

The Graph Data Structure

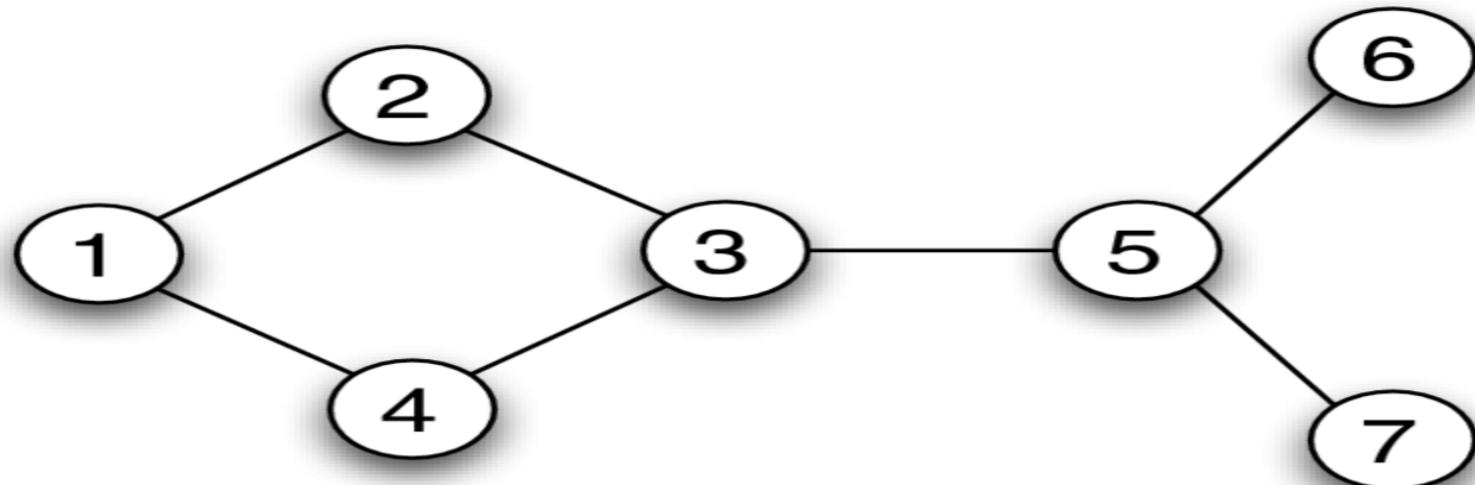
Dr. Amit Praseed

Graphs

- A graph is a data structure that is commonly represented as $G = (V, E)$, where V is a set of vertices and E is a set of edges
- Graphs are commonly used to represent a large number of real world problems
 - Railways, roadways, airline routes, transmission towers etc.
 - Routing traffic over the Internet
 - Representing game outcomes
 - Representing a problem search space

Graph Terminology

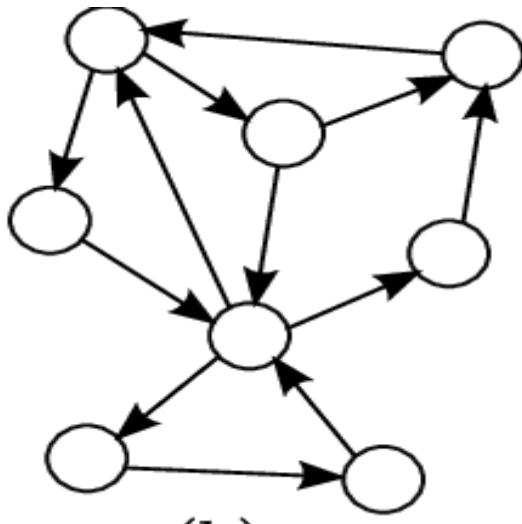
- A graph is a collection of vertices, V and a collection of edges, E
- Every edge $e \in E$ can be represented as $\{u, v\}$ where $u, v \in V$



Graph Terminology

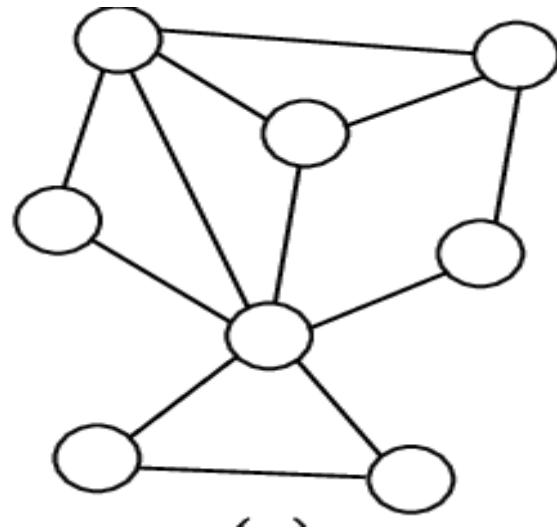
Directed Graphs

- A graph in which the edges are represented as **ordered pairs** (u, v) are called directed graphs
- Edges have direction



Undirected Graphs

- A graph in which the edges are represented as **unordered pairs** (u, v) are called undirected graphs
- Edges do not have direction



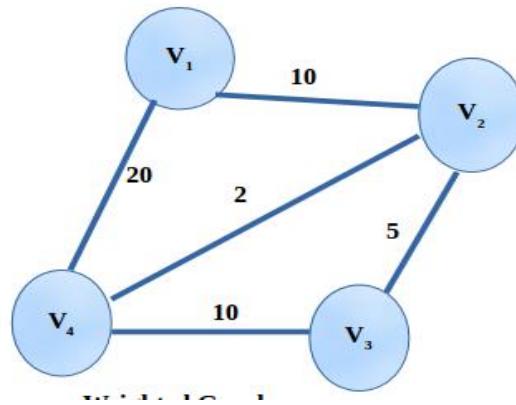
Graph Terminology

- Graphs without loops and parallel edges are often called simple graphs; non-simple graphs are sometimes called multigraphs
- For any edge $u-v$ in an undirected graph, we call u a neighbor of v and vice versa, and we say that u and v are adjacent.
- The degree of a node is its number of neighbors.
- For any directed edge $u-v$, we call u a predecessor of v, and we call v a successor of u. The in-degree of a vertex is its number of predecessors; the out-degree is its number of successors

Graph Terminology

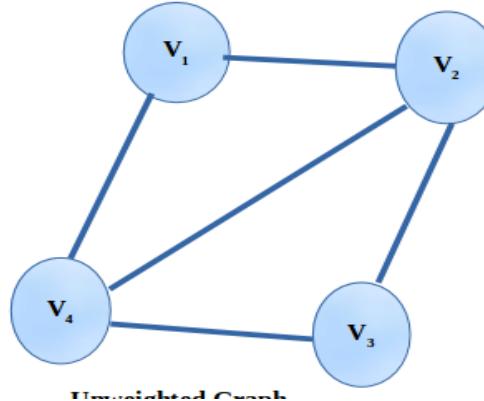
Weighted Graphs

- A graph in which the edges are assigned a numeric value (called weight) are called weighted graphs
- Can represent path length, delay etc.



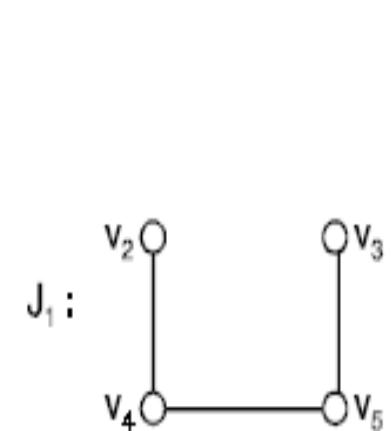
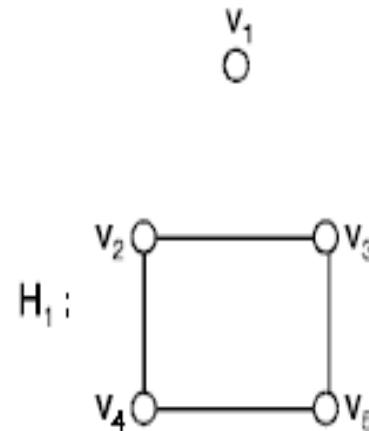
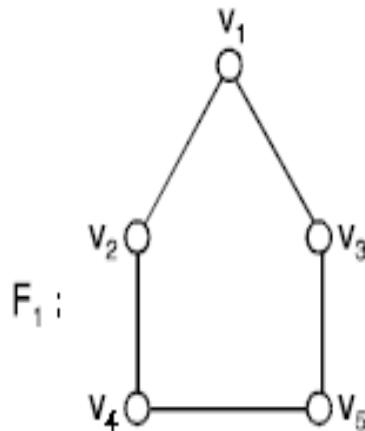
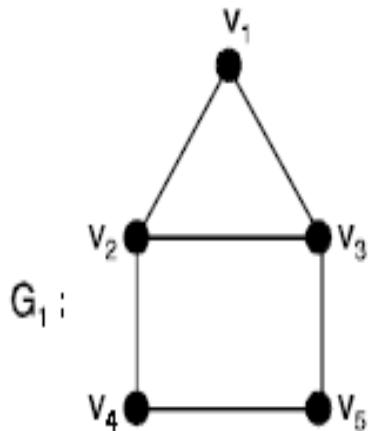
Undirected Graphs

- A graph in which the edges do not have any numeric value associated with them are called unweighted graphs



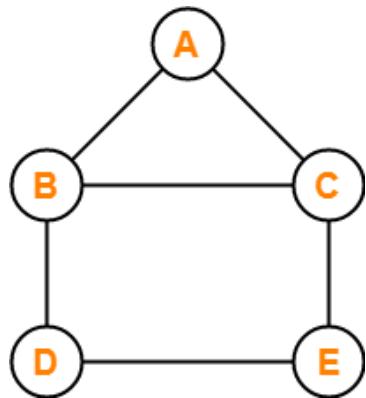
Graph Terminology

- A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$
 - By definition, G is a subgraph of itself
- A proper subgraph of G is any subgraph other than G itself.



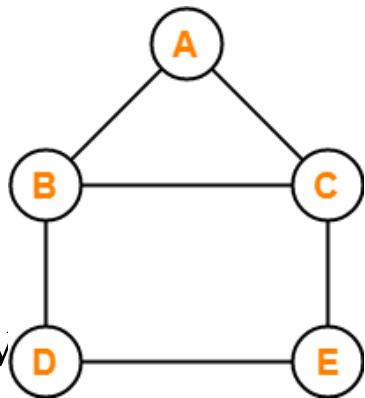
Walks, Trails, Paths

- A walk is a sequence of vertices, where each adjacent pair of vertices are adjacent in G
 - A vertex can be traversed more than once
 - An edge can be used more than once
- Eg: d b a c e d e c is a walk of length 7
- If a walk starts and ends at the same vertex it is called a closed walk
 - Otherwise it is called an open walk



Walks, Trails, Paths

- A path is a walk in which each vertex is visited at most once
 - A vertex cannot be traversed more than once
 - Eg: a b c e d is a path of length 4
- A trail is a walk in which an edge can be traversed at most once
 - A vertex can be visited more than once
 - An edge can only be visited once

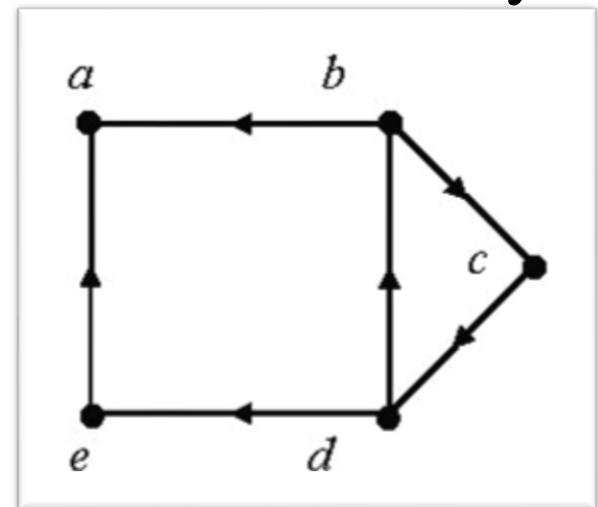


Connected Graphs

- Two vertices u and v in a graph G , v is said to be reachable from u if there exists a path between u and v .
- An undirected graph is connected if every vertex is reachable from every other vertex.
- Every undirected graph consists of one or more components, which are its maximal connected subgraphs; two vertices are in the same component if and only if there is a path between them

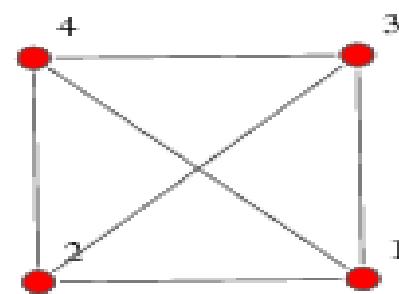
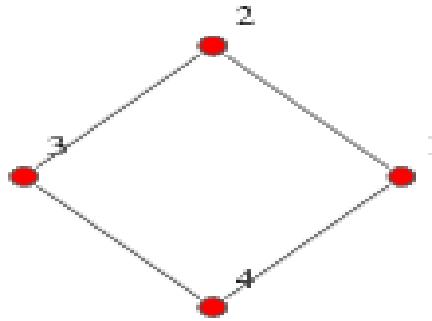
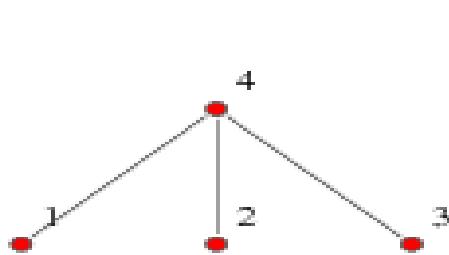
Connected Graphs

- A directed graph G is said to be strongly connected if there exists a path between any two vertices u and v
 - Note that the path needs to be a directed path
- The given graph is not strongly connected (why?)
- A directed graph G is said to be weakly connected if the graph is not strongly connected, but the underlying undirected graph is connected



Adjacency Matrix

- An adjacency matrix is a $|V| \times |V|$ matrix
 - $A[i, j] = 1$ if there is an edge from vertex u to vertex v
 - Otherwise, $A[i, j] = 0$



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

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

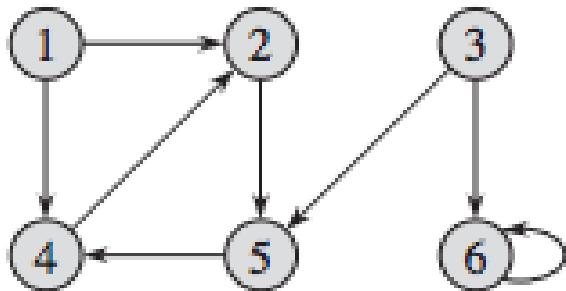
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Adjacency Matrix

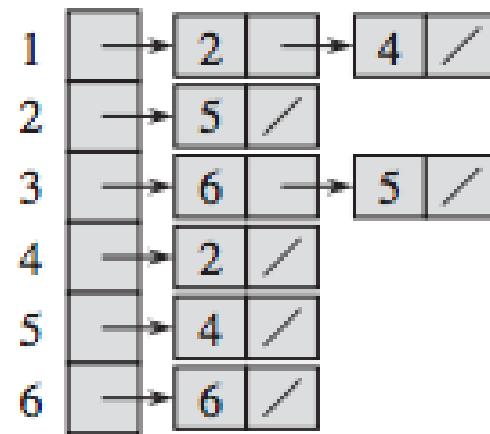
- For undirected graphs, the adjacency matrix is always symmetric, meaning $A[u, v] = A[v, u]$ for all vertices u and v and the diagonal entries $A[u, u]$ are all zeros.
- For directed graphs, the adjacency matrix may or may not be symmetric, and the diagonal entries may or may not be zero.
- Given an adjacency matrix
 - We can decide in $\theta(1)$ time whether two vertices are connected by an edge
 - We can also list all the neighbors of a vertex in $\theta(V)$ time.
 - Adjacency Matrices require $\theta(V^2)$ space, regardless of how many edges the graph actually has
- Wastage of space and time for sparse graphs

Adjacency List

- An adjacency list is an array of lists, each containing the neighbors of one of the vertices
 - For undirected graphs, each edge $u-v$ is stored twice; for directed graphs, each edge $u-v$ is stored only once



(a)



(b)

Adjacency List

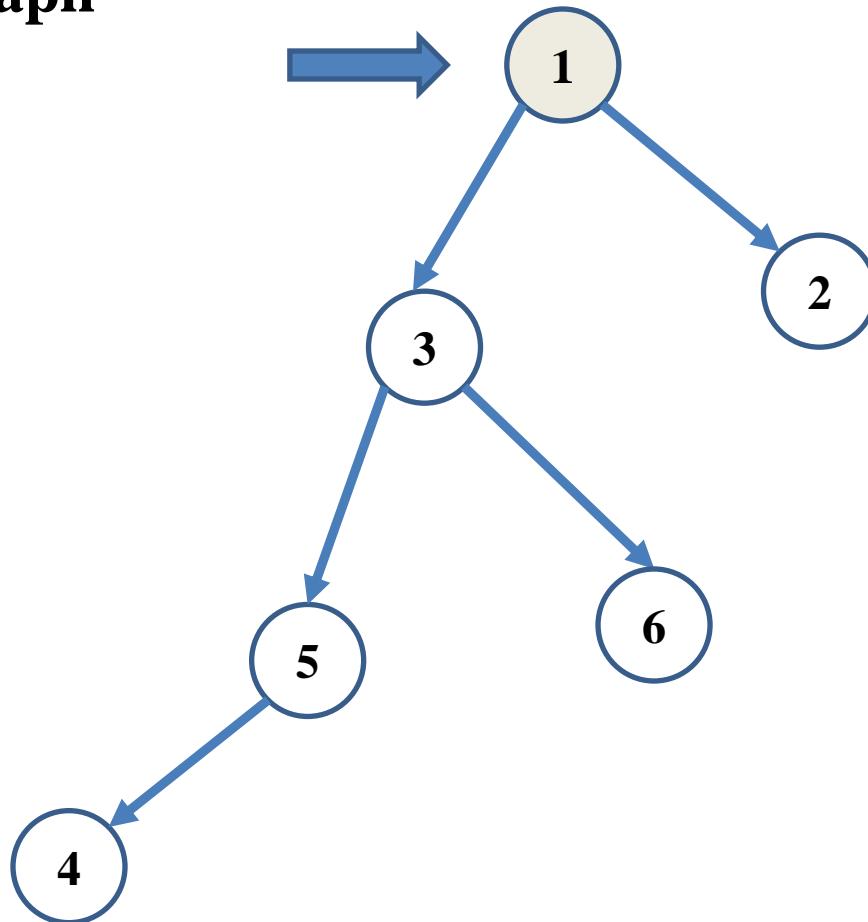
- Given an adjacency list
 - We can list the neighbors of a node v in $O(1 + \deg(v))$ time
 - We can determine whether $u-v$ is an edge in $O(1 + \deg(u))$ time
 - We can traverse the graph in $O(V + E)$ time
 - Adjacency lists require a space complexity of $O(V + E)$
- Adjacency lists are usually preferred for representing sparse graphs, but dense graphs can be more efficiently represented using adjacency matrices

Depth First Search

- Depth First Search (DFS) is a graph traversal algorithm
- As the name suggests, DFS explores one path in a graph completely before exploring a new one
- When DFS hits a dead end on a path, it backtracks and starts exploring a new path from the previous node
- This behaviour is suggestive of a LIFO data structure – STACK
- DFS can be implemented easily with recursion which uses an implicit stack

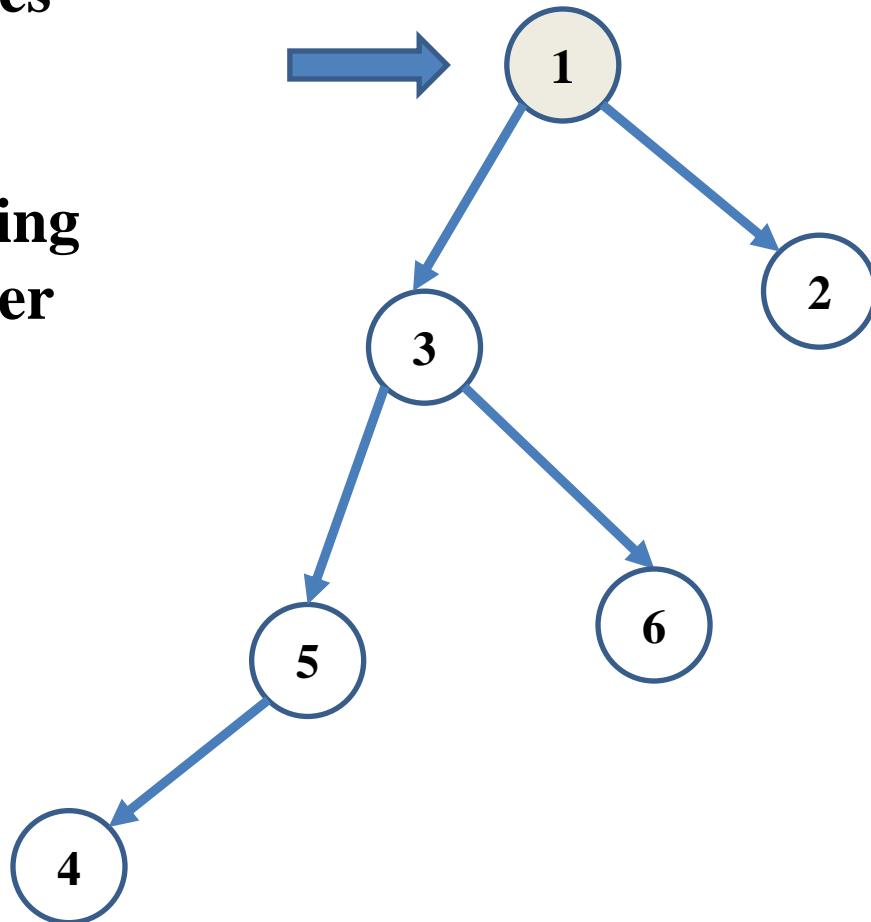
Depth First Search

DFS is a graph traversal algorithm



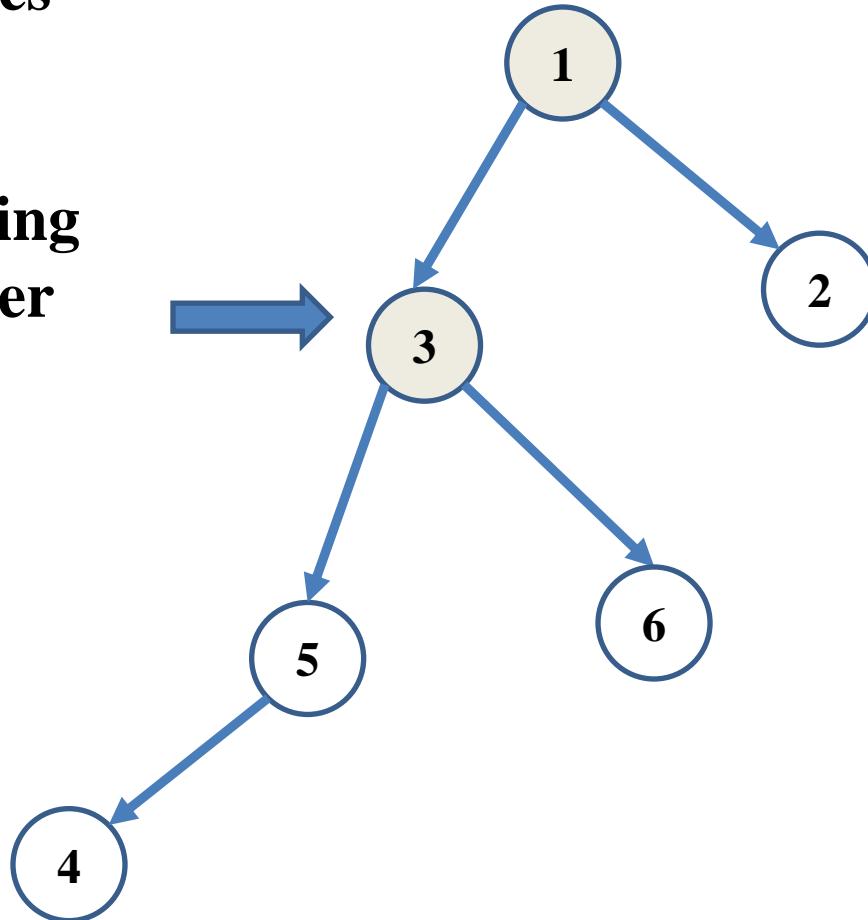
Depth First Search

**DFS explores
one path
completely
before moving
on to another**



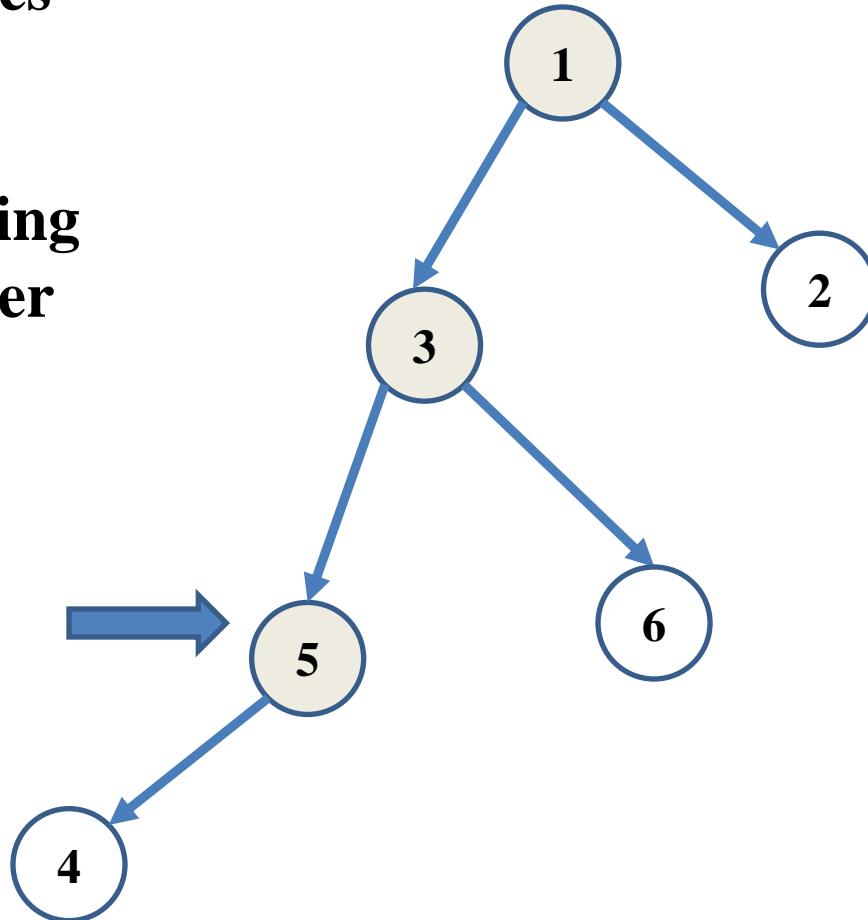
Depth First Search

**DFS explores
one path
completely
before moving
on to another**



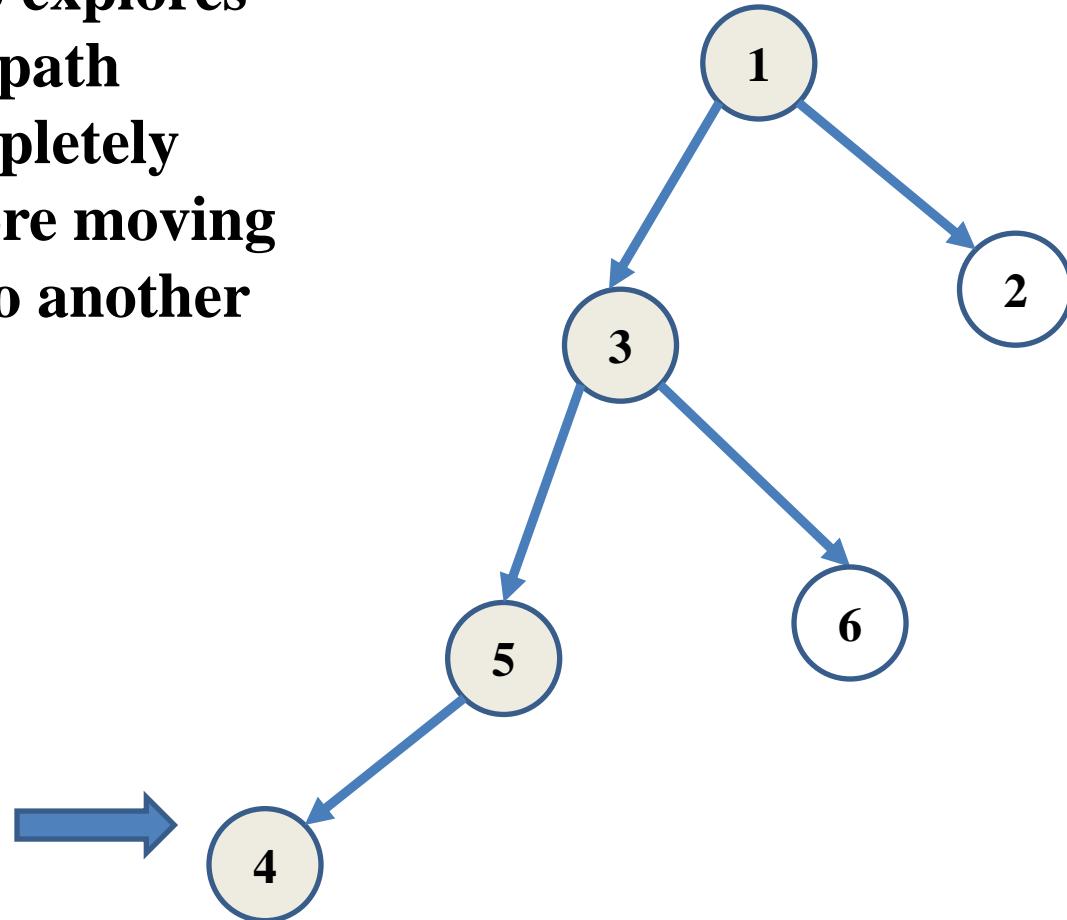
Depth First Search

**DFS explores
one path
completely
before moving
on to another**



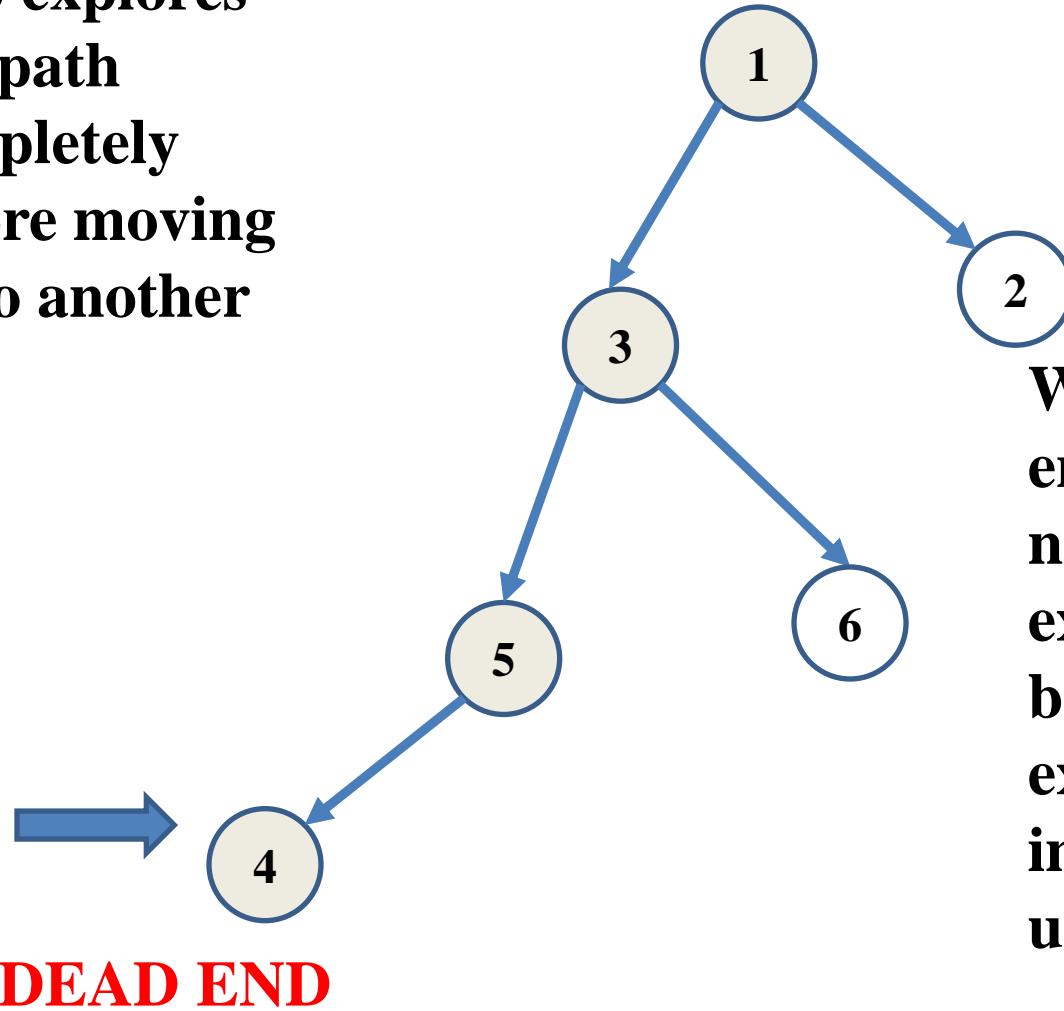
Depth First Search

**DFS explores
one path
completely
before moving
on to another**



Depth First Search

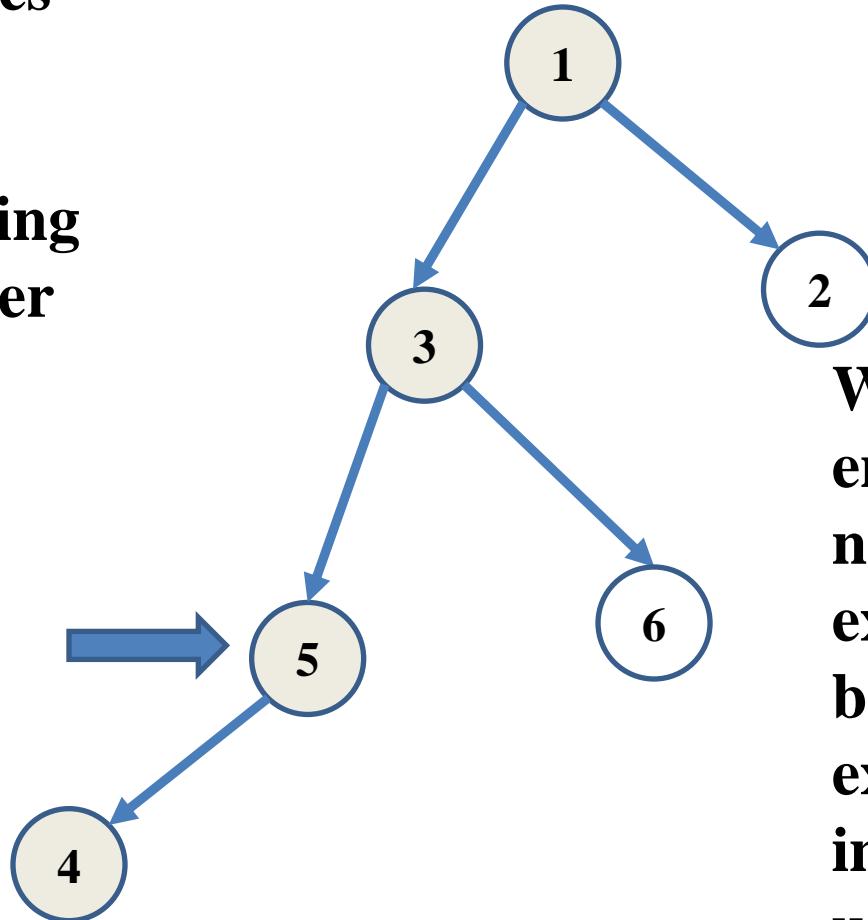
**DFS explores
one path
completely
before moving
on to another**



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

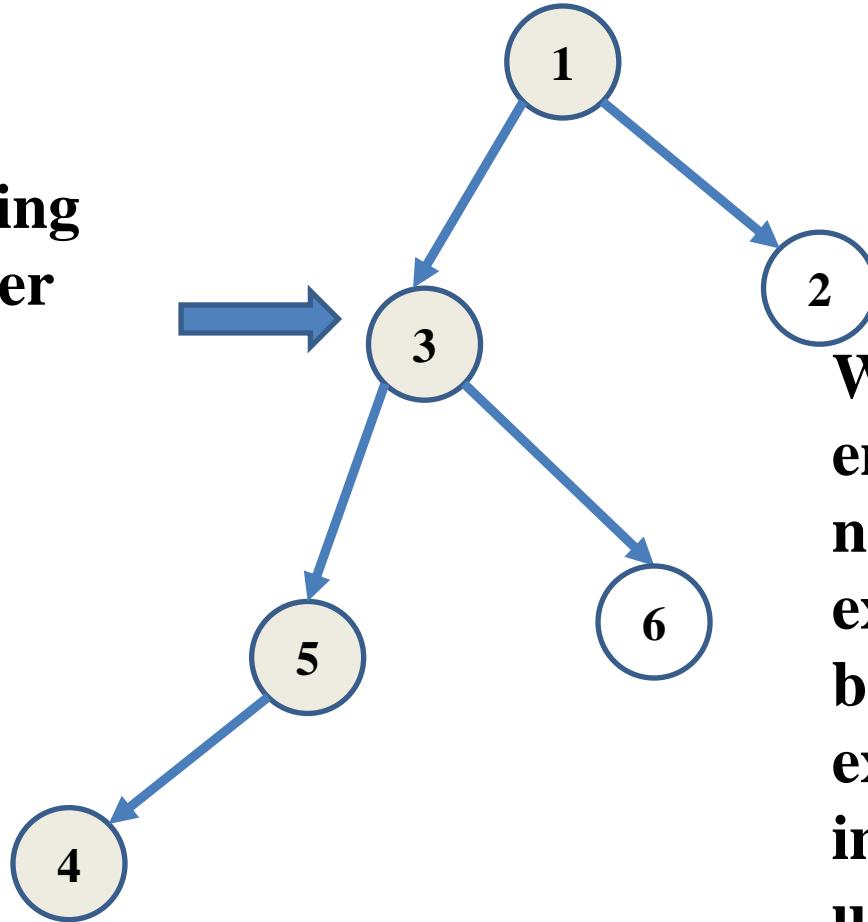
**DFS explores
one path
completely
before moving
on to another**



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

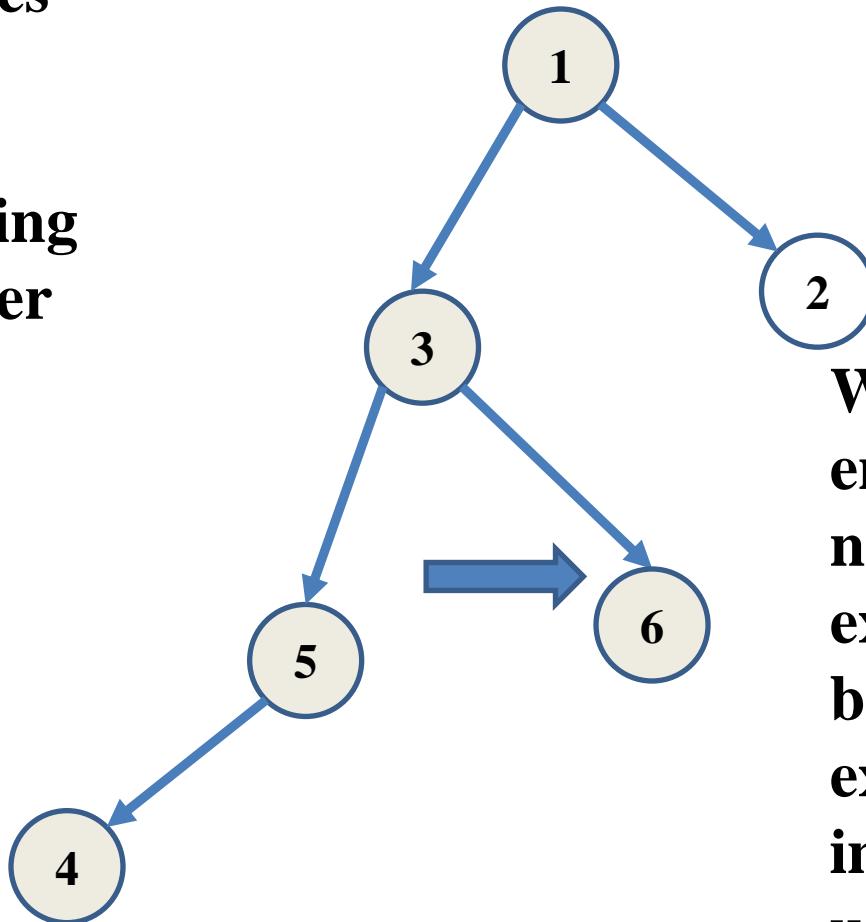
**DFS explores
one path
completely
before moving
on to another**



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

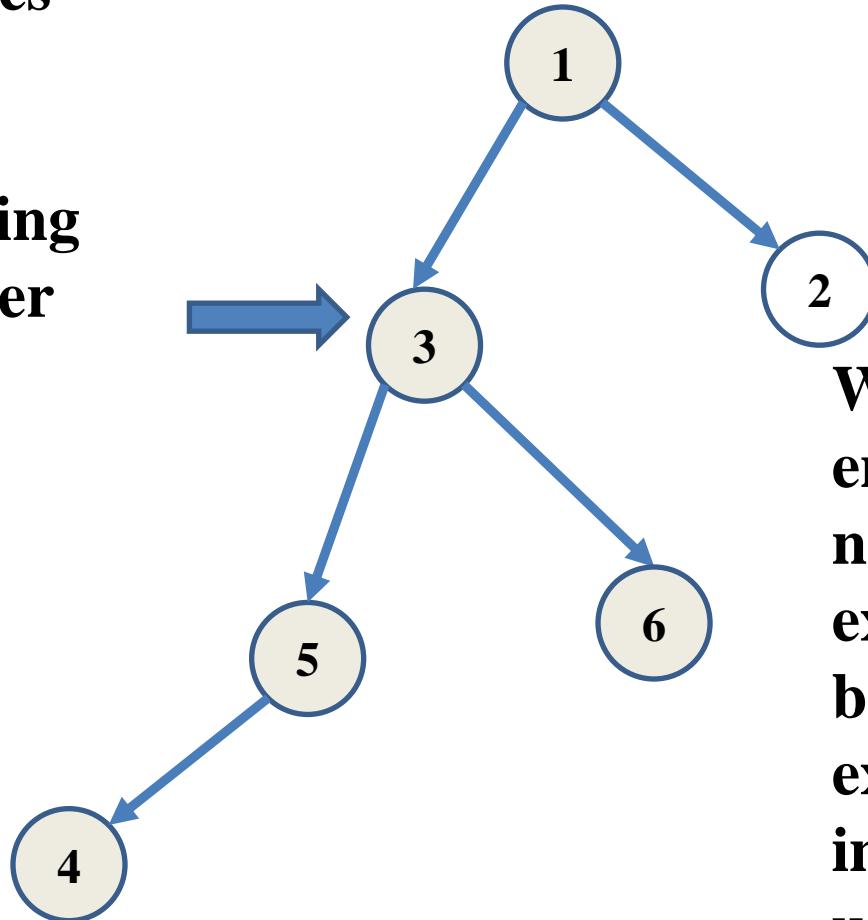
**DFS explores
one path
completely
before moving
on to another**



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

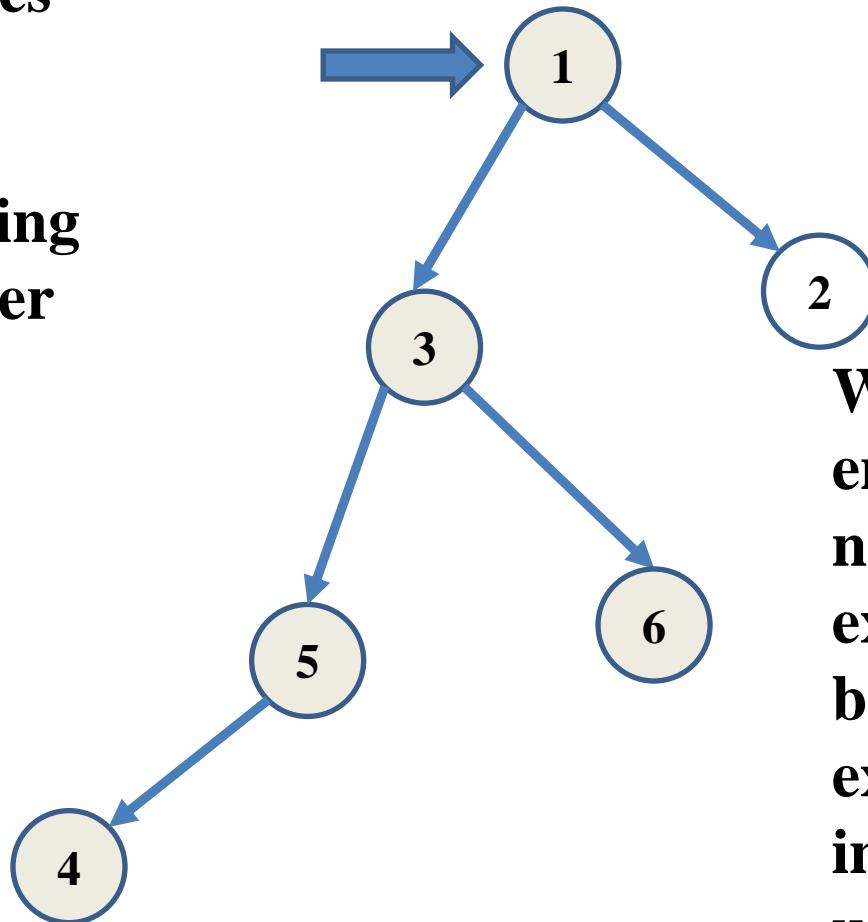
**DFS explores
one path
completely
before moving
on to another**



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

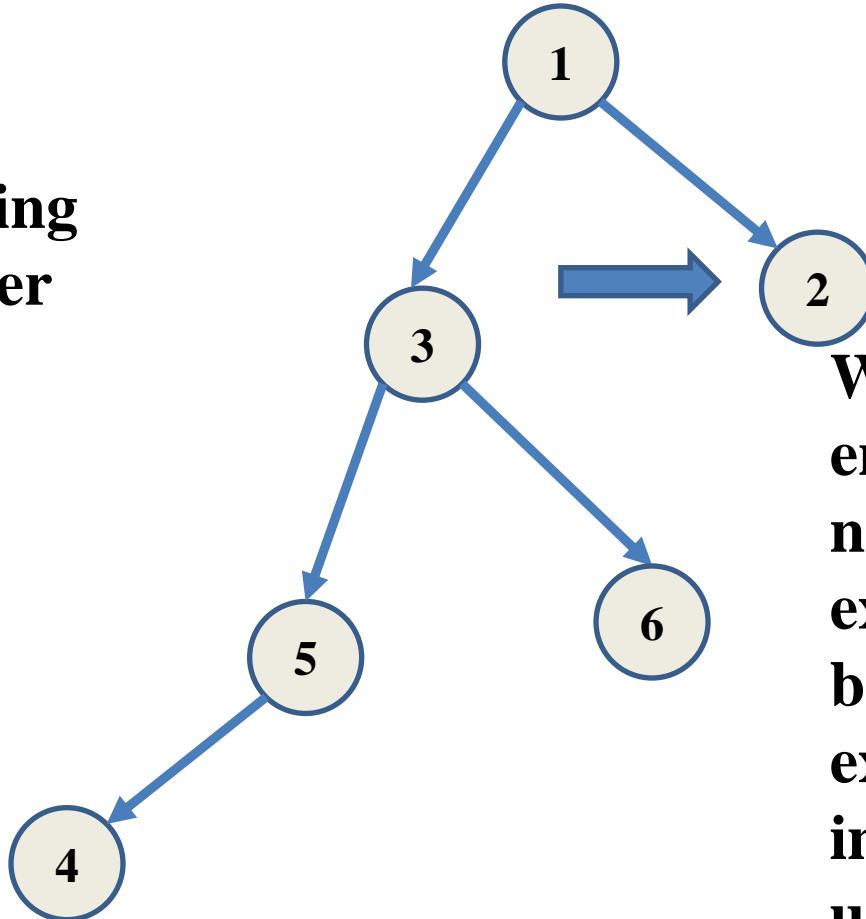
**DFS explores
one path
completely
before moving
on to another**



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Depth First Search

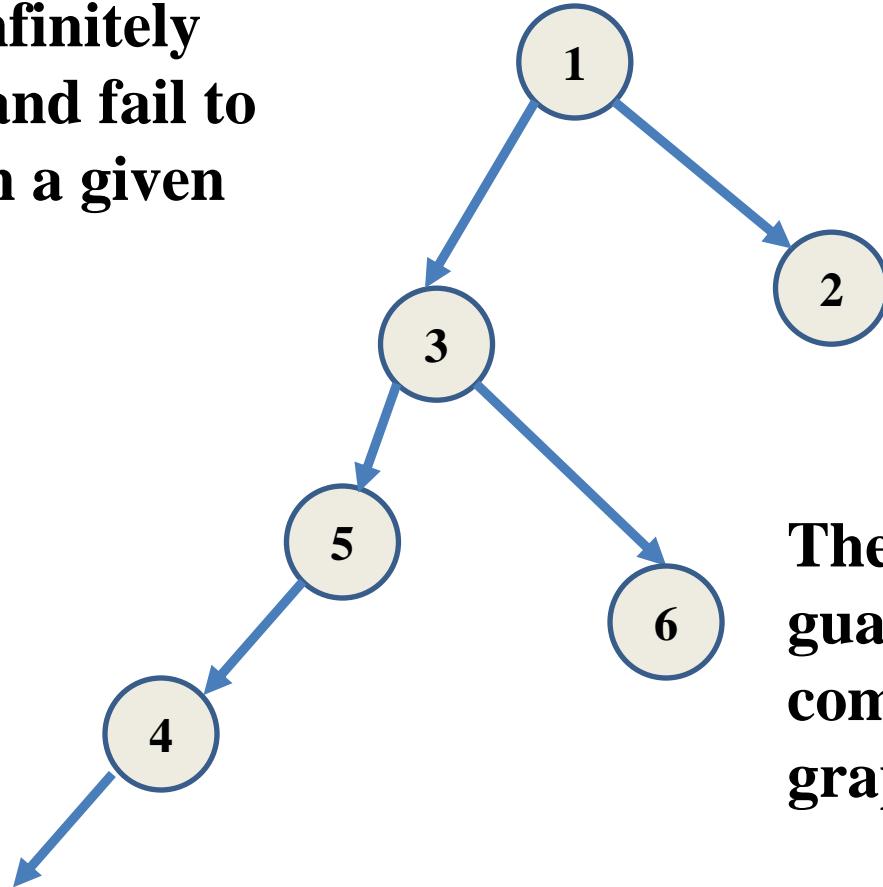
**DFS explores
one path
completely
before moving
on to another**



