Introduction to Algorithms

Mong-Jen Kao (高孟駿)

Tuesday 10:10 – 12:00

Thursday 15:30 – 16:20

Graph Algorithms

– Graph problem pervades computer science, and Algorithms for graphs are fundamental to the field.

Graph Traversal / Searching

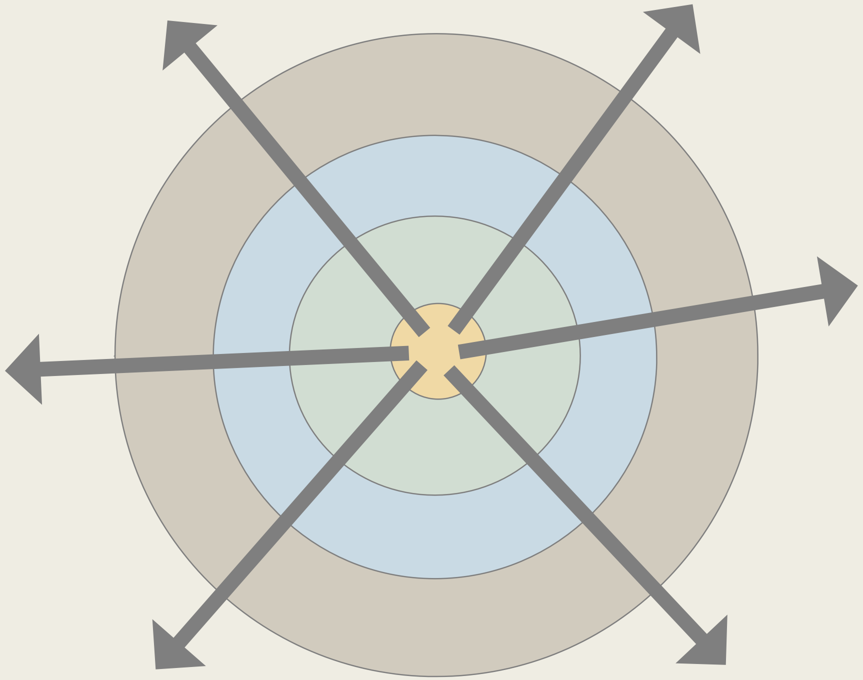
To explore / search the vertices / edges in a graph in a systematic way.

Graph Traversal

- Traversal / Searching is a fundamental problem in graphs.
  - To explore every vertex / edge in the graph
  - To search for a particular vertex / edge in the graph

- In this lecture, we examine two different ways to do this.
  1. ***<u>Breadth-first search (BFS)</u>*** – which reveals the shortest-path map / distance information for the source vertex.
  2. ***<u>Depth-first search (DFS)</u>*** – which reveals certain structural properties / information of the graph.

Breadth-First Search (BFS)

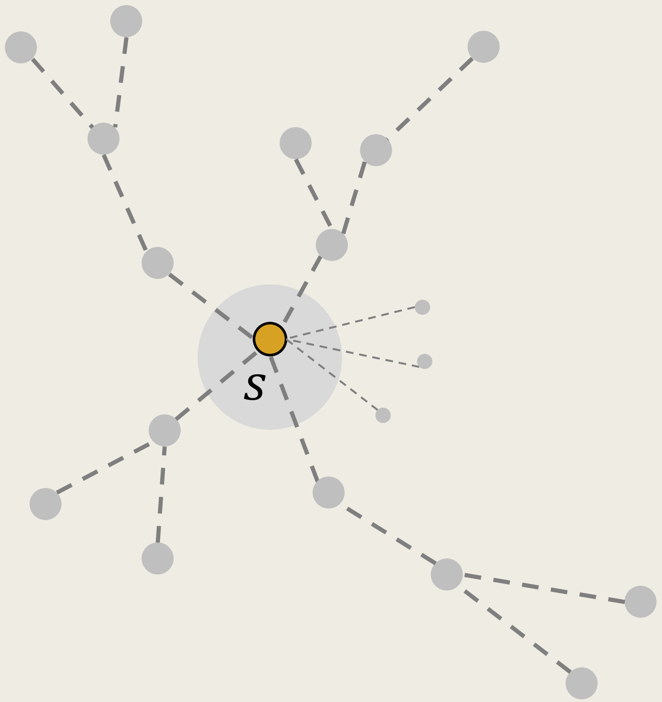


During the process, the shortest-path map from the source vertex is revealed.

To explore the vertices in an **equidistant contour** (concentric) order.

Breadth-First Search (BFS)

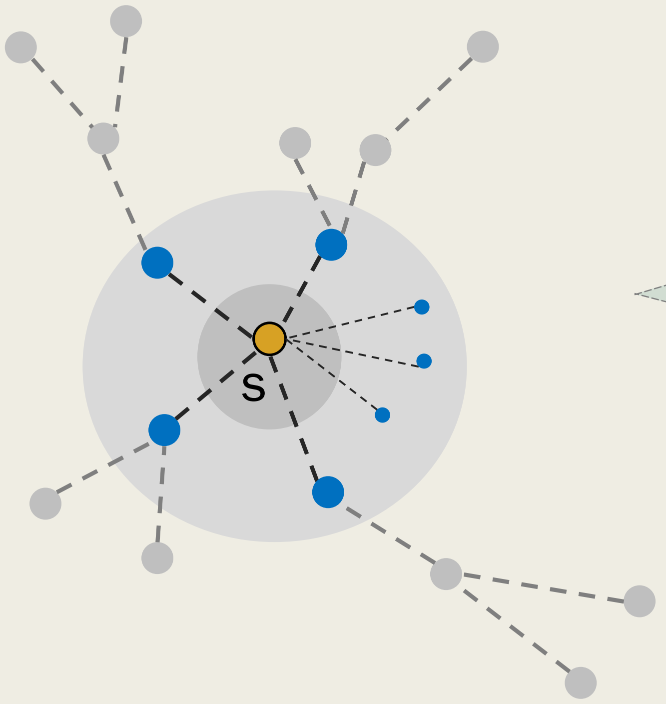
- This process explores the vertices in the order of their (shortest) distances to the source vertex s.



The process starts from the source vertex s.

Breadth-First Search (BFS)

- This process explores the vertices in the order of their (shortest) distances to the source vertex s.

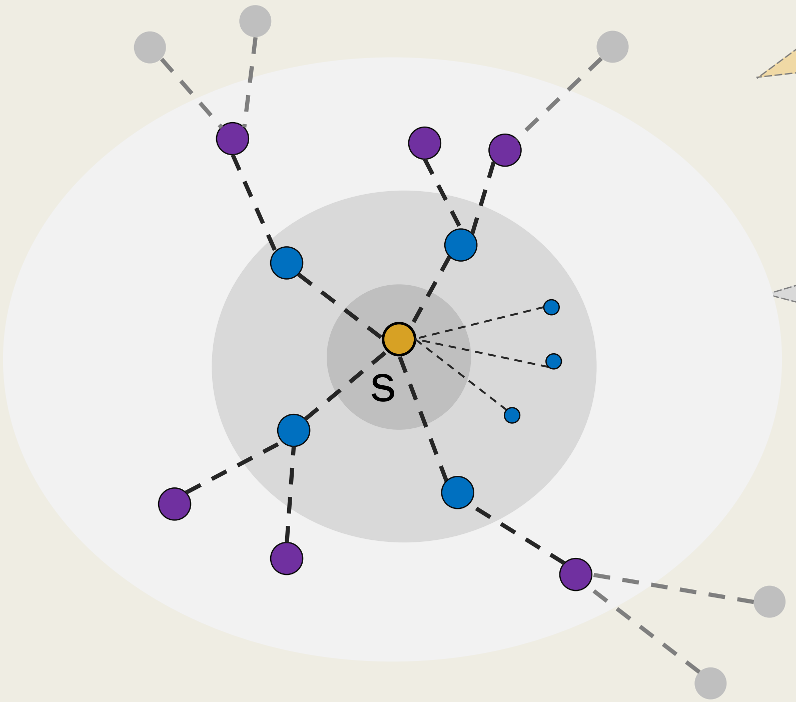


Next, the vertices with distance 1 to s are explored.

The process starts from the source vertex s.

Breadth-First Search (BFS)

- This process explores the vertices in the order of their (shortest) distances to the source vertex s.



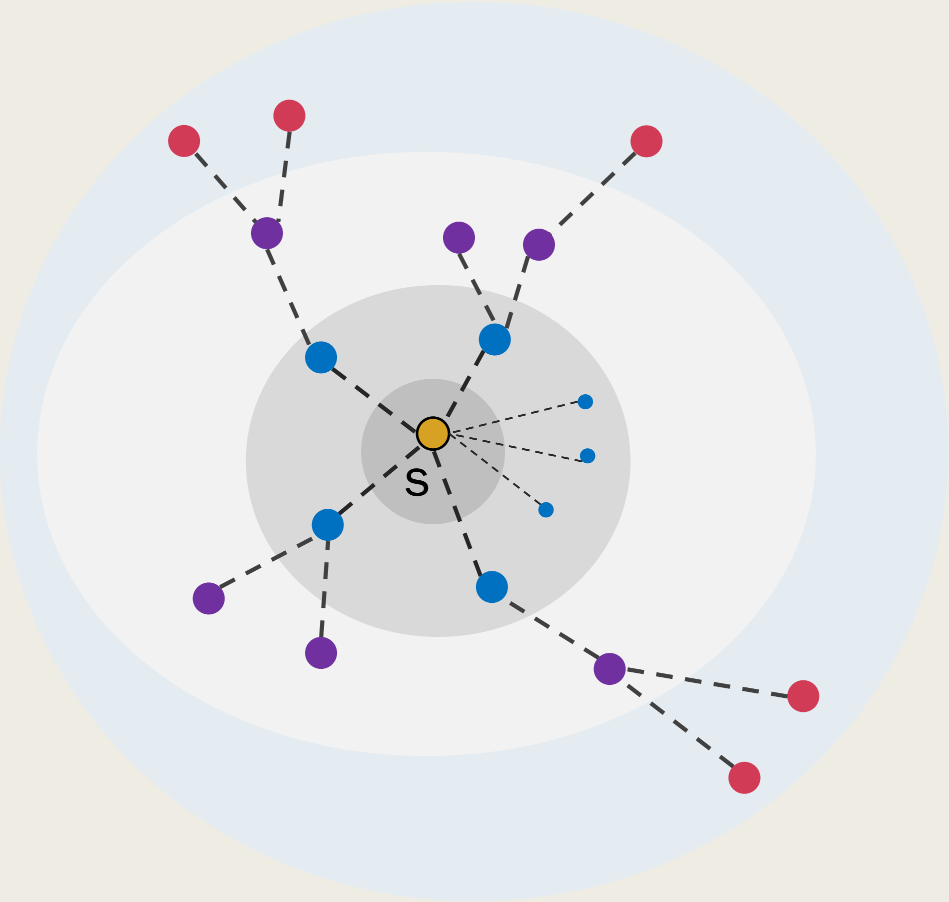
Next, the vertices with distance 2 to s are explored.

Next, the vertices with distance 1 to s are explored.

The process starts from the source vertex s.

Breadth-First Search (BFS)

- This process explores the vertices in the order of their (shortest) distances to the source vertex s.

