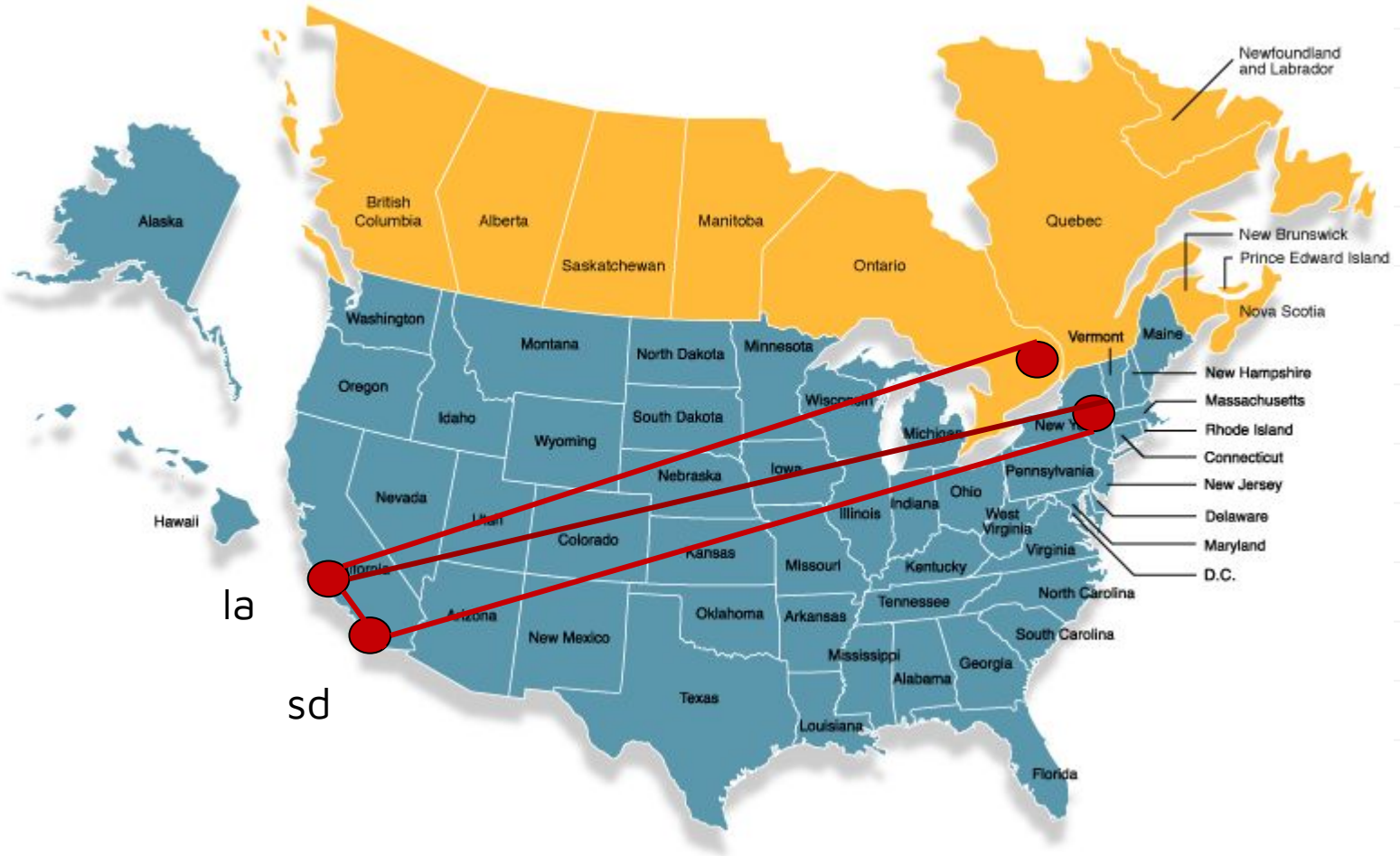
DSC 40B

Lecture 21 :

Shortest Path

Shortest Paths

Example



la

sd

Recall

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

Recall

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

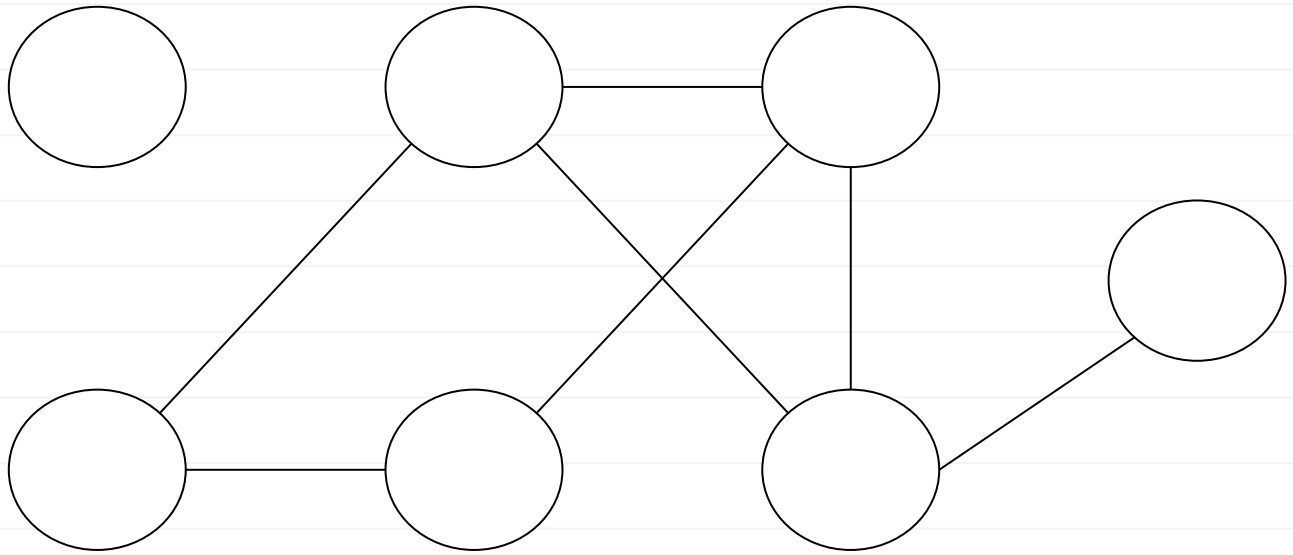
Definitions

- 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*.

