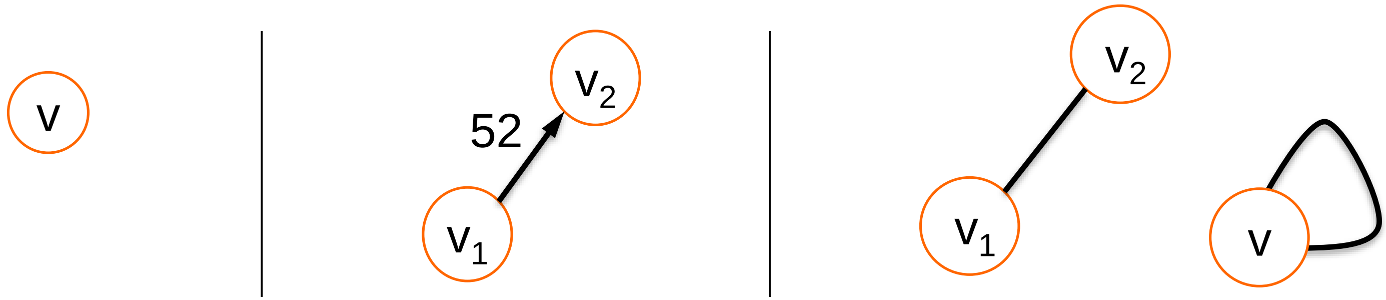
# EE 441 Data Structures

## Chapter 8: Graphs

Klaus Werner SCHMIDT
Uğur HALICI

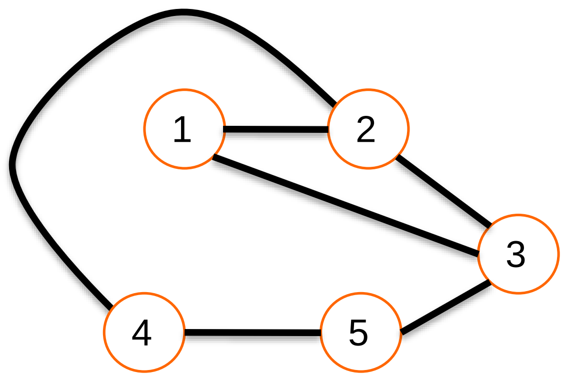
# Graphs

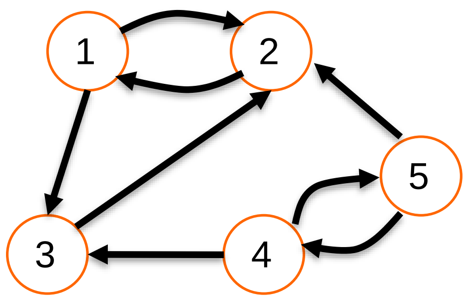
- A graph G is an ordered pair $G = (V, E)$ with a set of **vertices** or nodes $(V)$ and the **edges** or arcs $(E)$ that connect them.
- E is a binary relation on $V$, each edge is a tuple $< v_1, v_2 >$, where $v_1, v_2$ in $(V)$
- $|E| \leq |V|^2$
- The edges indicate how we can move through the graph.
- A **weighted graph** is one where each edge has a cost for traveling between the nodes
- Typical examples for the edges:



# Graphs

- Undirected graph: edges allow travel in either direction
- Directed graph: edges allow travel in one direction




Undirected graph

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

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

- Degree of a vertex: number of edges incident to that vertex
- For directed graph,
  - In-degree of a vertex $v$
    $\rightarrow$ number of edges that have $v$ as the head
  - Out-degree of a vertex $v$
    $\rightarrow$ 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

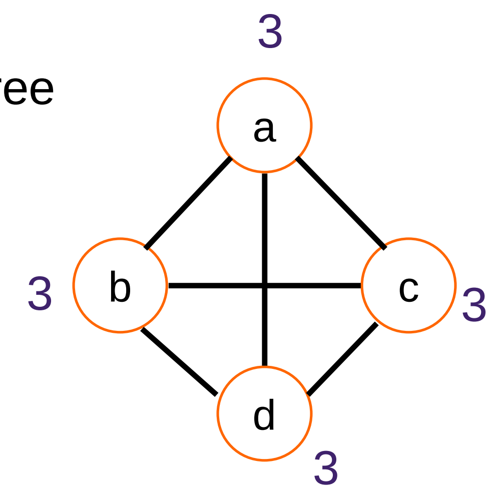
$$|e| = \frac{1}{2} \cdot \sum_{i=0}^{n-1} d_i$$

Explanation:
Each edge is counted twice since
Vertices are adjacent to each edge

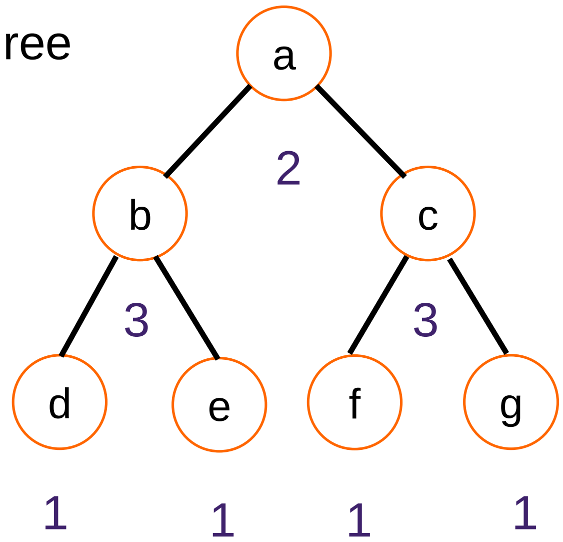
- A node with in-degree 0 is a root.

# Examples

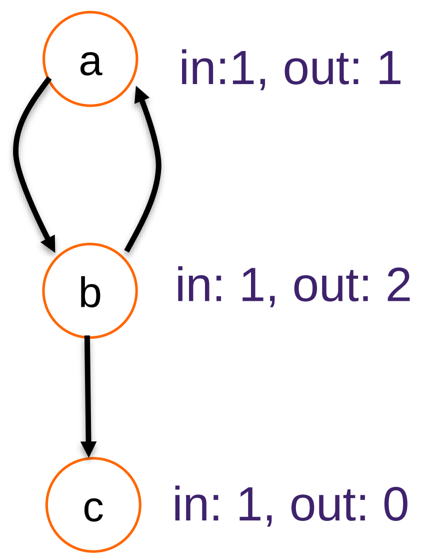
Degree




Degree




In-degree and
Out-degree



# 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 $w$

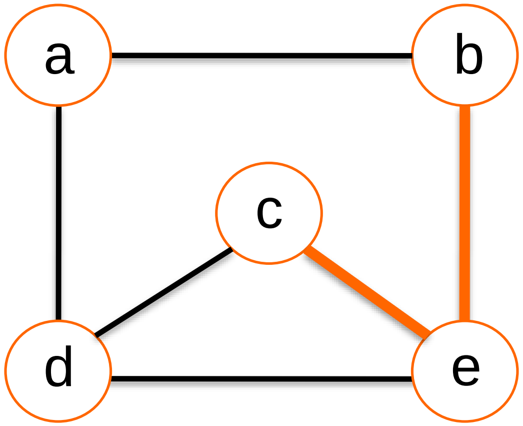
# Path

- 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 for $1 \leq i \leq k-1$
  - i.e. $v_1, v_2, \ldots v_k$ is a path iff $< \ v_i, v_{i+1} \ > \in E$ for $1 \leq i \leq k-1$
- 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

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

  For example: bec




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

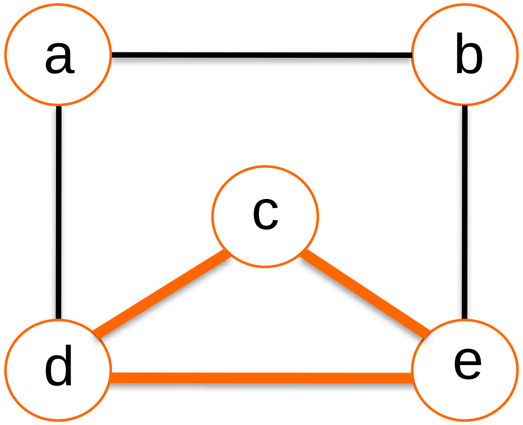
  For example: edce

  For example: dcedabed

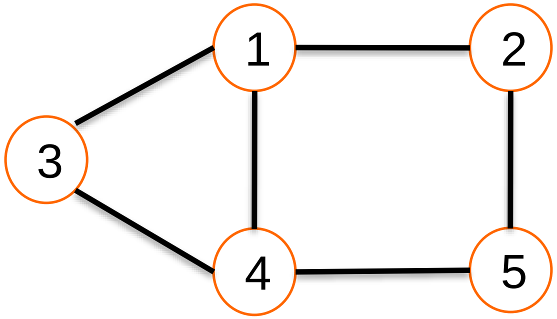
  Special case: $< \; v, v \; >$ is a cycle of length 1

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

  For example: edce



# Example – Undirected Graph




A graph G (undirected)

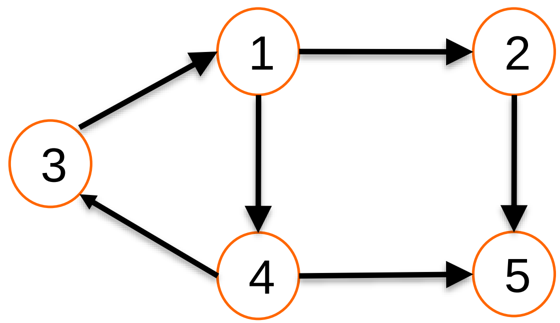
The graph G= (V,E) has 5 vertices and 6 undirected (12 directed) edges

