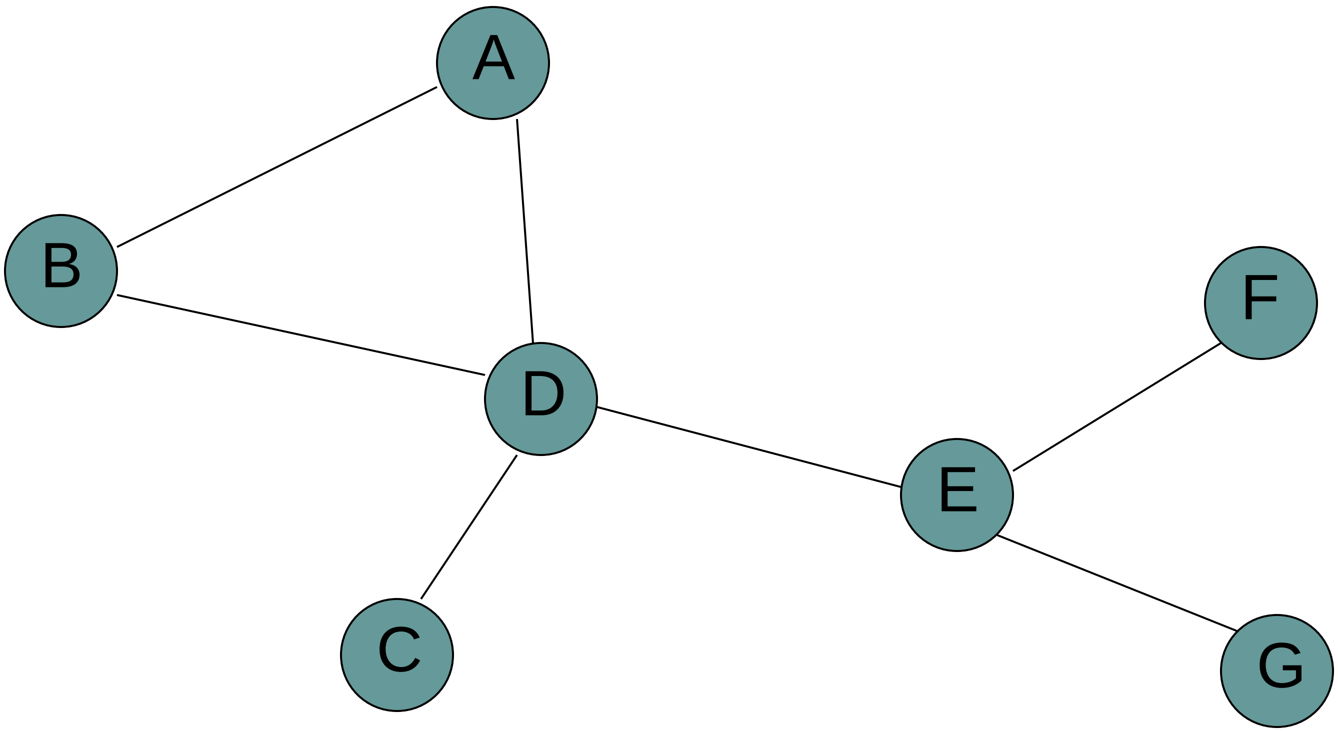
Graphs

When do we see graphs in real life problems?

- Transportation networks (flights, roads, etc.)
- Communication networks
- Web
- Social networks
- Circuit design
- Bayesian networks

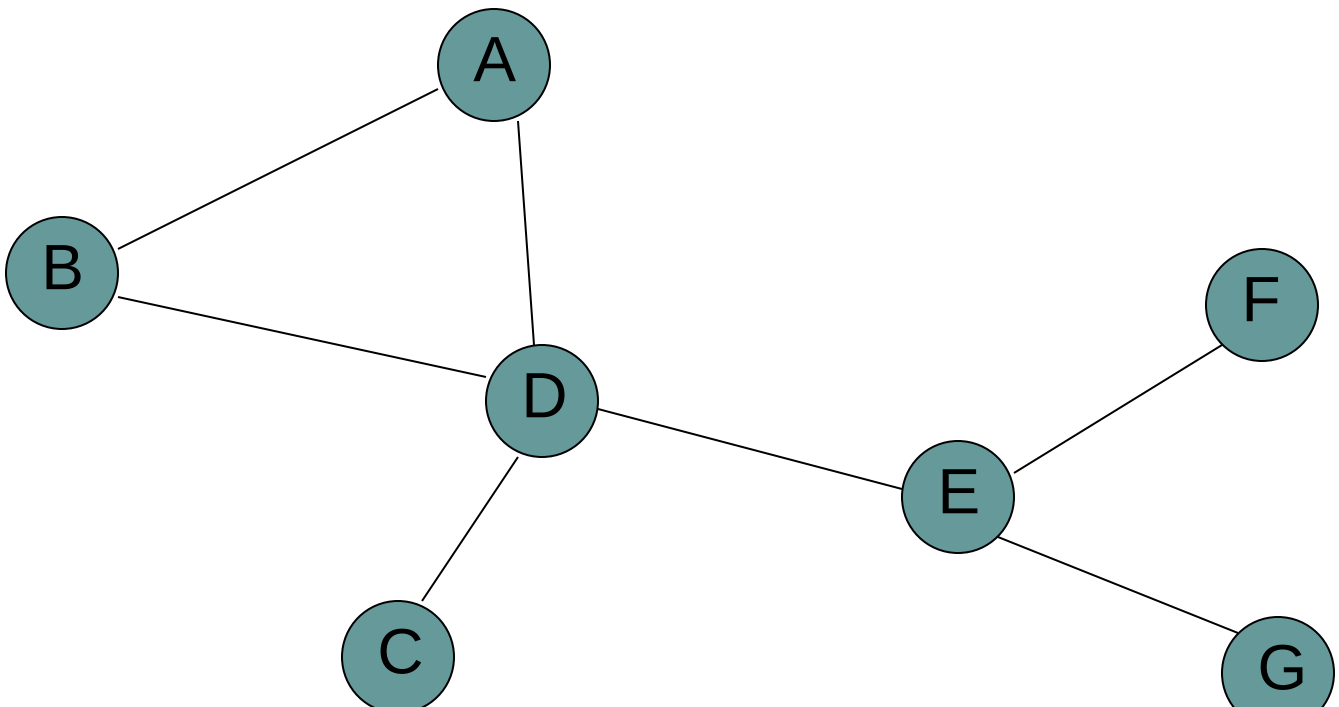
Graphs

- A graph is a set of vertices V and a set of edges (u,v) $\in$ E where u,v $\in$ V



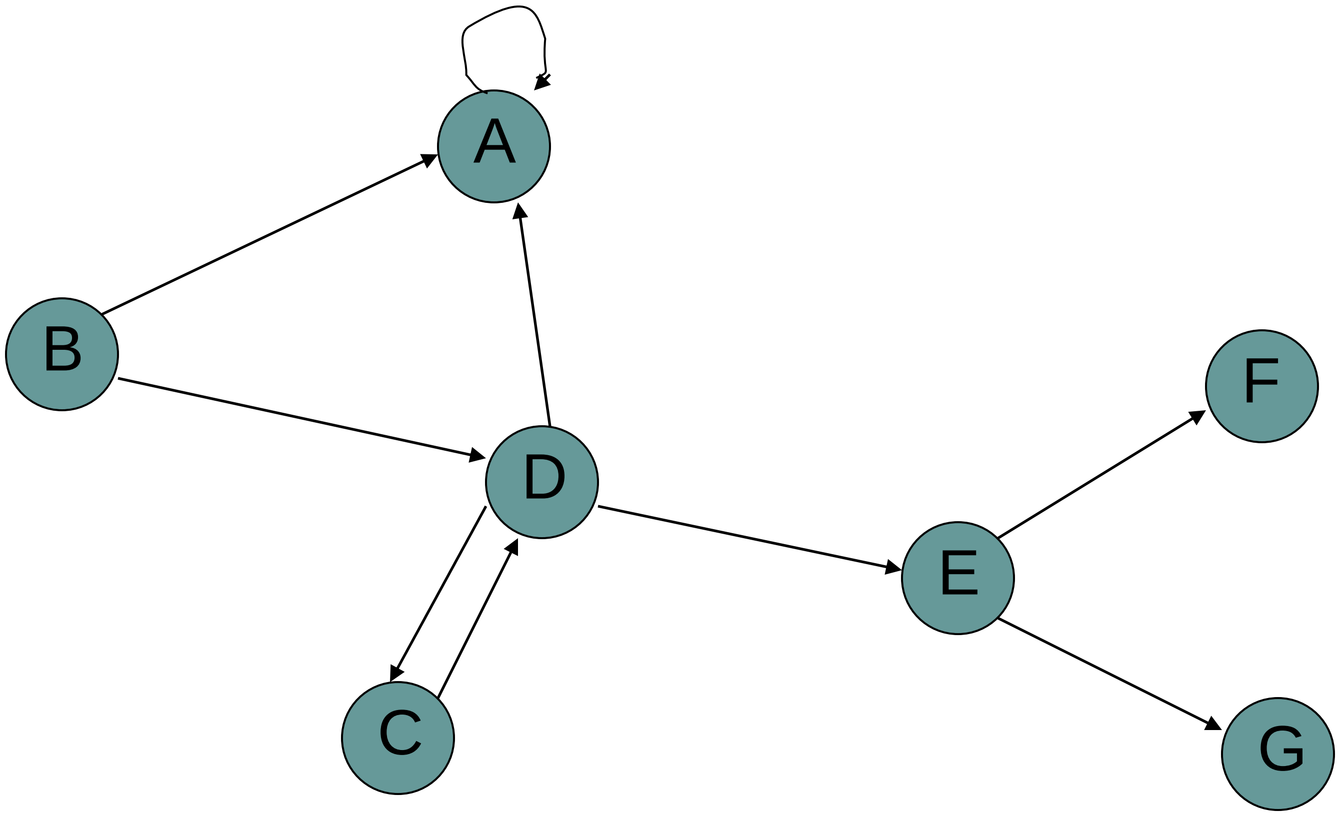
Different types of graphs

- Undirected – edges do not have a direction



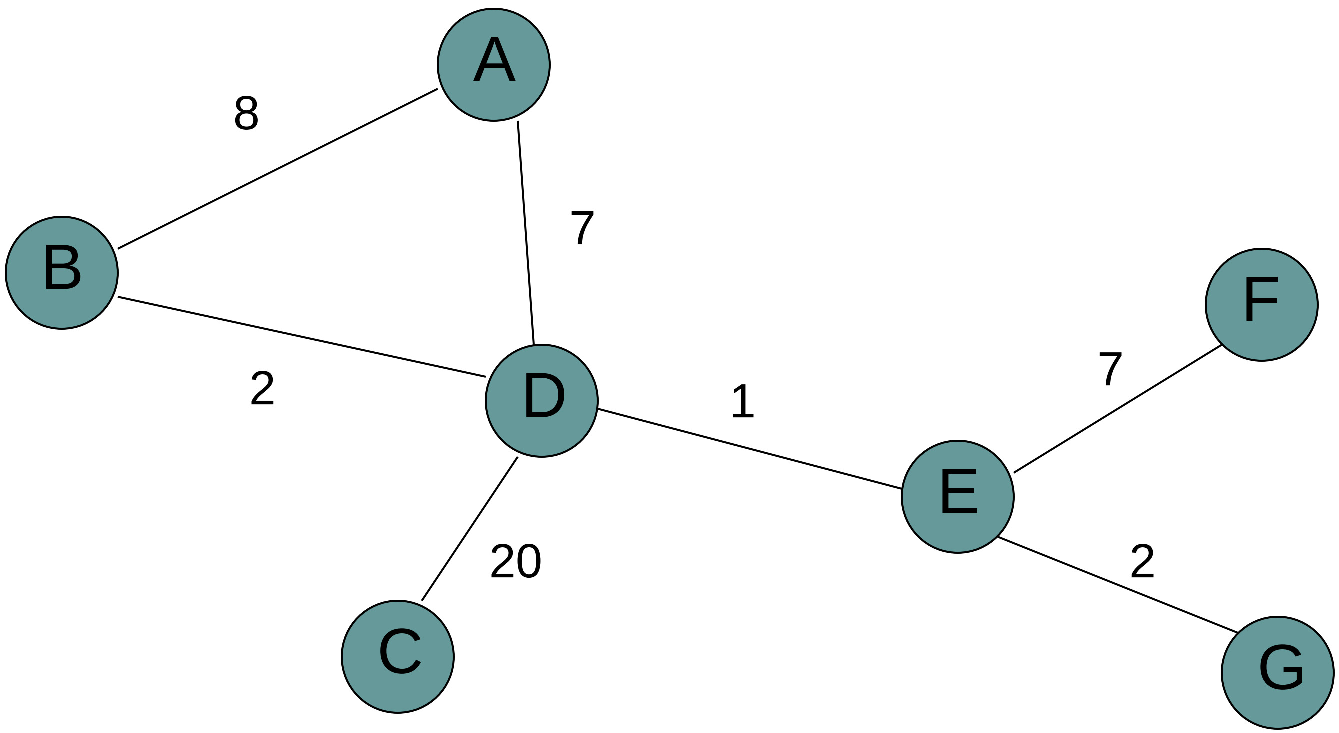
Different types of graphs

- Directed – edges **do** have a direction



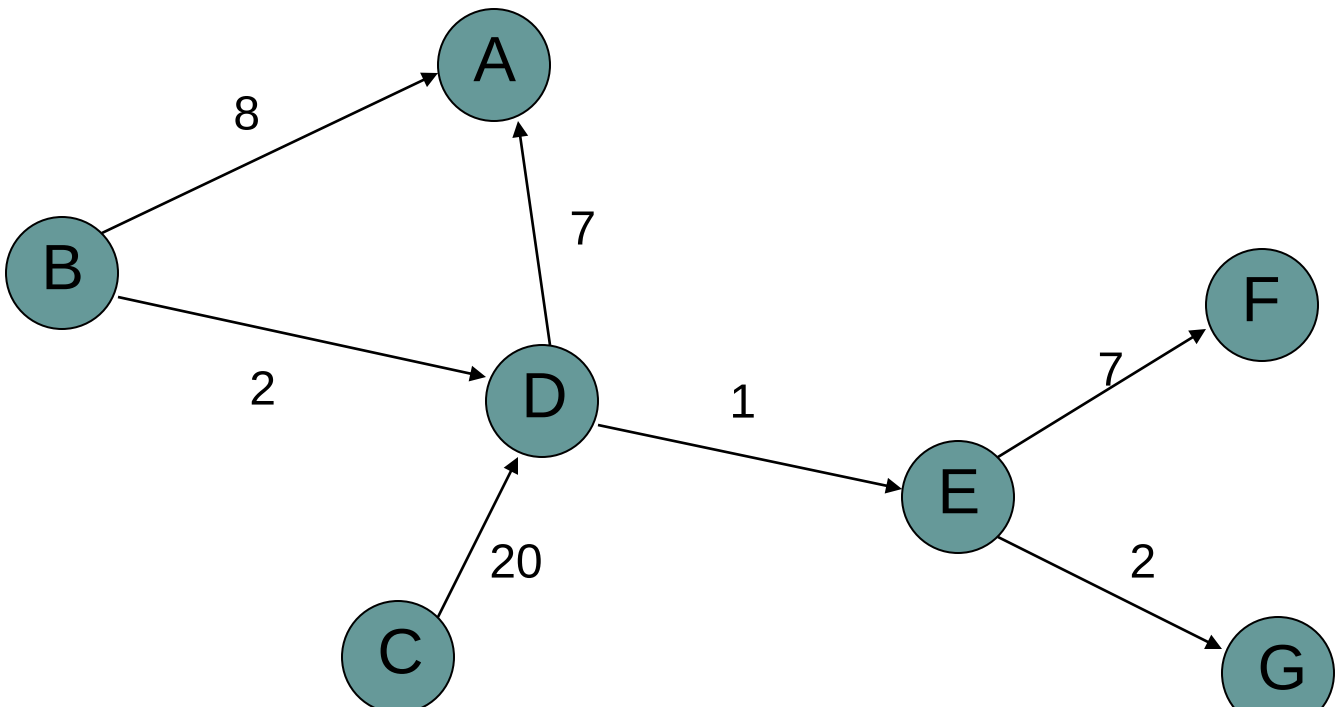
Different types of graphs

- Weighted – edges have an associated weight



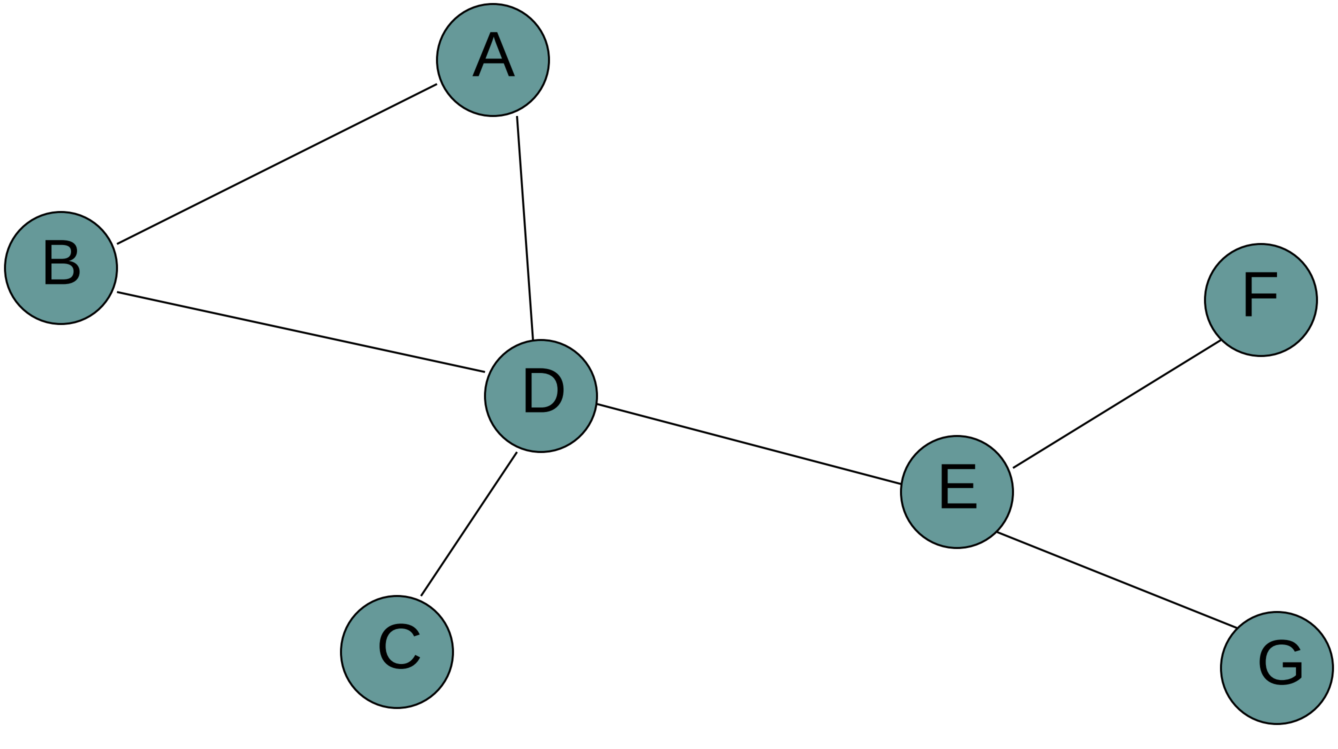
Different types of graphs

- Weighted – edges have an associated weight



Terminology

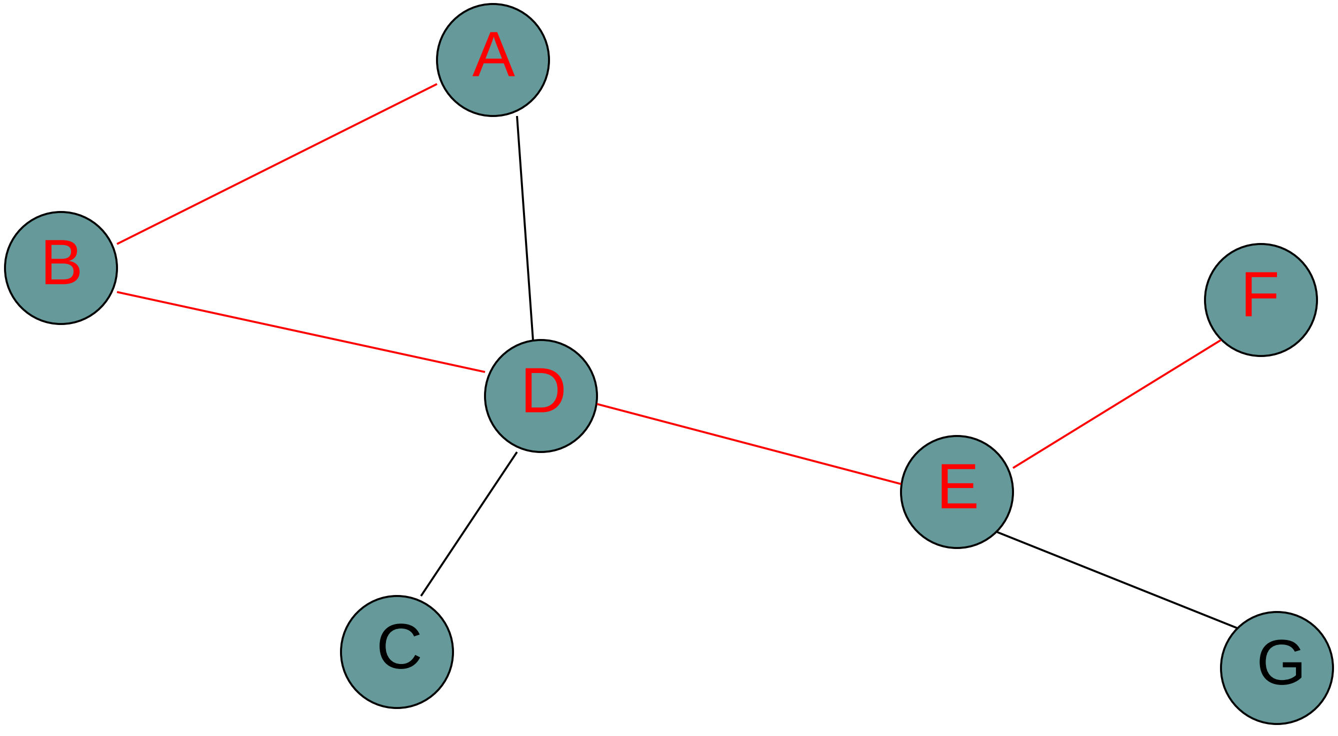
- Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$



Terminology

- Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$

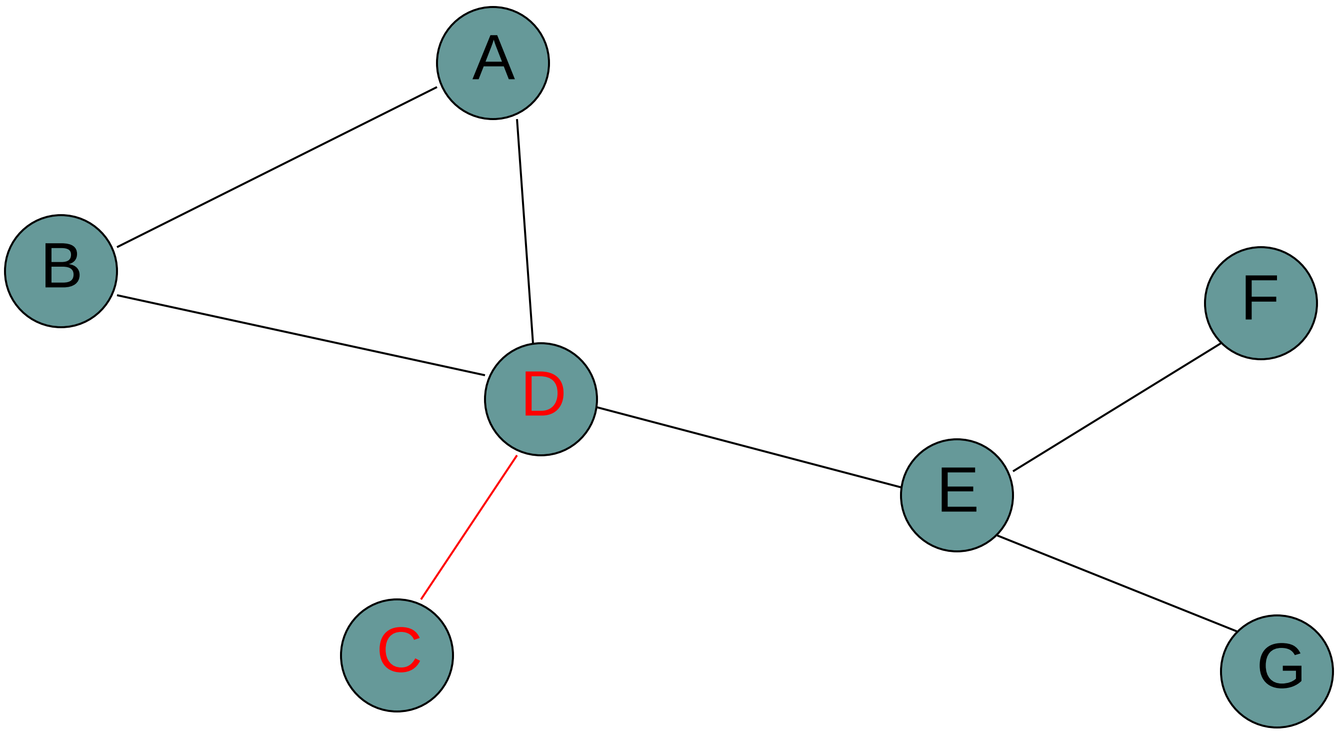
{A, B, D, E, F}



Terminology

- Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$

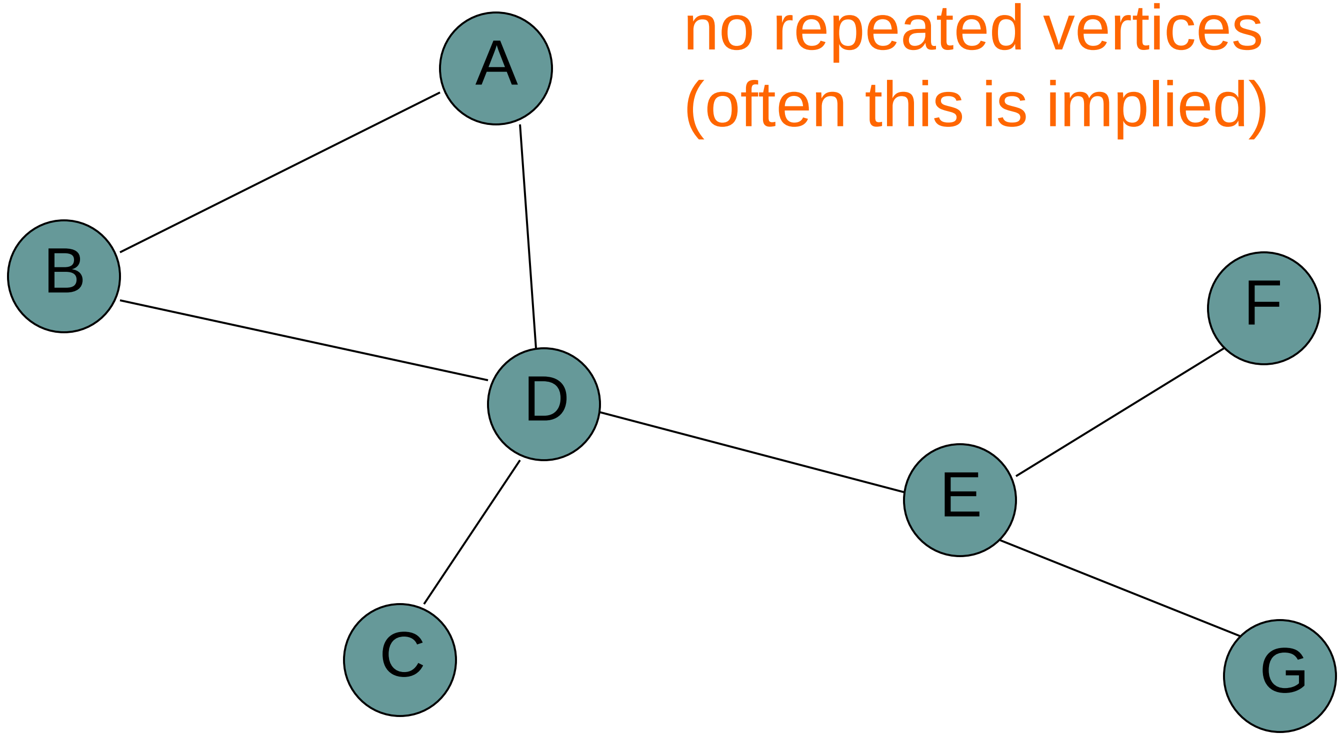
{C, D}



Terminology

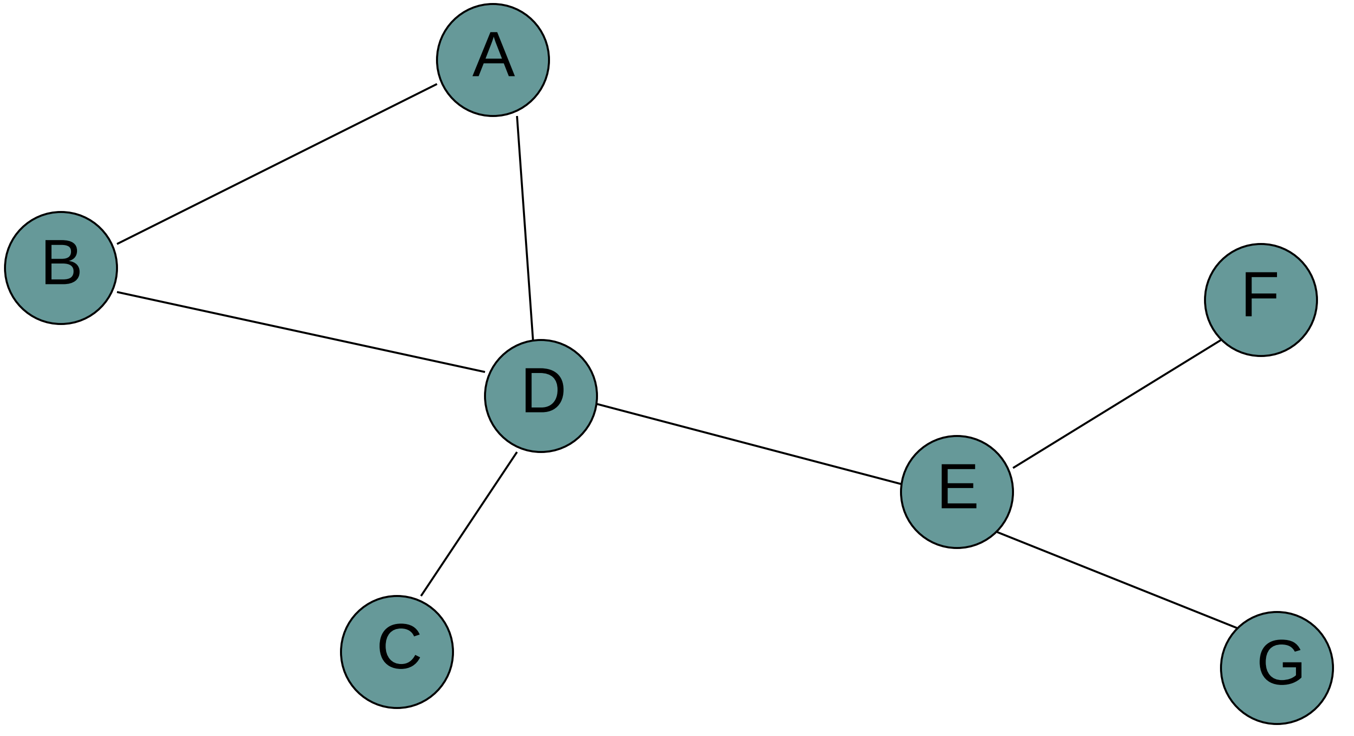
- Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$

A *simple* path contains no repeated vertices (often this is implied)



Terminology

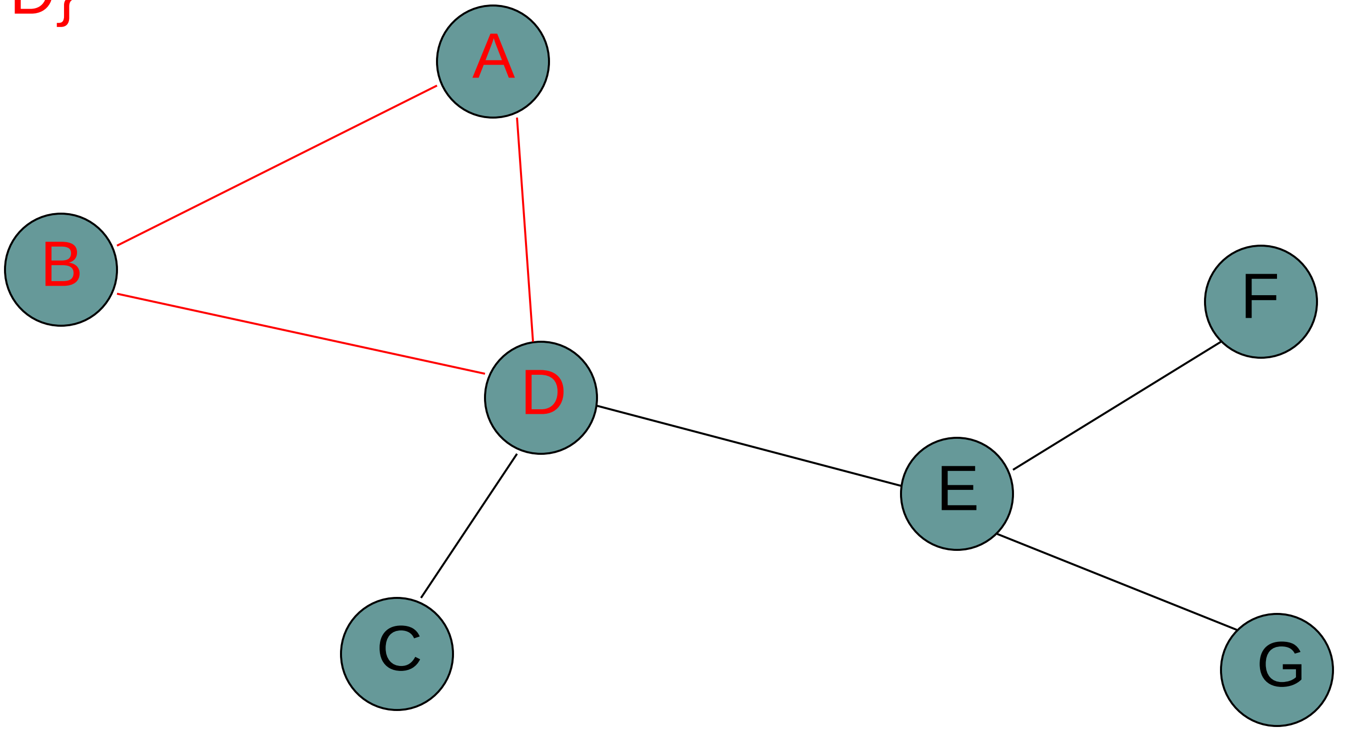
- Cycle – A subset of the edges that form a path such that the first and last node are the same



Terminology

- Cycle – A subset of the edges that form a path such that the first and last node are the same

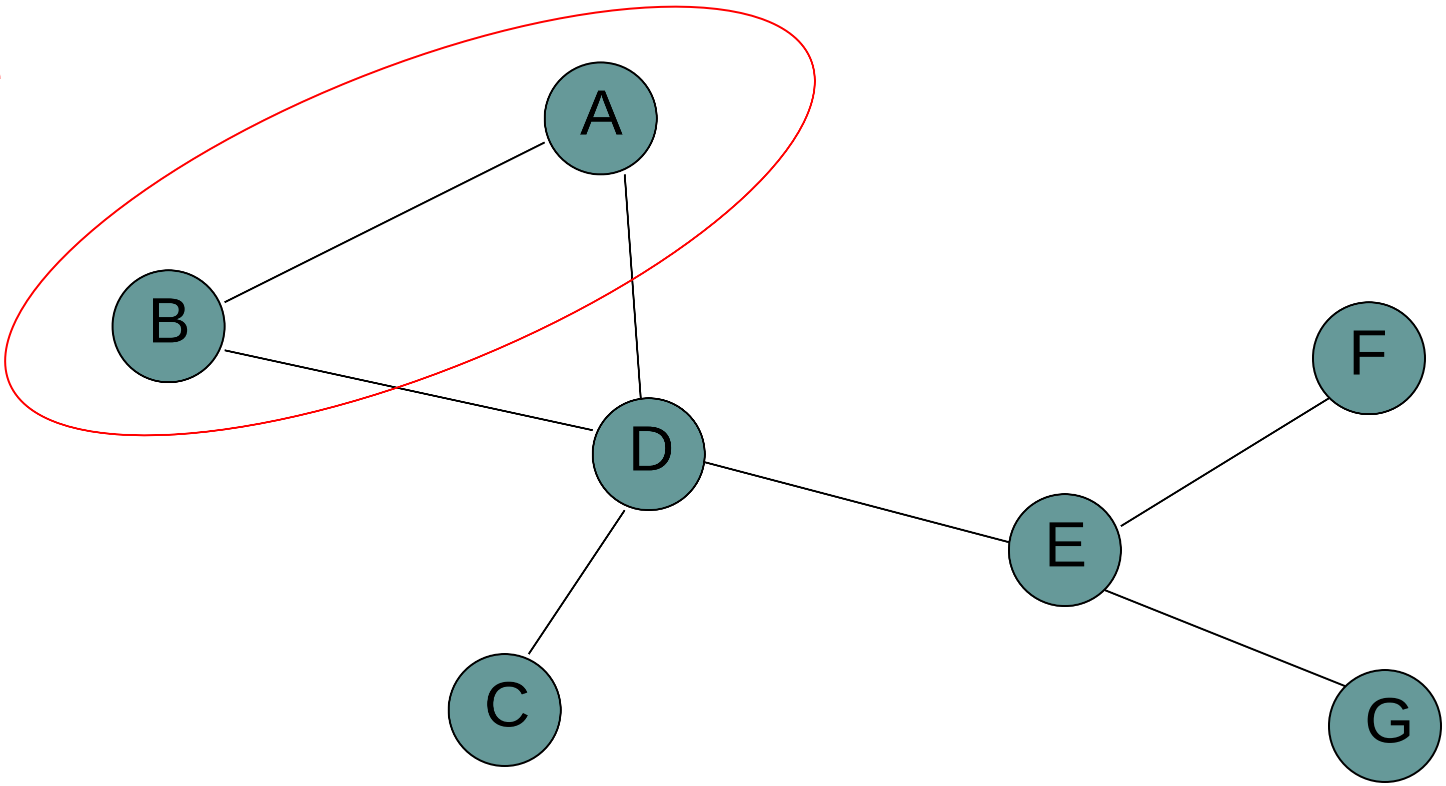
{A, B, D}



Terminology

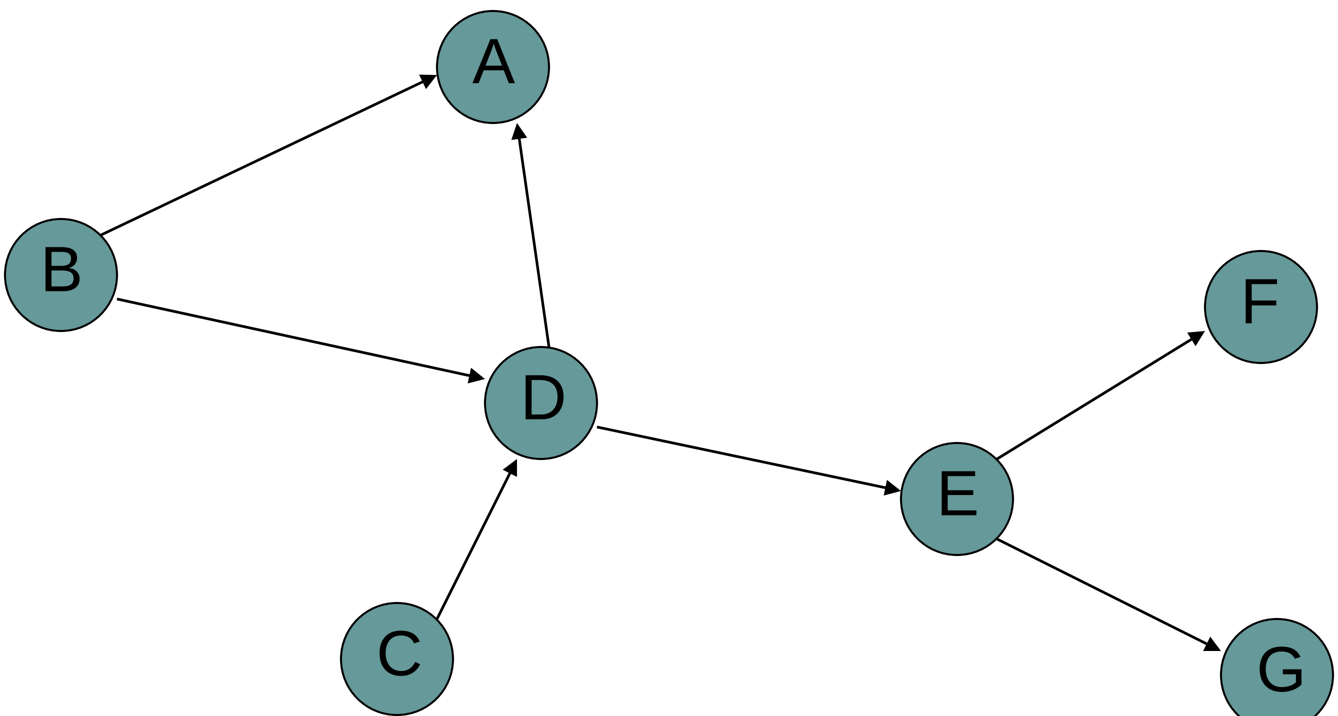
- Cycle – A subset of the edges that form a path such that the first and last node are the same

not a cycle



Terminology

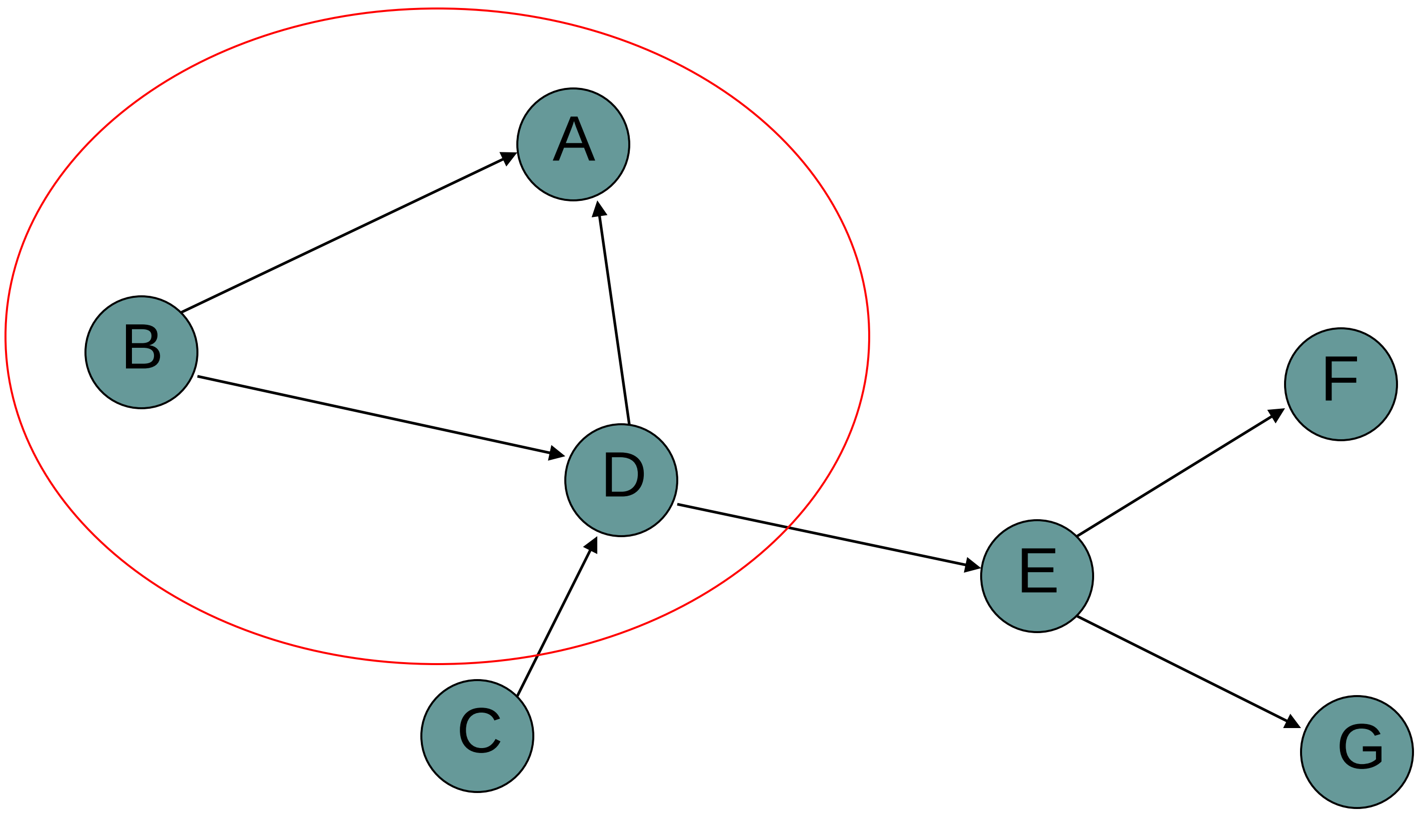
- Cycle – A subset of the edges that form a path such that the first and last node are the same