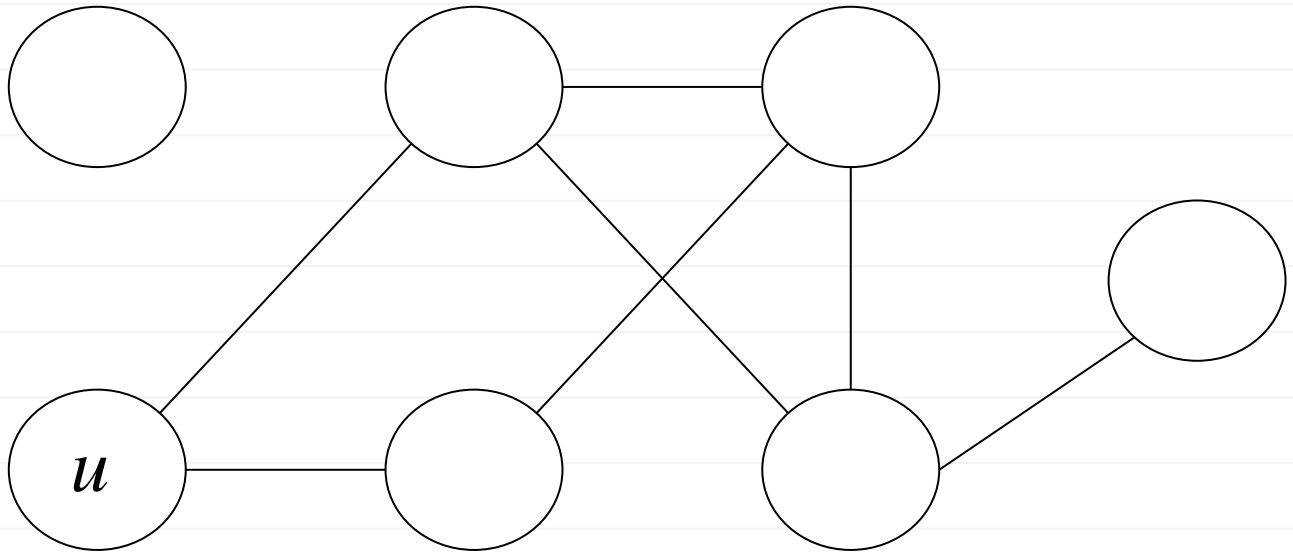
Today: Shortest Paths

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

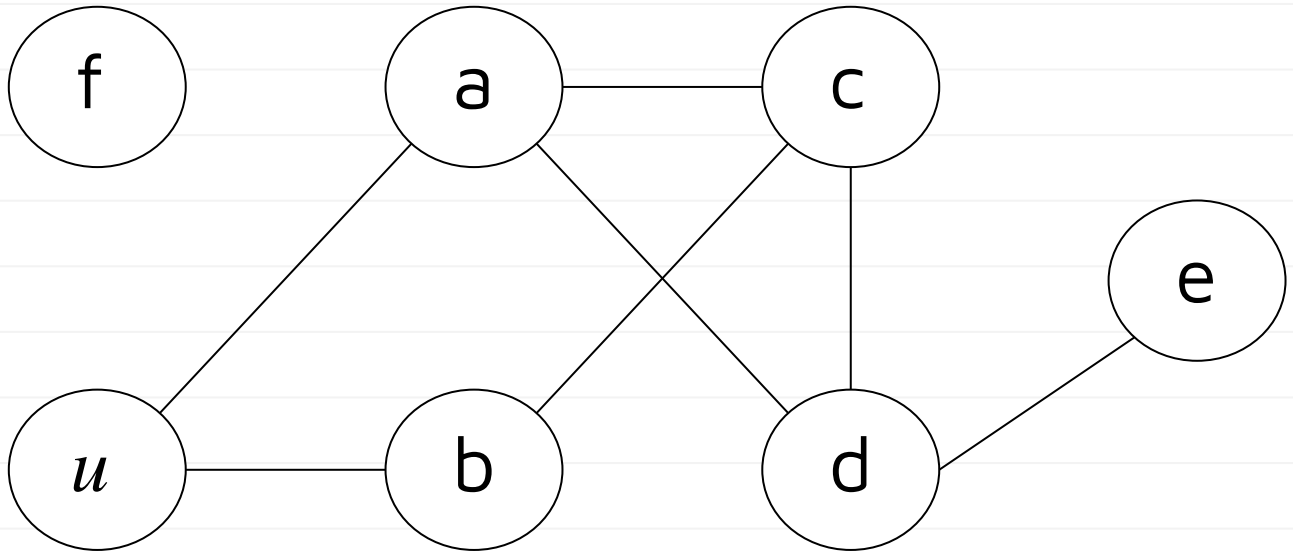
Example



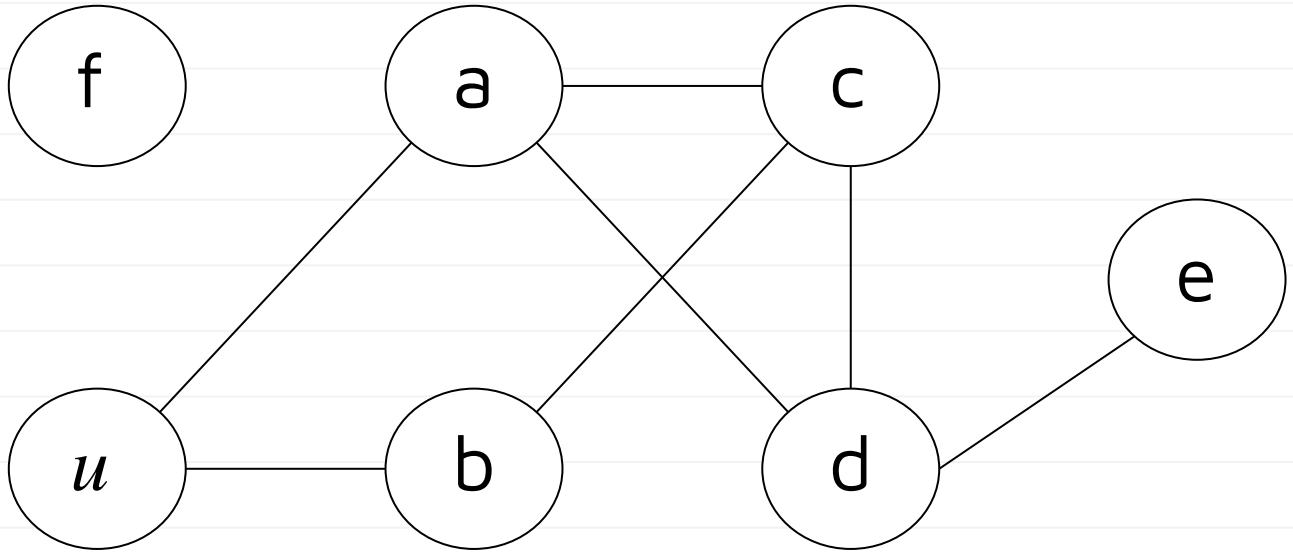
Example



Example

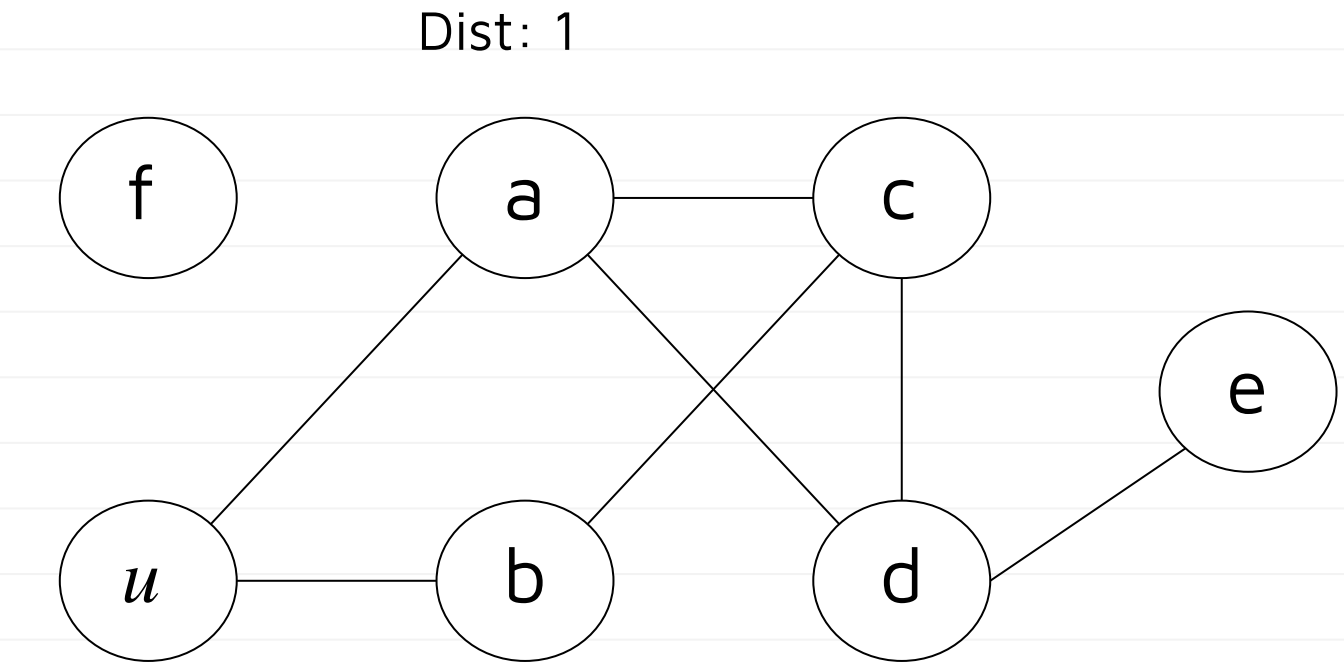


Example

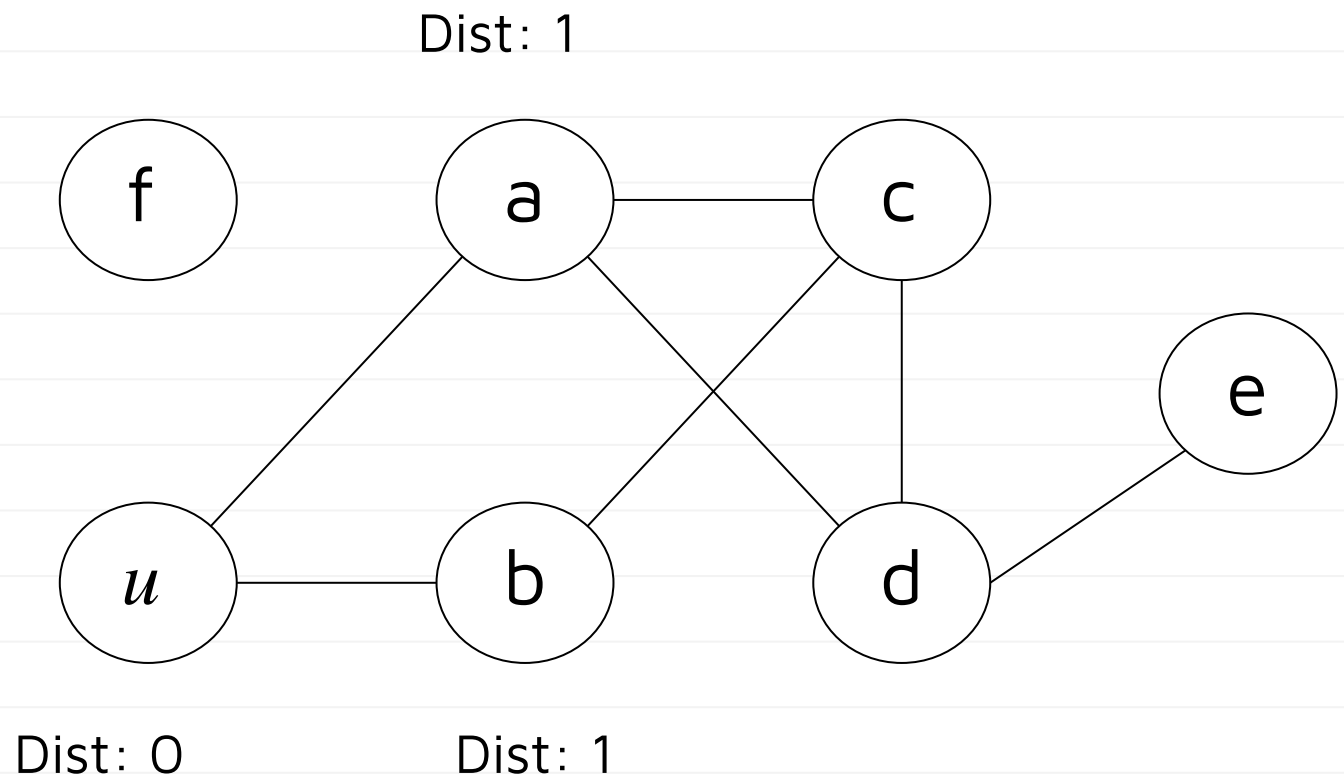


Dist: 0

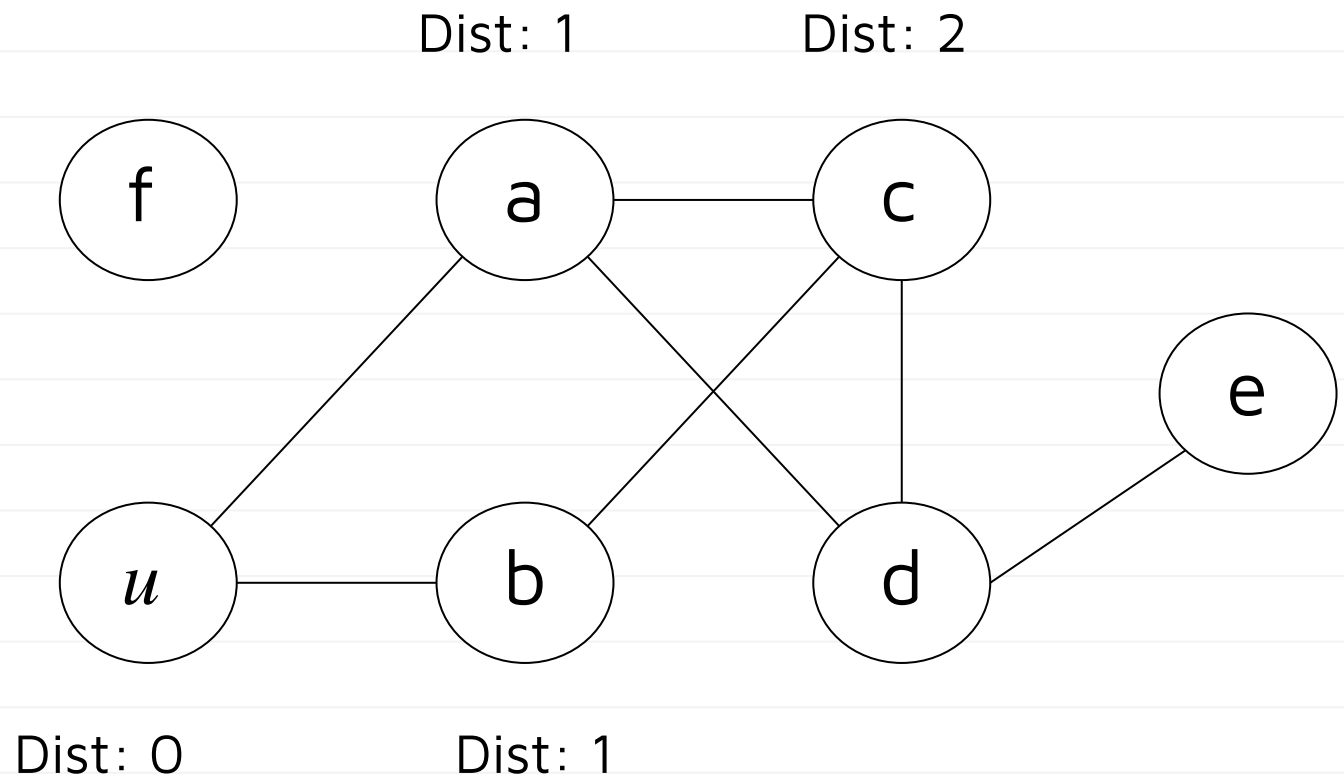
Example



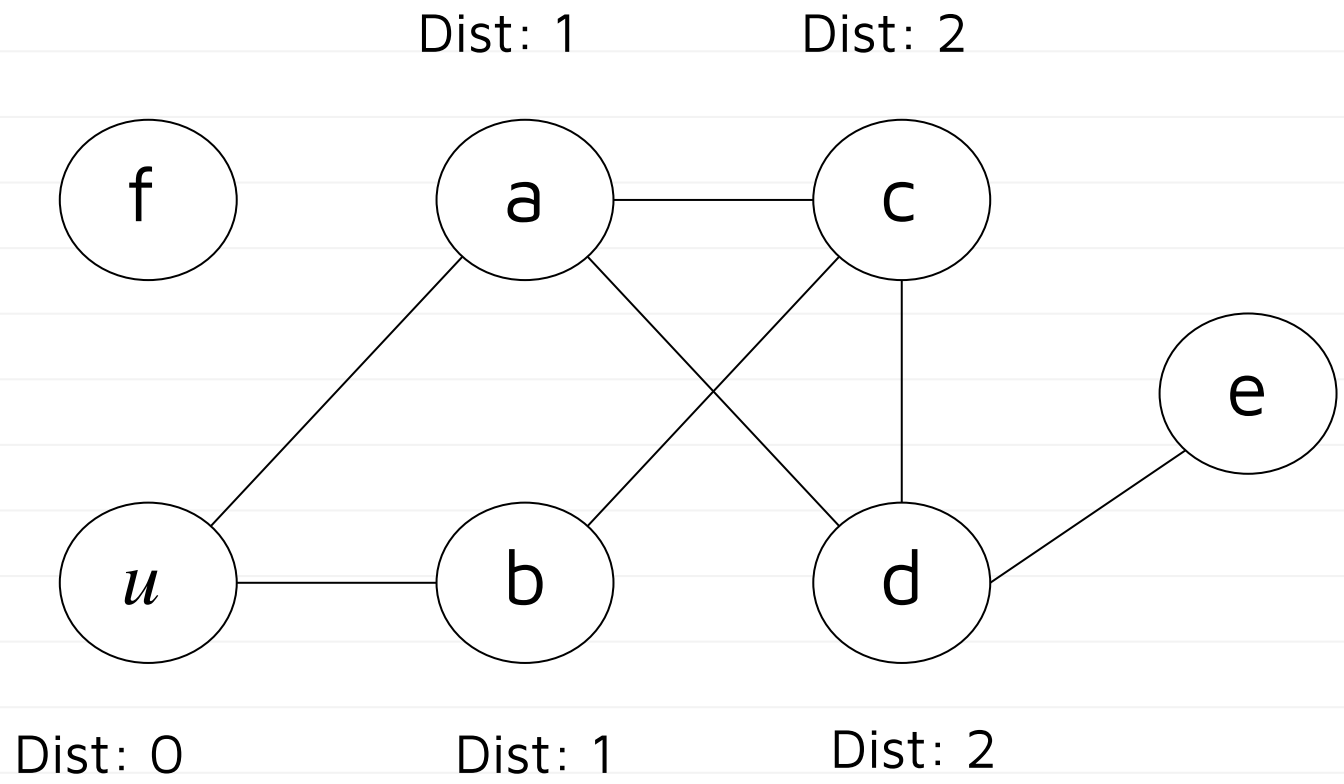
Example



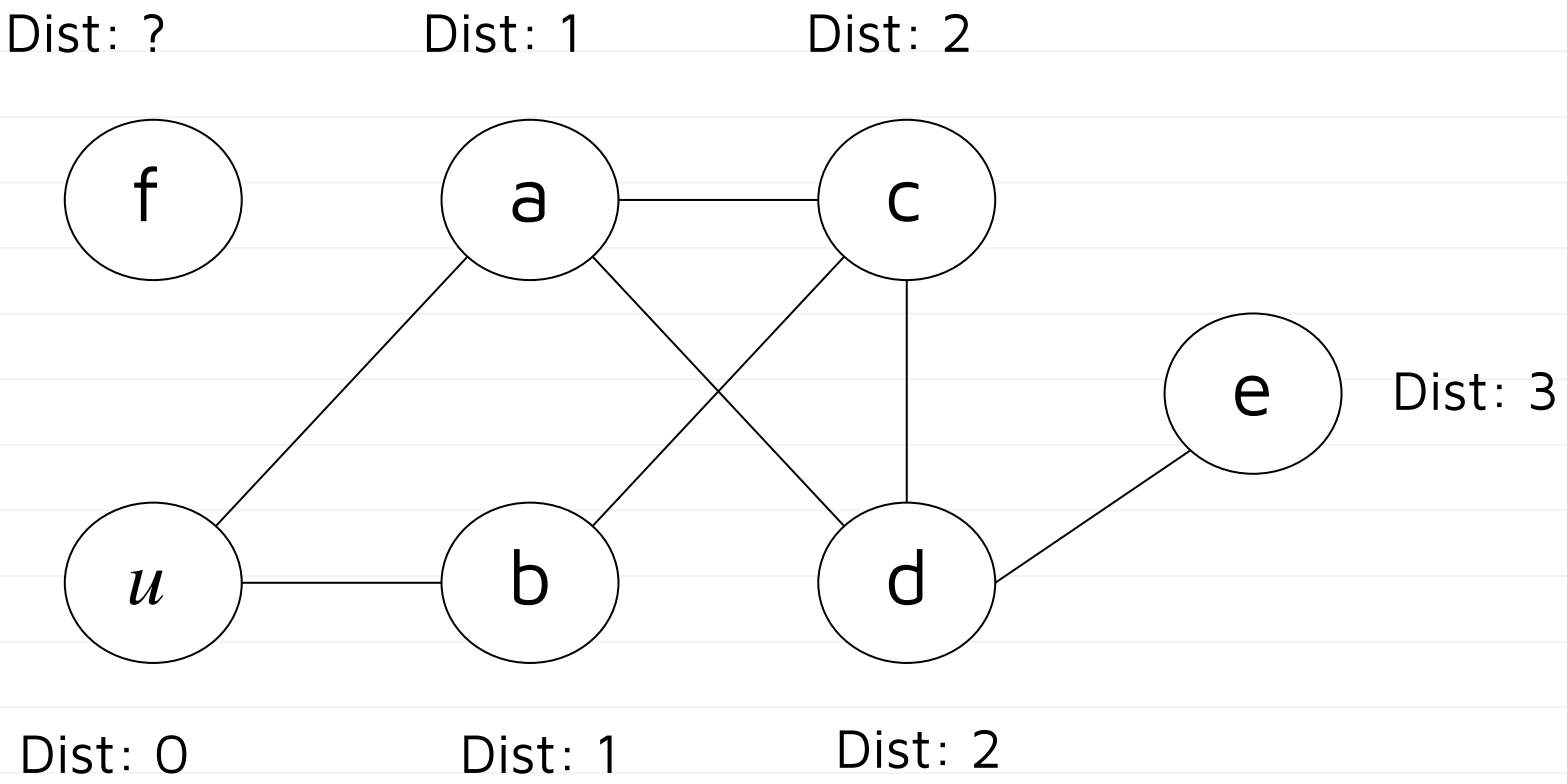
Example



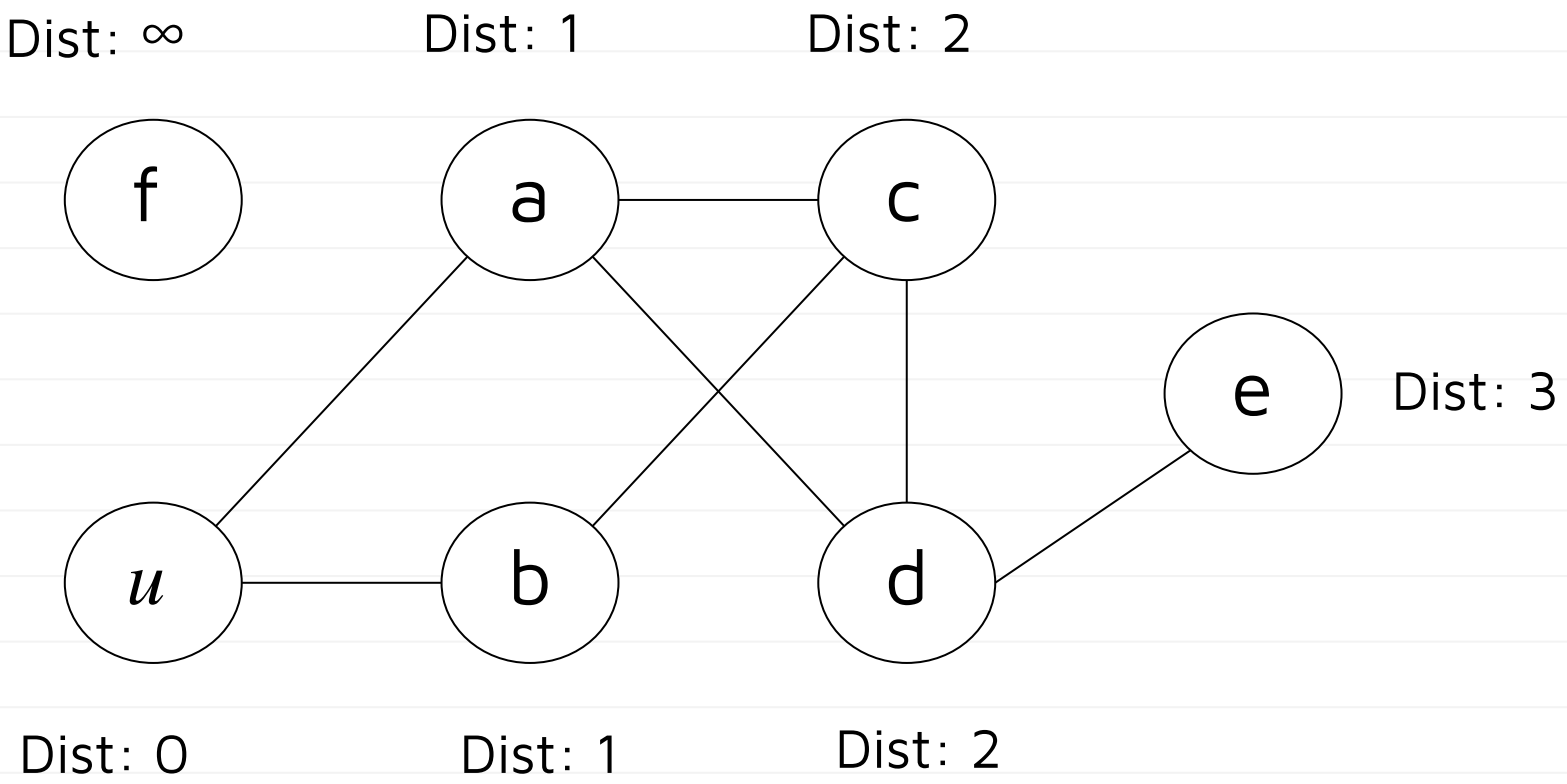
Example



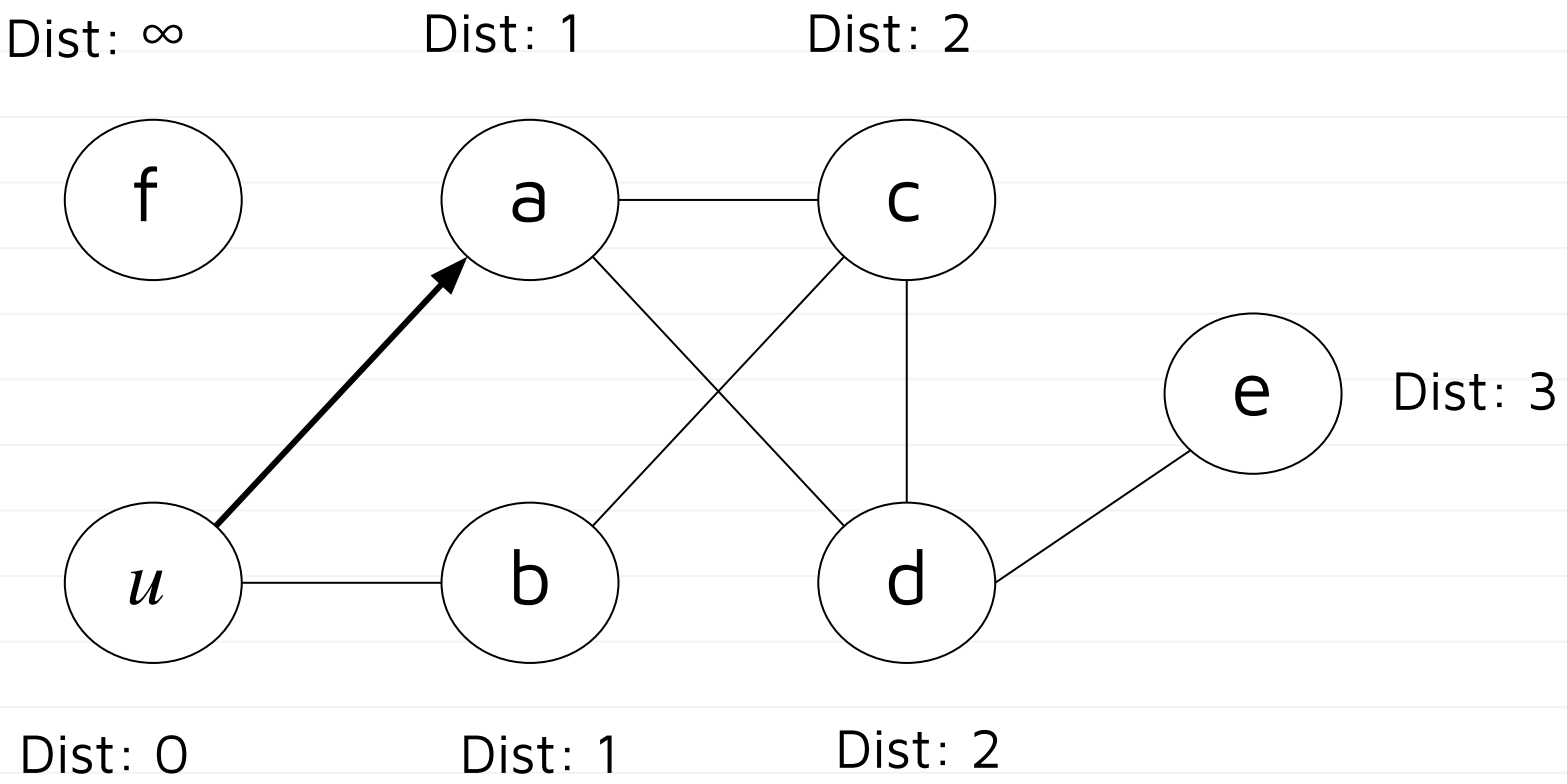
Example



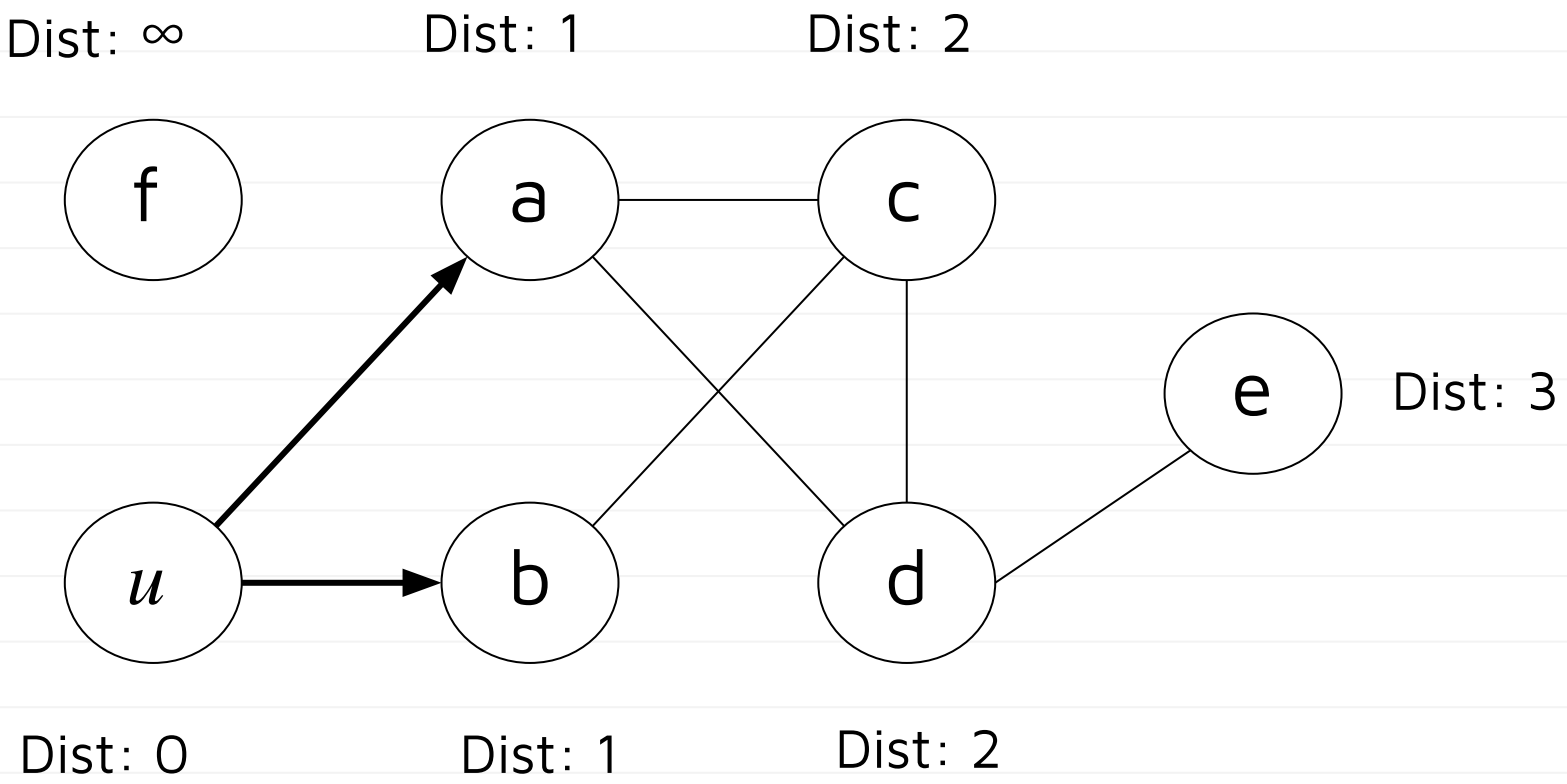
Example



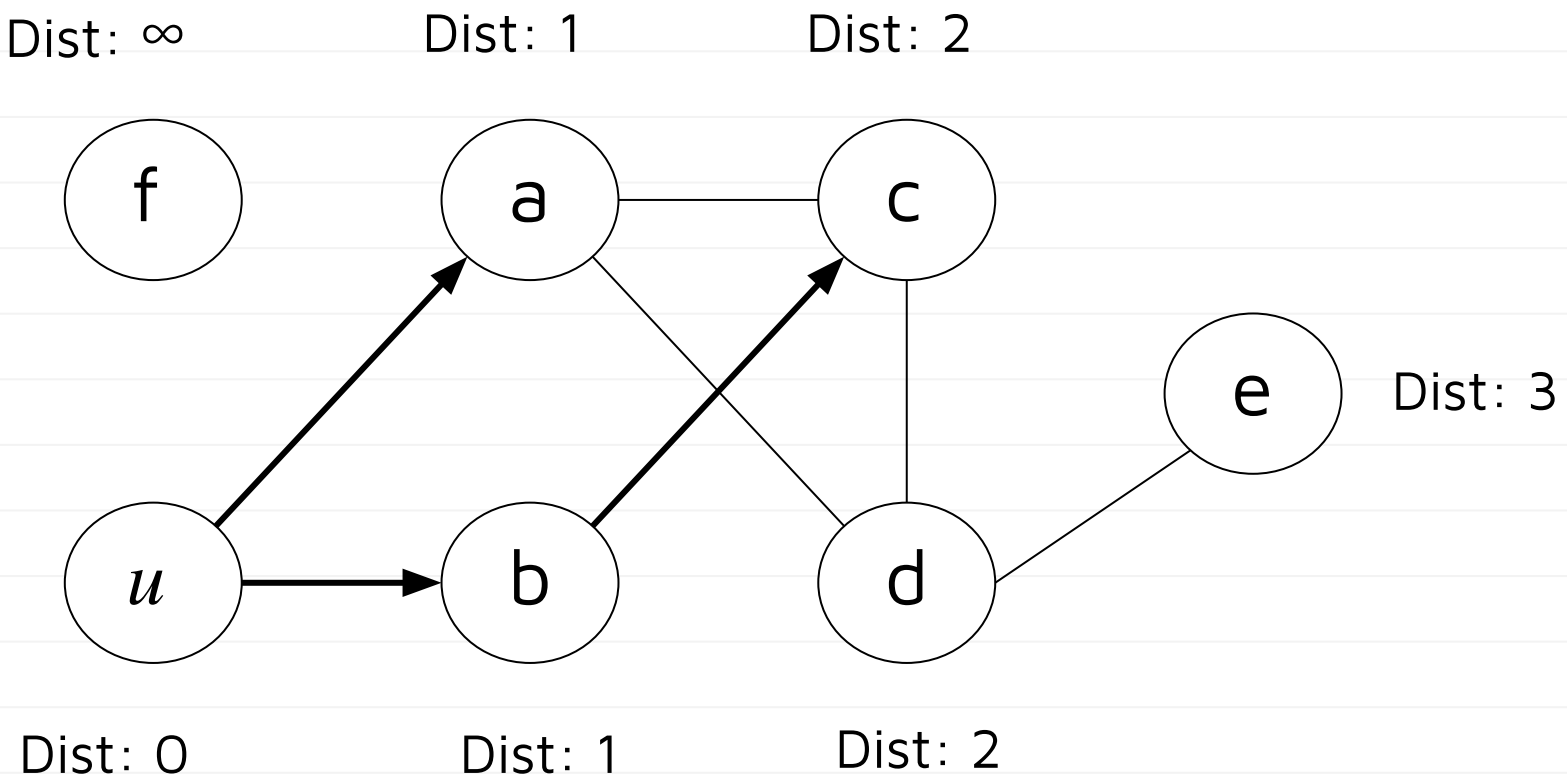
Example



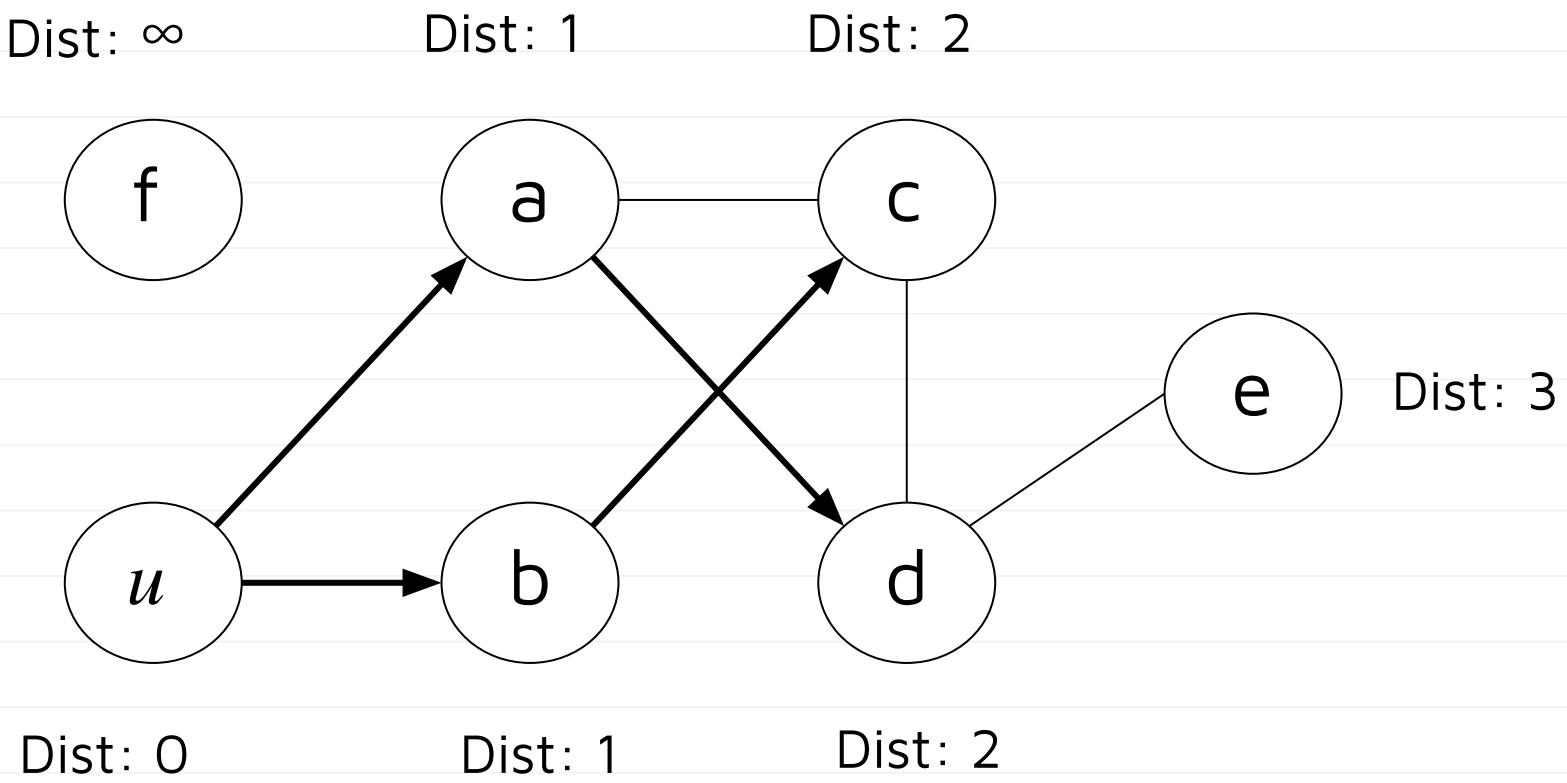
Example



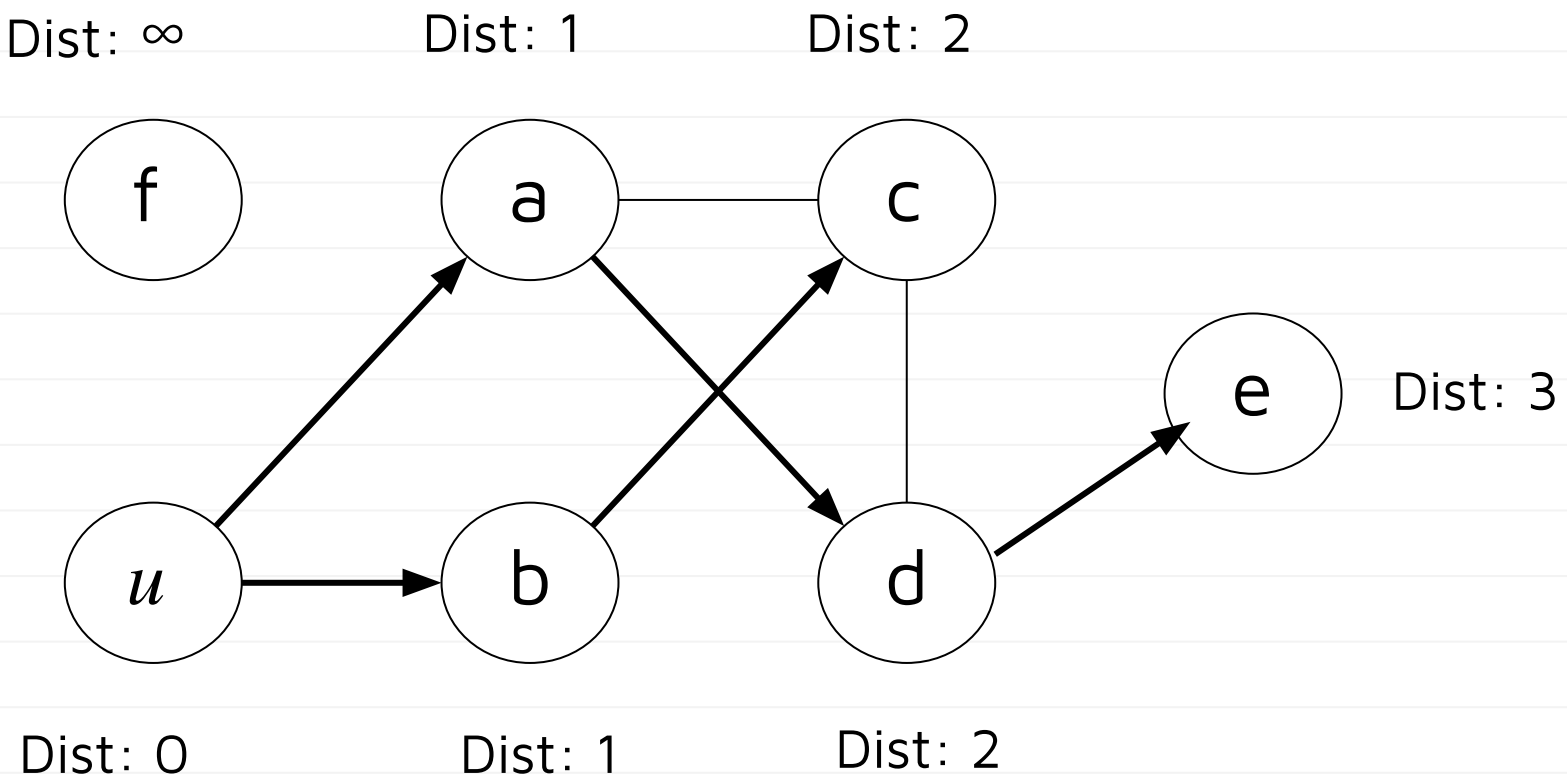
Example



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 mic

- 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

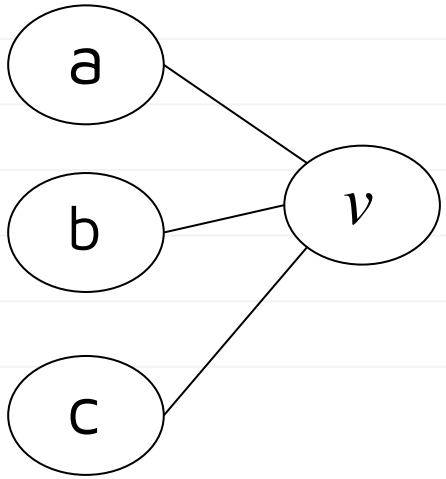
D: 8

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.

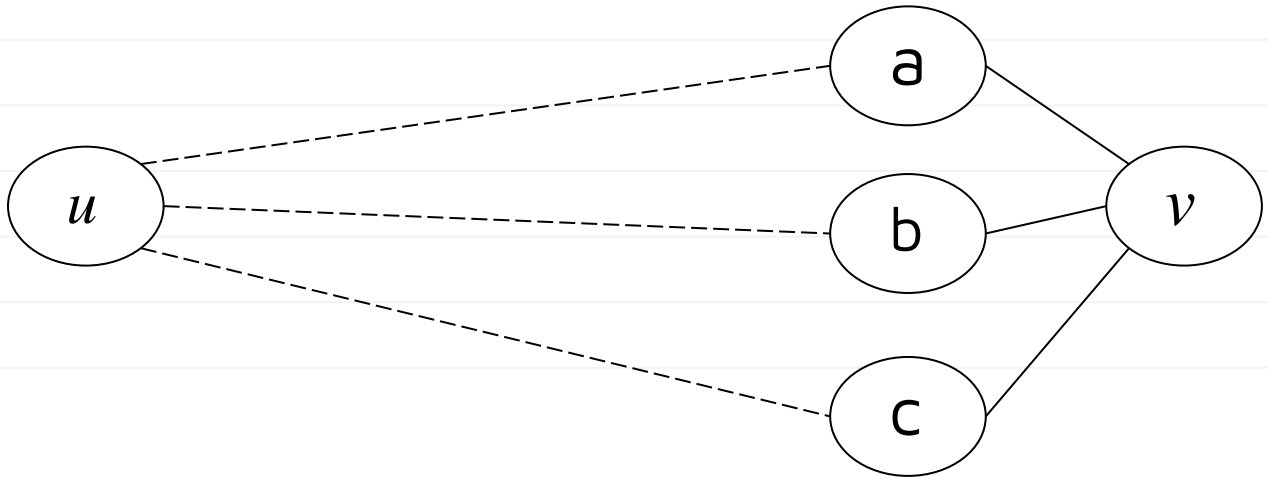
What is the distance from u to v?



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.

