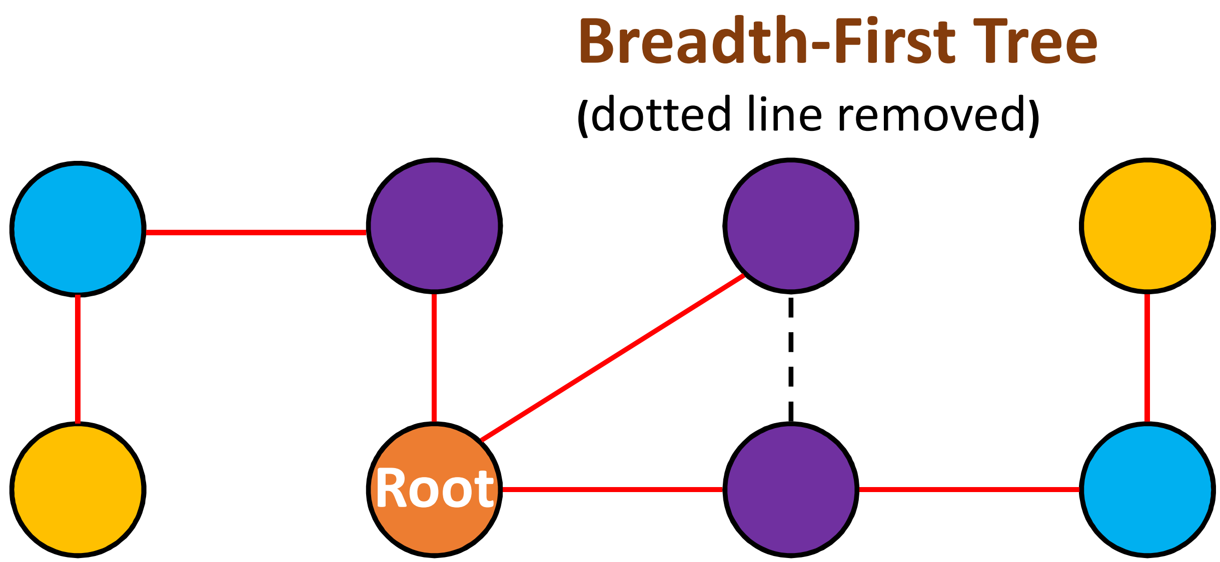
CSE 105: Data Structures and Algorithms-I (Part 2)

Instructor

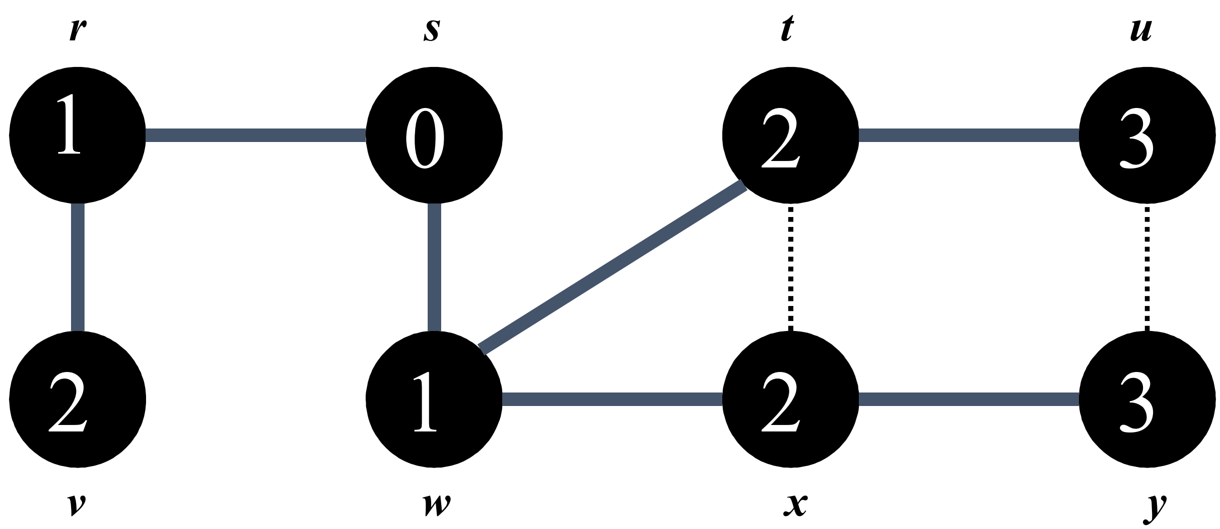
Dr Md Monirul Islam

Graph Searching

Breadth-First Tree



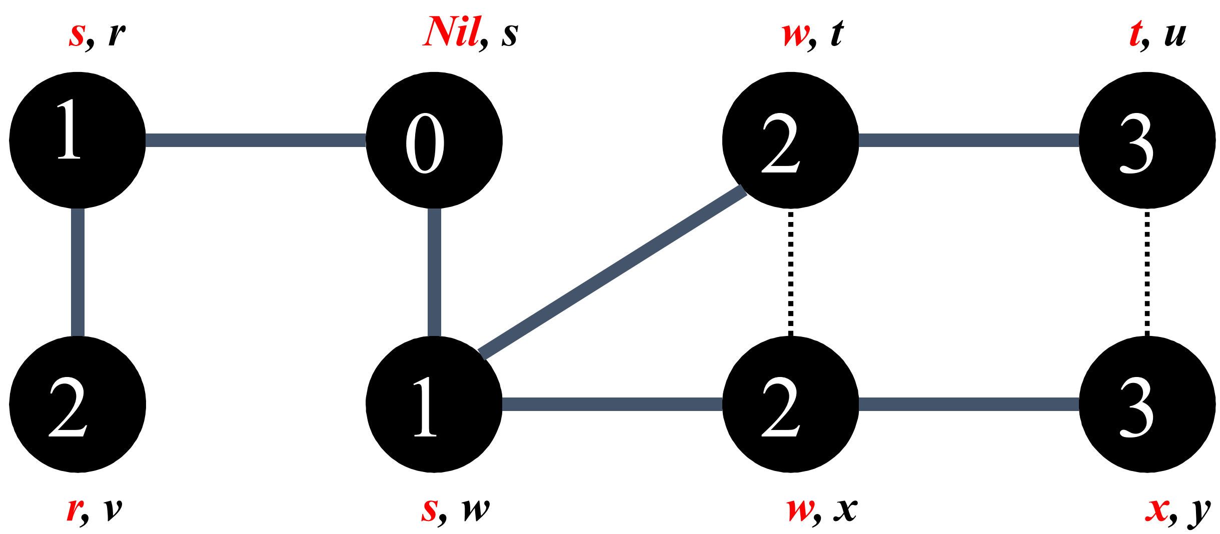
Breadth-First Tree



Breadth-First Tree

(dotted lines removed)

Breadth-First Tree

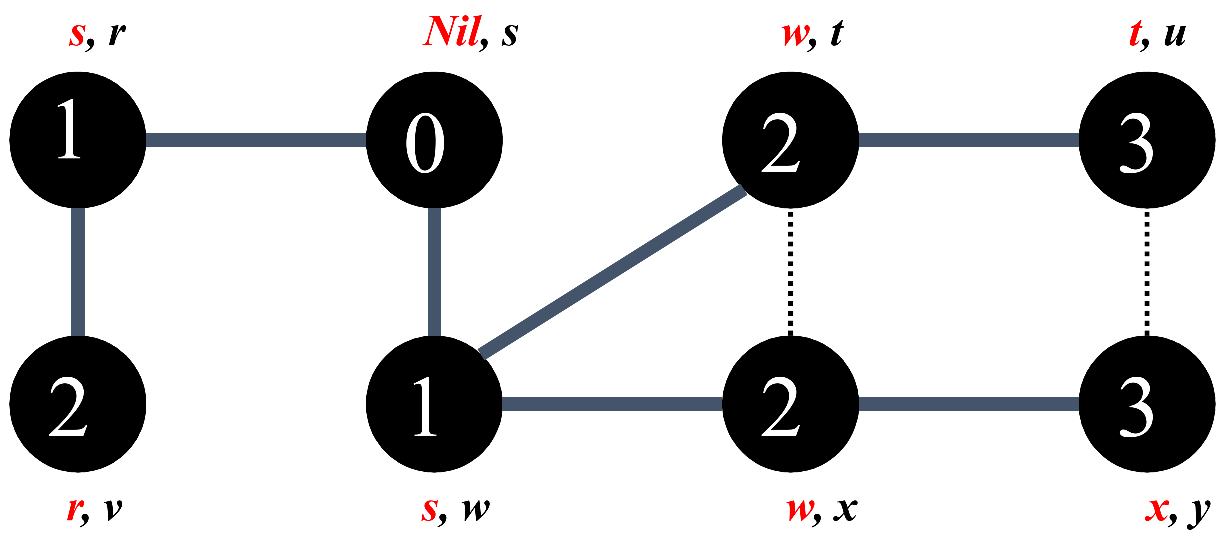


Parent is shown as **in RED**

Breadth-First Tree

(dotted lines removed)

Breadth-First Tree



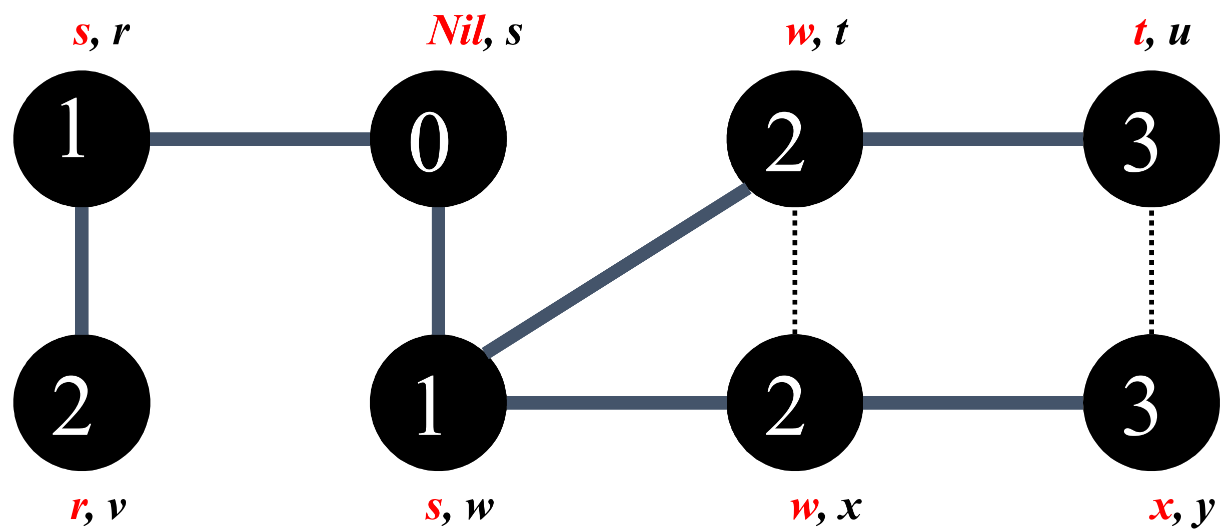
Parent is shown as **in RED**

Edges in the Tree: $E_\pi = \{(s, w), (s, r), (w, t), (w, x), (r, v), (t, u), (x, y)\}$

