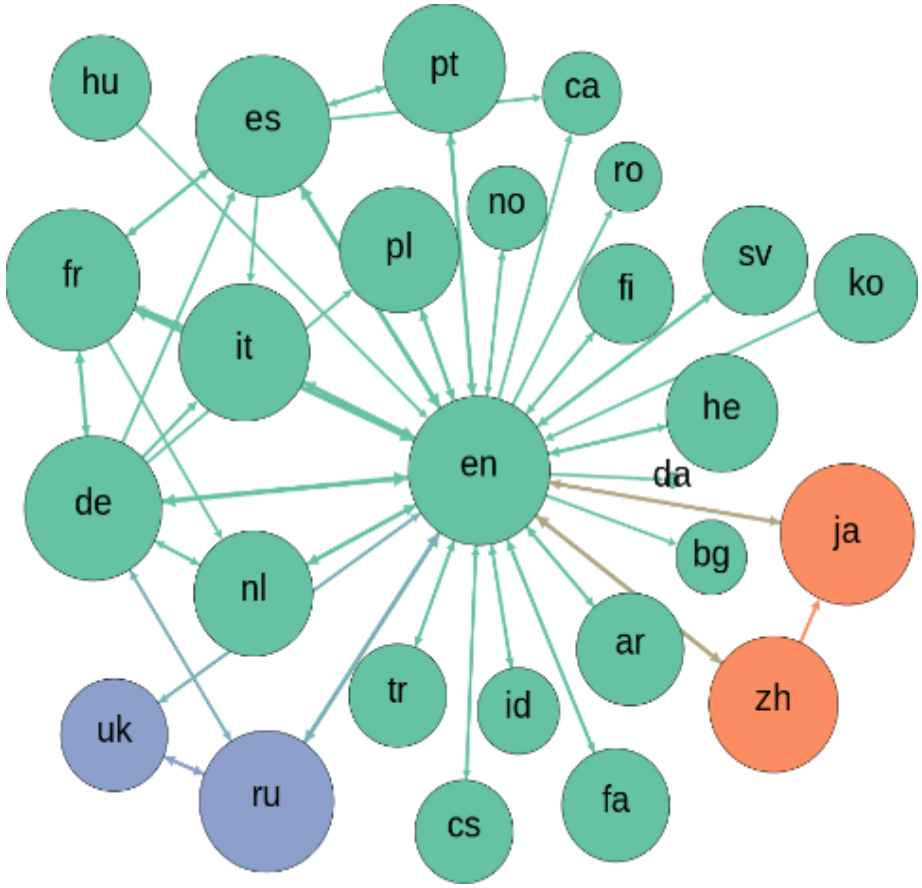



# Graph

Kiran Waghmare

**Source:** Wikipedia

# Graph

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices .
- A graph *G* is defined as follows:

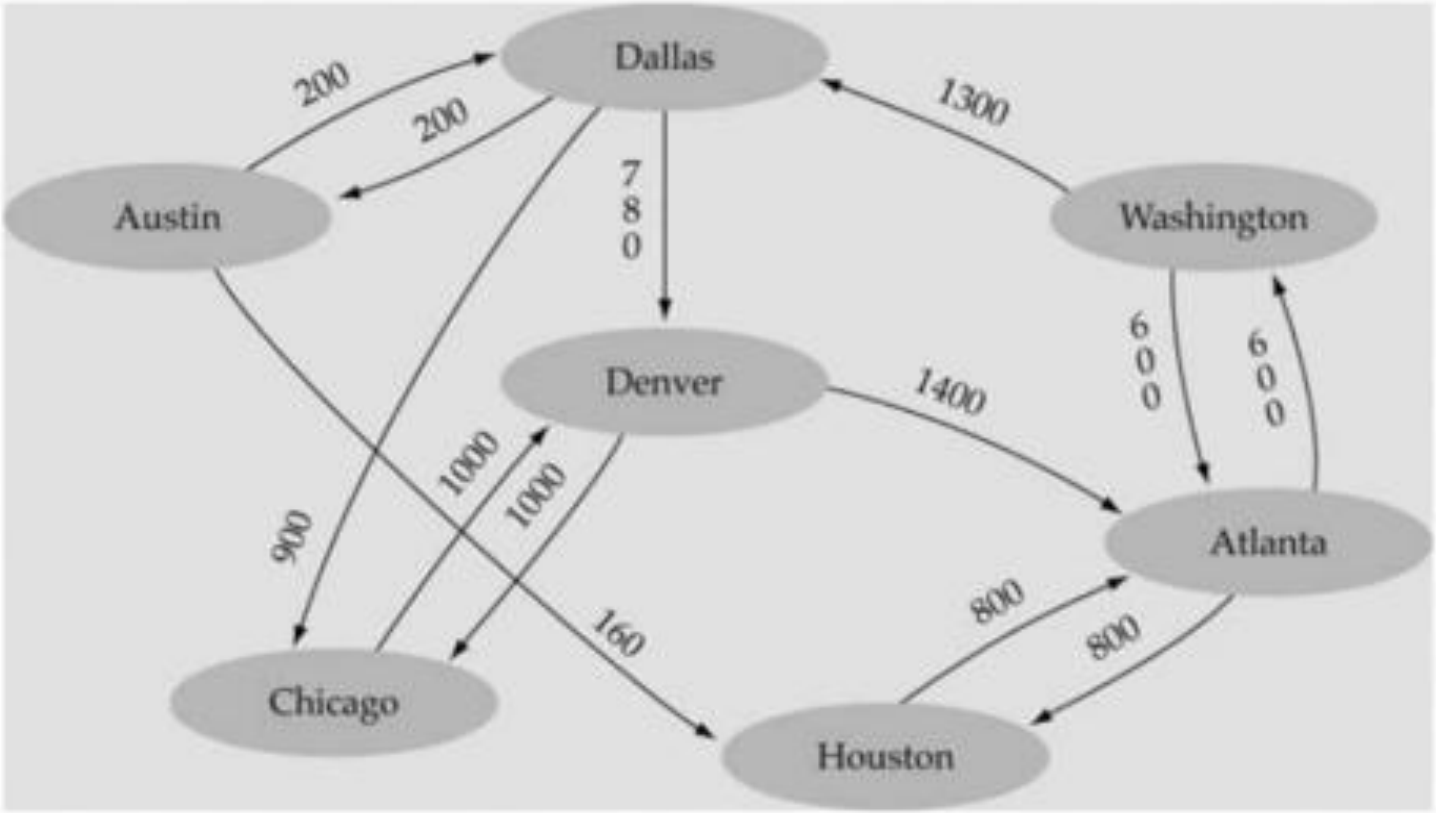
$$G=(V,E)$$

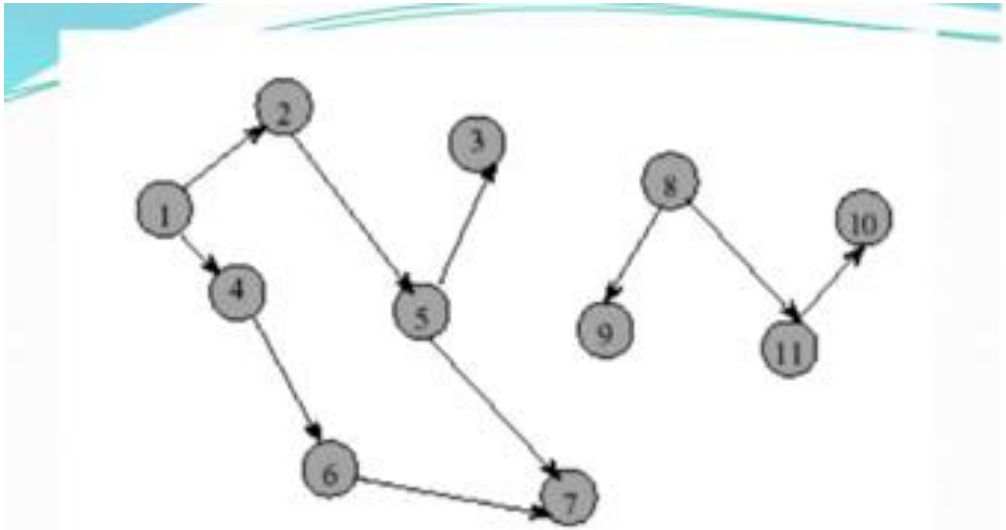
*V(G):* a finite, nonempty set of vertices

*E(G):* a set of edges (pairs of vertices)

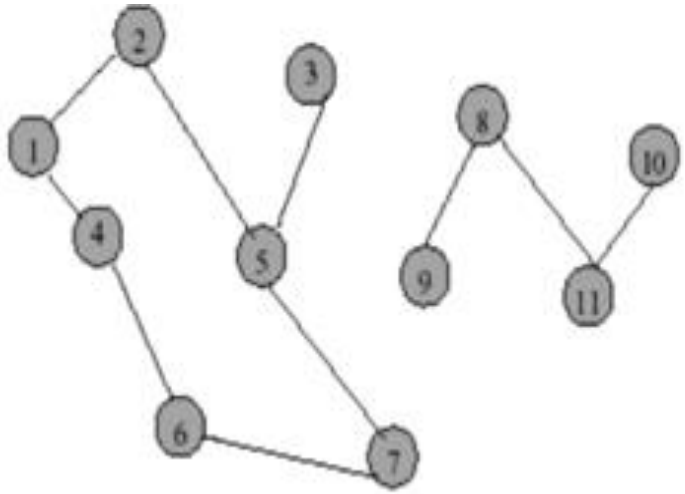
# Weighted graph:

-a graph in which each edge carries a value






Directed graph




Undirected graph

## Directed Graph

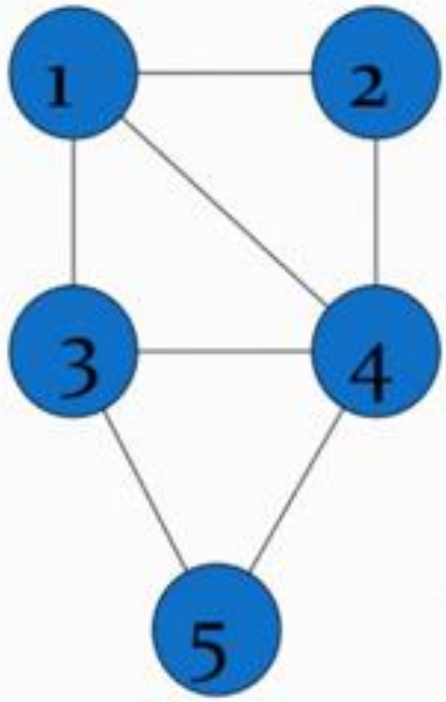
- Directed edge (i, j) is incident to vertex j and incident from vertex i
- Vertex i is adjacent to vertex j, and vertex j is adjacent from vertex i

# Graph Representation

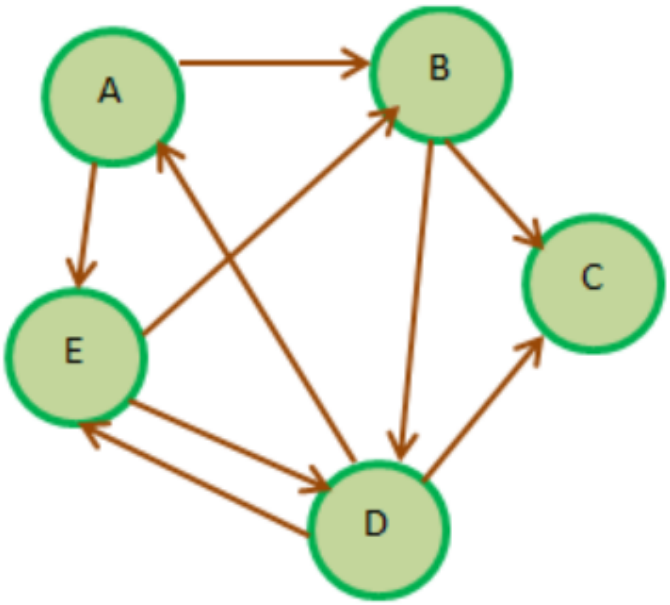
- For graphs to be computationally useful, they have to be conveniently represented in programs
- There are two computer representations of graphs:
  - Adjacency matrix representation
  - Adjacency lists representation