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}

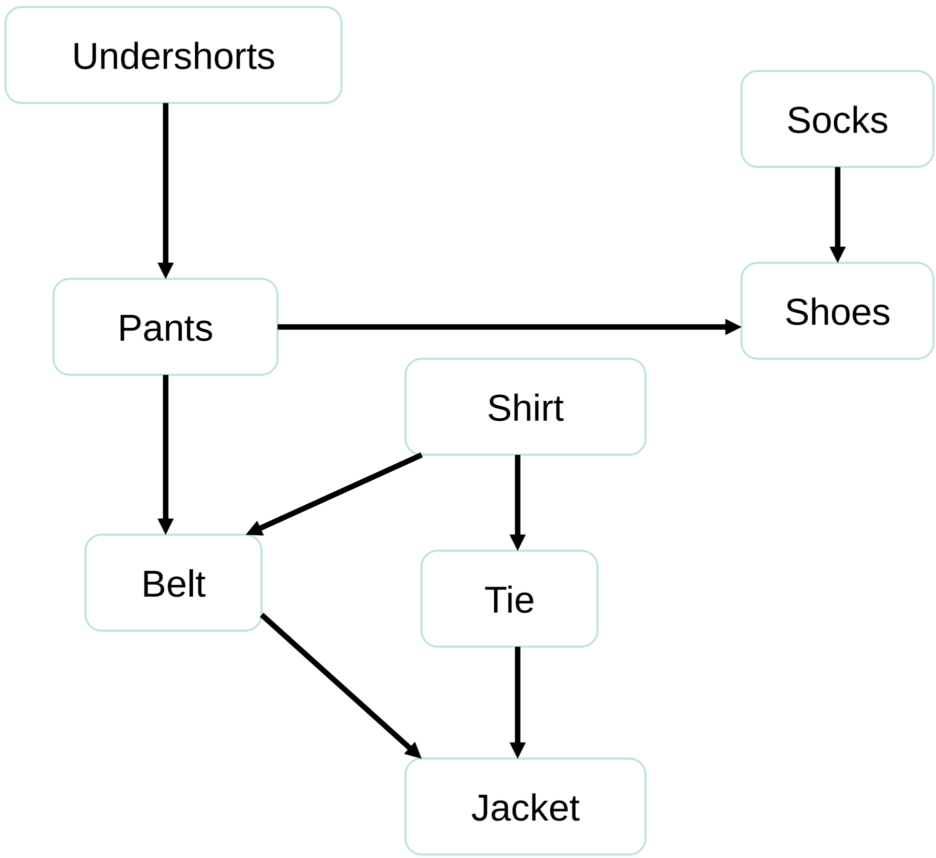
E = { (1,2),(1,4),(2,5),(4,5),(3,1),(4,3) }

- *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 (a directed cycle),

# Acyclic Graph

An **acyclic graph** is one that has no cycles

A Directed Acyclic Graph implies an ordering on events

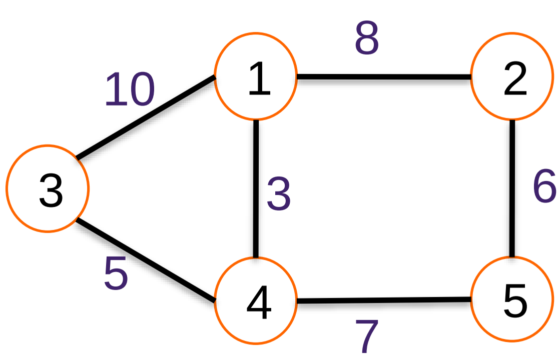


# Weighted Graph

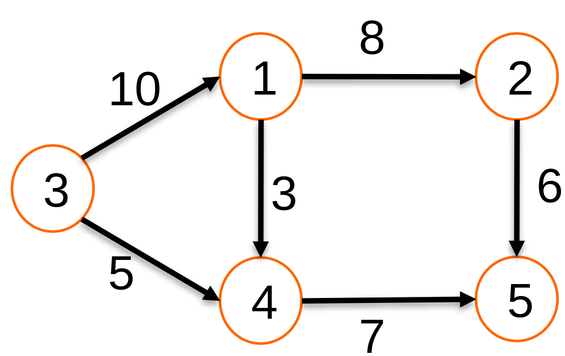
- We can label the edges of a graph with numeric values

  → the resulting graph is called a ***weighted graph***

Weighted (Undirected) Graph

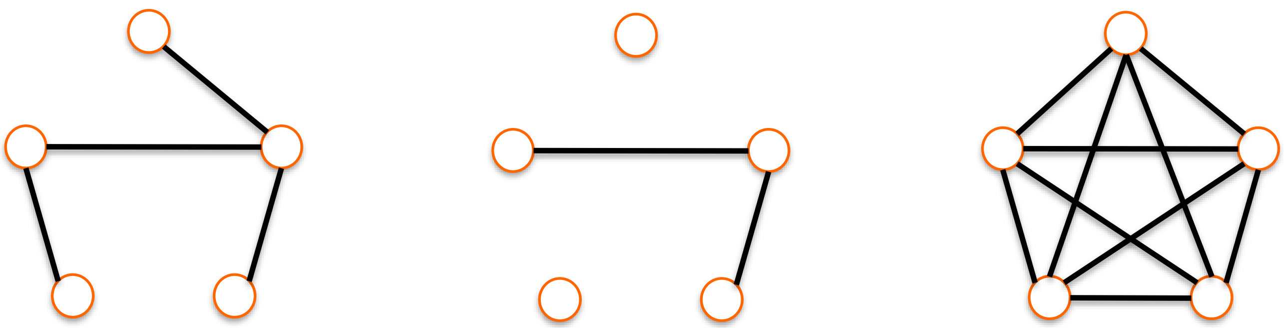



Weighted Directed Graph



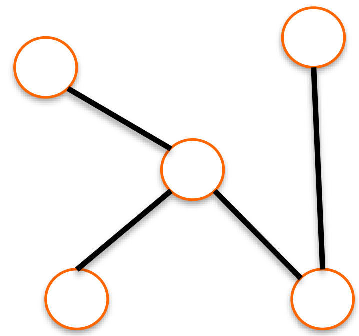
# Connected Graph

- 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 complete graph:** every pair of distinct vertices is connected by a unique edge



a) connected b) disconnected c) complete

# More on Connectivity



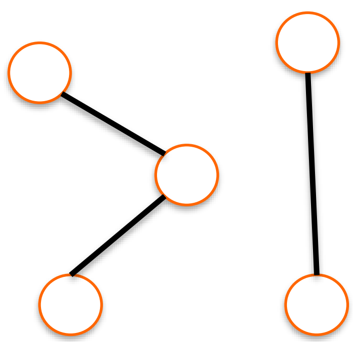
n=5
m=4

**n** = #vertices

**m** = #edges

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

If **m** < **n** - 1, G is not connected



n=5
m=3

Note: The trees that we covered were all directed starting from the root node.

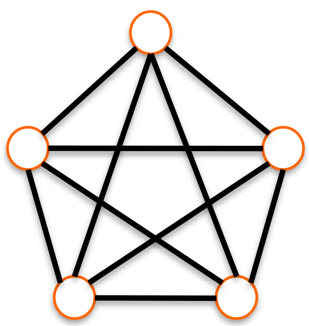
# More on Complete Graphs

- 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!

Intuitively, $m = \frac{n(n-1)}{2}$

- Therefore, if a graph is not complete, $m < \frac{n(n-1)}{2}$

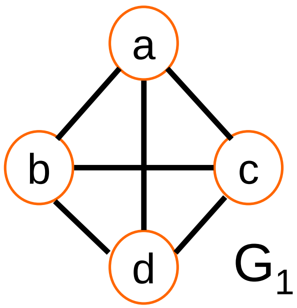


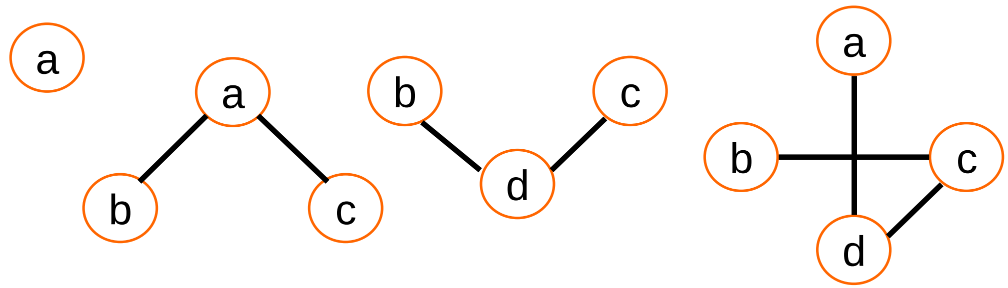
n=5;

m=$\frac{5\cdot 4}{2} = 10$

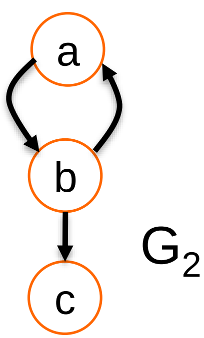
# Subgraph

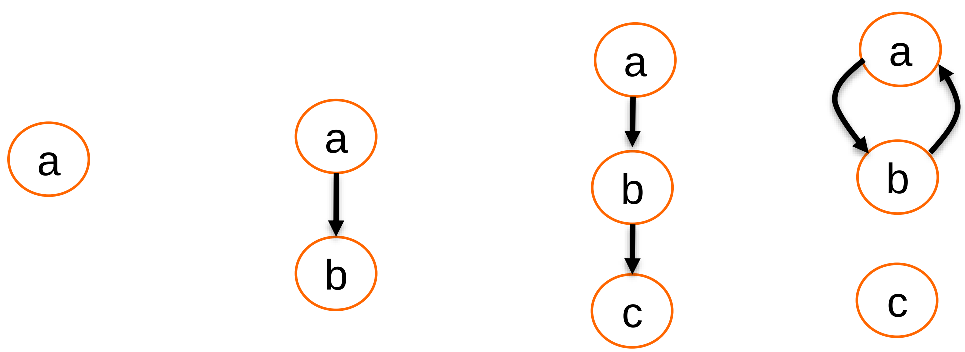
Subset of vertices and edges forming a graph