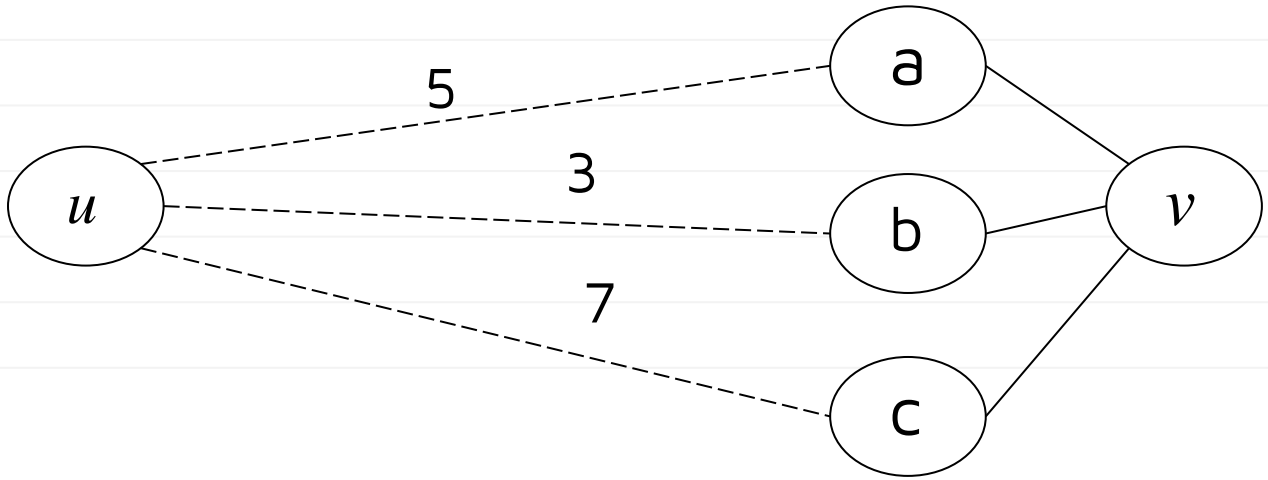
What is the distance from u to v?



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.

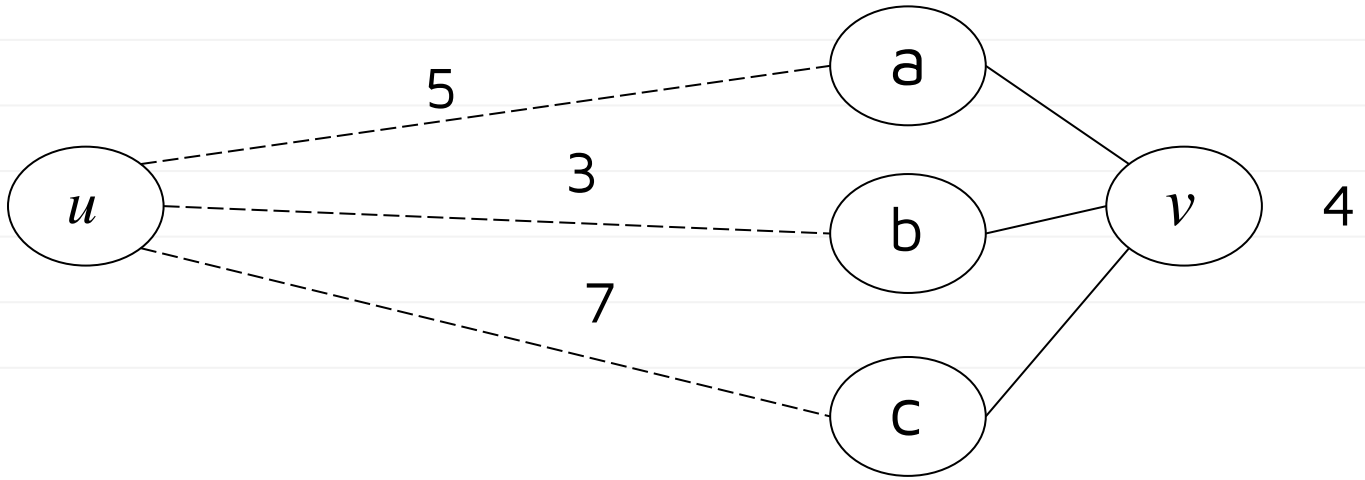
What is the distance from u to v?



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.

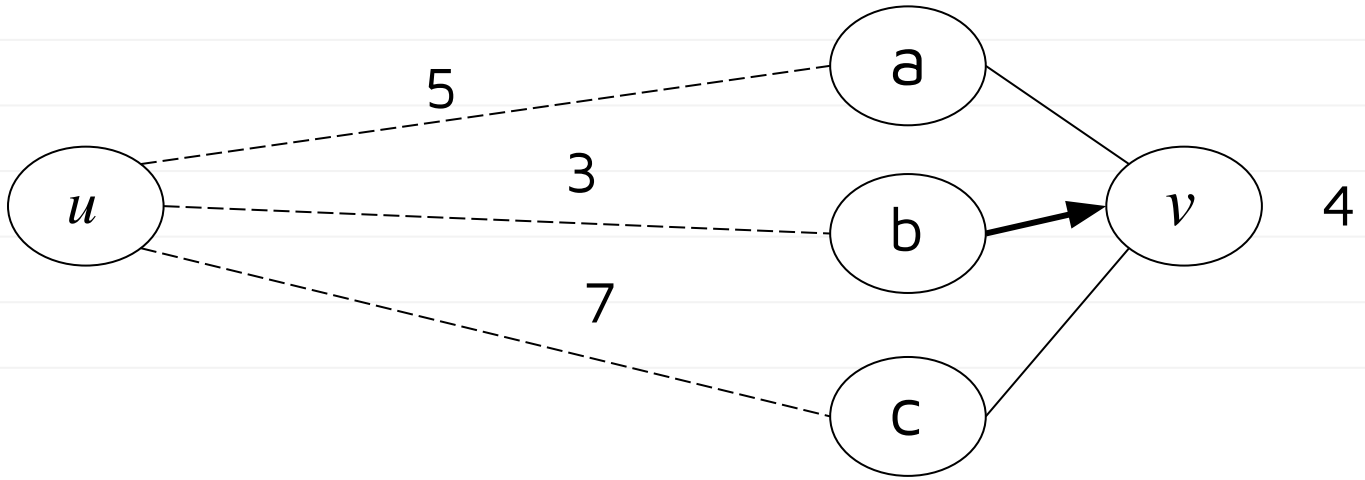
What is the distance from u to v?



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?

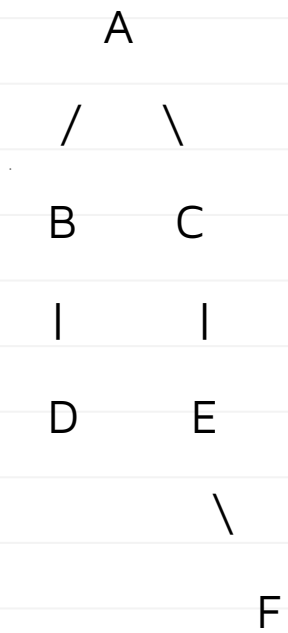


Algorithm Idea

- 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.

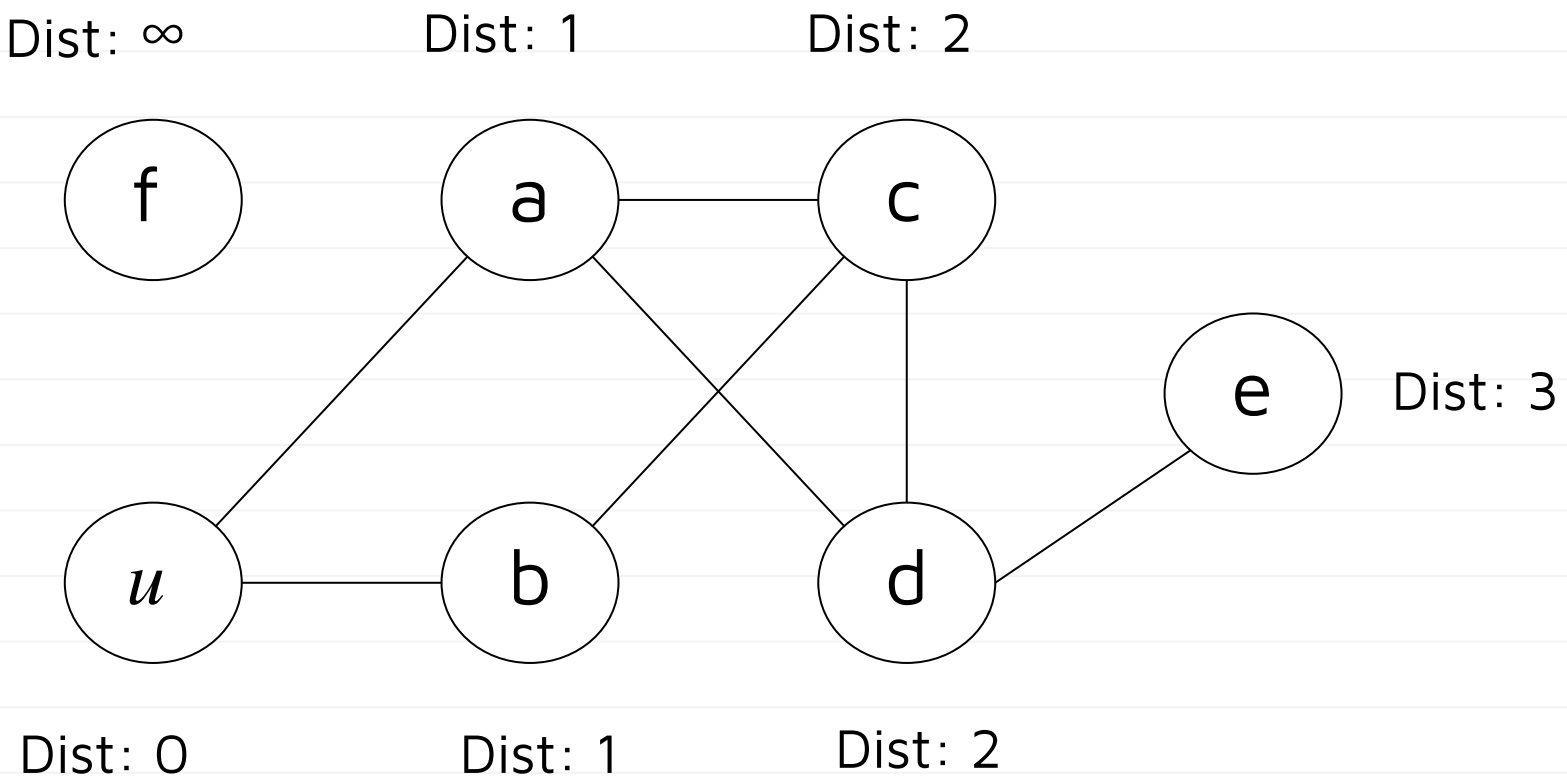


BFS for Shortest Paths

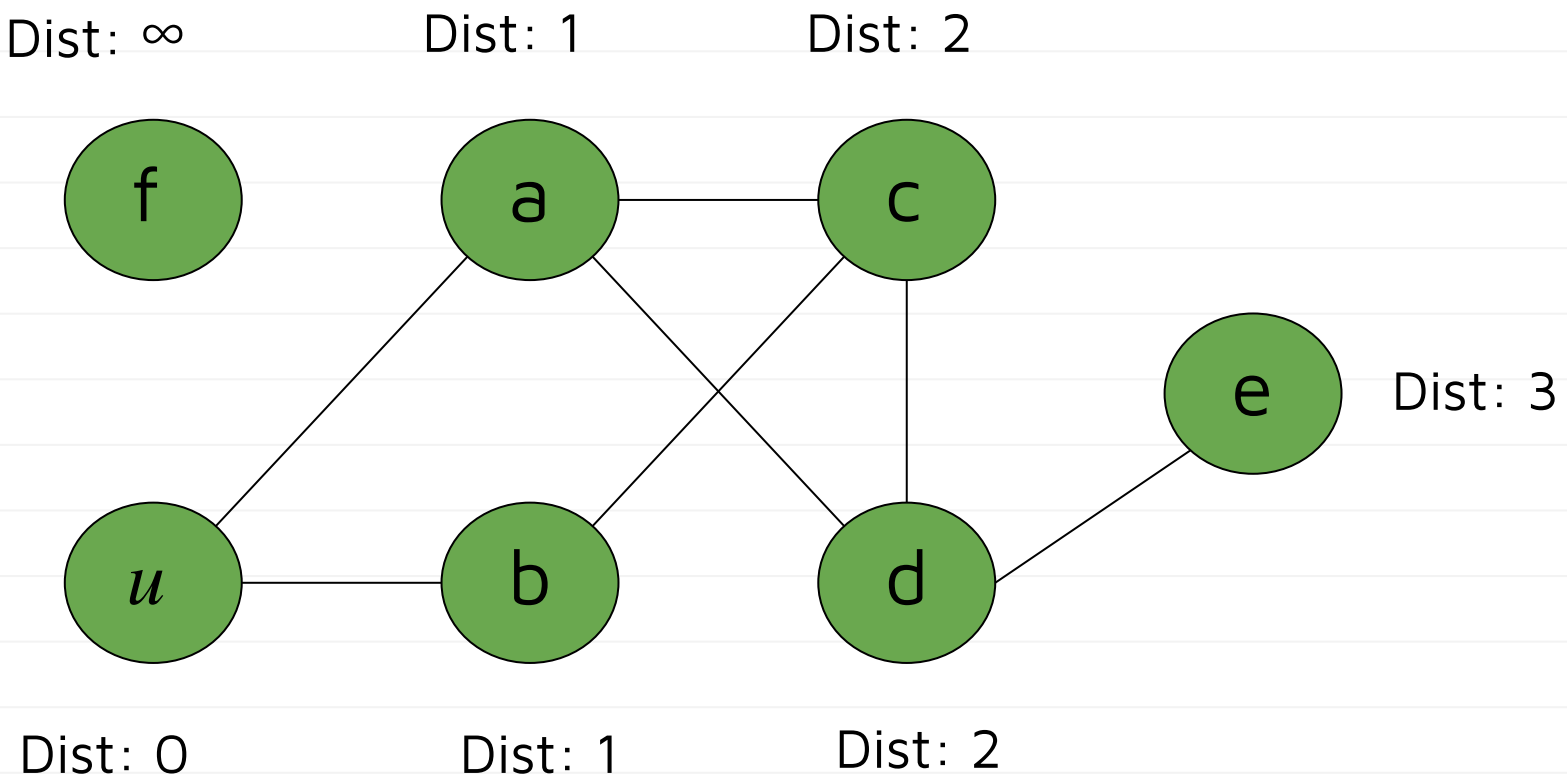
Key Property of BFS

- For any $k \geq 1$ you choose: **#take k = 10**
- All nodes distance $k - 1$ **(9)** from source are added to the queue before any node of distance k (**10**).
- Therefore, nodes are "processed" (popped from queue) in **order of distance** from source.

Example

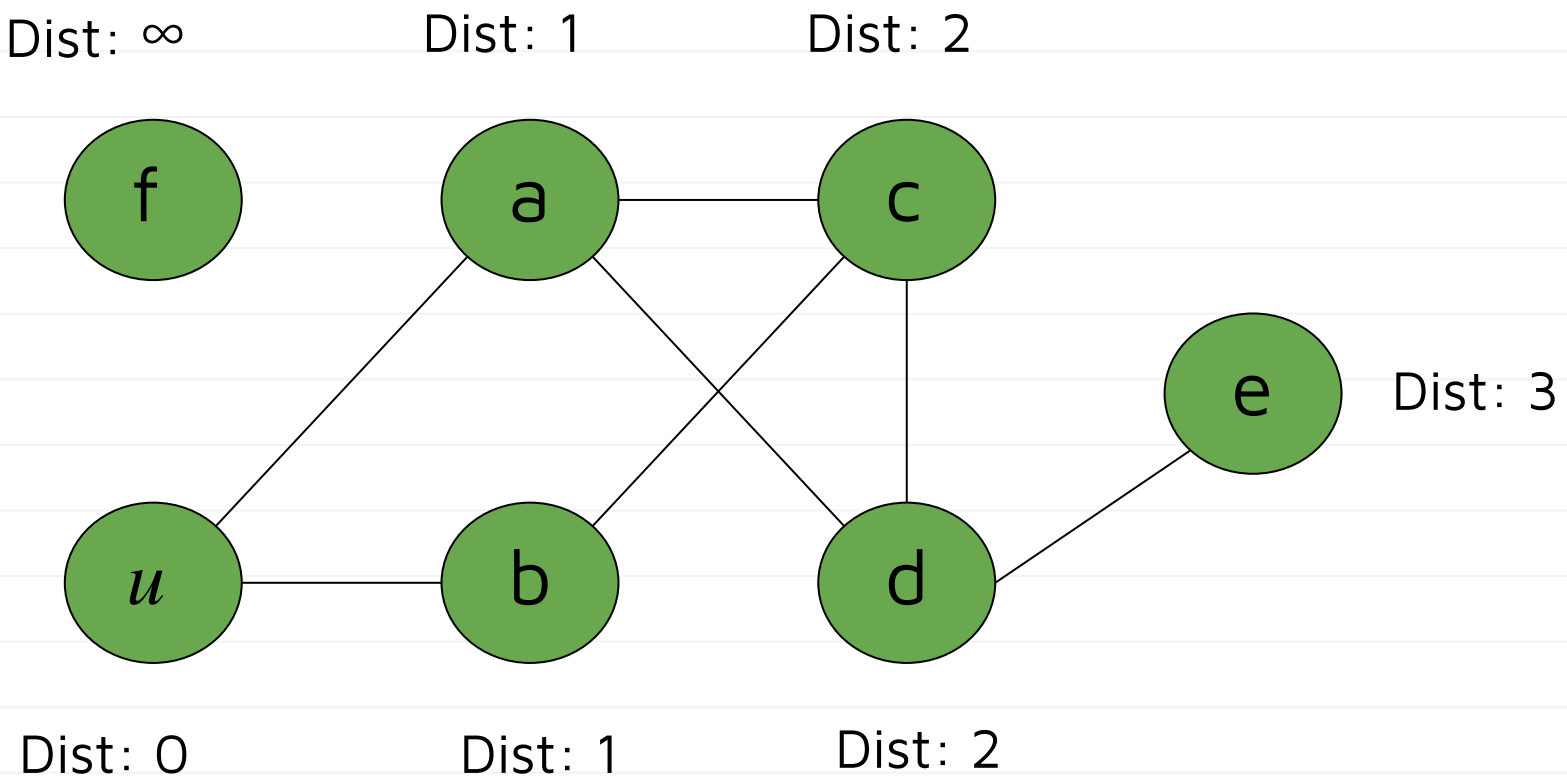


Example



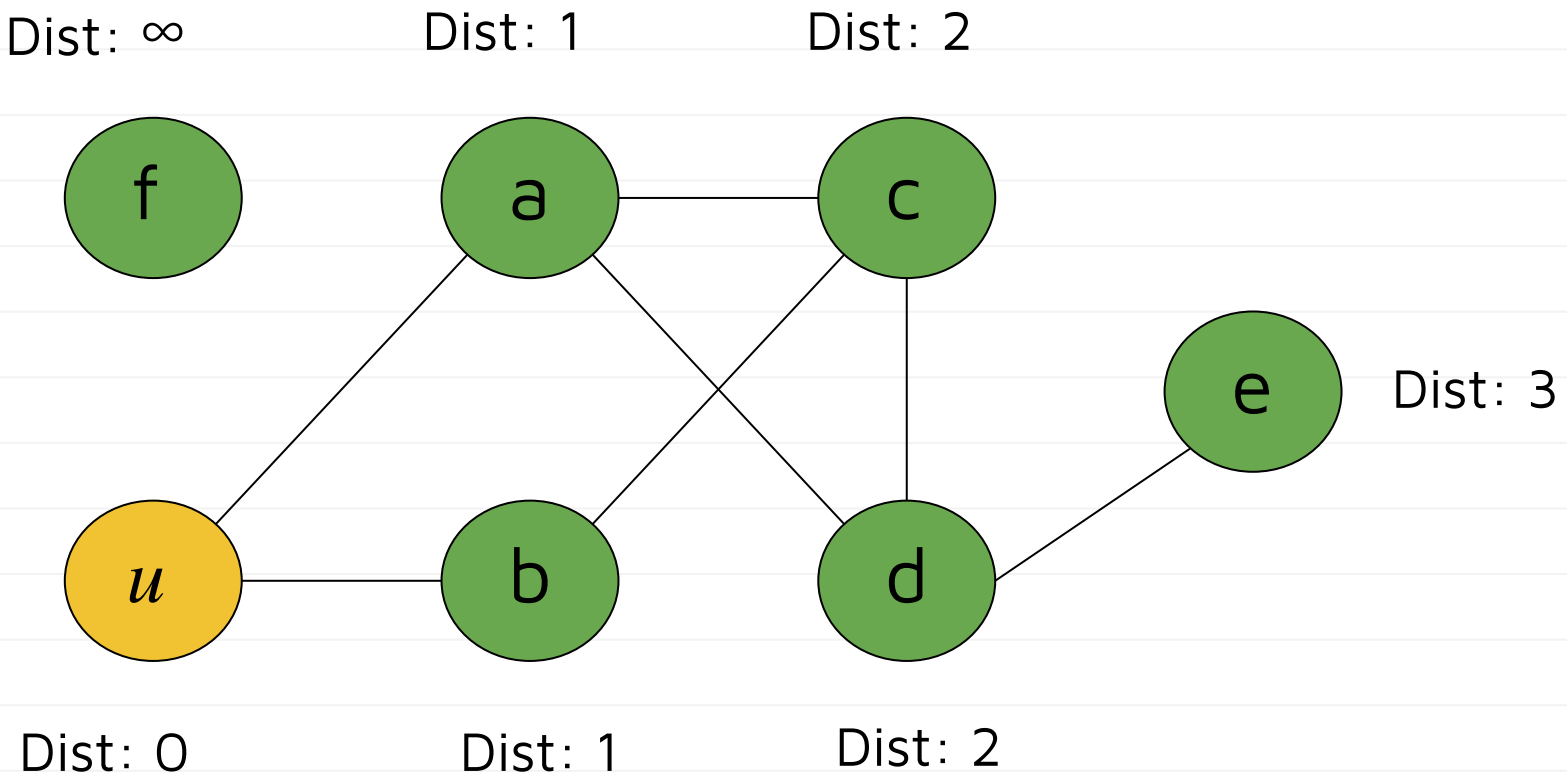
Example

[ ]



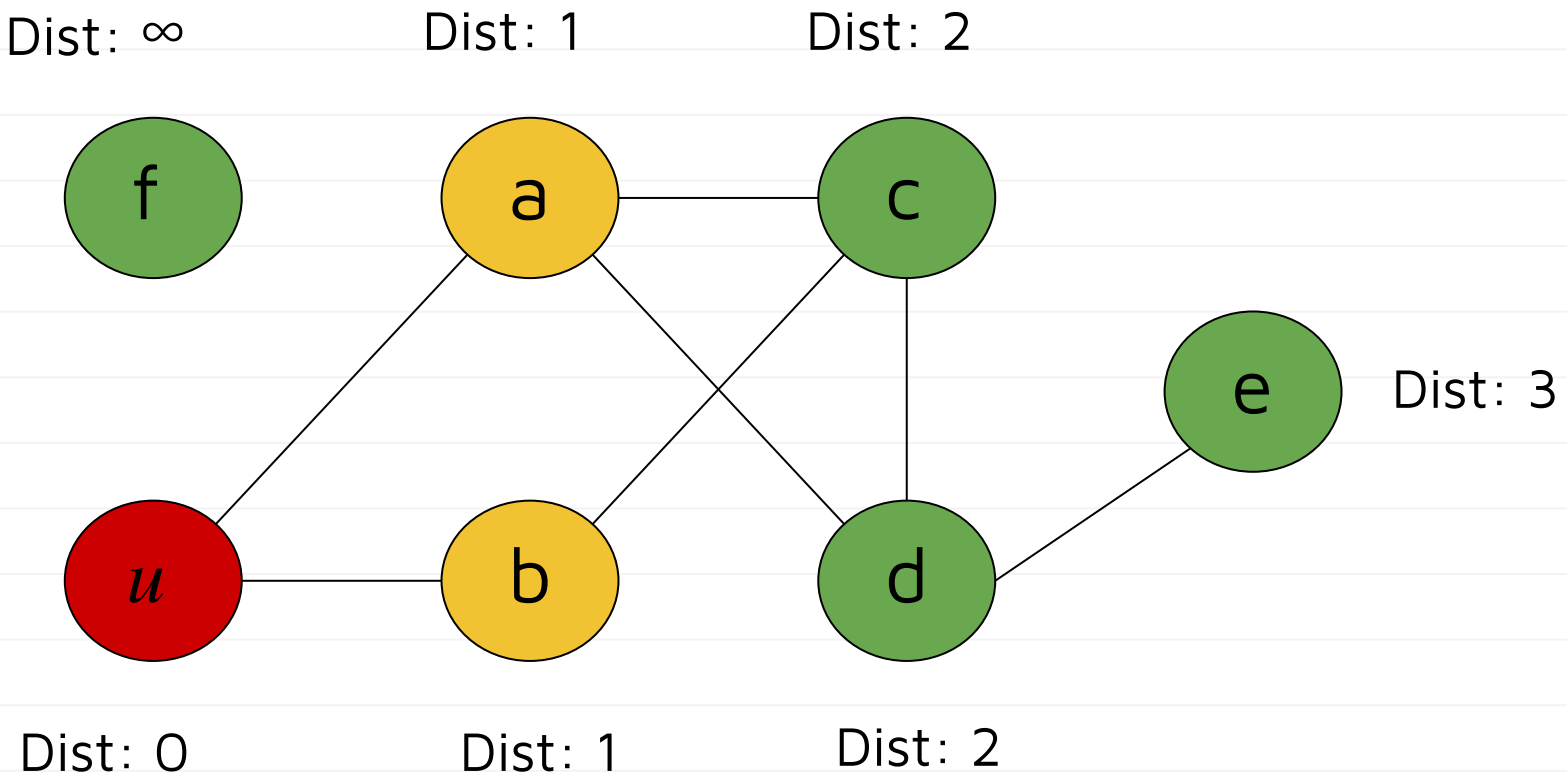
Example

[$u(0)$]