Terminology

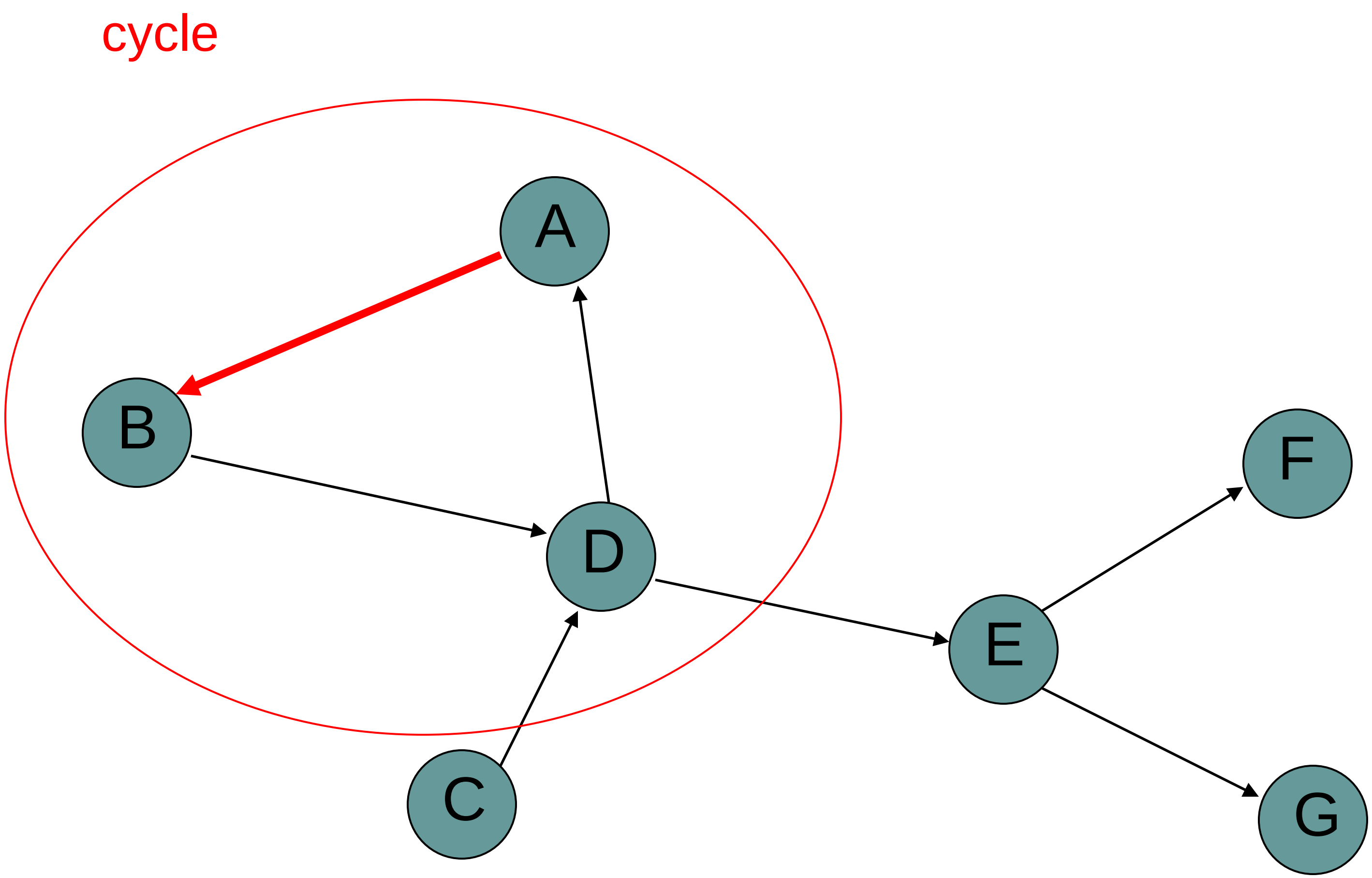
- Cycle – A subset of the edges that form a path such that the first and last node are the same

not a cycle



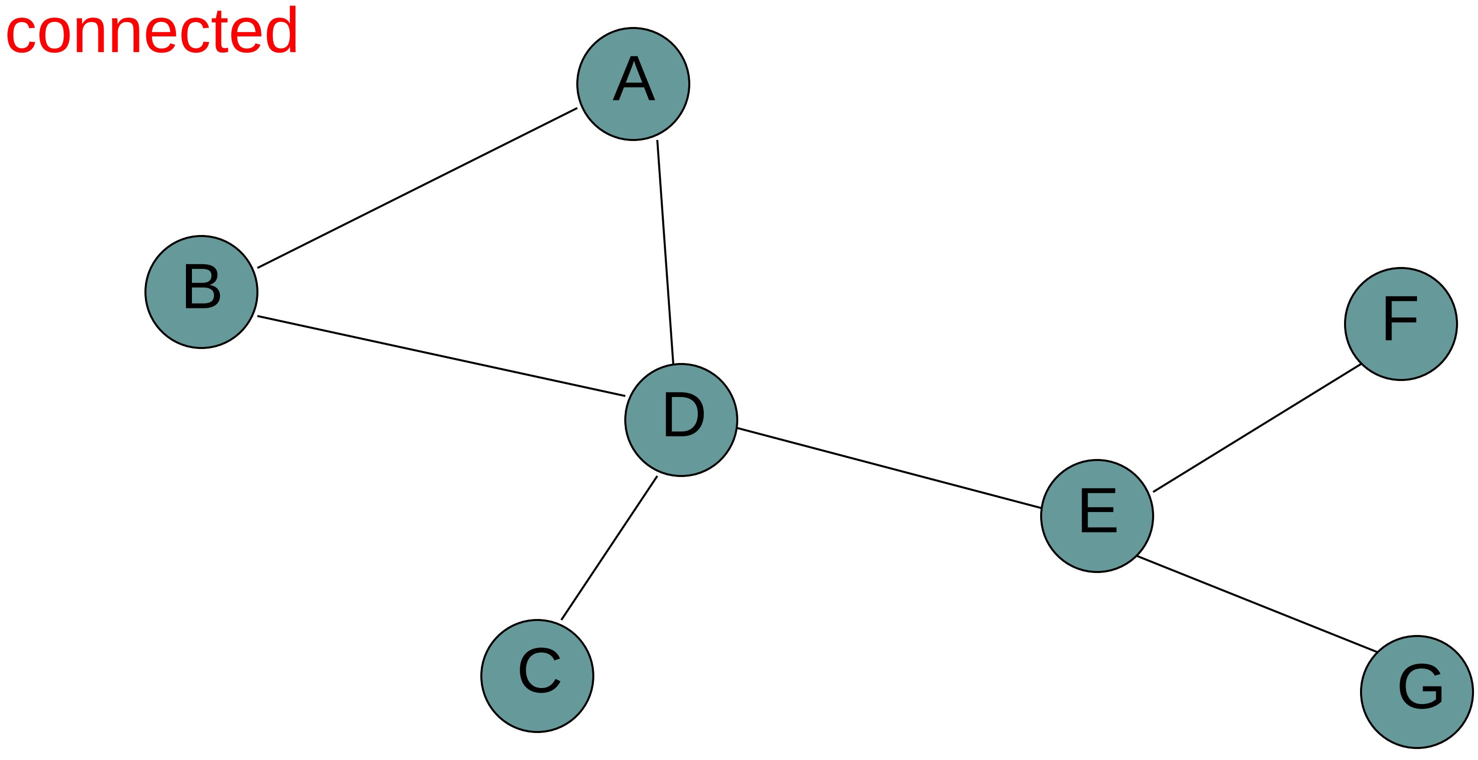
Terminology

- Cycle – A path $p_1, p_2, \ldots p_k$ where $p_1 = p_k$



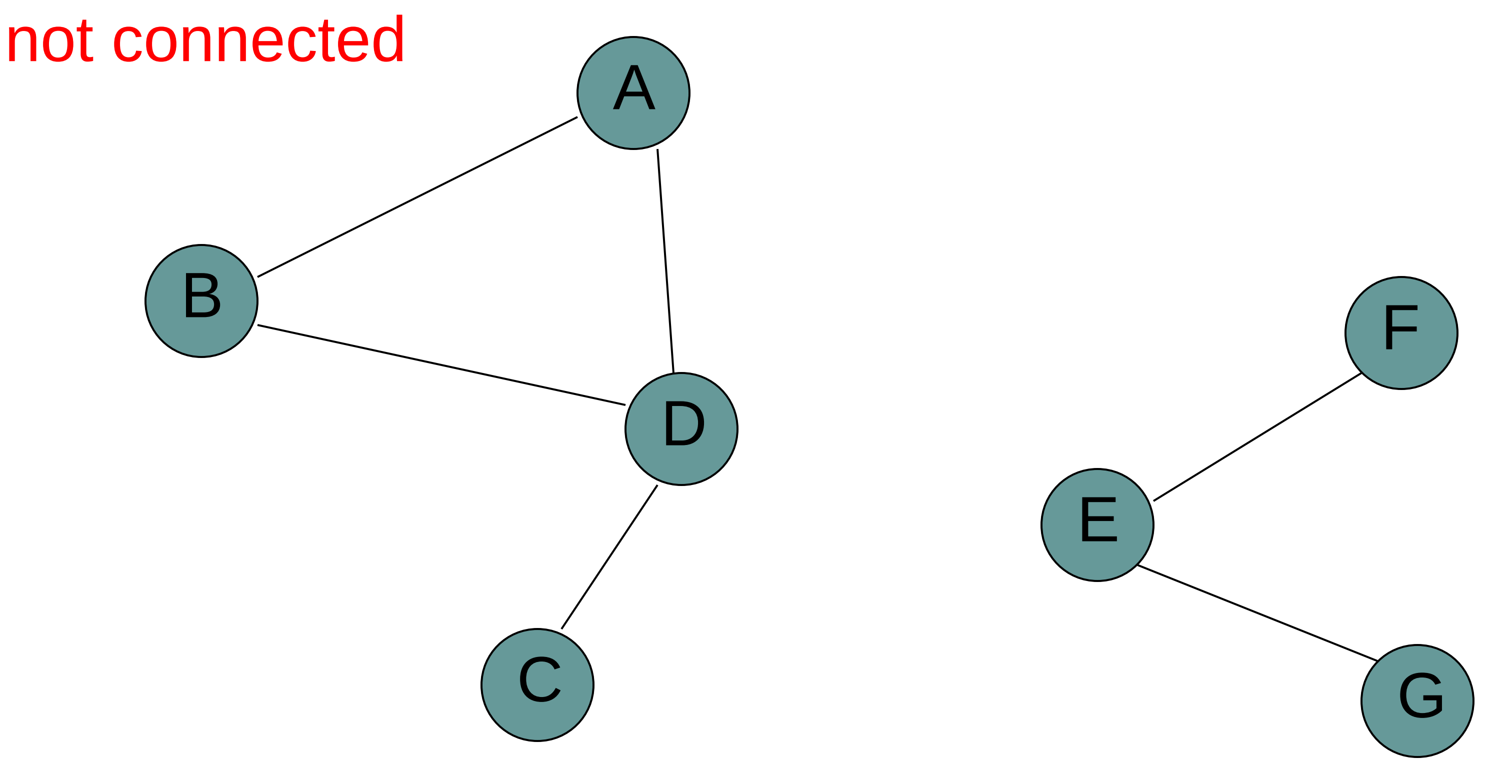
Terminology

- Connected – every pair of vertices is connected by a path



Terminology

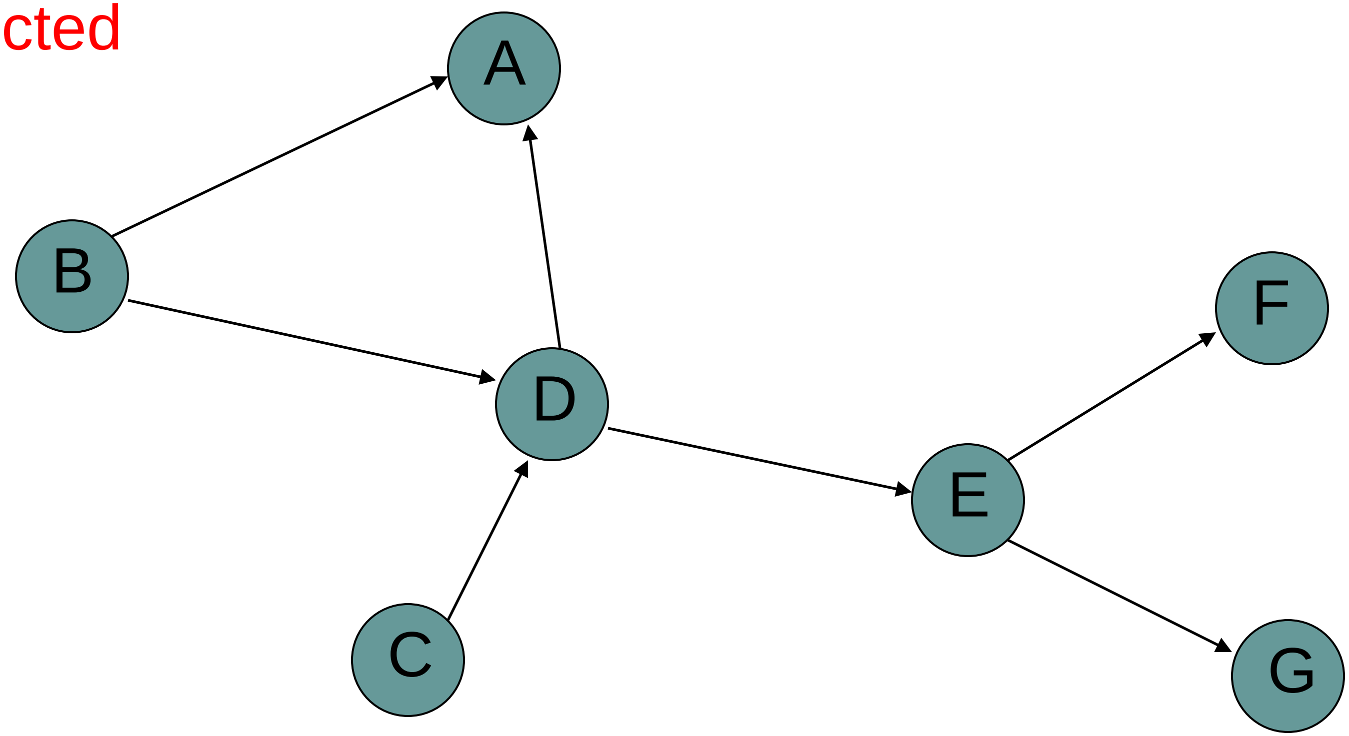
- Connected (undirected graphs) – every pair of vertices is connected by a path



Terminology

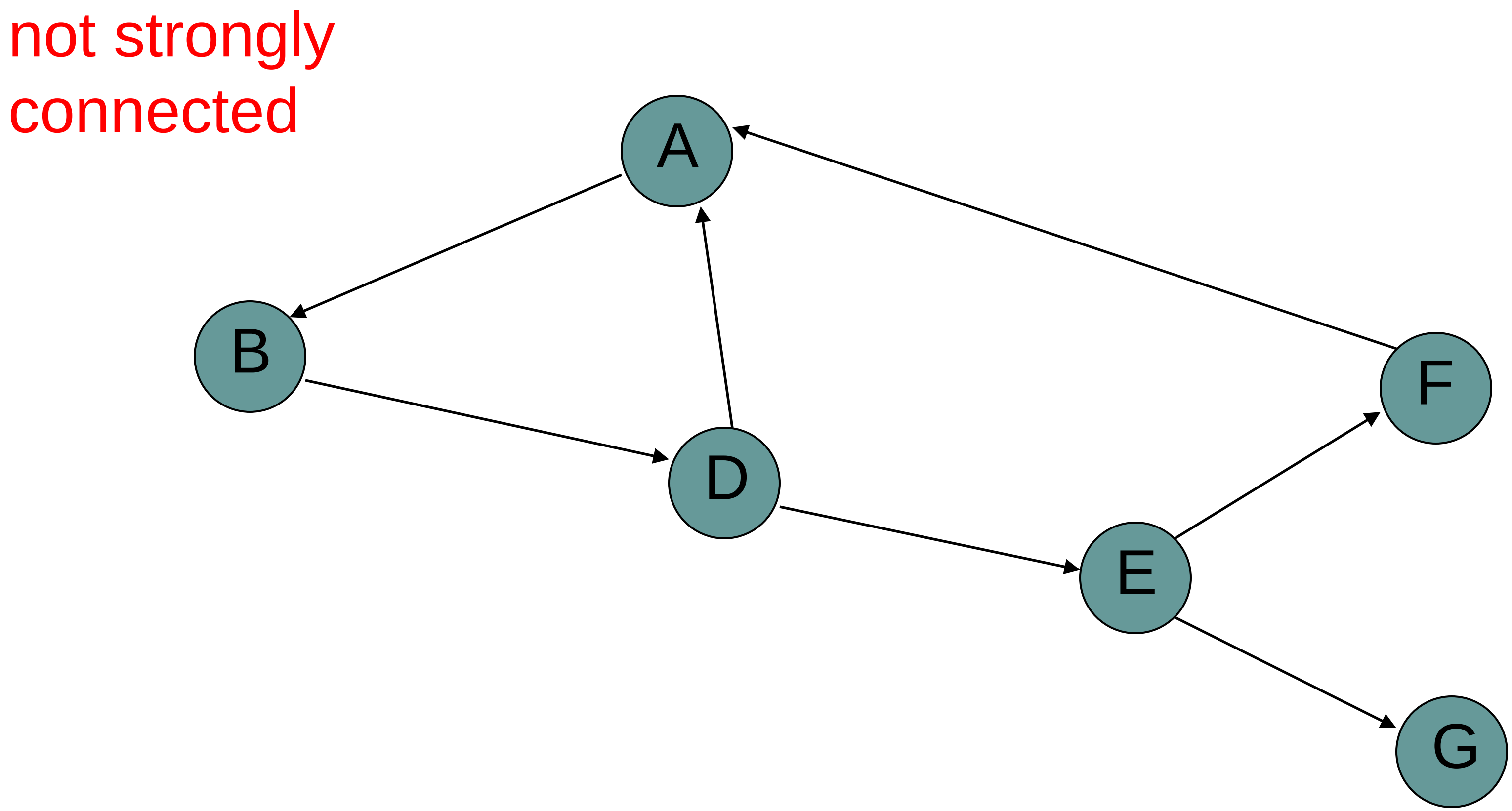
- Strongly connected (directed graphs) – Every two vertices are reachable by a path

not strongly connected



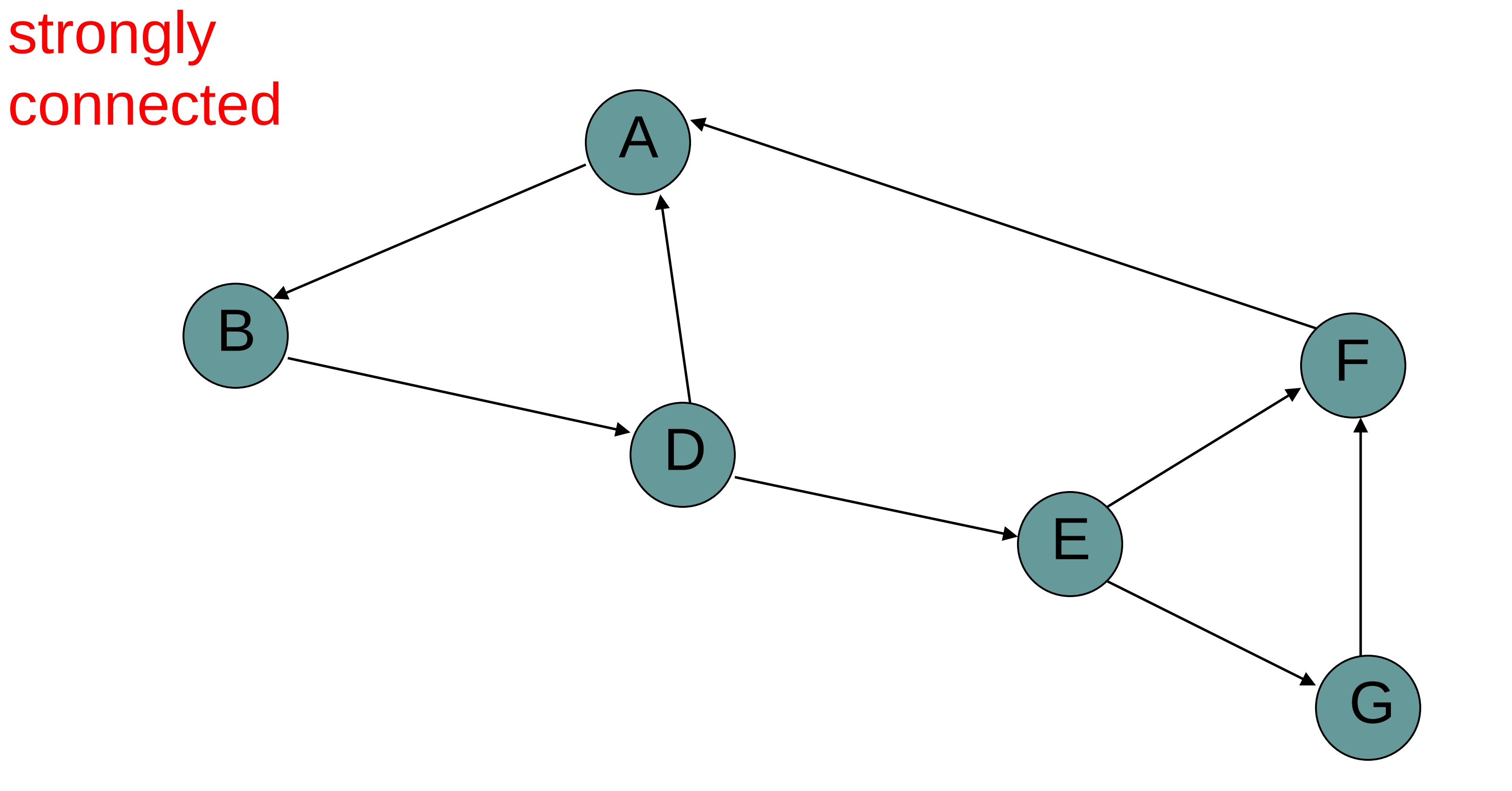
Terminology

- Strongly connected (directed graphs) – Every two vertices are reachable by a path



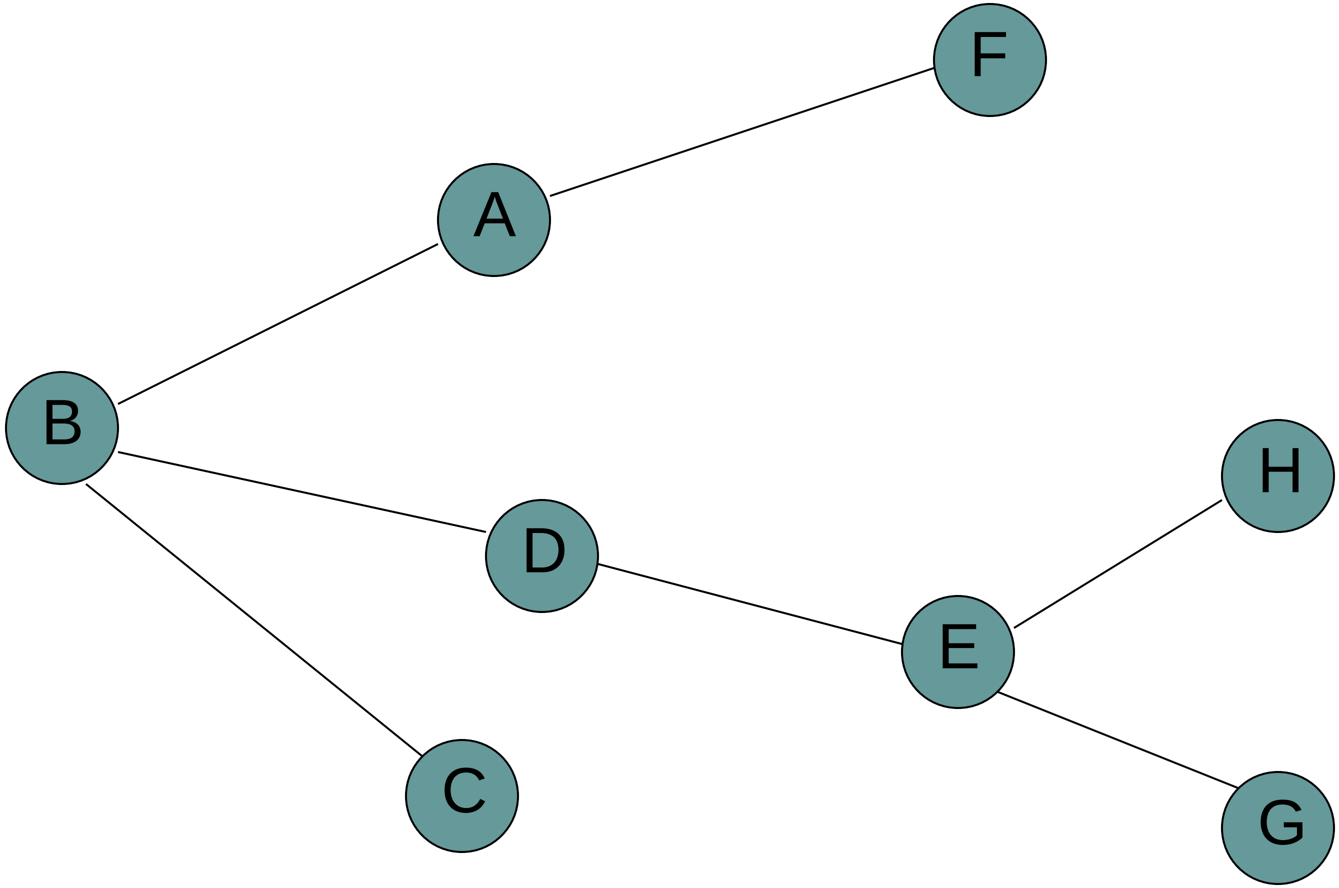
Terminology

- Strongly connected (directed graphs) – Every two vertices are reachable by a path



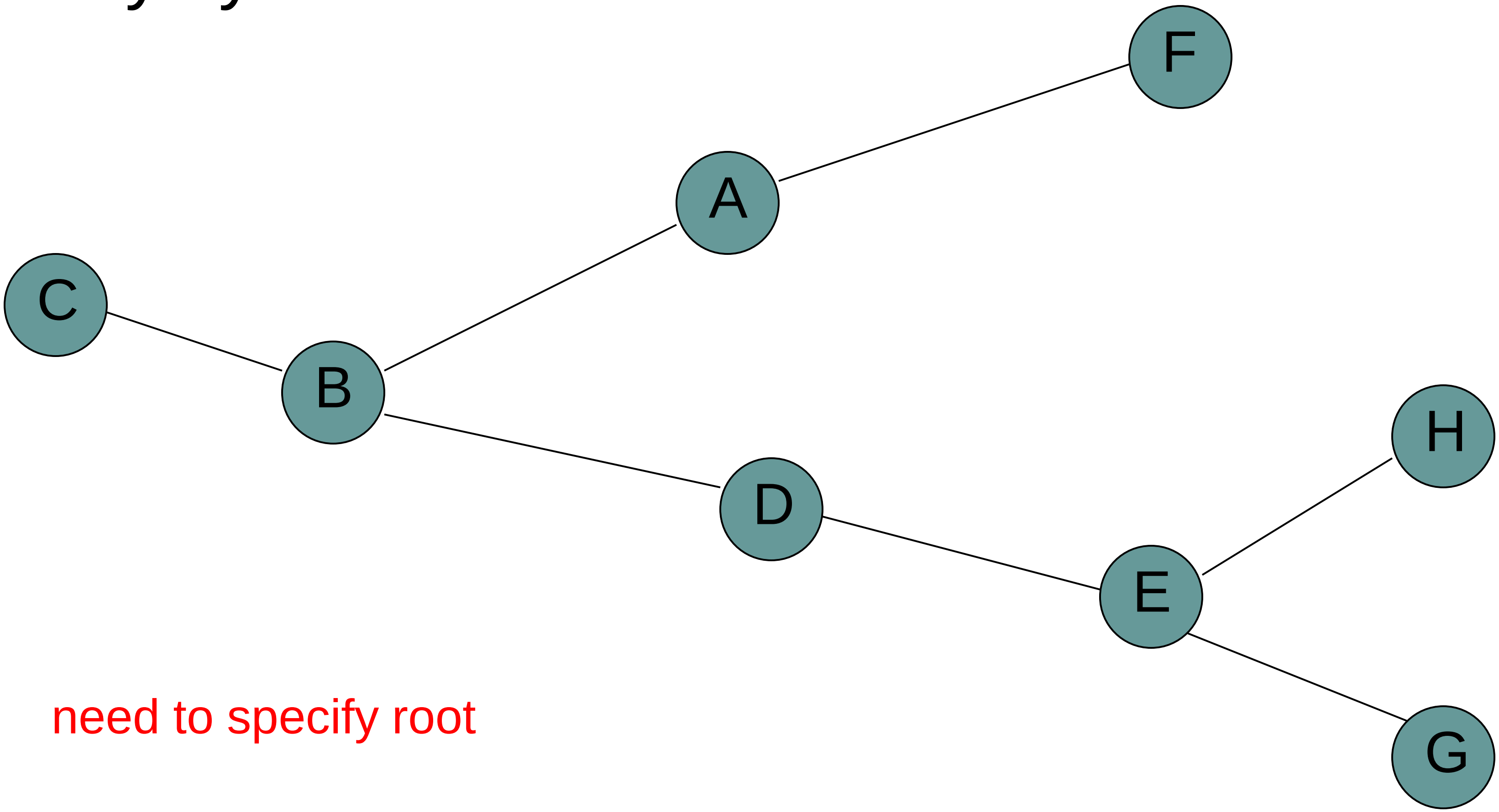
Different types of graphs

- Tree – connected, undirected graph without any cycles



Different types of graphs

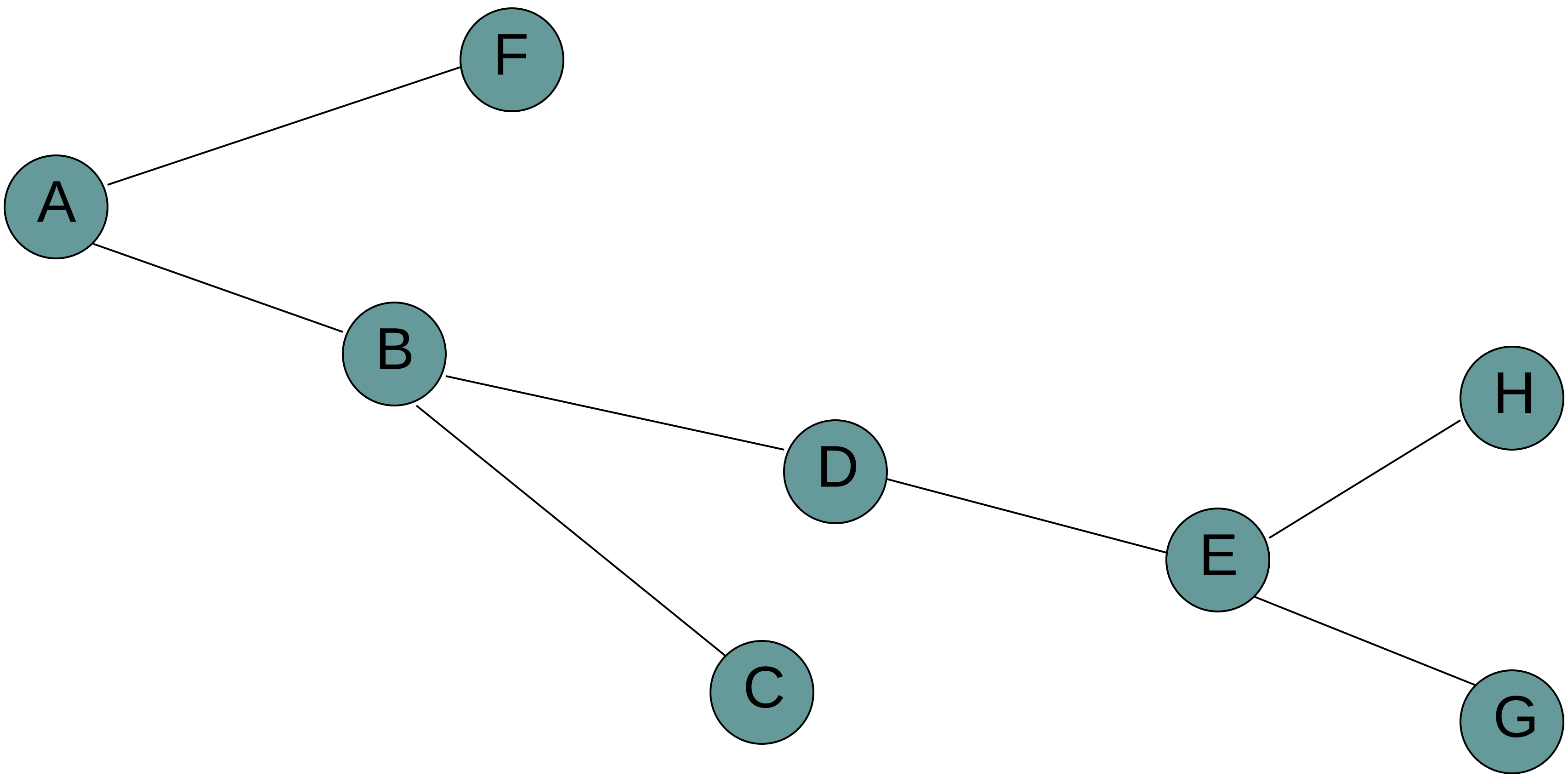
- Tree – connected, undirected graph without any cycles



need to specify root

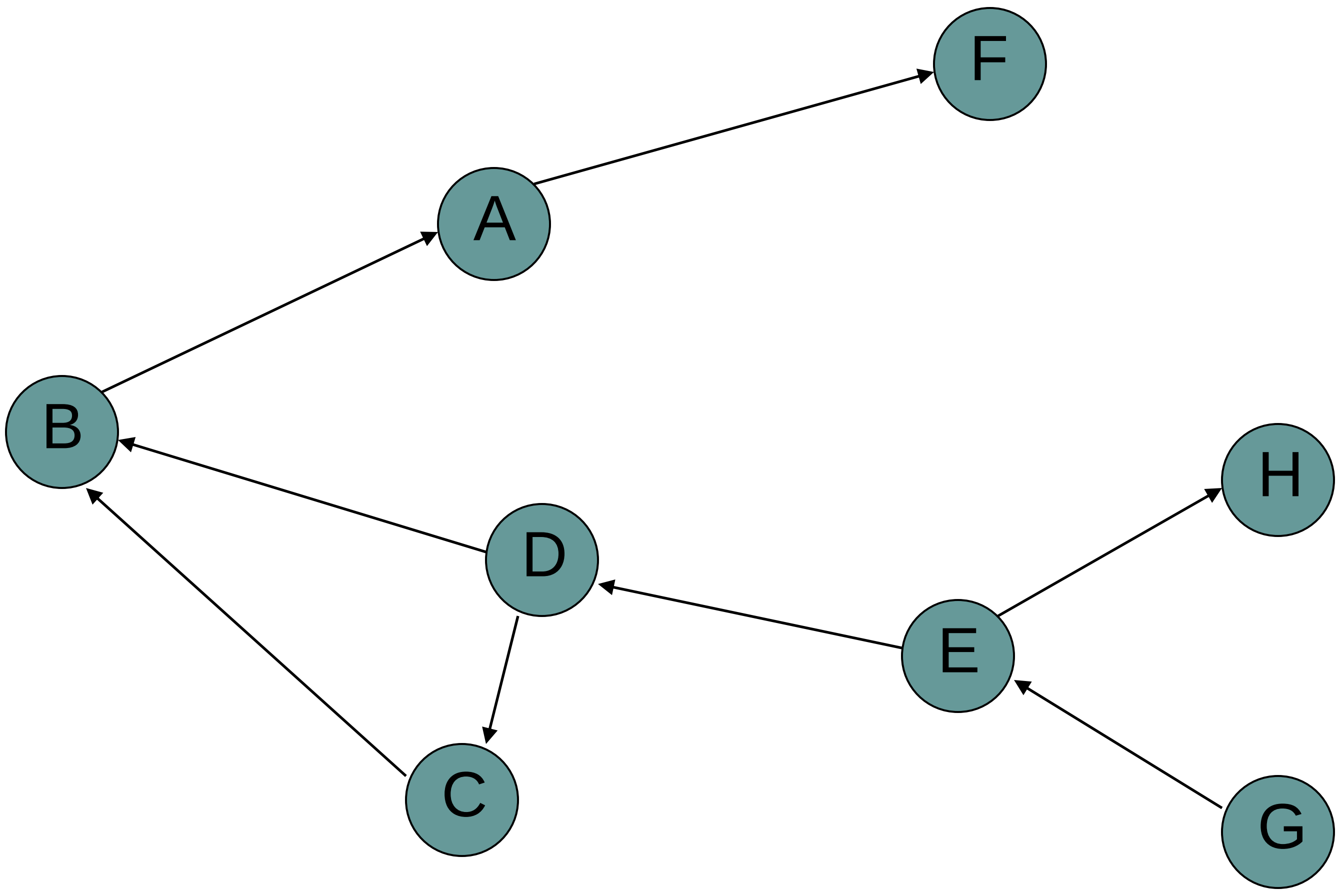
Different types of graphs

- Tree – connected, undirected graph without any cycles



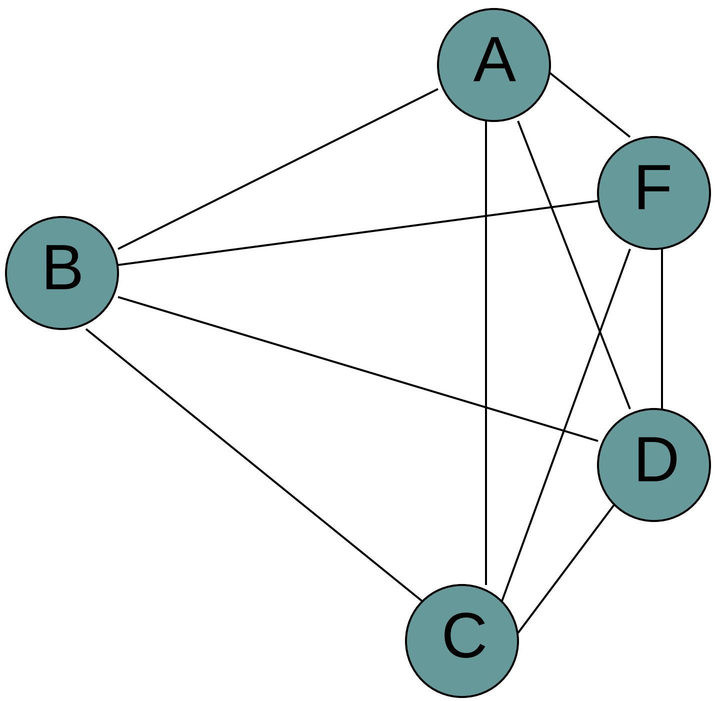
Different types of graphs

- DAG – directed, acyclic graph



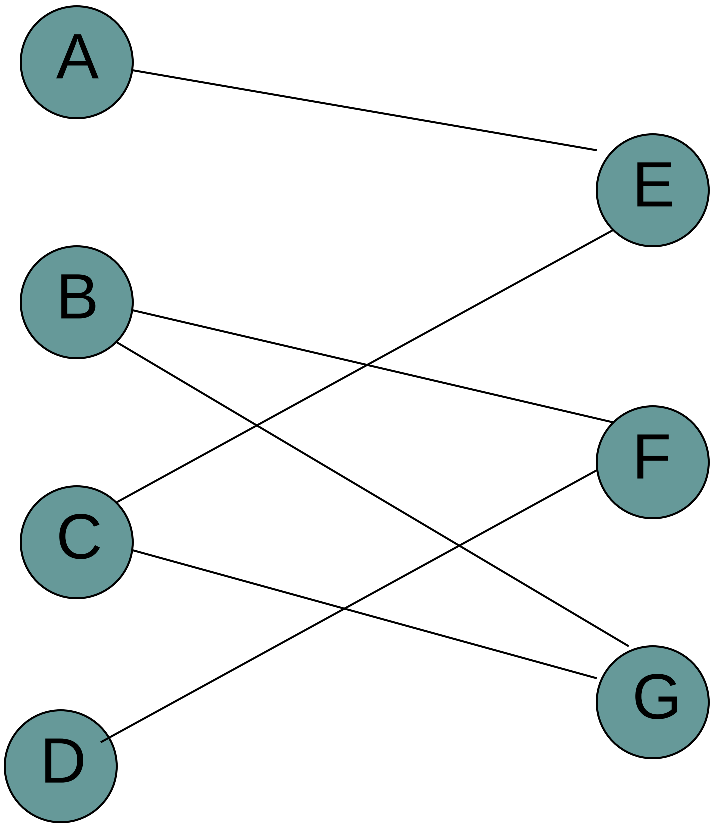
Different types of graphs

- Complete graph – an edge exists between every pair of two nodes



Different types of graphs

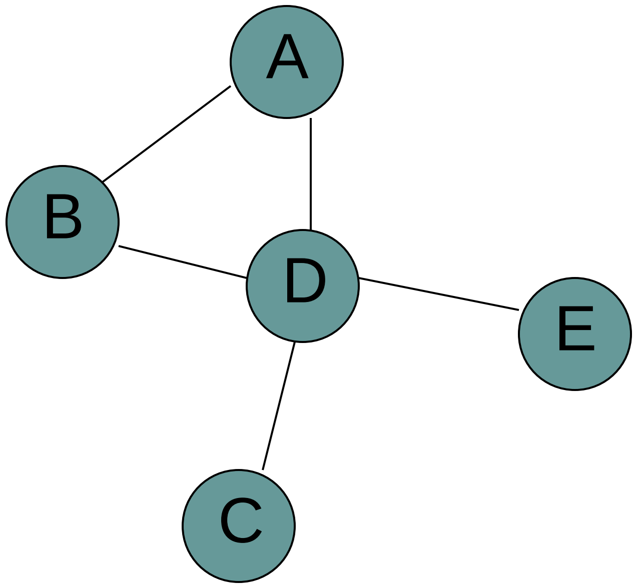
- Bipartite graph – a graph where every vertex can be partitioned into two sets X and Y such that all edges connect a vertex $u \in X$ and a vertex $v \in Y$

