DSC 40B

Lecture 19 : BFS, Part 2

Review: Aggregate Analysis

Question

What is the time complexity of the following code in terms of $|V|$ and $|E|$?

```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print(”Edge:”, u, v)
```

Question

What is the time complexity of the following code in terms of $|V|$ and $|E|$?

```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print("Edge:", u, v)
```

How many times do we execute this line?

Question

What is the time complexity of the following code in terms of $|V|$ and $|E|$?

```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print("Edge:", u, v)
```

How many times do we execute this line? **E or 2*E**

Question

What is the time complexity of the following code in terms of $|V|$ and $|E|$? **Θ(V + E)** *#V to cover the case when a graph does not have edges.*

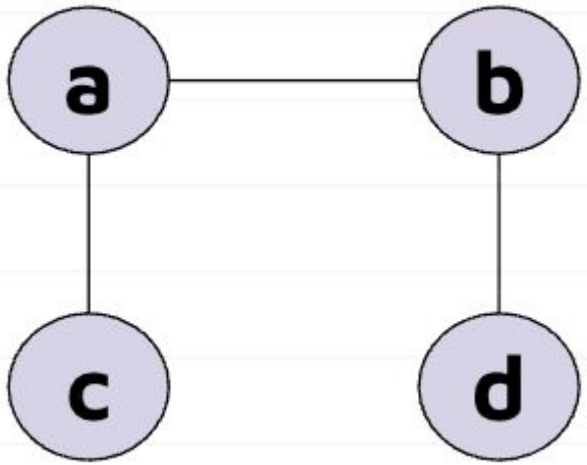
```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print("Edge:", u, v)
```

Answer

```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print(”Edge:”, u, v)
```

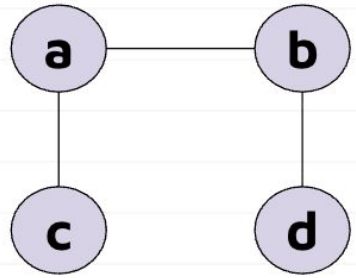
- Time complexity: $\Theta(V + E)$
- The **print** executes:
  - **once** for each edge, if the graph is **directed**.
  - **twice** for each edge, if the graph is **undirected**.

dict-of-sets representation




```
adj = {

    "a":  {"b", "c"},

    "b":  {"a", "d"},

    "c":  {"a"},

    "d": {"b"}

}
```

Another Look...




```
adj = {
    "a":  {"b", "c"},
    "b":  {"a", "d"},
    "c":  {"a"},
    "d": {"b"}
}
```

Consider the graph's `dict-of-sets` representation.

```
for u in graph.nodes:
    for v in graph.neighbors(u):
        print("Edge:", u, v)
```

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

$\Theta(\ ?\ )$

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

$\Theta(V)$

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

$\Theta(V)$

How many times do we call this line?

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending no
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

$\Theta(V)$

What is the graph if my goal is to make full_bfs to call bfs as many times as possible?

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Θ(V)

No more than V, can be ignored

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

$\Theta(V)$

No more than V, can be ignored

What line of code is executed the most?

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

$\Theta(V)$

No more than V, can be ignored

```python
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Θ(V)

No more than V, can be ignored

print(u,v) how many times?

```
def full_bfs(graph):
    status = {node: 'undiscovered' for node in graph.nodes}
    for node in graph.nodes:
        if status[node] == 'undiscovered'
            bfs(graph, node, status)

