
	f	f	abcdgf		f
	(backtrack)		abcdg	f	g
	(backtrack)		abcd	g	d
7	h	h	abcdh		h
	(backtrack)		abcd	h	d
	(backtrack)		abc	d	c
	(backtrack)		ab	c	b
	(backtrack)		a	b	a
8	i	i	ai		i
	(backtrack)		a	i	a
	(backtrack)		(empty)		-

Interesting features of DFS

- Complexity: O(|V| + |E|)
  All vertices visited once, then marked
  For each vertex on queue, we examine all edges
  In other words, we traverse all edges once
- DFS does not necessarily find shortest path
  - Why?

Breadth-First Traversal (BFT, or BFS)

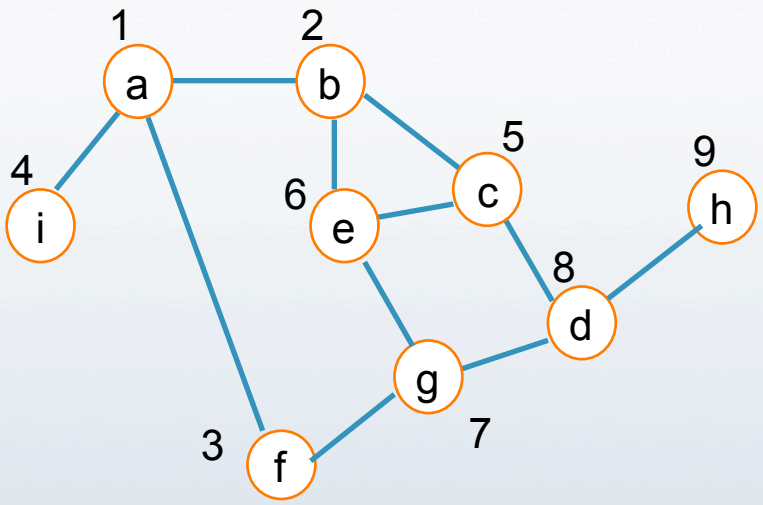
- After visiting a given vertex v, the **breadth-first traversal (search)** algorithm visits every unvisited vertex adjacent to v before visiting any other vertex.
- Thus, from the starting node, we follow all paths of length one.
- Then we follow paths of length two that go to unvisited nodes.
- We continue increasing the length of the paths until there are no unvisited nodes along any of the paths.
- The breath-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v. We may visit the vertices adjacent to v in sorted order.

Iterative Breadth-First Traversal

```
bft(in v:Vertex) {
// Traverses a graph beginning at vertex v iteratively
// by using breath-first strategy
   queue q;
   // add v to the queue and mark it
   q.insert(v);
   Mark v as visited;
   while (!q.isEmpty()) {
      q.delete(w);
      for (each unvisited vertex u adjacent to  w) {
         Mark u as visited;
         q.insert(u);
      }
   }

}
```

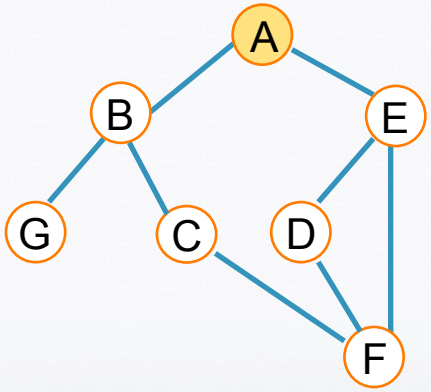
Trace of Iterative BFT



starting from vertex a

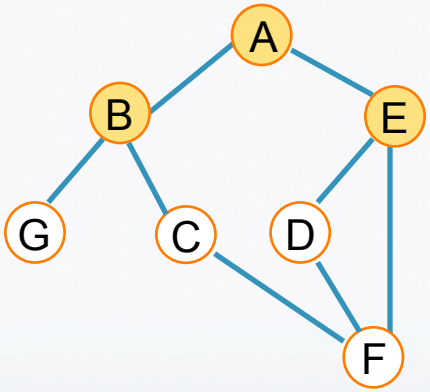
node visited	Queue (front to back)
a	a
	(empty)
b	b
f	bf
i	bfi
	fi
c	fic
e	fice
	ice
g	iceg
	ceg
	eg
d	egd
	gd
	d
	(empty)
h	h
	empty