Some of the subgraphs of $G_1$

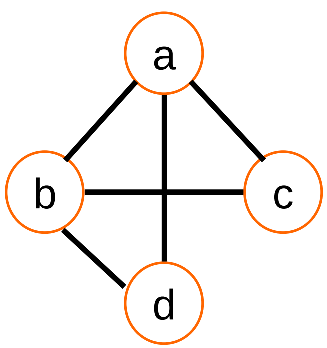


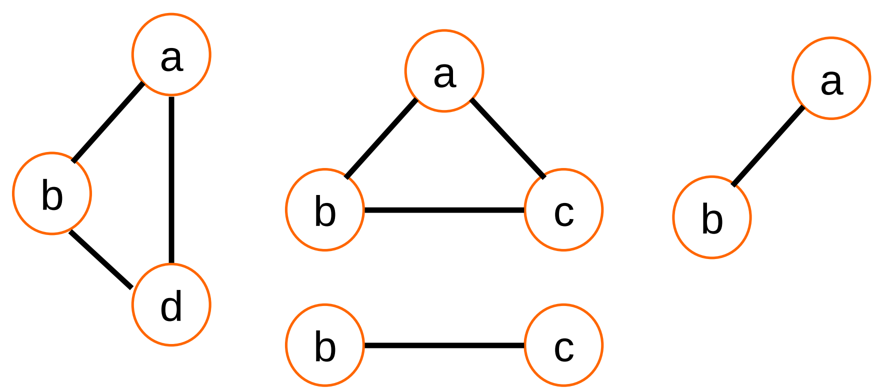

Some of the subgraphs of $G_2$

# 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 $G_1$

# Adjacency Matrix

- Let $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ be a graph with n vertices.
- The adjacency matrix $A$ of $G$ is a two-dimensional
  - $\mathrm{n} \times \mathrm{n}$ array, say adj_mat: `int *adj_mat[n][n];`
  - If the edge $(\mathrm{v_i}, v_j)$ is in $E(G)$: `adj_mat[i][j] = 1`
  - If there is no such edge in $E(G)$: `adj_mat[i][j] = 0`
- Properties:
  - The adjacency matrix for an undirected graph is symmetric: $A = A^T$
- The adjacency matrix for a directed graph need not be symmetric
- “1” in $< j, j >$ means there’s a self-loop in vertex *j*

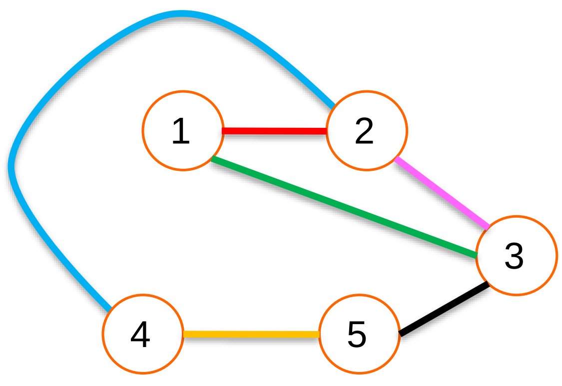
*Remember: Adjacent in directed graphs*

2 is adjacent to 1, but 1 is NOT adjacent to 2

# Adjacency Matrix

## Example : Undirected Graph

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

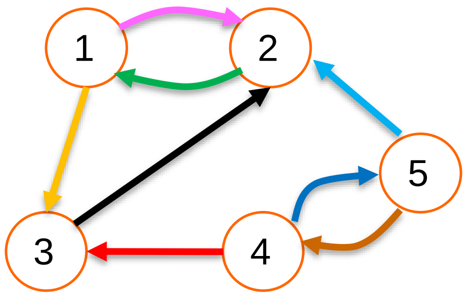



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & \boxed{1} & \boxed{1} & 0 & 0 \\ \boxed{1} & 0 & \boxed{1} & \boxed{1} & 0 \\ \boxed{1} & \boxed{1} & 0 & 0 & \boxed{1} \\ 0 & \boxed{1} & 0 & 0 & \boxed{1}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

- For undirected graph:
  - Two vertices $x, y$ are **adjacent** if $< x, y >$ is an edge
  - The adjacency matrix is symmetric

# Adjacency Matrix

## Example : Directed Graph



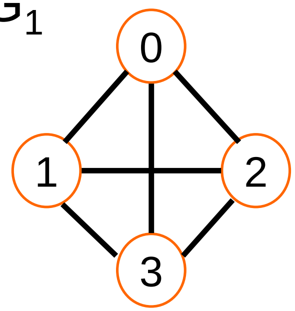

$$\begin{array}{c c} & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & \boxed{1} & \boxed{1} & 0 & 0 \\ \boxed{1} & 0 & 0 & 0 & 0 \\ 0 & \boxed{1} & 0 & 0 & 0 \\ 0 & 0 & \boxed{1} & 0 & \boxed{1} \\ 0 & \boxed{1} & 0 & \boxed{1}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

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)})

- For directed graphs
  - Vertex $w$ is **adjacent** to $v$ iff $(v, w) \in E$
    $\rightarrow$ there is a direct edge from $v$ to $w$
  - $w$ is successor of $v$
  - $v$ is predecessor of $w$
  - The adjacency matrix need not be symmetric

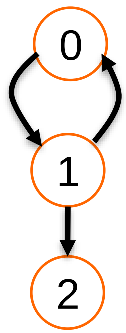
# Symmetry Examples in Adjacency Matrix

$G_1$




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

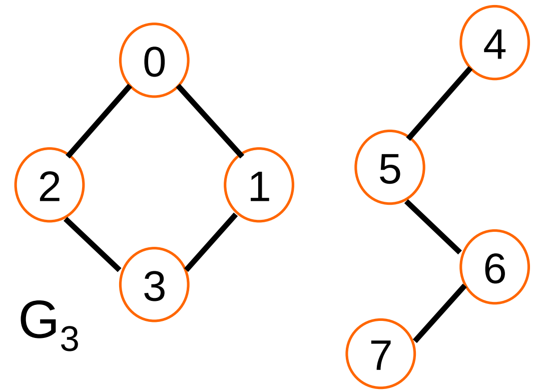
symmetric




$G_2$

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

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: $n^2$

```
enum visited {UNVISITED, VISITED}; // enum for marking vertices
// Graph class definition with adjacency matrix
class Graph{
    private:
        int numVertex, numEdge; // mumber of edges and vertices
        int **matrix; // pointer for adjacency matrix
        int *mark; //array of visited nodes
    public:
    Graph(int n); // Constructor
    ~Graph(); // Destructor
    int n() const; // number of vertices
    int e() const; // number of edges
    int first(int v); // first neighbor of vertex v
    int next(int v, int w); // v's next neighbor after w
    void setEdge(int v1, int v2, int wt); // set edge with weight wt
    void delEdge(int v1, int v2); // delete edge
    bool isEdge(int v1, int v2); // is (v1,vj) an edge?
    int weight(int v1, int v2); // get the weight of (v1,v2)
    int getMark(int v); // get the mark of vertex v
    void setMark(int v, int val); // set the mark
    void clearMark(); // clear all marks
};
```

```
// Constructor
Graph::Graph(int n){
        int i;
        numVertex = n;
        numEdge = 0;
        mark = new int[numVertex];      // 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. Size is numVertex*numVertex
        matrix = new int*[numVertex];
        for(i=0; i < numVertex; i++)
           matrix[i] = new int[numVertex];
        for(i=0; i < numVertex; i++) // Initialize to 0
                for (int j=0; j < numVertex; j++)
                         matrix[i][j] = 0;
}
// Destructor, Return dynamically allocated memory
Graph::~Graph(){
        delete[] mark;
        for (int i=0; i < numVertex; i++)
           delete[] matrix[i];
        delete [] matrix;
}
```

```
int Graph::n() const{ // Return number of vertices
    return numVertex;
}

int Graph::e() const{ // Return number of edges
    return numEdge;
}

int Graph::first(int v) {// Return first neighbor of v
     for (int i=0; i<numVertex; i++)
         if (matrix[v][i] != 0)
            return i;
     return numVertex; // Return n if none
}

int Graph::next(int v, int w) { // Return v's next neighbor after 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 Graph::setEdge(int v1, int v2, int wt) {
    assert(wt > 0);
     //http://www.cplusplus.com/reference/cassert/assert/
    if (matrix[v1][v2] == 0)
        numEdge++;
    matrix[v1][v2] = wt;
}

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

bool Graph::isEdge(int i, int j){ // Is (v1,v2) an edge?
    return matrix[i][j] != 0;
}

int Graph::weight(int v1, int v2){ // Return weight of (v1,v2)
    return matrix[v1][v2];
}
```

```
int Graph::getMark(int v) {// Get mark of vertex v
    return mark[v];
}

void Graph::setMark(int v, int val) {// Set mark of vertex v
    mark[v] = val;
}

void Graph::clearMark(void){// clear all marks
    for (int i=0; i<numVertex; i++)
        mark[i] = UNVISITED;
}
```

Note: For all of the methods that pass nodes, you can implement an additional check that determines if the node value is valid (less than `numVertex`)