Not included in E_π : $(t, x), (u, y)$

Predecessor Subgraph, G_π

- A graph where all predecessors are defined

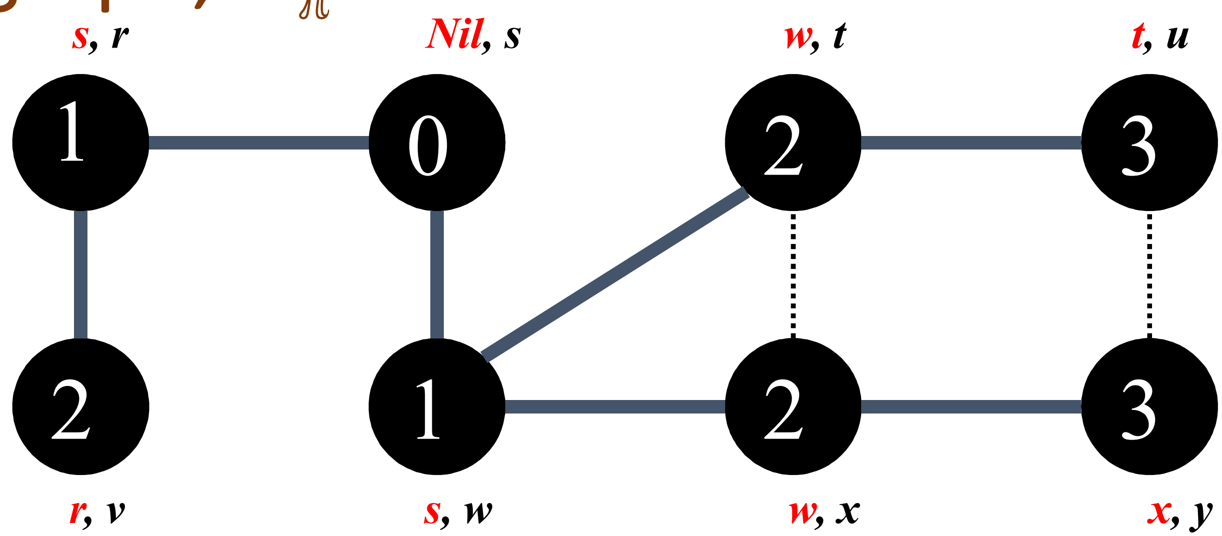


Predecessor Subgraph, G_π

- More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ and

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



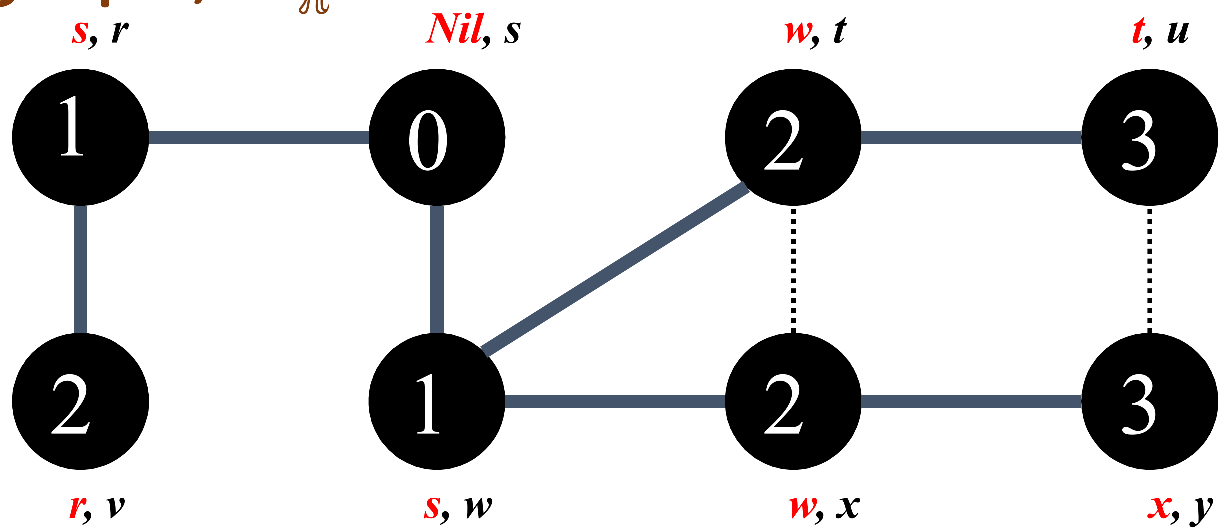
Predecessor Subgraph, G_π

- More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

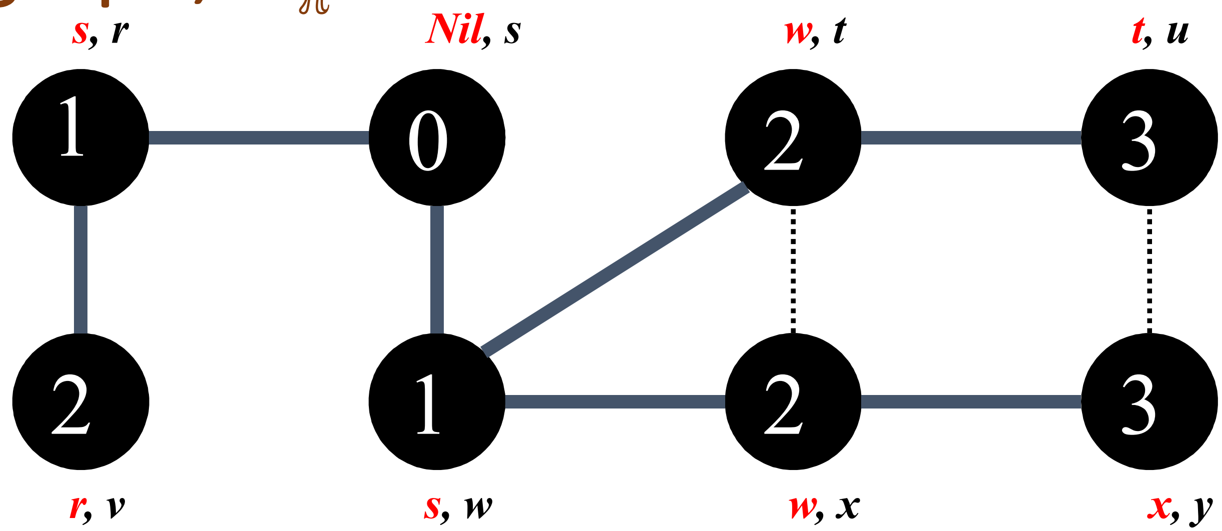
$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ and

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

G_π is a tree? How?



Predecessor Subgraph, G_π



- More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ and

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

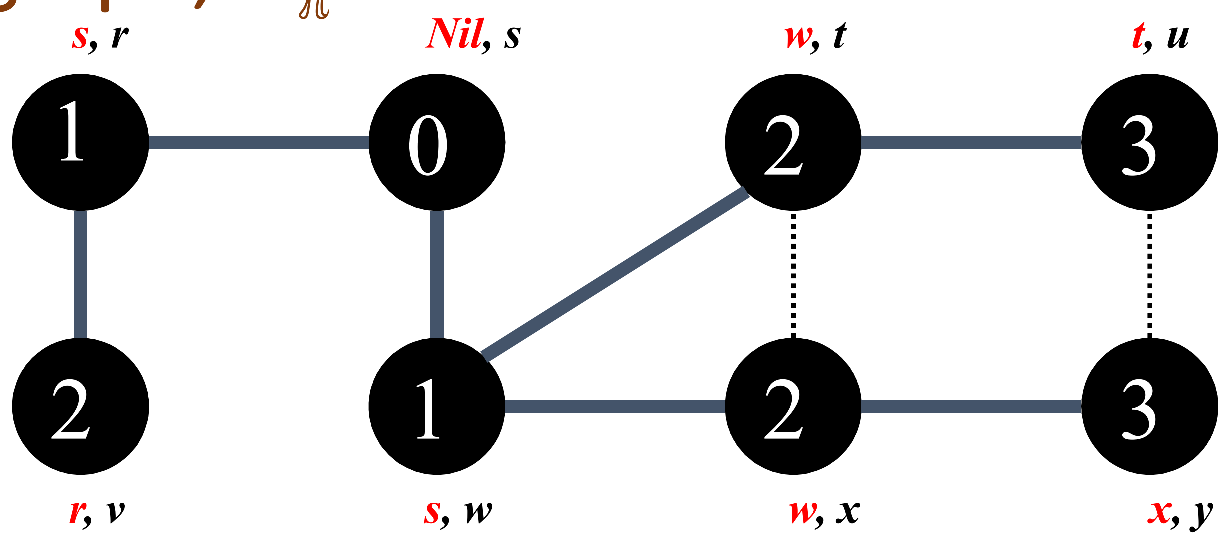
G_π is a tree? How?

V_π consists of (1) vertex s plus

(2) those unique vertices that have a parent

Predecessor Subgraph, G_π

- More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where



$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ and

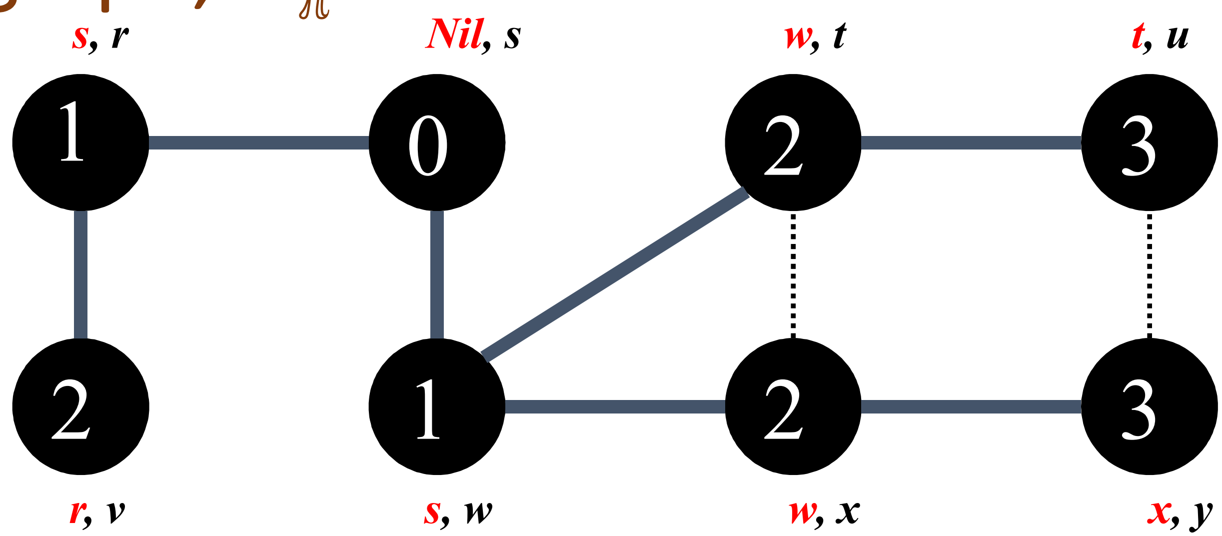
$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$

G_π is a tree? How?

E_π consists of edges from vertices of $\{V_\pi - \{s\}\}$ to their parents

ONLY s has NO connection to its parents

Predecessor Subgraph, G_π



- More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ and

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

G_π is a tree? How?

$$|E_\pi| = |V_\pi| - 1$$

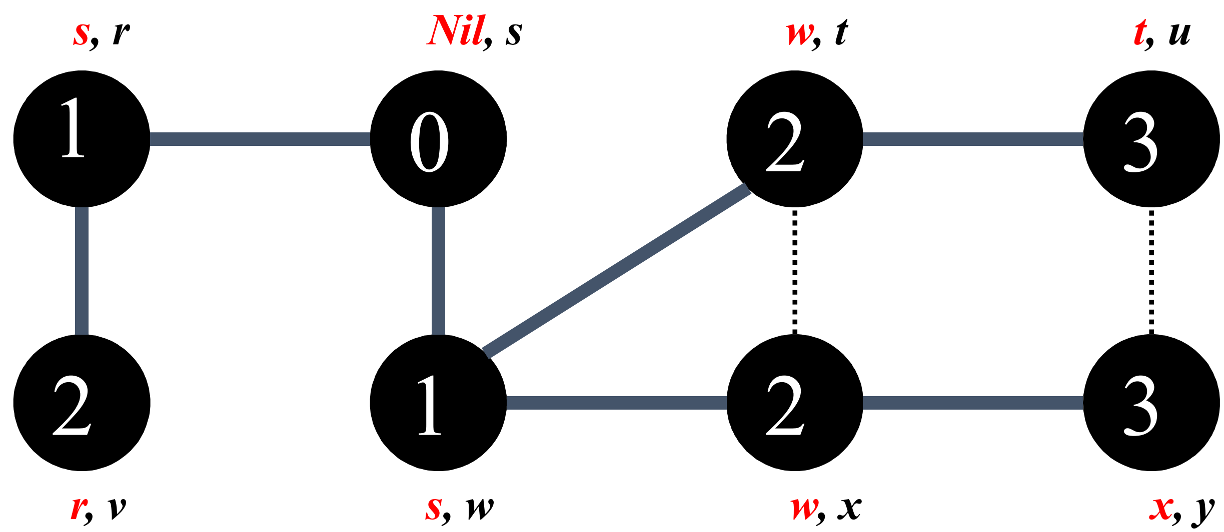
That means, G_π is a tree.