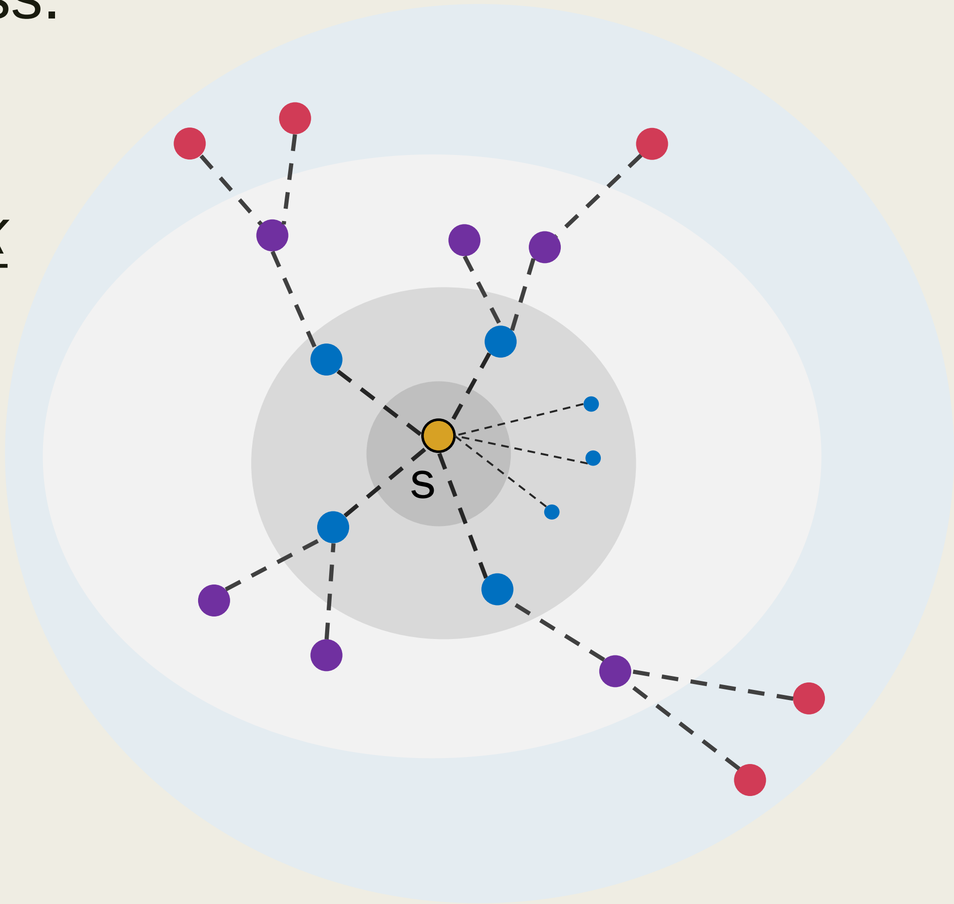


- The vertices discovered at each level forms ***equidistant contours*** concentered at the source vertex s.

- A ***shortest-path tree*** (SPT) from the source vertex s is produced.

The BFS Algorithm

- The algorithm uses a ***first-in first-out (FIFO) queue*** to implement the aforementioned process.
  - In each iteration, the algorithm *extracts the first vertex* in the queue and process it.
  - For each vertex discovered (*which belongs to the next contour*), the algorithm *appends the vertex to the tail of the queue*.



Formal Description of the BFS Algorithm

- The algorithm maintains the following information during the process.
  - $\forall v \in V$, the color (status) of v, denoted $\mathrm{color}[v]$.
    - white: not discovered yet.
    - gray: discovered, not yet processed.
    - black: discovered and processed.
  - $\forall v \in V$, the predecessor (parent) of v, denoted $\pi[v]$, in the search.
    - i.e., the vertex that discovers v during the search.
    - $\pi[v]$ is **NIL** if v has no predecessor (yet).
      - v is the source vertex, or v is not yet discovered.

Formal Description of the BFS Algorithm

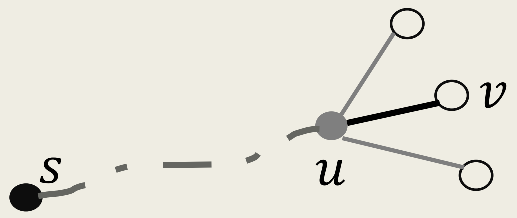
- The algorithm maintains the following information during the process.
  - $\forall v \in V$, the distance to the source vertex, denoted $\mathrm{d}[v]$.
    - Initially, $\mathrm{d}[v] = \infty$ for all $v \in V$.
  - A first-in first-out (FIFO) queue Q.
    - The queue is used to store the current gray vertices in the order they are discovered by the algorithm.

- BFS(G, s) - $G = (V, E)$ the input graph, $s \in V$ the start vertex.

A. For each $v \in V$,
   set $\text{color}[v] \leftarrow \text{white}$, $d[v] \leftarrow \infty$, and $\pi[v] \leftarrow \text{NIL}$.

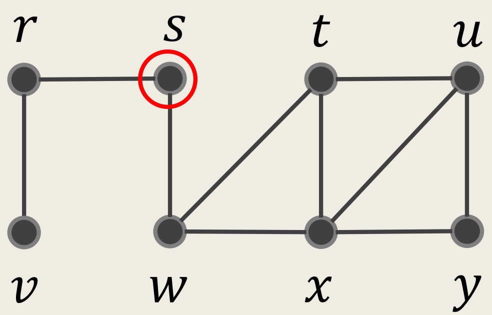
B. Set $\text{color}[s] \leftarrow \text{gray}$ and $d[s] \leftarrow 0$.
   $\text{ENQUEUE}(Q, s)$.

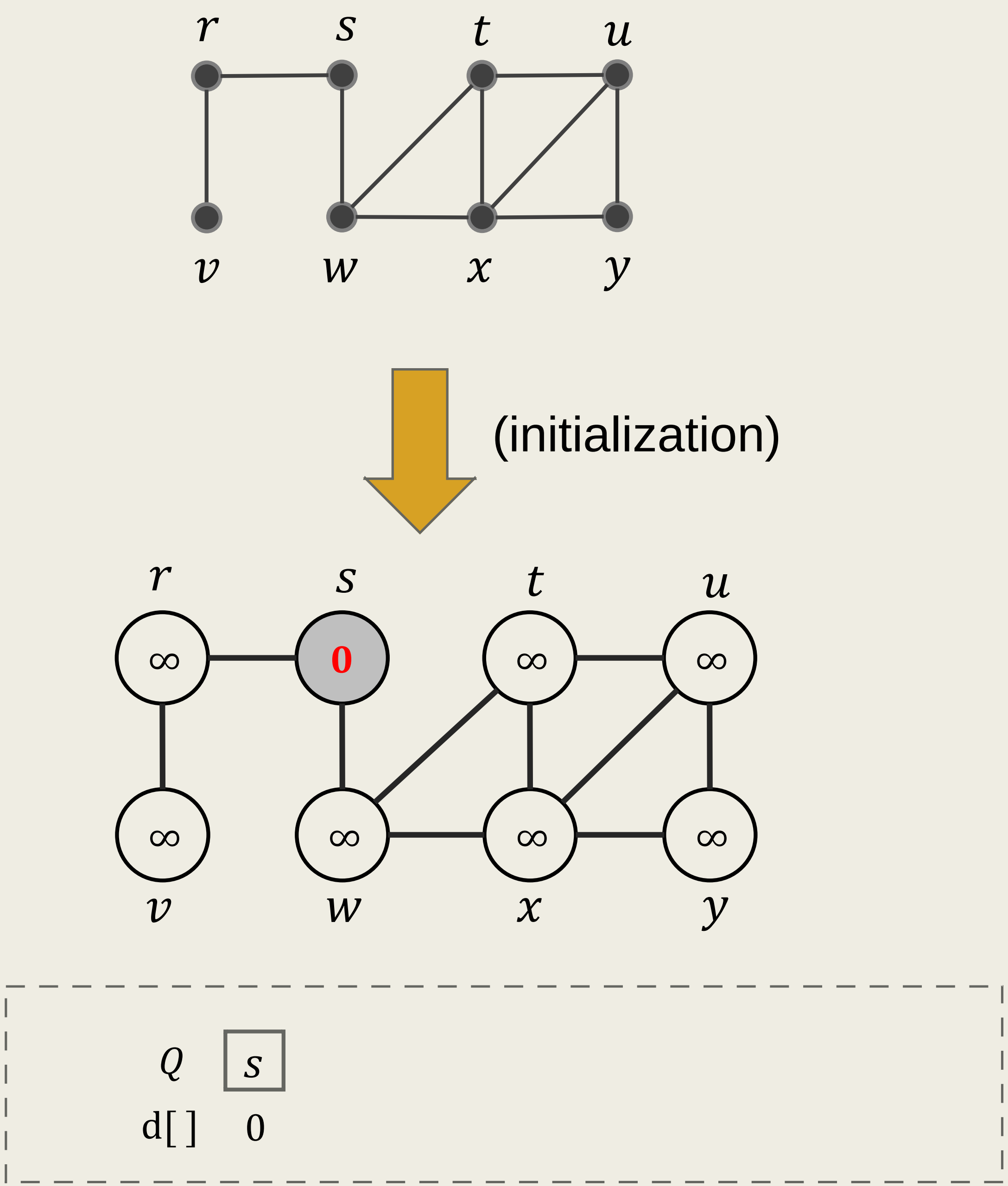
C. While $Q \neq \emptyset$, do the following.
   - $u \leftarrow \text{DEQUEUE}(Q)$.
   - For each $v \in N[u]$, do the following.
     - If $\text{color}[v] = \text{white}$, then
       - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
         $ENQUEUE(Q, v)$.
   - Set $\text{color}[u] \leftarrow \text{black}$.



An Example

- Consider the following graph and
  the execution of the BFS algorithm with the source vertex s.





- Initialization

A. For each $v \in V$, set

color$[v] \leftarrow$ white,

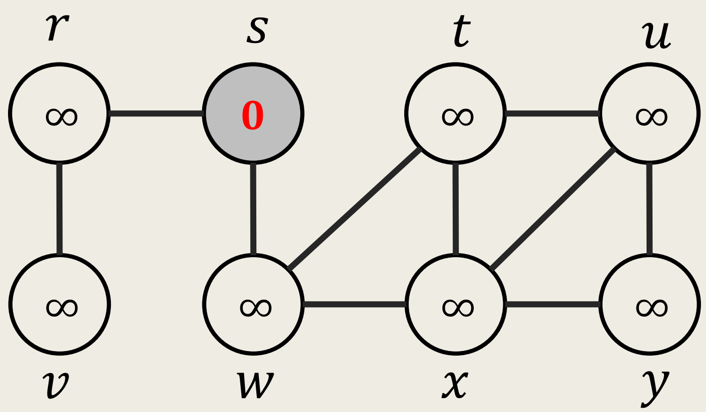
$d[v] \leftarrow \infty$, and

$\pi[v] \leftarrow$ NIL.

B. Set color$[s] \leftarrow$ gray and

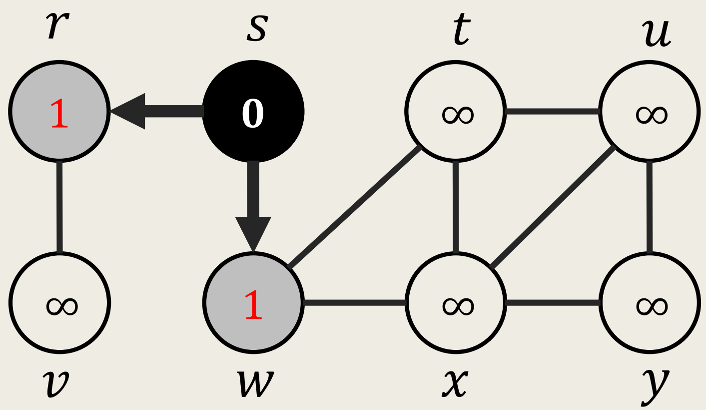
$d[s] \leftarrow 0$.

ENQUEUE(Q, s).