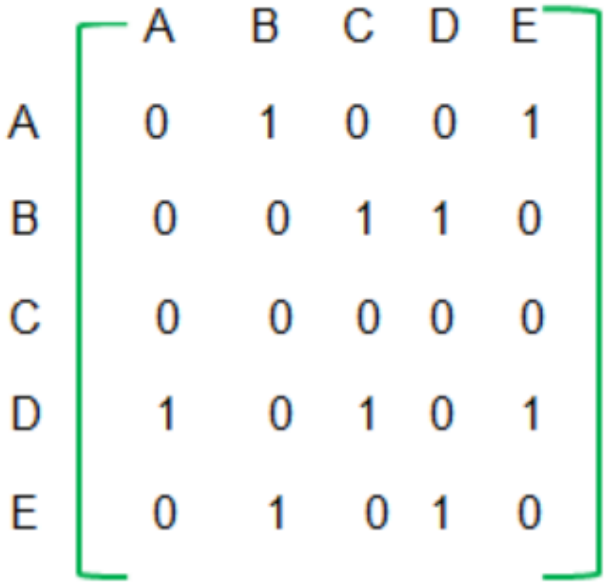
- *Adjacency Matrix*
  - A square grid of boolean values
  - If the graph contains N vertices, then the grid contains N rows and N columns
  - For two vertices numbered I and J, the element at row I and column J is true if there is an edge from I to J, otherwise false

# Adjacency Matrix




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 | 1 |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | 0 | 0 | 1 | 1 | 0 |



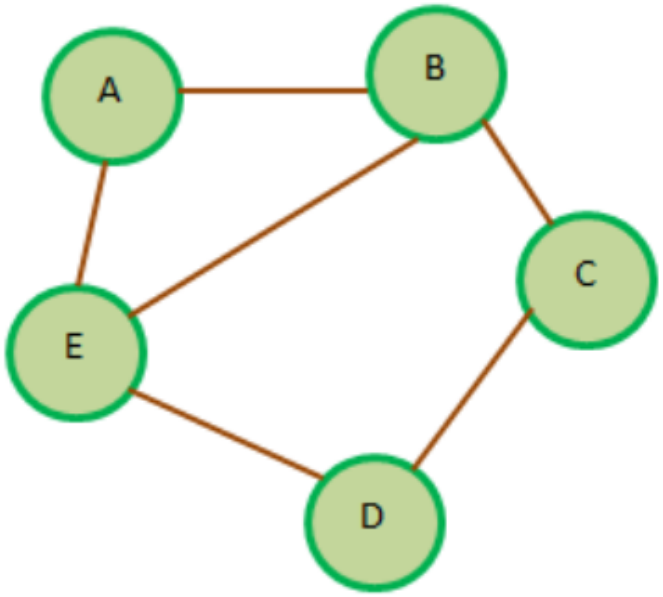
| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 0 | 1 |
| B | 0 | 0 | 1 | 1 | 0 |
| C | 0 | 0 | 0 | 0 | 0 |
| D | 1 | 0 | 1 | 0 | 1 |
| E | 0 | 1 | 0 | 1 | 0 |

*Directed graph*

*Adjacency* Matrix

# Graph representation: adjacency matrix

A graph with $n = |V|$ vertices, $v_1, \cdots, v_n$,




*Undirected graph*

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 0 | 1 |
| B | 1 | 0 | 1 | 0 | 1 |
| C | 0 | 1 | 0 | 1 | 0 |
| D | 0 | 0 | 1 | 0 | 1 |
| E | 1 | 1 | 0 | 1 | 0 |

*Adjacency* Matrix

Space: $O(n^2)$

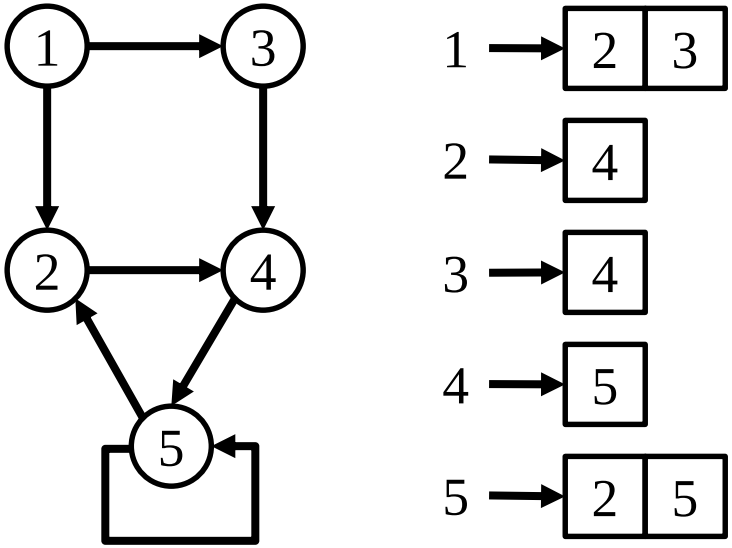
For undirected graphs, the matrix is symmetric.

# Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

# Graph representation: adjacency list

A graph can be represented by $|V|$ lists, one per vertex. The list for vertex $u$ holds the vertices connected to the outgoing edges from $u$.



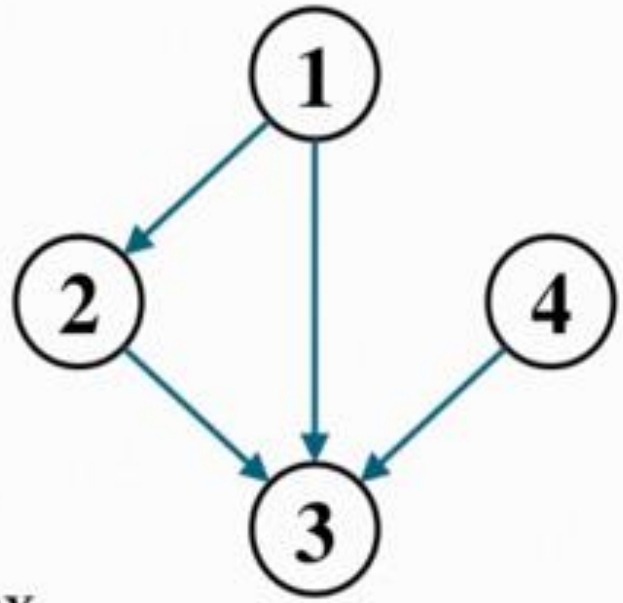

The lists can be implemented in different ways (vectors, linked lists, …)

Space: $O(|E|)$

**Undirected graphs:** use bi-directional edges

# Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to $v$
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex



# Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex $v$ = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is

$$\Sigma \text{ out-degree}(v) = |E|$$

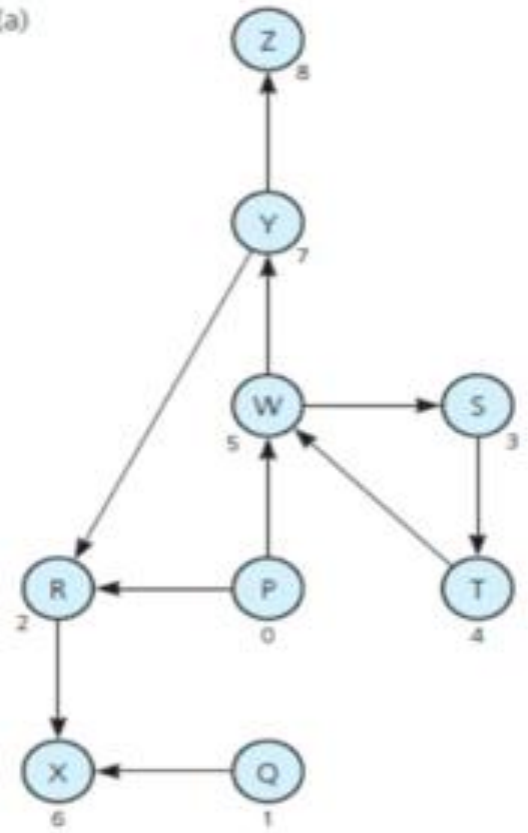
For undirected graphs, # items in adjacency lists is

$$\Sigma \text{ degree}(v) = 2\,|E|$$

- So: Adjacency lists take O(V+E) storage

# Implementing Graphs

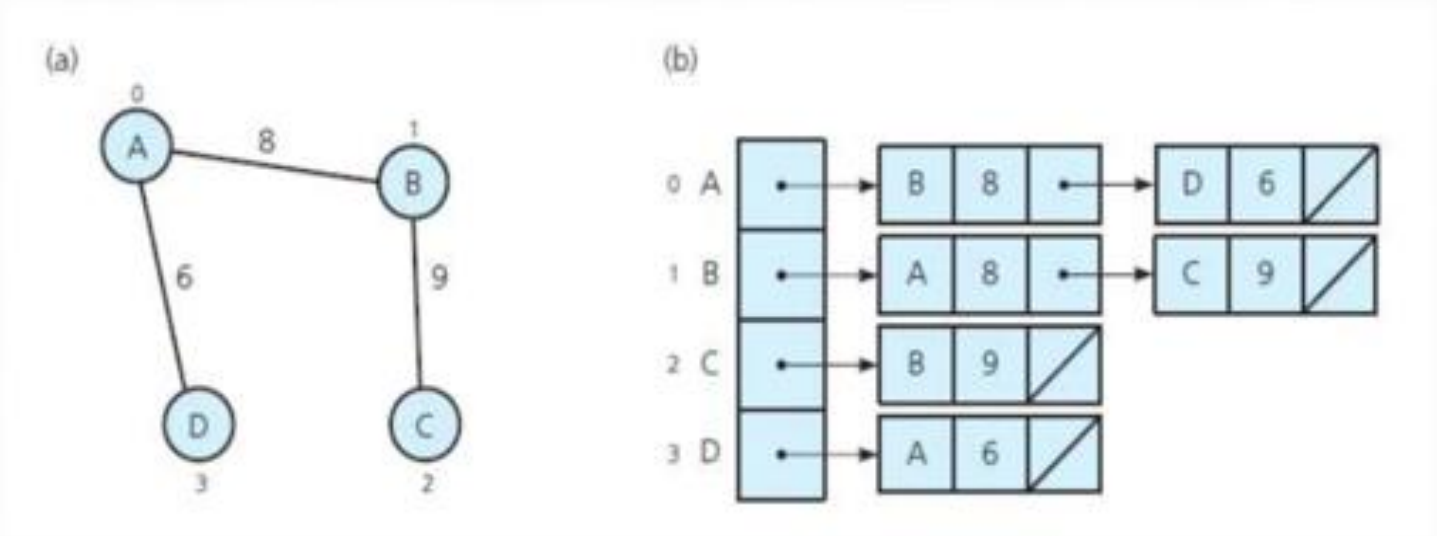
(a)




(b)

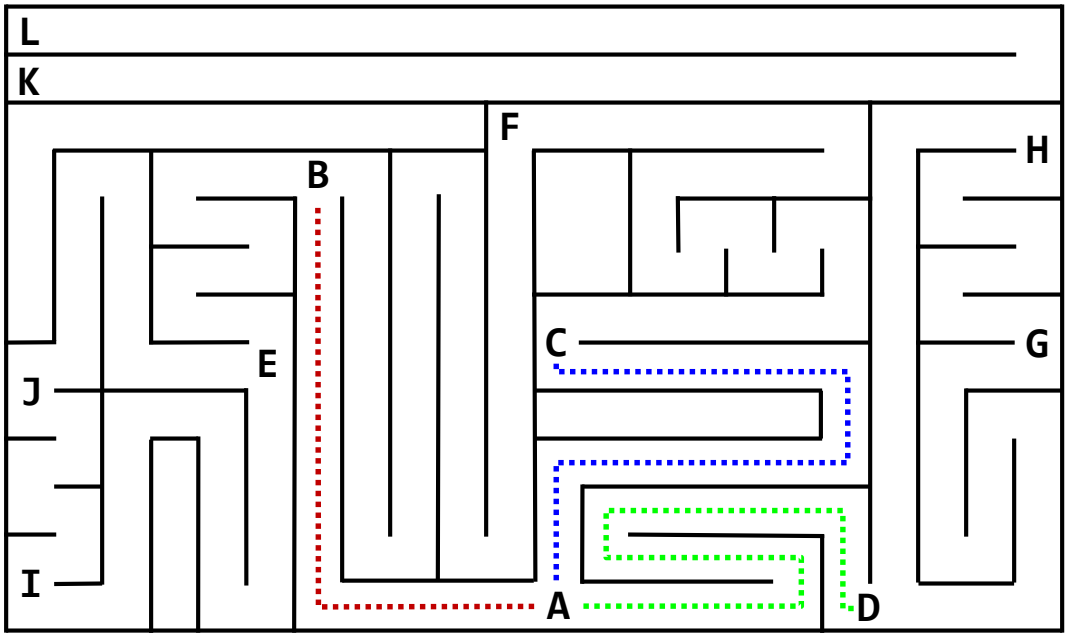
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | P | Q | R | S | T | W | X | Y | Z |
| 0 | P | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | Q | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | R | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | S | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 4 | T | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 5 | W | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 6 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | Y | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 8 | Z | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

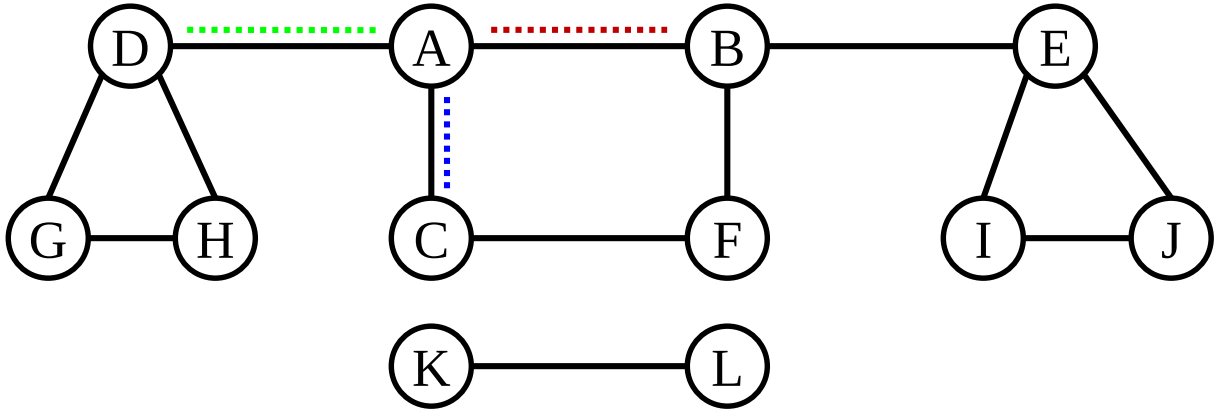
# Implementing Graphs




- (a) A weighted undirected graph and (b) its adjacency list

# Reachability: exploring a maze




Which vertices of the graph are reachable from a given vertex?

