Data Structures & Algorithms

Subodh Kumar

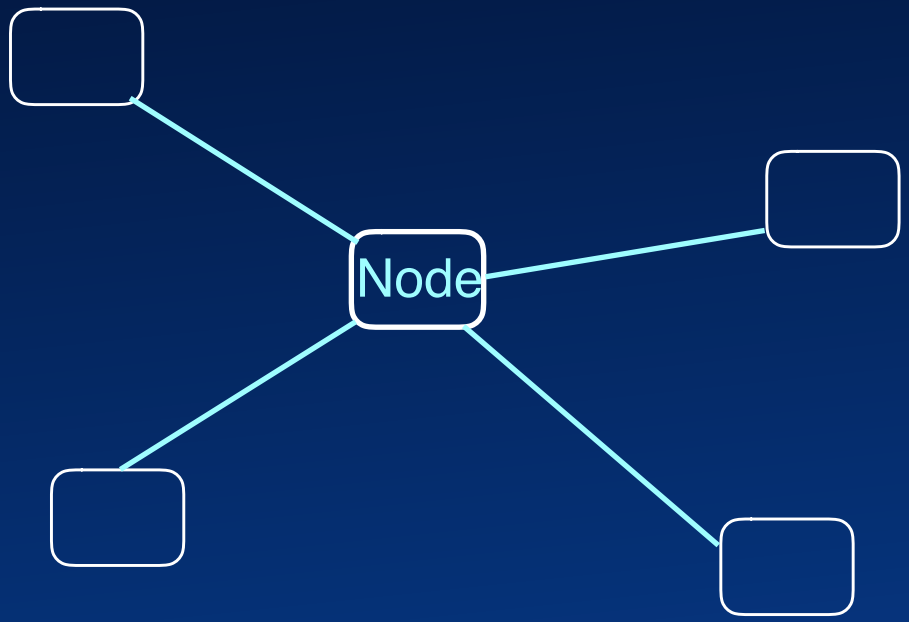
(subodh@iitd.ac.in, Bharti 422)

Dept of Computer Sc. & Engg.

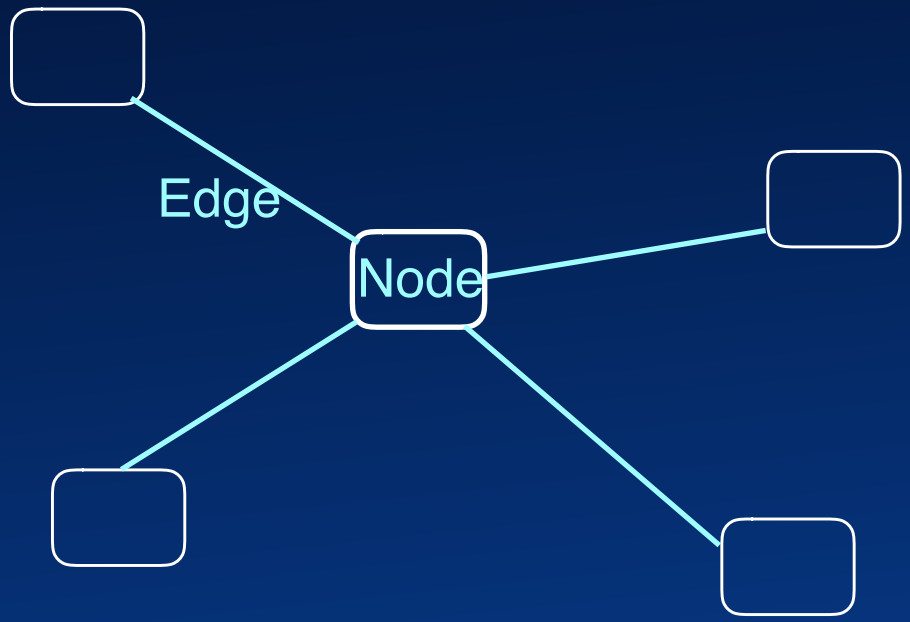
Graph

Graph

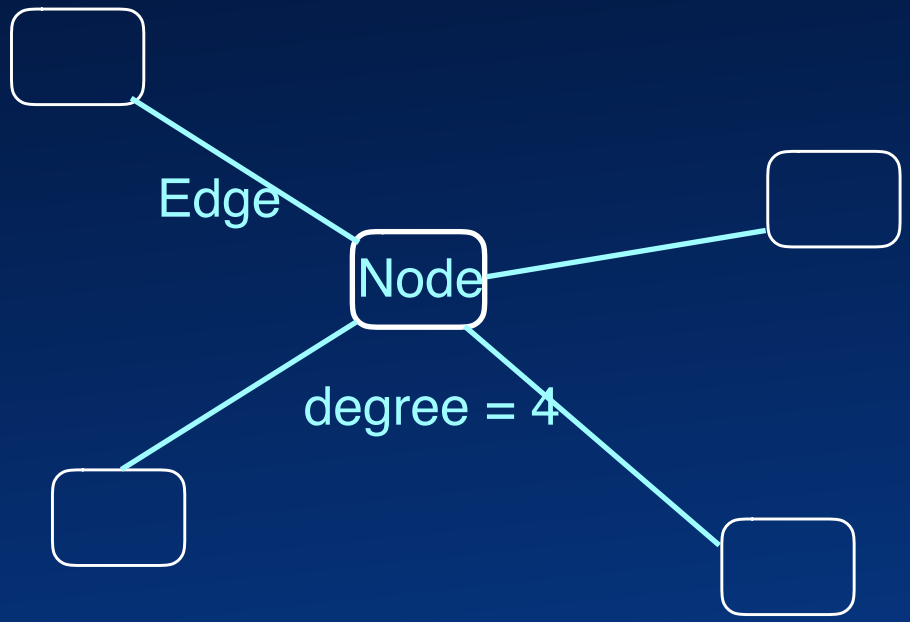
Graph



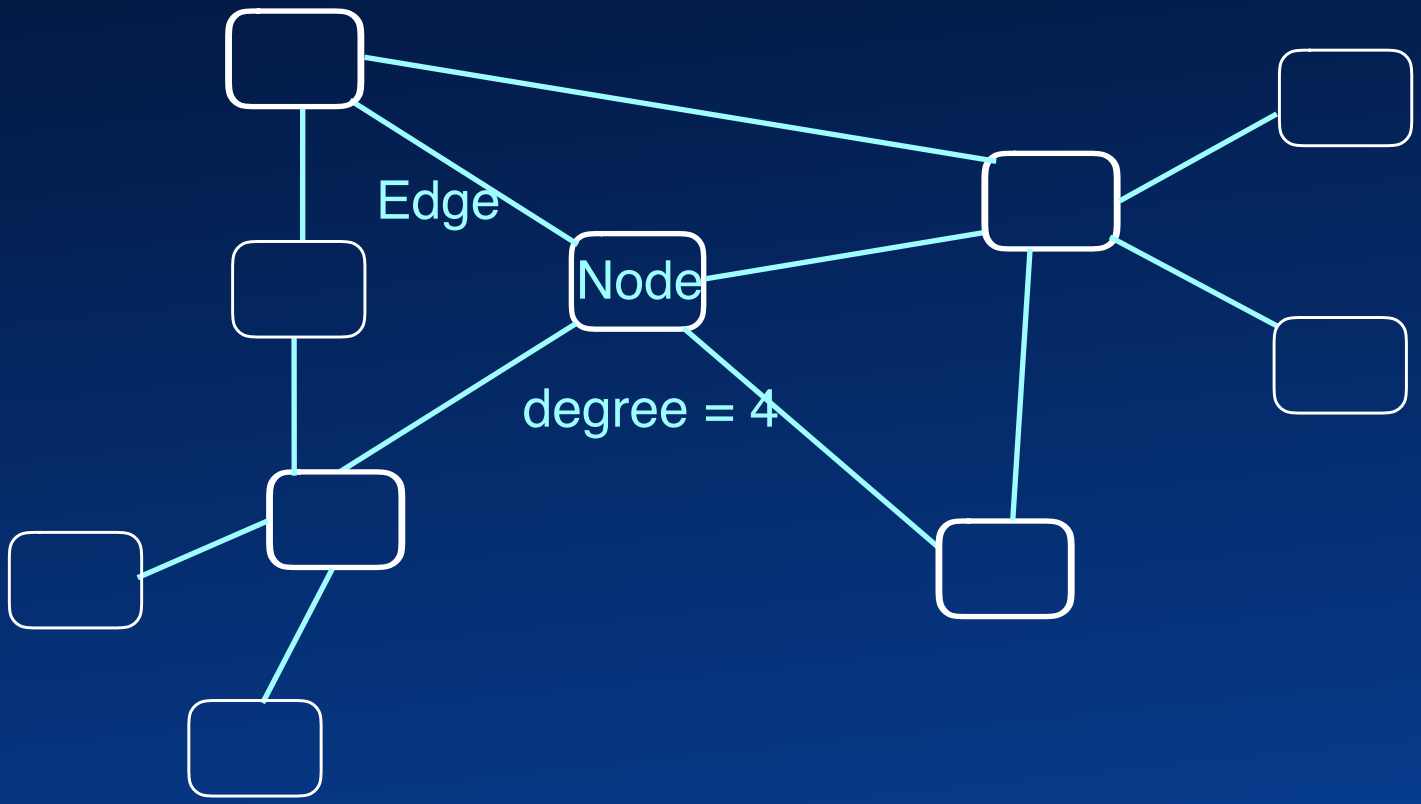
Graph



Graph

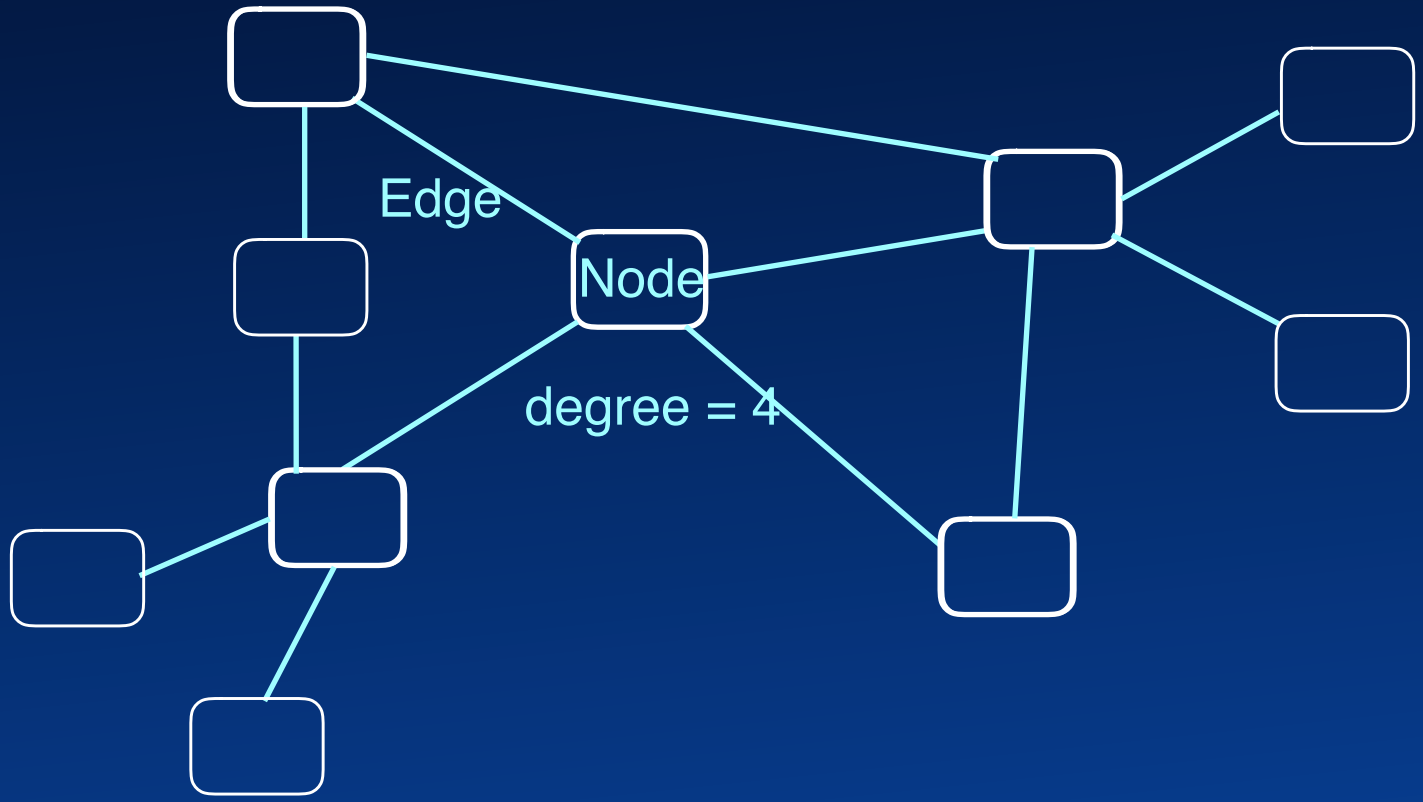


Graph



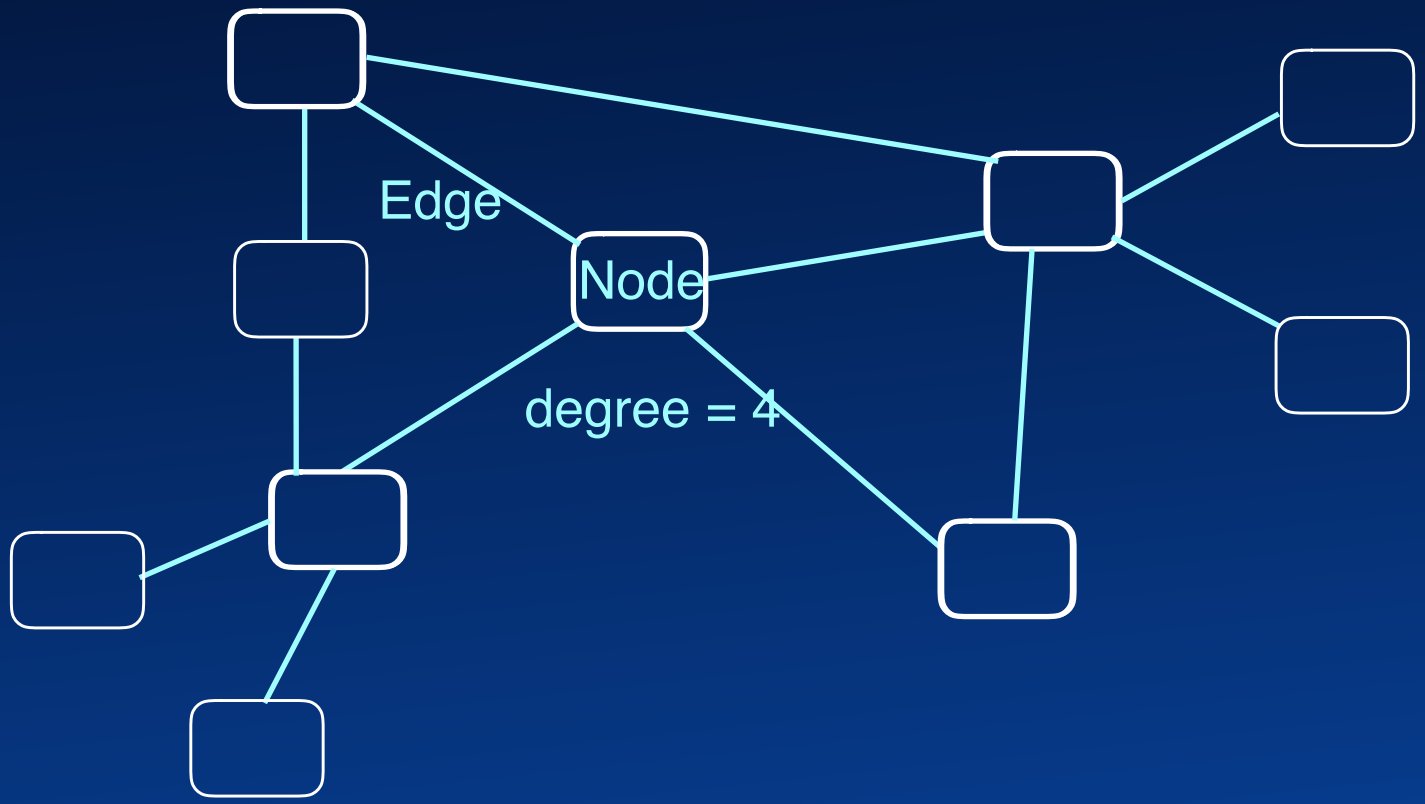
Graph

Undirected
(Symmetric)



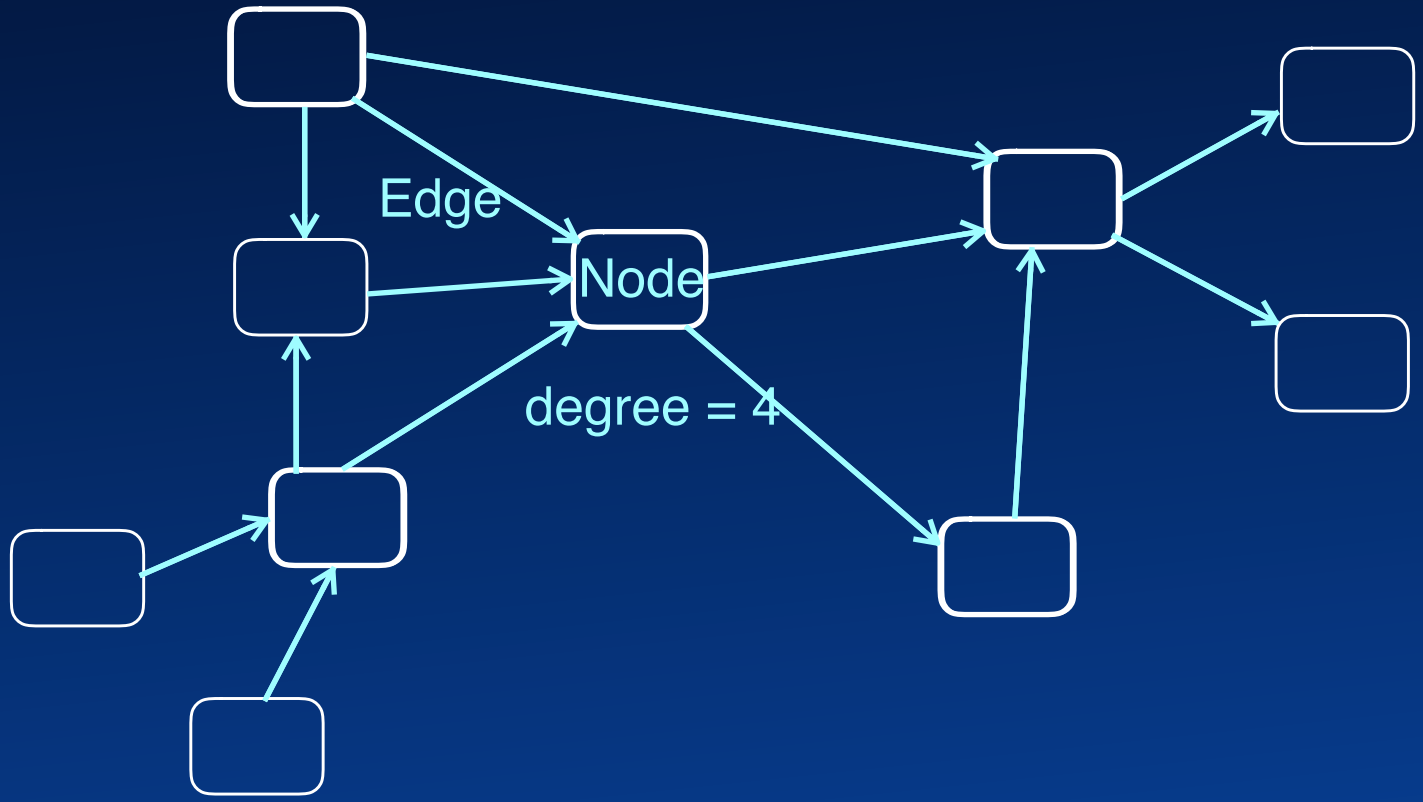
Graph

Undirected (Symmetric) = Undirected graph



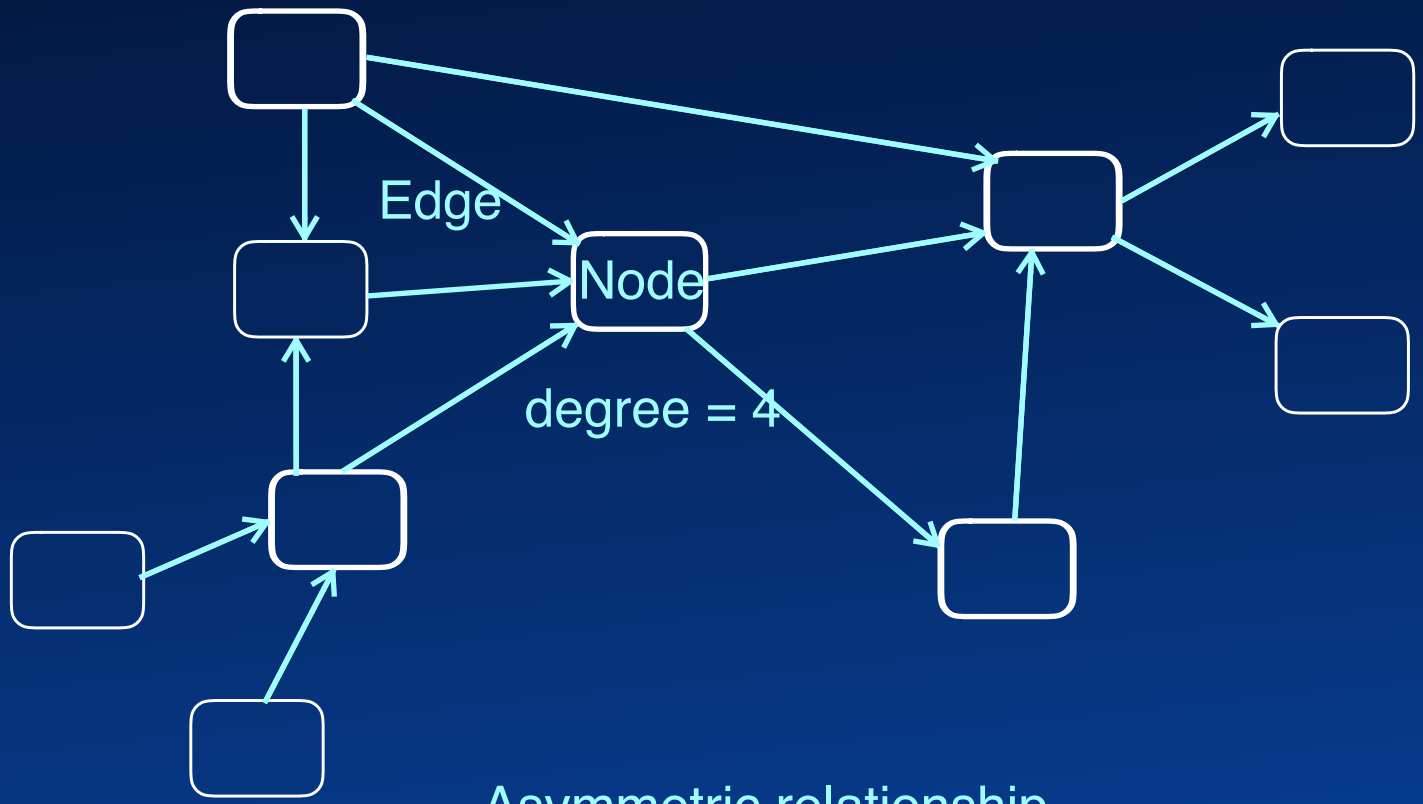
Graph

Undirected (Symmetric) = Undirected graph



Graph

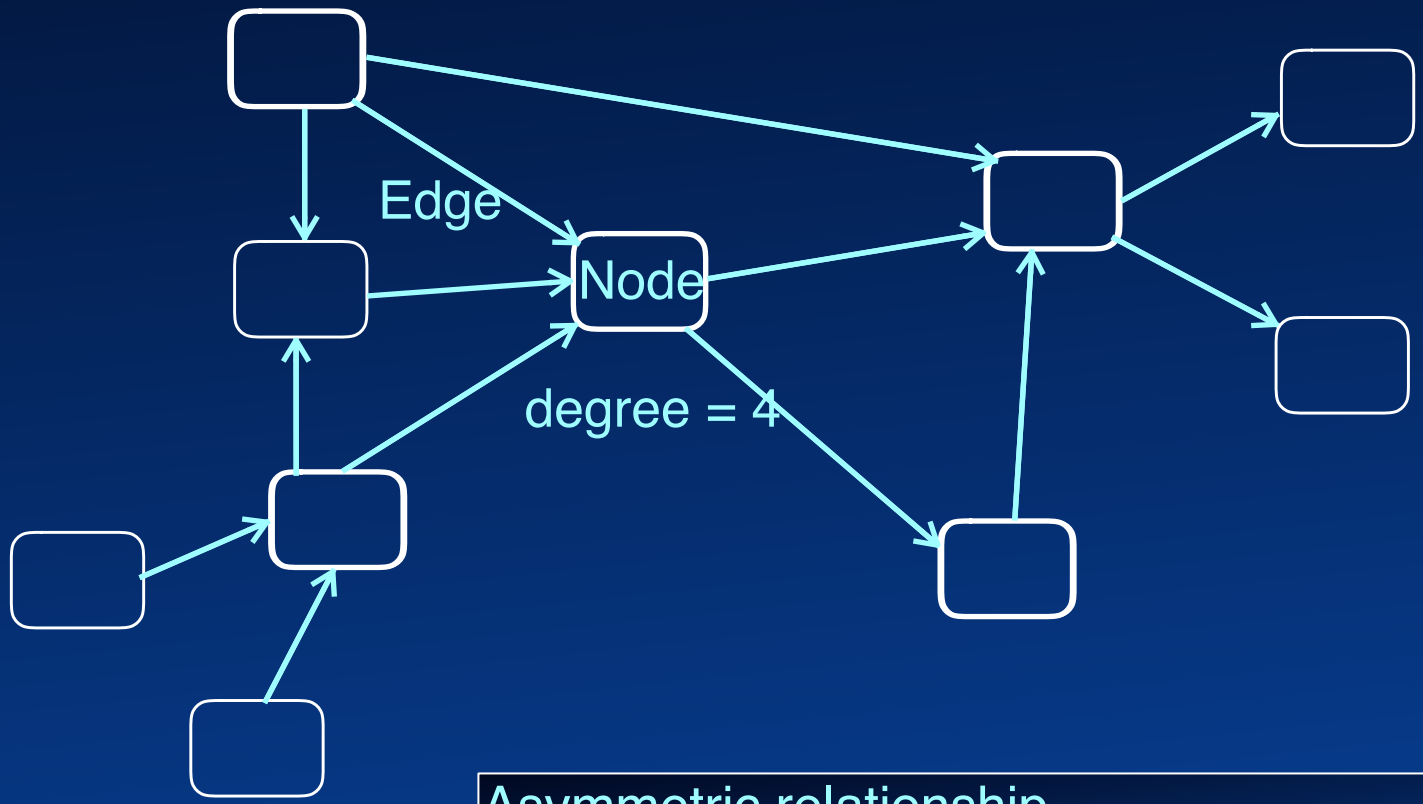
Undirected (Symmetric) = Undirected graph



Asymmetric relationship
(Edges are directed)

Graph

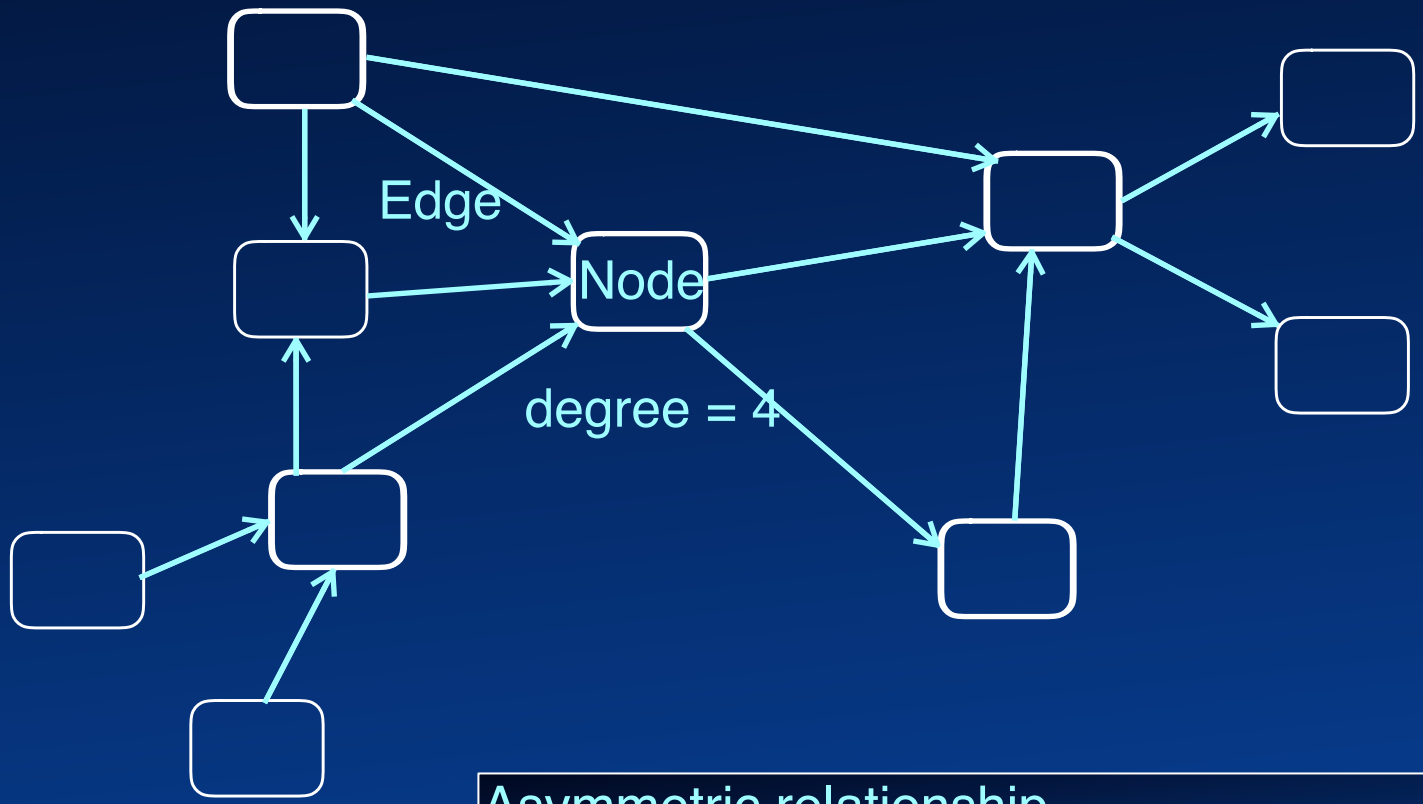
Undirected (Symmetric) = Undirected graph



Asymmetric relationship
(Edges are directed) = Directed graph

Graph

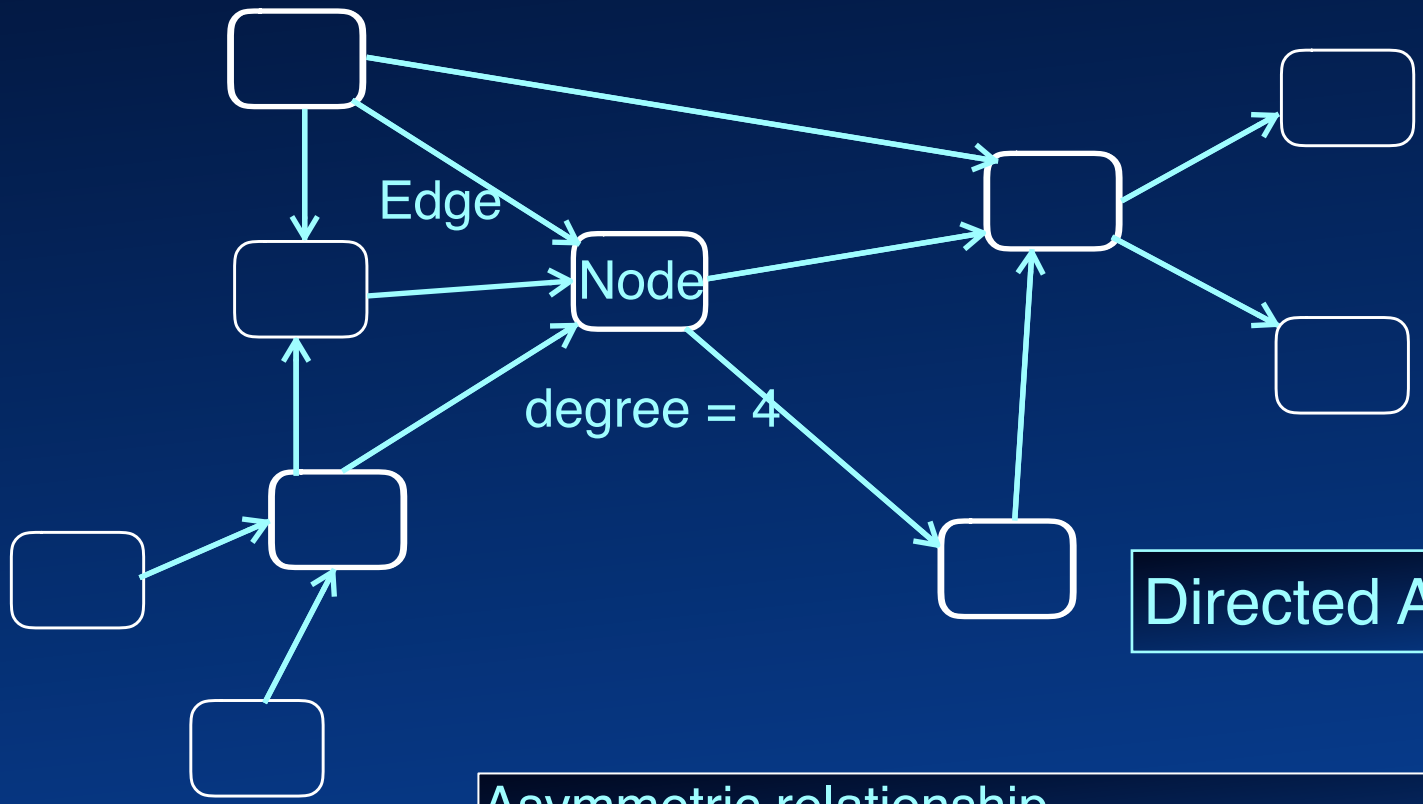
Undirected (Symmetric) = Undirected graph



Asymmetric relationship
(Edges are directed) = Directed graph e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph



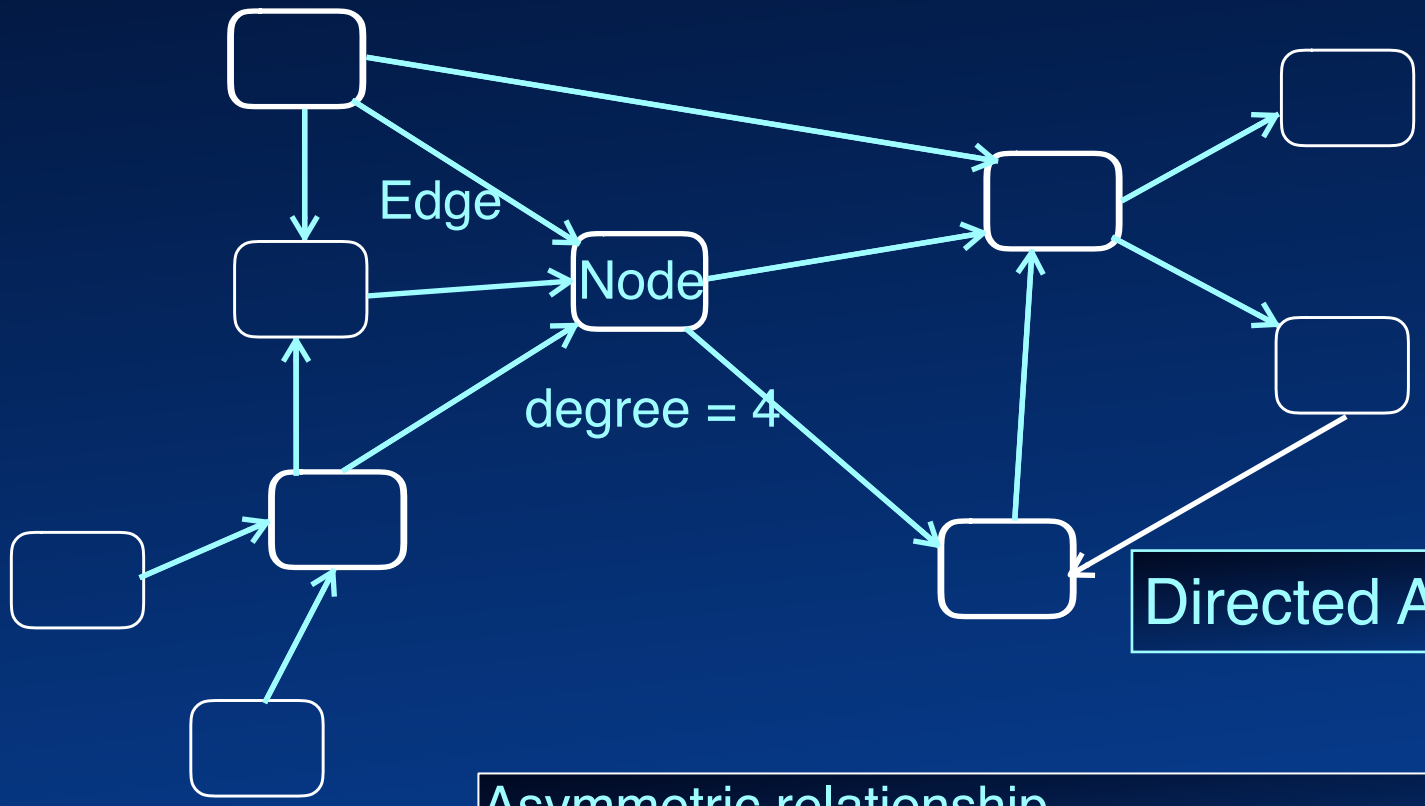
Directed Acyclic Graph

Asymmetric relationship
(Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph



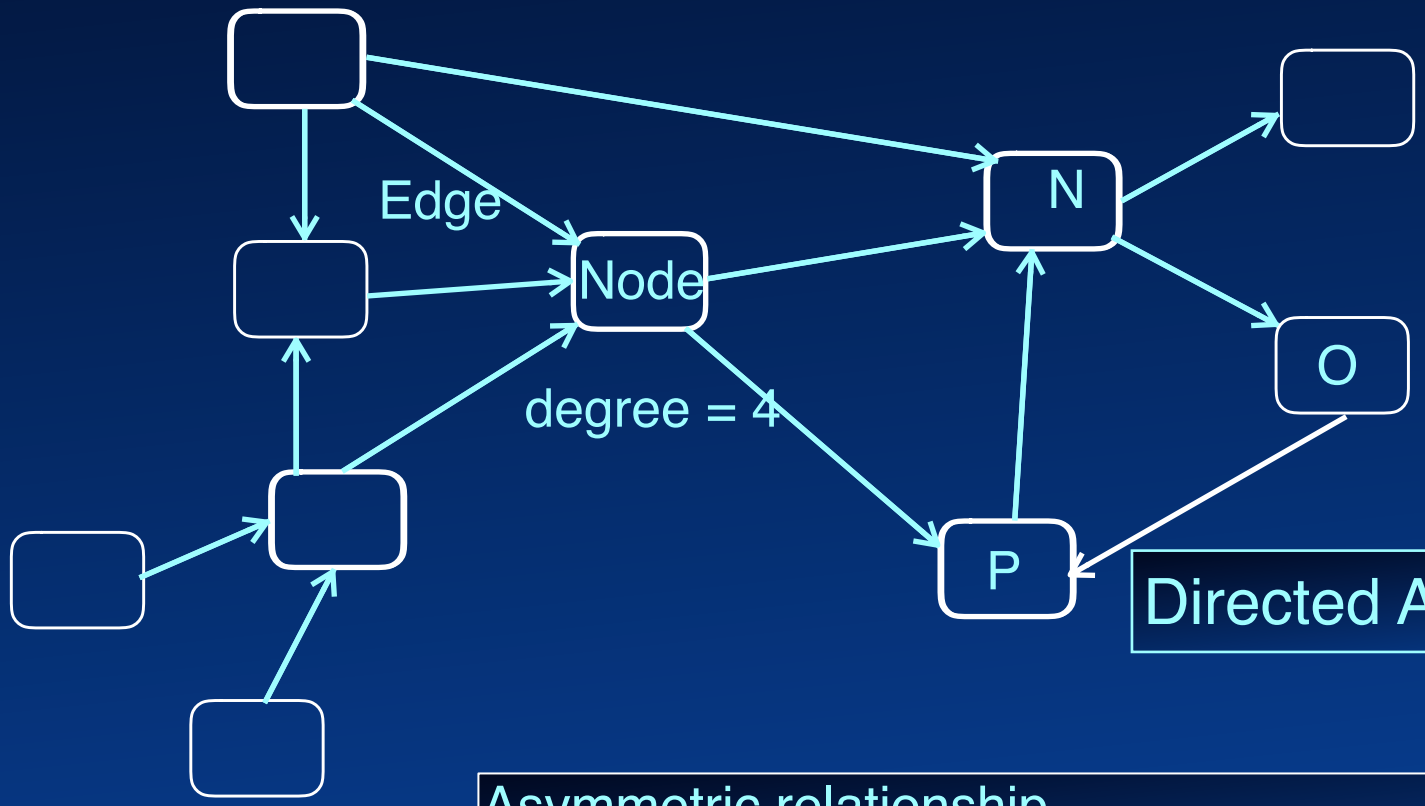
Directed Acyclic Graph

Asymmetric relationship
(Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph

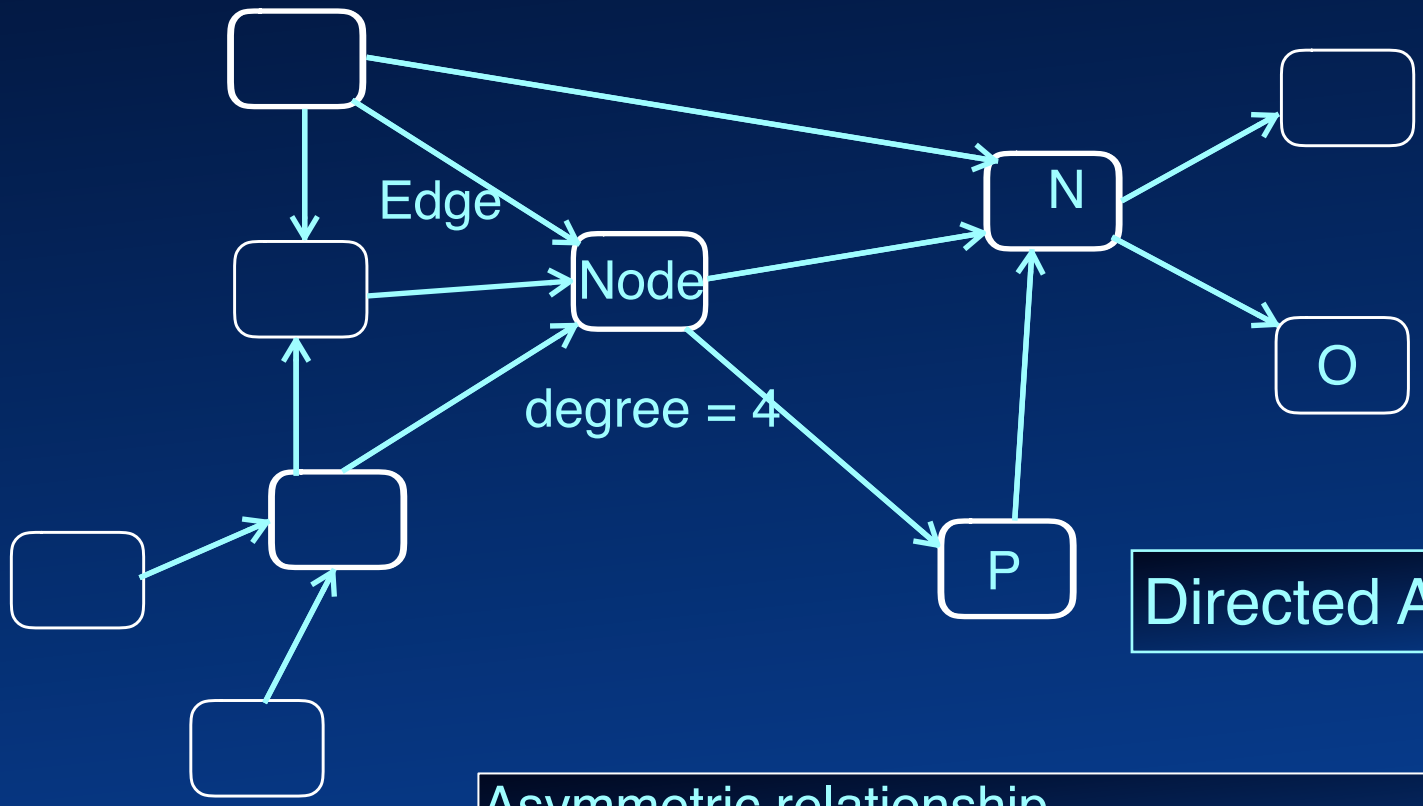


Directed Acyclic Graph

Asymmetric relationship
(Edges are directed) = Directed graph e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph

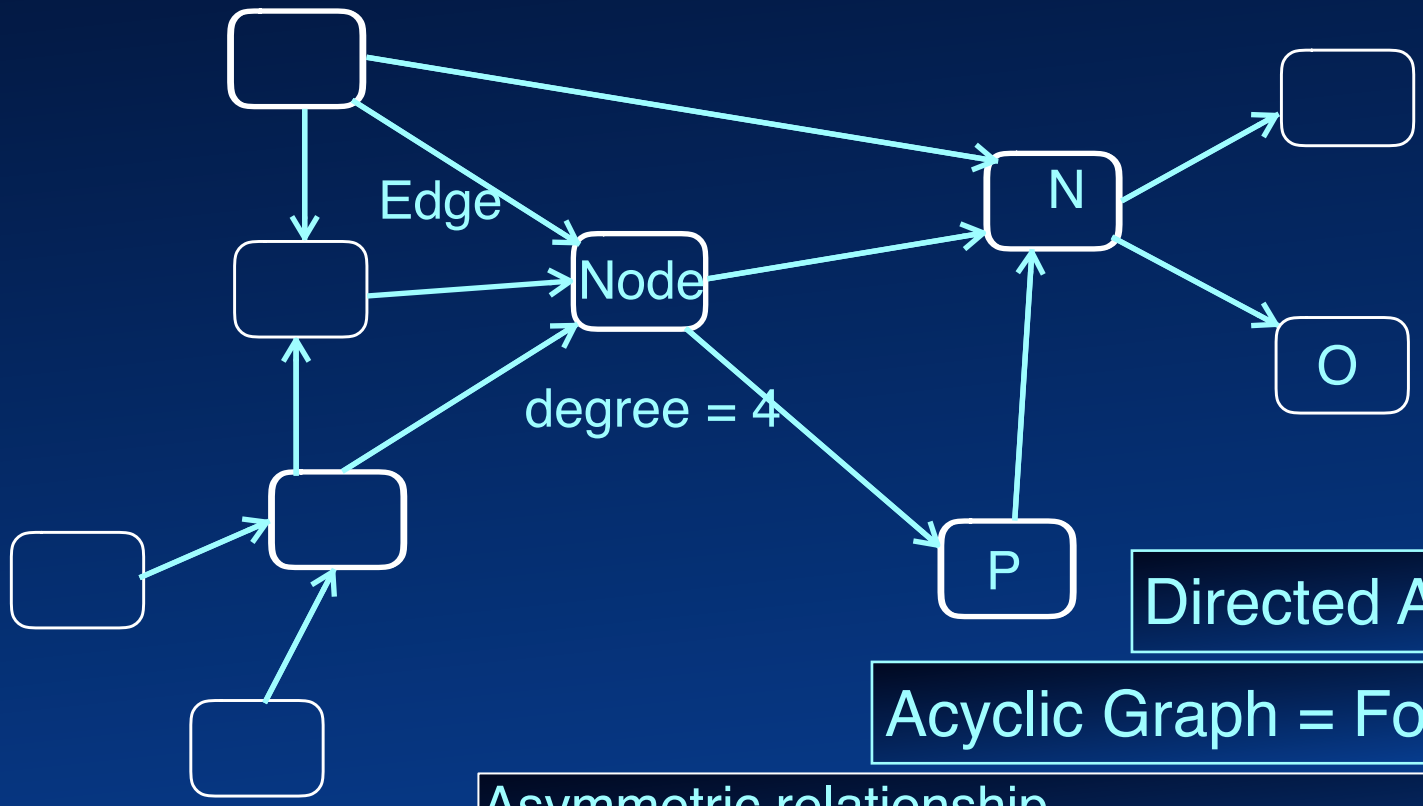


Directed Acyclic Graph

Asymmetric relationship
(Edges are directed) = Directed graph e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph



Directed Acyclic Graph

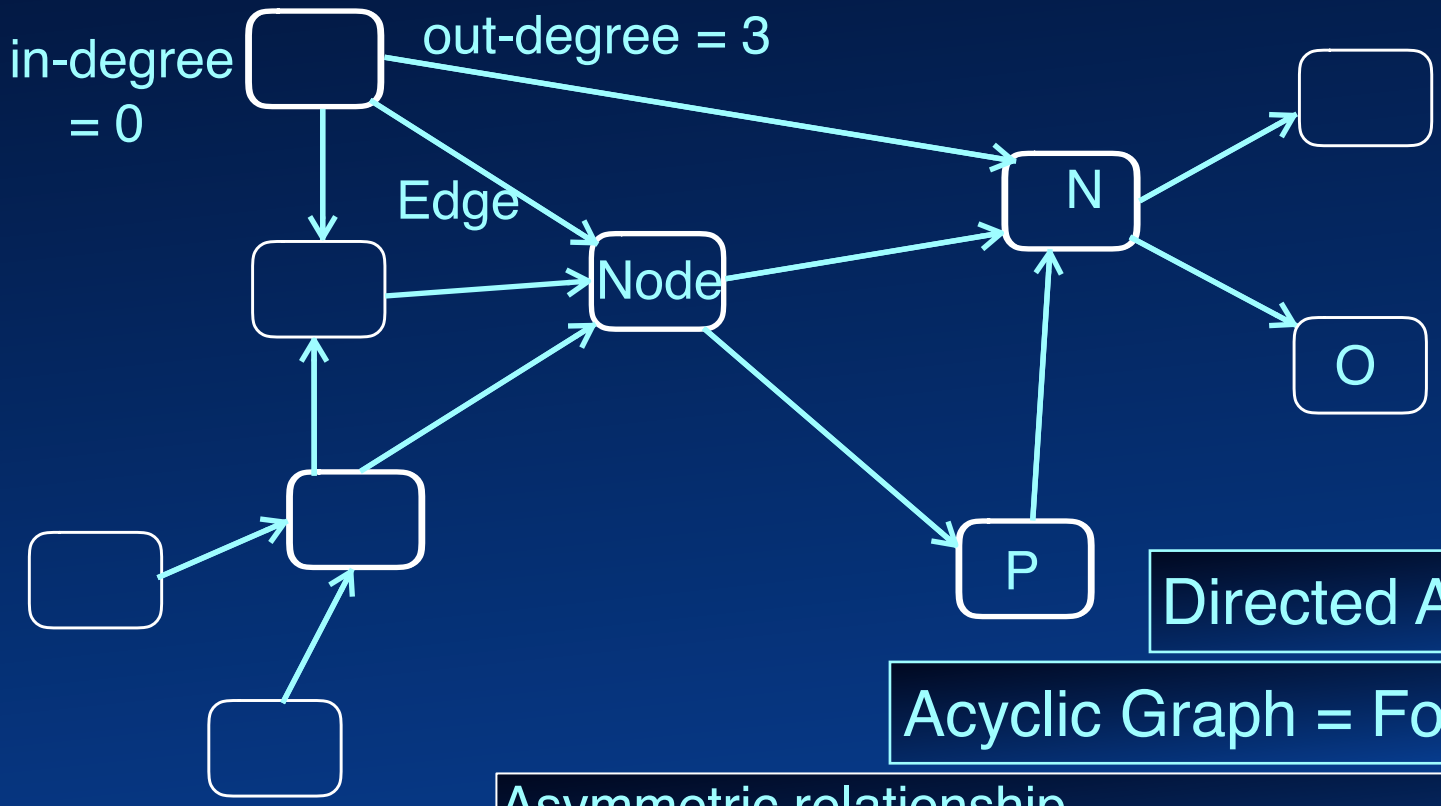
Acyclic Graph = Forest

Asymmetric relationship
(Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph



Directed Acyclic Graph

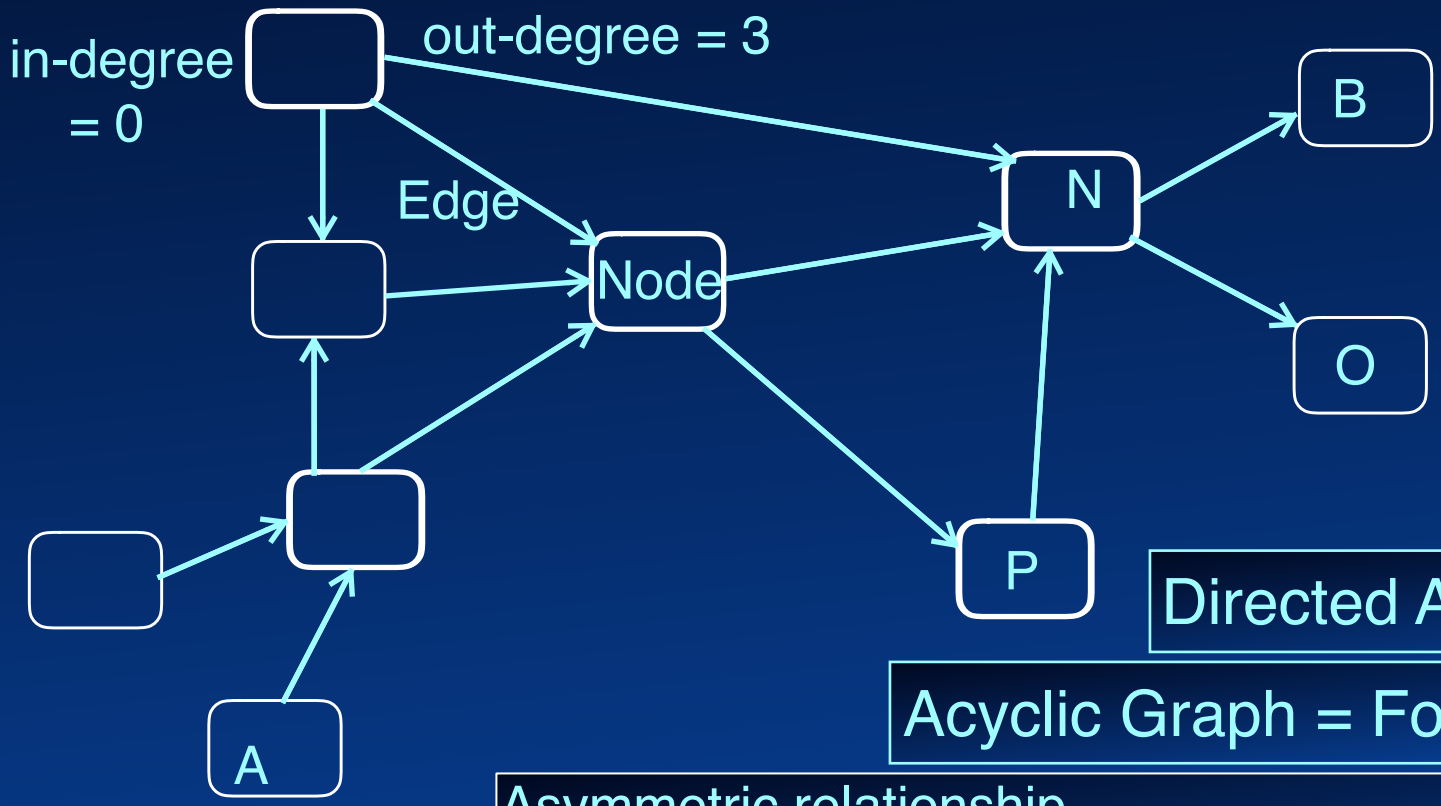
Acyclic Graph = Forest

Asymmetric relationship (Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph



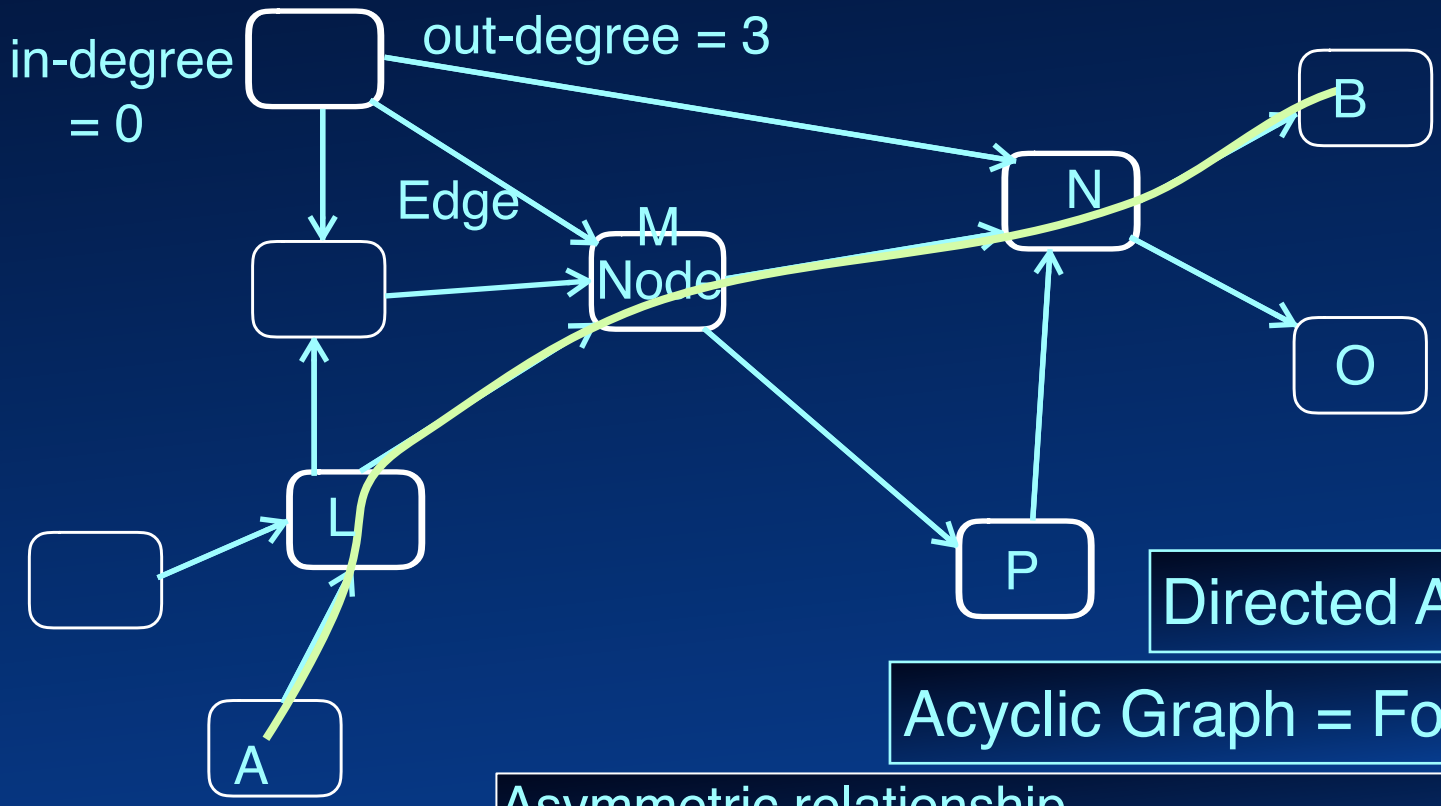
Directed Acyclic Graph

Acyclic Graph = Forest

Asymmetric relationship (Edges are directed) = Directed graph e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph



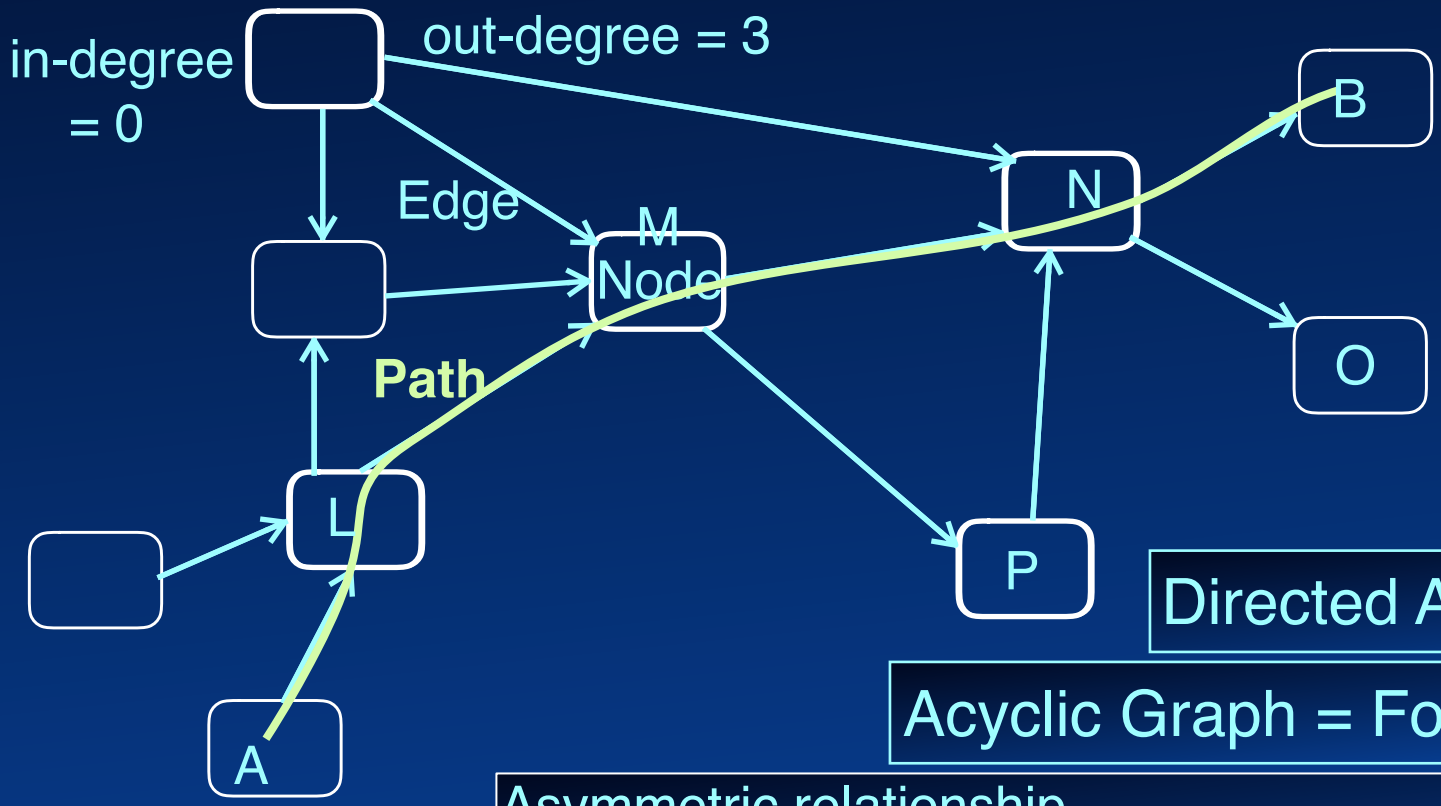
Directed Acyclic Graph

Acyclic Graph = Forest

Asymmetric relationship (Edges are directed) = Directed graph e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph



Directed Acyclic Graph

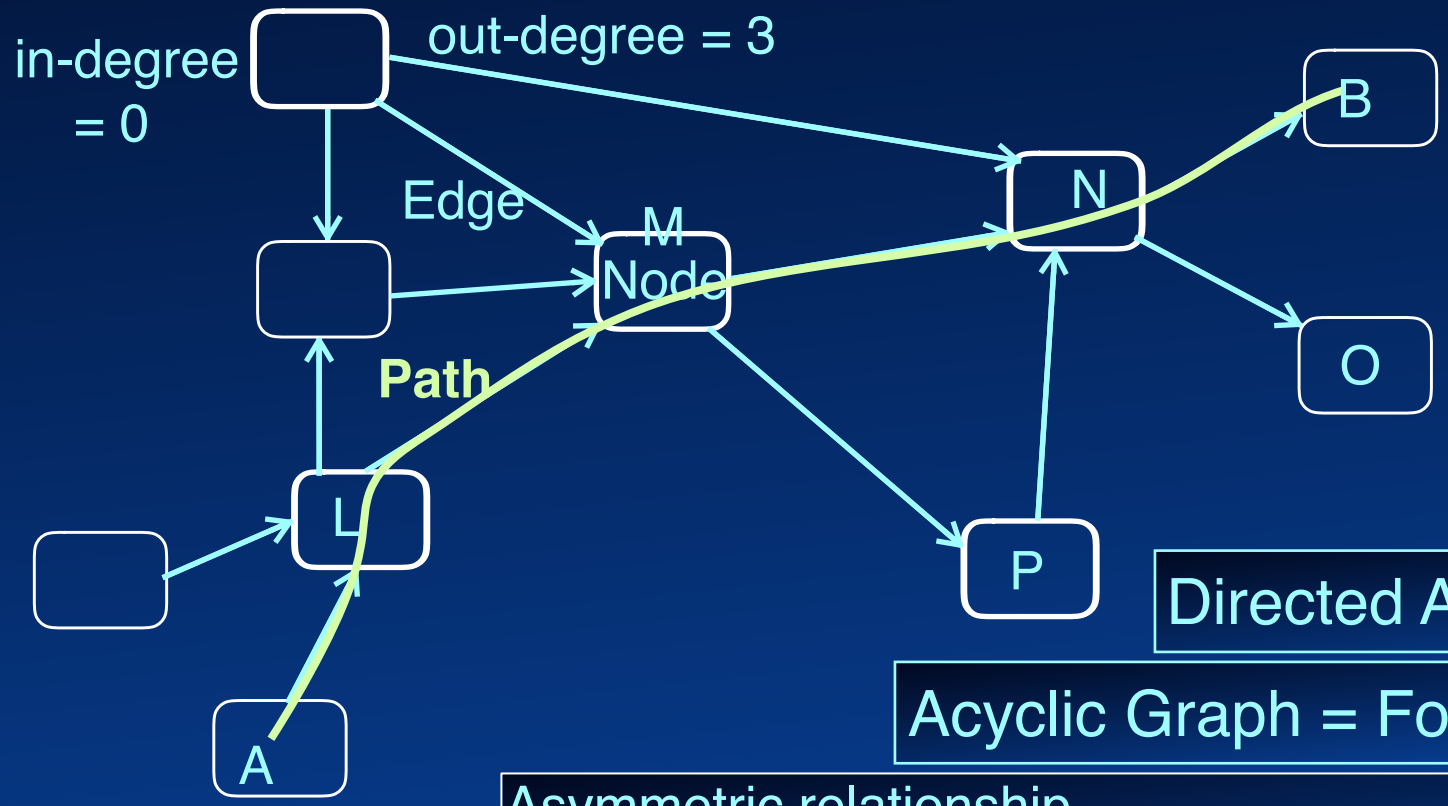
Acyclic Graph = Forest

Asymmetric relationship (Edges are directed) = Directed graph e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph

A is connected to B



Directed Acyclic Graph

Acyclic Graph = Forest

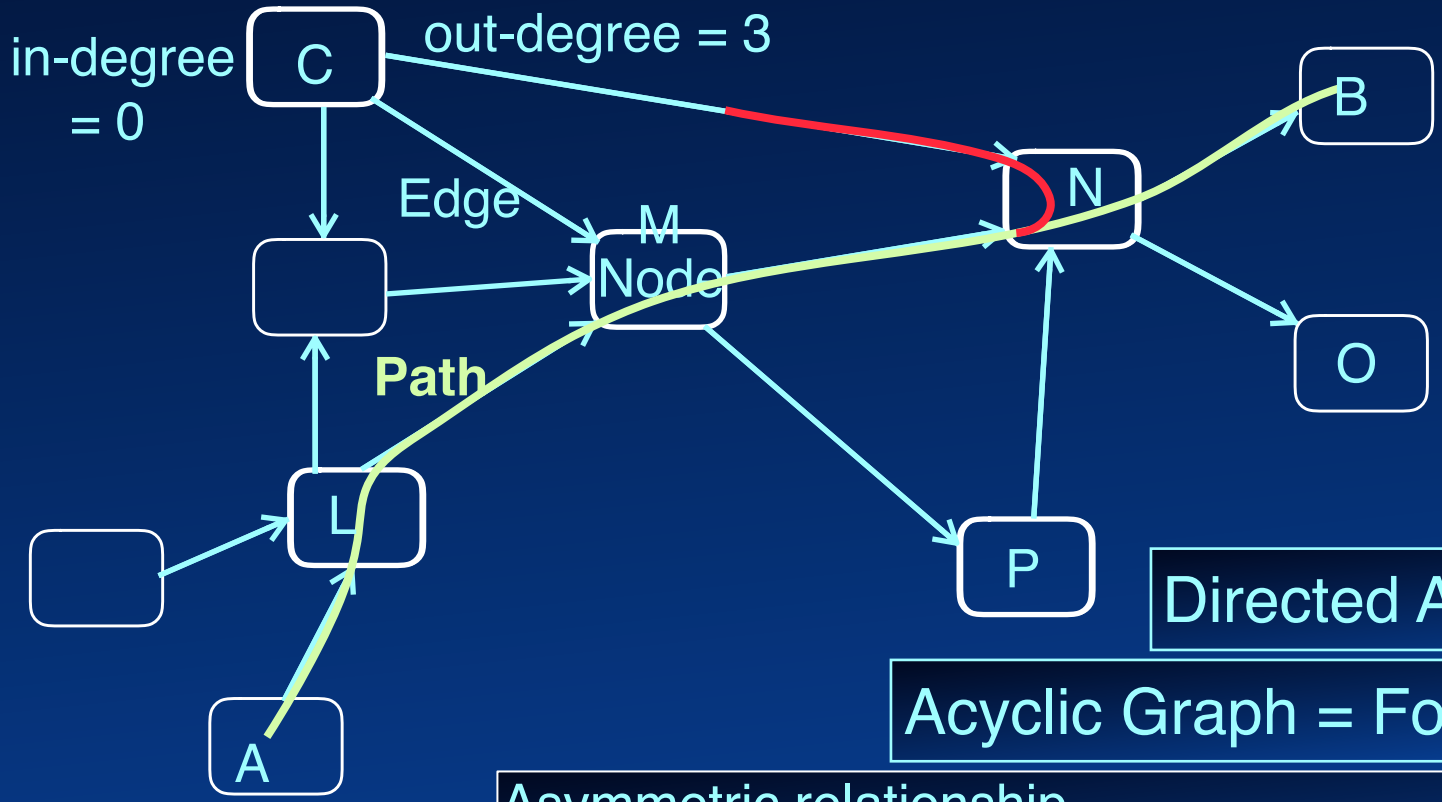
Asymmetric relationship
(Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph

A is connected to B



Directed Acyclic Graph

Acyclic Graph = Forest

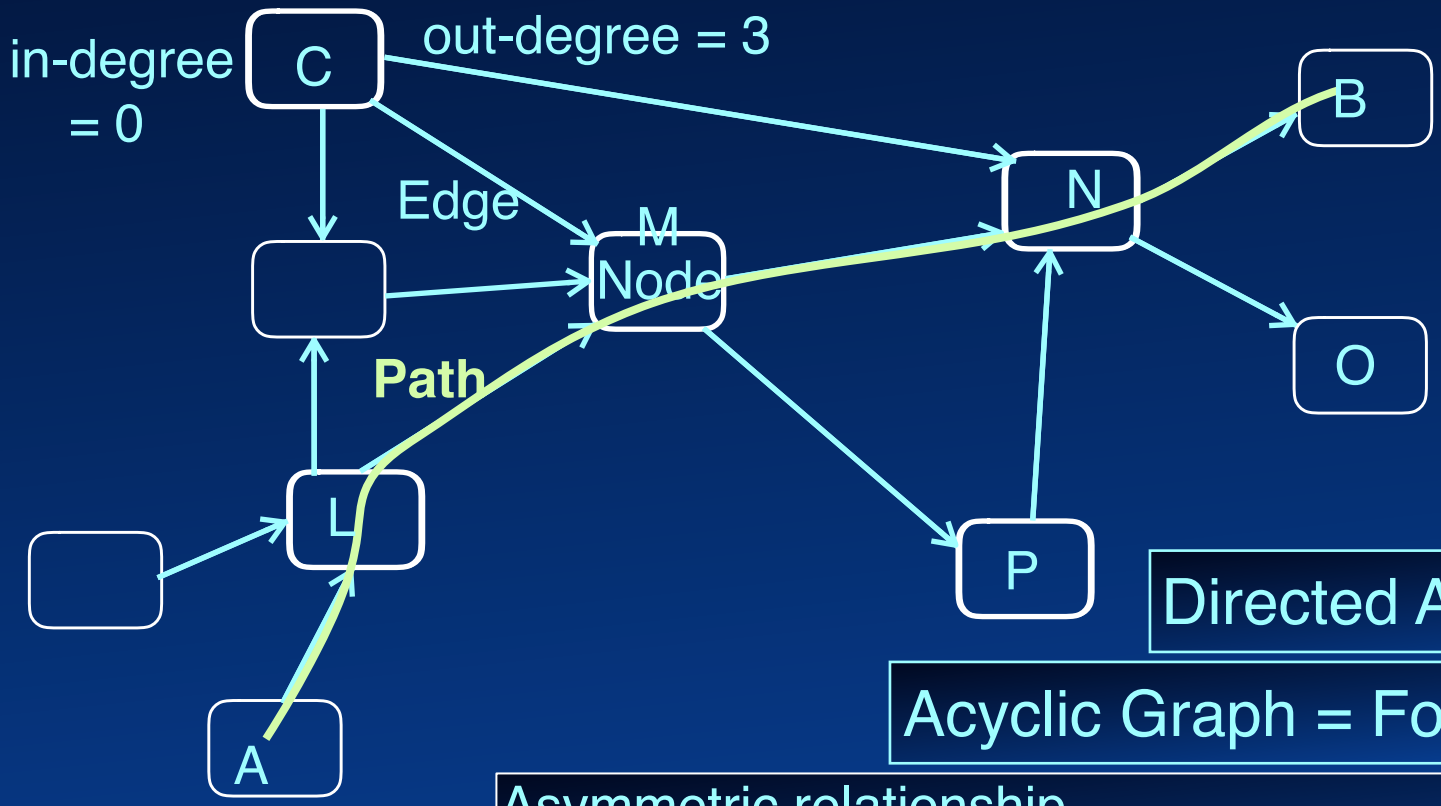
Asymmetric relationship (Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph

A is connected to B



Directed Acyclic Graph

Acyclic Graph = Forest

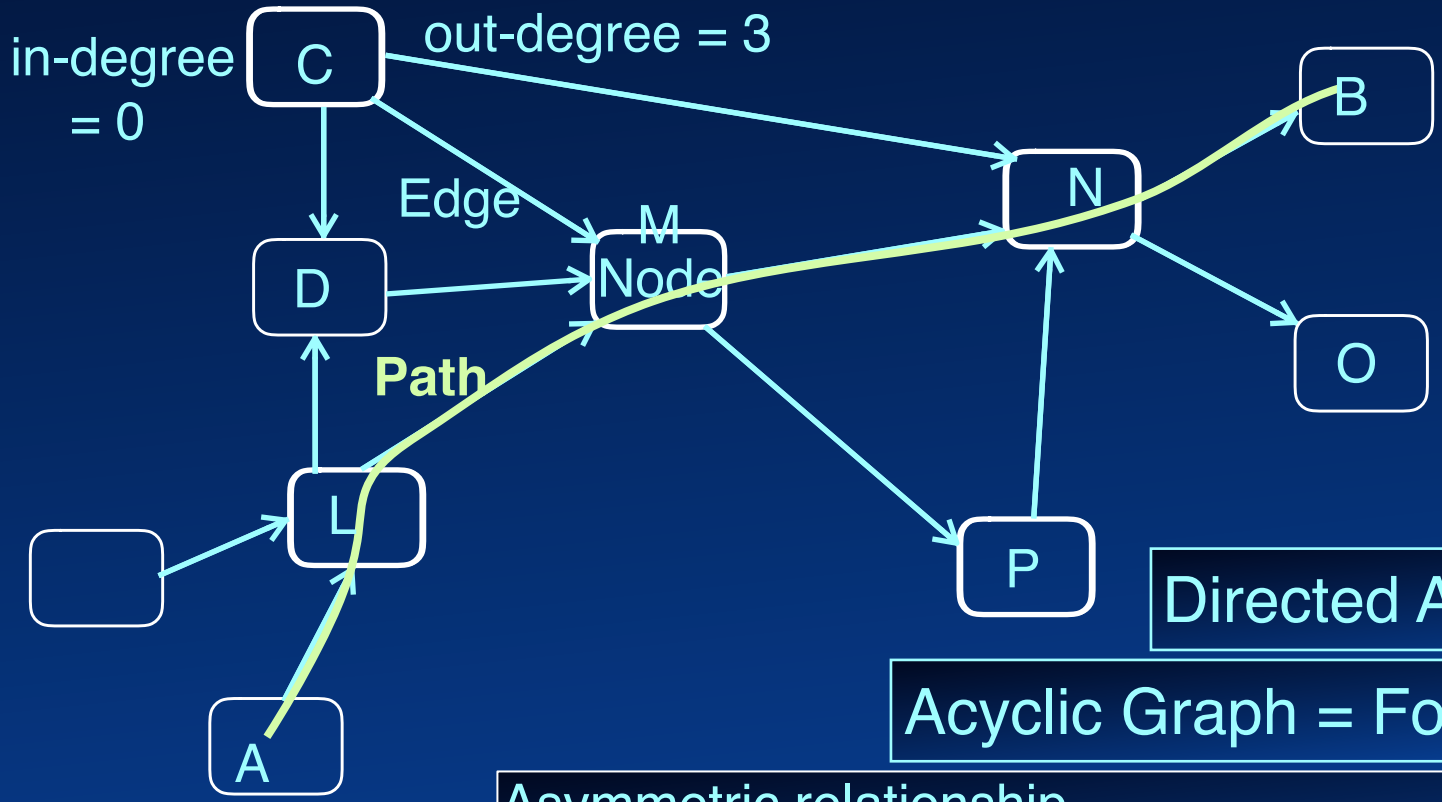
Asymmetric relationship
(Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph

A is connected to B



Directed Acyclic Graph

Acyclic Graph = Forest

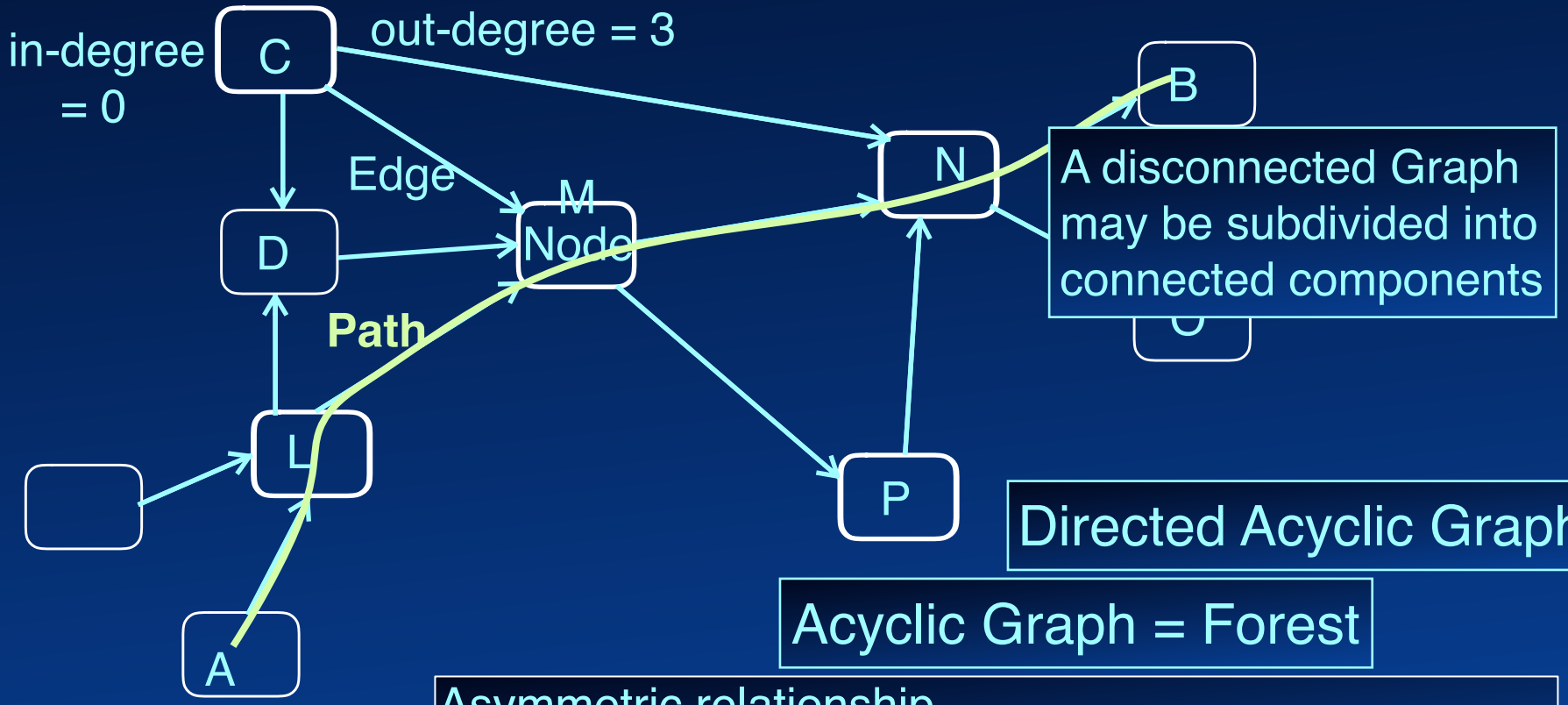
Asymmetric relationship (Edges are directed) = Directed graph

e.g., Tree

Graph

Undirected (Symmetric) = Undirected graph

A is connected to B

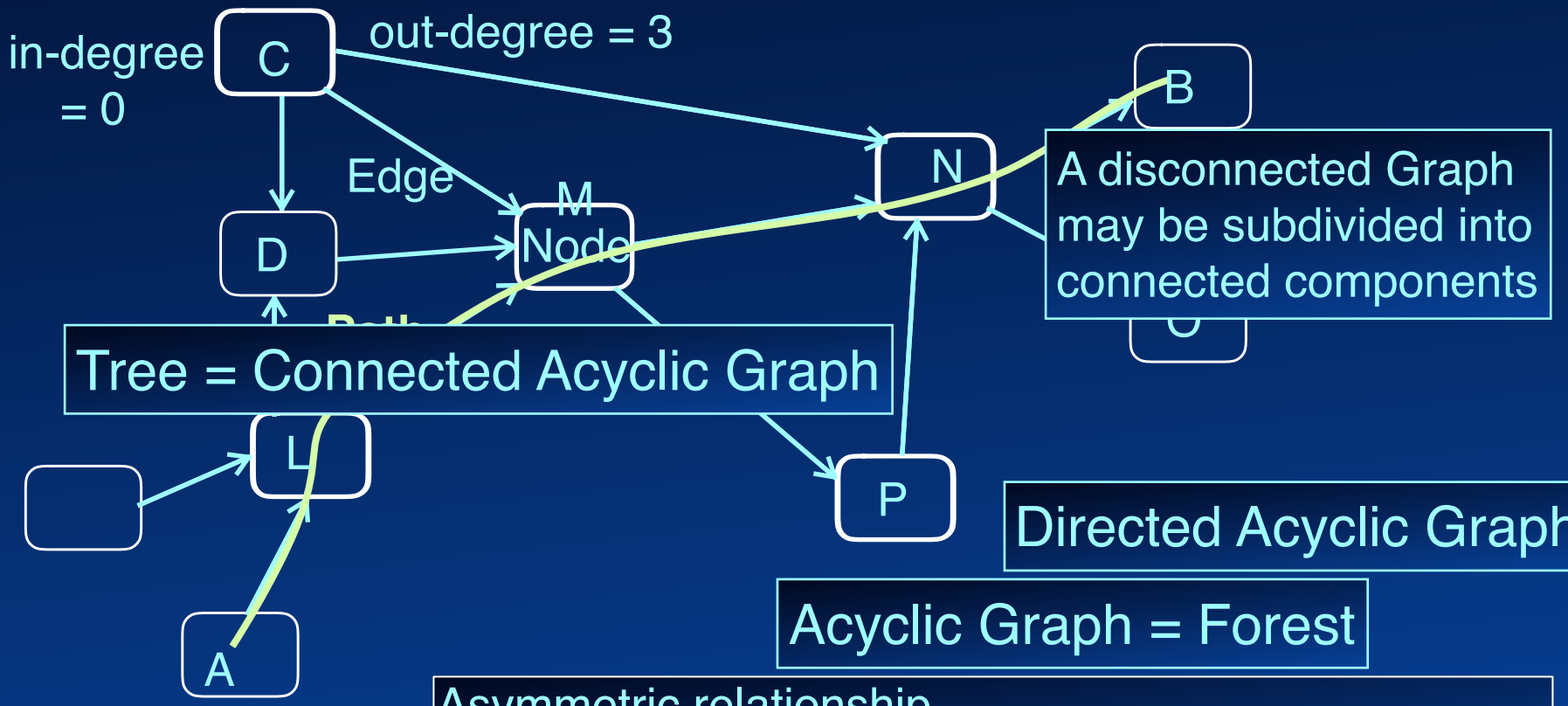


Directed Acyclic Graph

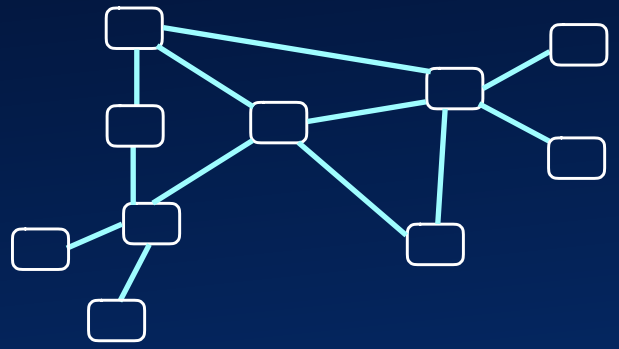
Acyclic Graph = Forest

Asymmetric relationship (Edges are directed) = Directed graph e.g., Tree

Graph

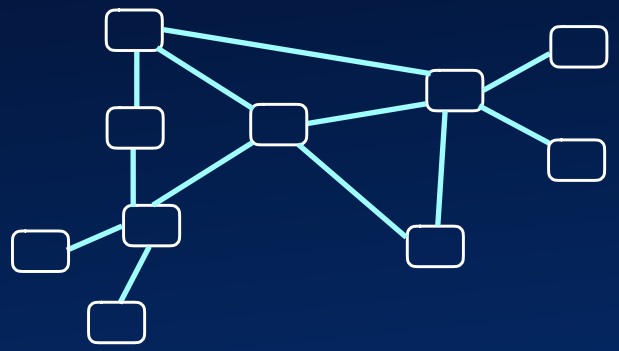

Undirected (Symmetric) = Undirected graph
A is connected to B
in-degree = 0
C
out-degree = 3
B
N
Edge
M
Node
D
A disconnected Graph may be subdivided into connected components
Tree = Connected Acyclic Graph
L
P
Directed Acyclic Graph
Acyclic Graph = Forest
A
Asymmetric relationship
(Edges are directed) = Directed graph
e.g., Tree

Graph Representations



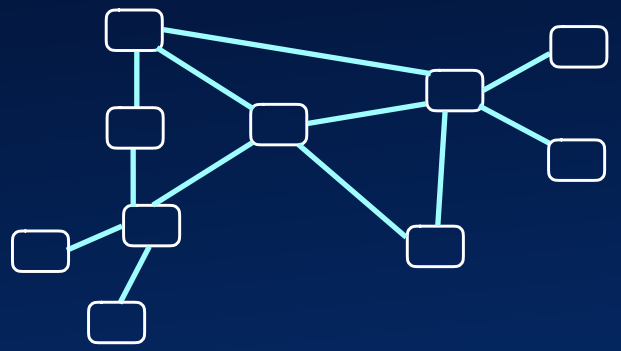
Graph Representations

```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```



Graph Representations

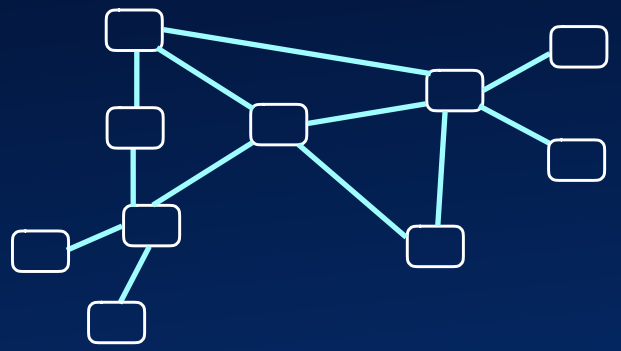
```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```



```
class NodE<NV,EV> {
    NV value;
    List<Pair<NodE,EV>> adjacent;
}
```

Graph Representations

```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```

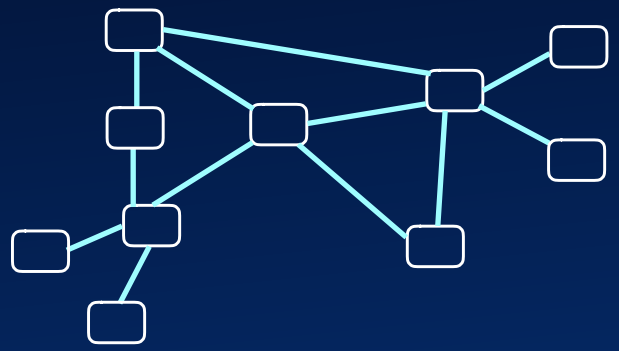


```
class NodE<NV,EV> {
    NV value;
    List<Pair<NodE,EV>> adjacent;
}
```

```
class Edge<N,EV> {
    EV value;
    N source;
    N dest;
}
```

Graph Representations

```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```



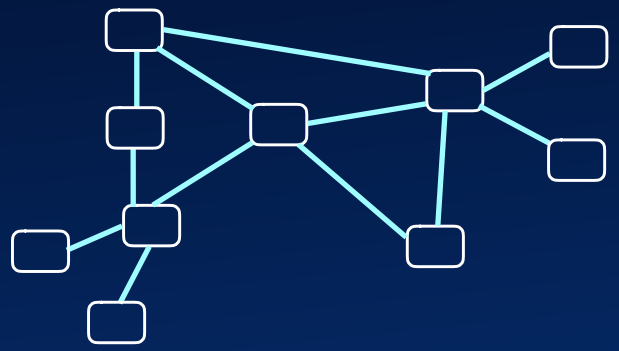
```
class NodE<NV,EV> {
    NV value;
    List<Pair<NodE,EV>> adjacent;
}
```

```
class Edge<N,EV> {
    EV value;
    N source;
    N
}
```

```
class Edge<NV,EV> {
    EV value;
    Node<NV> source;
    Node<NV> dest;
}
```

Graph Representations

```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```



```
class NodE<NV,EV> {
    NV value;
    List<Pair<NodE,EV>> adjacent;
}
```

```
class Graph<NV,EV> {
    List<Node<NV>> nodes;
    List<Edge<NV,EV>> edges;
}
```

```
class Edge<N,EV> {
    EV value;
    N source;
    N
}
```

```
class Edge<NV,EV> {
    EV value;
    Node<NV> source;
    Node<NV> dest;
}
```

Graph Representations

```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```

```
class NodE<NV,EV> {
    NV value;
    List<Pair<NodE,EV>> adja
}
```

```
class Graph<NV,EV> {
    List<Node<NV>> nodes;
    List<Edge<NV,EV>> edges;
}
```

```
class Graph<NV,EV> {

    class Edge<EV> {
        int source, dest;
        EV value;
    }

    ArrayList<NV> nodes;
    List<Edge<EV>> edges;
}
```

```
EV> {
    EV value;
    Node<NV> source;
    Node<NV> dest;
}
```

Graph Representations

```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```

```
class Graph<NV,EV> {

    class Edge<EV> {
        int source, dest;
        EV value;
    }

    ArrayList<NV> nodes;
    List<Edge<EV>> edges;
```

```
class NodE<NV,EV> {
    NV value;
    List<Pair<NodE,EV>> adja
}
```

```
class Graph<NV,EV> {
    List<Node<NV>> nodes;
    Map<Node<NV>, EV> edges;
}
```

```
class Grapl
    List<Noc
    List<Edg
}
```

```
EV> {
EV value;
Node<NV> source;
Node<NV> dest;
```

Graph Representations

```
class Node<NV> {
    NV value;
    List<Node> adjacent;
}
```

```
class Graph<NV,EV> {

    class Edge<EV> {
        int source, dest;
        EV value;
    }
```

```
class NodE<NV,EV> {
    NV value;
    List<Pair<Noc
}
```

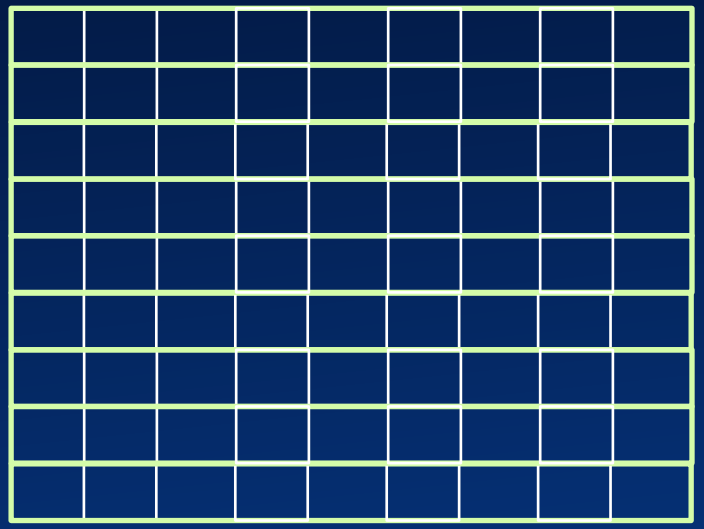
```
class NodE<NV,EV> {
    NV value;
    List<Edge<NodE,EV>> incident;
}
```

```
class Grapl
    List<Noc
    List<Edg
}
```

```
class
    List<Node<NV>> nodes;
    Map<Node<NV>, EV> edges;
}
```

```
EV> {
    EV value;
    Node<NV> source;
    Node<NV> dest;
}
```

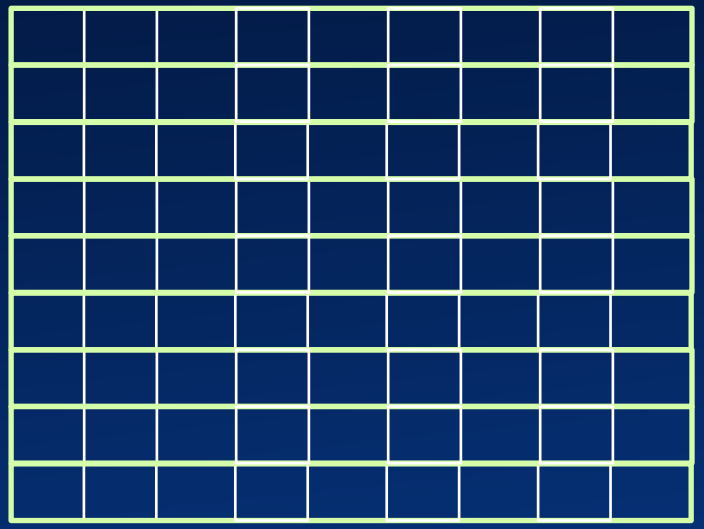
Adjacency Matrix



edges[i][j] contains special value
if node i and node j are not adjacent

Adjacency Matrix

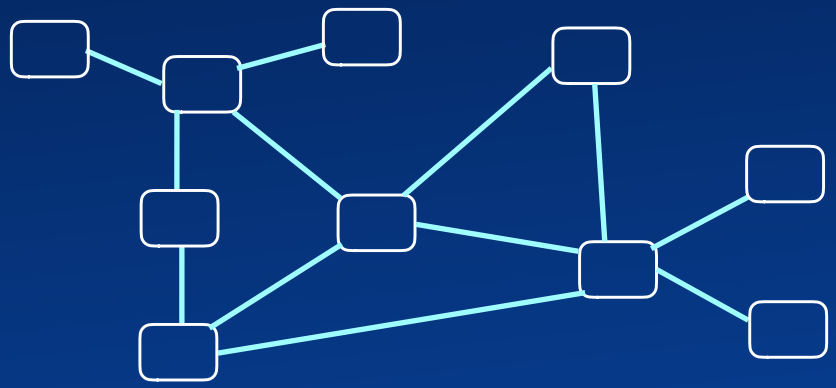
```
class Graph<NV,EV> {

    class Edge<EV> {
        int source, dest;
        EV value;
    }

    ArrayList<NV> nodes;
    Array2D<EV> edges;
}
```



edges[i][j] contains special value
if node i and node j are not adjacent

Graph Size

- #Edges: m
- #vertices: n
- $m = O(n^2)$

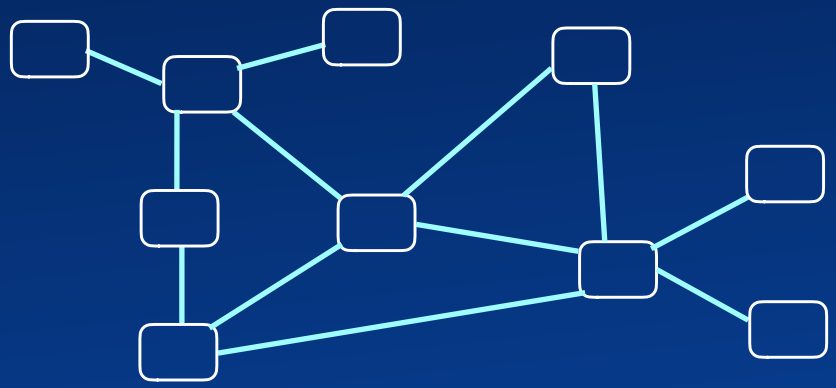


Graph Size

- #Edges: m
- #vertices: n
- $m = O(n^2)$

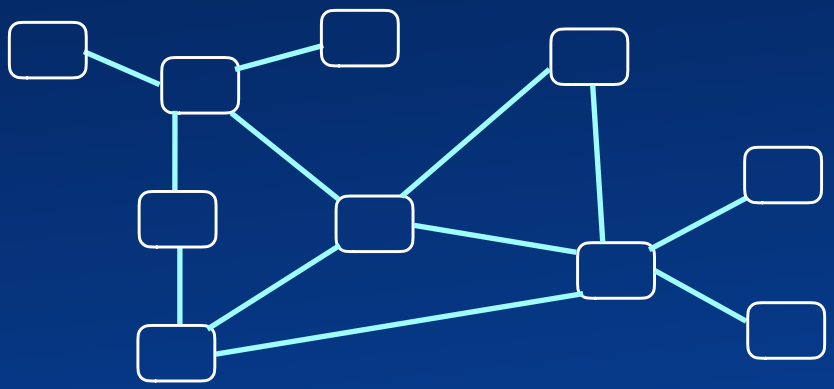
- Undirected graph

$$\sum_{v \in G} \text{degree}(v) = 2m$$



Graph Size

- #Edges: m
- #vertices: n
- $m = O(n^2)$



- Undirected graph

$$\sum_{v \in G} \text{degree}(v) = 2m$$

- Directed Graph:

$$\sum_{v \in G} \text{indegree}(v) = m$$

$$\sum_{v \in G} \text{outdegree}(v) = m$$

Graph Queries

- Iterator<Node> neighbor_nodes(Node n)
- Iterator<Edge> incident_edges(Node n)
- boolean isAdjacent(Node n1, Node n2)
- boolean isConnected(Node n1, Node n2)
- EV edgeValue(Node n1, Node n2)
- List<Node> apath(Node n1, Node n2)
- List<Node> shortestPath(Node n1, Node n2)
- void traverse()
- void traverse(Node n)

T/F

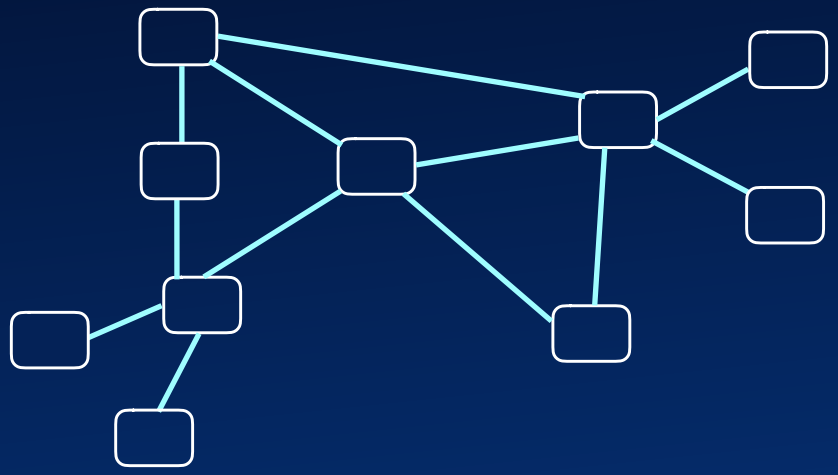
- An AVL tree is also a directed acyclic graph
- A directed acyclic graph is also a tree
- A graph with n vertices and n-1 edges is connected

Mail: col106quiz@cse.iitd.ac.in
Format: t,f,f

Traversal

- **Start at any vertex**

```
traverse(v):
    process(v)
    for u in v.adjacent()
        traverse(u)
```



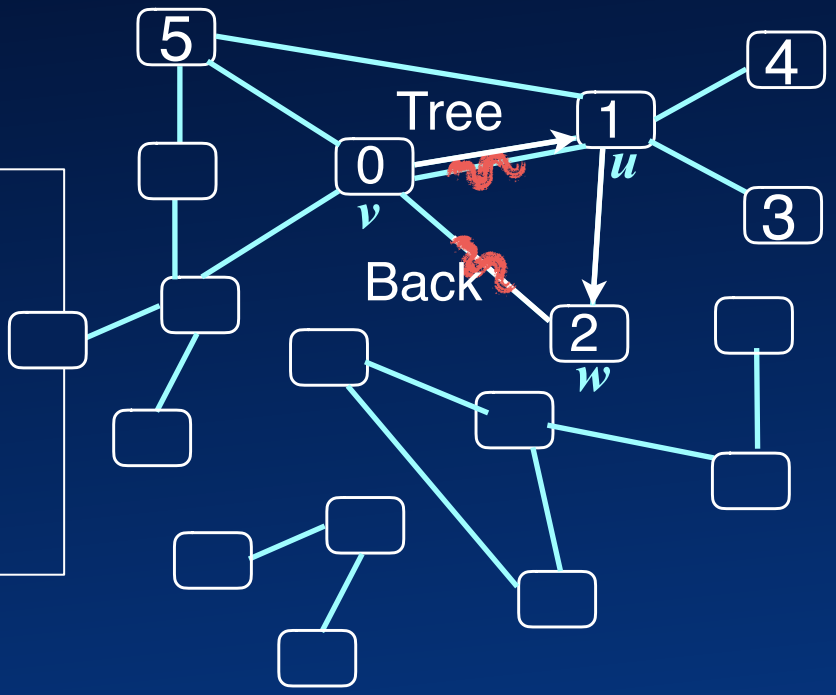
DFS Traversal

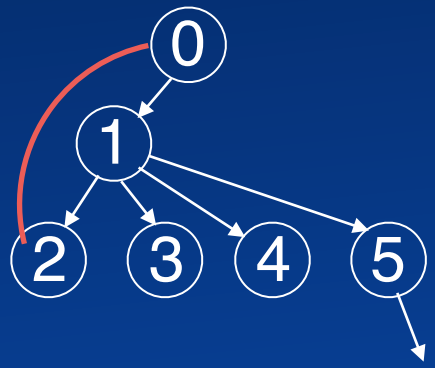
- **Start at any vertex**

```
traverse(v, parent):
    process(v) and mark(v)
    for u in v.adjacent() - parent
        visit(edge to u)
        traverse(u,v) if (! u.marked)
```

```
While ∃ unmarked v:
    traverse(v, null)
```

O(*m*+*n*)





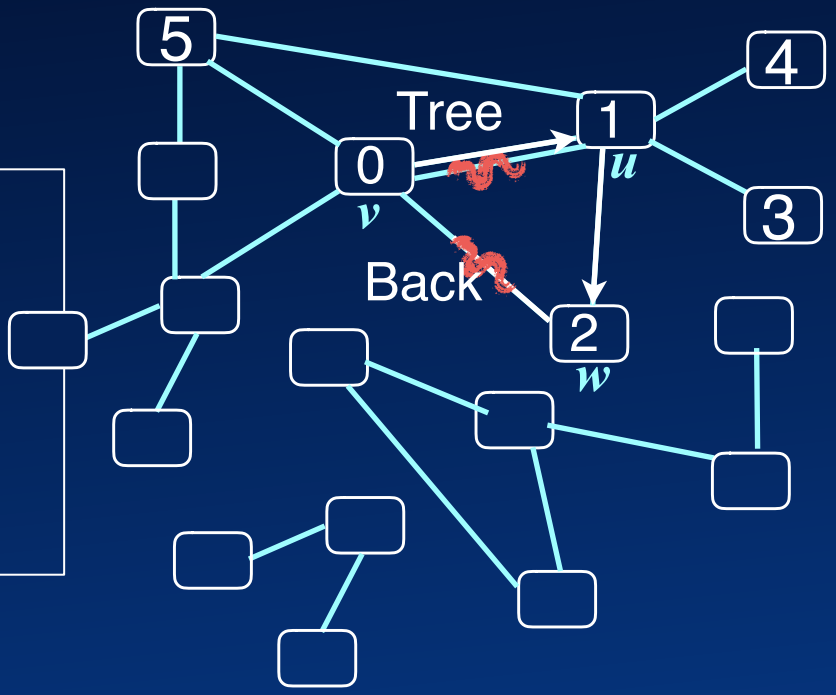
DFS Traversal

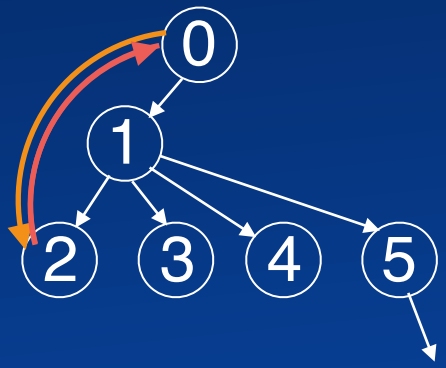
- **Start at any vertex**

```
traverse(v, parent):
    process(v) and mark(v)
    for u in v.adjacent() - parent
        visit(edge to u)
        traverse(u,v) if (! u.marked)
```

```
While ∃ unmarked v:
    traverse(v, null)
```

O(*m*+*n*)





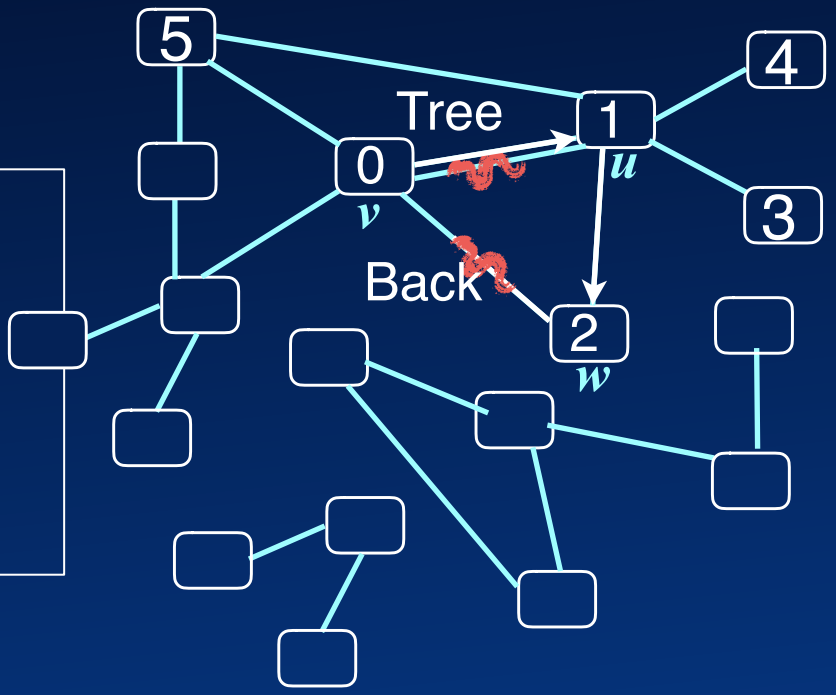
DFS Traversal

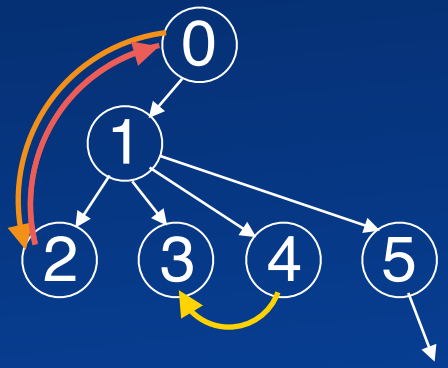
- **Start at any vertex**

```
traverse(v, parent):
    process(v) and mark(v)
    for u in v.adjacent() - parent
        visit(edge to u)
        traverse(u,v) if (! u.marked)
```

```
While ∃ unmarked v:
    traverse(v, null)
```

O(*m*+*n*)





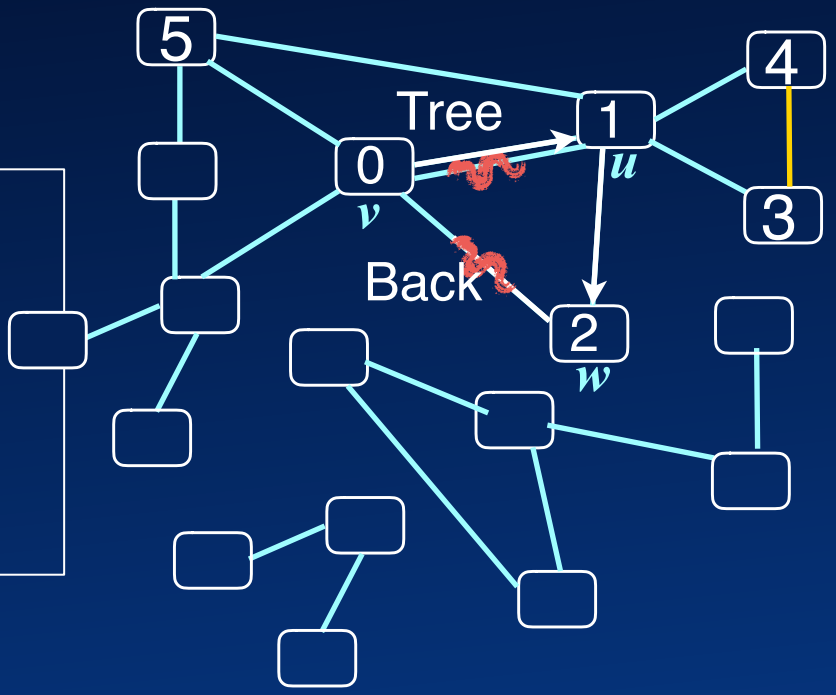
DFS Traversal

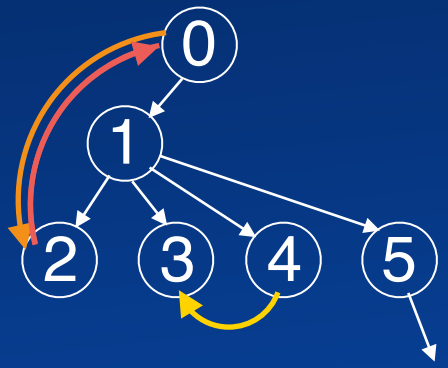
- **Start at any vertex**

```
traverse(v, parent):
    process(v) and mark(v)
    for u in v.adjacent() - parent
        visit(edge to u)
        traverse(u,v) if (! u.marked)
```

```
While ∃ unmarked v:
    traverse(v, null)
```

O(*m*+*n*)





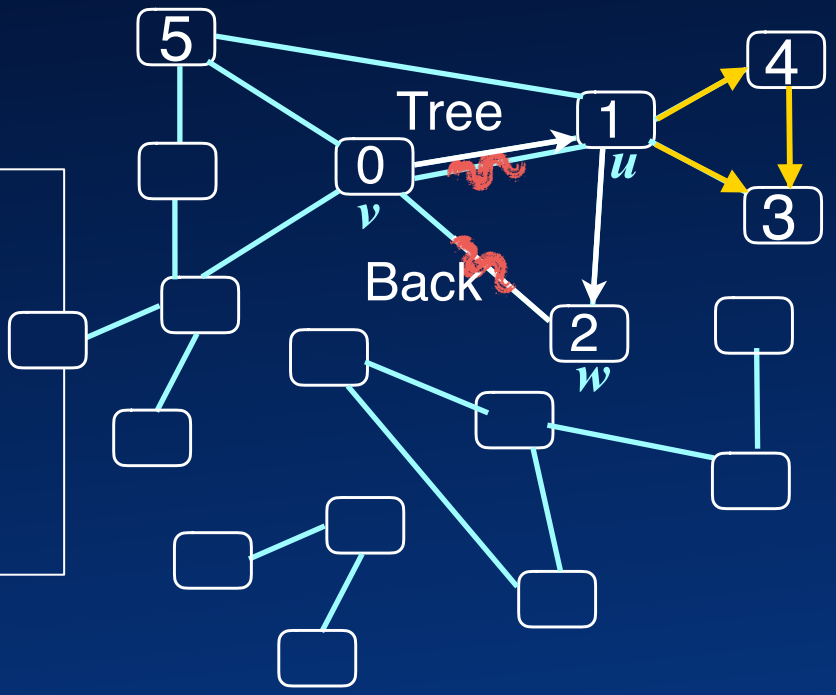
DFS Traversal

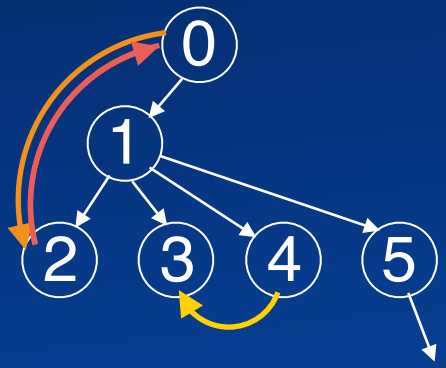
- **Start at any vertex**