def bfs(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}

    status[source] = 'pending'
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft()
        for v in graph.neighbors(u):
            # explore edge (u,v)
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                # append to right
                pending.append(v)
        status[u] = 'visited'
```

Θ(V)

No more than V, can be ignored

print(u,v) |E| or 2|E| in aggregate

Question

Suppose we run a full_bfs is called on a graph with no edges, does the comparison line gets executed?

A: Yes, many times

B: Yes, but once

C: No

D: I do not know :(

Question

Suppose `bfs` is called on an undirected graph using source node u.

If u is part of a connected component with nodes V_1 and edges E_1, what is the time complexity of the call to bfs?

A: $\Theta(E_1)$

B: $\Theta(V_1)$

C: $\Theta(E_1 + V_1)$

D: $\Theta(V_1 * E_1)$

E: Something else

Answer

- Time complexity: $\Theta(V_1 + E_1)$
- `bfs` explores all nodes and edges in the connected component.

Time Complexity of Full BFS

- `full_bfs` calls `bfs` **once** for **each** connected component.
- **Time complexity:**

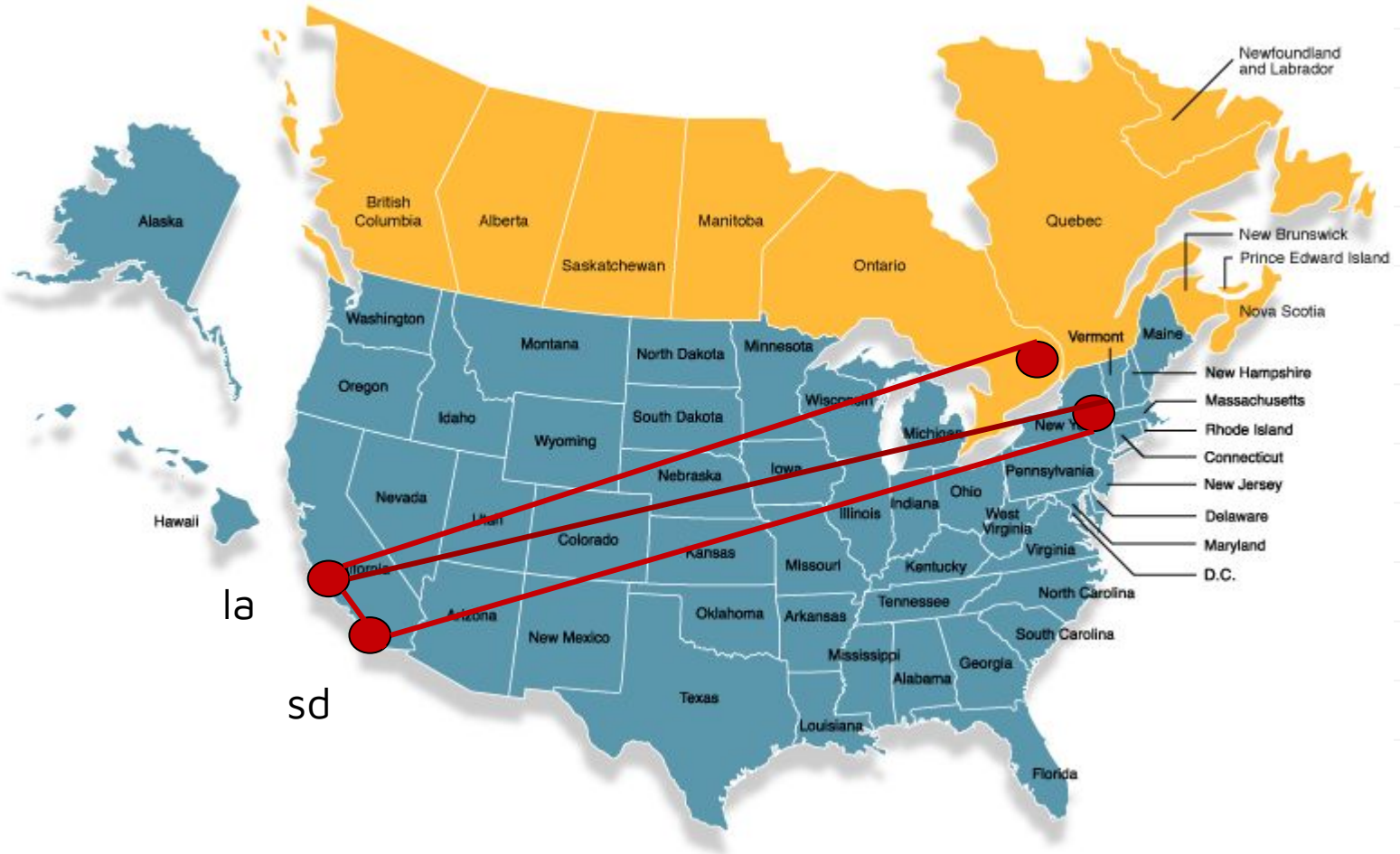
$$\Theta((V_1 + E_1) + (V_2 + E_2) + \ldots + (V_k + E_k))$$
$$= \Theta((V_1 + V_2 + \ldots + V_k) + (E_1 + E_2 + \ldots + E_k))$$
$$= \Theta(V + E)$$

Time Complexity

- Full BFS takes $\Theta(V + E)$
- Why not just $\Theta(E)$?