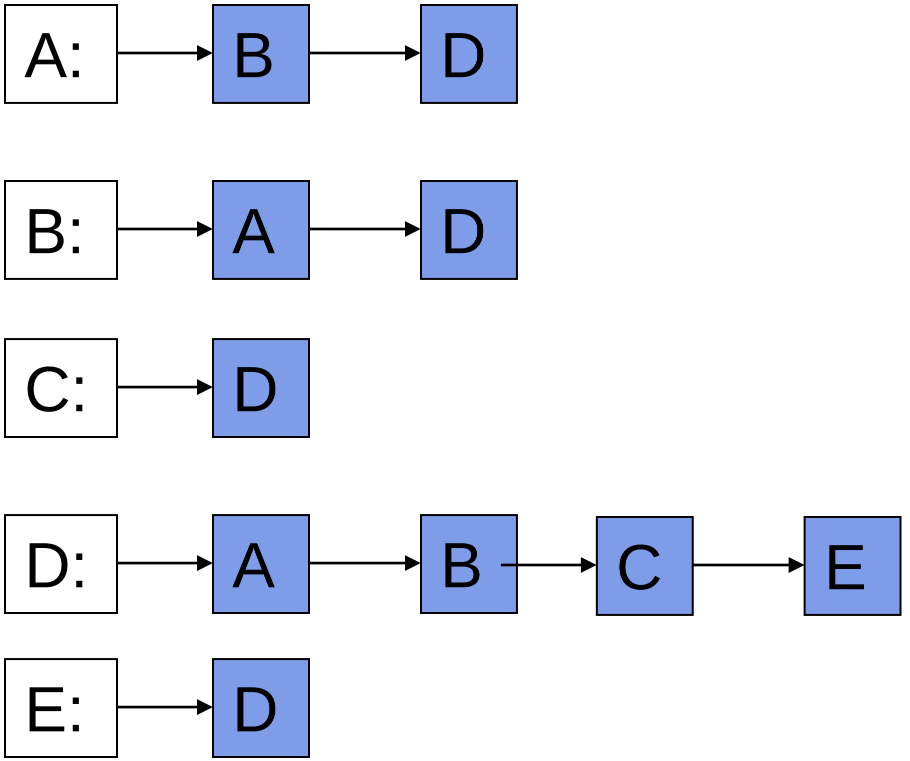


Representing graphs

Representing graphs

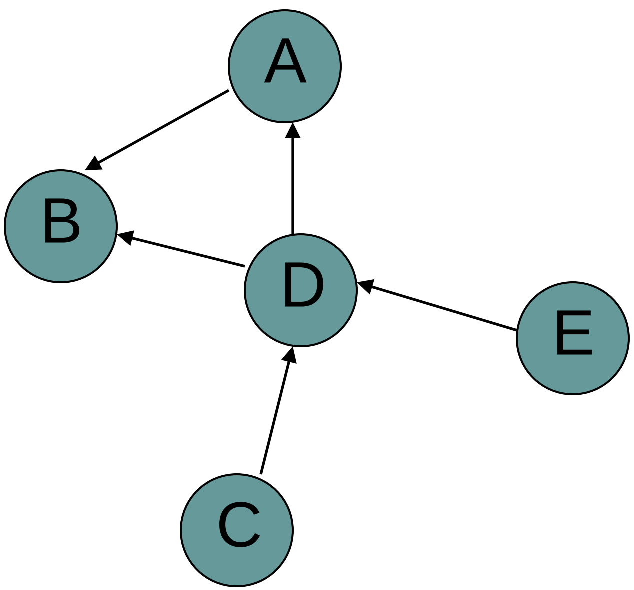
- Adjacency list – Each vertex u ∈ V contains an adjacency list of the set of vertices v such that there exists an edge (u,v) ∈ E

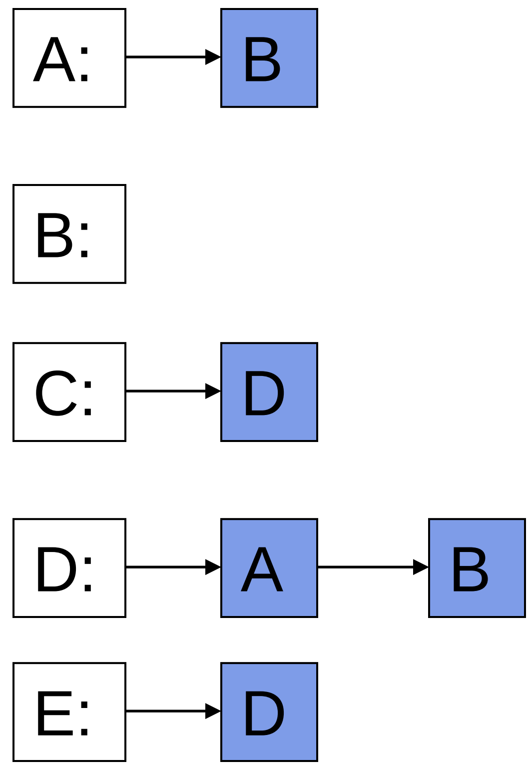




Representing graphs

- Adjacency list – Each vertex u $\in$ V contains an adjacency list of the set of vertices v such that there exists an edge (u,v) $\in$ E

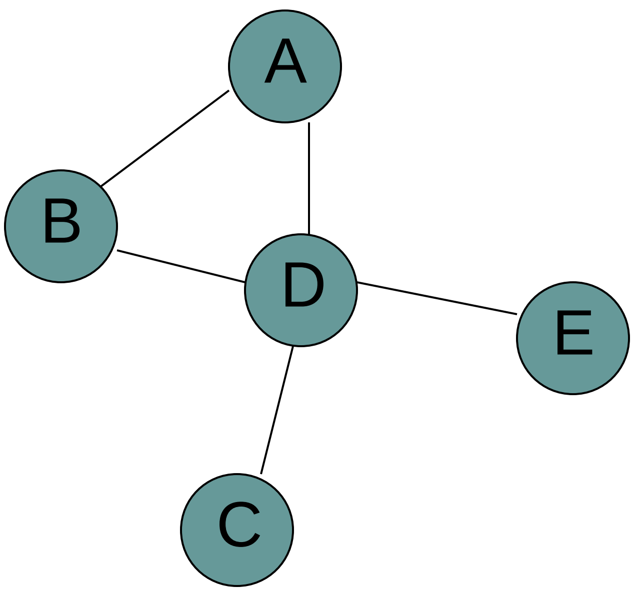




Representing graphs

- Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

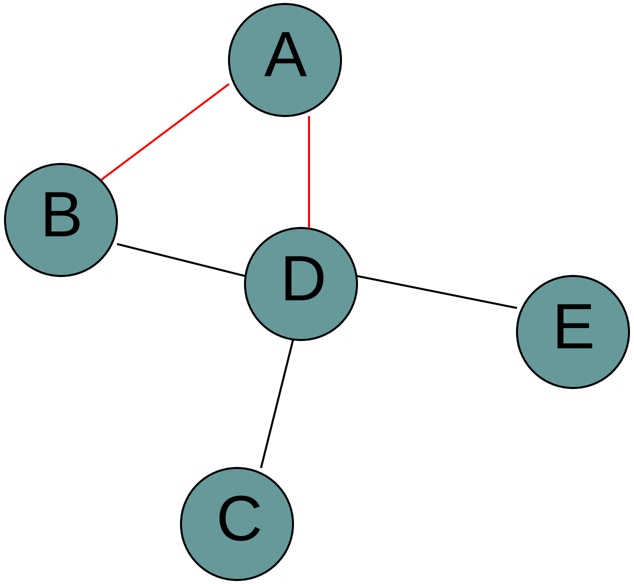


	A	**B**	**C**	**D**	**E**
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

Representing graphs

- Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

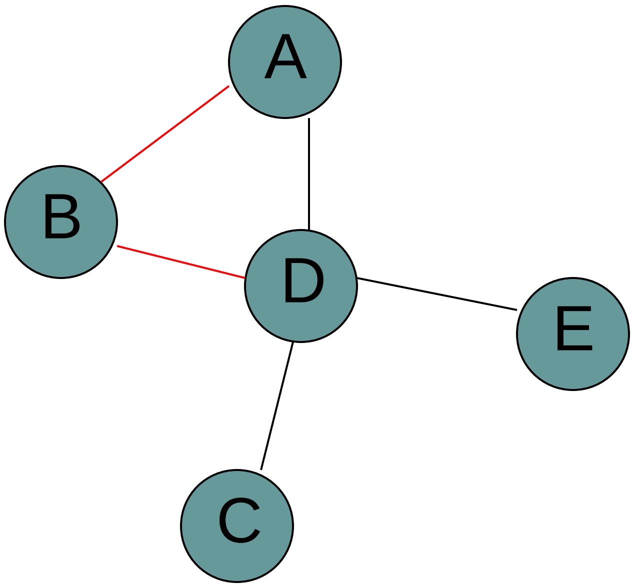


	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

Representing graphs

- Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

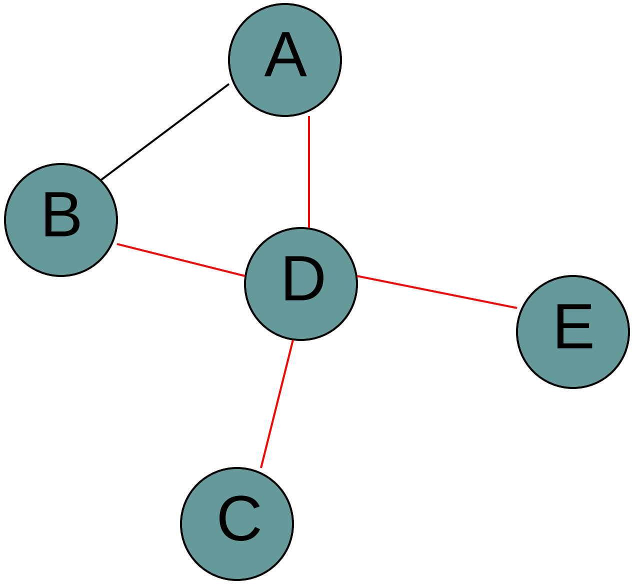


	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

Representing graphs

- Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

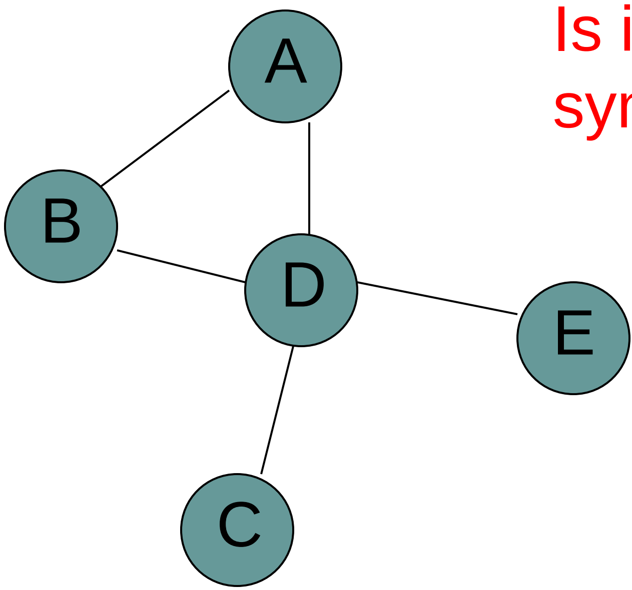


	A	**B**	**C**	**D**	**E**
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	**1**	**1**	**1**	0	**1**
E	0	0	0	1	0

Representing graphs

- Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$



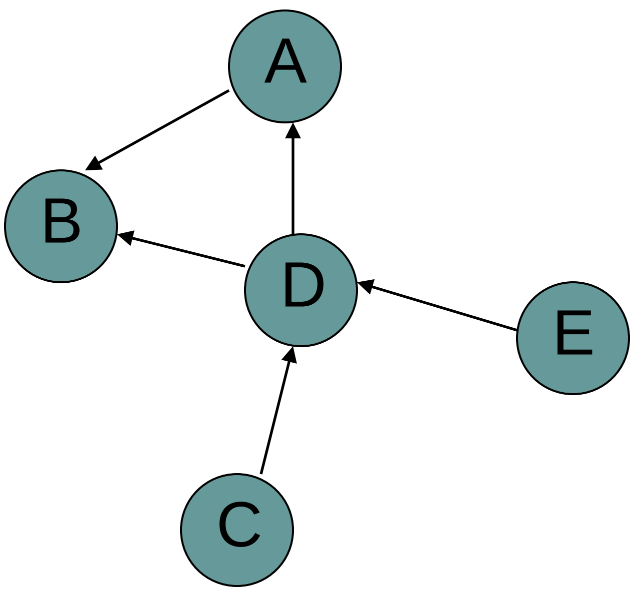
Is it always symmetric?

$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \hline \mathbf{A} & 0 & 1 & 0 & 1 & 0 \\ \mathbf{B} & 1 & 0 & 0 & 1 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 1 & 0 \\ \mathbf{D} & 1 & 1 & 1 & 0 & 1 \\ \mathbf{E} & 0 & 0 & 0 & 1 & 0 \end{array}$$

Representing graphs

- Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$



	A	**B**	**C**	**D**	**E**
A	0	1	0	0	0
B	0	0	0	0	0
C	0	0	0	1	0
D	1	1	0	0	0
E	0	0	0	1	0

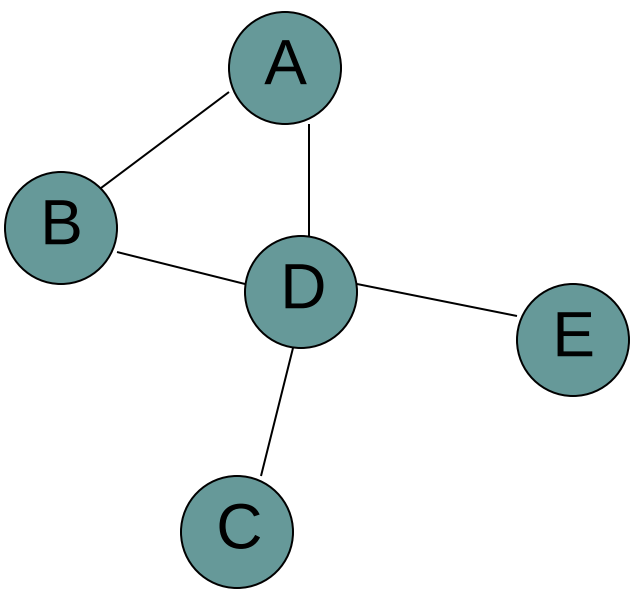
Adjacency list vs. adjacency matrix

Adjacency list	Adjacency matrix
Sparse graphs (e.g. web)	Dense graphs
Space efficient	Constant time lookup to discover if an edge exists
Must traverse the adjacency list to discover is an edge exists	simple to implement
	for non-weighted graphs, only requires boolean matrix

Can we get the best of both worlds?

Sparse adjacency matrix

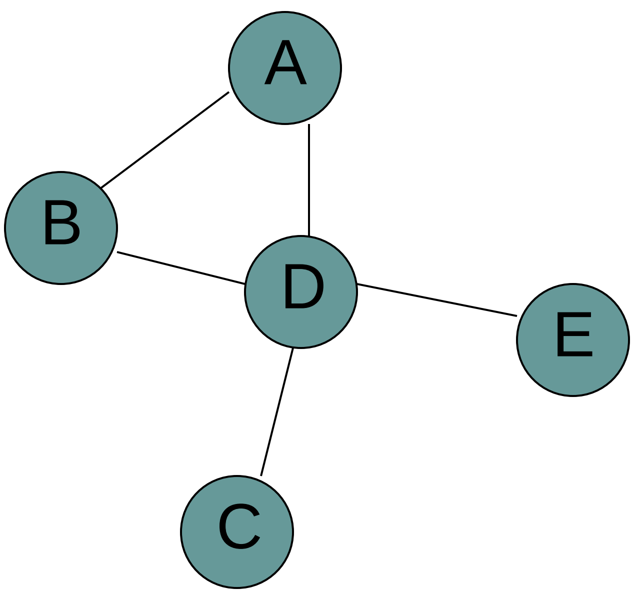
- Rather than using an adjacency list, use an adjacency hashtable



A:	hashtable [B,D]
B:	hashtable [A,D]
C:	hashtable [D]
D:	hashtable [A,B,C,E]
E:	hashtable [D]

Sparse adjacency matrix

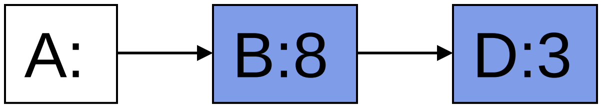
- Constant time lookup
- Space efficient
- Not good for dense graphs

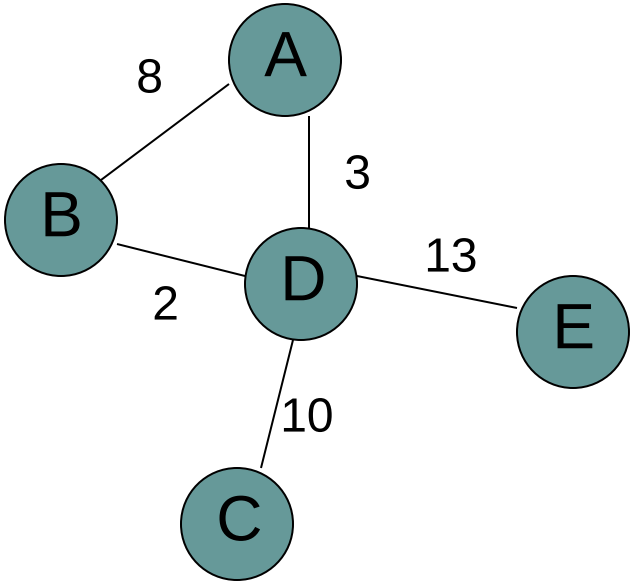


A:	hashtable [B,D]
B:	hashtable [A,D]
C:	hashtable [D]
D:	hashtable [A,B,C,E]
E:	hashtable [D]

Weighted graphs

- Adjacency list
  - store the weight as an additional field in the list

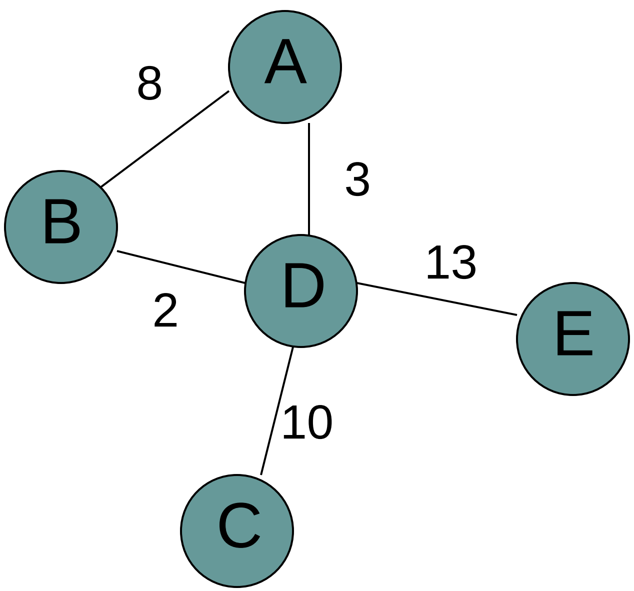




Weighted graphs

- Adjacency matrix

$$a_{ij} = \begin{cases} \mathit{weight} & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$



	A	B	C	D	E
A	0	8	0	3	0
B	8	0	0	2	0
C	0	0	0	10	0
D	3	2	10	0	13
E	0	0	0	13	0

Graph algorithms/questions

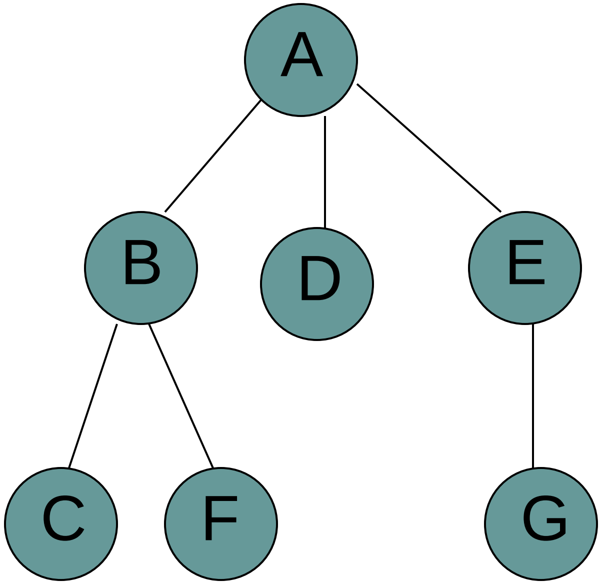
- Graph traversal (BFS, DFS)
- Shortest path from a to b
  - unweighted
  - weighted positive weights
  - negative/positive weights
- Minimum spanning trees
- Are all nodes in the graph connected?
- Is the graph bipartite?

Breadth First Search (BFS) on Trees

```
TreeBFS(T)
    Enqueue(Q, Root(T))
    while !Empty(Q)
          v ← Dequeue(Q)
          Visit(v)
          for all c ∈ Children(v)
                Enqueue(Q, c)
```

Tree BFS

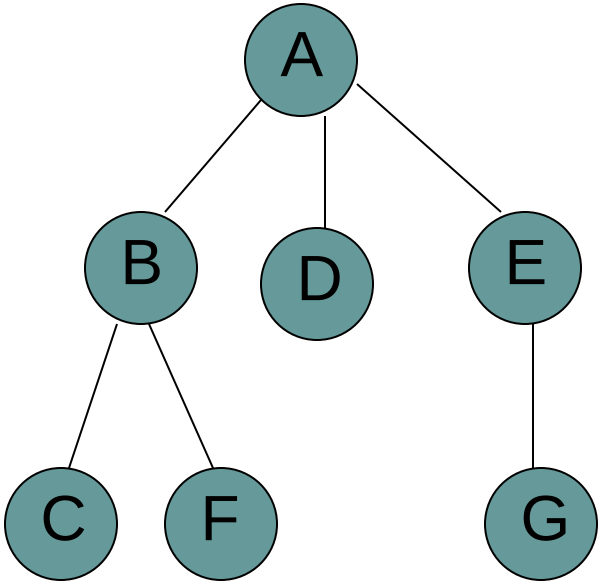
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q:

Tree BFS

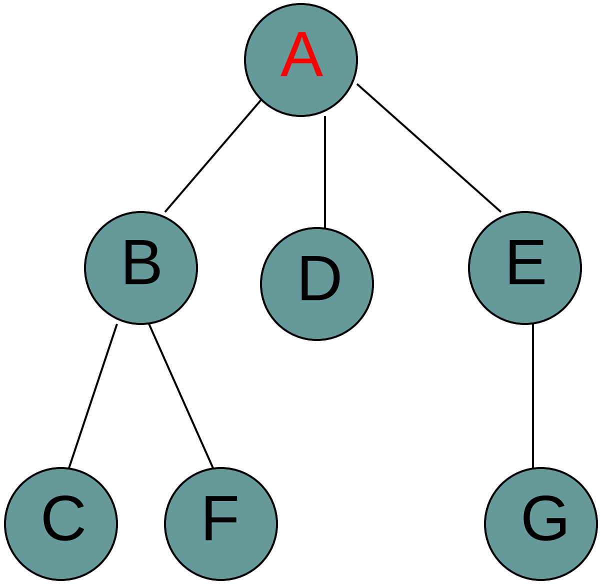
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: A

Tree BFS

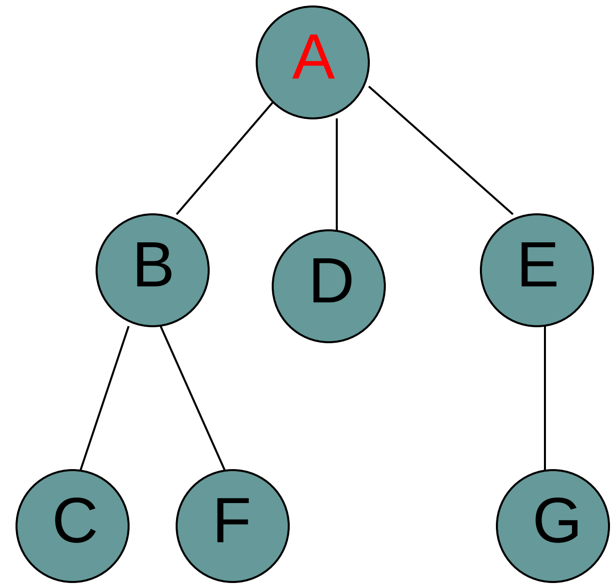
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q:

Tree BFS

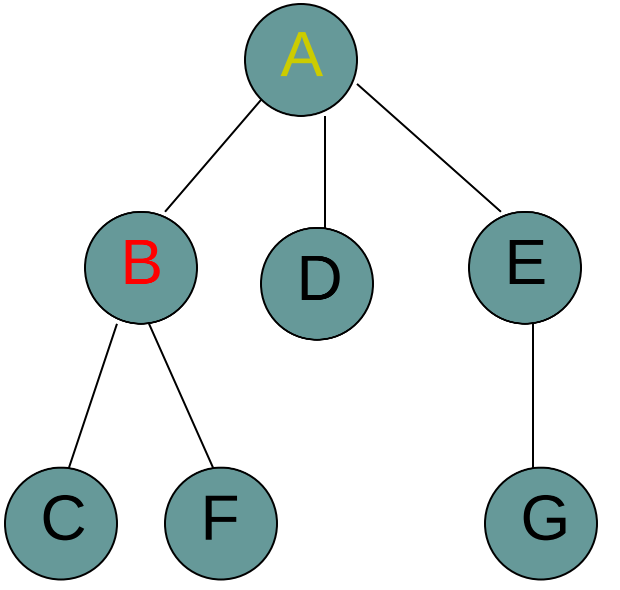
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: B, D, E

Tree BFS

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

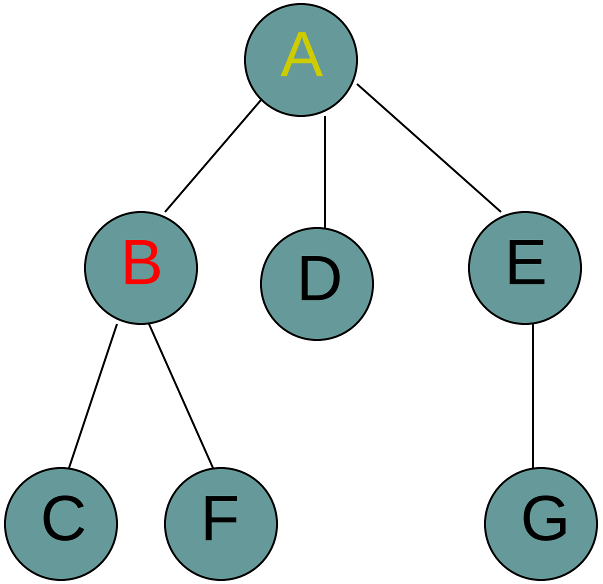


Q: D, E

Tree BFS

TreeBFS(T)

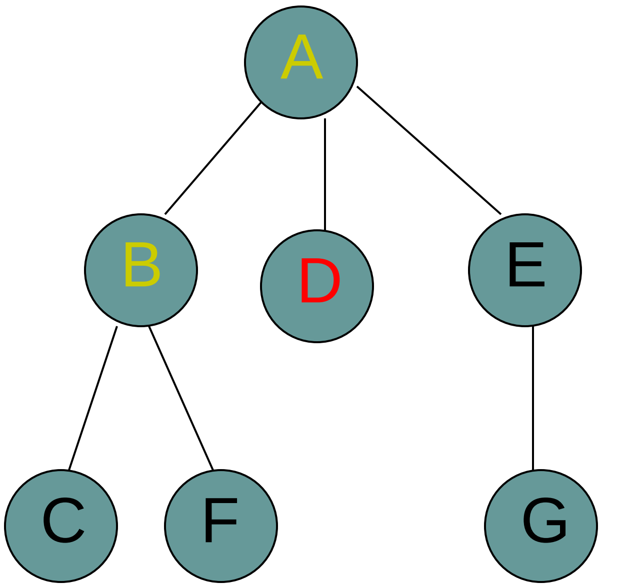
1. Enqueue(Q, Root(T))
2. **while** !Empty(Q)
3. $v \leftarrow$ Dequeue(Q)
4. Visit(v)
5. **for all** $c \in$ Children(v)
6. Enqueue(Q, c)



Q: D, E, C, F

Tree BFS

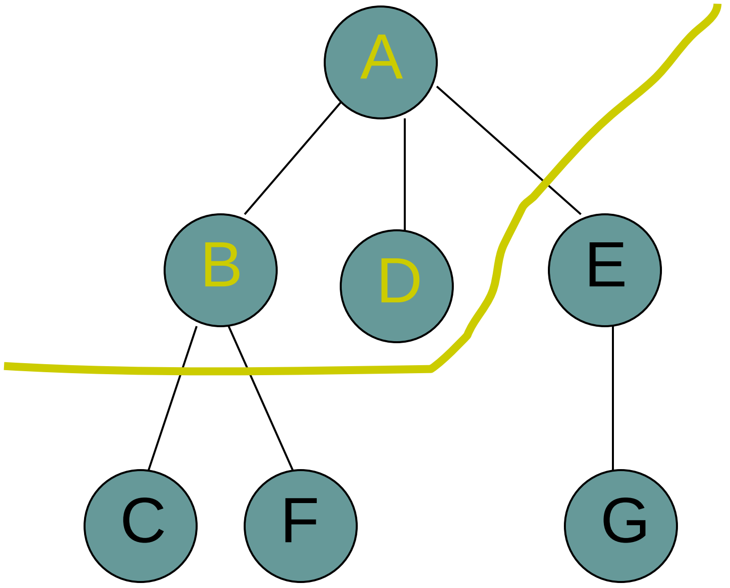
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: E, C, F

Tree BFS

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

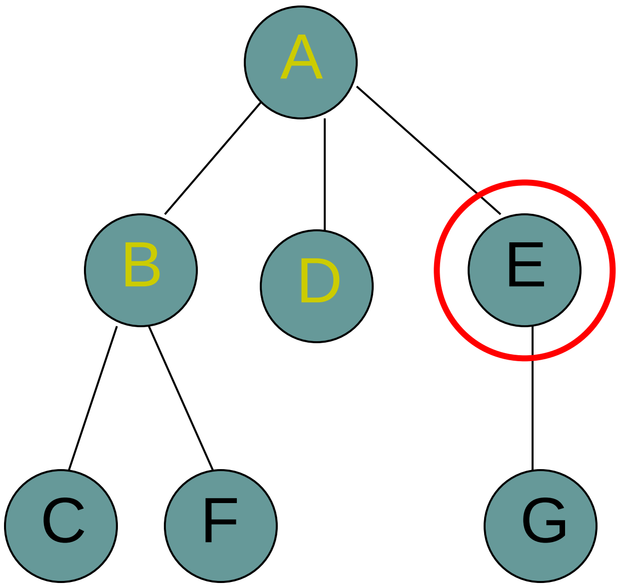


Q: E, C, F

Frontier: the set of vertices that have been visited so far

Tree BFS

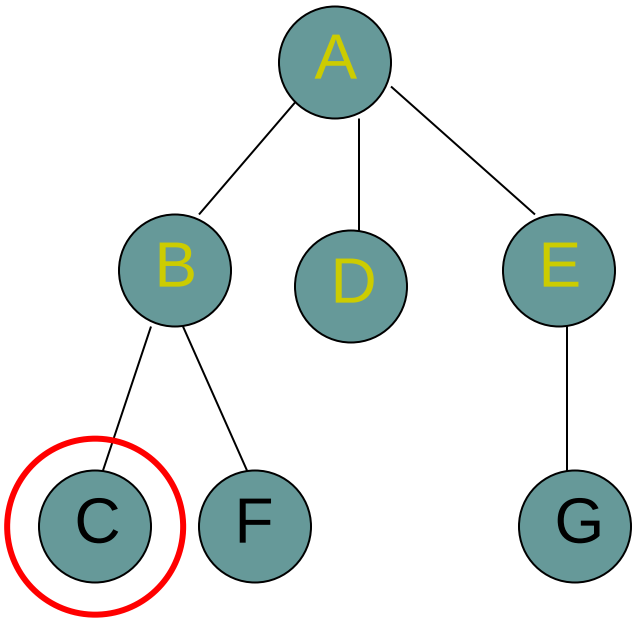
```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
              Enqueue(Q, c)
```



Q: C, F, G

Tree BFS

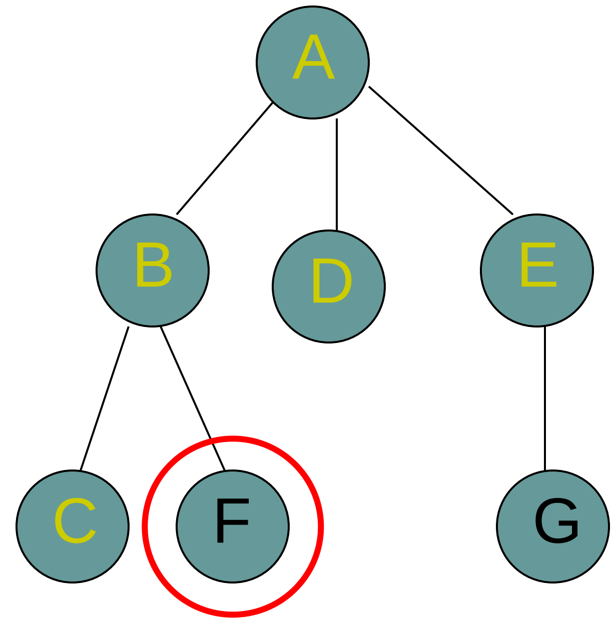
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: F, G

Tree BFS

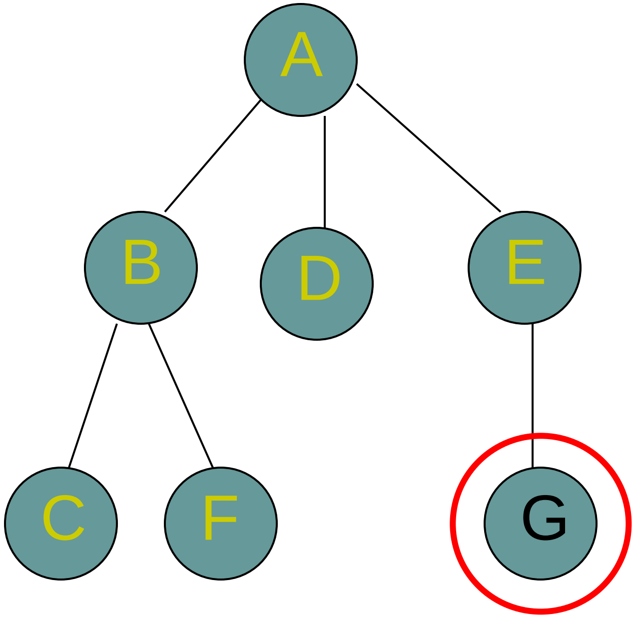
```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
              Enqueue(Q, c)
```



Q: G

Tree BFS

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: Empty

Tree BFS

- What order does the algorithm traverse the nodes?
- BFS traversal visits the nodes in increasing distance from the root

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

Tree BFS

- Does it visit all of the nodes?