```
traverse(v, parent):
    process(v) and mark(v)
    for u in v.adjacent() - parent
        visit(edge to u)
        traverse(u,v) if (! u.marked)
```

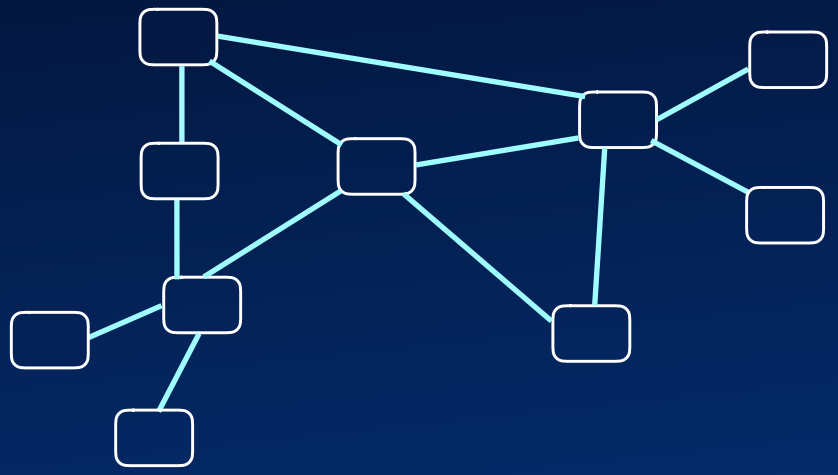
```
While ∃ unmarked v:
    traverse(v, null)
```

O(*m*+*n*)





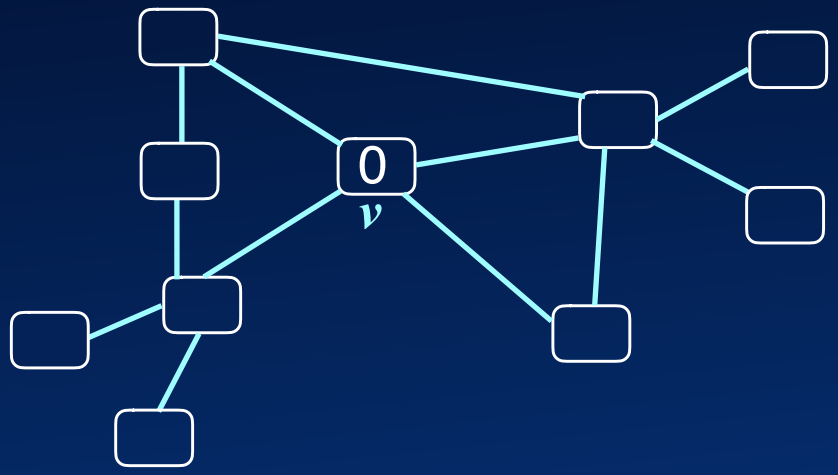
BFS Traversal



```
bfs():
   while(Q.hasNext()):
       v = Q.next();
       process(v);
       for u in v.adjacent
          if(! u.marked):
              mark u and enqueue(u)
```

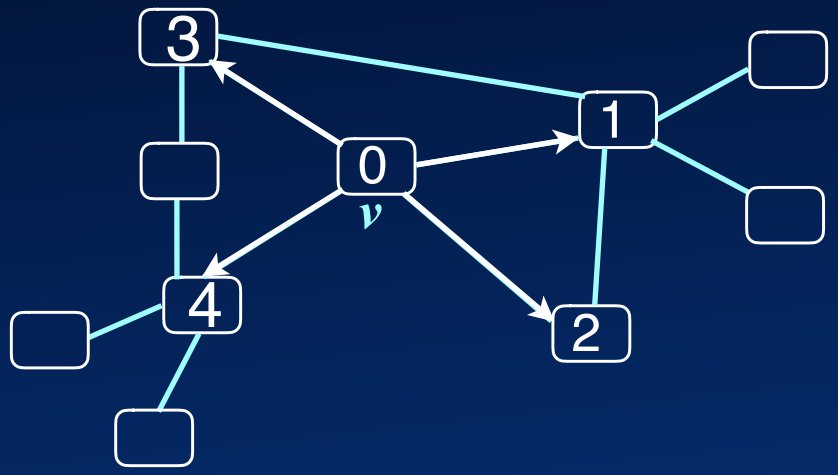
BFS Traversal

```
bfs():
   while(Q.hasNext()):
       v = Q.next();
       process(v);
       for u in v.adjacent
          if(! u.marked):
              mark u and enqueue(u)
```



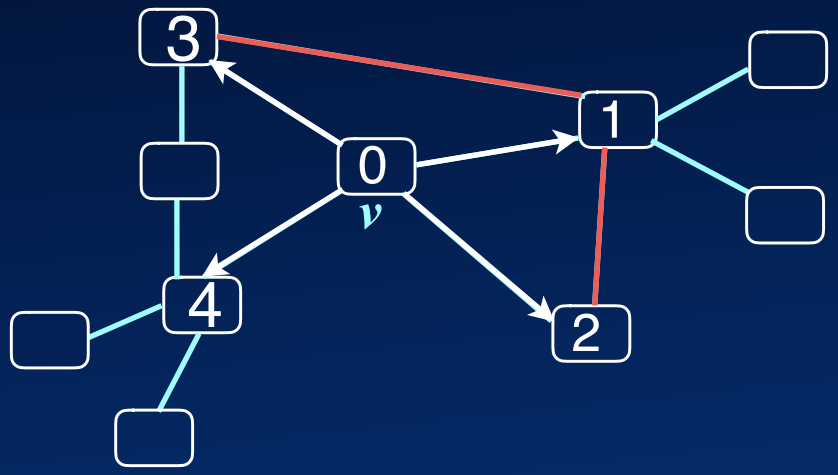
BFS Traversal

```
bfs():
   while(Q.hasNext()):
       v = Q.next();
       process(v);
       for u in v.adjacent
          if(! u.marked):
              mark u and enqueue(u)
```



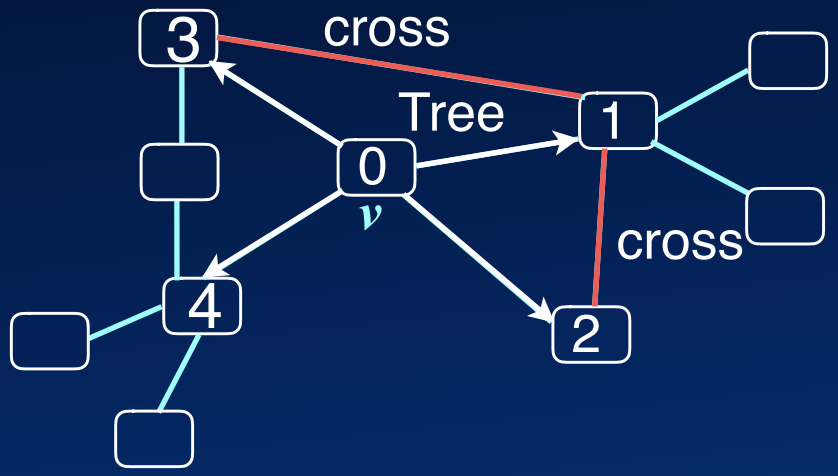
BFS Traversal

```
bfs():
   while(Q.hasNext()):
      v = Q.next();
      process(v);
      for u in v.adjacent
         if(! u.marked):
            mark u and enqueue(u)
```



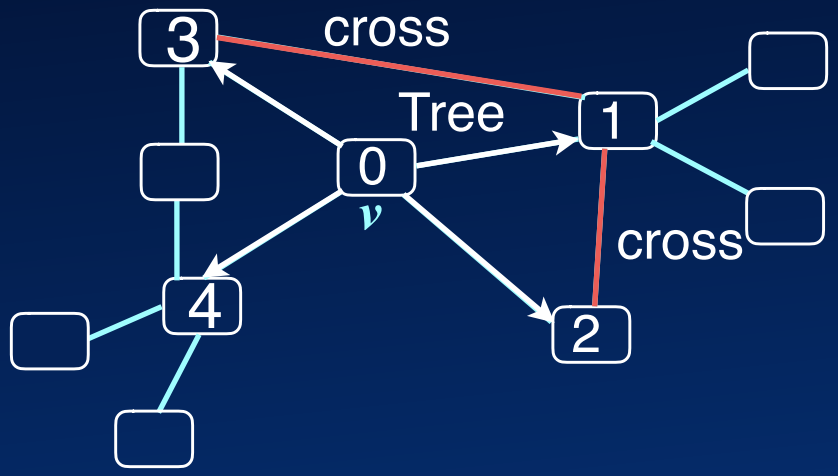
BFS Traversal

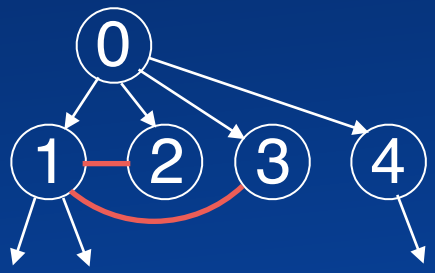
```
bfs():
   while(Q.hasNext()):
       v = Q.next();
       process(v);
       for u in v.adjacent
          if(! u.marked):
              mark u and enqueue(u)
```



BFS Traversal

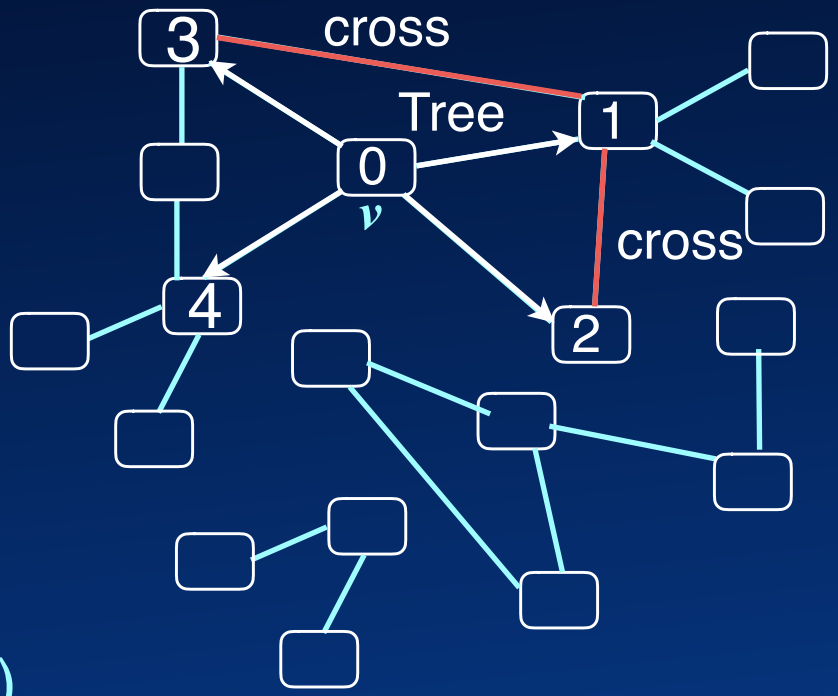
```
bfs():
   while(Q.hasNext()):
      v = Q.next();
      process(v);
      for u in v.adjacent
         if(! u.marked):
            mark u and enqueue(u)
```

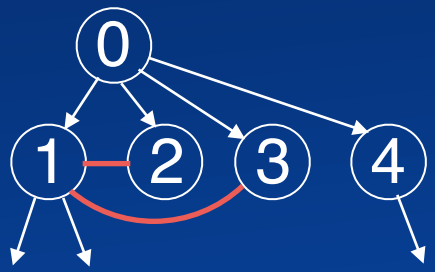




BFS Traversal

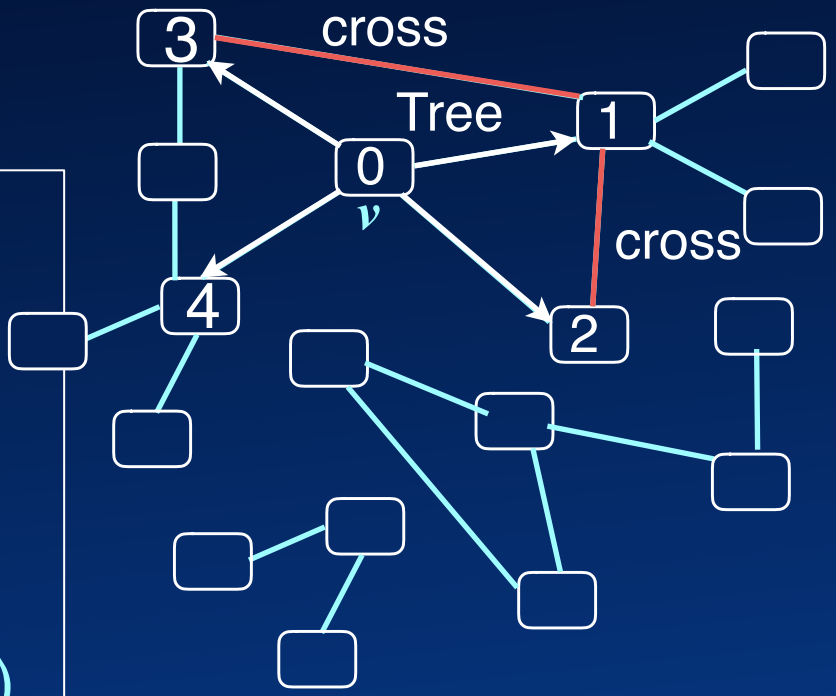
```
bfs():
   while(Q.hasNext()):
      v = Q.next();
      process(v);
      for u in v.adjacent
         if(! u.marked):
            mark u and enqueue(u)
```

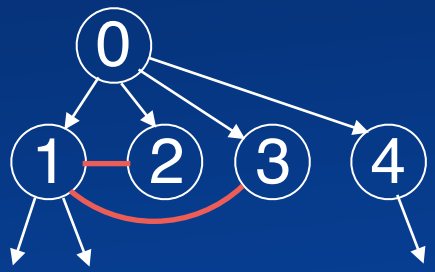




BFS Traversal

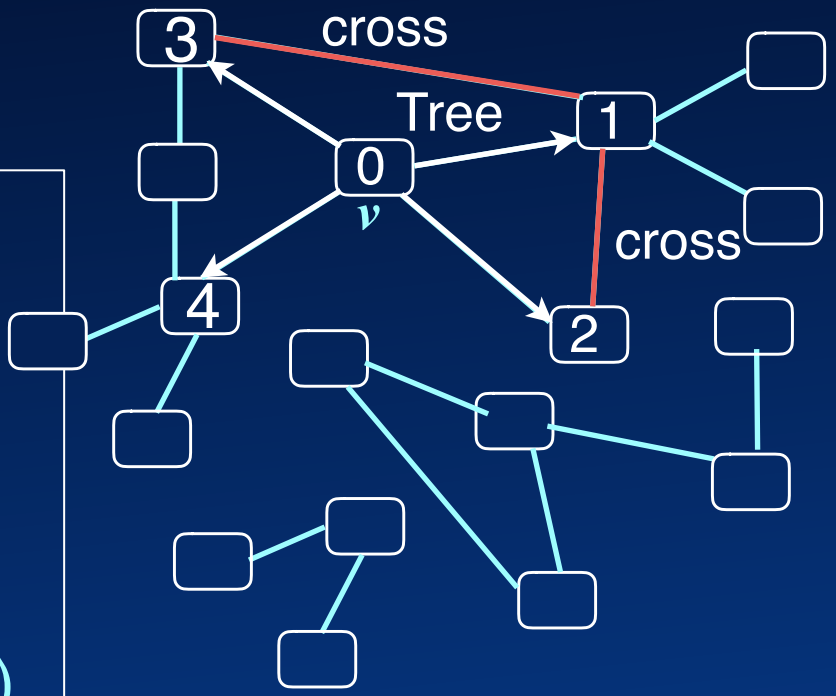
```
bfs():
   while(Q.hasNext()):
       v = Q.next();
       process(v);
       for u in v.adjacent
          if(! u.marked):
               mark u and enqueue(u)
```



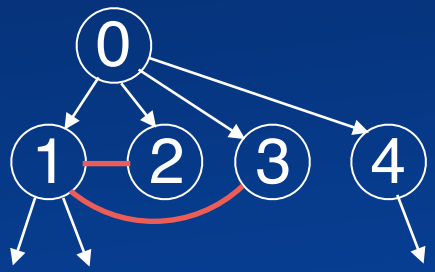


BFS Traversal

```
bfs():
   while(Q.hasNext()):
       v = Q.next();
       process(v);
       for u in v.adjacent
          if(! u.marked):
               mark u and enqueue(u)
```

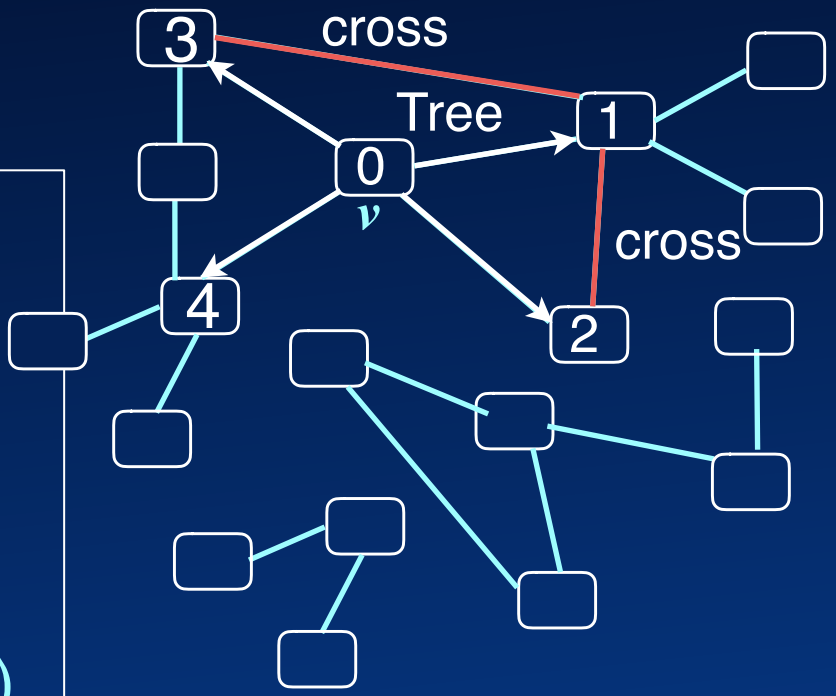



```
While ∃ unmarked v:
   enqueue(v)
   bfs()
```



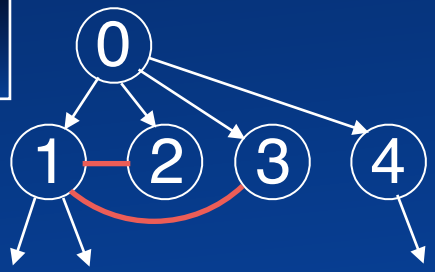
BFS Traversal

```
bfs():
   while(Q.hasNext()):
      v = Q.next();
      process(v);
      for u in v.adjacent
         if(! u.marked):
            mark u and enqueue(u)
```