# Merits of Adjacency Matrix

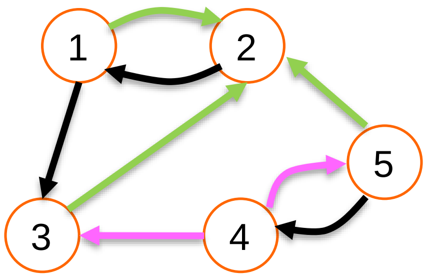
- Storage complexity: $O(|V|^2)$.
- Determining the connection of vertices is easy from the adjacency matrix.
- Edge existence query: O(1) (just array lookup)
- The degree of a vertex $i$ is

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

- For a directed graph (unweighted), the row sum is the out-degree, while the column sum is the in-degree

$$\text{in\_deg}(i) = \sum_{j=0}^{n-1} \text{adj\_mat}[j][i]$$

$$\text{out\_deg}(i) = \sum_{j=0}^{n-1} \text{adj\_mat}[i][j]$$




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

$\text{in\_deg}(2) = 3$

$\text{out\_deg}(4) = 2$

# But, ….

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

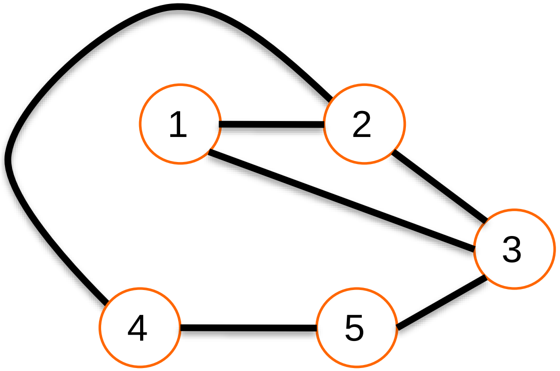
# Graph Representations

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

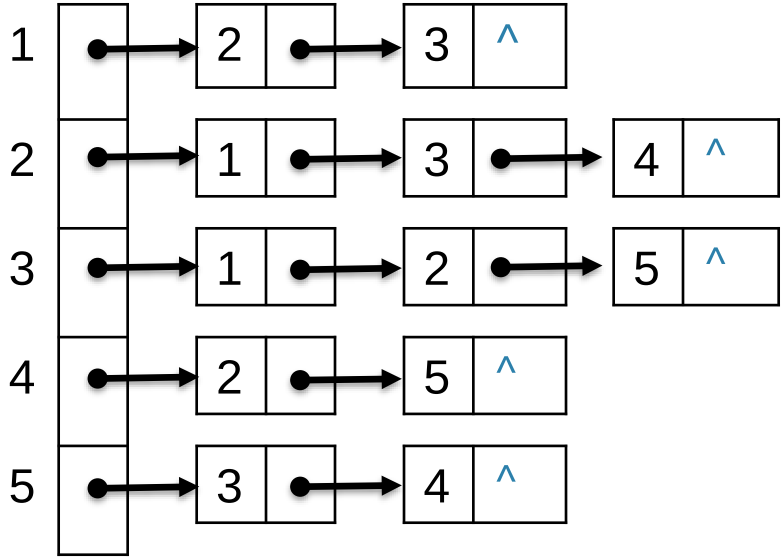
- **Adjacency Lists:** For each vertex we keep a list of adjacent vertices
  - appropriate when graph is sparse
  - $|E| << |V|^2$

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

# Adjacency List Example




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



# Adjacency List

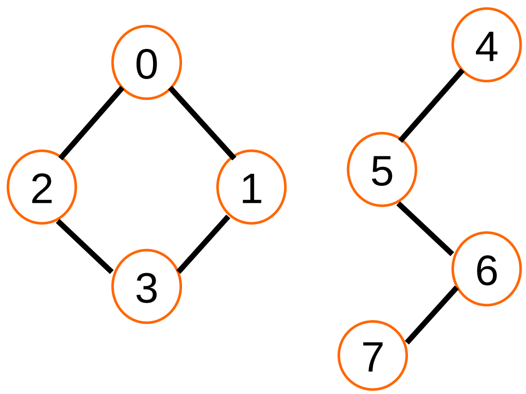
- A list of pointers, one for each node of the graph
- These pointers are the start of a linked list of nodes that can be reached by one edge of the graph
- Adj[u] is the list of all vertices adjacent to u
- List does not have to be sorted
- For a weighted graph, this list would also include the weight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

- Notice that

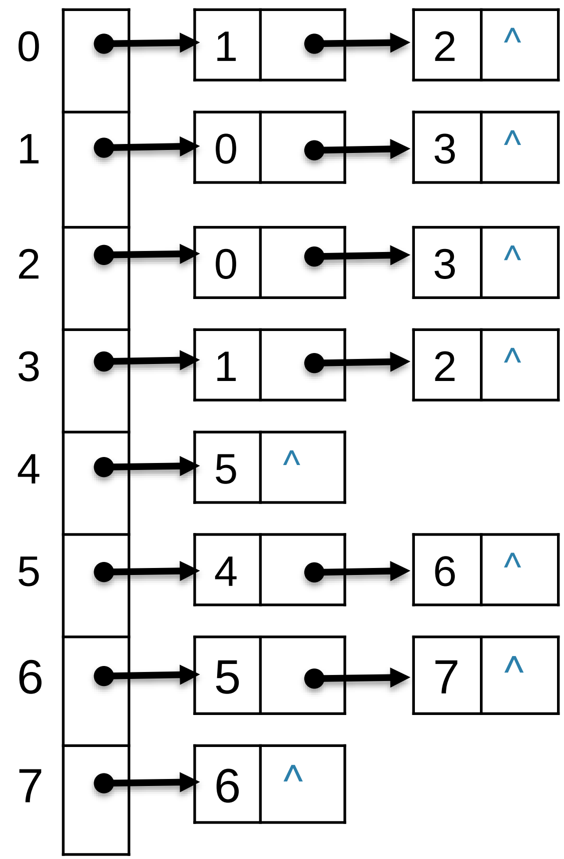
  Adjacency matrix is better if the graph is dense (many edges)

  Adjacency list is better if the graph is sparse (few edges)

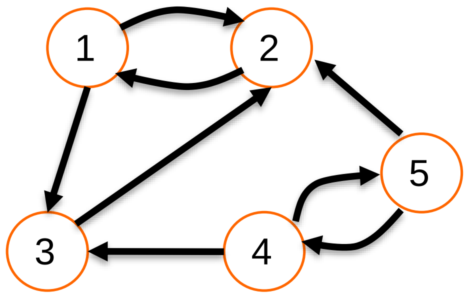
# Adjacency List Example



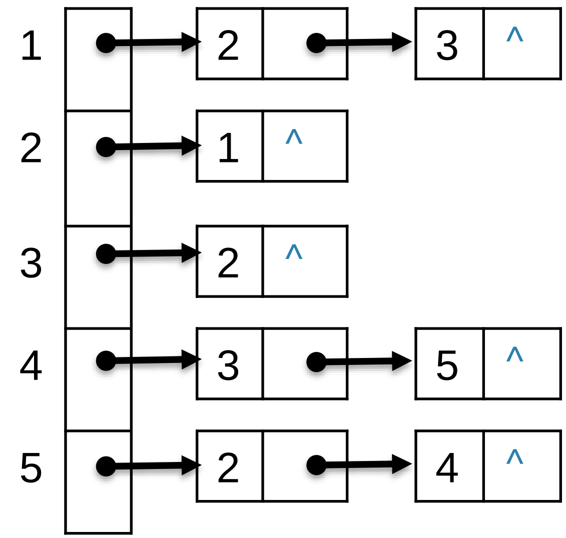

$G_3$



# 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: $O(|V| + 2 \cdot |E|) = O(|V| + |E|)$

Edge query check:

$O(|V| + |E|)$ 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

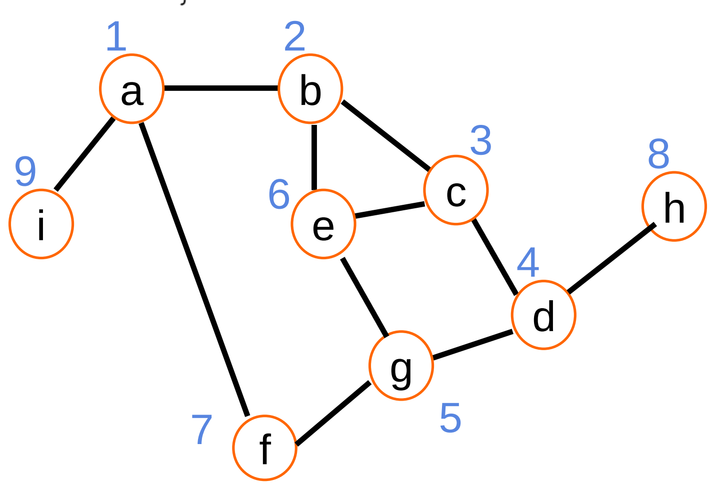
- A *graph-traversal* algorithm starts from a vertex $v$, visits all of the vertices that can be reachable from the vertex $v$
- A graph-traversal algorithm visits all vertices if and only if the graph is connected
- A **connected component** is the subset of vertices visited during a traversal algorithm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