When DFS hits a dead end (no further neighbours to be explored), it backtracks and explores the immediately previous unexplored path

Completeness of DFS

DFS could get stuck exploring infinitely long paths and fail to terminate in a given time

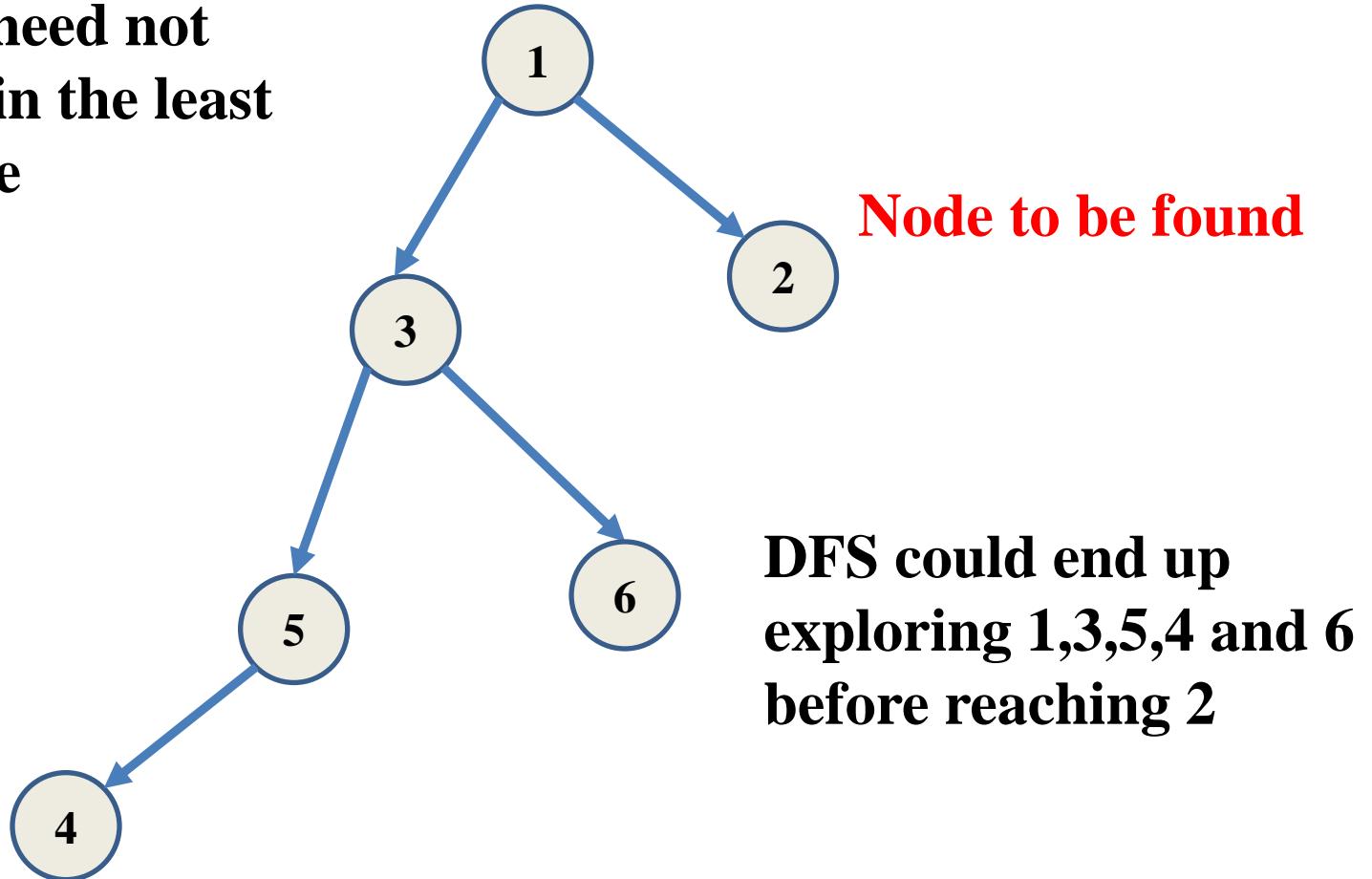


Potentially continues to ∞

The DFS algorithm is guaranteed to be complete for finite graphs

Optimality of DFS

DFS algorithm is not optimal - it need not find a node in the least possible time



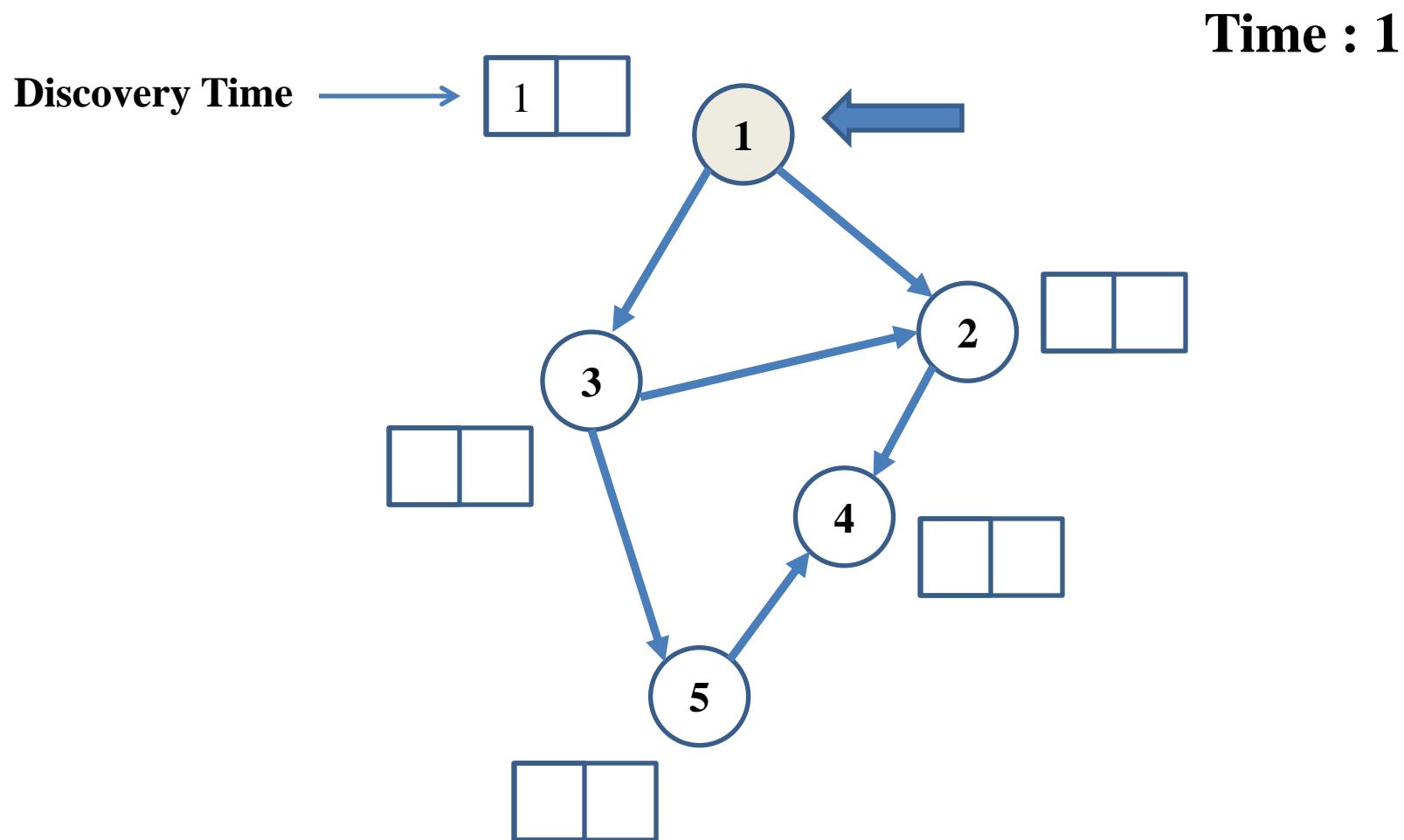
Applications of DFS

- Cycle detection
- Path Detection
- Finding strongly connected components
- Job Scheduling with dependencies (Topological Sort)

Associated Notations

- **Nodes are assigned colours**
 - WHITE : The node has not been visited
 - GRAY : The node has been visited, but all of its branches have not been visited completely
 - BLACK : A node and its branches have been explored completely
- Often, two other values are associated with a node
 - **Discovery Time (d)** : The time at which the node becomes gray
 - **Finishing Time (f)** : The time at which the node becomes black
- **Predecessor Subgraph** : Each time a node is visited, its parent is noted to construct the **DFS tree / forest**

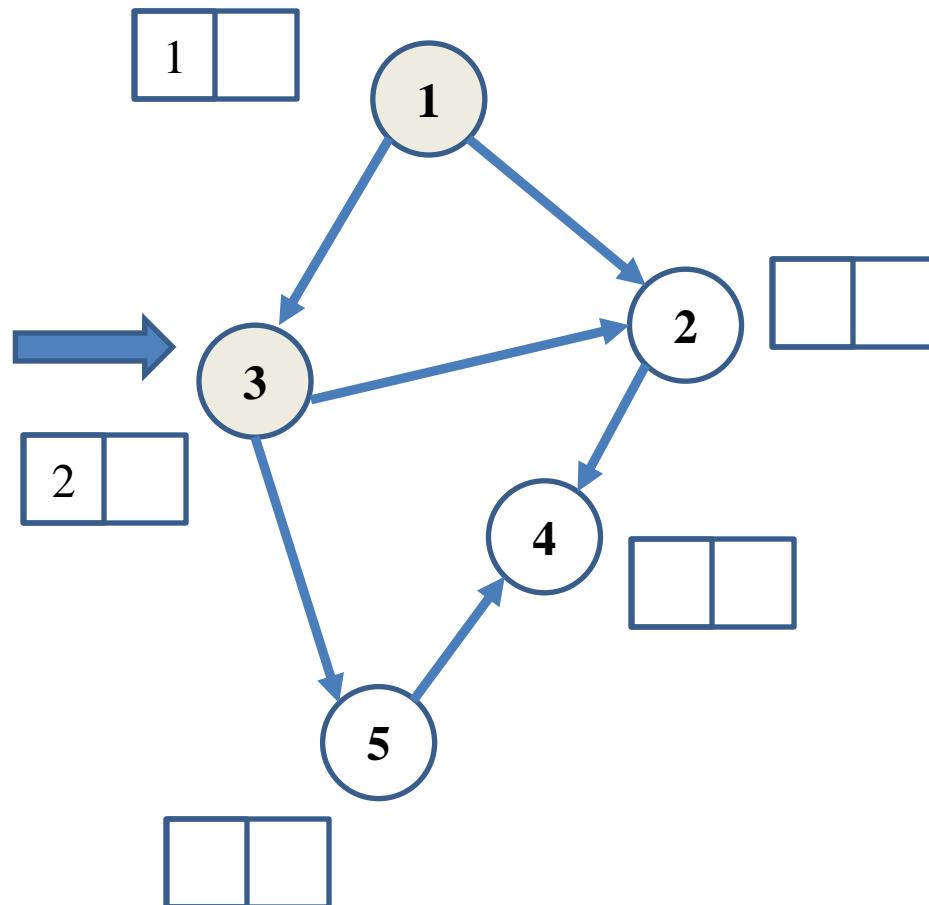
DFS in Action



DFS in Action

$\Pi(3) = 1$

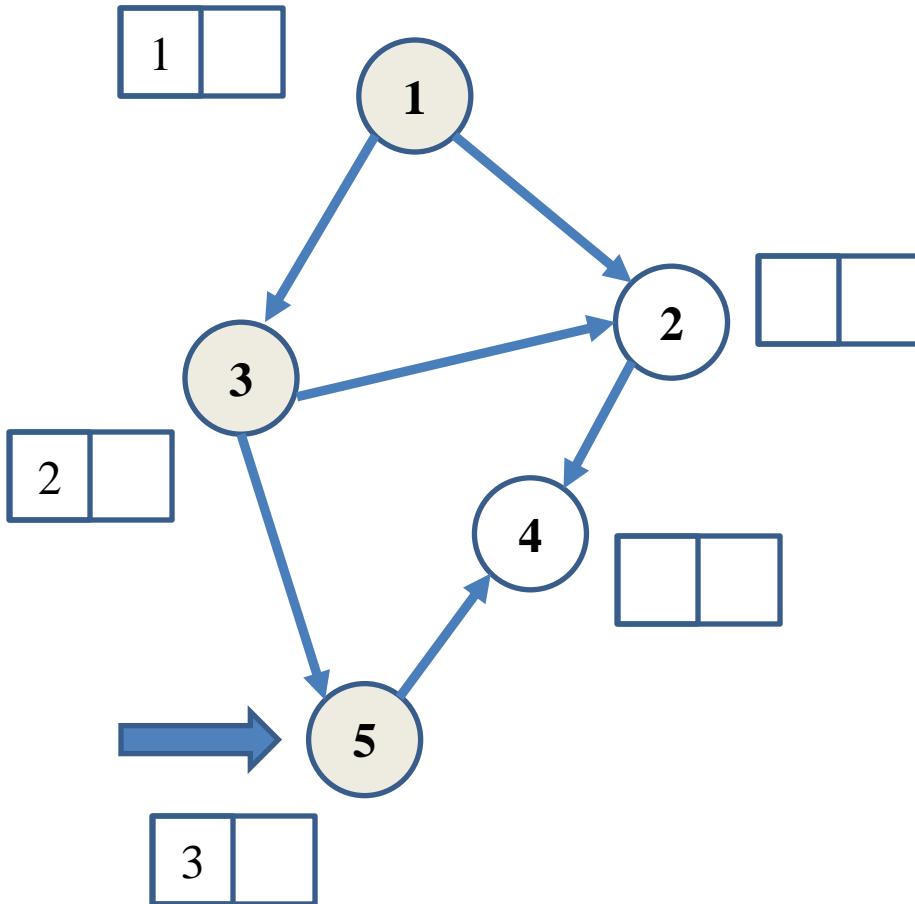
Time : 2



DFS in Action

$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3\end{aligned}$$

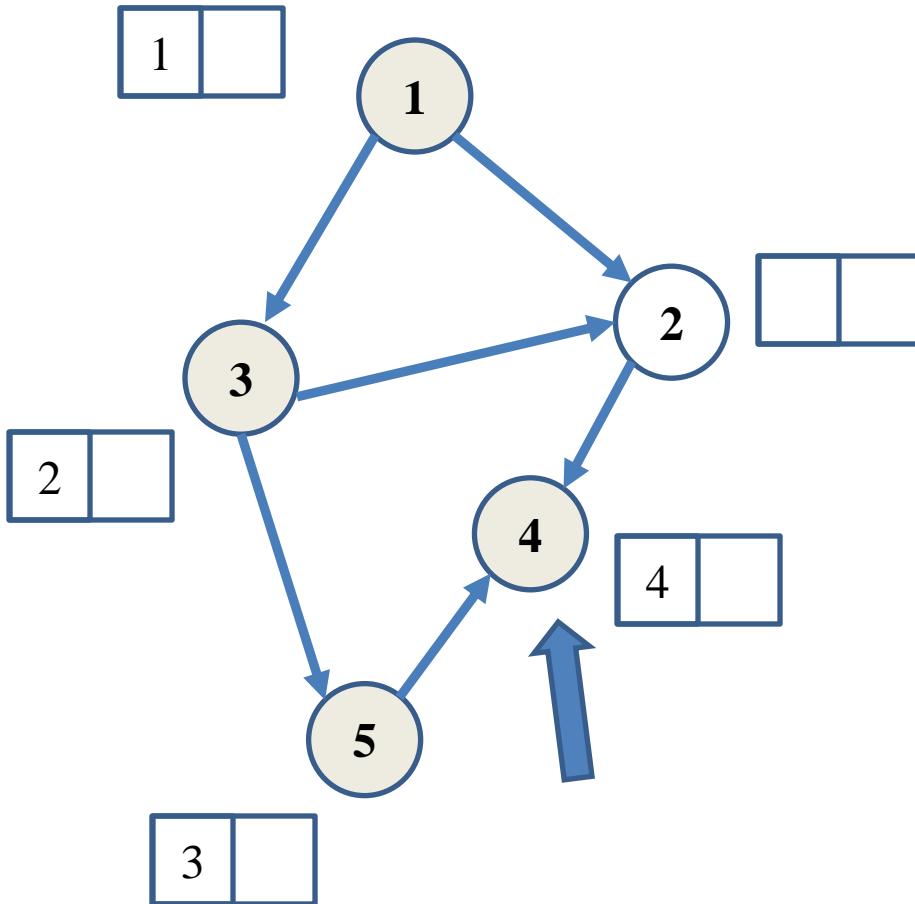
Time : 3



DFS in Action

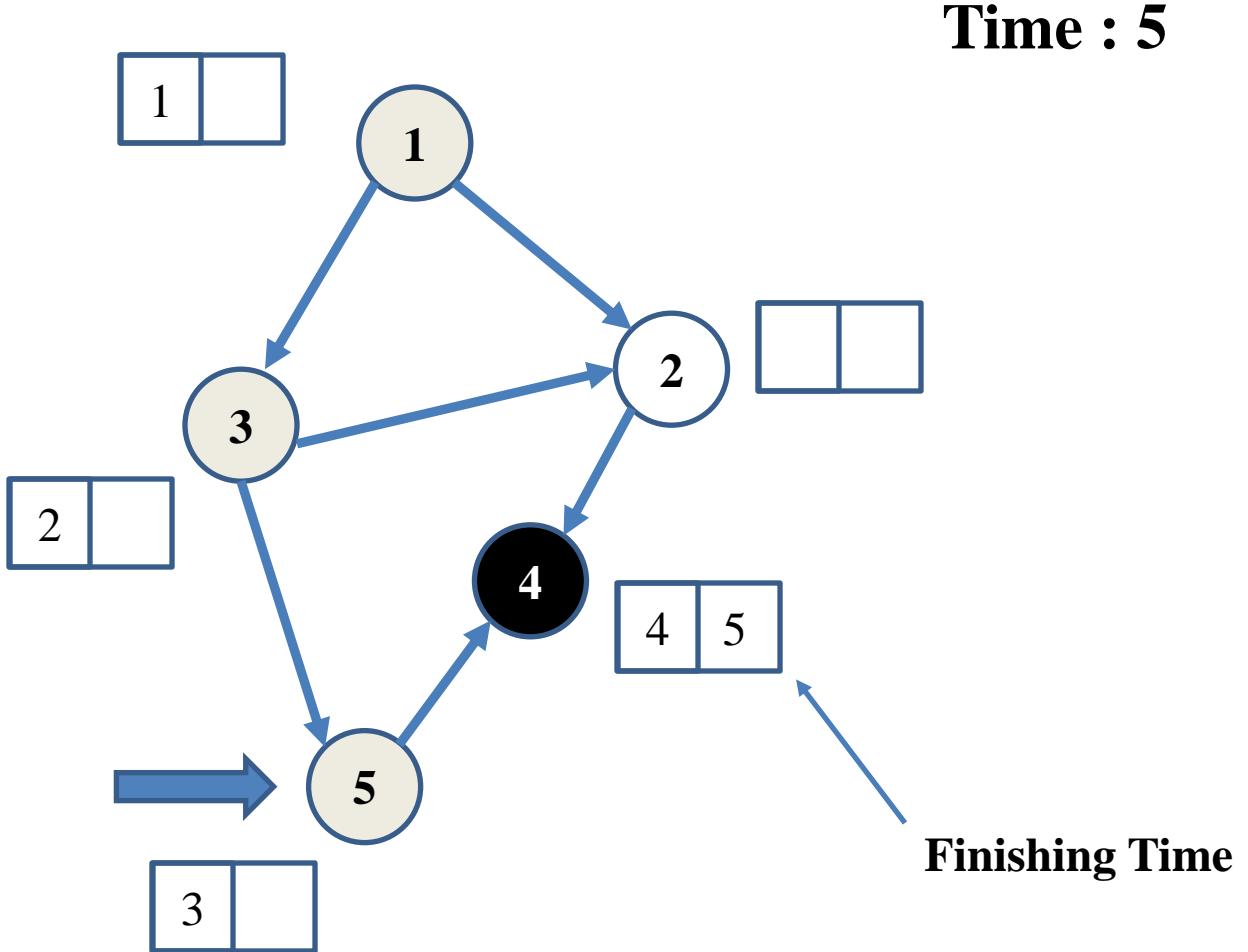
Time : 4

$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3 \\ \Pi(4) &= 5\end{aligned}$$



DFS in Action

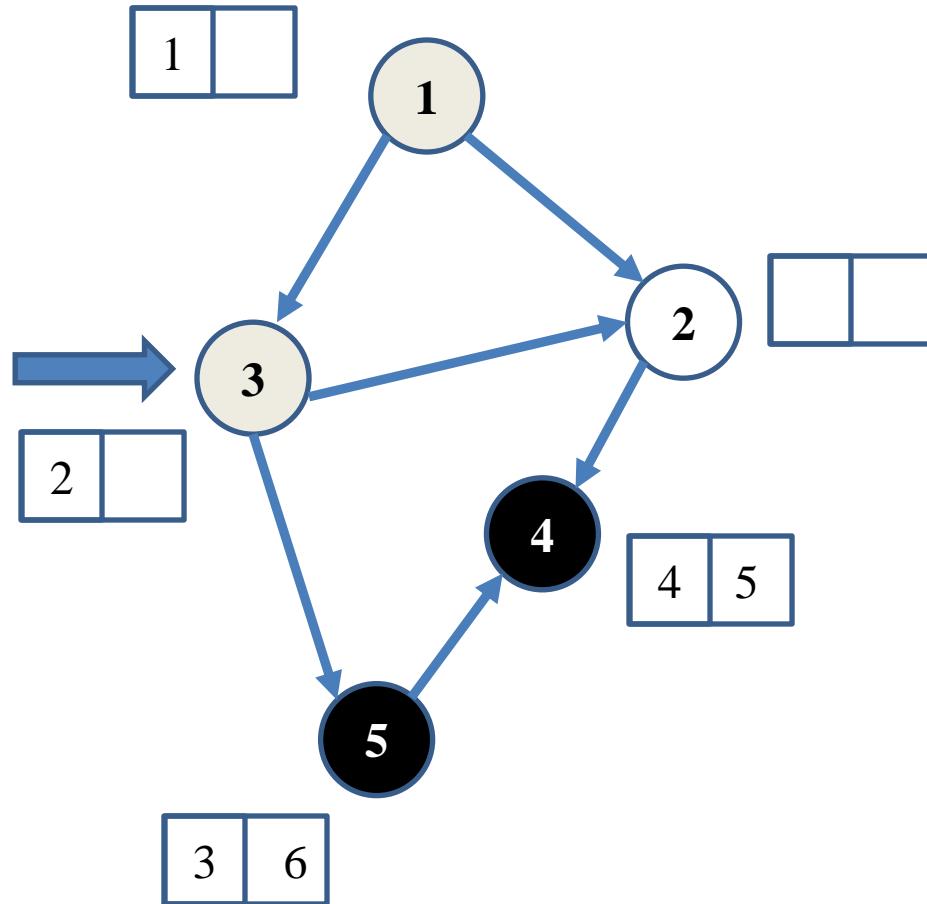
$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3 \\ \Pi(4) &= 5\end{aligned}$$



DFS in Action

Time : 6

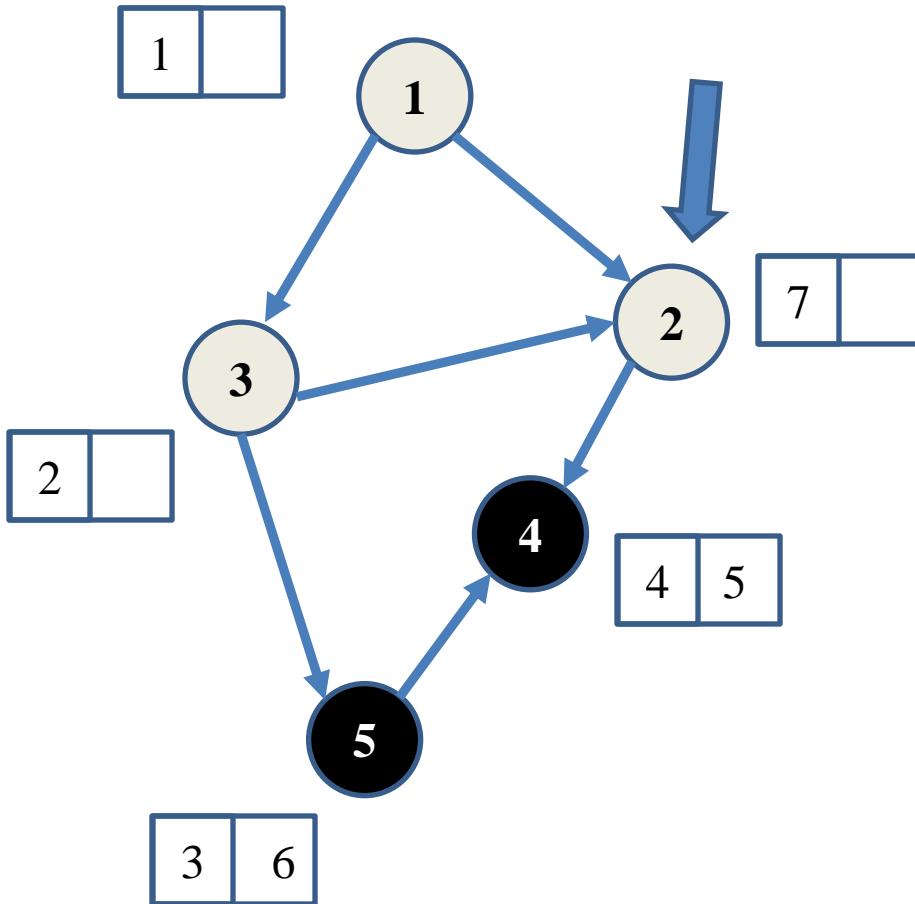
$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3 \\ \Pi(4) &= 5\end{aligned}$$



DFS in Action

Time : 7

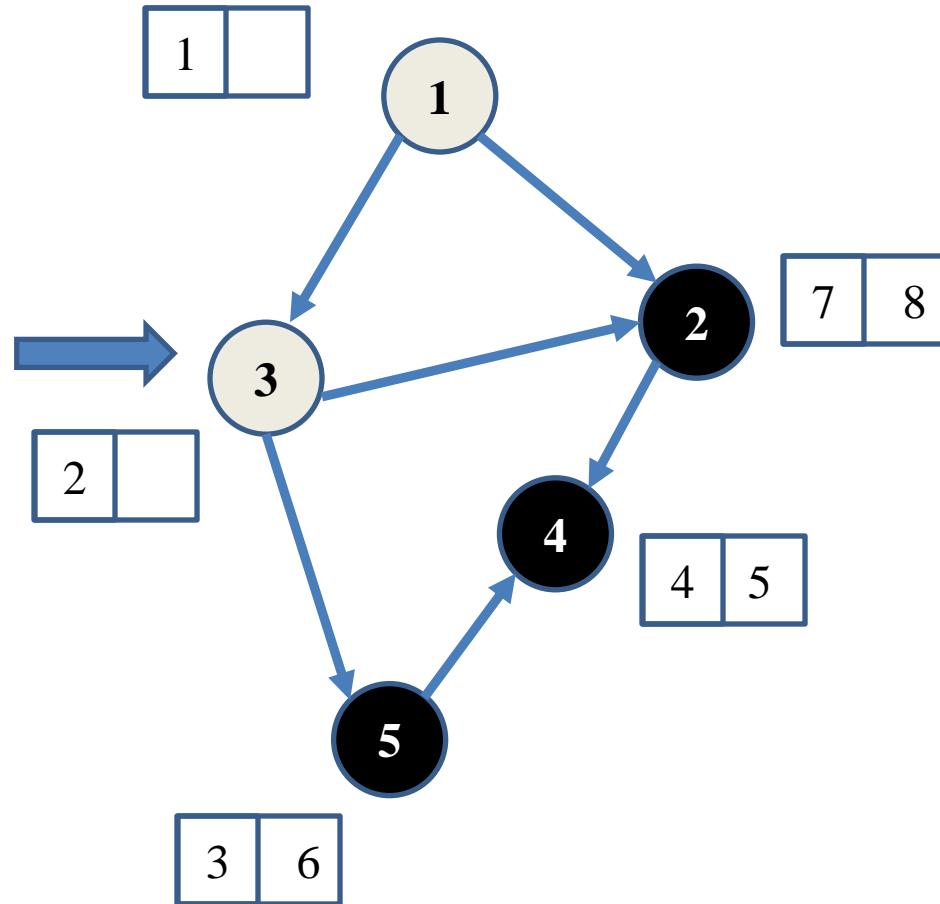
$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3 \\ \Pi(4) &= 5 \\ \Pi(2) &= 3\end{aligned}$$



DFS in Action

Time : 8

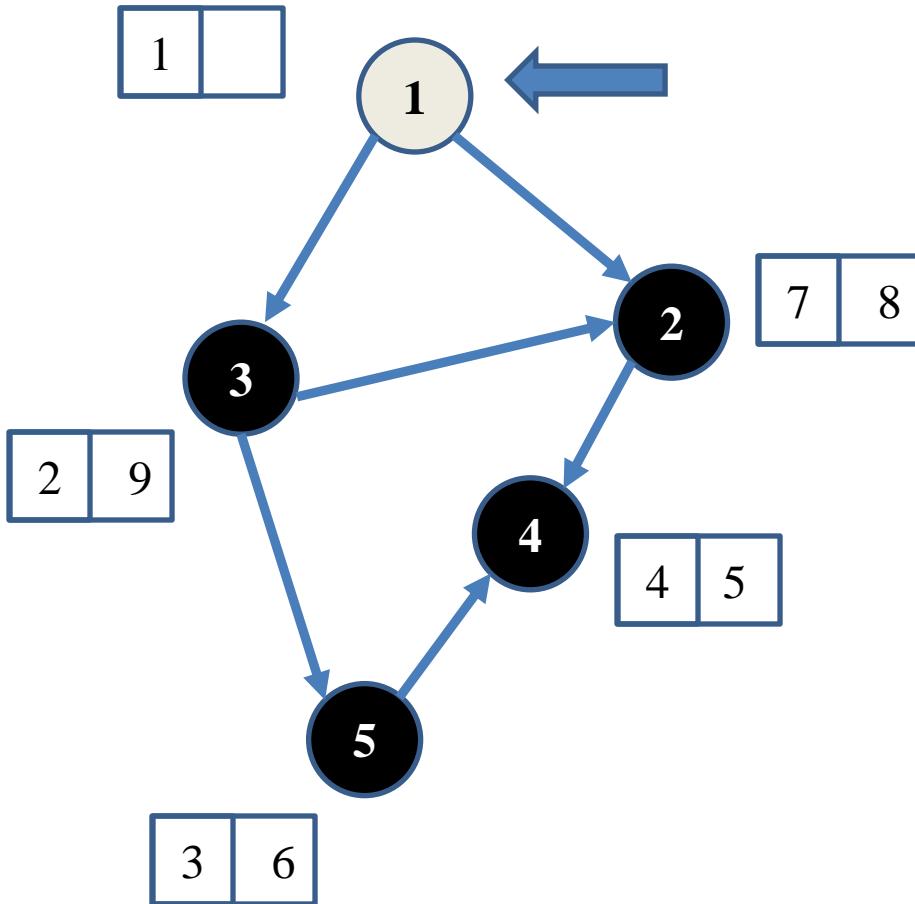
$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3 \\ \Pi(4) &= 5 \\ \Pi(2) &= 3\end{aligned}$$



DFS in Action

Time : 9

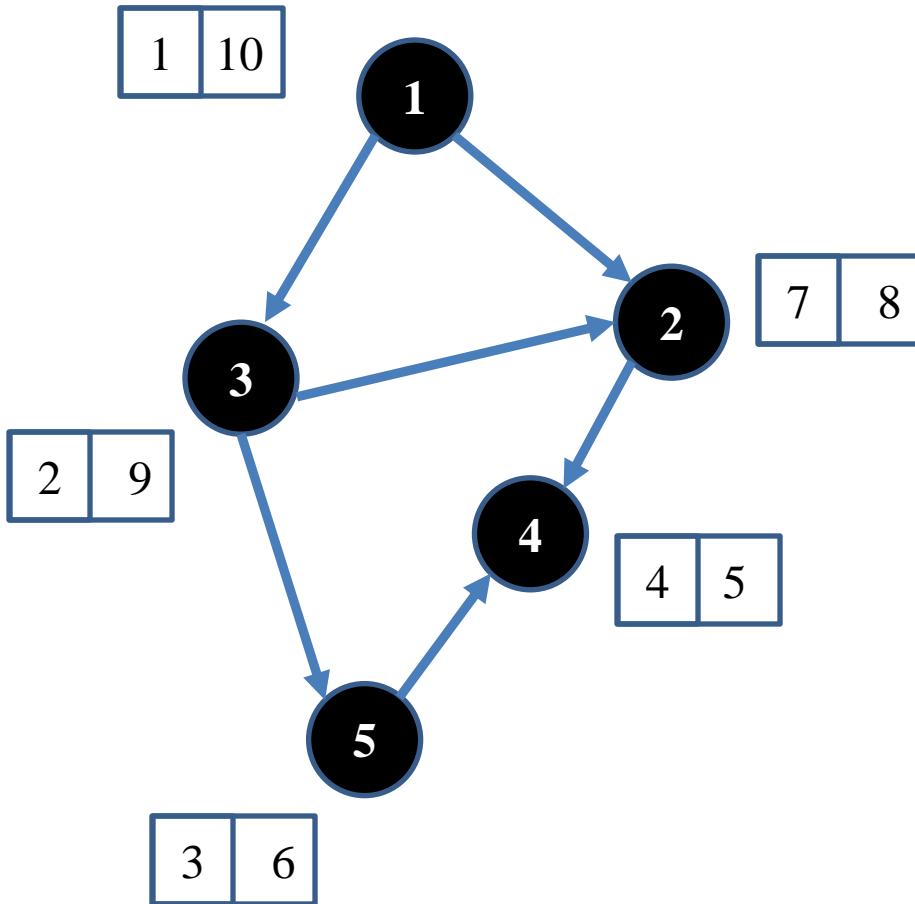
$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3 \\ \Pi(4) &= 5 \\ \Pi(2) &= 3\end{aligned}$$



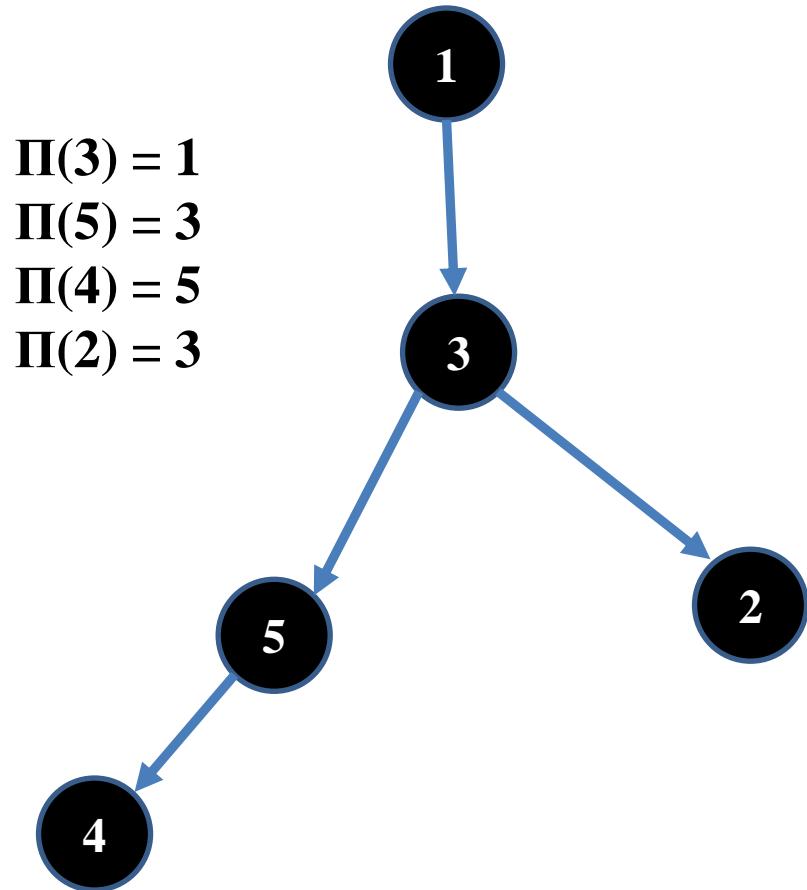
DFS in Action

Time : 10

$$\begin{aligned}\Pi(3) &= 1 \\ \Pi(5) &= 3 \\ \Pi(4) &= 5 \\ \Pi(2) &= 3\end{aligned}$$

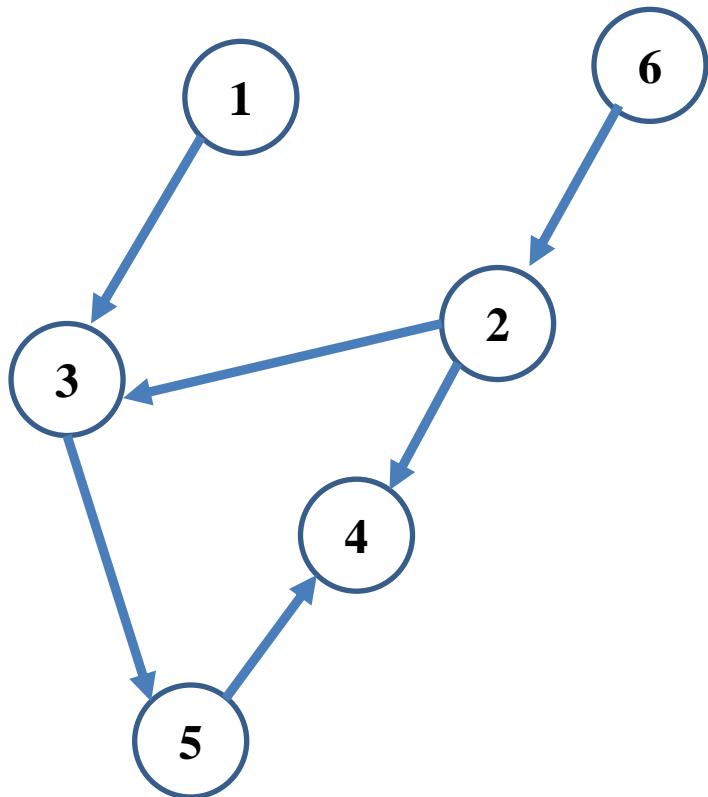


DFS Tree



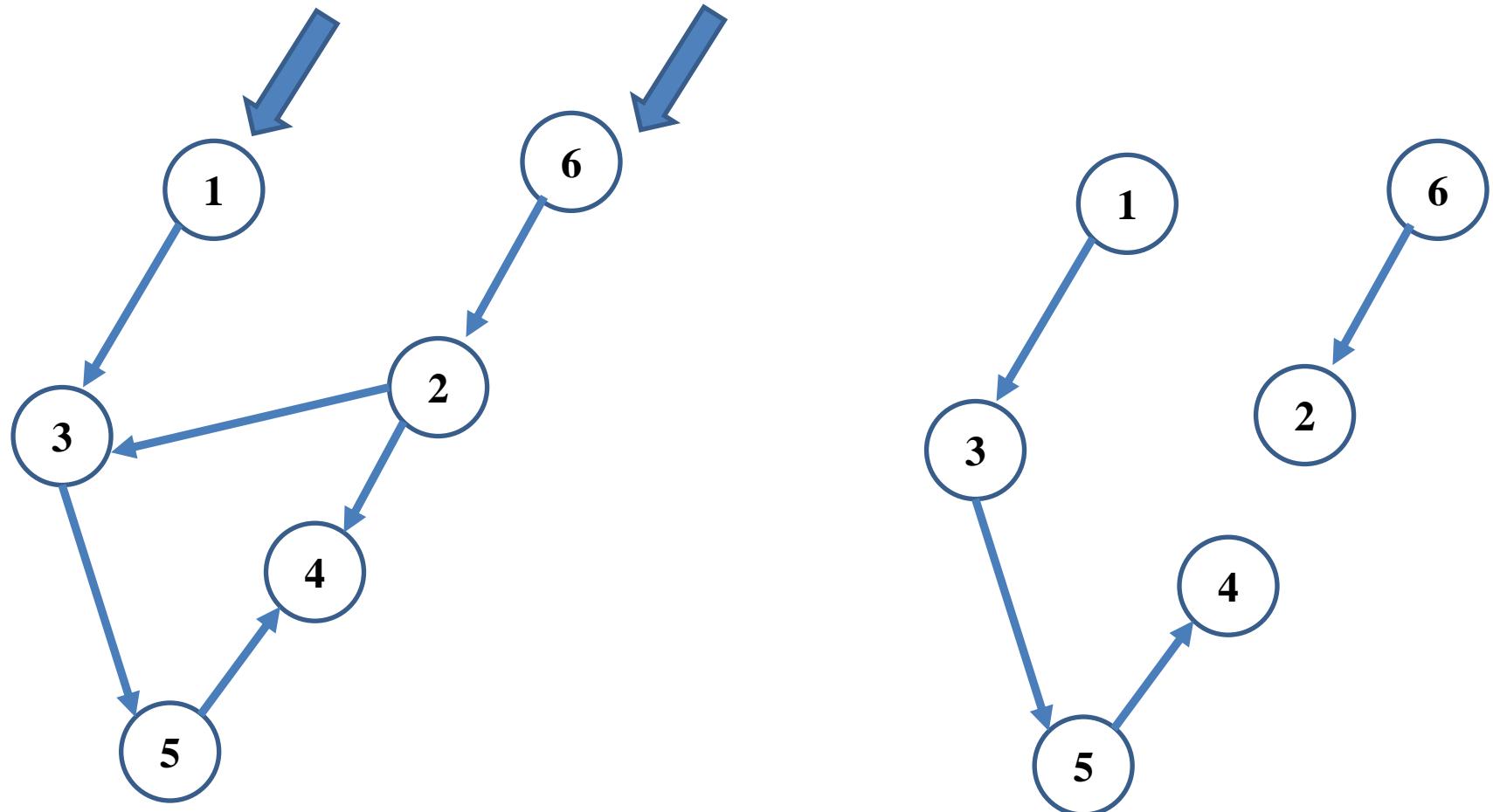
- DFS need not always form a single tree for a graph
- Sometimes, DFS may need to be performed multiple times from different starting nodes to discover all the nodes in the graph
- This gives rise to a number of DFS trees, collectively called a **DFS Forest**

Need for Multiple DFS Runs



- Nodes 2 and 6 are not reachable from 1
- Node 1 is not reachable from 6
- There is no starting node such that all the vertices in the graph are reachable
- We need to run DFS from at least two vertices – say 1 and 6

DFS Forest



If DFS is run from node 1 and then from node 6, we get two DFS trees (a DFS forest)

DFS Algorithm

DFS(G)

for each vertex u in V[G]

colour[u] = WHITE

Π[u]= NIL

time=0

for each vertex u in V[G]

if colour[u]= WHITE

DFS_VISIT(u)

DFS_VISIT Algorithm

DFS_VISIT(u)

colour[u] = GRAY

time = time + 1

d[u] = time

for each v in Adj[u]

if colour[v] = WHITE

$\Pi[v] = u$

DFS_VISIT(v)

colour[u] = BLACK

time = time + 1

f[u] = time

Time Complexity of DFS

- Every node is explored EXACTLY ONCE --- $\Theta(|V|)$
- For every node u , DFS explores all the edges in $\text{Adj}[u]$
- When summed over all the nodes in the graph, this amounts to the number of edges in the graph

$$\sum_{u \in V} \text{Adj}[u] = \Theta(|E|)$$

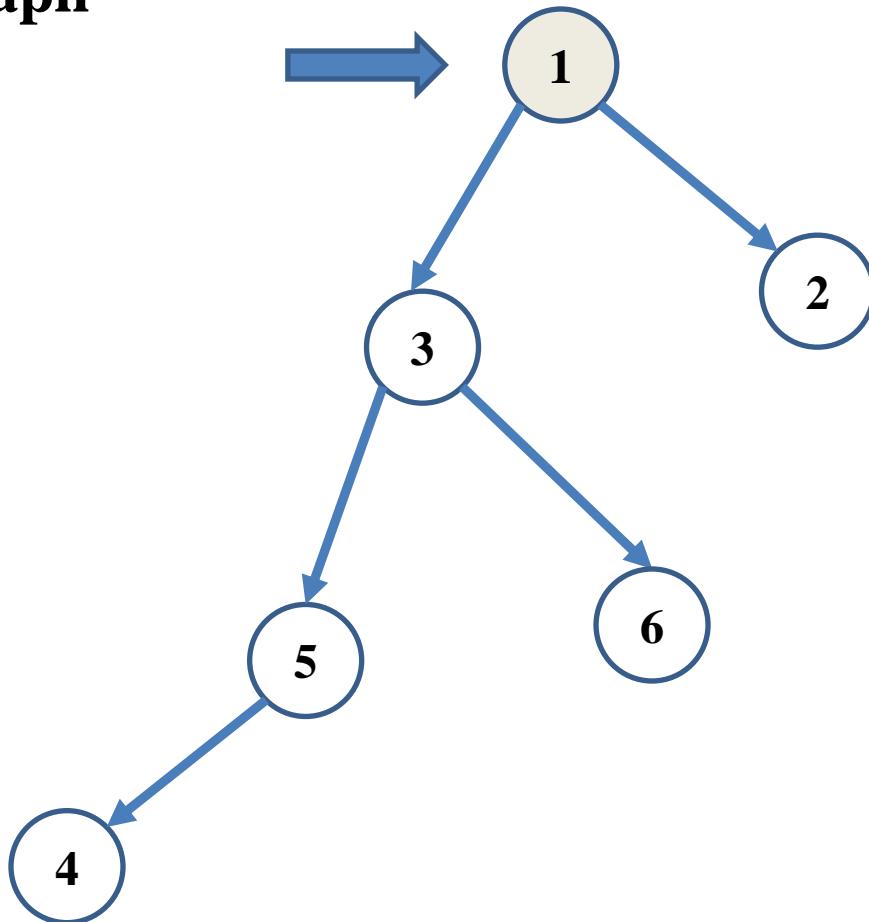
- Thus, **total complexity of DFS is $\Theta(|V| + |E|)$**

Breadth First Search

- Breadth First Search (BFS) is another graph traversal algorithm
- As the name suggests, BFS explores all the neighbours of a node before exploring any other node
- When BFS hits a dead end on a path, it backtracks and starts exploring a new path from the previous node
- This behaviour is suggestive of a FIFO data structure
 - QUEUE