Predecessor Subgraph and Breadth-First Tree

- More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$ and

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



G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G.

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = Ø
9  ENQUEUE(Q, s)
10 while Q ≠ Ø
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

Lemma 22.6

When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Line 16 sets $v.\pi = u$ iff (u, v) in E, and $\delta(s, v) < \infty$, that is v is reachable from s.

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = ∅
9  ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

Lemma 22.6

When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Line 16 sets $v.\pi = u$ iff (u, v) in E, and $\delta(s, v) < \infty$, that is v is reachable from s.

This proves that V_π consists of all vertices reachable from s.

```
BFS(G, s)
1  for each vertex u ∈ G.V − {s}
2      u.color = WHITE
3      u.d = ∞
4      u.π = NIL
5  s.color = GRAY
6  s.d = 0
7  s.π = NIL
8  Q = Ø
9  ENQUEUE(Q, s)
10 while Q ≠ Ø
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18     u.color = BLACK
```

Lemma 22.6

When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Line 16 sets $v.\pi = u$ iff (u, v) in E, and $\delta(s, v) < \infty$, that is v is reachable from s.

This proves that V_π consists of all vertices reachable from s.

As G_π forms a tree, it contains unique simple path from s to every vertex in V_π.

```
BFS(G, s)
1   for each vertex u ∈ G.V − {s}
2       u.color = WHITE
3       u.d = ∞
4       u.π = NIL
5   s.color = GRAY
6   s.d = 0
7   s.π = NIL
8   Q = ∅
9   ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

Lemma 22.6

When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Line 16 sets $v.\pi = u$ iff (u, v) in E, and $\delta(s, v) < \infty$, that is v is reachable from s.

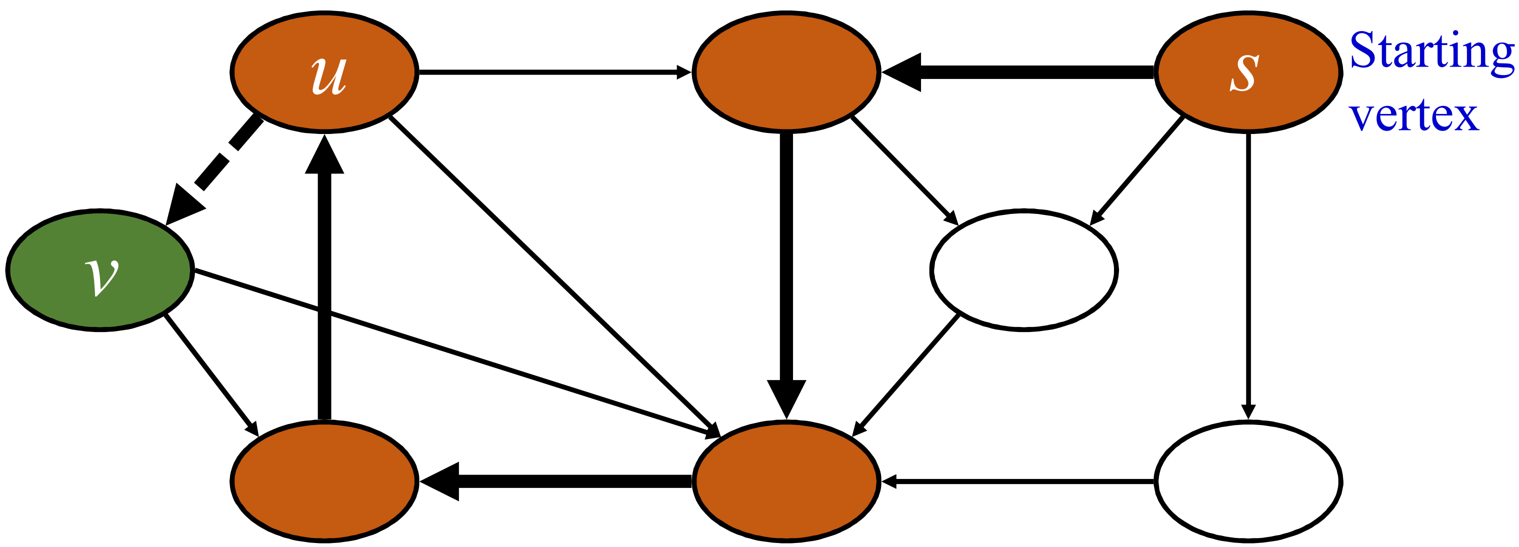
This proves that V_π consists of all vertices reachable from s.

As G_π forms a tree, it contains unique simple path from s to every vertex in V_π.

Theorem 22.5 proves that each such path is a shortest path.