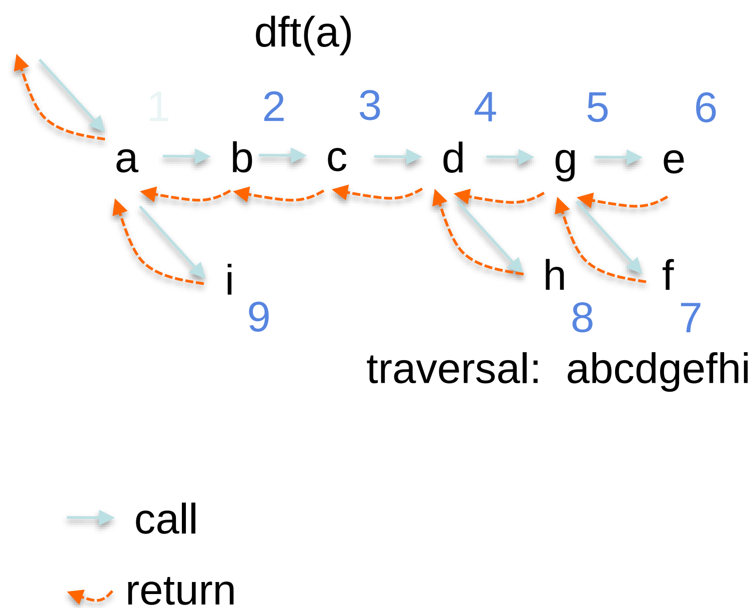
# Depth-First Traversal (DFT)

- 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)
}
```

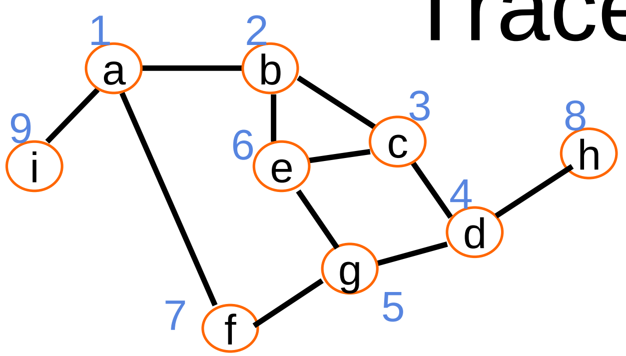


# Iterative Depth-First Traversal Algorithm

```
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 (all vertices that are adjacent
          to the vertex on the top of stack are visited)
        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

```
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 (all vertices that are adjacent
           to the vertex on the top of stack are visited)
         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;
      }
   }
}
```

| 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

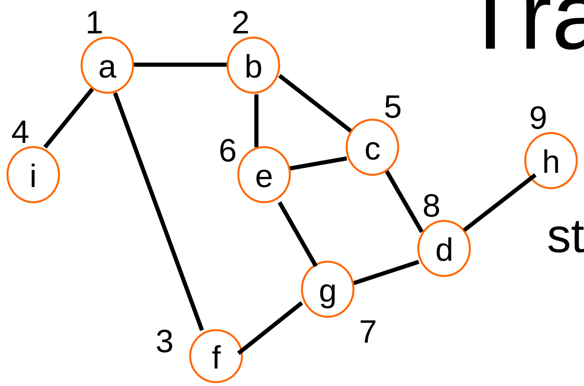
# Breadth-First Traversal (BFT, or BFS)

- 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()) {
     w = q.delete();
     for (each unvisited vertex u adjacent to w) {
        Mark u as visited;
        q.insert(u);
     }
  }
}
```

# Trace of Iterative BFT

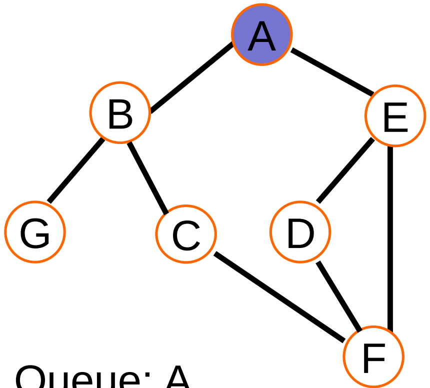



starting from vertex a

```
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()) {
     w= q.delete();
      for (each unvisited vertex u adjacent
   to  w) {
         Mark u as visited;
         q.insert(u);}
   }}
```

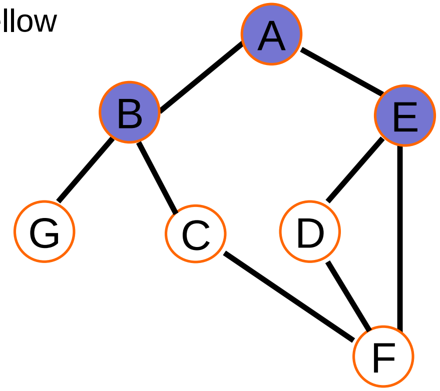
| node visited | Queue (front to back) | Visited |
|---|---|---|
| a | a | a |
| | (empty) | a |
| b | b | a,b |
| f | bf | a,b,f |
| i | bfi | a,b,f,i |
| | fi | a,b,f,i |
| c | fic | a,b,f,i,c |
| e | fice | a,b,f,i,c,e |
| | ice | a,b,f,i,c,e |
| g | iceg | a,b,f,i,c,e,g |
| | ceg | a,b,f,i,c,e,g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

the nodes visited (marked) are shown as yellow



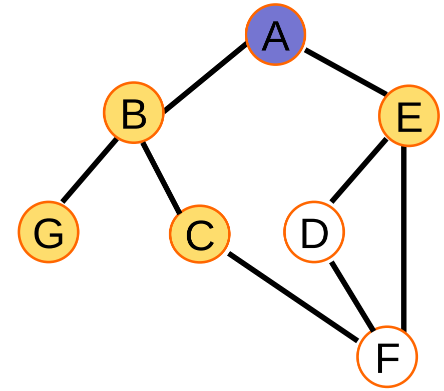

Queue: A

Start with A. Mark it




Queue: ABE

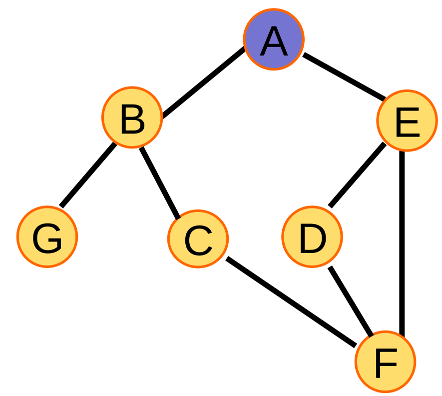
Expand A's adjacent vertices. Mark them and put them in queue.




Queue: ABECG

Now take B off queue, and queue its neighbors.

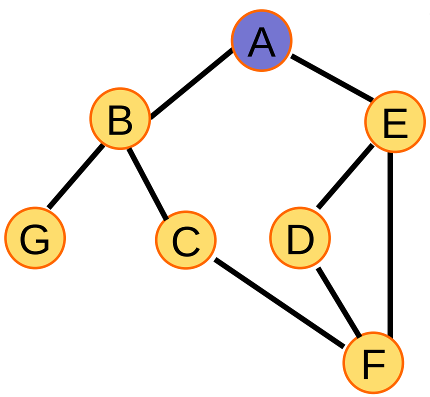
```
bft(in v:Vertex) {
// Traverses a graph beginning at vertex
   v iteratively
// by using breath-first strategy
   queue q;
   // add v to the queue and mark it
   q.insert(v);
   Mark v as visited;
   while (!q.isEmpty()) {
      w=q.delete();
      for (each unvisited vertex u
   adjacent to  w)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



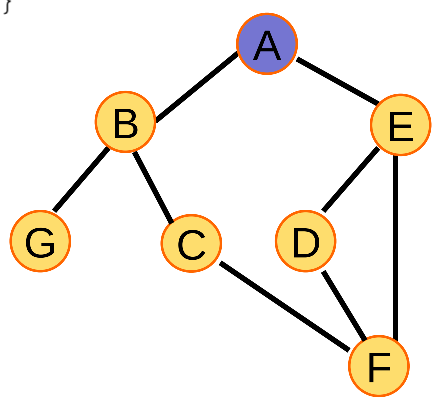

Queue: ABECGDF

Do same with E.