Q	s
d[ ]	0

process (expand) s

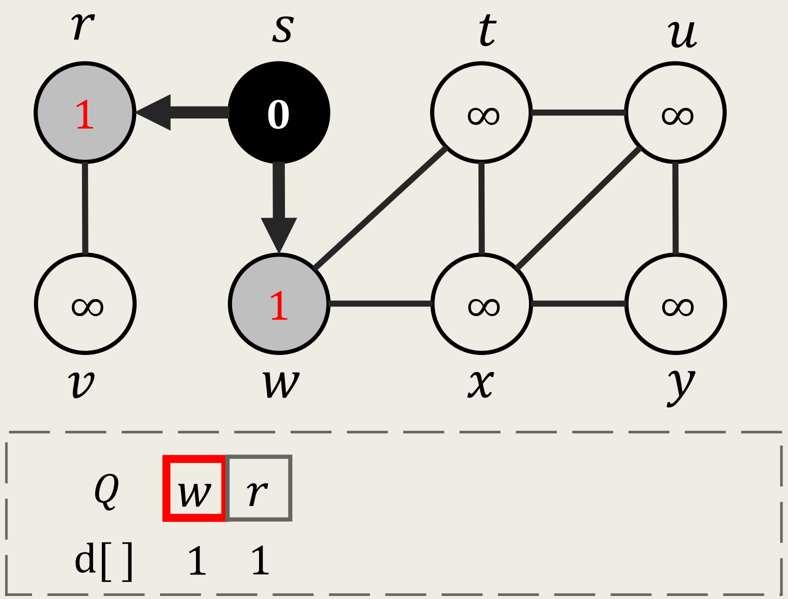


Q	w	r
d[ ]	1	1

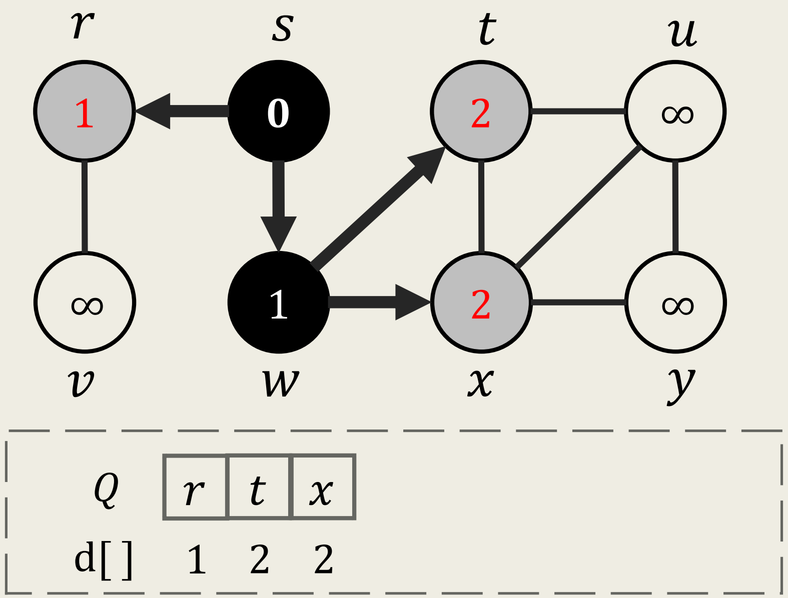
- Process the gray nodes until Q becomes empty.

A. While $Q \neq \emptyset$, do the following.
   - $u \leftarrow \text{DEQUEUE}(Q)$.
   - For each $v \in N[u]$, do
     - If $\text{color}[v] = \text{white}$, then
       - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
       - $\text{ENQUEUE}(Q, v)$.
   - Set $\text{color}[u] \leftarrow \text{black}$.

Discovery of distance-1 vertices is done.

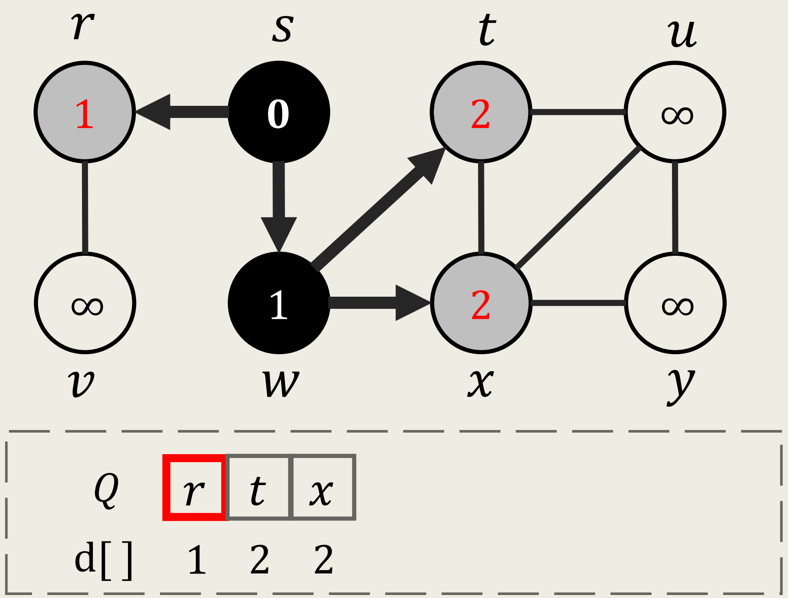


process (expand) w

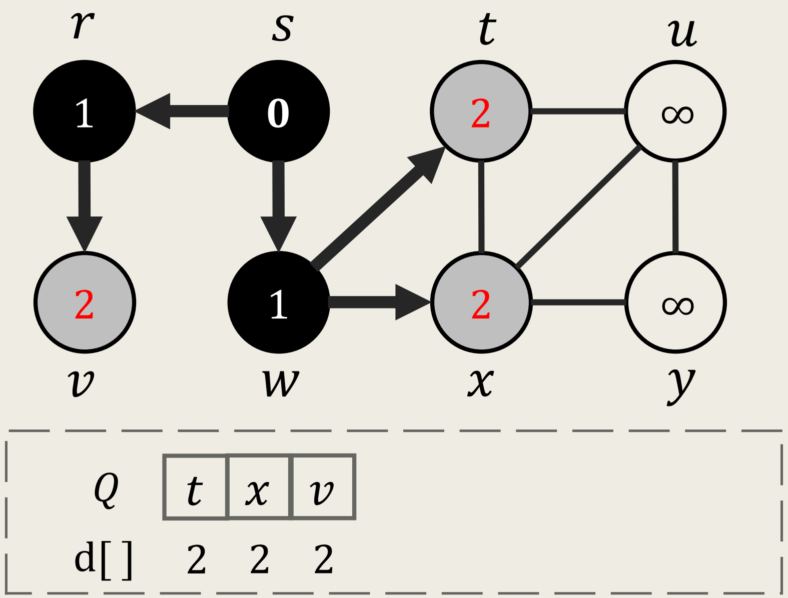


- Process the gray nodes until Q becomes empty.

A. While $Q \neq \emptyset$, do the following.
  - $u \leftarrow \text{DEQUEUE}(Q)$.
  - For each $v \in N[u]$, do
    - If $\text{color}[v] = \text{white}$, then
      - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
      - $\text{ENQUEUE}(Q, v)$.
  - Set $\text{color}[u] \leftarrow \text{black}$.



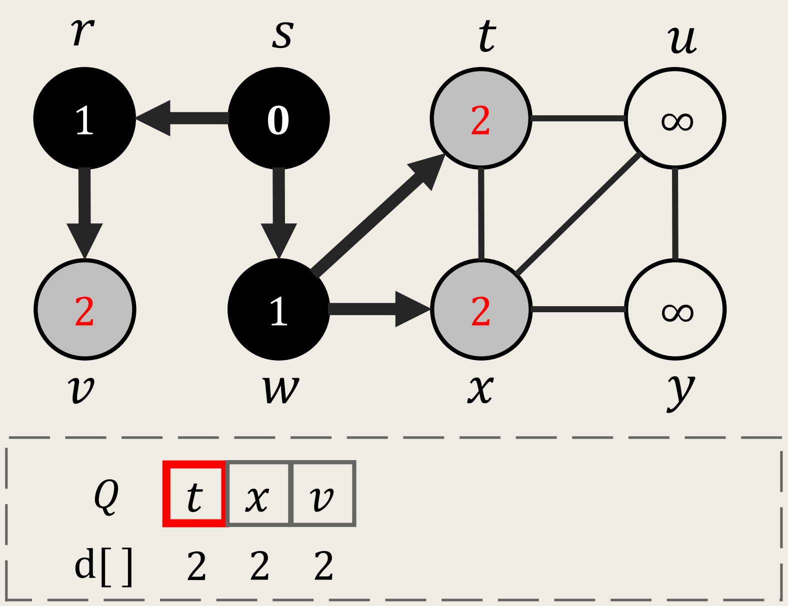
process (expand) r



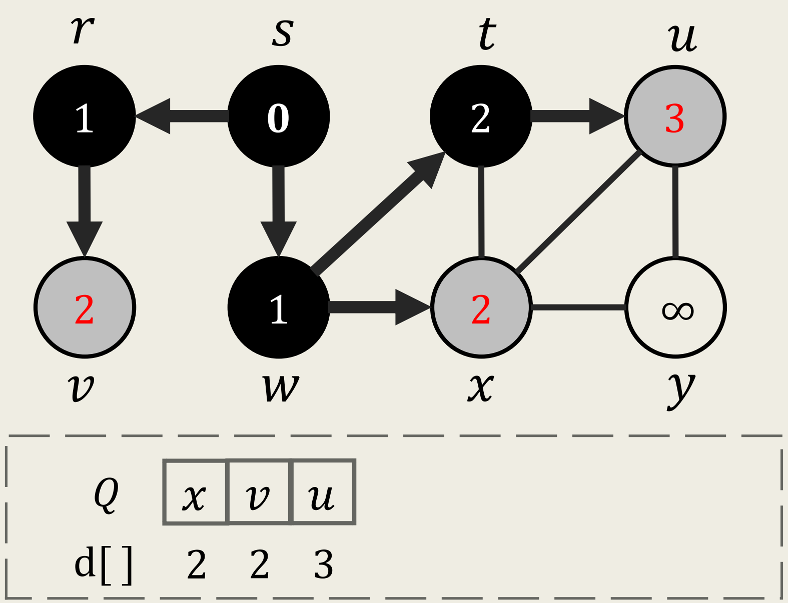
- Process the gray nodes until Q becomes empty.

A. While $Q \neq \emptyset$, do the following.
  - $u \leftarrow \text{DEQUEUE}(Q)$.
  - For each $v \in N[u]$, do
    - If $\text{color}[v] = \text{white}$, then
      - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
      - $\text{ENQUEUE}(Q, v)$.
  - Set $\text{color}[u] \leftarrow \text{black}$.

Discovery of distance-2 vertices is done.

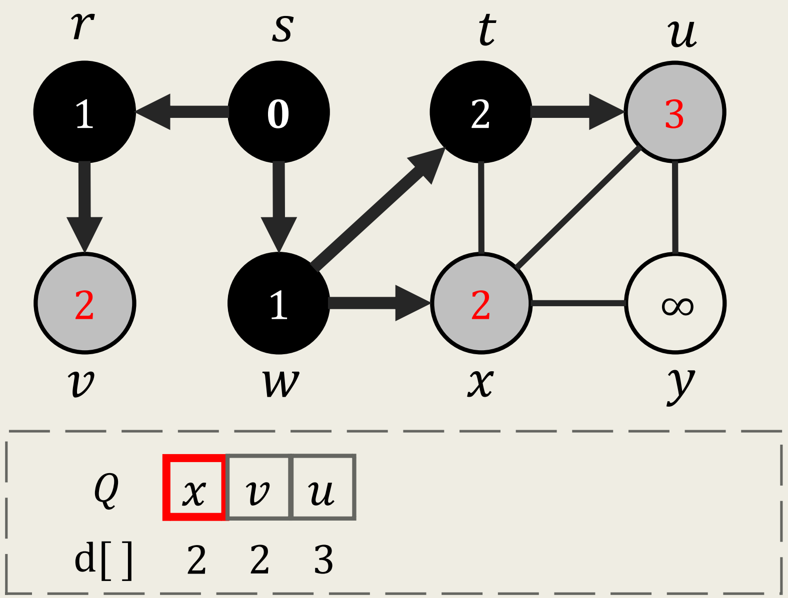


process (expand) t

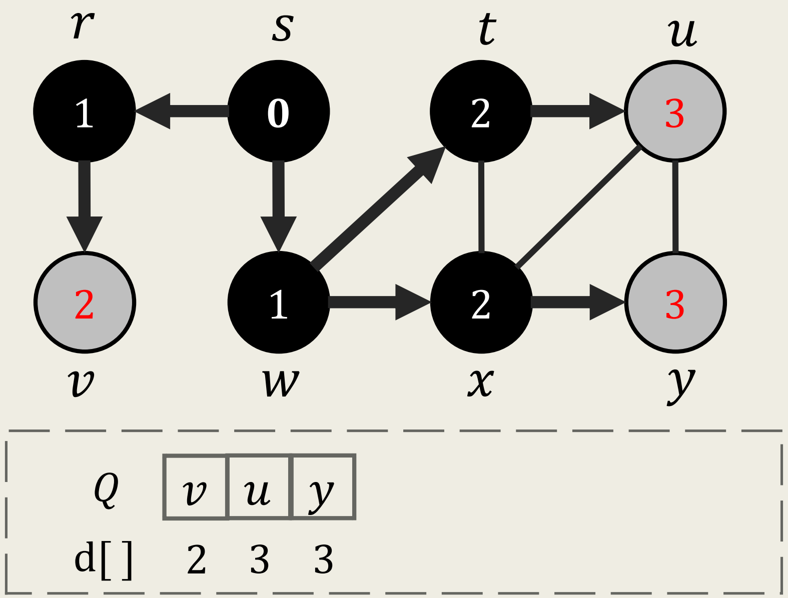


- Process the gray nodes until Q becomes empty.