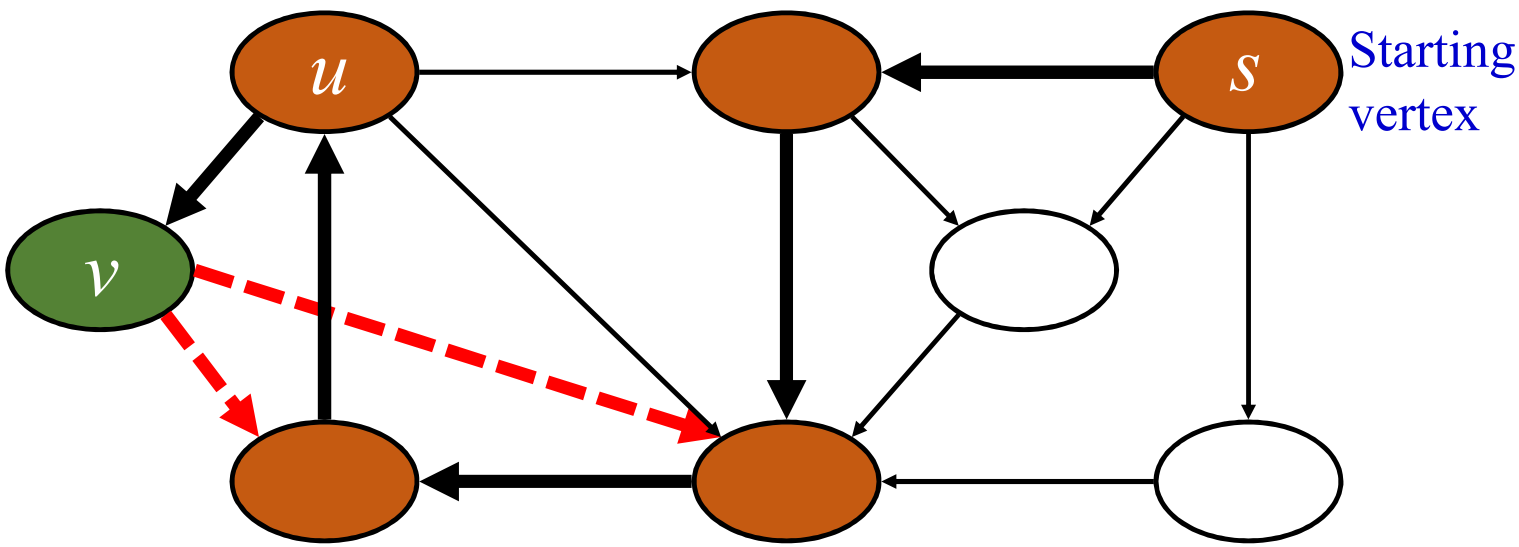
Depth-First Search

- Explore "deeper" in the graph whenever possible



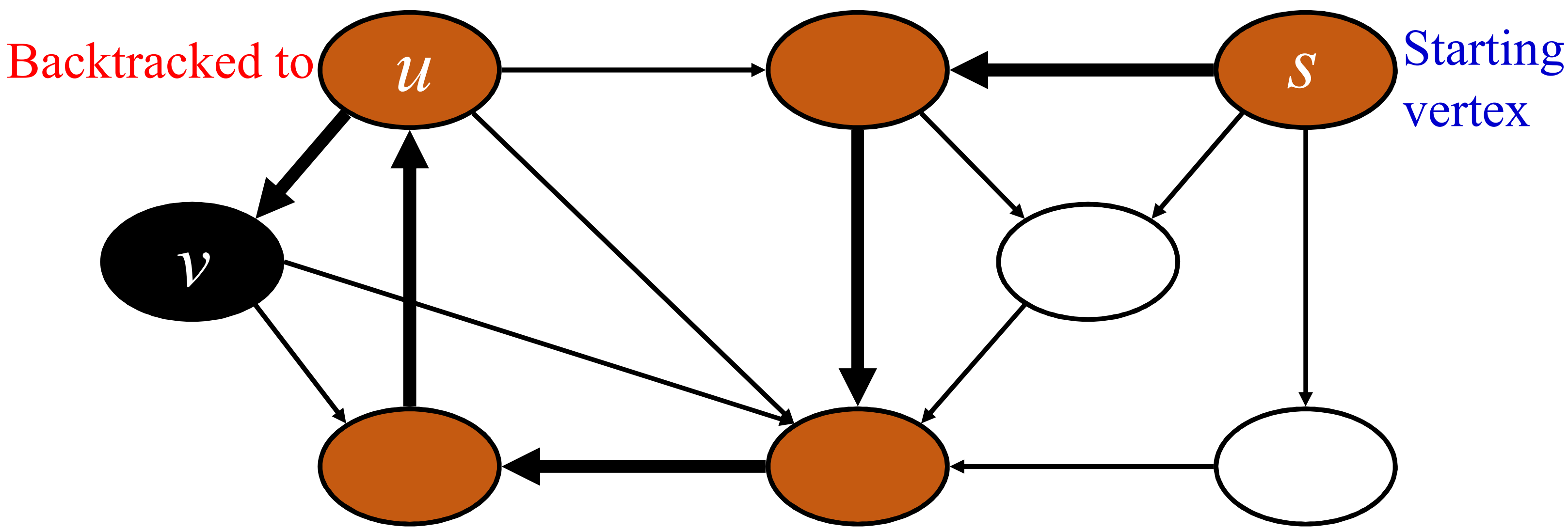
Depth-First Search

- Explore "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges



Depth-First Search

- Explore "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex v that still has unexplored edges
- When all of v's edges have been explored, backtrack to the vertex from which v was discovered (i.e., its parent)



Depth-First Search

- Explore "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex v that still has unexplored edges
- When all of v's edges have been explored, backtrack to the vertex from which v was discovered (i.e., its parent)

- Vertices initially colored white
- Then colored grey when discovered
- Then black when finished

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

- records predecessors in π attributes
- Produces multiple trees

- we define the ***predecessor subgraph*** of G as $G_\pi = (V, E_\pi)$, where

  $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq NIL\}$