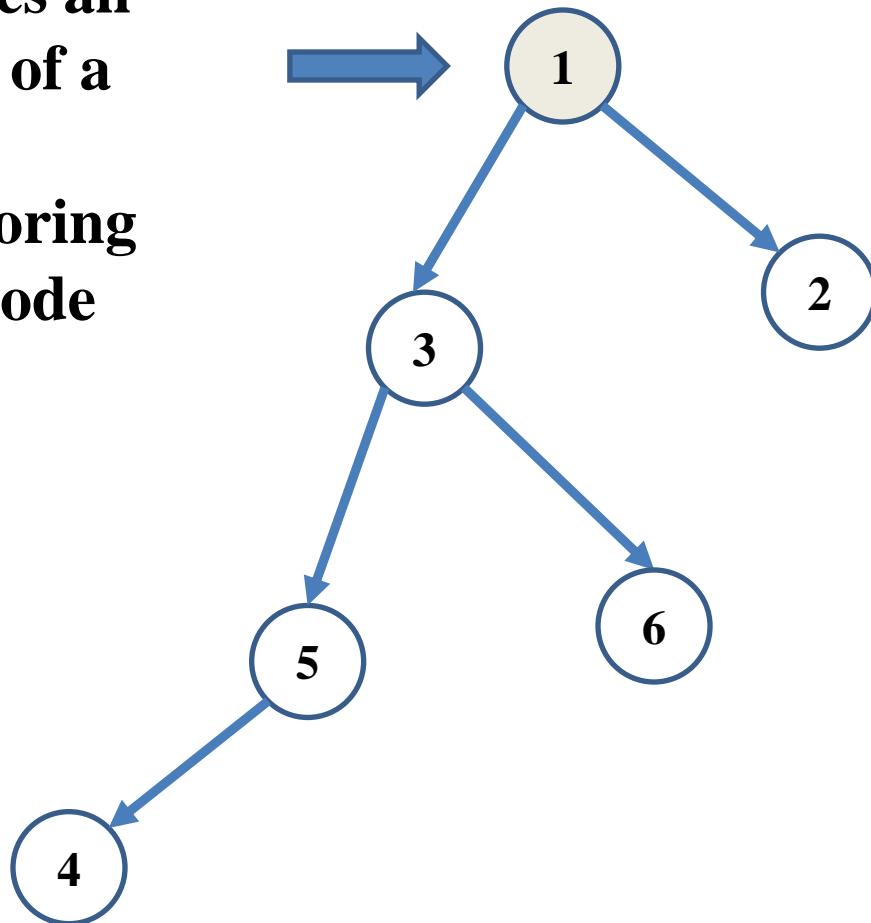
Breadth First Search

BFS is a graph traversal algorithm



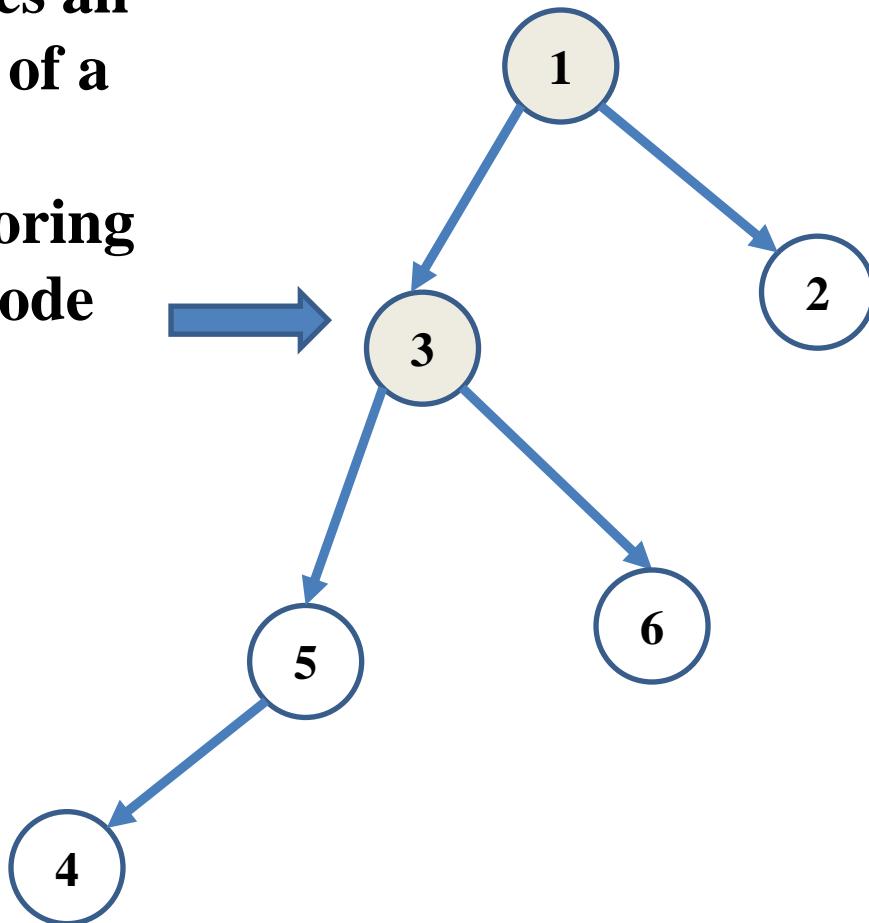
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



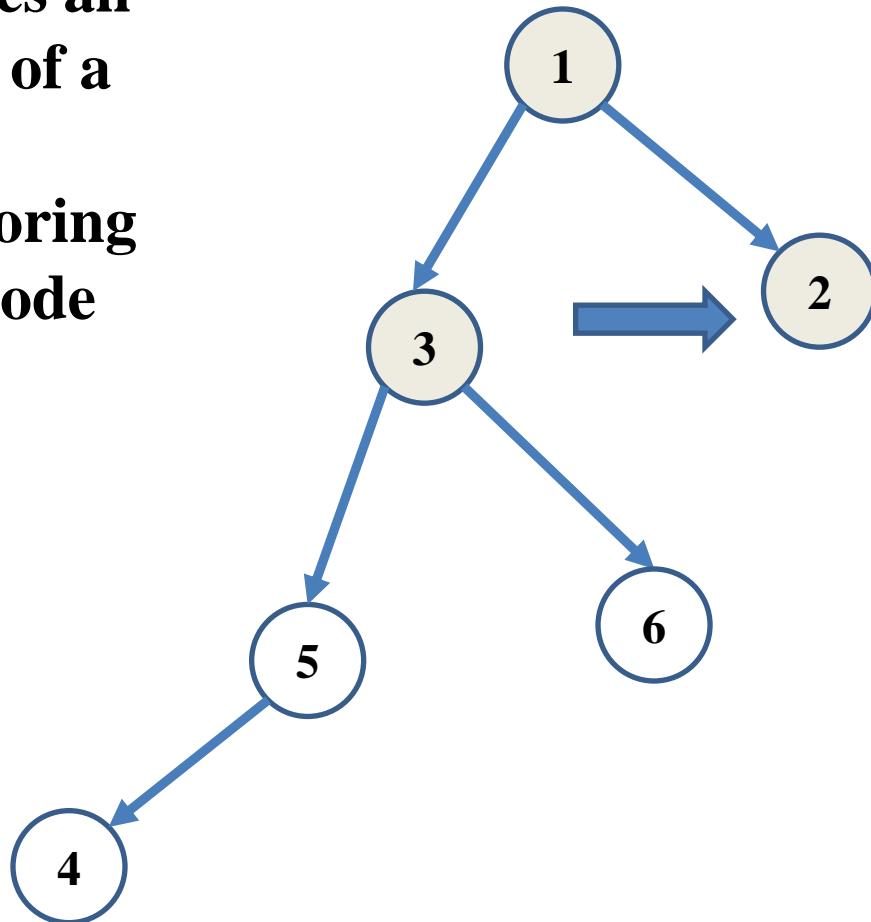
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



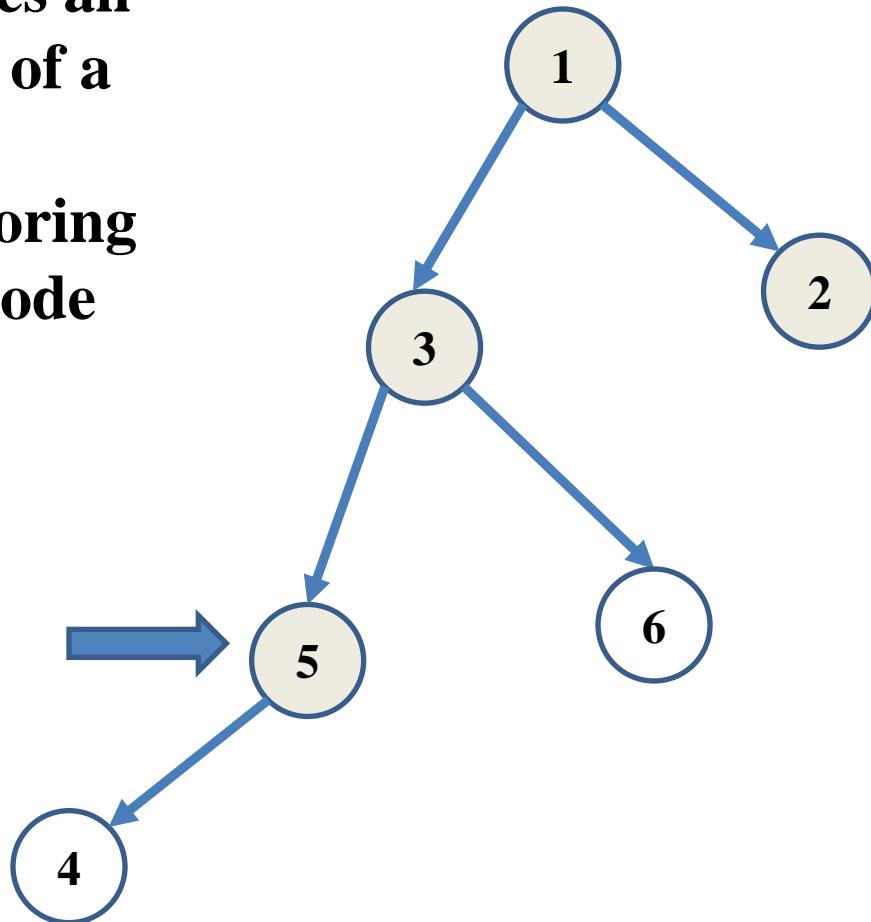
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



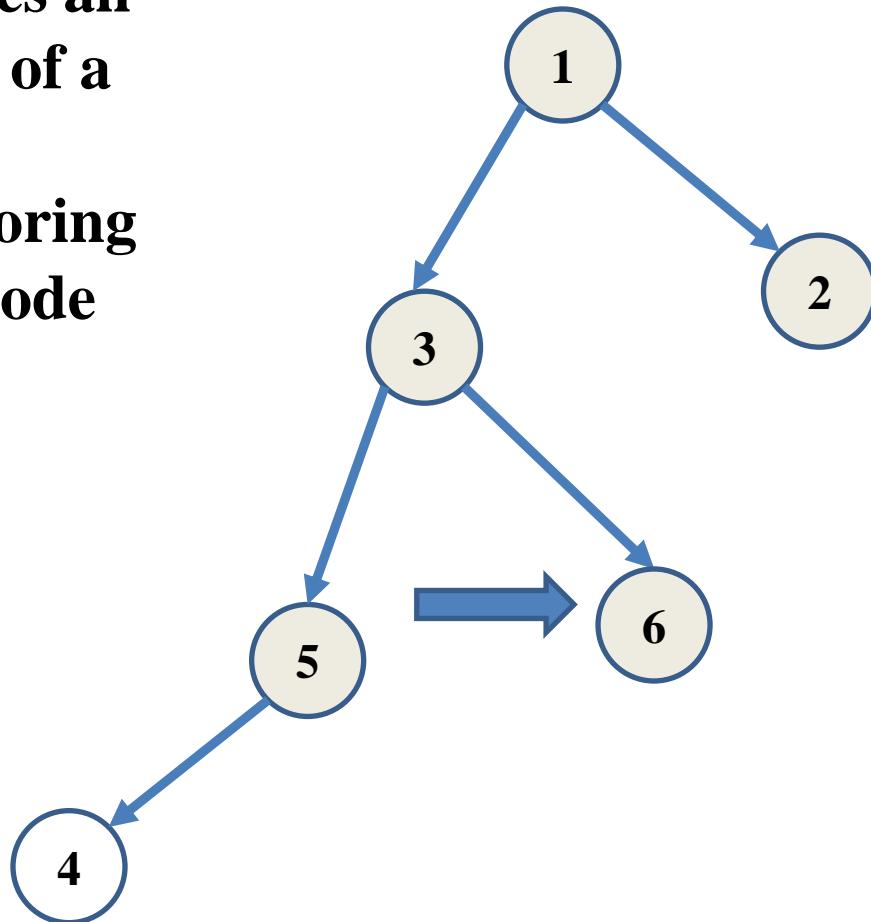
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



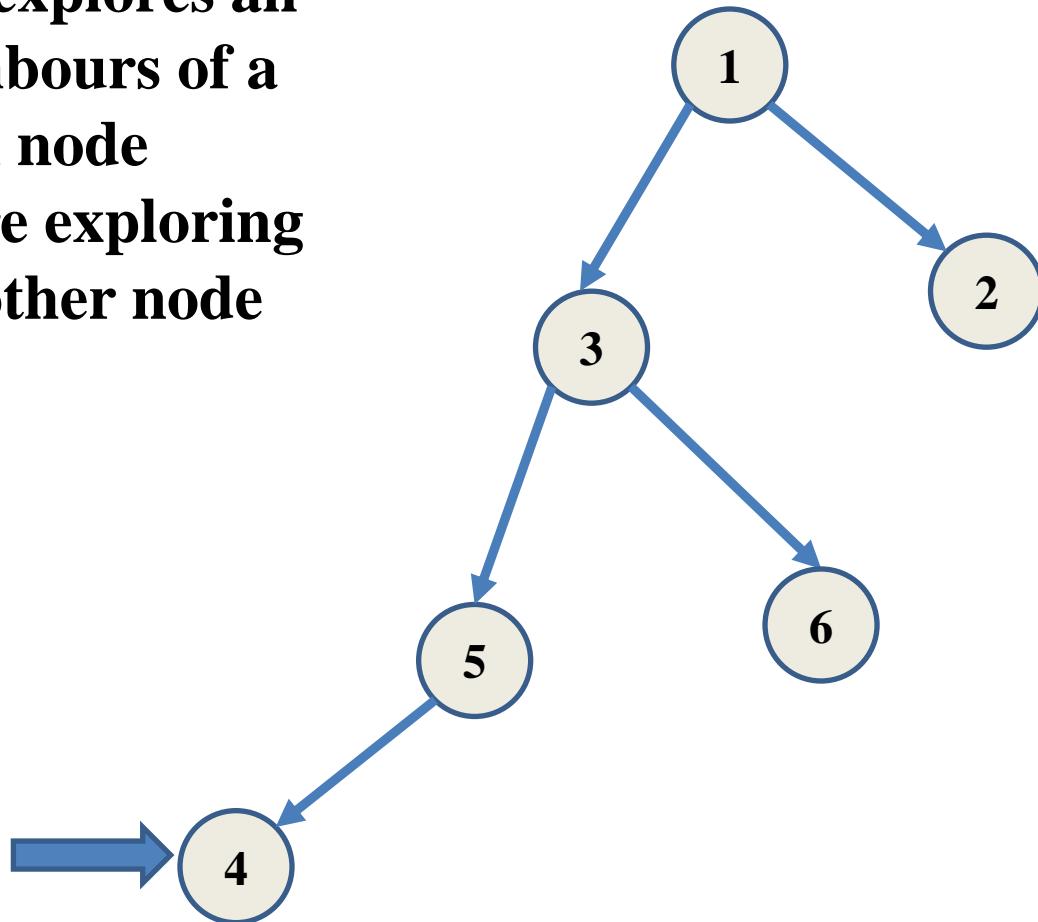
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



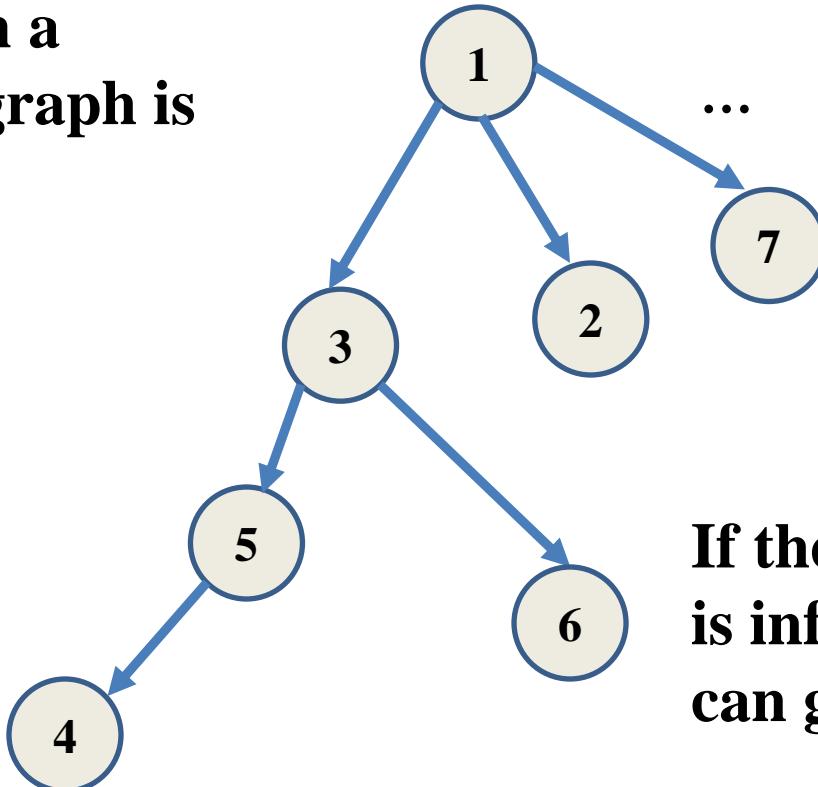
Breadth First Search

BFS explores all neighbours of a given node before exploring any other node



Completeness of BFS

BFS algorithm
terminates with a
solution if the graph is
finite



Potentially continues to ∞

If the branching factor
is infinite, then BFS
can get stuck in a loop

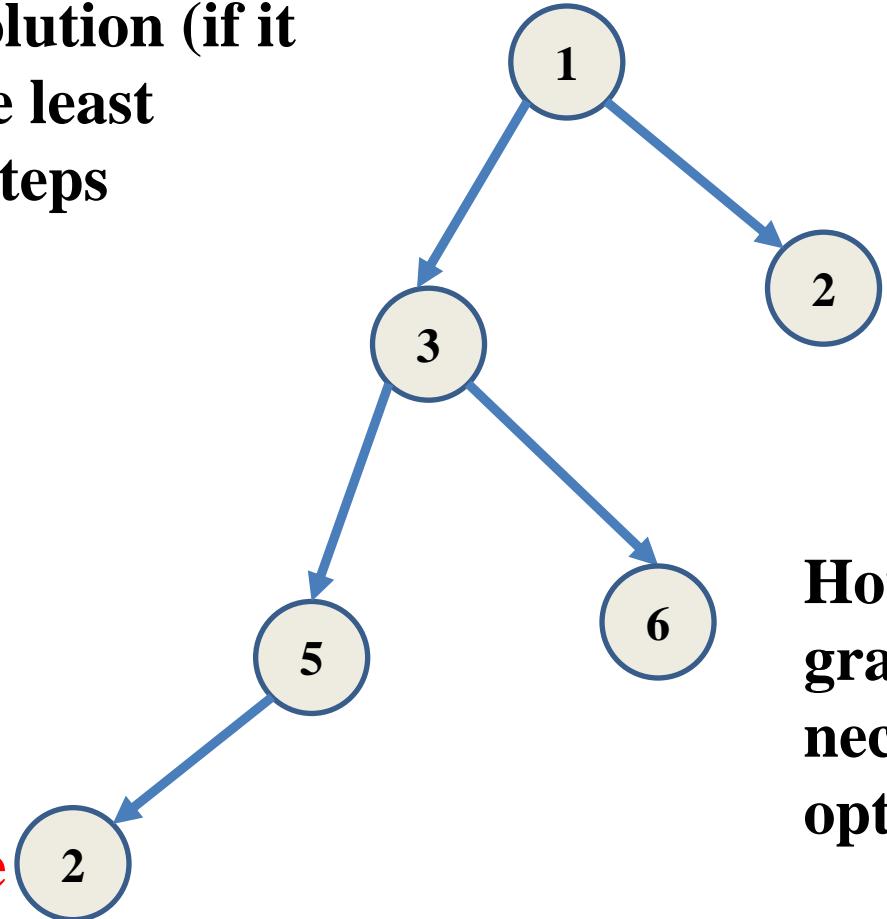
Optimality of BFS

BFS algorithm will
find the a solution (if it
exists) in the least
number of steps

Key = 2

Present here

Present here



However, in a weighted
graph, BFS need not
necessarily give an
optimal solution

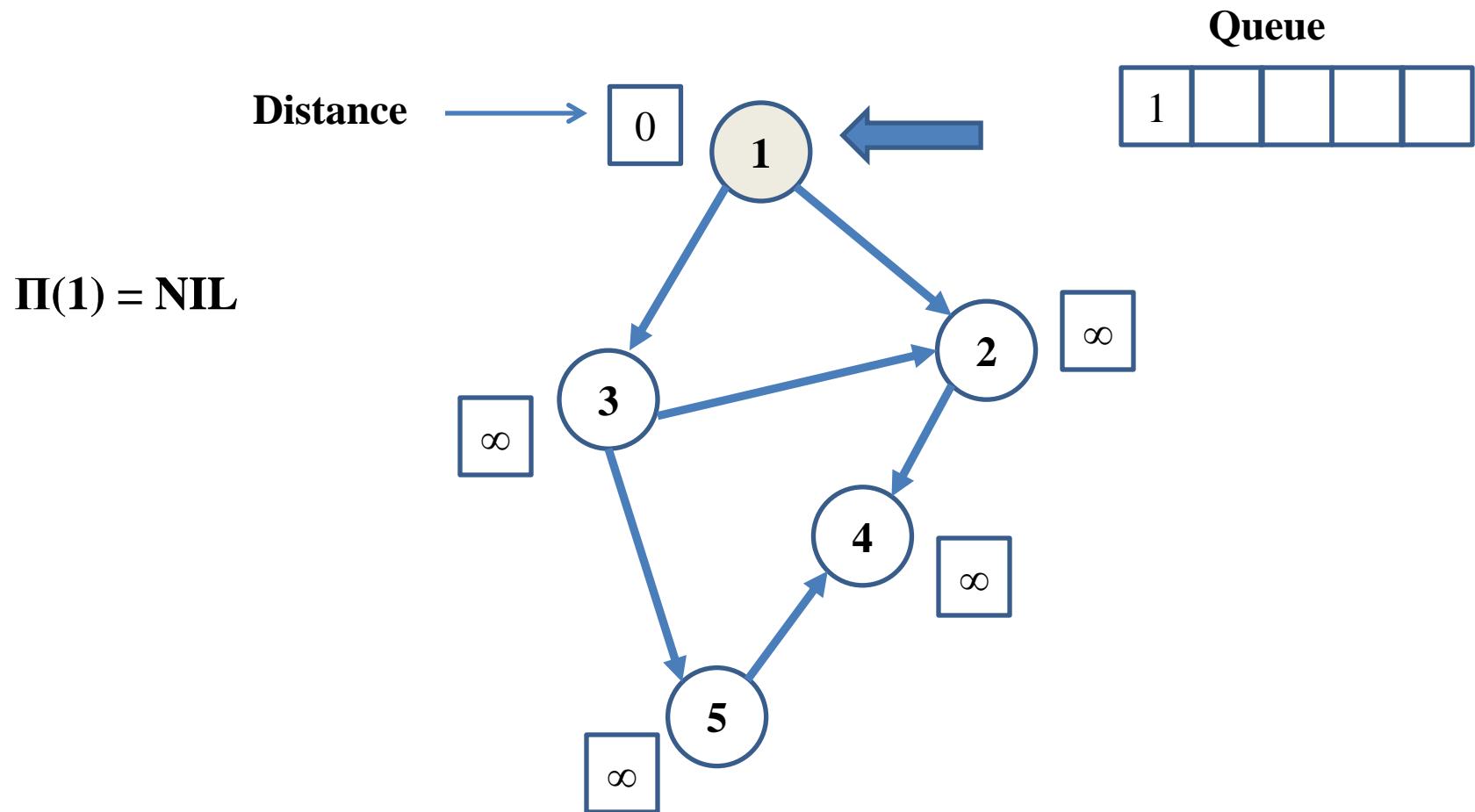
Applications of BFS

- Cycle detection
- Path Detection
- Finding strongly connected components

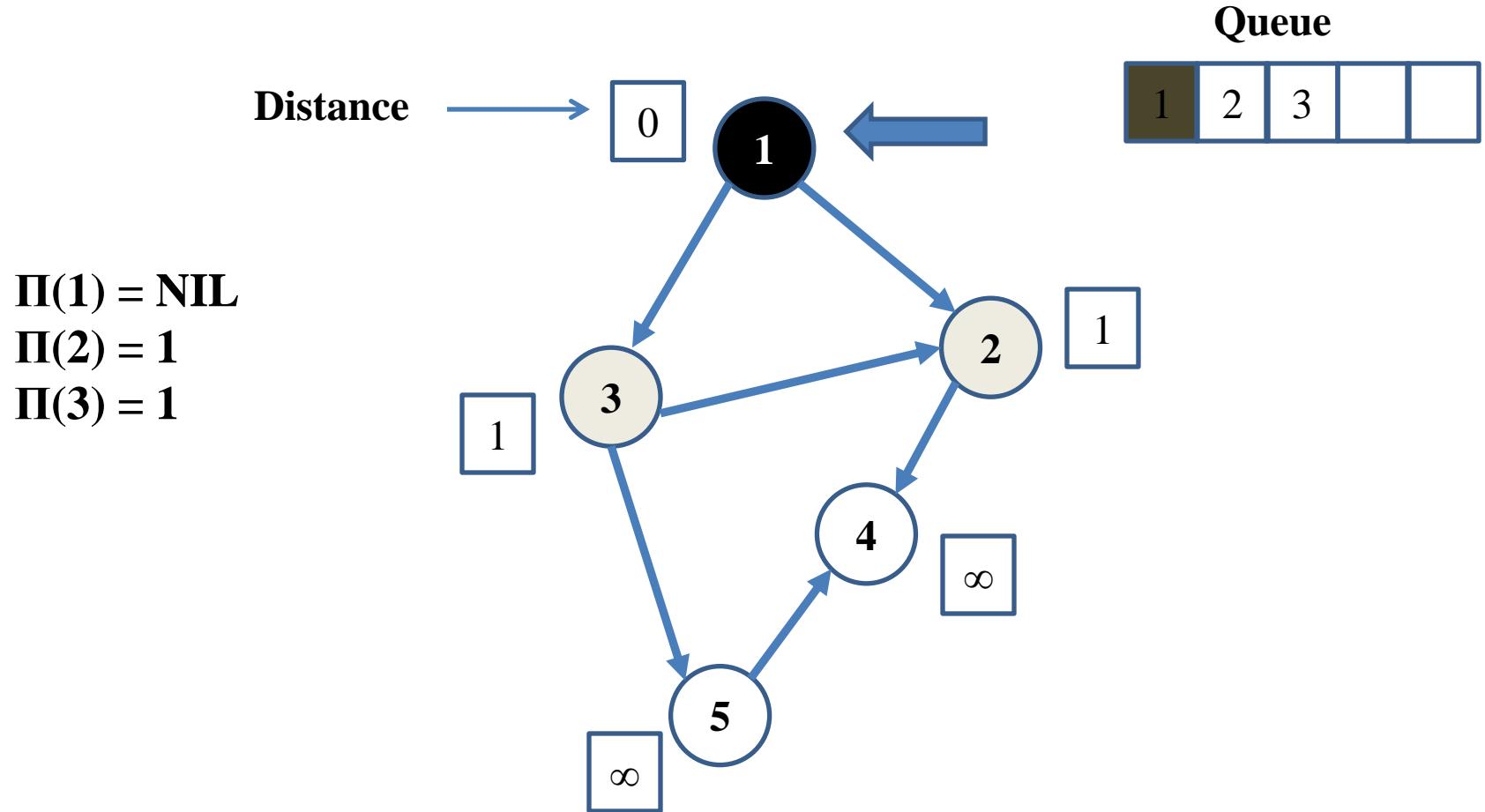
Associated Notations

- **Nodes are assigned colours**
 - WHITE : The node has not been visited
 - GRAY : The node has been visited, but all of its branches have not been visited completely
 - BLACK : A node and its branches have been explored completely
- Every node is also assigned a distance value which is the number of steps it took to reach that node from the source vertex

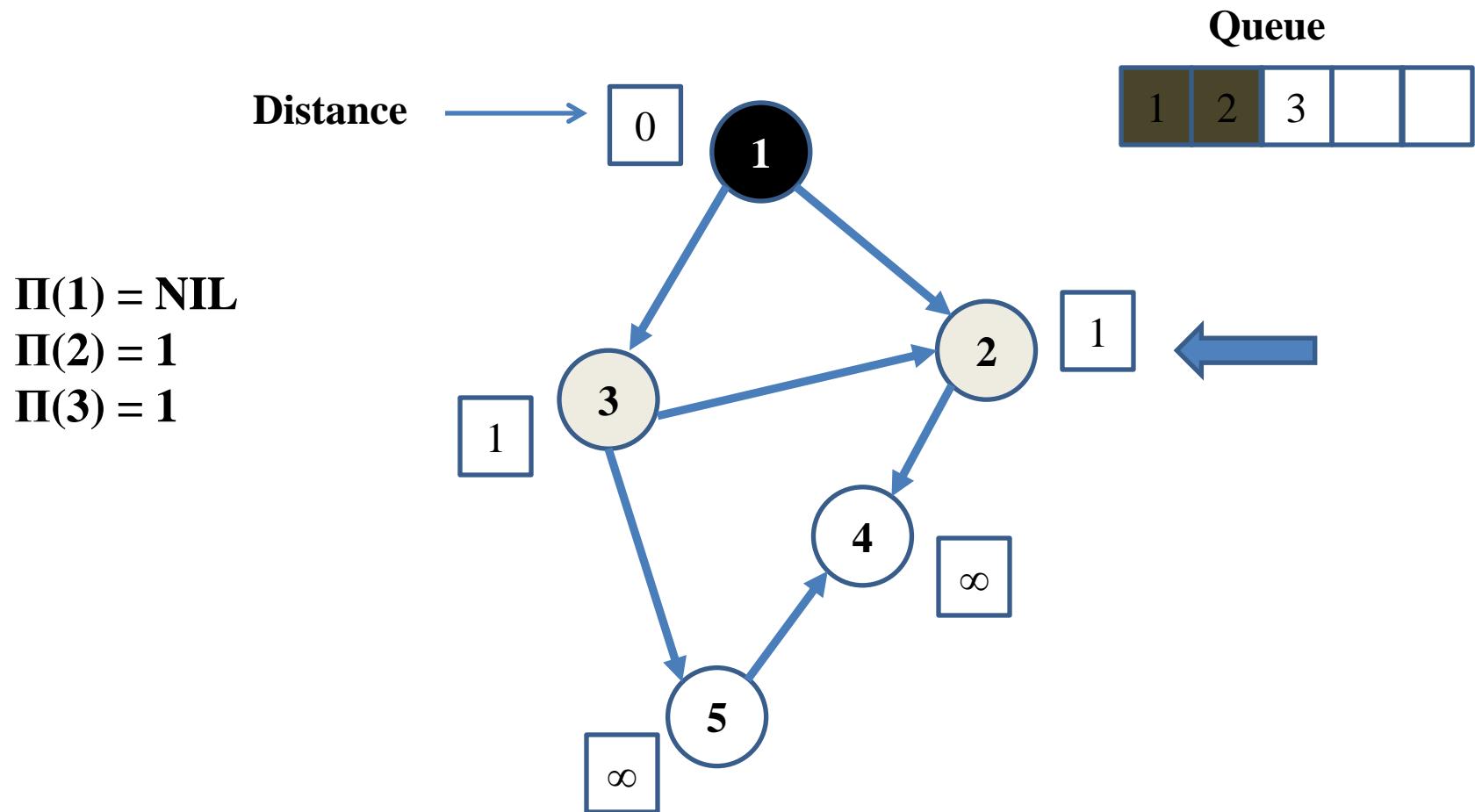
BFS in Action



BFS in Action

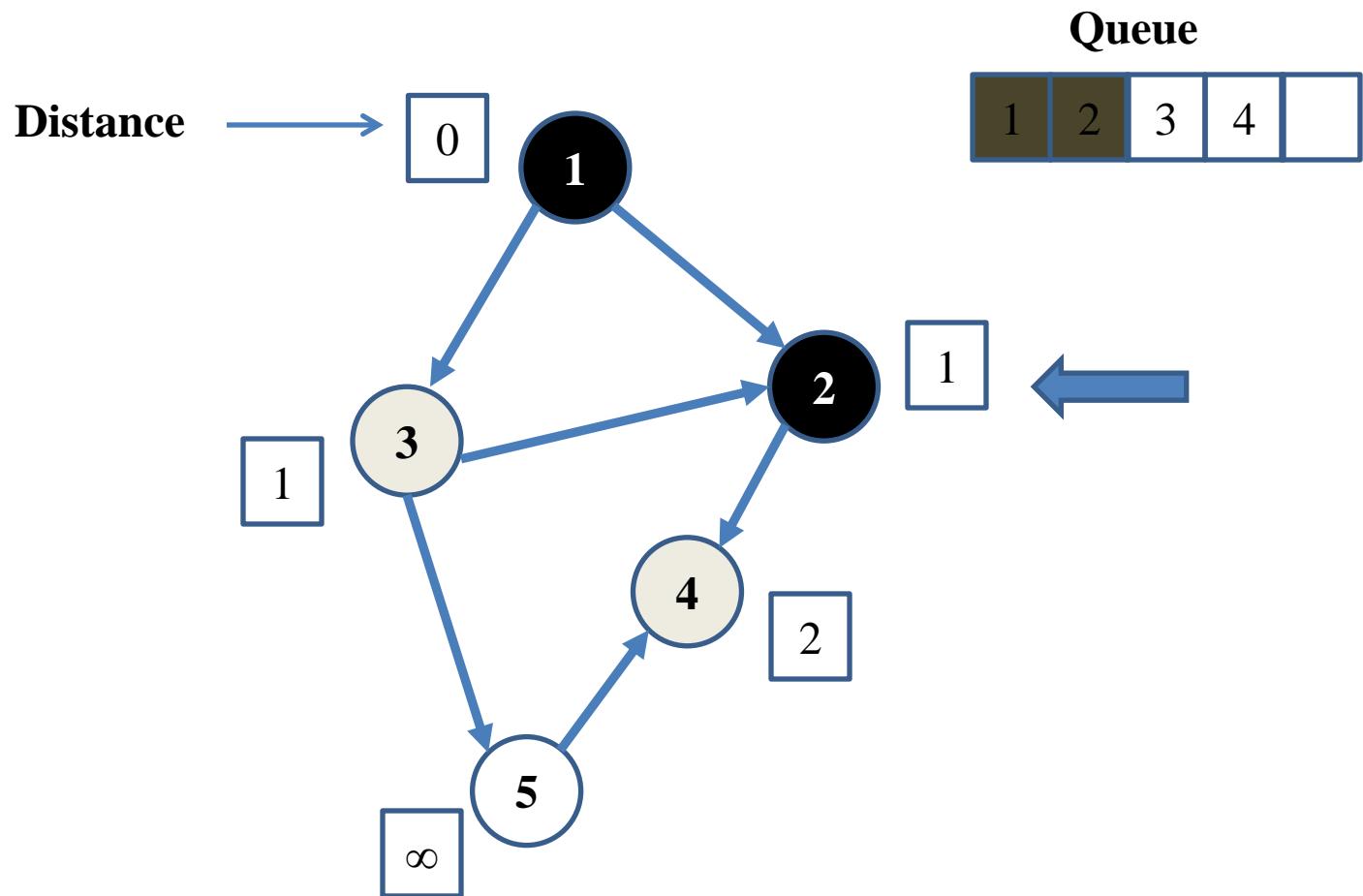


BFS in Action

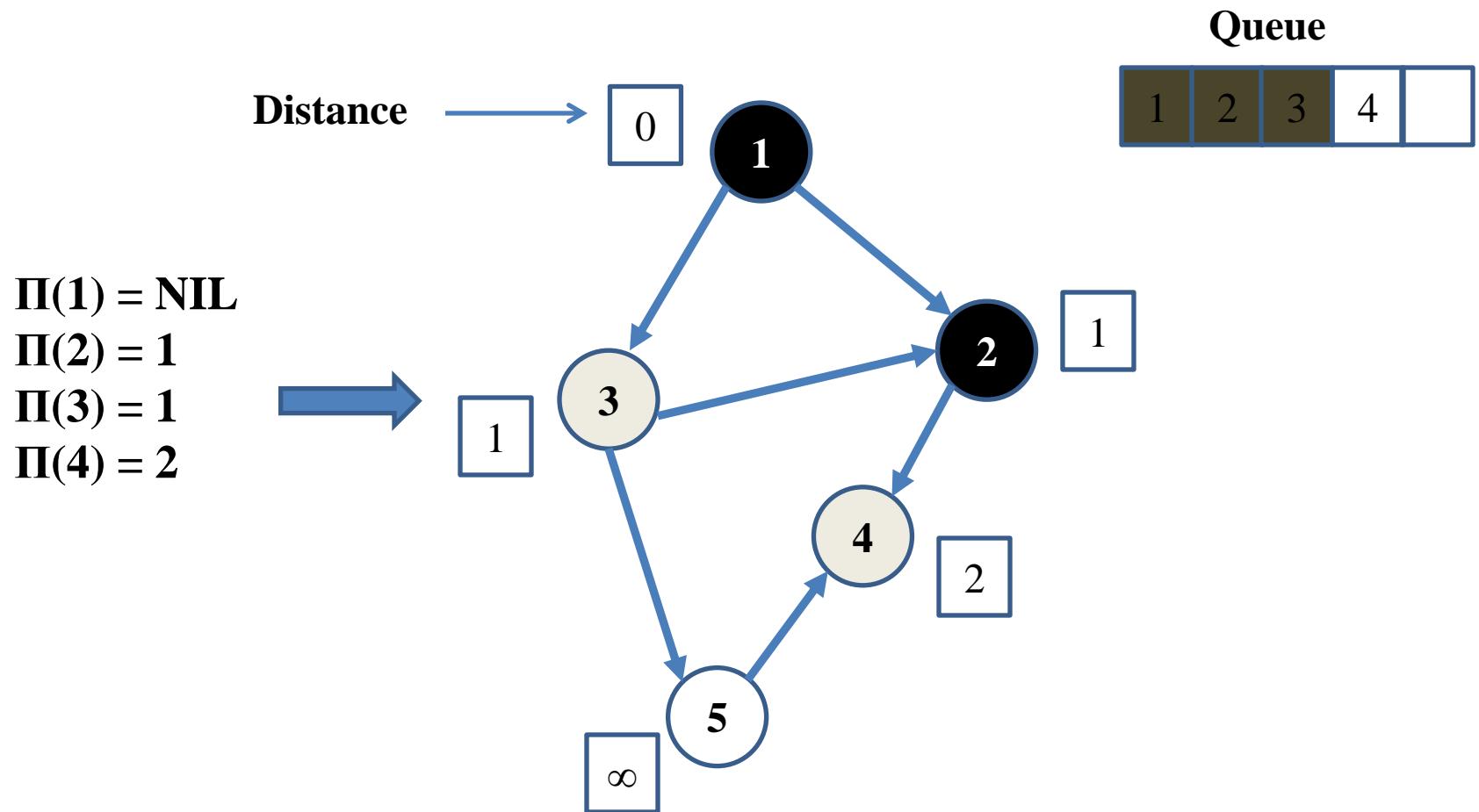


BFS in Action

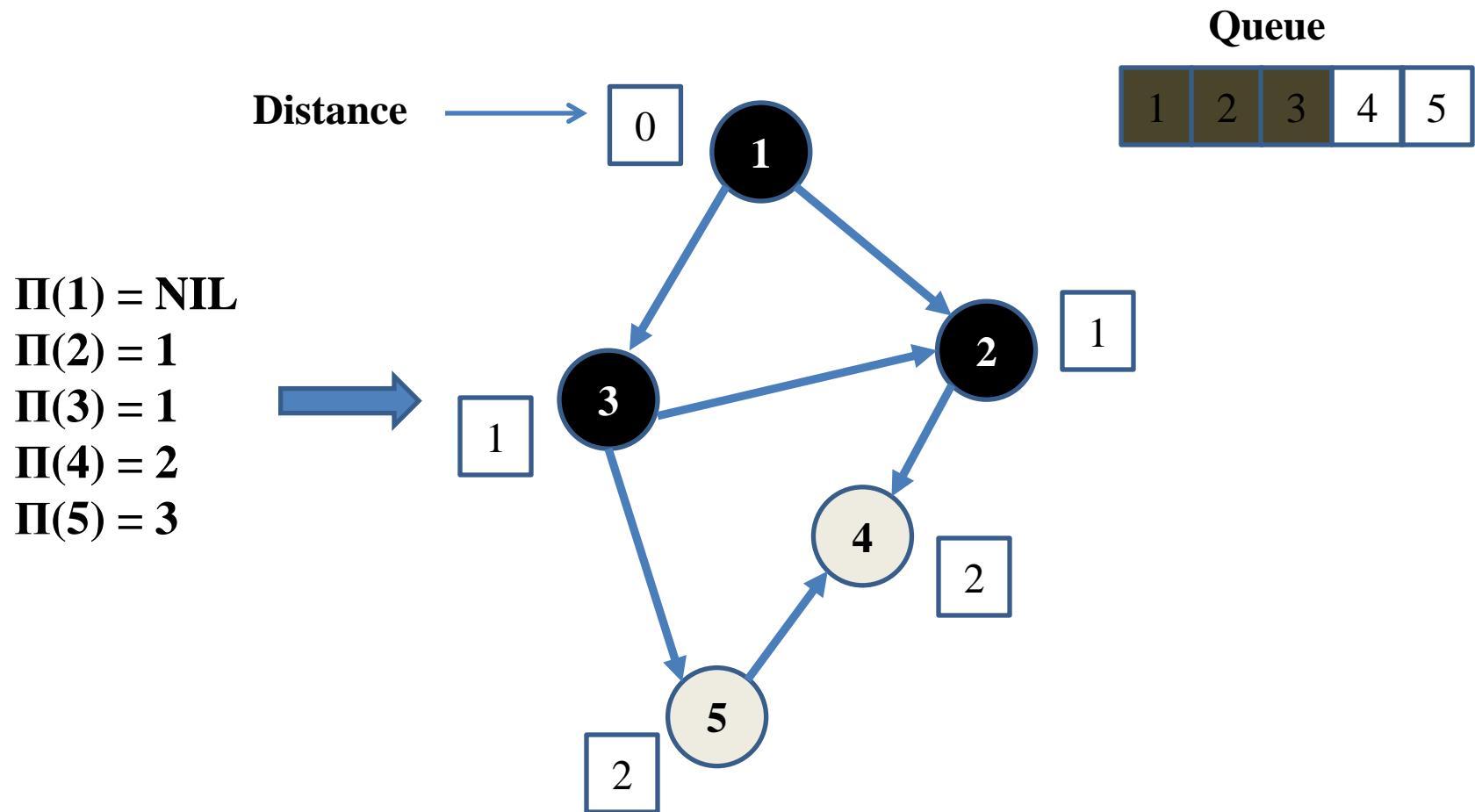
$\Pi(1) = \text{NIL}$
 $\Pi(2) = 1$
 $\Pi(3) = 1$
 $\Pi(4) = 2$