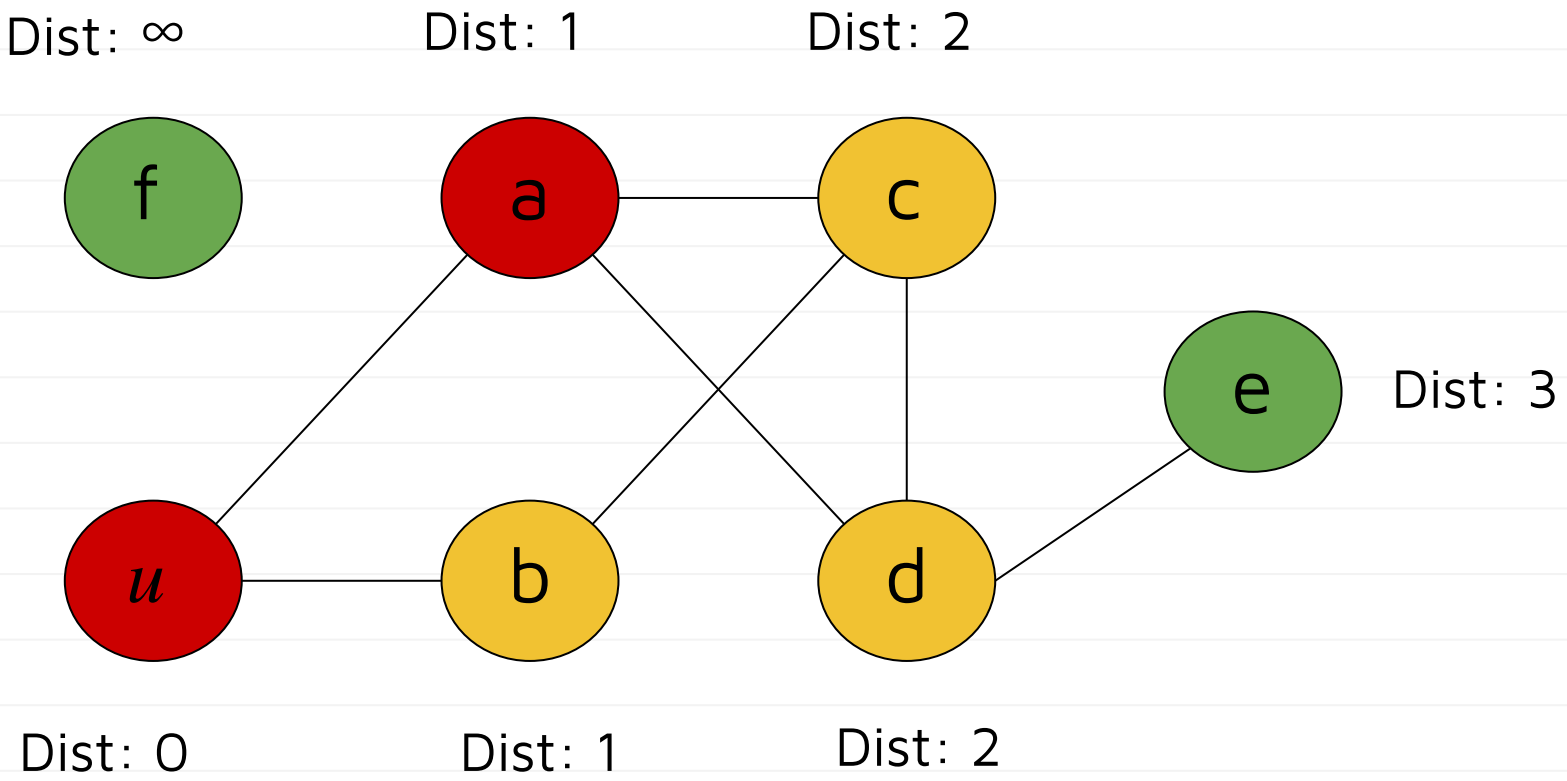
Example

[~~u(0)~~, a(1), b(1)]



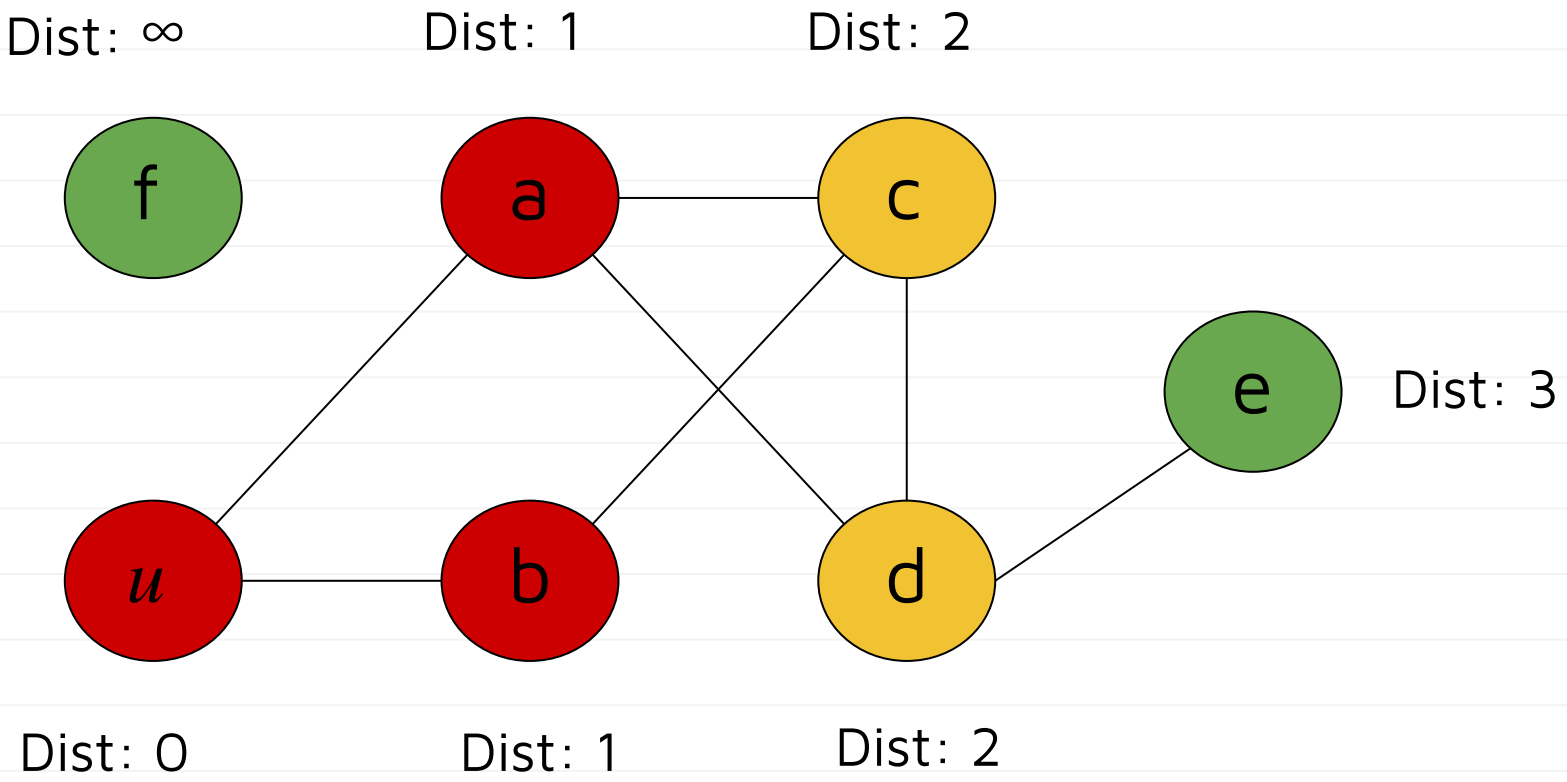
Example

[~~*u*(0)~~, ~~a(1)~~, b(1), c(2), d(2)]



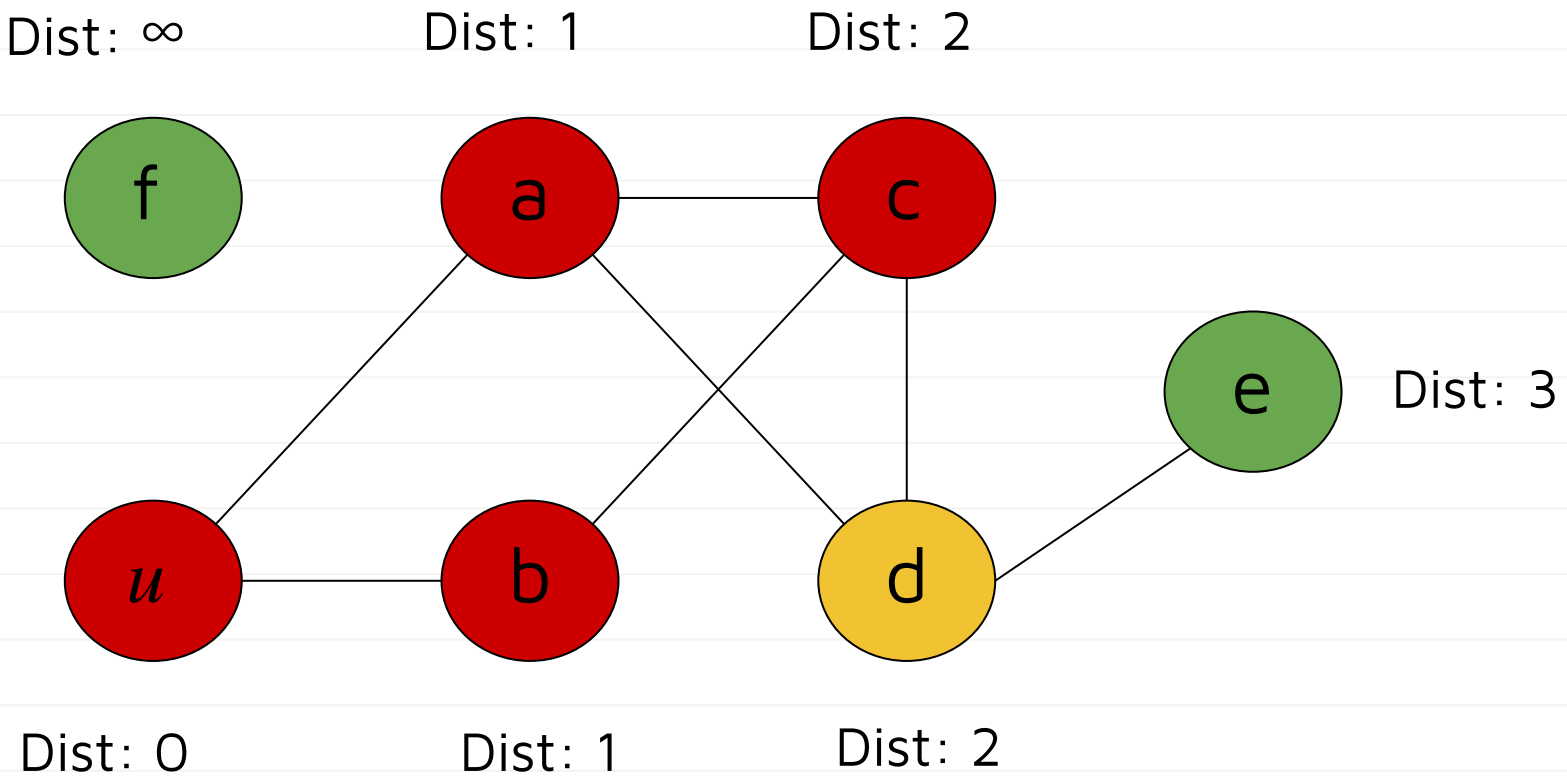
Example

[~~*u*(0)~~, ~~a(1)~~, ~~b(1)~~, c(2), d(2)]



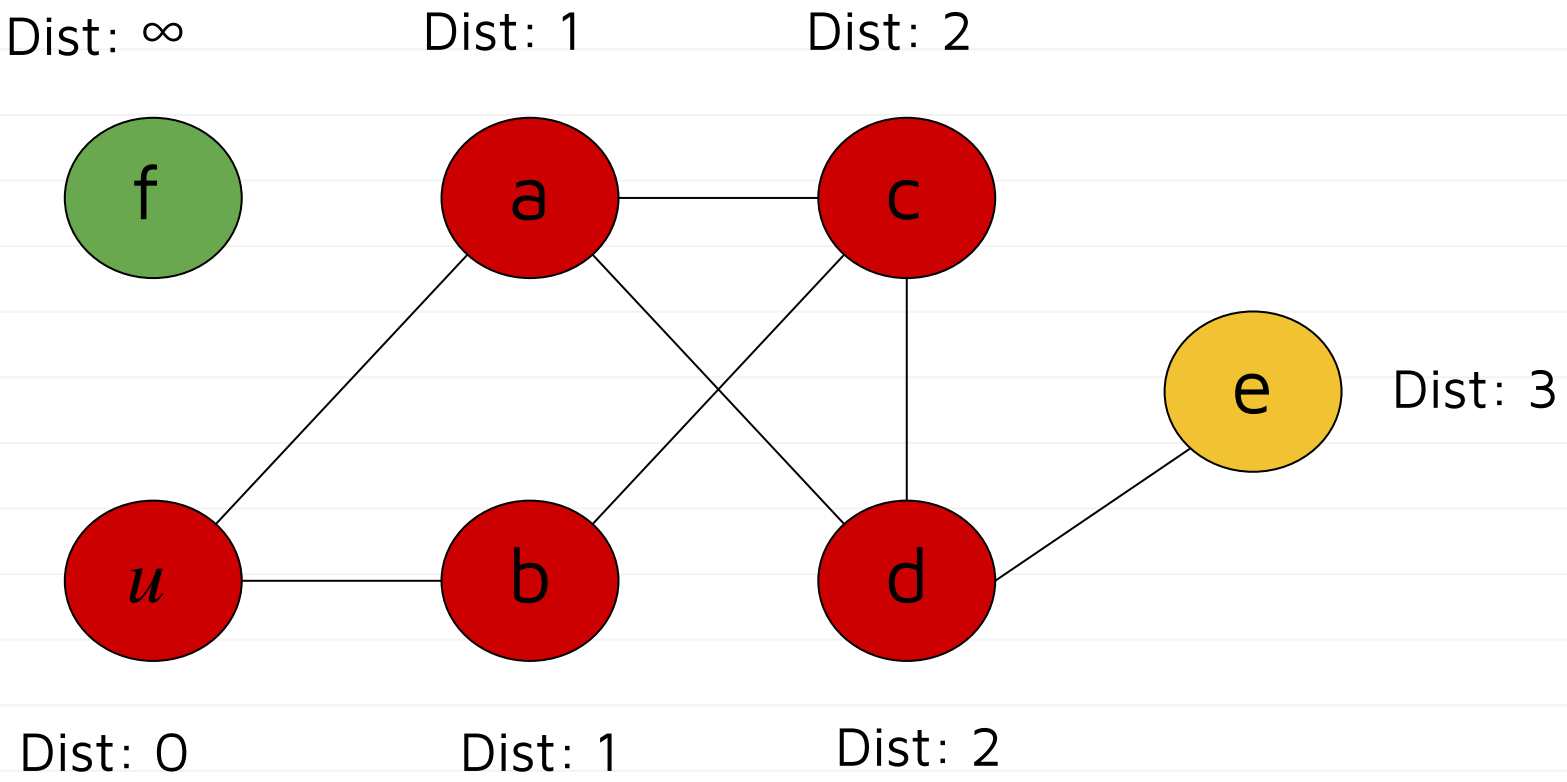
Example

[~~*u*(0)~~, ~~a(1)~~, ~~b(1)~~, ~~c(2)~~, d(2)]



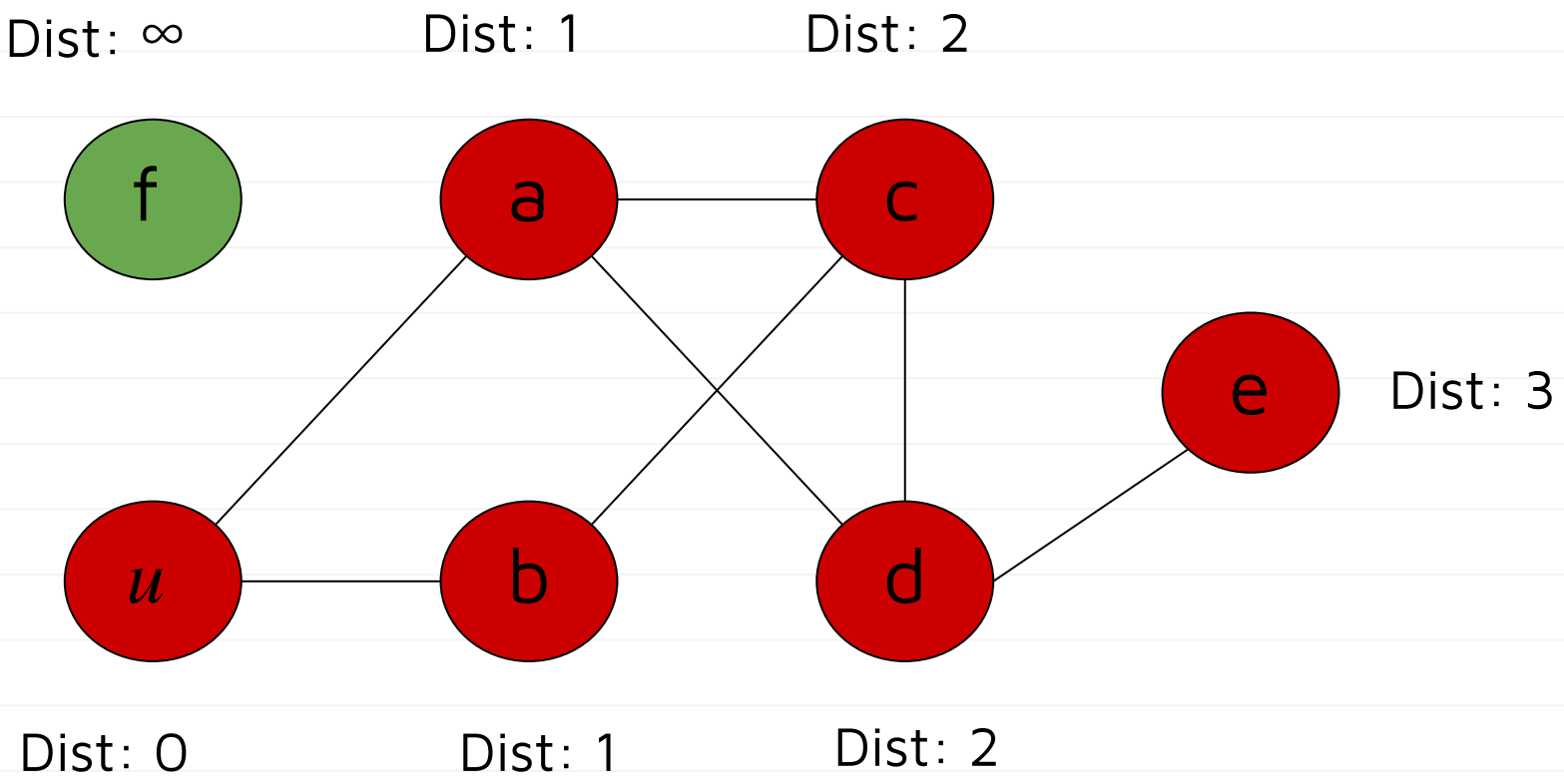
Example

[~~*u*(0)~~, ~~a(1)~~, ~~b(1)~~, ~~c(2)~~, ~~d(2)~~, e(3)]



Example

[~~*u*(0)~~, ~~a(1)~~, ~~b(1)~~, ~~c(2)~~, ~~d(2)~~, ~~e(3)~~]



Discovering Shortest Paths

- We “discover” shortest paths when we pop a node from queue and look at its neighbors.
- But the neighbor’s status matters!

Consider This

- We pop a node s.
- It has a neighbor v whose status is **undiscovered**.
- We've discovered a **shortest path** to v through s!

Consider This

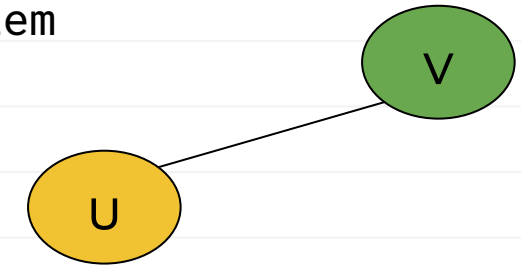
- We pop a node s.
- It has a neighbor v whose status is **pending** or **visited**.
- We already have a shortest path to v.

Modifying BFS

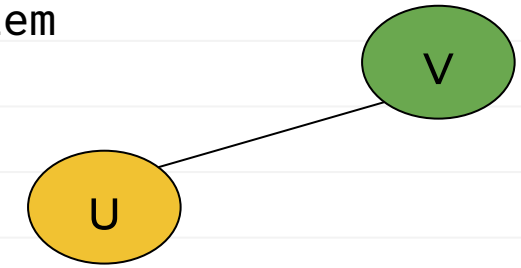
- Use BFS "framework".
- Return dictionary of **search predecessors**.
  - If v is discovered while visiting u, we say that u is the **BFS predecessor** of v.
  - This encodes the shortest paths.
- Also return dictionary of shortest path distances.

```
def bfs_shortest_path(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft() #remove the first elem
        for v in graph.neighbors(u):
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                pending.append(v)
        status[u] == 'visited'
    return predecessor, distance
```

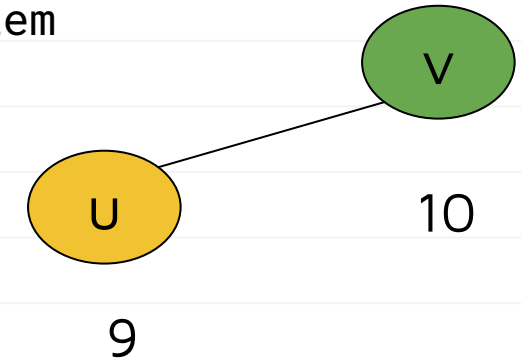
```
def bfs_shortest_path(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft() #remove the first elem
        for v in graph.neighbors(u):
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                pending.append(v)
        status[u] == 'visited'
    return predecessor, distance
```

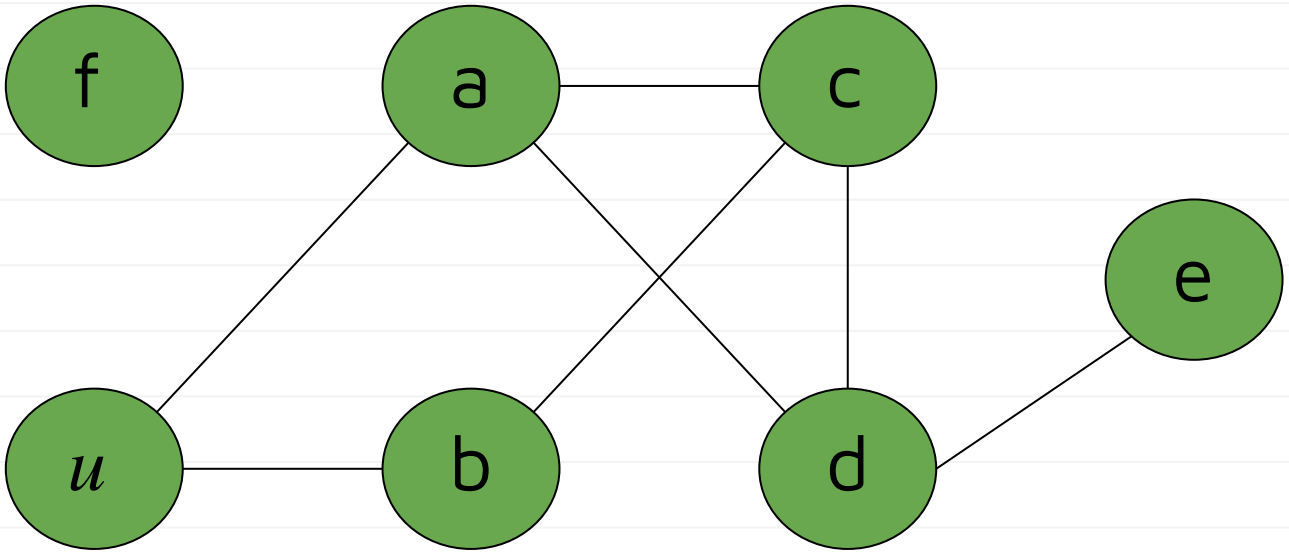


```
def bfs_shortest_path(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft() #remove the first elem
        for v in graph.neighbors(u):
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                pending.append(v)
        status[u] == 'visited'
    return predecessor, distance
```

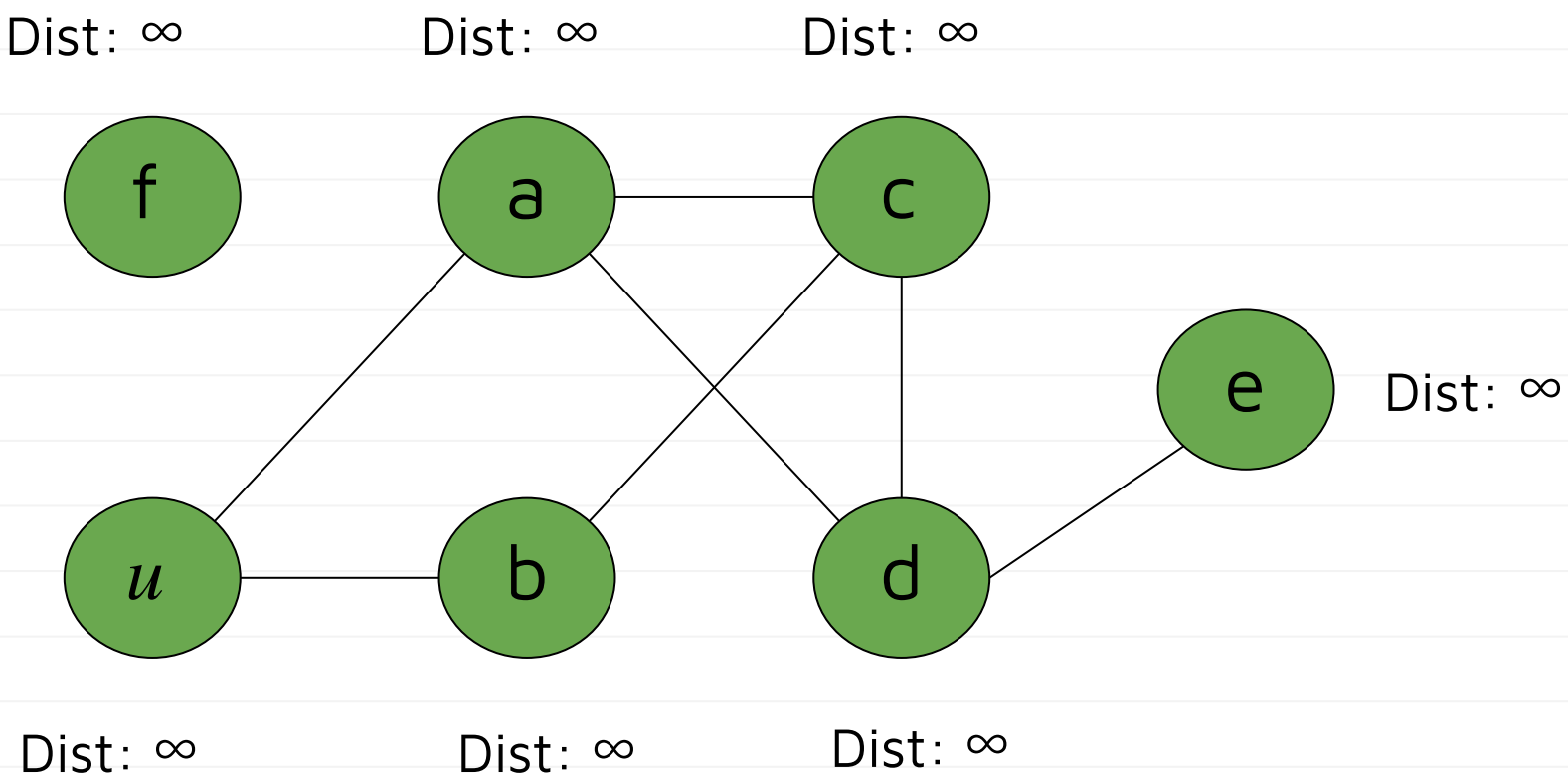


```
def bfs_shortest_path(graph, source):
    """Start a BFS at `source`."""
    status = {node: 'undiscovered' for node in graph.nodes}
    distance = {node: float('inf') for node in graph.nodes}
    predecessor = {node: None for node in graph.nodes}

    status[source] = 'pending'
    distance[source] = 0
    pending = deque([source])

    # while there are still pending nodes
    while pending:
        u = pending.popleft() #remove the first elem
        for v in graph.neighbors(u):
            if status[v] == 'undiscovered':
                status[v] = 'pending'
                distance[v] = distance[u] + 1
                predecessor[v] = u
                pending.append(v)
        status[u] == 'visited'
    return predecessor, distance
```