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

Running time of Tree BFS

- Adjacency list
  - How many times does it visit each vertex?
  - How many times is each edge traversed?
  - O(|V|+|E|)
- Adjacency matrix
  - For each vertex visited, how much work is done?
  - O(|V|^2)

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

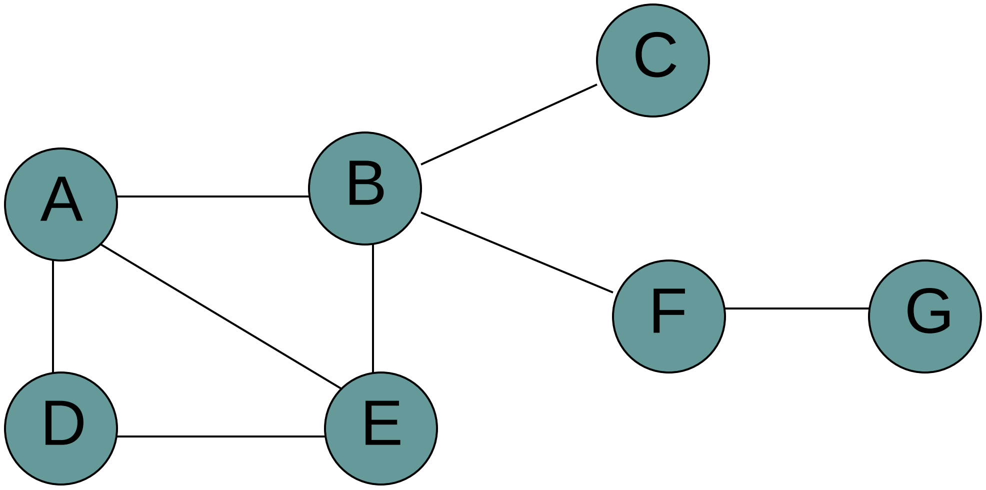
BFS Recursively

Hard to do!

```
TreeBFS(T)
    Enqueue(Q, Root(T))
    while !Empty(Q)
            v ← Dequeue(Q)
            Visit(v)
            for all c ∈ Children(v)
                    Enqueue(Q, c)
```

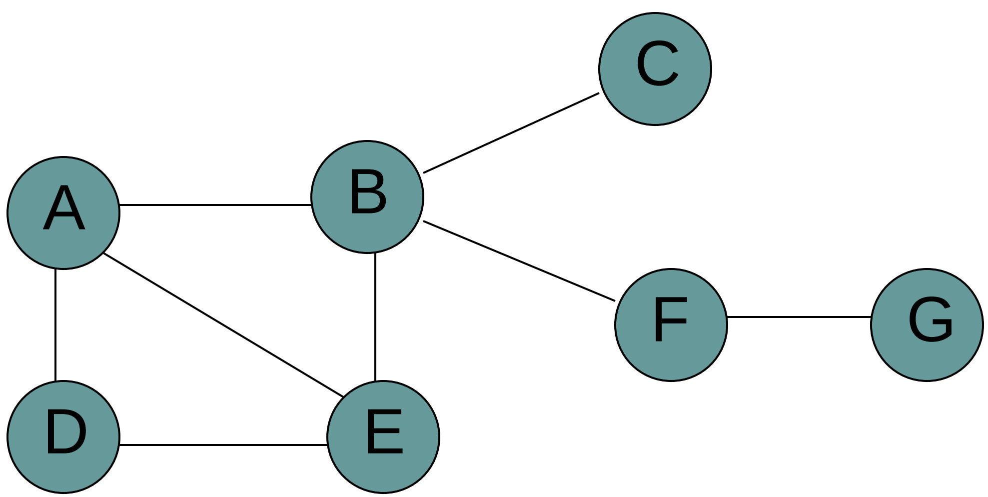
BFS for graphs

- What needs to change for graphs?
- Need to make sure we don't visit a node multiple times



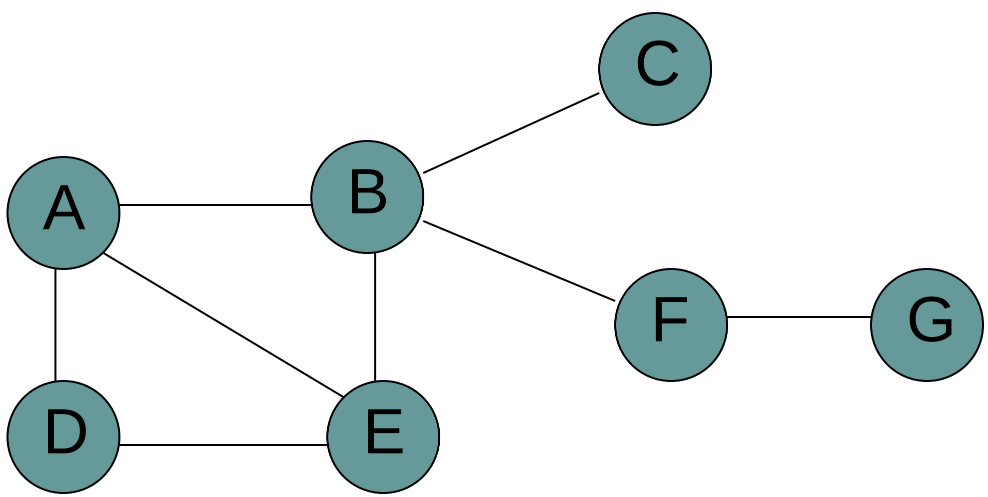
```
BFS(G, s)
  for each v ∈ V
        dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
              if dist[v] = ∞
                    Enqueue(Q, v)
                    dist[v] ← dist[u] + 1
```

distance variable keeps track of how far from the starting node and whether we've seen the node yet



```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

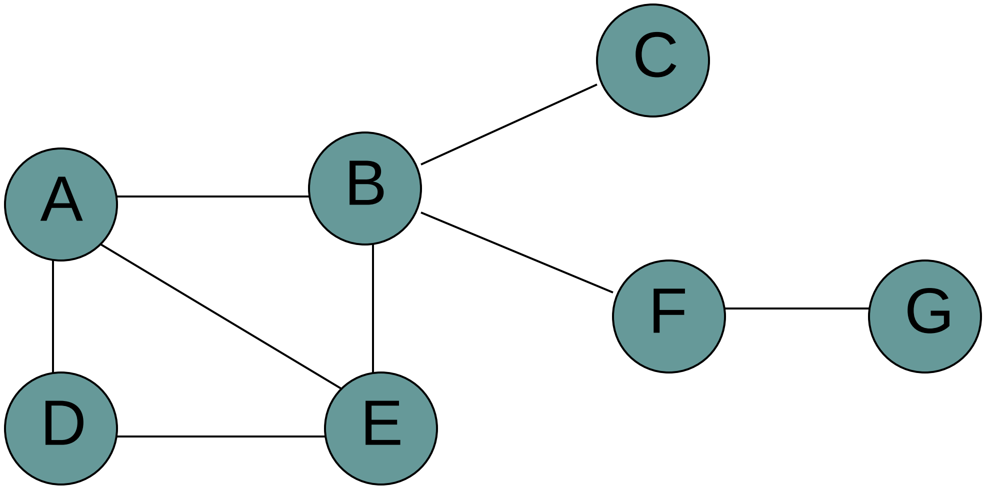
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



BFS(G, s)

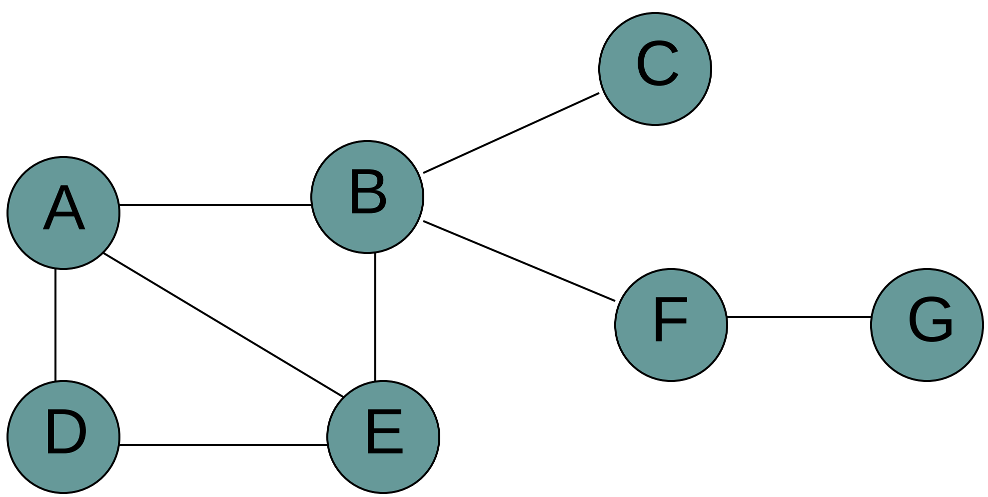
```
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

set all nodes
as unseen



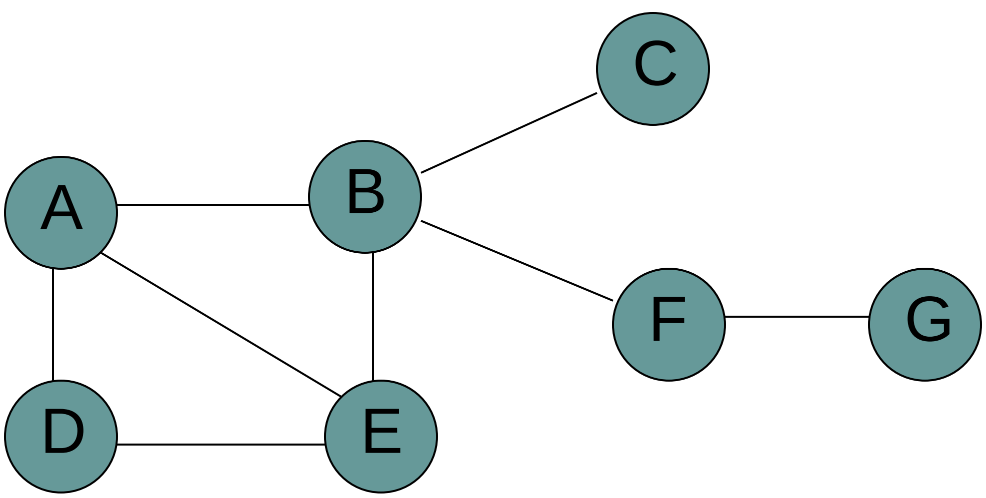
```
BFS(G, s)
    for each v ∈ V
            dist[v] = ∞
    dist[s] = 0
    ENQUEUE(Q, s)
    while !EMPTY(Q)
            u ← DEQUEUE(Q)
            VISIT(U)
            for each edge (u, v) ∈ E
                    if dist[v] = ∞
                            ENQUEUE(Q, v)
                            dist[v] ← dist[u] + 1
```

check if the node has been seen



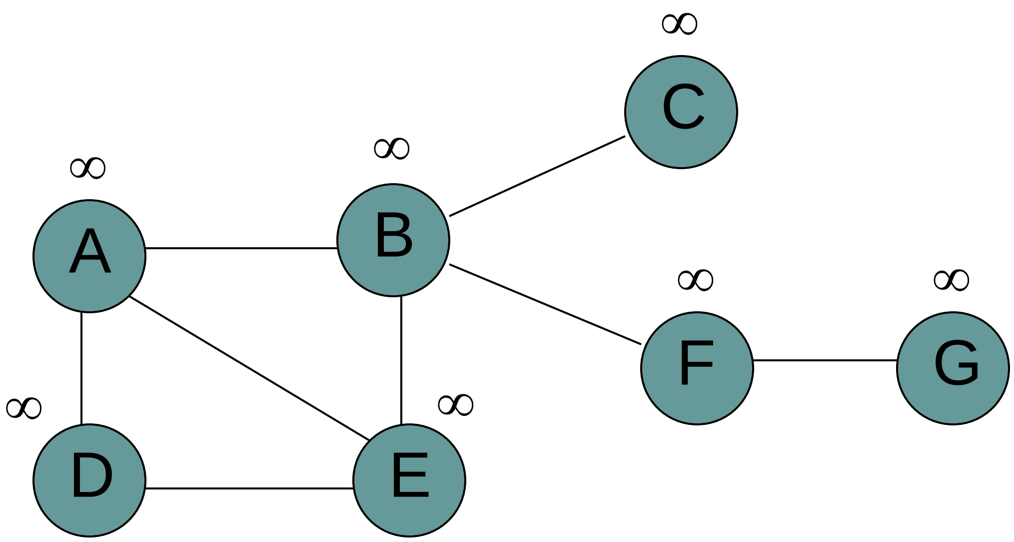
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

set the node as seen
and record distance



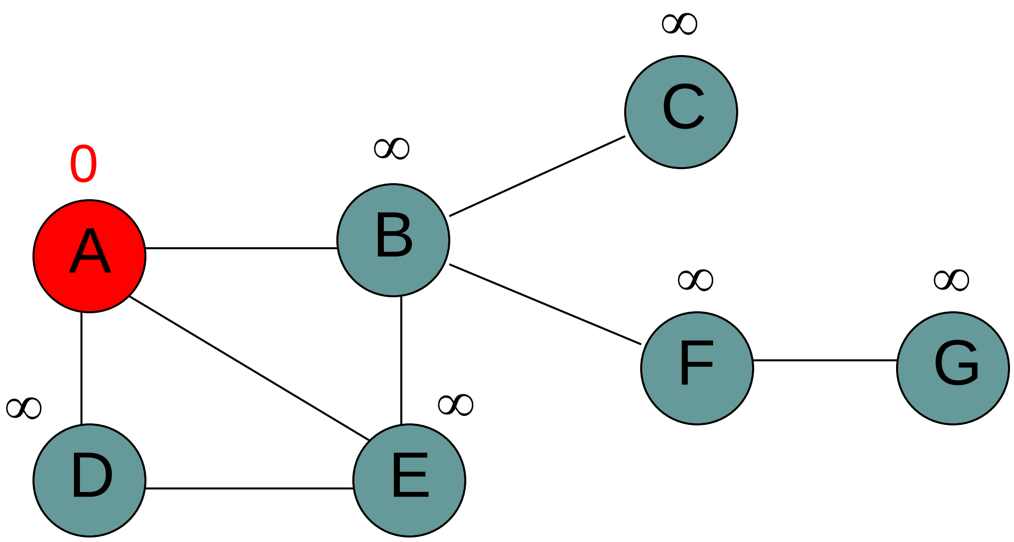
BFS(G, s)

```
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```



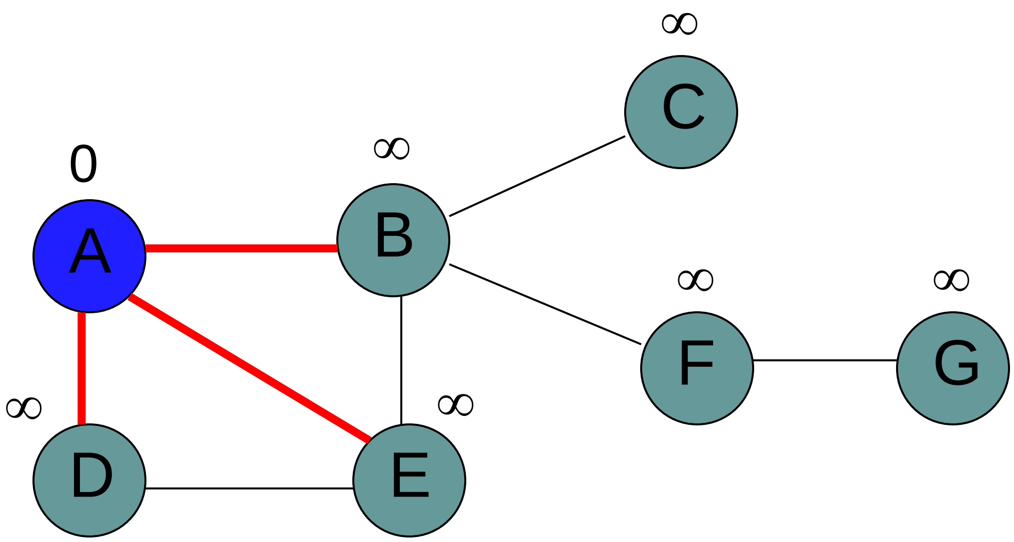
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: A



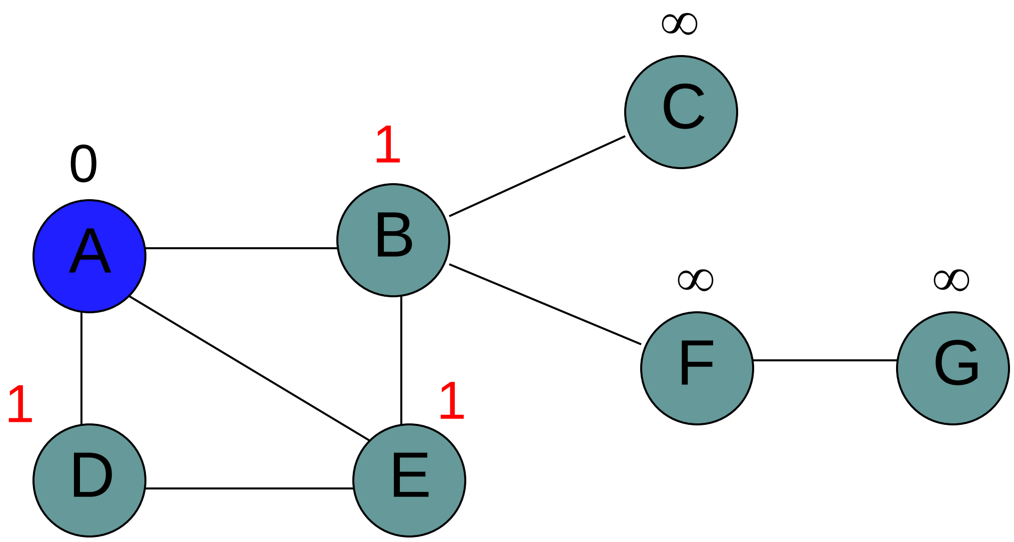
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q:



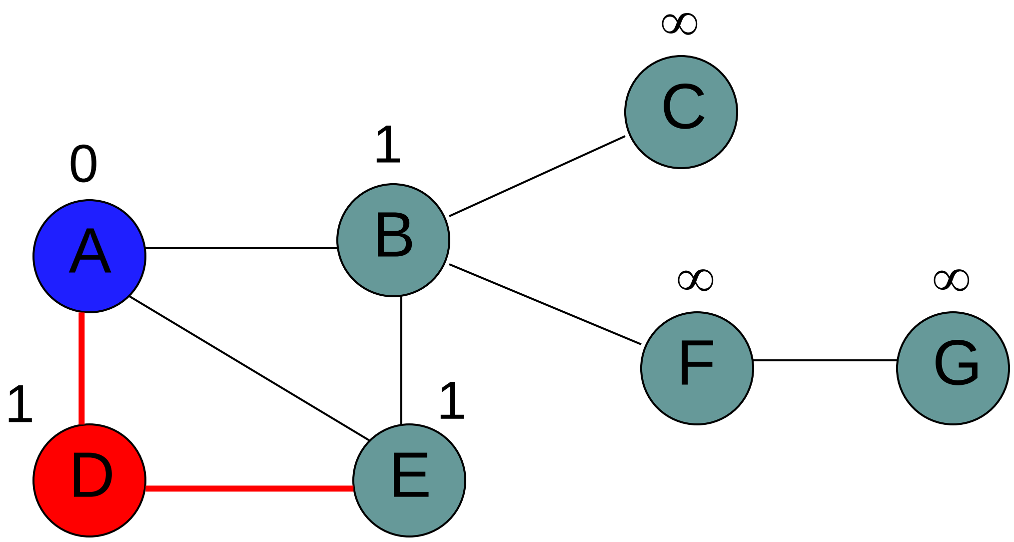
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: D, E, B



```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

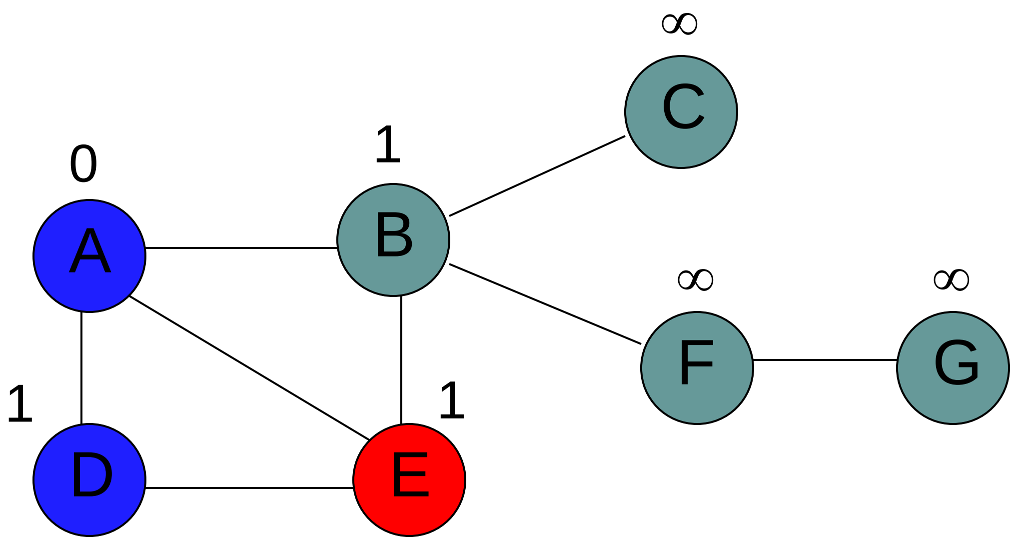
Q: E, B



BFS(G, s)

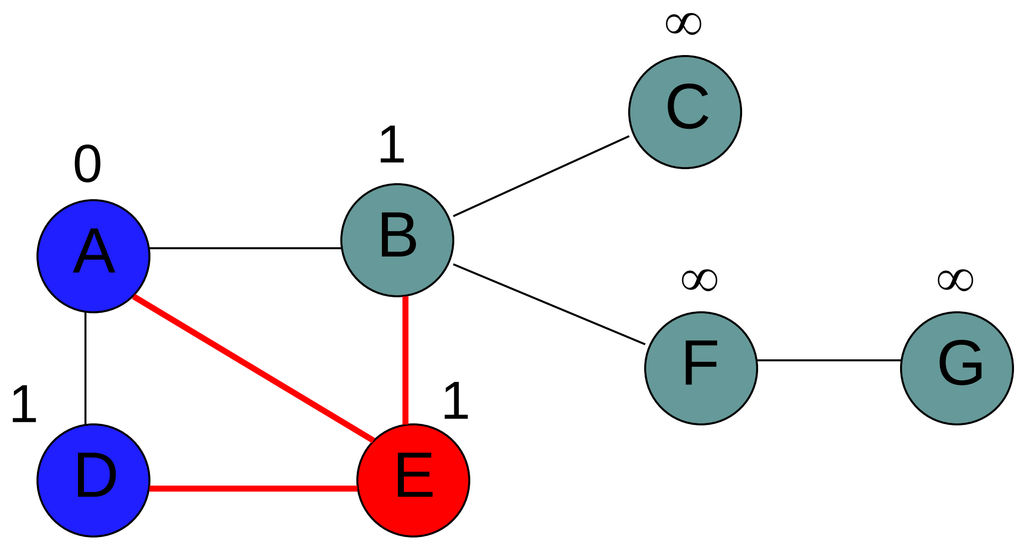
```
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(U)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```

Q: B



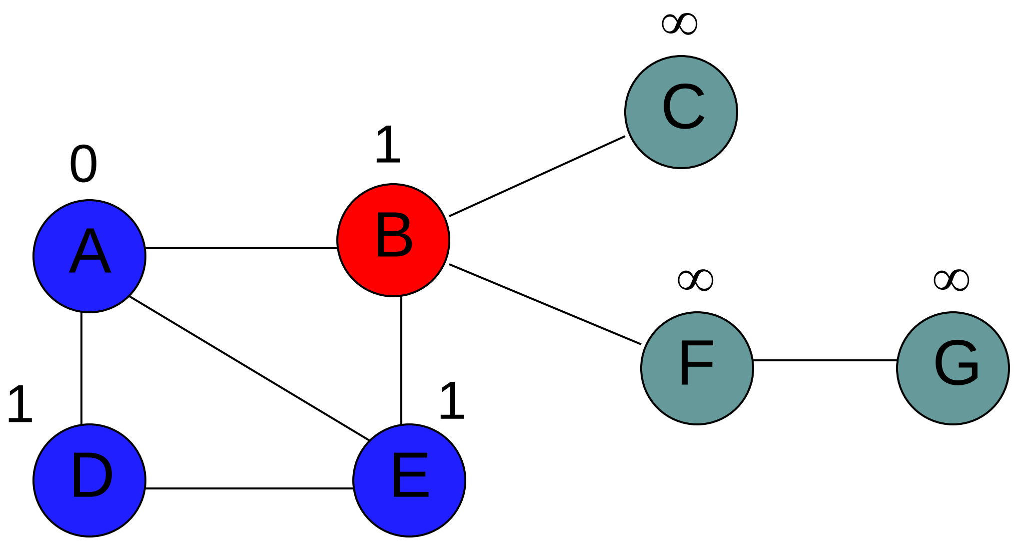
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: B



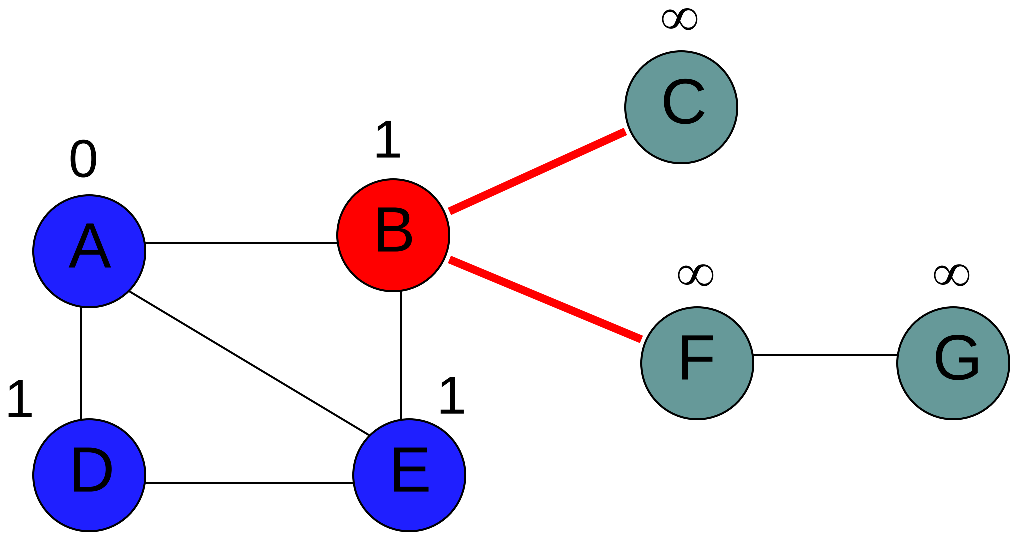
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q:



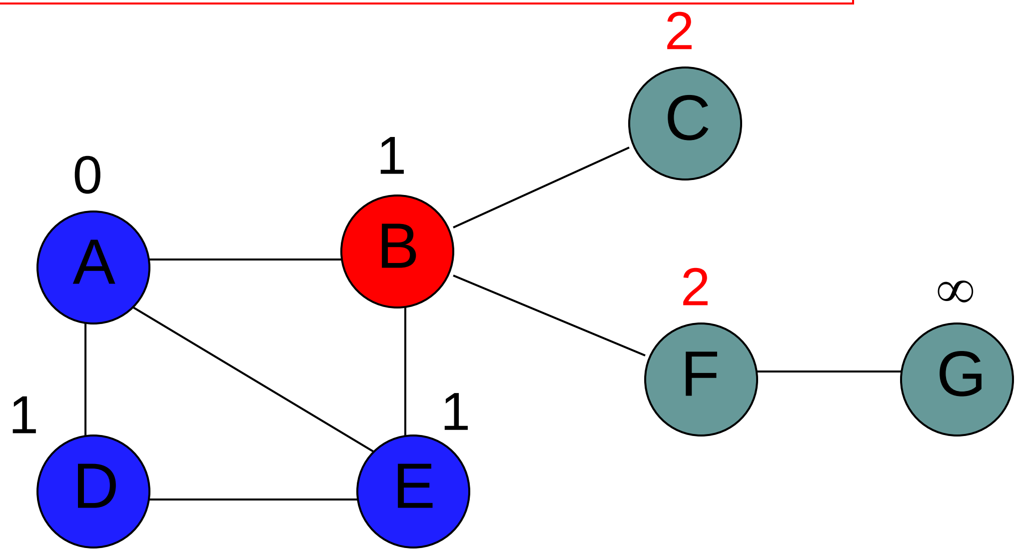
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Q:



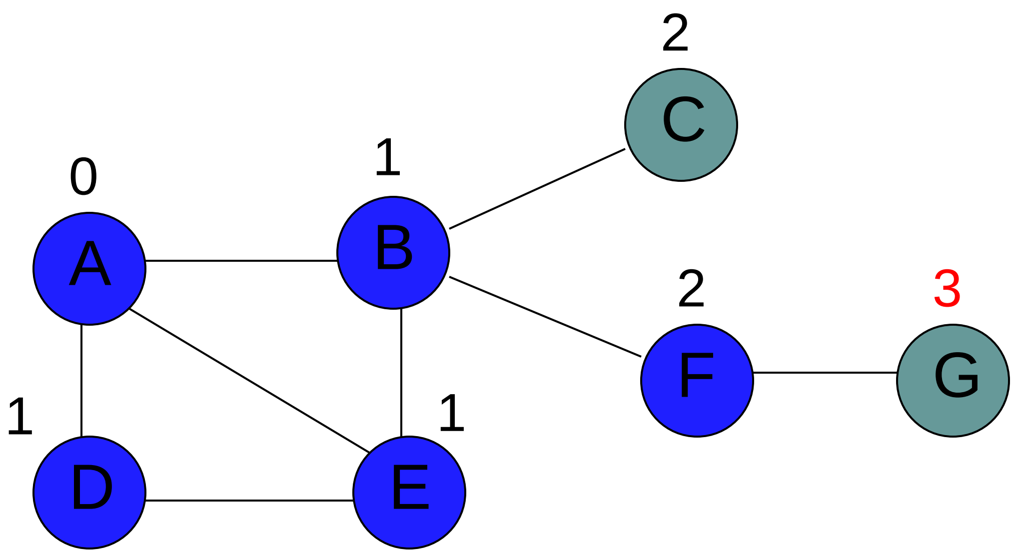
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: F, C

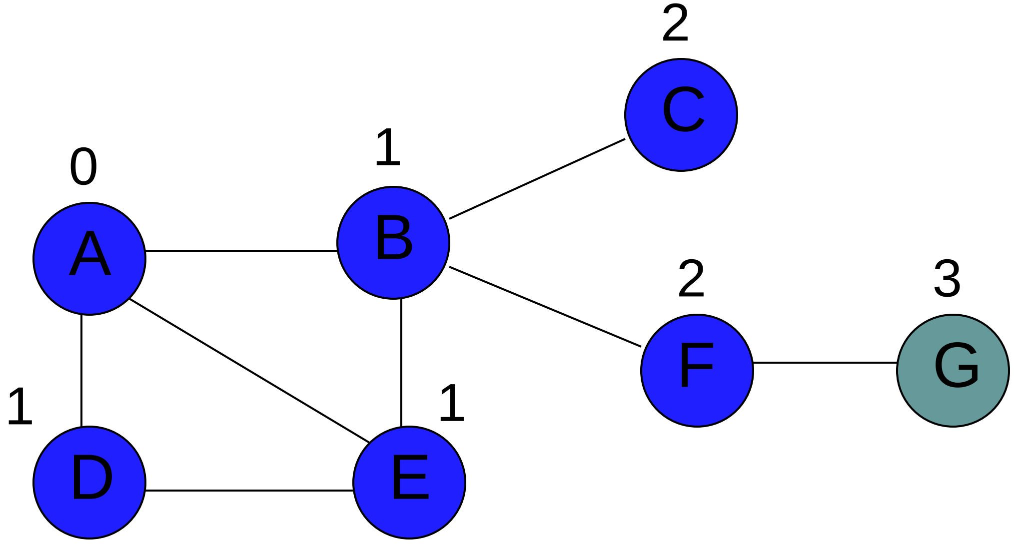


BFS(G, s)

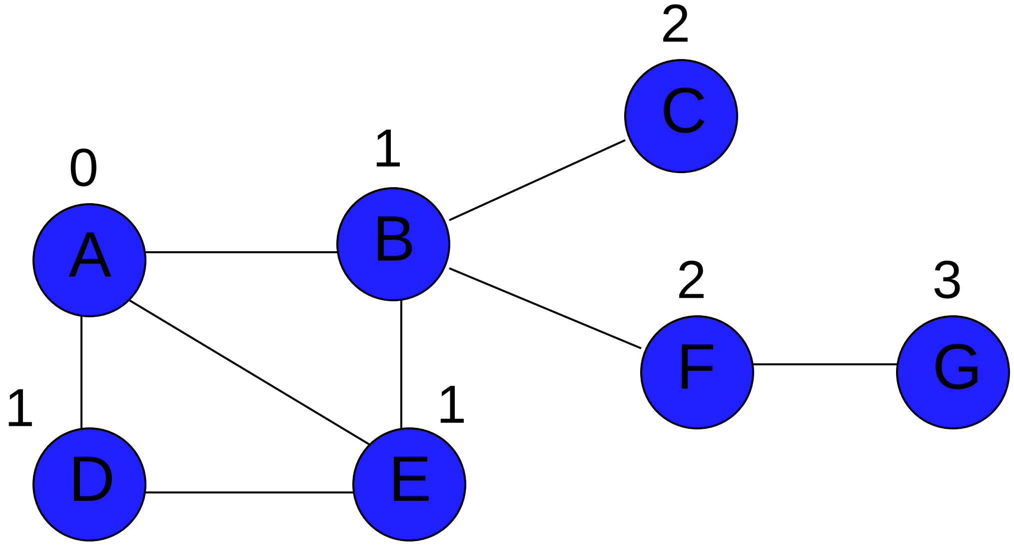
```
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```



```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

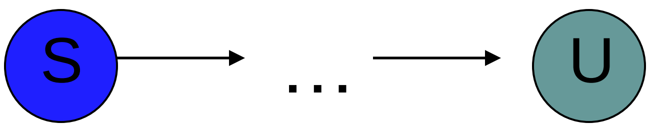


```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```



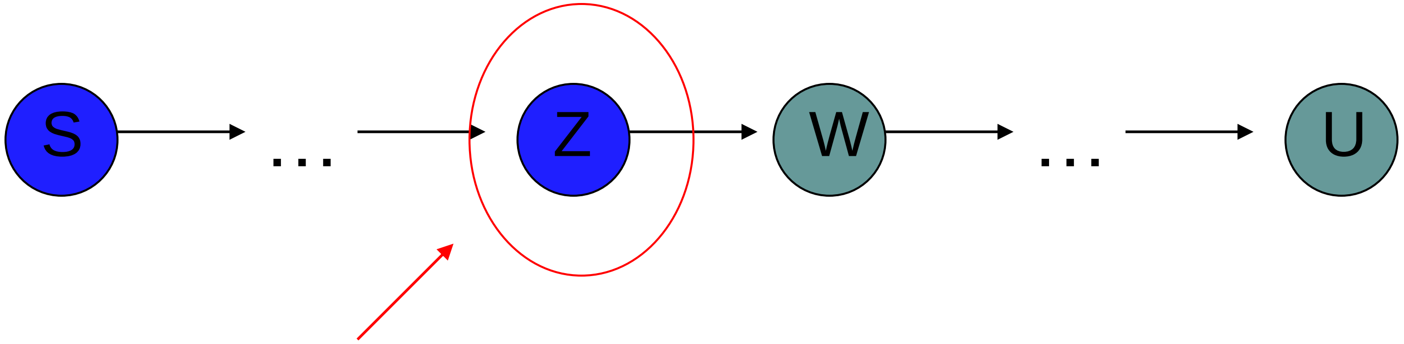
Is BFS correct?

- Does it visit all nodes reachable from the starting node?
- Can you prove it?
- Assume we "miss" some node 'u', i.e. a path exists, but we don't visit 'u'



Is BFS correct?

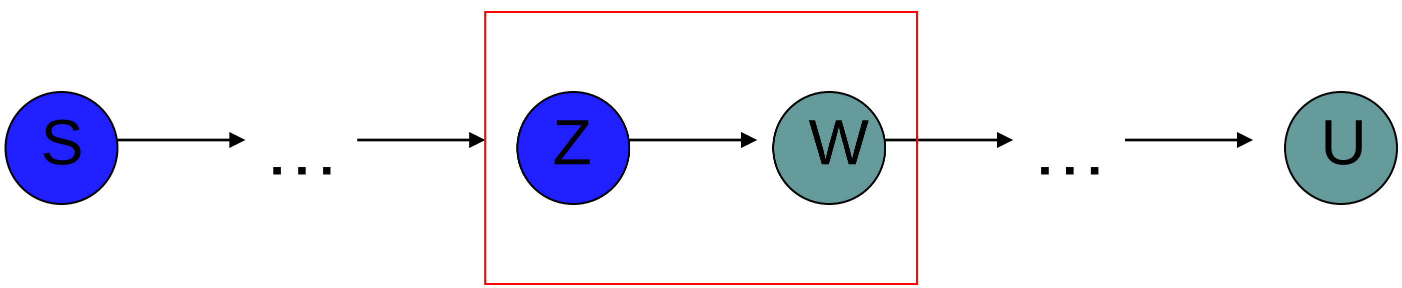
- Does it visit all nodes reachable from the starting node?
- Can you prove it?
- Find the last node along the path to ‘u’ that was visited



why do we know that
such a node exists?

Is BFS correct?

- Does it visit all nodes reachable from the starting node?
- Can you prove it?
- We visited ‘z’ but not ‘w’, which is a contradiction, given the pseudocode

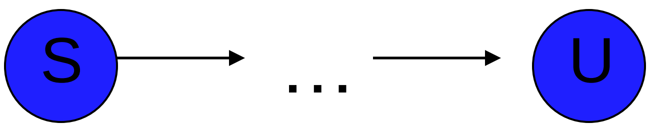


contradiction

Is BFS correct?

- Does it correctly label each node with the shortest distance from the starting node?

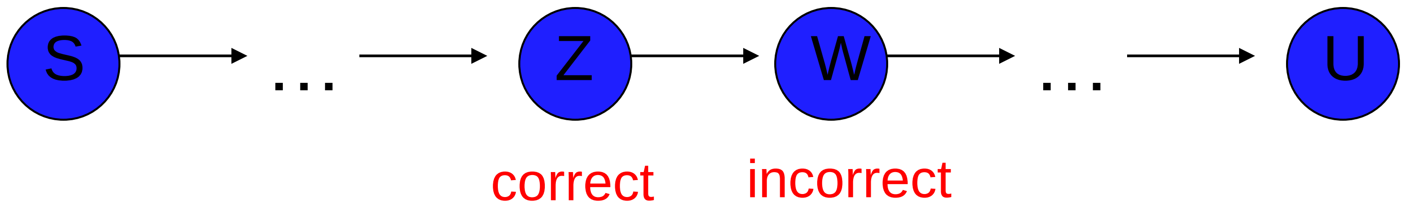
- Assume the algorithm labels a node with a longer distance. Call that node 'u'



Is BFS correct?

- Does it correctly label each node with the shortest distance from the starting node?

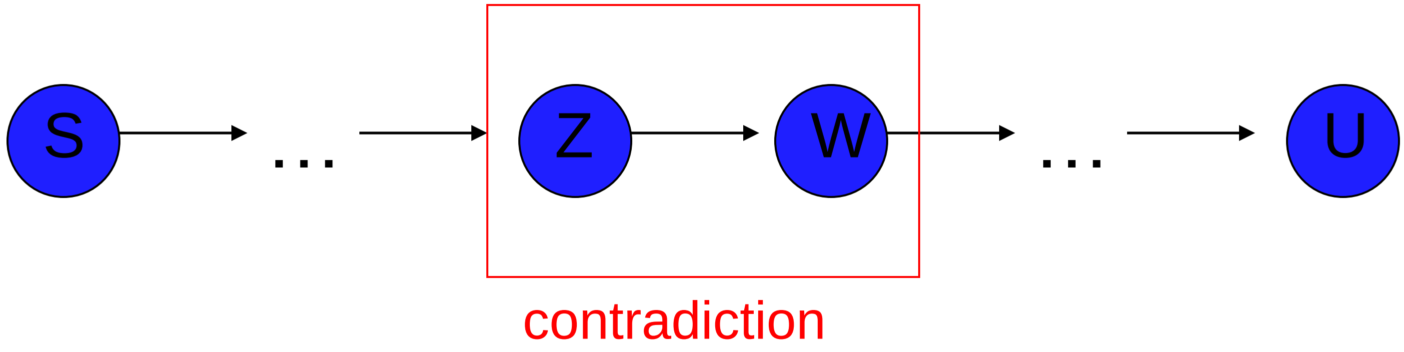
- Find the last node in the path with the correct distance



Is BFS correct?

- Does it correctly label each node with the shortest distance from the starting node?

- Find the last node in the path with the correct distance



Runtime of BFS

- Nothing changed over our analysis of TreeBFS

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

Runtime of BFS

- Adjacency list: O(|V| + |E|)
- Adjacency matrix: O($|V|^2$)

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Depth First Search (DFS)

```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```

Depth First Search (DFS)

```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```

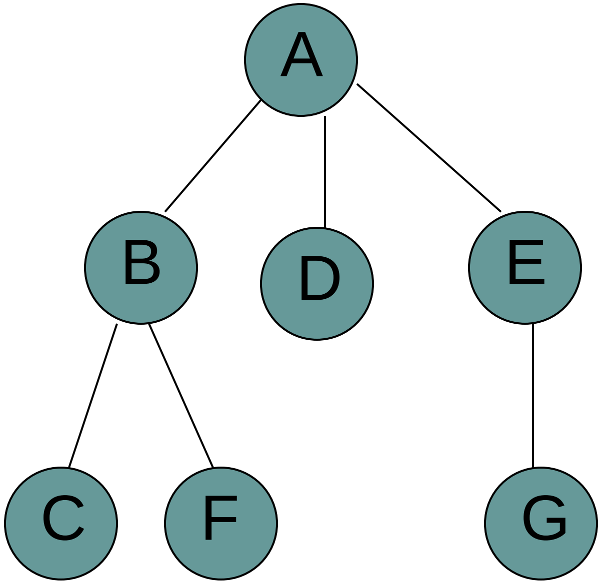
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

Depth First Search (DFS)

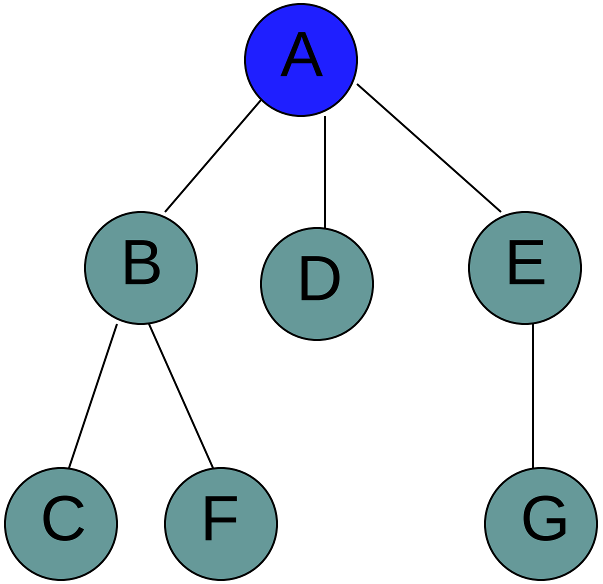
```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

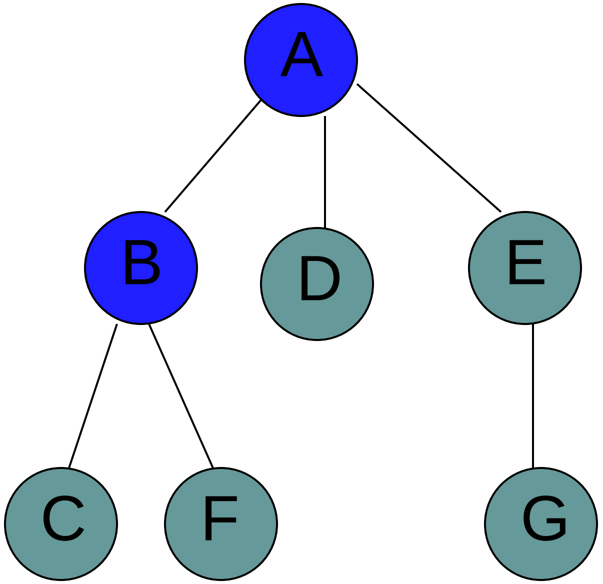
Tree DFS



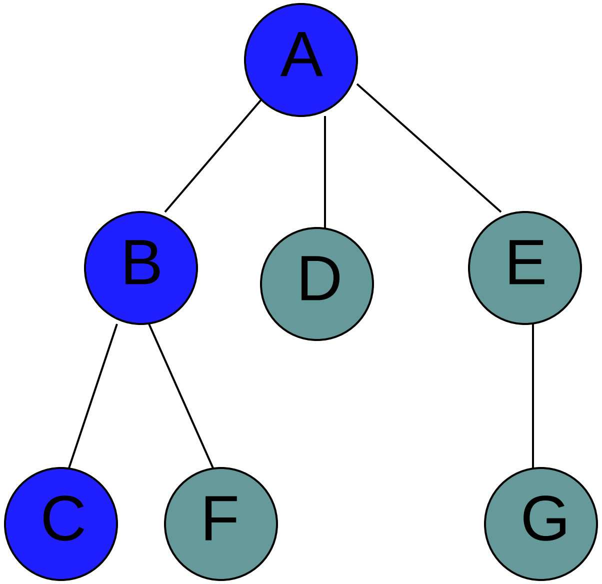
Tree DFS



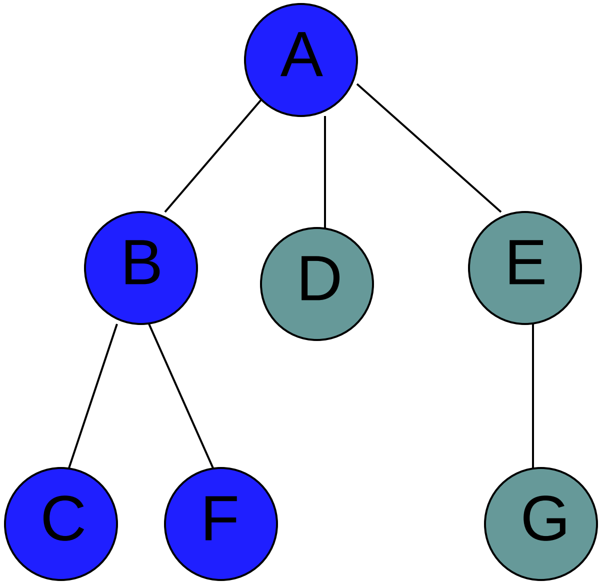
Tree DFS



Tree DFS

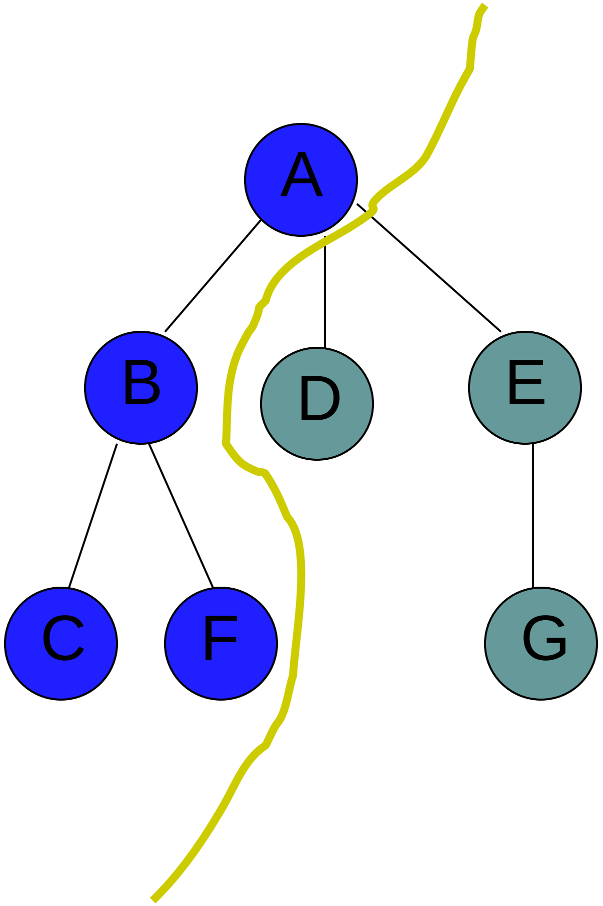


Tree DFS

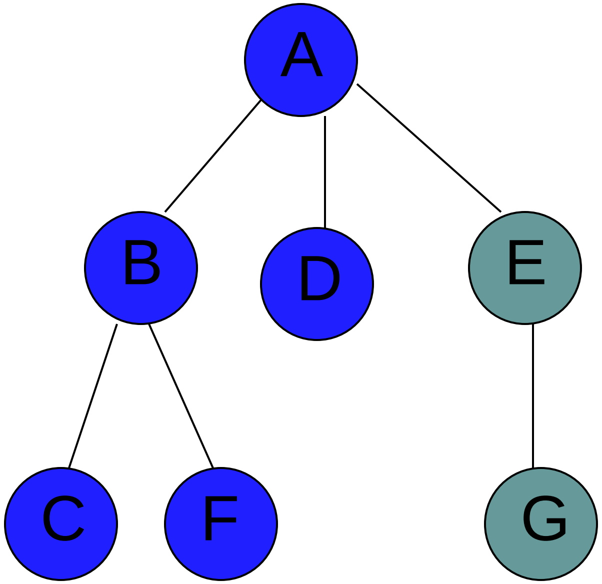


Frontier?

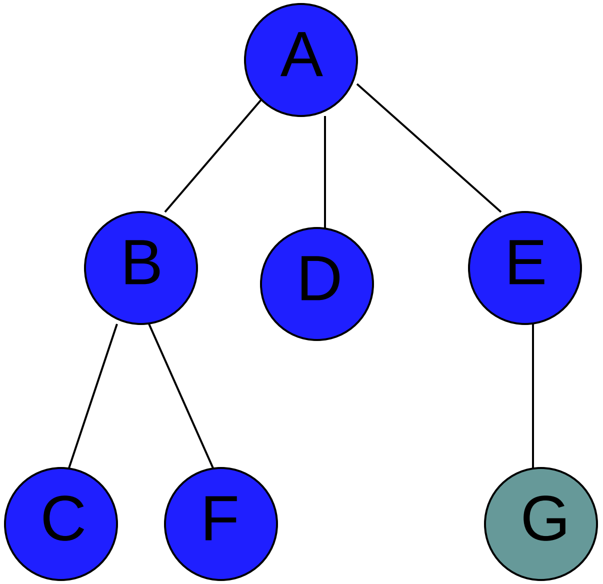
Tree DFS



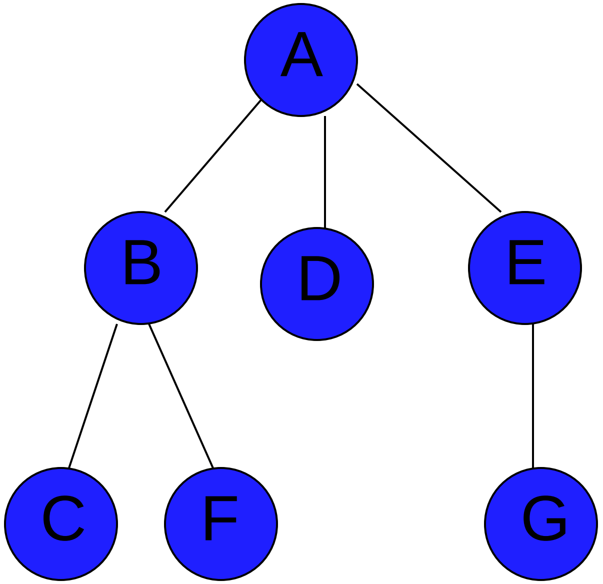
Tree DFS



Tree DFS



Tree DFS



Depth-first Search (DFS) on Graphs

- Explore edges out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its *predecessor*).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search on Graphs

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.

Pseudo-code

DFS(*G*)