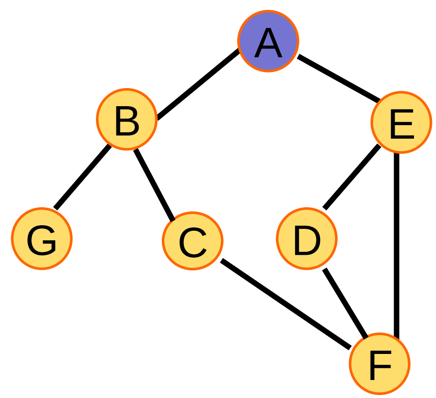
```
bft(in v:Vertex) {
// Traverses a graph beginning at vertex
   v iteratively
// by using breath-first strategy
   queue q;
   // add v to the queue and mark it
   q.insert(v);
   Mark v as visited;
   while (!q.isEmpty()) {
      w=q.delete( );
      for (each unvisited vertex u
   adjacent to  w)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



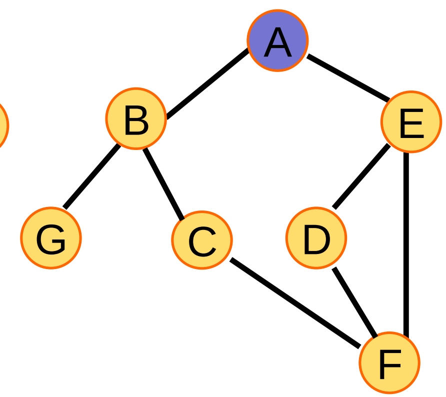

Queue: ABECGDF
Take C off queue.
Its neighbor F is already marked, so not queued.




Queue: ABECGDF
Take G off queue.

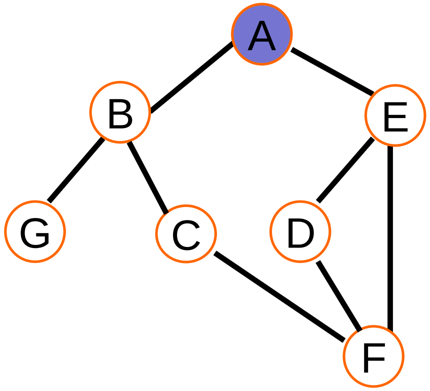



Queue: ABECGDF
Take D off queue.
F, E marked so not queued.



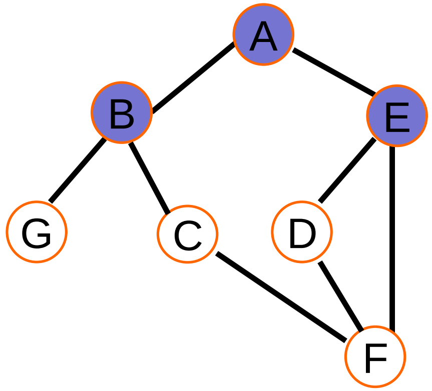

Queue:ABECGDF
Take F off queue.
E, D, C marked, so not queued again.

the nodes visited (marked) are shown as yellow



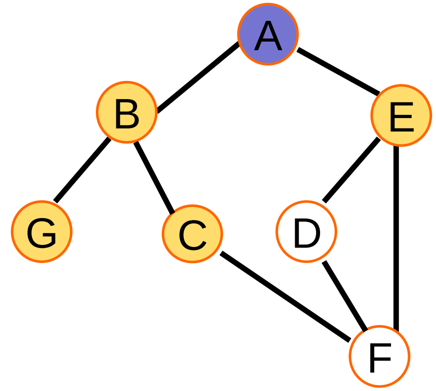

Queue: A

Start with A. Mark it



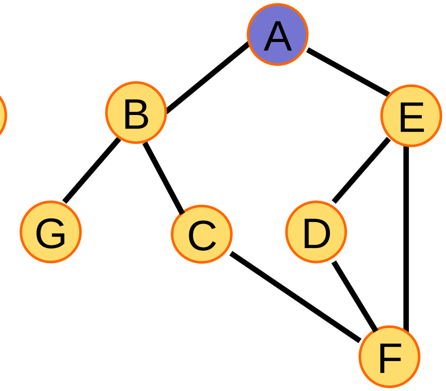

Queue: ABE

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



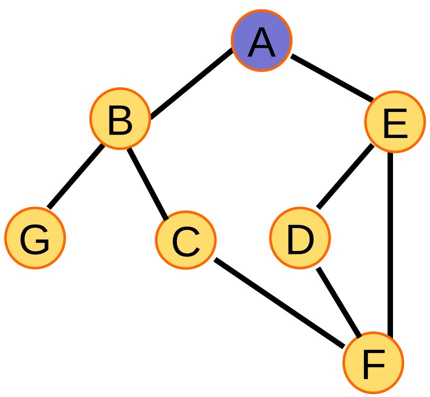

Queue: ABECG

Now take B off queue, and queue its neighbors.



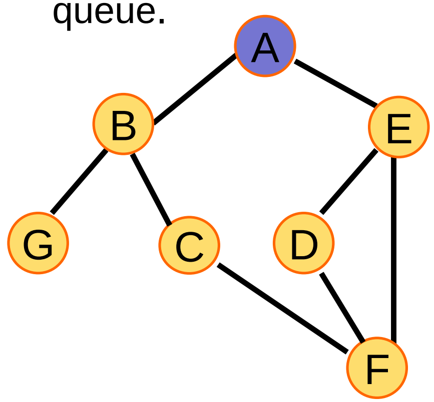

Queue: ABECGDF

Do same with E.



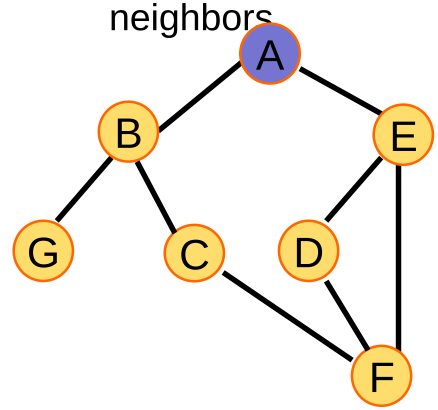

Queue: ABECGDF

Take C off queue.
Its neighbor F is already marked, so not queued.



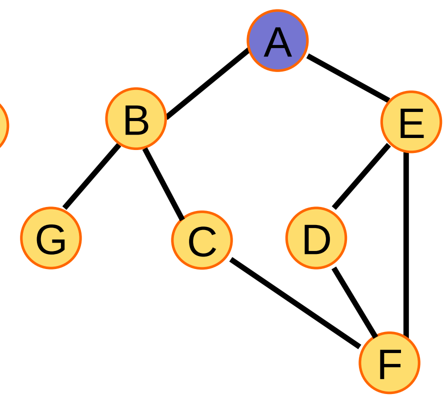

Queue: ABECGDF

Take G off queue.




Queue: ABECGDF

Take D off queue.
F, E marked so not queued.



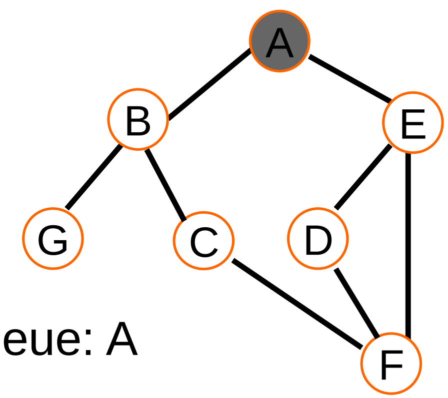

Queue:ABECGDF

Take F off queue.
E, D, C marked, so not queued again.

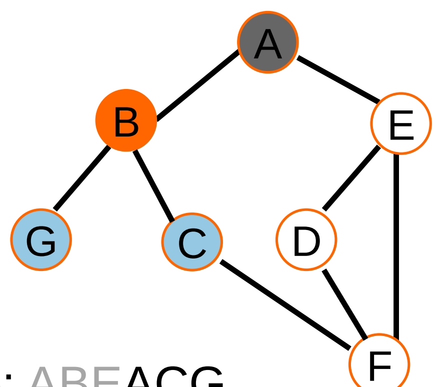
Marking the nodes is very important to handle cycles! What would happen if marking was not used?

Start with A.

Put in the queue (nodes in Queue are blue)



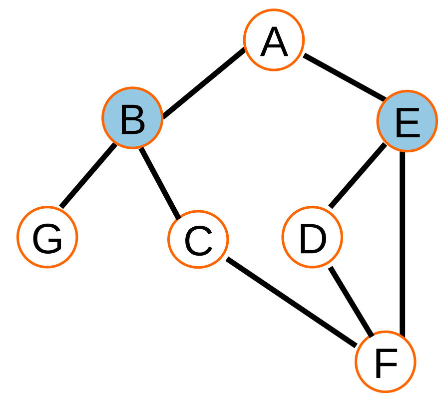

Queue: A




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




Queue: ABE

B and E are next

# 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 the shortest path from $s$ to each vertex

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

# Traversal Analysis - Summary

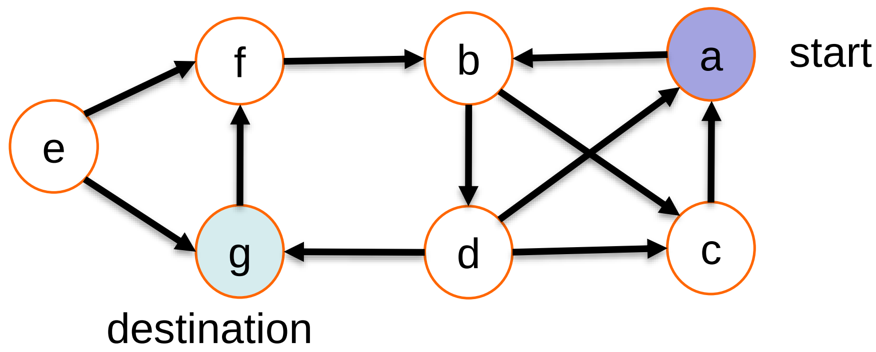
- Graph as state space (node = state, edge = action)
- BFS and DFS each search the state space for a best move
- If the graph is connected, these methods will visit each node exactly once
- If the search is exhaustive, they will find the same solution, but if there is a time limit and the search space 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)

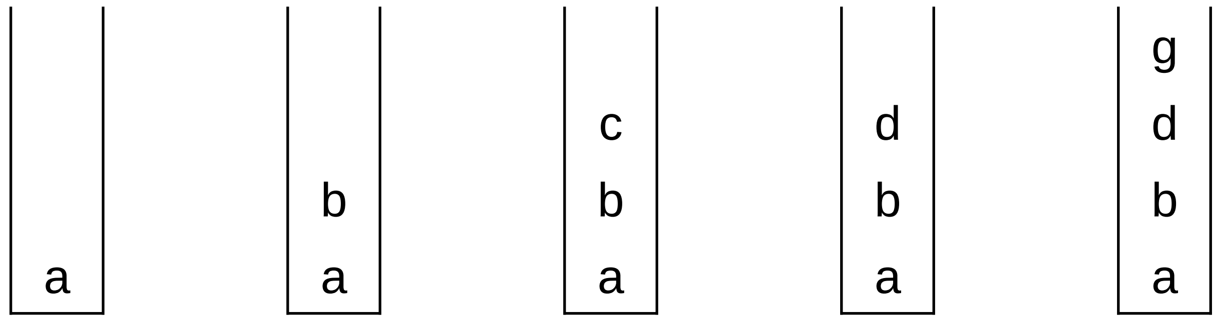
# Finding a Path

- 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 breadth – first search?

DFS Process

# DFS vs. BFS




DFS on a

DFS on b

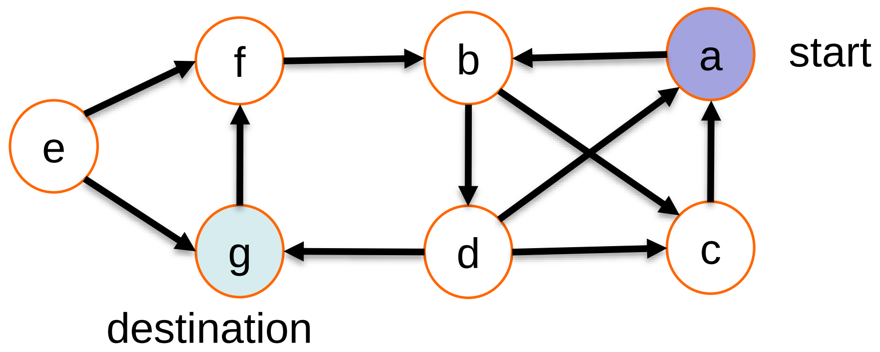
DFS on c

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 | rear front | rear front | rear front |
|---|---|---|---|
| a | b | c d | d |
| Initial call to BFS on a<br>Add a to queue | Dequeue a<br>Add b | Dequeue b<br>Add c, d | Dequeue c<br>Nothing to add |

rear front

g

Dequeue d
Add g

## Destination found - done!

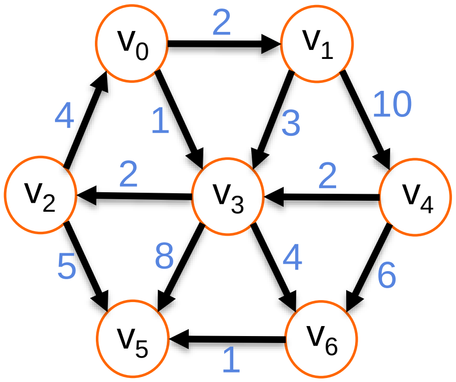
To extract the path, previous vertices must be remembered as the nodes are inserted into Queue. So follow the previous nodes starting from destination and insert in a stack until start node is reached. Then the path is in the stack

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

a
b
d
g

# 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




**Unweighted shortest path: all weights are 1**

# Dijkstra's algorithm

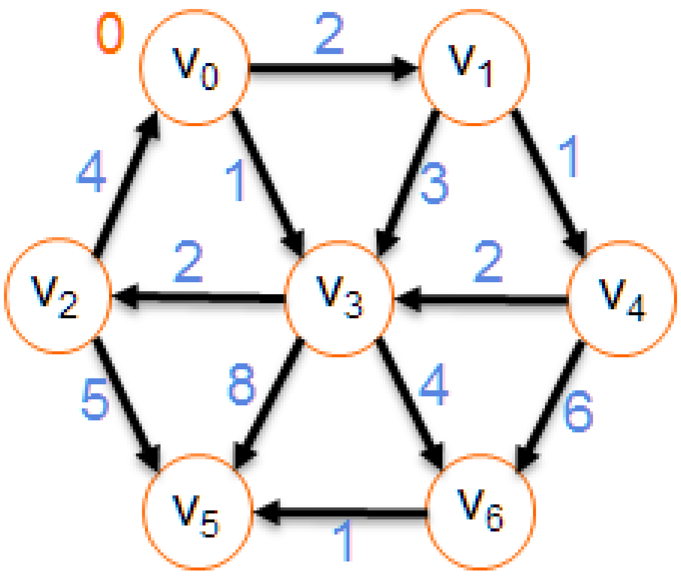
```
function Dijkstra(Graph, source):
// Initializations
for each vertex v in Graph:
   // Unknown dist from source to v
   dist[v] := infinity ;
   // Previous node in optimal path
   // from source
   previous[v] := undefined ;
end for
// Distance from source to itself
dist[source] := 0 ;
// All nodes 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 ;
```

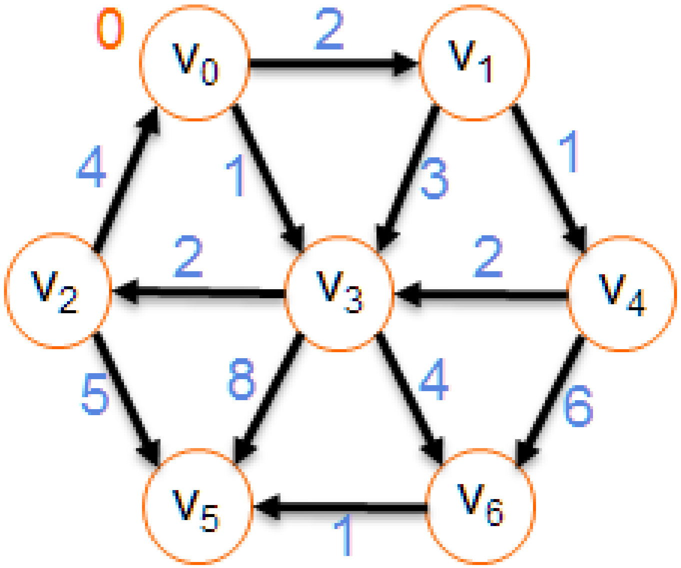