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

- Records timestamps for each vertex, *v*
  - Discovery time, *d*: when *v* is discovered
  - Finishing time, *f* : when *v's adjacency list* is finished

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

Θ(V)

Θ(V) EXCLUDING the time required for DFS-VISIT().

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

$\Theta(V)$

$\Theta(V)$ EXCLUDING the time required for DFS-VISIT().

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

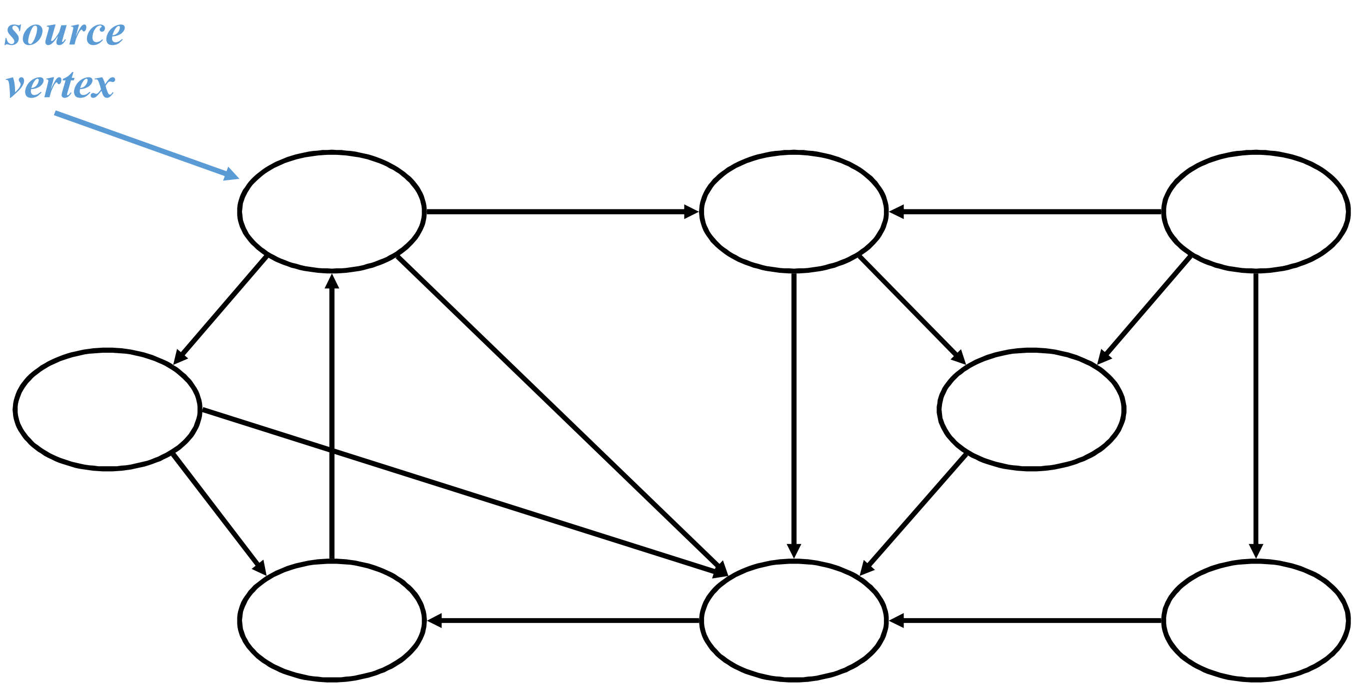
$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

$\Theta(V + E)$

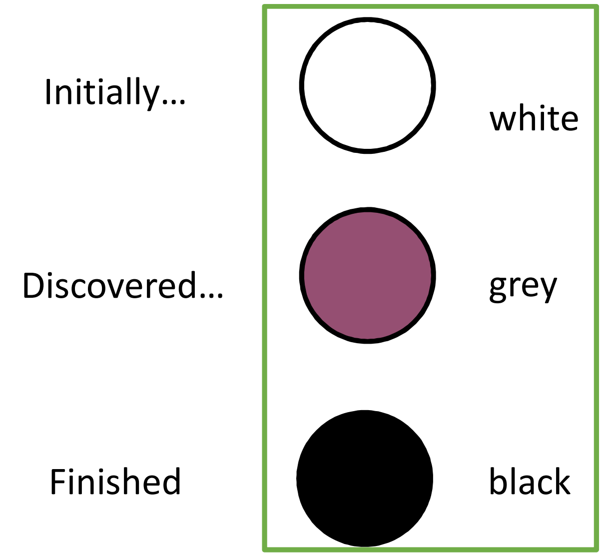
How many times DFS-Visit() is called?

- The procedure DFS-VISIT is called exactly once for each vertex since:
  - the vertex u on which DFS-VISIT() is invoked must be white
  - the first thing DFS-VISIT does is paint vertex u gray

DFS Example



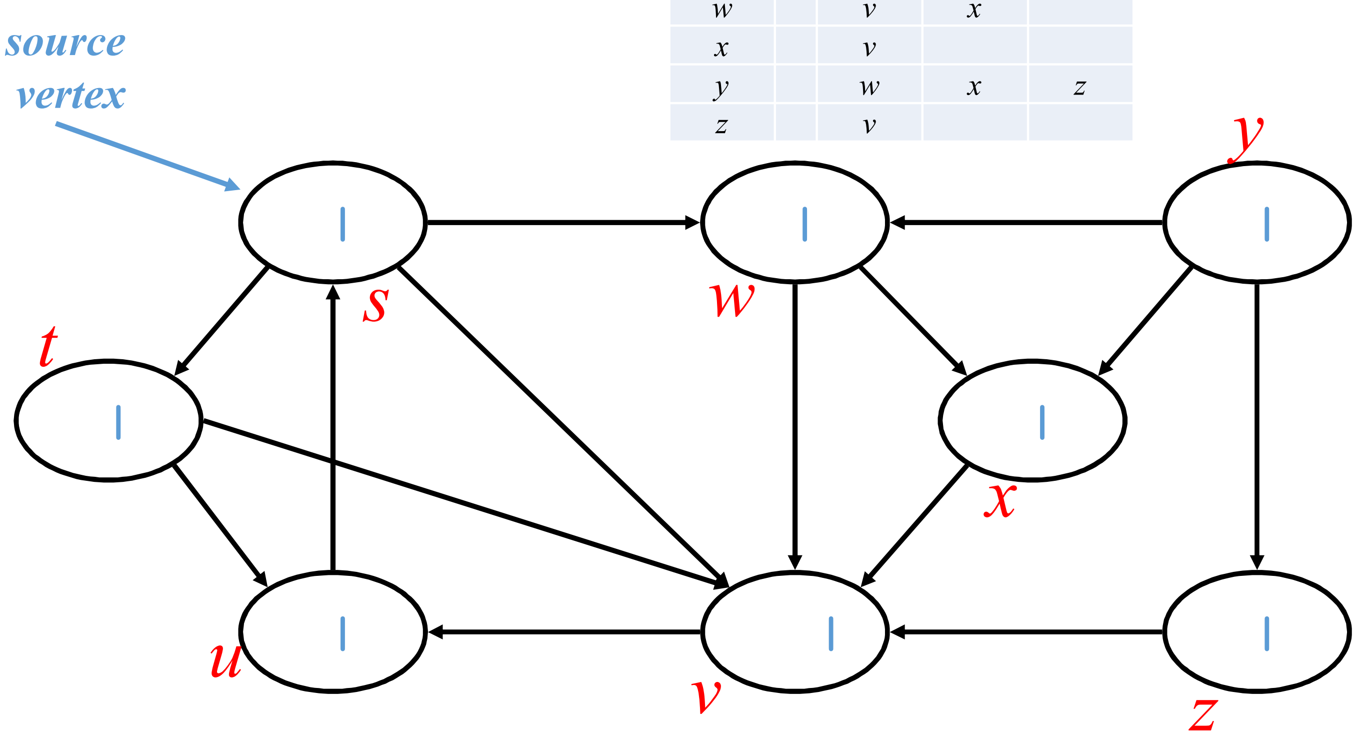
source
vertex

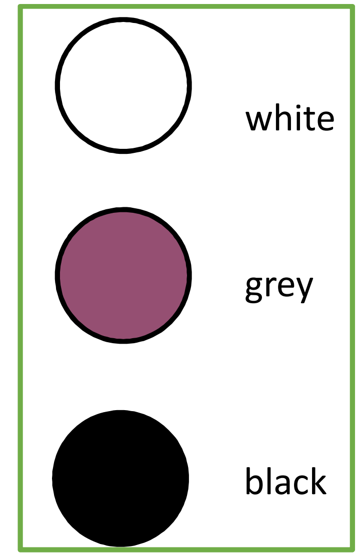


Initially…
white
Discovered…
grey
Finished
black