the nodes visited (marked) are shown as yellow



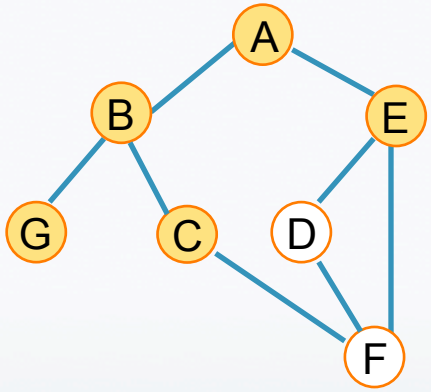
Queue: A

Start with A. Mark it



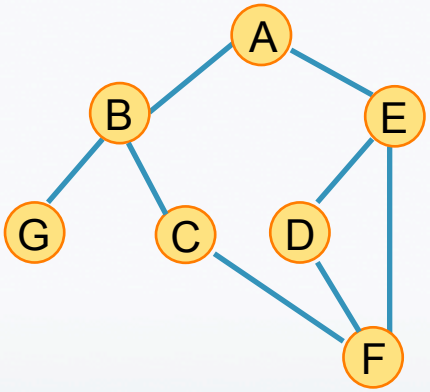
Queue: ABE

Expand A's adjacent vertices. Mark them and put them in queue.



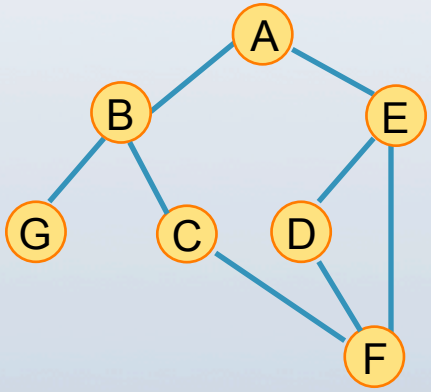
Queue: ABECG

Now take B off queue, and queue its neighbors.



Queue: ABECGDF

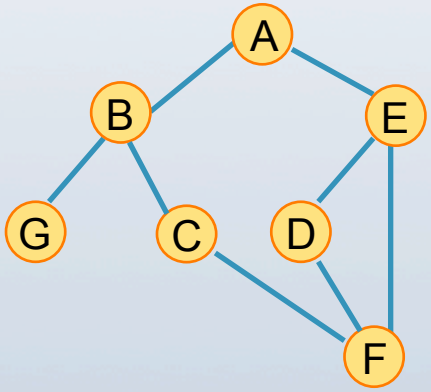
Do same with E.



Queue: ABECGDF

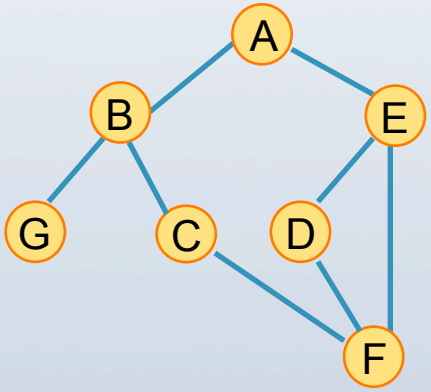
Take C off queue.

Its neighbor F is already marked, so not queued.



Queue: ABECGDF

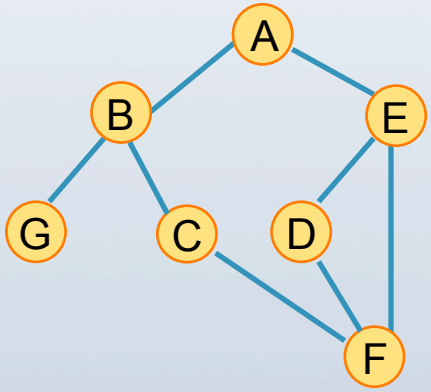
Take G off queue.



Queue: ABECGDF

Take D off queue.

F, E marked so not queued.



Queue:ABECGDF

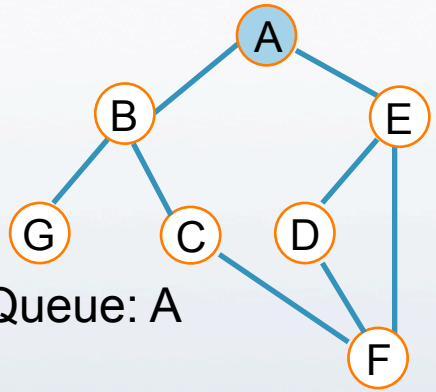
Take F off queue.