BFS in Action

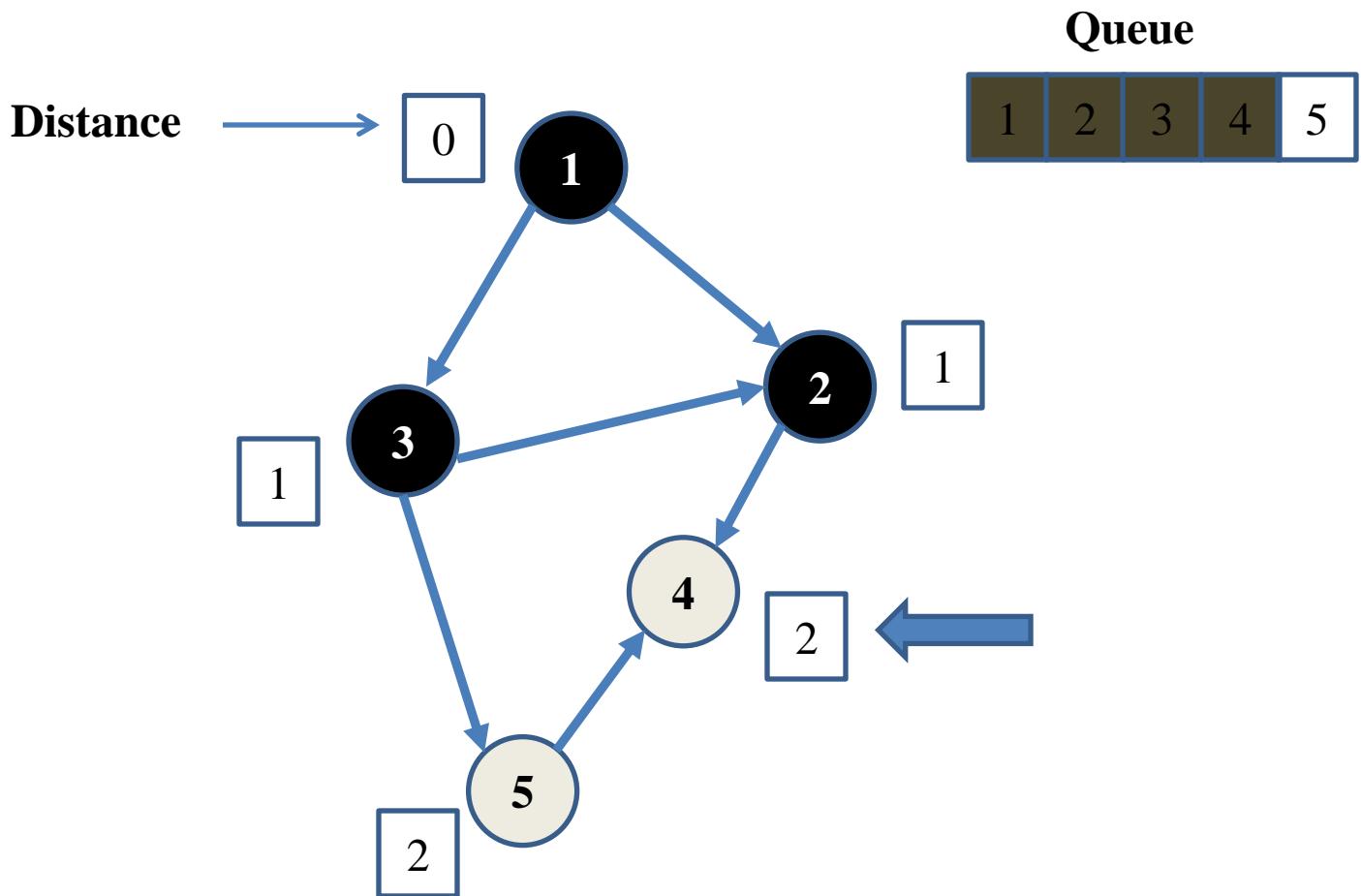


BFS in Action



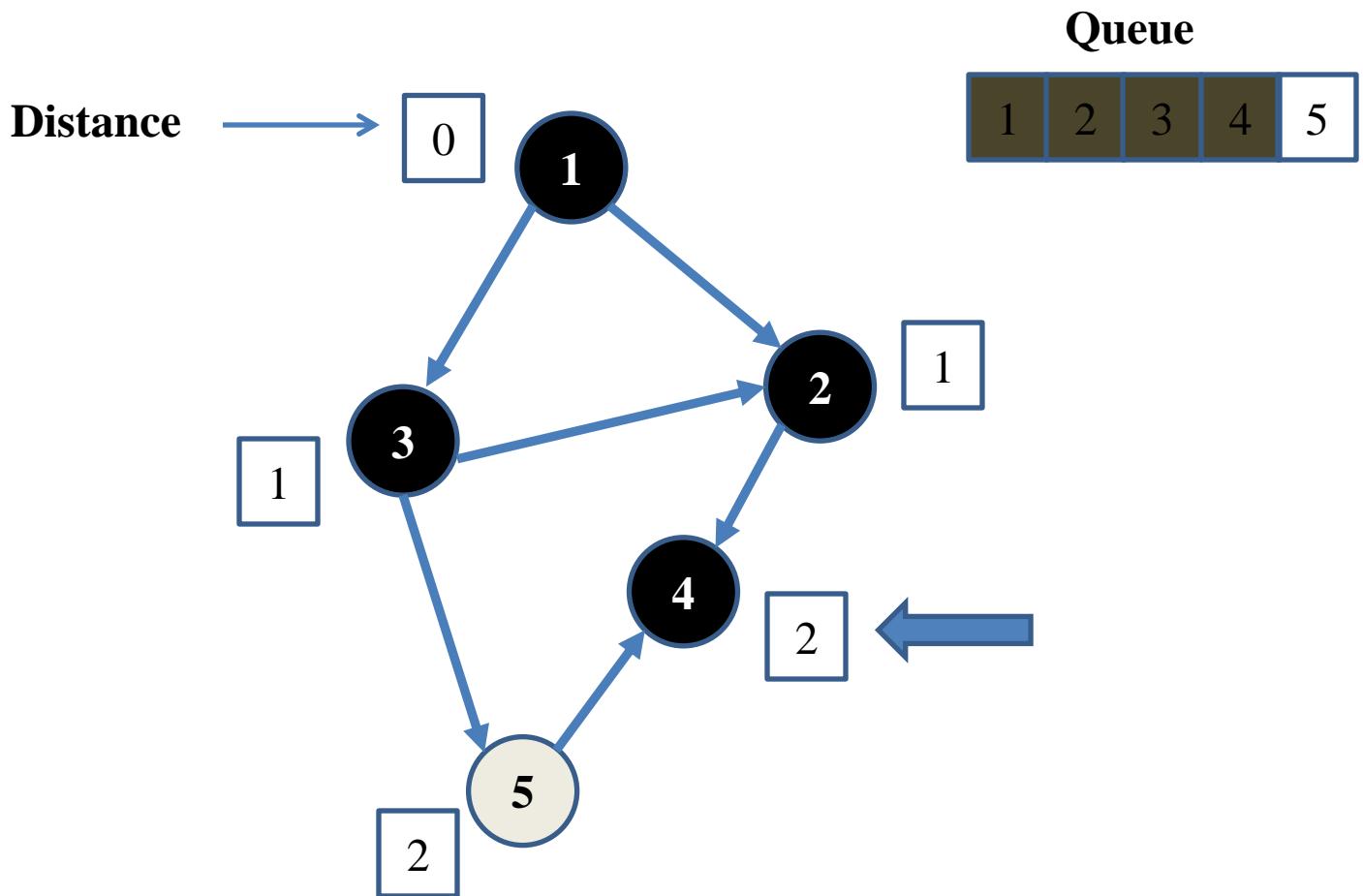
BFS in Action

$\Pi(1) = \text{NIL}$
 $\Pi(2) = 1$
 $\Pi(3) = 1$
 $\Pi(4) = 2$
 $\Pi(5) = 3$



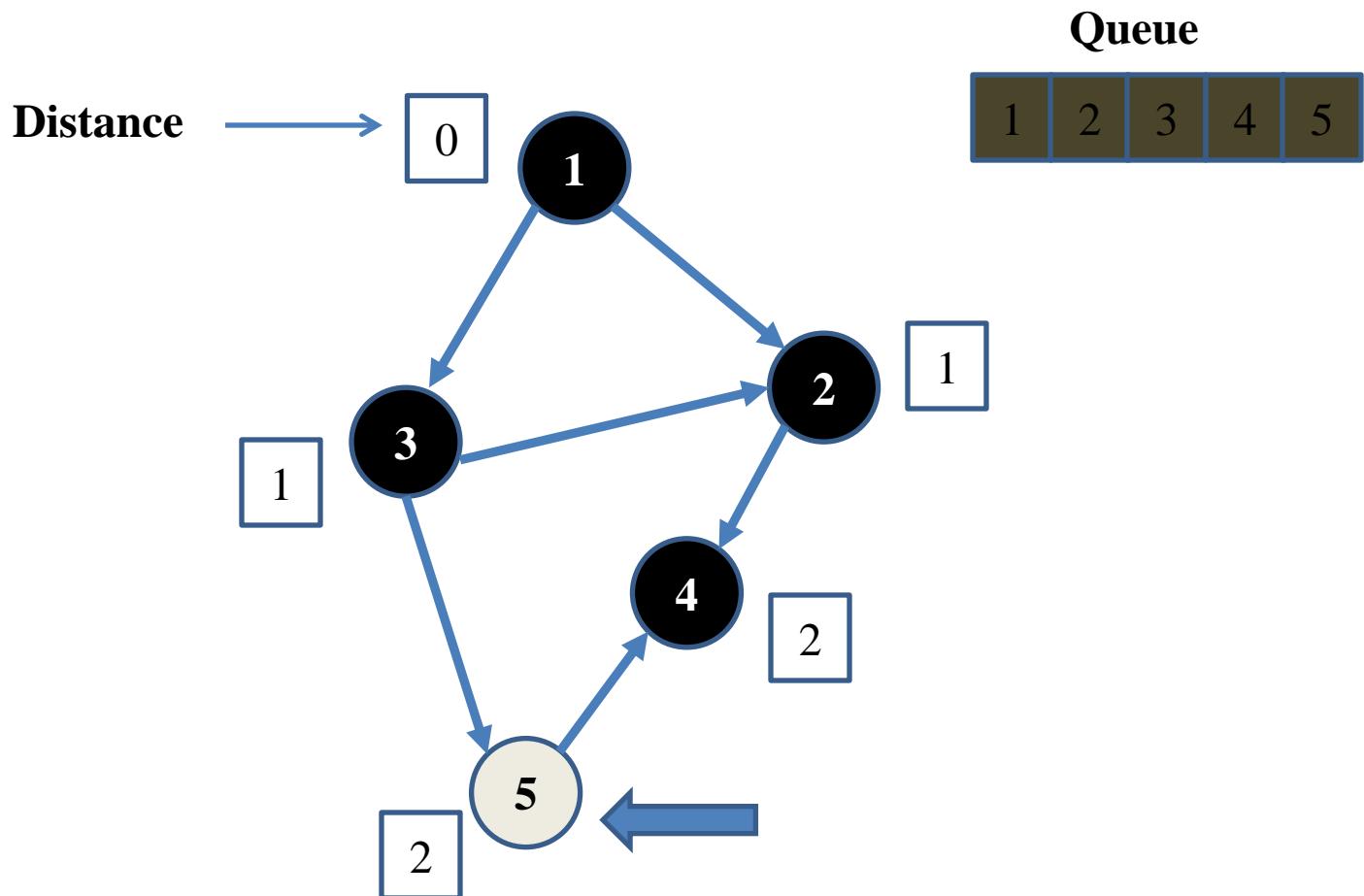
BFS in Action

$\Pi(1) = \text{NIL}$
 $\Pi(2) = 1$
 $\Pi(3) = 1$
 $\Pi(4) = 2$
 $\Pi(5) = 3$



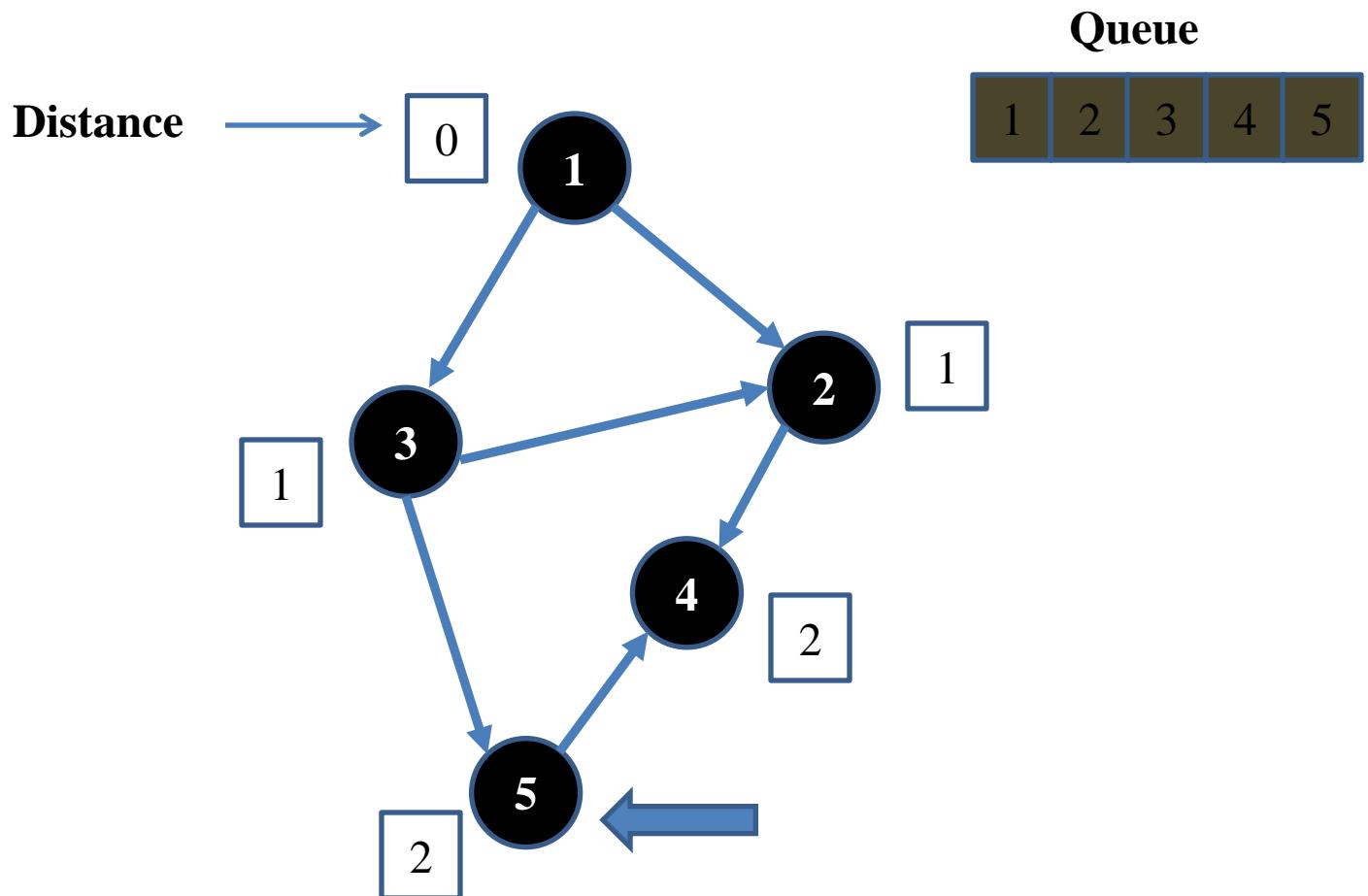
BFS in Action

$\Pi(1) = \text{NIL}$
 $\Pi(2) = 1$
 $\Pi(3) = 1$
 $\Pi(4) = 2$
 $\Pi(5) = 3$

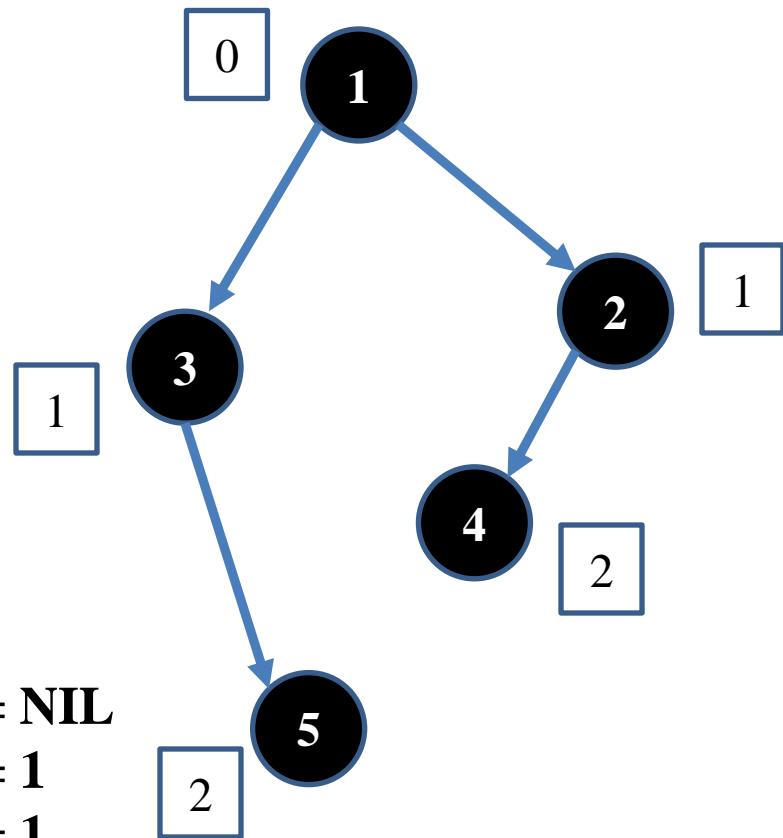


BFS in Action

$\Pi(1) = \text{NIL}$
 $\Pi(2) = 1$
 $\Pi(3) = 1$
 $\Pi(4) = 2$
 $\Pi(5) = 3$



BFS Tree



- The breadth first tree gives the nodes reachable from the source node
- For an unweighted graph, it also shows the shortest path from the source vertex to every other vertex in the graph

BFS Algorithm

$BFS(G, s)$

for each vertex u in $V[G] - \{s\}$

$\text{colour}[u] = \text{WHITE}$

$\Pi[u] = \text{NIL}$

$d[u] = \infty$

$\text{colour}[s] = \text{GRAY}$

$d[s] = 0$

$\Pi[u] = \text{NIL}$

$Q = \emptyset$

$\text{ENQUEUE}(Q, s)$

BFS Algorithm

while $Q \neq \emptyset$

$u = DEQUEUE()$

for each v **in** $Adj[u]$

if $colour[v] = WHITE$

$\Pi[v] = u$

$d[v] = d[u] + 1$

$ENQUEUE(v)$

$colour[u] = BLACK$

Time Complexity of BFS

- Every node is explored EXACTLY ONCE --- $\Theta(|V|)$
- For every node u , BFS explores all the edges in $\text{Adj}[u]$
- When summed over all the nodes in the graph, this amounts to the number of edges in the graph

$$\sum_{u \in V} \text{Adj}[u] = \Theta(|E|)$$

- Thus, **total complexity of BFS is also $\Theta(|V| + |E|)$**

Thank You

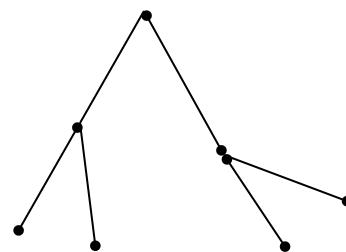
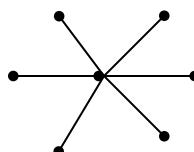
Minimum Spanning Trees

Slide and Content Credits:

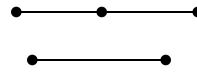
Prof. Roger Crawfis
Ohio State University

Tree

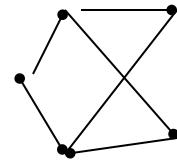
- We call an undirected graph a **tree** if the graph is *connected* and contains *no cycles*.
- Trees:



- Not Trees:



Not
connected



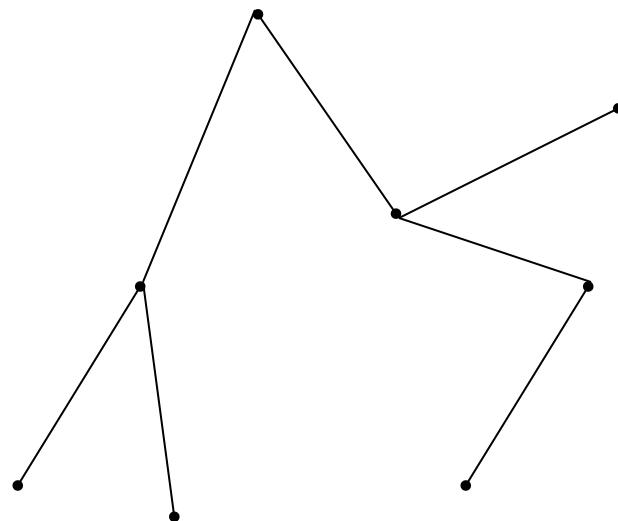
Has a cycle

Number of Vertices

- If a graph is a tree, then the number of edges in the graph is one less than the number of vertices.
- A tree with n vertices has $n - 1$ edges.
 - Each node has one parent except for the root.
 - Note: Any node can be the root here, as we are not dealing with rooted trees.

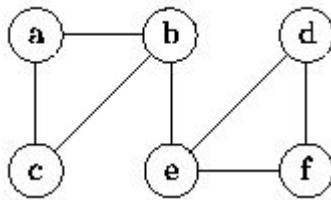
Connected Graph

- A **connected graph** is one in which there is *at least one path* between each pair of vertices.

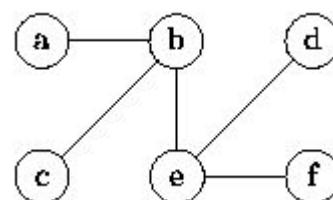


Spanning Tree

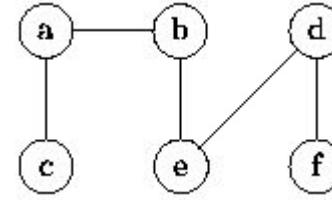
- In a tree there is always **exactly one path** from each vertex in the graph to any other vertex in the graph.
- A **spanning tree** for a graph is a subgraph that includes every vertex of the original, and is a tree.



(a) Graph G



(b) Breadth-first
spanning tree of
G rooted at b



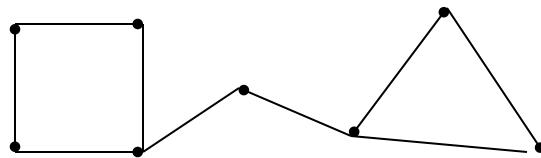
(c) Depth-first
spanning tree of
G rooted at c

Non-Connected Graphs

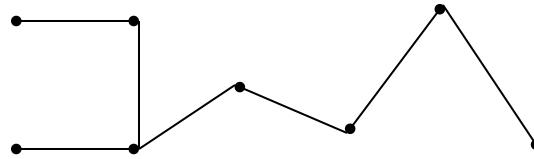
- If the graph is not connected, we get a spanning tree for each **connected component** of the graph.
 - That is we get a forest.

Finding a Spanning Tree

Find a spanning tree for the graph below.



We could break the two cycles by removing a single edge from each. One of several possible ways to do this is shown below.



Was breadth-first or depth-first search (or neither) used to create this?

Minimum Spanning Tree

- A spanning tree that has minimum total weight is called a **minimum spanning tree** for the graph.
 - Technically it is a minimum-weight spanning tree.
- If all edges have the same weight, breadth-first search or depth-first search will yield minimum spanning trees.
 - For the rest of this discussion, we assume the edges have weights associated with them.

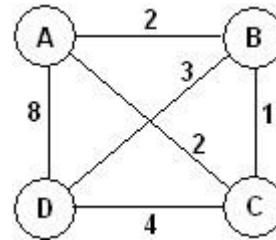
Note, we are strictly dealing with undirected graphs here, for directed graphs we would want to find the optimum branching of the directed graph.

Minimum Spanning Tree

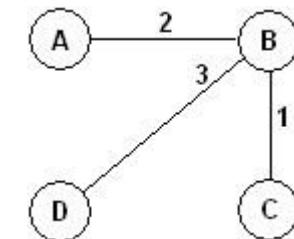
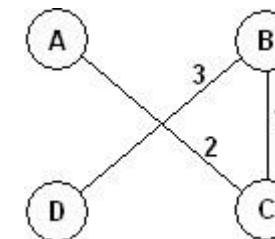
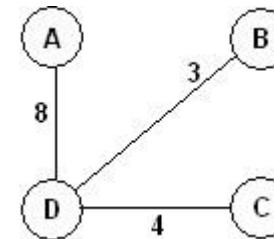
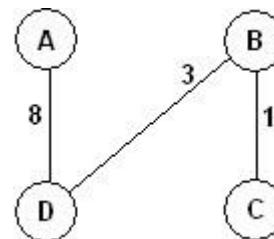
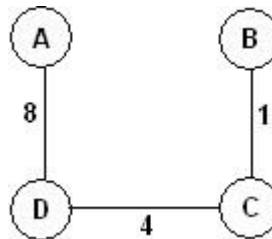
- Minimum-cost spanning trees have many applications.
 - Building cable networks that join n locations with minimum cost.
 - Building a road network that joins n cities with minimum cost.
 - Obtaining an independent set of circuit equations for an electrical network.
 - In pattern recognition minimal spanning trees can be used to find noisy pixels.

Minimum Spanning Tree

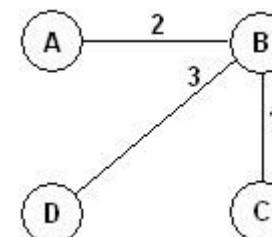
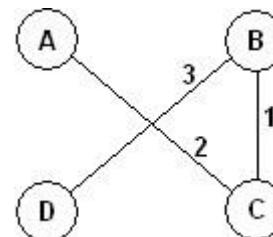
- Consider this graph.



- It has 20 spanning trees. Some are:



- There are two minimum-cost spanning trees, each with a cost of 6:



Minimum Spanning Tree

- Brute Force option:
 1. For all possible spanning trees
 - i. Calculate the sum of the edge weights
 - ii. Keep track of the tree with the minimum weight.
- Step i) requires $N-1$ time, since each tree will have exactly $N-1$ edges.
- If there are M spanning trees, then the total cost will $O(MN)$.
- Consider a complete graph, with $N(N-1)$ edges.
How big can M be?

Brute Force MST

- For a complete graph, it has been shown that there are N^{N-2} possible spanning trees!
- Alternatively, given N items, you can build N^{N-2} distinct trees to connect these items.

MST-Greedy Techniques

- There are many approaches to computing a minimum spanning tree. We could try to detect cycles and remove edges, but the two algorithms we will study build them from the bottom-up in a *greedy* fashion.
- **Kruskal's Algorithm** – *starts with a forest of single node trees* and then adds the edge with the minimum weight to connect two components.
- **Prim's Algorithm** – *starts with a single vertex* and then adds the minimum edge to extend the spanning tree.

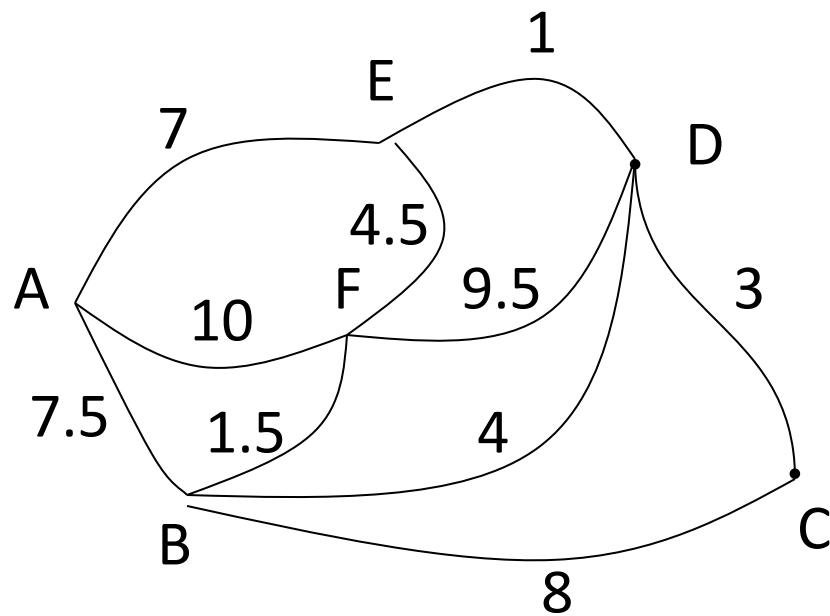
Kruskal's Algorithm

- Greedy algorithm to choose the edges as follows.

Step 1	First edge: choose any edge with the minimum weight.
Step 2	Next edge: choose any edge with minimum weight from <i>those not yet selected</i> . (The subgraph can look disconnected at this stage.)
Step 3	Continue to choose edges of minimum weight from those not yet selected, except do not select any edge that creates a cycle in the subgraph.
Step 4	Repeat step 3 until the subgraph connects all vertices of the original graph.

Kruskal's Algorithm

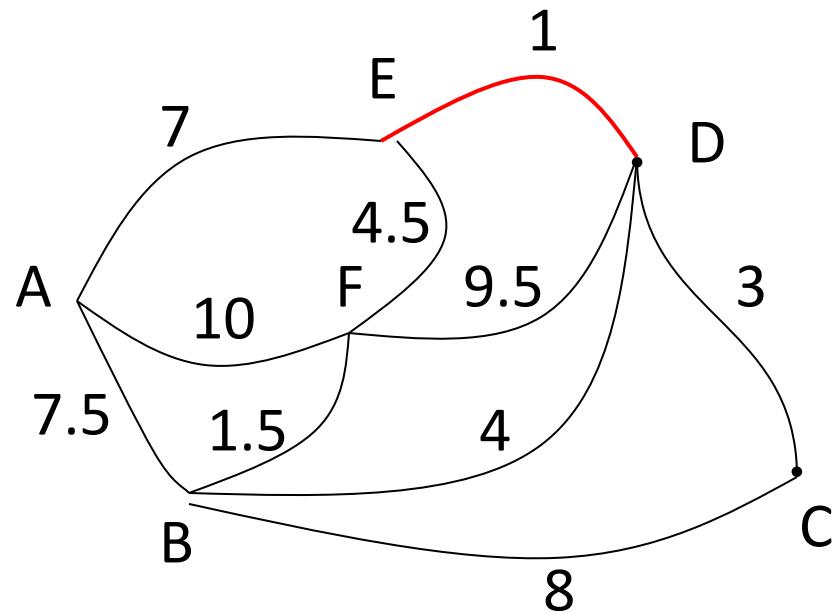
Use Kruskal's algorithm to find a minimum spanning tree for the graph.



Kruskal's Algorithm

Solution

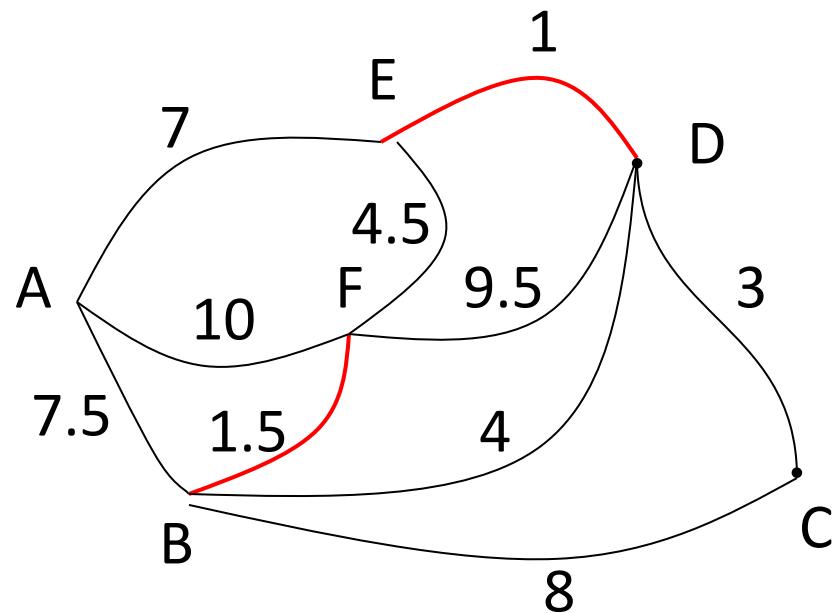
First, choose ED (the smallest weight).



Kruskal's Algorithm

Solution

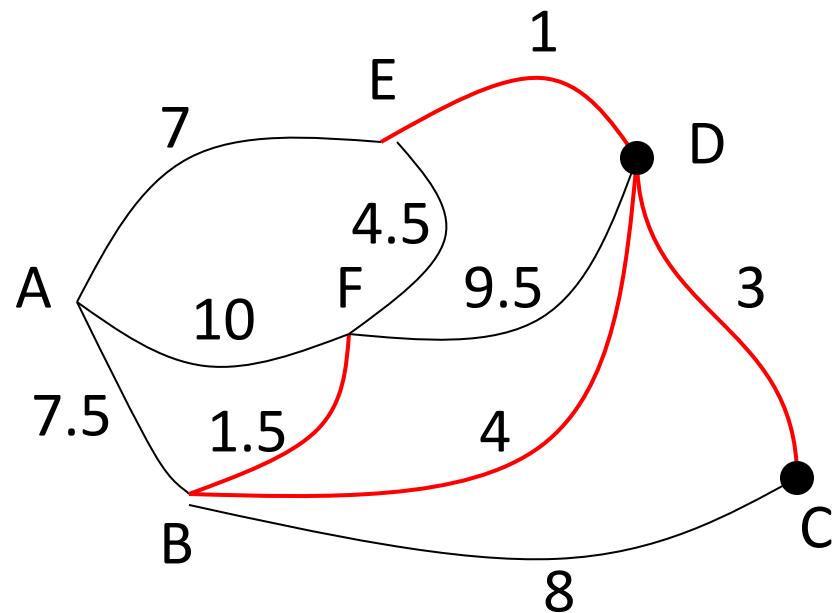
Now choose BF (the smallest remaining weight).



Kruskal's Algorithm

Solution

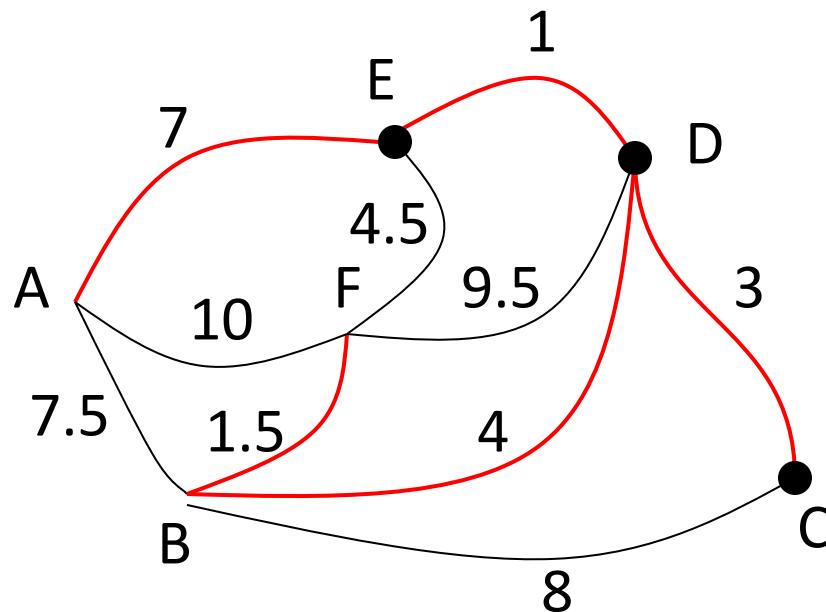
Now CD and then BD.



Kruskal's Algorithm

Solution

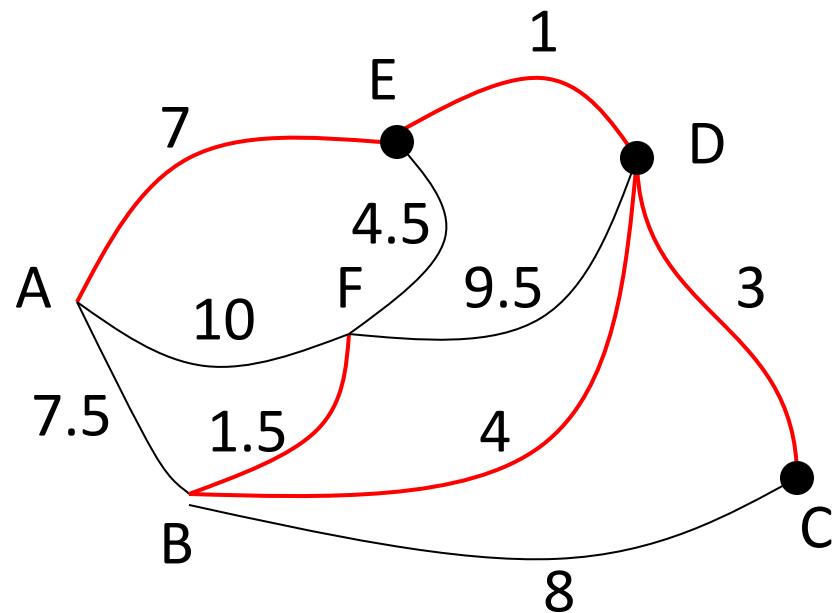
Note EF is the smallest remaining, but that would create a cycle. Choose AE and we are done.



Kruskal's Algorithm

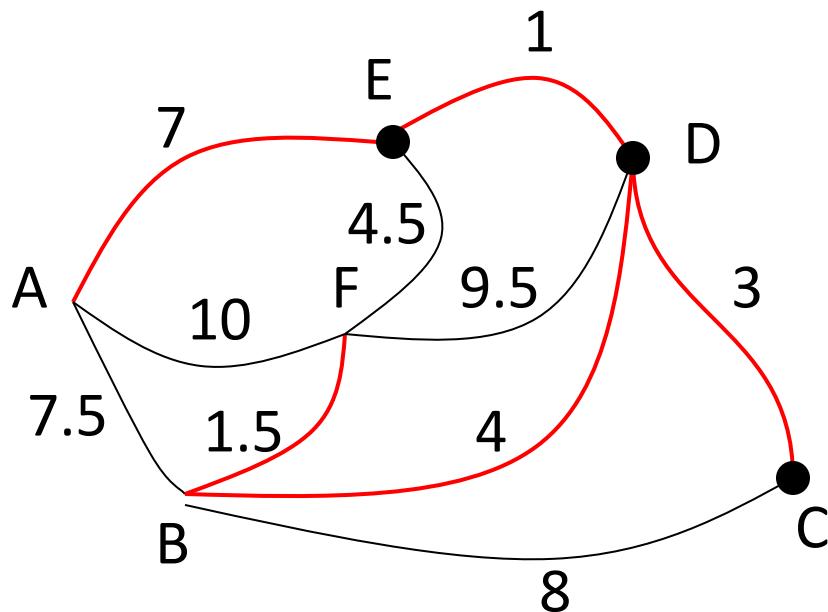
Solution

The total weight of the tree is 16.5.



Kruskal's Algorithm

- Some questions:
 1. How do we know we are finished?
 2. How do we check for cycles?



Kruskal's Algorithm

Build a priority queue (min-based) with all of the edges of G.

T = \varnothing ;

while(queue is not empty)

{

get minimum edge e from priorityQueue;

if(e does not create a cycle with edges in T)

add e to T;

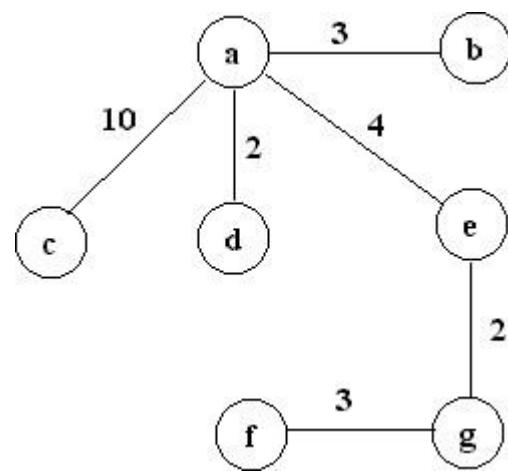
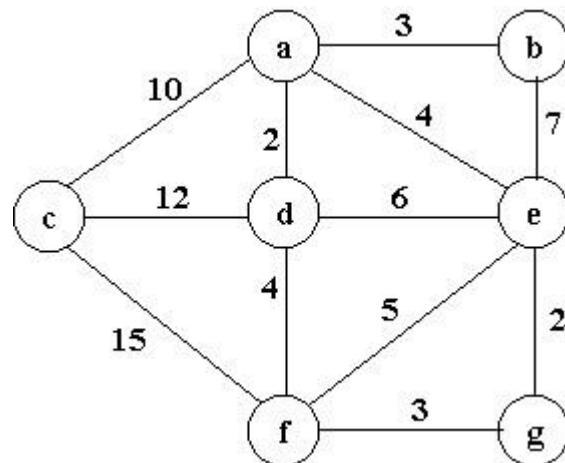
}

return T;

Kruskal's Algorithm

edge	ad	eg	ab	fg	ae	df	ef	de	be	ac	cd	cf
weight	2	2	3	3	4	4	5	6	7	10	12	15
insertion status	✓	✓	✓	✓	✓	x	x	x	x	✓	x	x
insertion order	1	2	3	4	5					6		

- Trace of Kruskal's algorithm for the undirected, weighted graph:



The minimum cost is: 24

Kruskal's Algorithm – Time complexity

- Steps
 - Initialize forest $O(|V|)$
 - Sort edges $O(|E|\log|E|)$
 - Check edge for cycles $O(|V|)$ x
 - Number of edges $O(|V|) = O(|V|^2)$
 - Total $O(|V| + |E|\log|E| + |V|^2)$
 - Since $|E| = O(|V|^2)$ $O(|V|^2 \log|V|)$
 - Thus we would class MST as $O(n^2 \log n)$ for a graph with n vertices
 - This is an *upper bound*, some improvements on this are known.

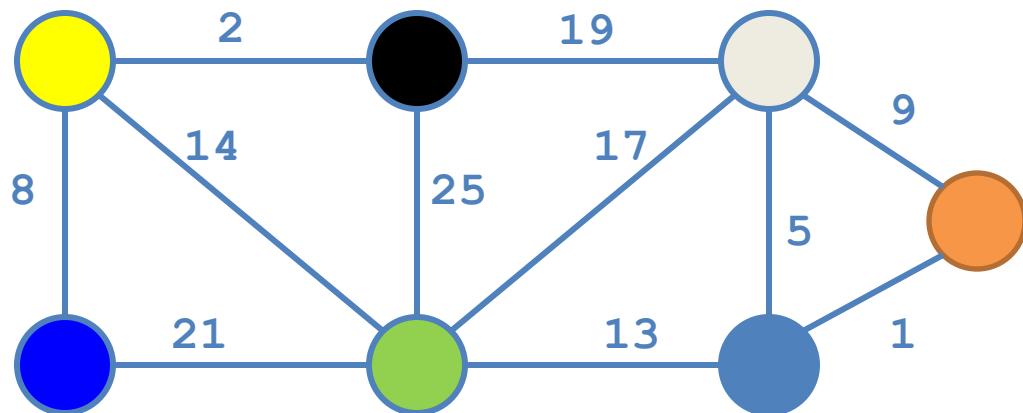
Kruskal's Algorithm

- Another implementation is based on sets (see Chapter 21).

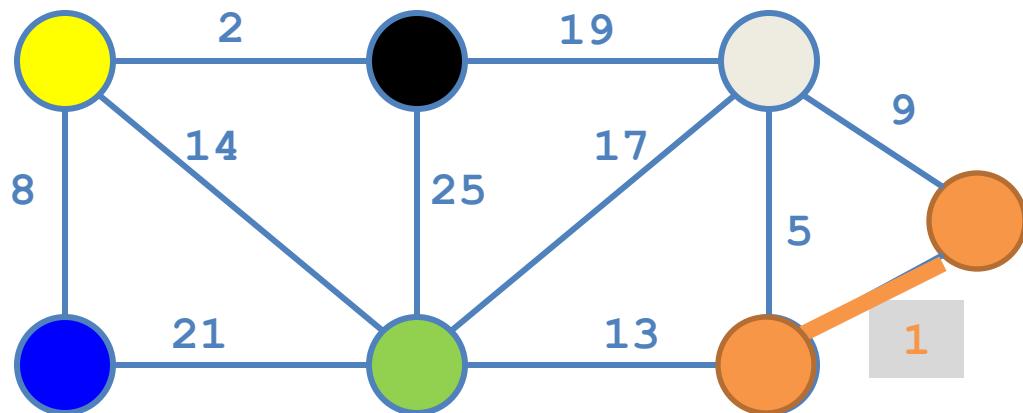
```
Kruskal()
```

```
{  
    T = Ø;  
    for each v ∈ V  
        MakeSet(v);  
    sort E by increasing edge weight w  
    for each (u,v) ∈ E (in sorted order)  
        if FindSet(u) ≠ FindSet(v)  
            T = T ∪ {{u,v}};  
            Union(FindSet(u), FindSet(v));  
}
```

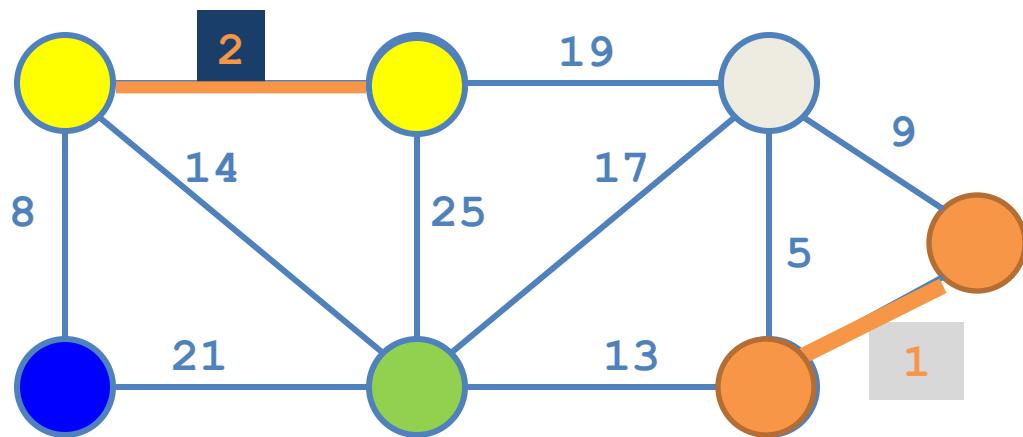
Kruskal's Algorithm



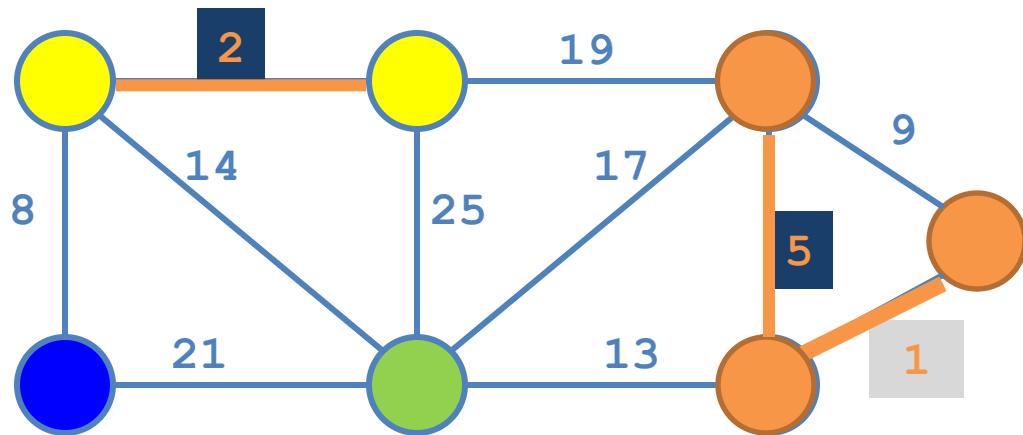
Kruskal's Algorithm



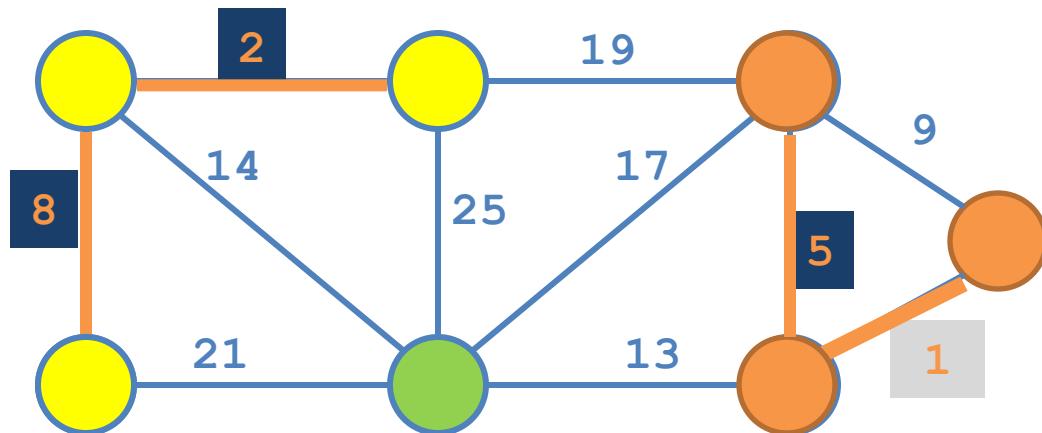
Kruskal's Algorithm



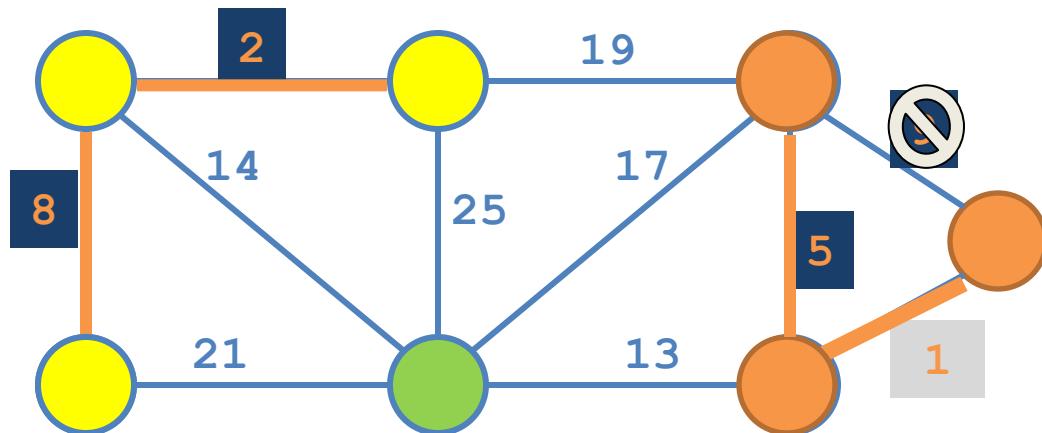
Kruskal's Algorithm



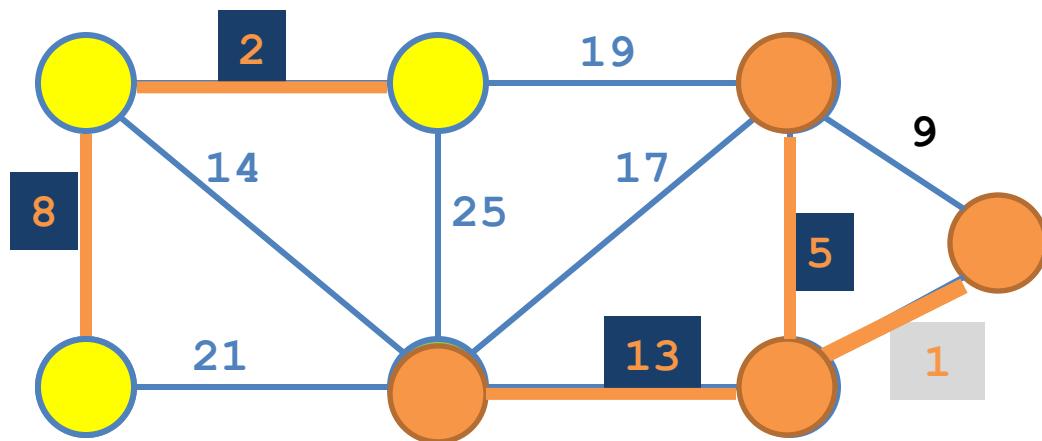
Kruskal's Algorithm



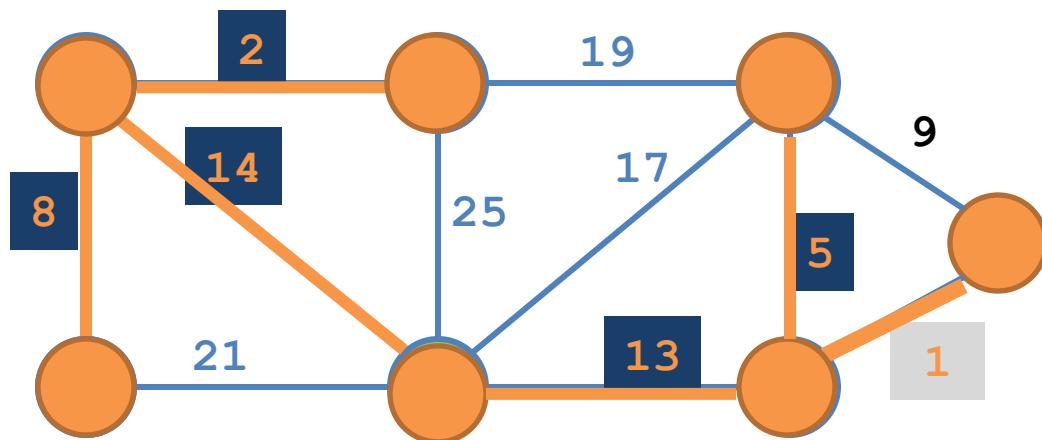
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm



Prim's Algorithm

- Prim's algorithm finds a minimum cost spanning tree by selecting edges from the graph one-by-one as follows:
- It starts with a tree, T , consisting of a single starting vertex, x .
- Then, it finds the shortest edge emanating from x that connects T to the rest of the graph (i.e., a vertex not in the tree T).
- It adds this edge and the new vertex to the tree T .
- It then picks the shortest edge emanating from the revised tree T that also connects T to the rest of the graph and repeats the process.