DFS Example

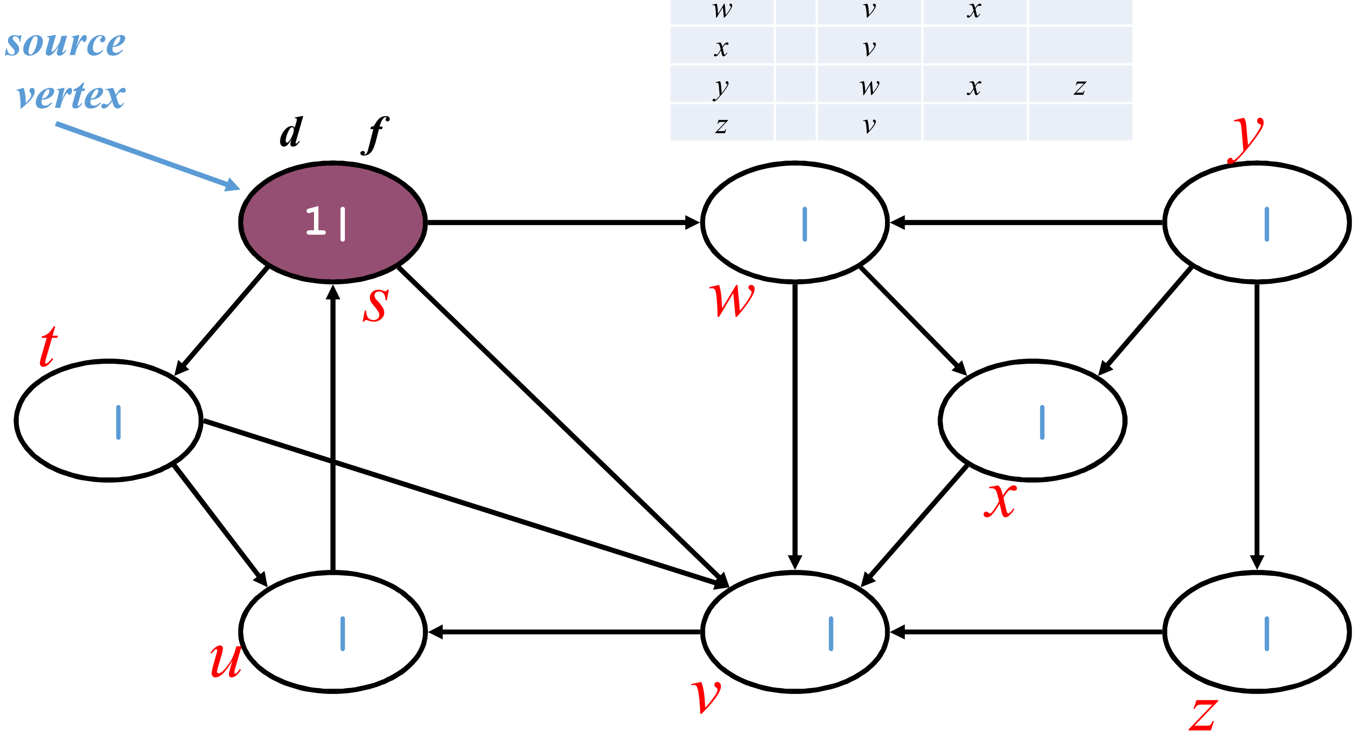
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

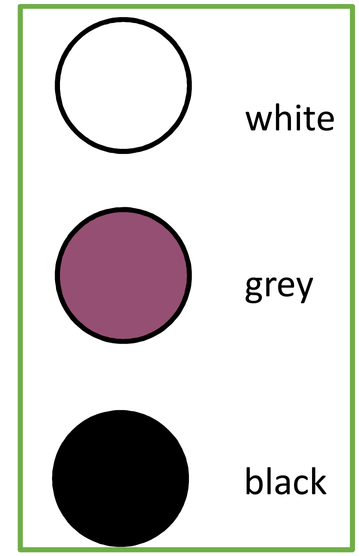




DFS Example

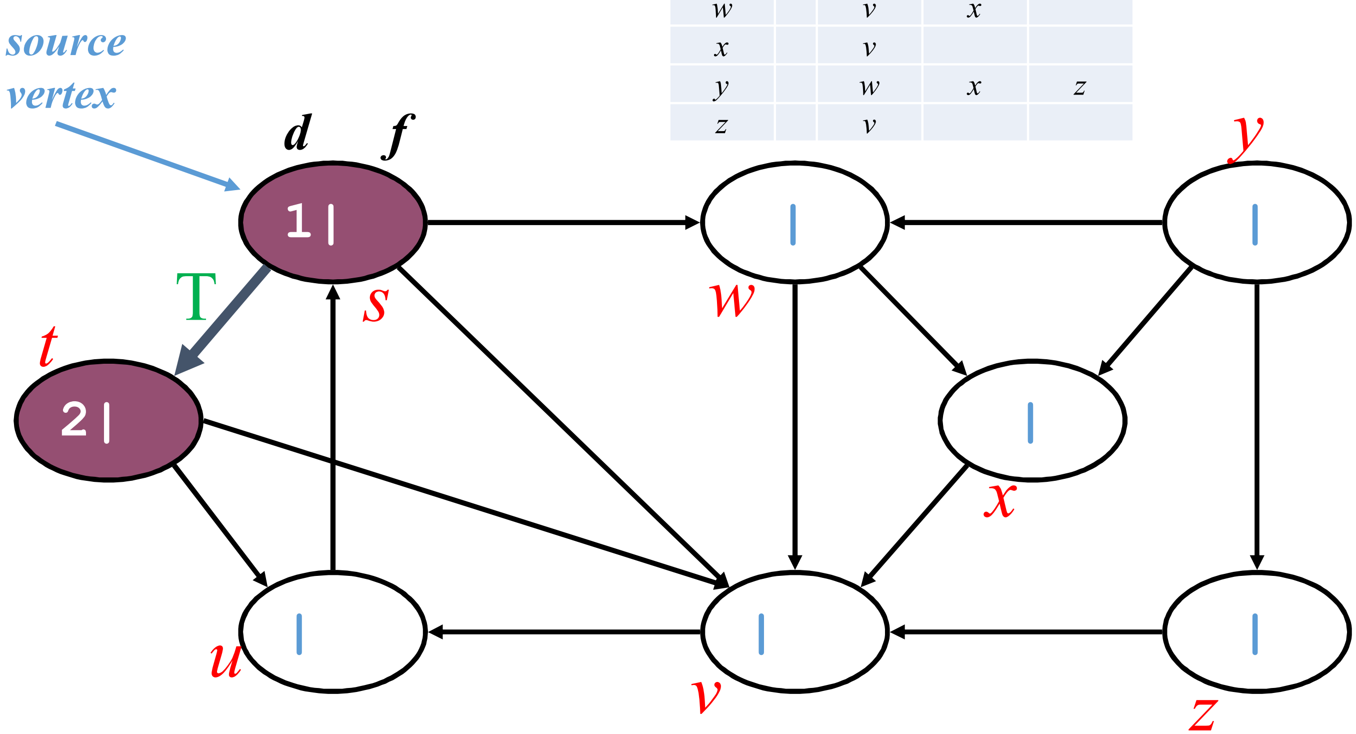
Vertices		Adjacency list	
s	*t*	*v*	*w*
t	*u*	*v*	
u	*s*		
v	*u*		
w	*v*	*x*	
x	*v*		
y	*w*	*x*	*z*
z	*v*		

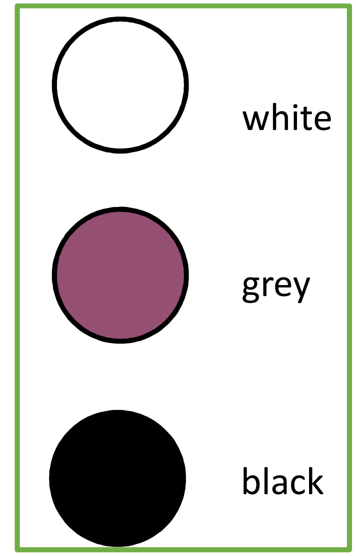




DFS Example

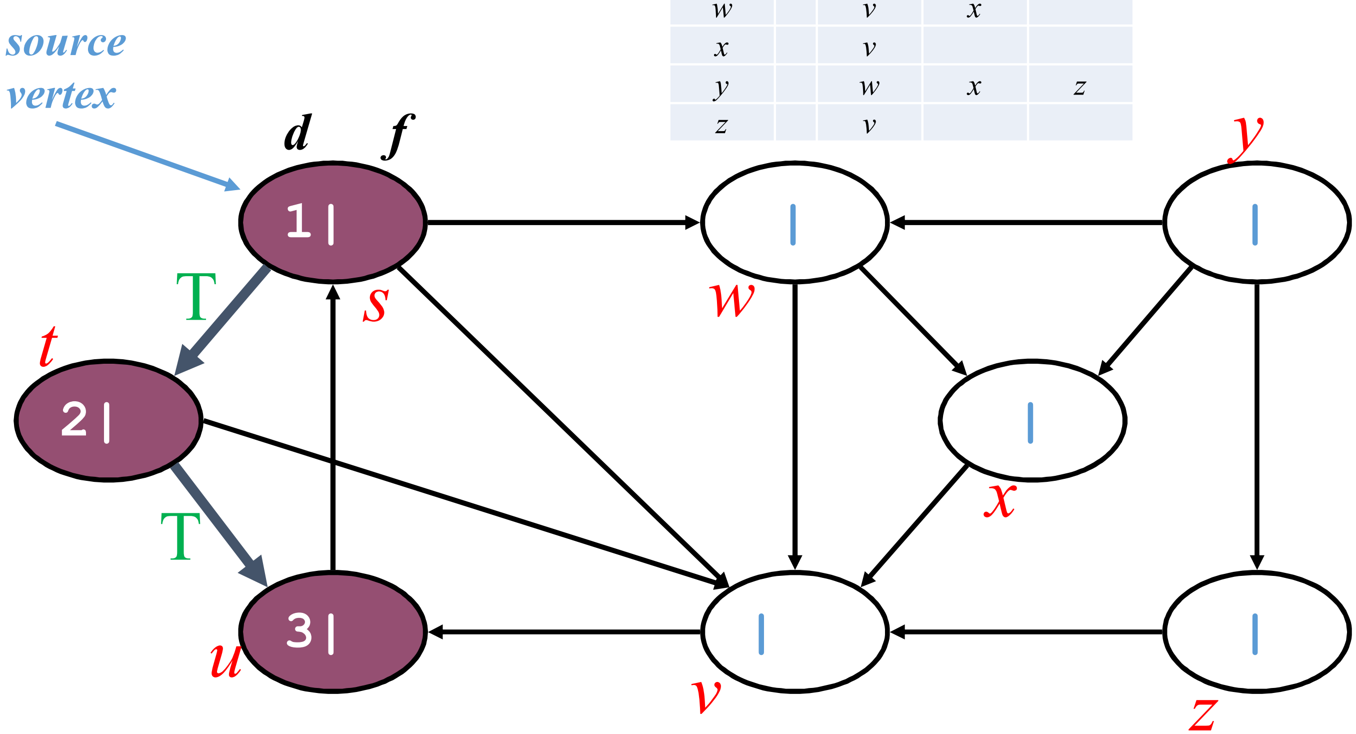
Vertices	Adjacency list		
s	*t*	*v*	*w*
t	*u*	*v*	
u	*s*		
v	*u*		
w	*v*	*x*	
x	*v*		
y	*w*	*x*	*z*
z	*v*		

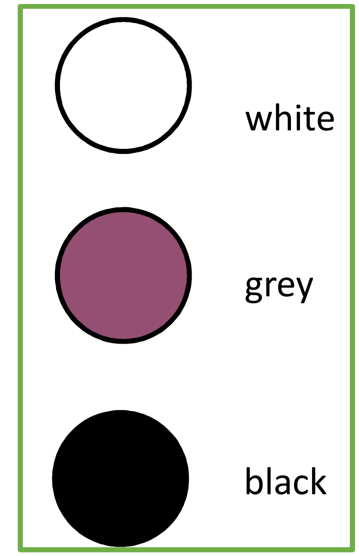




DFS Example

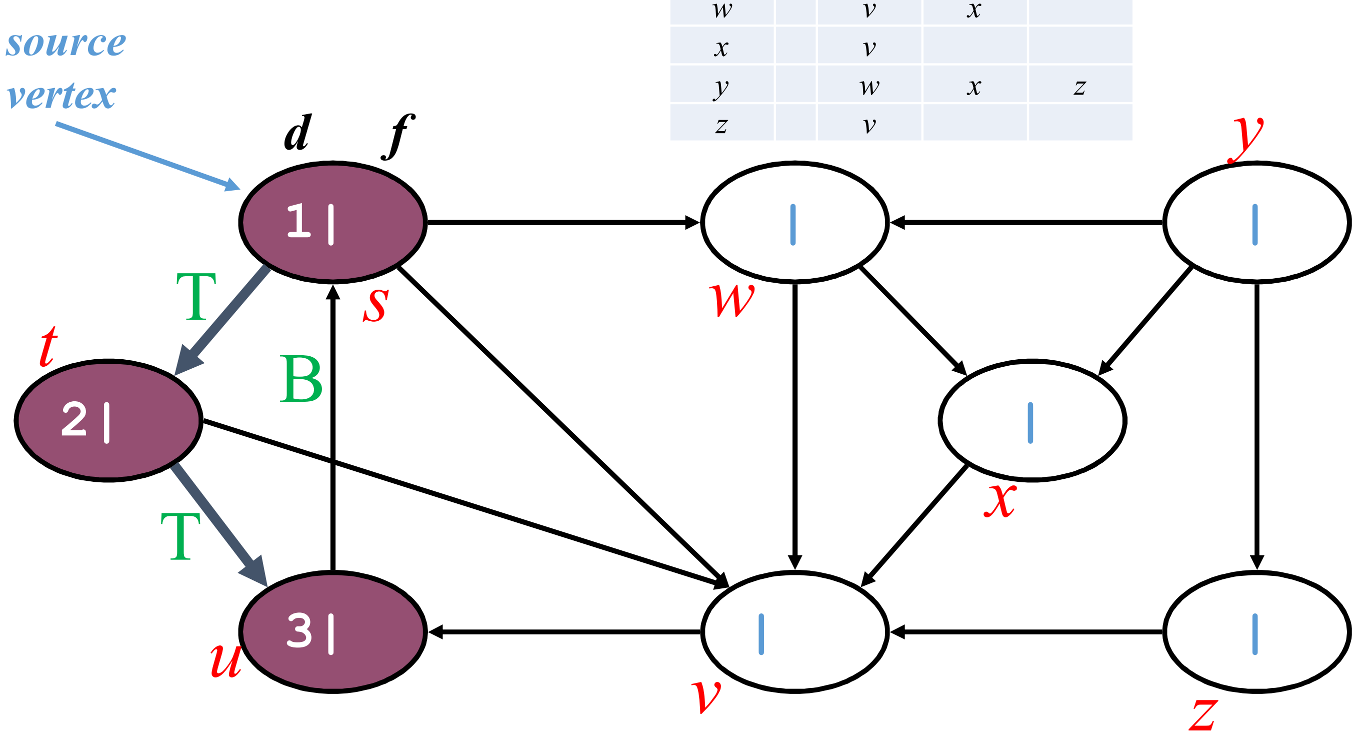
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

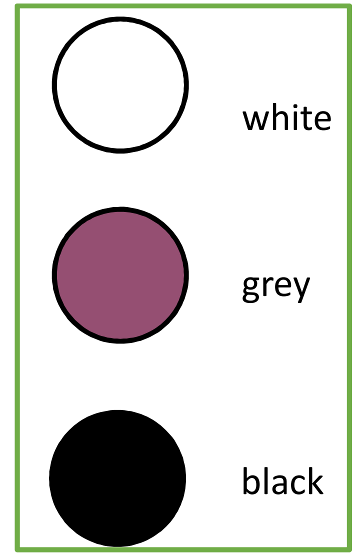




DFS Example

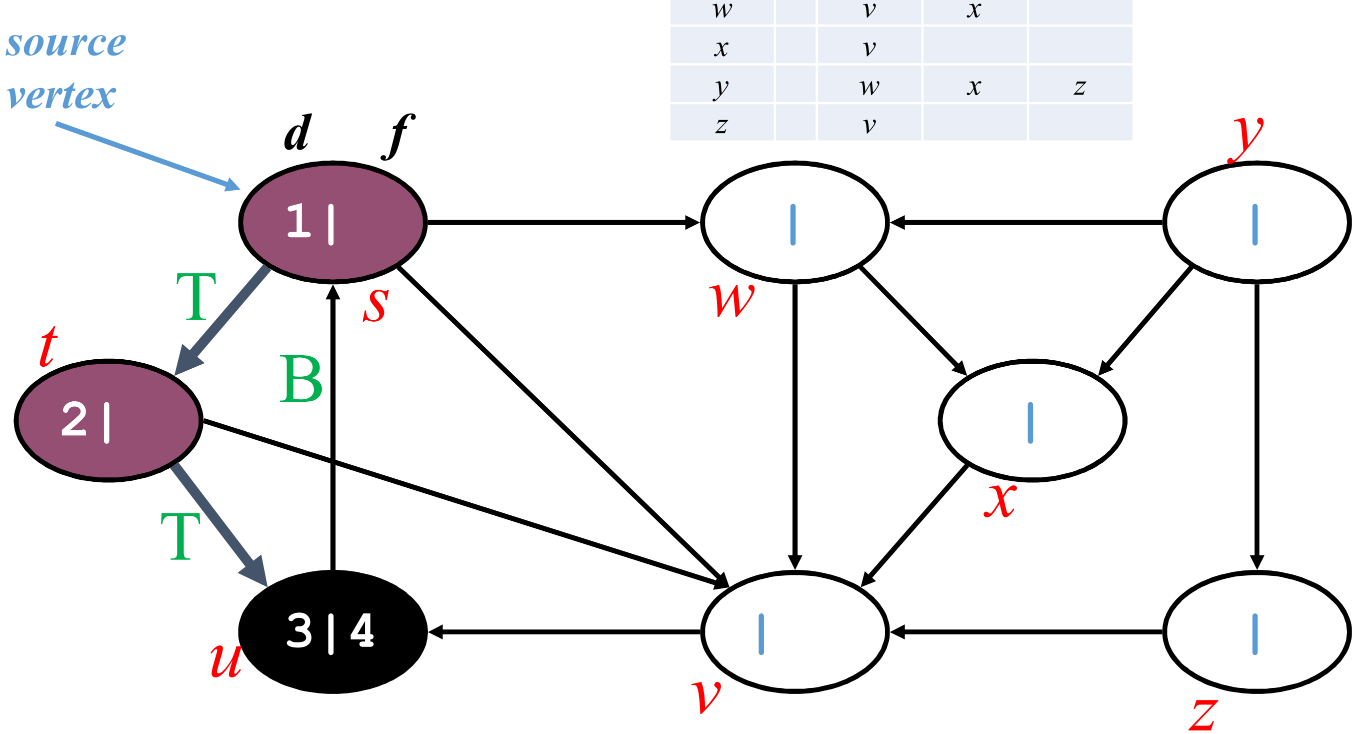
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

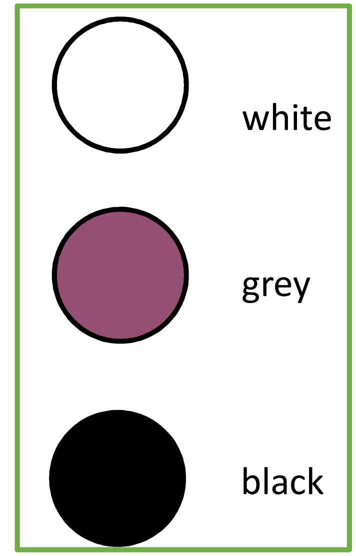




DFS Example

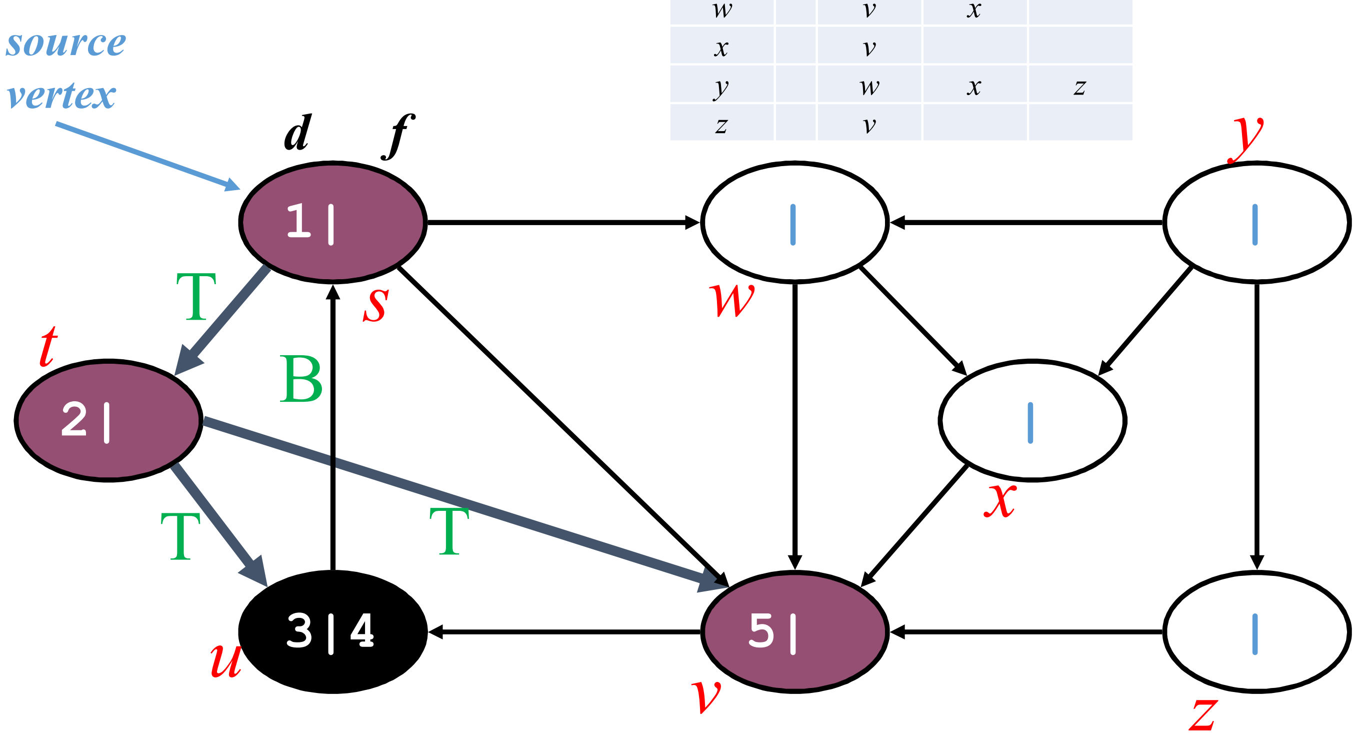
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

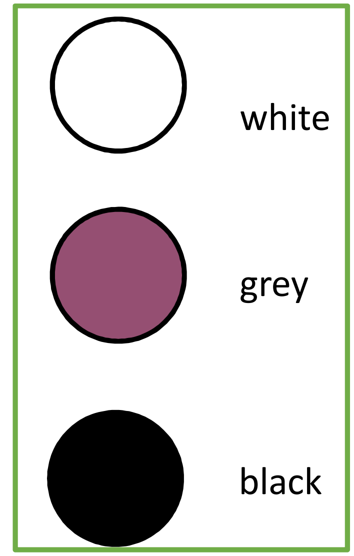




DFS Example

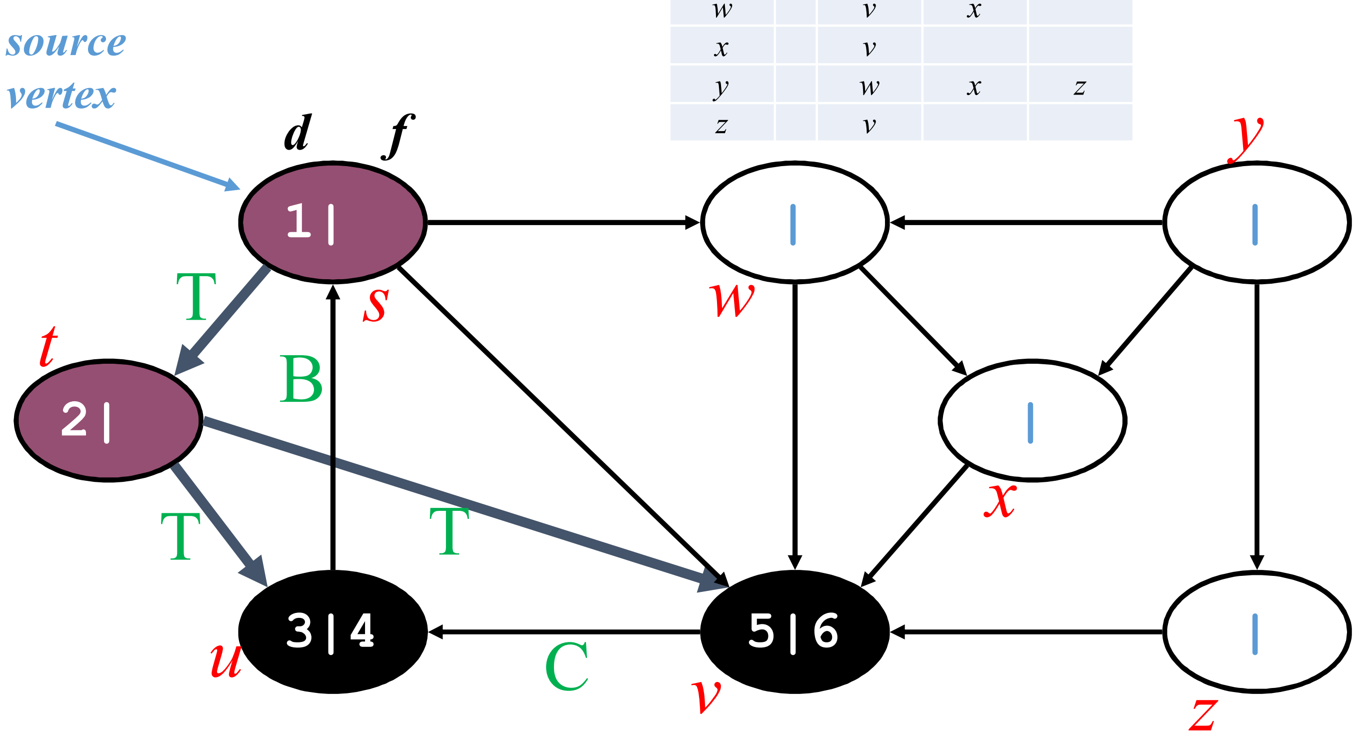