- $\Theta(V + E)$ works for *all graphs*.
  - If we know more about the number of edges, we might be able to simplify.
  - E.g., if the graph is **complete**, $E = \Theta(V^2)$, so time complexity is $\Theta(V + V^2) = \Theta(V^2)$.

Shortest Paths

Example



la

sd

Recall

- The **length** of a path is ?

Recall

- The **length** of a path is (# of nodes) – 1

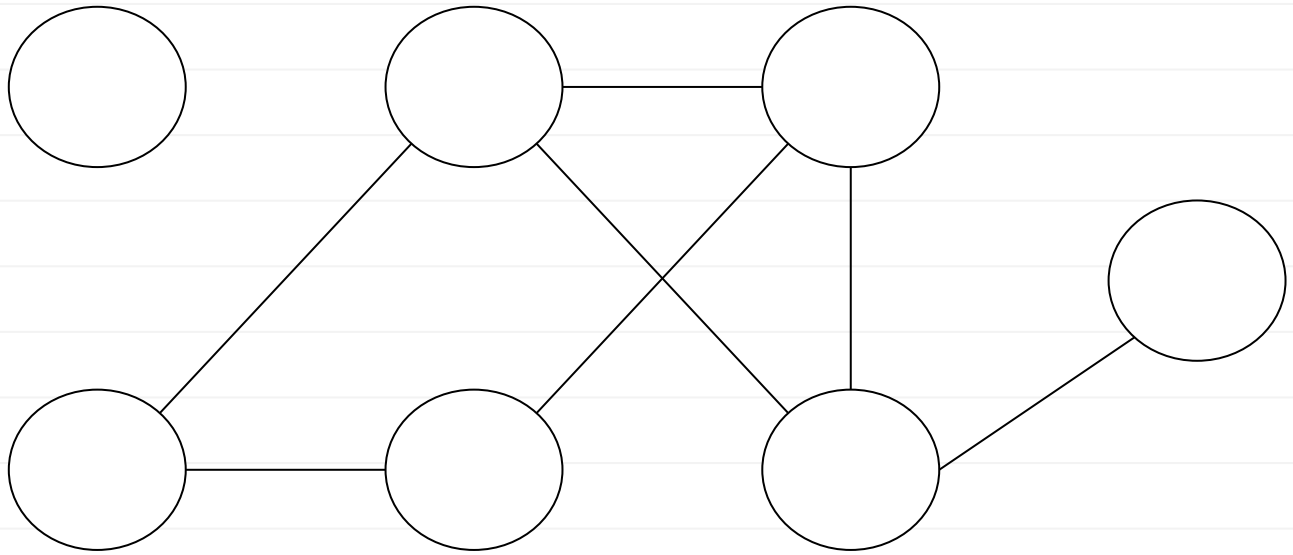
Definitions

- A **shortest path** between u and v is a path between u and v with **smallest** possible length.
  - There may be several, or none at all.
- The **shortest path distance** is the length of a shortest path.
  - Convention: ∞ if no path exists.
  - "the distance between u and v" means *spd*.

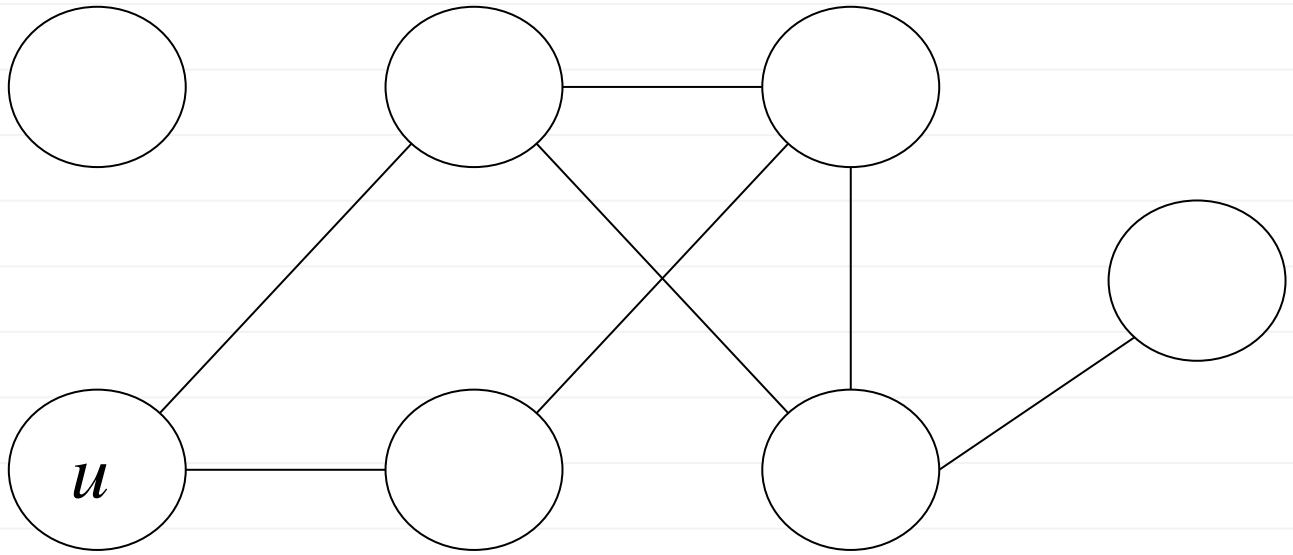