# Finding the nodes reachable from another node

- function explore($\boldsymbol{G}$, $\boldsymbol{v}$):
  // Input: $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is a graph
  // Output: visited($\boldsymbol{u}$) is true for all the
  // nodes reachable from $\boldsymbol{v}$

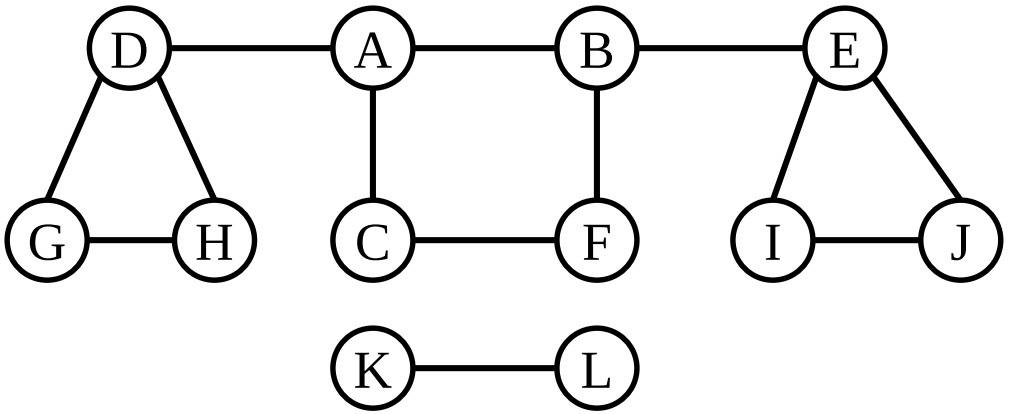
- visited($\boldsymbol{v}$) = true
- previsit($\boldsymbol{v}$)
- for each edge $(\boldsymbol{v}, \boldsymbol{u}) \in \boldsymbol{E}$:
- if not visited($\boldsymbol{u}$): explore($\boldsymbol{G}, \boldsymbol{u}$)
- postvisit($\boldsymbol{v}$)

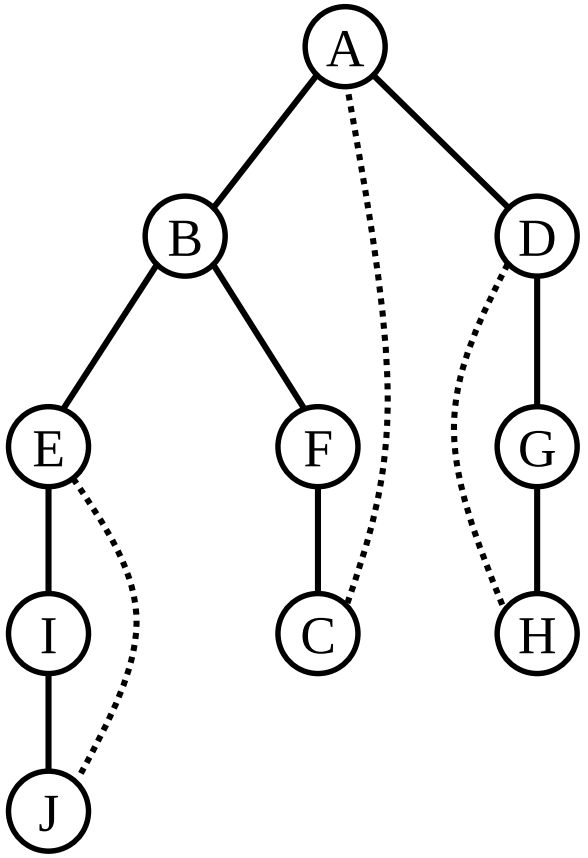
**Notes:**

- Initially, visited($v$) is assumed to be *false* for every $v \in V$.
- pre/postvisit functions are not required now.

# Finding the nodes reachable from another node

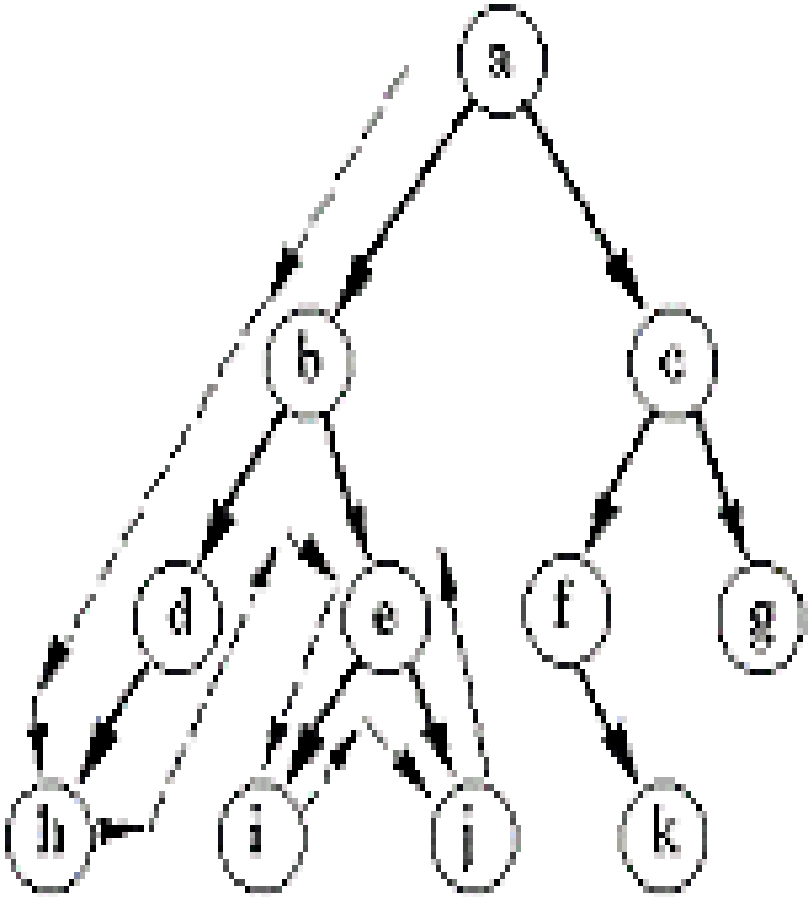
- function explore($\boldsymbol{G}$, $\boldsymbol{v}$):
  visited($\boldsymbol{v}$) = true
  for each edge $(\boldsymbol{v}, \boldsymbol{u}) \in \boldsymbol{E}$:
    if not visited($\boldsymbol{u}$): explore($\boldsymbol{G}, \boldsymbol{u}$)



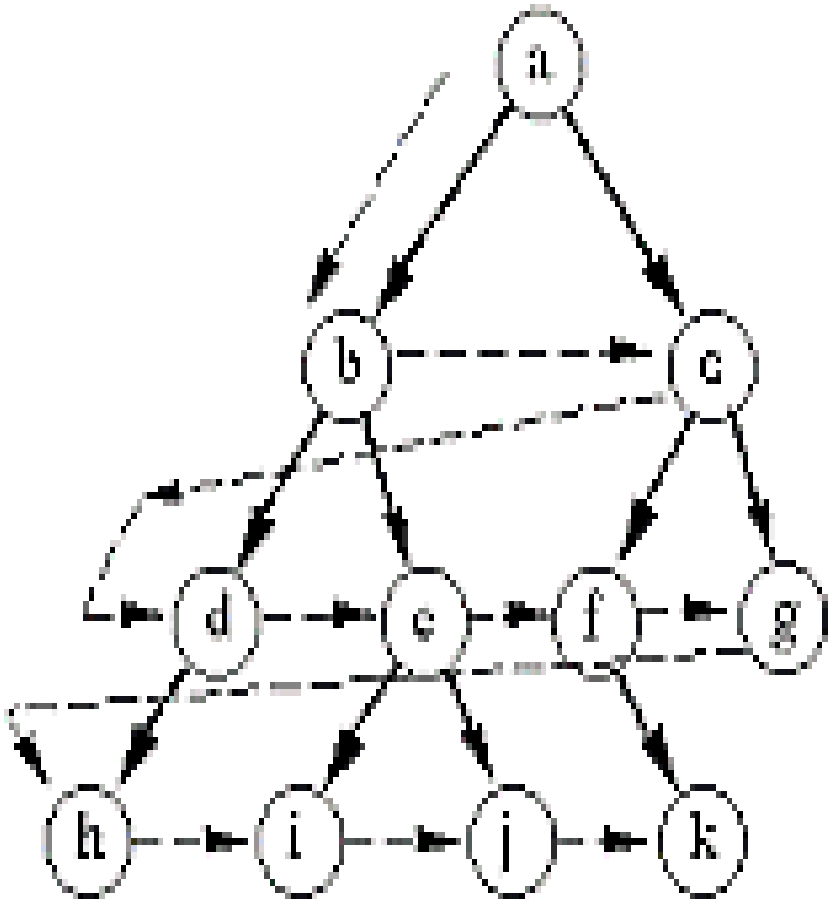




Dotted edges are ignored (*back edges*): they lead to previously visited vertices.

The solid edges (*tree edges*) form a tree.




Depth-first search




Breadth-first search

# Depth-first search

```
function DFS(G):
  for all v ∈ V:
    visited(v) = false
  for all v ∈ V:
    if not visited(v): explore(G, v)
```

DFS traverses the entire graph.

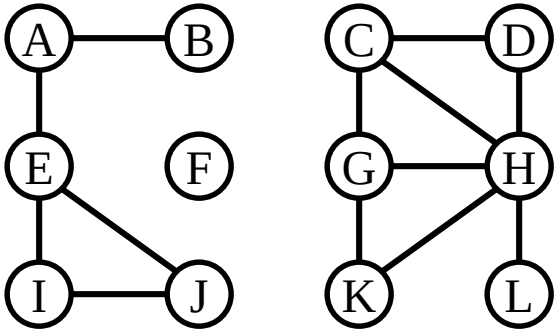
**Complexity:**

- Each vertex is visited only once (thanks to the chalk marks)
- For each vertex:
  - A fixed amount of work (pre/postvisit)
  - All adjacent edges are scanned

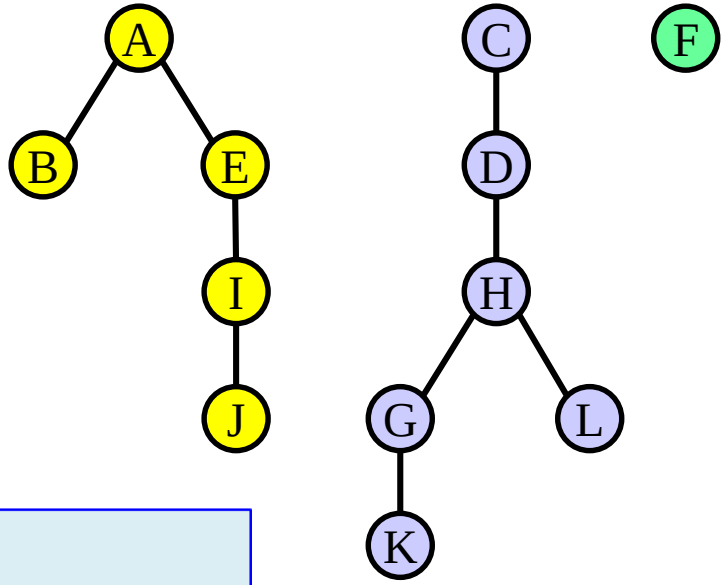
**Running time** is $O(|V| + |E|)$.
Difficult to improve: reading a graph already takes $O(|V| + |E|)$.