```
1. for each vertex u ∈ V[G]
2.       do color[u] ← white
3.             π[u] ← NIL
4. time ← 0
5. for each vertex u ∈ V[G]
6.       do if color[u] = white
7.                then DFS-Visit(u)
```

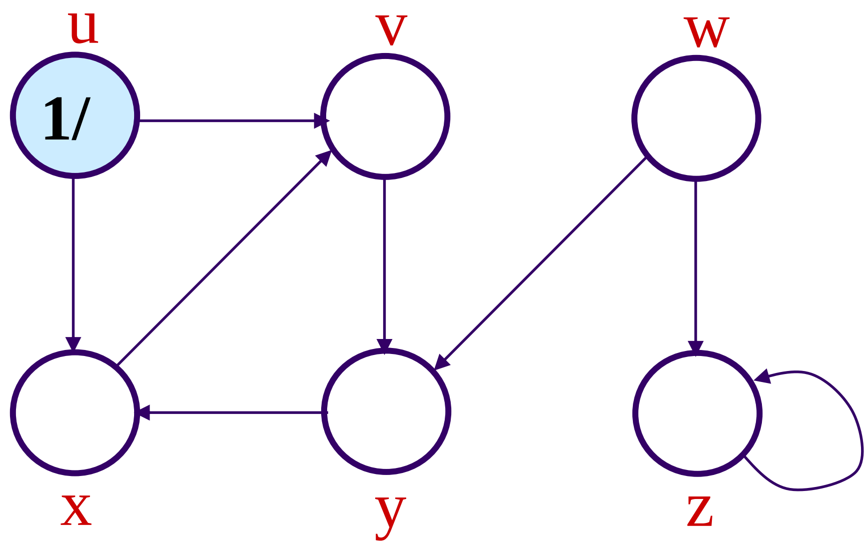
DFS-Visit(*u*)

```
1.    color[u] ← GRAY  ∇ White vertex u
      has been discovered
2.    time ← time + 1
3.    d[u] ← time
4.    for each v ∈ Adj[u]
5.         do if color[v] = WHITE
6.               then π[v] ← u
7.                    DFS-Visit(v)
8.    color[u] ← BLACK    ∇ Blacken u;
      it is finished.
9.    f[u] ← time ← time + 1
```

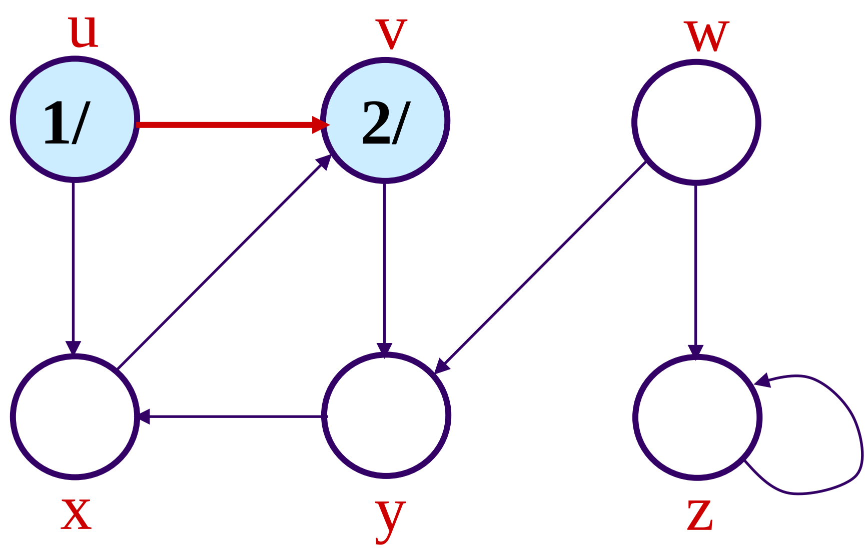
Uses a global timestamp ***time***.

Example: animation.

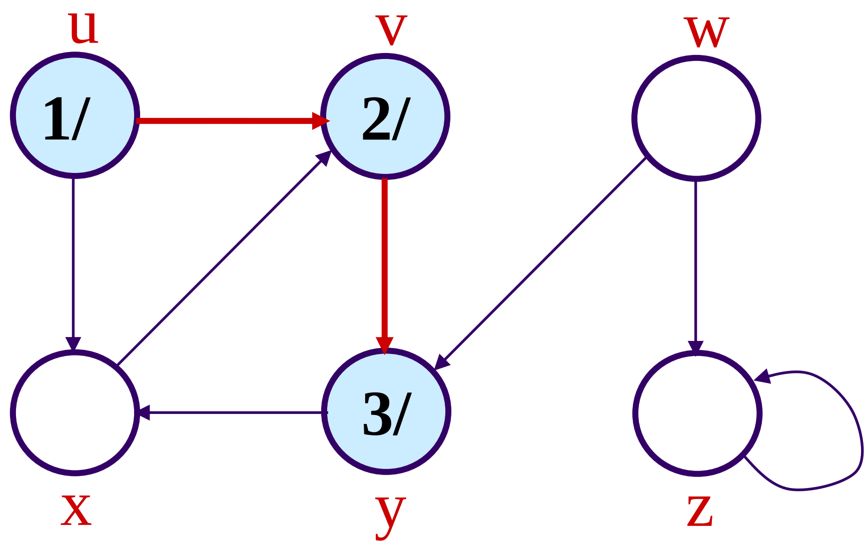
Example (DFS)



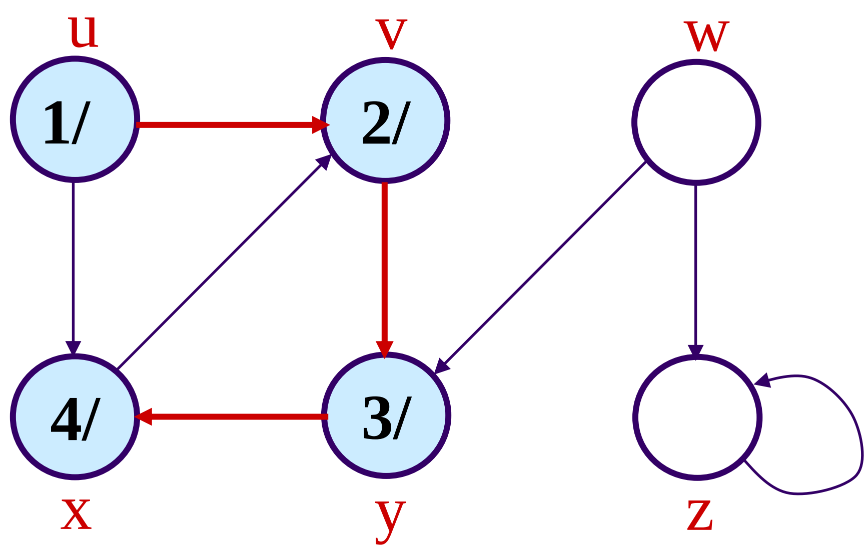
Example (DFS)



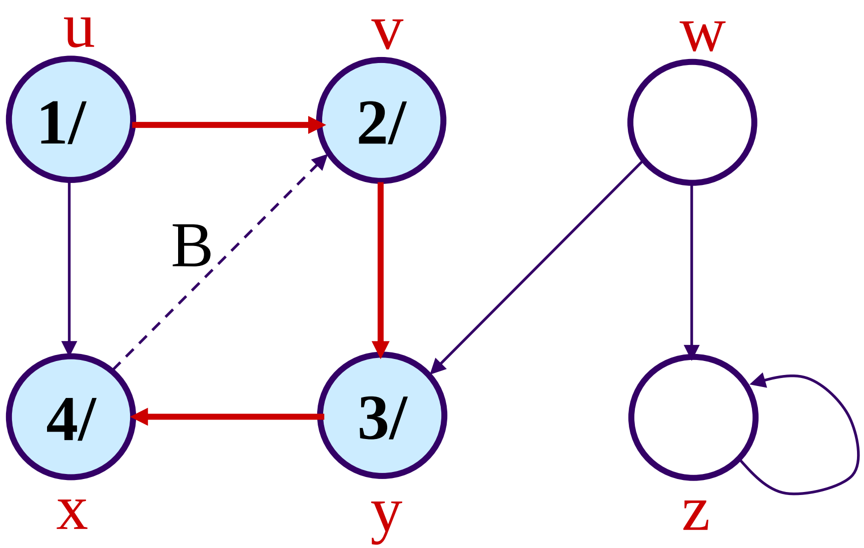
Example (DFS)



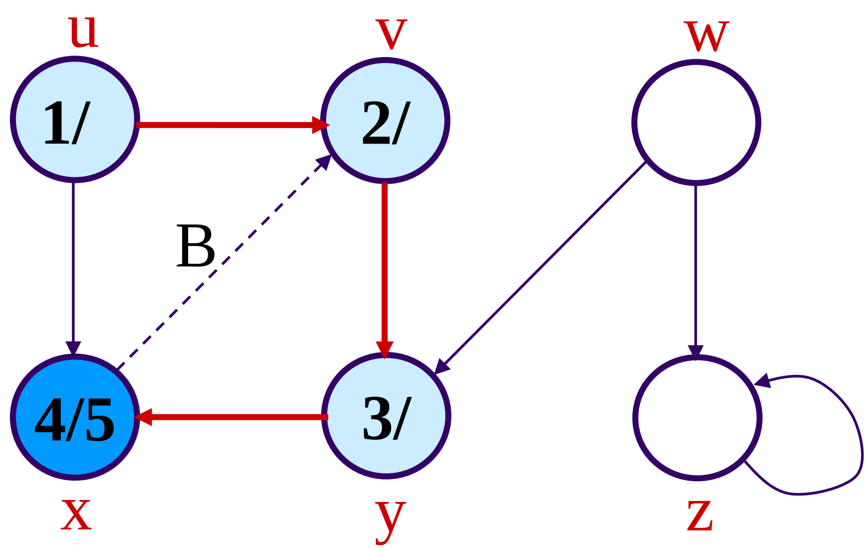
Example (DFS)



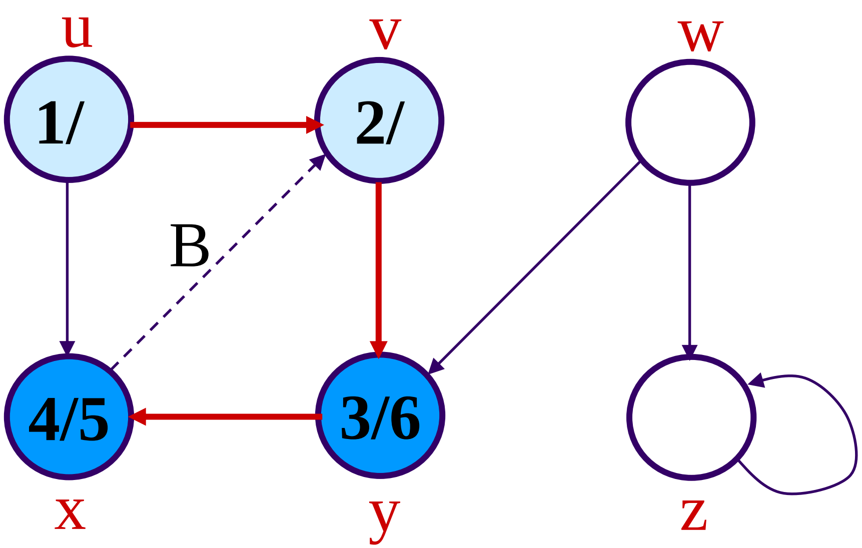
Example (DFS)



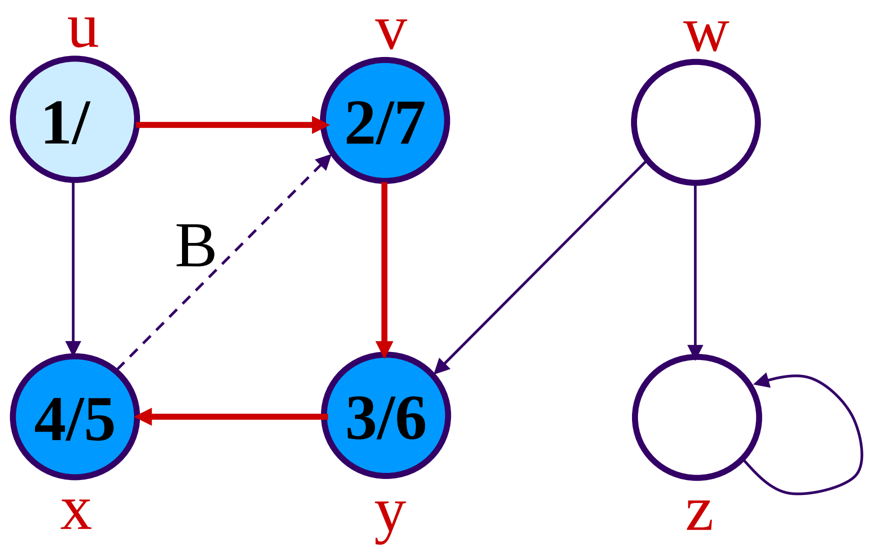
Example (DFS)



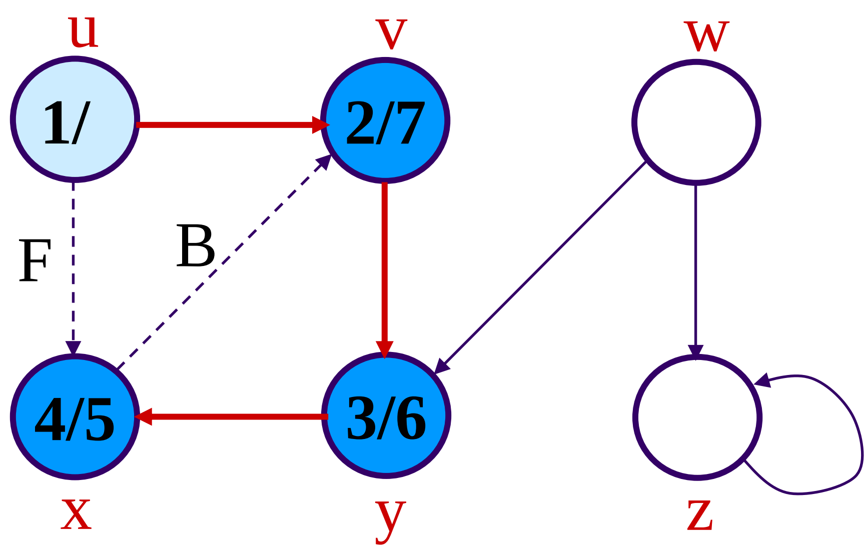
Example (DFS)



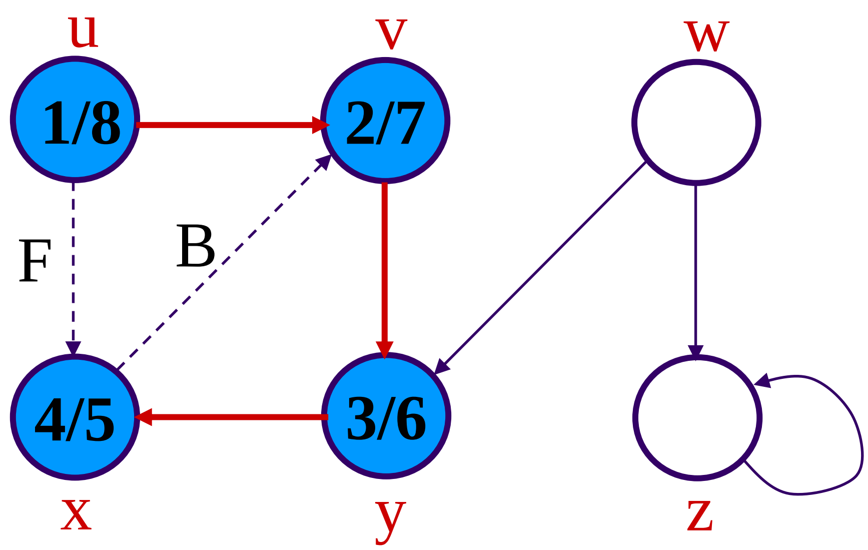
Example (DFS)



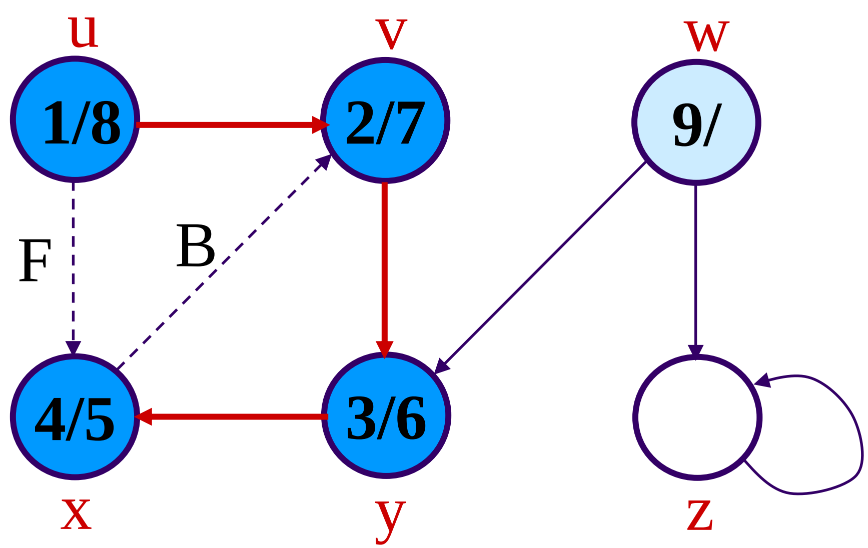
Example (DFS)



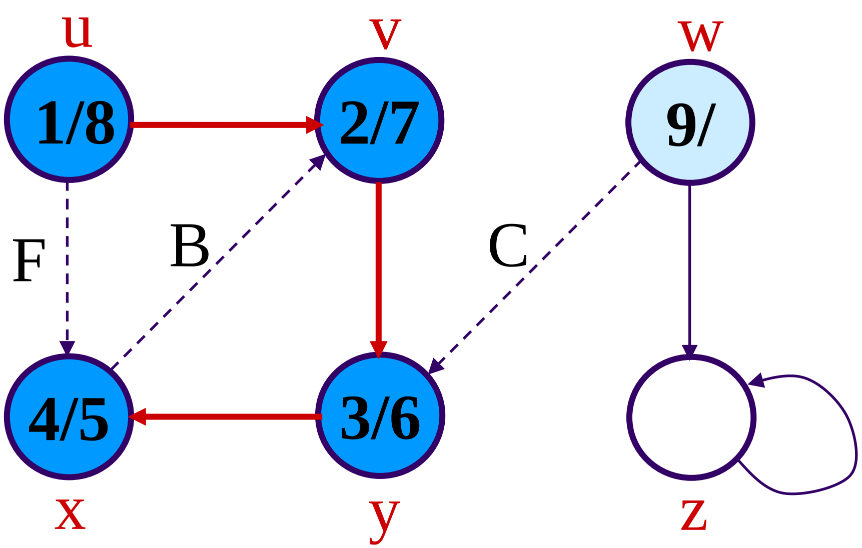
Example (DFS)



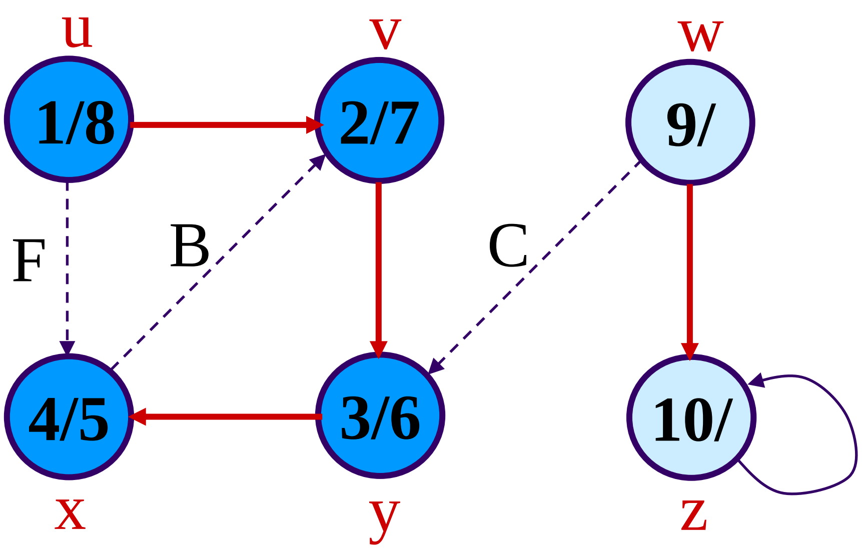
Example (DFS)



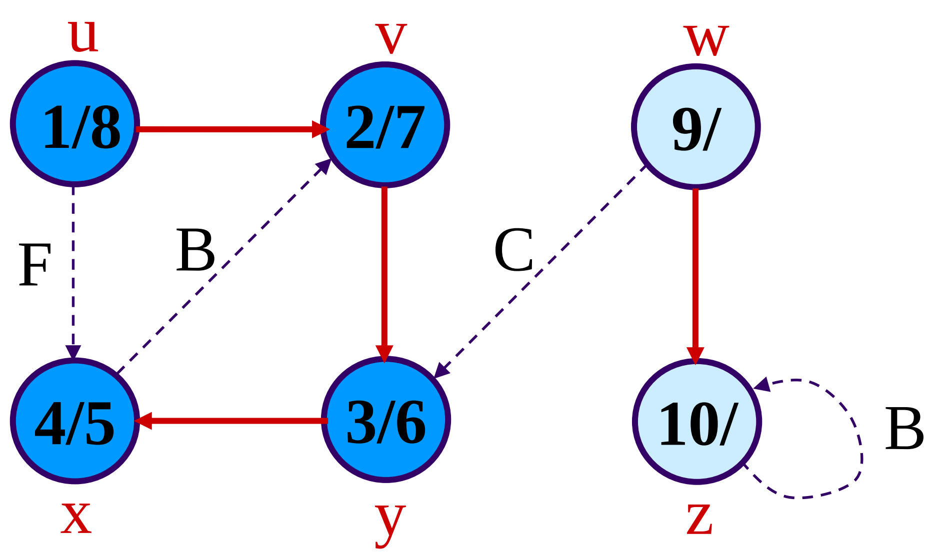
Example (DFS)



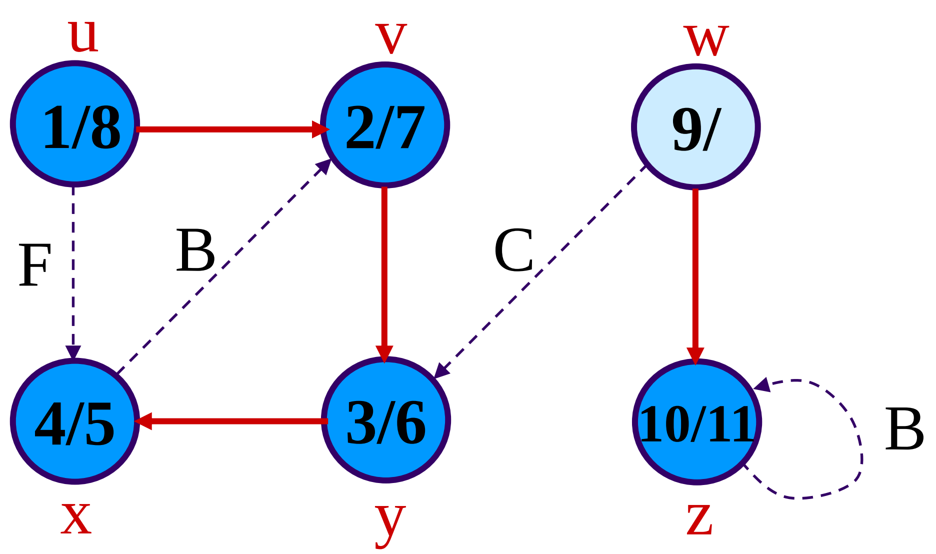
Example (DFS)



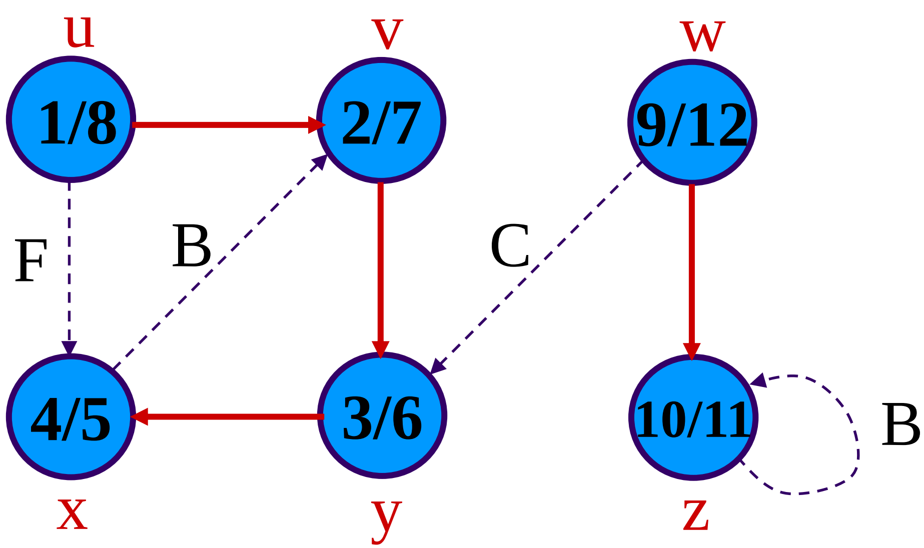
Example (DFS)



Example (DFS)



Example (DFS)



Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 3-6 of DFS-Visit is executed |Adj[*v*]| times. The total cost of executing DFS-Visit is $\sum_{v \in V} |Adj[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(V+E)$.

DFS-Visit(*u*)

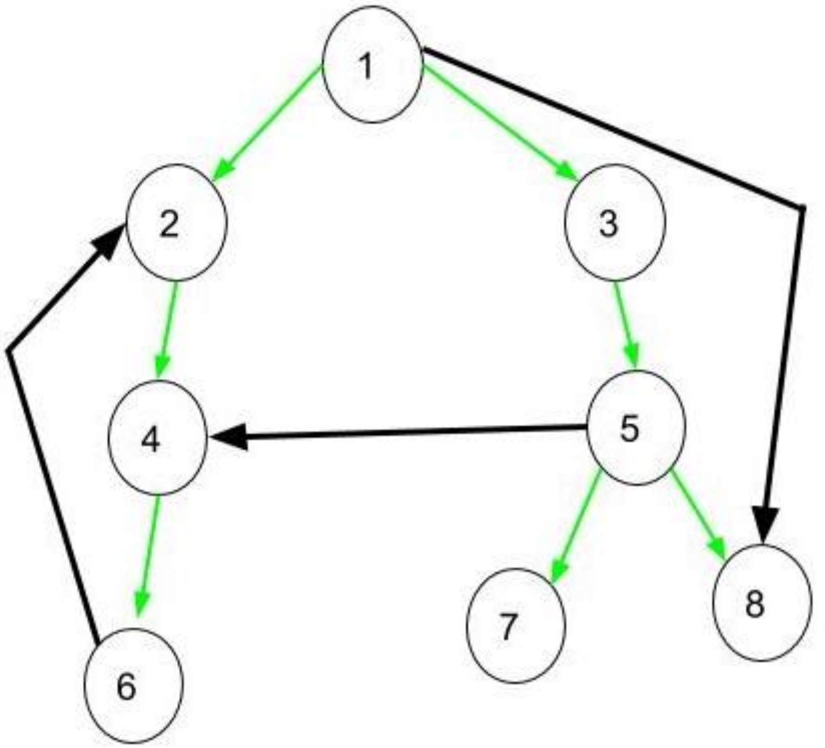
1. *color*[*u*] ← GRAY ∇ White vertex *u* has been discovered
2. *time* ← *time* + 1
3. *d*[*u*] ← *time*
4. **for** each *v* ∈ *Adj*[*u*]
5. **do if** *color*[*v*] = WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. *color*[*u*] ← BLACK ∇ Blacken *u*; it is finished.
9. *f*[*u*] ← *time* ← *time* + 1

Classification of Edges

- **Tree edge:** It is an edge which is present in the tree obtained after applying DFS on the graph.
- **Back edge:** It is an edge (u, v) such that v is the ancestor of node u but is not part of the DFS tree.
- **Forward edge:** It is an edge (u, v) such that v is a descendant but not part of the DFS tree.
- **Cross edge:** It is an edge that connects two nodes such that they do not have any ancestor and a descendant relationship between them.

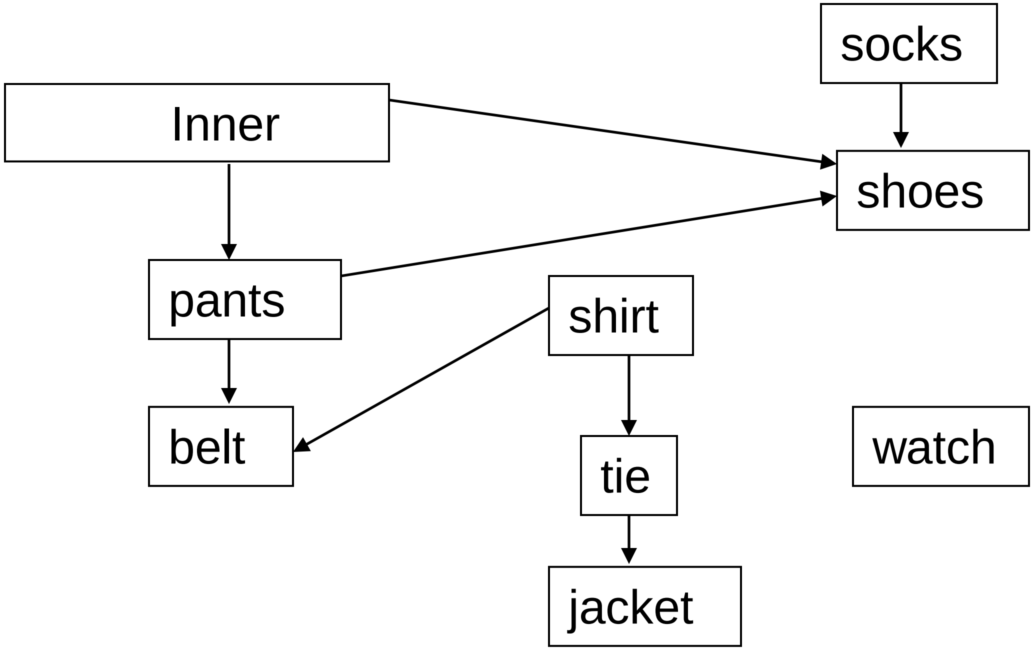
Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Classification of Edges



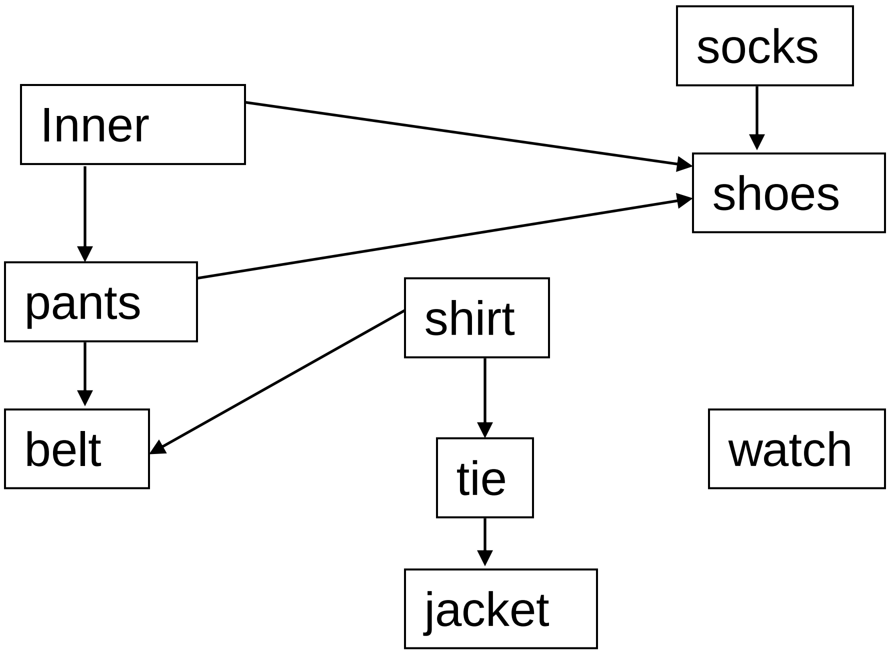
DAGs

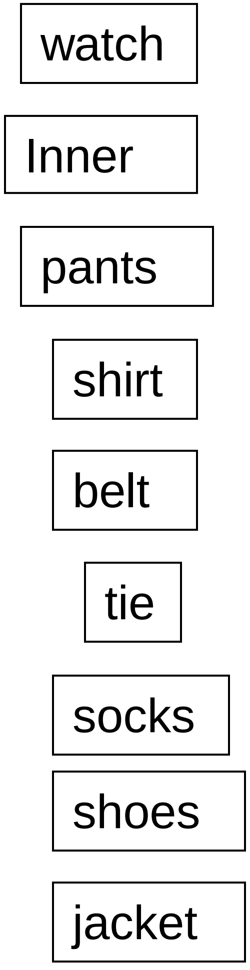
Can represent dependency graphs



Topological sort

- A linear ordering of all the vertices such that for all edges (u,v) $\in$ E, u appears before v in the ordering
- An ordering of the nodes that "obeys" the dependencies, i.e. an activity can't happen until it's dependent activities have happened





Topological sort

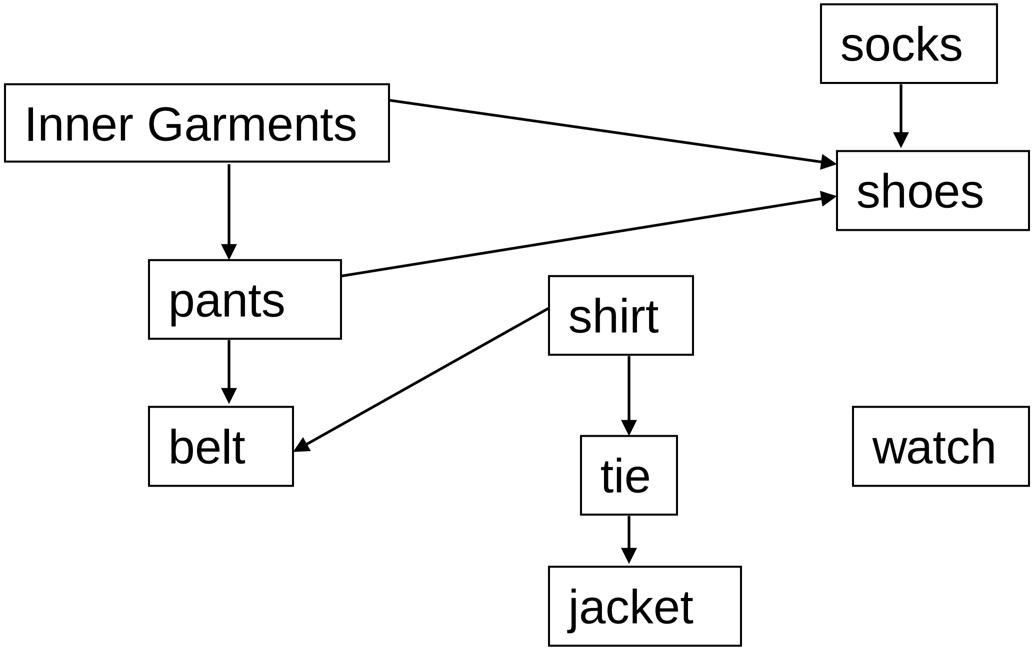
TOPOLOGICAL-SORT1(G)

1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)

Topological sort

TOPOLOGICAL-SORT1(G)

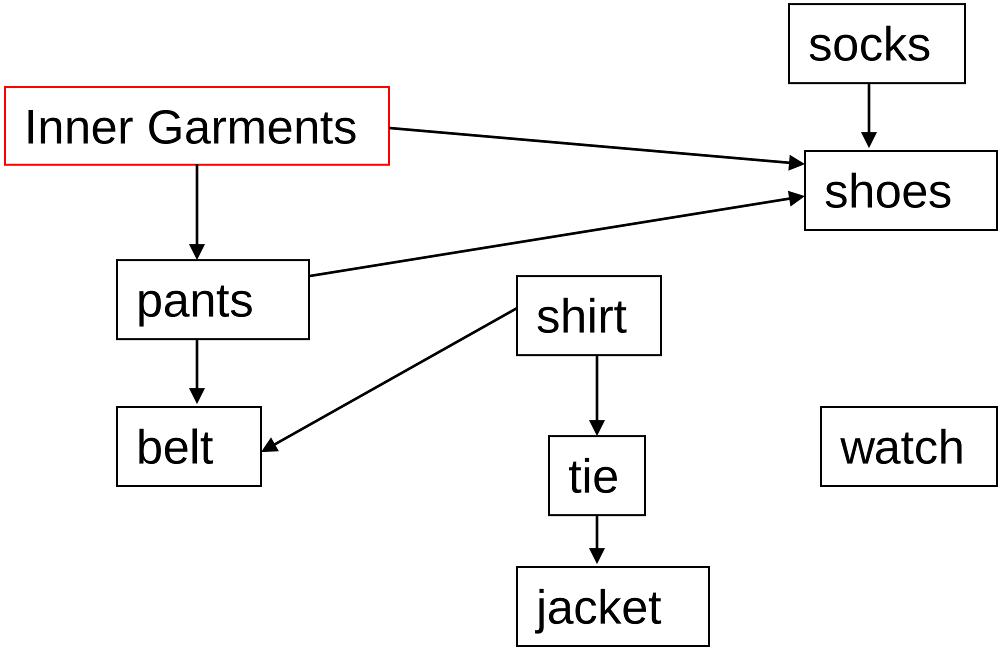
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)



Topological sort

TOPOLOGICAL-SORT1(G)

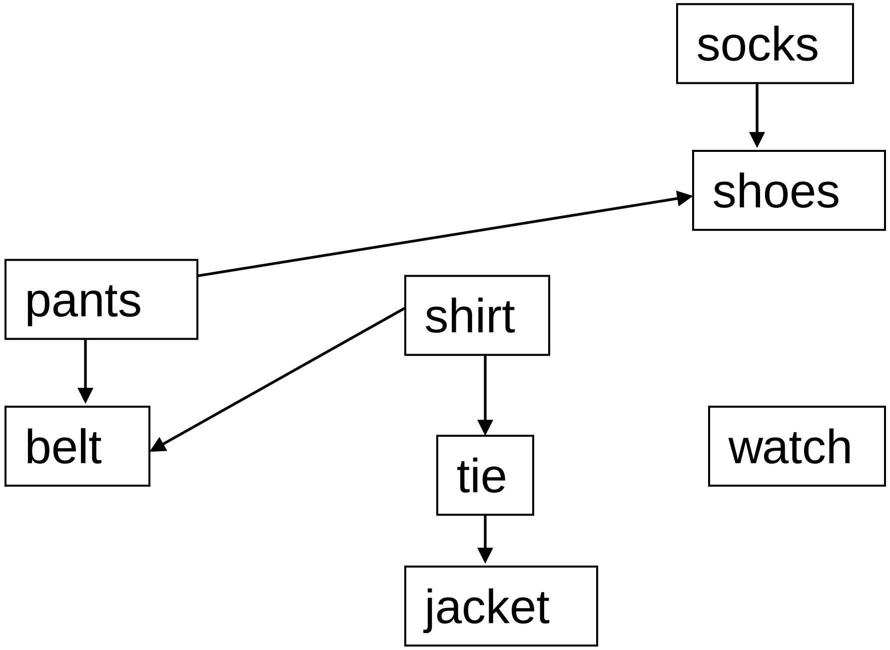
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)



Topological sort

Inner Garments

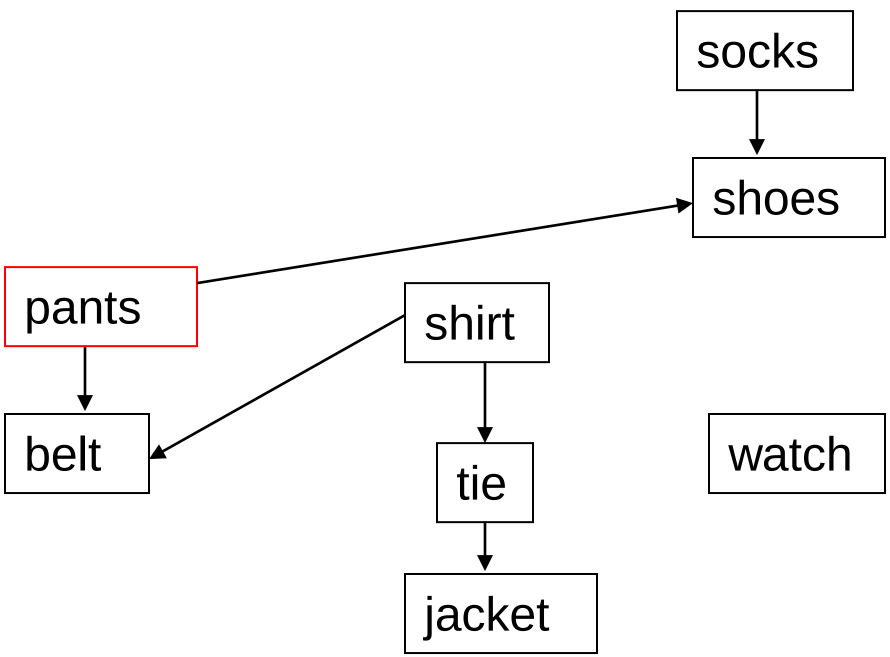
```
TOPOLOGICAL-SORT1(G)
Find a node v with no incoming edges
Delete v from G
Add v to linked list
TOPOLOGICAL-SORT1(G)
```



Topological sort

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```

Inner Garments

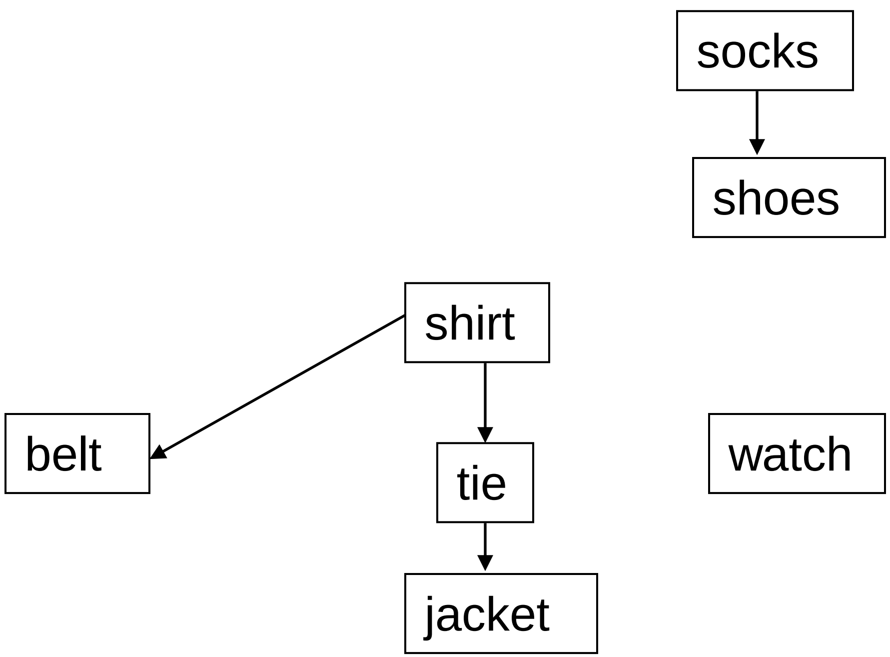


Topological sort