Prim's Algorithm Abstract

Consider a graph $G=(V, E)$;

Let T be a tree consisting of only the starting vertex x ;

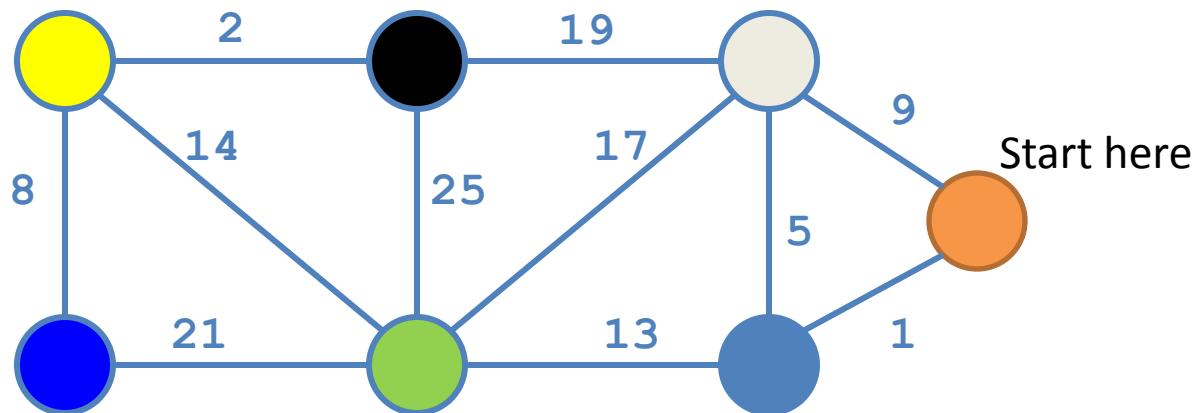
while (T has fewer than $|V|$ vertices)

{

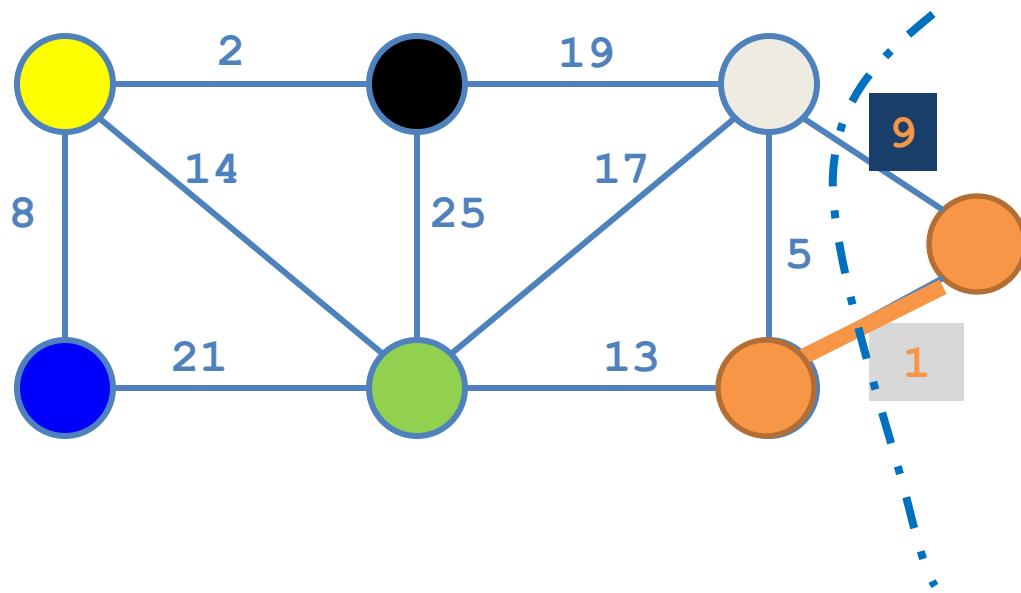
 find a smallest edge connecting T to $G-T$;
 add it to T ;

}

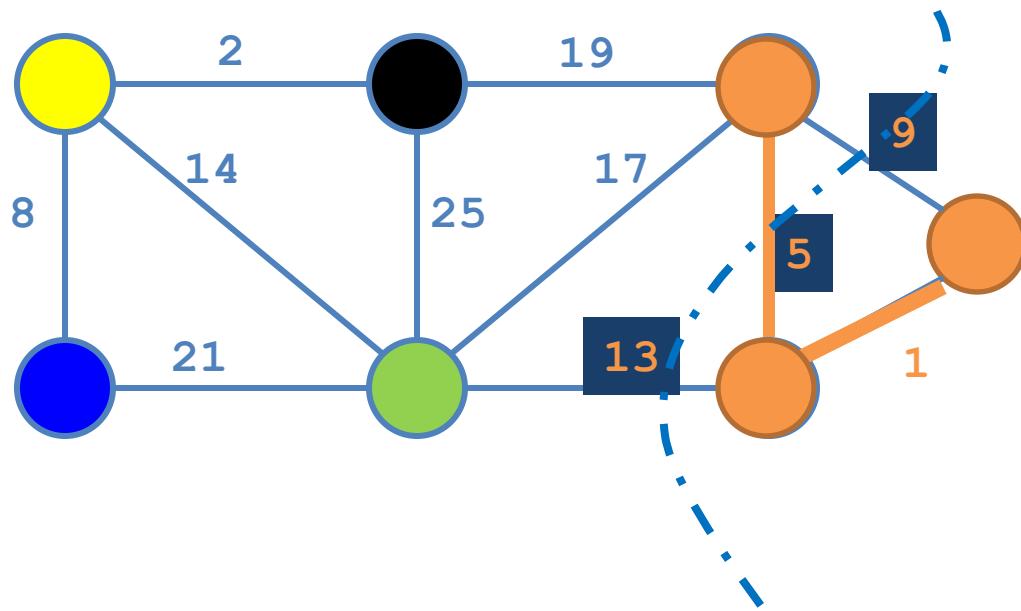
Prim's Algorithm



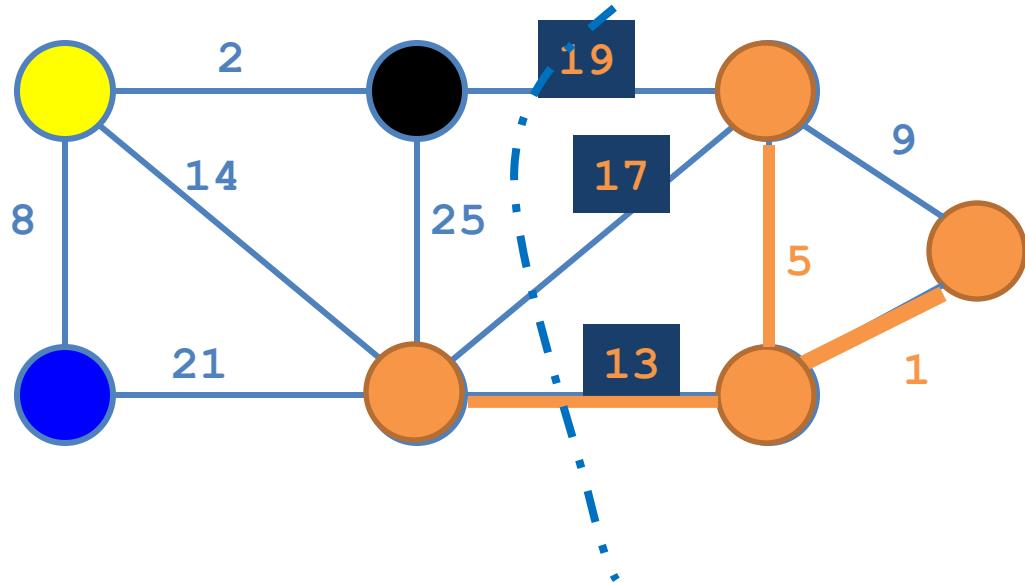
Prim's Algorithm



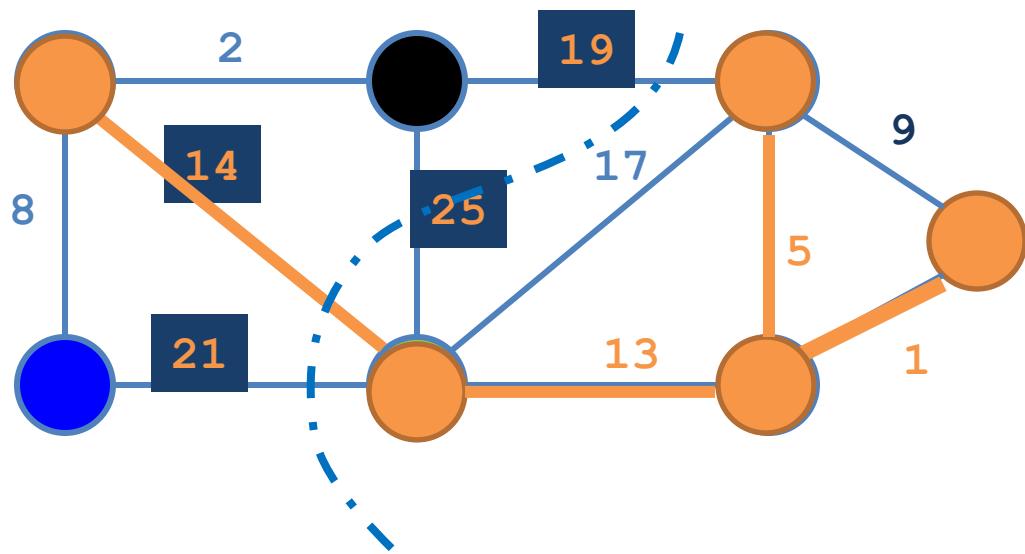
Prim's Algorithm



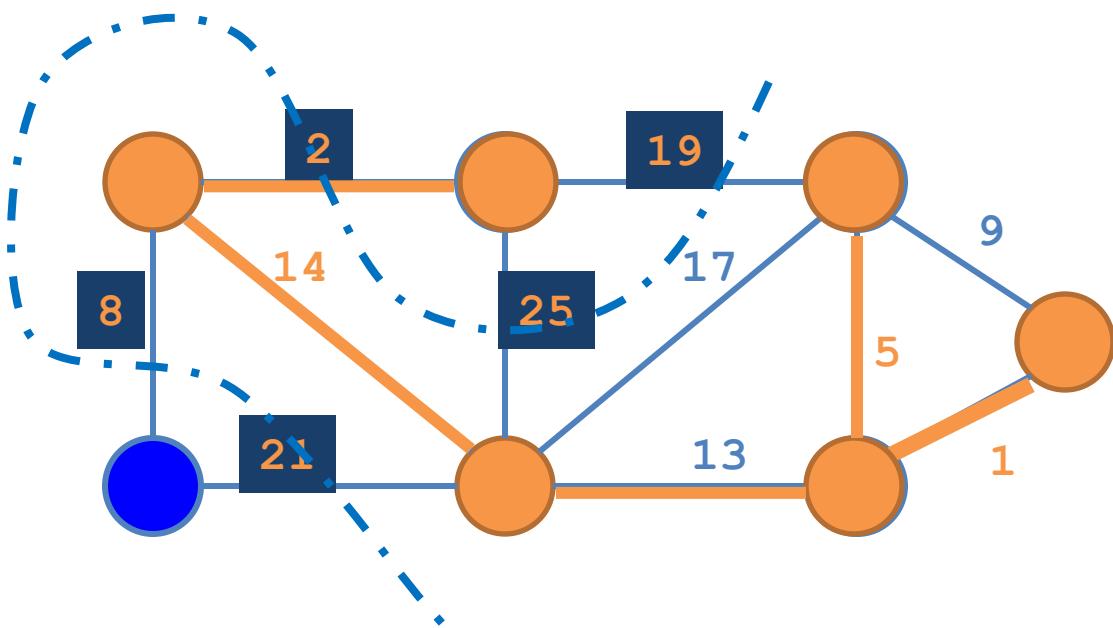
Prim's Algorithm



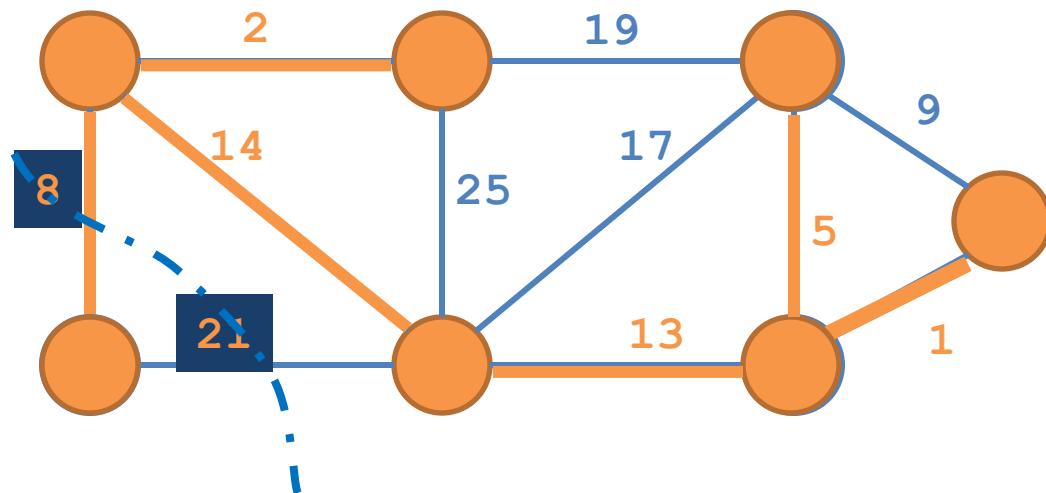
Prim's Algorithm



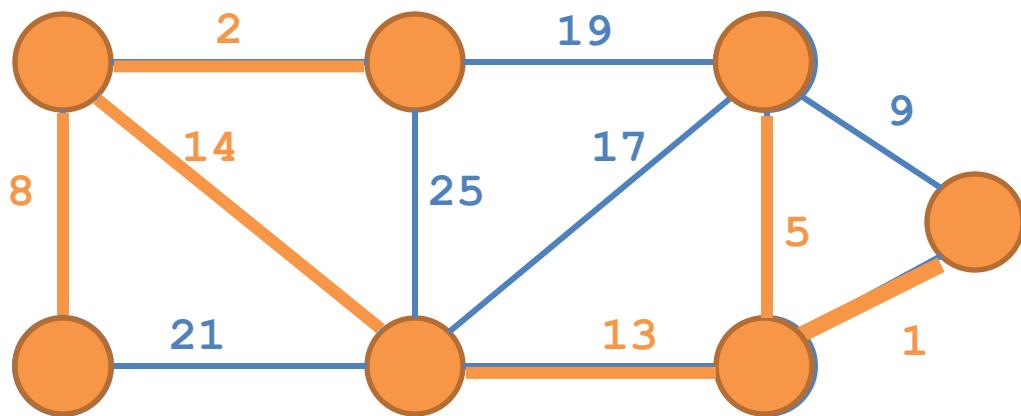
Prim's Algorithm



Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

```
function Prim(G = <N,A>: graph ; length : A → R+) : set of edges
```

```
//B-> MST Set and T-> MST edge set
```

```
{initialisation} T ← Ø
```

```
B ← {an arbitrary member of N}
```

```
While B ≠ N do
```

```
    find e = {u , v} of minimum length such u ∈ B and v ∈ N - B
```

```
    T ← T ∪ {e}
```

```
    B ← B ∪ {v}
```

```
Return T
```

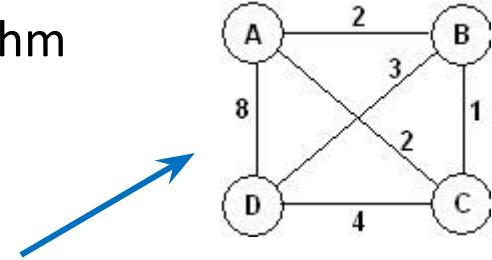
Complexity:

Outer loop: n-1 times

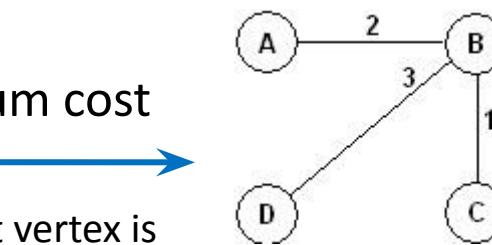
Inner loop: n times O(n²)

Prim's and Kruskal's Algorithms

- It is not necessary that Prim's and Kruskal's algorithm generate the same minimum-cost spanning tree.



- For example for the graph shown on the right:

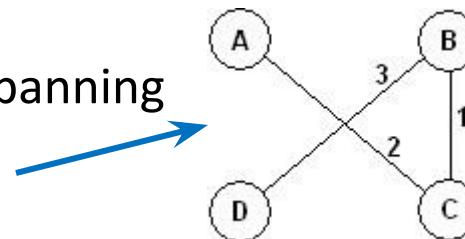


- Kruskal's algorithm results in the following minimum cost spanning tree:



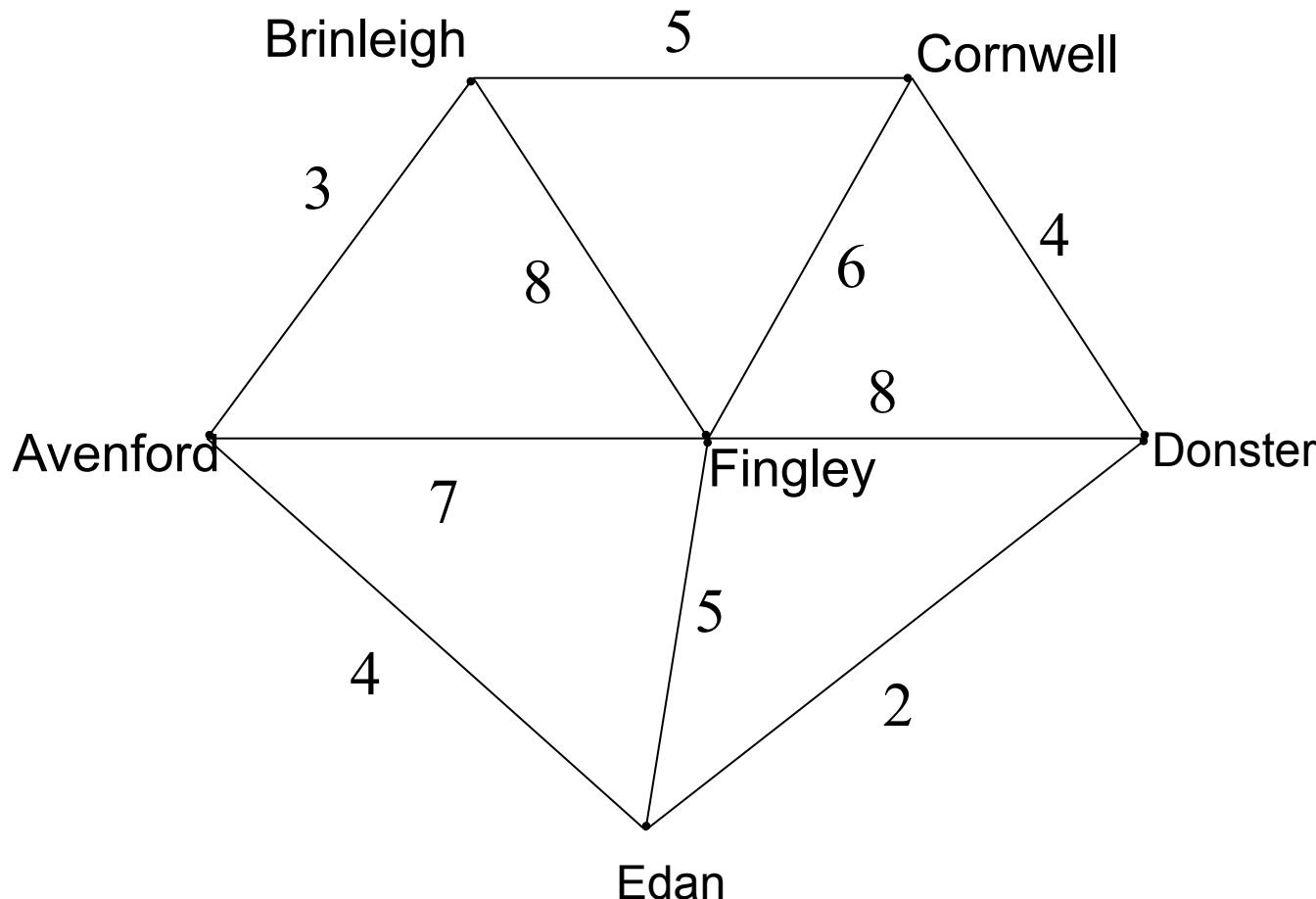
- The same tree is generated by Prim's algorithm if the start vertex is any of: A, B, or D.

- However if the start vertex is C the minimum cost spanning tree generated by Prim's algorithm is:



Prim's algorithm with an Adjacency Matrix

A cable company want to connect five villages to their network which currently extends to the market town of Avenford. What is the minimum length of cable needed?

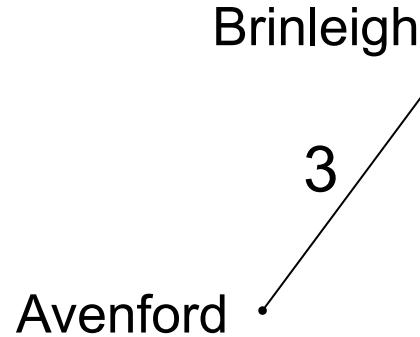


Prim's algorithm with an Adjacency Matrix

Note, this example has outgoing edges on the columns and incoming on the rows, so it is the transpose of adjacency matrix mentioned in class. Actually, it is an undirected, so $A^T = A$.

	A	B	C	D	E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

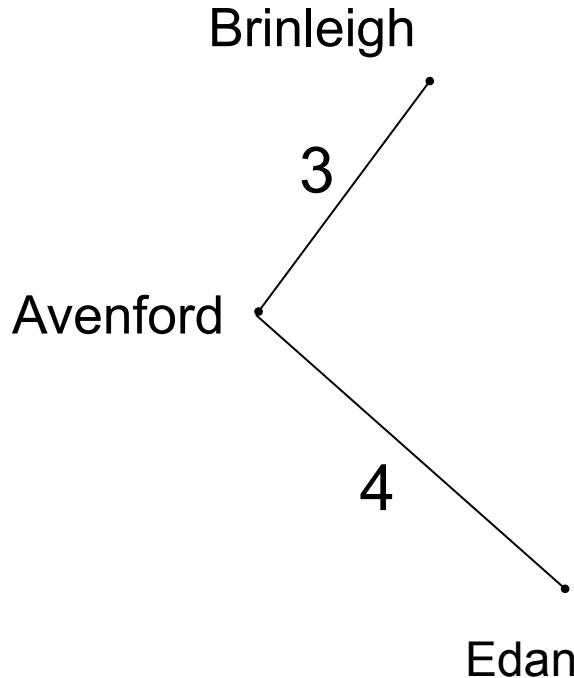
- Start at vertex A. Label column A “1” .
- Delete row A
- Select the smallest entry in column A (AB, length 3)



1

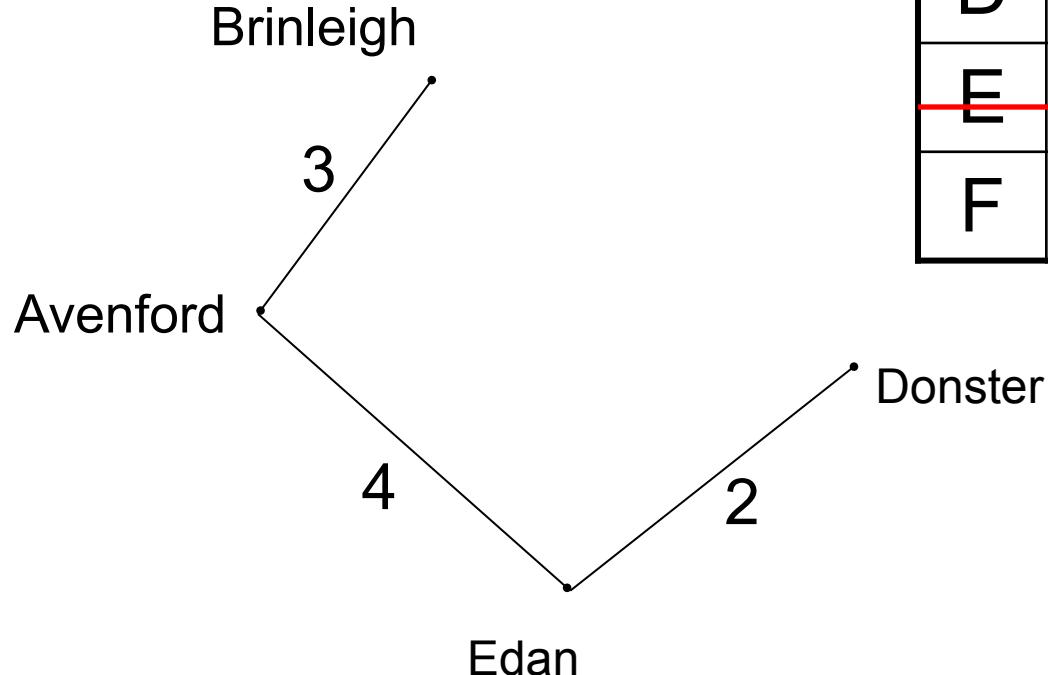
	A	B	C	D	E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

- Label column B “2”
- Delete row B
- Select the smallest uncovered entry in either column A or column B (AE, length 4)



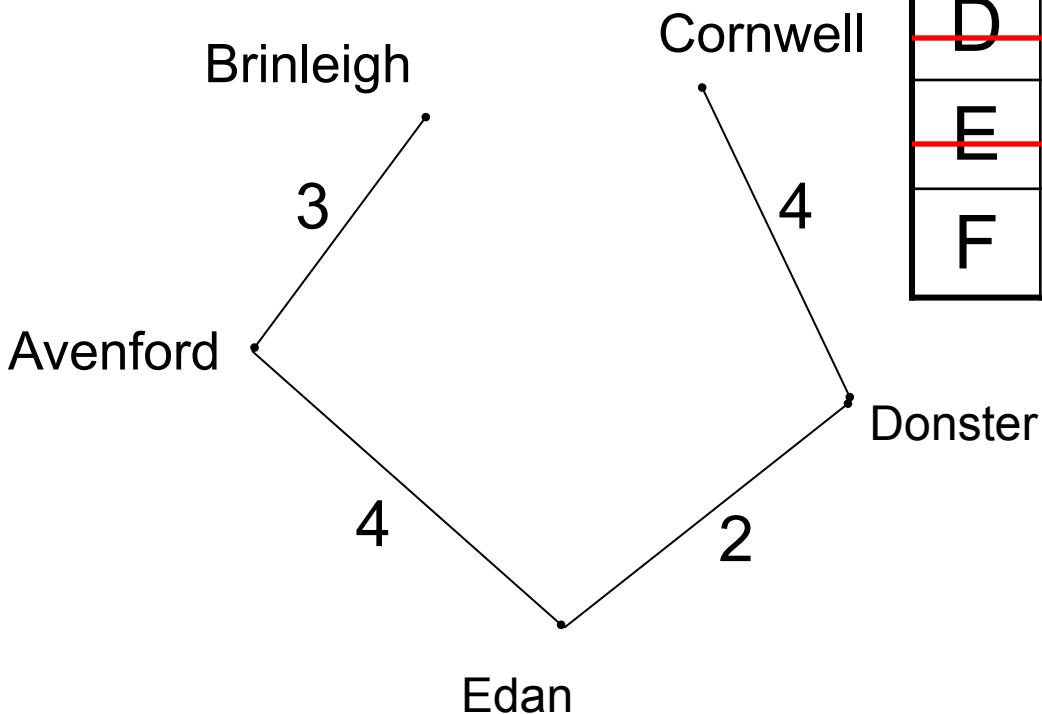
	1	2					
	A	B	C	D	E	F	
A	-	3	-	-	4	7	
B	3	-	5	-	-	8	
C	-	5	-	4	-	6	
D	-	-	4	-	2	8	
E	4	-	-	2	-	5	
F	7	8	6	8	5	-	

- Label column E “3”
- Delete row E
- Select the smallest uncovered entry in either column A, B or E (ED, length 2)



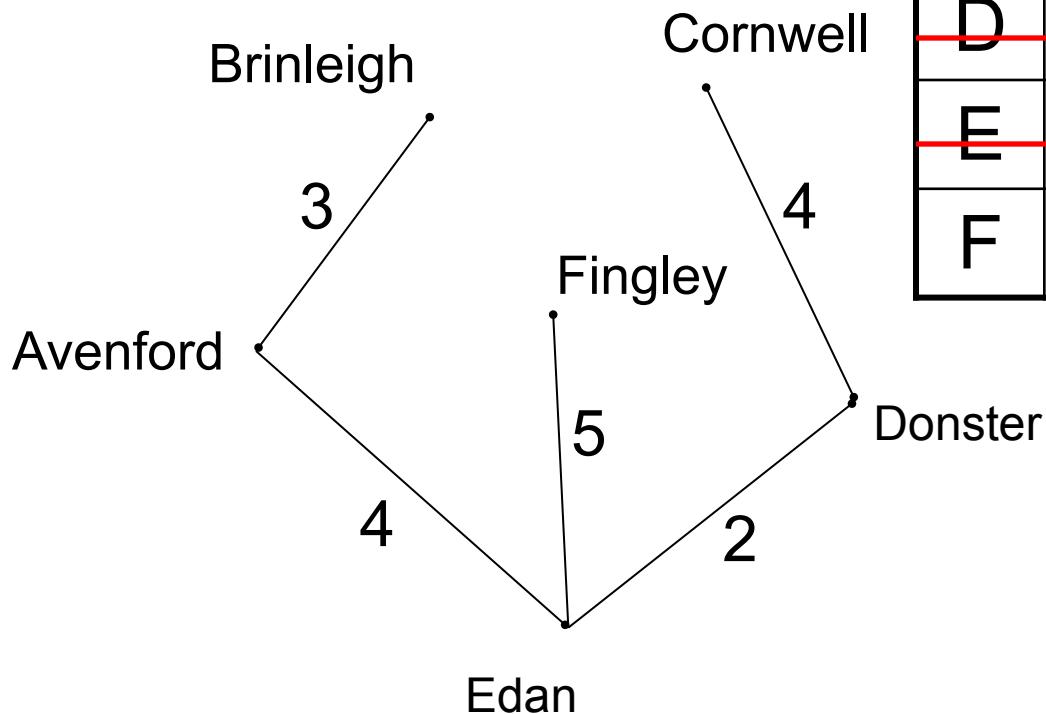
	1	2	3			
	A	B	C	D	E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

- Label column D “4”
- Delete row D
- Select the smallest uncovered entry in either column A, B, D or E (DC, length 4)



	1	2	4	3	E	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

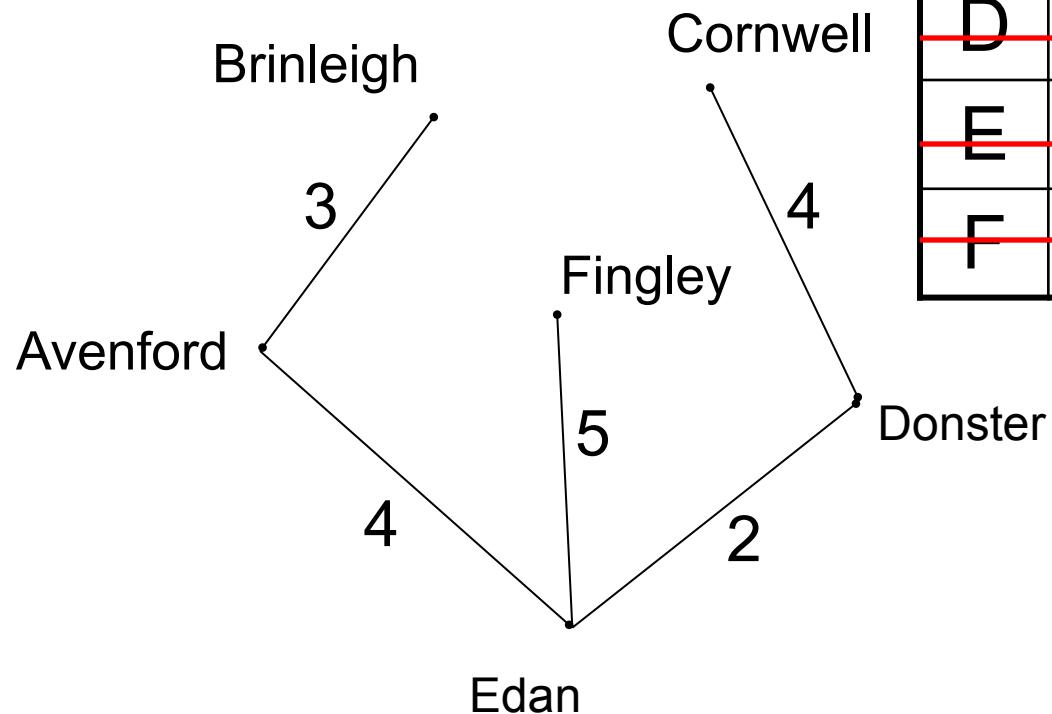
- Label column C “5”
- Delete row C
- Select the smallest uncovered entry in either column A, B, D, E or F (EF, length 5)



	1	2	5	4	3	F
A	-	3	-	-	4	7
B	3	-	5	-	-	8
C	-	5	-	4	-	6
D	-	-	4	-	2	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

FINALLY

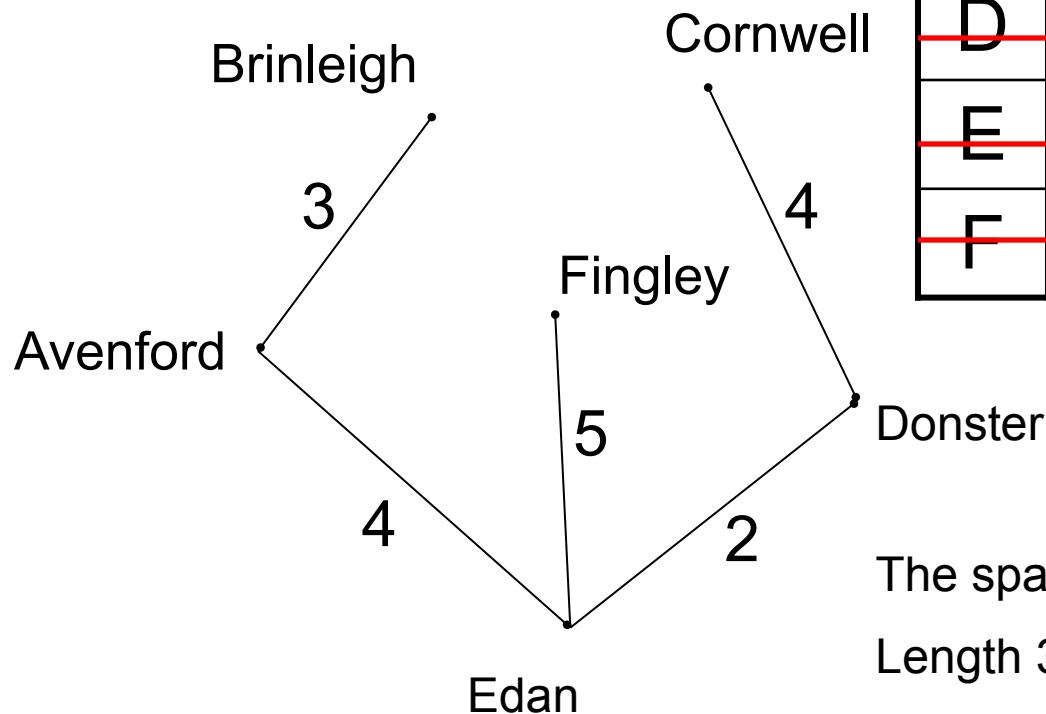
- Label column F “6”
- Delete row F



	1	2	5	4	3	6
A	-	B	C	D	E	F
B	3	-	5	-	-	8
C	5	-	4	-	-	6
D	-	4	-	2	-	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

FINALLY

- Label column F “6”
- Delete row F



	1	2	5	4	3	6
A	-	B	C	D	E	F
B	3	-	5	-	-	8
C	5	-	4	-	-	6
D	-	4	-	2	-	8
E	4	-	-	2	-	5
F	7	8	6	8	5	-

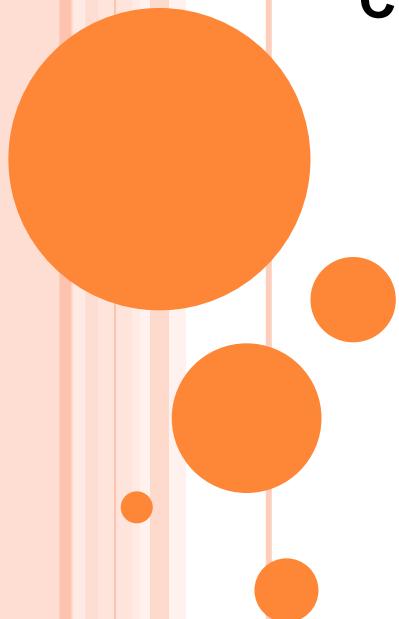
The spanning tree is shown in the diagram
Length $3 + 4 + 4 + 2 + 5 = 18\text{Km}$

Kruskal vs. Prim

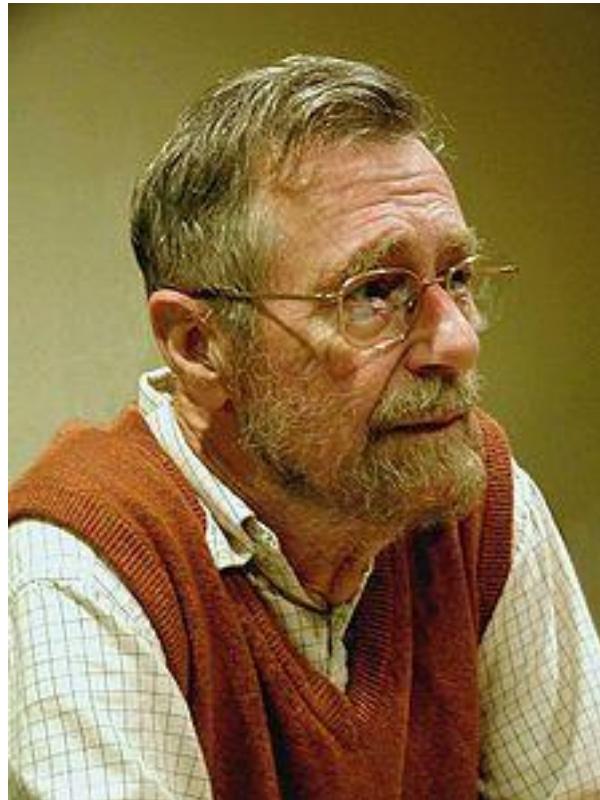
- Both are Greedy algorithms
 - Both take the next minimum edge
 - Both are optimal (find the global min)
- Different sets of edges considered
 - Kruskal – all edges
 - Prim – Edges from Tree nodes to rest of G.
- Both need to check for cycles
 - Kruskal – set containment and union.
 - Prim – Simple boolean.
- Both can terminate early
 - Kruskal – when $|V|-1$ edges are added.
 - Prim – when $|V|$ nodes are added (or $|V|-1$ edges).
- Both are $O(|E| \log |V|)$
 - Prim can be $O(|E| + |V| \log |V|)$ w/ Fibonacci Heaps
 - Prim with an adjacency matrix is $O(|V|^2)$.

DIJKSTRA'S ALGORITHM

Credits: Laksman Veeravagu and Luis Barrera



THE AUTHOR: EDSGER WYBE DIJKSTRA



"Computer Science is no more about computers than astronomy is about telescopes."

<http://www.cs.utexas.edu/~EWD/>



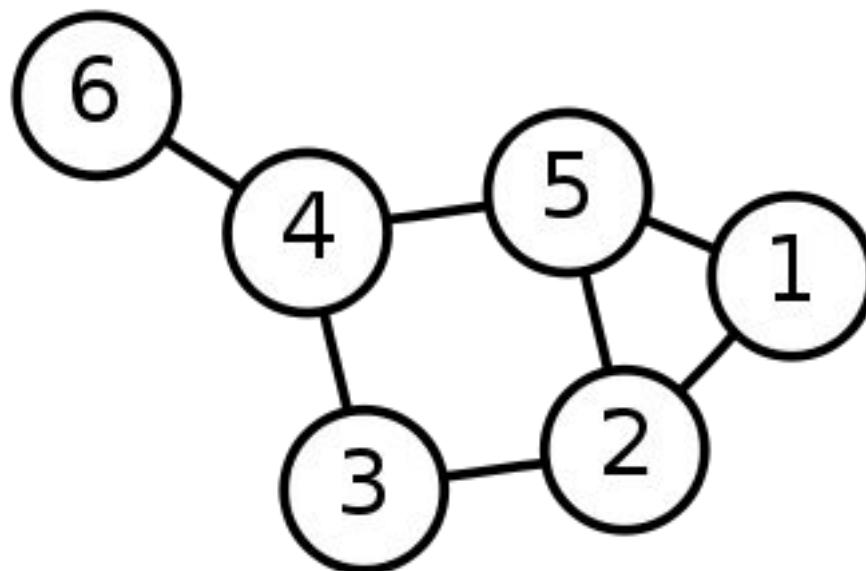
EDSGER WYBE DIJKSTRA

- May 11, 1930 – August 6, 2002
- Received the 1972 A. M. Turing Award, widely considered the most prestigious award in computer science.
- The Schlumberger Centennial Chair of Computer Sciences at The University of Texas at Austin from 1984 until 2000
- Made a strong case against use of the GOTO statement in programming languages and helped lead to its deprecation.
- Known for his many essays on programming.



SINGLE-SOURCE SHORTEST PATH PROBLEM

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex v to all other vertices in the graph.



DIJKSTRA'S ALGORITHM

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices



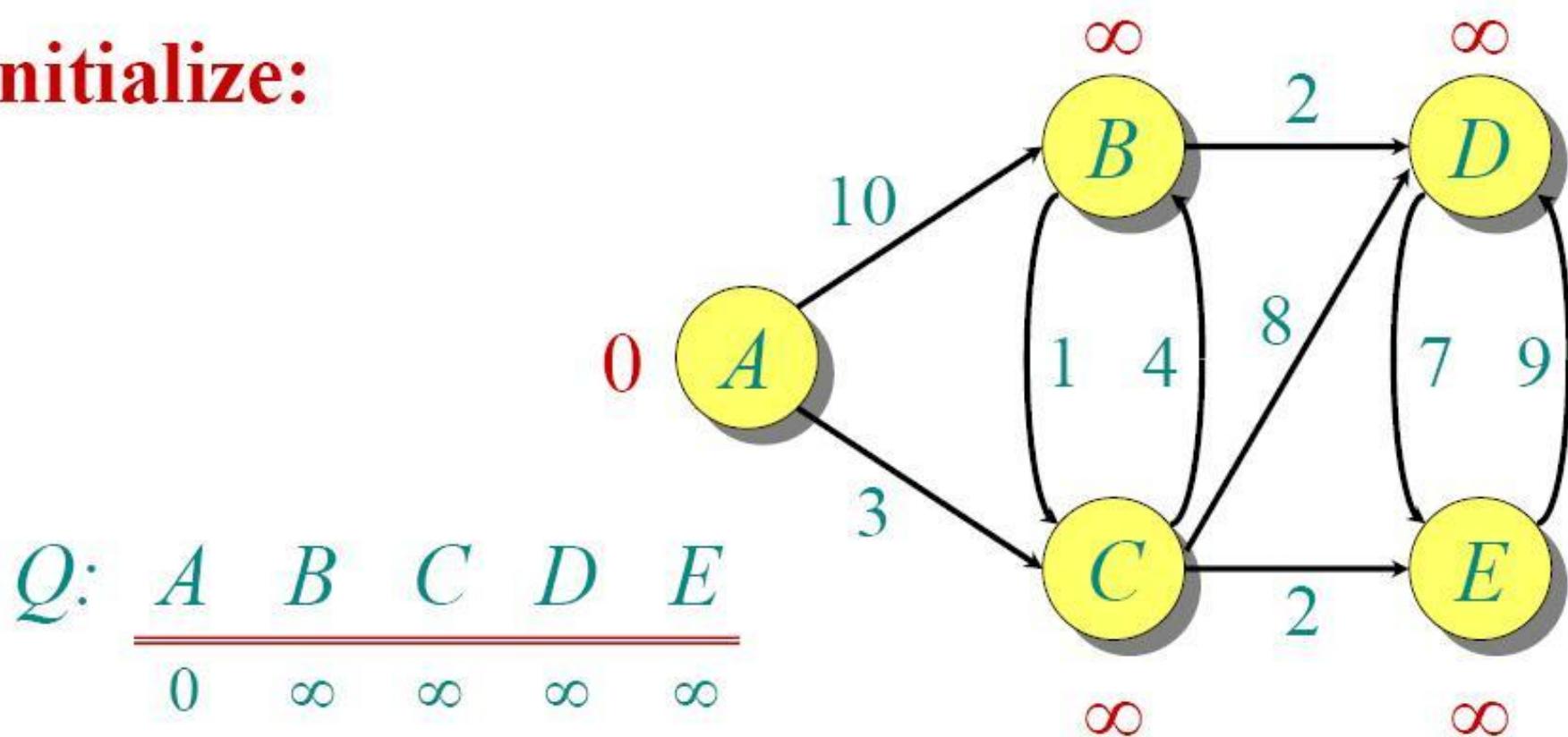
DIJKSTRA'S ALGORITHM - PSEUDOCODE

```
dist[s] ← 0          (distance to source vertex is zero)
for all v ∈ V-{s}
    do dist[v] ← ∞   (set all other distances to infinity)
S←∅                 (S, the set of visited vertices is initially empty)
Q←V                 (Q, the queue initially contains all vertices)
while Q ≠ ∅          (while the queue is not empty)
do u ← mindistance(Q,dist) (select the element of Q with the min. distance)
    S←S ∪ {u}        (add u to list of visited vertices)
    for all v ∈ neighbors[u]
        do if dist[v] > dist[u] + w(u, v)  (if new shortest path found)
              then d[v] ← d[u] + w(u, v)    (set new value of shortest path)
                  (if desired, add traceback code)
return dist
```



DIJKSTRA ANIMATED EXAMPLE

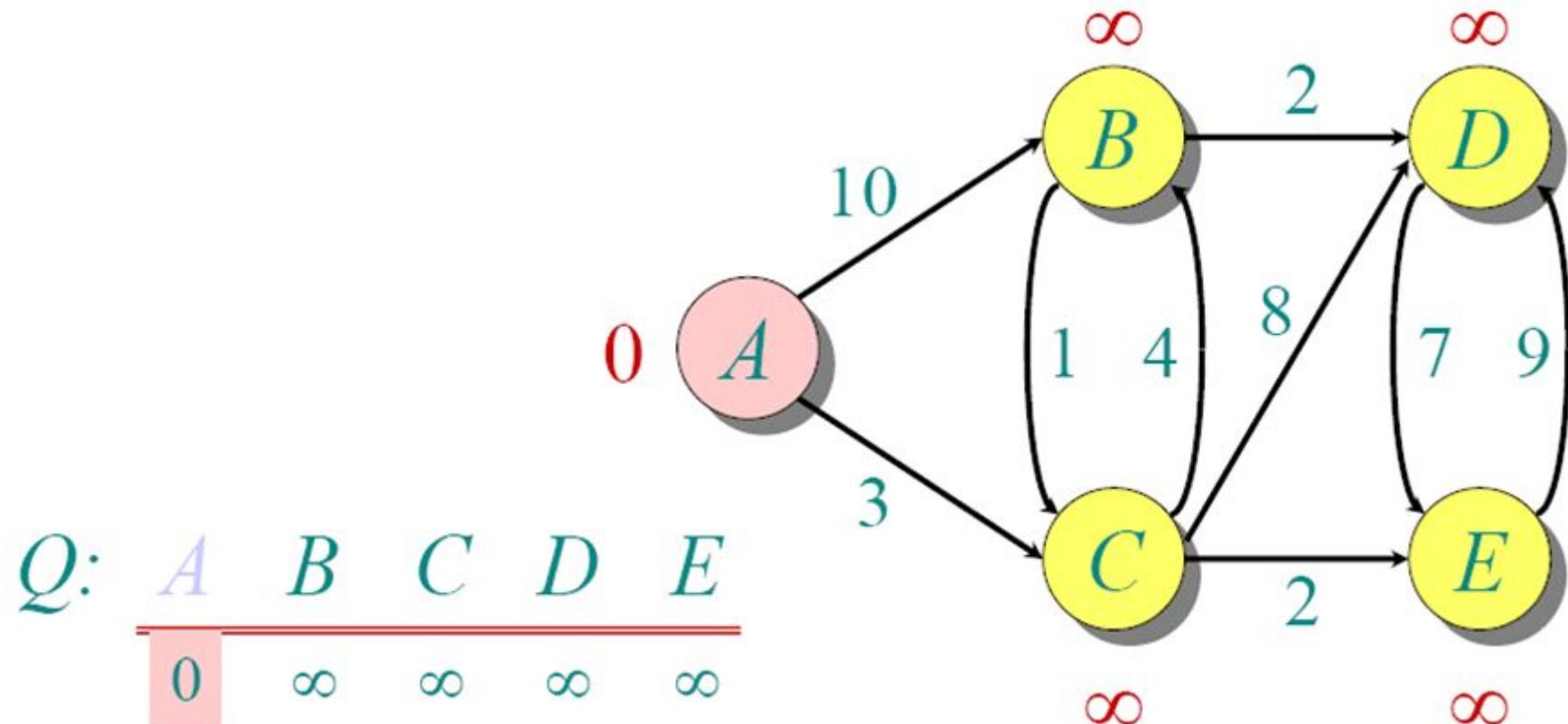
Initialize:



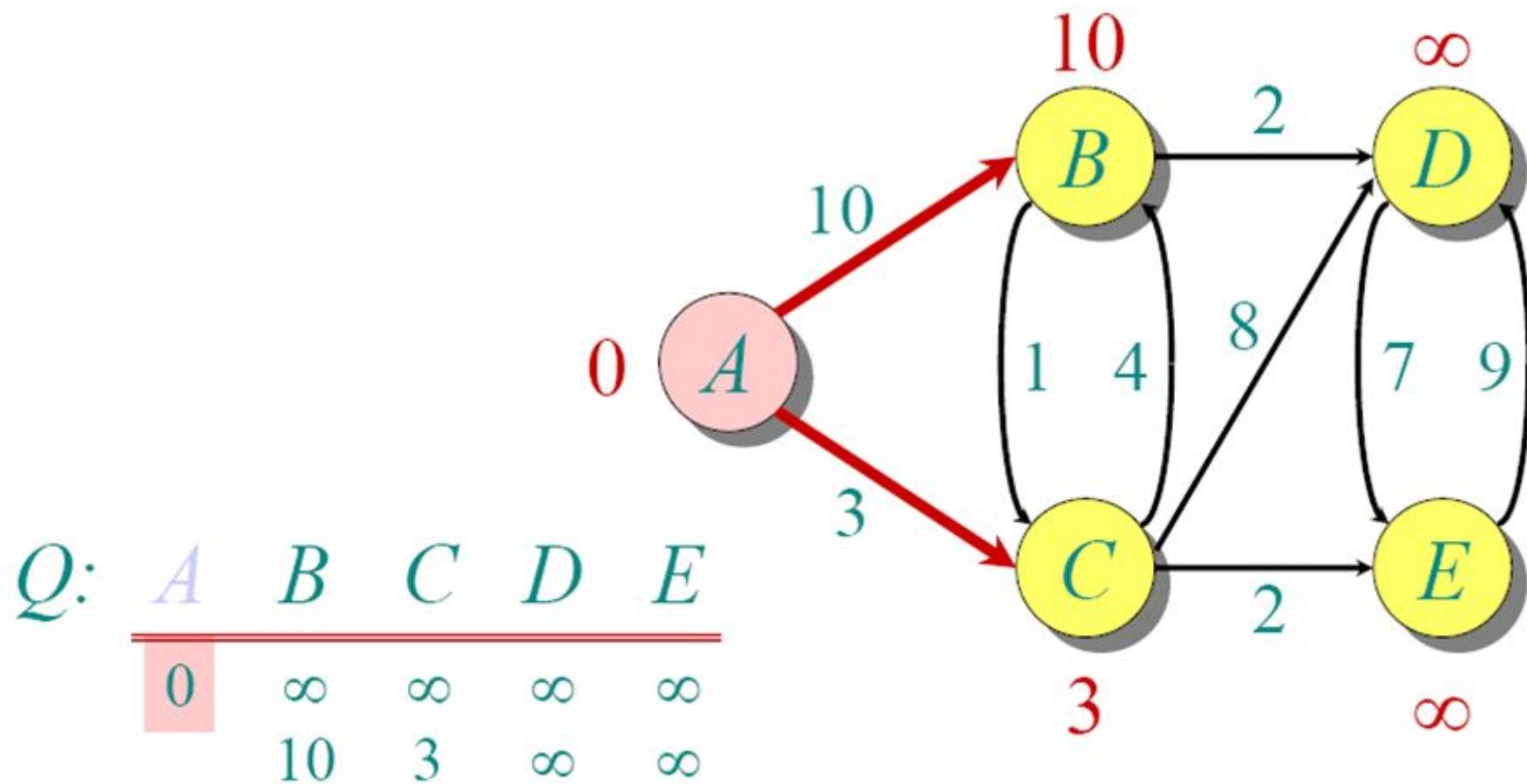
$S: \{\}$



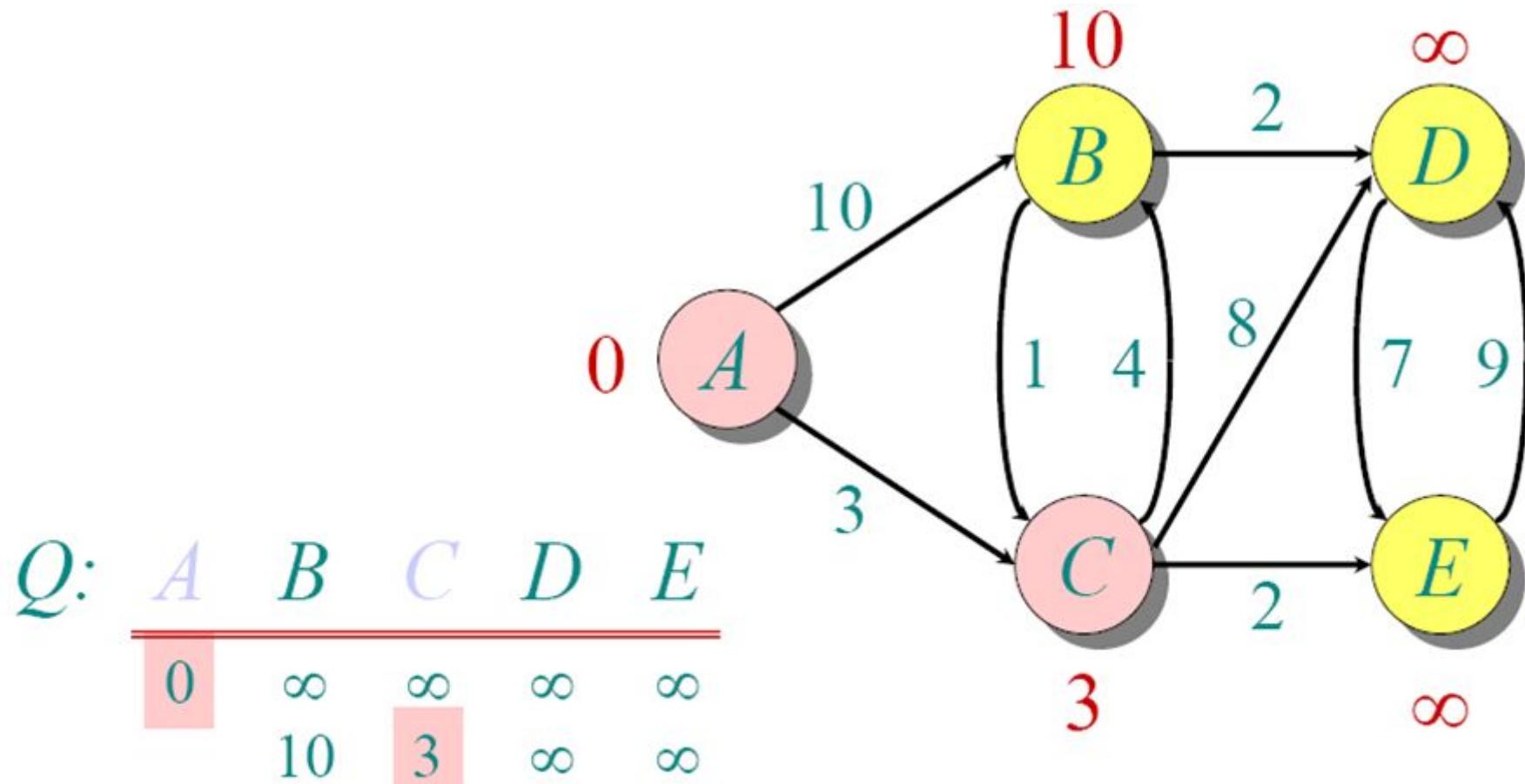
DIJKSTRA ANIMATED EXAMPLE



DIJKSTRA ANIMATED EXAMPLE



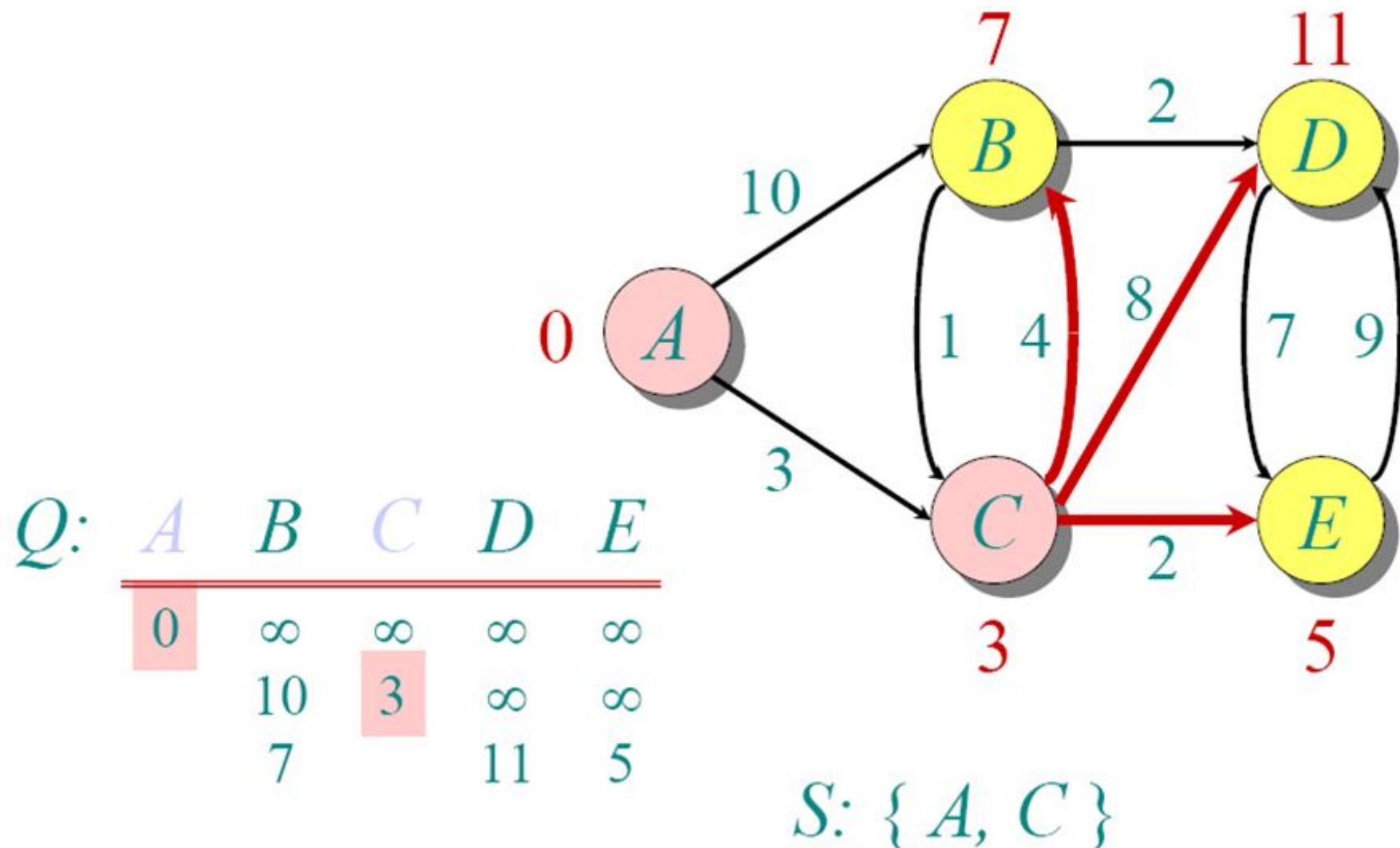
DIJKSTRA ANIMATED EXAMPLE



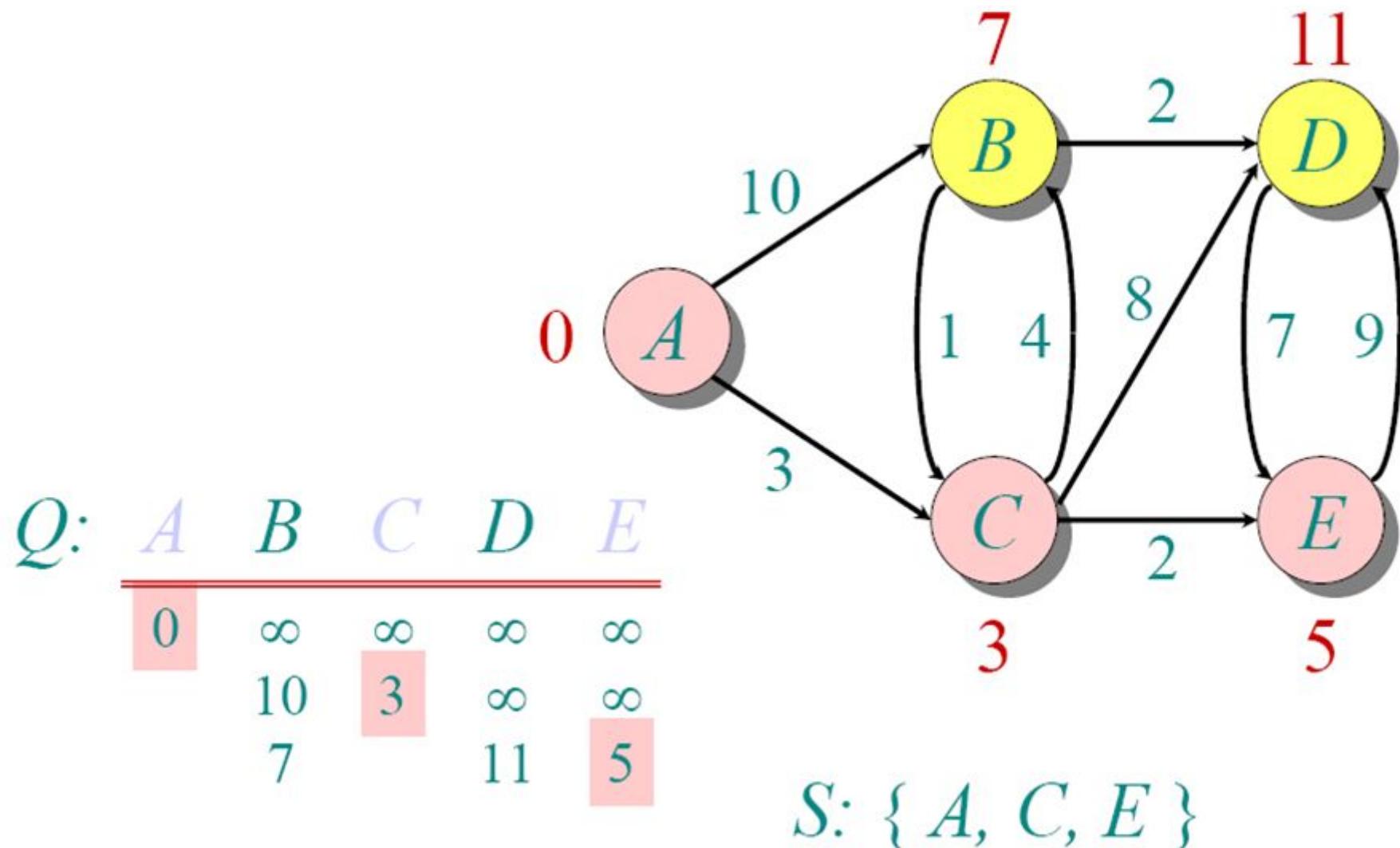
S: { A, C }



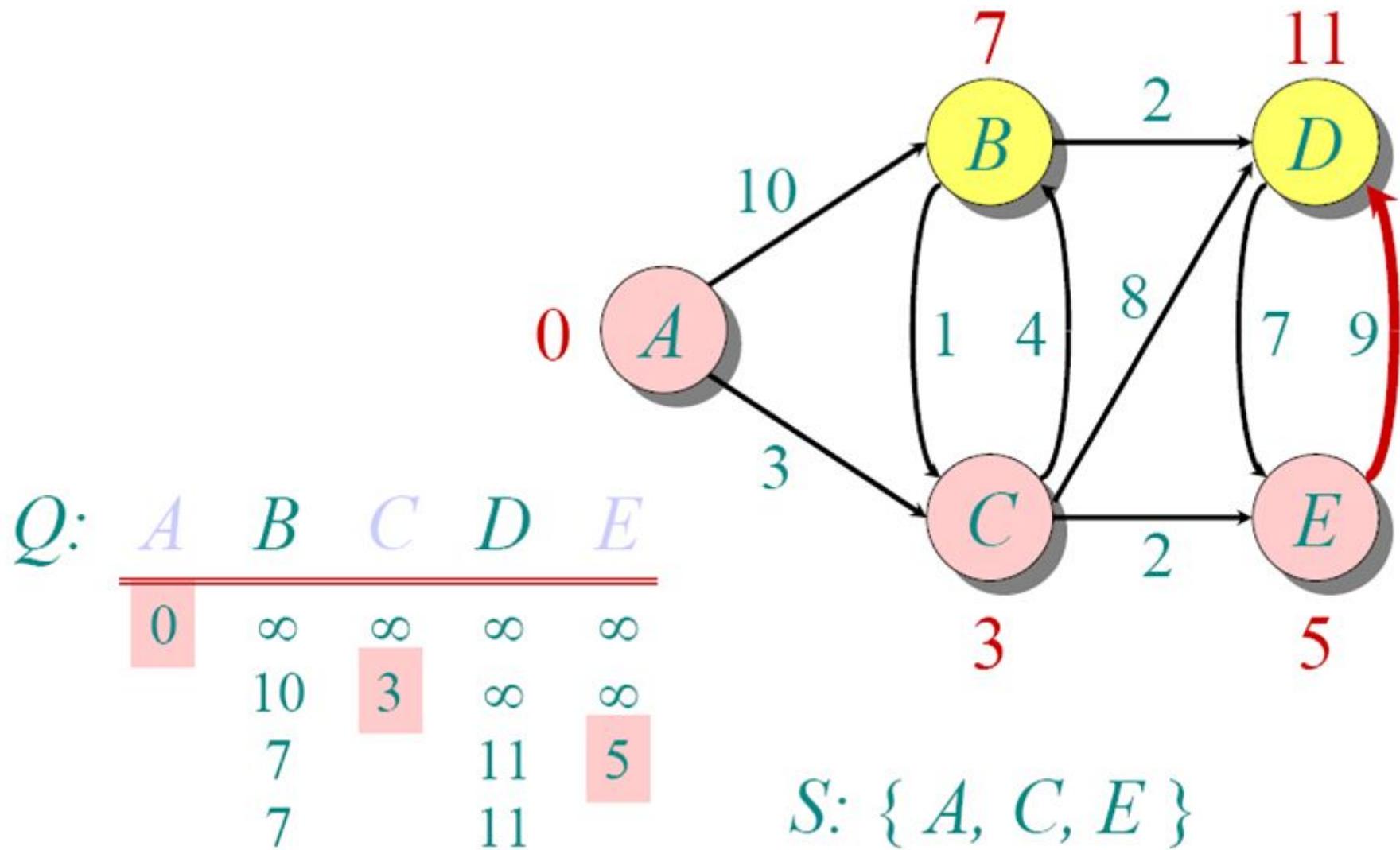
DIJKSTRA ANIMATED EXAMPLE



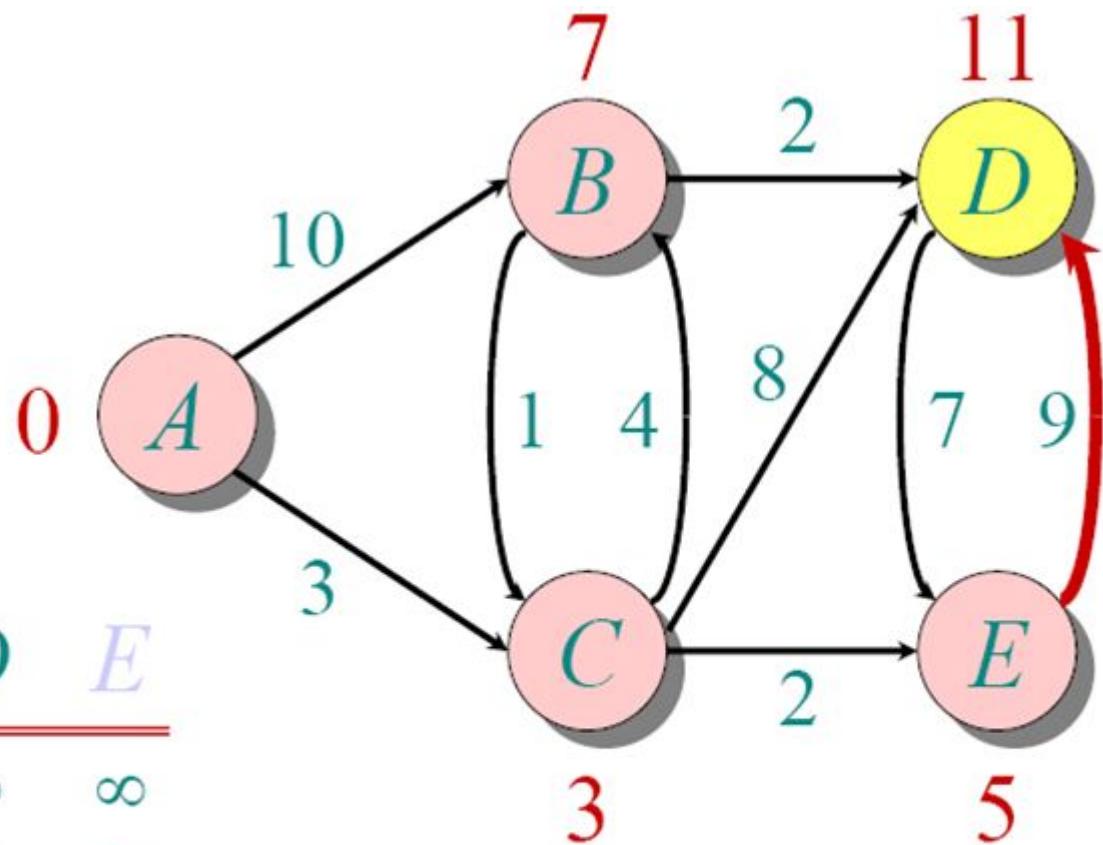
DIJKSTRA ANIMATED EXAMPLE



DIJKSTRA ANIMATED EXAMPLE



DIJKSTRA ANIMATED EXAMPLE



$Q:$

	A	B	C	D	E
0	0	∞	∞	∞	∞
10		3	∞	∞	
7	7		3	11	5

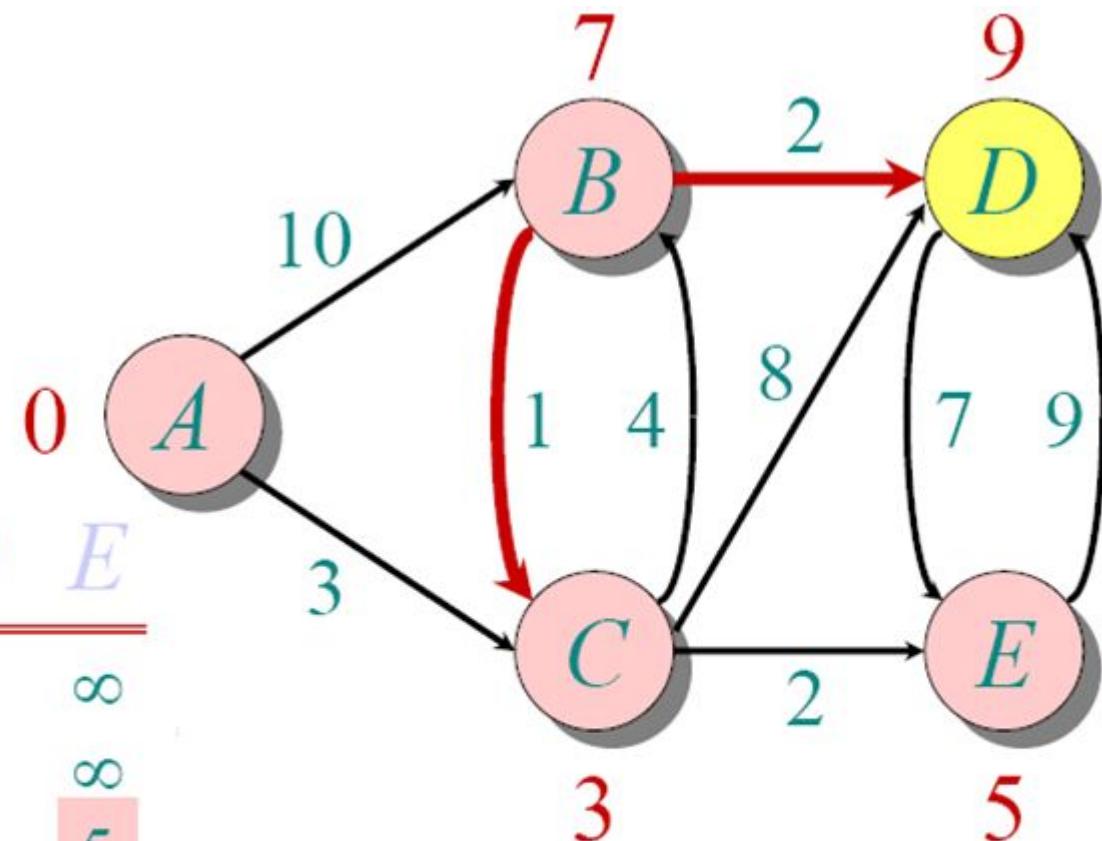
$S: \{ A, C, E, B \}$



DIJKSTRA ANIMATED EXAMPLE

$Q:$

A	B	C	D	E
0	∞	∞	∞	∞
10	3	∞	∞	5
7	7	11	11	9

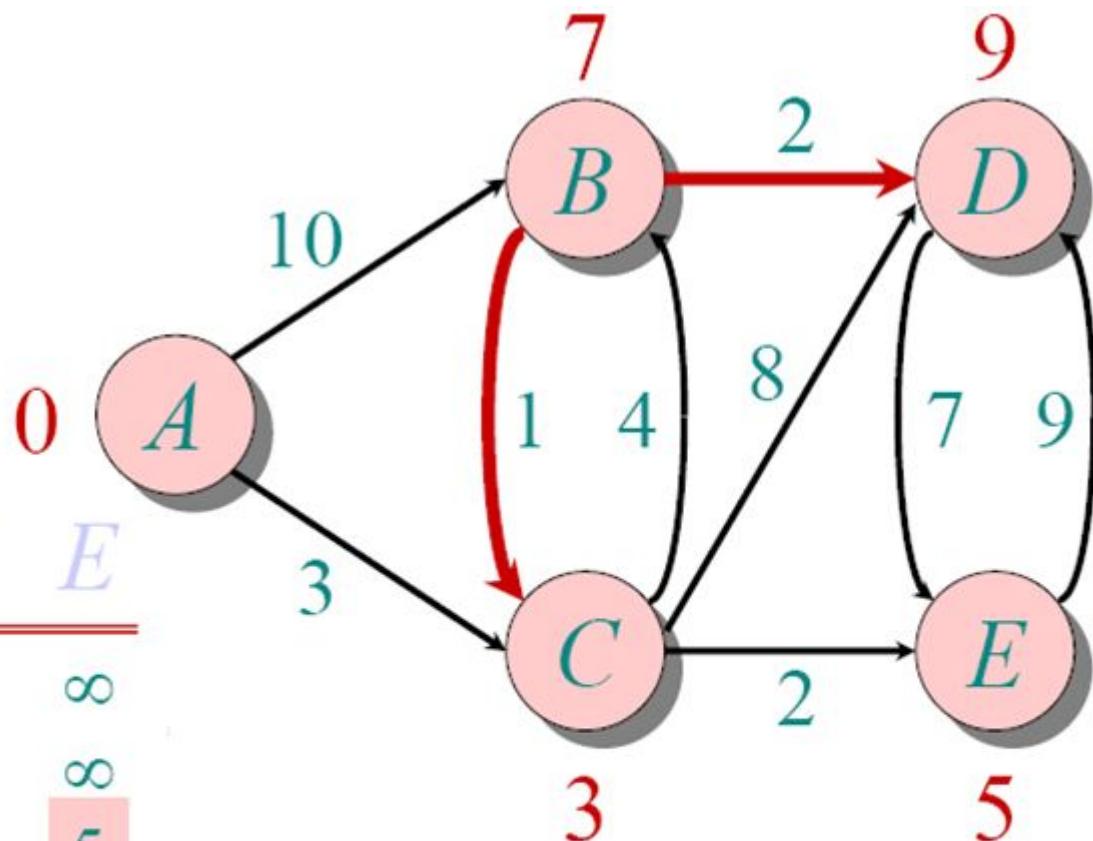


$S: \{ A, C, E, B \}$

DIJKSTRA ANIMATED EXAMPLE

$Q:$

A	B	C	D	E
0	∞	∞	∞	∞
10	3	∞	∞	5
7	7	11	11	9



$S: \{ A, C, E, B, D \}$

IMPLEMENTATIONS AND RUNNING TIMES

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

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

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

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



DIJKSTRA'S ALGORITHM - WHY IT WORKS

- As with all greedy algorithms, we need to make sure that it is a correct algorithm (e.g., it *always* returns the right solution if it is given correct input).
- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively.
- If you can't sleep unless you see a proof, see the second reference or ask us where you can find it.



DIJKSTRA'S ALGORITHM - WHY IT WORKS

- To understand how it works, we'll go over the previous example again. However, we need two mathematical results first:
- **Lemma 1:** Triangle inequality
If $\delta(u,v)$ is the shortest path length between u and v ,
$$\delta(u,v) \leq \delta(u,x) + \delta(x,v)$$
- **Lemma 2:**
The subpath of any shortest path is itself a shortest path.
- The key is to understand why we can claim that anytime we put a new vertex in S , we can say that we already know the shortest path to it.
- Now, back to the example...



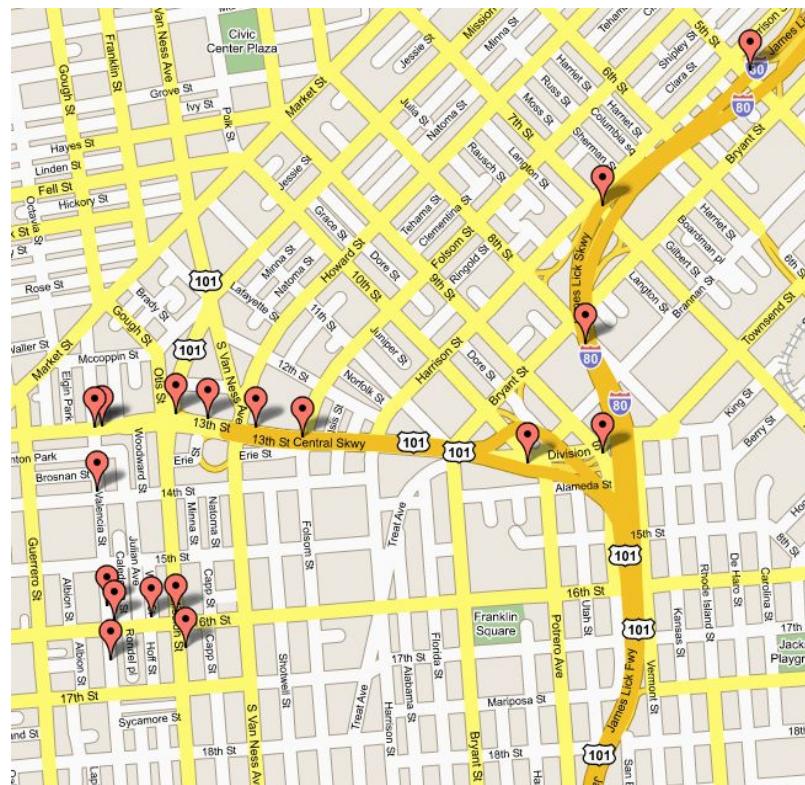
DIJKSTRA'S ALGORITHM - WHY USE IT?

- As mentioned, Dijkstra's algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex u to every vertex using Dijkstra's as it is to calculate the shortest path to some particular vertex v .
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra's algorithm.



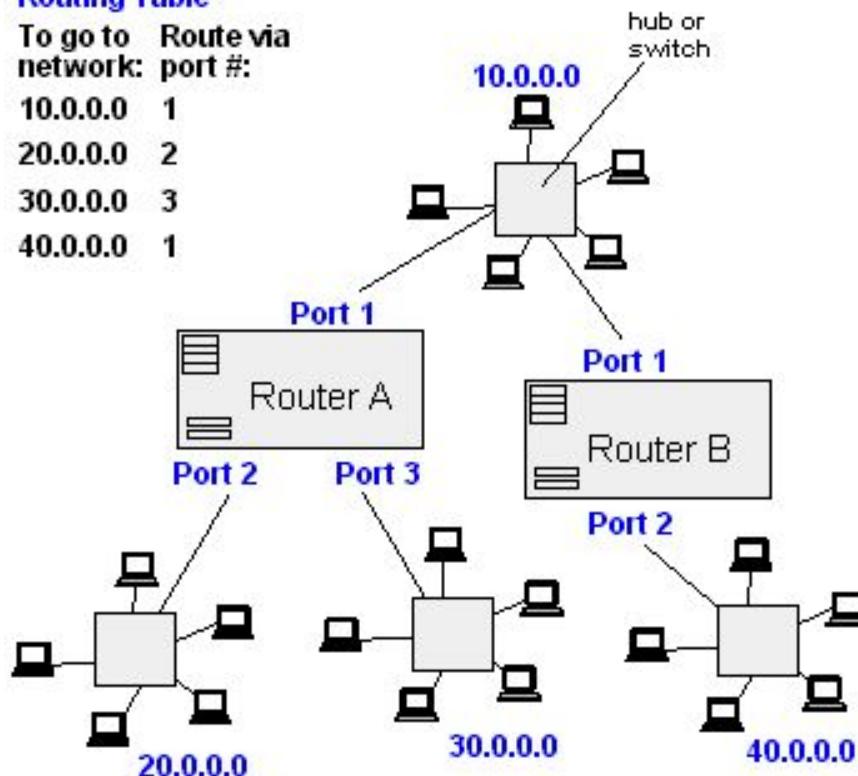
APPLICATIONS OF DIJKSTRA'S ALGORITHM

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



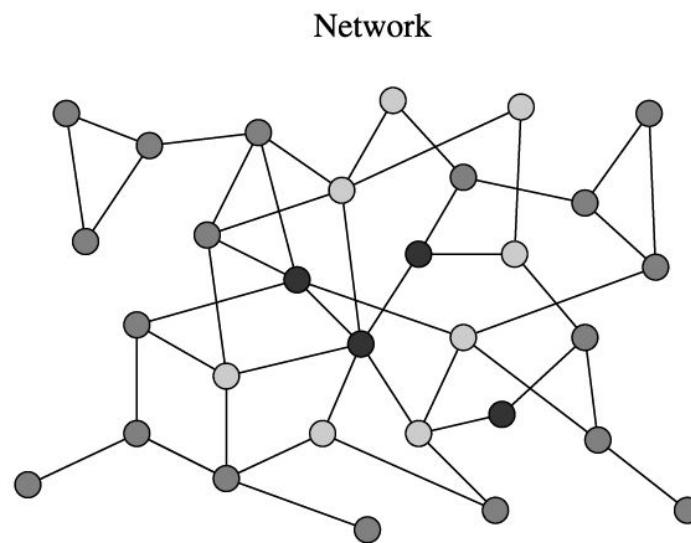
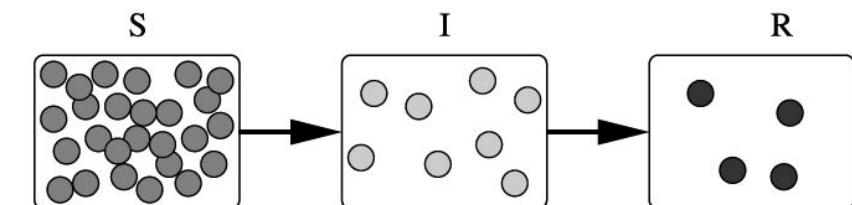
From Computer Desktop Encyclopedia
© 1998 The Computer Language Co., Inc.

Router A Routing Table	
To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1



APPLICATIONS OF DIJKSTRA'S ALGORITHM

- One particularly relevant this week: epidemiology
- Prof. Lauren Meyers (Biology Dept.) uses networks to model the spread of infectious diseases and design prevention and response strategies.
- Vertices represent individuals, and edges their possible contacts. It is useful to calculate how a particular individual is connected to others.
- Knowing the shortest path lengths to other individuals can be a relevant indicator of the potential of a particular individual to infect others.



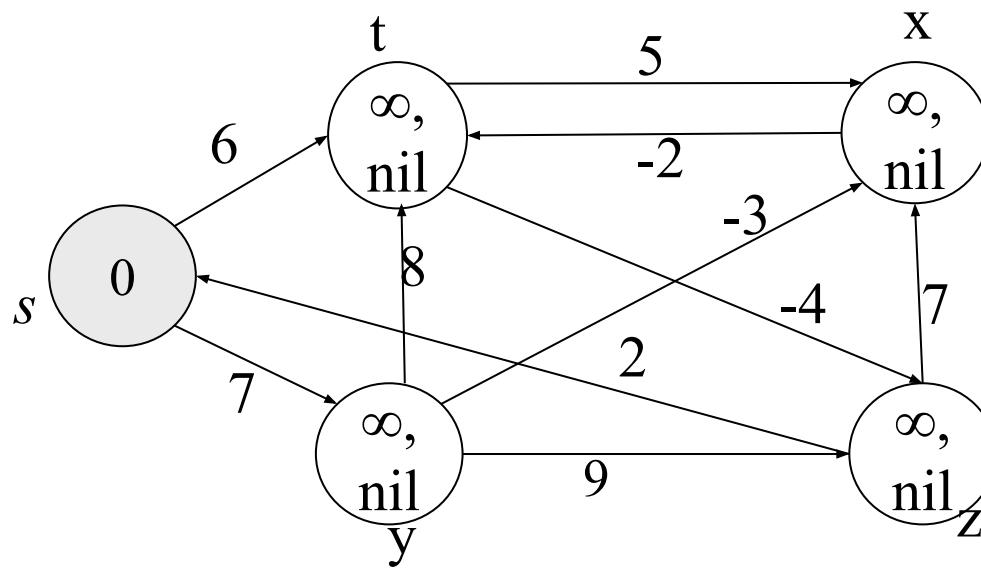
SHORTEST PATH PROBLEM

- Weighted path length (cost): The sum of the weights of all links on the path.
- The single-source shortest path problem: Given a weighted graph G and a source vertex s , find the shortest (minimum cost) path from s to every other vertex in G .

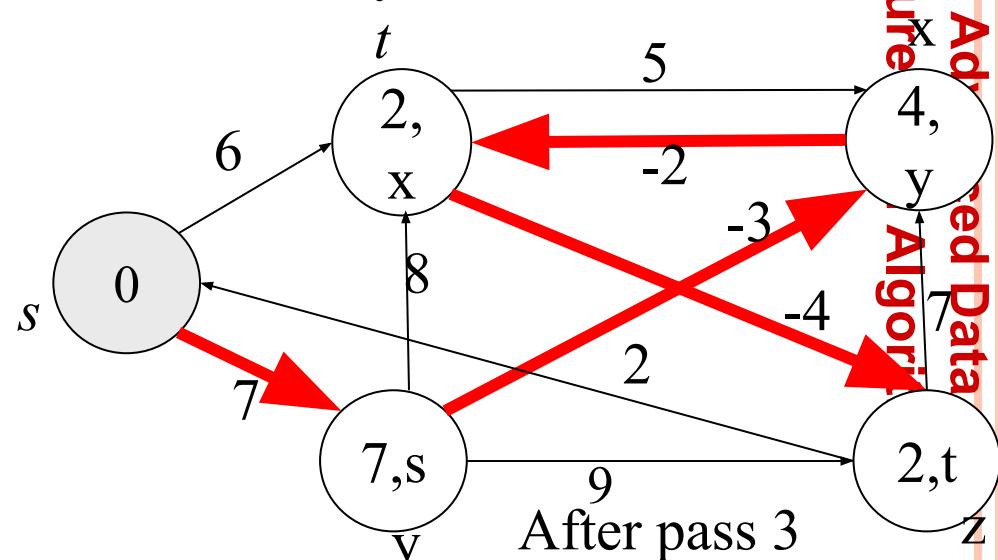
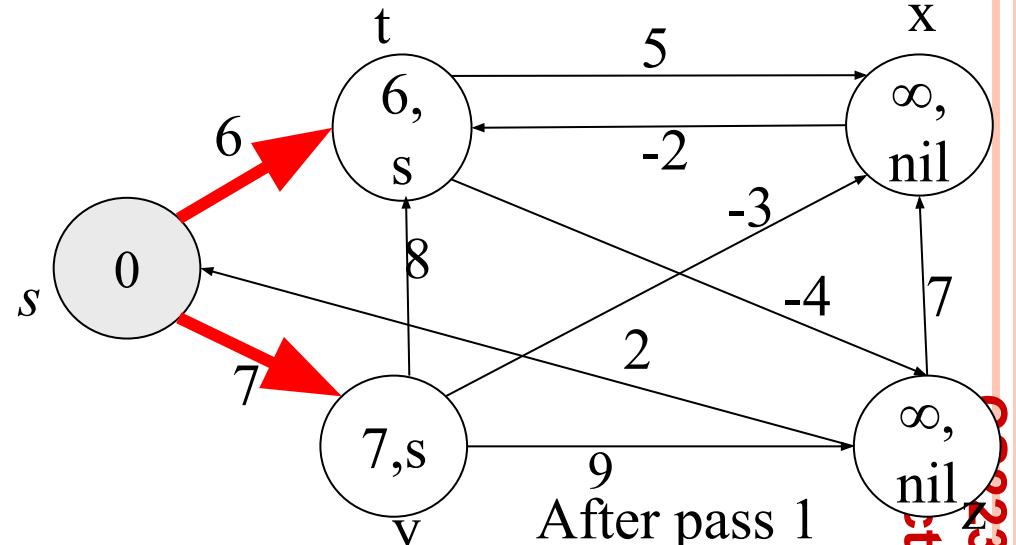
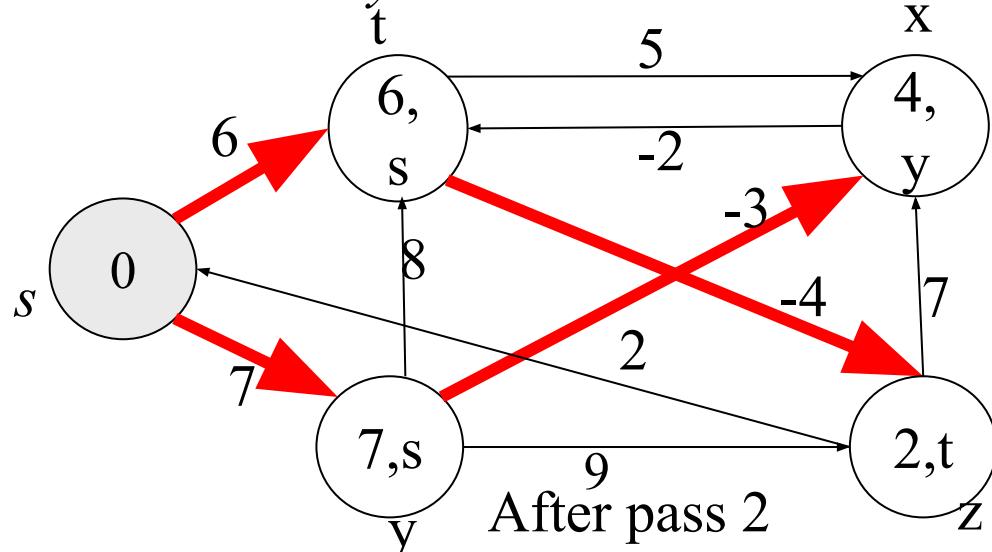
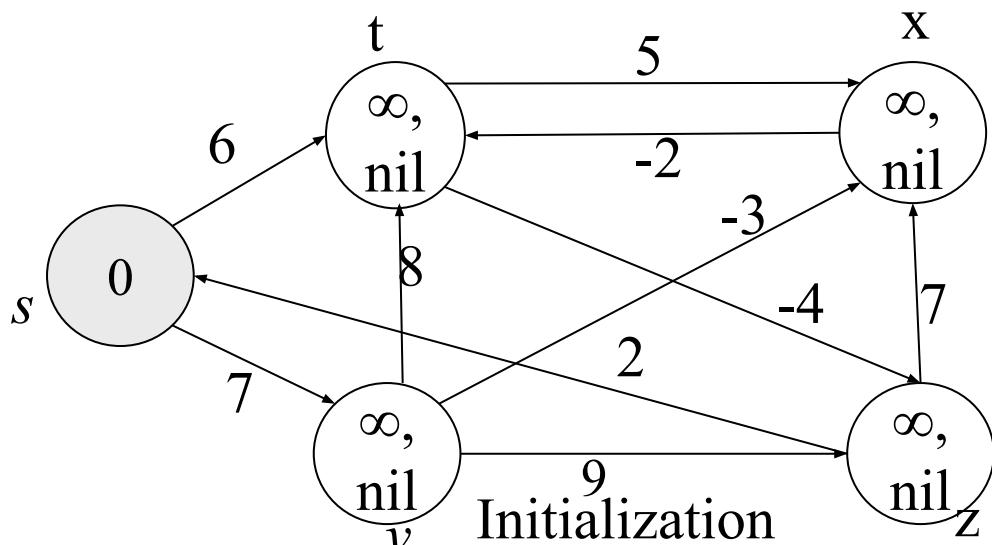
DIFFERENCES

- Negative link weight: The Bellman-Ford algorithm works; Dijkstra's algorithm doesn't.
- Distributed implementation: The Bellman-Ford algorithm can be easily implemented in a distributed way. Dijkstra's algorithm cannot.
- Time complexity: The Bellman-Ford algorithm is higher than Dijkstra's algorithm.

THE BELLMAN-FORD ALGORITHM



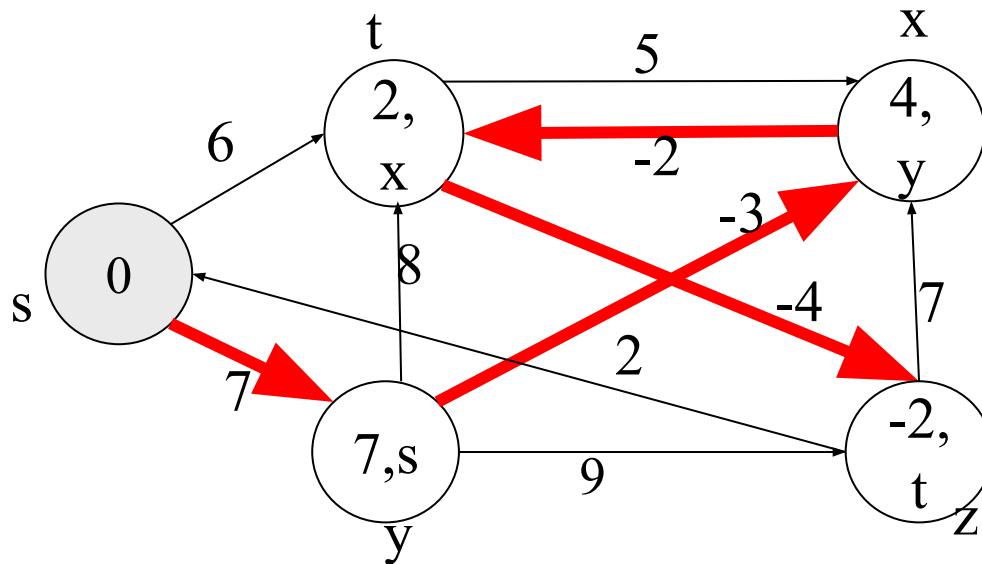
THE BELLMAN-FORD ALGORITHM



The order of edges examined in each pass:

$(t, x), (t, z), (x, t), (y, x), (y, t), (y, z), (z, x), (z, s), (s, t), (s, y)$

THE BELLMAN-FORD ALGORITHM



After pass 4

The order of edges examined in each pass:

(t, x), (t, z), (x, t), (y, x), (y, t), (y, z), (z, x), (z, s), (s, t), (s, y)

THE BELLMAN-FORD ALGORITHM

Bellman-Ford(G, w, s)

1. Initialize-Single-Source(G, s)
2. for $i := 1$ to $|V| - 1$ do
3. for each edge $(u, v) \in E$ do
4. Relax(u, v, w)
5. for each vertex $v \in u.\text{adj}$ do
6. if $d[v] > d[u] + w(u, v)$
7. then return False // there is a negative cycle
8. return True

Relax(u, v, w)

- if $d[v] > d[u] + w(u, v)$
- then $d[v] := d[u] + w(u, v)$
- $\text{parent}[v] := u$

TIME COMPLEXITY

Bellman-Ford(G, w, s)

1. Initialize-Single-Source(G, s) $O(|V|)$
2. for $i := 1$ to $|V| - 1$ do
3. for each edge $(u, v) \in E$ do $O(|V||E|)$
4. Relax(u, v, w)
5. for each vertex $v \in u.adj$ do $O(|E|)$
6. if $d[v] > d[u] + w(u, v)$
7. then return False // there is a negative cycle
8. return True

Time complexity: $O(|V||E|)$

Algorithm	Negative Edge Weights	Positive Edge Weights > 1	Undirected Cycles	Runtime
DFS	✓	✓	✗	$O(n + e)$
BFS	✗	✗	✓	$O(n + e)$ or $O(g^d)$
Dijkstra	✗	✓	✓	$O(e + n \log(n))$
Bellman-Ford	✓	✓	✓	$O(n * e)$

REFERENCES

- Dijkstra's original paper:
E. W. Dijkstra. (1959) *A Note on Two Problems in Connection with Graphs.* Numerische Mathematik, 1. 269-271.
- MIT OpenCourseware, 6.046J Introduction to Algorithms.
<
<http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/> > Accessed 4/25/09
- Meyers, L.A. (2007) Contact network epidemiology: Bond percolation applied to infectious disease prediction and control. *Bulletin of the American Mathematical Society* 44: 63-86.
- Department of Mathematics, University of Melbourne. *Dijkstra's Algorithm.*
<<http://www.ms.unimelb.edu.au/~moshe/620-261/dijkstra/dijkstra.html> > Accessed 4/25/09



Dynamic Programming

Introduction

- Divide and conquer
 - Partition the problem into independent subproblems
 - Solve the subproblems recursively
 - Combine the solutions to solve the original problem.
 - **Dynamic programming** solves every subproblem just once and stores the answer in a table.
- **Main approach:** recursive, holds answers to a sub problem in a table, can be used without recomputing.
 - It follows a bottom-up technique by which it start with smaller and hence simplest subinstances.

Dynamic Programming

- A dynamic-programming algorithm **solves each subsubproblem just once and then saves its answer** in a table, thereby avoiding the work of **recomputing** the answer every time it solves each subsubproblem.
- The technique of storing sub-problem solutions is called **memoization**
- Typically the Dynamic programming is applied to optimization problems (**Problems can have many possible solutions**)
- **Steps to delevolp the Dynamic-programming algorithm**
 - . Characterize the structure of an optimal solution.
 - . Recursively define the value of an optimal solution.
 - . Compute the value of an optimal solution, typically in a bottom-up fashion.
 - . Construct an optimal solution from computed information.

Example: Rod cutting

- Dynamic programming to solve a simple problem in deciding where to cut steel rods.
 - Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free.

Serling Enterprises to know the best way to cut up the rods

We are given prices p_i and rods of length i :

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Question: We are given a rod of length n , and want to **maximize** revenue, by cutting up the rod into pieces and selling each of the pieces

Example: Rod cutting

We'll first list the solutions:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

1.) Cut into 2 pieces of length 2: $p_2 + p_2 = 5 + 5 = 10$

2.) Cut into 4 pieces of length 1: $p_1 + p_1 + p_1 + p_1 = 1 + 1 + 1 + 1 = 4$

3-4.) Cut into 2 pieces of length 1 and 3 (3 and 1): $p_1 + p_3 = 1 + 8 = 9$ $p_3 + p_1 = 8 + 1 = 9$

5.) Keep length 4: $p_4 = 9$

6-8.) Cut into 3 pieces, length 1, 1 and 2 (and all the different orders)

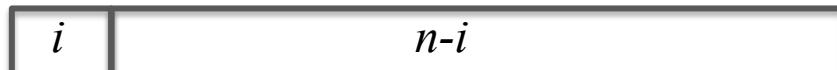
$$p_1 + p_1 + p_2 = 7 \quad p_1 + p_2 + p_1 = 7 \quad p_2 + p_1 + p_1 = 7$$

Total: 8 cases for $n = 4$ ($= 2^{n-1}$). We can slightly reduce by always requiring cuts in non-decreasing order. **But still a lot!**

Example: Rod cutting

Note: We've computed a brute force solution; all possibilities for this simple small example. **But we want more optimal solution!**

One solution:
 recurse on further



What are we doing?

- Cut rod into length i and $n-i$
 - Only remainder $n-i$ can be further cut (recursed)

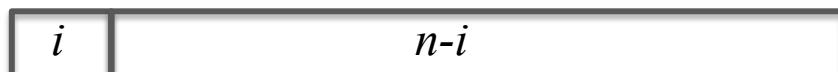
We need to define:

- a.) **Maximum revenue** for log of size n : r_n (that is the solution we want to find).
- b.) **Revenue (price)** for single log of length i : p_i

Example: Rod cutting

Example: If we cut log into length i and $n-i$:

recurse on further



Revenue: $p_i +$
 r_{n-i}

Can be seen by recursing on $n-i$

What are we going to do?

There are many possible choices of i :

$$r_n = \max \left\{ \begin{array}{l} p_1 + r_{n-1} \\ p_1 + r_{n-2} \\ \dots \\ p_n + r_0 \end{array} \right\}$$

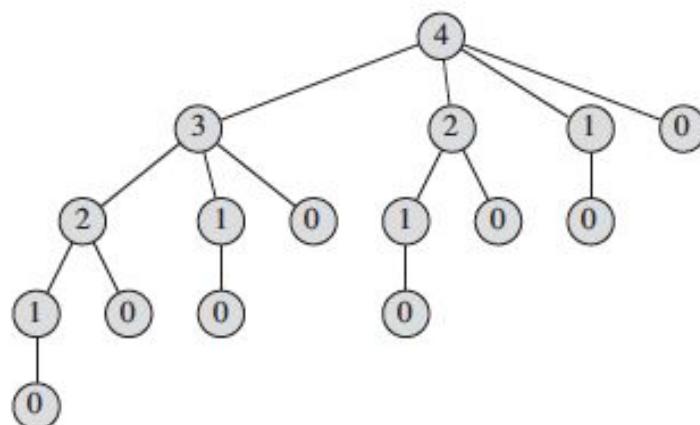
Example: Rod cutting

Recursive (top-down) pseudo code:

```
CUT-ROD( $p, n$ )
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  return  $q$ 
```

Problem? **Slow runtime** (it's essential brute force)!

But why? Cut-rod calls itself repeatedly with the same parameter values (tree):



Example: Rod cutting

- Recursive solution is inefficient, since it repeatedly calculates a solution of the same subproblem (overlapping subproblem).
- Instead, solve each subproblem only once AND save its solution. Next time we encounter the subproblem look it up in a hashtable or an array
- (**Memoization**, recursive top-down solution).
- We save the solution of subproblems Of increasing size (i.e. in order) in an array. Each time we will fall back on solutions that we obtained in previous steps and stored in an array (bottom-up solution).
-

Recursive top-down solution: Cut-Rod with Memoization

- **Step 1 Initialization:**

Creates array for holding memoized results, and initialized to minus infinity. Then calls the main auxiliary function.