E, D, C marked, so not queued again.

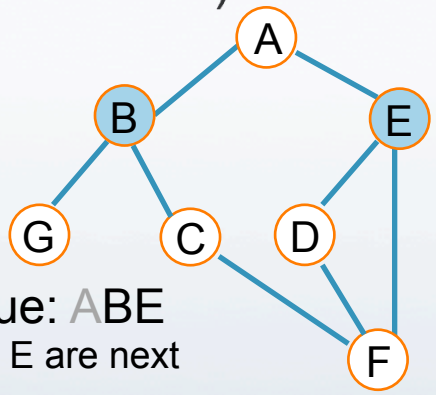
Marking the nodes is very important to handle cycles! If marking was not used

Start with A.

Put in the queue (the nodes in Queue are blue)

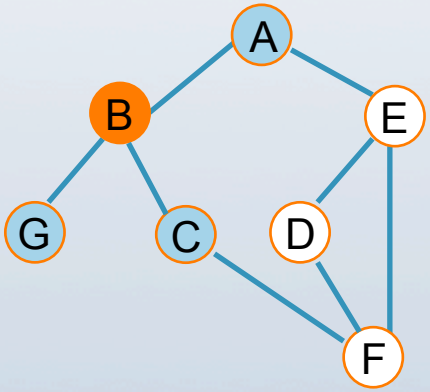


Queue: A



Queue: ABE

B and E are next



Queue: ABEACG

When we go to B, we put A,C and G in the queue

A is queued again although it was processed previously

Interesting features of BFS

Complexity: O(|V| + |E|)

- All vertices put on queue exactly once.
- For each vertex on queue, we expand its edges.
- In other words, we traverse all edges once.

BFS finds shortest path from s to each vertex

- Shortest in terms of number of edges.
- Why does this work?

Traversal Analysis - Summary

- Graph as state space (node = state, edge = action)
- BFS and DFS each search the state space for a best move.
- If the graph is connected, these methods will visit each node exactly once.
- If the search is exhaustive, they will find the same solution, but if there is a time limit and the search space is large...
  - DFS explores a few possible moves, looking at the effects far in the future
  - BFS explores many solutions but only sees effects in the near future (often finds shorter solutions)

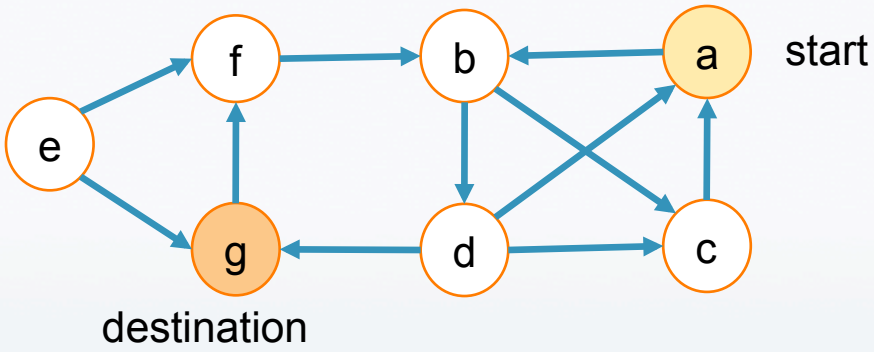
Finding a Path

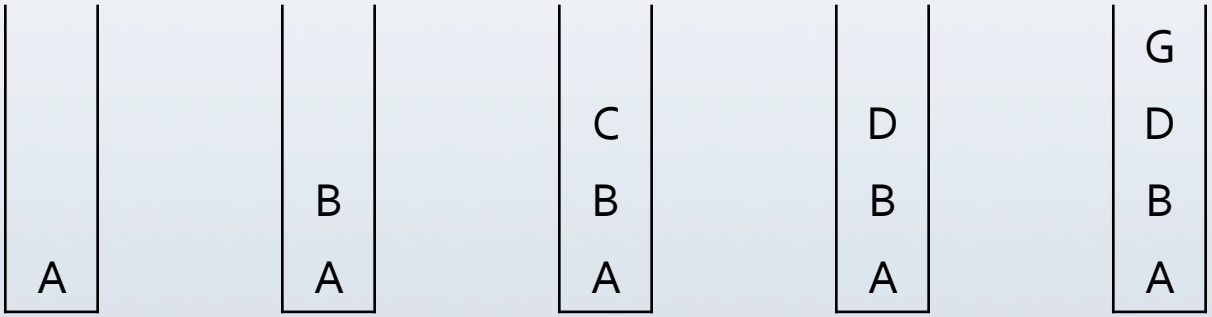
- Find path from source vertex s to destination vertex d.
- Use graph search starting at s and terminating as soon as we reach d.