```
TOPOLOGICAL-SORT1(G)
Find a node v with no incoming edges
Delete v from G
Add v to linked list
TOPOLOGICAL-SORT1(G)
```

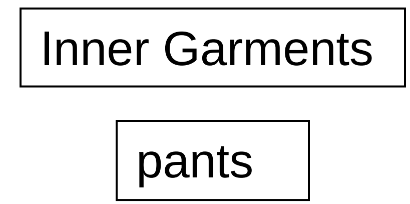
Inner Garments

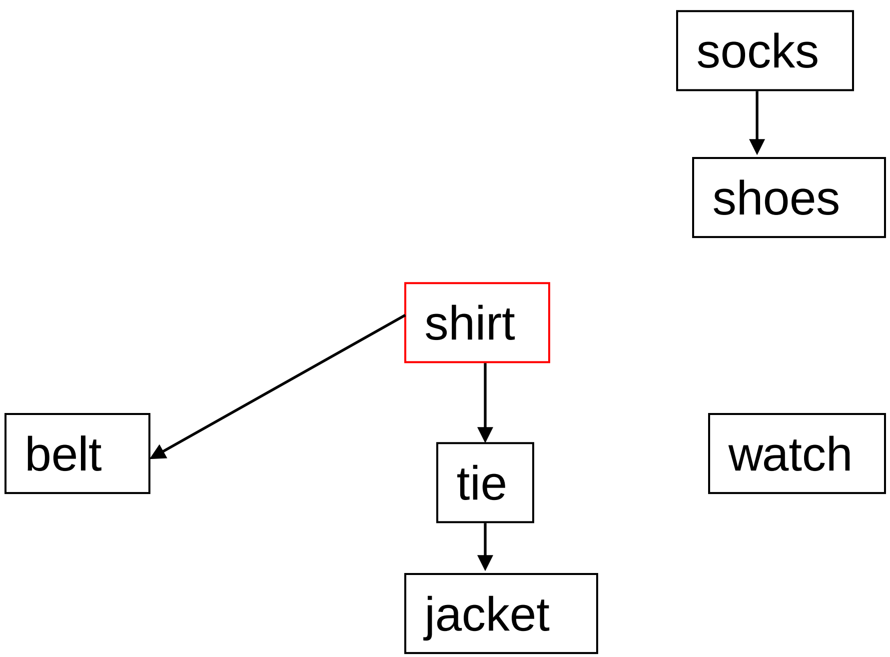
pants



Topological sort

```
TOPOLOGICAL-SORT1(G)
Find a node v with no incoming edges
Delete v from G
Add v to linked list
TOPOLOGICAL-SORT1(G)
```

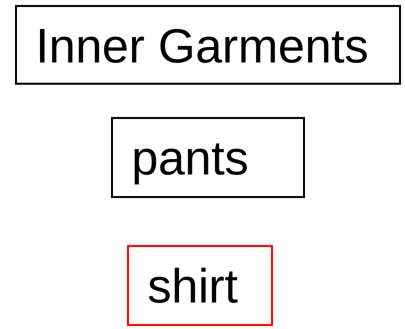


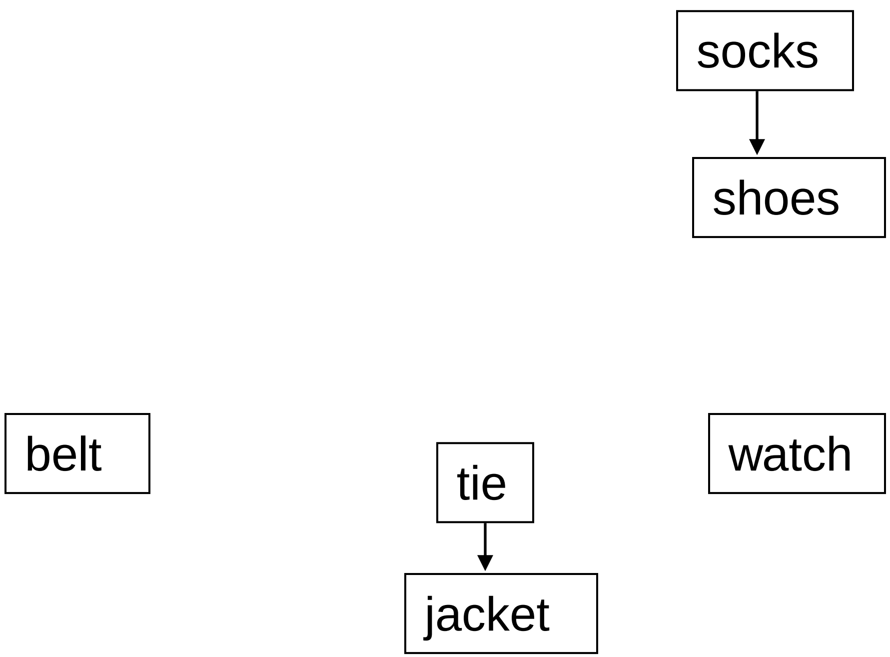


Topological sort

TOPOLOGICAL-SORT1(G)

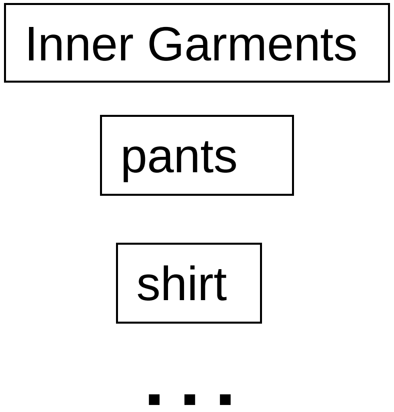
```
Find a node v with no incoming edges
Delete v from G
Add v to linked list
TOPOLOGICAL-SORT1(G)
```

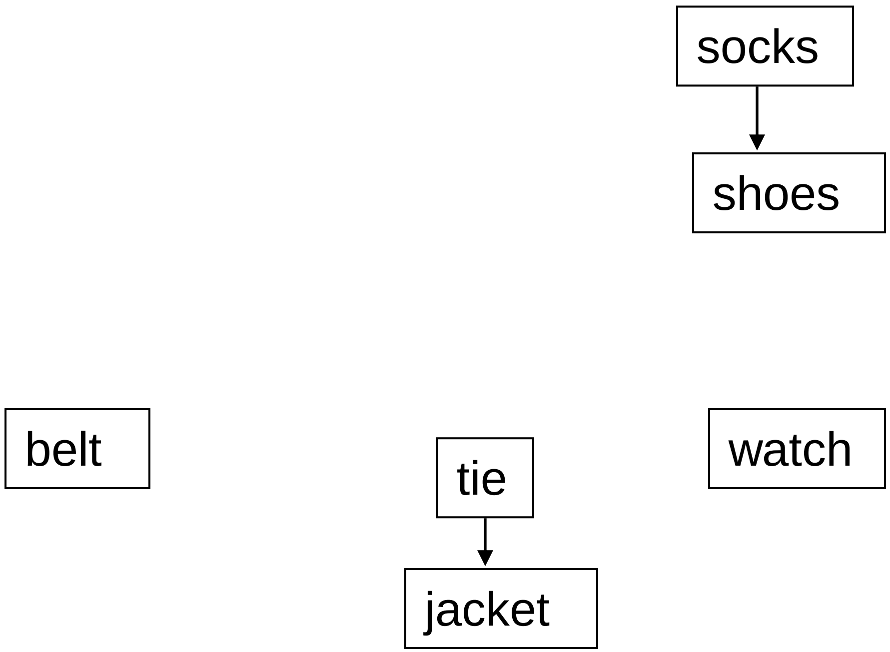




Topological sort

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```





Running time?

TOPOLOGICAL-SORT1(G)

1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)

Running time?

Topological-Sort1(G)

1 Find a node v with no incoming edges

2 Delete v from G

3 Add v to linked list

4 Topological-Sort1(G)

O(|V|+|E|)

Running time?

TOPOLOGICAL-SORT1(G)

1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)

O(E) overall

Running time?

TOPOLOGICAL-SORT1(G)

1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)

How many calls? |V|

Running time?

Topological-Sort1(G)

1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 $\boxed{\text{Topological-Sort1}(G)}$

Overall running time?

$O(|V|^2+|V|\ |E|)$

Can we do better?

TOPOLOGICAL-SORT1(G)

1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)

Topological sort 2

TOPOLOGICAL-SORT2(G)

for all edges $(u, v) \in E$
    $active[v] \leftarrow active[v] + 1$
for all $v \in V$
    if $active[v] = 0$
        ENQUEUE(S, v)
while !EMPTY(S)
    $u \leftarrow$ DEQUEUE(S)
    add u to linked list
    for each edge $(u, v) \in E$
        $active[v] \leftarrow active[v] - 1$
        if $active[v] = 0$
            ENQUEUE(S, v)

Topological sort 2

```
TOPOLOGICAL-SORT2(G)
 for all edges (u, v) ∈ E
         active[v] ← active[v] + 1
 for all v ∈ V
         if active[v] = 0
                 ENQUEUE(S, v)
 while !EMPTY(S)
         u ← DEQUEUE(S)
         add u to linked list
         for each edge (u, v) ∈ E
                 active[v] ← active[v] − 1
                 if active[v] = 0
                         ENQUEUE(S, v)
```

Topological sort 2

```
TOPOLOGICAL-SORT2(G)
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                ENQUEUE(S, v)
while !EMPTY(S)
        u ← DEQUEUE(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        ENQUEUE(S, v)
```

Topological sort 2

TOPOLOGICAL-SORT2(G)

```
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                ENQUEUE(S, v)
while !EMPTY(S)
        u ← DEQUEUE(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        ENQUEUE(S, v)
```

Running time?

- How many times do we process each node?
- How many times do we process each edge?
- O(|V| + |E|)

```
Topological-Sort2(G)
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                Enqueue(S, v)
while !Empty(S)
        u ← Dequeue(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        Enqueue(S, v)
```

Topological sort 3

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finishing times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

Running time?

O(|V| + |E|)

```
TOPOLOGICAL-SORT(G)
    call DFS(G) to compute finishing times v.f for each vertex v
    as each vertex is finished, insert it onto the front of a linked list
    return the linked list of vertices
```

Connectedness

Given an undirected graph, for every node u $\in$ V, can we reach all other nodes in the graph?

Run BFS or DFS-Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false.

Running time: O(|V| + |E|)

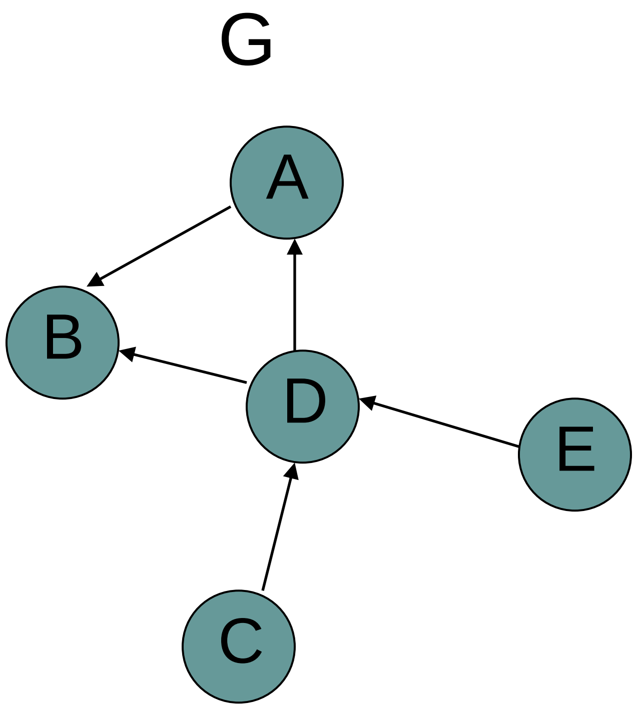
Strongly connected

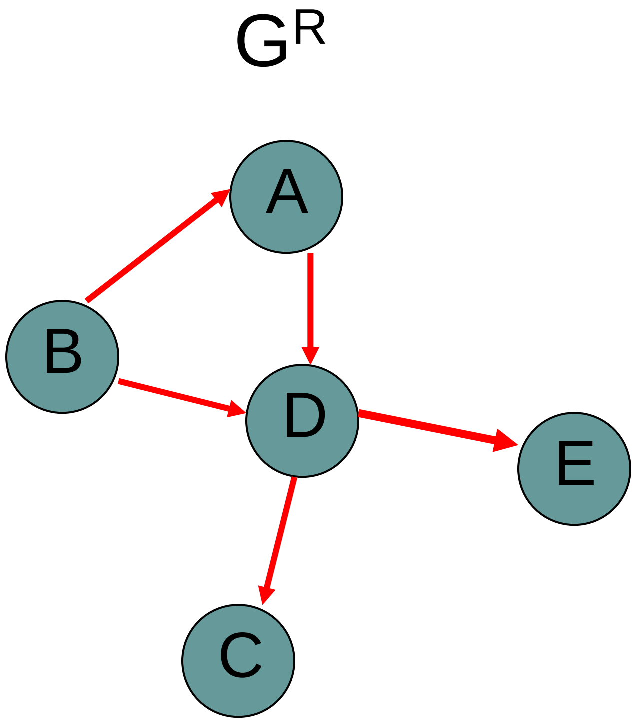
Given a directed graph, can we reach any node v from any other node u?

Ideas?

Transpose of a graph

- Given a graph G, we can calculate the transpose of a graph G^R by reversing the direction of all the edges





Running time to calculate G^R? O(|V| + |E|)

Strongly connected

Strongly-Connected(G)

Run DFS or BFS from some node u
if not all nodes are visited
    return false
Create graph G^R by reversing all edge dirctions
Run DFS or BFS on G^R from node u
if not all nodes are visited
    return false
return true

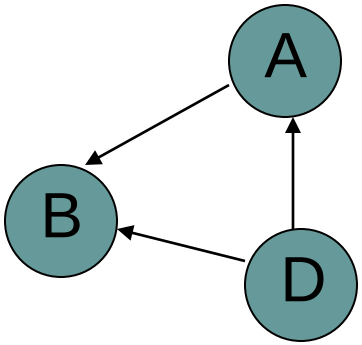
Runtime?

STRONGLY-CONNECTED(G)	
Run DFS or BFS from some node u	O(|V| + |E|)
if not all nodes are visited	O(|V|)
return false	
Create graph G^R by reversing all edge dirctions	O(|V| + |E|)
Run DFS or BFS on G^R from node u	O(|V| + |E|)
if not all nodes are visited	O(|V|)
return false	
return true	

O(|V| + |E|)

Detecting cycles

- Undirected graph
  - BFS or DFS. If we reach a node we've seen already, then we've found a cycle
- Directed graph



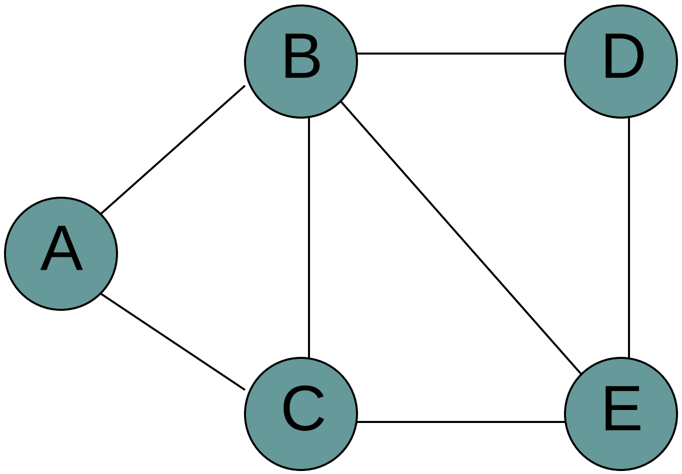
have to be careful

Detecting cycles

- Undirected graph
  - BFS or DFS. If we reach a node we've seen already, then we've found a cycle
- Directed graph
  - Apply DFS
  - Identify back edge, if back edge exists then there is cycle

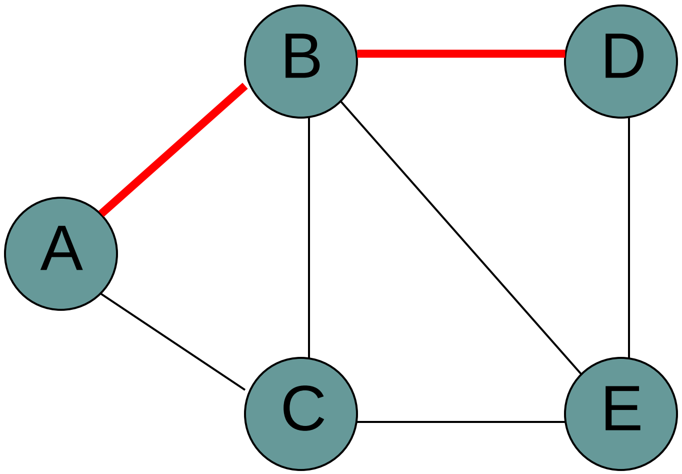
Shortest paths

- What is the shortest path from a to d?



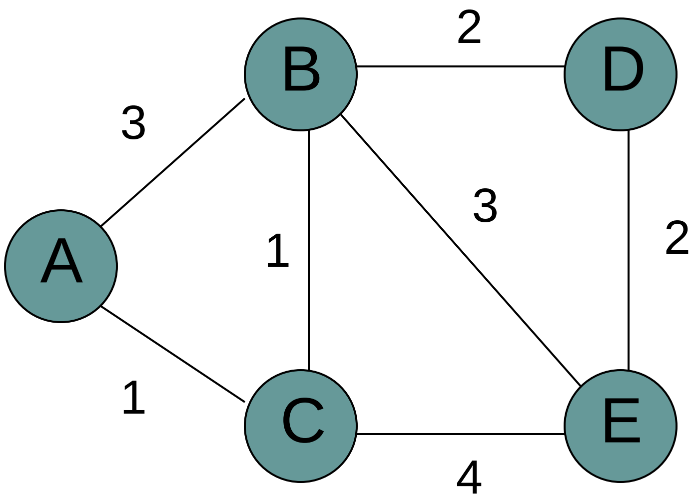
Shortest paths

- BFS



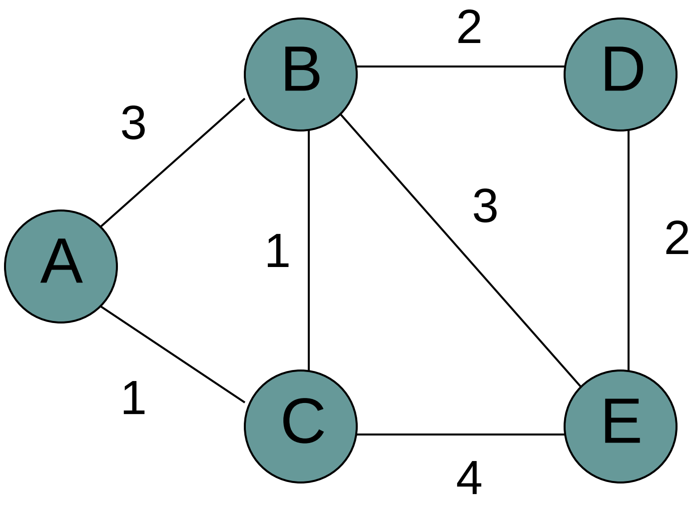
Shortest paths

- What is the shortest path from a to d?



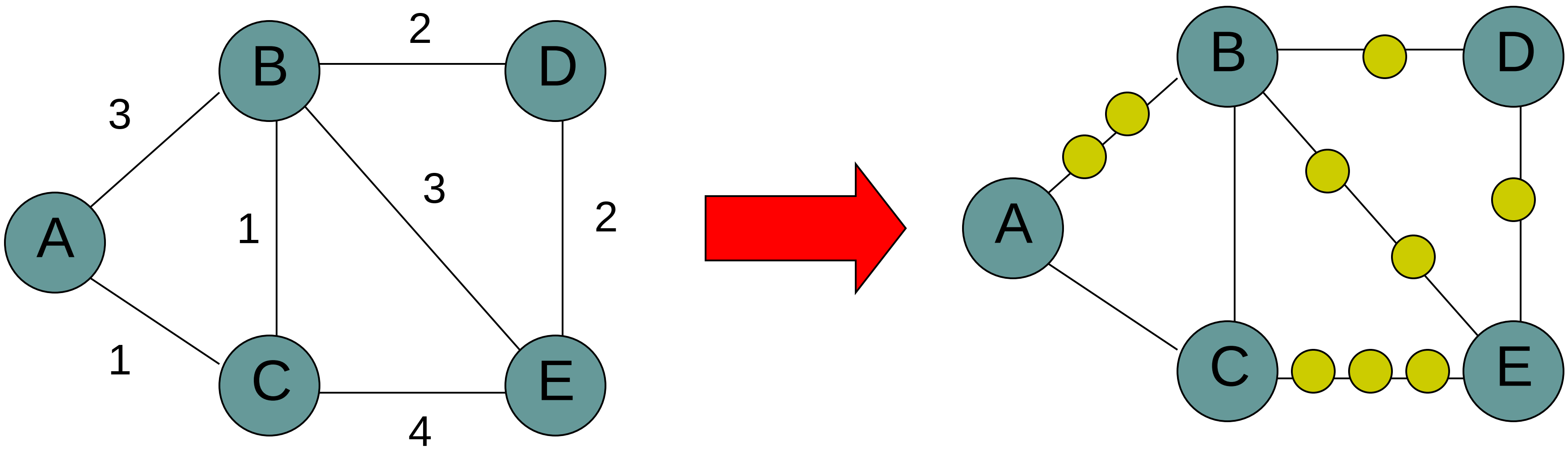
Shortest paths

- We can still use BFS



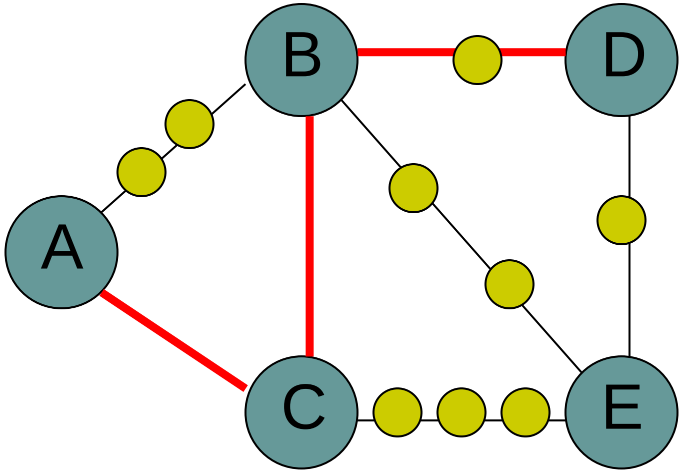
Shortest paths

- We can still use BFS



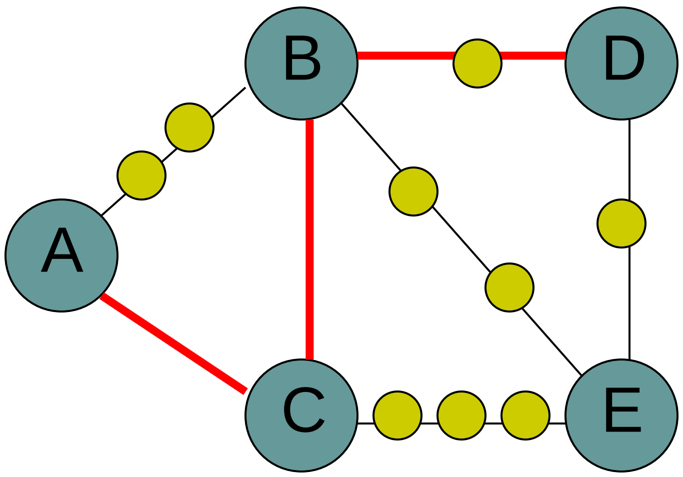
Shortest paths

- We can still use BFS



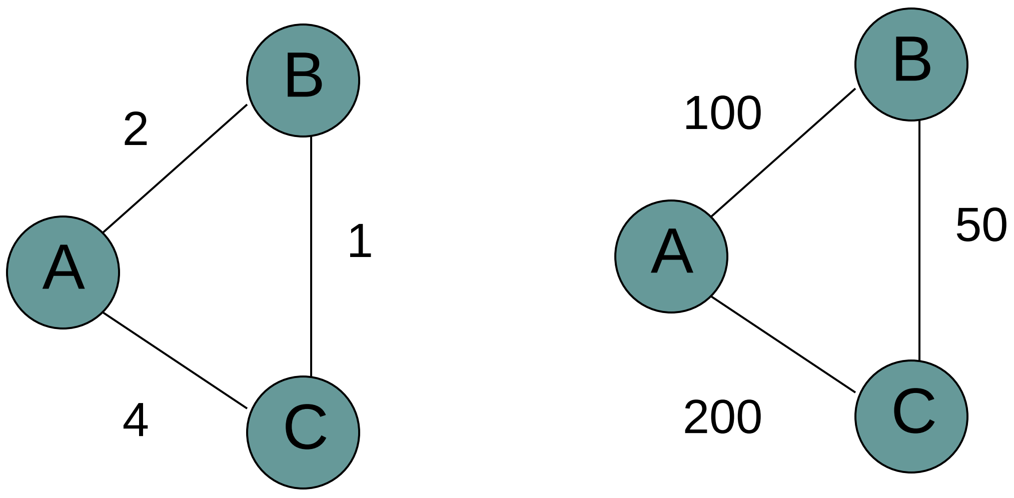
Shortest paths

- What is the problem?

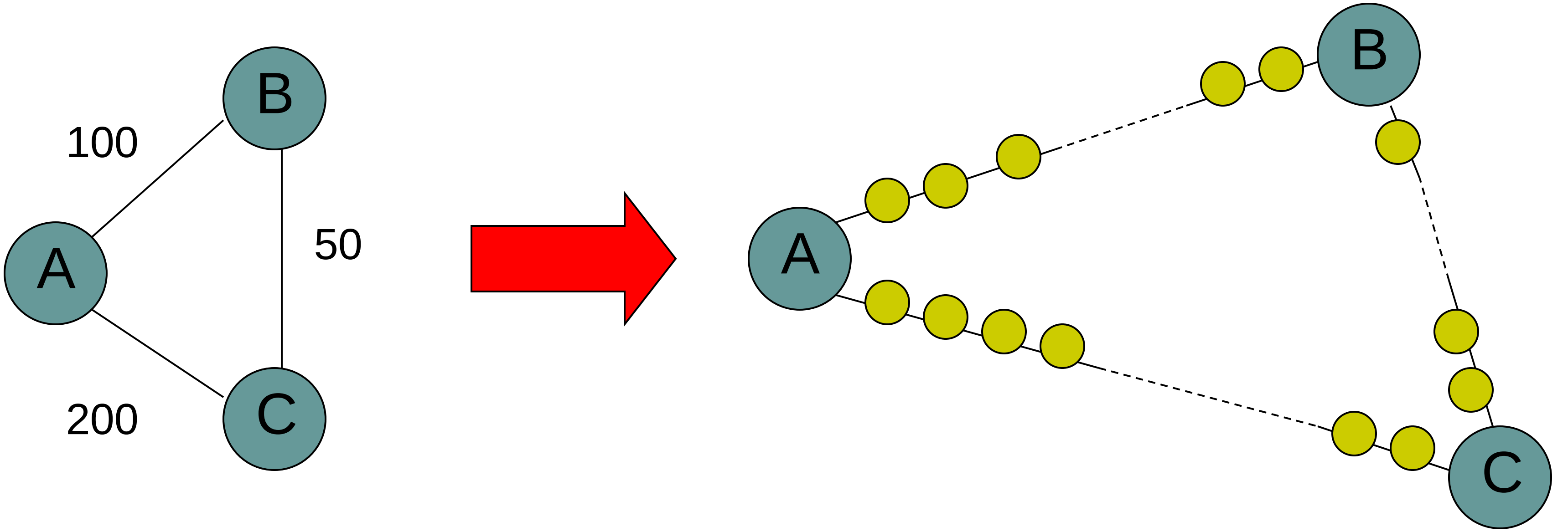


Shortest paths

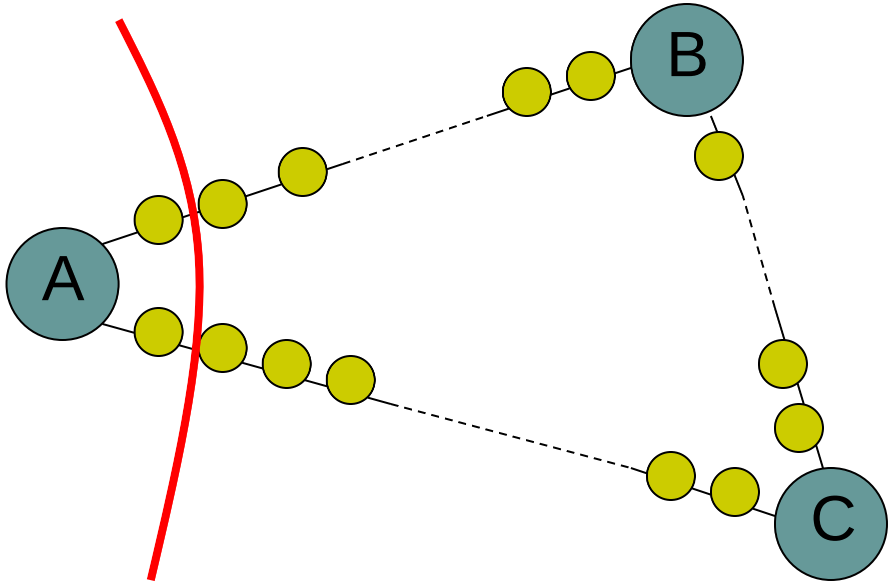
- Running time is dependent on the weights



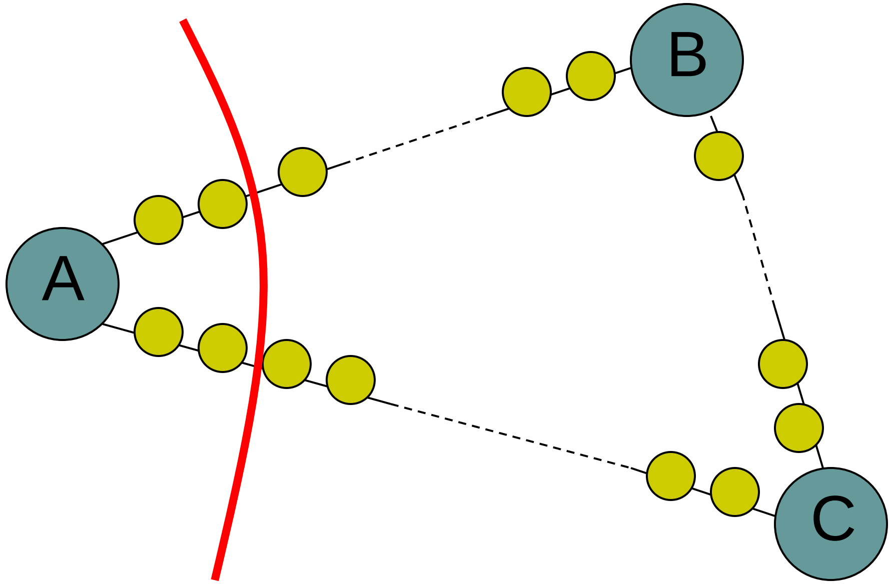
Shortest paths



Shortest paths

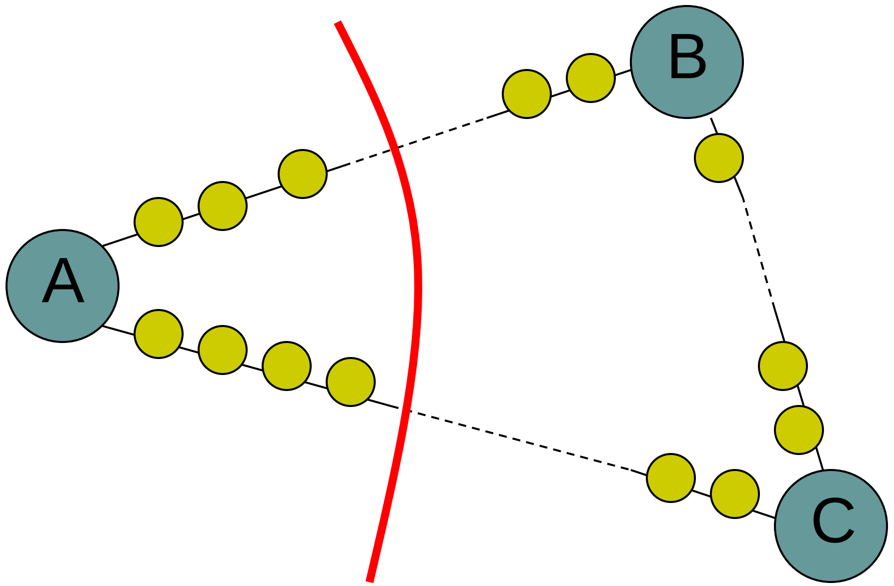


Shortest paths



Shortest paths

Nothing will change as we expand the frontier until we've gone out 100 levels



Dijkstra’s algorithm

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Dijkstra’s algorithm

$\textsc{Dijkstra}(G, s)$

$\textbf{for}$ all $v \in V$
  $dist[v] \leftarrow \infty$
  $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
$Q \leftarrow \textsc{MakeHeap}(V)$
$\textbf{while}$ $!\textsc{Empty}(Q)$
  $u \leftarrow \textsc{ExtractMin}(Q)$
  $\textbf{for}$ all edges $(u, v) \in E$
    $\textbf{if}$ $dist[v] > dist[u] + w(u, v)$
      $dist[v] \leftarrow dist[u] + w(u, v)$
      $\textsc{DecreaseKey}(Q, v, dist[v])$
      $prev[v] \leftarrow u$

$\text{BFS}(G, s)$

$\textbf{for}$ each $v \in V$
  $dist[v] = \infty$
$dist[s] = 0$
$\textsc{Enqueue}(Q, s)$
$\textbf{while}$ $!\textsc{Empty}(Q)$
  $u \leftarrow \textsc{Dequeue}(Q)$
  $\textsc{Visit}(\textsc{u})$
  $\textbf{for}$ each edge $(u, v) \in E$
    $\textbf{if}$ $dist[v] = \infty$
      $\textsc{Enqueue}(Q, v)$
      $dist[v] \leftarrow dist[u] + 1$

Dijkstra’s algorithm

prev keeps track of
the shortest path

Dijkstra(G, s)

```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

BFS(G, s)

```
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra’s algorithm

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

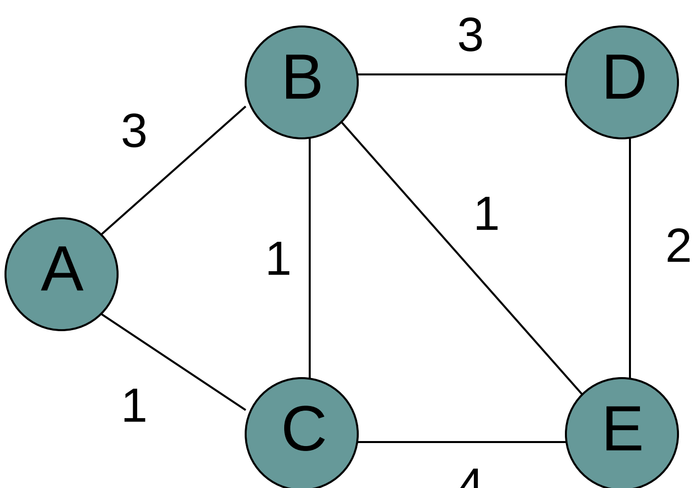
Dijkstra’s algorithm

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

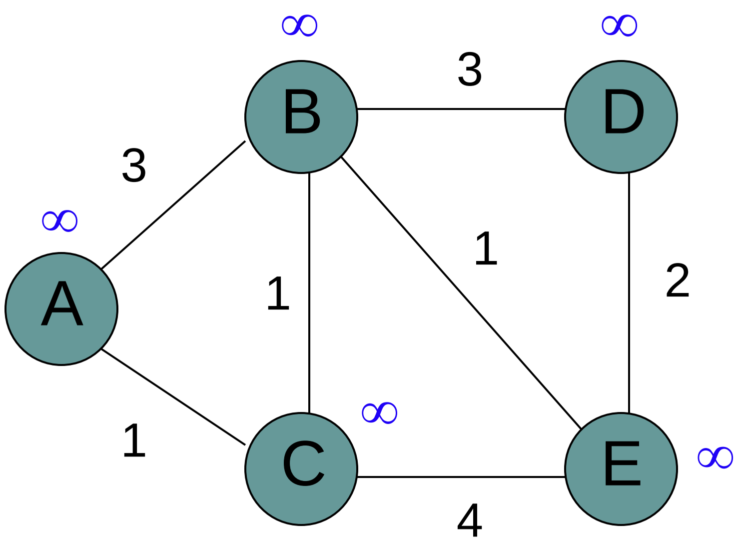
```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra(G, s)

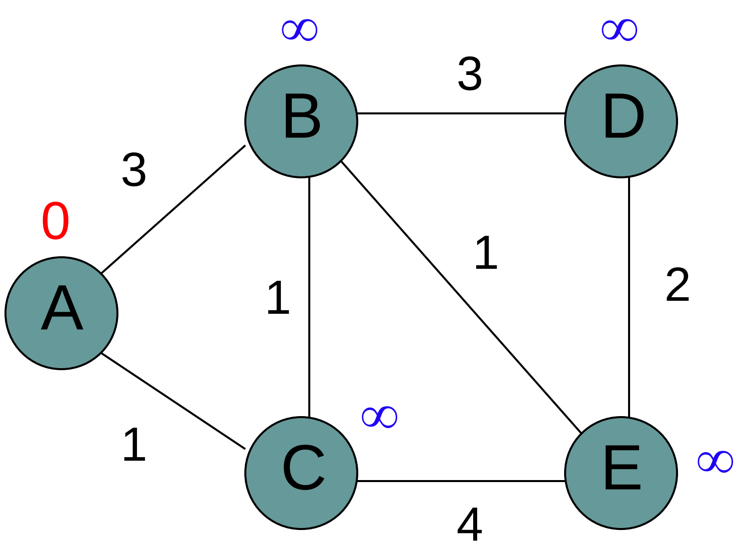
```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



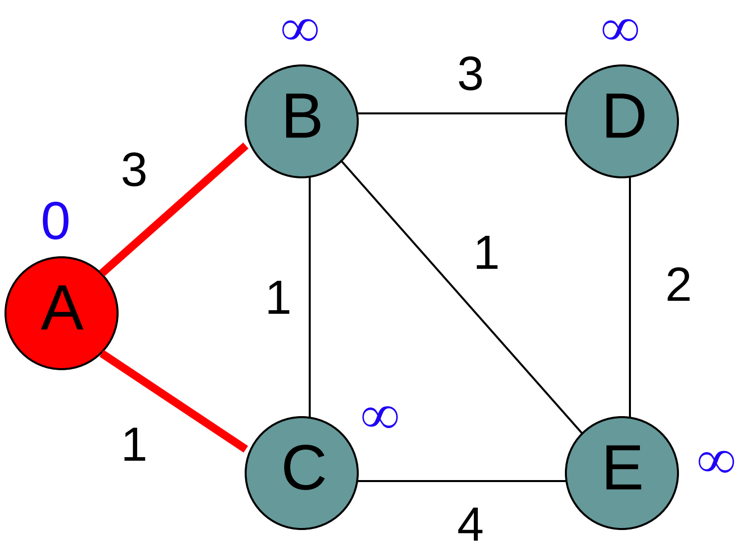
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

A	0
B	∞
C	∞
D	∞
E	∞

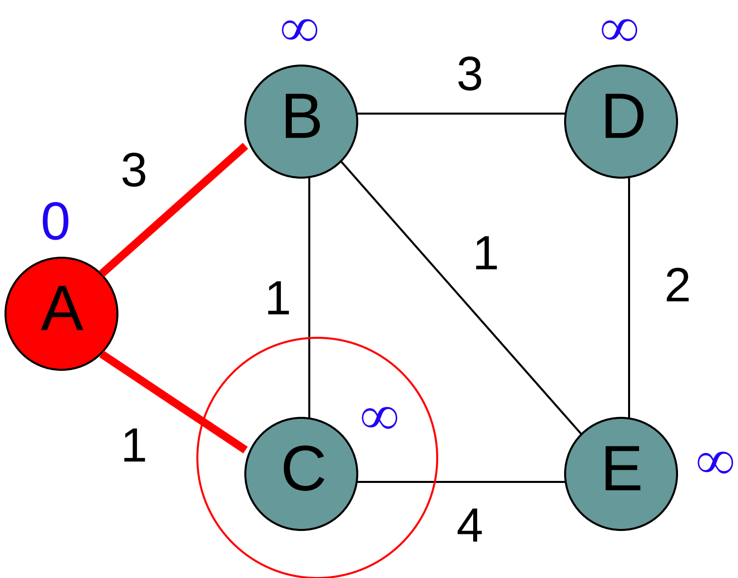
```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B	∞
C	∞
D	∞
E	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B ∞