Today: Shortest Paths

- **Given**: directed/undirected graph G, source u
- **Goal**: find shortest path from u to **every other node**.

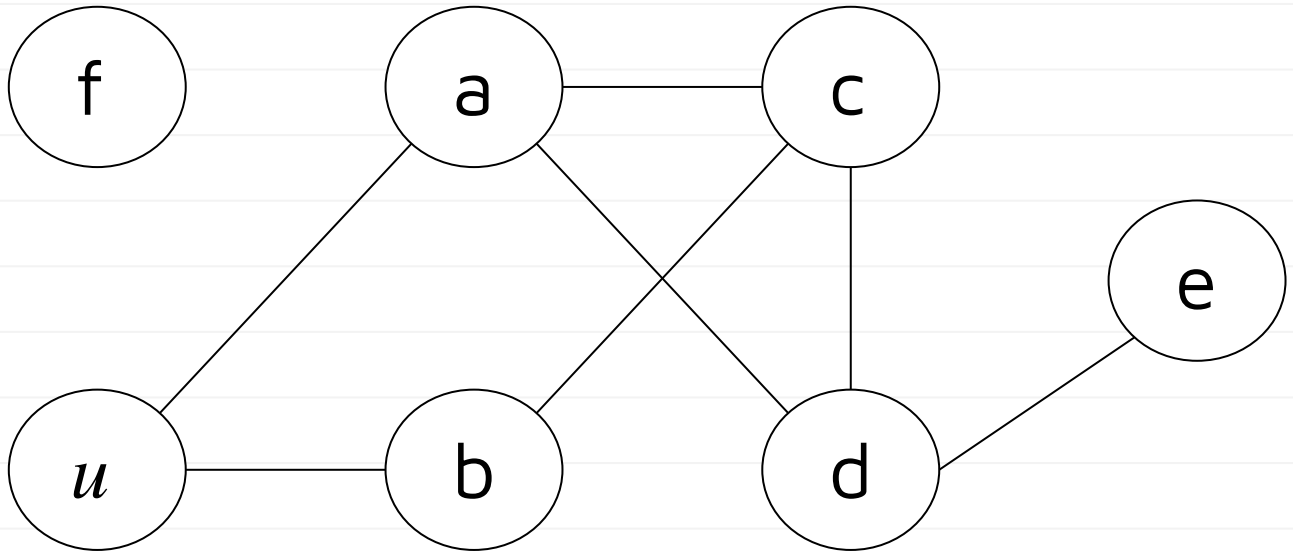
Example



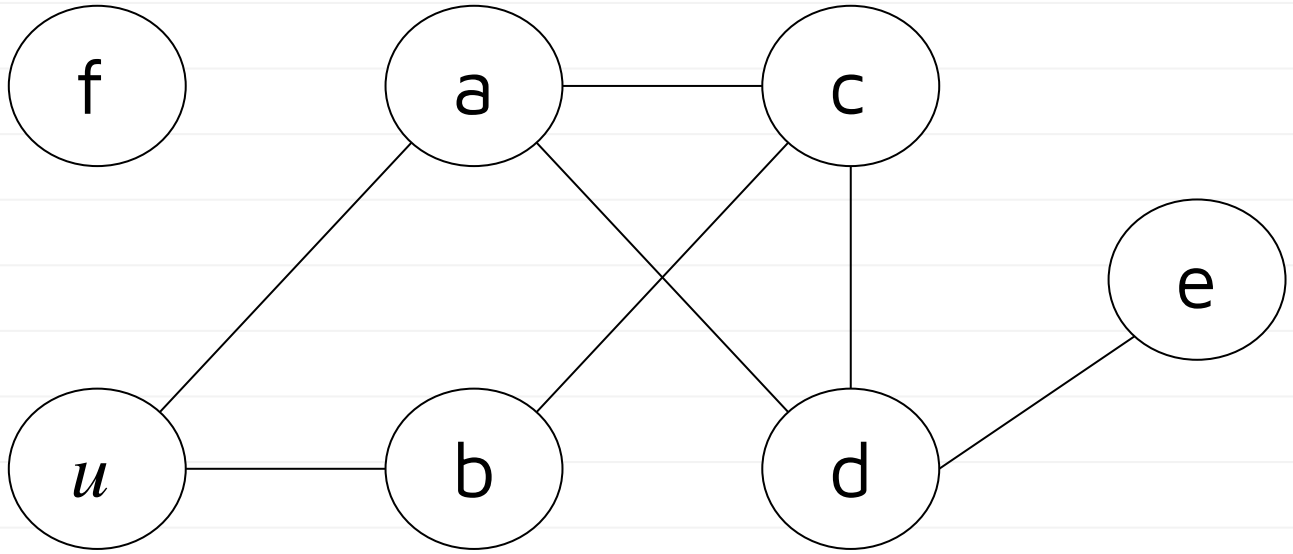
Example



Example

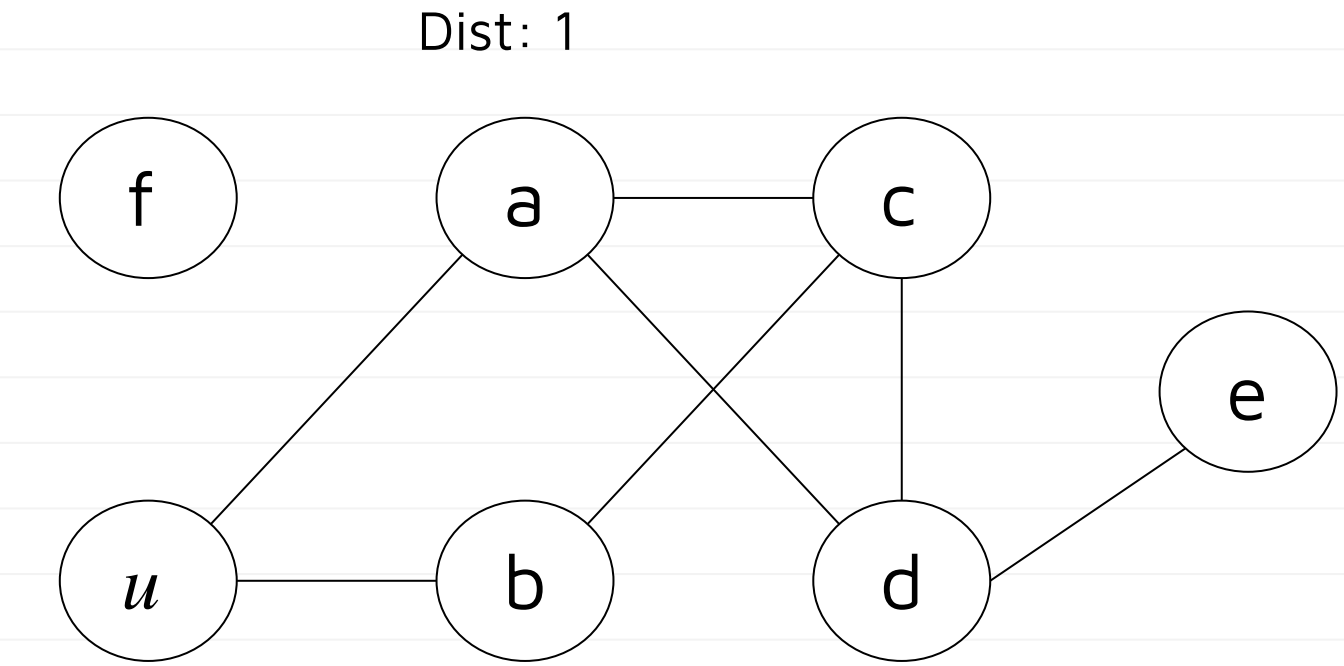


Example

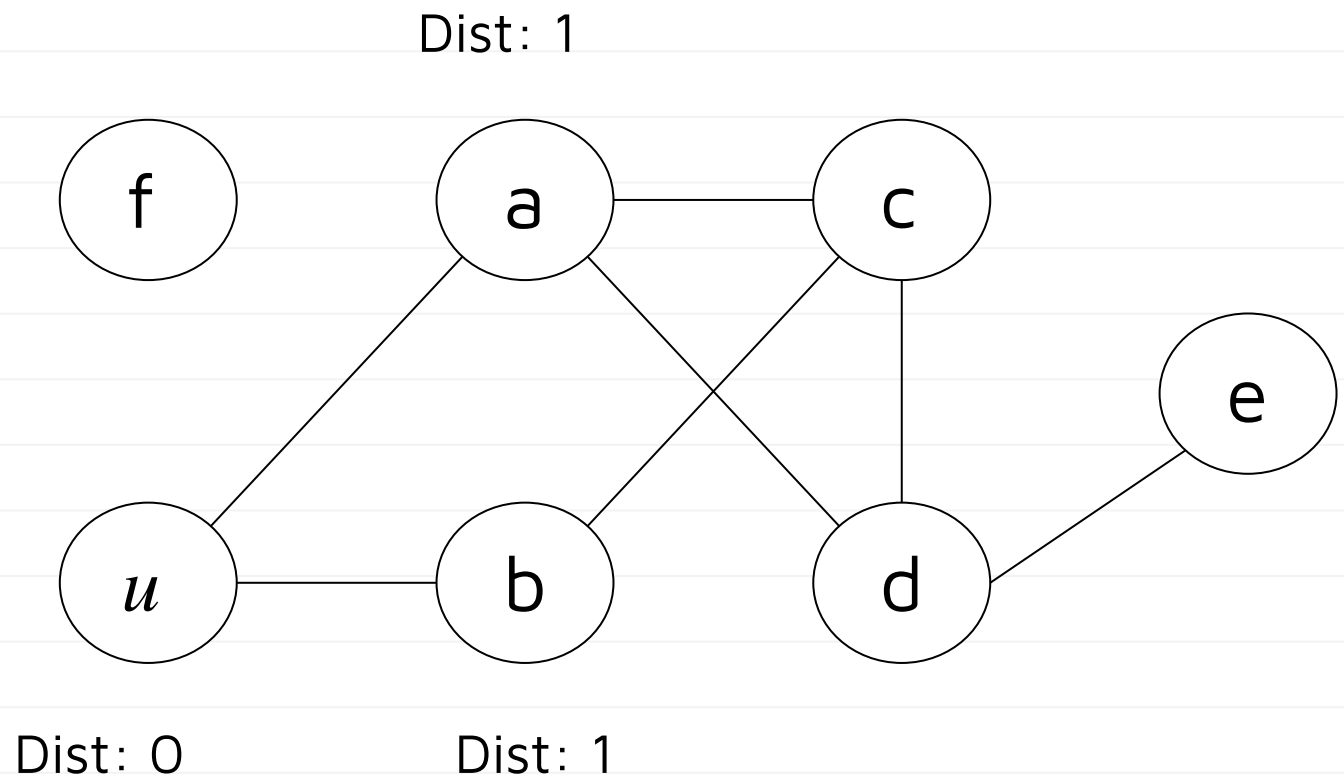


Dist: 0

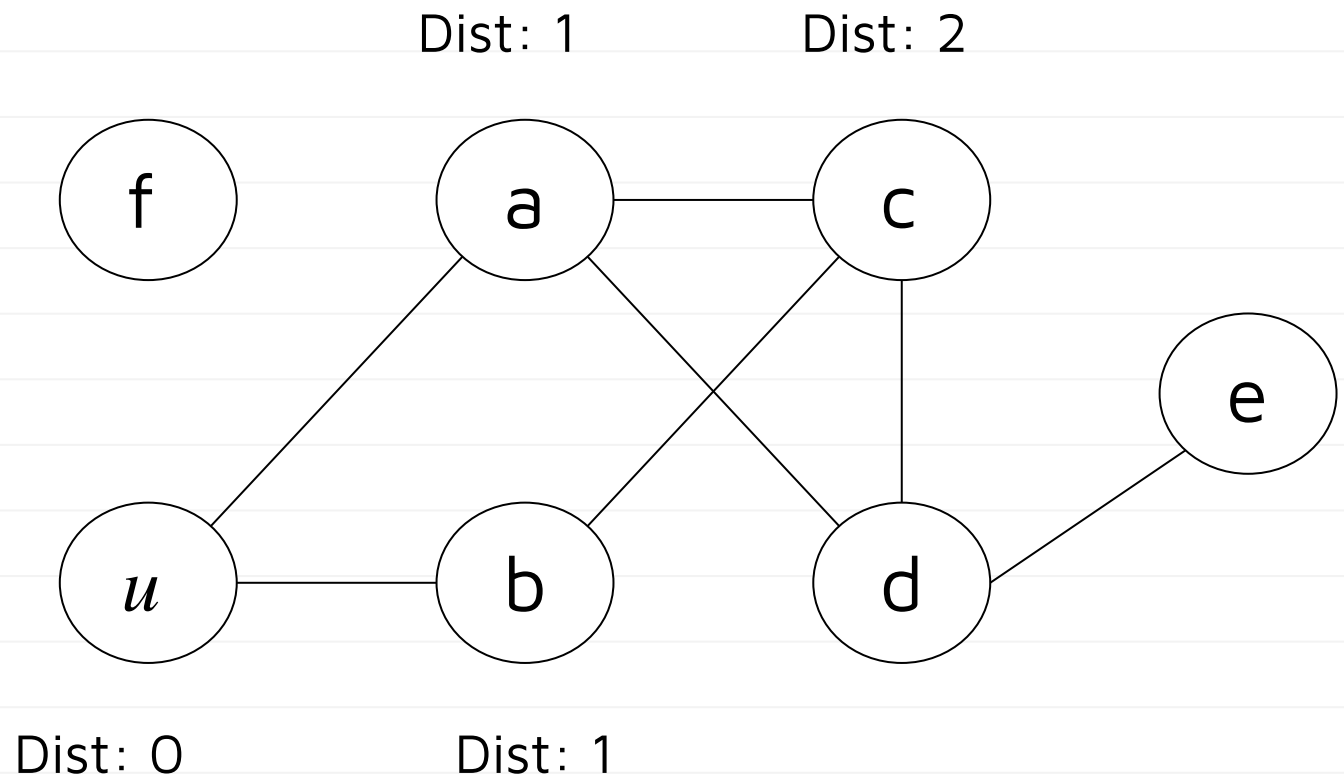
Example



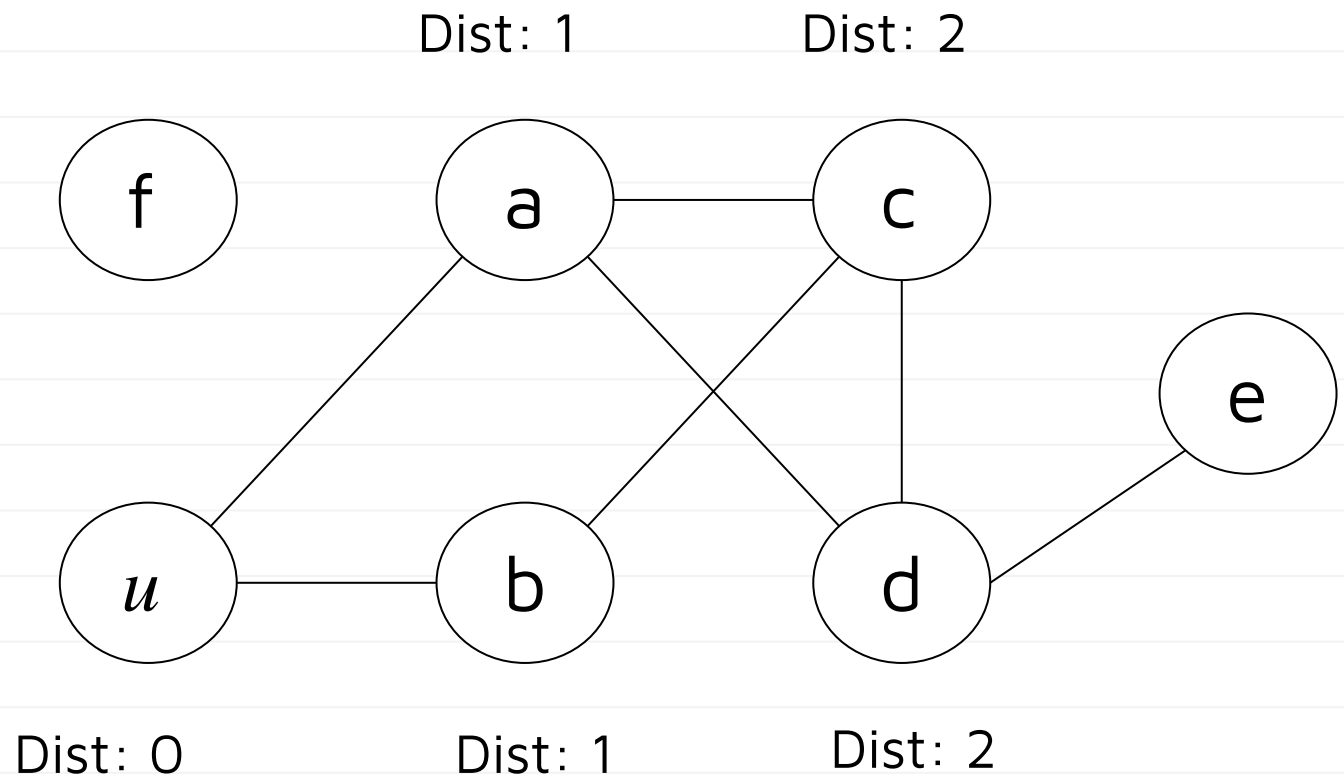
Example



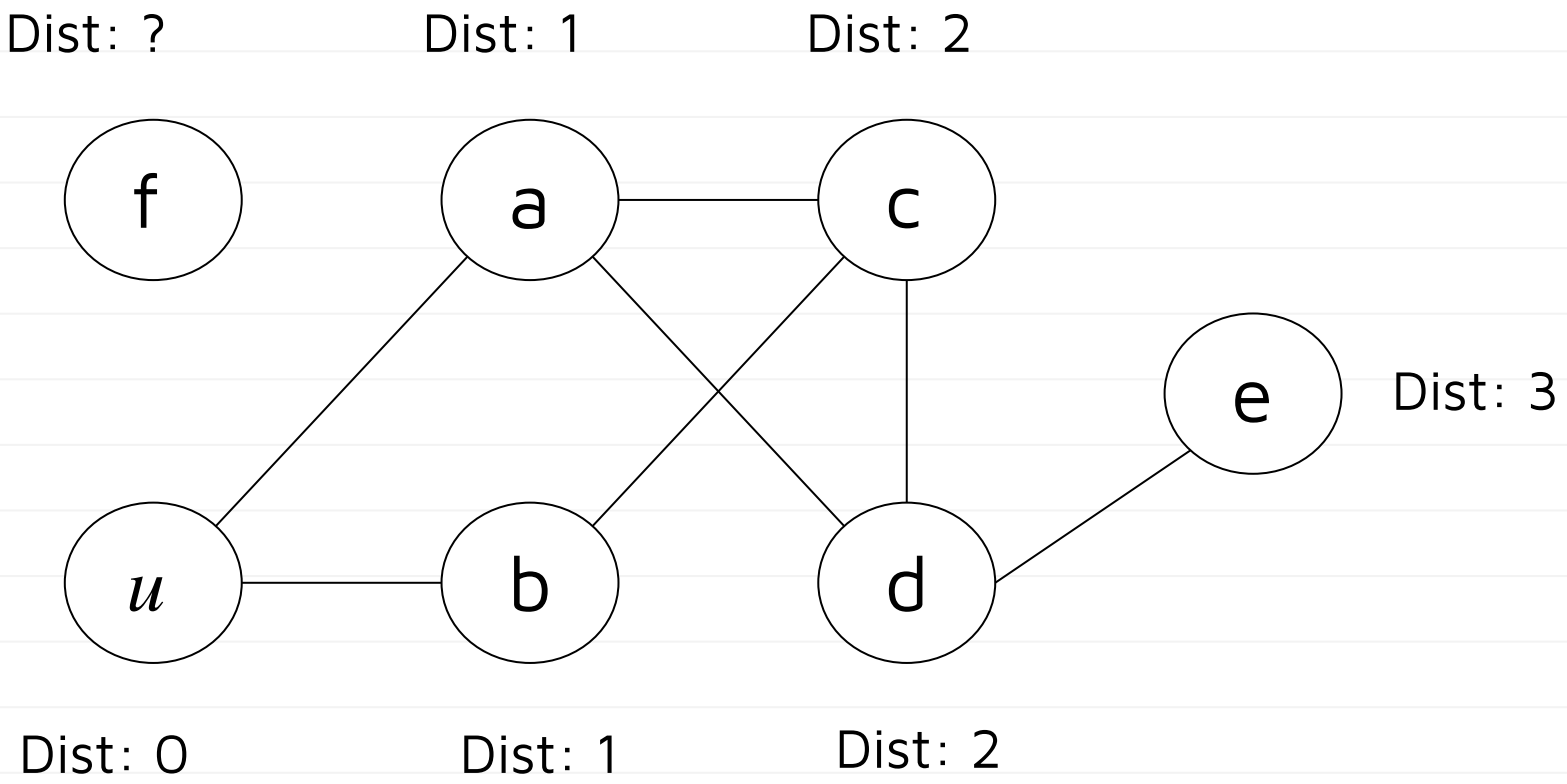
Example



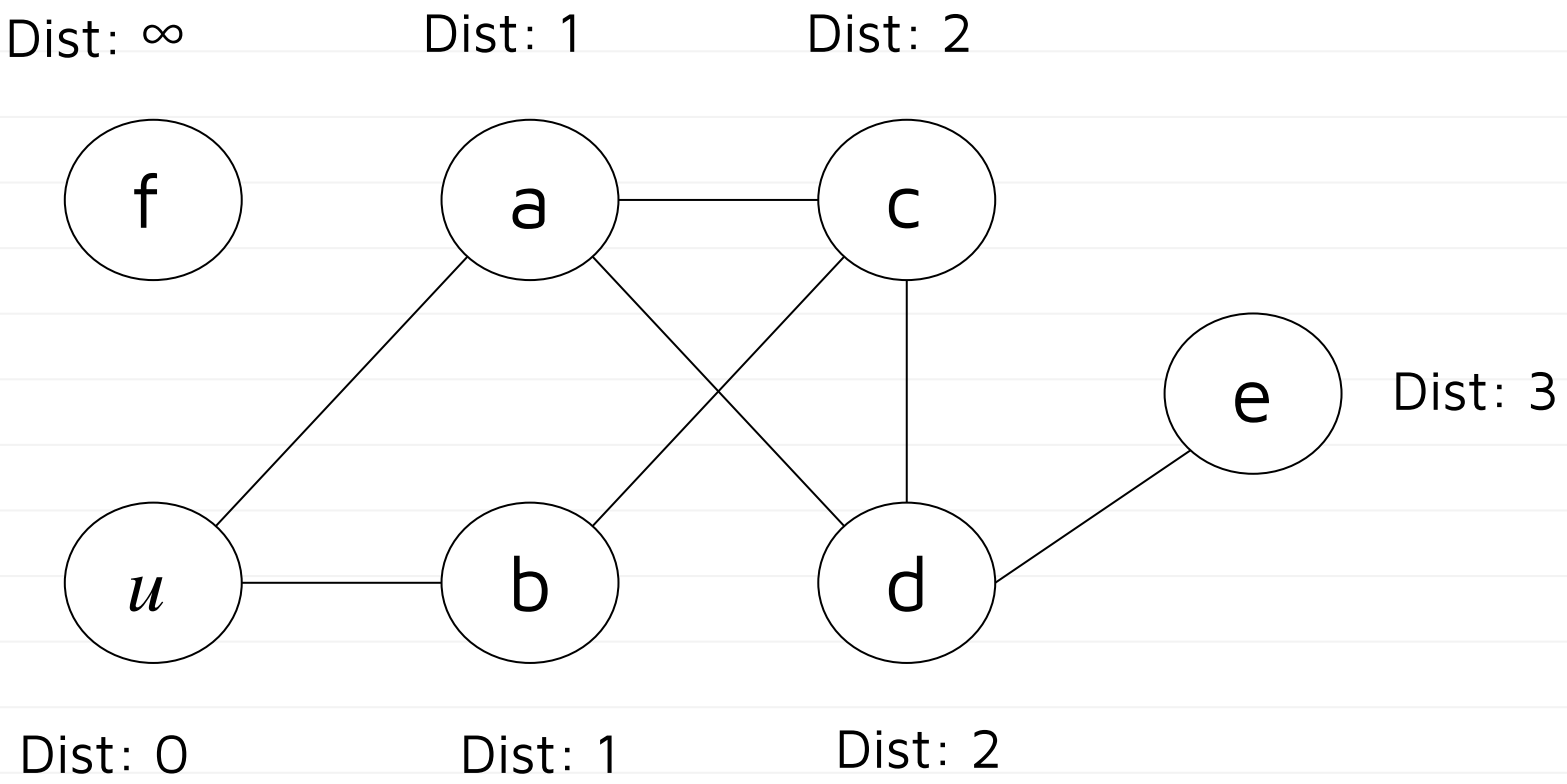
Example



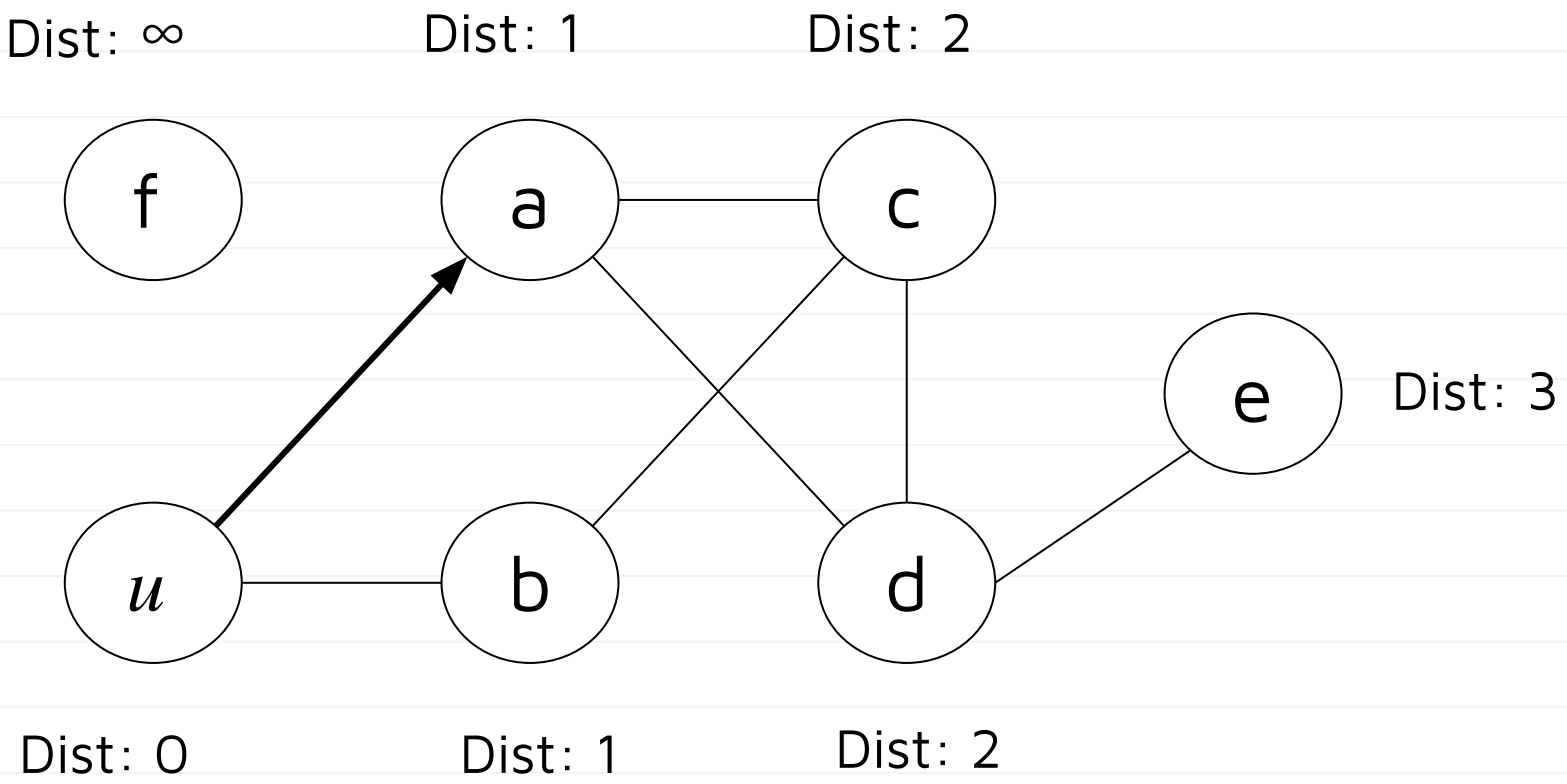
Example



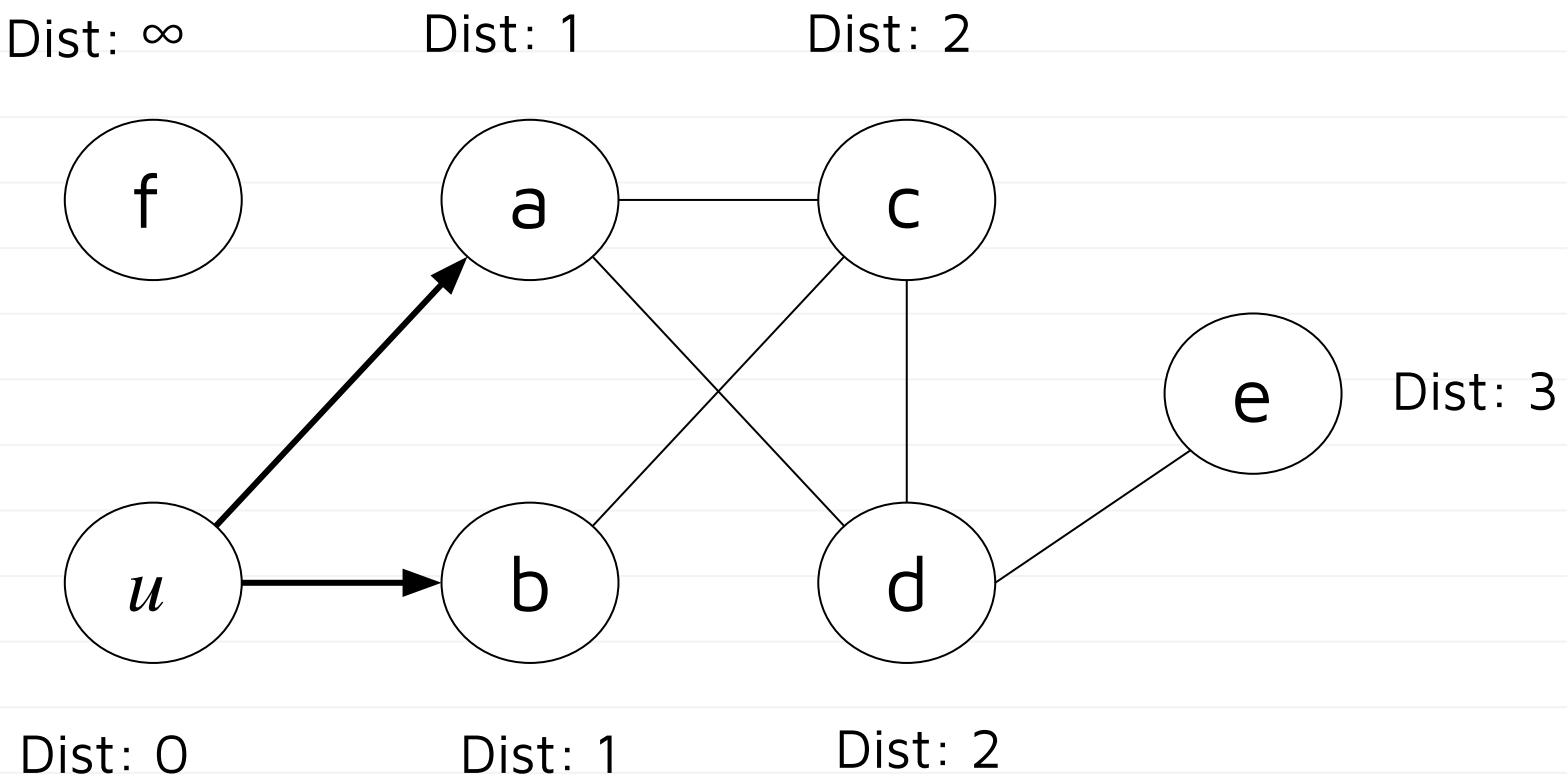
Example



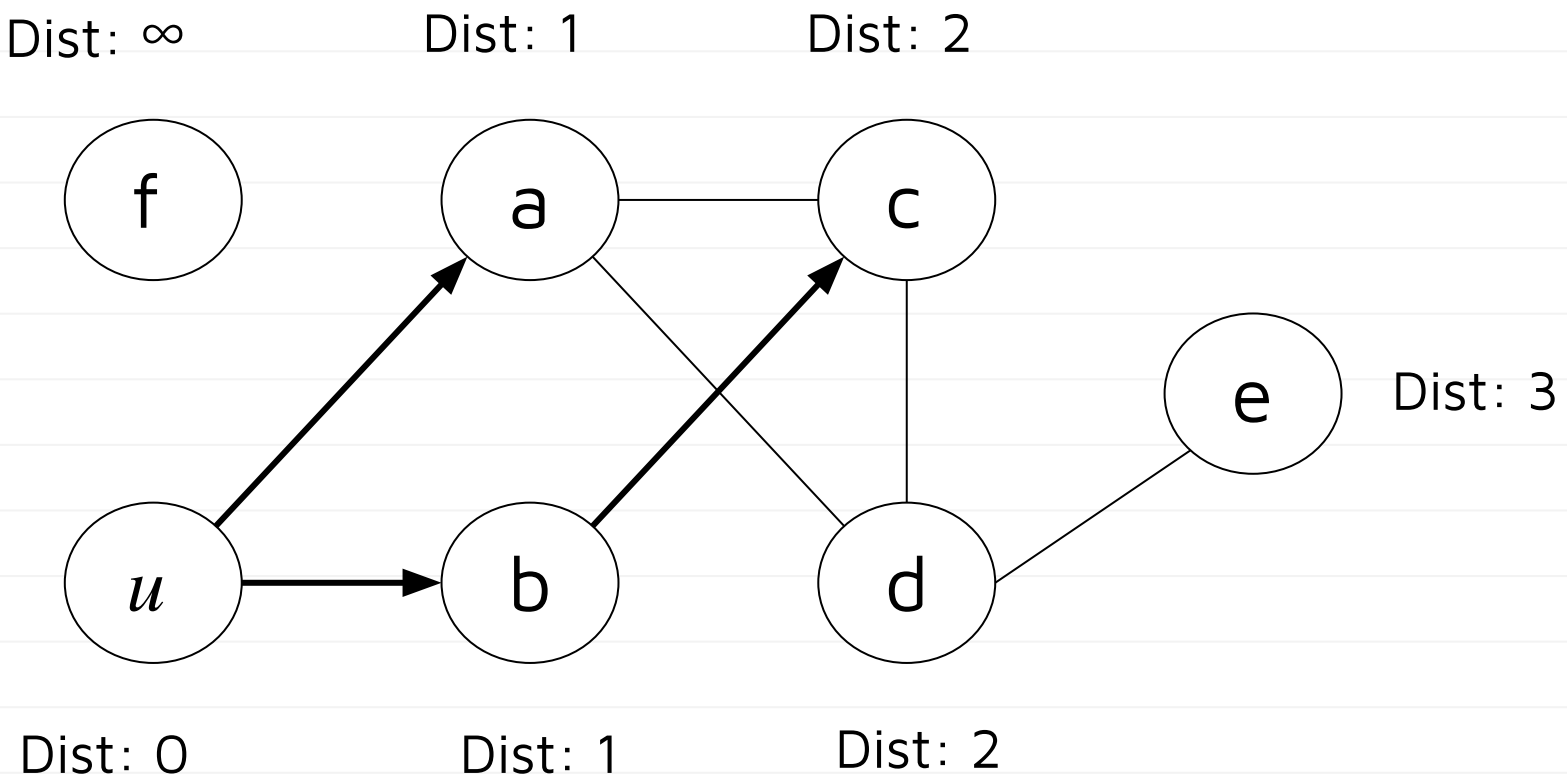