A. While $Q \neq \emptyset$, do the following.
   - $u \leftarrow \text{DEQUEUE}(Q)$.
   - For each $v \in N[u]$, do
     - If $\text{color}[v] = \text{white}$, then
       - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
       - $\text{ENQUEUE}(Q, v)$.
   - Set $\text{color}[u] \leftarrow \text{black}$.

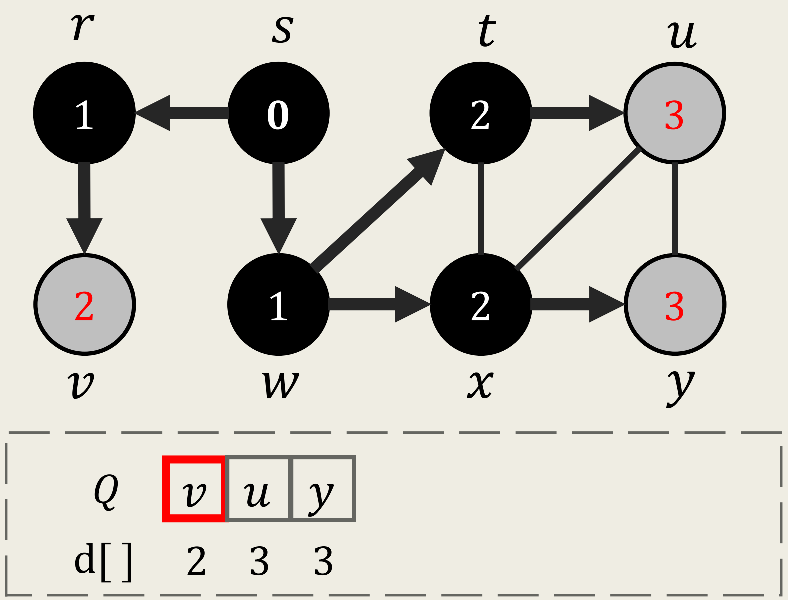


process (expand) x

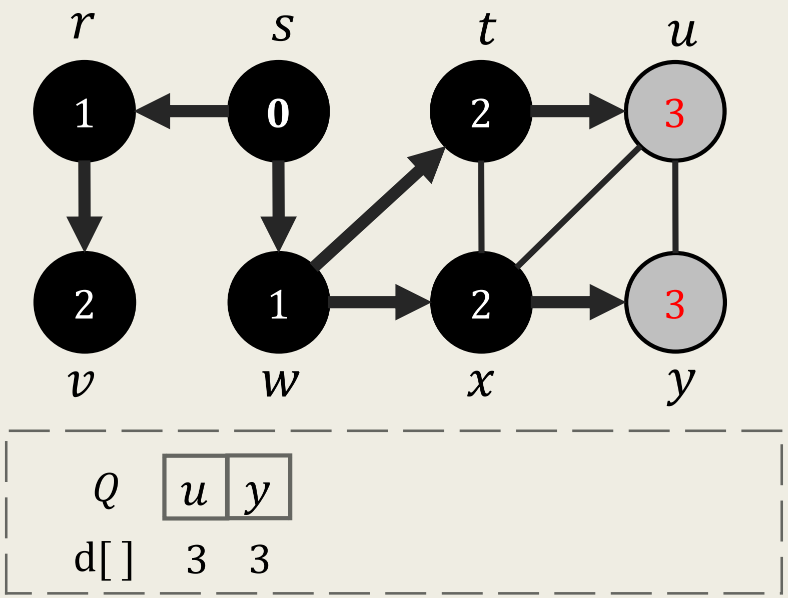


- Process the gray nodes until Q becomes empty.

A. While $Q \neq \emptyset$, do the following.
   - $u \leftarrow \text{DEQUEUE}(Q)$.
   - For each $v \in N[u]$, do
     - If $\text{color}[v] = \text{white}$, then
       - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
       - $\text{ENQUEUE}(Q, v)$.
   - Set $\text{color}[u] \leftarrow \text{black}$.



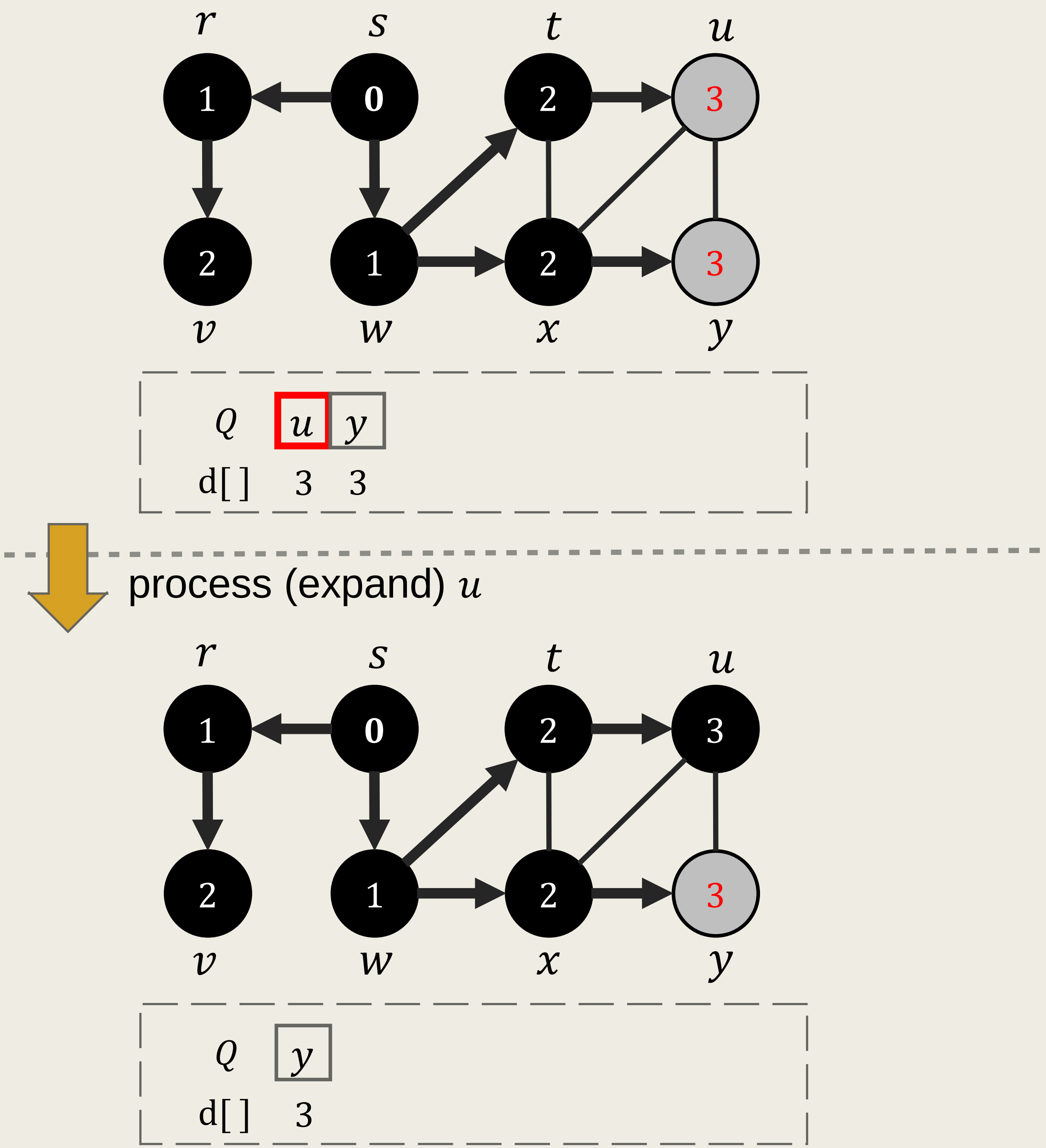
process (expand) v



- Process the gray nodes until Q becomes empty.

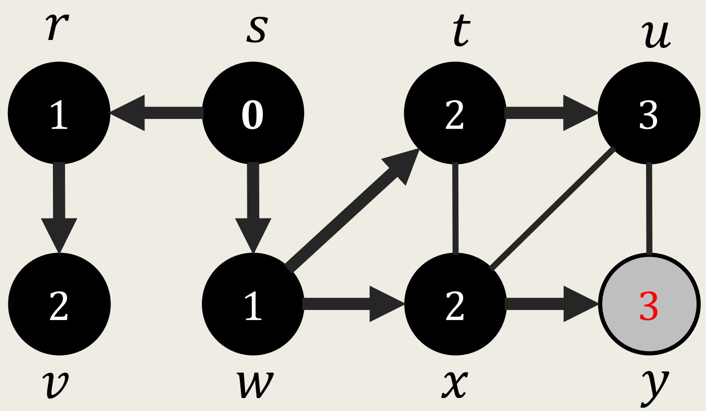
A. While $Q \neq \emptyset$, do the following.
   - $u \leftarrow \text{DEQUEUE}(Q)$.
   - For each $v \in N[u]$, do
     - If $\text{color}[v] = \text{white}$, then
       - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
       - $\text{ENQUEUE}(Q, v)$.
   - Set $\text{color}[u] \leftarrow \text{black}$.

Discovery of distance-3 vertices is done.



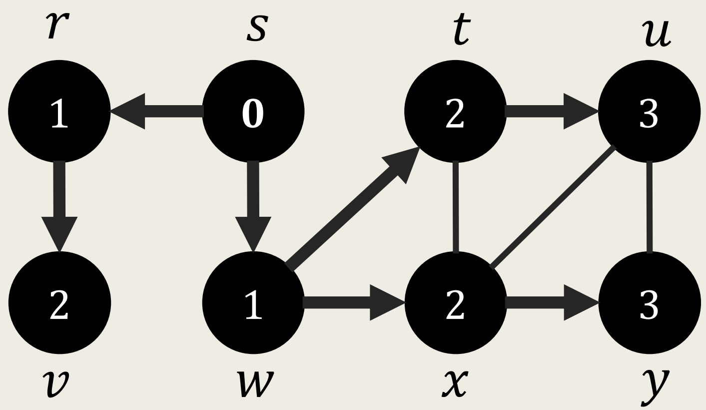
- Process the gray nodes until Q becomes empty.

A. While $Q \neq \emptyset$, do the following.
   - $u \leftarrow \mathrm{DEQUEUE}(Q)$.
   - For each $v \in N[u]$, do
     - If $\mathrm{color}[v] = \mathrm{white}$, then
       - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
       - $\mathrm{ENQUEUE}(Q, v)$.
   - Set $\mathrm{color}[u] \leftarrow \mathrm{black}$.



Q	y
d[ ]	3

process (expand) y



Q	
d[ ]	

- Process the gray nodes until Q becomes empty.

A. While $Q \neq \emptyset$, do the following.
   - $u \leftarrow \text{DEQUEUE}(Q)$.
   - For each $v \in N[u]$, do
     - If $\text{color}[v] = \text{white}$, then
       - Set $color[v] \leftarrow gray$, $d[v] \leftarrow d[u] + 1$, and $\pi[v] \leftarrow u$.
       - $\text{ENQUEUE}(Q, v)$.
   - Set $\text{color}[u] \leftarrow \text{black}$.

Q becomes empty, and the graph is traversed.

Analysis of the BFS Algorithm

Time Complexity of the BFS Algorithm

- The initialization step takes $\mathrm{O}(|V|)$ time.
- Consider the while loop.
  - The while loop repeats for at most $\mathrm{O}(|V|)$ times since every vertex enters the queue Q exactly once.
- Consider the inner for loop.
  - For any vertex $v \in V$, it takes $\mathrm{O}(\deg(v))$ time if adjacency list representation is used.
- The overall time complexity is $O(|V| + |E|)$ if adjacency list representation is used.

Correctness of the BFS Algorithm

Definition.

For any $\mathrm{u}, v \in V$, let $\delta(u, v)$ denote the distance between u and v in G.
$\delta(u, v) \equiv \infty$ if there is no path connecting u and v.

- We will prove the following theorem.

Theorem 1. (Correctness of Breadth-First Search)

When the algorithm terminates, we have $\mathrm{d}[v] = \delta(s, v)$ for all $v \in V$.