# DFS example

Graph




DFS forest



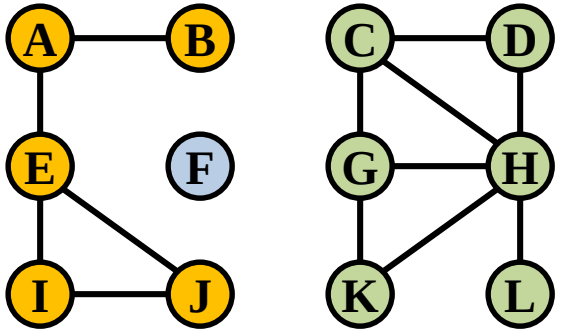

```
function DFS(G):
  for all v ∈ V:
    visited(v) = false
  for all v ∈ V:
    if not visited(v): explore(G,v)
```

- The outer loop of DFS calls ***explore*** three times (for A, C and F)
- Three trees are generated. They constitute a *forest*.

# Connectivity

- An undirected graph is connected if there is a path between any pair of vertices.

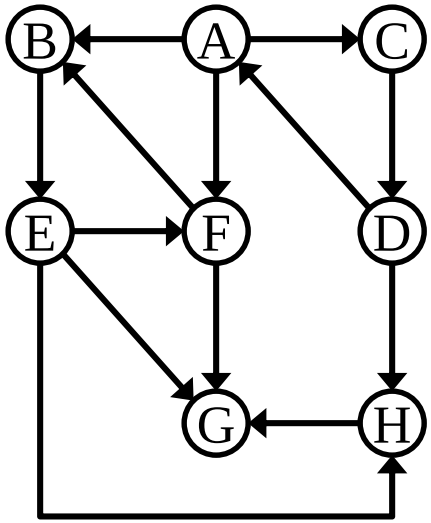
- A disconnected graph has disjoint *connected components*.




- Example: this graph has 3 connected components:

$$\{A, B, E, I, J\} \quad \{C, D, G, H, K, L\} \quad \{F\}.$$

# Cycles in graphs




A ***cycle*** is a circular path:

$$v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0.$$

Examples:

$$B \to E \to F \to B$$
$$C \to D \to A \to C$$

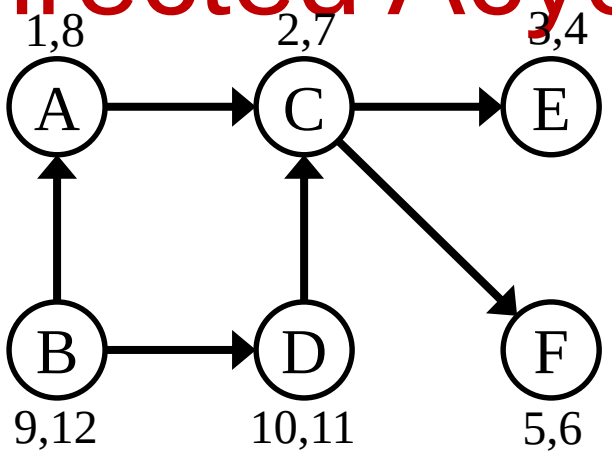
**Property:** A directed graph has a cycle *iff* its DFS reveals a back edge.

**Proof:**

$\Leftarrow$ If $(u, v)$ is a back edge, there is a cycle with $(u, v)$ and the path from $v$ to $u$ in the search tree.

$\Rightarrow$ Let us consider a cycle $v_0 \to v_1 \to v_2 \to \cdots \to v_k \to v_0$. Let us assume that $v_i$ is the first discovered vertex (lowest pre number). All the other $v_j$ on the cycle are reachable from $v_i$ and will be its descendants in the DFS tree. The edge $v_{i-1} \to v_i$ leads from a vertex to its ancestor and is thus a back edge.

# Directed Acyclic Graphs (DAGs)

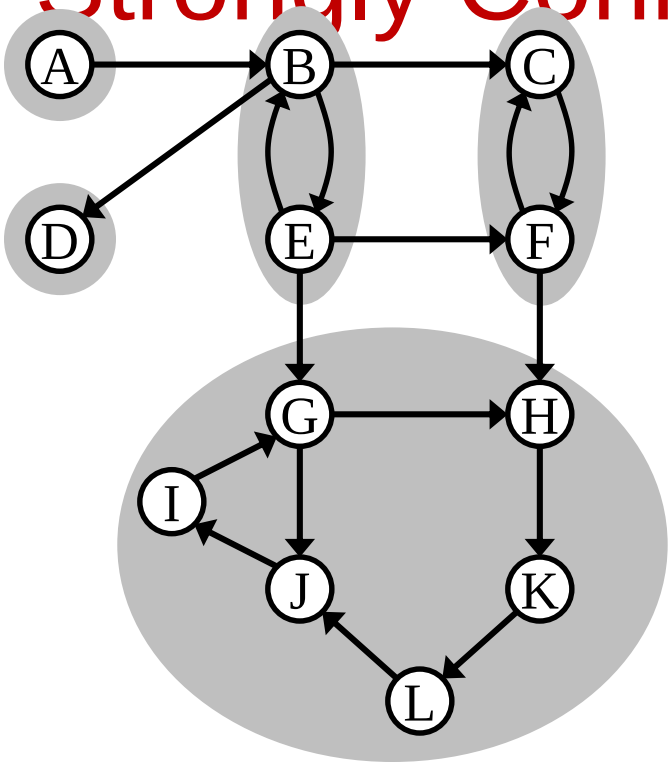



A **DAG** is a directed graph without cycles.

DAGs are often used to represent causalities or temporal dependencies, e.g., task A must be completed before task C.

- Cyclic graphs cannot be linearized.
- All DAGs can be linearized. How?
  - Decreasing order of the post numbers.
  - The only edges $(u, v)$ with post[$u$] < post[$v$] are back edges (do not exist in DAGs).
- **Property:** In a DAG, every edge leads to a vertex with a lower post number.
- **Property:** Every DAG has at least one source and at least one sink. (source: highest post number, sink: lowest post number).

# Strongly Connected Components




The graph is not *strongly connected*.

Two nodes $u$ and $v$ of a directed graph are connected if there is a path from $u$ to $v$ and a path from $v$ to $u$.

The *connected* relation is an equivalence relation and partitions $V$ into disjoint sets of *strongly connected components*.

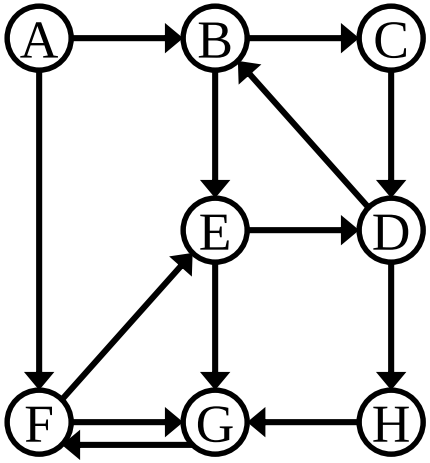
This graph is connected (undirected view), but there is no path between any pair of nodes.

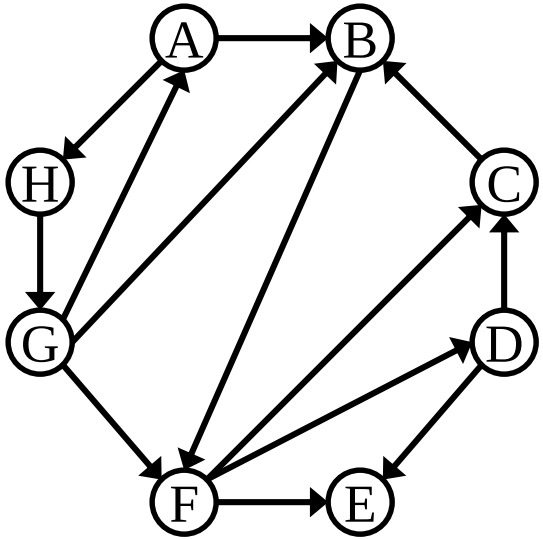
For example, there is no path $K \rightarrow \cdots \rightarrow C$ or $E \rightarrow \cdots \rightarrow A$.

**Strongly Connected Components**

$\{A\}$
$\{B, E\}$
$\{C, F\}$
$\{D\}$
$\{G, H, I, J, K, L\}$

# DFS (from [DPV2008])



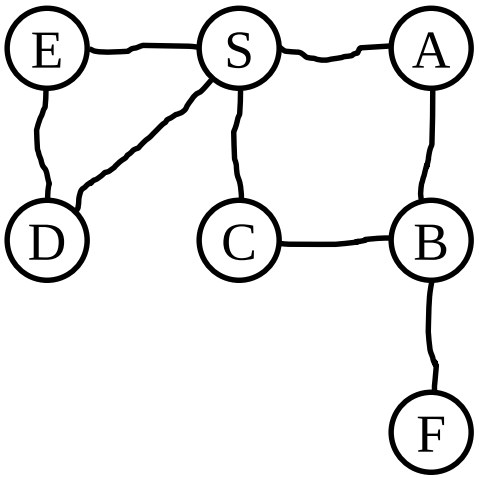




Perform DFS on the two graphs. Whenever there is a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge or cross edge, and give the `pre` and `post` number of each vertex.

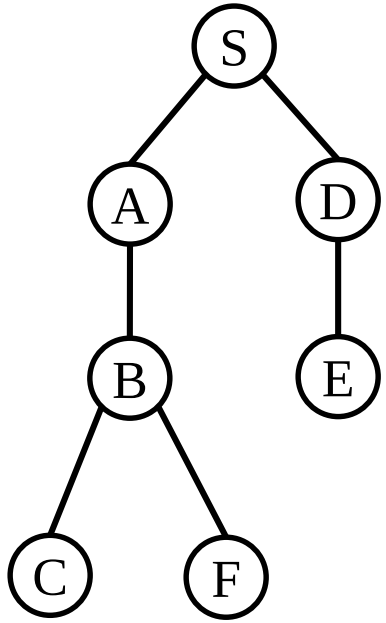
# Distance in a graph

Depth-first search finds vertices reachable from another given vertex. The paths are not the shortest ones.

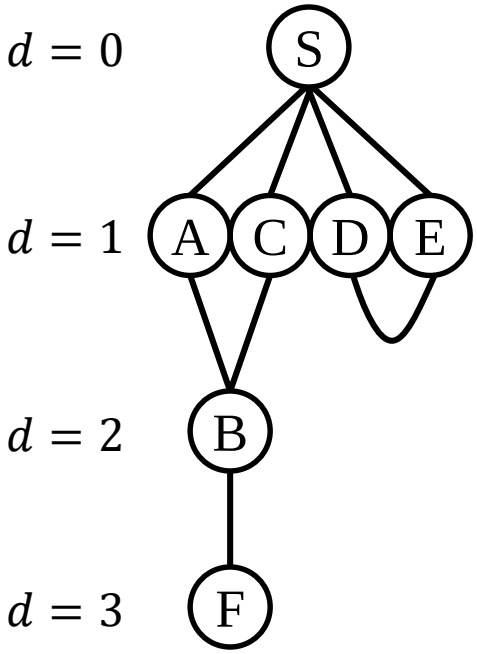
Graph




DFS tree




Shortest distance

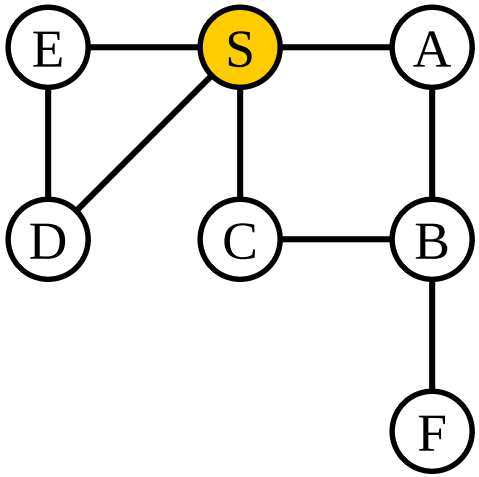



Distance between two nodes: length of the shortest path between them

# BFS algorithm

- BFS visits vertices layer by layer: $0,1,2,\ldots,d$.
- Once the vertices at layer $d$ have been visited, start visiting vertices at layer $d+1$.
- Algorithm with two active layers:
  - Vertices at layer $d$ (currently being visited).
  - Vertices at layer $d+1$ (to be visited next).
- Central data structure: a queue.