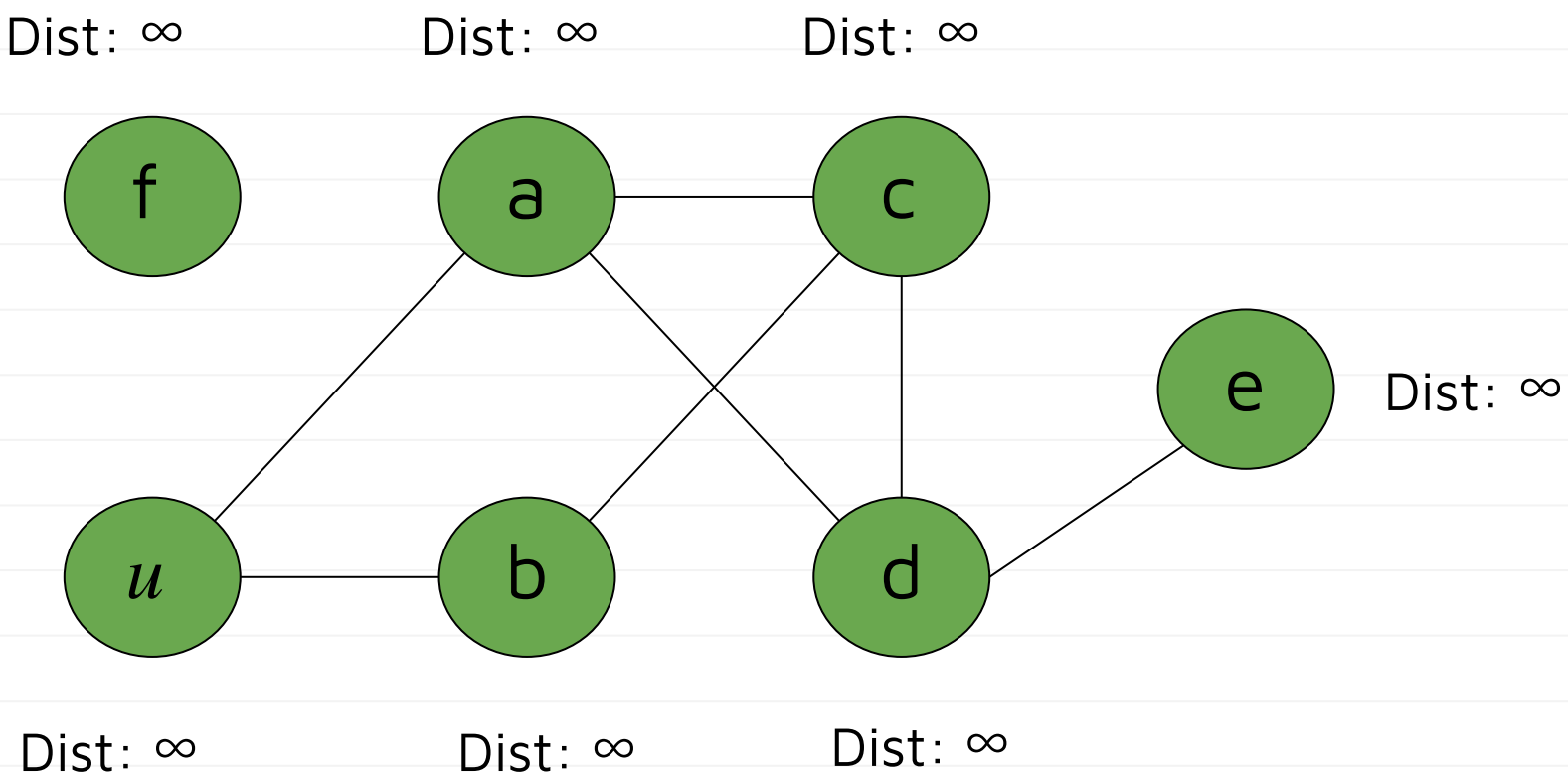


distance = {



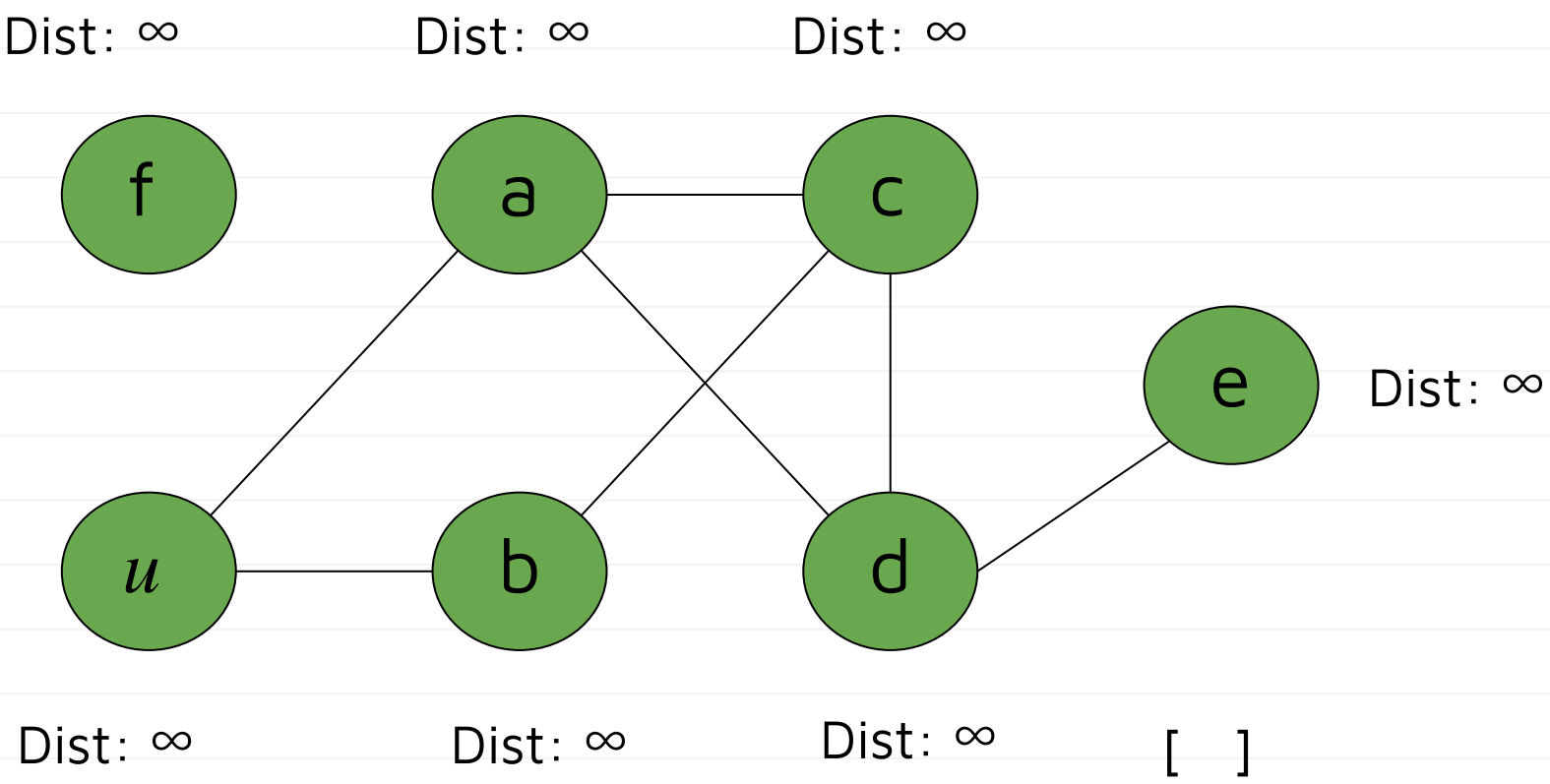
distance = {

predec = {



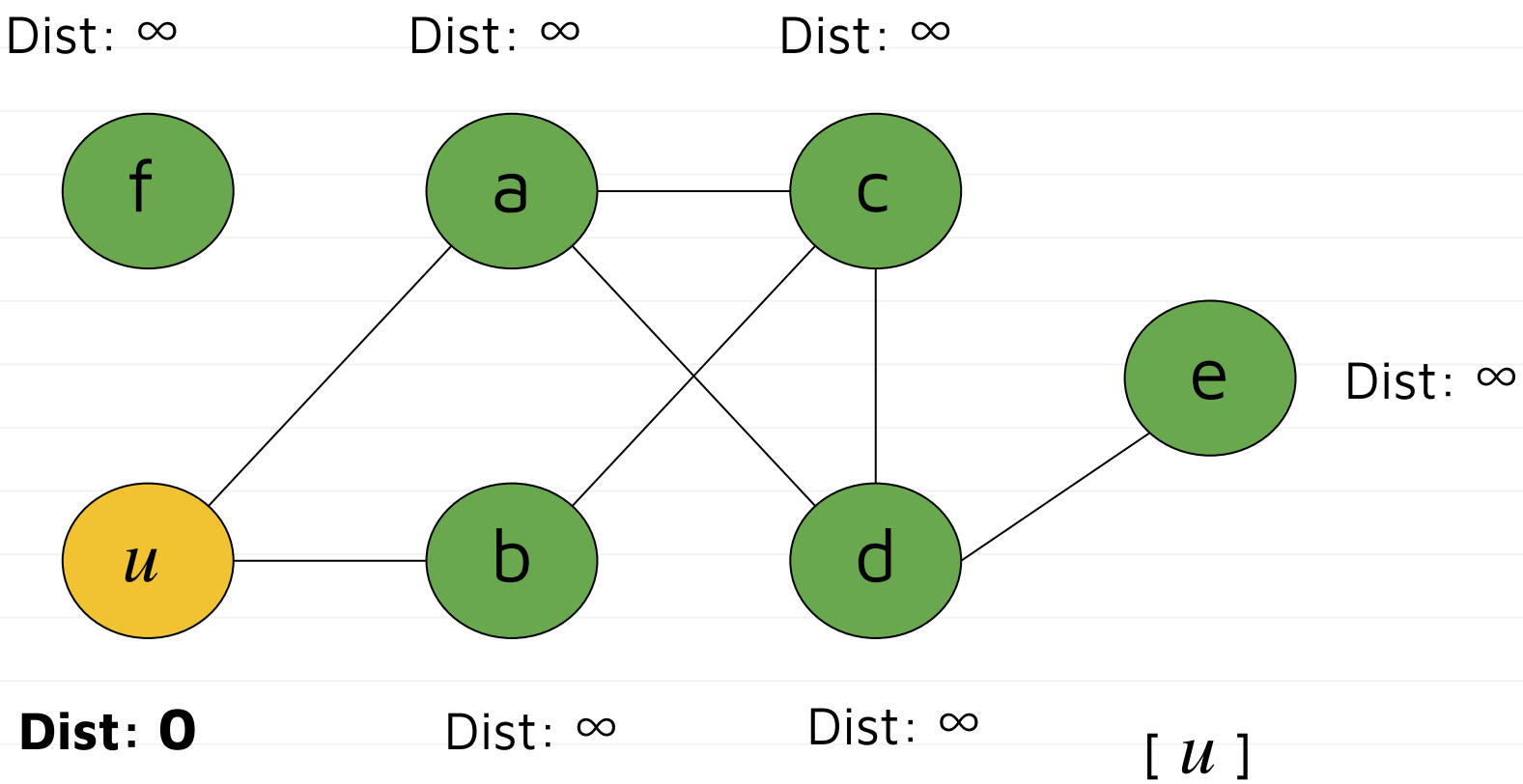
distance = {

predec = {



distance = {'u': 0

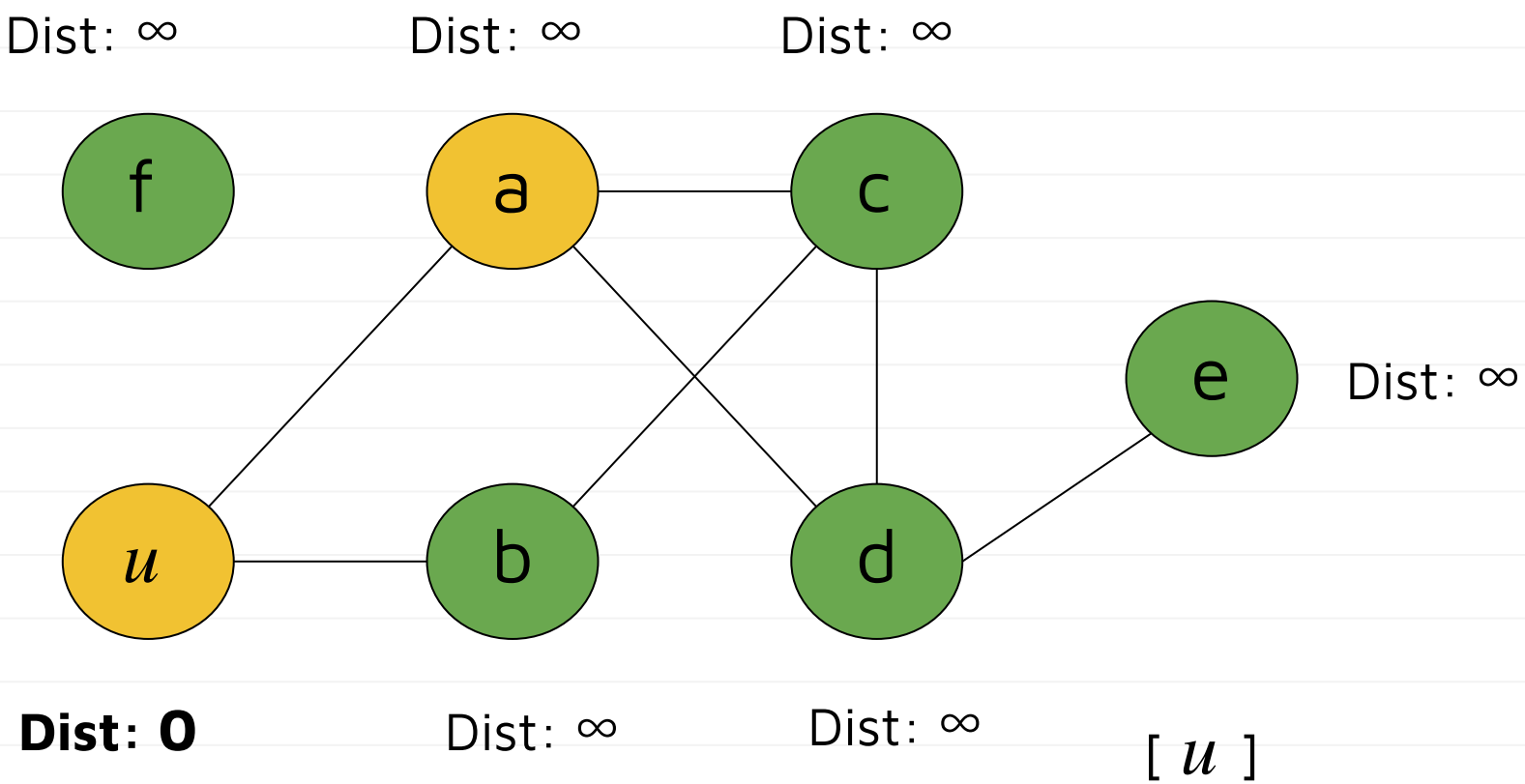
predec = {



[u]

distance = { 'u': 0

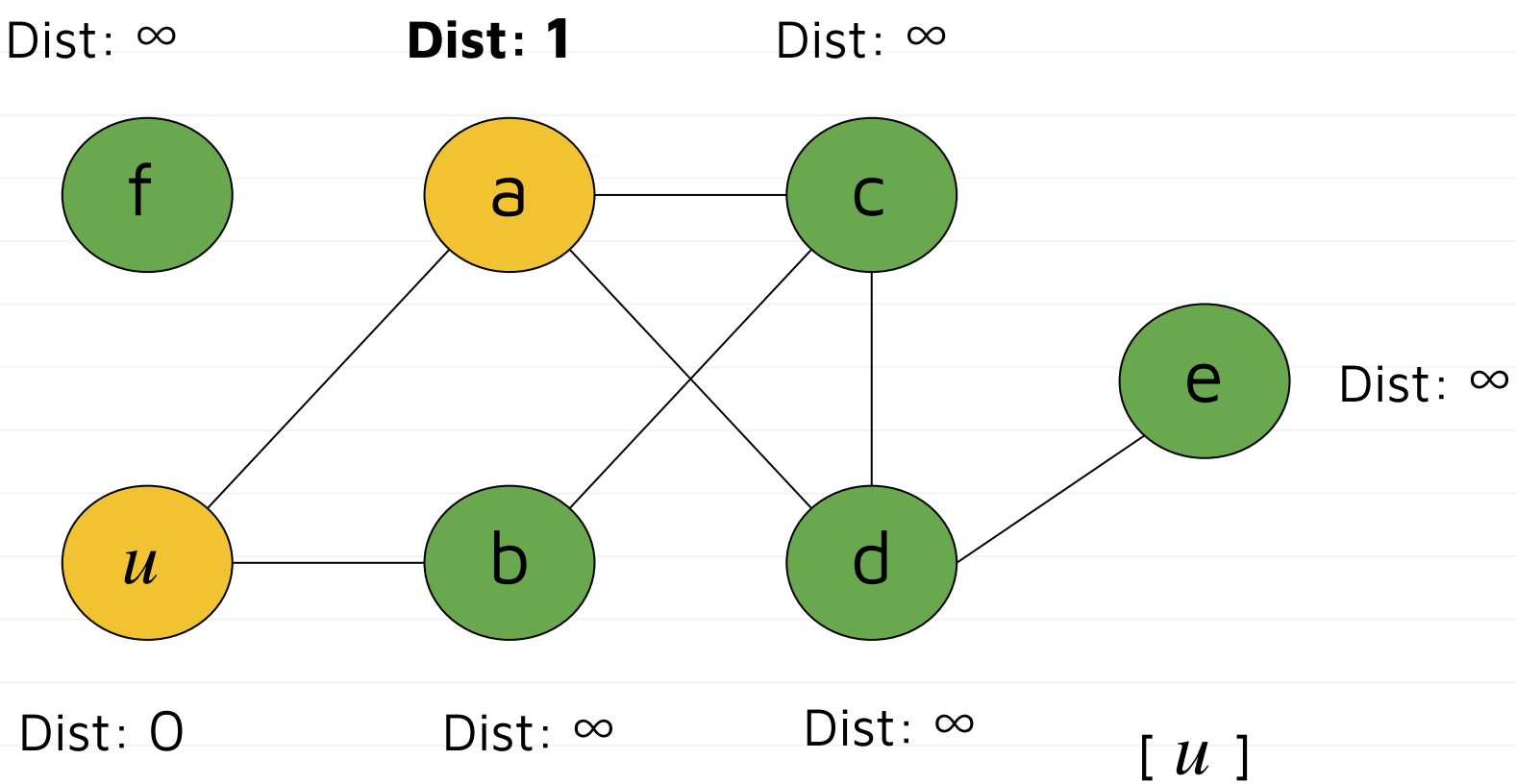
predec = {



[u]

distance = { 'u': 0, 'a': 1

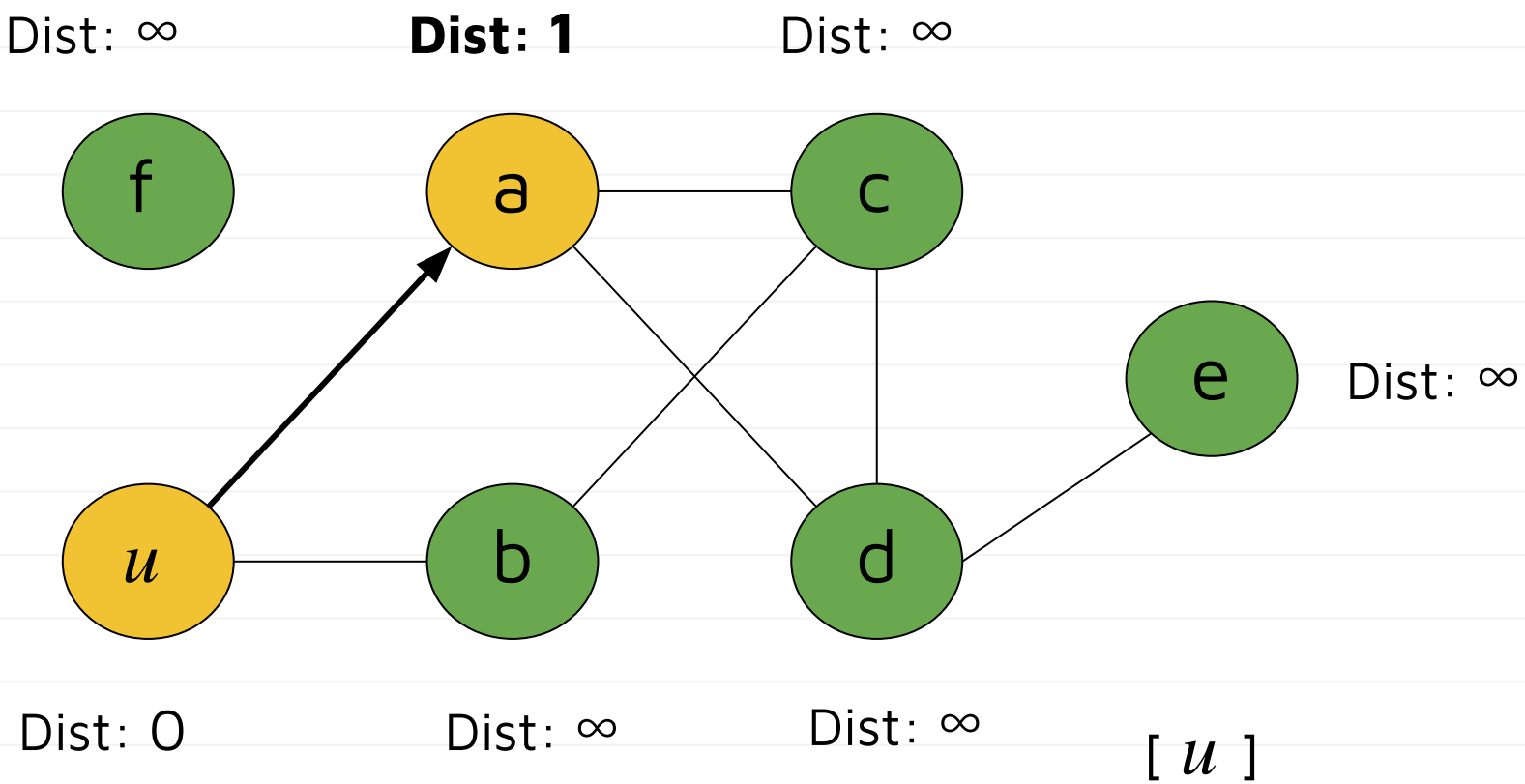
predec = {



[*u*]

distance = { 'u': 0, 'a': 1

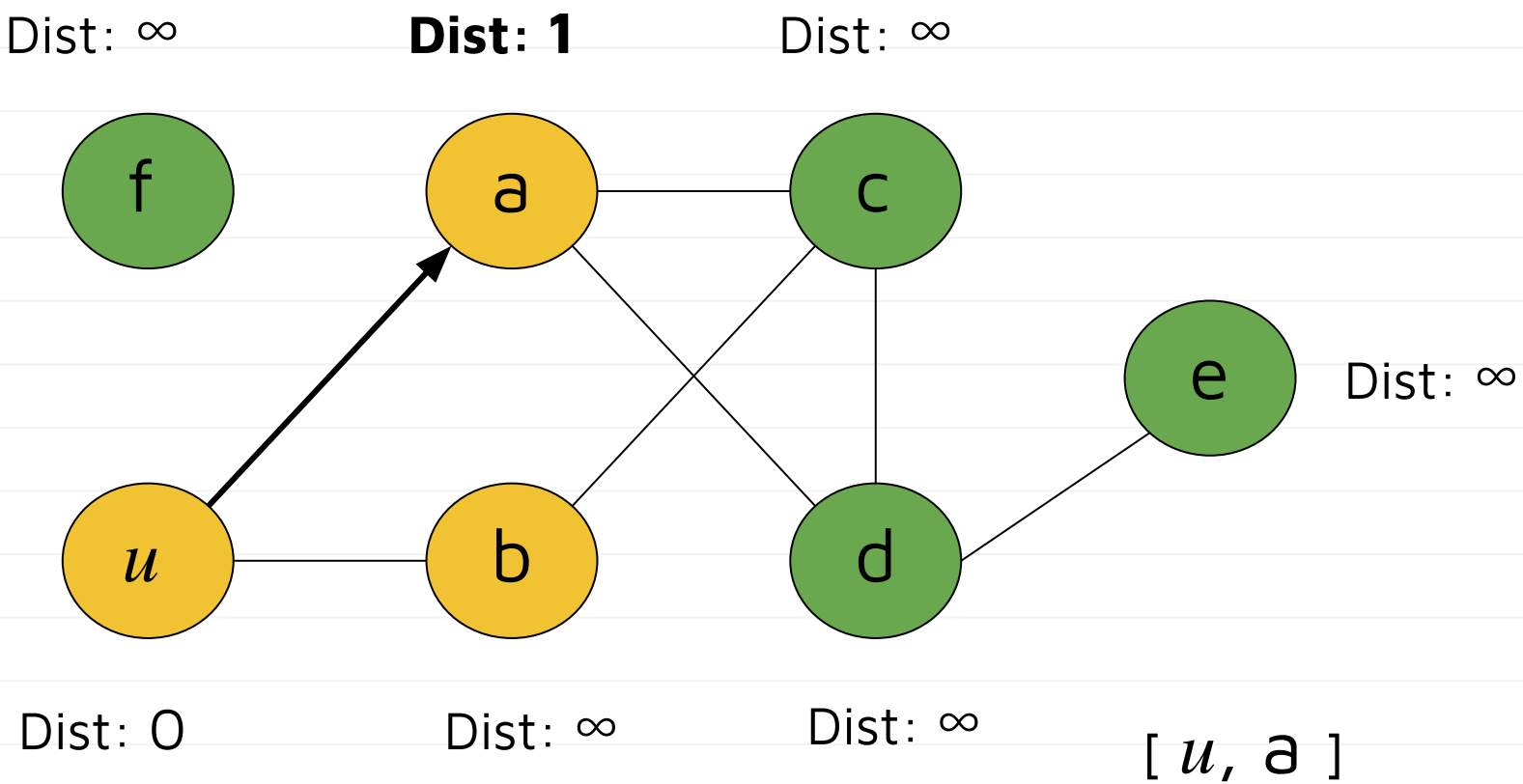
predec = {'a': 'u'