  Need to remember edges traversed.
- Use depth – first search ?
- Use breath – first search?

DFS vs. BFS

DFS Process





DFS on A

DFS on B

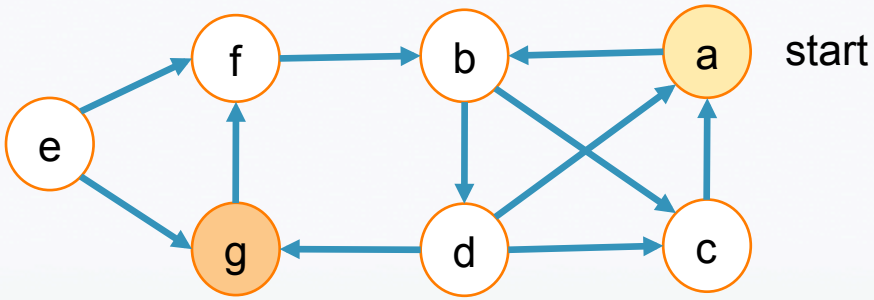
DFS on C

DFS on D
Return to
call on B

DFS on G
Destination found
Done!

Path is implicitly stored in DFS recursion. Path is: A, B, D, G
(use another stack to correct the order)

DFS vs. BFS



BFS Process

rear	front
	A

Initial call to BFS on A
Add A to queue

rear	front
	B

Dequeue A
Add B

rear	front
	C D

Dequeue B
Add C, D

rear	front
	D

Dequeue C
Nothing to add

rear	front
	G

Dequeue D
Add G

Destination found - done!

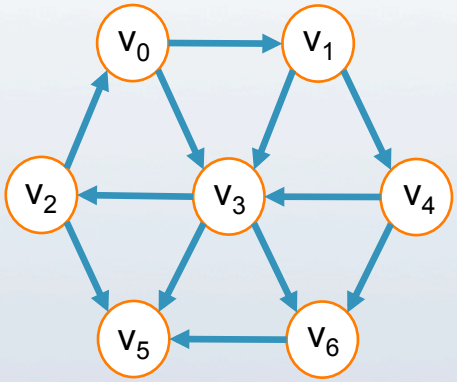
To extract the path, previous vertices must be remembered as the nodes are inserted into Queue. So follow the previous nodes starting from destination and insert in a stack until start node is reached. Then the path is in the stack