C ∞

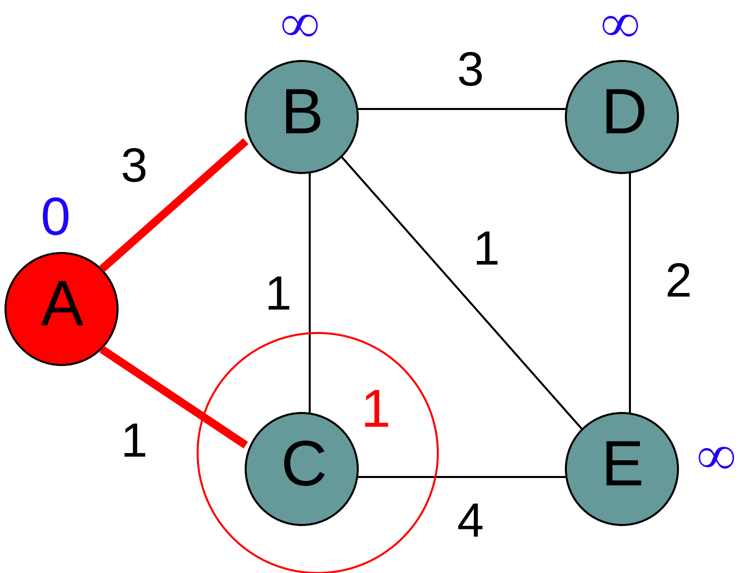
D ∞

E ∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

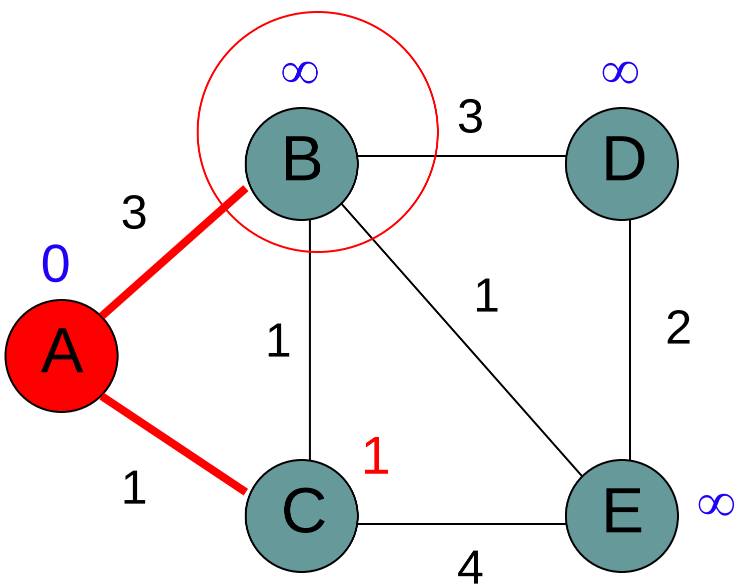
Heap

C 1
B ∞
D ∞
E ∞



DIJKSTRA(G, s)

for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
$Q \leftarrow$ MAKEHEAP(V)
while !EMPTY(Q)
    $u \leftarrow$ EXTRACTMIN(Q)
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            DECREASEKEY$(Q, v, dist[v])$
            $prev[v] \leftarrow u$

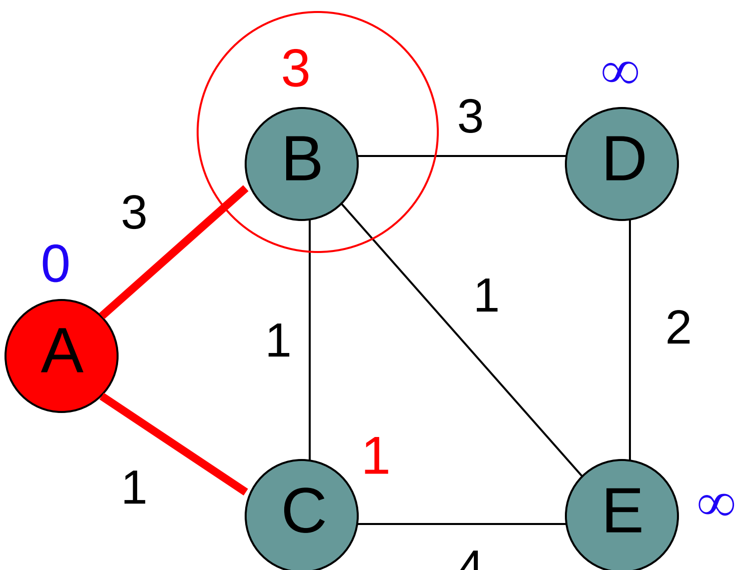


Heap

C 1
B ∞
D ∞
E ∞

DIJKSTRA(G, s)

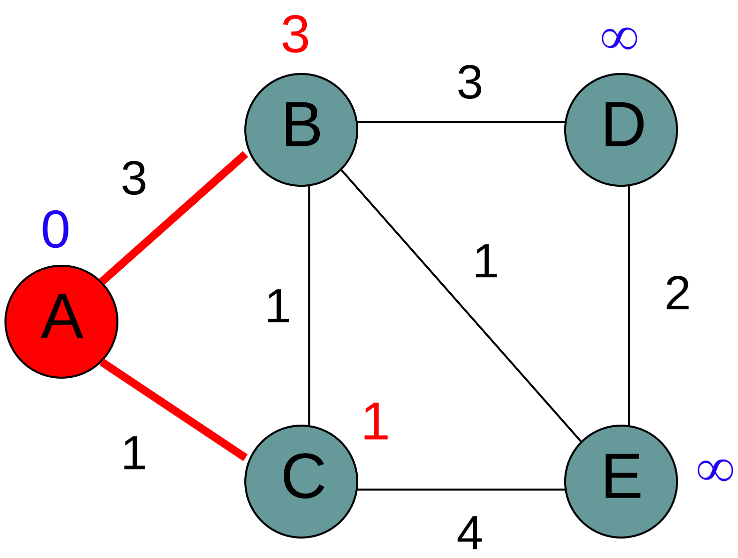
for all $v \in V$
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
$Q \leftarrow$ MAKEHEAP(V)
while !EMPTY(Q)
    $u \leftarrow$ EXTRACTMIN(Q)
    for all edges $(u, v) \in E$
        if $dist[v] > dist[u] + w(u, v)$
            $dist[v] \leftarrow dist[u] + w(u, v)$
            DECREASEKEY$(Q, v, dist[v])$
            $prev[v] \leftarrow u$



Heap

C 1
B 3
D ∞
E ∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

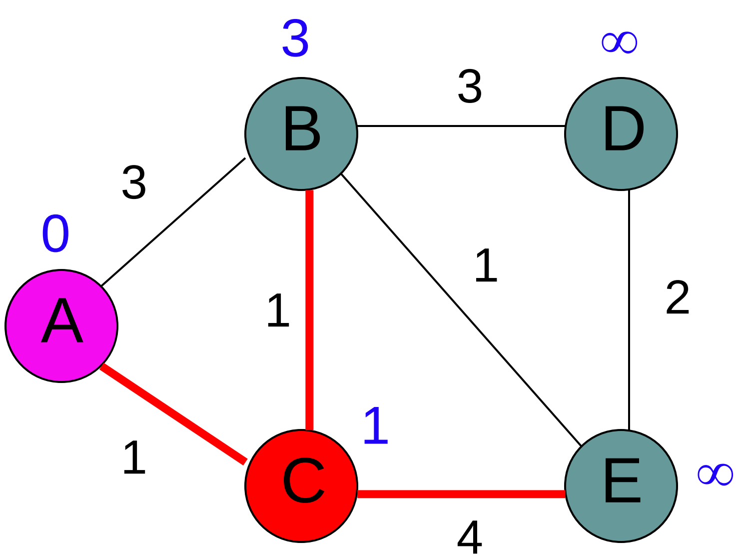


Heap	
C	1
B	3
D	∞
E	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap

B	3
D	∞
E	∞



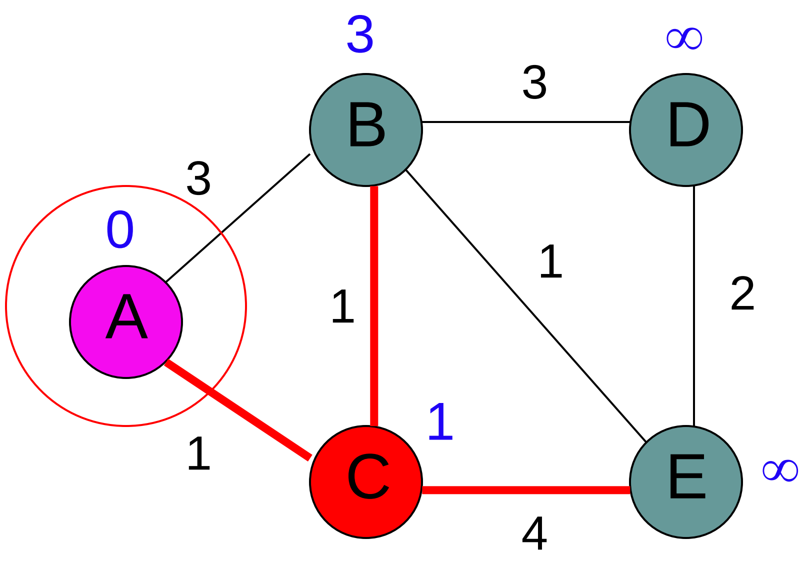
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap

B 3

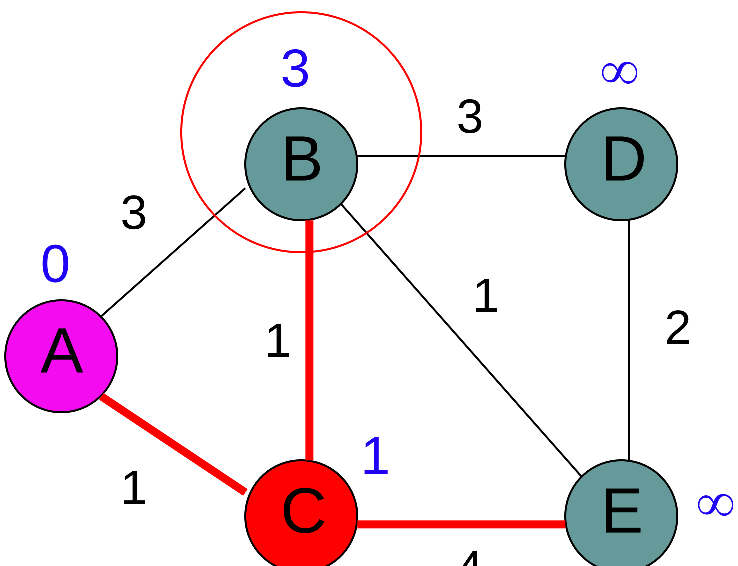
D ∞

E ∞



```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

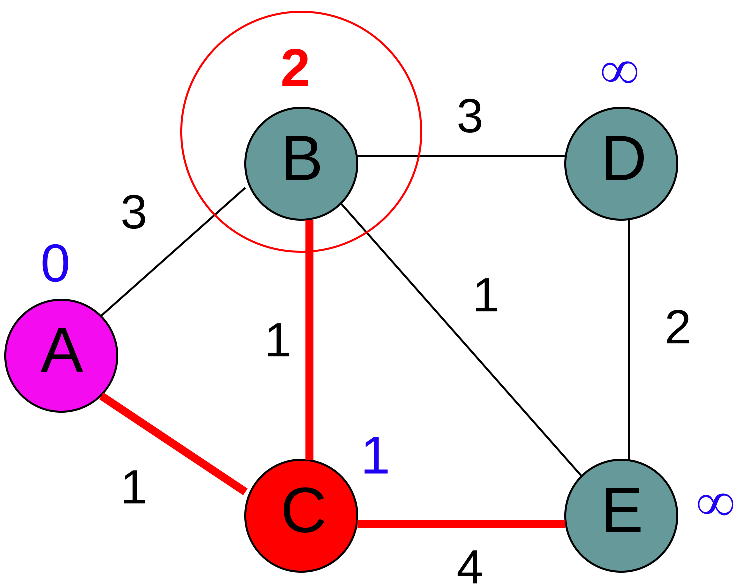
Heap	
B	3
D	∞
E	∞



```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

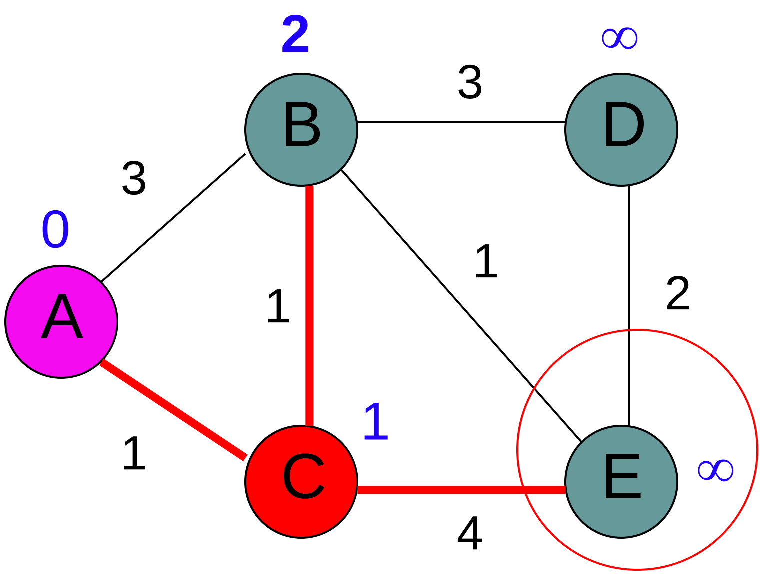
Heap

B 2
D ∞
E ∞



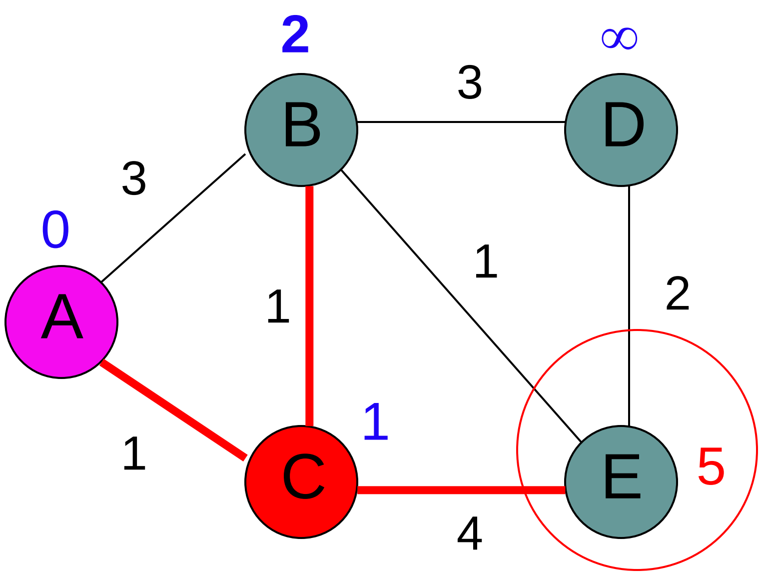
```
Dijkstra(G, s)
for all v ∈ V
    dist[v] ← ∞
    prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
    u ← ExtractMin(Q)
    for all edges (u, v) ∈ E
        if dist[v] > dist[u] + w(u, v)
            dist[v] ← dist[u] + w(u, v)
            DecreaseKey(Q, v, dist[v])
            prev[v] ← u
```

Heap	
B	2
D	∞
E	∞



DIJKSTRA(G, s)

```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

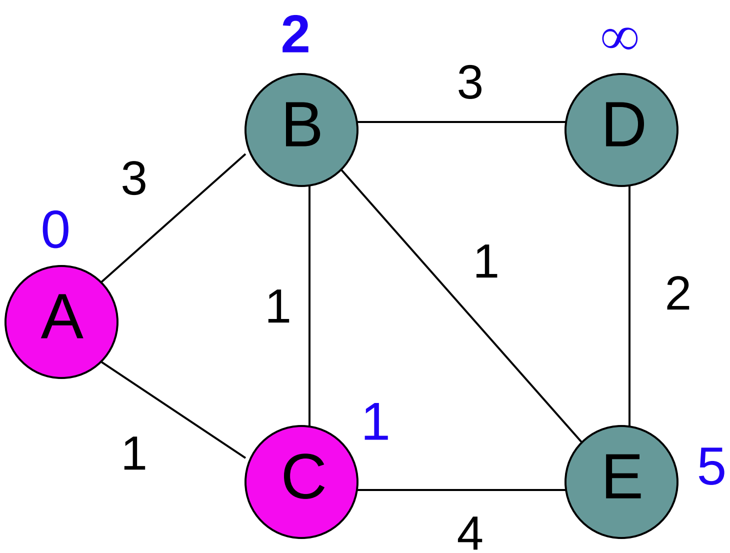


Heap

B	2
E	5
D	∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

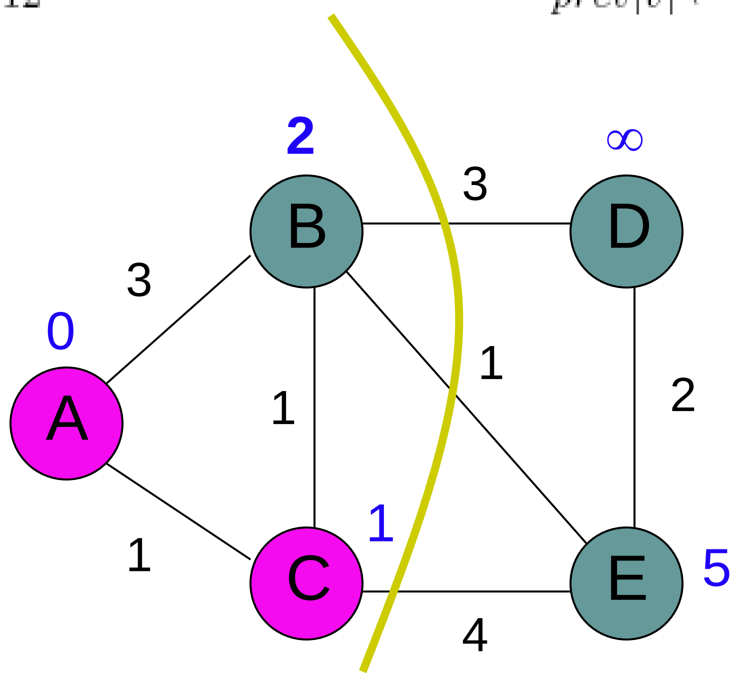
Heap	
B	2
E	5
D	∞



Frontier?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap	
B	2
E	5
D	∞



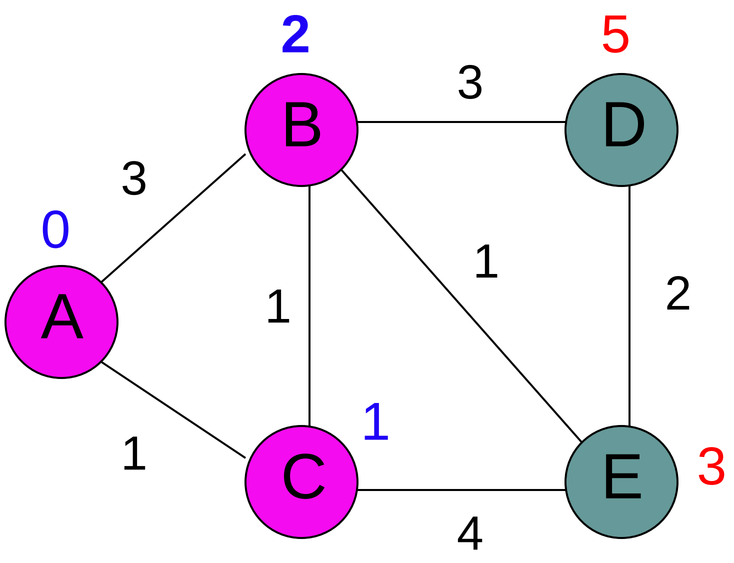
All nodes reachable from starting node within a given distance

DIJKSTRA(G, s)

```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap

E 3
D 5

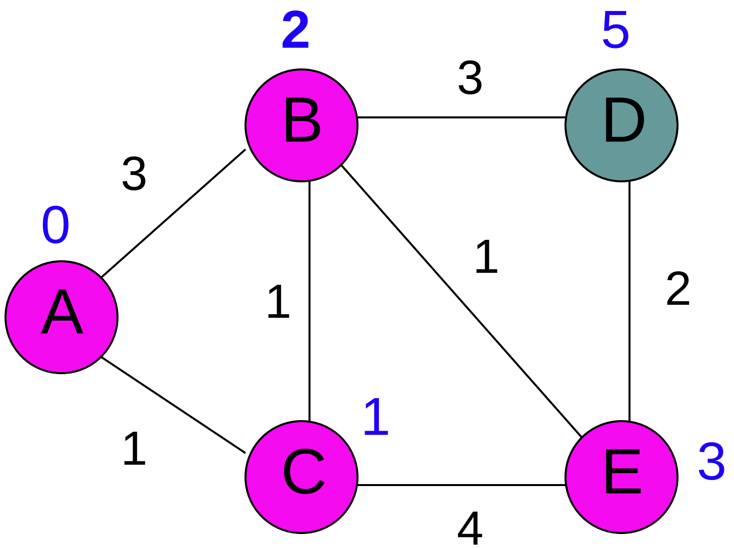


DIJKSTRA(G, s)

```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

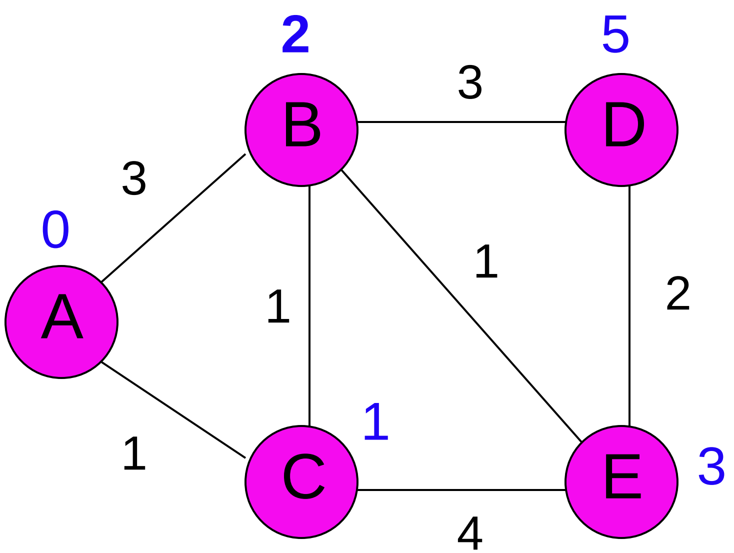
Heap

D 5



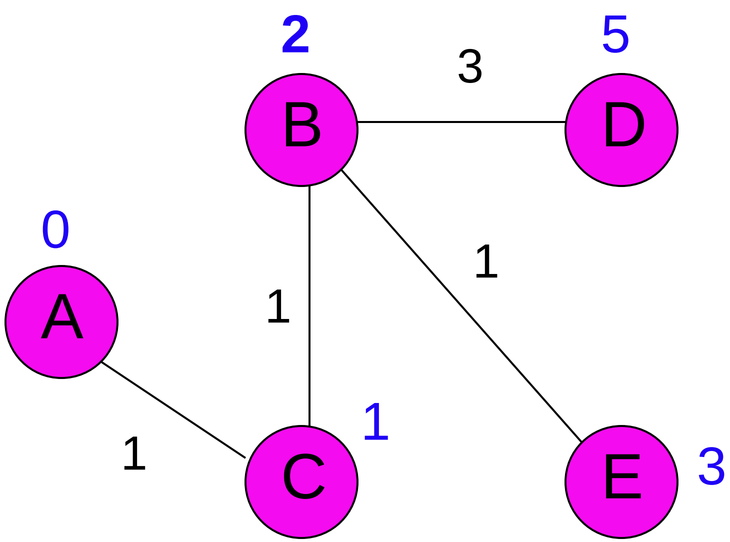
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap



```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap



Is Dijkstra’s algorithm correct?

Invariant:

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

Is Dijkstra's algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

proof?

Is Dijkstra's algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v

- The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex
- Therefore, there cannot be any other path that hasn't been visited already that would result in a shorter path

Running time?

```
Dijkstra(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

Running time?

```
Dijkstra(G, s)
  for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
  dist[s] ← 0
  Q ← MakeHeap(V)
  while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
              if dist[v] > dist[u] + w(u, v)
                    dist[v] ← dist[u] + w(u, v)
                    DecreaseKey(Q, v, dist[v])
                    prev[v] ← u
```

1 call to MakeHeap

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

|V| iterations

Running time?

DIJKSTRA(G, s)

for all $v \in V$
  $dist[v] \leftarrow \infty$
  $prev[v] \leftarrow null$
$dist[s] \leftarrow 0$
$Q \leftarrow$ MAKEHEAP(V)
while !EMPTY(Q)
  $u \leftarrow$ EXTRACTMIN(Q) |V| calls
  for all edges $(u, v) \in E$
    if $dist[v] > dist[u] + w(u, v)$
      $dist[v] \leftarrow dist[u] + w(u, v)$
      DECREASEKEY$(Q, v, dist[v])$
      $prev[v] \leftarrow u$

Running time?

$\textsc{Dijkstra}(G, s)$

```
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MakeHeap(V)
while !Empty(Q)
        u ← ExtractMin(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DecreaseKey(Q, v, dist[v])
                        prev[v] ← u
```

O(|E|) calls

Running time?

- Depends on the heap implementation

	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	O(|V|^2)	O(|E|)	O(|V|^2)
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)

Running time?

- Depends on the heap implementation

	1 MakeHeap	$\lvert V\rvert$ ExtractMin	$\lvert E\rvert$ DecreaseKey	Total
Array	$O(\lvert V\rvert)$	$O(\lvert V\rvert^2)$	$O(\lvert E\rvert)$	$O(\lvert V\rvert^2)$
Bin heap	$O(\lvert V\rvert)$	$O(\lvert V\rvert \log \lvert V\rvert)$	$O(\lvert E\rvert \log \lvert V\rvert)$	$O((\lvert V\rvert+\lvert E\rvert) \log \lvert V\rvert)$ $O(\lvert E\rvert \log \lvert V\rvert)$

Is this an improvement? If $|E| < |V|^2 / \log |V|$

Running time?

- Depends on the heap implementation

	1 MakeHeap	$\|V\|$ ExtractMin	$\|E\|$ DecreaseKey	Total
Array	$O(\|V\|)$	$O(\|V\|^2)$	$O(\|E\|)$	$O(\|V\|^2)$
Bin heap	$O(\|V\|)$	$O(\|V\| \log \|V\|)$	$O(\|E\| \log \|V\|)$	$O((\|V\|+\|E\|) \log \|V\|)$ $O(\|E\| \log \|V\|)$
Fib heap	$O(\|V\|)$	$O(\|V\| \log \|V\|)$	$O(\|E\|)$	$O(\|V\| \log \|V\| + \|E\|)$

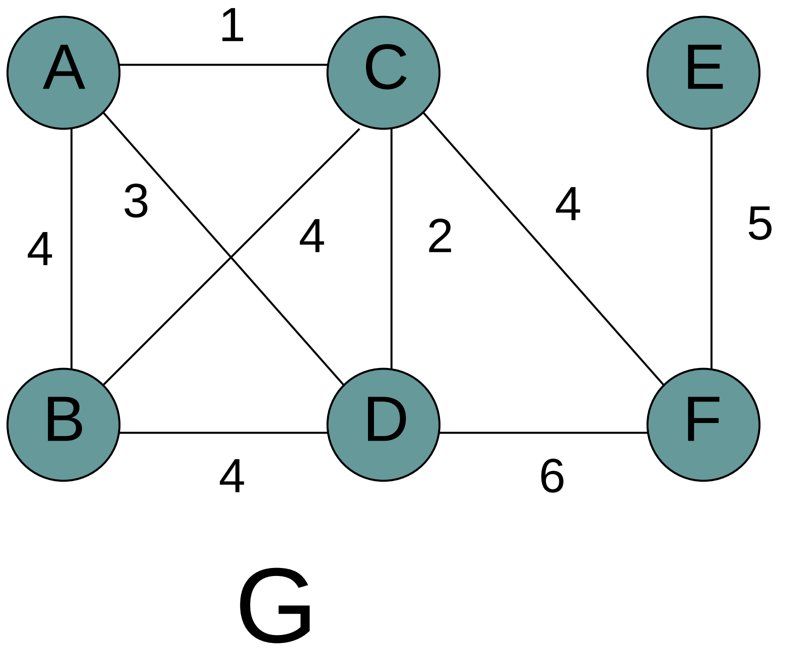
Kruskal's algorithm

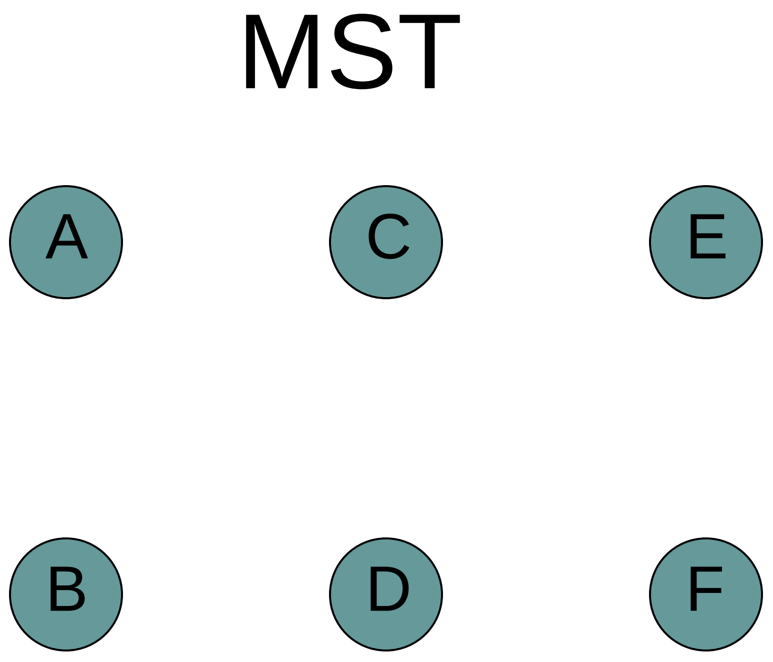
Given a partition S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.

```
Kruskal(G)
for all v ∈ V
        MakeSet(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if Find-Set(u) ≠ Find-Set(v)
                add edge to T
                Union(Find-Set(u),Find-Set(v))
```

Kruskal's algorithm

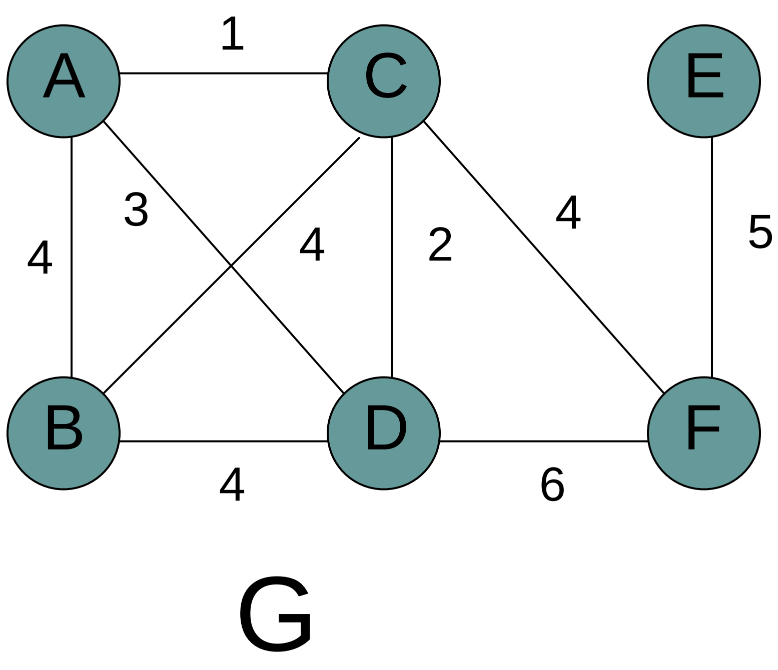
Add smallest edge that connects two sets not already connected

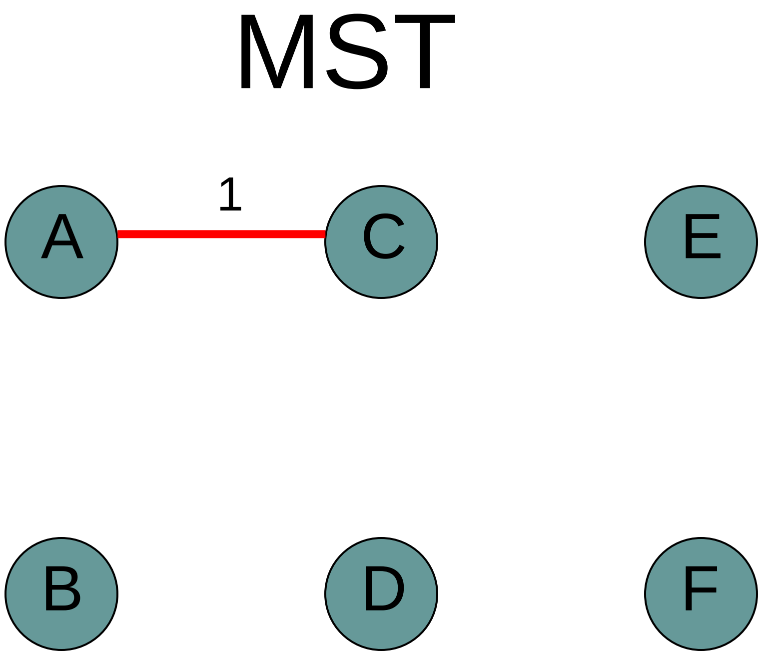




Kruskal's algorithm

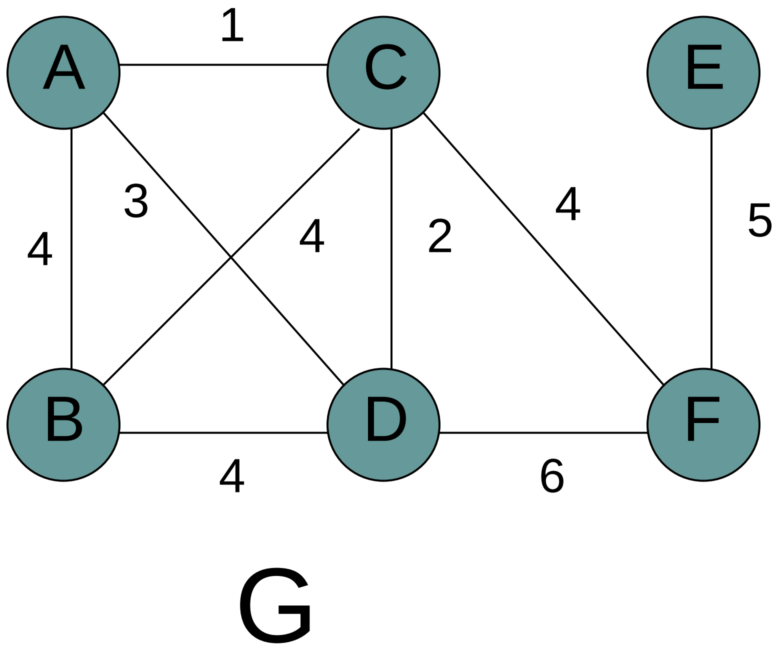
Add smallest edge that connects
two sets not already connected

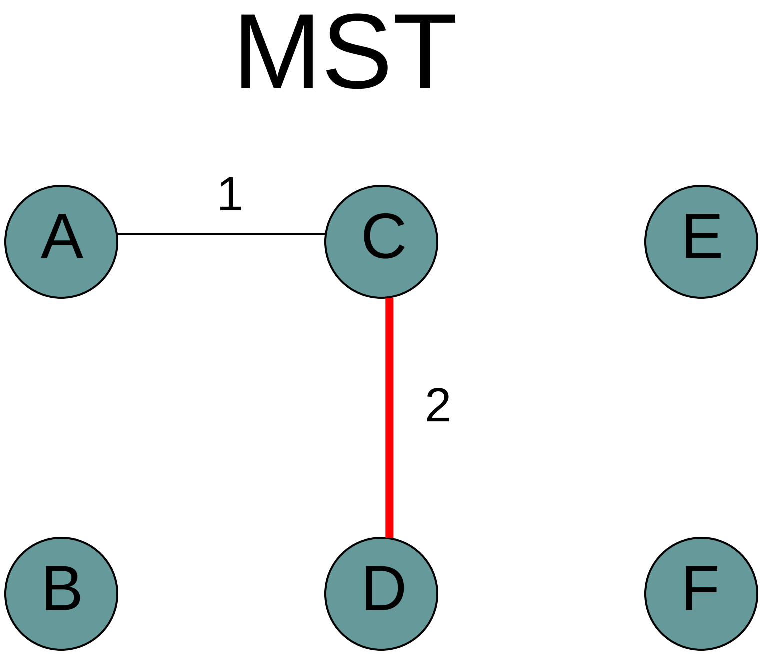




Kruskal's algorithm

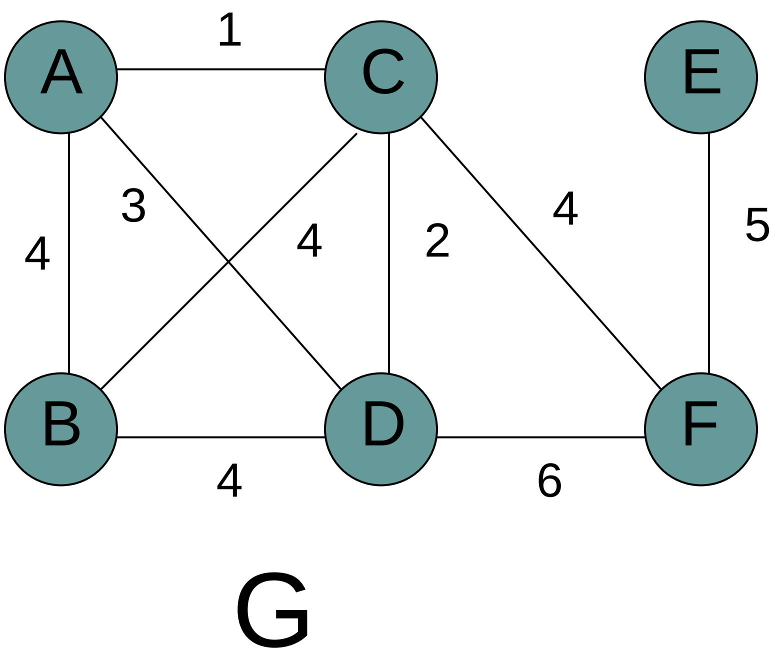
Add smallest edge that connects two sets not already connected

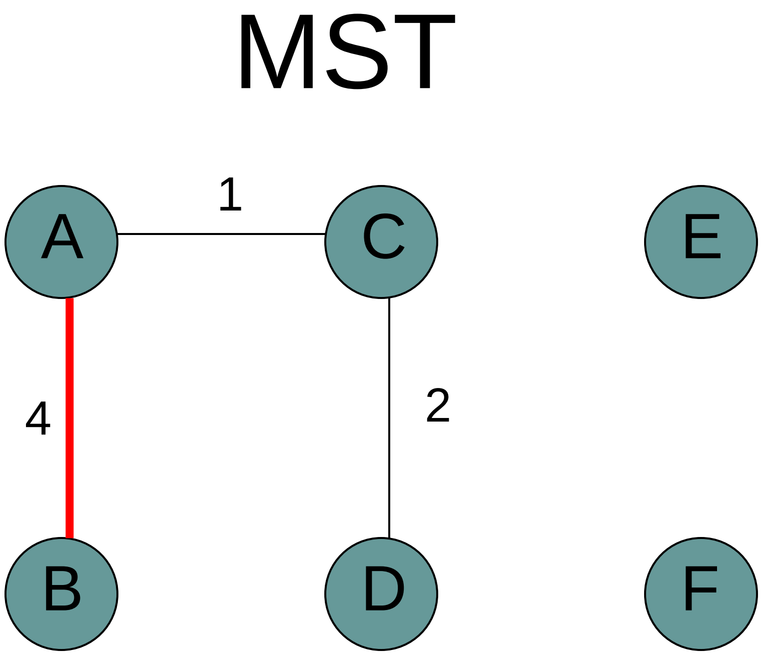




Kruskal’s algorithm

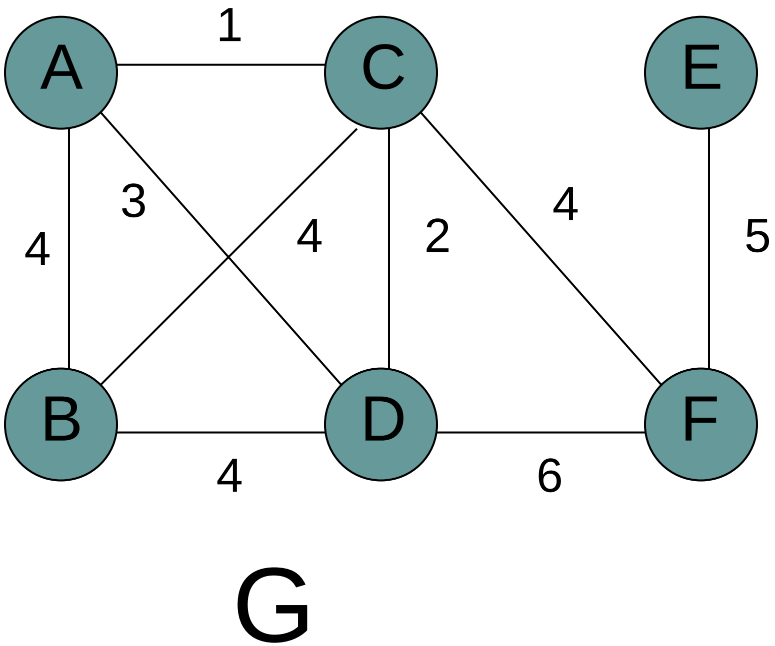
Add smallest edge that connects two sets not already connected

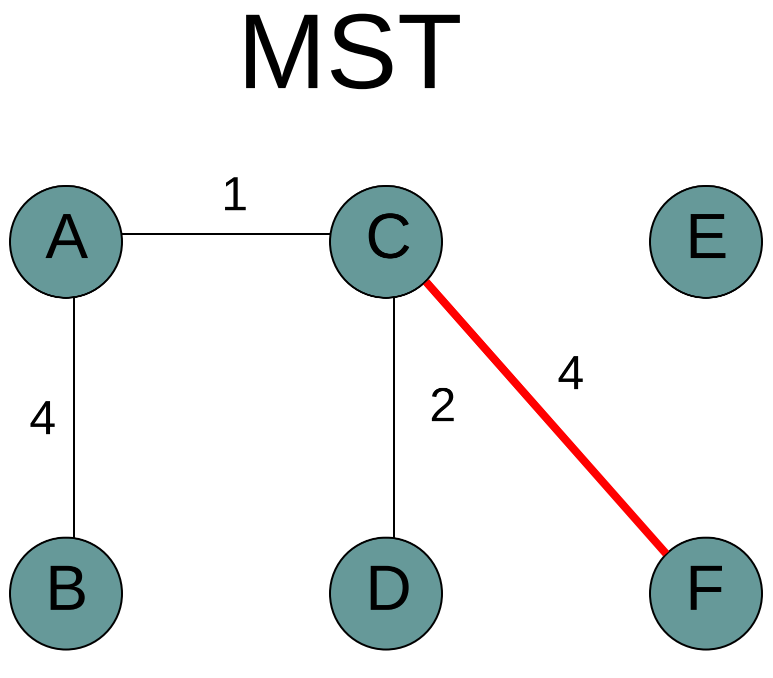




Kruskal’s algorithm

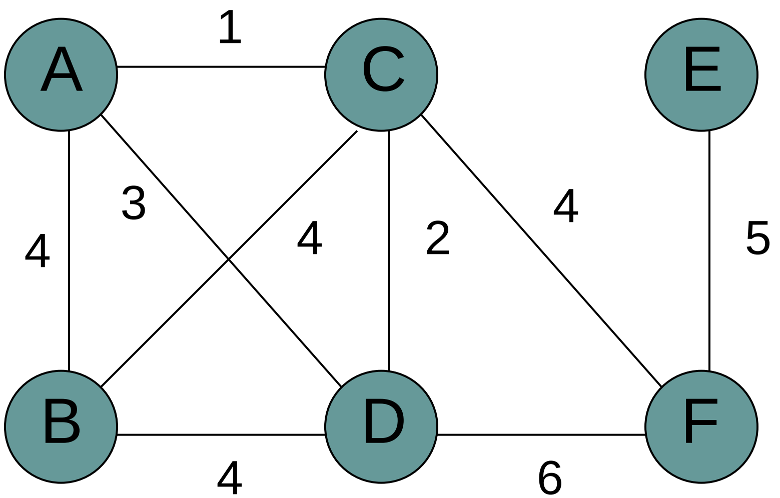
Add smallest edge that connects two sets not already connected





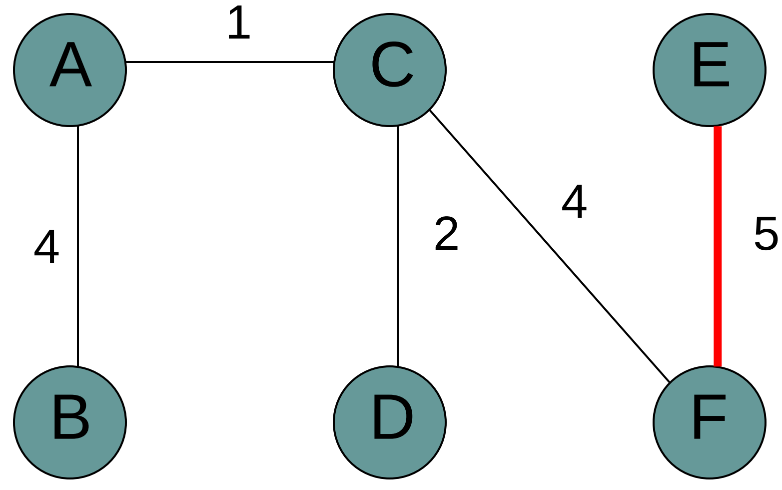
Kruskal’s algorithm

Add smallest edge that connects
two sets not already connected



G

MST



Correctness of Kruskal's

- Never adds an edge that connects already connected vertices
- Always adds lowest cost edge to connect two sets. By min cut property, that edge must be part of the MST

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

Running time of Kruskal's