[u]

distance = { 'u': 0, 'a': 1

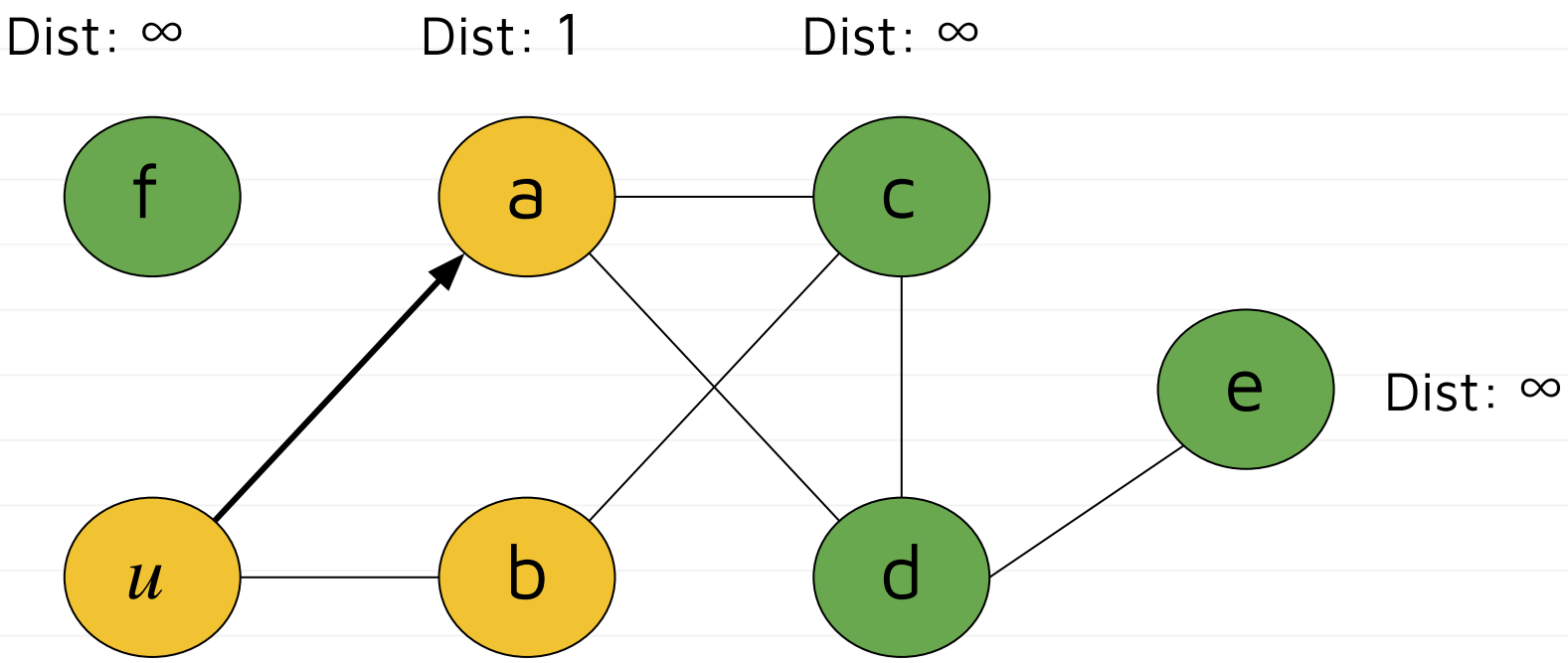
predec = {'a': 'u'



[u, a]

distance = { 'u': 0, 'a': 1, 'b': 1

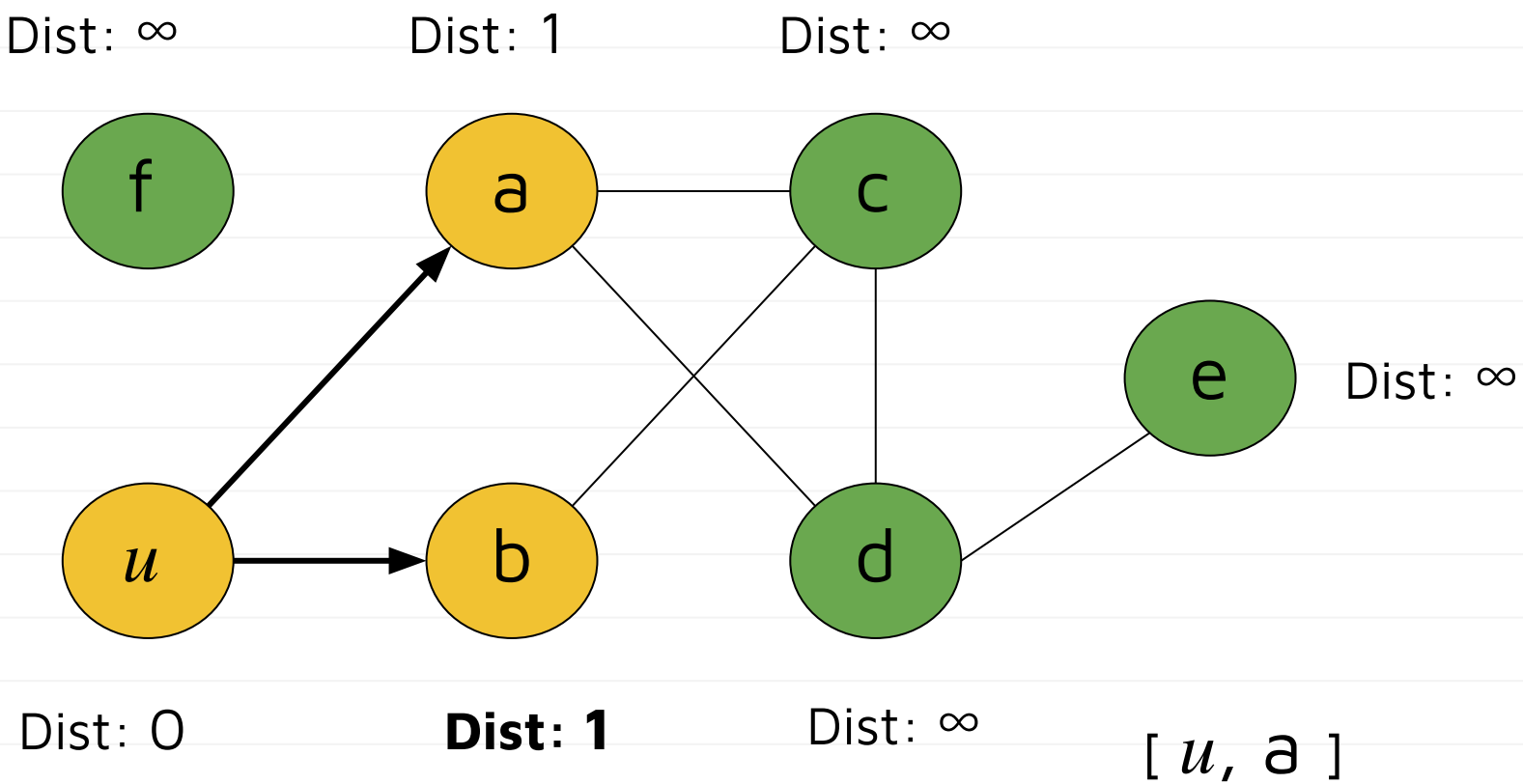
predec = {'a': 'u'



[u, a]

distance = { 'u': 0, 'a': 1, 'b': 1

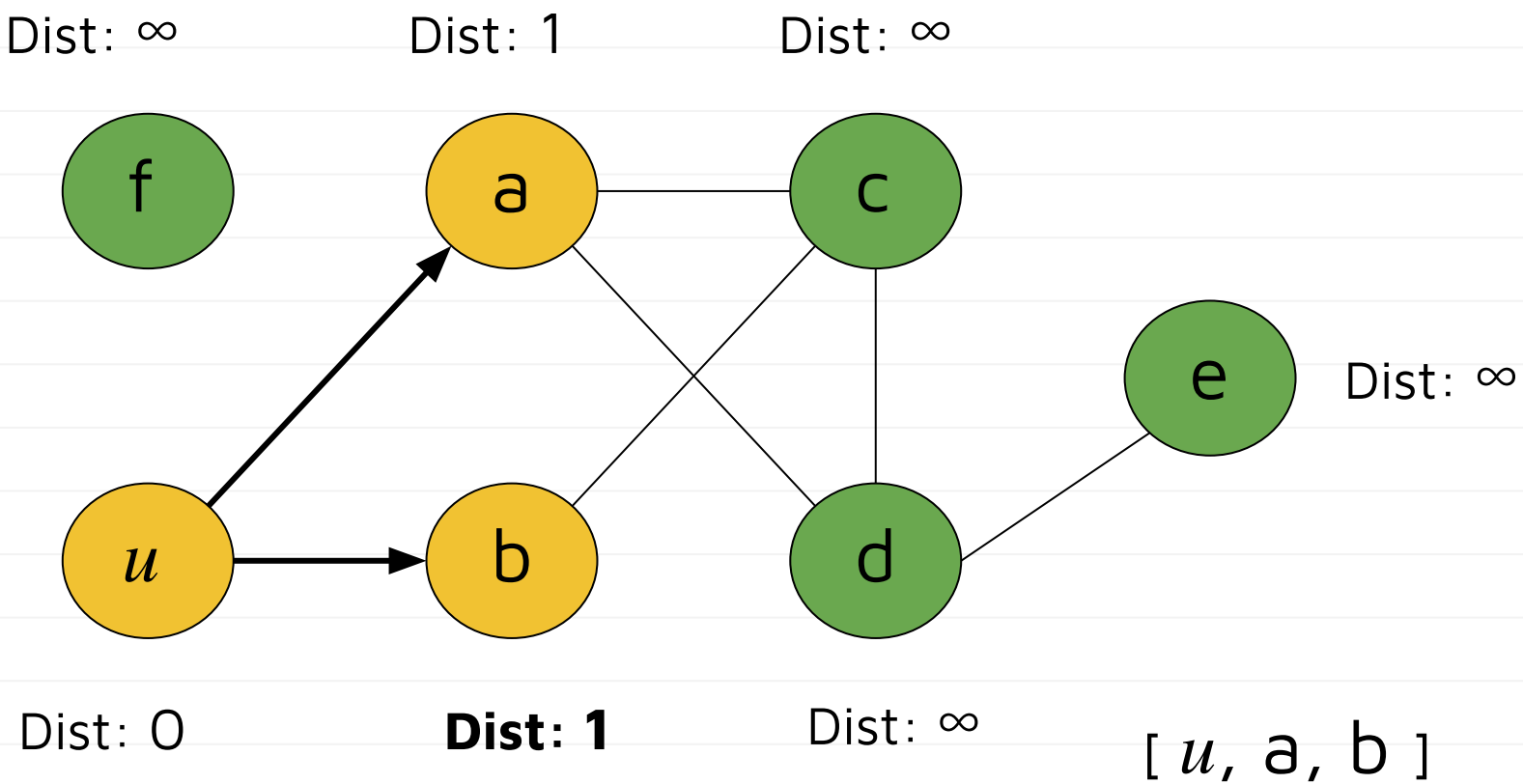
predec = {'a': 'u', 'b': 'u'



[u, a]

distance = { 'u': 0, 'a': 1, 'b': 1

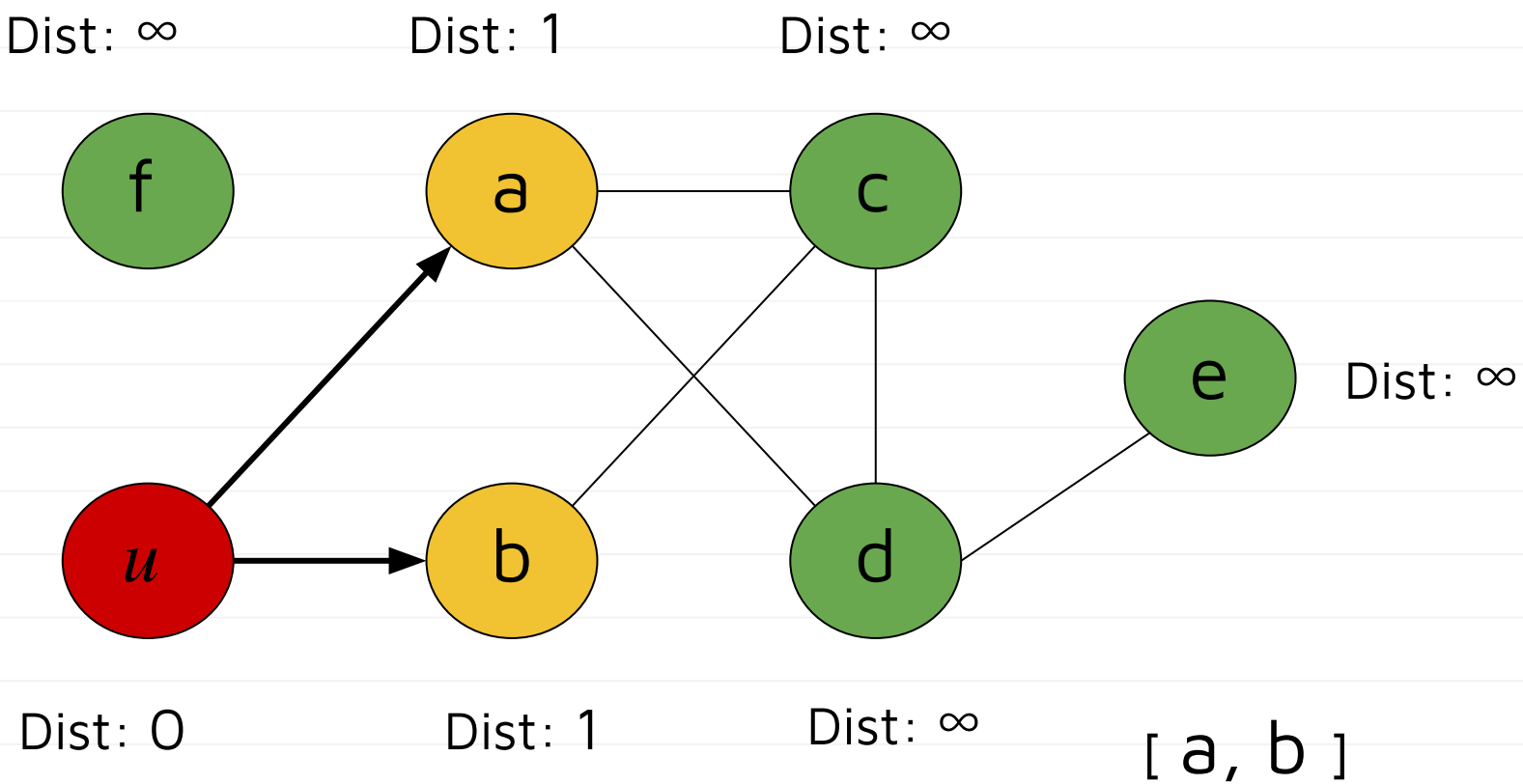
predec = {'a': 'u', 'b': 'u'



[u, a, b]

distance = { 'u': 0, 'a': 1, 'b': 1

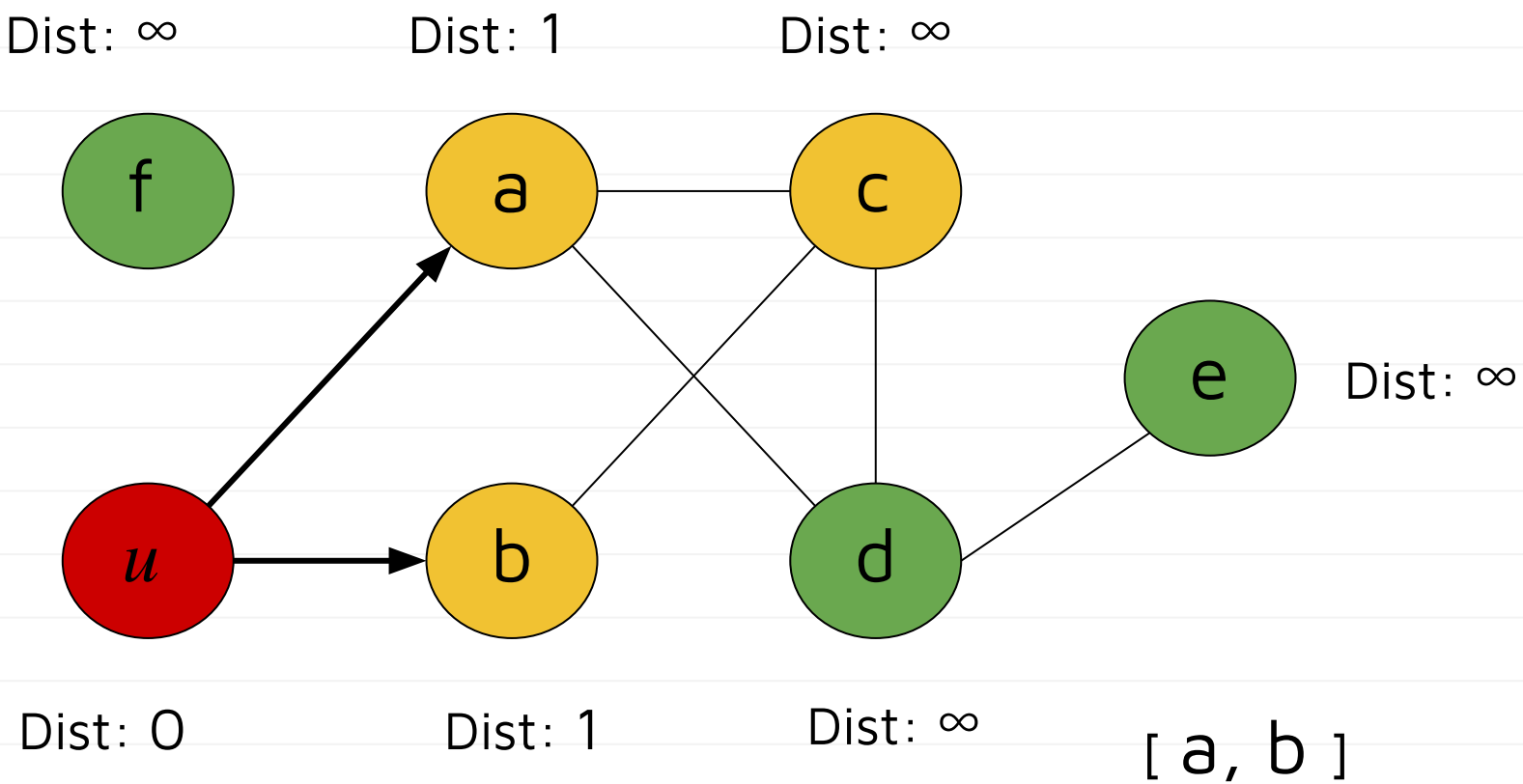
predec = {'a': 'u', 'b': 'u'



[a, b]

distance = { 'u': 0, 'a': 1, 'b': 1

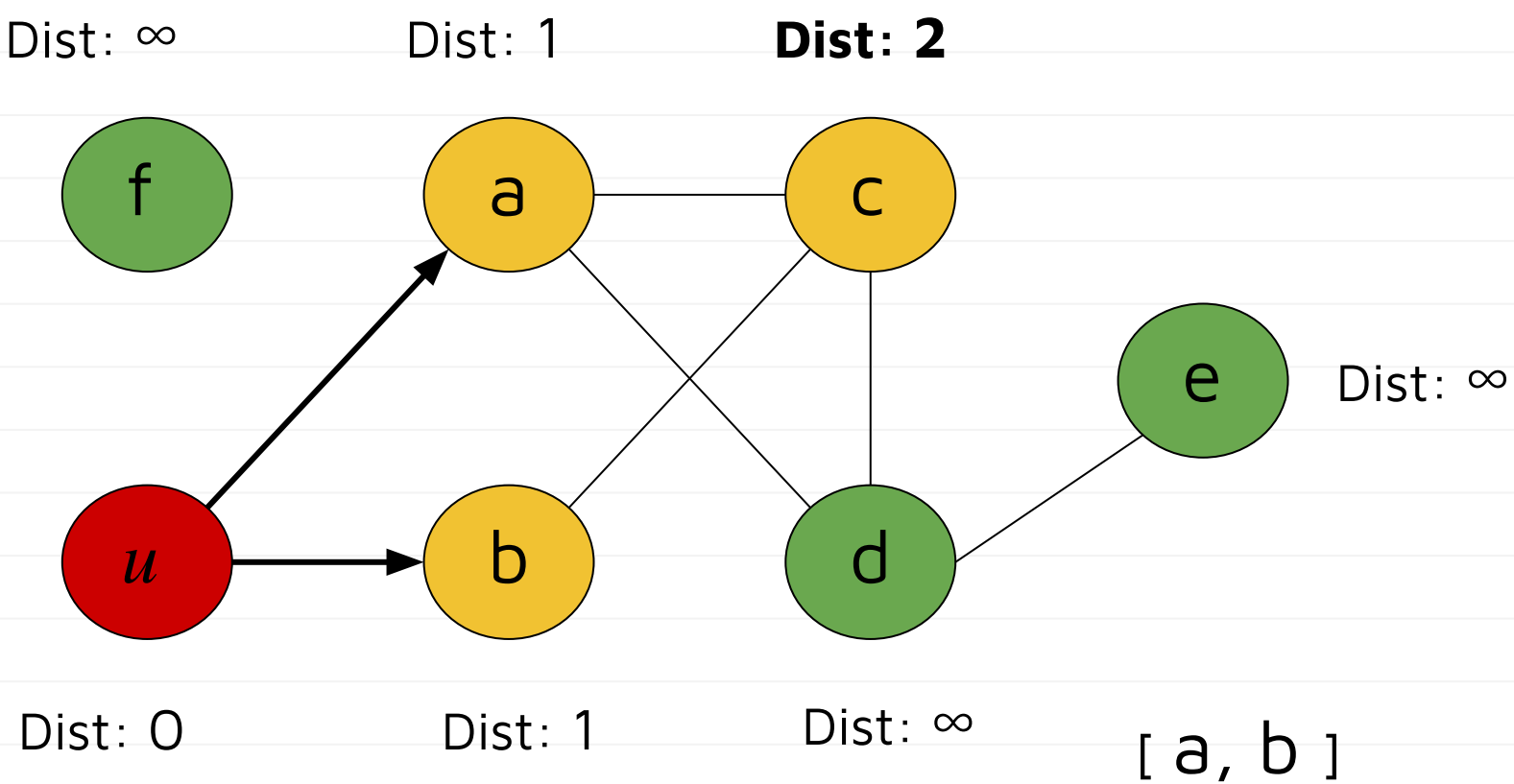
predec = {'a': 'u', 'b': 'u'



[a, b]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2

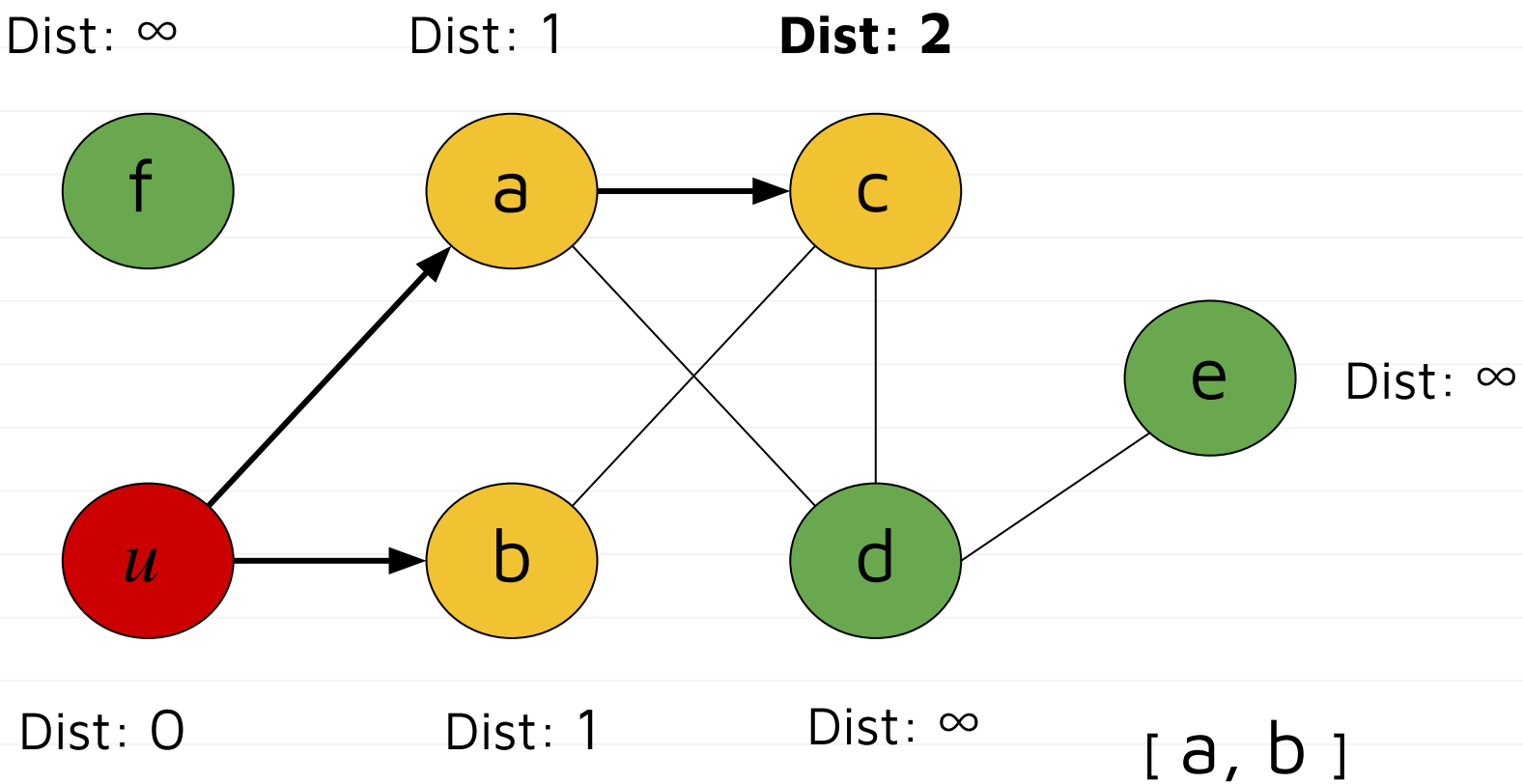
predec = {'a': 'u', 'b': 'u'