vertex	a	b	c	d	e	f	g
previous	-	a	b	b			d

a
b
d
g

Unweighted Shortest-Path problem

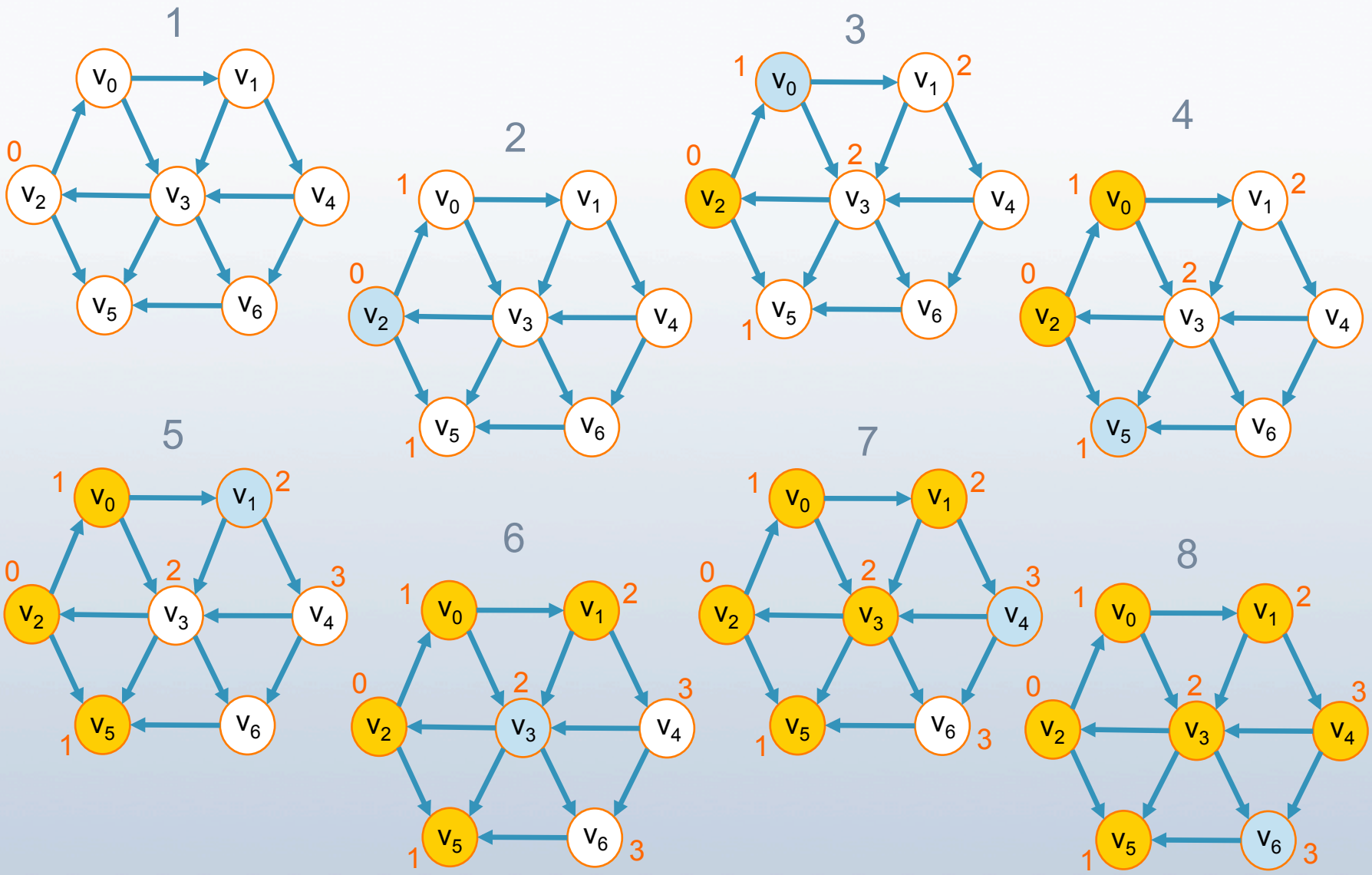
- *Find the shortest path (measured by number of edges) from a designated vertex S to every vertex.*



Algorithm

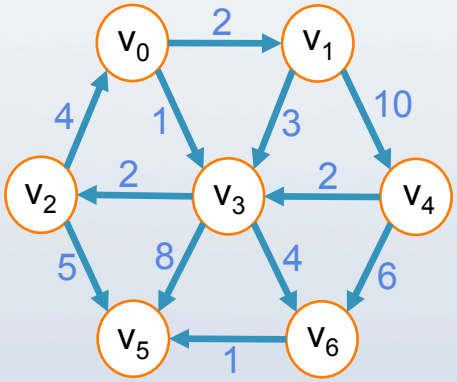
1. Start with an initial node s.
   - Mark the distance of s to s, D_s as 0.
   - Initially $D_i = \infty$ for all $i \neq s$.
2. Traverse all nodes starting from s as follows:
   1. If the node we are currently visiting is v, for all *w* that are adjacent to v:
      - Set $D_w = D_v + 1$ if $D_w = \infty$.
   2. Repeat step 2.1 with another vertex u that has not been visited yet, such that $D_u = D_v$ (if any).
   3. Repeat step 2.1 with another unvisited vertex u that satisfies $D_u = D_v + 1$.(if any)

Searching the graph in the unweighted shortest-path computation. The orange vertices have already been completely processed, the white vertices have not yet been used as *v*, and the blue vertex is the current vertex, *v*. ($s=v_2$)



Weighted Shortest-Path Problem

- *Find the shortest path (measured by total cost) from a designated vertex S to every vertex. All edge costs are nonnegative.*