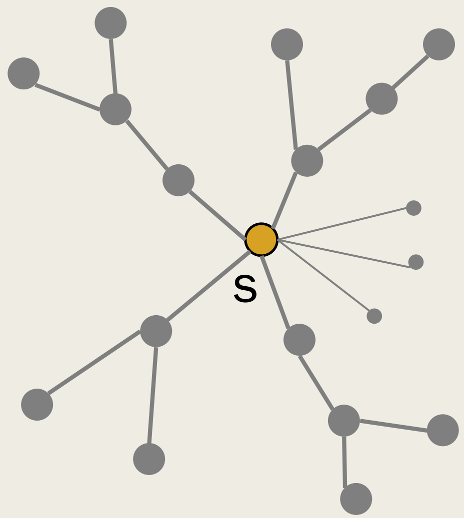
Moreover, for any $v \neq s$ that is reachable from s,
one of the shortest path from s to v consists of a shortest path from s to $\pi(v)$ followed by the edge $(\pi(v), v)$.

Breadth-First Tree (Shortest-Path Tree)

- Define the predecessor subgraph

$$G_\pi = (V_\pi, E_\pi), \text{ where } V_\pi = \{\, v \in V : \pi[v] \neq \text{NIL} \,\} \cup \{s\} \text{ and}$$

$$E_\pi = \{\, (\pi[v], v) : v \in V_\pi - \{s\} \,\} .$$



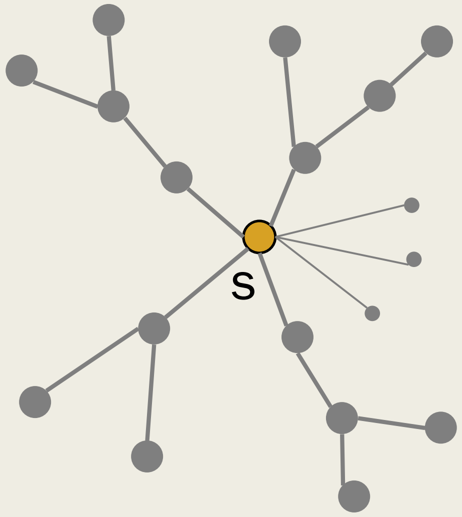
The predecessor graph G_π is connected and has exactly $|V_\pi| - 1$ edges.

It is a tree.

Breadth-First Tree (Shortest-Path Tree)

- Define the predecessor subgraph

$$G_\pi = (V_\pi, E_\pi), \text{ where } V_\pi = \{ v \in V : \pi[v] \neq \text{NIL} \} \cup \{s\} \text{ and}$$
$$E_\pi = \{ (\pi[v], v) : v \in V_\pi - \{s\} \} .$$



By Theorem 1,
for any $v \in V_\pi - \{s\}$, the $s{-}v$ path in G_π
must be a shortest $s{-}v$ path in the graph G.

We call G_π the Bread-First Tree, or,
the Shortest-Path Tree (SPT), induced by s.

- To prove Theorem 1, we need the following lemma, which follows from the design of the algorithm.

> **<u>Lemma 2.</u>**
>
> When the BFS algorithm terminates, for any edge $(u, v) \in E$, we have
>
> $$\mathrm{d}[u] < \infty \implies \mathrm{d}[v] \leq \mathrm{d}[u] + 1.$$

- If $d[v] > d[u]$, then consider the moment when u is processed in the while loop.
  - If v is already discovered, then $d[v]$ is either $d[u]$ or $d[u] + 1$.
  - Otherwise, v will be discovered by u and $d[v] = d[u] + 1$.

- Now let's prove Theorem 1. ☺

Theorem 1. (Correctness of Breadth-First Search)

When the algorithm terminates, we have $\mathrm{d}[v] = \delta(s, v)$ for all $v \in V$.

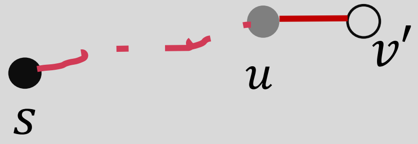
Moreover, for any $v \neq s$ that is reachable from s,
one of the shortest path from s to v consists of a shortest path from s to $\pi(v)$ followed by the edge $(\pi(v), v)$.

Proof.

- For any $v \in V$, the distance of the s-v path in the SPT is $d[v]$.
  - Hence, $d[v] \geq \delta(s, v)$.
  - It suffices to prove that $d[v] \leq \delta(s, v)$.

Proof. (continue)

Assume for contradiction that $\mathrm{d}[v] > \delta(s, v)$ for some $v \in V$.

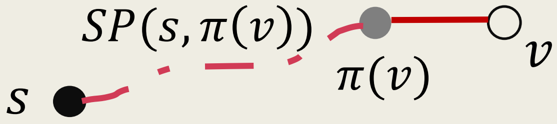


Let v' be such a vertex with the minimum $\delta(s, v')$.
It follows that $v' \neq s$ and $\delta(s, v') < \infty$.

Consider any shortest s-v' path, and let u be the vertex preceding v' on the path.
Hence $\delta(s, v') = \delta(s, u) + 1$, and by our assumption we have $\mathrm{d}[u] = \delta(s, u)$.

It follows that $\mathrm{d}[v'] > \delta(s, v') = \delta(s, u) + 1 = d[u] + 1$, a contradiction to Lemma 2.

The second part of the theorem follows directly from $\mathrm{d}[v] = \delta(s, v)$ for all $v \in V$.



$$\delta(s, v) = d[v] = d[\pi[v]] + 1 = \delta(s, \pi[v]) + 1.$$

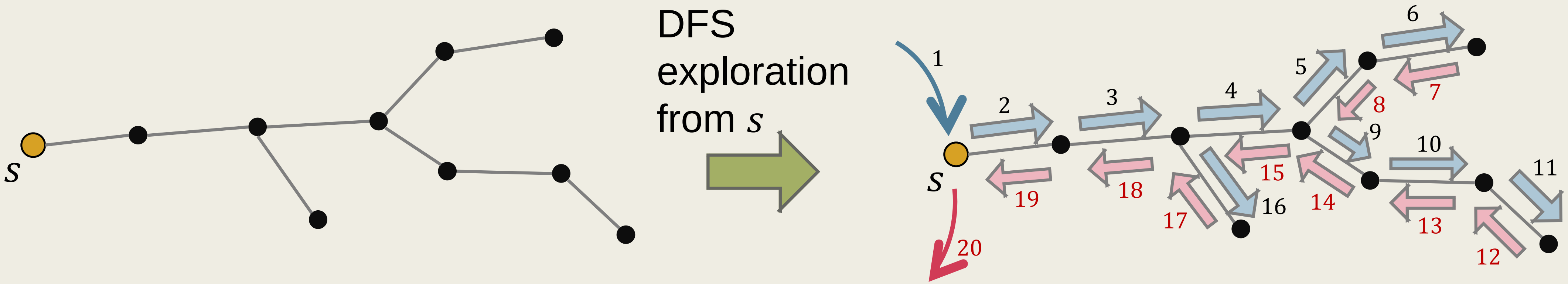
Depth-First Search (DFS)

Prefer depth over breadth.

Traverse / Search deeper whenever possible.

Depth-First Search (DFS)

- The DFS algorithm **search deeper in the graph whenever possible** until all the vertices are discovered.
  - At any vertex, it picks an *unexplored neighboring vertex* and search recursively until all neighboring vertices are explored.

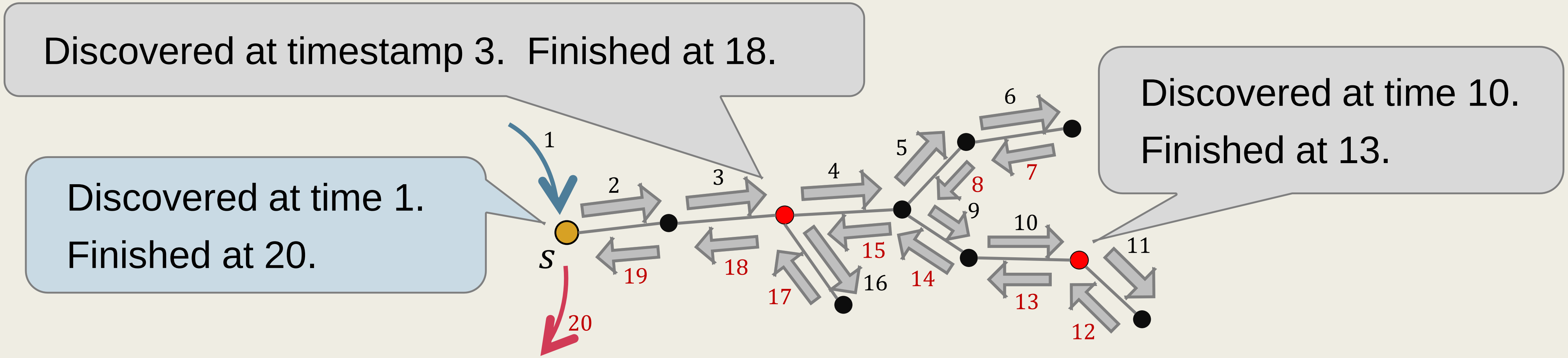


Formal Description of the DFS Algorithm

- The algorithm maintains the following information during its execution.
  - $\forall v \in V$, the color (status) of v, denoted $\mathrm{color}[v]$.
    - White: not yet discovered
    - Gray: discovered but not yet finished
    - Black: discovered & finished
  - $\forall v \in V$, the predecessor of $\pi[v]$ of v during the search.
    - NIL if v has no predecessor (yet).

Formal Description of the DFS Algorithm

- During the process, the DFS algorithm maintains the following data.
  - $\forall v \in V$,
    - $d[v]$: the timestamp when v is first discovered.
    - $f[v]$: the timestamp when the search from v is done.



Formal Description of the DFS Algorithm

DFS(G) - DFS on $G = (V, E)$.

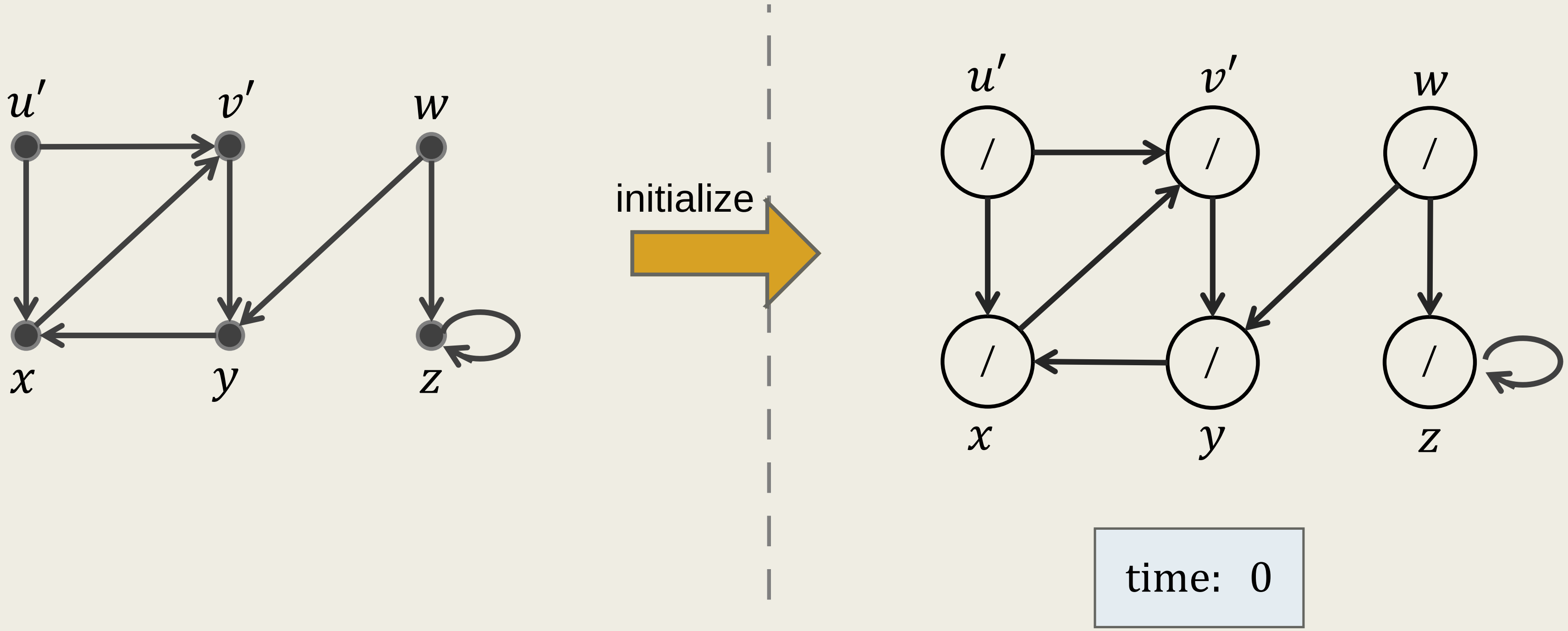
A. For each $v \in V$,
   set $\mathrm{color}[v] \leftarrow \mathrm{white}$ and
   $\pi[v] \leftarrow \mathrm{NIL}$.
B. Set $\mathrm{time} \leftarrow 0$.
C. For each $v \in V$,
   - If $\mathrm{color}[v] = \mathrm{white}$,
     then call DFS-Visit(v).