MEMOIZED-CUT-ROD(p, n)

```
1 let  $r[0..n]$  be a new array  
2 for  $i = 0$  to  $n$   
3      $r[i] = -\infty$   
4 return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

Step 2: The main auxiliary function, which goes through the lengths, computes answers to subproblems and memoizes if subproblem not yet encountered:

MEMOIZED-CUT-ROD-AUX(p, n, r)

```
1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 
```

Bottom-up solution: **Bottom Up Cut-Rod**

Each time we use previous values from arrays:

BOTTOM-UP-CUT-ROD(p, n)

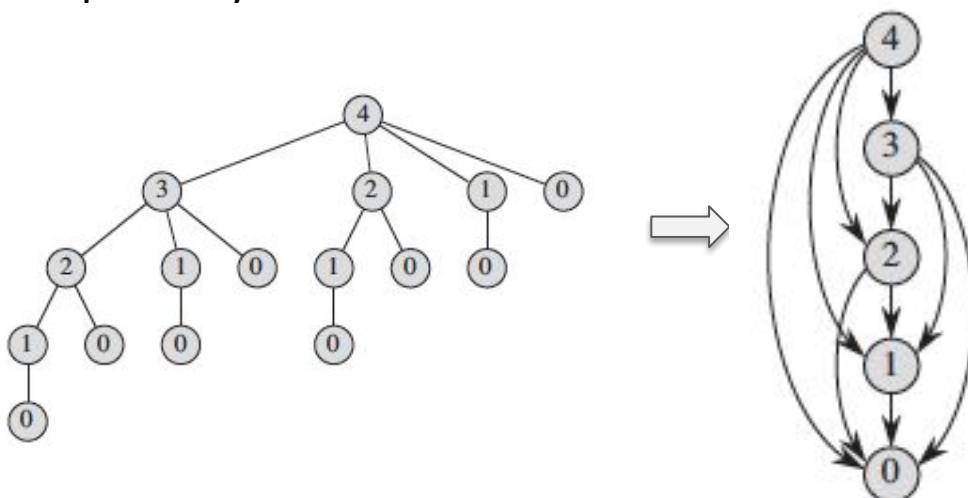
```
1 let  $r[0..n]$  be a new array      } Check if value already known or memoized  
2  $r[0] = 0$   
3 for  $j = 1$  to  $n$   
4      $q = -\infty$   
5     for  $i = 1$  to  $j$   
6          $q = \max(q, p[i] + r[j - i])$  } Compute maximum revenue  
7      $r[j] = q$                  if it hasn't already been  
8 return  $r[n]$                   computed.  
                                } Saving value
```

Run time:

Run time: For bottom-up and top-down approach
 $O(n^2)$

We can also view the subproblems encountered in graph form.

- We reduce the previous tree that included all the subproblems repeatedly



- each vertex represents subproblem of a given size
- Vertex label: subproblem size
- Edges from x to y : We need a solution for subproblem x when solving subproblem y

Run time:

Run time: Can be seen as number of edges $O(n^2)$

Note: Run time is a combination of number of items in table (n) and work per item (n).
The work per item because of the max operation (needed even if the table is filled
And we just take values from the table) is proportional to n as in the number of edges
in graph.

- Example-2: Fibonacci numbers
- Divide problem into two subproblems to calculate (n-1)th and (n-2)th Fibonacci numbers

Problem:

Let's consider the calculation of **Fibonacci** numbers:

$$F(n) = F(n-2) + F(n-1)$$

with seed values $F(1) = 1$, $F(2) = 1$

or

$$F(0) = 0, F(1) = 1$$

What would a series look like:

0, 1, 1, 2, 3, 4, 5, 8, 13, 21, 34, 55, 89, 144, ...

<https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

Example:2

Divide and Conquer

Recursive Algorithm:

```
Fib(n)
{
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

    Return Fib(n-1)+Fib(n-2)
}
```

Recurrence relation:

$$T(n) = T(n-1) + T(n-2) + o(1)$$

Time Complexity: $O(2^{n/2})$

- Using **Dynamic Programming** approach with memoization:
 - Use same recurrence relation and store the results of the problem that solved. i.e Memorize these result in an array

```
void fib () {
    fibresult[0] = 1;
    fibresult[1] = 1;
    for (int i = 2; i<n; i++)
        fibresult[i] = fibresult[i-1] + fibresult[i-2];
}
```

fib(20) = 10946

0	1	14	610
1	1	15	987
2	2	16	1597
3	3	17	2584
4	5	18	4181
5	8	19	6765
6	13	20	10946
7	21		
8	34		
9	55		
10	89		
11	144		
12	233		
13	377		

References

- <https://www.cs.usfca.edu/~galles/visualization/DPFib.html>

Dynamic Programming

02

KNAPSACK

“Given the weights and profits of ‘N’ items, we are asked to put these items in a knapsack that has a capacity ‘C’. The goal is to get the maximum profit from the items in the knapsack”

0-1 KNAPSACK PROBLEM

- A variation of a *bin packing* problem
 - You have a set of items
 - Each item has a weight and a value
 - You have a knapsack with a weight limit
 - **Goal:** Maximize the **value** of the items you put in the knapsack without exceeding the weight limit.
-
- In the 0-1 knapsack problem, we can't *exceed* the weight limit, but the optimal solution may be *less* than the weight limit

Example

Items: { Apple, Orange, Banana, Melon }

Weights: { 2, 3, 1, 4 }

Profits: { 4, 5, 3, 7 }

Knapsack capacity: 5

Different combinations of fruits in the knapsack, such that their total weight is not more than 5:

Apple + Orange (total weight 5) => 9 profit

Apple + Banana (total weight 3) => 7 profit

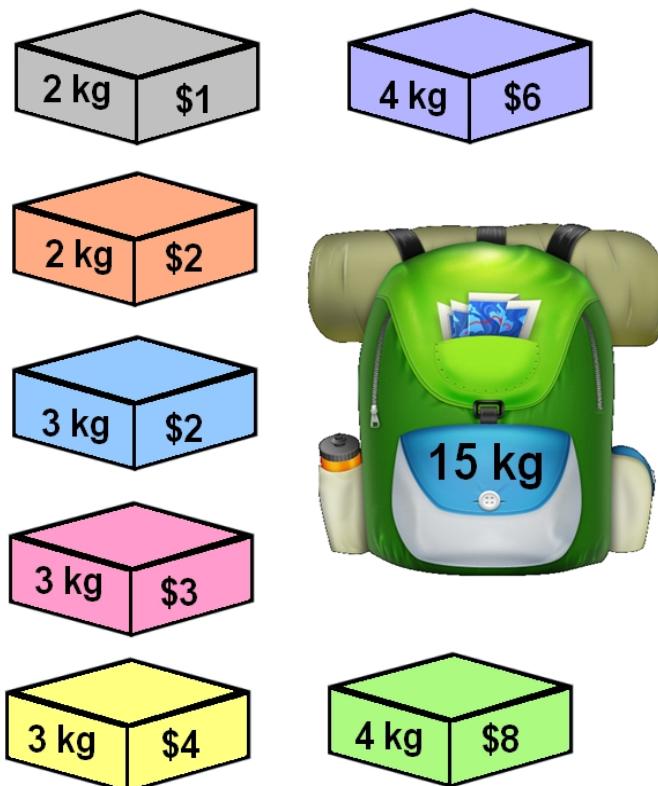
Orange + Banana (total weight 4) => 8 profit

Banana + Melon (total weight 5) => 10 profit

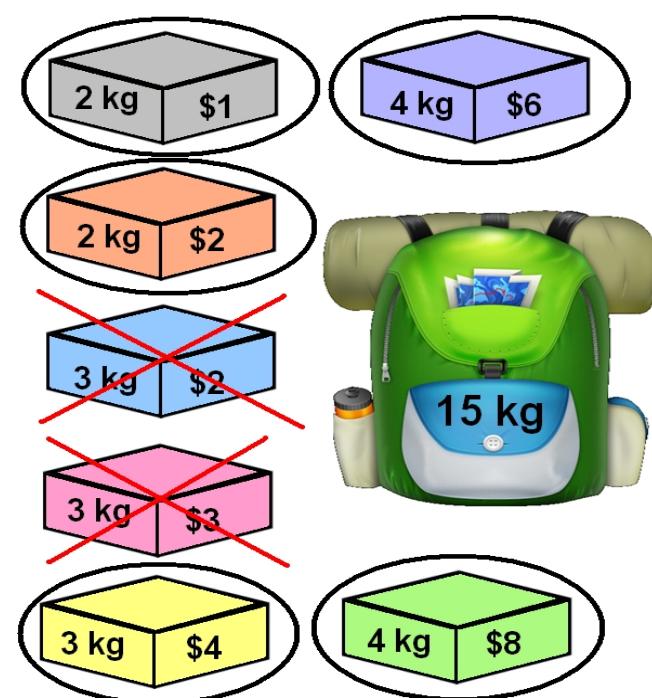
Banana + Melon is the best combination, as it gives us the maximum profit and the total weight does not exceed the capacity

Example

Problem



Solution



Example

- A thief breaks into a house, carrying a knapsack...
 - He can carry up to 25 pounds of loot
 - He has to choose which of N items to steal
 - Each item has some weight and some value
 - “0-1” because each item is stolen (1) or not stolen (0)
 - He has to select the items to steal in order to maximize the value of his loot, but cannot exceed 25 pounds

Problem :

- Given two integer arrays to represent weights and values of 'n' items, we need to find a subset of these items which will give us maximum profit such that their cumulative weight is not more than a given number 'C'
 - Input
 - Capacity 'C'
 - n items with weights w_i and values v_i
 - Output: a set of items S such that
 - the sum of weights of items in S is at most C and the sum of values of items in S is maximized

Solution

The straight forward way:

Example:

$$n = 3$$

$$(p_1, \quad p_2, \quad p_3) = (1, \quad 2, \quad 5)$$

$$(w_1, \quad w_2, \quad w_3) = (2, \quad 3, \quad 4)$$

$$M = 6$$

x_1	x_2	x_3	$\sum w_i x_i$	$\sum p_i x_i$
0	0	0	0	0
0	0	1	4	5
0	1	0	3	2
0	1	1	-	-
1	0	0	2	1
1	0	1	6	<u>6</u> ← solution
1	1	0	5	3
1	1	1	-	-

The complexity is $O(2^n)$

The Dynamic Programming way:

- Sub-problems:
 - Knapsack with a smaller knapsack.

- Recursive relationship

$$f_0(X) = 0$$

$$f_i(X) = \max \{ f_{i-1}(X), p_i + f_{i-1}(X - W_i) \}$$

The Dynamic Programming way:

$f_i(X) = \max$ profit generated from x_1, x_2, \dots, x_i
subject to the capacity X

$$\begin{cases} f_0(X) = 0 \\ f_i(X) = \max \left\{ \underbrace{f_{i-1}(X)}_{}, \underbrace{p_i + f_{i-1}(X - W_i)}_{} \right\} \end{cases}$$

Example:

$$\begin{aligned}(p_1 & \quad p_2 & \quad p_3 & \quad p_4 & \quad p_5 & \quad p_6) = (w_1 & \quad w_2 & \quad w_3 & \quad w_4 & \quad w_5 & \quad w_6) \\ &= (100 & \quad 50 & \quad 20 & \quad 10 & \quad 7 & \quad 3)\end{aligned}$$

$$M = 165$$

Question: to find $f_6(165)$

Use backward approach:

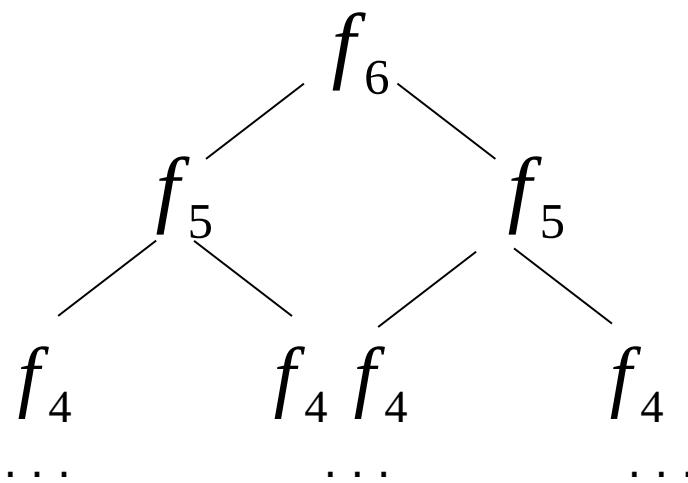
$$f_6(165) = \max\{ f_5(165), f_5(162) + 3 \} = \dots$$

$$f_5(165) = \max\{ f_4(165), f_4(158) + 7 \} = \dots$$

...

The result:

$$(x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6) = (1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1)$$



Therefore, the complexity of 0/1 Knapsack is $O(2^n)$

Example: $M = 6$

	Object 1	Object 2	Object 3	Object 4
p_i	3	4	8	5
w_i	2	1	4	3

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3
2	0	4	4	7	7	7	7
3	0	4	4	7	8	1	1
4	0	4	4	7	9	1	12

Max profit
is 12

By tracing which entries lead to this max-profit solution,
we can obtain the optimal solution - Object 1,2 and 4.

	0	1	...	$j-w_i$...	j	...	M
0	0	0	0	0	0	0	0	0
1								
...								
$i-1$				$f_{i-1}(j-w_i)$		$f_{i-1}(j)$		
i					$+p_i$		$f_i(j)$	
...								
n								

$(M+1)(n+1)$ entries and constant time each $\Rightarrow O(Mn)$

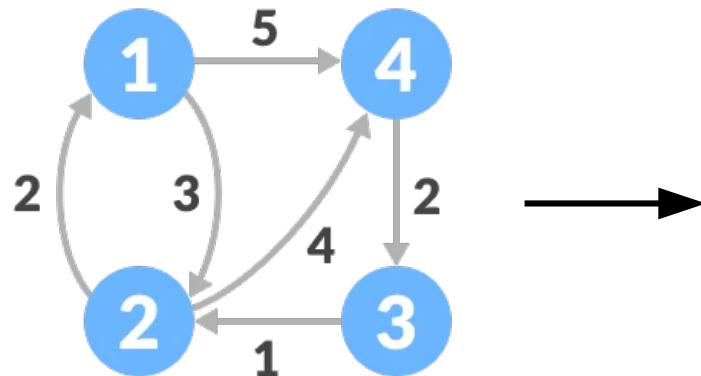
Floyd-Warshall algorithm

- Single-source shortest path in weighted graphs.
 - Bellman-Ford algorithm
 - Dijkstra's algorithm!
- Floyd-Warshall algorithm
 - An “all-pairs” shortest path algorithm
 - Another example of **dynamic programming**

Floyd's Algorithm: All pairs shortest path

- All pairs shortest path
 - **The problem:** find the shortest path between every pair of vertices of a graph.
 - The graph: may contain negative edges but no negative cycles.
 -
 - A representation: a weight matrix where
 $W(i,j)=0$ if $i=j$.
 $W(i,j)=-l$ if there is no edge between i and j .
 $W(i,j)$ =“weight of edge”
 -

Example:



Step 1. Create a matrix dimension $N \times N$ and Each cell $A[i][j]$ is filled with the distance from the i th vertex to the j th vertex.

$$A^0 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 5 \\ 2 & 0 & \infty & 4 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

Step:2: Create a matrix A^1 using matrix A^0 .

Let k be the intermediate vertex

In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k .

$$\rightarrow A^1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & 0 & & \\ \infty & 0 & & \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 8 \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

A direct distance from the source to the destination is greater than the path through the vertex k ,

\rightarrow if $(A[i][j] > A[i][k] + A[k][j])$

Step:3: Create a matrix A^2 using matrix A^1

k is the second vertex (i.e. vertex 2)

In this step, k is vertex 2. We calculate the distance from source vertex to destination vertex through this vertex k.

$$A^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & & 1 & 0 & \\ 4 & & \infty & & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & \infty & \\ 2 & 0 & 9 & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & 2 & 0 & \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix}$$

Step:4: Create a matrix A^3 using matrix A^2

k is the second vertex (i.e. vertex 3)

In this step, k is vertex .

Step:5: Create a matrix A^4 using matrix A^3

k is the second vertex (i.e. vertex 4)

In this step, k is vertex .

$$A^4 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & & 5 \\ 2 & 0 & & 4 \\ 3 & & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{bmatrix}$$

A4 gives the shortest path between each pair of vertices.

Floyd's Algorithm:

Floyd-Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles)

Pseudocode for this basic version follows:

```
1 let dist be a |V| × |V| array of minimum distances
  initialized to  $\infty$  (infinity)
2 for each edge (u,v)
3   dist[u][v]  $\leftarrow w(u,v)$  // the weight of the edge (u,v)
4 for each vertex v
5   dist[v][v]  $\leftarrow 0$ 
6 for k from 1 to |V|
7   for i from 1 to |V|
8     for j from 1 to |V|
9       if dist[i][j] > dist[i][k] + dist[k][j]
10        dist[i][j]  $\leftarrow$  dist[i][k] + dist[k][j]
11      end if
```

Subproblems ?

How can we define the shortest distance $d_{i,j}$ in terms of “smaller” problems?

Elements of dynamic programming

- Big problems break up into little problems.
 - eg, Shortest path with at most k edges
- The optimal solution of a problem can be expressed in terms of optimal solutions of smaller sub-problems.

eg, $d^{(k)}[b] \leftarrow \min\{ d^{(k-1)}[b], \min_a \{d^{(k-1)}[a] + \text{weight}(a,b)\} \}$

optimal sub-structure

The sub-problems overlap a lot.

- Lots of different entries of $d^{(k)}$ ask for $d^{(k-1)}[a]$.
“We can save time by solving a sub-problem just once and storing the answer”

Floyd-Warshall Algorithm

n = no of vertices

A = matrix of dimension n*n

for k = 1 to n

 for i = 1 to n

 for j = 1 to n

$$A^k[i, j] = \min (A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j])$$

return A

- Time Complexity
 - There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.
- Space Complexity
 - The space complexity of the Floyd-Warshall algorithm is $O(n^2)$

Floyd Warshall Algorithm Applications

- To find the shortest path in a directed graph
- To find the transitive closure of directed graphs
- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite

Dynamic programming Longest Common Subsequence

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory for later use
- More efficient than “*brute-force methods*”, which solve the same subproblems over and over again

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: $X = \{A\ B\ C\ B\ D\ A\ B\}$, $Y = \{B\ D\ C\ A\ B\ A\}$

Longest Common Subsequence:

$X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ B\ A$

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- if $|X| = m, |Y| = n$, then there are 2^m subsequences of x ; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is $O(n 2^m)$
- Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.
- Subproblems: “find LCS of pairs of *prefixes* of X and Y ”

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i , Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0, 0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i, 0] = 0$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- When we calculate $c[i, j]$, we consider two cases:
- First case:** $x[i] = y[j]$: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{j-1} , plus 1

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** $x[i] \neq y[j]$
- As symbols don't match, our solution is not improved, and the length of $\text{LCS}(X_i, Y_j)$ is the same as before (i.e. maximum of $\text{LCS}(X_i, Y_{j-1})$ and $\text{LCS}(X_{i-1}, Y_j)$)

Why not just take the length of $\text{LCS}(X_{i-1}, Y_{j-1})$?
11/05/21 8

LCS Length Algorithm

LCS-Length(X, Y)

1. $m = \text{length}(X)$ // get the # of symbols in X
2. $n = \text{length}(Y)$ // get the # of symbols in Y
3. for $i = 1$ to m $c[i,0] = 0$ // special case: Y_0
4. for $j = 1$ to n $c[0,j] = 0$ // special case: X_0
5. for $i = 1$ to m // for all X_i
6. for $j = 1$ to n // for all Y_j
7. if ($X_i == Y_j$)
8. $c[i,j] = c[i-1,j-1] + 1$
9. else $c[i,j] = \max(c[i-1,j], c[i,j-1])$
10. return c

LCS Example

We'll see how LCS algorithm works on the following example:

- $X = ABCB$
- $Y = BDCAB$

What is the Longest Common Subsequence of X and Y?

$$\text{LCS}(X, Y) = BCB$$

$X = A \ B \ C \ B$

$Y = \quad B \ D \ C \ A \ B$

LCS Example (0)

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi					
1	A					
2	B					
3	C					
4	B					

$$X = ABCB; \ m = |X| = 4$$

$$Y = BDCAB; \ n = |Y| = 5$$

Allocate array c[5,4]

LCS Example (1)

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0				
2	B	0				
3	C	0				
4	B	0				

for $i = 1$ to m $c[i,0] = 0$

for $j = 1$ to n $c[0,j] = 0$

LCS Example (2)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (3)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	
2	B	0				
3	C	0				
4	B	0				

```

if( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (4)

ABCB
BDCA₄B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

```

if( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (5)

ABCB
BDCA_B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (6)

A₀B₁C₂B₃
B₀D₁C₂A₃B₄

i	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1				
3	C	0					
4	B	0					

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (7)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (8)

ABC
BDCA B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

```

if( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (10)

ABC
BD CAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (11)