# BFS algorithm




| | | S | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|
| | $S_0$ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| $S_0$ | $A_1\ C_1\ D_1\ E_1$ | 0 | 1 | ∞ | 1 | 1 | 1 | ∞ |
| $A_1$ | $C_1\ D_1\ E_1\ B_2$ | 0 | 1 | 2 | 1 | 1 | 1 | ∞ |
| $C_1$ | $D_1\ E_1\ B_2$ | 0 | 1 | 2 | 1 | 1 | 1 | ∞ |
| $D_1$ | $E_1\ B_2$ | 0 | 1 | 2 | 1 | 1 | 1 | ∞ |
| $E_1$ | $B_2$ | 0 | 1 | 2 | 1 | 1 | 1 | ∞ |
| $B_2$ | $F_3$ | 0 | 1 | 2 | 1 | 1 | 1 | 3 |
| $F_3$ | | 0 | 1 | 2 | 1 | 1 | 1 | 3 |

# BFS algorithm

- function BFS($\boldsymbol{G}$, $\boldsymbol{s}$)

```
// Input: Graph G(V, E), source vertex s.
// Output: For each vertex u, dist[u] is
//         the distance from s to u.

  for all u ∈ V: dist[u]= ∞

  dist[s] = 0
  Q = {s} // Queue containing just s
  while not Q.empty():
    u = Q.pop_front()
    for all (u, v) ∈ E:
      if dist[v] = ∞:
        dist[v] = dist[u] + 1
        Q.push_back(v)
```

**Runtime** $O(|V| + |E|)$: Each vertex is visited once, each edge is visited once (for directed graphs) or twice (for undirected graphs).

# Reachability: BFS vs. DFS

**Input:** A graph $G$ and a source node $s$.

**Output:** $\forall u \in V$: reached[$u$] $\Leftrightarrow$ $u$ is reachable from $s$.

```
function BFS(G, s)

  for all u ∈ V:
    reached[u] = false

  Q = ⊏⊐ // Empty queue
  Q.push_back(s)
  reached[s] = true
  while not Q.empty():
    u = Q.pop_front()
    for all (u, v) ∈ E:
      if not reached[v]:
        reached[v] = true
        Q.push_back(v)
```

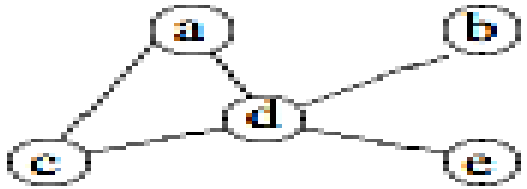
```
function DFS(G, s)

  for all u ∈ V:
    reached[u] = false

  S = ⊔    // Empty stack
  S.push(s)
  while not S.empty():
    u = S.pop()
    if not reached[u]:
      reached[u] = true
      for all (u, v) ∈ E:
        if not reached[v]:
          S.push(v)
```

# Spanning Tree

## Spanning Trees

**Spanning Trees:** A subgraph $T$ of a undirected graph $G = (V, E)$ is a spanning tree of $G$ if it is a tree and contains every vertex of $G$.

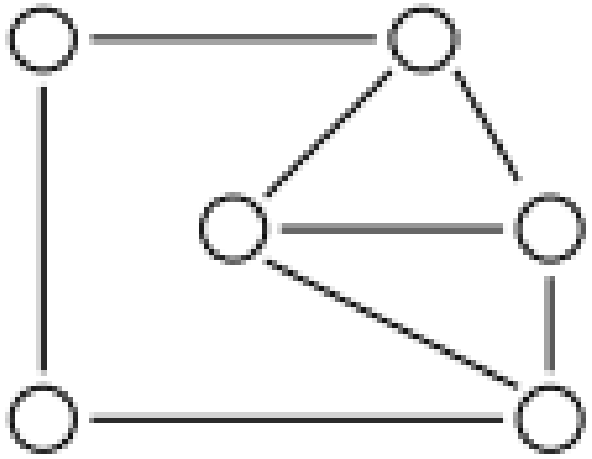
**Example:**




Graph

## Spanning Trees

**Theorem:** Every connected graph has a spanning tree.

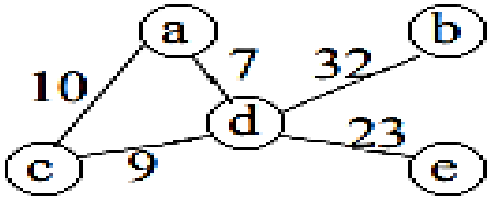


## Weighted Graphs

**Weighted Graphs:** A weighted graph is a graph, in which each edge has a weight (some real number).

**Weight of a Graph:** The sum of the weights of all edges.

**Example:**



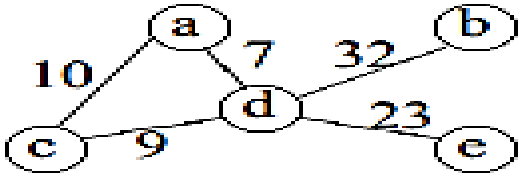

weighted graph

Minimum spanning tree

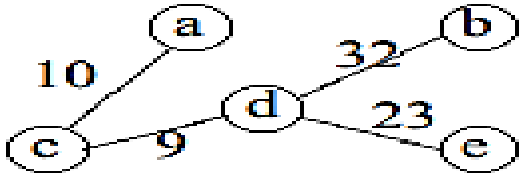
## Minimum Spanning Trees

A **Minimum Spanning Tree** in an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).

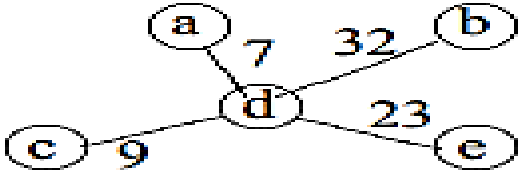
**Example:**



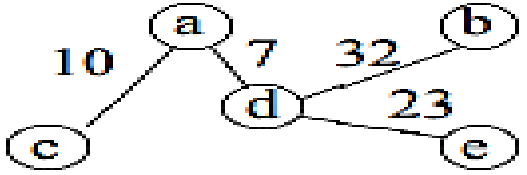

weighted graph




Tree 1, w=74




Tree 2, w=71



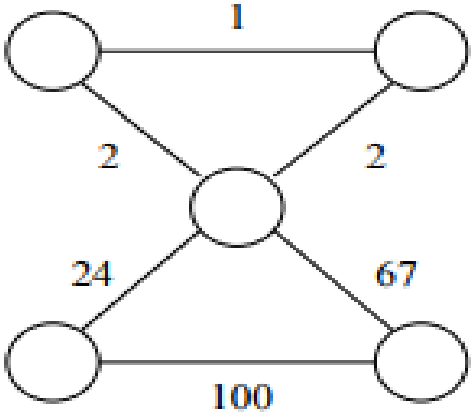

Tree 3, w=72

Minimum spanning tree

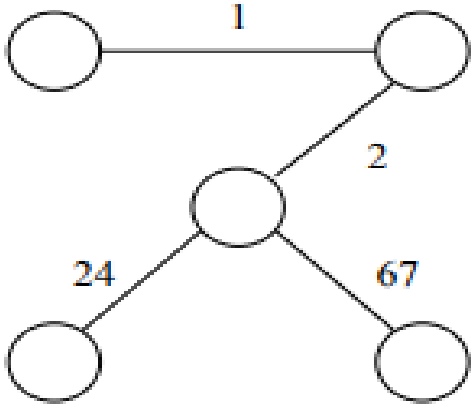
# Minimum Spanning Trees

**Remark:** The minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique (we won't prove this now).

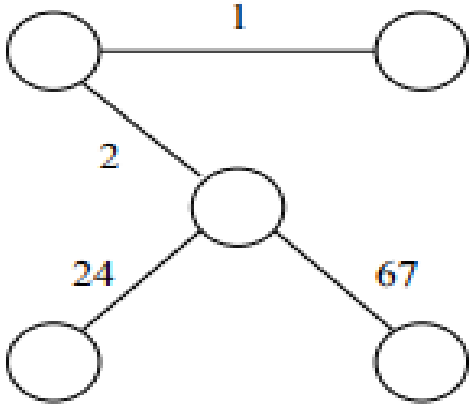
**Example:**



weighted graph



MST1



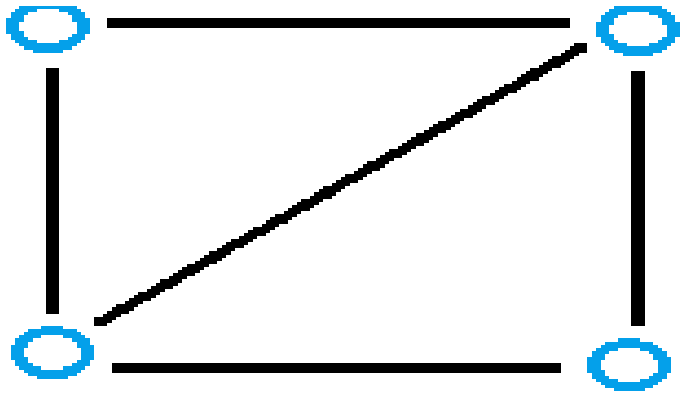
MST2

# Q. What is a Minimum Spanning Tree?

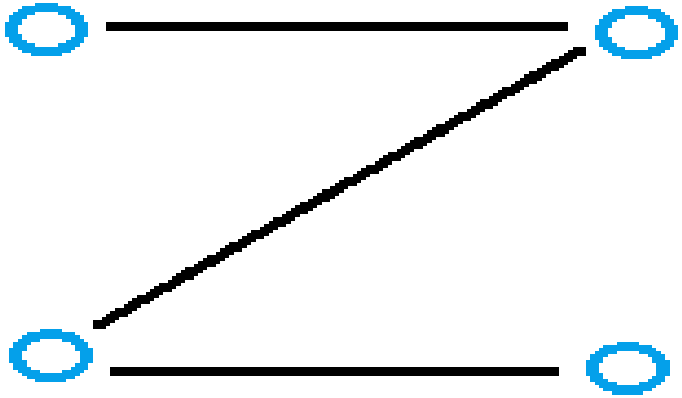
A minimum spanning tree is a special kind of tree that minimizes the lengths (or “weights”) of the edges of the tree.
**Example:** A cable company wanting to lay line to multiple neighborhoods; by minimizing the amount of cable laid, the cable company will save money.

A tree has one path joins any two vertices. A spanning tree of a graph is a tree that:

- Contains all the original graph’s vertices.
- Reaches out to (spans) all vertices.
- Is acyclic. In other words, the graph doesn’t have any nodes which loop back to itself.