DFS-Visit(u) - Search recursively at u.

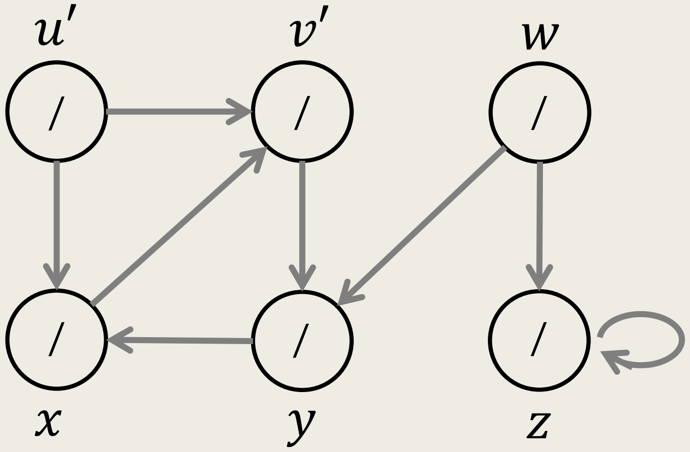
A. Set $\mathrm{color}[u] \leftarrow \mathrm{gray}$ and
   $d[u] \leftarrow (\ \mathrm{time} \leftarrow \mathrm{time} + 1\ )$.
B. For each $v \in N(u)$, do
   - If $\mathrm{color}[v] = \mathrm{white}$, then
     set $\pi[v] \leftarrow u$ and DFS-Visit(v).
C. Set $\mathrm{color}[u] \leftarrow \mathrm{black}$ and
   $f[u] \leftarrow (\ \mathrm{time} \leftarrow \mathrm{time} + 1\ )$.

An Example

- Consider the following graph and the execution of the DFS algorithm.



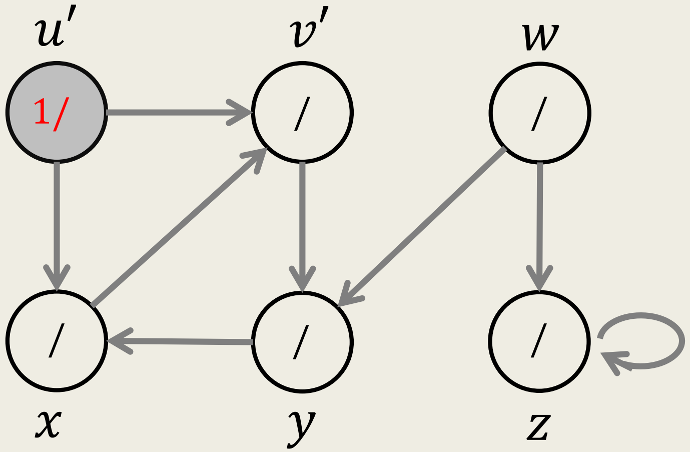
Current calls : DFS(G)



time: 0

examines u' and calls DFS-VISIT(u')

Current calls : DFS-VISIT(u')

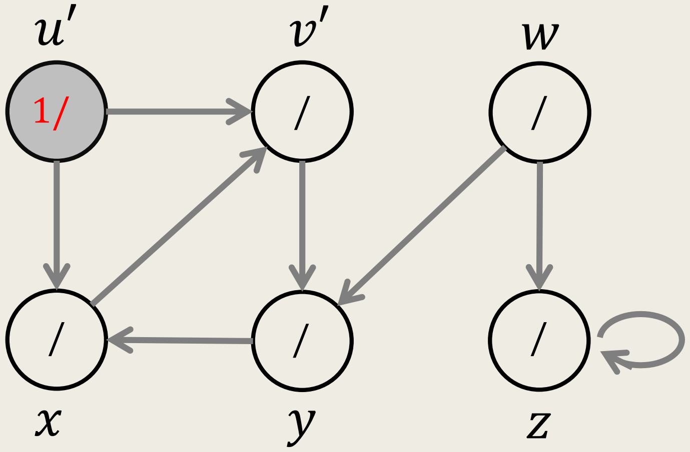


time: 1

```
DFS ( G )
    ...
    ...
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

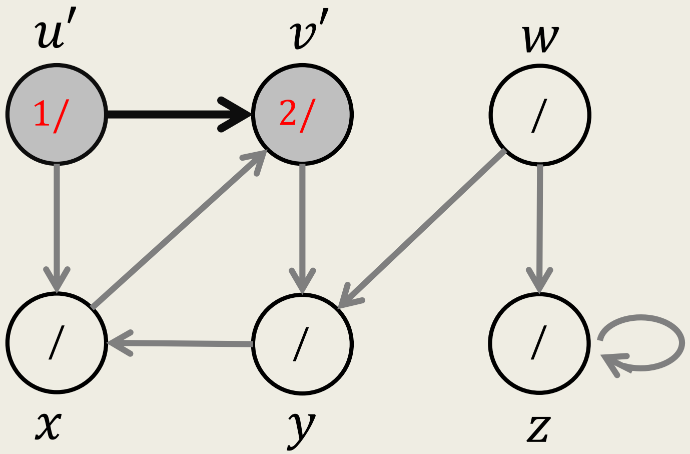
Current calls : DFS-VISIT(u')



time: 1

examines v' and calls DFS-VISIT(v')

Current calls : DFS-VISIT(v')



time: 2

DFS (G)

…

…

for each $u \in V$, **do**

if color$[u]$ is white, **then**

DFS-VISIT (u).

DFS-VISIT (u)

color$[u] \leftarrow$ gray.

$d[u] \leftarrow$ (time $\leftarrow$ time + 1).

for each $v \in$ Adj$[u]$, **do**

if color$[v]$ is white, **then**

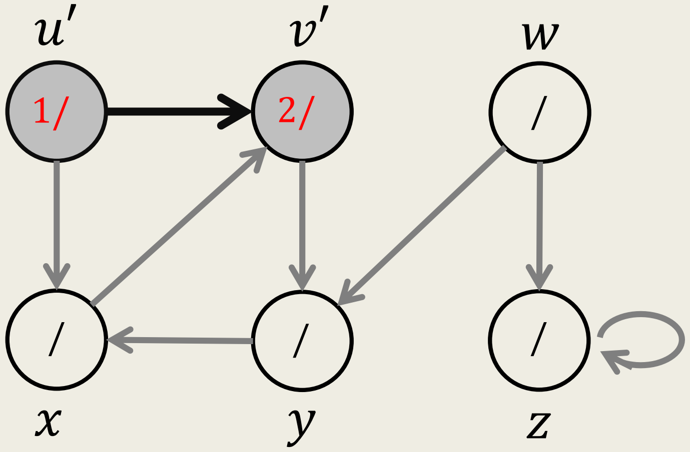
$\pi[v] \leftarrow u$.

DFS-VISIT (v).

color$[u] \leftarrow$ black.

$f[u] \leftarrow$ (time $\leftarrow$ time + 1).

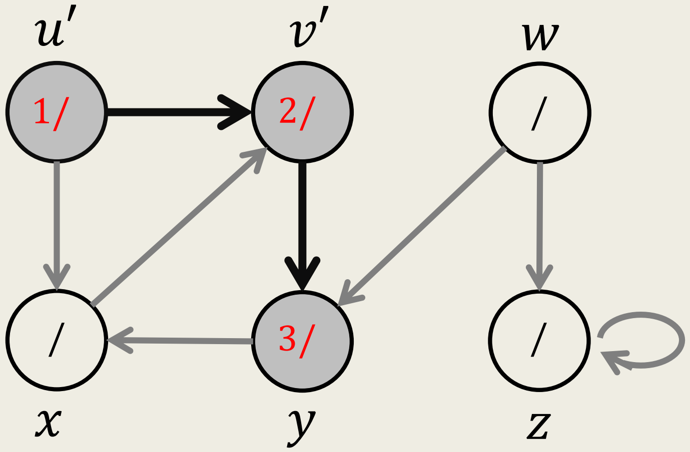
Current calls : DFS-VISIT(v')



time: 2

examines y and calls DFS-VISIT(y)

Current calls : DFS-VISIT(y)



time: 3

DFS (G)

…

…

for each $u \in V$, **do**

  if color[u] is white, **then**

    DFS-VISIT (u).

DFS-VISIT (u)

color[u] ← gray.

$d[u]$ ← (time ← time + 1).

for each $v \in \text{Adj}[u]$, **do**

  if color[v] is white, **then**

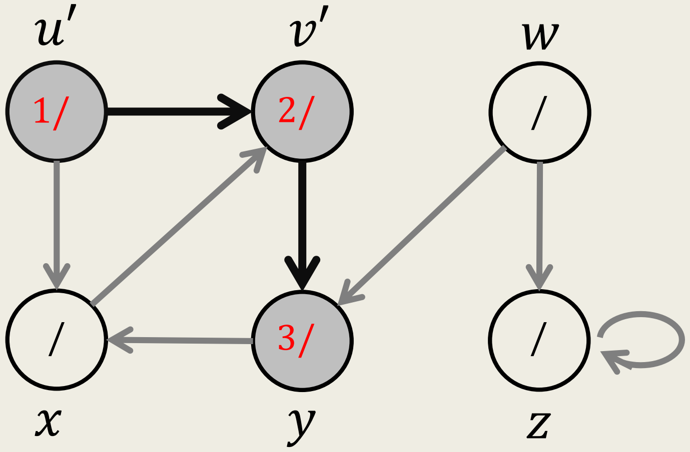
    $\pi[v] \leftarrow u$.

    DFS-VISIT (v).

color[u] ← black.

$f[u]$ ← (time ← time + 1).

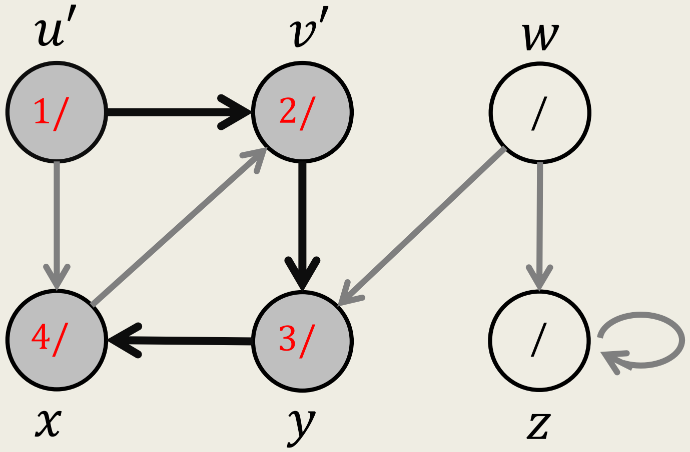
Current calls : DFS-VISIT(y)



time: 3

examines x and calls DFS-VISIT(x)

Current calls : DFS-VISIT(x)

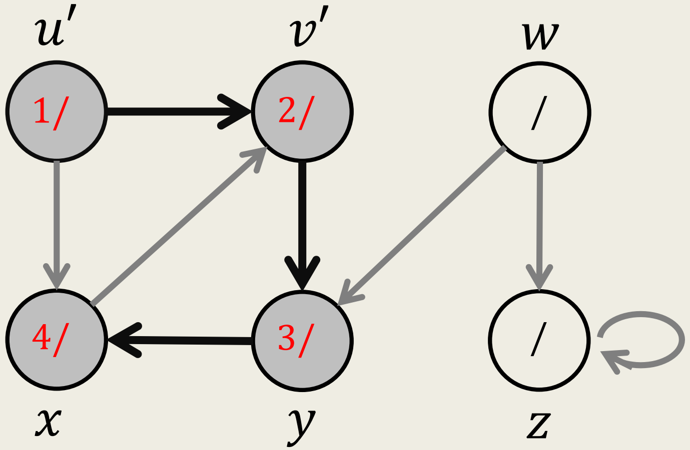


time: 4