```
While ∃ unmarked v:
   enqueue(v)
   bfs()
```

O(*m*+*n*)



T/F

- Time complexity of DFS in a graph with n nodes and m edges is O(m+n)
- Time complexity of BFS in a graph with n nodes and m edges is O(n^2)

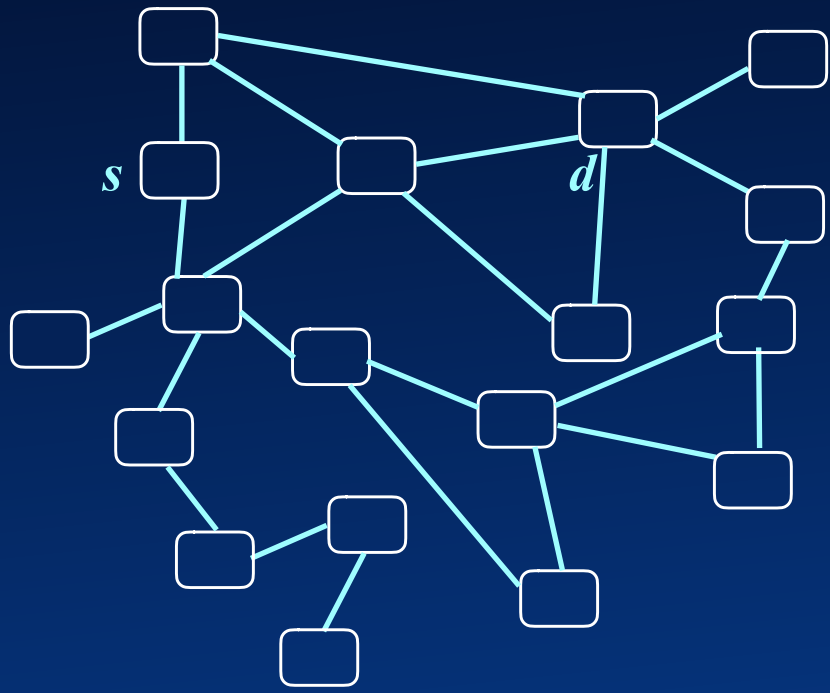
Mail: col106quiz@cse.iitd.ac.in
Format: t,t

Mail: col106quiz@cse.iitd.ac.in
Format: t,t,t

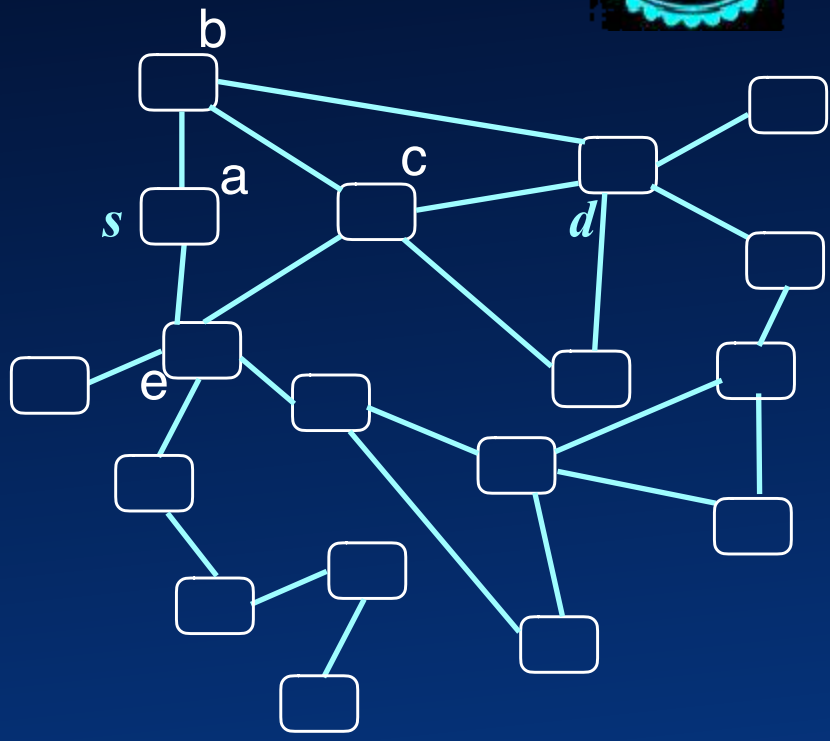
T/F

- The shortest path from node u to node v in a directed graph is equal to the shortest path from node v to node u.
- The shortest path from node u to node v in an undirected graph is equal to the shortest path from node v to node u.
- If P_1 is the shortest path from node u to node w in a directed graph, and P_2 is the shortest path from w to node v, P_1P_2 (i.e., P_1 followed by P_2) is the shortest path from u to v.

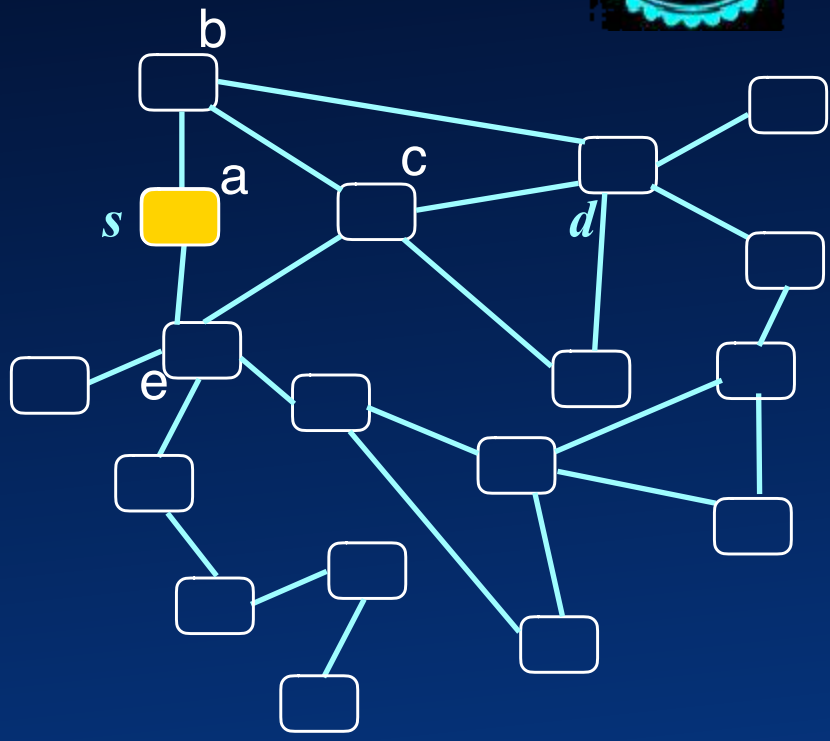
Shortest Path



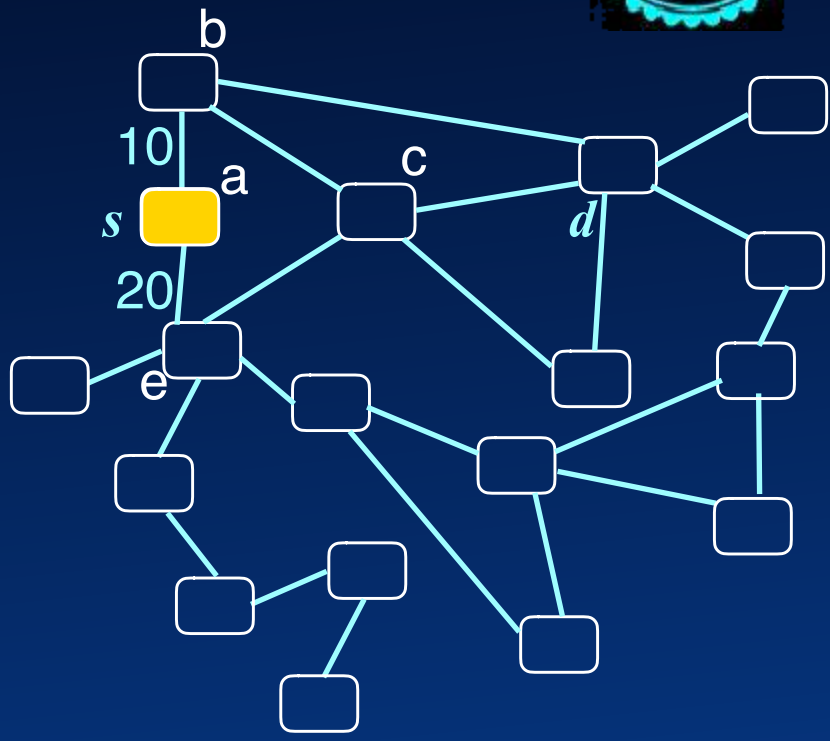
Shortest Path



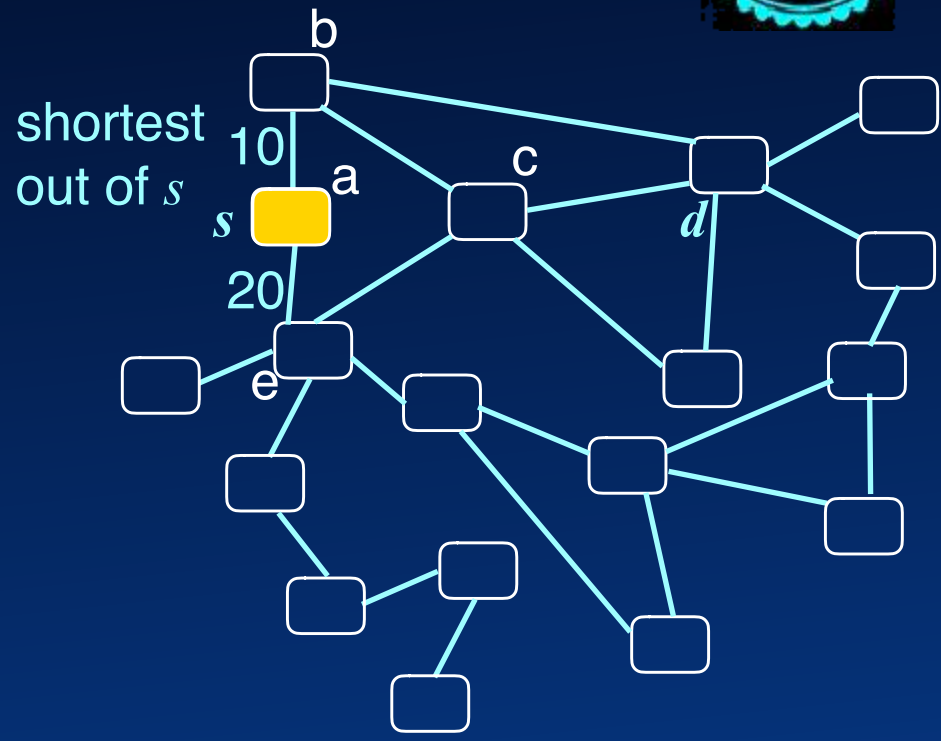
Shortest Path



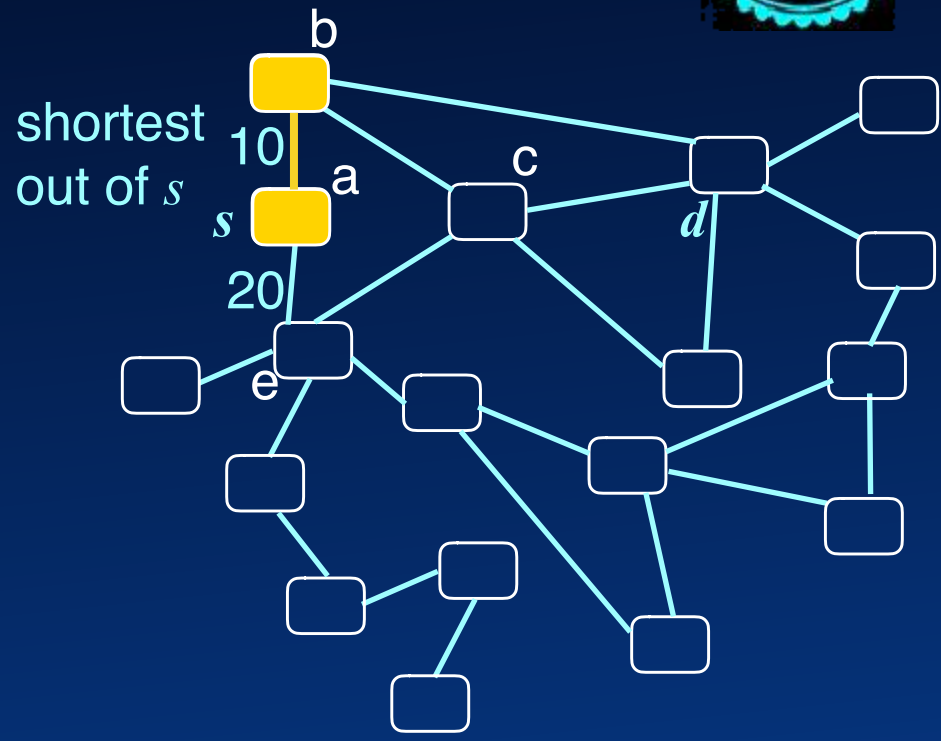
Shortest Path



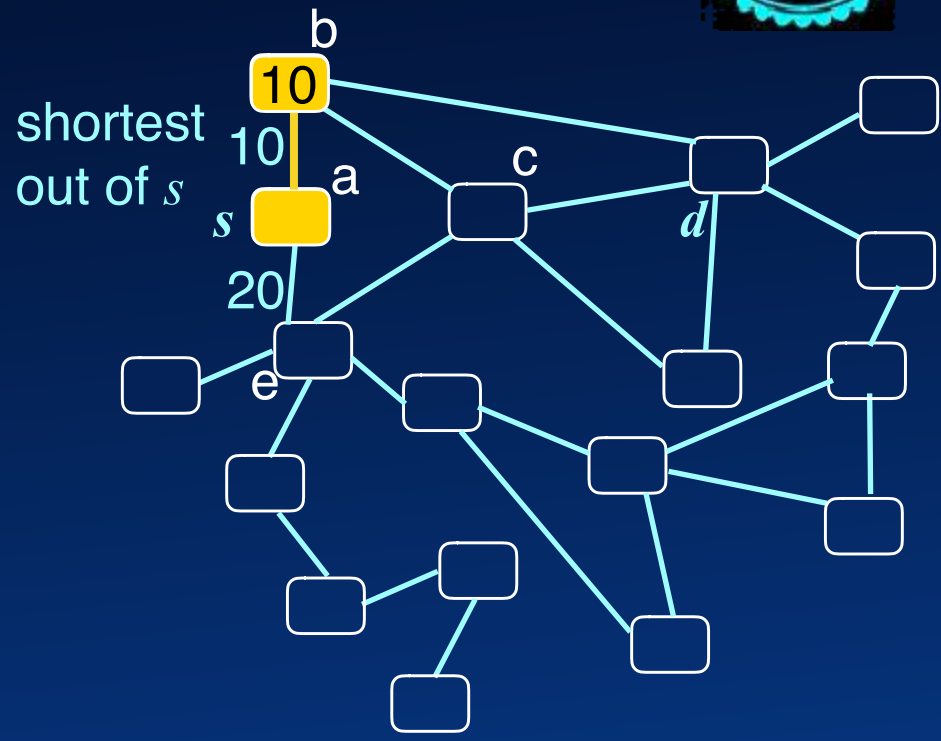
Shortest Path



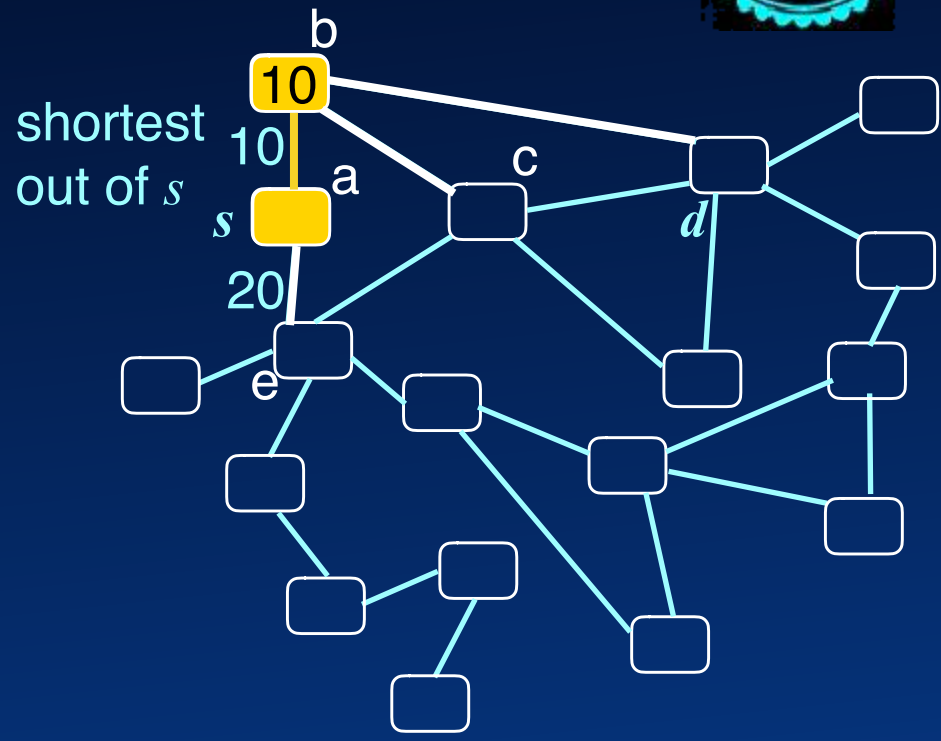
Shortest Path



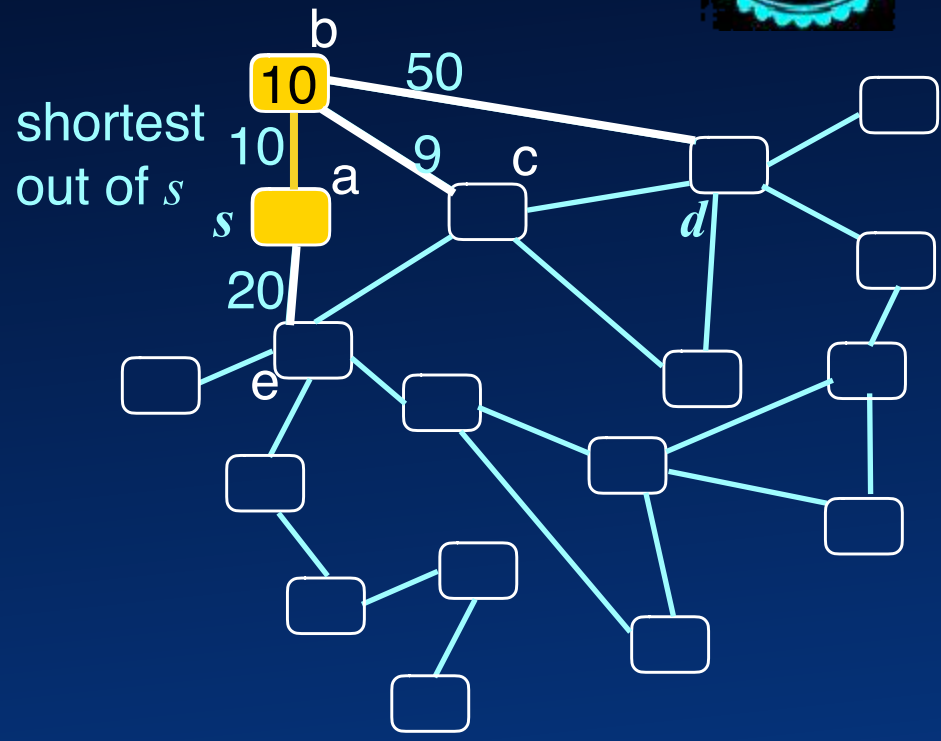
Shortest Path



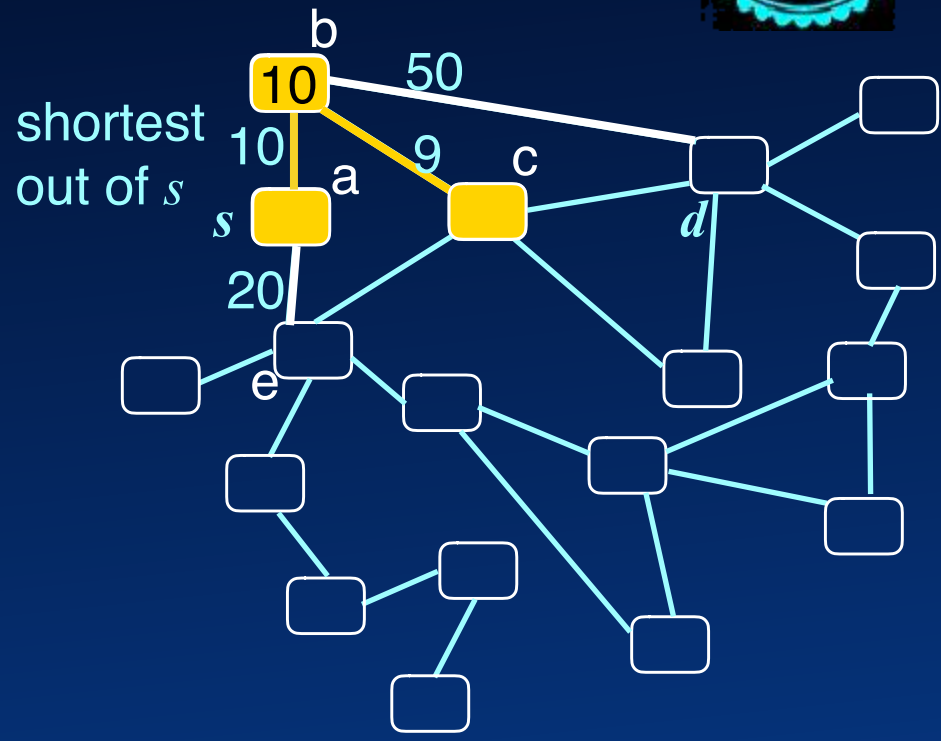
Shortest Path



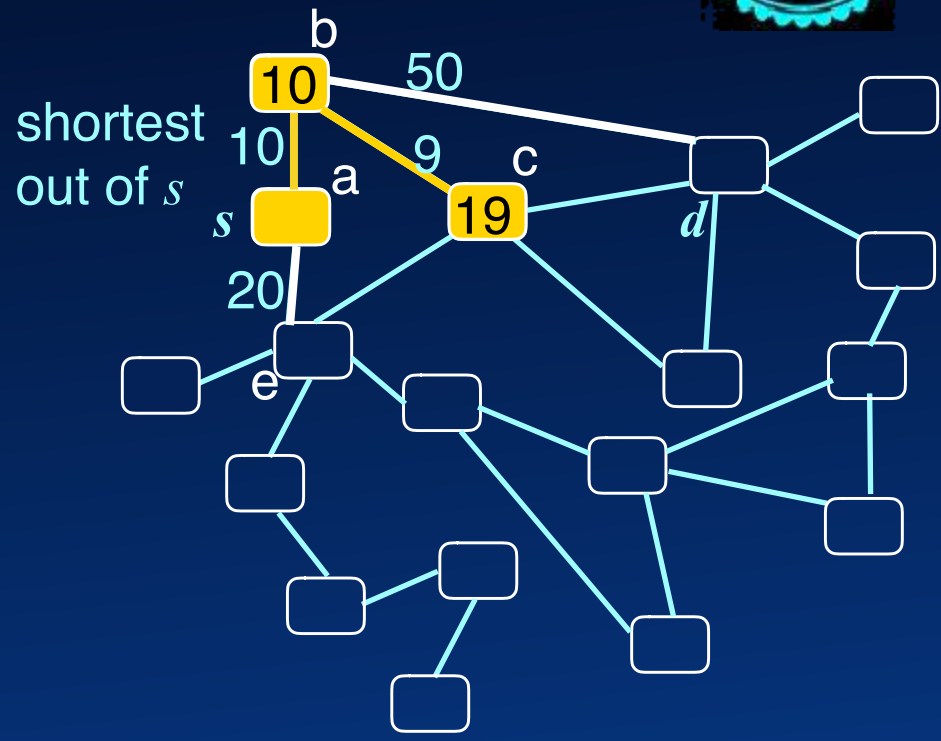
Shortest Path



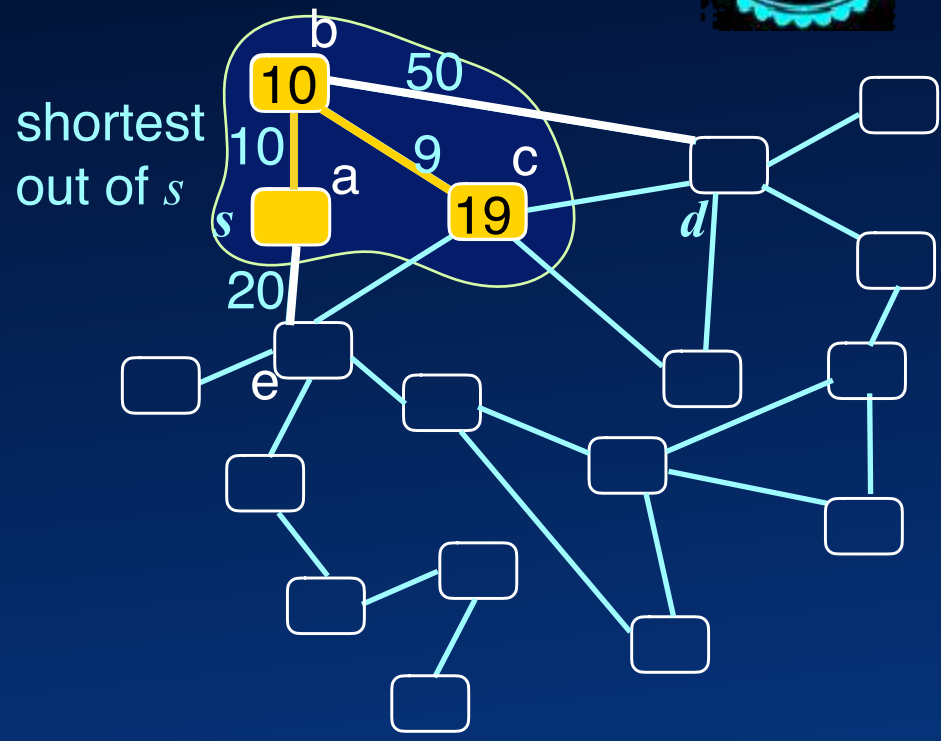
Shortest Path



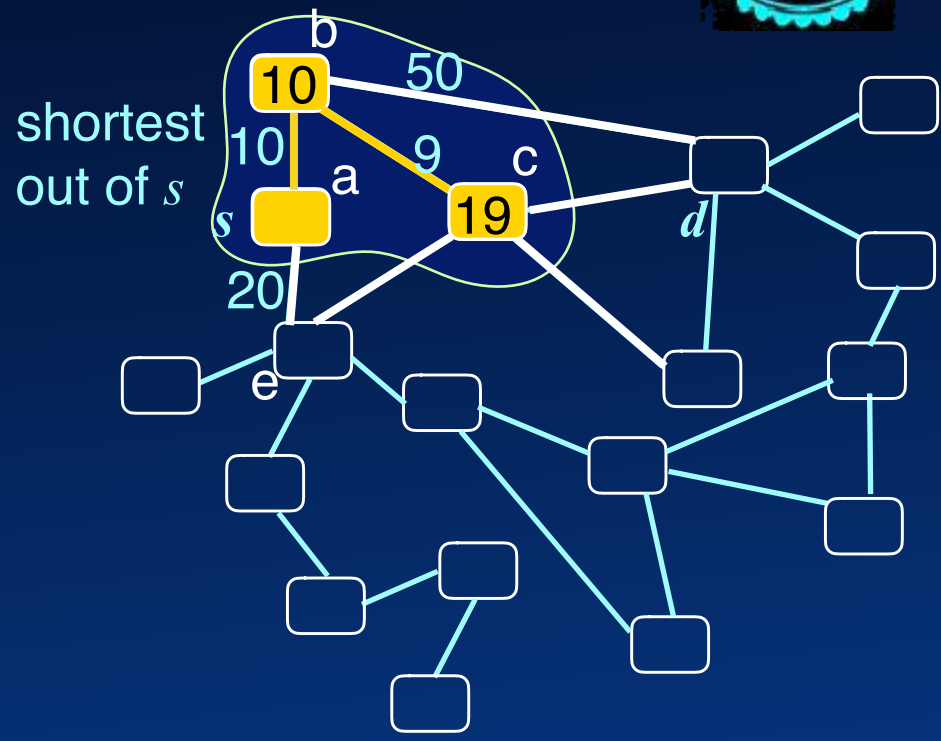
Shortest Path



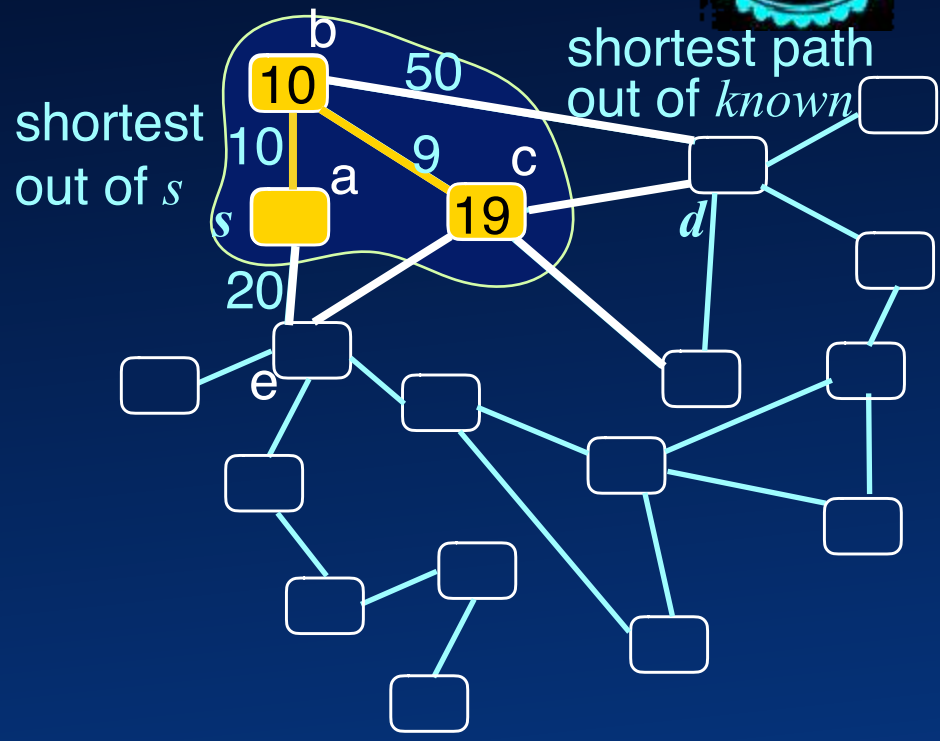
Shortest Path



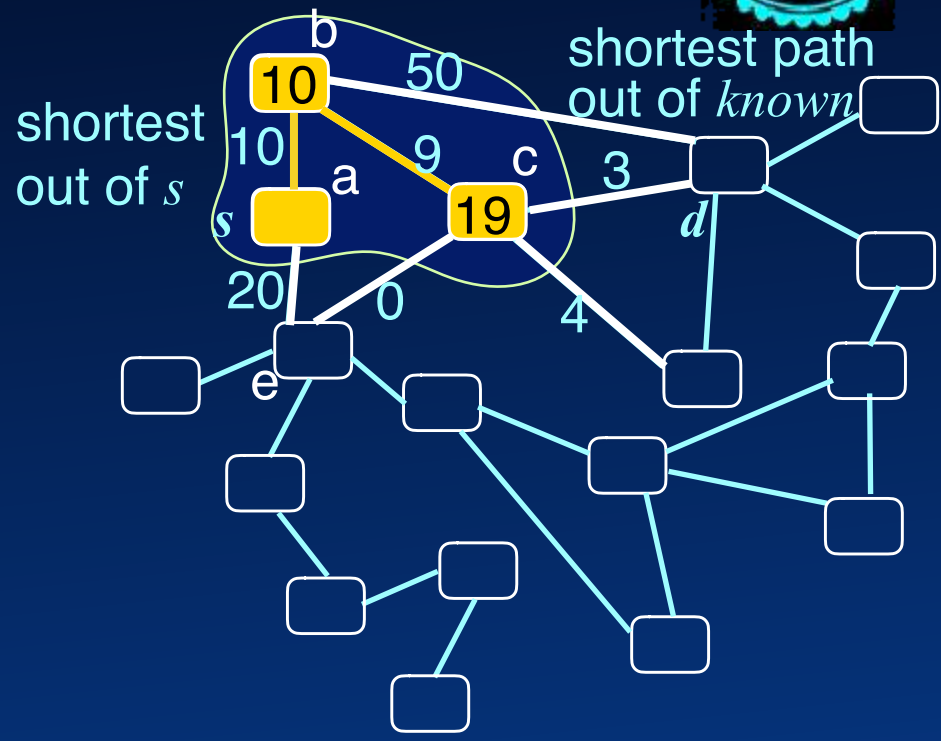
Shortest Path



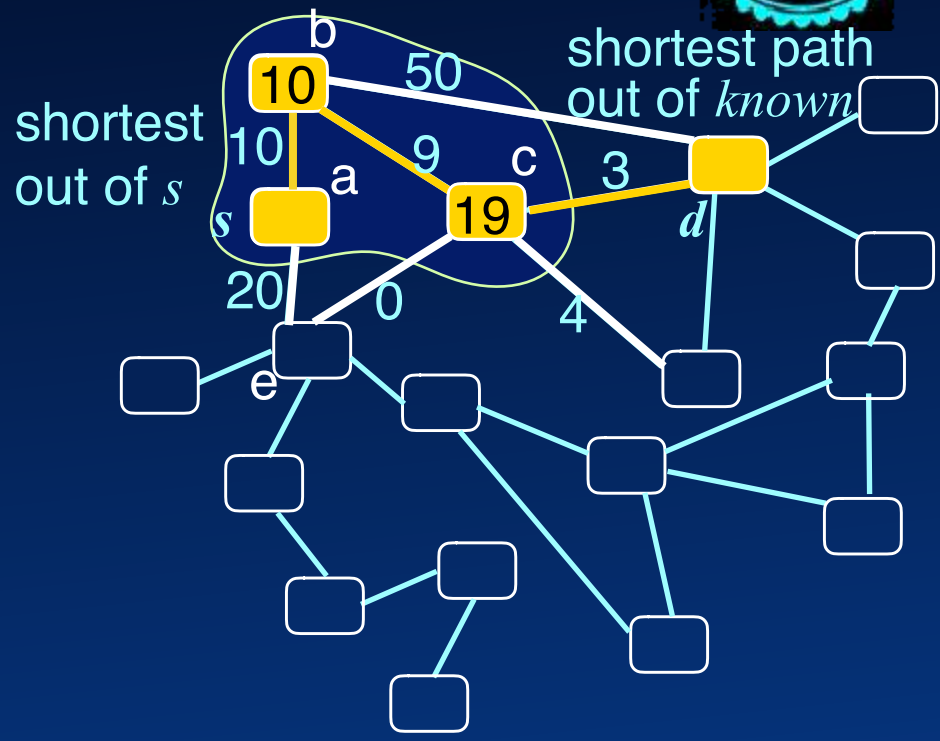
Shortest Path



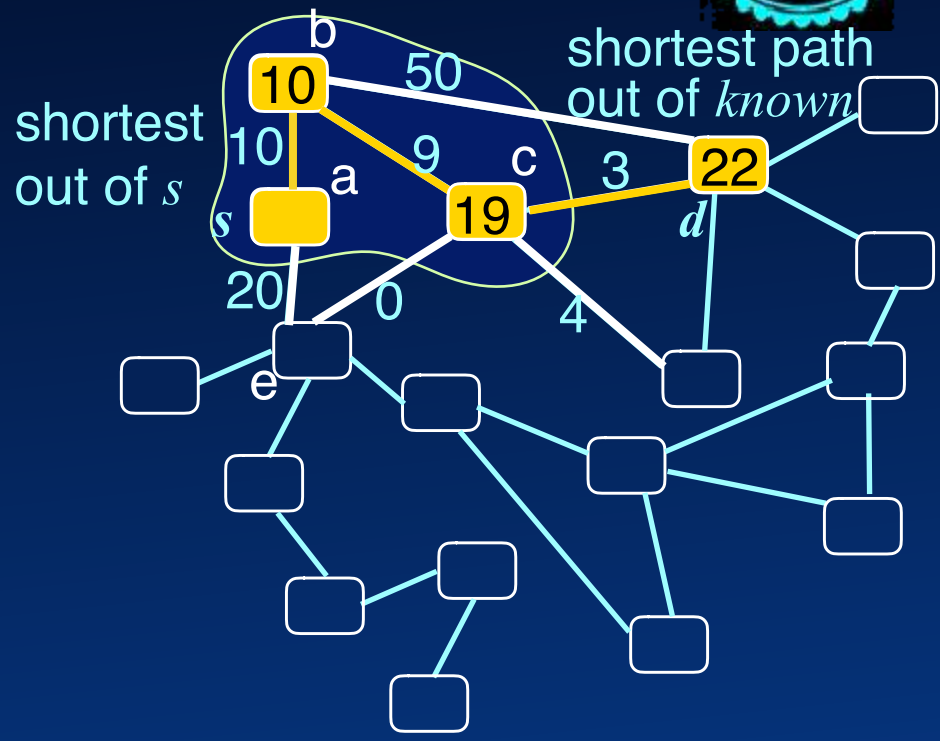
Shortest Path



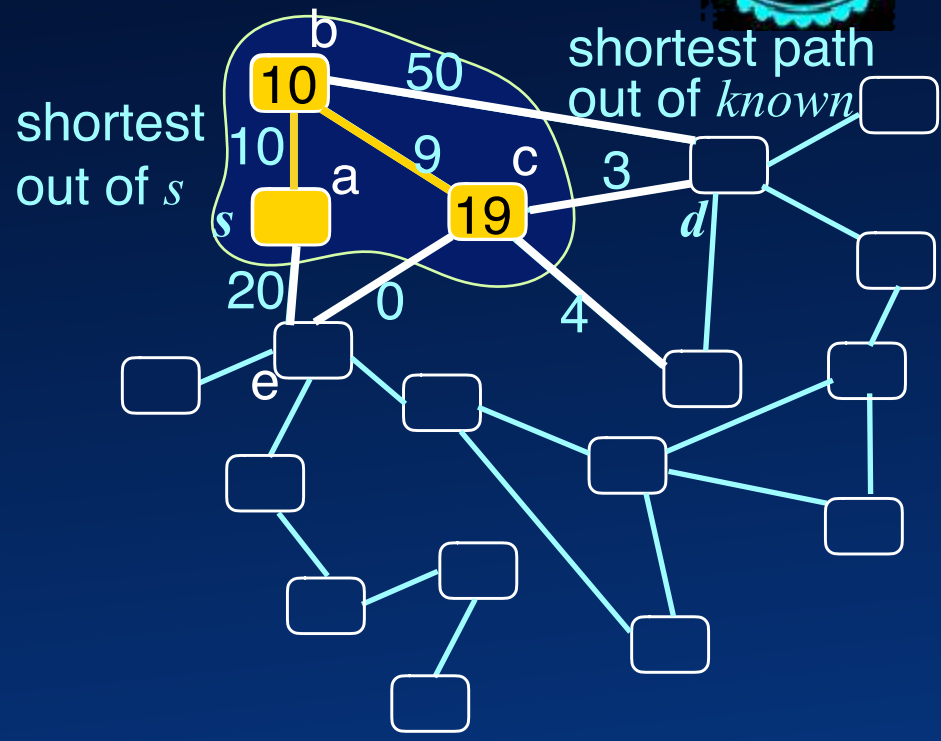
Shortest Path



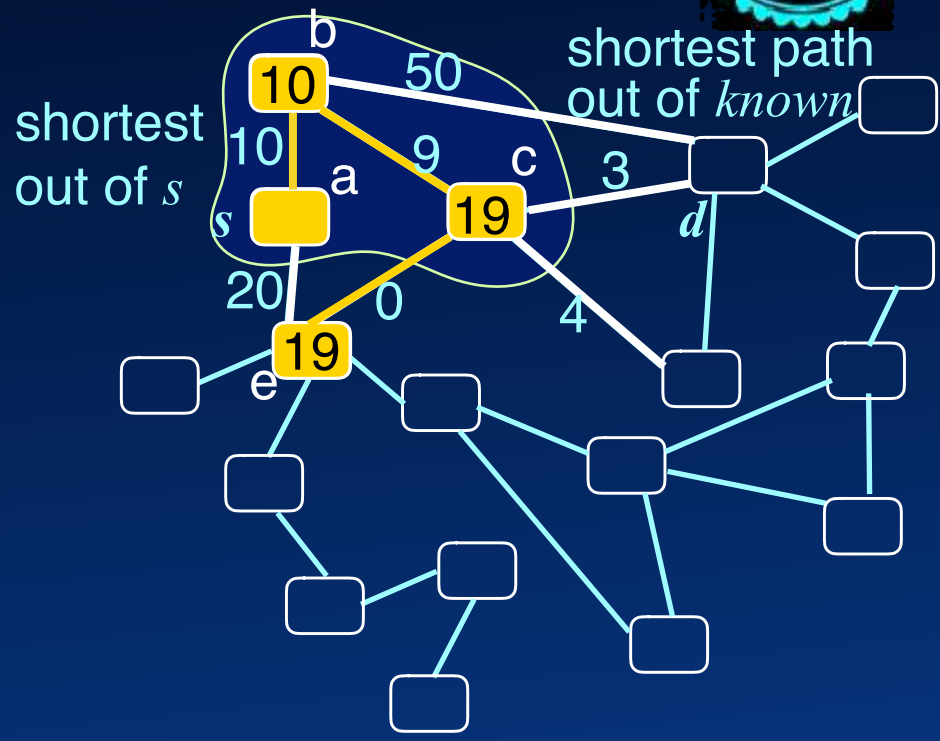
Shortest Path



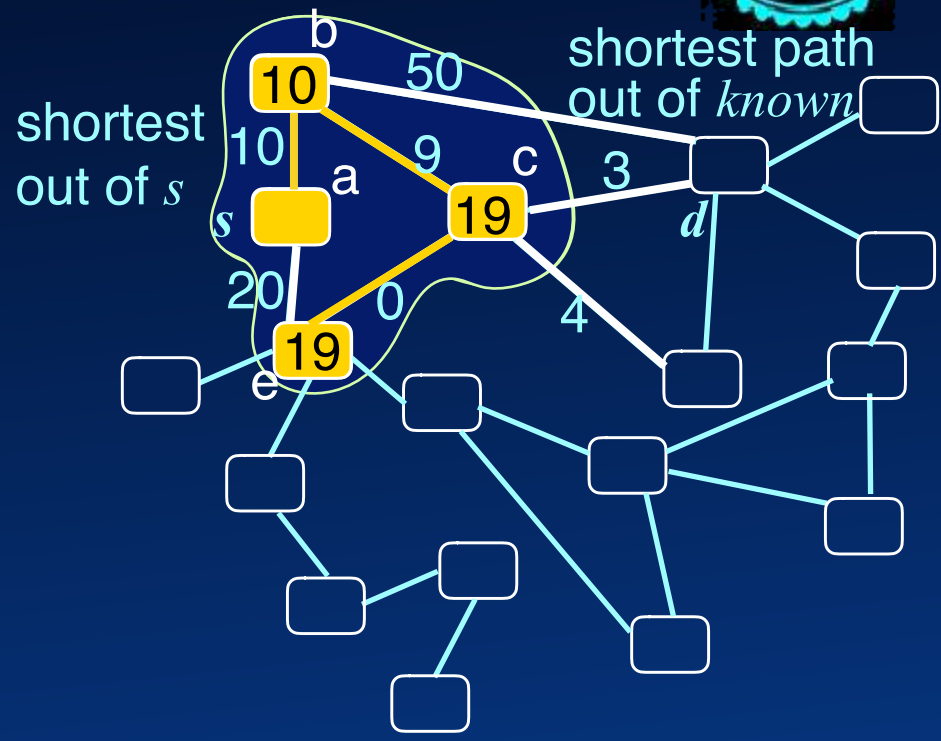
Shortest Path



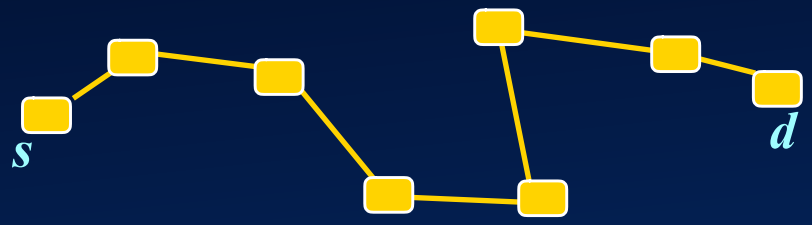
Shortest Path

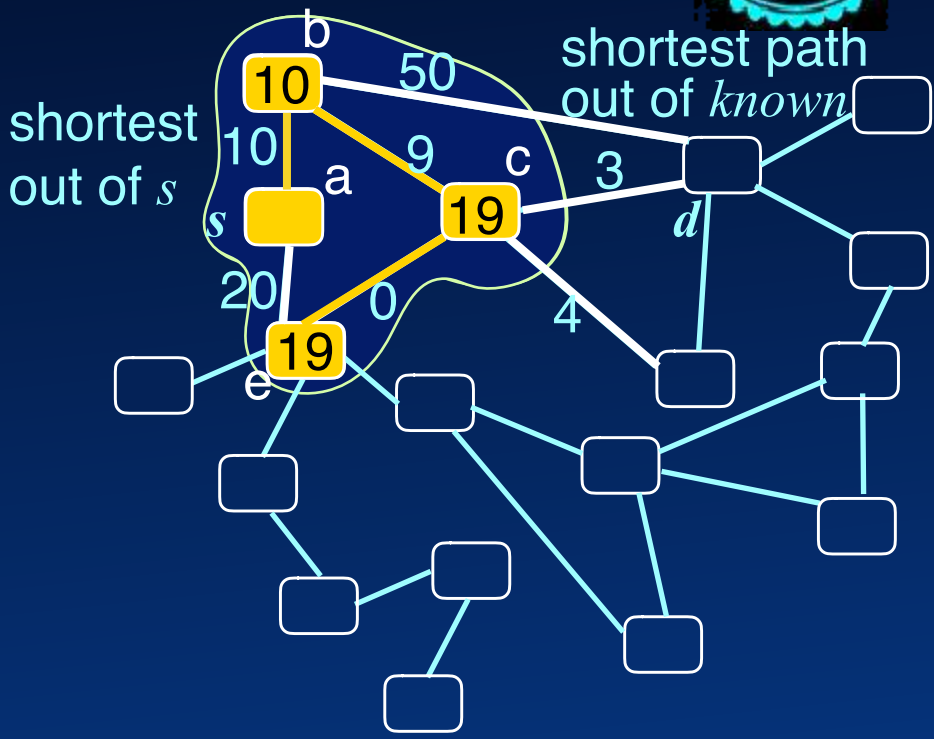


Shortest Path

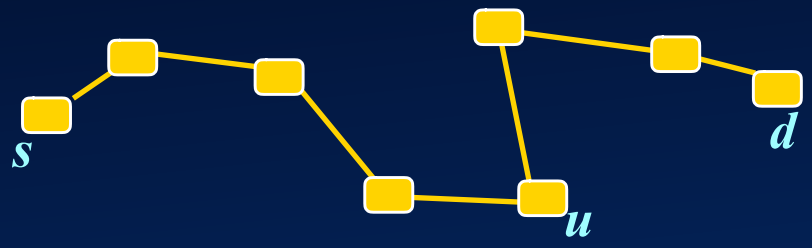


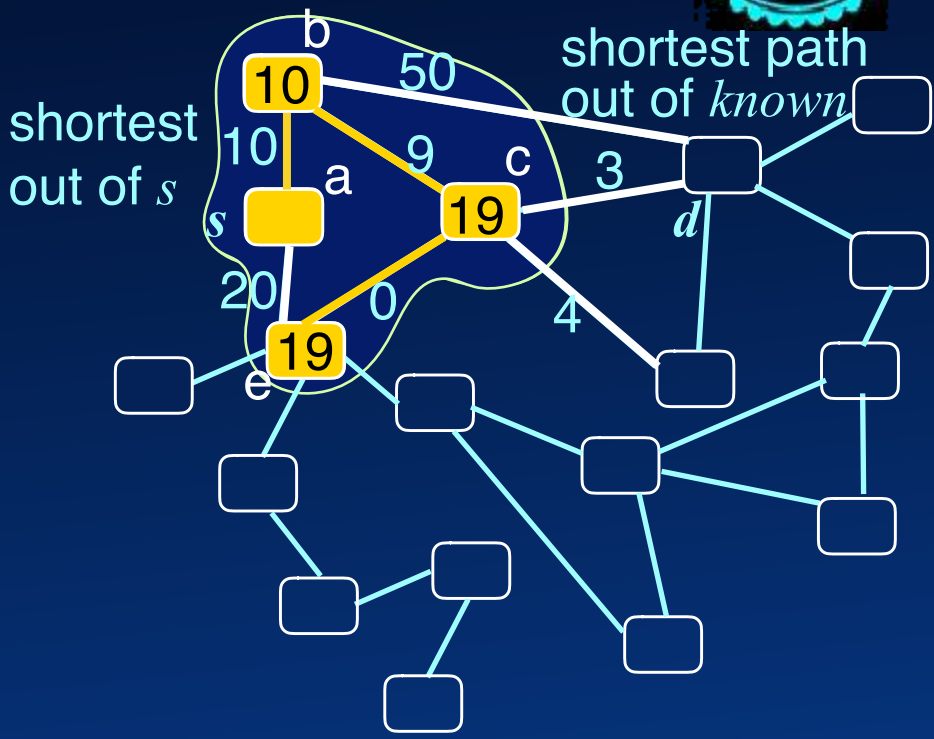
Shortest Path



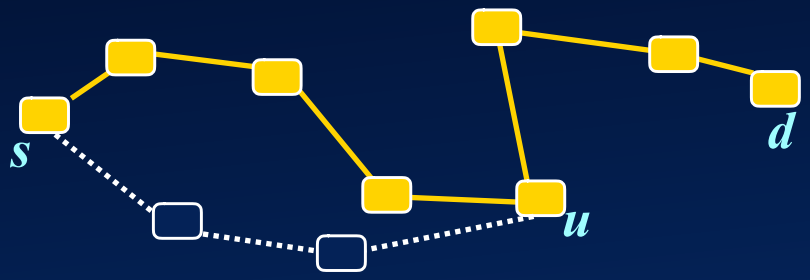


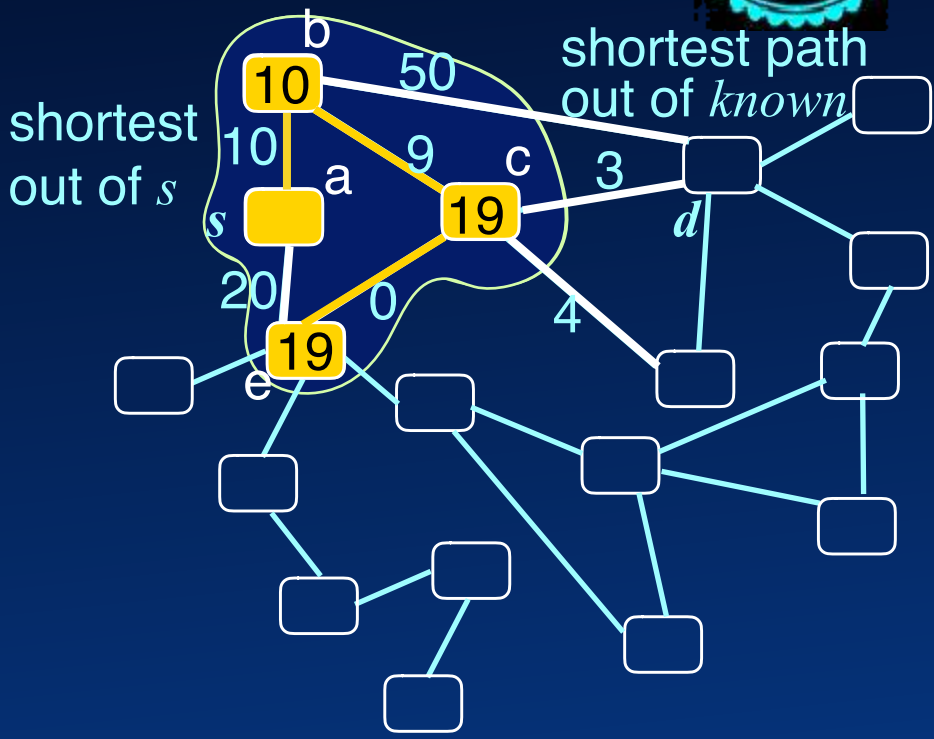
Shortest Path



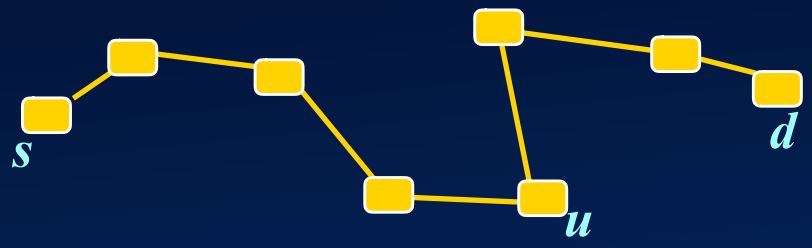


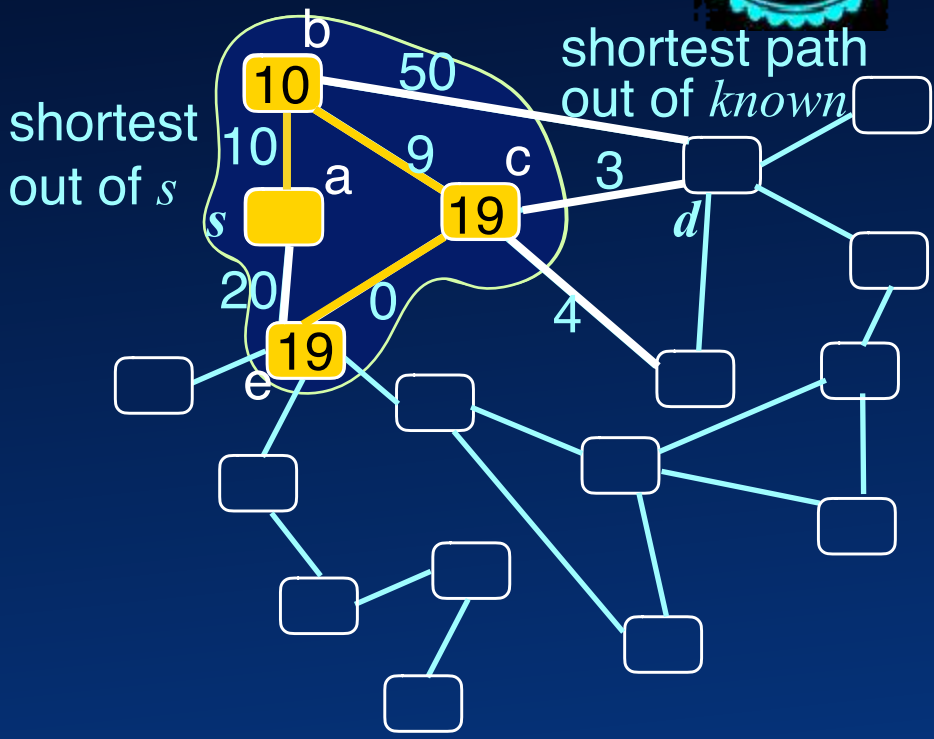
Shortest Path



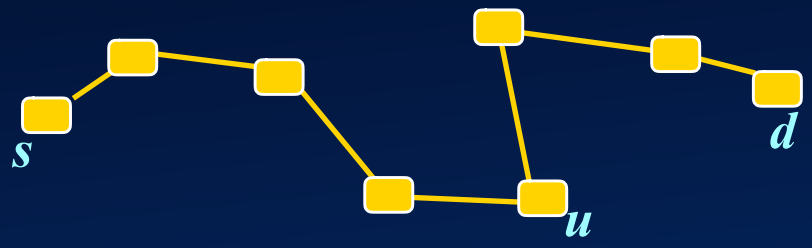


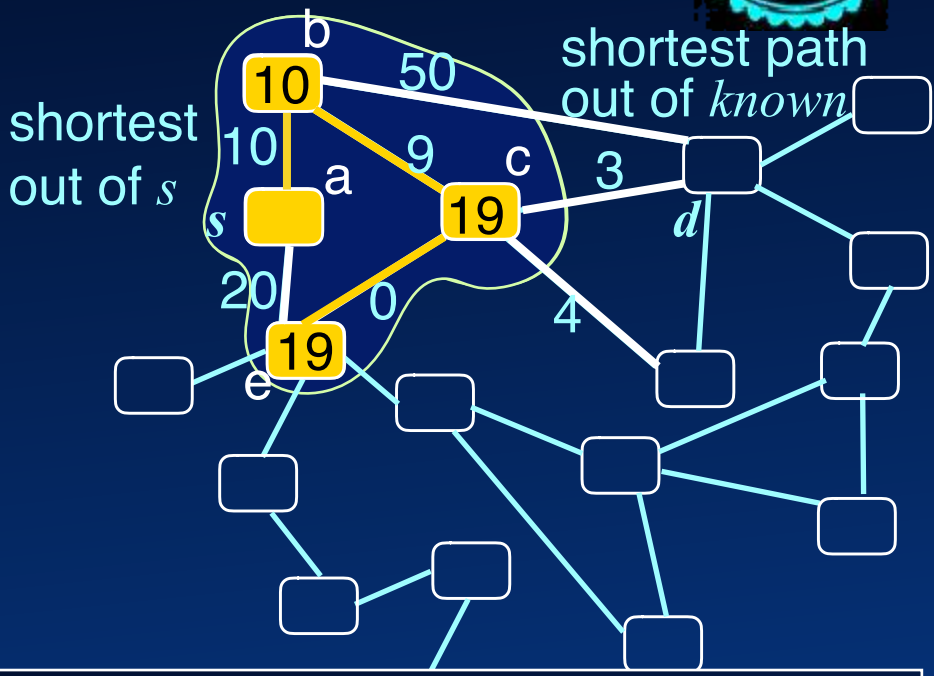
Shortest Path





Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

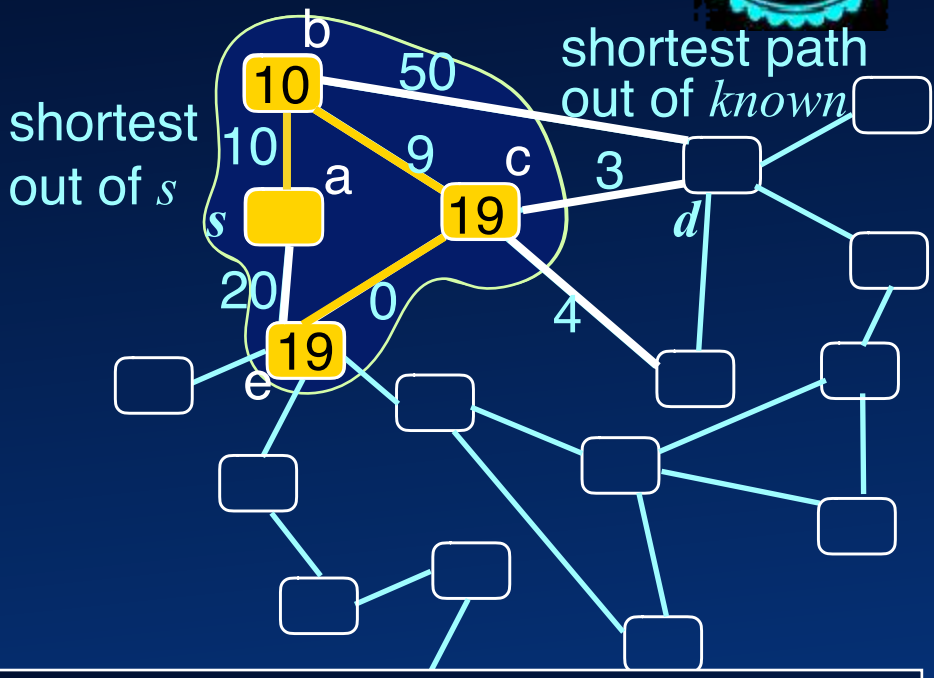
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