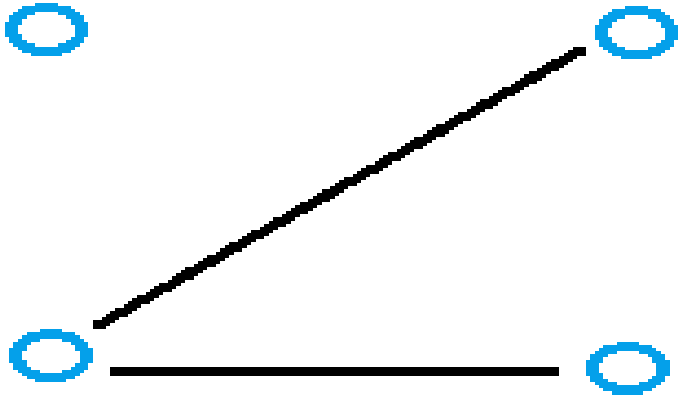


Not Acyclic



Acyclic

Connected



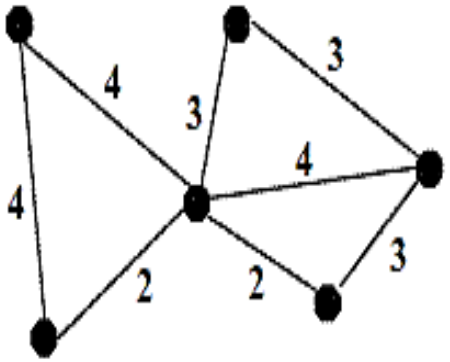
Not Connected

# Finding Minimum Spanning Trees

1. Kruskal's algorithm,
2. Prim's algorithm

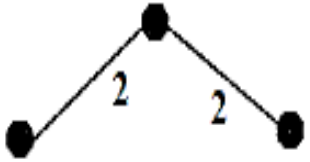
# Kruskal's Algorithm

**1** *Given a network.....*

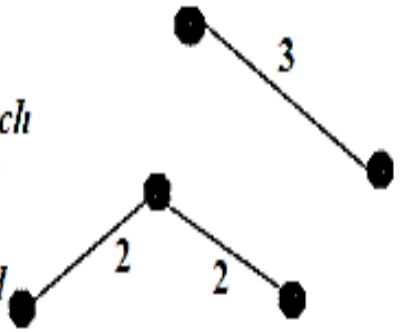



**2** *Choose the shortest edge (if there is more than one, choose any of the shortest).....*

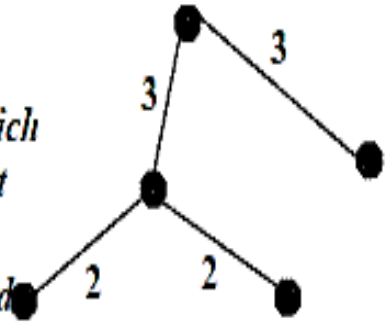

**3** *Choose the next shortest edge and add it.....*



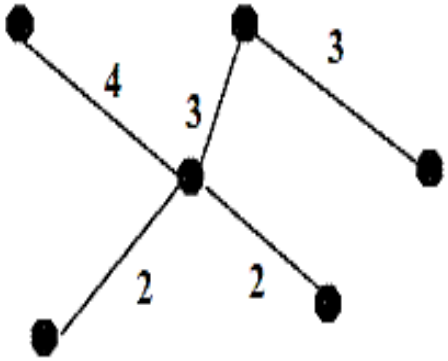

**4** *Choose the next shortest edge which wouldn't create a cycle and add it.*




**5** *Choose the next shortest edge which wouldn't create a cycle and add it.*

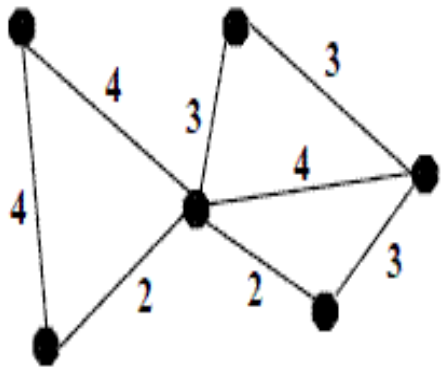



**6** *Repeat until you have a minimal spanning tree.*



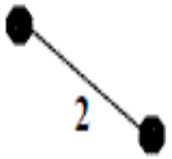
# Prim's Algorithm

**1** *Given a network.....*

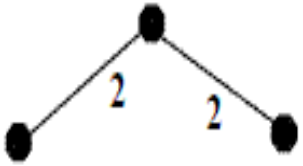



**2** *Choose a vertex*

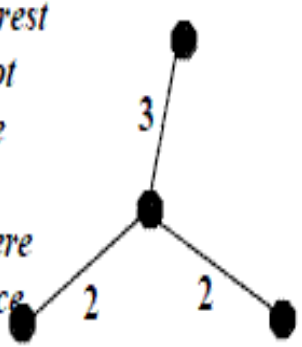
**3** *Choose the shortest edge from this vertex.*