```
while Q is not empty:
  u := vertex in Q with min
  dist[] value;
  remove u from Q ; /***/

 if dist[u] = infinity then
  break;
 for each neighbor v of u:
 alt:=dist[u]+dist_between(u,v);
       if (alt < dist[v])
          dist[v] := alt ;
          previous[v] := u ;
```

$Q = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$

| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| previous | - | - | - | - | - | - | - |
| distance | 0 | inf | inf | inf | inf | inf | inf |



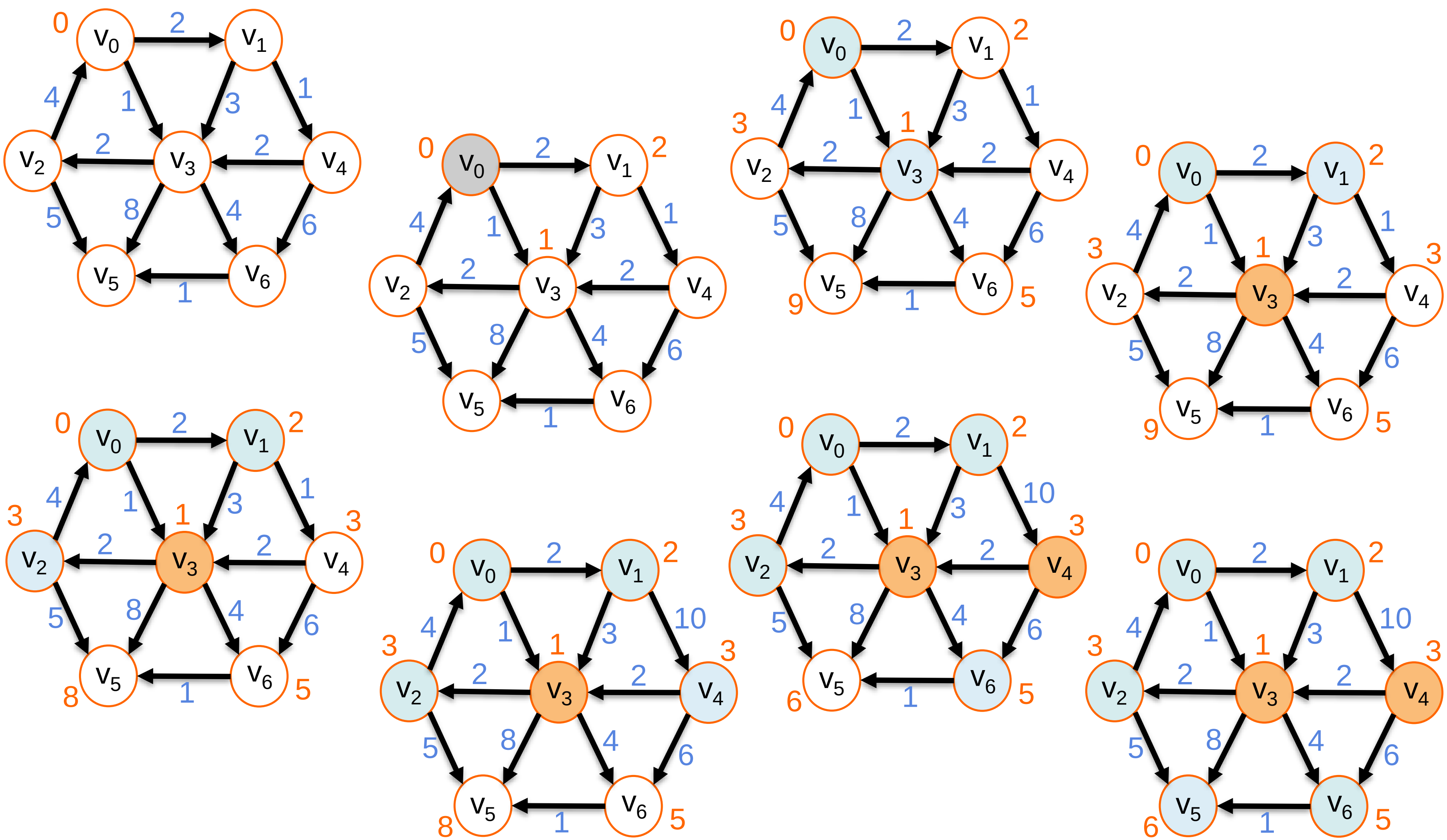