[a, b]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2

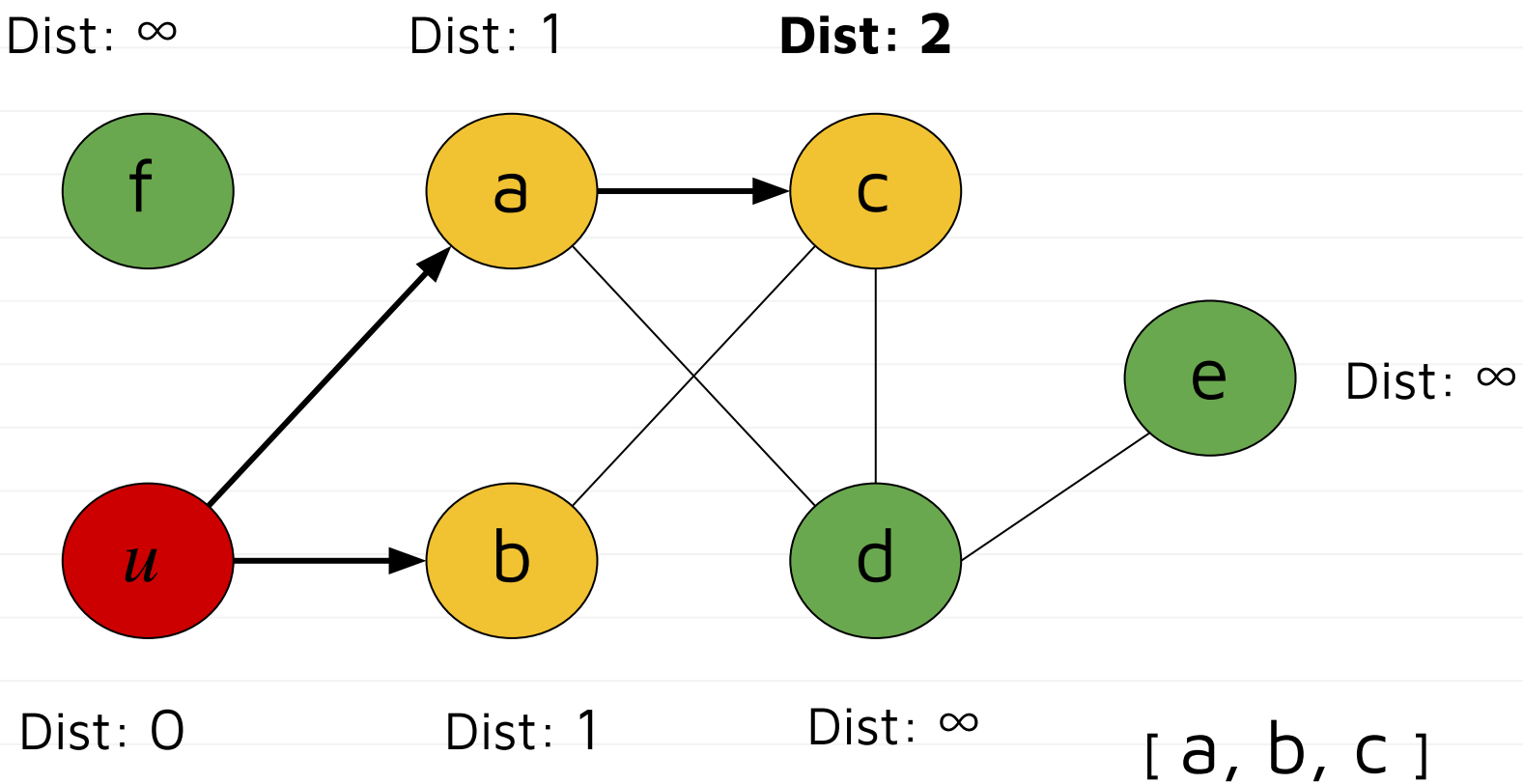
predec = {'a': 'u', 'b': 'u', 'c': 'a',



[a, b]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2

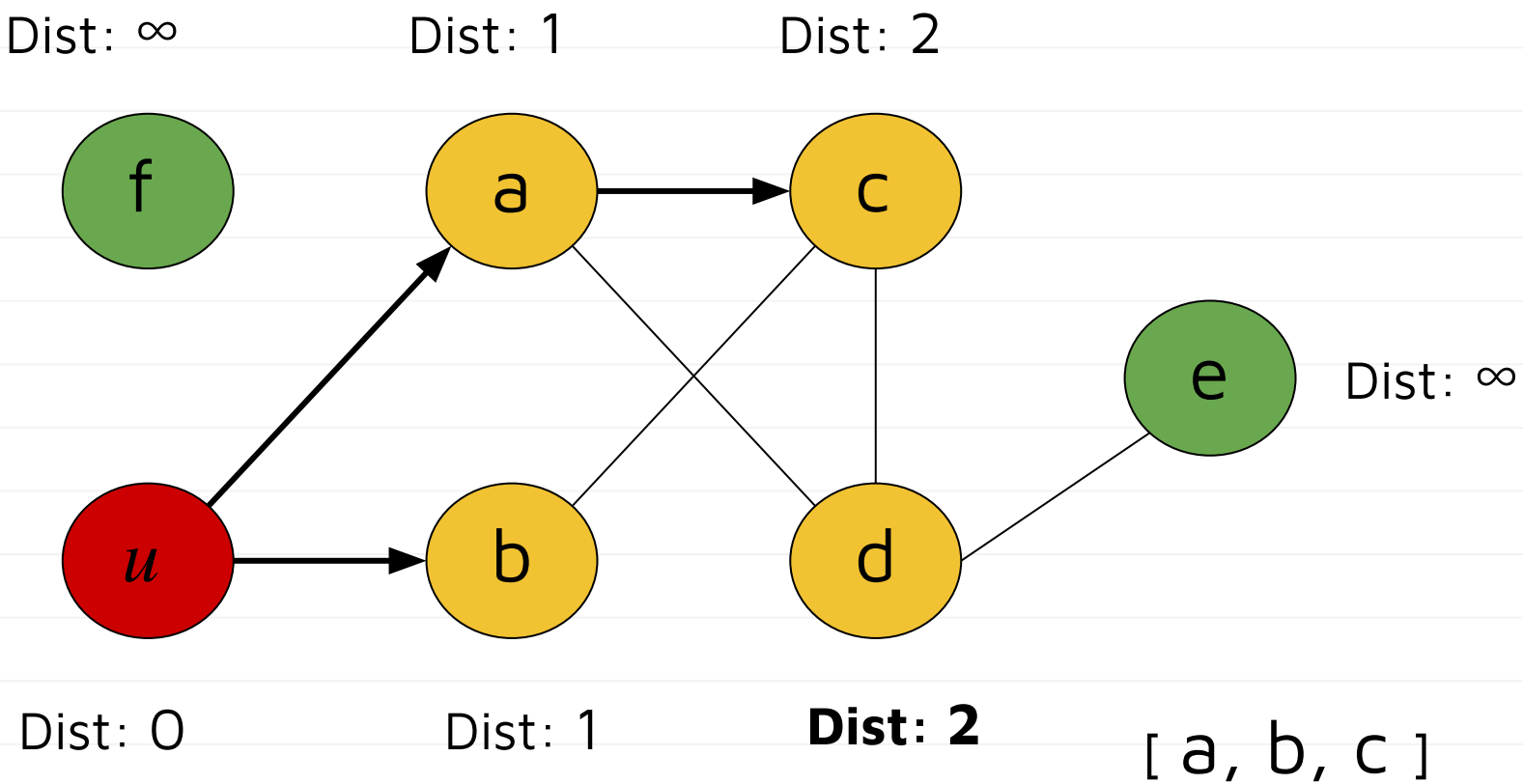
predec = {'a': 'u', 'b': 'u', 'c': 'a',



[a, b, c]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2

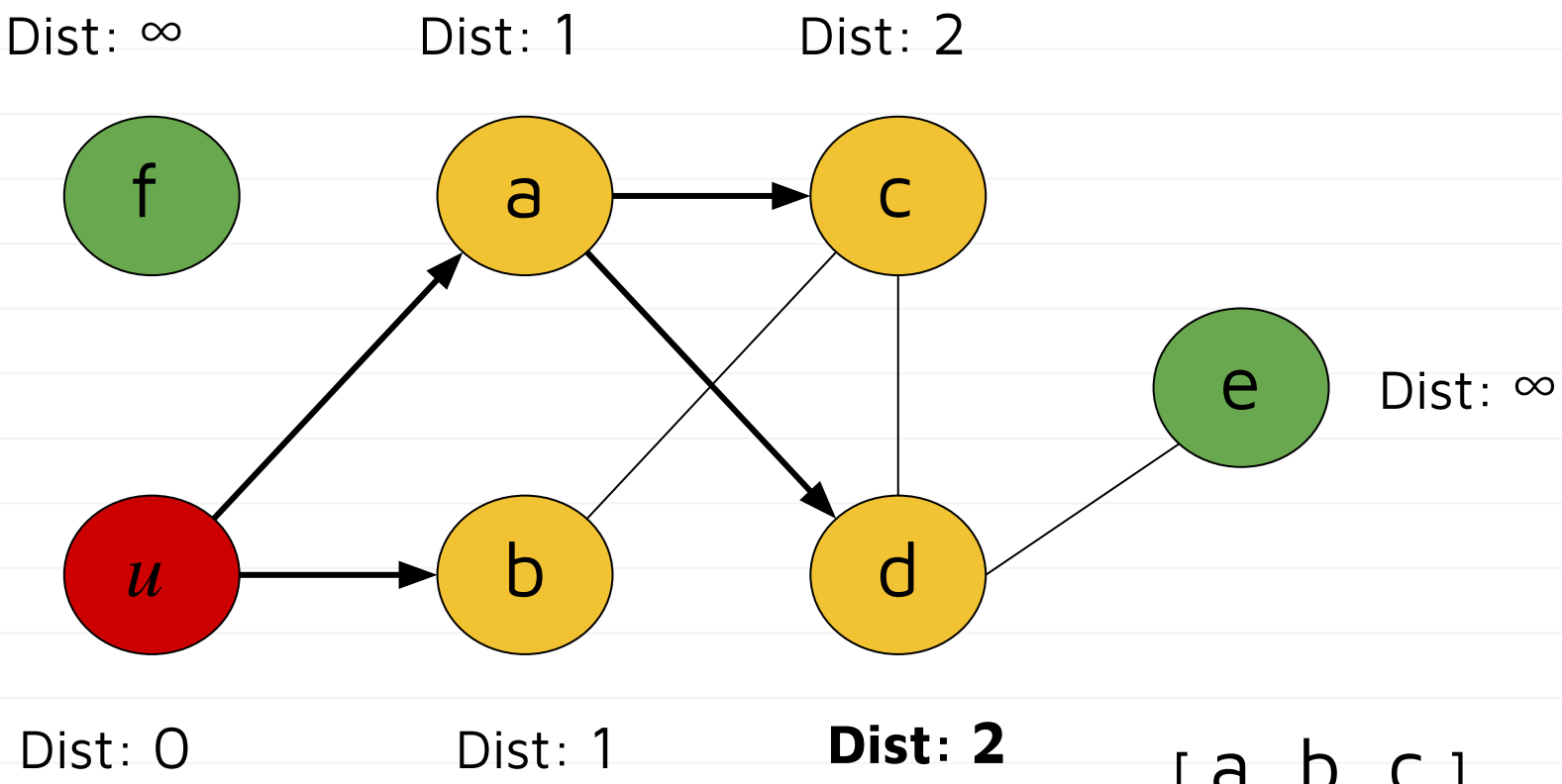
predec = {'a': 'u', 'b': 'u',
'c': 'a',



[a, b, c]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2

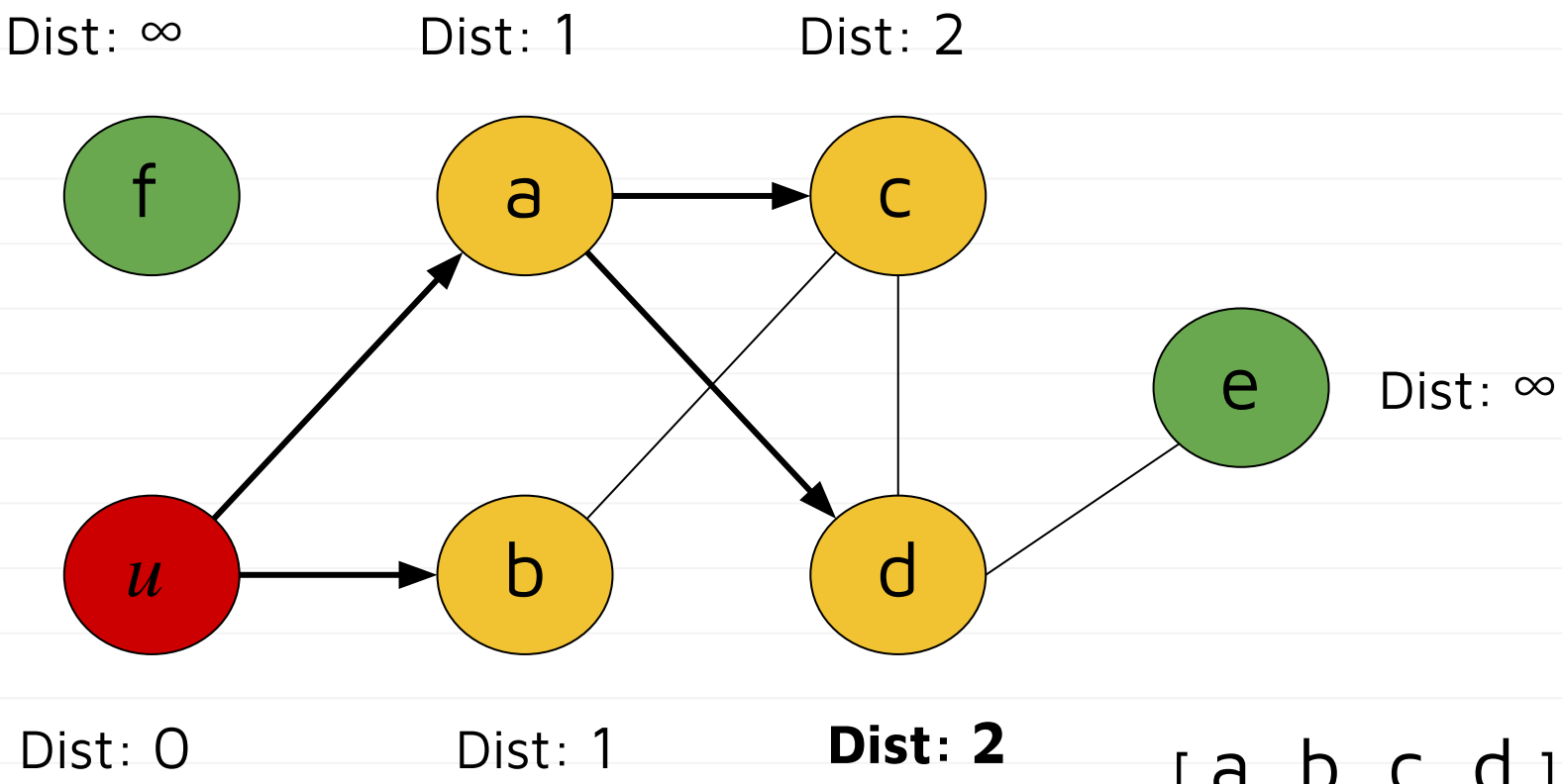
predec = {'a': 'u', 'b': 'u',
'c': 'a', 'd': 'a',



[a, b, c]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2

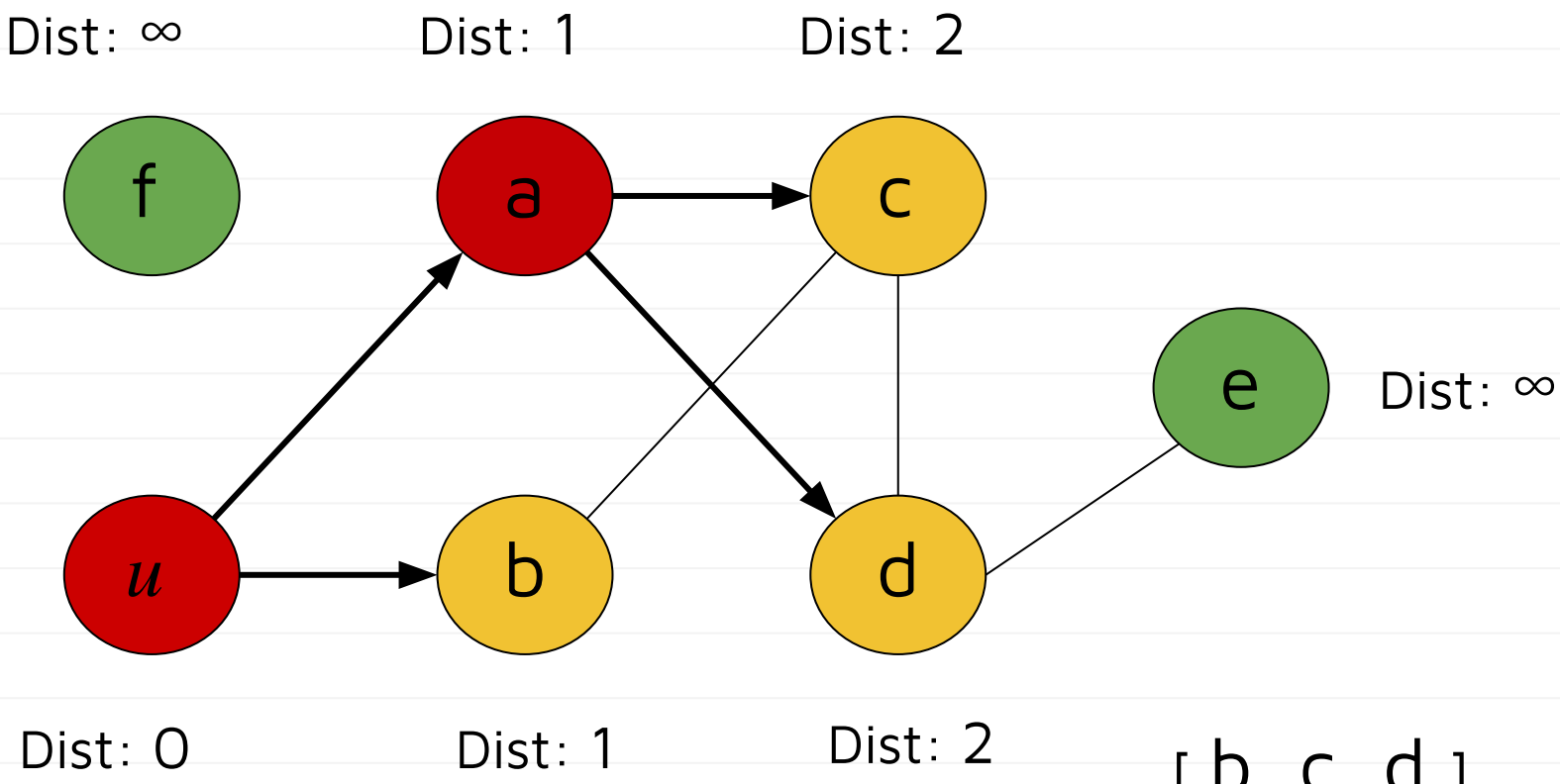
predec = {'a': 'u', 'b': 'u',
'c': 'a', 'd': 'a',



[a, b, c, d]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2

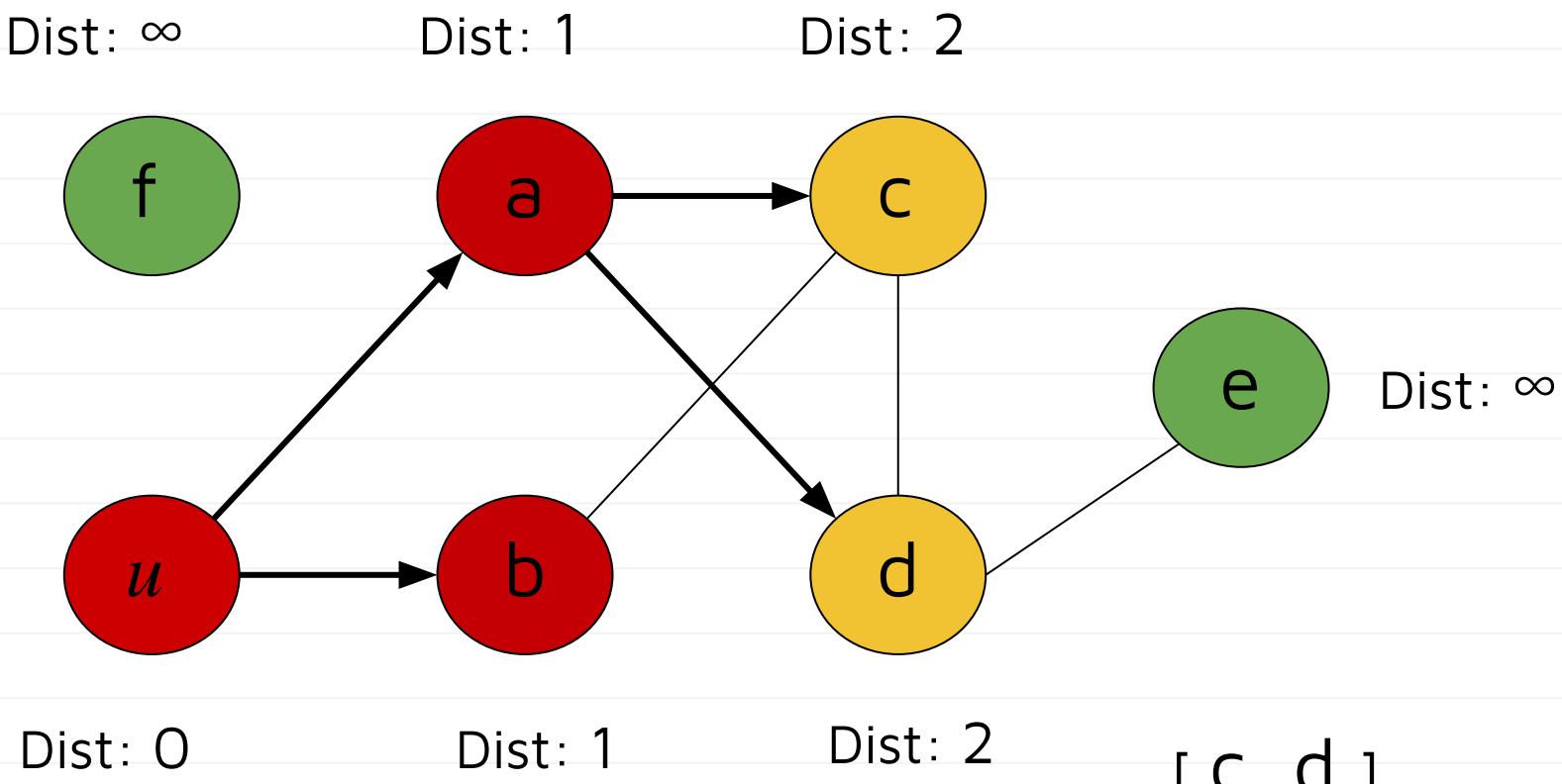
predec = {'a': 'u', 'b': 'u',
'c': 'a', 'd': 'a',



[b, c, d]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2

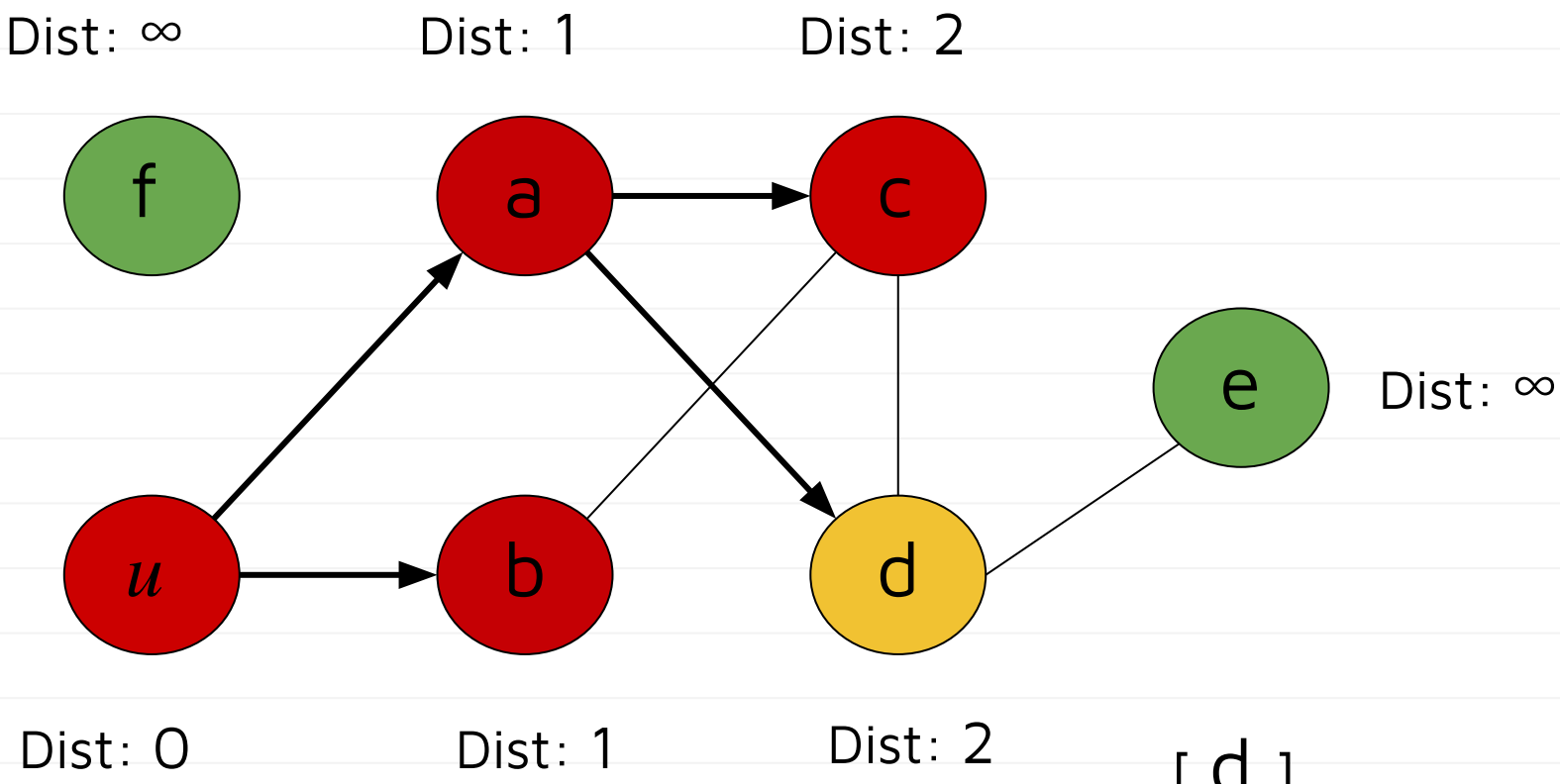
predec = {'a': 'u', 'b': 'u',
'c': 'a', 'd': 'a',



[c, d]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2

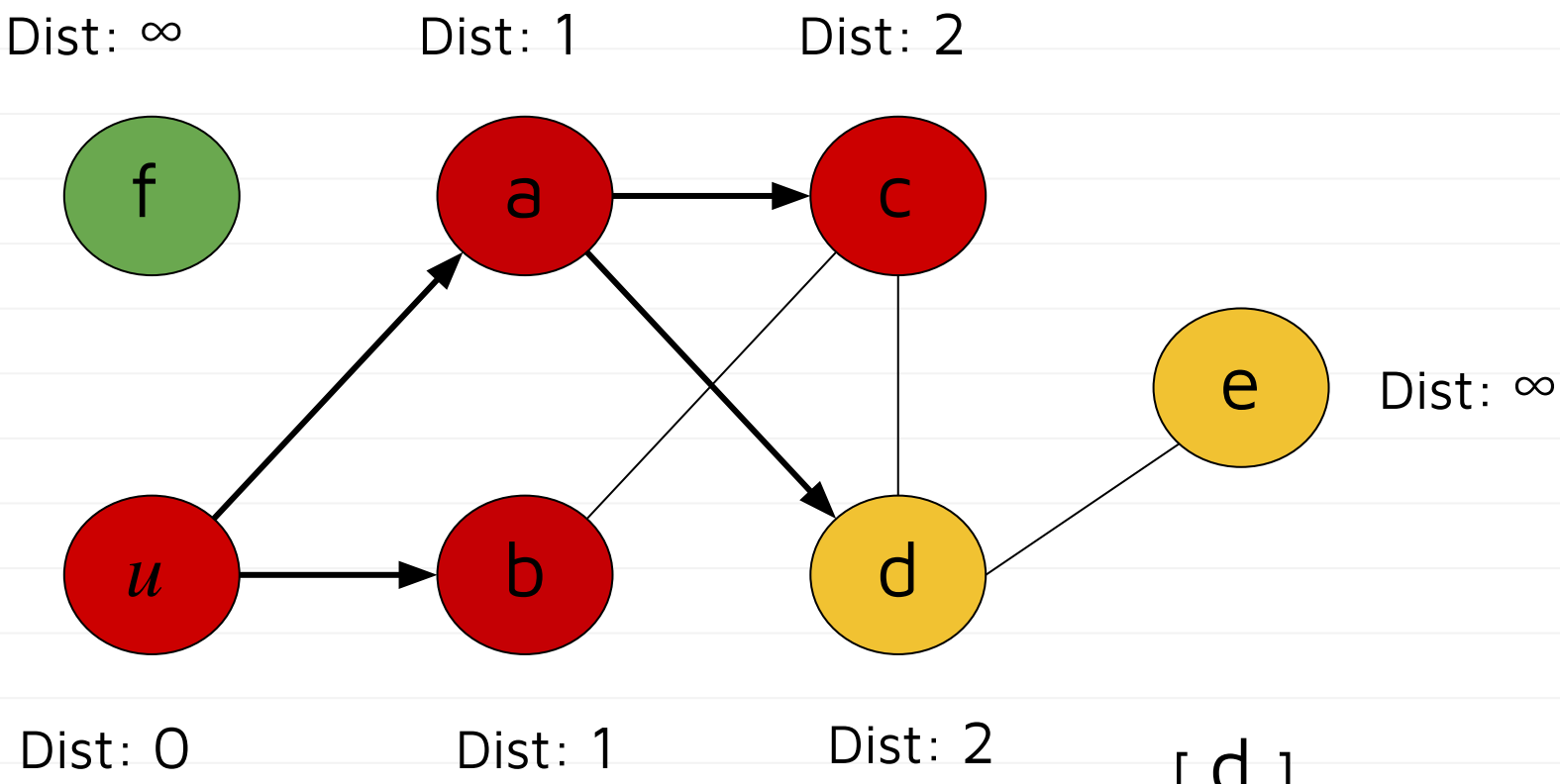
predec = {'a': 'u', 'b': 'u',
'c': 'a', 'd': 'a',