Example



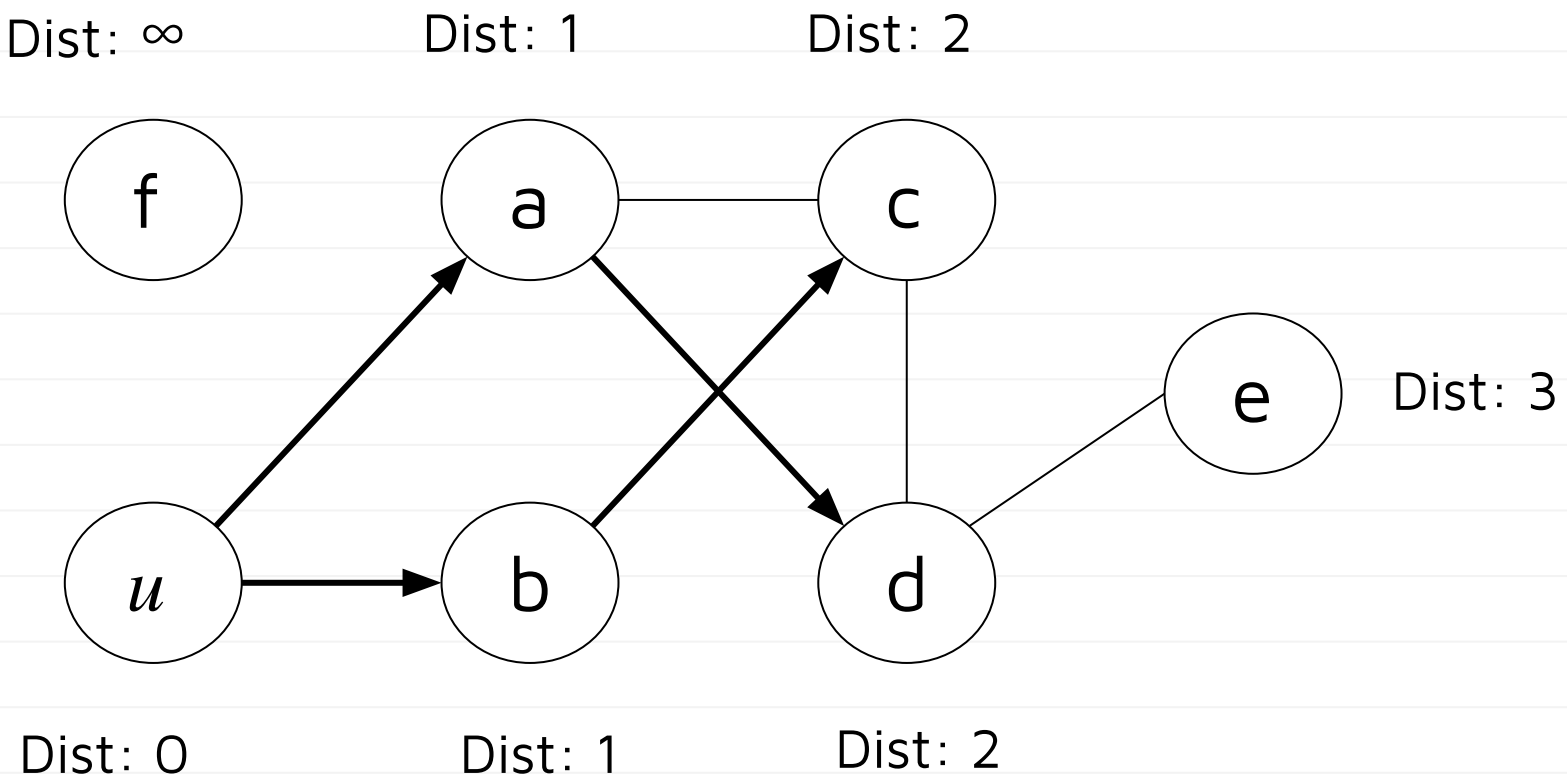
Example



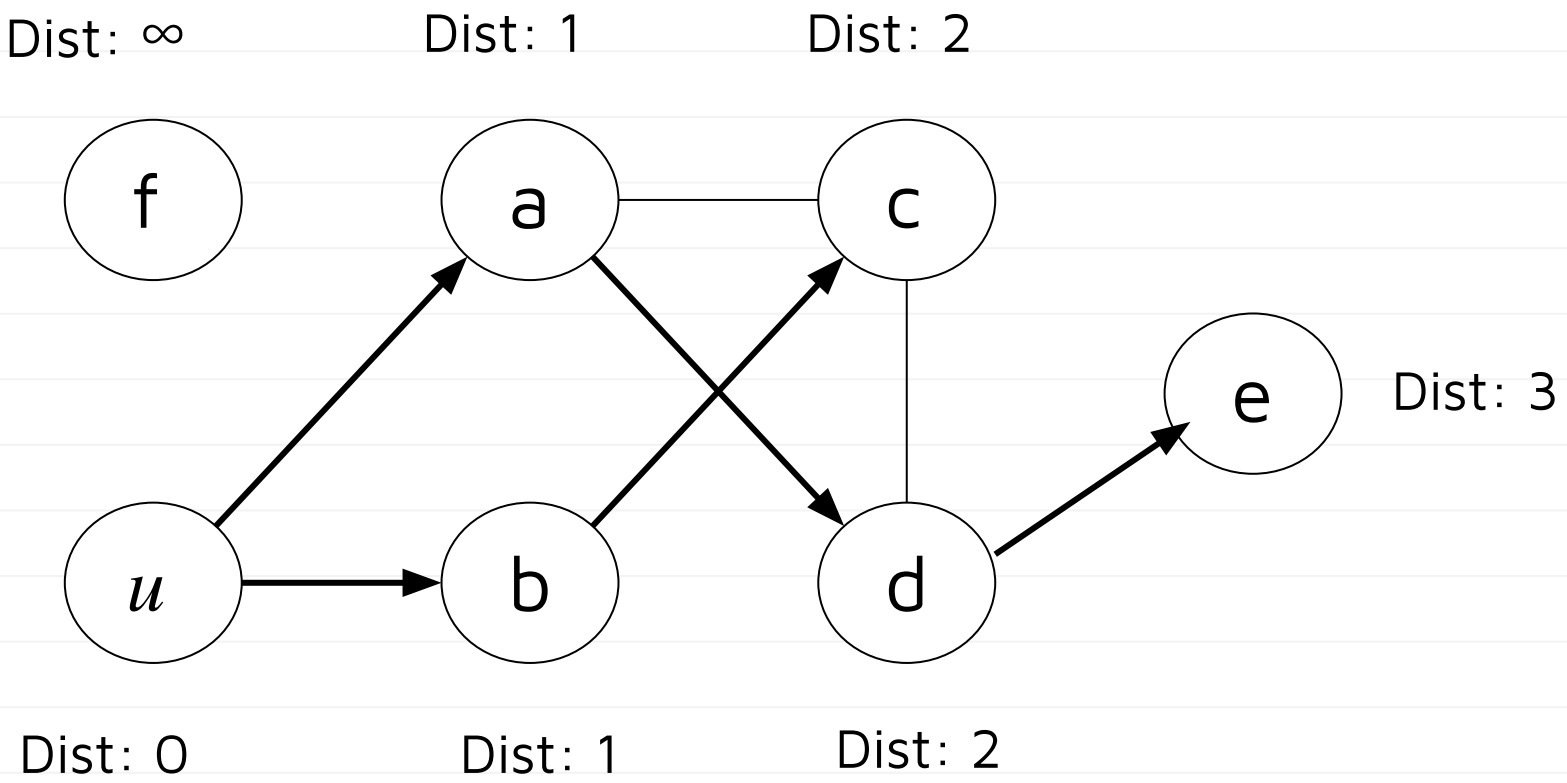
Example



Example



Example



Key Property of Shortest Paths

- *Suppose* you have shortest path from u to v.

Key Property of Shortest Paths

- *Suppose* you have shortest path from u to v.
- *Suppose* it goes through the edge (x, v).
  - x is only 1 edge away from v.

Key Property of Shortest Paths

- *Suppose* you have shortest path from u to v.
- *Suppose* it goes through the edge (x, v).
  - x is only 1 edge away from v.
- Then the part of that path from u to x is a **shortest** path.

Key Property, Restated

- A shortest path of length k is composed of:
  - A **shortest path** of length k - 1
  - Plus one edge