Dijkstra's algorithm

```
function Dijkstra(Graph, source):
// Initializations
for each vertex v in Graph:
   // Unknown dist from source to v
   dist[v] := infinity ;
   // Previous node in optimal path
   // from source
   previous[v] := undefined ;
end for
// Distance from source to itself
dist[source] := 0 ;
// All nodes in the graph are
// unoptimized - thus are in Q
Q := the set of all nodes in Graph ;
// The main loop
while Q is not empty:
  u := vertex in Q with min dist[] value;
  remove u from Q ; /***/
  if dist[u] = infinity then break;
  //all remaining vertices are
  // inaccessible
  for each neighbor v of u:
  // where v has not yet been
  // removed from Q.
    alt:=dist[u]+dist_between(u,v);
      if (alt < dist[v])
         dist[v] := alt ;
         previous[v] := u ;
         decrease-key v in Q;
         // Reorder v in the Queue
      end if
   end for
end while
return dist;
```

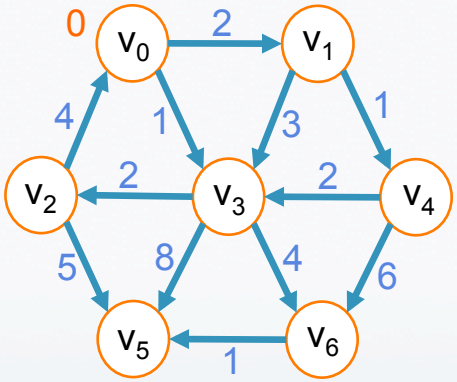
If we are only interested in a shortest path between vertices *source and target,*
*then we can terminate the search if u = target at line marked /***/*

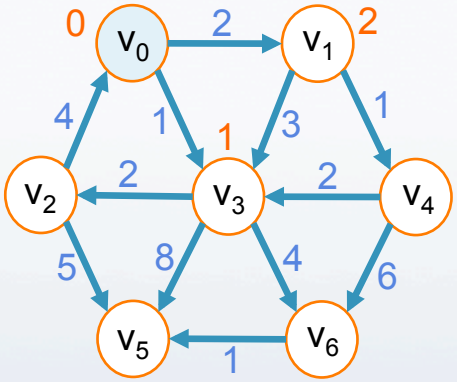
Dijkstra's algorithm (continues)

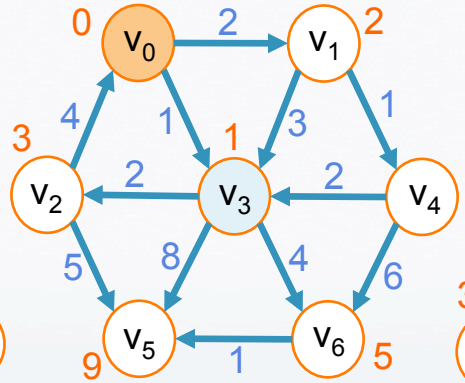
Now we can read the shortest path from source to target by reverse iteration considering previous array

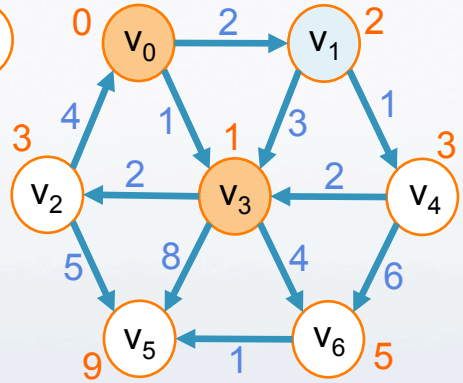
```
S := empty sequence
u := target
// Construct the shortest path with a stack S
while previous[u] is defined:
   // Push the vertex into the stack
   insert u at the beginning of S
   // Traverse from target to source
   u := previous[u]
end while ;
```

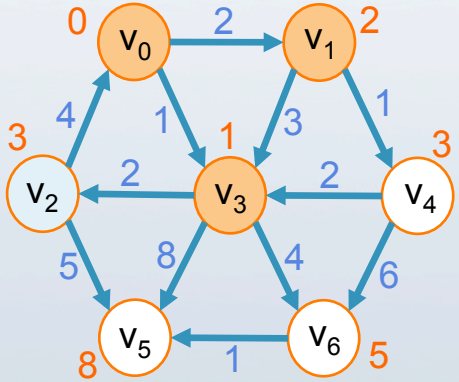
Stages of Dijkstra's algorithm ($s=v_0$)