```
DFS ( G )
    …
    …
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

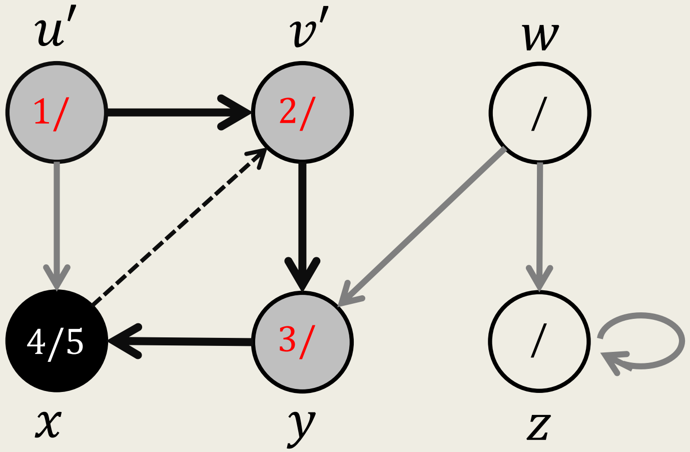
Current calls : DFS-VISIT(x)



time: 4

examines v', finishes x, and returns

Current calls : DFS-VISIT(y)

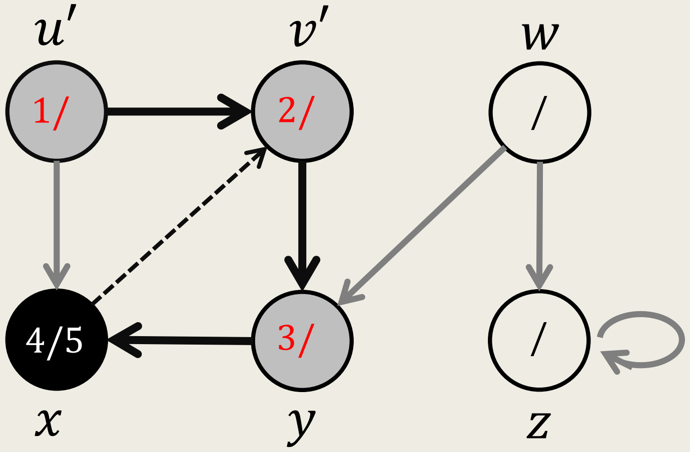


time: 5

```
DFS ( G )
    …
    …
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

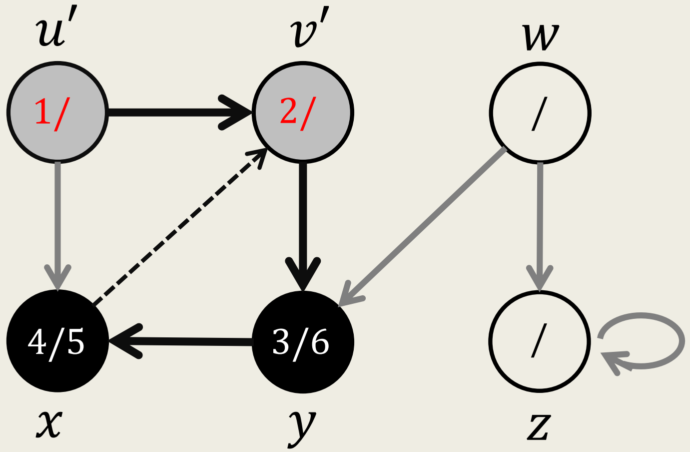
Current calls : DFS-VISIT(y)



time: 5

finishes y and returns

Current calls : DFS-VISIT(v')



time: 6

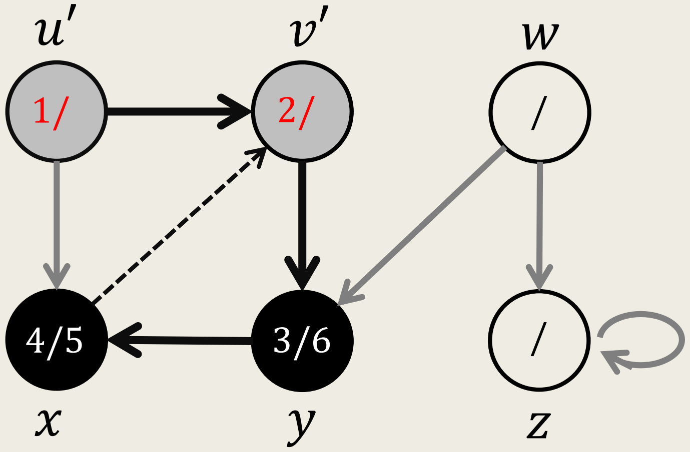
DFS (G)

- …
- …
- **for each** $u \in V$, **do**
  - **if** color[u] is white, **then**
    - **DFS-VISIT** (u).

DFS-VISIT (u)

- color[u] ← gray.
- $d[u]$ ← (time ← time + 1).
- **for each** $v \in \mathrm{Adj}[u]$, **do**
  - **if** color[v] is white, **then**
    - $\pi[v] \leftarrow u$.
    - **DFS-VISIT** (v).
- color[u] ← black.
- $f[u]$ ← (time ← time + 1).

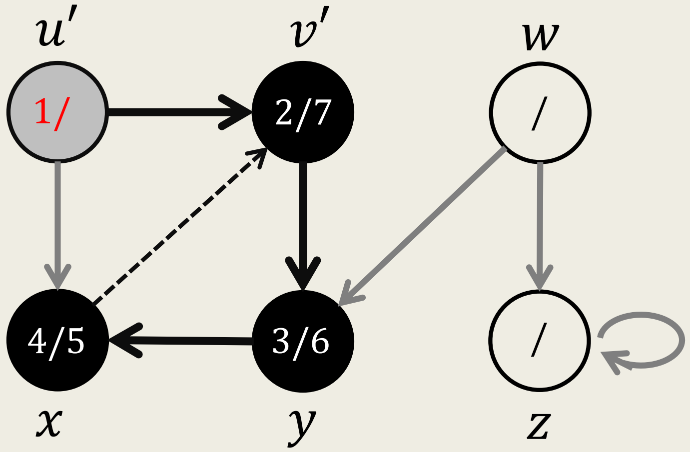
Current calls : DFS-VISIT(v')



time: 6

finishes v' and returns

Current calls : DFS-VISIT(u')

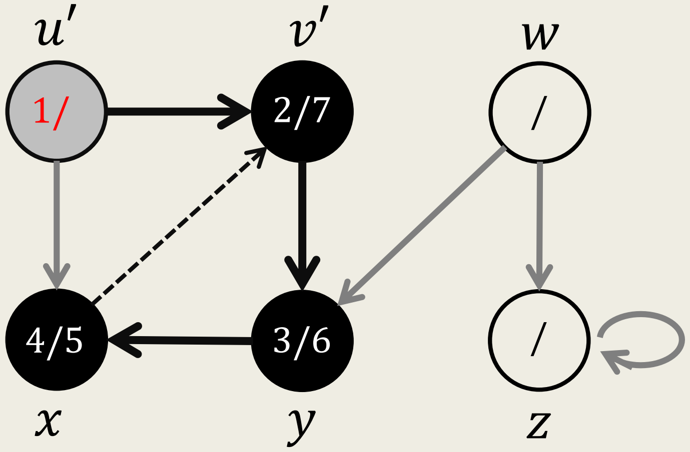


time: 7

```
DFS ( G )
    ...
    ...
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

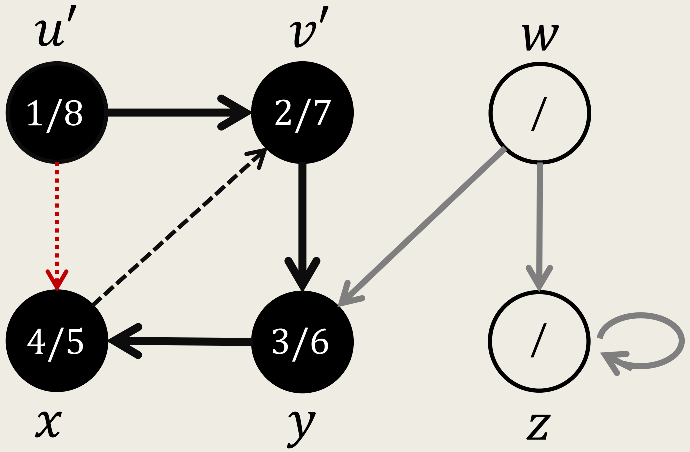
Current calls : DFS-VISIT(u')



time: 7

examines x, finishes u' and returns

Current calls : DFS(G)



time: 8

DFS (G)

…

…

for each $u \in V$, **do**

  if color[u] is white, **then**

   DFS-VISIT (u).

DFS-VISIT (u)

color[u] ← gray.

$d[u]$ ← (time ← time + 1).

for each $v \in$ Adj[u], **do**

  if color[v] is white, **then**

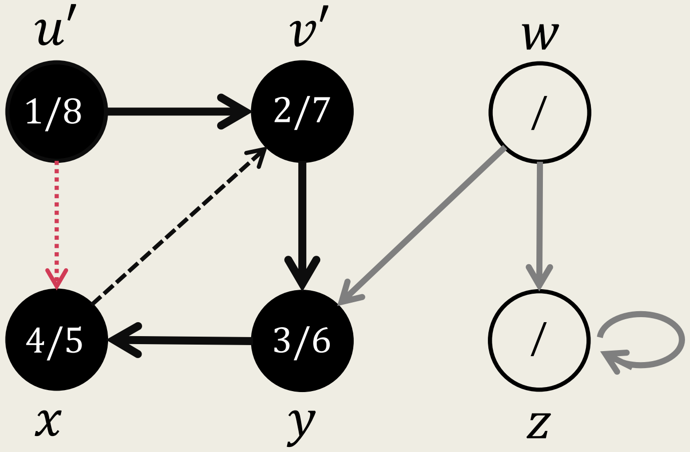
   $\pi[v] \leftarrow u$.

   DFS-VISIT (v).

color[u] ← black.

$f[u]$ ← (time ← time + 1).

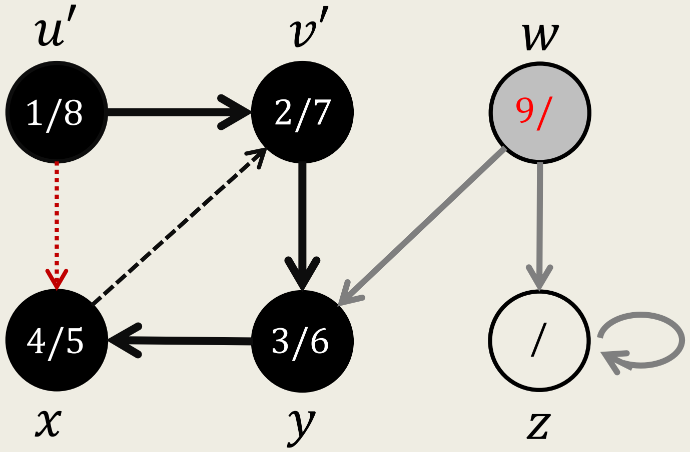
Current calls : DFS(G)



time: 8

examines v' and w, and calls DFS-VISIT(w)

Current calls : DFS-VISIT(w)