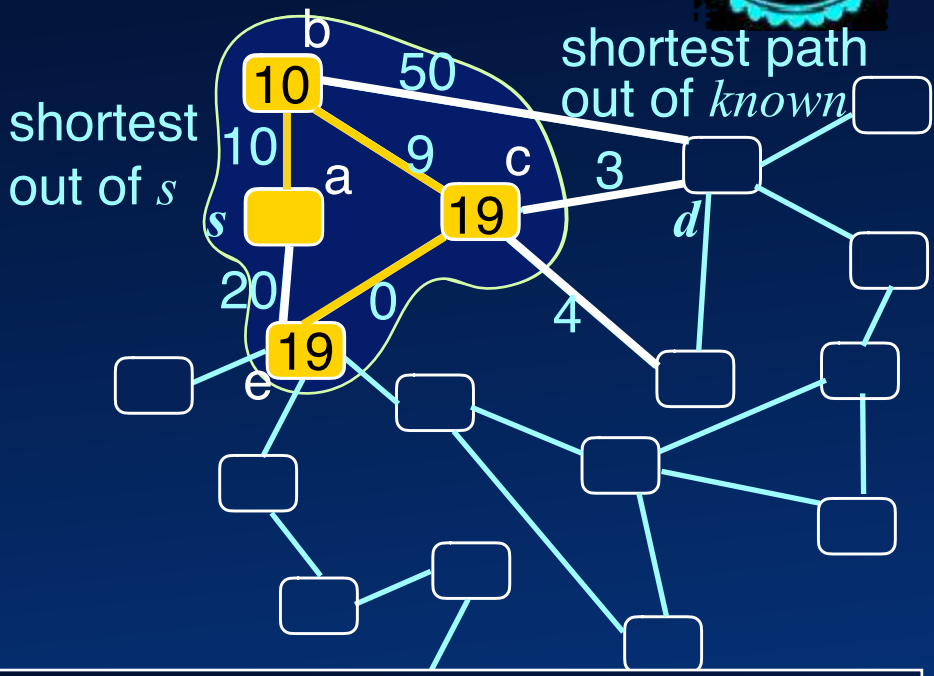
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

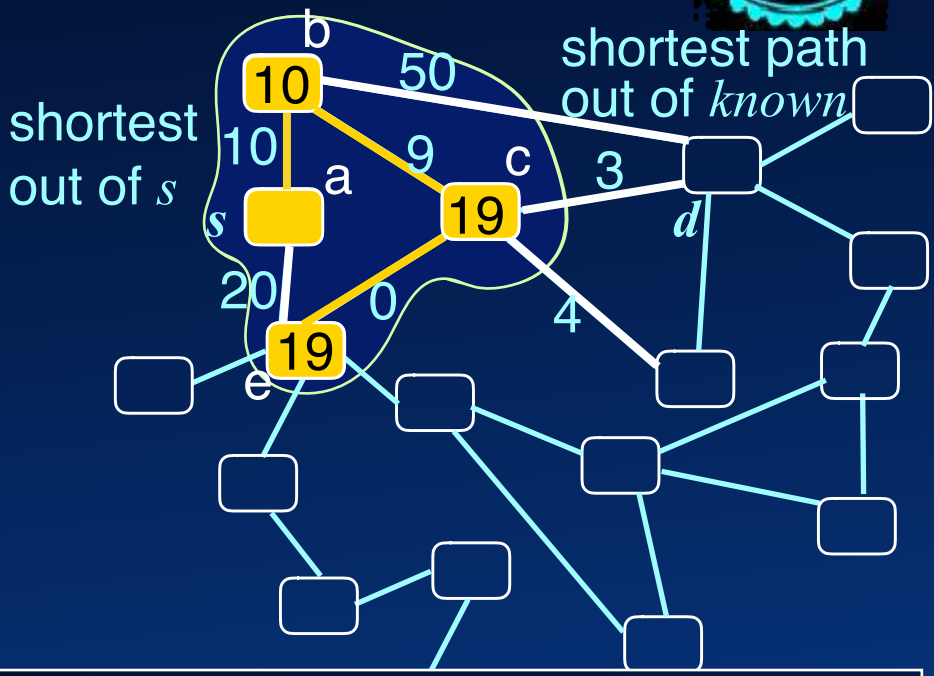
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

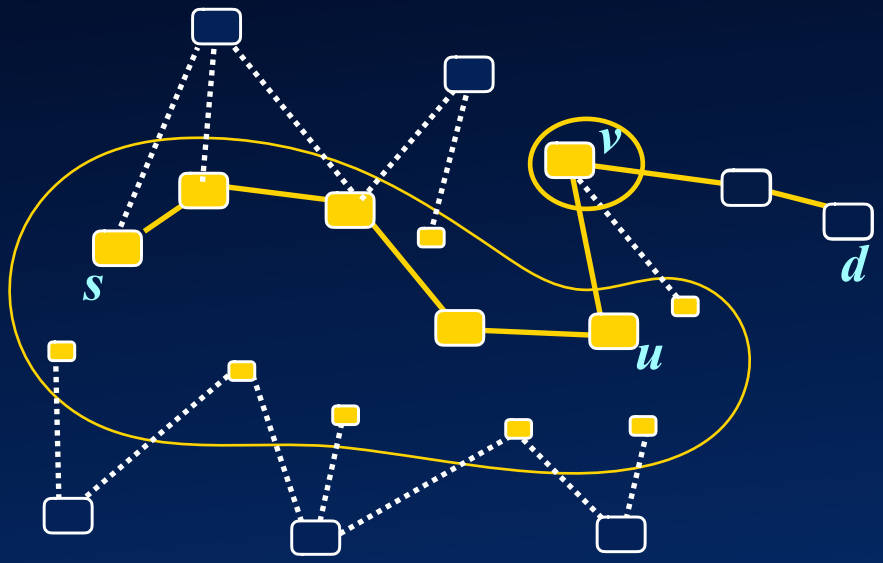
Induction step:

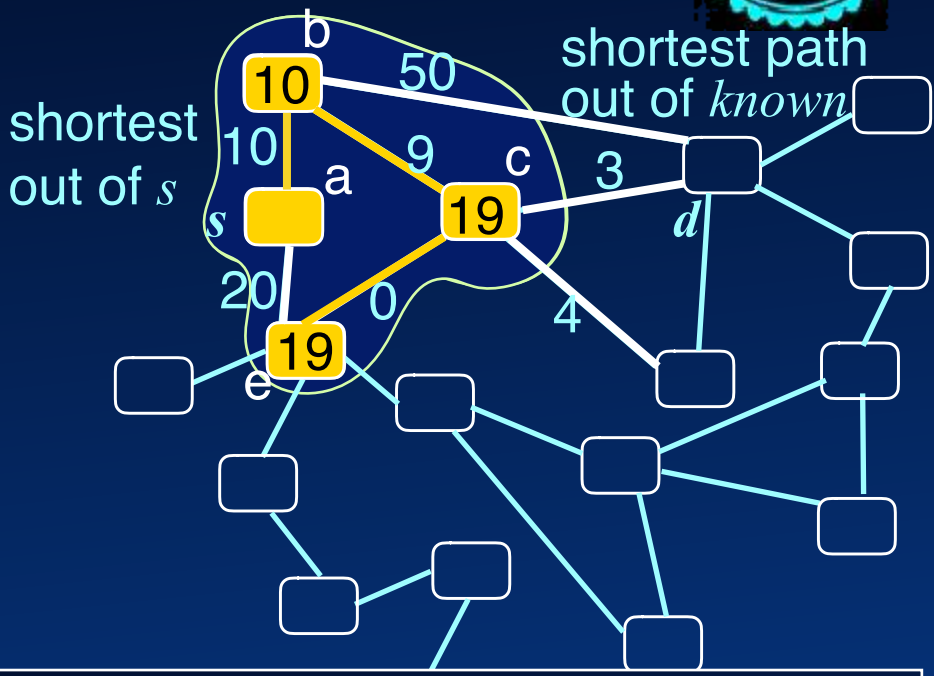
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

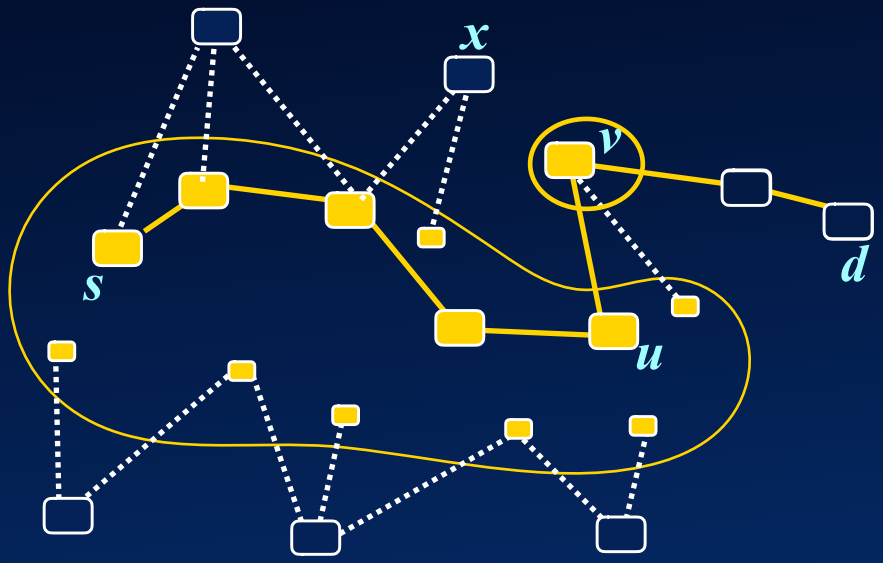
Induction step:

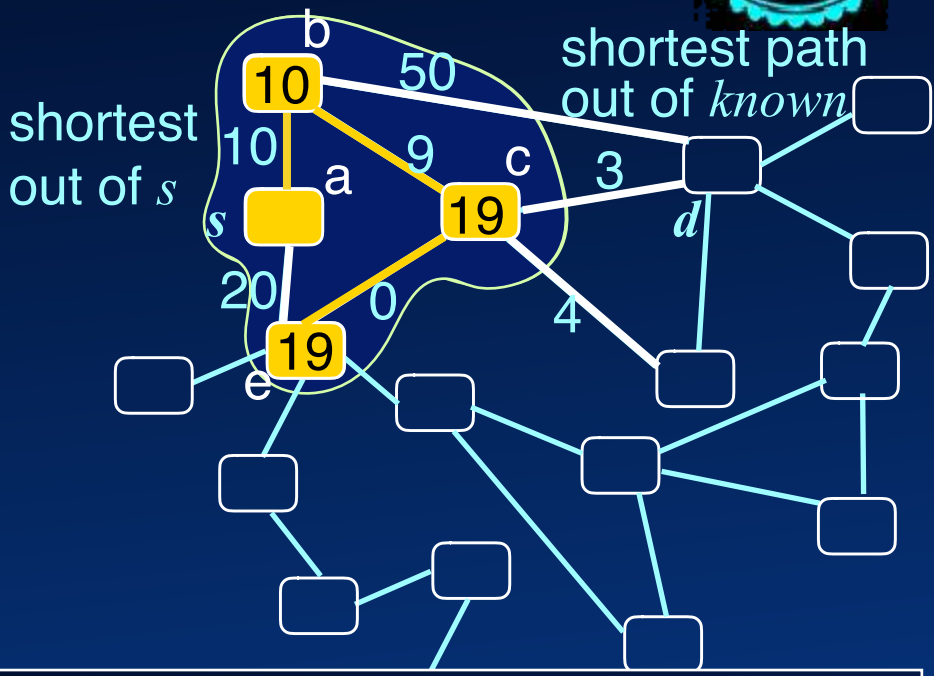
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

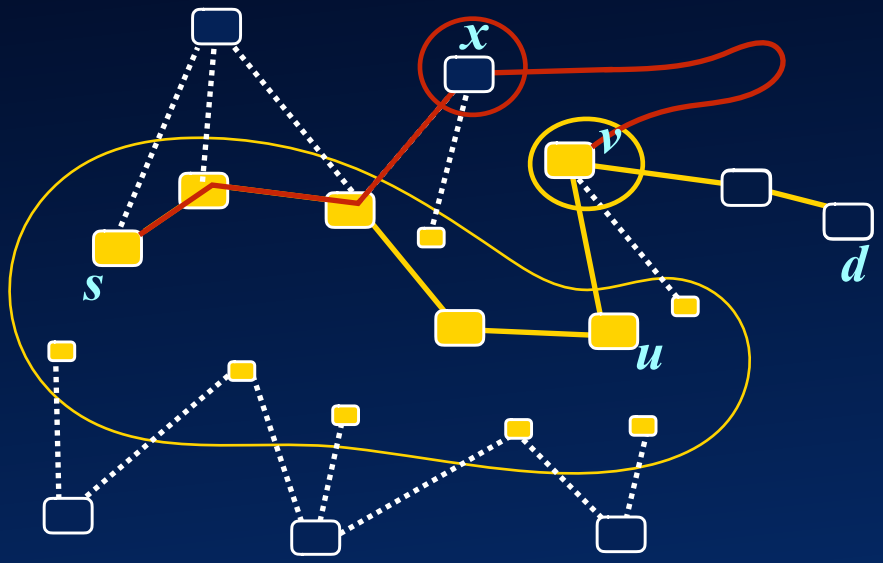
Induction step:

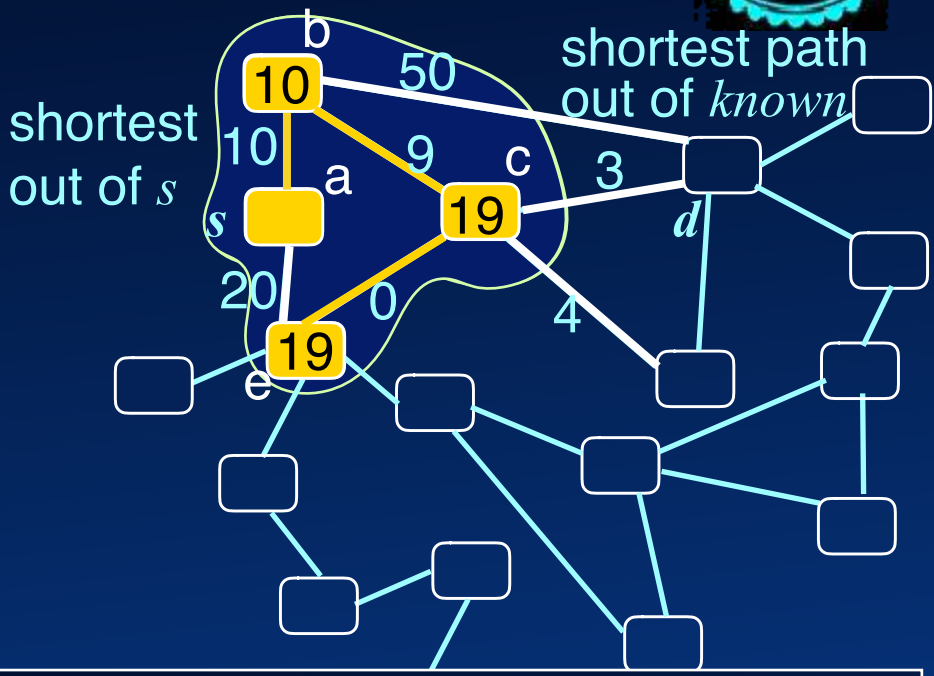
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

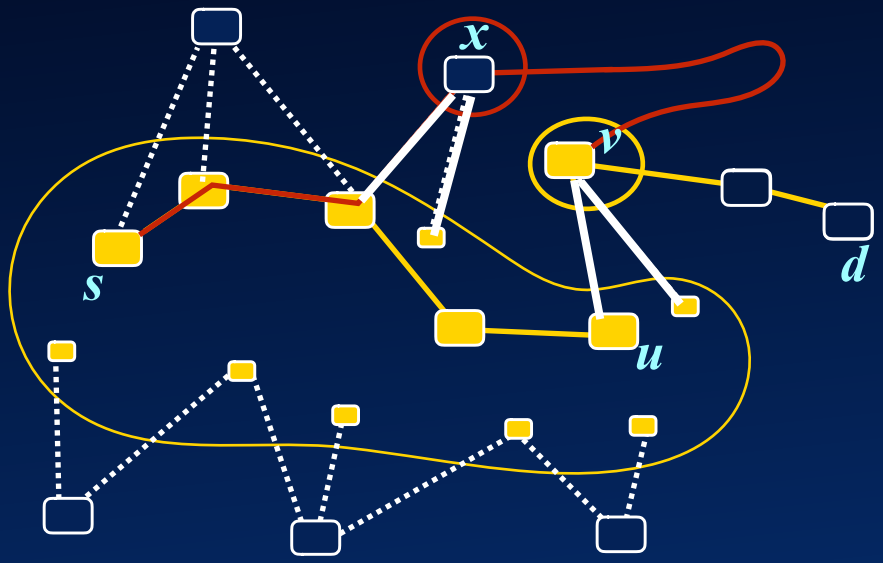
Induction step:

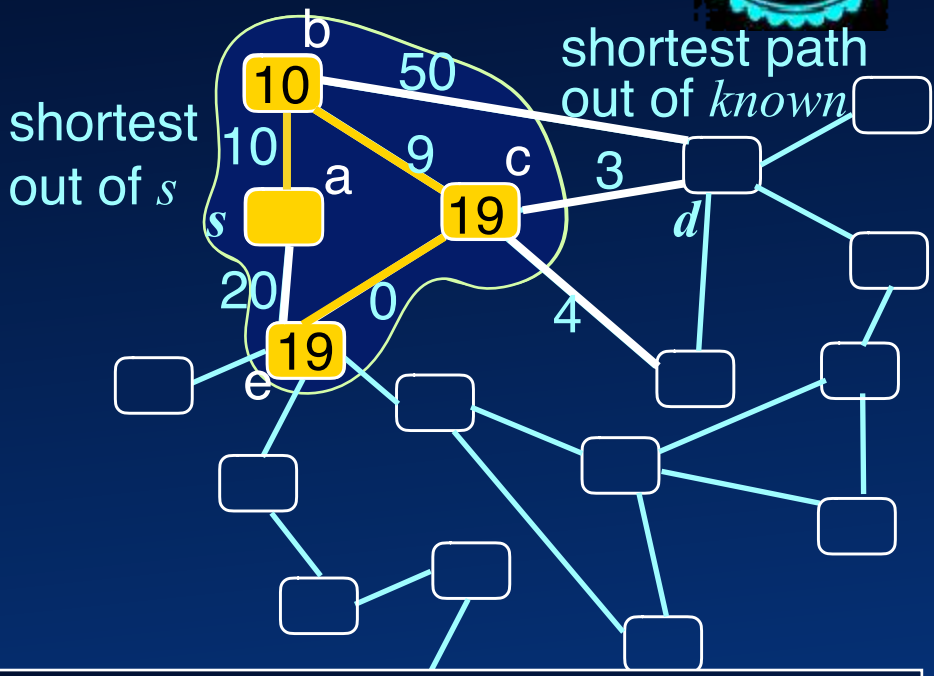
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

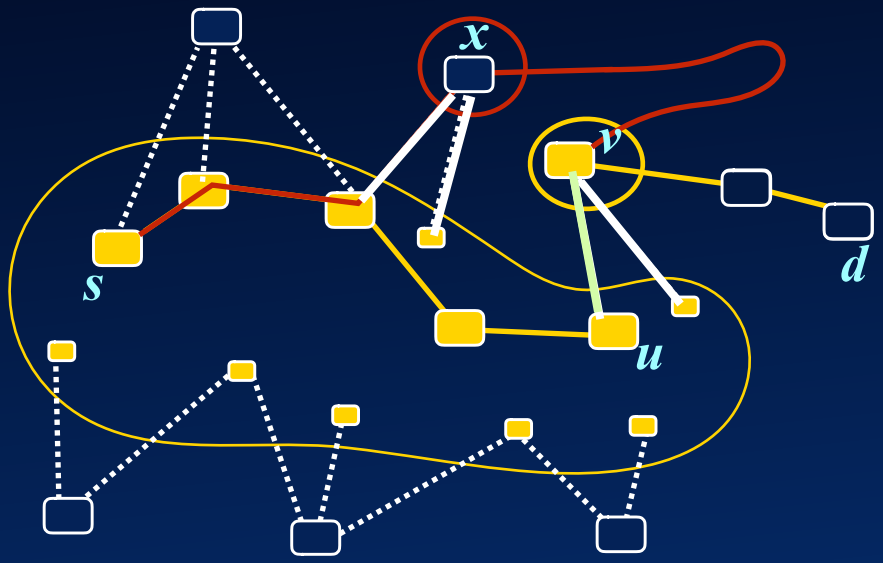
Induction step:

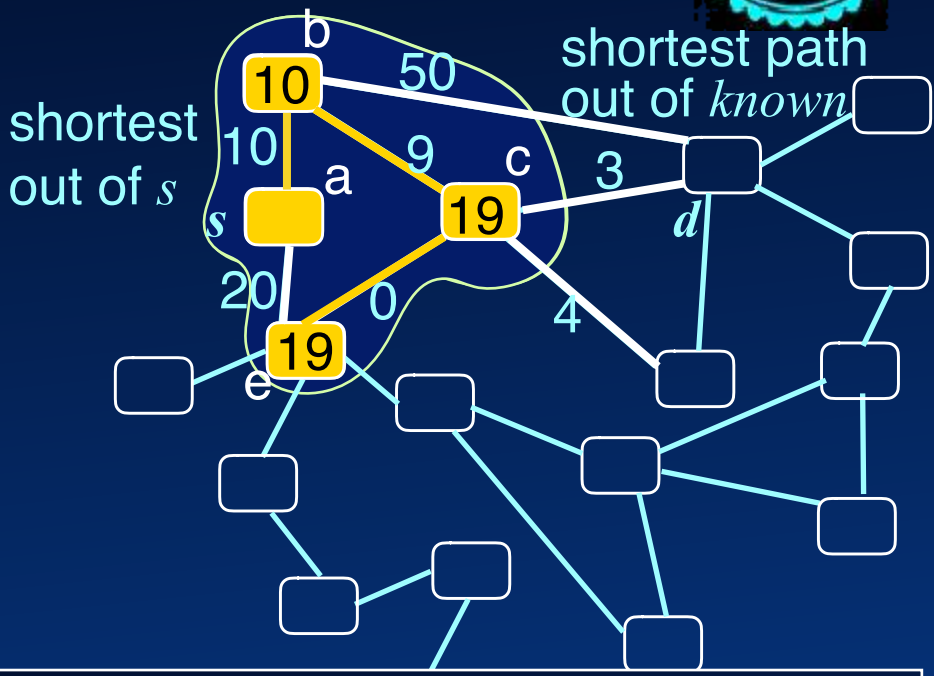
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

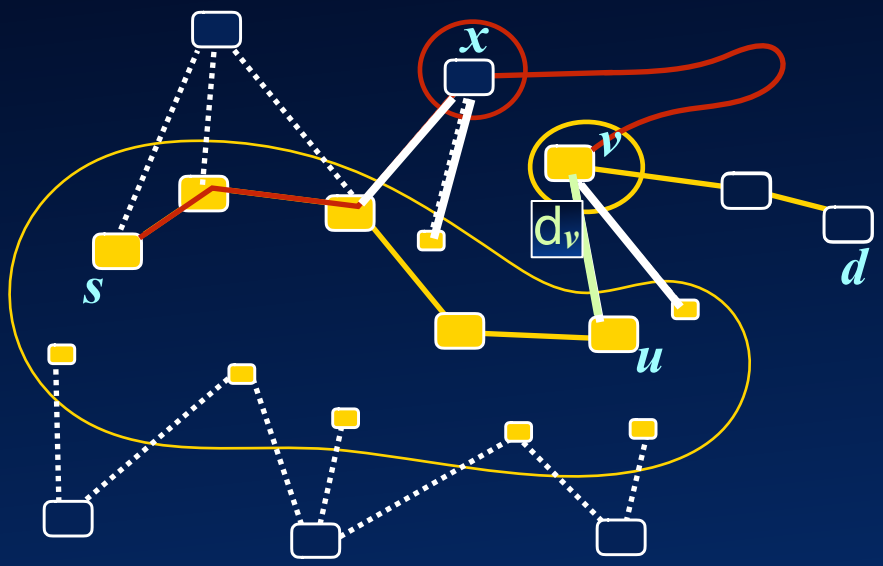
Induction step:

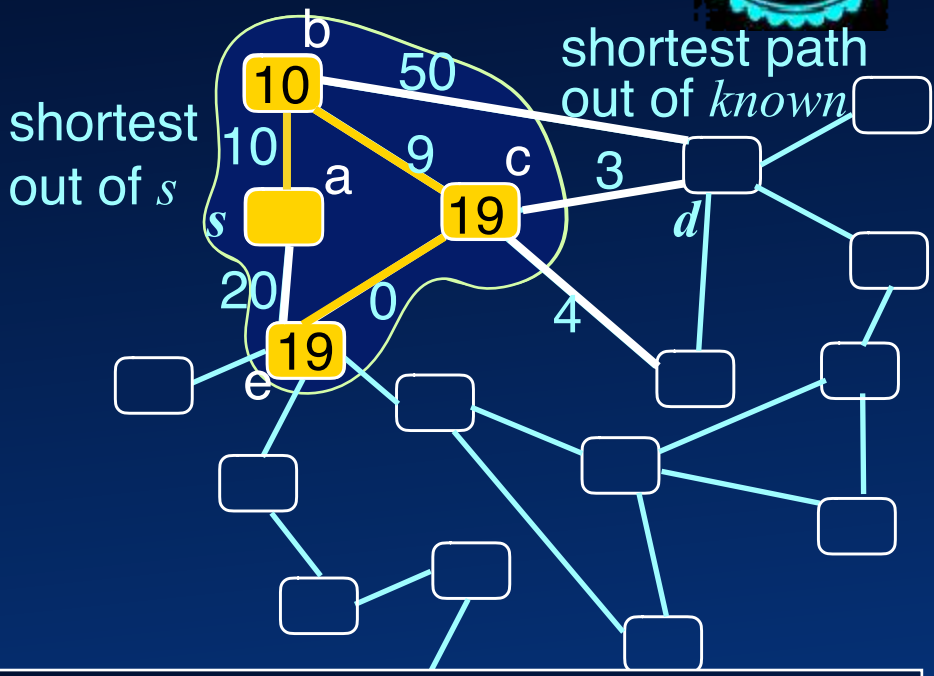
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

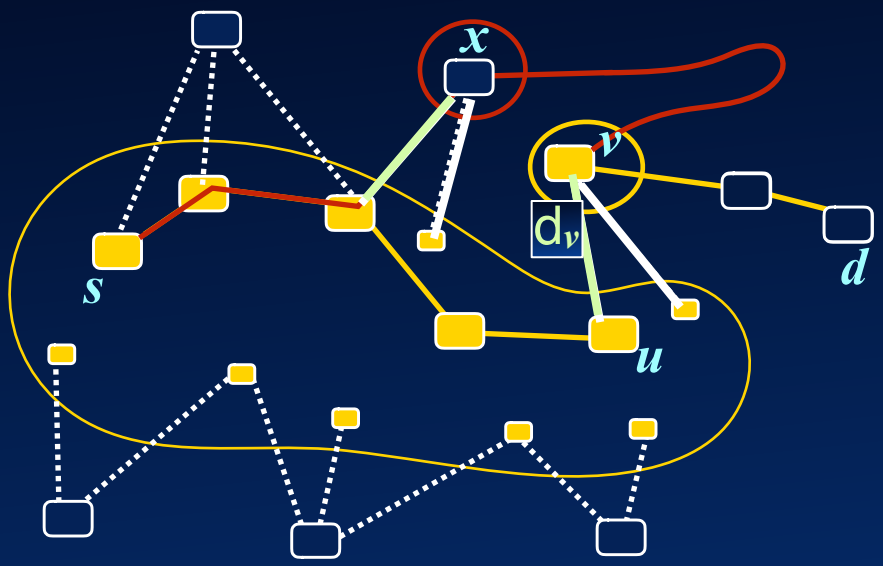
Induction step:

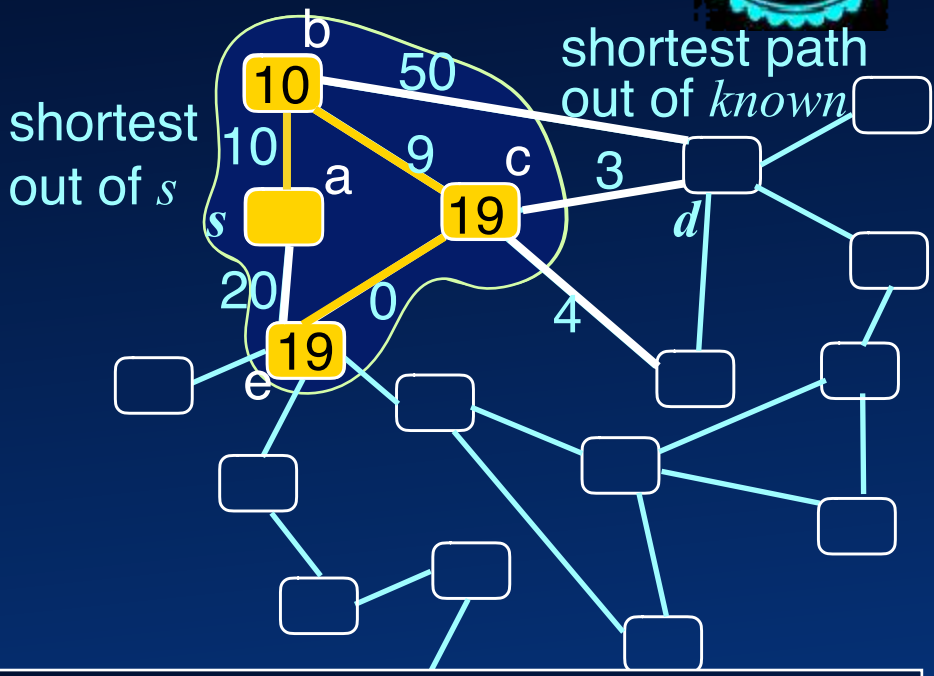
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

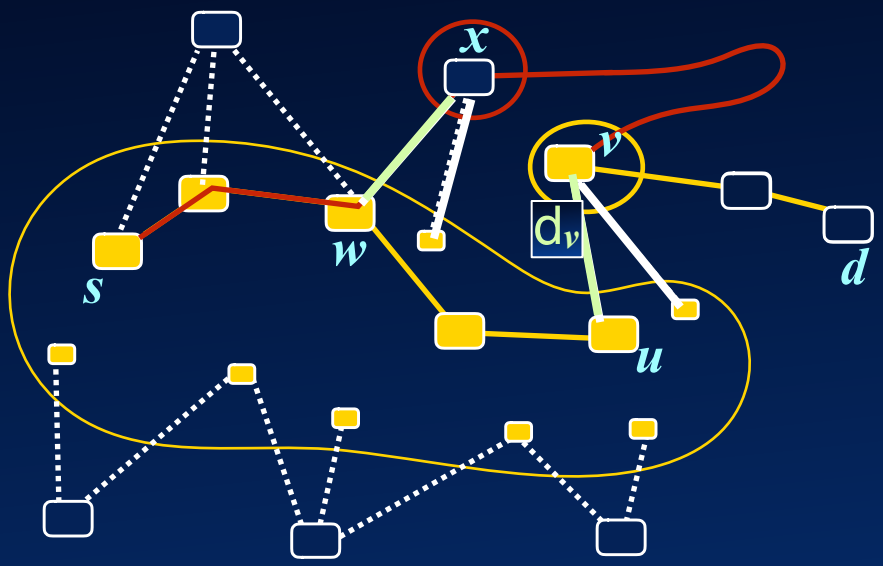
Induction step:

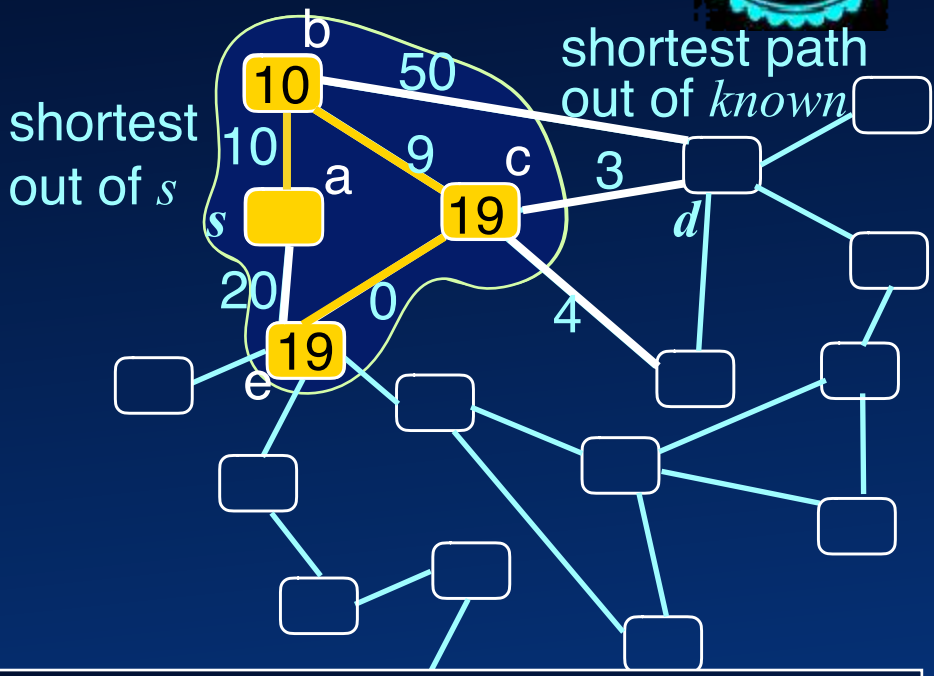
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

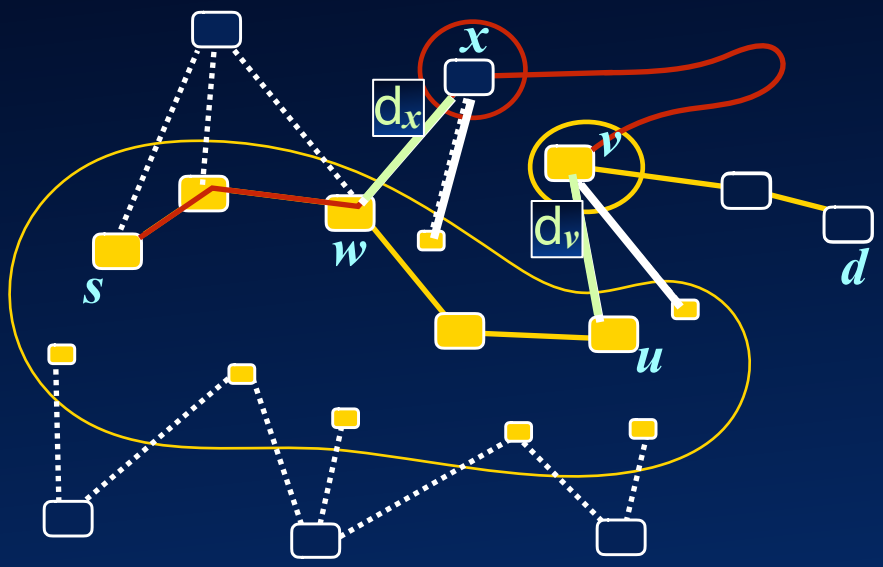
Induction step:

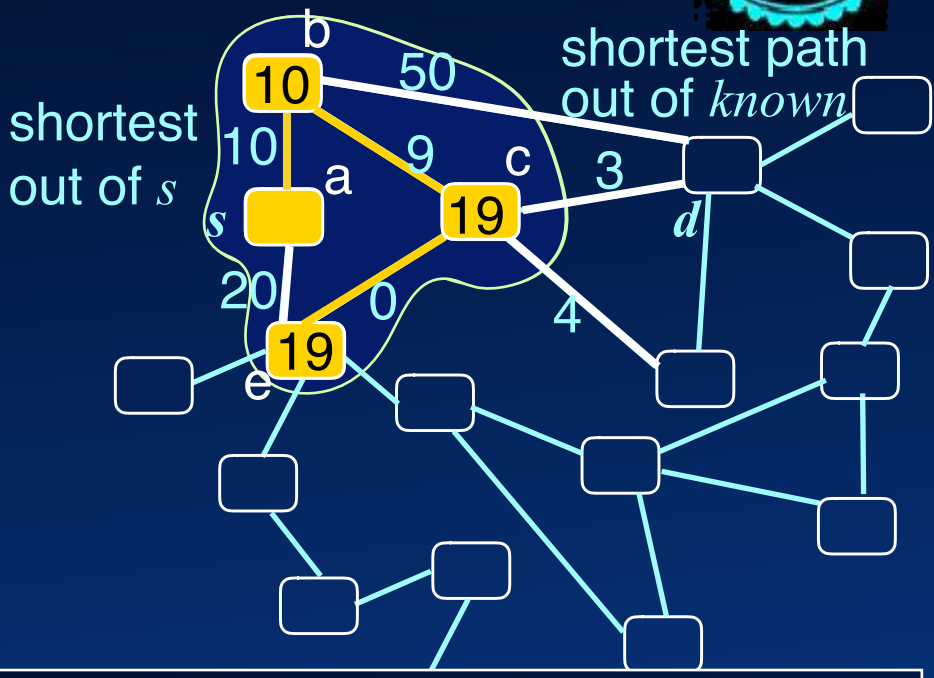
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

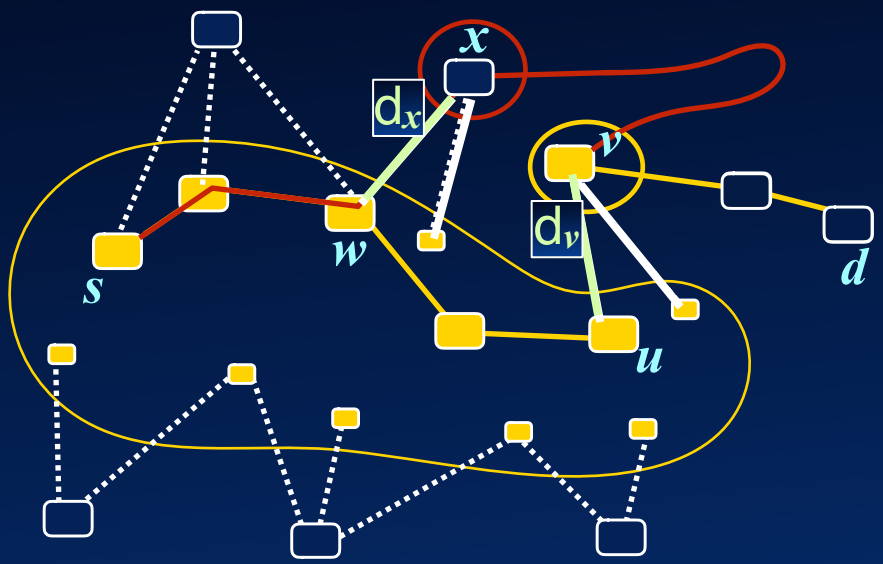
Induction step:

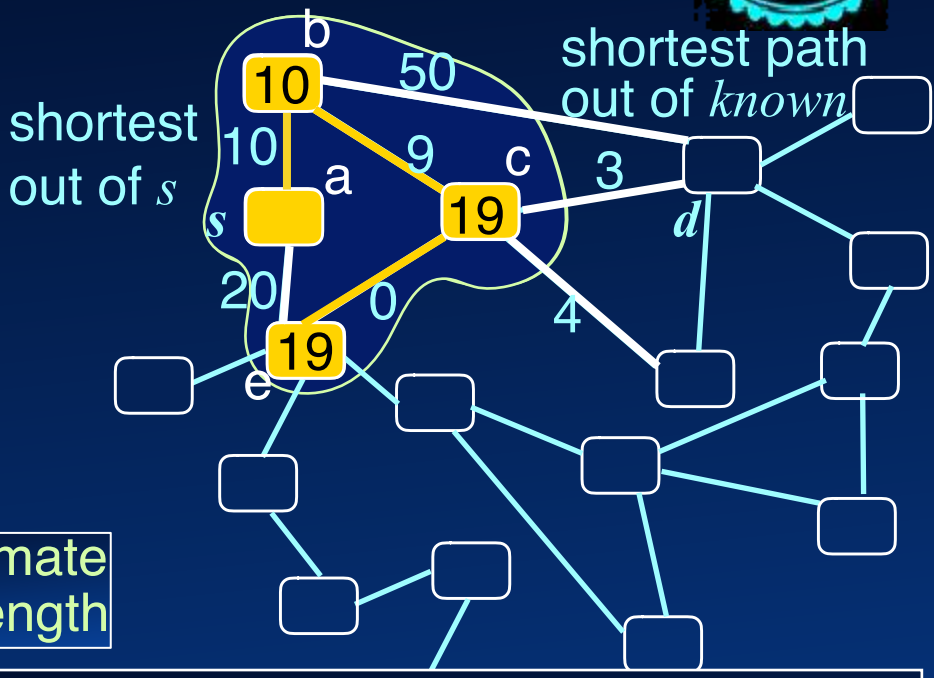
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length

Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

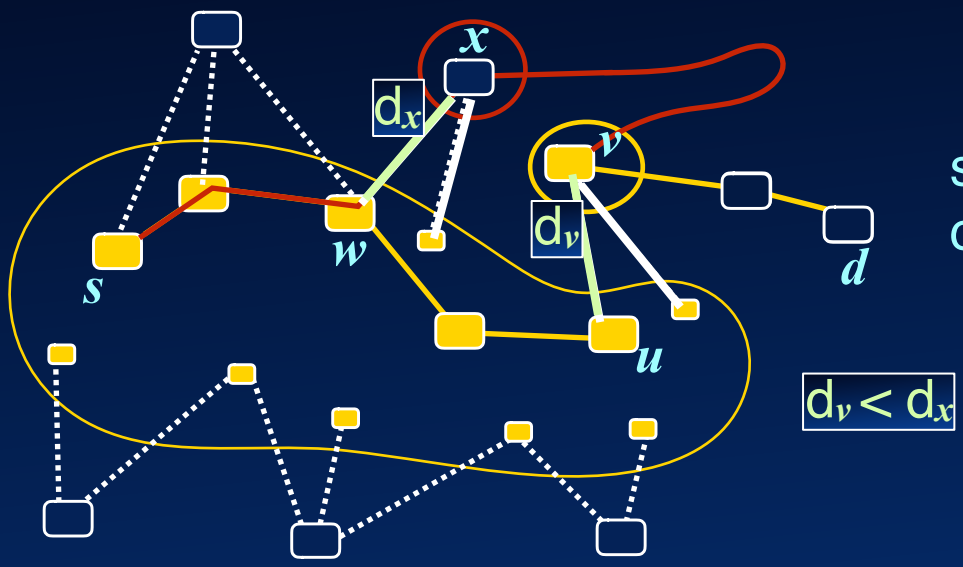
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

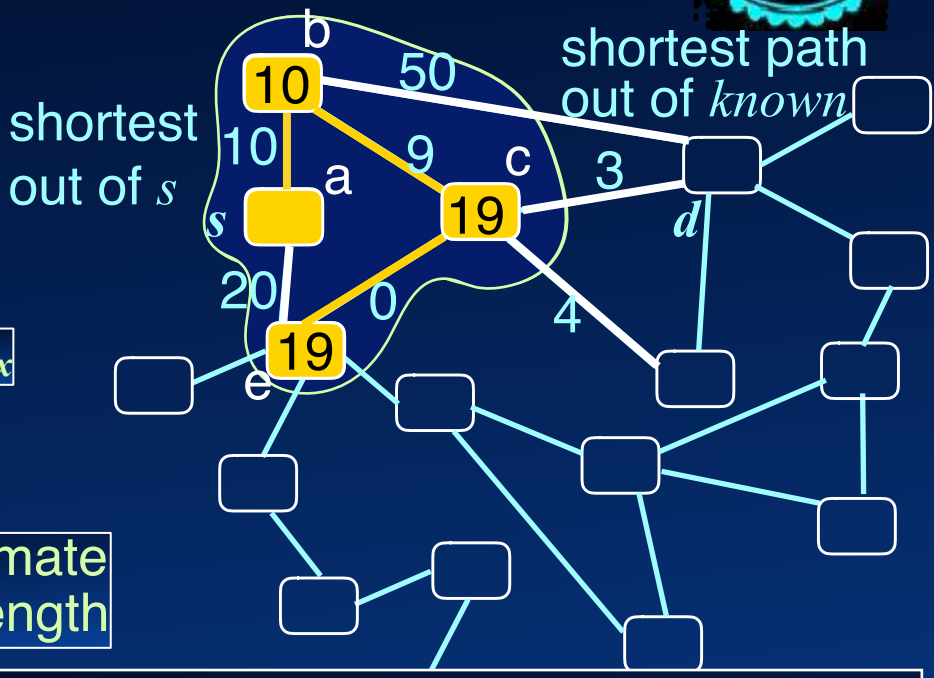
Cluster = {s}

Shortest Path



$d_v < d_x$

d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

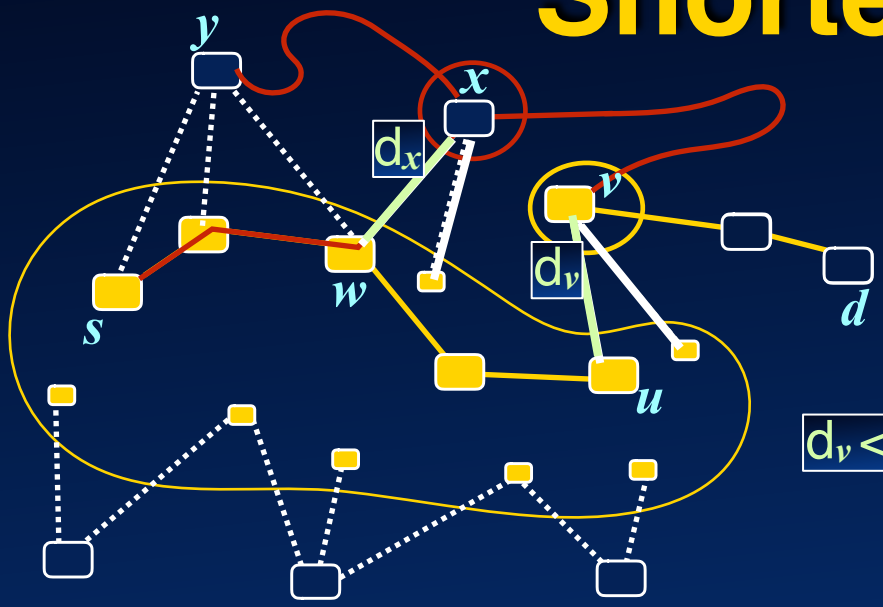
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

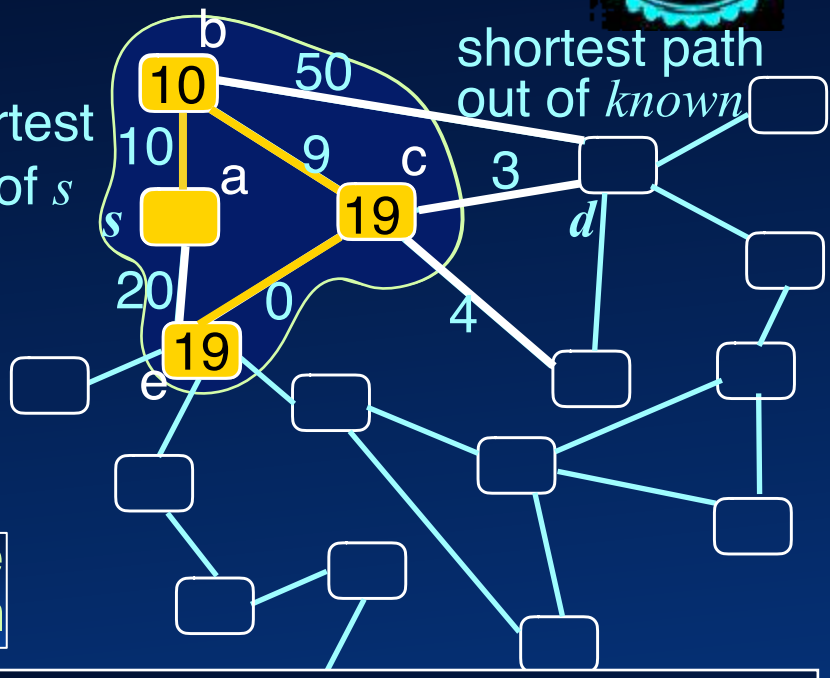
Cluster = {s}

Shortest Path



$d_v < d_x$

d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

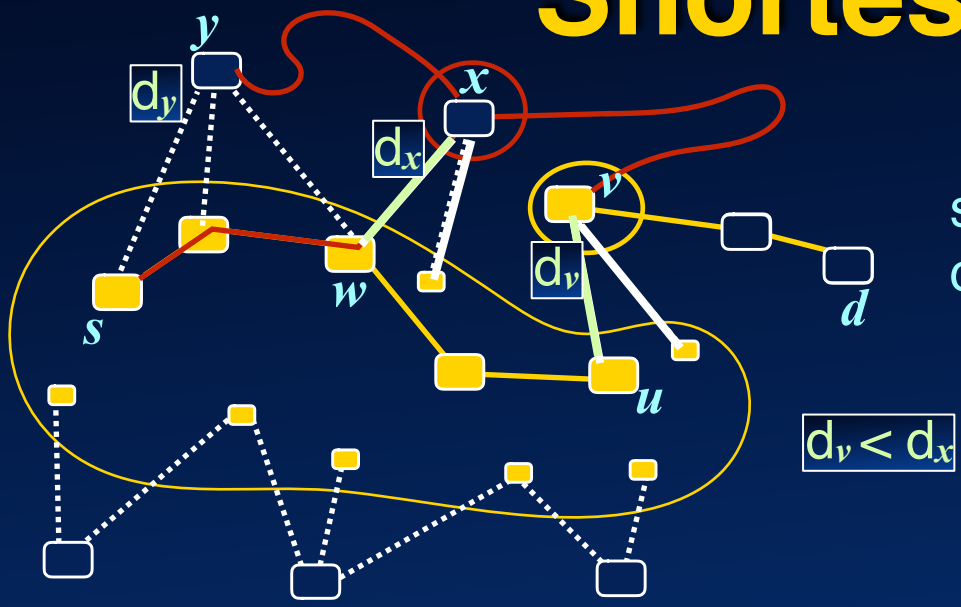
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

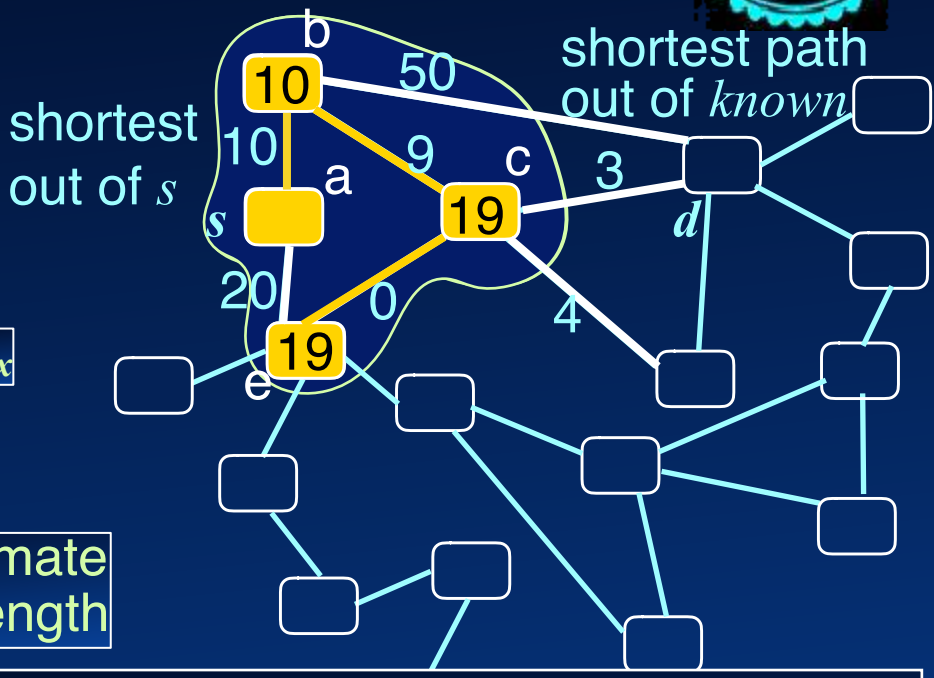
Cluster = {s}

Shortest Path



$d_v < d_x$

d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

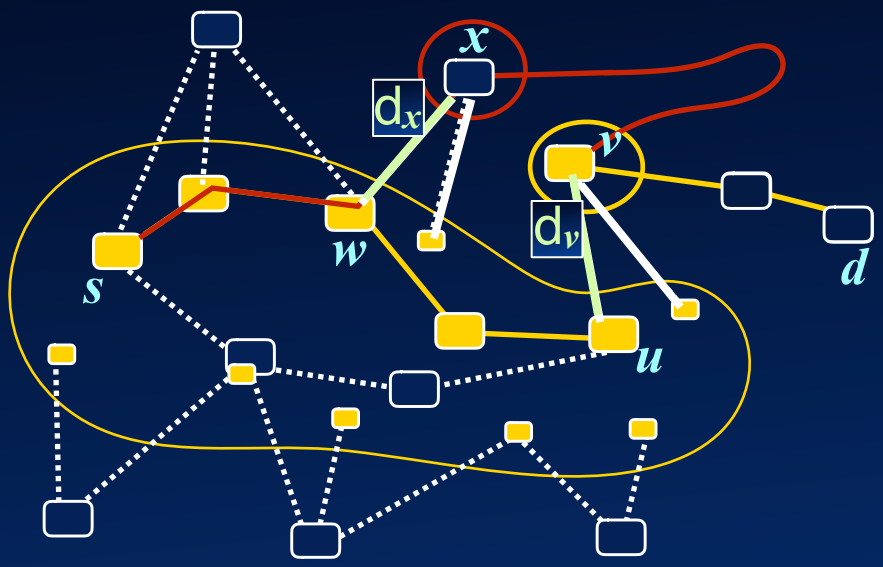