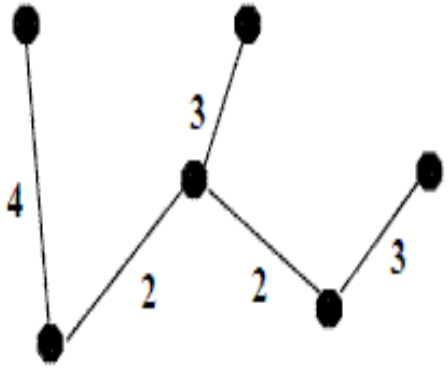

**4** *Choose the nearest vertex not yet in the solution.*



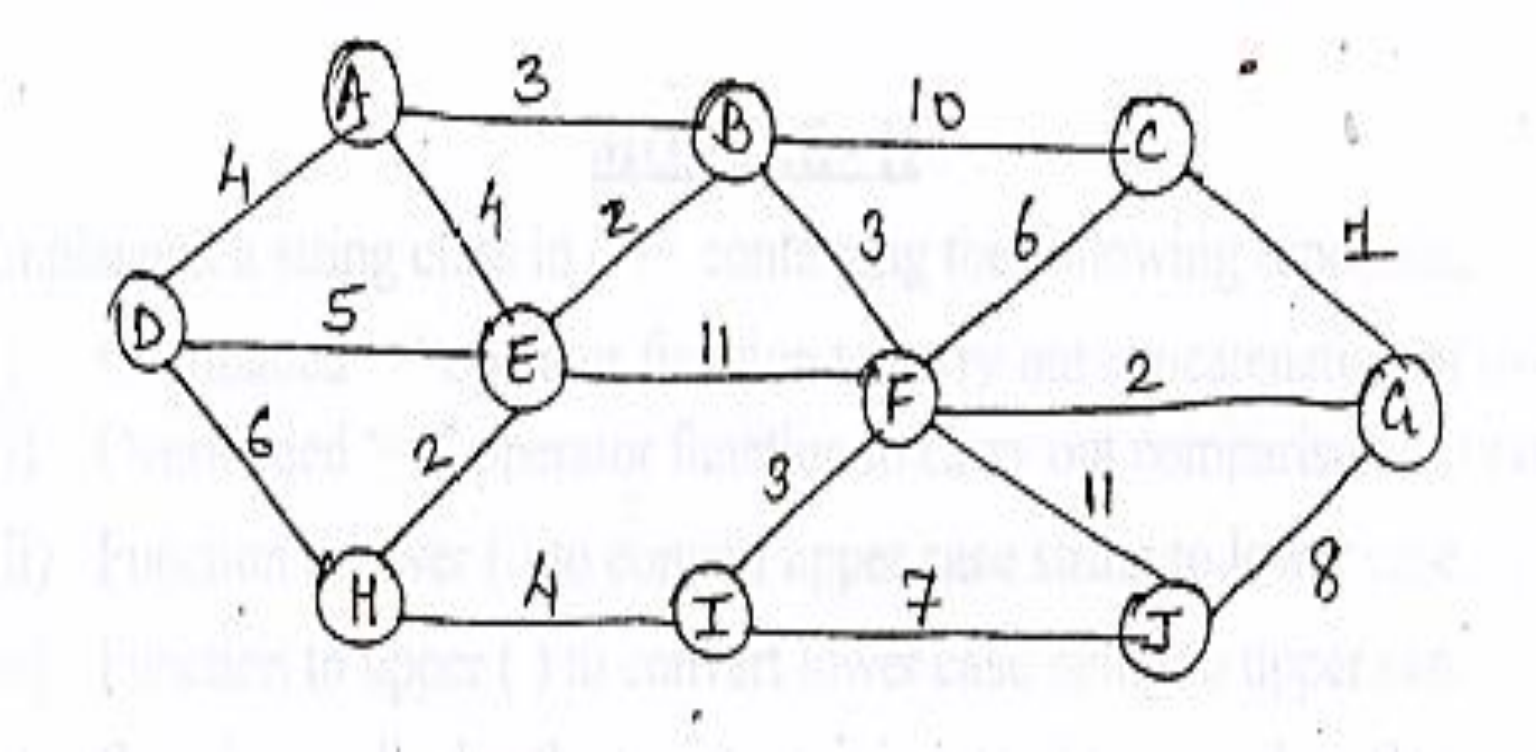

**5** *Choose the next nearest vertex not yet in the solution, when there is a choice choose either.*




**6** *Repeat until you have a minimal spanning tree.*

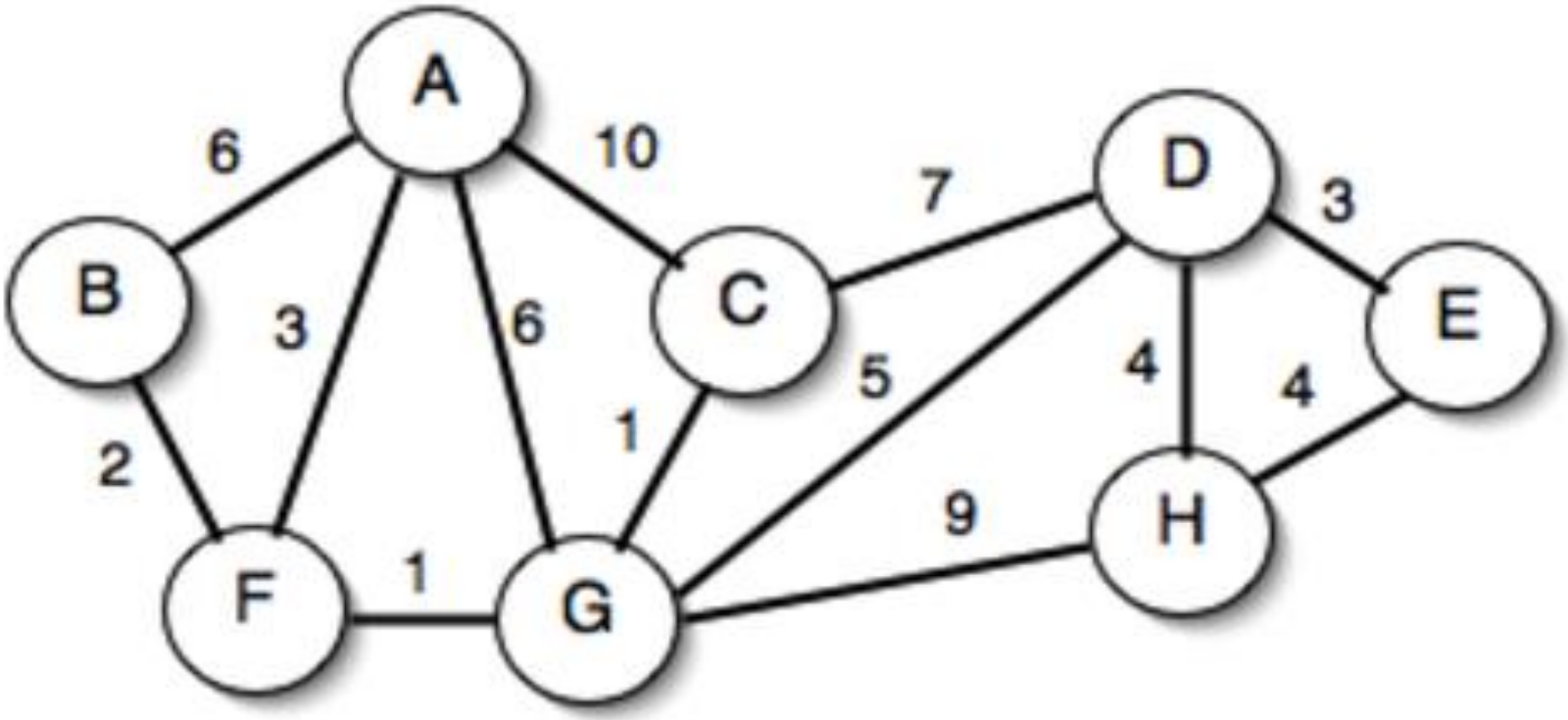


# Obtain minimum cost spanning tree for the following graph using Kruskal's algorithm.

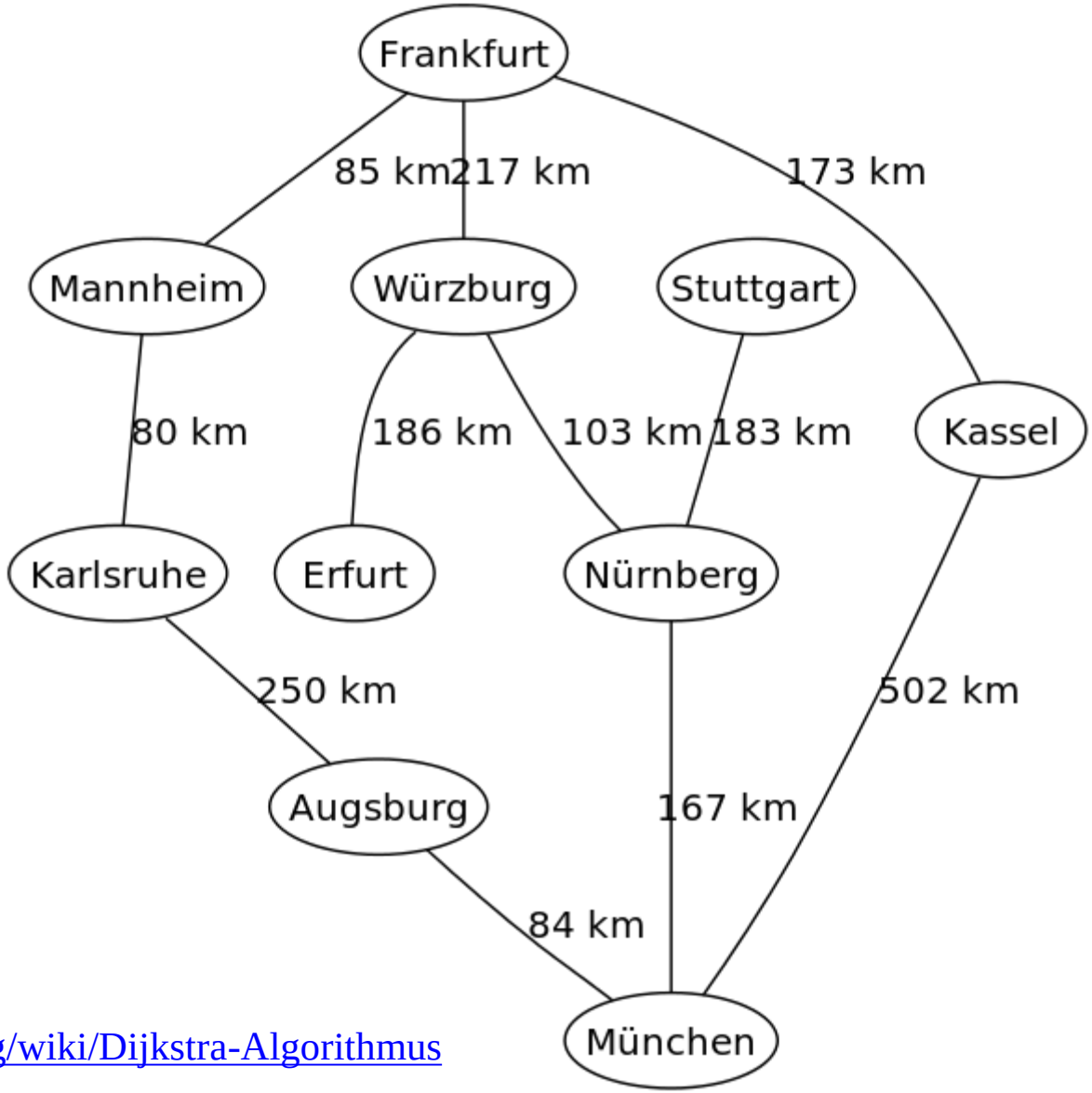


Obtain minimum cost spanning tree for the following graph using Kruskal's algorithm.

8

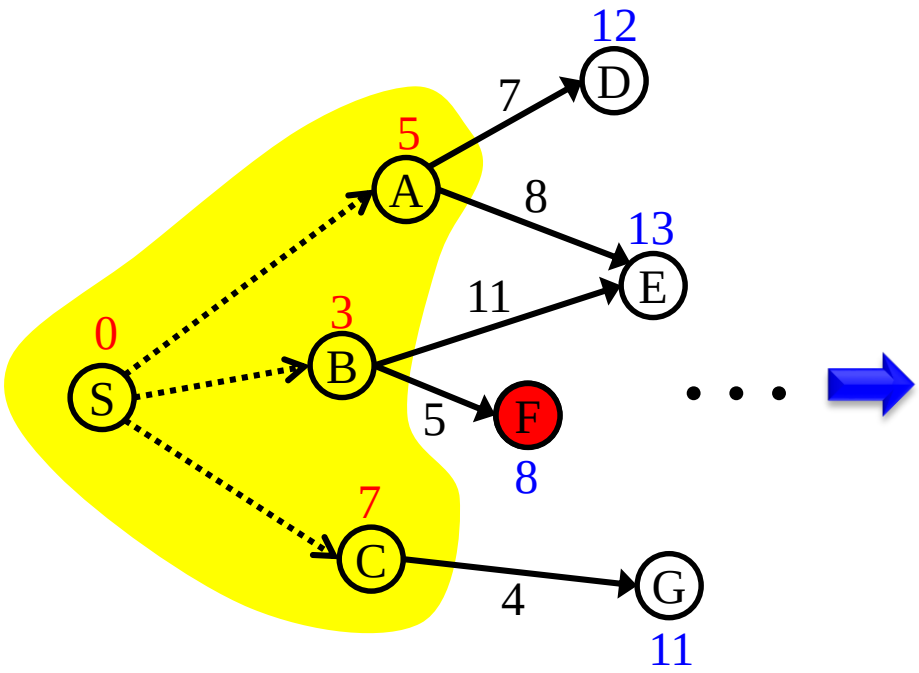


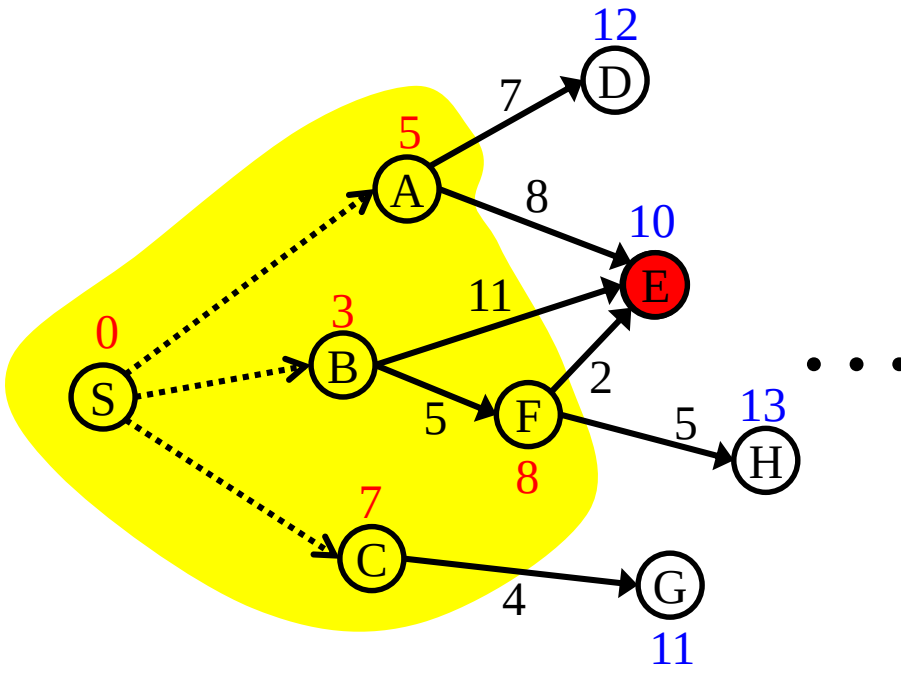
# Distances on edges




https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

# Dijkstra's algorithm: invariant




Shortest paths already computed (completed vertices)

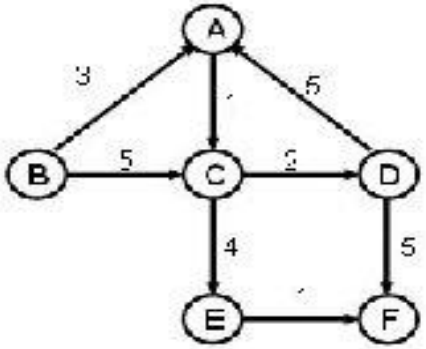
Frontier

Data structure:

The set of non-completed vertices with their shortest distance from S using only the completed vertices.

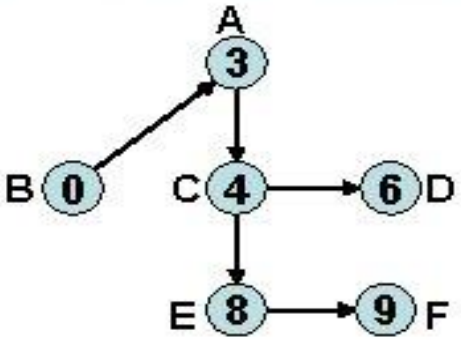
# Example

Tracing Dijkstra's algorithm starting at vertex B:

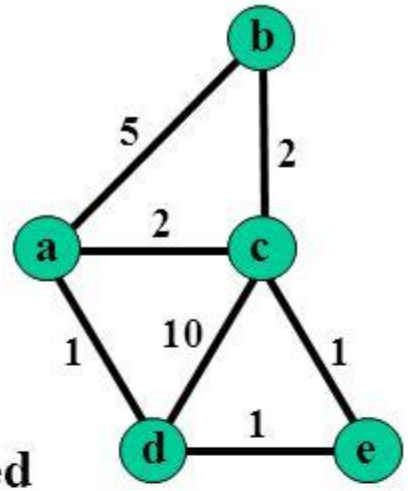



| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Initial (B) | ∞ | 0 | ∞ | ∞ | ∞ | ∞ |
| Process B | 3 | | 5 | ∞ | ∞ | ∞ |
| Process A | | | 4 | ∞ | ∞ | ∞ |
| Process C | | | | 6 | 8 | ∞ |
| Process D | | | | | 8 | 11 |
| Process E | | | | | | 9 |
| | | | | | | |
| Predecessor | B | _ | A | C | C | E |

The resulting vertex-weighted graph is:



# Trace of Dijkstra's Algorithm

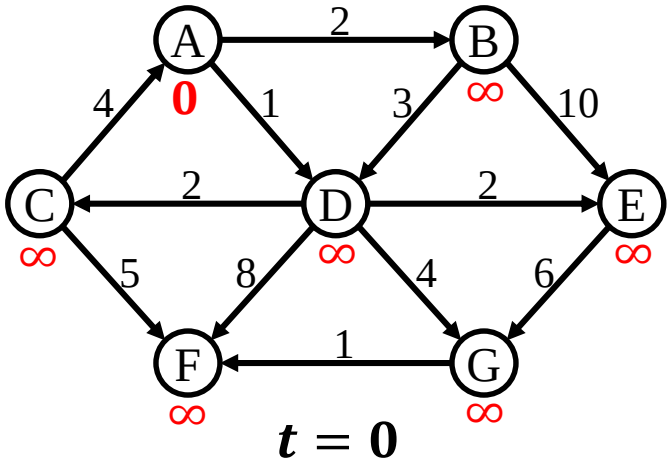


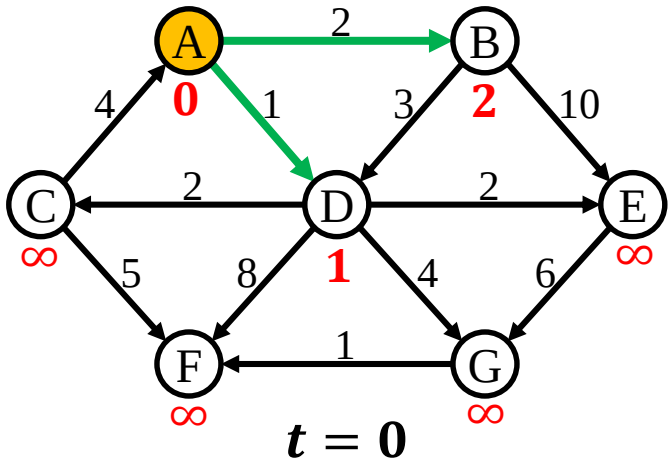

1. Initialize D (start=a)
2. Until all nodes are settled:
   - -find smallest distance & settle node.
   - -adjust distances in D for unsettled nodes.

| iteration | | distance | | | | | settled |
|---|---|---|---|---|---|---|---|
| | | a | b | c | d | e | |
| | 0 | (0) | 5 | 2 | 1 | ∞ | a |
| adjust w/d | 1 | | 5 | 2 | (1) | ~~∞~~ 2 | a,d |
| adjust w/c | 2 | | ~~5~~ 4 | (2) | | 2 | a,d,c |
| adjust w/e | 3 | | 4 | | | (2) | a,d,c,e |
| | 4 | | (4) | | | | a,d,c,e,b |
| | | | | | | | all nodes settled |