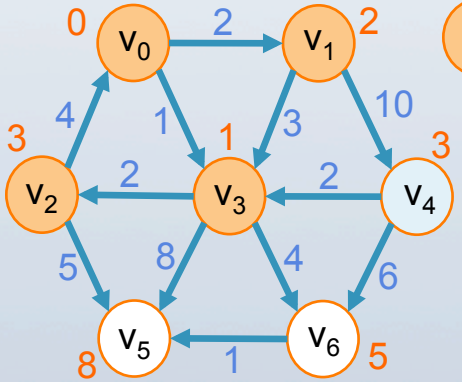


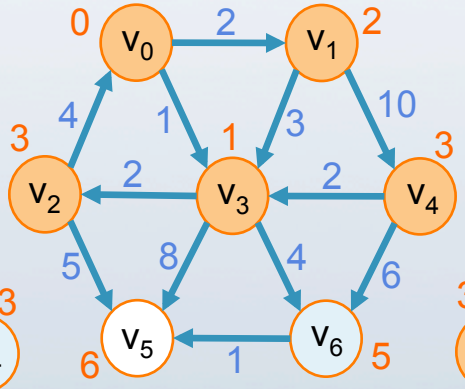


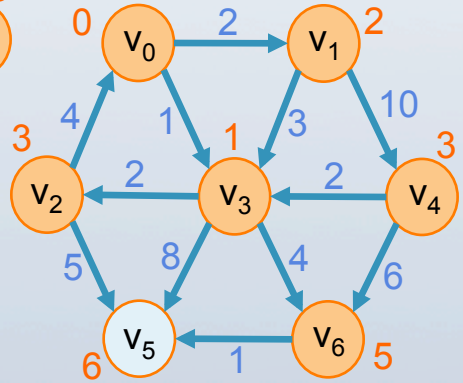






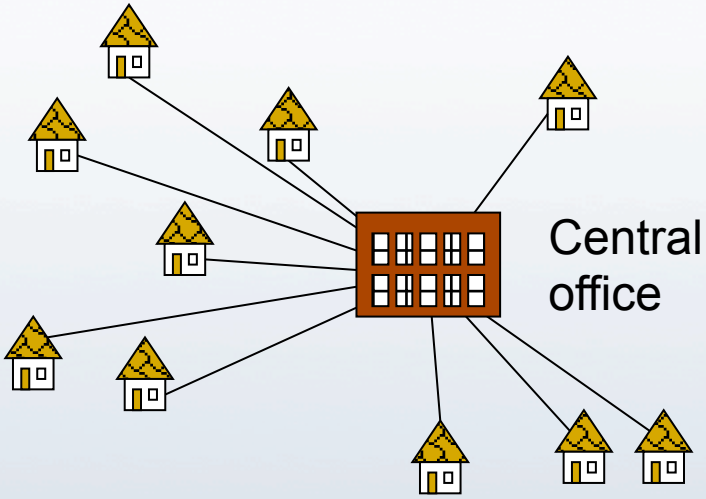




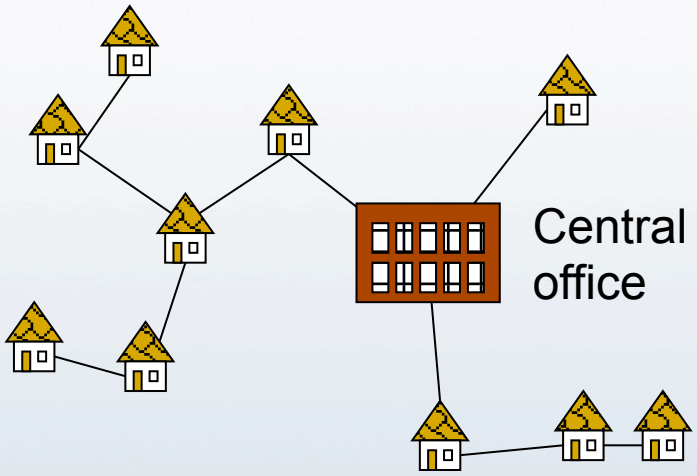


vertex	0	1	2	3	4	5	6
previous	-	0	3	0	2	~~3~~ ~~2~~ 6	3

Minimum Spanning Tree Problem: Laying Telephone Wire



Naïve Approach
expensive



Minimum spanning tree:
Minimizes the total length of wire connecting the customers