ABC
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	2
3	C	0	1	1	2	
4	B	0				

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (12)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (13)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCA B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

```

if( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (15)

ABCB
BDCA_B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array $c[m,n]$
- So what is the running time?

$O(m*n)$

since each $c[i,j]$ is calculated in constant time, and there are $m*n$ elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each $c[i,j]$ depends on $c[i-1,j]$ and $c[i,j-1]$
or $c[i-1, j-1]$

For each $c[i,j]$ we can say how it was acquired:

2	2
2	3

For example, here
 $c[i,j] = c[i-1,j-1] + 1 = 2+1=3$

How to find actual LCS - continued

- Remember that

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- So we can start from $c[m, n]$ and go backwards
- Whenever $c[i, j] = c[i - 1, j - 1] + 1$, remember $x[i]$ (because $x[i]$ is a part of LCS)
- When $i=0$ or $j=0$ (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

Finding LCS (2)

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): B C B

LCS (straight order): B C B
(this string turned out to be a palindrome)

ADSA – Unit 6

Advanced Topics

Dr. Sreeja S R

Topics to be covered

- Network Flows
- Randomized Algorithms
- Computational Complexity
 - NP-completeness
 - Polytime reductions

Network Flows

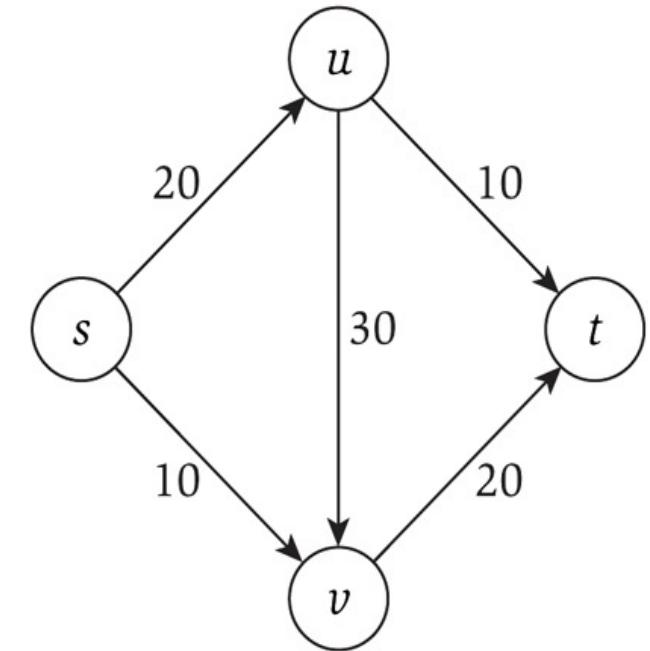
Network Flows

- A wide variety of engineering and management problems involve optimization of network flows – that is, how objects move through a network.
- Examples include
 - coordination of trucks in a transportation system
 - routing of packets in a communication network
 - sequencing of legs for air travel.

Flow Networks

A *flow network* is a directed graph $G(V, E)$

- Each edge $e \in E$ has a capacity $c(e) > 0$.
- There is a single *source* node $s \in V$.
- There is a single *sink* node $t \in V$.
- Nodes other than s and t are *internal*.

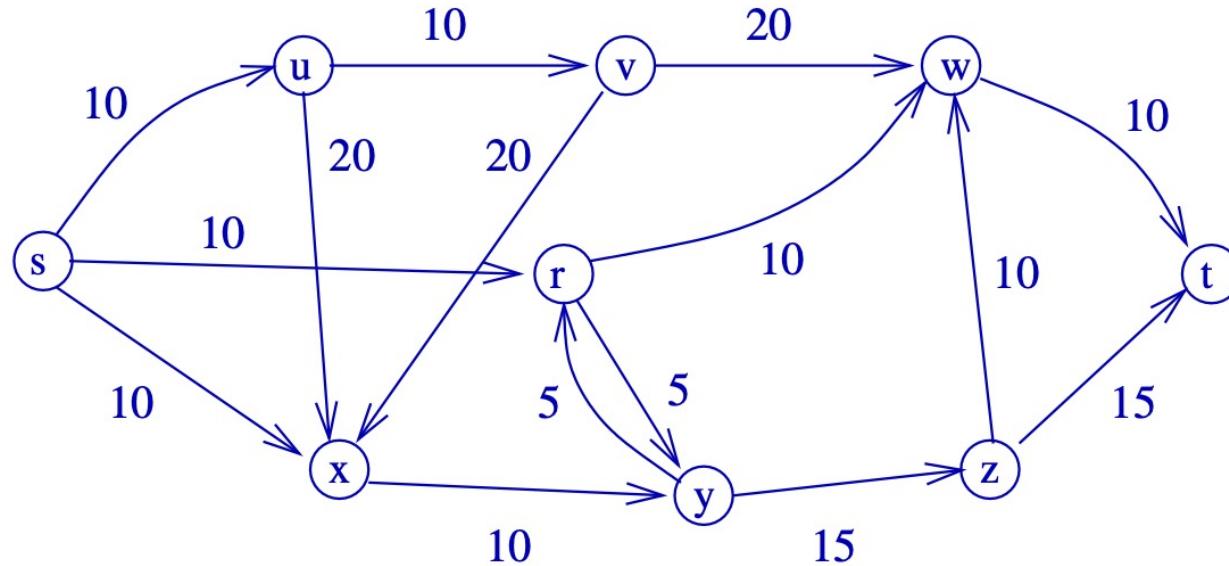


A flow network with source s and sink t . The numbers next to the edges are the capacities.

Note: A capacity function $c: V \times V \rightarrow \mathbb{R}$ such that $c(u, v) \geq 0$ if $(u, v) \in E$ and $c(u, v) = 0$ for all $(u, v) \notin E$.

Flow Networks

Example:



For this graph, $V = \{s, r, u, v, w, x, y, z, t\}$

The edge set is

$$E = \{(s, u), (s, r), (s, x), (u, v), (u, x), (v, x), (v, w), (r, w), (r, y), (x, y), (y, r), (y, z), (z, w), (z, t), (w, t)\}.$$

Some examples of capacities are $c(s, x) = 10$, $c(r, y) = 5$, $c(v, x) = 20$ and $c(v, r) = 0$ (since there is no arc from v to r).

Network Flows

Let $\mathcal{N} = (G = (V, E), c, s, t)$ be a flow network.

A **flow** in \mathcal{N} is a function $f: V \times V \rightarrow \mathbb{R}$ satisfying the following conditions:

Capacity constraint: $f(u, v) \leq c(u, v)$ for all $u, v \in V$.

Flow conservation: For all $u \in V \setminus \{s, t\}$,

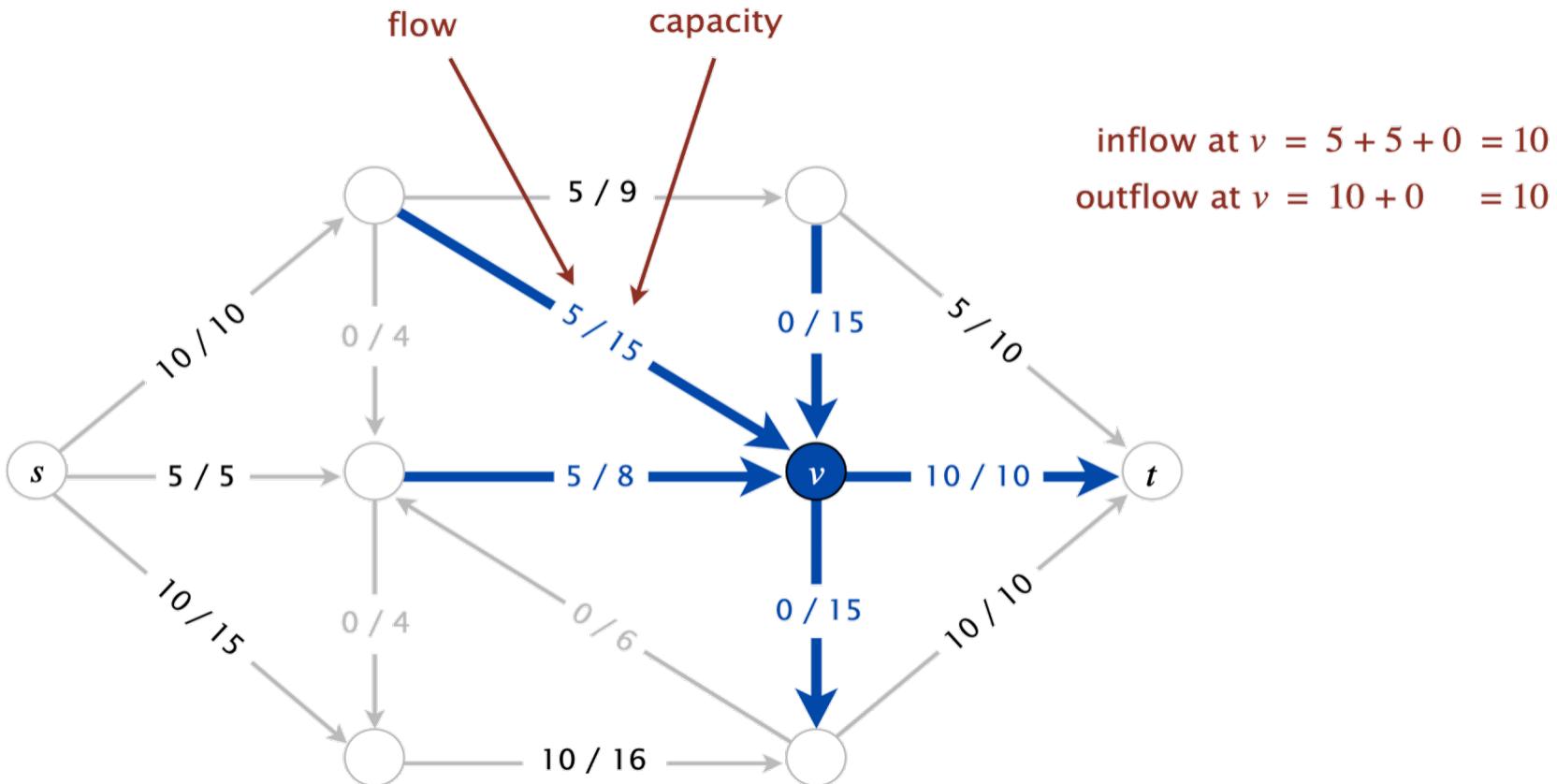
$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

flow into u flow out of u

Note: In the case that flow conservation is satisfied, one can prove that the net flow out of s equals the net flow into t . This quantity is called the **flow value**, or simply the magnitude, of f .

Network Flows

- **Capacity:** We cannot overload an edge
- **Conservation:** Flow entering any vertex must be equal to flow leaving that vertex
- We want to maximize the **value of a flow**, subject to these constraints
- A **saturated edge** is at maximum capacity



Maximum-Flow Problem

Input: Network \mathcal{N}

Output: Flow of maximum value in \mathcal{N}

The problem is to find the flow f such that $|f| = \sum_{v \in V} f(s, v)$ is the largest possible (over all “legal” flows).

A **maximum flow** is defined as the maximum amount of flow that the graph or network would allow to flow from the source node to its sink node.

Assumptions:

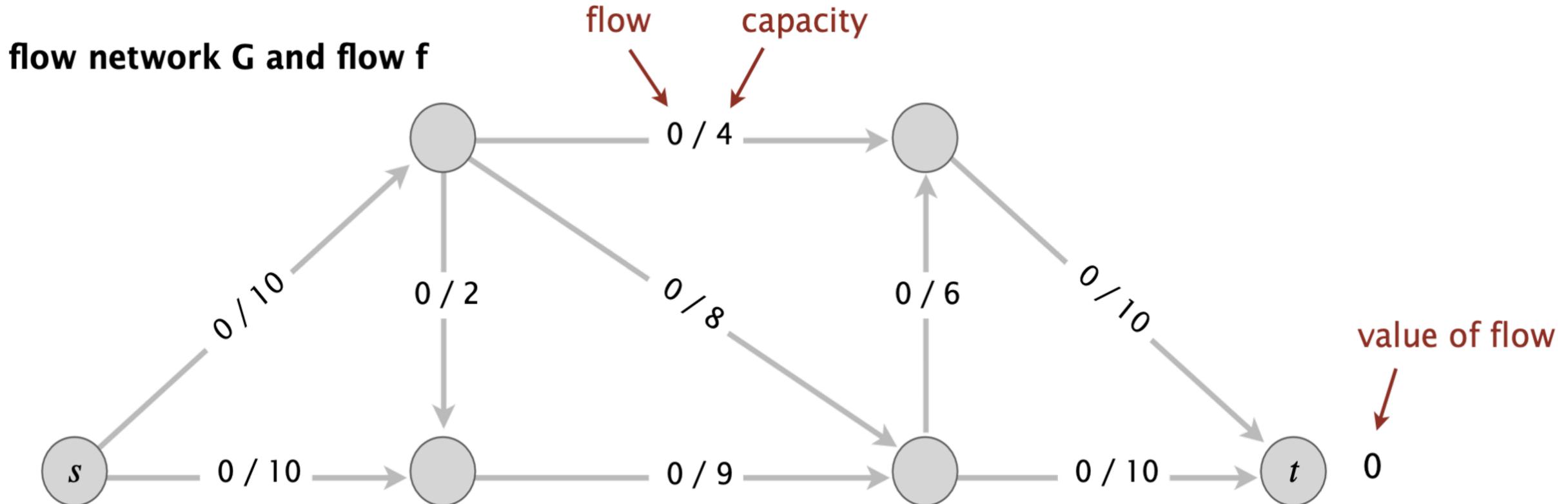
1. No edges enter s , no edges leave t .
2. There is at least one edge incident on each node.
3. All edge capacities are integers.

Maximum-Flow Problem

No known dynamic programming algorithm to solve this problem.

Let us take a greedy approach.

1. Start with zero flow along all edges.

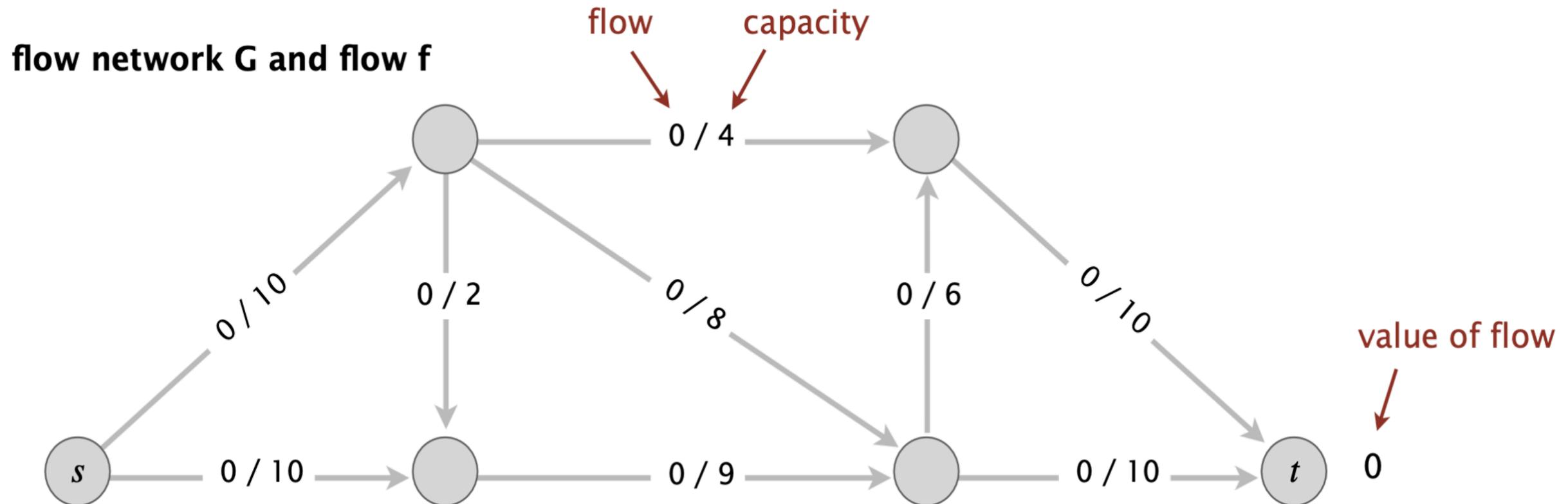


Maximum-Flow Problem

No known dynamic programming algorithm to solve this problem.

Let us take a greedy approach.

1. Start with zero flow along all edges.

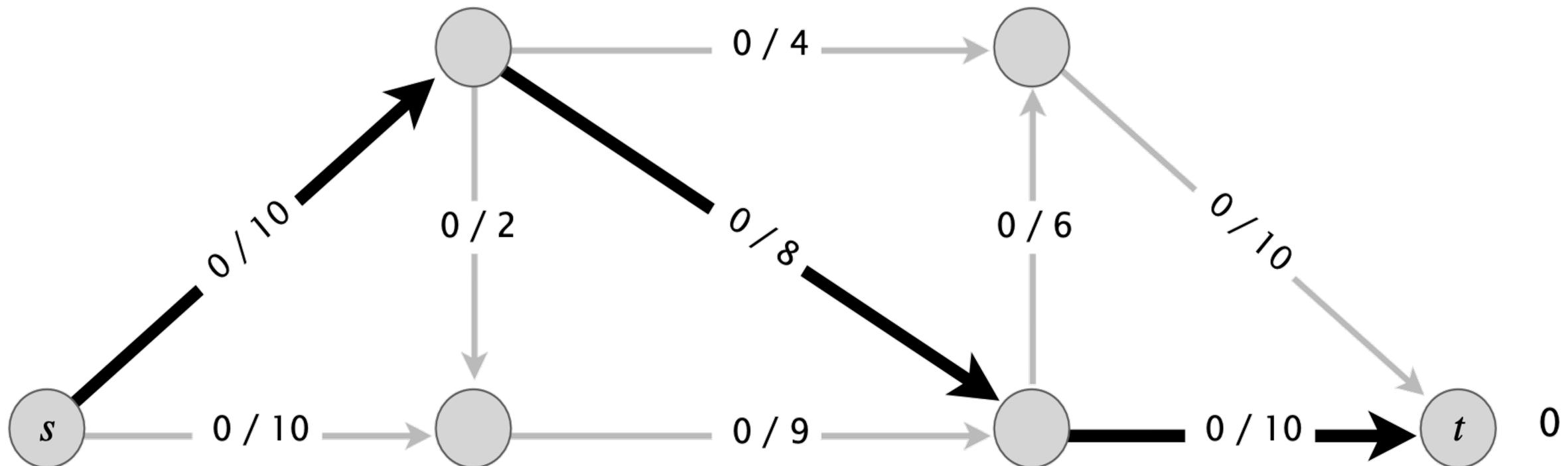


Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s-t path P where each edge has $f(u, v) < c(u, v)$

flow network G and flow f

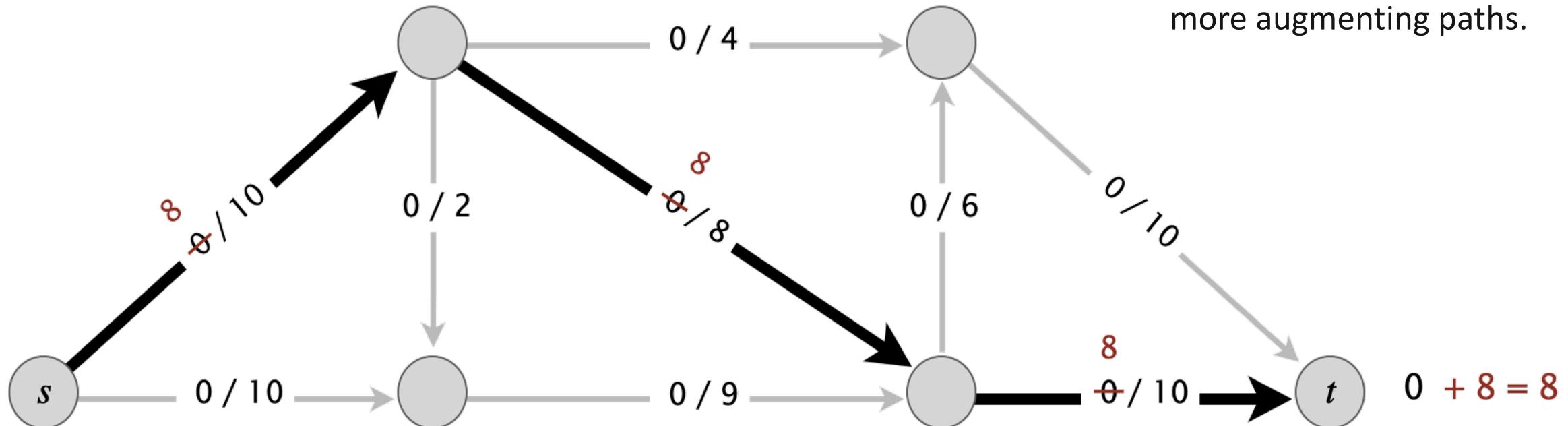


Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s - t path P where each edge has $f(u, v) < c(u, v)$
3. Augment flow along path P .

flow network G and flow f

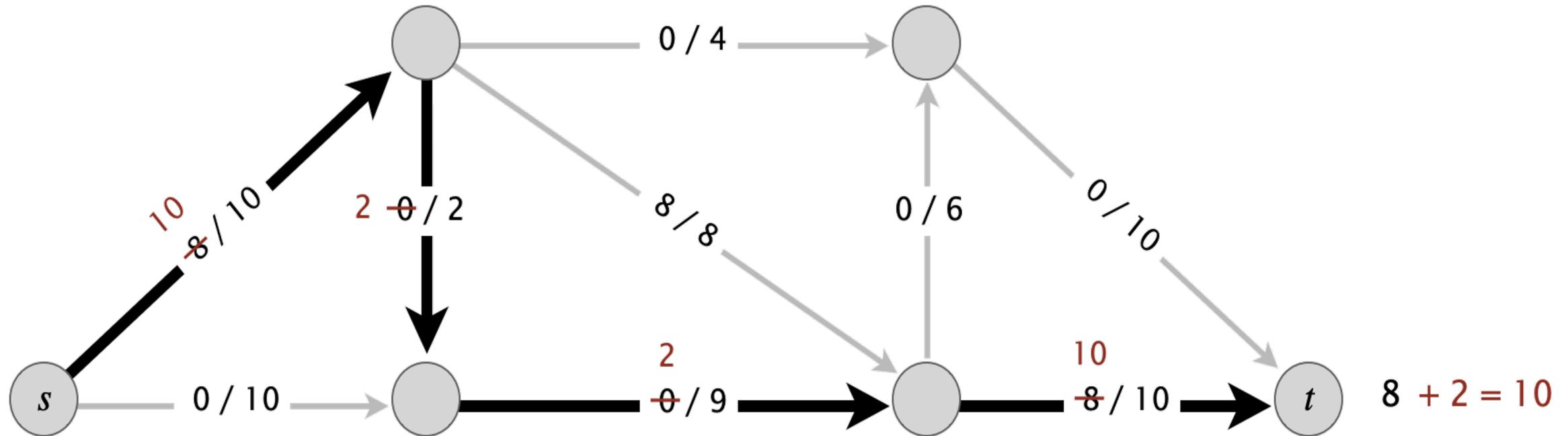


Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s - t path P where each edge has $f(u, v) < c(u, v)$
3. Augment flow along path P .
4. Repeat until you get stuck.

flow network G and flow f

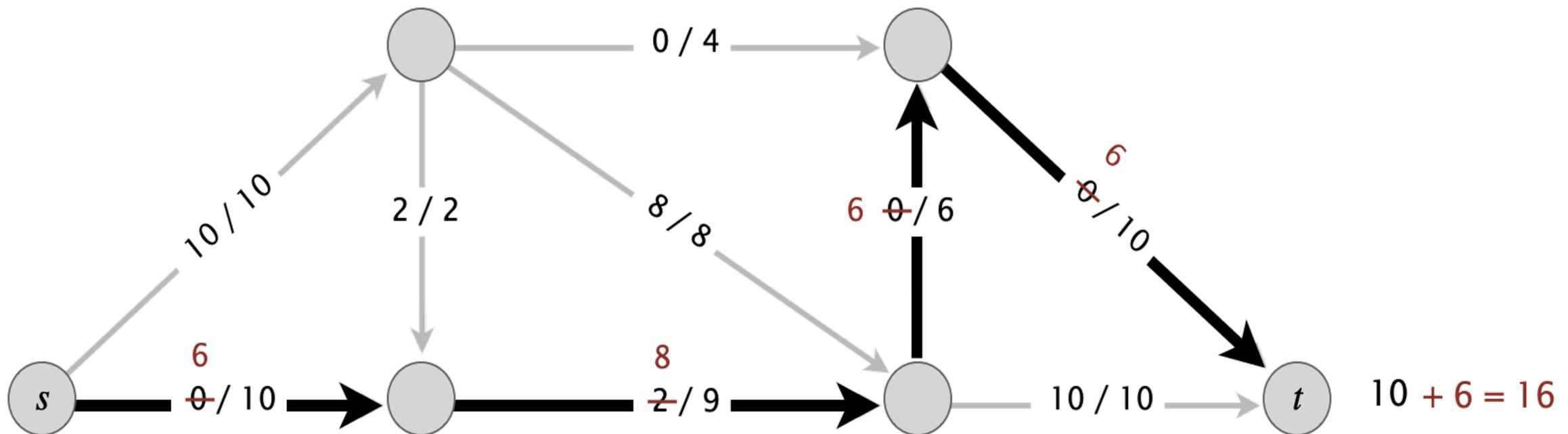


Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s - t path P where each edge has $f(u, v) < c(u, v)$
3. Augment flow along path P .
4. Repeat until you get stuck.

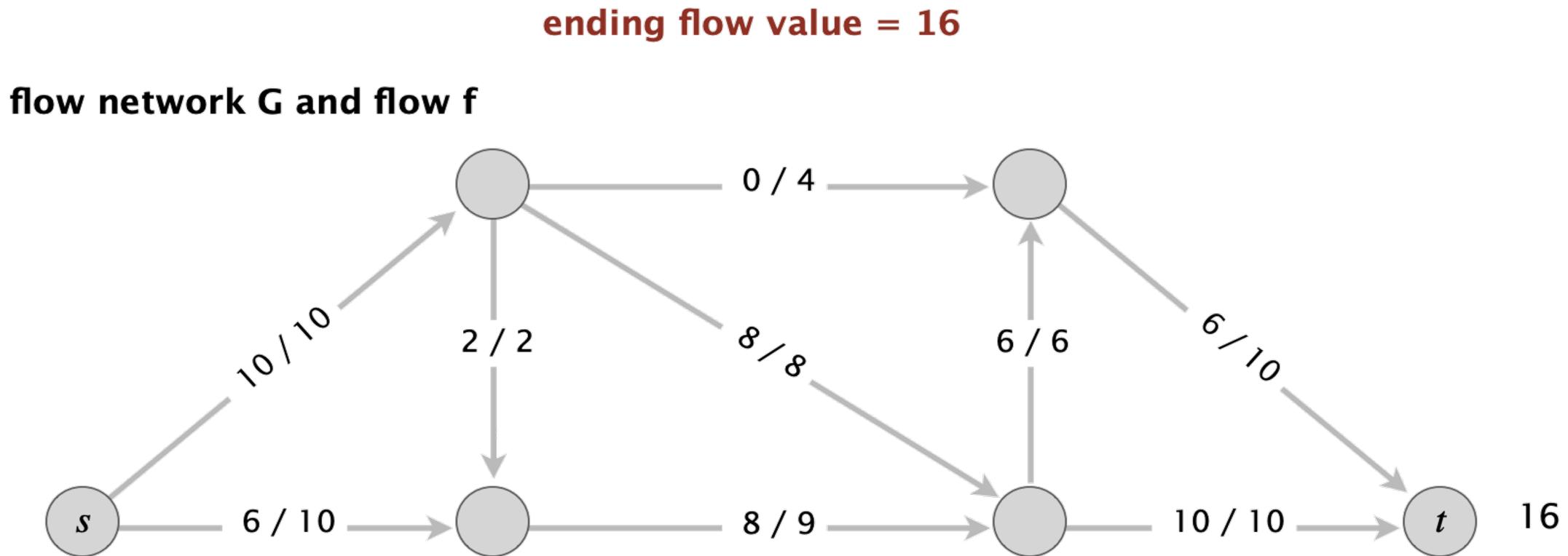
flow network G and flow f



Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s-t path P where each edge has $f(u, v) < c(u, v)$
3. Augment flow along path P .
4. Repeat until you get stuck.



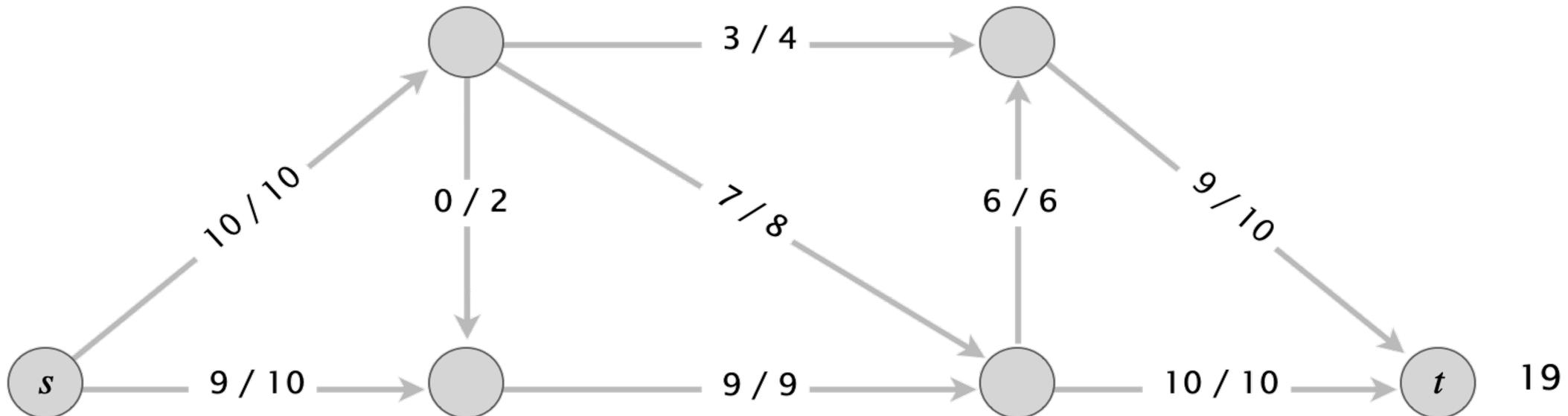
Maximum-Flow Problem

Greedy algorithm.

1. Start with zero flow along all edges.
2. Find an s-t path P where each edge has $f(u, v) < c(u, v)$
3. Augment flow along path P .
4. Repeat until you get stuck.

but max-flow value = 19

flow network G and flow f



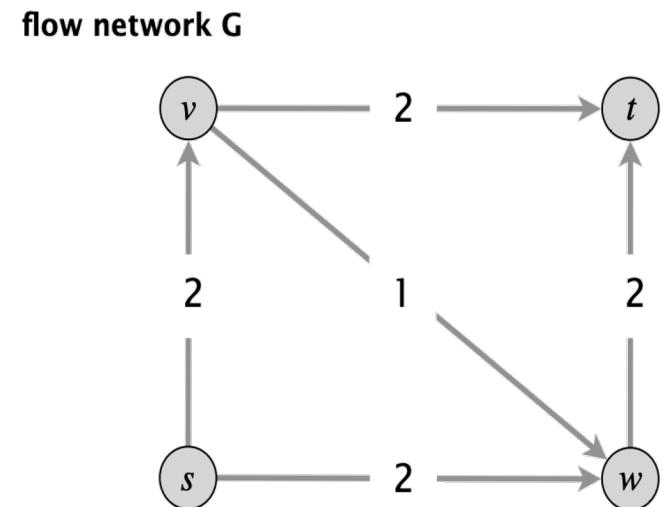
Why does the greedy algorithm fail?

Q. Why does the greedy algorithm fail?

A. Once greedy algorithm increases flow on an edge, it never decreases it.

Ex. Consider flow network G

- Greedy algorithm could choose $s \rightarrow v \rightarrow w \rightarrow t$ as first path.



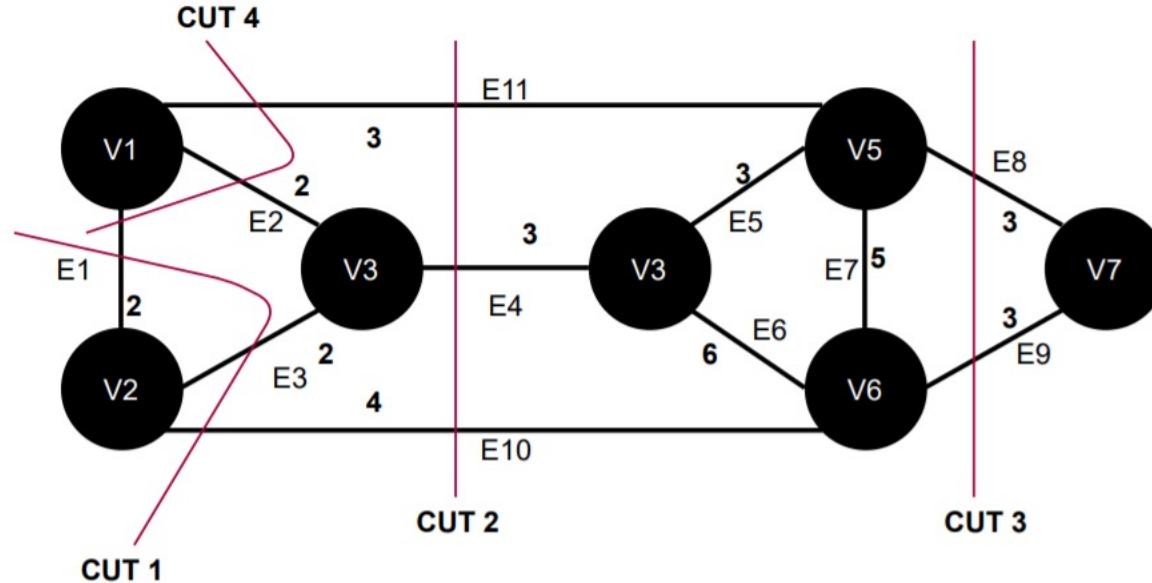
We need some mechanism to **undo** a bad decision. Can we do it by just modifying the graph?

- Yes.

Minimum Cut in a Graph

A **cut** is a set of edges whose removal divides a connected graph into two disjoint subsets.

The **minimum cut** of a weighted graph is defined as the minimum sum of weights of edges that, when removed from the graph, divide the graph into two sets.



Here in this graph, **CUT 3** is an example of a minimum cut. It removes the edges E8 and E9, and the sum of weights of these two edges are minimum among all other cuts in this graph.

Max-Flow Min-Cut Theorem

The **max-flow min-cut theorem** states that the maximum flow through any network from a given source to a given sink is exactly equal to the minimum sum of a cut.

The **Ford-Fulkerson algorithm** is an algorithm that tackles the max-flow min-cut problem. This algorithm finds the maximum flow of a network or graph.

The Ford-Fulkerson algorithm is based on the three important concepts: **the residual network, augmented path and cut**.

Ford–Fulkerson algorithm

The **Ford–Fulkerson algorithm** is an elegant solution to the maximum flow problem.

Ford–Fulkerson augmenting path algorithm.

- Start with $f(e) = 0$ for each edge $e \in E$.
- Find an $s \rightarrow t$ path P in the **residual network** G_f .
- **Augment flow** along path P .
- Repeat until you get stuck.

Note: If there are multiple possible augmenting paths, the decision of which path to use is completely arbitrary. Thus, like any terminating greedy algorithm, the Ford–Fulkerson algorithm will find a locally optimal solution.

Residual Graph

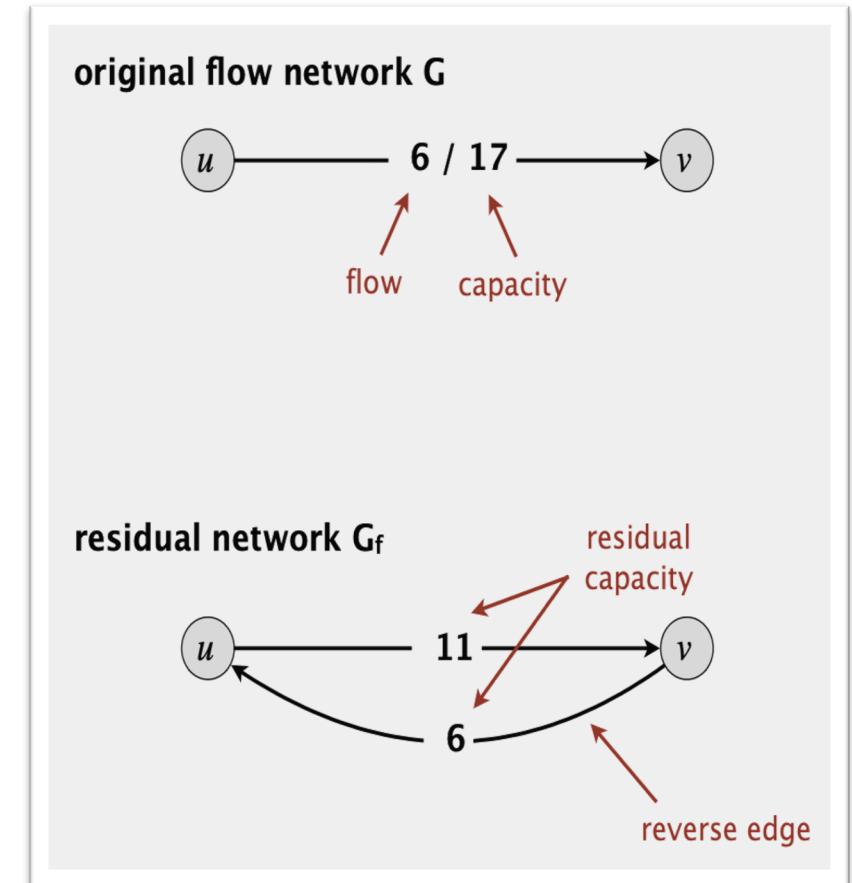
Residual capacity is an important attribute and it plays an important role in residual graph creation. Residual capacity is defined as the new capacity after a given flow has been taken away. In other words, for a given edge (u, v) the residual capacity, C_f is defined as

$$c_f(u, v) = c(u, v) - f(u, v)$$

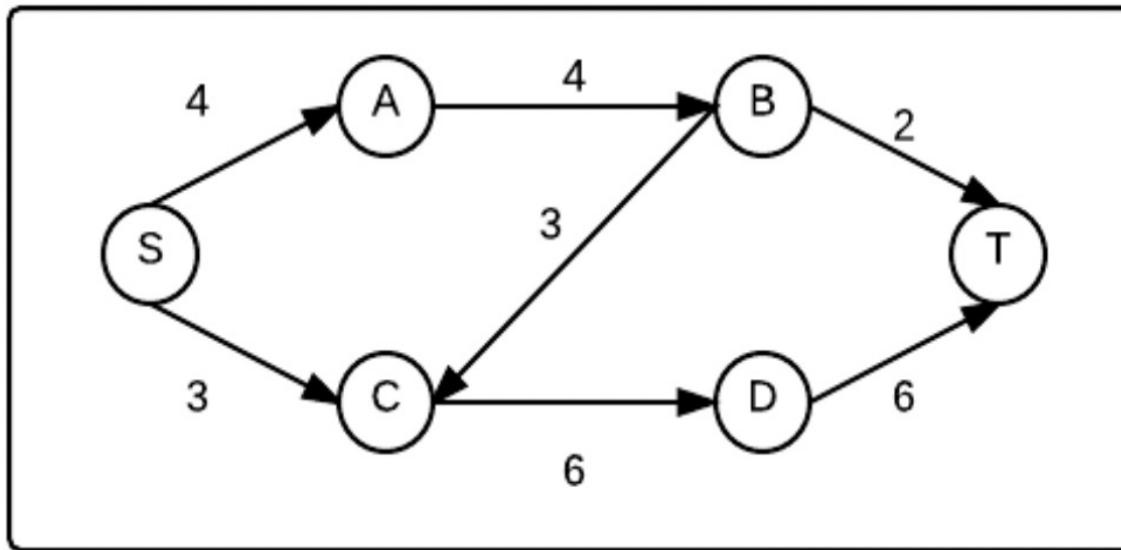
However, there must also be a residual capacity for the **reverse edge** as well. The max-flow min-cut theorem states that flow must be preserved in a network. So, the following equality always holds:

$$f(u, v) = -f(v, u)$$

With these tools, it is possible to calculate the residual capacity of any edge, forward or backward, in the flow network. Then, these residual capacities are used to make a residual network, G_f



Example



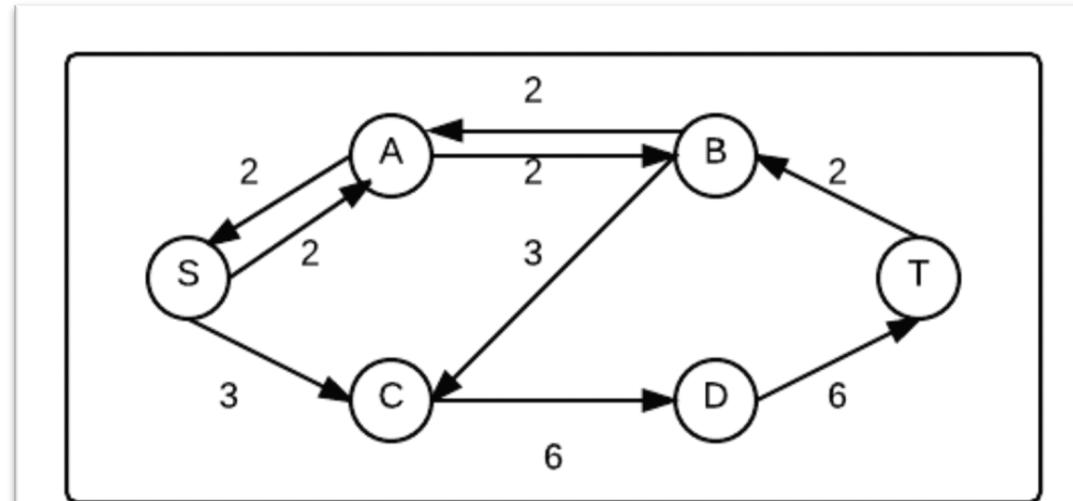
Flow network before Ford-Fulkerson

Example

Initially, 2 units of flow are pushed along the top-most path (S , A , (A, B) , and (B, T)). Two things happen:

1. In the forward direction, the edges now have a residual capacity equal to $c_f(u, v) = c(u, v) - f(u, v)$. The flow is equal to 2, so the residual capacity of (S, A) and (A, B) is reduced to 2, while the edge (B, T) has a residual capacity of 0.

2. In the backward direction, the edges now have a residual capacity equal to $c_f(v, u) = c(v, u) - f(v, u)$. Because of flow preservation, this can be written $c_f(v, u) = c(v, u) + f(u, v)$. And since the capacity of those backward edges was initially 0, all of the backward edges (T, B) , (B, A) , and (A, S) now have a residual capacity of 2.

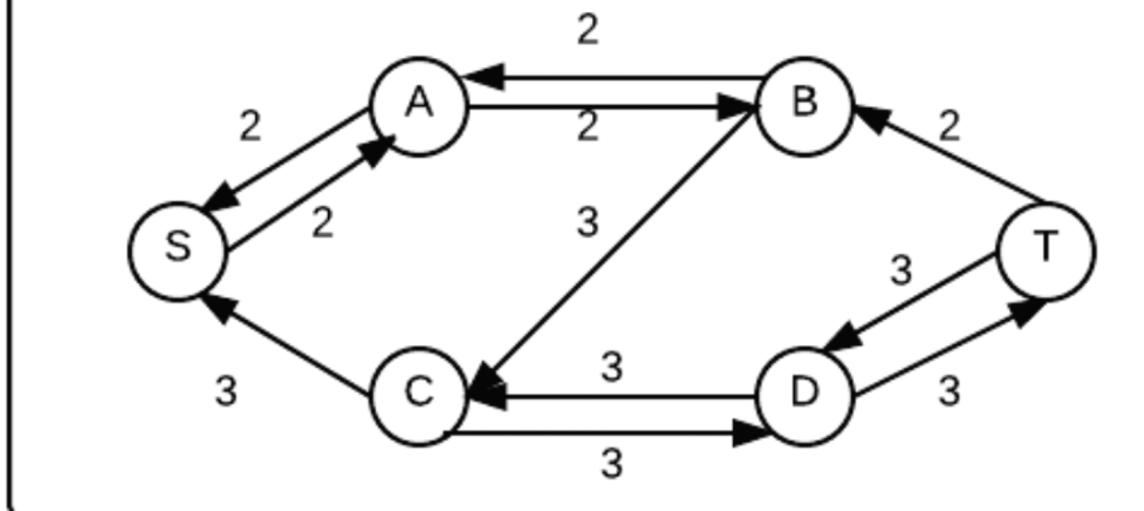


Residual graph after 1 round

When a new residual graph is constructed with these new edges, any edges with a residual capacity of 0—like (B, T) —are not included.

Example

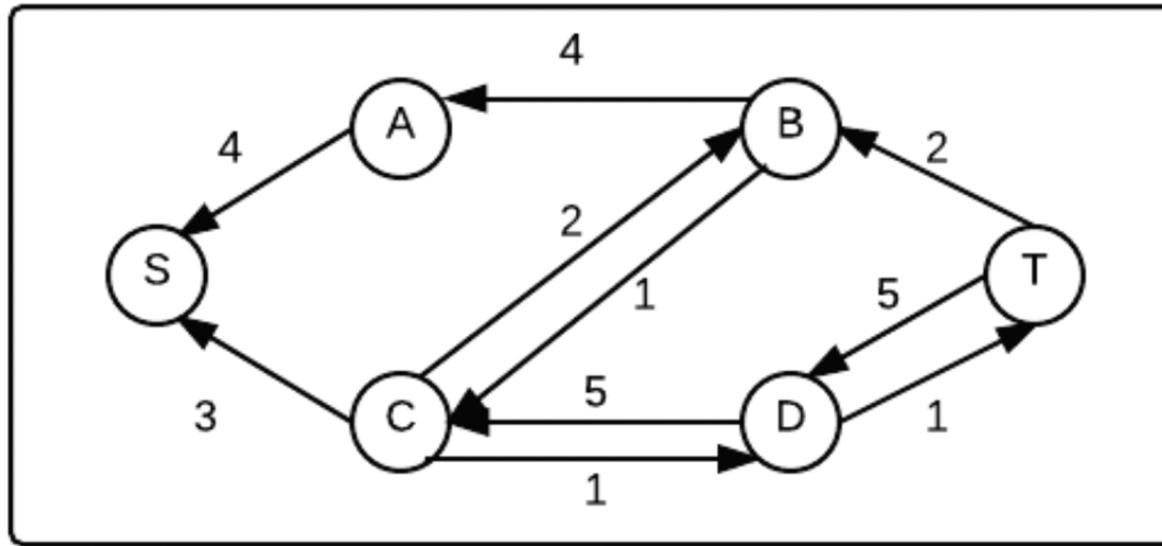
Now, a new augmenting path must be found (the top-most path can never be used again because the edge (B, T) was erased). The bottom path can be chosen and flow of 3 can be sent along it.



Residual graph after 2 rounds

Example

Finally, a flow of 2 can be sent along the path $[(S, A), (A, B), (B, C), (C, D), (D, T)]$ because the minimum residual capacity along that path is 2. The final residual graph is given here.



Residual graph after 3 rounds

There are no more paths from the source to the sink, so there can be no more augmenting paths. Therefore, the loop is complete. The flow, 7, is a maximum flow.

Complexity of Ford–Fulkerson algorithm

The analysis of Ford-Fulkerson depends heavily on how the augmenting paths are found. The typical method is to use breadth-first search to find the path. If this method is used, Ford-Fulkerson runs in polynomial time.

If all flows are integers, then the while loop of Ford-Fulkerson is run at most $|f^*|$ times, where f^* is the maximum flow. This is because the flow is increased, at worst, by 1 in each iteration. Finding the augmenting path inside the while loop takes $O(V + E')$, where E' is the set of edges in the residual graph. This can be simplified to $O(E)$.

So, the runtime of Ford-Fulkerson is $O(E \cdot |f^*|)$

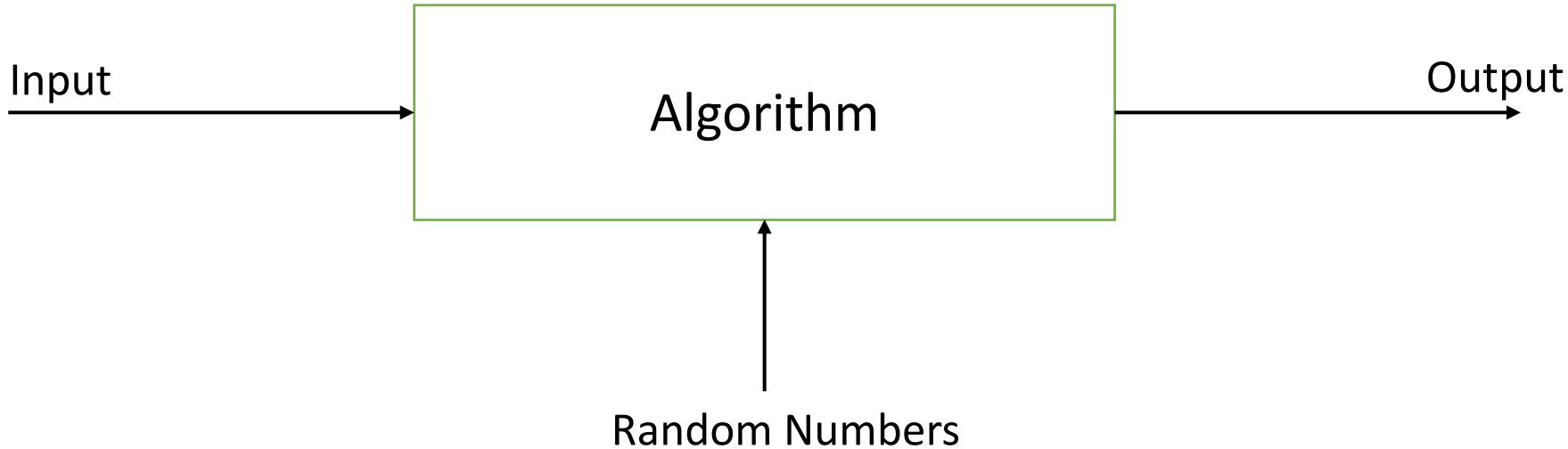
Randomized Algorithms

Deterministic Algorithms



Goal: To prove that the algorithm solves the problem correctly (always) and quickly (typically, the number of steps should be polynomial in the **size of the input**).

Randomized Algorithms



- In addition to input, algorithm takes a source of random numbers and makes random choices during execution.
- Behavior can vary even on a fixed input.
- Design algorithm and analysis so that this behavior is likely to be good, on every input.

Randomized Algorithms

- A randomized algorithm is a technique that uses a source of randomness as part of its logic.
- It is typically used to reduce either the running time, or time complexity; or the memory used, or space complexity, in a standard algorithm.
- The algorithm works by generating a random number, r , within a specified range of numbers, and making decisions based on r 's value.

Randomized Algorithms

- Randomized algorithms are usually designed in one of two common forms: as a **Las Vegas** or as a **Monte Carlo algorithm**.
- Practically speaking, computers cannot generate completely random numbers, so randomized algorithms in computer science are approximated using a pseudorandom number generator in place of a true source of random number, such as the drawing of a card, flipping a coin, etc.

Advantages of randomized algorithms

- Simplicity
- Performance

For many problems, a randomized algorithm is the simplest, the fastest, or both.

Monte Carlo and Las Vegas

- A **Monte Carlo algorithm** runs for a fixed number of steps, and produces an answer that is correct with probability $\geq 1/3$.
- A **Las Vegas algorithm** always produces the correct answer; its running time is a random variable whose expectation is bounded (say by a polynomial).
- These probabilities/expectations are only over the random choices made by the algorithm (i.e., independent of the input).
- Thus independent repetitions of Monte Carlo algorithms drive down the failure probability exponentially.

Monte Carlo Algorithms

- The term for this algorithm, Monte Carlo, was coined by mathematicians Nicholas Metropolis, Stanislaw Ulam, and John von Neumann, working on the Manhattan Project, around the 1940s.
- The name was found in a research paper published in 1949, attributed by some sources to the fact that Ulam's uncle made a yearly trip to gamble at Monte Carlo, in Monaco.

Monte Carlo Algorithms - Example

- The game, Wheel of Fortune, can be played using a Monte Carlo randomized algorithm.
- Instead of mindfully choosing letters, a player (or computer) picks random letters to obtain a solution, as shown in the image.
- The more letters a player reveals, the more confident a player becomes in their solution. However, if a player does not guess quickly, the chance that other players will guess the solution also increases.
- Therefore, a Monte Carlo algorithm is given a deterministic amount of time, in which it must come up with a "guess" based on the information revealed; the best solution it can come up with.



Example contd..

- This allows for the possibility of being wrong, maybe even a large probability of being wrong if the Monte Carlo algorithm did not have sufficient time to reveal enough useful letters.
- But providing it with a time limit controls the amount of time the algorithm will take, thereby decreasing the risk of another player guessing and getting the prize.
- **Note: A useful property of a Monte Carlo algorithm is if the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time.**

Approximating π

- A classic example of a Monte Carlo algorithm lies in the solution to approximating pi, π , the ratio of a circle's circumference to its diameter.
- Imagine you're blindfolded and someone asks you to find the shape of an object. Intuitively, the solution would be to touch the object in as many places as possible until a familiar object come to mind, at which point you make a guess. The same strategy is applied when approximating π .
- **For better understanding of Monte Carlo simulation for approximating π , you can watch [this video](#).**

Las Vegas Algorithms

- The term for this algorithm, Las Vegas, is attributed to mathematician Laszlo Babai, who coined it in 1979 simply as a parallel to the much older Monte Carlo algorithm, as both are major world gambling centers.
- However, the gambling styles of the two have nothing to do with the styles of the algorithms, as it cannot be said that gambling in Las Vegas always gives a correct, or even positive turnout.

Las Vegas Algorithms

- A Las Vegas algorithm runs within a specified amount of time. If it finds a solution within that timeframe, the solution will be exactly correct; however, it is possible that it runs out of time and does not find any solutions.
- A Las Vegas algorithm is by definition a Monte Carlo algorithm with error probability 0.

Las Vegas Algorithm - Example

- A Las Vegas algorithm could be thought of as the strategy used by a user who searches for something online.
- Since searching every single website online is extremely inefficient, the user will generally use a search engine to get started. The user will then surf the web until a website is found which contains exactly what the user is looking for.
- Since clicking through links is a randomized process, assuming the user does not know exactly what's contained on the website at the other end, the time complexity ranges from getting lucky and reaching the target website on the first link, to being unlucky and spending countless hours to no avail.

Why this example follows Las Vegas algorithm?

Example contd..

Because here the user knows exactly what she is looking for, so once the website is found, there is no probability of error. Similarly, if the user's allotted time to surf the web is exceeded, she will terminate the process knowing that the solution was not found.

Randomized Quicksort

- A common Las Vegas randomized algorithm is **quicksort**, a sorting algorithm that sorts elements in place, using no extra memory.
- Since this is a comparison based algorithm, the worst case scenario will occur when performing pairwise comparison, taking $O(n^2)$, where the time taken grows as a square of the number of digits to be sorted grows. However, through randomization, the runtime of this algorithm can be reduced up to $O(n \log(n))$.

Complexity of Randomized Algorithms

- Randomized algorithms have a complexity class of their own.
- The basic probabilistic complexity class is called **RP**, randomized polynomial time algorithms.
- It encompass problems with an efficient randomized algorithm, taking **polynomial time**, and recognizes bad solutions with absolute certainty, and correct solutions with probability of at least $\frac{1}{2}$.

Complexity of Randomized Algorithms

- Randomized algorithms with polynomial time runtime complexity, whose output is always correct (Las Vegas algorithms), are said to be in **ZPP**, or zero-error probabilistic polynomial time algorithms.
- Lastly, the class of problems for which both YES and NO-instances are allowed to be identified with some error, commonly known as Monte Carlo algorithms, are in the complexity class called **BPP**, bounded-error probabilistic polynomial time.

Randomized Algorithms - Examples

Primality Testing

- Testing whether a number is prime or composite
- Modern cryptographic techniques securing important information depend on it.
- Primality Testing was one of the first randomized algorithms formally developed in the 1970s.
- Randomized algorithms are used to perform primality testing in order to avoid a brute force search, which would consist of a time consuming linear search of every prime number leading up to the number at hand.

Randomized Algorithms - Examples

Randomized Minimum Cut

- The Max-Flow Min-cut algorithm is another basic randomized algorithm applied on network flow and general graph problems.
- The goal is to find the smallest total weight of edges which, if removed, would disconnect the source from the sink in a max-flow network problem.
- Due to the fact that every edge must be checked in order to confidently assure that the right answer is found, randomized algorithms are developed.
- The famous randomized algorithm for a minimum cut in a graph is Ford-Fulkerson Algorithm.

Randomized Algorithms - Examples

Frievalds' Algorithm for Matrix Product Checking

- Given three $n \times n$ matrices, A, B, and C, a common problem is to verify whether $A \times B = C$. Since matrix multiplication is a rather costly process and checking the stated equality would require term-by-term comparisons of both sides of the equation.
- Randomized algorithms have been developed in order to avoid this brute force method of checking the equality.
- The algorithm is named after Rusins Frievalds, who realized that by using randomization, he could reduce the running time of this problem from brute force matrix multiplication using Strassen's algorithm, taking a runtime of $O(n^{2.37})$ to $O(kn^2)$, where k is the number of times the algorithm is executed.

THANK YOU

Dynamic programming Longest Common Subsequence

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory for later use
- More efficient than “*brute-force methods*”, which solve the same subproblems over and over again

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: $X = \{A\ B\ C\ B\ D\ A\ B\}$, $Y = \{B\ D\ C\ A\ B\ A\}$

Longest Common Subsequence:

$X = A\ B\ C\ B\ D\ A\ B$

$Y = B\ D\ C\ A\ B\ A$

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- if $|X| = m, |Y| = n$, then there are 2^m subsequences of x ; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is $O(n 2^m)$
- Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.
- Subproblems: “find LCS of pairs of *prefixes* of X and Y ”

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i , Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0, 0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i, 0] = 0$

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- When we calculate $c[i, j]$, we consider two cases:
- First case:** $x[i] = y[j]$: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{j-1} , plus 1

LCS recursive solution

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** $x[i] \neq y[j]$
- As symbols don't match, our solution is not improved, and the length of $\text{LCS}(X_i, Y_j)$ is the same as before (i.e. maximum of $\text{LCS}(X_i, Y_{j-1})$ and $\text{LCS}(X_{i-1}, Y_j)$)

Why not just take the length of $\text{LCS}(X_{i-1}, Y_{j-1})$?
11/05/21 8

LCS Length Algorithm

LCS-Length(X, Y)

1. $m = \text{length}(X)$ // get the # of symbols in X
2. $n = \text{length}(Y)$ // get the # of symbols in Y
3. for $i = 1$ to m $c[i,0] = 0$ // special case: Y_0
4. for $j = 1$ to n $c[0,j] = 0$ // special case: X_0
5. for $i = 1$ to m // for all X_i
6. for $j = 1$ to n // for all Y_j
7. if ($X_i == Y_j$)
8. $c[i,j] = c[i-1,j-1] + 1$
9. else $c[i,j] = \max(c[i-1,j], c[i,j-1])$
10. return c

LCS Example

We'll see how LCS algorithm works on the following example:

- $X = ABCB$
- $Y = BDCAB$

What is the Longest Common Subsequence of X and Y?

$$\text{LCS}(X, Y) = BCB$$

$X = A \ B \ C \ B$

$Y = \quad B \ D \ C \ A \ B$

LCS Example (0)

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi					
1	A					
2	B					
3	C					
4	B					

$X = ABCB; m = |X| = 4$

$Y = BDCAB; n = |Y| = 5$

Allocate array $c[5,4]$

LCS Example (1)

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0				
2	B	0				
3	C	0				
4	B	0				

for $i = 1$ to m $c[i,0] = 0$

for $j = 1$ to n $c[0,j] = 0$

LCS Example (2)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (3)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
Xi	0	0	0	0	0	0
A	0	0	0	0		
B	0					
C	0					
B	0					

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (4)

ABCB
BDCA₄B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

```

if( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (5)

ABCB
BDCA_B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (6)

A₀B₁C₂B₃
B₀D₁C₂A₃B₄

i	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1				
3	C	0					
4	B	0					

```

if ( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (7)

ABC
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (8)

ABC
BDCA B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

```

if( Xi == Yj )
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )

```

LCS Example (10)

ABC
BD CAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (11)

ABC
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	2
3	C	0	1	1	2	
4	B	0				

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (12)

ABC
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	1
3	C	0	1	1	2	2
4	B	0				

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (13)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCA B

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

```

if(  $X_i == Y_j$  )
     $c[i,j] = c[i-1,j-1] + 1$ 
else  $c[i,j] = \max( c[i-1,j], c[i,j-1] )$ 

```

LCS Example (15)

ABCB
BDCA_B

j	0	1	2	3	4	5
i	Y _j	B	D	C	A	B
0	X _i	0	0	0	0	0
1	A	0	0	0	0	1
2	B	0	1	1	1	1
3	C	0	1	1	2	2
4	B	0	1	1	2	2

$\text{if } (X_i == Y_j)$
 $c[i,j] = c[i-1,j-1] + 1$
 $\text{else } c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array $c[m,n]$
- So what is the running time?

$O(m*n)$

since each $c[i,j]$ is calculated in constant time, and there are $m*n$ elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each $c[i,j]$ depends on $c[i-1,j]$ and $c[i,j-1]$
or $c[i-1, j-1]$

For each $c[i,j]$ we can say how it was acquired:

2	2
2	3

For example, here
 $c[i,j] = c[i-1,j-1] + 1 = 2+1=3$

How to find actual LCS - continued

- Remember that

$$c[i, j] = \begin{cases} c[i - 1, j - 1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j - 1], c[i - 1, j]) & \text{otherwise} \end{cases}$$

- So we can start from $c[m, n]$ and go backwards
- Whenever $c[i, j] = c[i - 1, j - 1] + 1$, remember $x[i]$ (because $x[i]$ is a part of LCS)
- When $i=0$ or $j=0$ (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

Finding LCS (2)

i	j	0	1	2	3	4	5
	Yj	B	D	C	A	B	
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): B C B

LCS (straight order): B C B
(this string turned out to be a palindrome)

ADSA – Unit 6

Advanced Topics

Dr. Sreeja S R

Topics to be covered

- Network Flows
- Randomized Algorithms
- Computational Complexity
 - NP-completeness
 - Polytime reductions

P = NP

- In computer science, there exist several famous unresolved problems, and $\mathcal{P}=\mathcal{NP}$ is one of the most studied ones.
- Until now, the answer to this problem is mainly “no”.
- And, this is accepted by the majority of the academic world.
- We probably wonder why this problem is still not resolved. And in the end, hopefully, we would have a better understanding of why $\mathcal{P}=\mathcal{NP}$ is still an open problem.

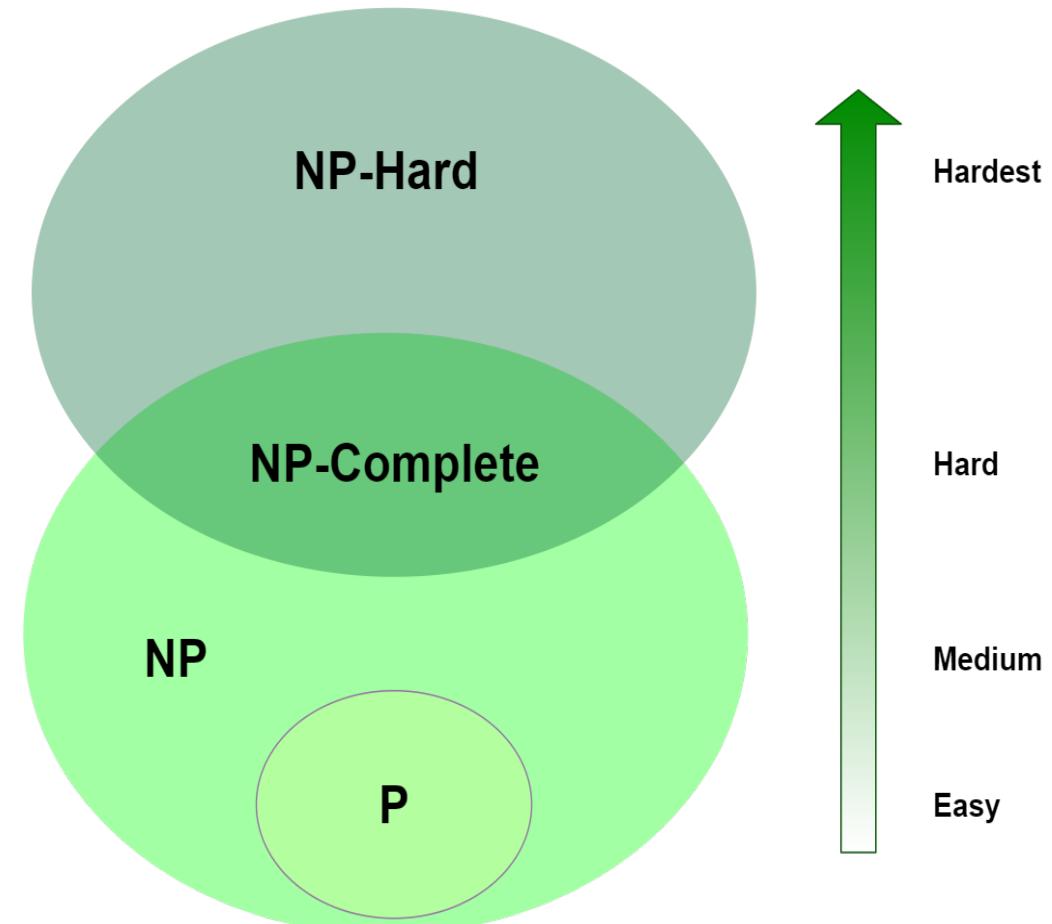
Classification

- To explain \mathcal{P} , \mathcal{NP} and others, let's use the same mindset that we use to classify problems in real life. Let's classify it in Easy-to-Hard scale.
- In theoretical computer science, the classification and complexity of common problem definitions have two major sets; \mathcal{P} which is **Polynomial** time and \mathcal{NP} which is **Non-deterministic Polynomial** time.
- There are also \mathcal{NP} -*hard* and \mathcal{NP} -*complete* sets, which we use to express more sophisticated problems.

Classification

- Easy $\rightarrow \mathcal{P}$
 - Medium $\rightarrow \mathcal{NP}$
 - Hard $\rightarrow \mathcal{NP}$ -complete
 - Hardest $\rightarrow \mathcal{NP}$ -hard
-
- Using the diagram, we assume that \mathcal{P} and \mathcal{NP} are not the same set, or, in other words, we assume that $\mathcal{P} \neq \mathcal{NP}$. This is apparently-true, but yet-unproven assertion.
 - Of course, another interesting aspect of this diagram is that we've got some overlap between \mathcal{NP} and \mathcal{NP} -hard.
 - We call \mathcal{NP} -complete when the problem belongs to both of these sets.

And we can visualize their relationship, too:



Algorithmic complexity

- $O(1)$ - constant-time
- $O(\log n)$ - logarithmic-time
- $O(n)$ - linear-time
- $O(n^2)$ - quadratic-time
- $O(n^k)$ - polynomial-time
- $O(k^n)$ - exponential-time
- $O(n!)$ - factorial-time

Where k is a constant and n is the input size. The size of n also depends on the problem definition. For example, using a number set with a size of n , the search problem has an average complexity between linear-time and logarithmic-time depending on the **data structure** in use.

Polynomial Algorithms

- The first set of problems are polynomial algorithms that we can solve in polynomial time, like logarithmic, linear or quadratic time.
- If an algorithm is polynomial, we can formally define its time complexity as:
 $T(n) = O(c * n^k)$ where $C > 0$ and $k > 0$ where C and k are constants and n is input size.
- In general, for polynomial-time algorithms k is expected to be less than n .

Many algorithms complete in polynomial time:

- All basic mathematical operations; addition, subtraction, division, multiplication
- Testing for primacy
- Hashtable lookup, string operations, sorting problems
- Shortest Path Algorithms; Dijkstra, Bellman-Ford, Floyd-Warshall
- Linear and Binary Search Algorithms for a given set of numbers

Polynomial Algorithms

- All of these have a complexity of $O(n^k)$ for some k .
- we don't always have just one input, n . But, so long as each input is a polynomial, multiplying them will still be a polynomial.
- For example, in graphs, we use E for edges and V for vertices, which gives $O(E * V)$ for Bellman-Ford's shortest path algorithm. Even if the size of the edge set is $E = V^2$, the time complexity is still a polynomial, $O(V^3)$, still in \mathcal{P} .

Examples

Example 1: Game of checkers

- What is the complexity of determining the optimal move on a given turn?
- If the size of the board is constrained to $8 * 8$, then this is believed to be a polynomial-time problem, placing it in P.
- If the size of the board is constrained to $N * N$, it's no longer in P.

Example 2: Stable roommate problem

- To match a roommate without a tie, takes polynomial-time.
- But when ties are not allowed it's not in P.

If the input size is going to be near k , then the algorithm is going to behave more like an exponential. These variants are actually **NP-Complete** .

NP Algorithms

- The second set of problems cannot be solved in polynomial time. However, they can be verified (or certified) in polynomial time.
- If an algorithm to have an exponential complexity, can be formally define its time complexity as: $T(n) = O(C_1 * k^{C_2*n})$ where $C_1 > 0, C_2 > 0$ and $k > 0$ where C_1, C_2 and k are constants and n is the input size.
- $T(n)$ is a function of exponential-time when at least $C_1 = 1, C_2 = 1$, we will get $O(k^n)$.
- For example, complexities like $O(n^n), O(2^n), O(2^{0.00001*n})$.

There are several algorithms that fit this description.

- Integer Factorization and
- Graph Isomorphism

NP Algorithms

- Both of these have two important characteristics: Their complexity is $O(k^n)$ for some k and their results can be verified in polynomial time.
- Those two facts place them all in \mathcal{NP} which is Non-deterministic Polynomial algorithms.
- These problems must be decision problems – have a yes or no answer – all function problems can be transformed into decision problems. This distinction helps to nail down what we mean by verified.
- These type of algorithms are in NP, if it can't be solved in polynomial time and the set of solutions to any decision problem can be verified in polynomial time by a Deterministic Turing Machine.

NP-Complete Algorithms

- This is similar to the previous set, but they are hard problems.
- There are more than 3000 of these problems, and the theoretical computer science community populates the list quickly.
- What makes them different from other NP problems is a useful distinction called **completeness**.
- For any NP problem that's complete, there exists a polynomial-time algorithm that can transform the problem into any other NP-Complete problem. This transformation requirement is also called **polytime reduction**.

NP-Complete Algorithms

As stated already, there are numerous NP problems proven to be complete. Among them are:

- Traveling Salesman
- Knapsack (**How we reduced the time from exponential to polynomial?**)
- Graph Coloring

Curiously, what they have in common, aside from being in NP, is that **each can be reduced into the other in polynomial time**. These facts together place them in NP-Complete.

NP-Hard Algorithms

- The last set of problems contains the hardest, most complex problems in computer science.
- They are not only hard to solve but are hard to verify as well.
- In fact, some of these problems aren't even decidable.

Among the hardest computer science problems are:

- K-means Clustering
- Traveling Salesman Problem, and
- Graph Coloring

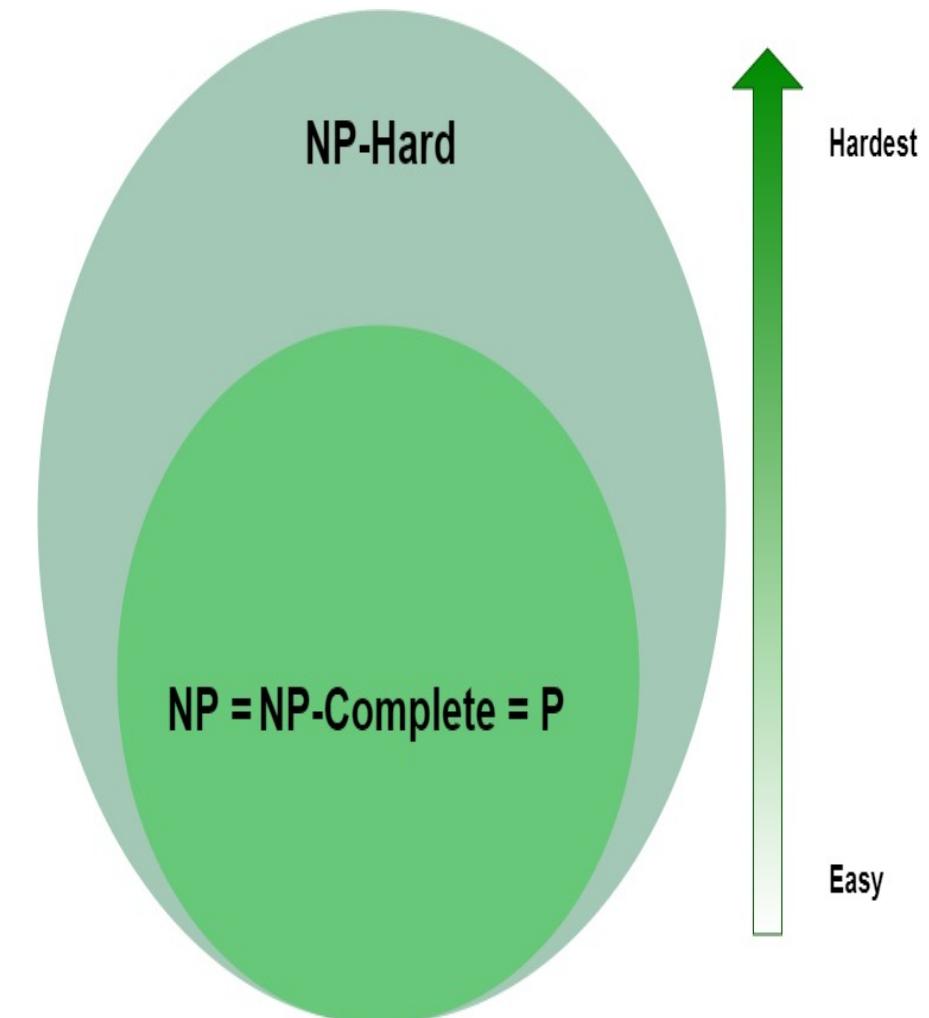
NP-Hard Algorithms

- These algorithms have a property similar to ones in NP-Complete - they can all be reduced to any problem in NP.
- Because of that, these are in NP-Hard and are at least as hard as any other problem in NP. A problem can be both in NP and NP-Hard, which is another aspect of being NP-Complete.

This characteristic has led to a debate about whether or not Traveling Salesman is indeed NP-Complete. Since NP and NP-Complete problems can be verified in polynomial time, proving that an algorithm cannot be verified in polynomial time is also sufficient for placing the algorithm in NP-Hard.

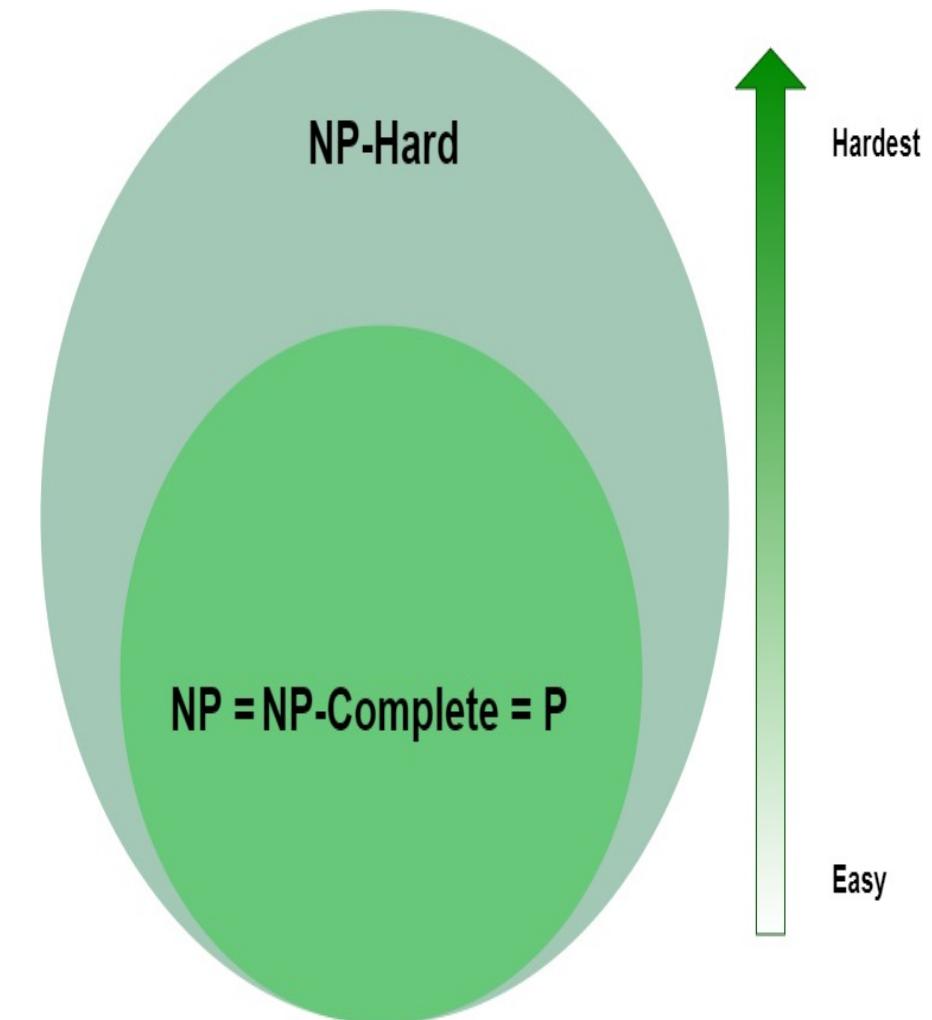
Does P=NP?

- The assumption is $P \neq NP$, however, $P=NP$ may be possible.
- If it were so, aside from NP or NP-Complete problems being solvable in polynomial time, certain algorithms in NP-Hard would also dramatically simplify.
- For example, if their verifier is NP or NP-Complete, then it follows that they must also be solvable in polynomial time, moving them into $P = NP = NP\text{-Complete}$ as well.



Does P=NP?

- If P=NP means a radical change in computer science and even in the real-world scenarios.
- Currently, some security algorithms have the basis of being a requirement of too long calculation time.
- Many encryption schemes and algorithms in cryptography are based on the number factorization which the best-known algorithm with exponential complexity. If we find a polynomial-time algorithm, these algorithms become vulnerable to attacks.



Conclusion

- P problems are quick to solve.
- NP problems are quick to verify but slow to solve.
- NP-Complete problems are also quick to verify, slow to solve and can be reduced to any other NP problem.
- NP-Hard problems are slow to verify, slow to solve and can be reduced to any other NP problem.

THANK YOU