```
S := empty sequence
u := target
// Construct the shortest path
   with a stack S
while previous[u] is defined:
    // Push the vertex into the
   stack
    insert u at the beginning of
   S
    // Traverse from target to
   source
    u := previous[u]
end while ;
```

$Q = \{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$

| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| previous | - | 0 | 3 | 0 | 1 | 6 | 3 |
| distance | 0 | 2 | 3 | 1 | 3 | 6 | 5 |

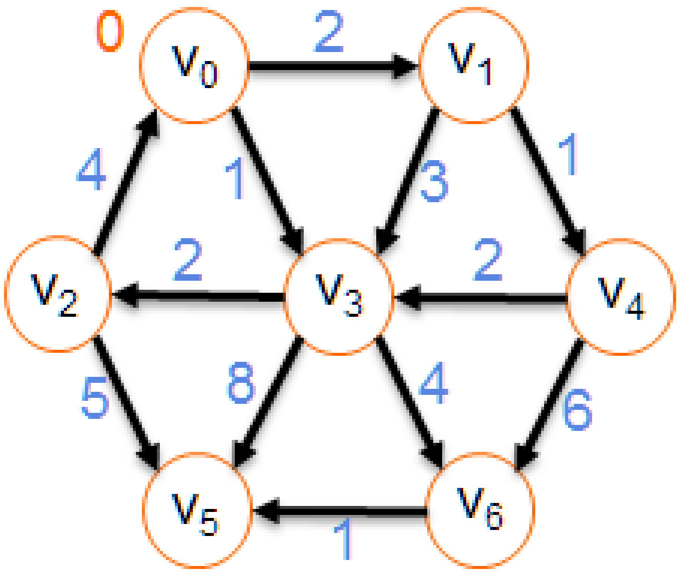
Example
Source:0
Target: 5
Construct the Path

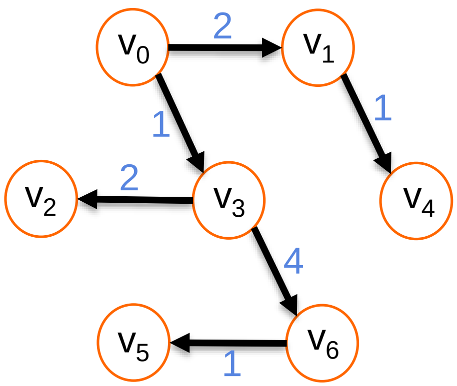
# Stages of Dijkstra's algorithm    ($s=v_0$)



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

# Spanning tree

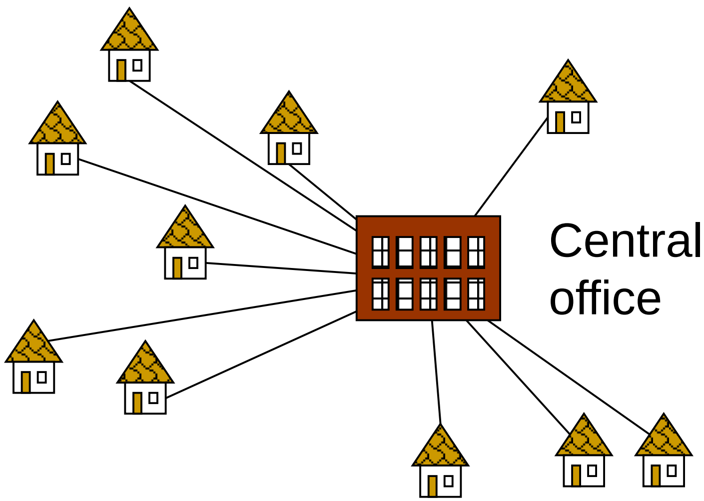


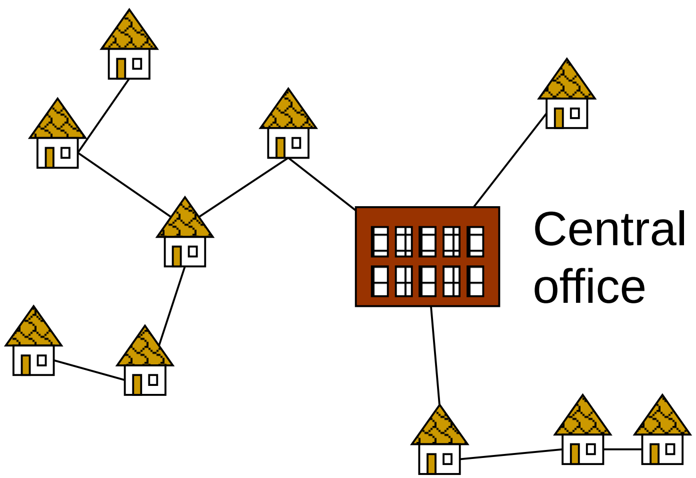




Spanning tree
starting from v0

| vertex | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| previous | - | 0 | 3 | 0 | 1 | 6 | 3 |
| distance | 0 | 2 | 3 | 1 | 3 | 6 | 5 |

# Minimum Spanning Tree Problem: Laying Telephone Wire



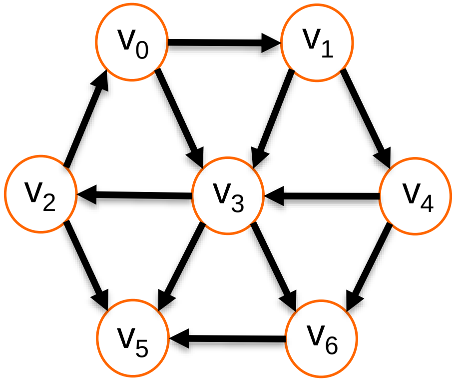




**Naïve Approach**
expensive

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

# Unweighted Shortest-Path problem

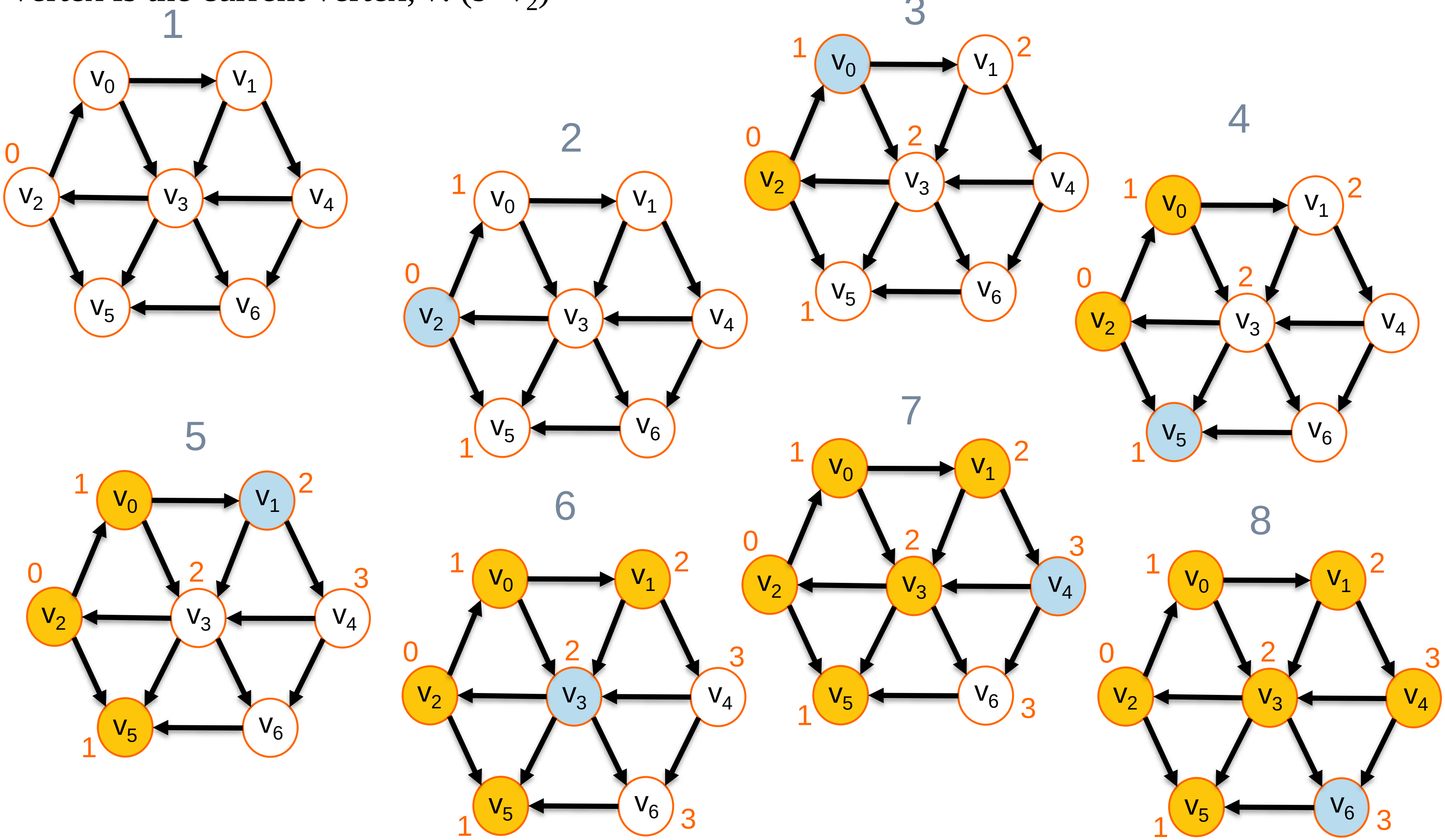
- *Find the shortest path (measured by number of edges) from a designated vertex S to every vertex*



# 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=$v_2$)



# EE 441 Data Structures

## Chapter 8: Graphs

Klaus Werner SCHMIDT

Uğur HALICI