Cluster = {s}

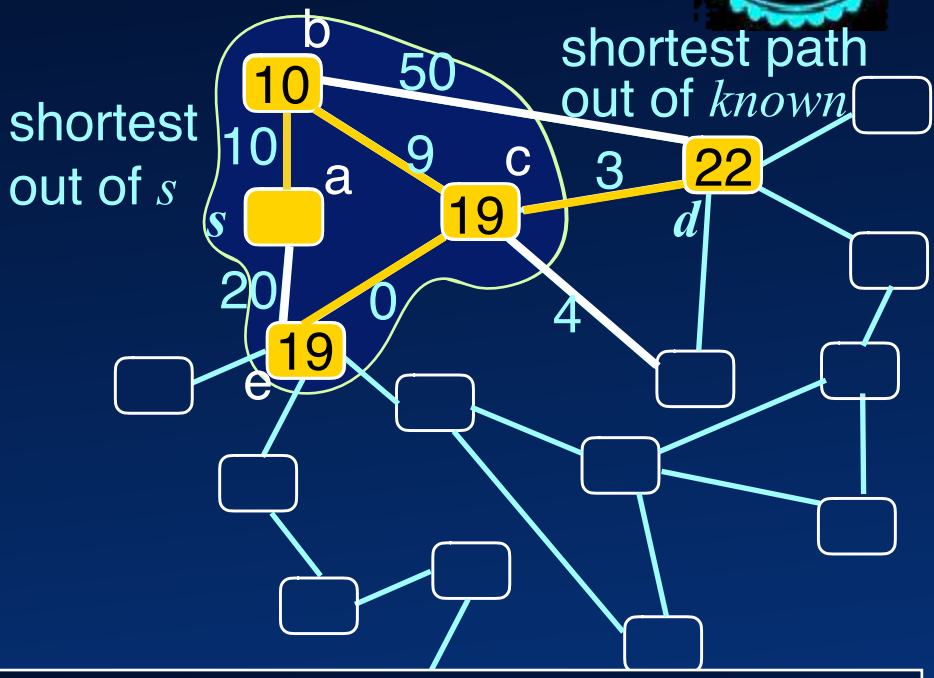
Mail: col106quiz@cse.iitd.ac.in
Format: f,t,t/f

T/F

- The shortest path from node u to node v in a directed graph is equal to the shortest path from node v to node u.
- The shortest path from node u to node v in an undirected graph is equal to the shortest path from node v to node u.
- If P_1 is the shortest path from node u to node w in a directed graph, and P_2 is the shortest path from w to node v, P_1P_2 (i.e., P_1 followed by P_2) is the shortest path from u to v.

Shortest Path





Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

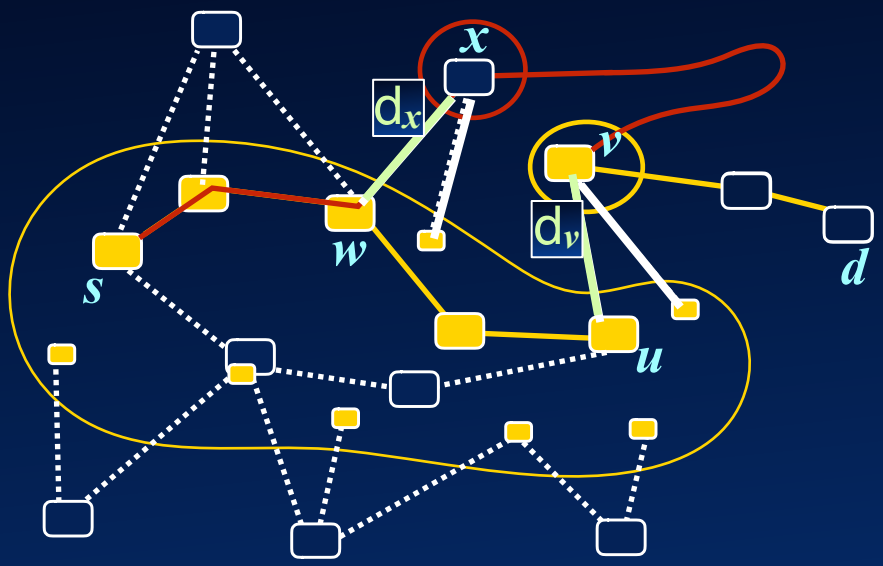
Induction step:

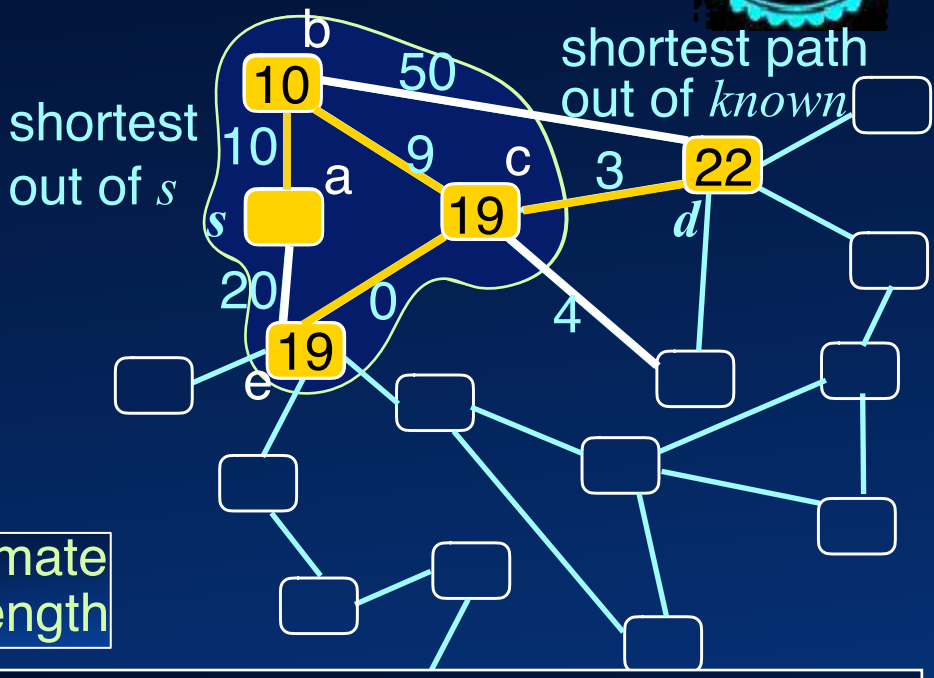
Find the shortest path from s to the next node added to the cluster

Base:

Cluster = {s}

Shortest Path





d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length

Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

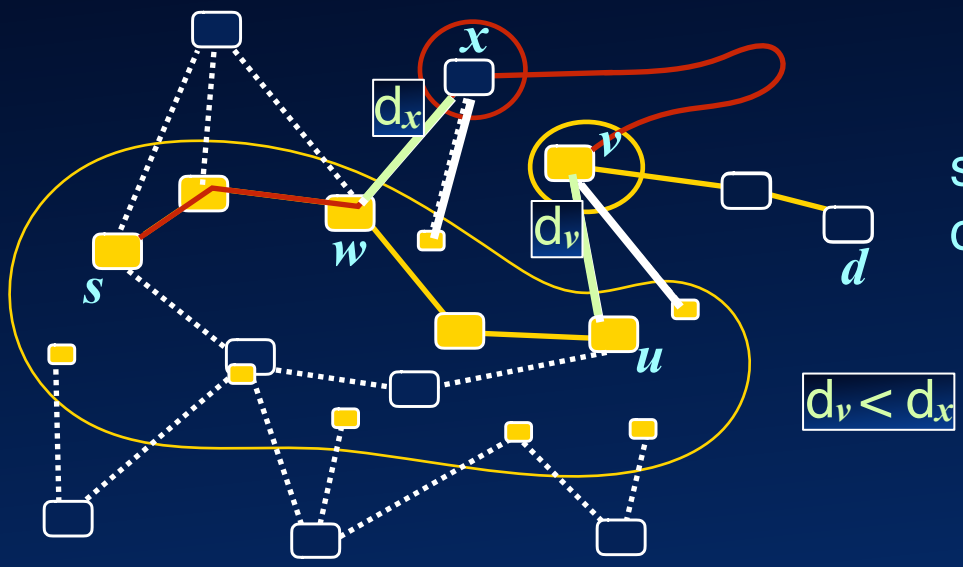
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

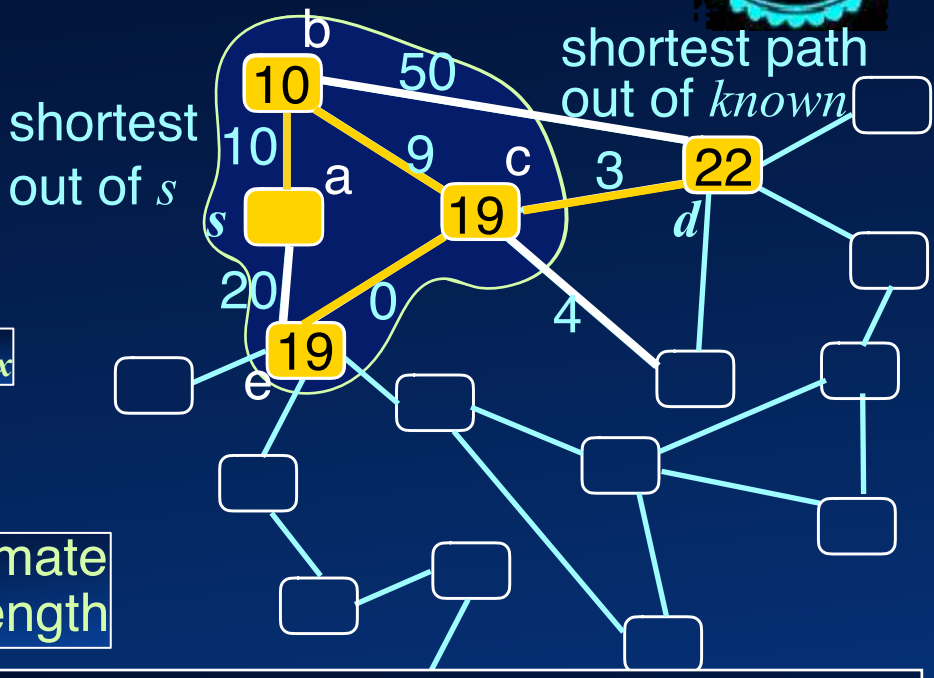
Cluster = {s}

Shortest Path



$d_v < d_x$

d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

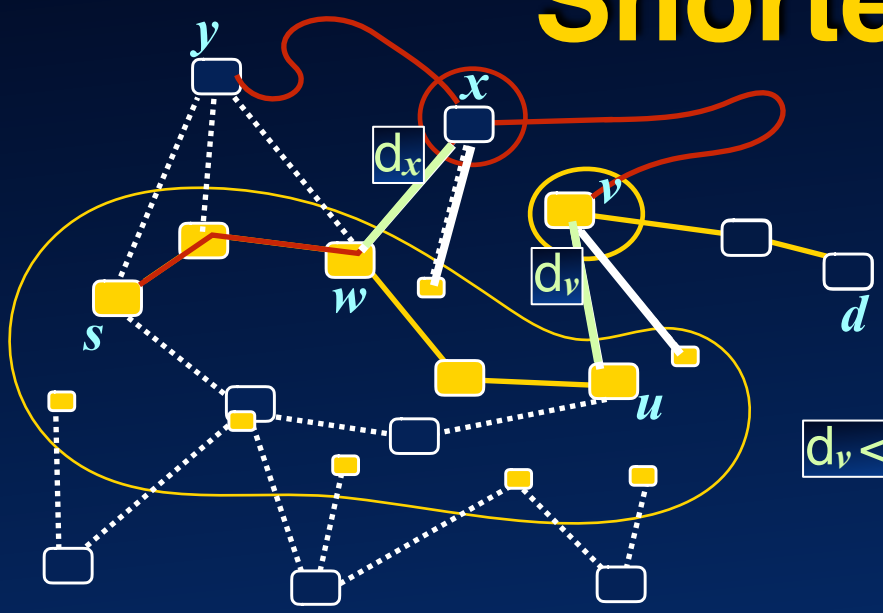
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

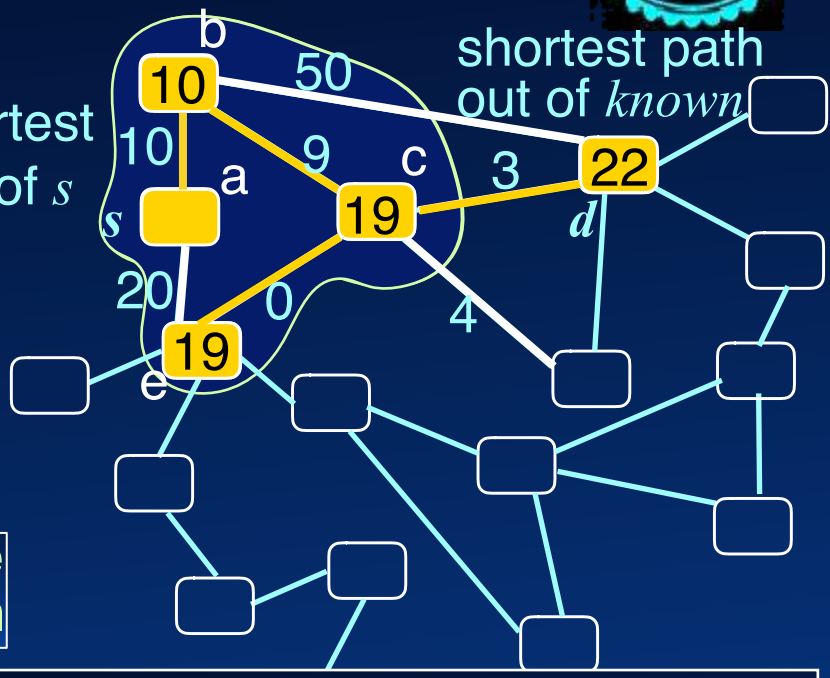
Cluster = {s}

Shortest Path



$d_v < d_x$

d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

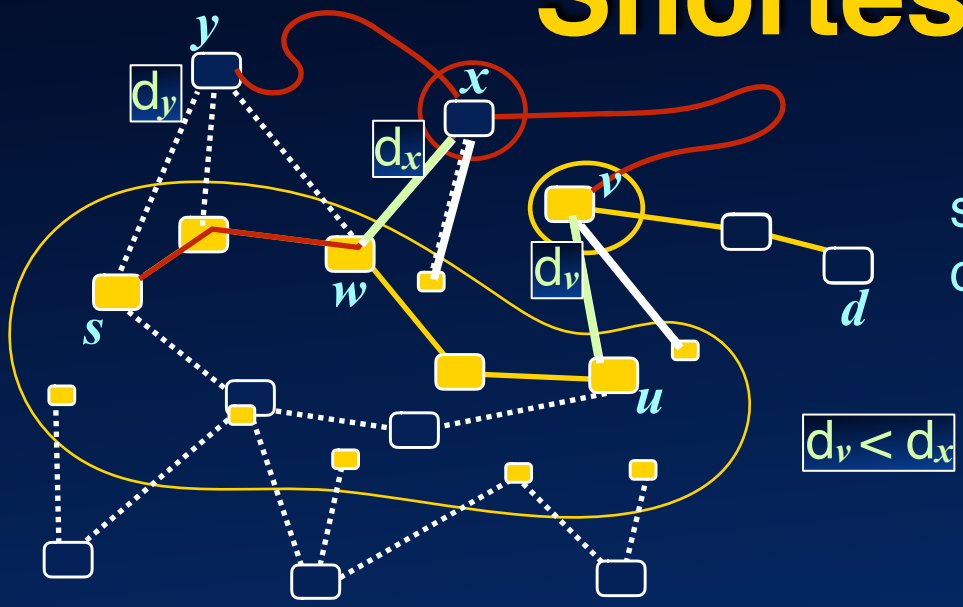
Induction step:

Find the shortest path from s to the next node added to the cluster

Base:

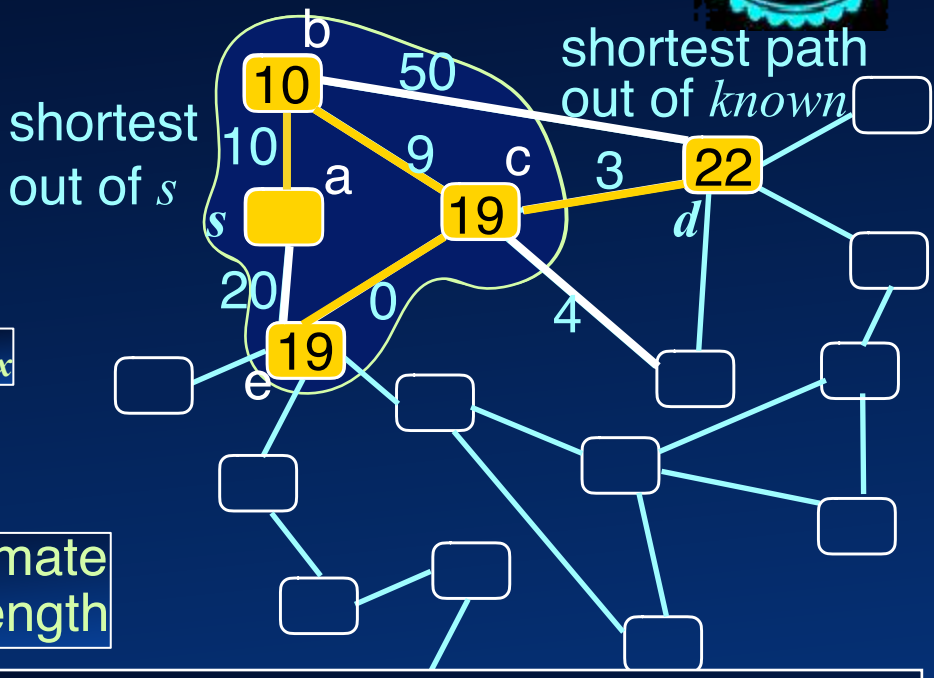
Cluster = {s}

Shortest Path



$d_v < d_x$

d_v is the smallest estimate
$\Rightarrow$ d_v is the shortest length



Induction Hypothesis:

Shortest path from s to all nodes in cluster is given

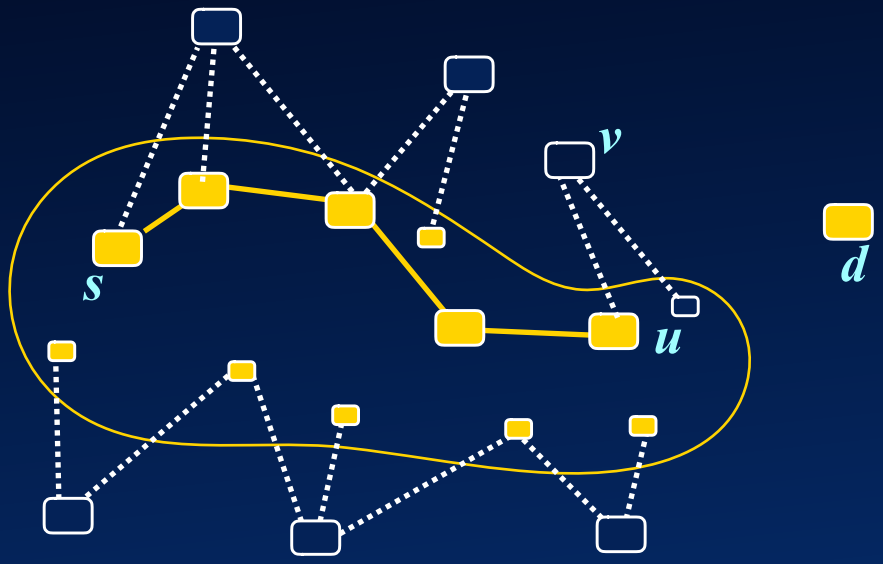
Induction step:

Find the shortest path from s to the next node added to the cluster

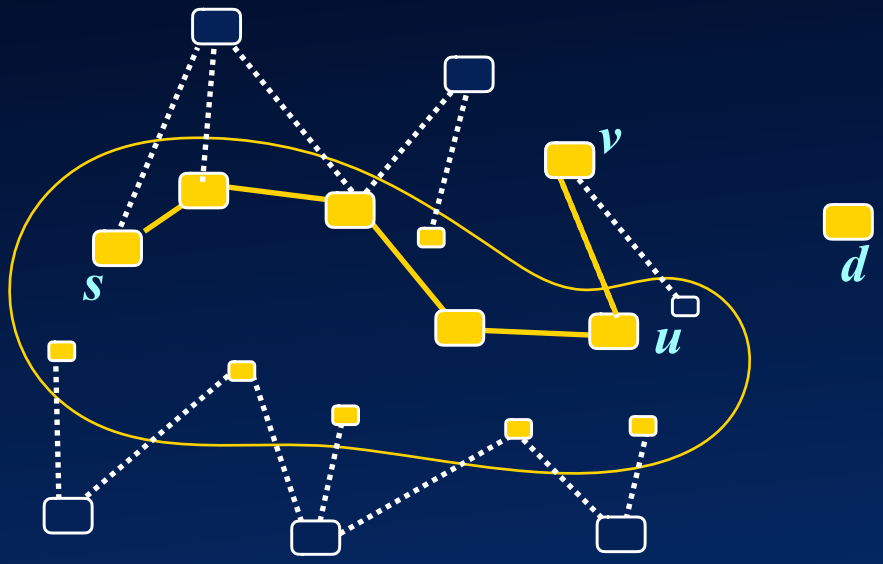
Base:

Cluster = {s}

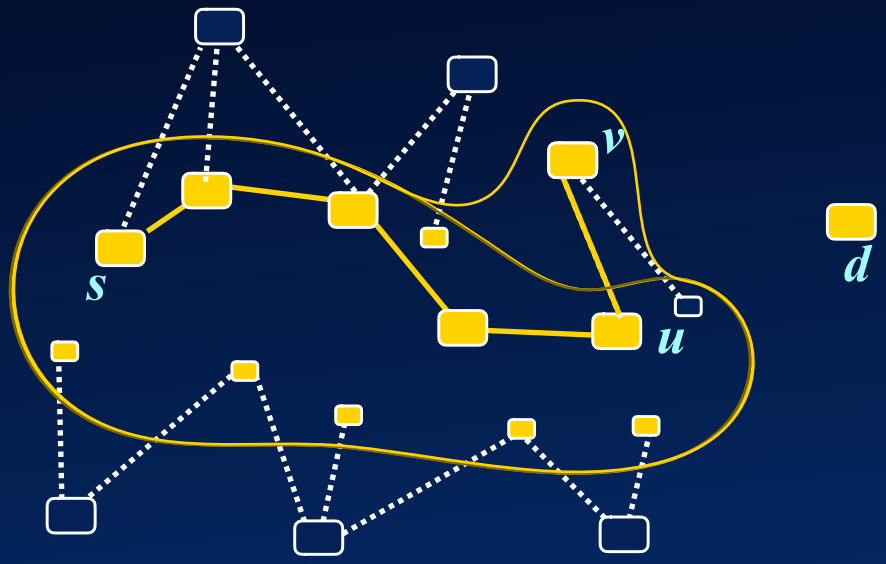
Shortest Path



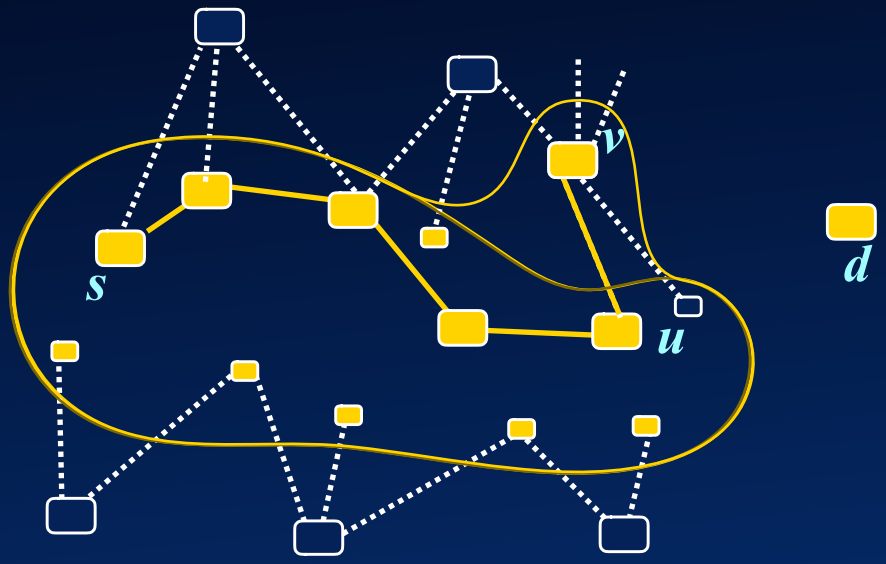
Shortest Path



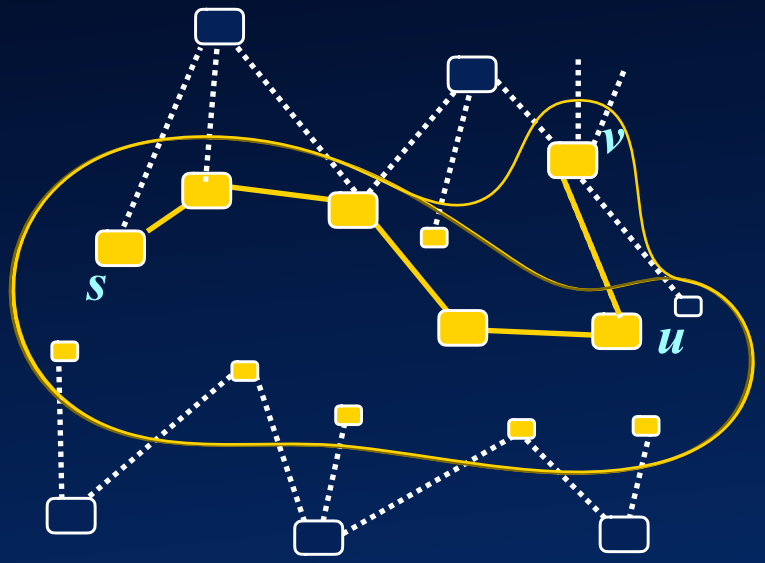
Shortest Path



Shortest Path



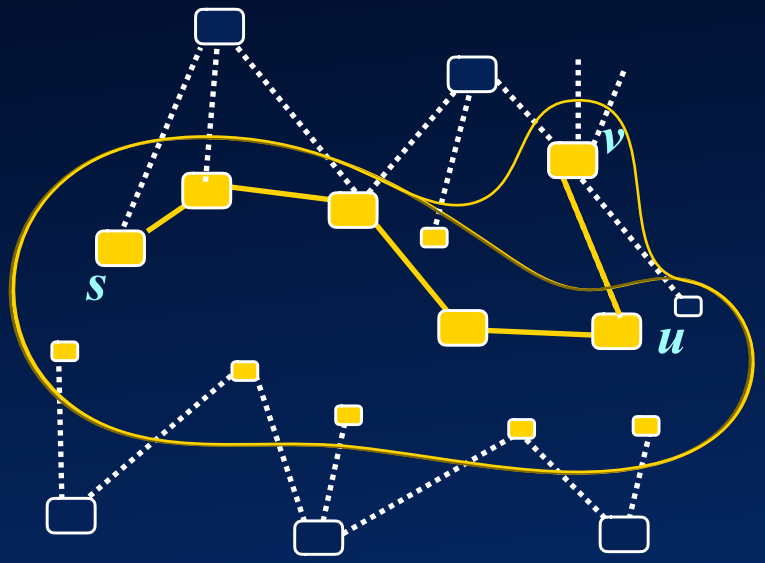
Shortest Path



Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
  PQ.insert(G.nodes)
  while PQ.hasNext():
     v = PQ.next()
     for w in v.adjacent():
        if(w.dist > v.dist + length(v,w))
           w.dist = v.dist + length(v,w)
           if(w.equals(d)) return w.dist
           PQ.edit(w)
```