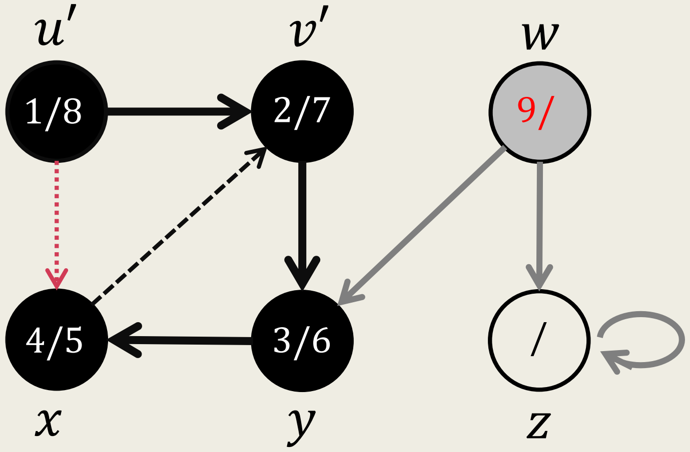


time: 9

```
DFS ( G )
    …
    …
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

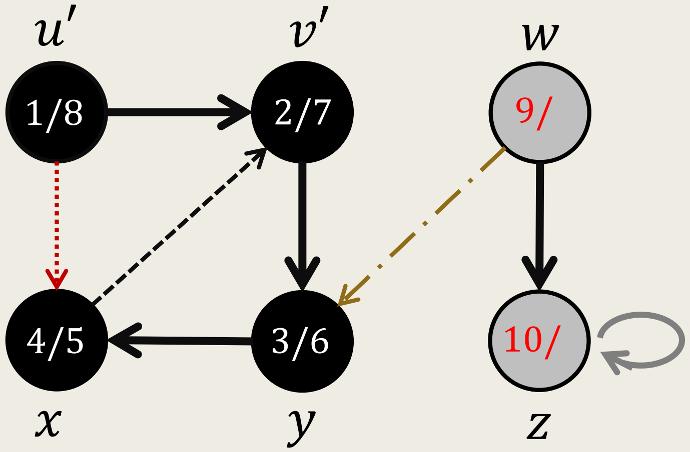
Current calls : DFS-VISIT(w)



time: 9

examines y and z, and calls DFS-VISIT(z)

Current calls : DFS-VISIT(z)

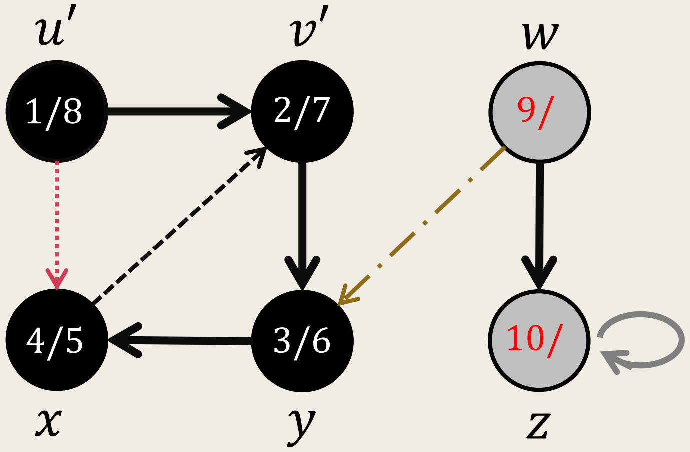


time: 10

```
DFS ( G )
    ...
    ...
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

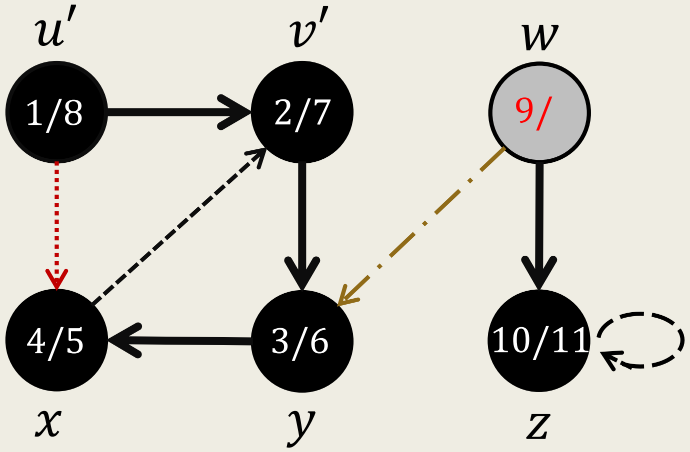
Current calls : DFS-VISIT(z)



time: 10

examines z, finishes z, and returns

Current calls : DFS-VISIT(w)



time: 11

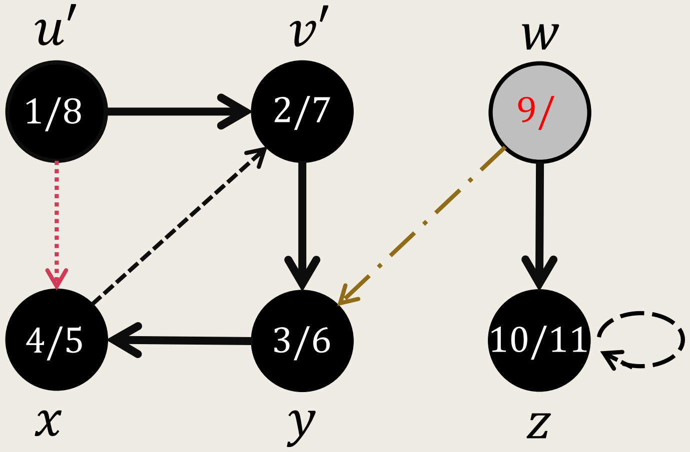
DFS (G)

- …
- …
- **for each** $u \in V$, **do**
  - **if** color[u] is white, **then**
    - **DFS-VISIT** (u).

DFS-VISIT (u)

- color[u] ← gray.
- $d[u]$ ← (time ← time + 1).
- **for each** $v \in$ Adj[u], **do**
  - **if** color[v] is white, **then**
    - $\pi[v] \leftarrow u$.
    - **DFS-VISIT** (v).
- color[u] ← black.
- $f[u]$ ← (time ← time + 1).

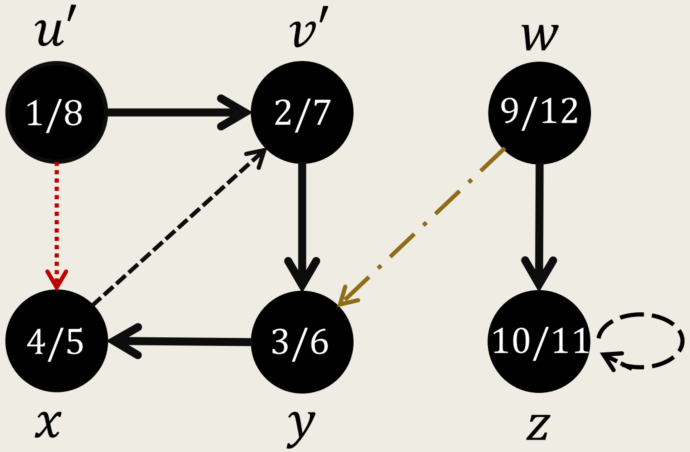
Current calls : DFS-VISIT(w)



time: 11

finishes w, and returns

Current calls : DFS(G)

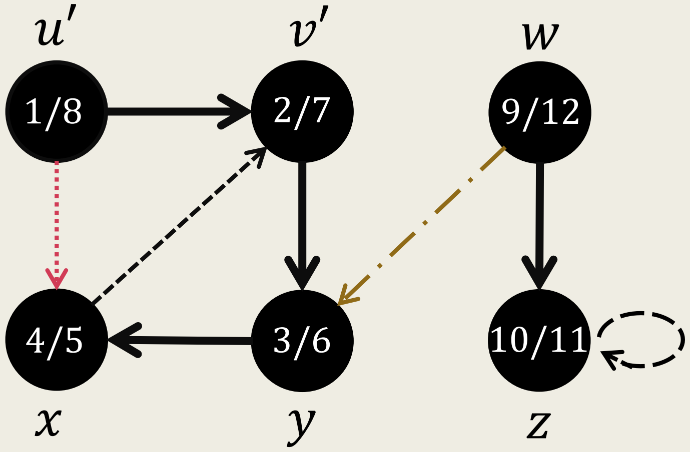


time: 12

```
DFS ( G )
    …
    …
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

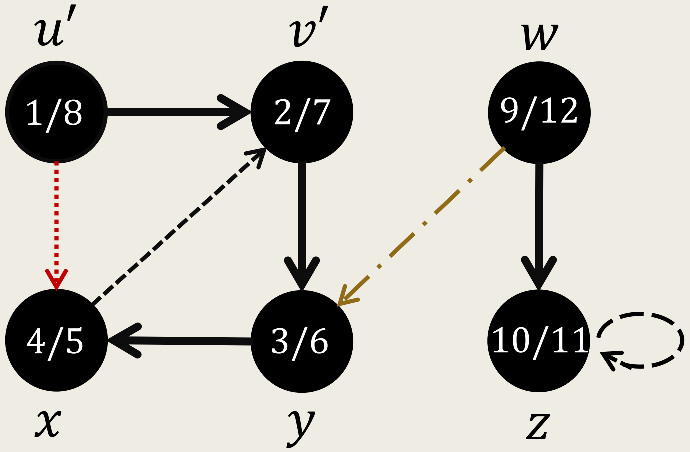
Current calls : DFS(G)



time: 12

examines x, y, z and returns

Current calls :



time: 12

```
DFS ( G )
    …
    …
    for each u ∈ V, do
        if color[u] is white, then
            DFS-VISIT (u).
```

```
DFS-VISIT ( u )
    color[u] ← gray.
    d[u] ← ( time ← time + 1 ).
    for each v ∈ Adj[u], do
        if color[v] is white, then
            π[v] ← u.
            DFS-VISIT (v).
    color[u] ← black.
    f[u] ← ( time ← time + 1 ).
```

Properties and Analysis of the DFS Algorithm

Time Complexity

- By the algorithm design,
  once a vertex is colored, it never becomes white.
  - So, the DFS algorithm…
    - Visits each vertex exactly once upon DFS-VISIT calls.
    - Examines each edge at most twice in the for loop.
- The DFS algorithm runs in $O(|V| + |E|)$ time.

Properties of the DFS Algorithm

- For any vertex $v \in V$,
  consider the time interval $I_v \coloneqq \left[d[v], f[v]\right]$.
- The DFS algorithm ensures the *laminar (nesting or disjoint)* property of the intervals of the nodes.

Theorem 3. (Parenthesis Theorem)

For any $u, v \in V$, exactly one of the followings holds.

1. $I_u \cap I_v = \emptyset$.
2. $I_u \subset I_v$ or $I_v \subset I_u$.

Consider the DFS forest.

None is a proper descendant of the other.

One is a proper descendant of the other.

- Consider the DFS forest G_π.

  The followings are obtained from Theorem 3.

Corollary 3.

For any $u, v \in V$, vertex v is a proper descendant of u if and only if $d[u] < d[v] < f[v] < f[u]$.

Theorem 3. (White-Path Theorem)

Consider the DFS forest G_π.

For any $u, v \in V$, vertex v is a descendant of u if and only if

at time $d[u]$, there is a u-v path in G that contains only white vertices.

Classification of the Edges

- One important characteristics of the DFS algorithm is that it classifies the edges of the input graphs into *four categories*.

  – **Tree edge**

    The edges that lead to undiscovered vertices during the search.

    - Formally, for each $v \in V$ with $\pi[v] \neq \mathrm{NIL}$, the edge $(\pi[v], v)$ is called a tree edge.
    - These are exactly the edges in the predecessor graph G_π, or, the DFS-tree.

Classification of the Edges

- One important characteristics of the DFS algorithm is that it classifies the edges of the input graphs into *four categories*.
  - **Back edge**

    The edges that lead to non-parent gray vertices during the search.

    - These are the edges that connect the current vertex **back to one of its predecessor vertices** during the search.
    - Formally, on the call DFS-Visit on $v \in V$, for any $u \in N(v)$ such that $\mathrm{color}[u] = \mathrm{gray}$ and $u \neq \pi[v]$, the edge (v, u) is called a back edge.

Classification of the Edges

- One important characteristics of the DFS algorithm is that it classifies the edges of the input graphs into *four categories*.
  - **Tree edge**
  - **Back edge**
- Note that, in ***undirected graphs***, edges are either ***tree edges*** or ***back edges***.

Classification of the Edges

- One important characteristics of the DFS algorithm is that it classifies the edges of the input graphs into *four categories*.
  - **Forward edge**

    The non-tree edges that lead to a proper descendant in the DFS tree.

    - Formally, on the call DFS-Visit on $v \in V$,
      for any $u \in N(v)$ such that $\mathrm{color}[u] = \mathrm{black}$ and $d[v] < d[u]$,
      then the edge (v, u) is a back edge.

Classification of the Edges

- One important characteristics of the DFS algorithm is that it classifies the edges of the input graphs into *four categories*.
  - **Cross edge**

    All other edges that go between vertices in the DFS forest, as long as one is not an ancestor of the other.

    - Formally, for any edge (u, v), if $f[v] < d[u]$, then the (u, v) is referred to as a cross edge.

Classification of the Edges

- One important characteristics of the DFS algorithm is that it classifies the edges of the input graphs into ***four categories***.
  - **Forward edge**
  - **Cross edge**
- Note that, forward edges and cross edges only occur in directed graphs.

Classification of the Edges

- One important characteristics of the DFS algorithm is that it classifies the edges of the input graphs into *four categories*.
  - Tree edge, Back edge, Forward edge, and Cross edge

- By the parenthesis theorem, any edge in the graph belongs to one of the above four categories.