[d]

distance = { 'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2

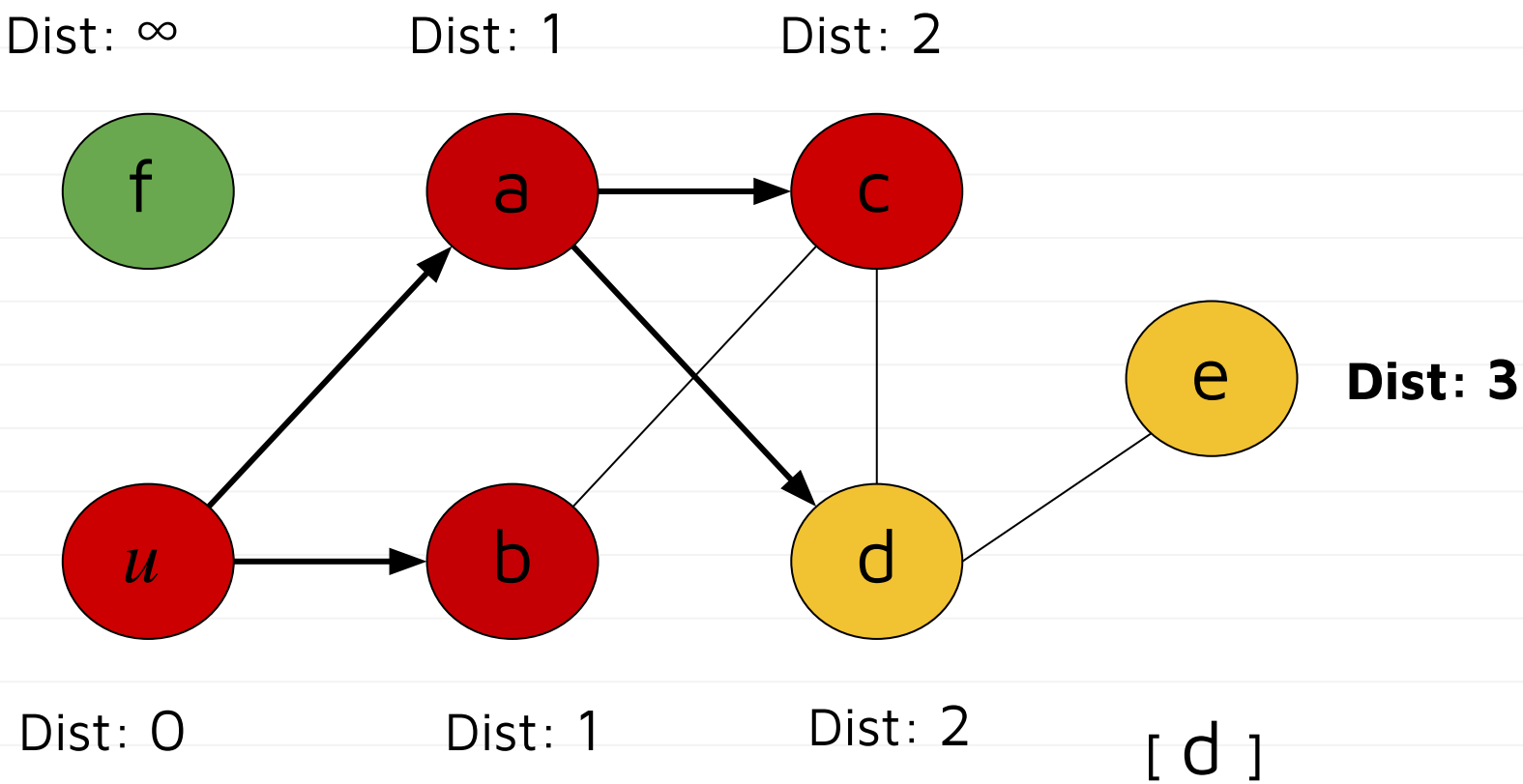
predec = {'a': 'u', 'b': 'u',
'c': 'a', 'd': 'a',



[d]

distance = {'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2, 'e': 3}

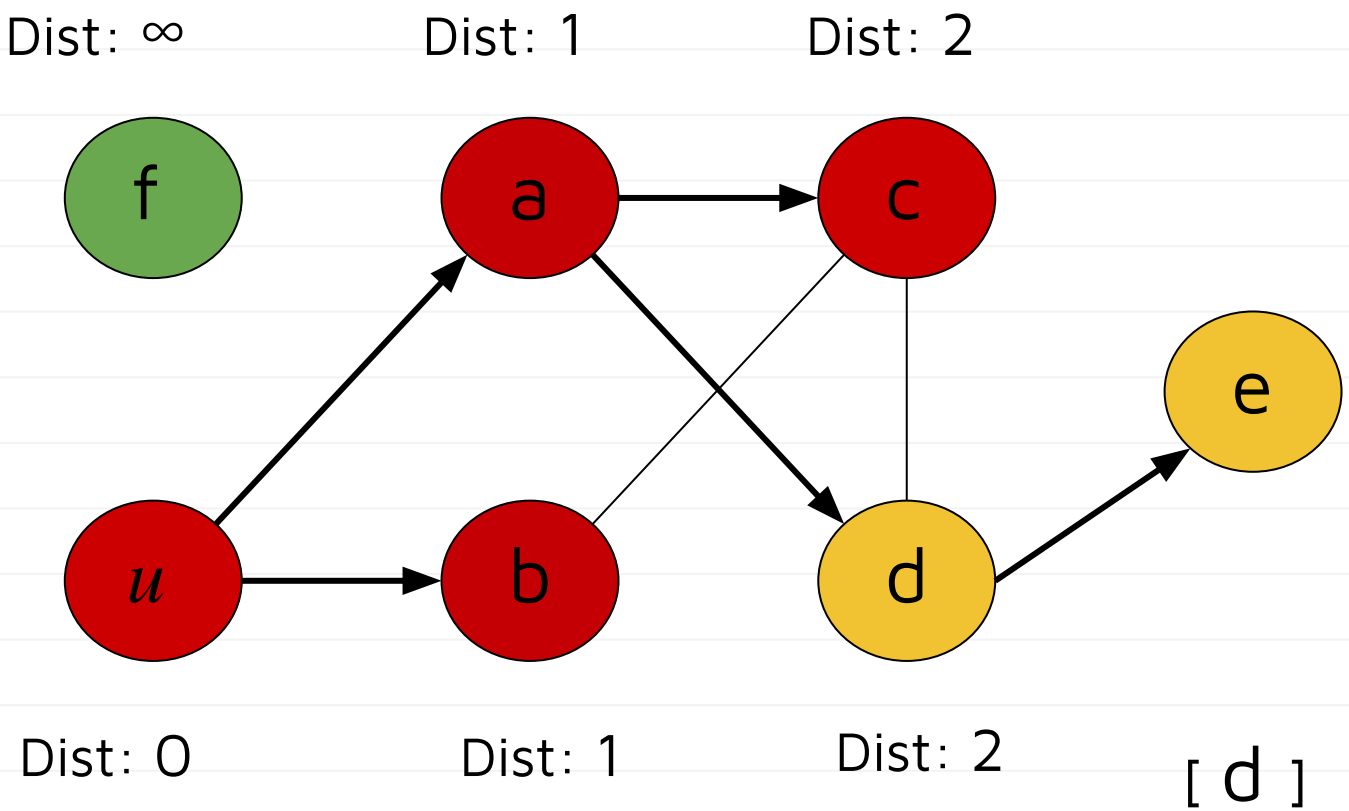
predec = {'a': 'u', 'b': 'u',
'c': 'a', 'd': 'a',



[d]

distance = {'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2, 'e': 3}

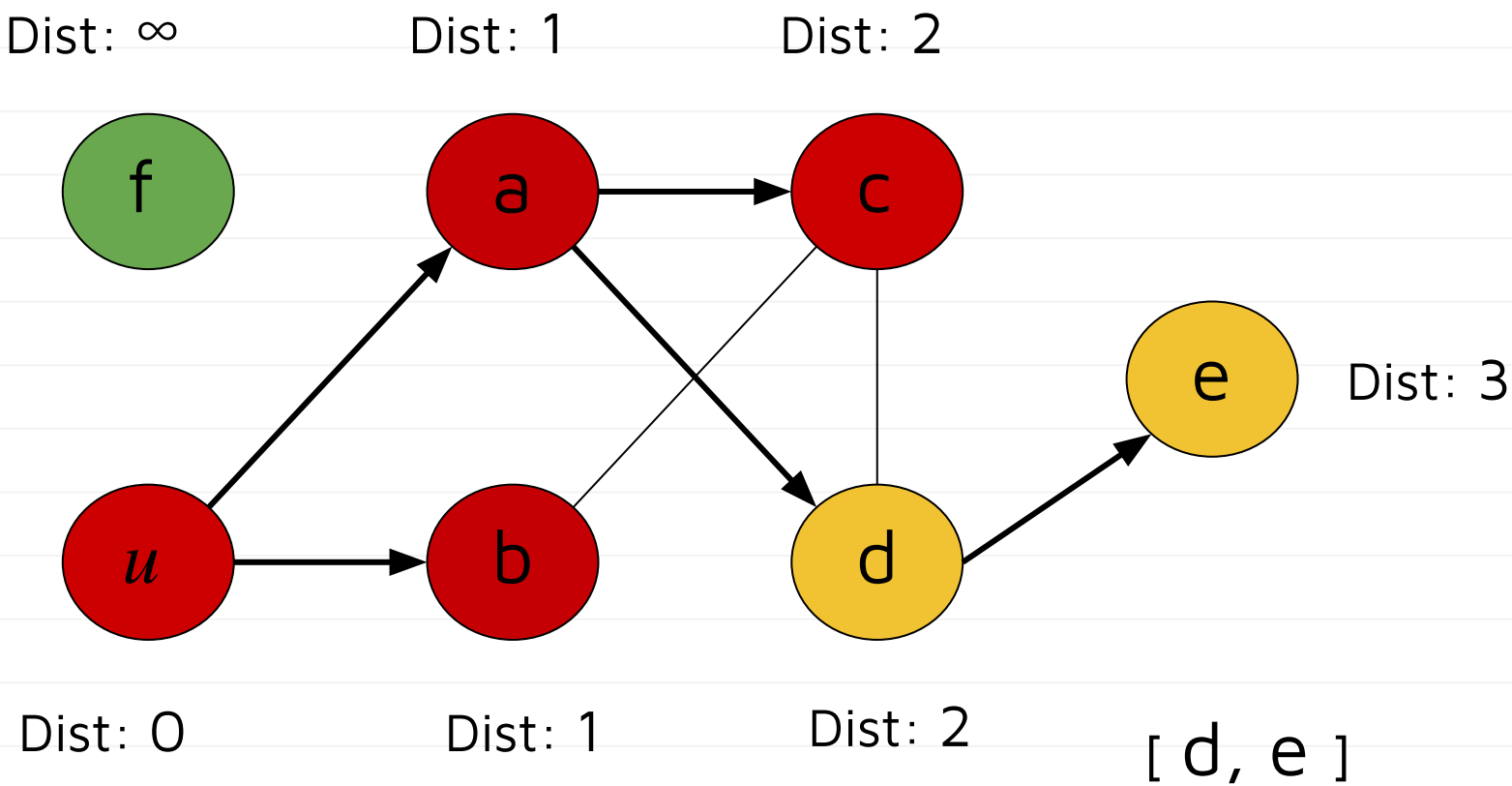
predec = {'a': 'u', 'b': 'u', 'c': 'a', 'd': 'a', 'e': 'd'}



[d]

distance = {'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2, 'e': 3}

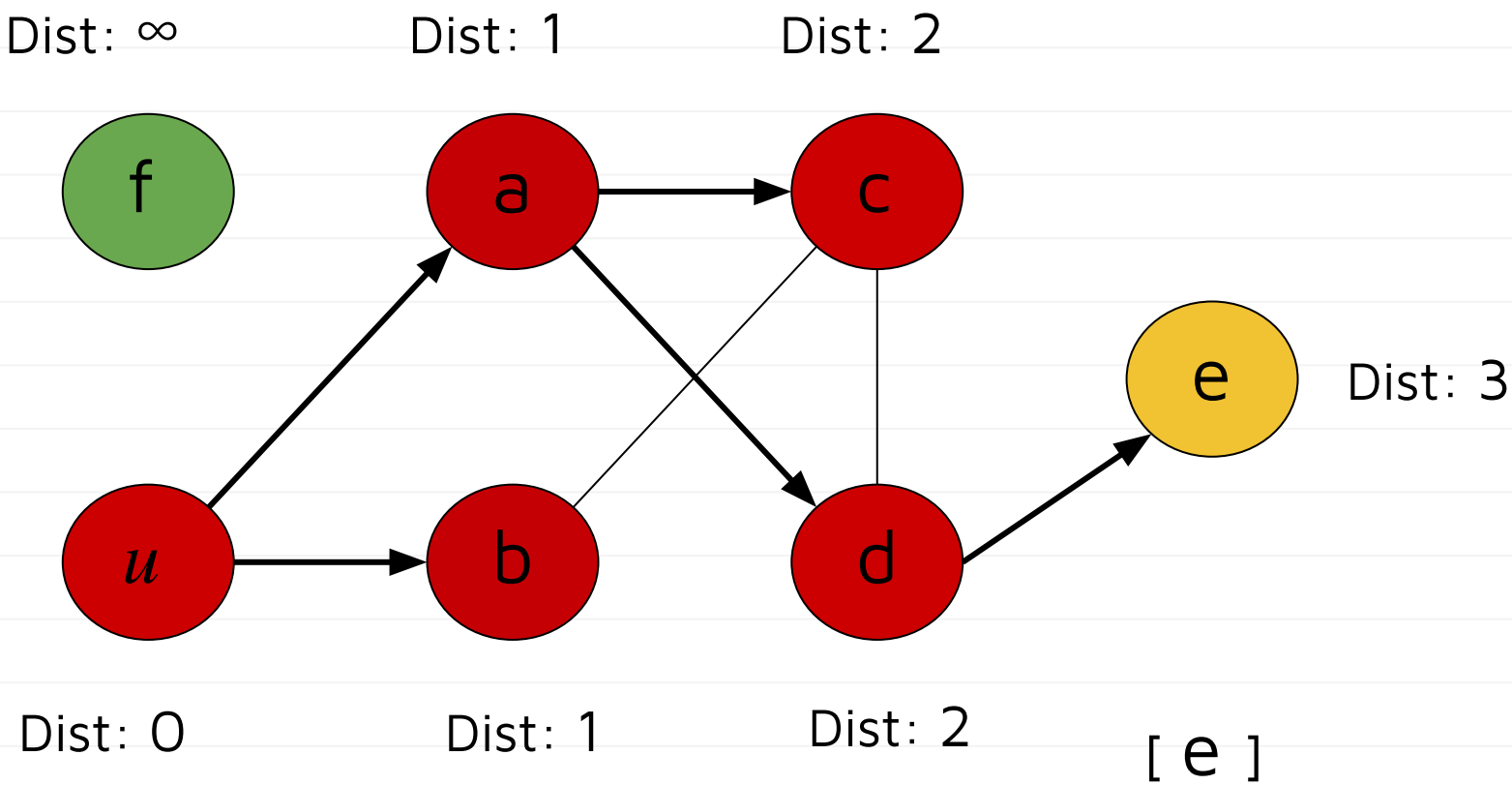
predec = {'a': 'u', 'b': 'u', 'c': 'a', 'd': 'a', 'e': 'd'}



[d, e]

distance = {'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2, 'e': 3}

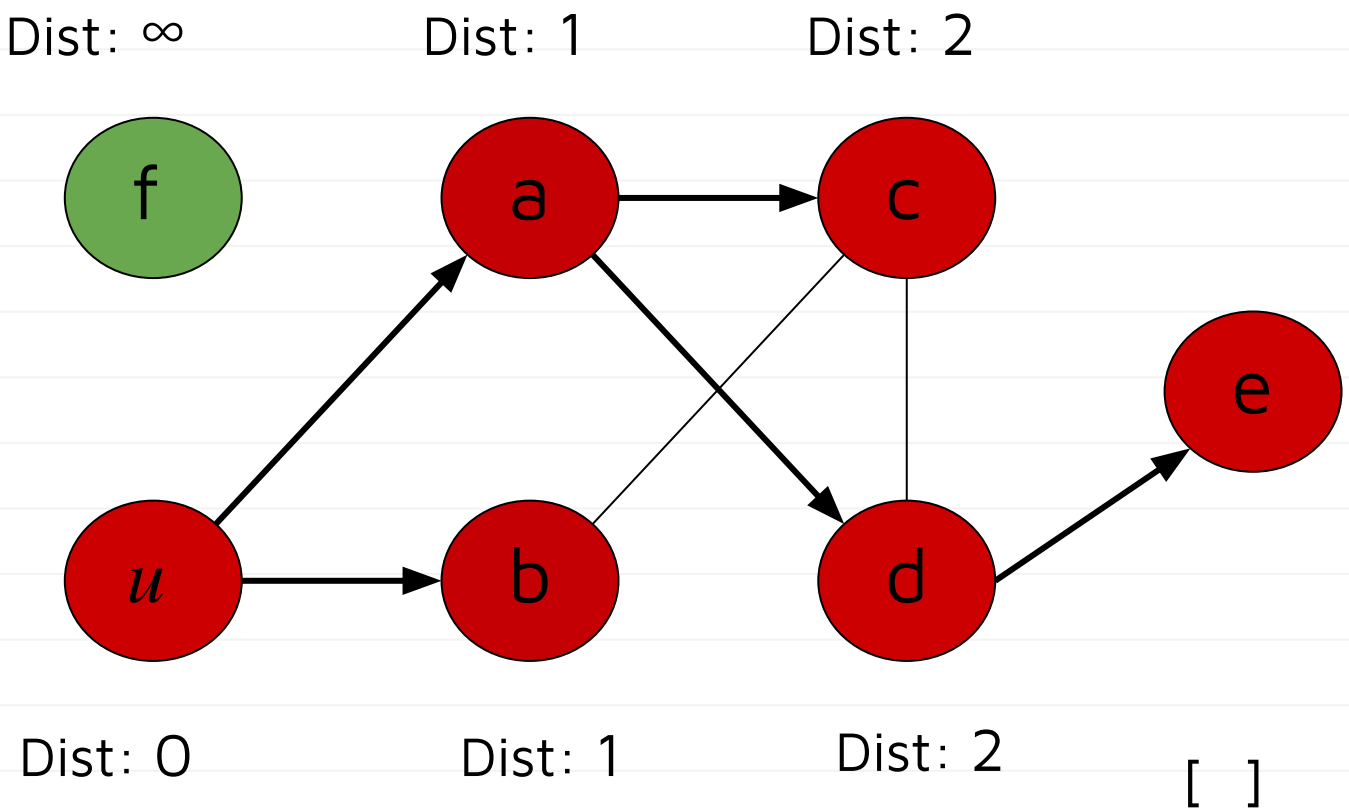
predec = {'a': 'u', 'b': 'u', 'c': 'a', 'd': 'a', 'e': 'd'}



[e]

distance = {'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2, 'e': 3}

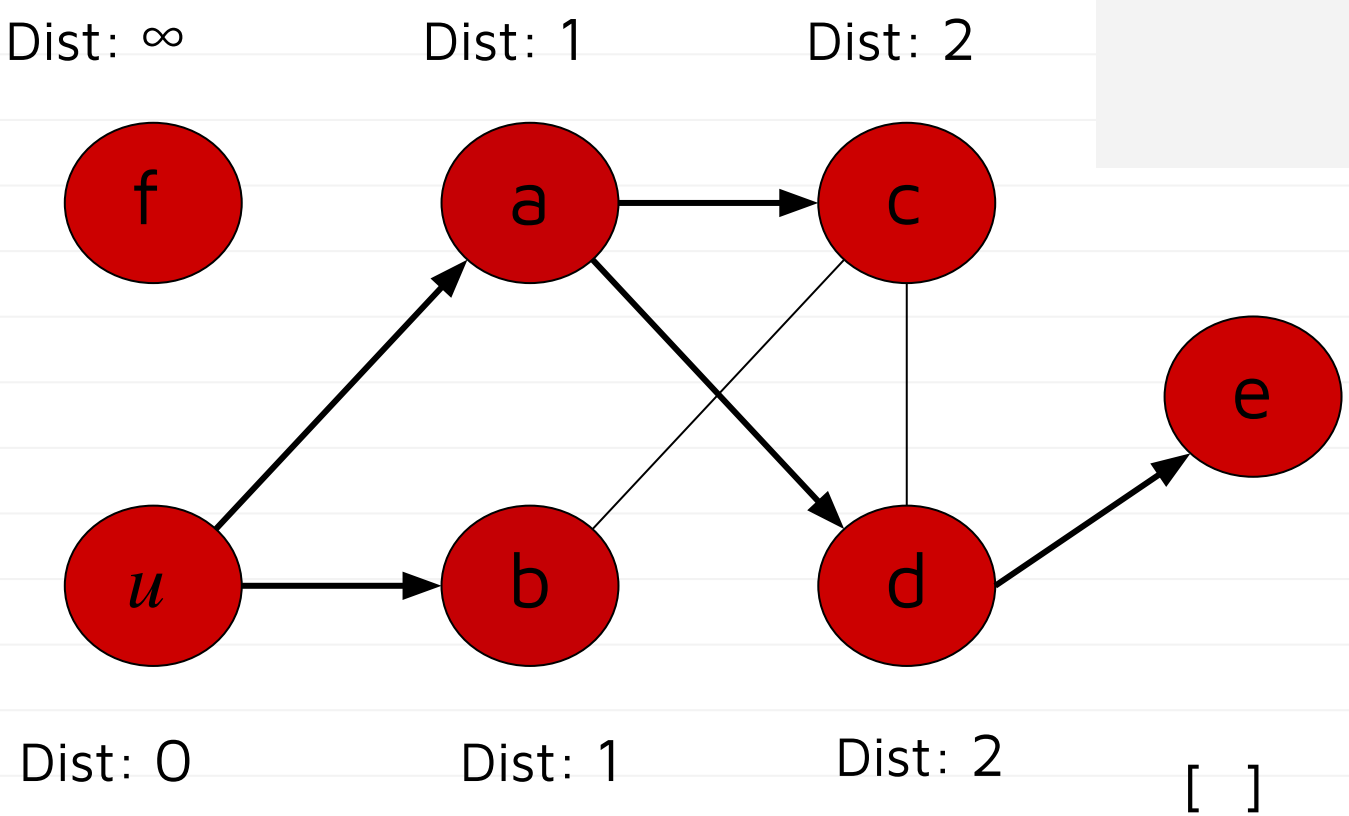
predec = {'a': 'u', 'b': 'u', 'c': 'a', 'd': 'a', 'e': 'd'}



[ ]

distance = {'u': 0, 'a': 1, 'b': 1, 'c':2, 'd': 2, 'e': 3, 'f' : ∞ }

predec = {'a': 'u', 'b': 'u', 'c': 'a', 'd': 'a', 'e': 'd', 'f': None, 'u': None}



[ ]

BFS Trees

Result of BFS

- Each node reachable from source has a single BFS predecessor.
  - Except for the source itself.
- The result is a **tree** (or forest).

Trees

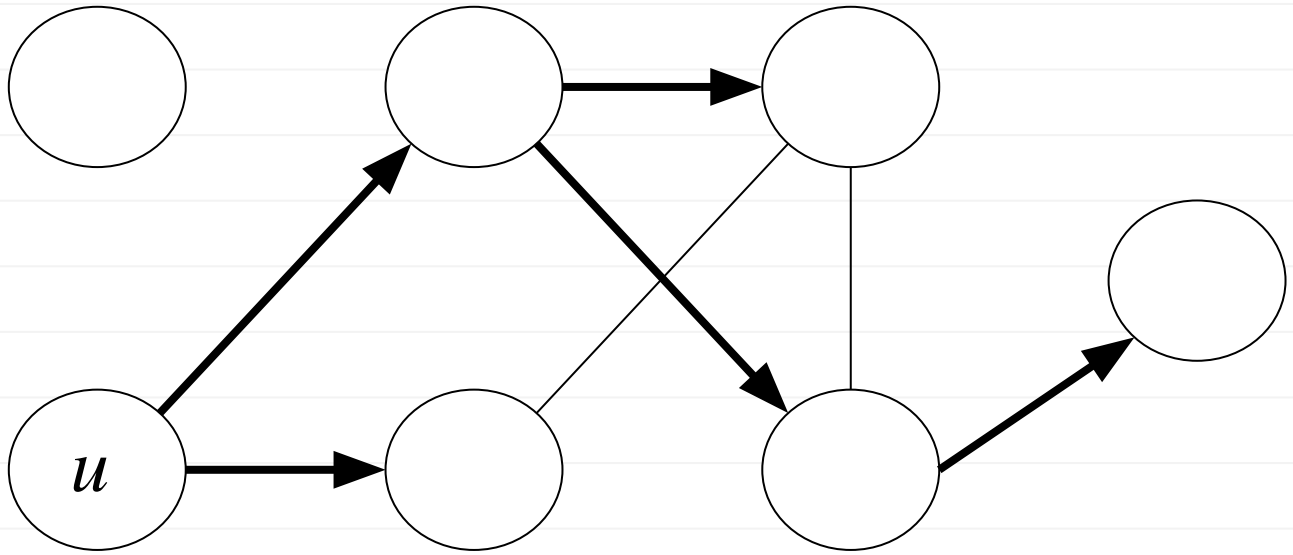
- A (free) **tree** is an undirected graph $T = (V, E)$ such that T is *connected* and $|E| = |V| - 1$.
- A **forest** is graph in which each connected component is a tree.

BFS Trees (Forests)

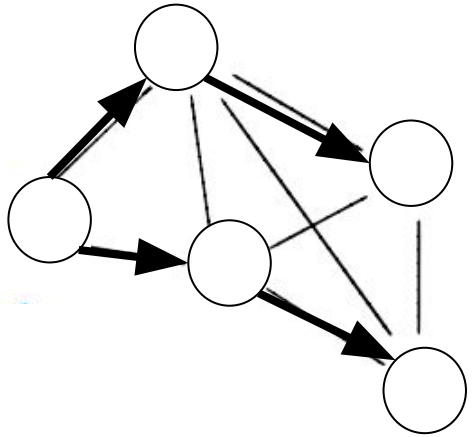
- If the input is connected, BFS produces a **tree**.
- If the input is not connected, BFS produces a **forest**.

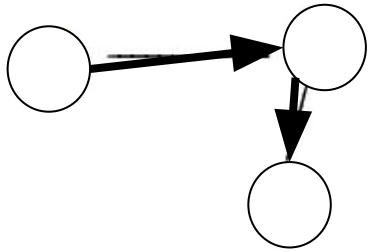
Example

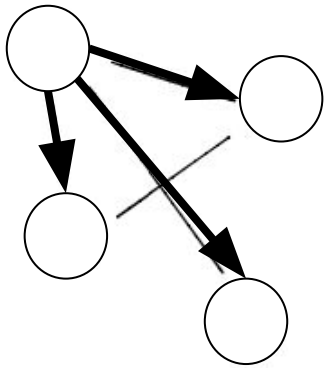
How many nodes?
How many edges?



Example







BFS Trees

- BFS trees and forests encode **shortest path distances.**

Thank you!

Do you have any questions?

CampusWire!