Question

Node v has three neighbors: a, b, and c. The distance from:

- u to a is 5.
- u to b is 3.
- u to c is 7.

What is the distance from u to v?

A: 6

B: 3

C: 4

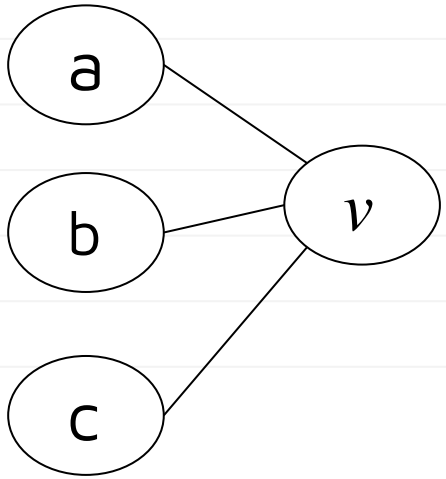
D: 8

E: Not enough info

Node v has three neighbors: a, b, and c. The distance from:

- u to a is 5.
- u to b is 3.
- u to c is 7.

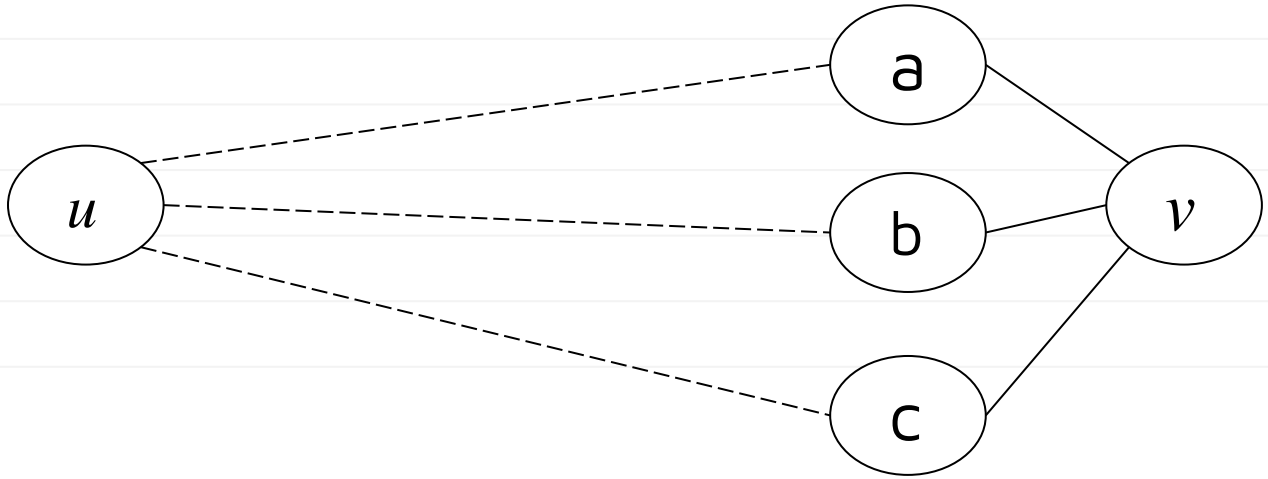
What is the distance from u to v?



Node v has three neighbors: a, b, and c. The distance from:

- u to a is 5.
- u to b is 3.
- u to c is 7.

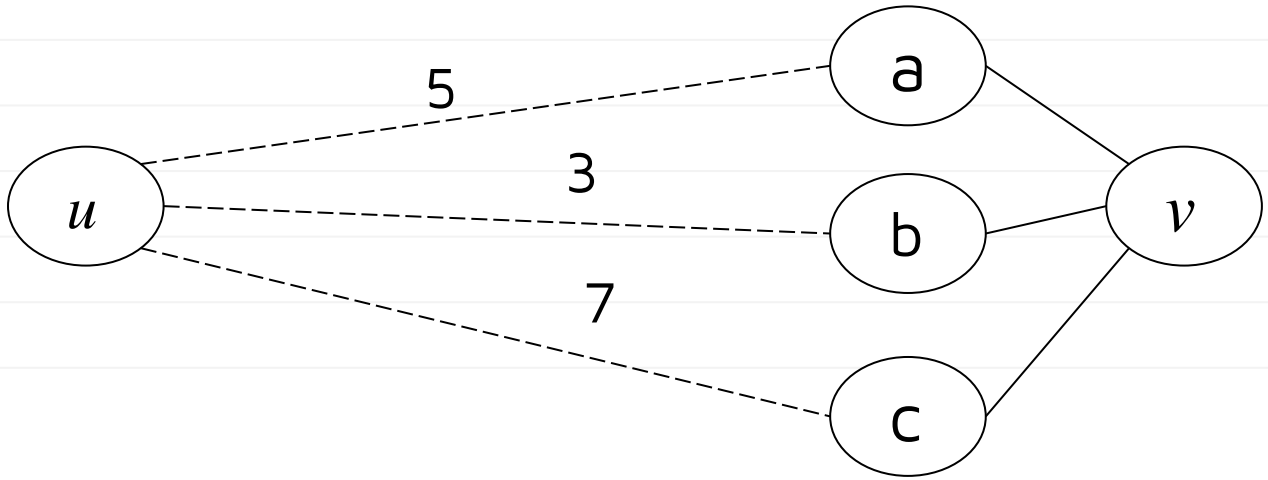
What is the distance from u to v?



Node v has three neighbors: a, b, and c. The distance from:

- u to a is 5.
- u to b is 3.
- u to c is 7.