**Circled numbers: shortest path lengths from a.**

# Example

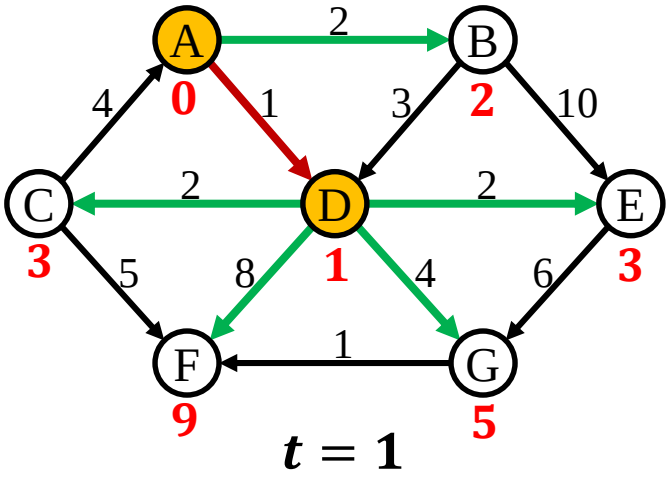


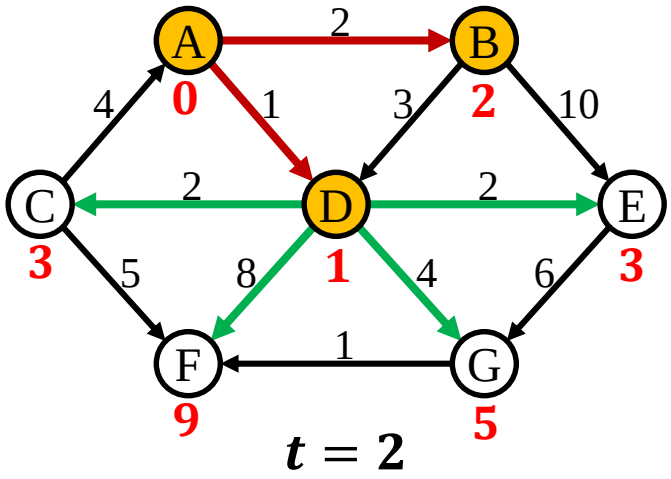




| Done | Queue |
|---|---|
| | **A:0** |
| | **B:∞** |
| | **E:∞** |
| | **D:∞** |
| | **C:∞** |
| | **F:∞** |
| | **G:∞** |

| Done | Queue |
|---|---|
| **A:0** | **D:1** |
| | **B:2** |
| | **E:∞** |
| | **C:∞** |
| | **F:∞** |
| | **G:∞** |
| | |

# Example

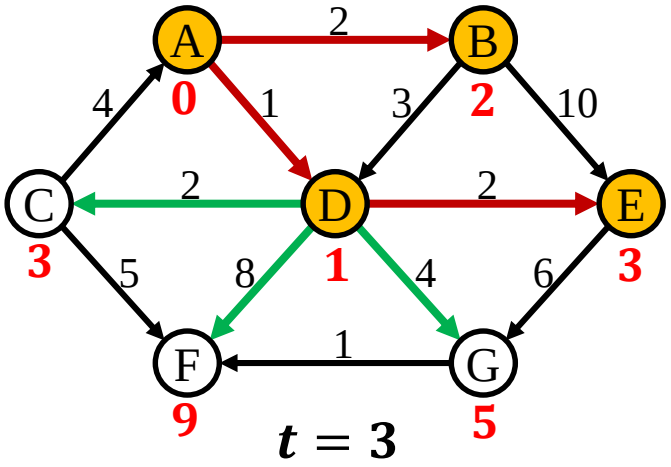


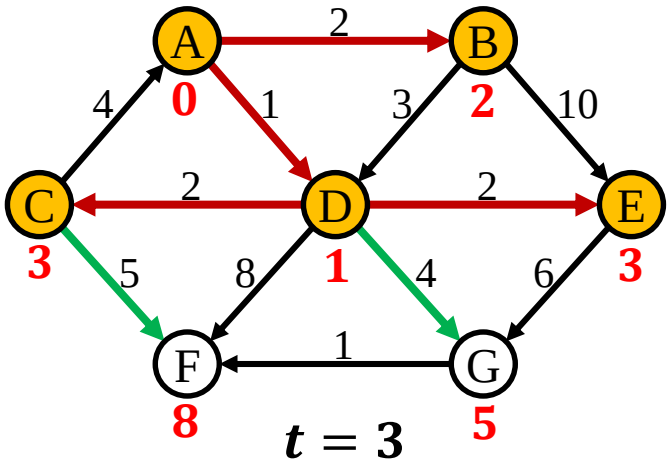




| Done | Queue |
|---|---|
| A:0 | B:2 |
| D:1 | E:3 |
| | C:3 |
| | G:5 |
| | F:9 |
| | |
| | |

| Done | Queue |
|---|---|
| A:0 | E:3 |
| D:1 | C:3 |
| B:2 | G:5 |
| | F:9 |
| | |
| | |
| | |

# Example

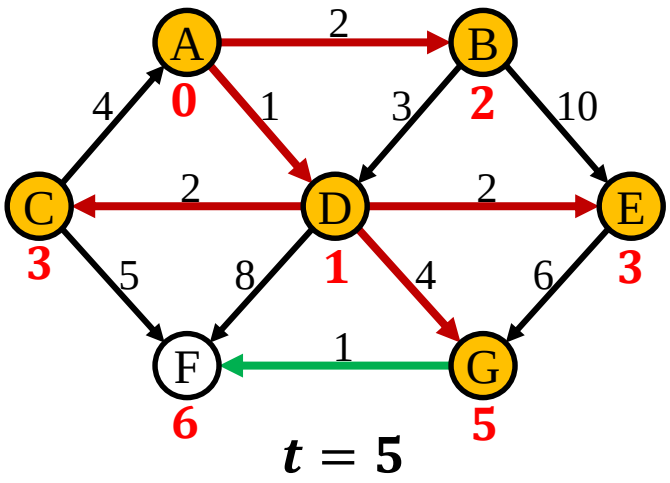



| Done | Queue |
|---|---|
| A:0 | C:3 |
| D:1 | G:5 |
| B:2 | F:9 |
| E:3 | |
| | |
| | |
| | |

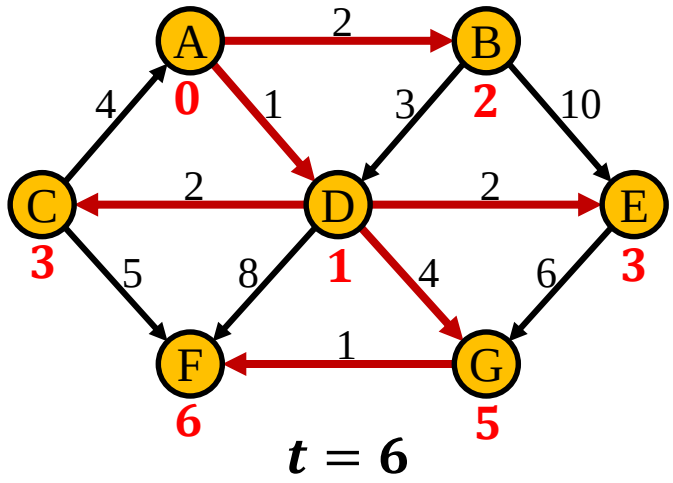
| Done | Queue |
|---|---|
| A:0 | G:5 |
| D:1 | F:8 |
| B:2 | |
| E:3 | |
| C:3 | |
| | |
| | |

# Example




$t = 5$

| Done | Queue |
|---|---|
| A:0 | F:6 |
| D:1 | |
| B:2 | |
| E:3 | |
| C:3 | |
| G:5 | |
| | |

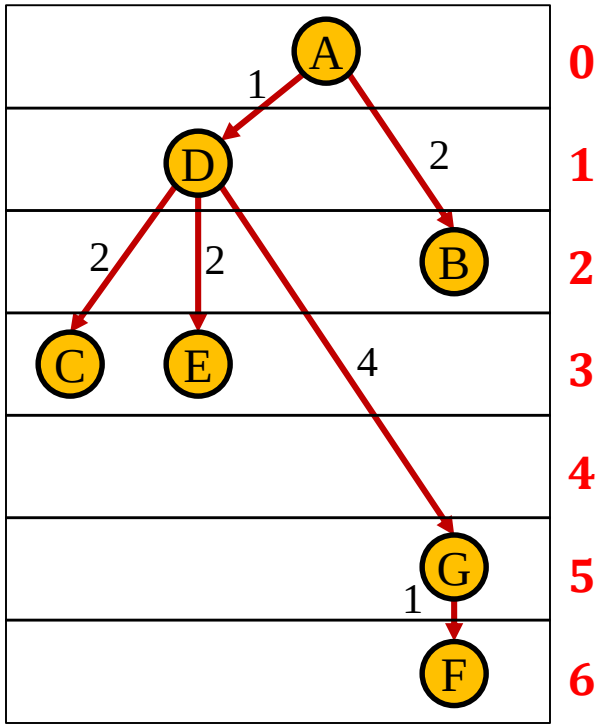


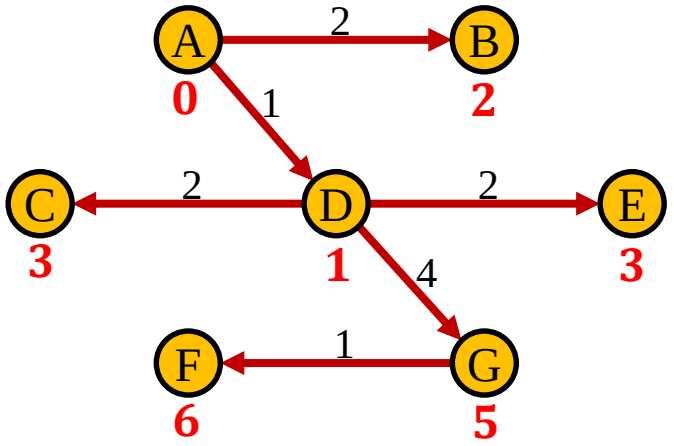

$t = 6$

| Done | Queue |
|---|---|
| A:0 | |
| D:1 | |
| B:2 | |
| E:3 | |
| C:3 | |
| G:5 | |
| F:6 | |

# Example

Shortest-path tree




We need to:

- keep a list non-completed vertices and their expected distances.
- select the non-completed vertex with shortest distance.
- update the distances of the neighbouring vertices.

# Dijkstra's algorithm for shortest paths

- function ShortestPaths($\boldsymbol{G}$, $\boldsymbol{l}$, $\boldsymbol{s}$)

```
// Input: Graph G(V,E), source vertex s,
//        positive edge lengths {l_e: e ∈ E}
// Output: dist[u] has the distance from s,
//         prev[u] has the predecessor in the tree

  for all u ∈ V:
    dist[u] = ∞
    prev[u] = nil

  dist[s] = 0
        Q = makequeue(V)  // using dist as keys

  while not Q.empty():
    u = Q.deletemin()
    for all (u,v) ∈ E:
      if dist[v] > dist[u] + l(u,v):
        dist[v] = dist[u] + l(u,v)
        prev[v] = u
        Q.decreasekey(v)
```

# Dijkstra's algorithm: complexity

```
Q = makequeue(V)
while not Q.empty():
    u = Q.deletemin()
    for all (u, v) ∈ E:
        if dist[v] > dist[u] + l(u, v):
            dist[v] = dist[u] + l(u, v)
            prev[v] = u
            Q.decreasekey(v)
```

$|V|$ **times**

$|E|$ **times**

- The skeleton of Dijkstra's algorithm is based on BFS, which is $O(|V| + |E|)$
- We need to account for the cost of:
  - **makequeue**: insert $|V|$ vertices to a list.
  - **deletemin**: find the vertex with min dist in the list ($|V|$ times)
  - **decreasekey**: update dist for a vertex ($|E|$ times)

- Let us consider two implementations for the list: **vector** and **binary heap**

# Dijkstra's algorithm: complexity

| Implementation | deletemin | insert/ decreasekey | Dijkstra's complexity |
|---|---|---|---|
| **Vector** | $O(\|V\|)$ | $O(1)$ | $O(\|V\|^2)$ |
| **Binary heap** | $O(\log \|V\|)$ | $O(\log \|V\|)$ | $O((\|V\| + \|E\|) \log \|V\|)$ |

**Binary heap:**

- The elements are stored in a complete (balanced) binary tree.
- **Insertion:** place element at the bottom and let it *bubble up* swapping the location with the parent (at most $\log_2 |V|$ levels).
- **Deletemin:** Remove element from the root, take the last node in the tree, place it at the root and let it *bubble down* (at most $\log_2 |V|$ levels).
- **Decreasekey:** decrease the key in the tree and let it *bubble up* (same as insertion). A data structure might be required to known the location of each vertex in the heap (table of pointers).

**For connected graphs:** $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$

| | Average | Worst |
|---|---|---|
| Dijkstra's algorithm | $O(\|E\| \log \|V\|)$ | $O(\|V\|^2)$ |
| A* search algorithm | $O(\|E\|)$ | $O(b^d)$ |
| Prim's algorithm | $O(\|E\| \log \|V\|)$ | $O(\|V\|^2)$ |
| Bellman–Ford algorithm | $O(\|E\| \cdot \|V\|)$ | $O(\|E\| \cdot \|V\|)$ |
| Floyd-Warshall algorithm | $O(\|V\|^3)$ | $O(\|V\|^3)$ |
| Topological sort | $O(\|V\| + \|E\|)$ | $O(\|V\| + \|E\|)$ |