```
KRUSKAL(G)
for all v ∈ V
        MAKESET(v)
T ← {}
sort the edges of E by weight
for all edges (u, v) ∈ E in increasing order of weight
        if FIND-SET(u) ≠ FIND-SET(v)
                add edge to T
                UNION(FIND-SET(u),FIND-SET(v))
```

Running time of Kruskal's

KRUSKAL(G)

for all $v \in V$
    MAKESET(v)

|V| calls to MakeSet

$T \leftarrow \{\}$

sort the edges of E by weight

O(|E| log |E|)

for all edges $(u, v) \in E$ in increasing order of weight
    if FIND-SET(u) $\neq$ FIND-SET(v)

2 |E| calls to FindSet

        add edge to T
        UNION(FIND-SET(u),FIND-SET(v))

|V| calls to Union

Running time of Kruskal's

Disjoint set data structure

$O(|E| \log |E|) +$

	MakeSet	FindSet $|E|$ calls	Union $|V|$ calls	Total
Linked lists	$|V|$	$O(|V|\,|E|)$	$|V|$	$O(|V||E| + |E| \log |E|)$ $O(|V|\,|E|)$
Linked lists + heuristics	$|V|$	$O(|E| \log |V|)$	$|V|$	$O(|E| \log |V| + |E| \log |E|)$ $O(|E| \log |E|)$

Prim's algorithm

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

Prim’s algorithm

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Prim's algorithm

```
PRIM(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MAKEHEAP(key)
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

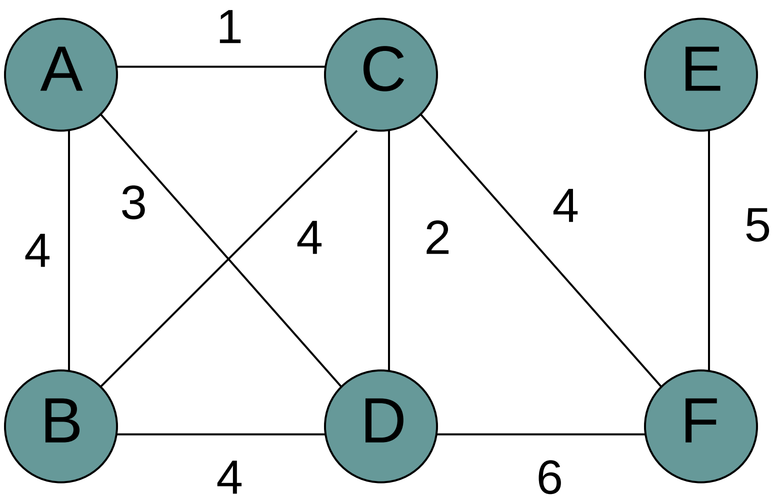
Prim's algorithm

Start at some root node and build out the MST by adding the lowest weighted edge at the frontier

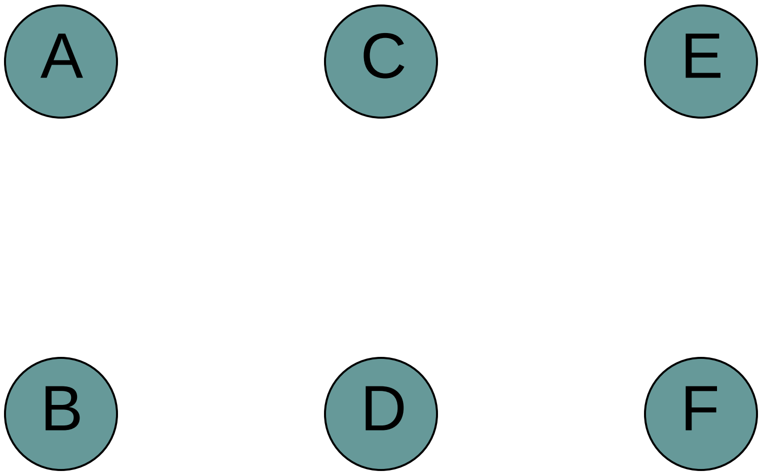
```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

Prim's

while $!Empty(H)$
    $u \leftarrow$ EXTRACT-MIN(H)
    $visited[u] \leftarrow true$
    for each edge $(u, v) \in E$
        if $!visited[v]$ and $w(u, v) < key(v)$
            DECREASE-KEY$(v, w(u, v))$
            $prev[v] \leftarrow u$

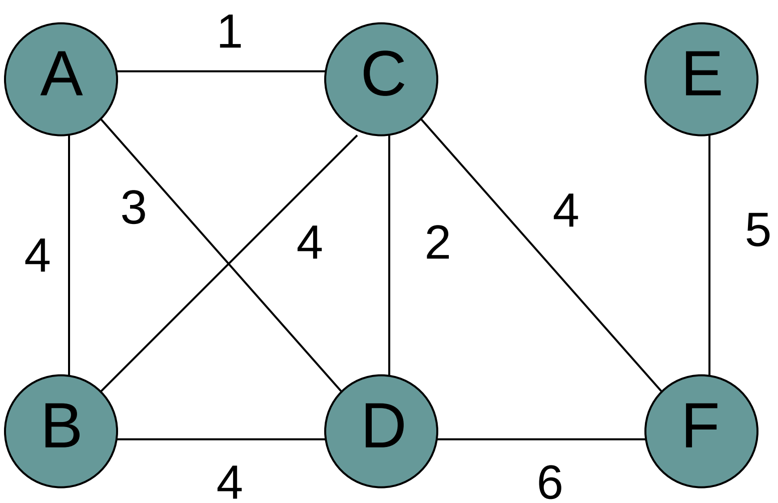


MST

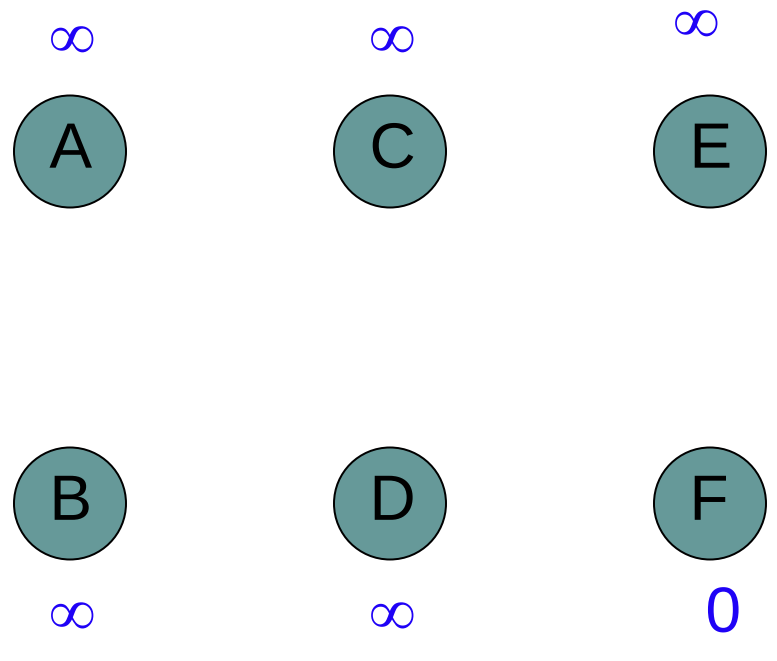


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

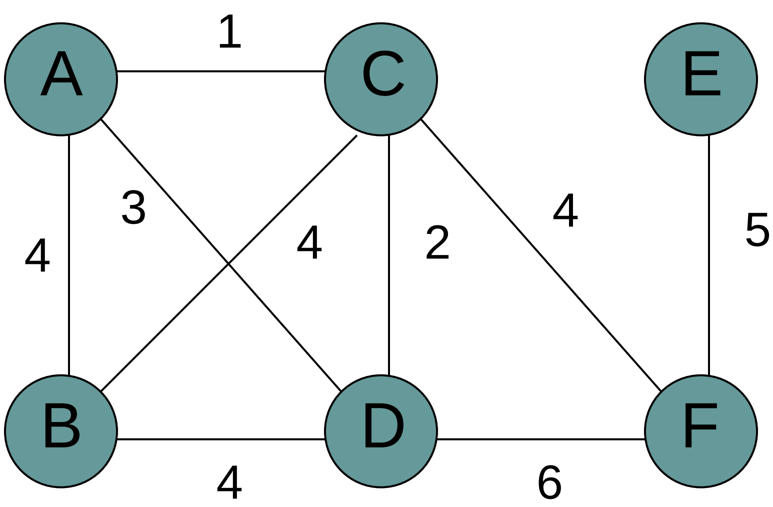


MST

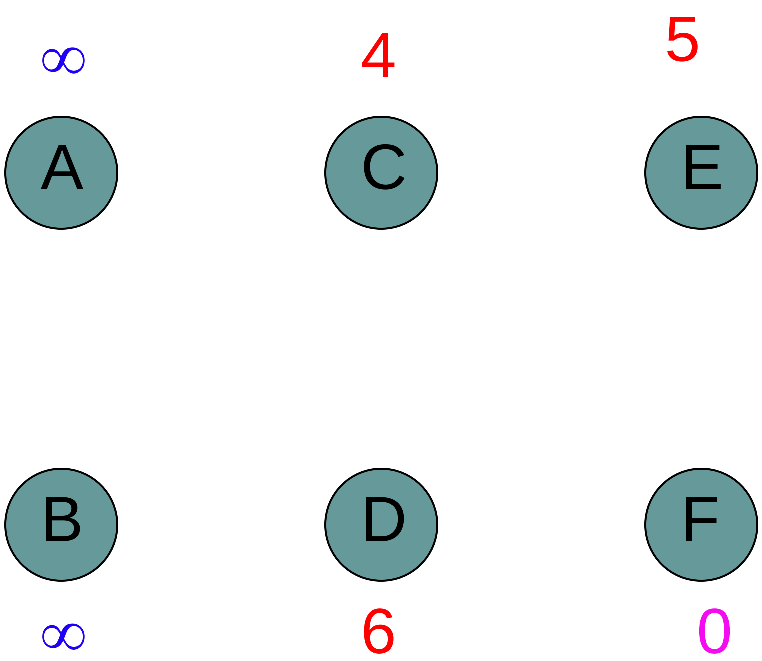


Prim's

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

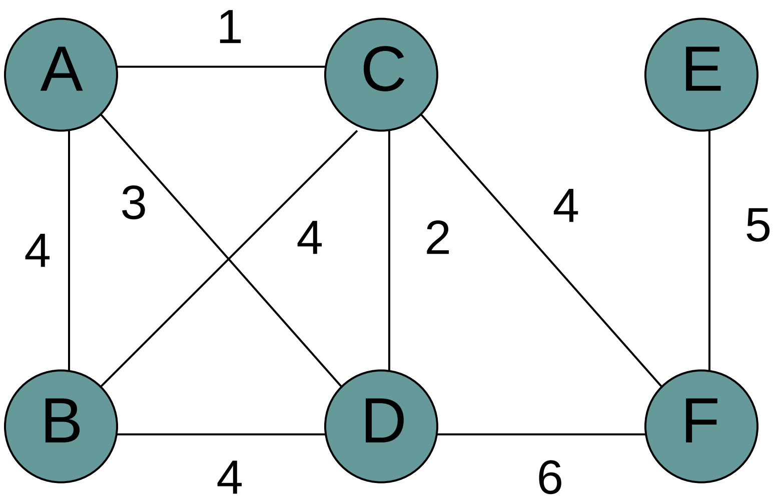


MST

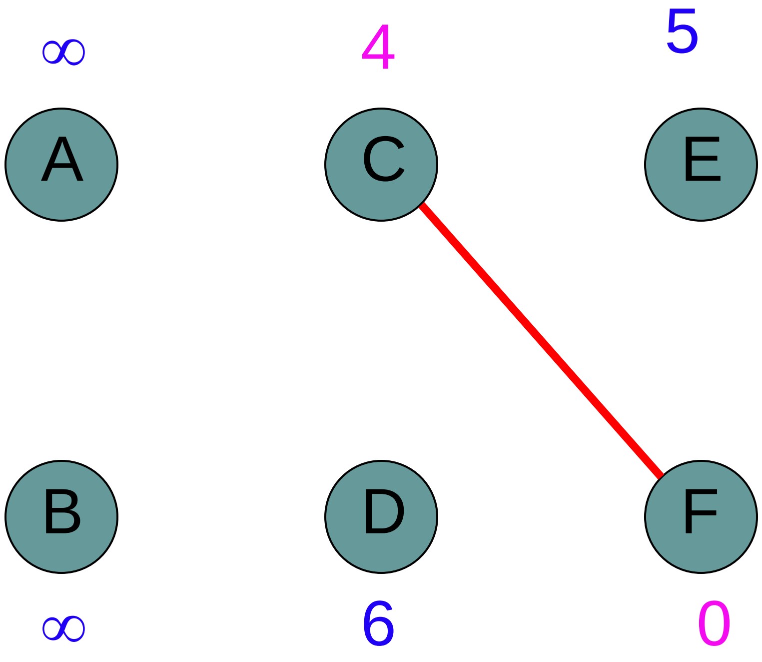


Prim's

```
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
            if !visited[v] and w(u, v) < key(v)
                DECREASE-KEY(v, w(u, v))
                prev[v] ← u
```

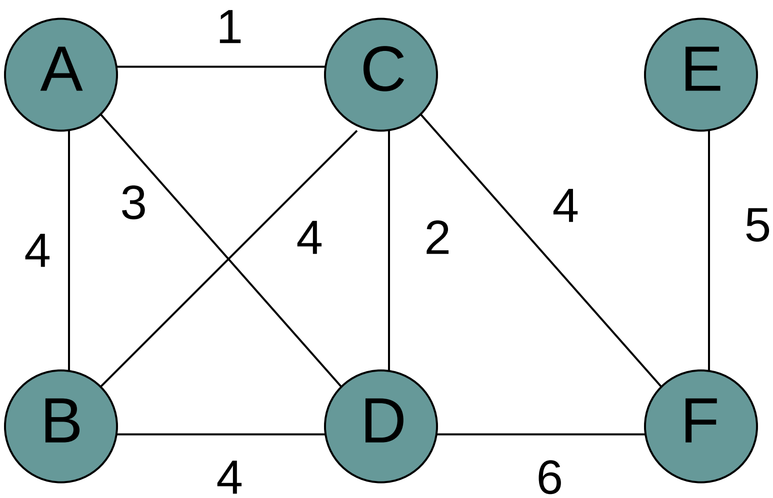


MST

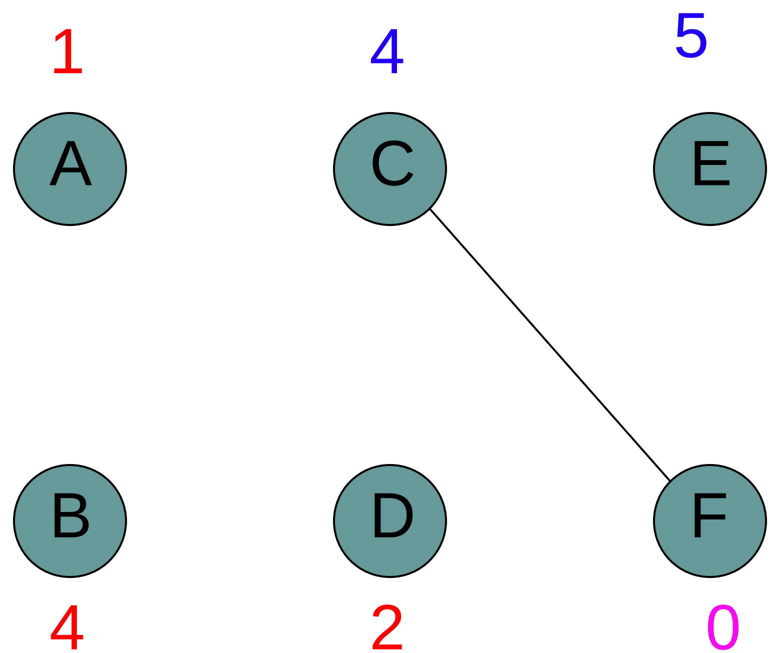


Prim's

```
while !Empty(H)
        u ← EXTRACT-MIN(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        DECREASE-KEY(v, w(u, v))
                        prev[v] ← u
```

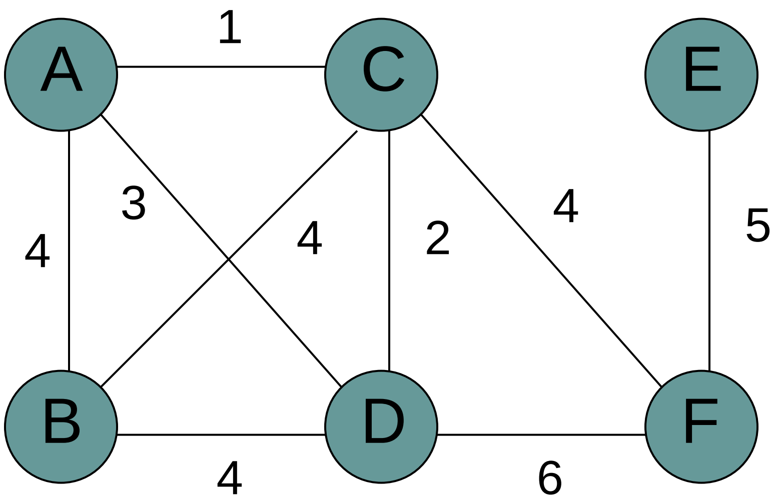


MST

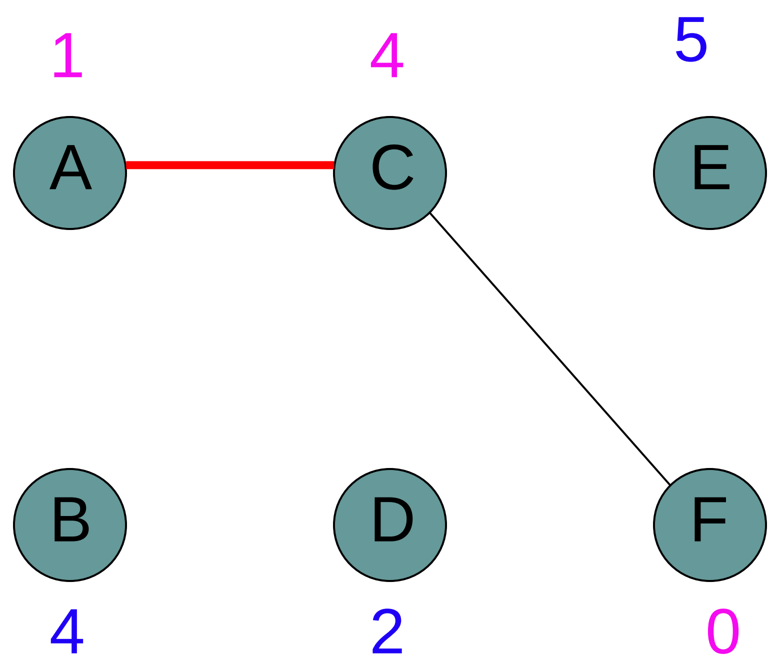


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

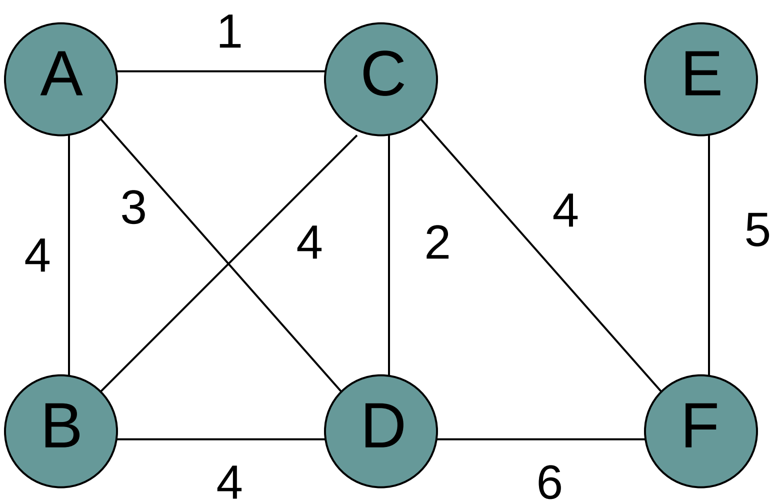


MST

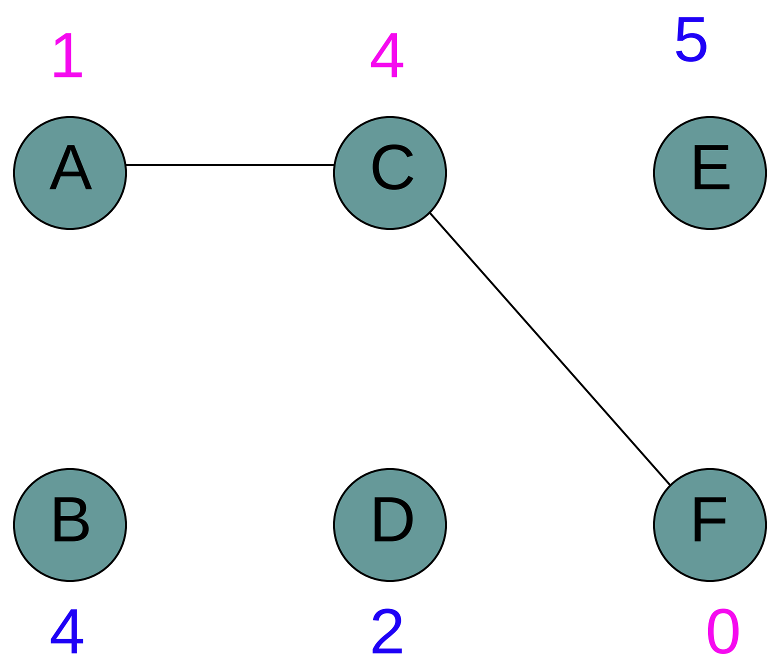


Prim's

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

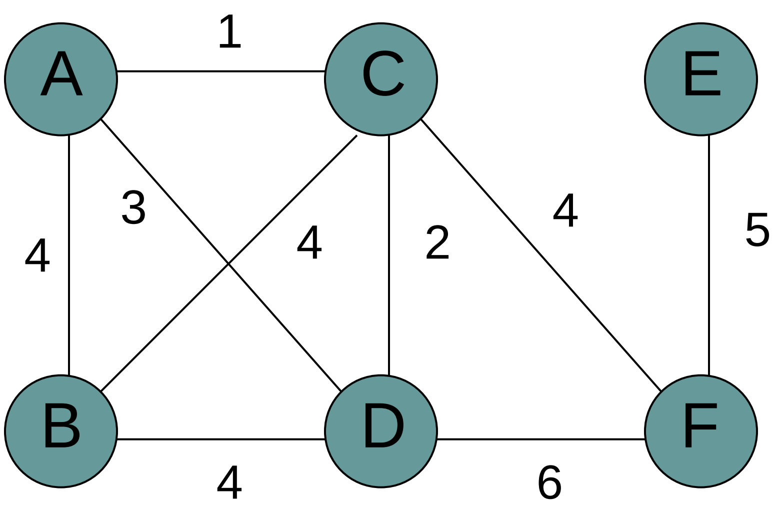


MST

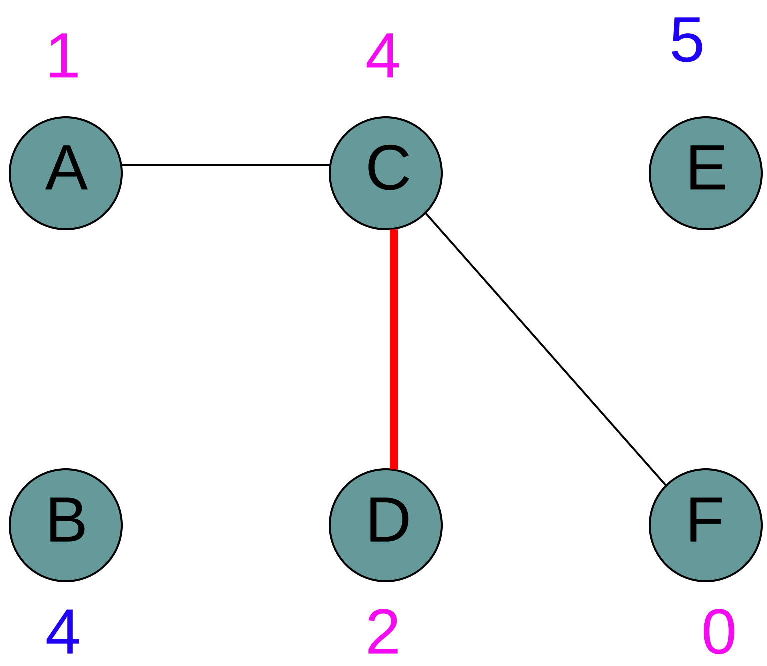


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

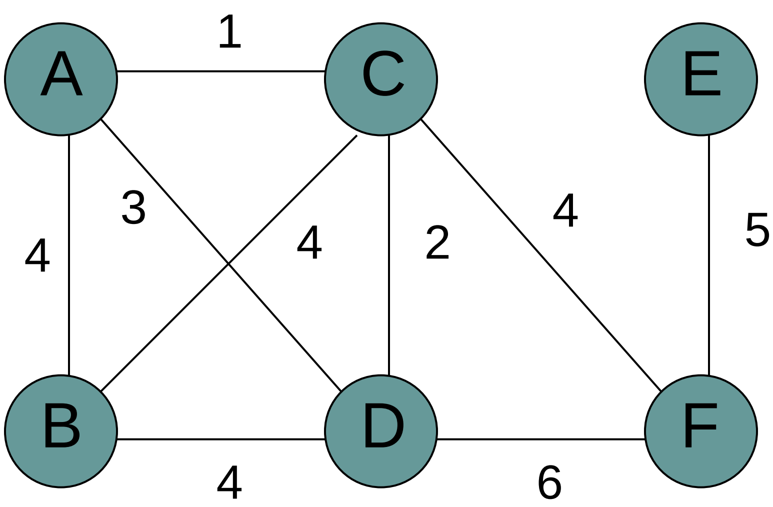


MST

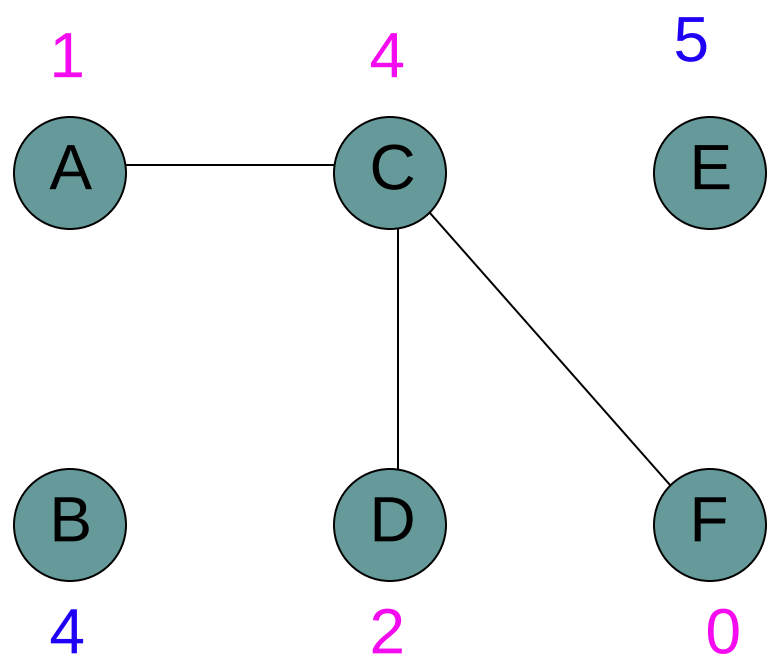


```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

Prim's

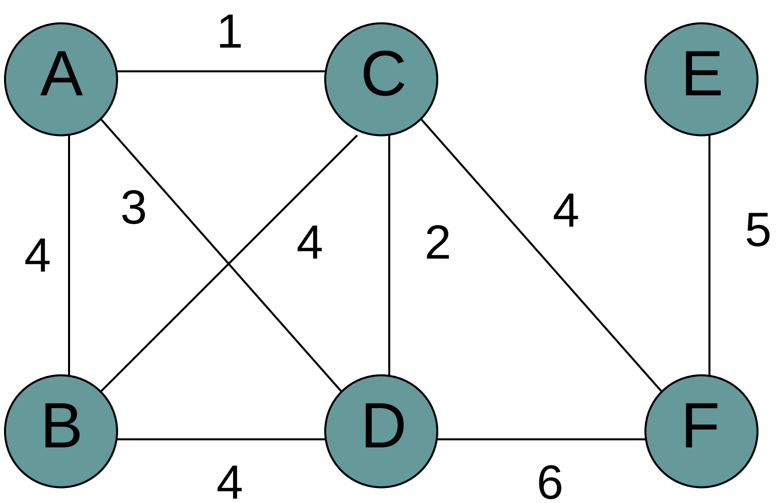


MST

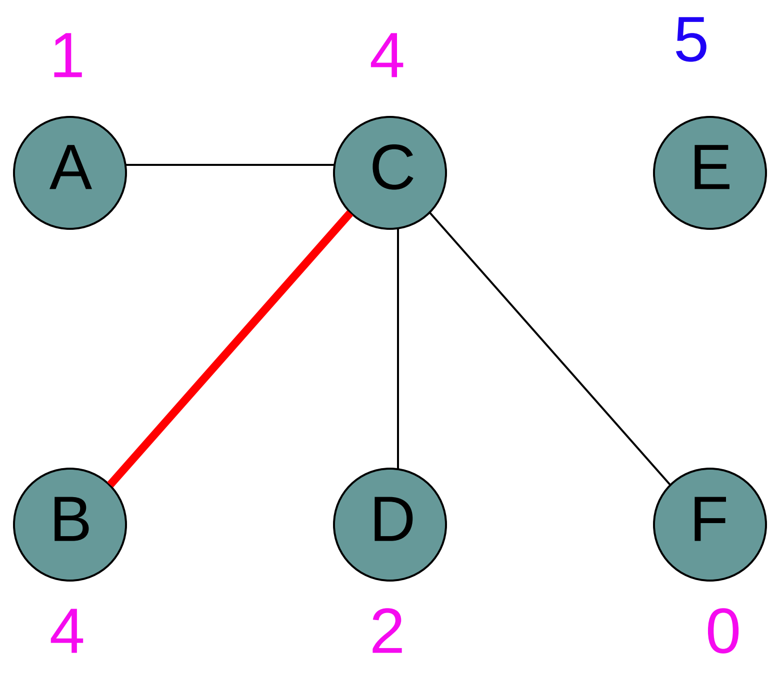


Prim's

```
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

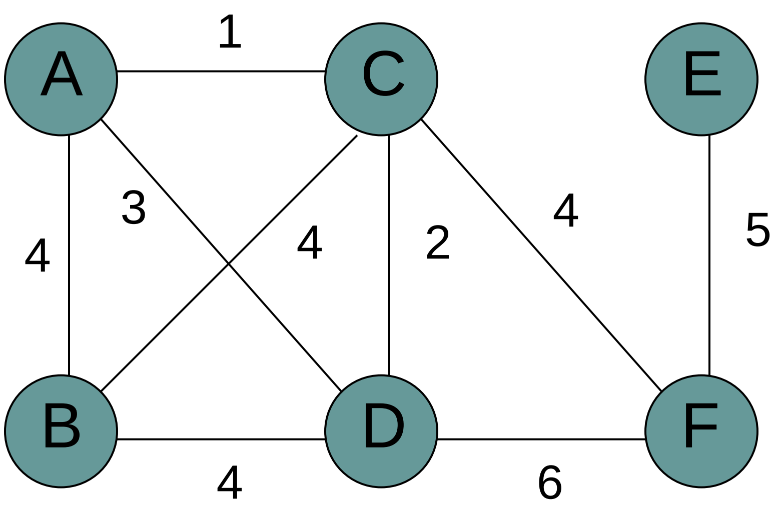


MST

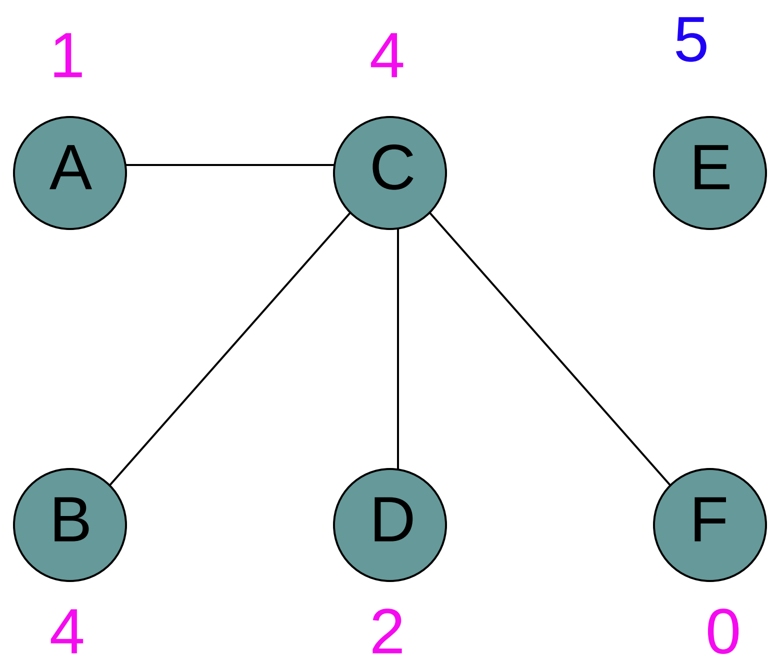


Prim's

```
while !Empty(H)
    u ← Extract-Min(H)
    visited[u] ← true
    for each edge (u, v) ∈ E
        if !visited[v] and w(u, v) < key(v)
            Decrease-Key(v, w(u, v))
            prev[v] ← u
```

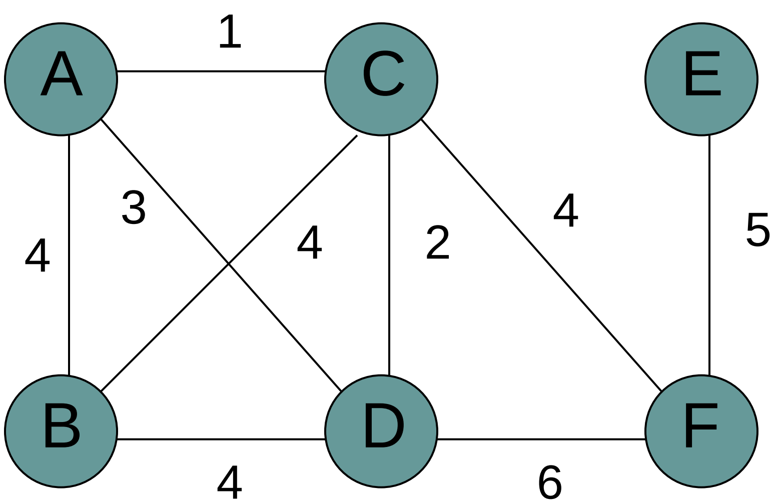


MST

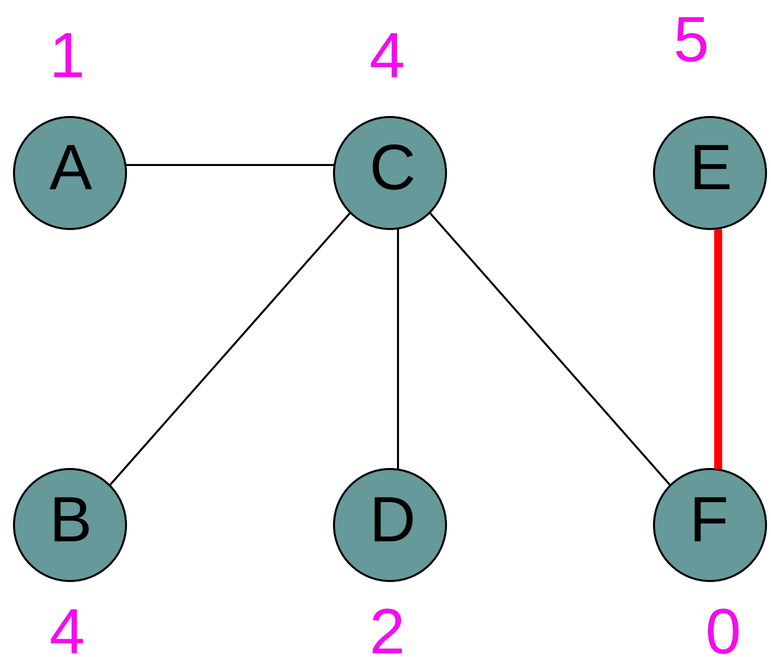


Prim's

$$
\begin{array}{l}
\textbf{while } !Empty(H) \\
\quad u \leftarrow \textsc{Extract-Min}(H) \\
\quad visited[u] \leftarrow true \\
\quad \textbf{for } \text{each edge } (u, v) \in E \\
\quad\quad \textbf{if } !visited[v] \text{ and } w(u, v) < key(v) \\
\quad\quad\quad \textsc{Decrease-Key}(v, w(u, v)) \\
\quad\quad\quad prev[v] \leftarrow u
\end{array}
$$



MST



Correctness of Prim's?

- Can we use the min-cut property?
  - Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.
- Let S be the set of vertices visited so far
- The only time we add a new edge is if it's the lowest weight edge from S to V-S

Running time of Prim's

```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

Running time of Prim's

```
Prim(G, r)
for all v ∈ V
        key[v] ← ∞
        prev[v] ← null
key[r] ← 0
H ← MakeHeap(key)
while !Empty(H)
        u ← Extract-Min(H)
        visited[u] ← true
        for each edge (u, v) ∈ E
                if !visited[v] and w(u, v) < key(v)
                        Decrease-Key(v, w(u, v))
                        prev[v] ← u
```

$\Theta(|V|)$

$\Theta(|V|)$

$|V|$ calls to Extract-Min

$|E|$ calls to Decrease-Key

Running time of Prim's

Same as Dijksta's algorithm

	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	O(|V|^2)	O(|E|)	O(|V|^2)
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)
Fib heap	O(|V|)	O(|V| log |V|)	O(|E|)	**O(|V| log |V| + |E|)**

Kruskal's: O(|E| log |E|)