Shortest Path

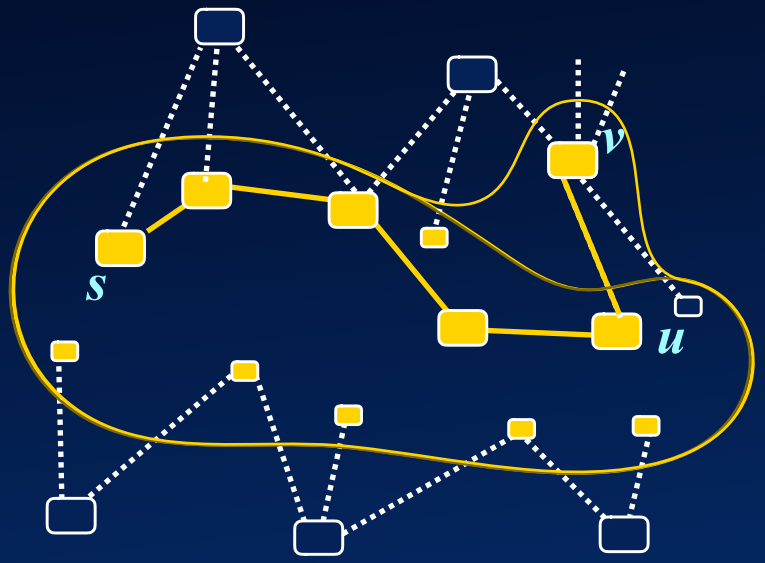


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
  PQ.insert(G.nodes)
  while PQ.hasNext():
     v = PQ.next()
     for w in v.adjacent():
        if(w.dist > v.dist + length(v,w))
           w.dist = v.dist + length(v,w)
           if(w.equals(d)) return w.dist
           PQ.edit(w)
```

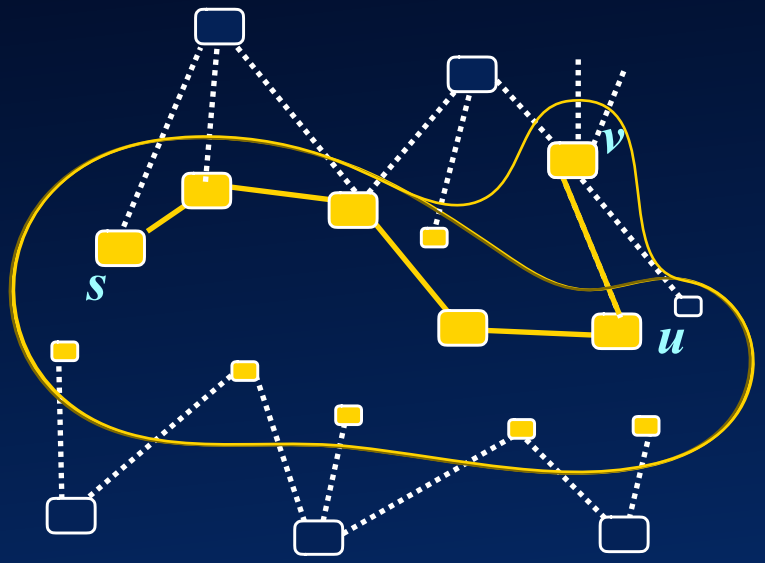
Shortest Path



Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
  PQ.insert(G.nodes)
  while PQ.hasNext():
     v = PQ.next()   👍 if v == d
     for w in v.adjacent():
        if(w.dist > v.dist + length(v,w))
           w.dist = v.dist + length(v,w)
           if(w.equals(d)) return w.dist
           PQ.edit(w)
```

Shortest Path

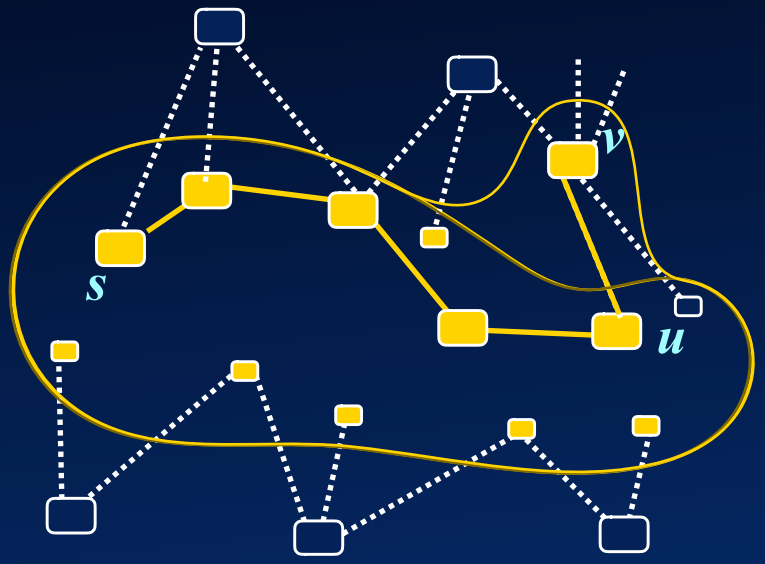


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
👉PQ.insert(G.nodes)
  while PQ.hasNext():
    v = PQ.next()  👍 if v == d
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        ~~if(w.equals(d)) return w.dist~~
        PQ.edit(w)
```

Shortest Path

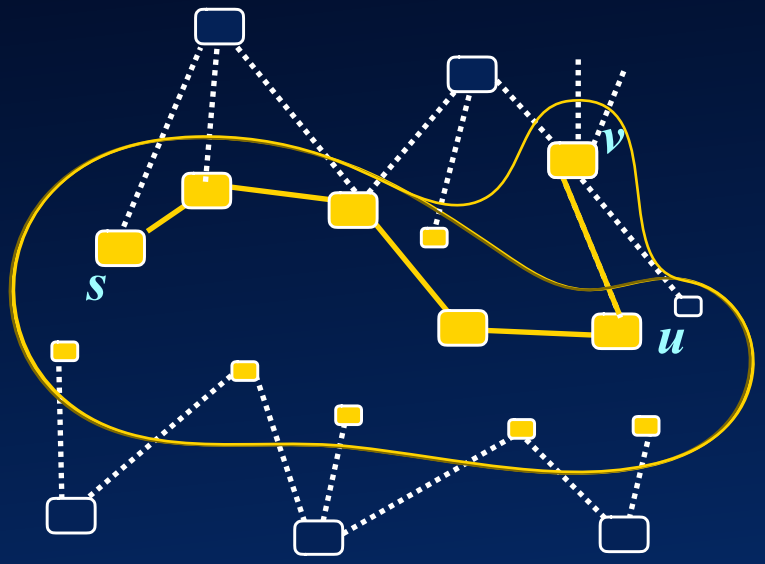


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
👉PQ.insert(G.nodes)
  while PQ.hasNext():
    v = PQ.next()   👍 if v == d
👉  for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path

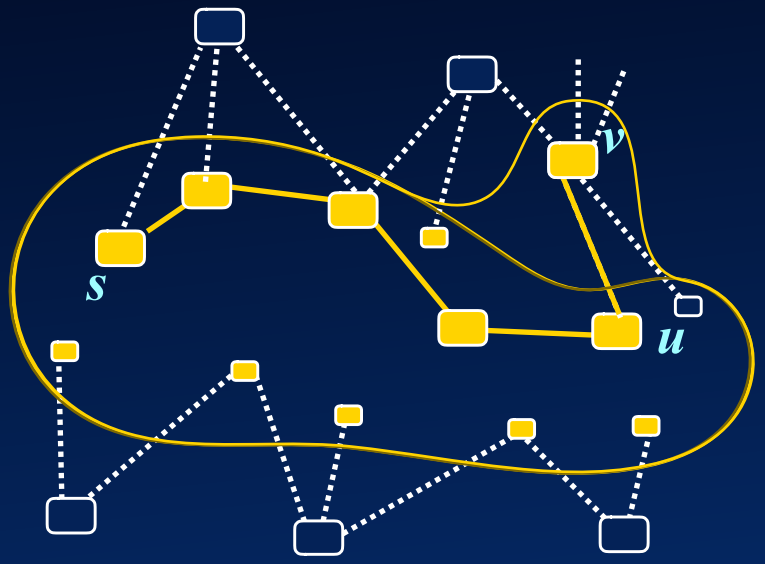


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
  PQ.insert(G.nodes)
  while PQ.hasNext():
    v = PQ.next()   if v == d
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path



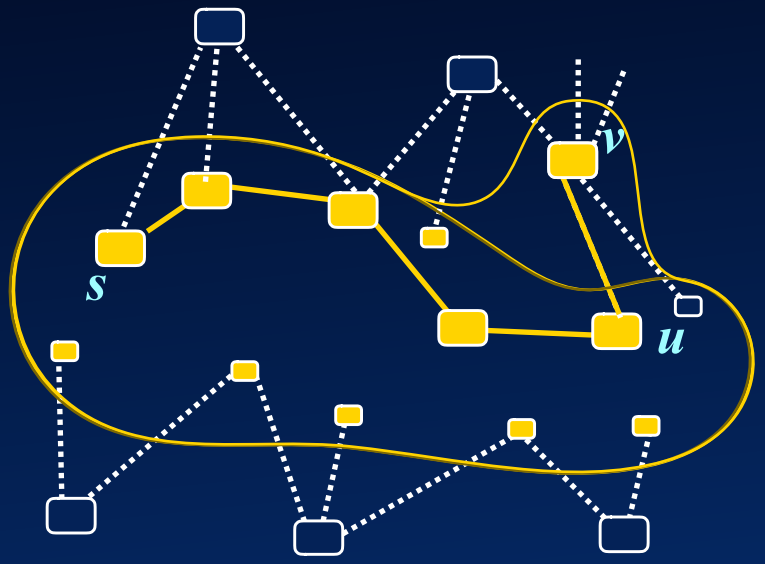
d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
  PQ.insert(G.nodes)
  while PQ.hasNext():
    v = PQ.next()
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

n

Shortest Path



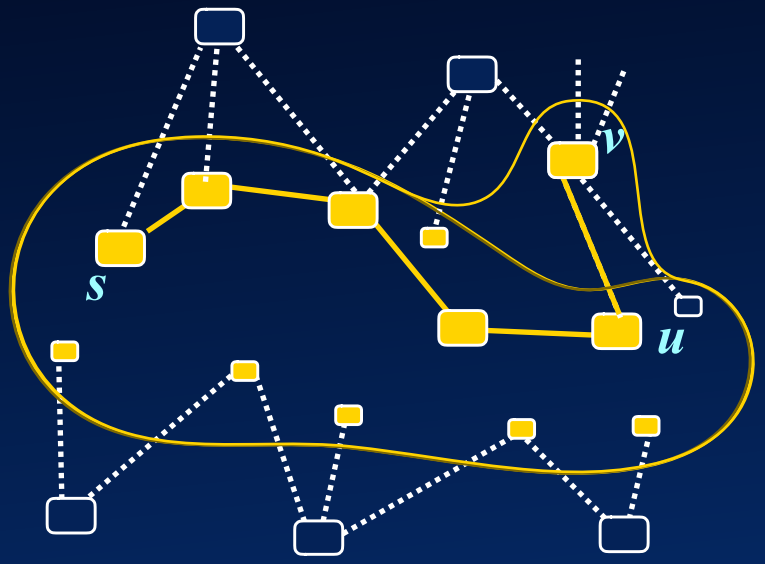
d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s
  PQ.insert(G.nodes)
  while PQ.hasNext():
    v = PQ.next()   if v == d
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

n

Shortest Path

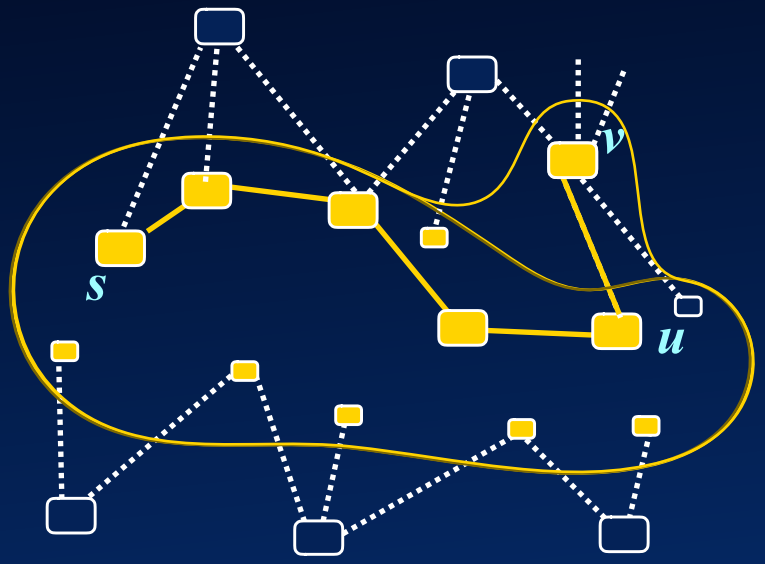


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s          ← n
  PQ.insert(G.nodes)            ← n
  while PQ.hasNext():
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path

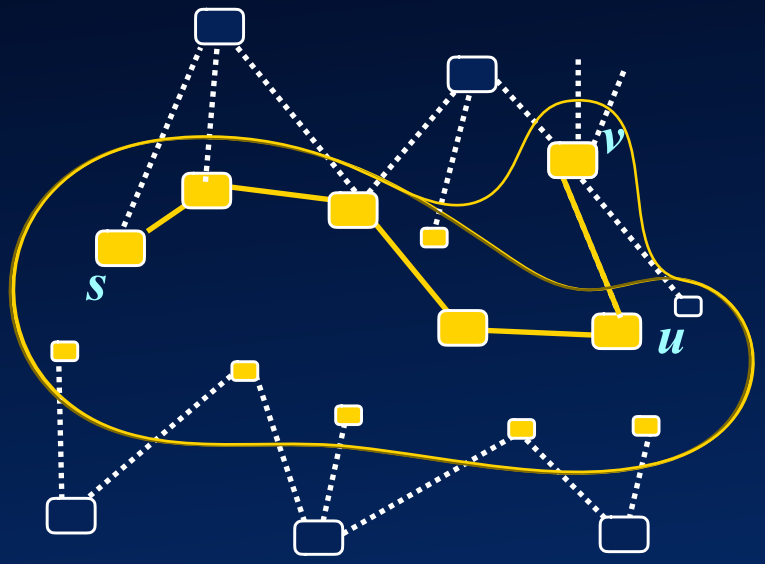


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s          ← n
  PQ.insert(G.nodes)            ← n
  while PQ.hasNext():           n
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path

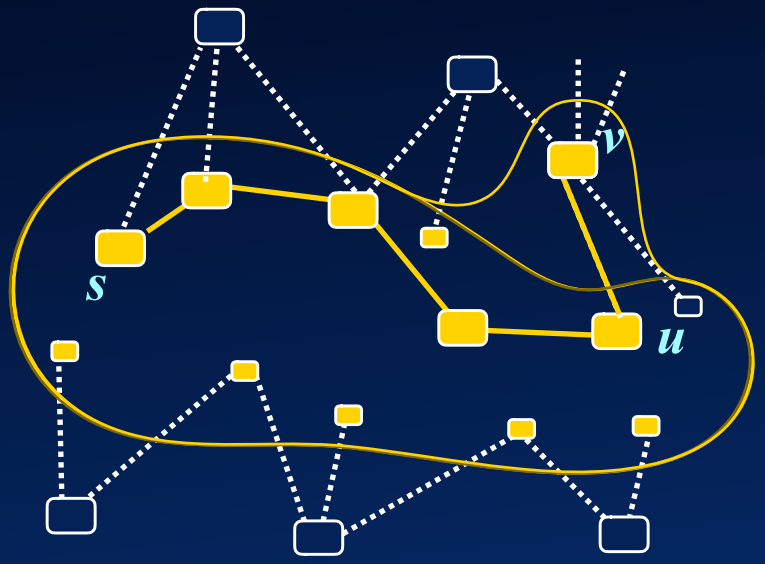


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s          ← n
  PQ.insert(G.nodes)            ← n
  while PQ.hasNext():           ← n
    v = PQ.next()   if v == d
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path

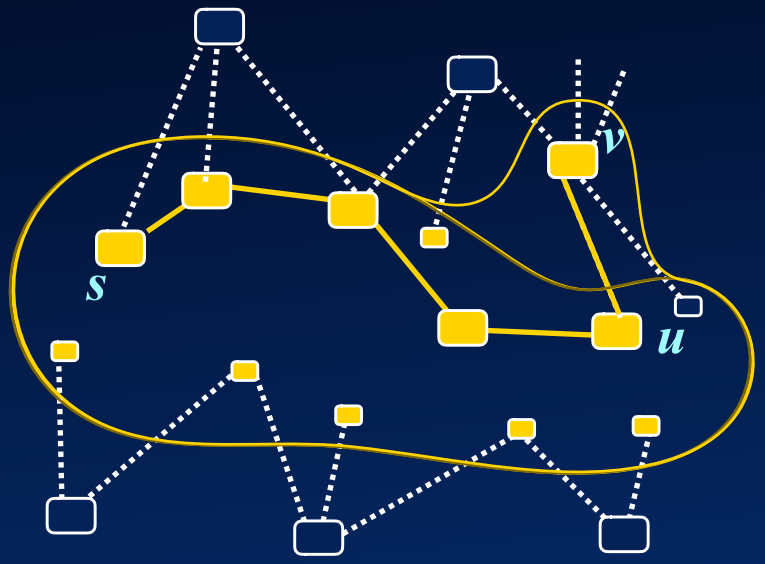


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s          ← n
  PQ.insert(G.nodes)            ← n
  while PQ.hasNext():           ← n
    v = PQ.next()   if v == d
    for w in v.adjacent():
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path

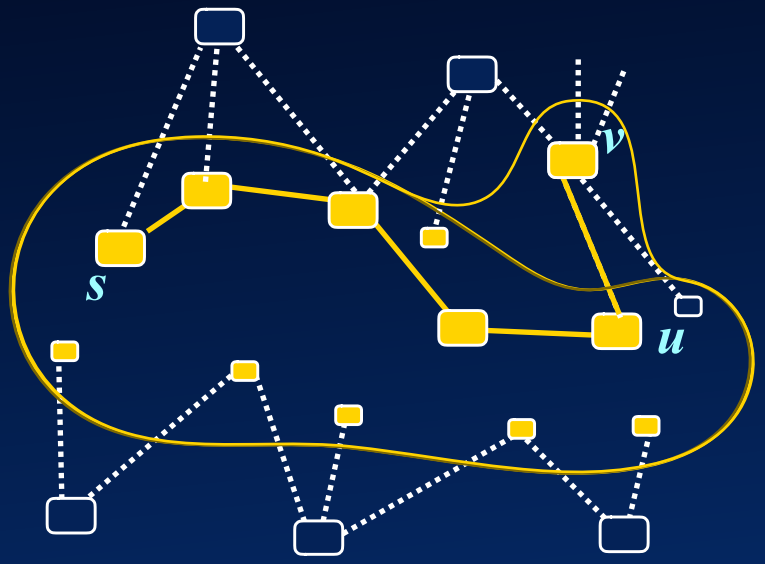


d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s                    ← n
  PQ.insert(G.nodes)                      ← n
  while PQ.hasNext():                     ← n
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():                ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path



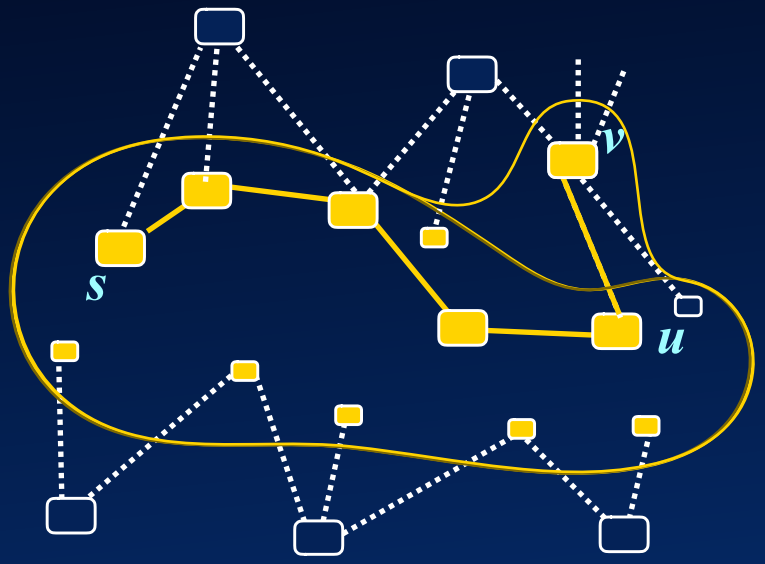
d

∑degree(u) = *m*

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s          ← n
  PQ.insert(G.nodes)            ← n
  while PQ.hasNext():           ← n
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():      ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

Shortest Path



d

Dijkstra's Algorithm

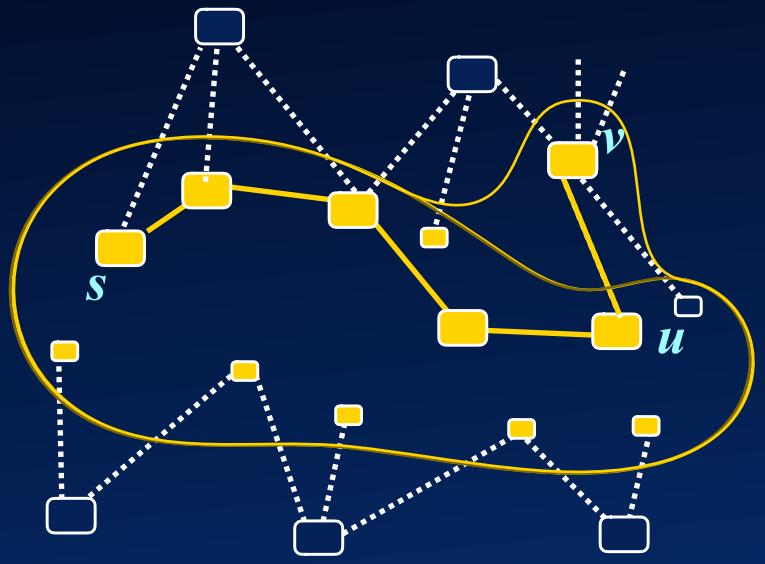
```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s                    ← n
  PQ.insert(G.nodes)                      ← n
  while PQ.hasNext():                     ← n
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():                ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        ~~if(w.equals(d)) return w.dist~~
        PQ.edit(w)
```

$\sum$degree(u) = m

n * O(log n): PQ.next()

m * O(log n): PQ.edit

Shortest Path



d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s                    ← n
  PQ.insert(G.nodes)                      ← n
  while PQ.hasNext():                     ← n
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():                ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        ~~if(w.equals(d)) return w.dist~~
        PQ.edit(w)
```

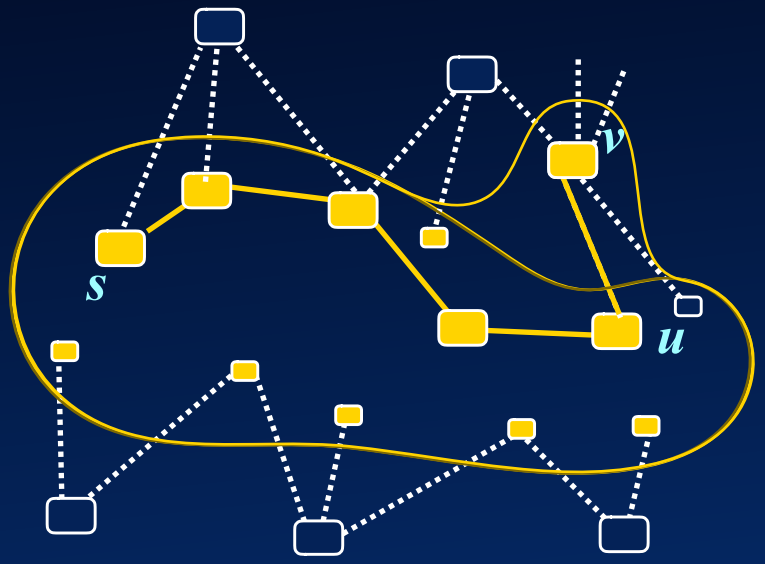
$\sum$degree(u) = m

n * O(log n): PQ.next()

m * O(log n): PQ.edit

$O(m+n)\log n$ if PQ is a heap

Shortest Path



Heap

d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s          ← n
  PQ.insert(G.nodes)            ← n
  while PQ.hasNext():           ← n
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():      ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        ~~if(w.equals(d)) return w.dist~~
        PQ.edit(w)
```

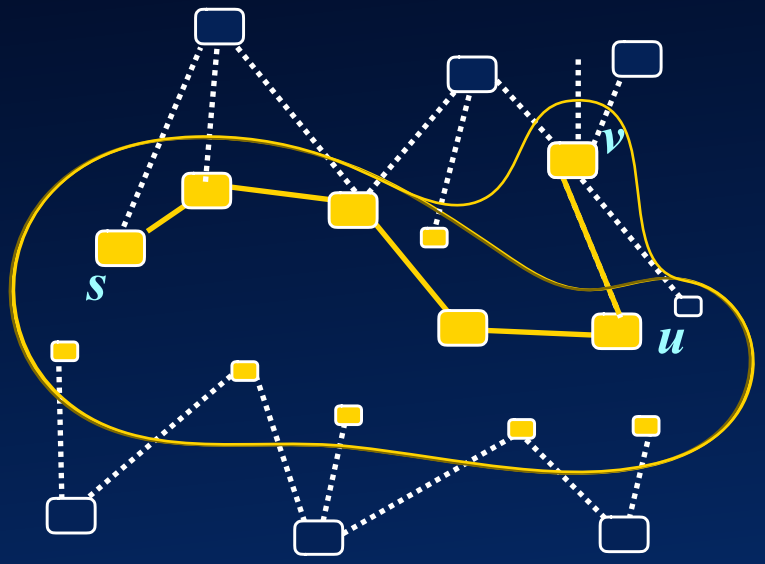
$\sum$degree(u) = m

n * O(log n): PQ.next()

m * O(log n): PQ.edit

$O(m+n)\log n$ if PQ is a heap

Shortest Path



Heap

d

Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s                    ← n
  PQ.insert(G.nodes)                      ← n
  while PQ.hasNext():                     ← n
    v = PQ.next()   if v == d
    for w in v.adjacent():                ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        ~~if(w.equals(d)) return w.dist~~
        PQ.edit(w)
```

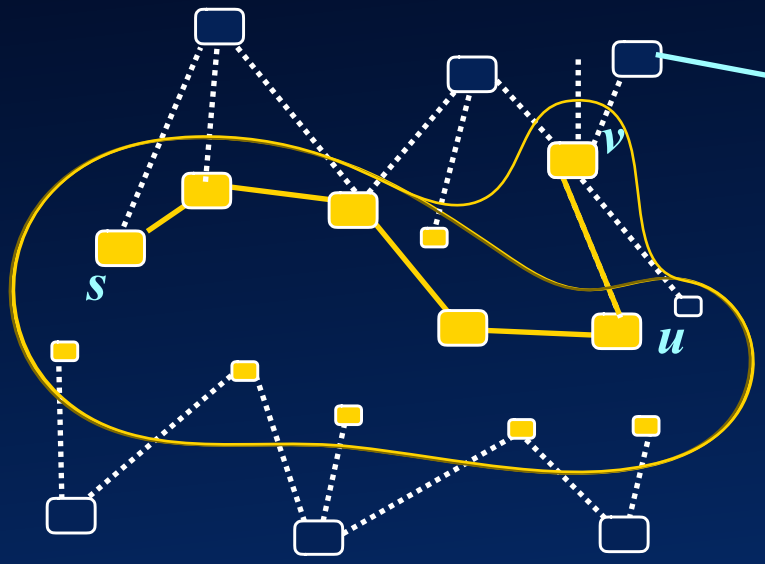
$\sum$degree(u) = m

n * O(log n): PQ.next()

m * O(log n): PQ.edit

$O(m+n)\log n$ if PQ is a heap

Shortest Path



Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s                    ← n
  PQ.insert(G.nodes)                      ← n
  while PQ.hasNext():                     ← n
    v = PQ.next()   if v == d
    for w in v.adjacent():                ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

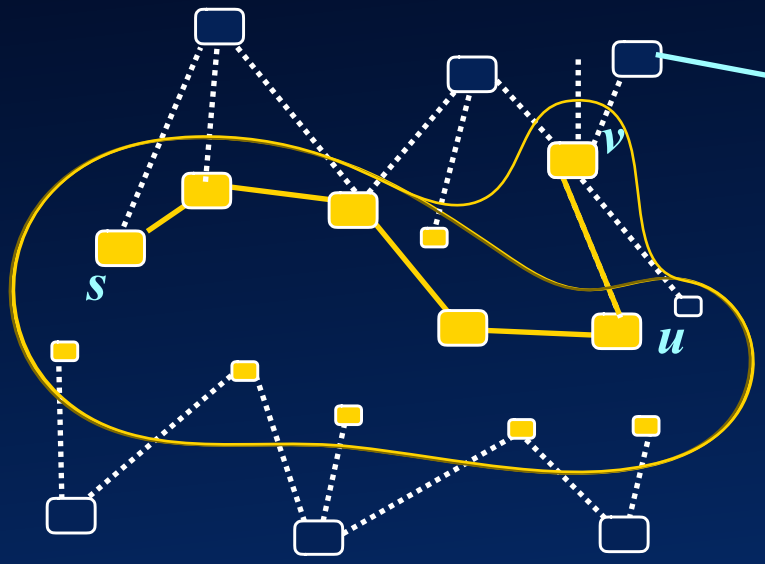
$\sum$degree(u) = m

n * O(log n): PQ.next()

m * O(log n): PQ.edit

$O(m+n)\log n$ if PQ is a heap

Shortest Path



Dijkstra's Algorithm

```
Shortest(s,d):
  s.dist = 0
  u.dist = ∞ ∀ u ∈ G-s                    ← n
  PQ.insert(G.nodes)                      ← n
  while PQ.hasNext():                     ← n
    v = PQ.next()   👍 if v == d
    for w in v.adjacent():                ← degree
      if(w.dist > v.dist + length(v,w))
        w.dist = v.dist + length(v,w)
        if(w.equals(d)) return w.dist
        PQ.edit(w)
```

$\sum$degree(u) = m

n * O(log n): PQ.next()

m * O(log n): PQ.edit

$O(m+n)\log n$ if PQ is a heap

$O(n^2)$ if PQ is a list