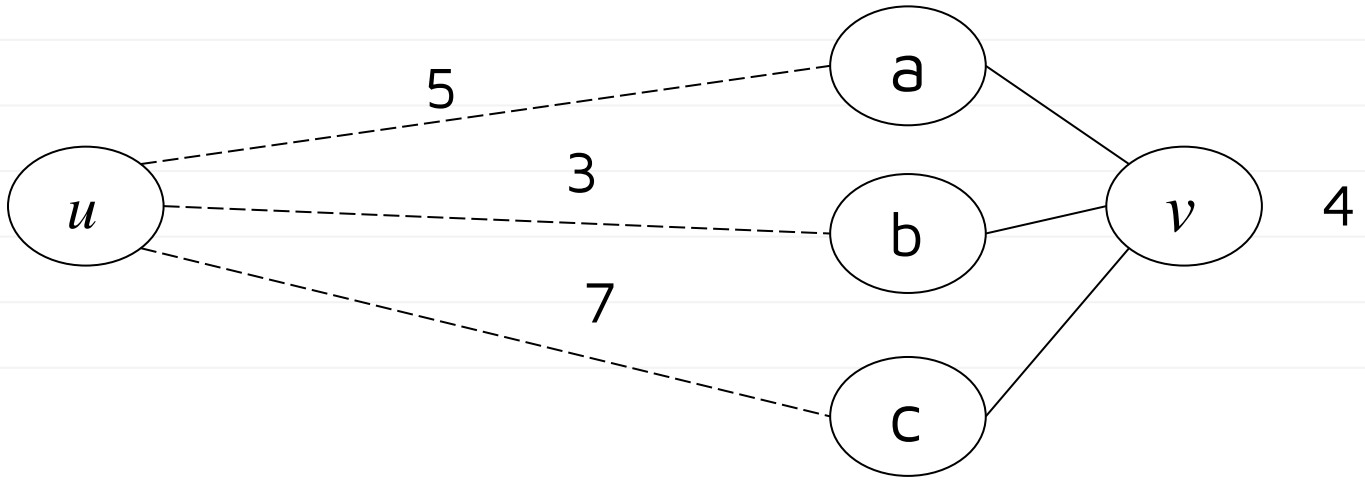
What is the distance from u to v?



Node v has three neighbors: a, b, and c. The distance from:

- u to a is 5.
- u to b is 3.
- u to c is 7.

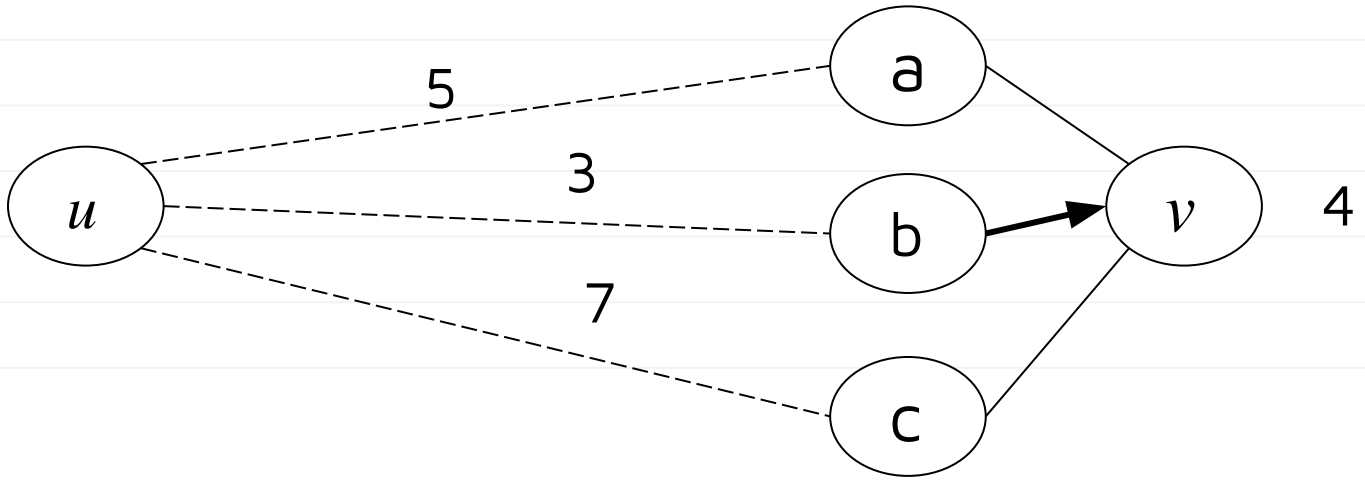
What is the distance from u to v?



Node v has three neighbors: a, b, and c. The distance from:

- u to a is 5.
- u to b is 3.
- u to c is 7.

What is the distance from u to v?

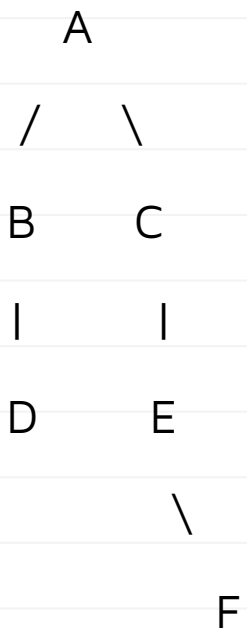


Algorithm Idea

- Find all nodes *distance 1* from source.
- Use these to find all nodes *distance 2* from source.
- Use these to find all nodes *distance 3* from source.
- …..

It turns out...

...this is exactly what BFS does.



Thank you!

Do you have any questions?

CampusWire!