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

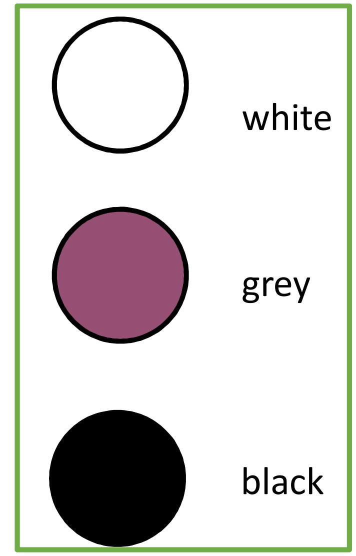




DFS Example

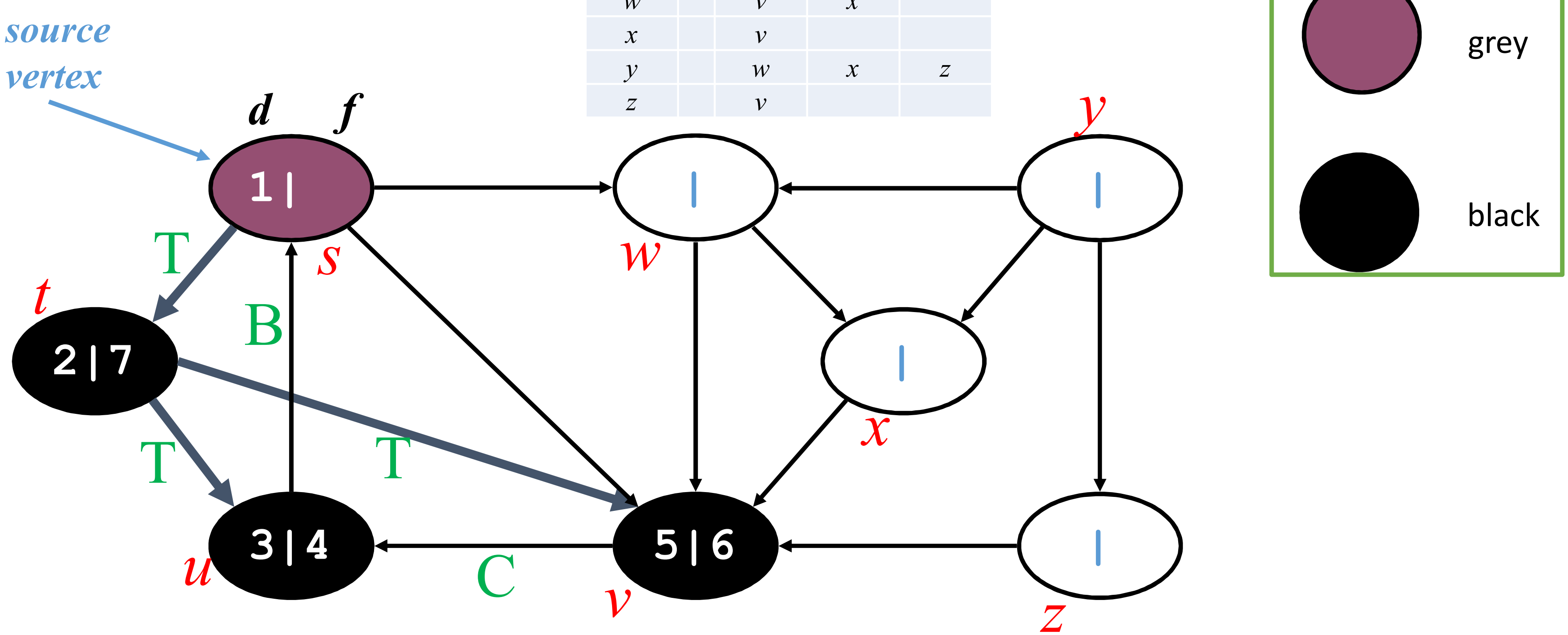
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		





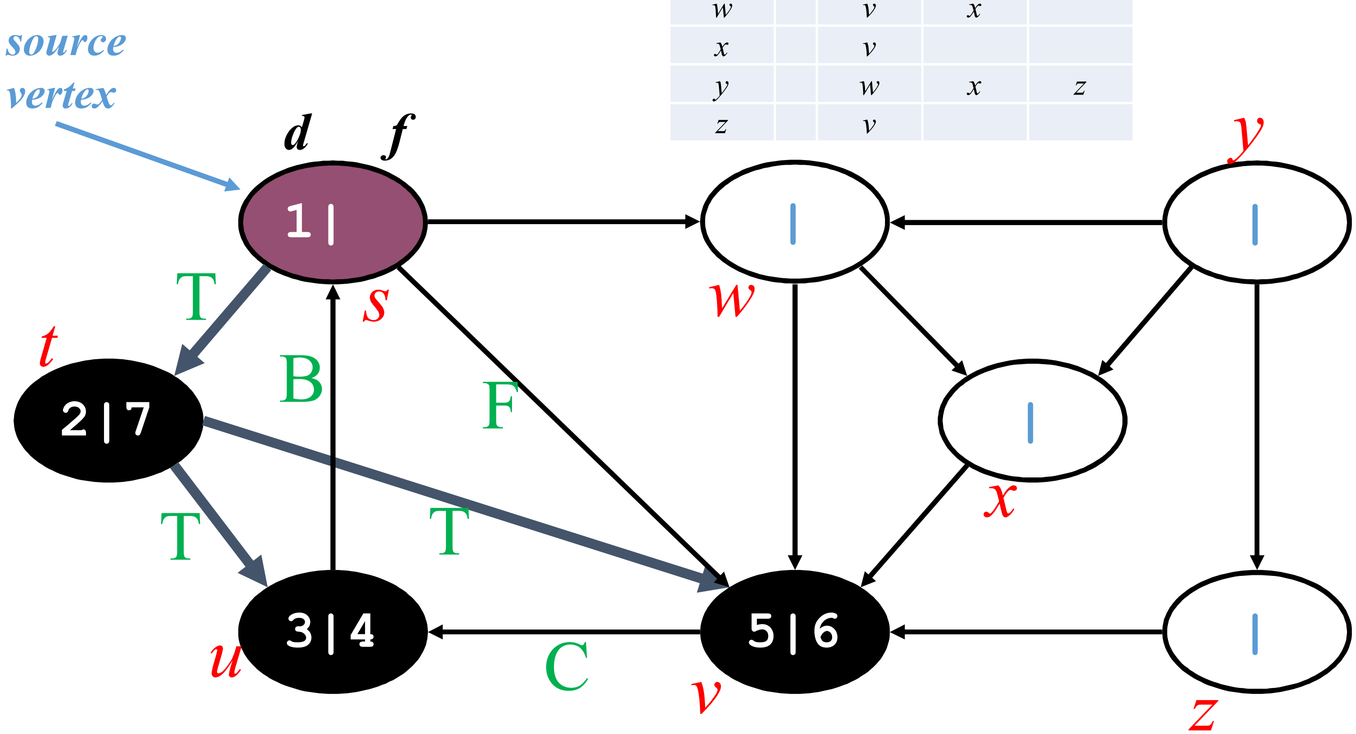
DFS Example

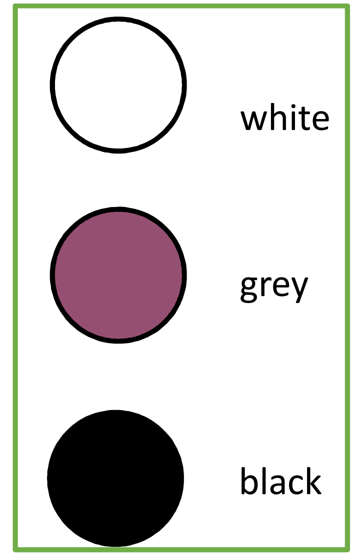
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		



DFS Example

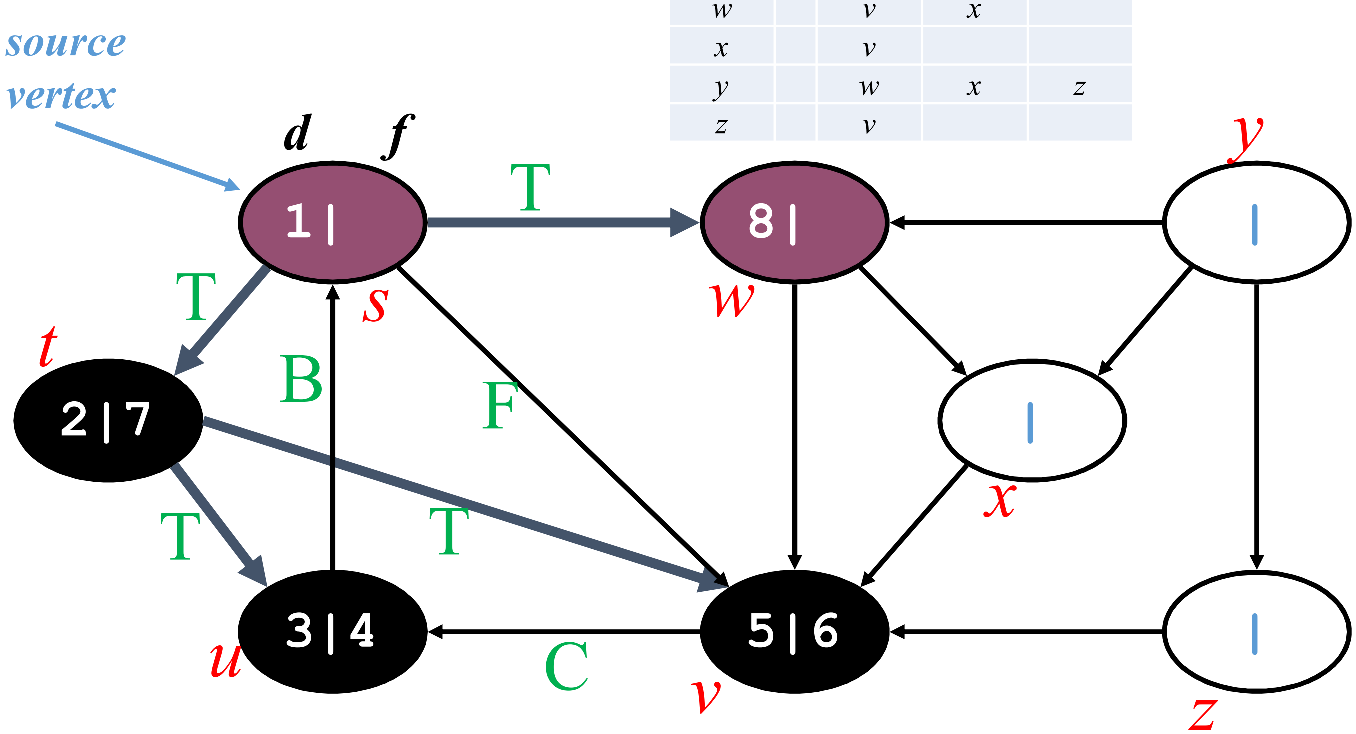
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

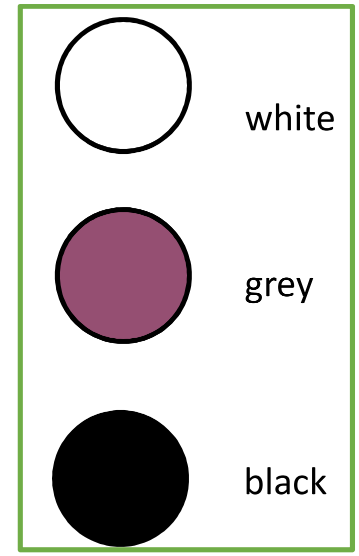




DFS Example

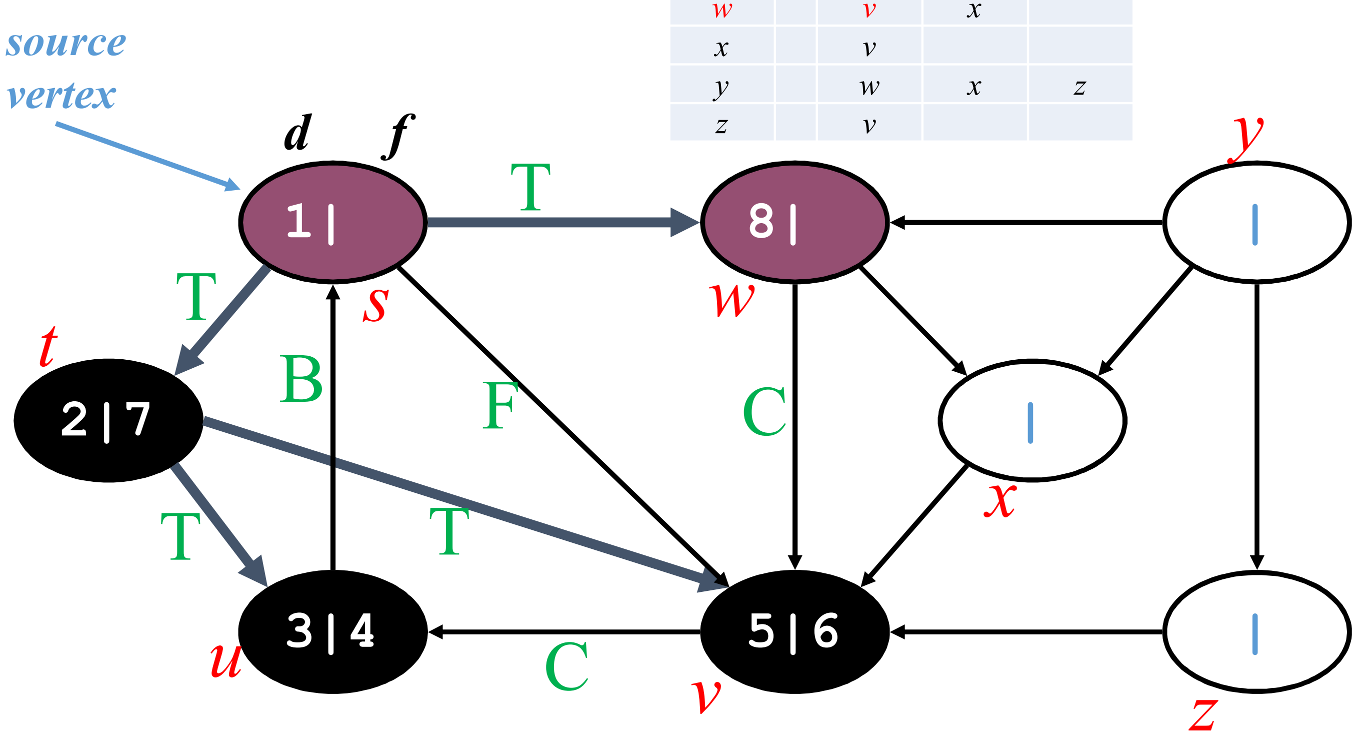
Vertices		Adjacency list	
s	*t*	*v*	*w*
t	*u*	*v*	
u	*s*		
v	*u*		
w	*v*	*x*	
x	*v*		
y	*w*	*x*	*z*
z	*v*		

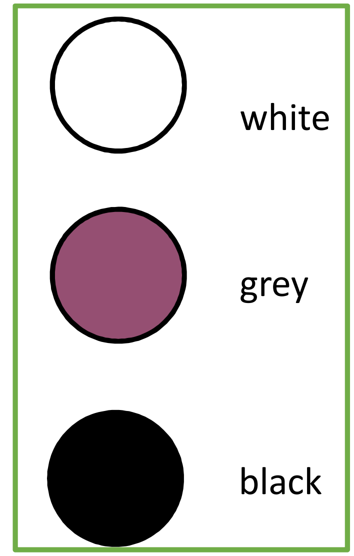




DFS Example

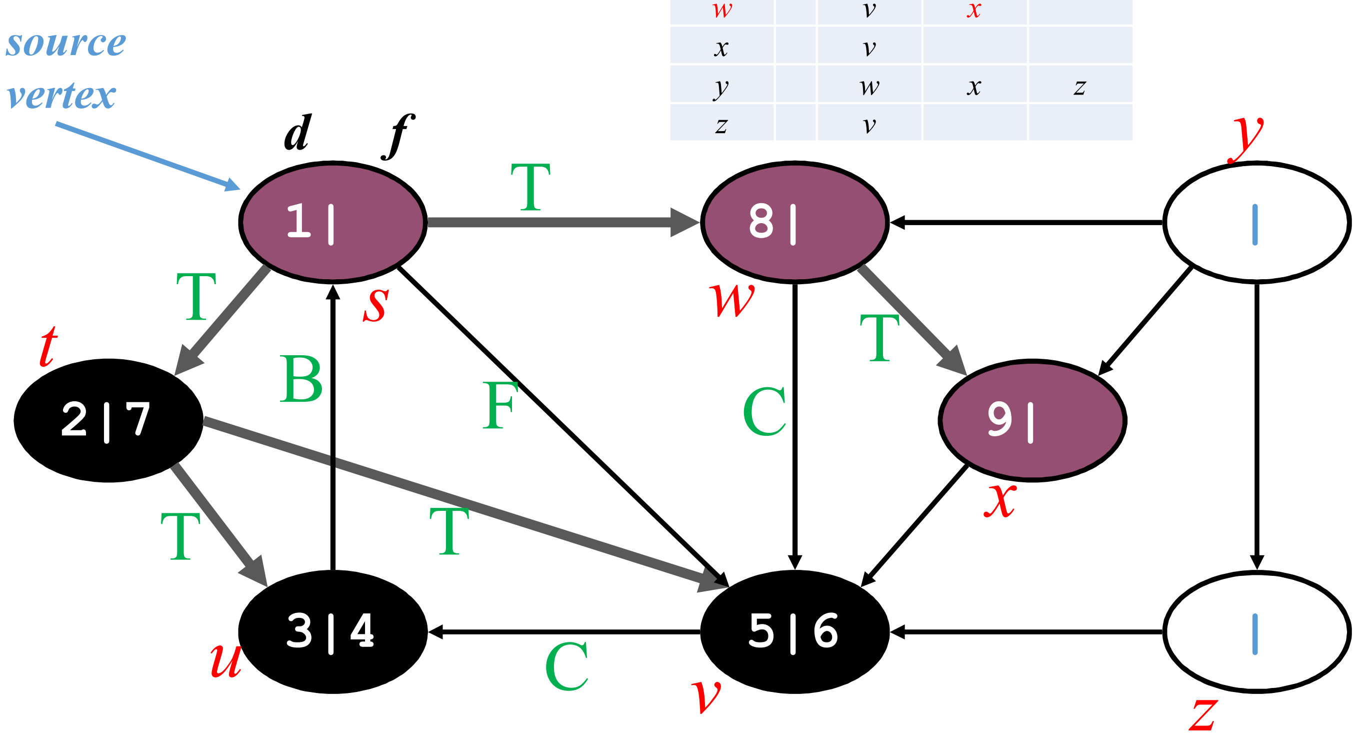
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

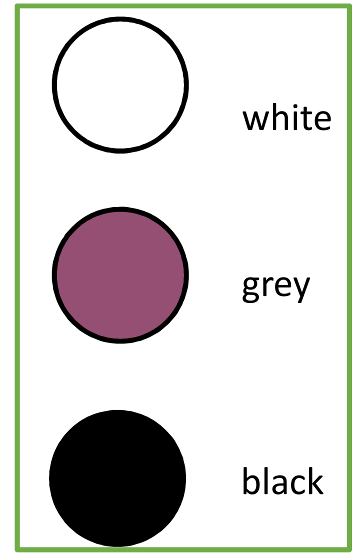




DFS Example

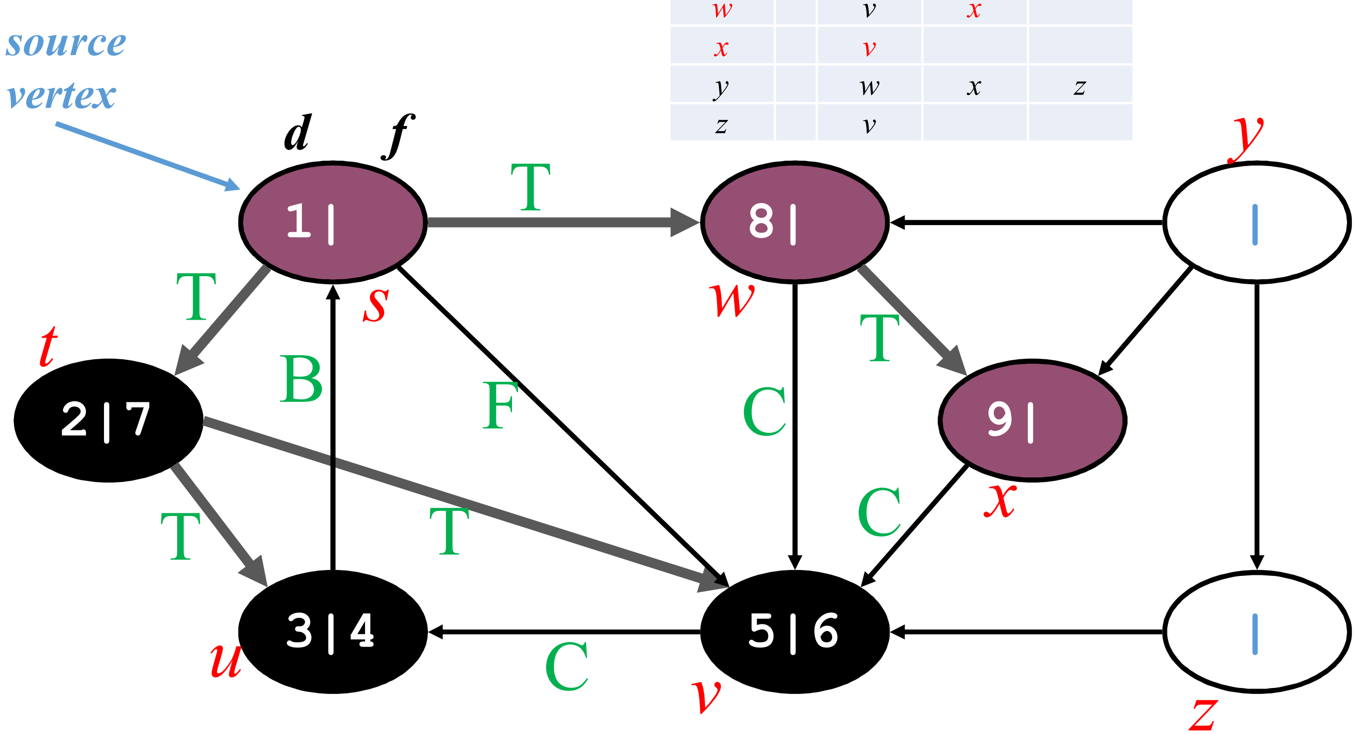
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

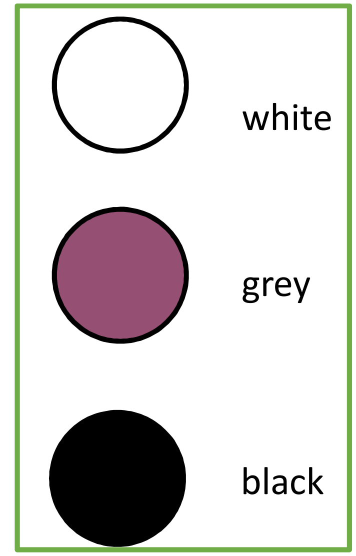




DFS Example

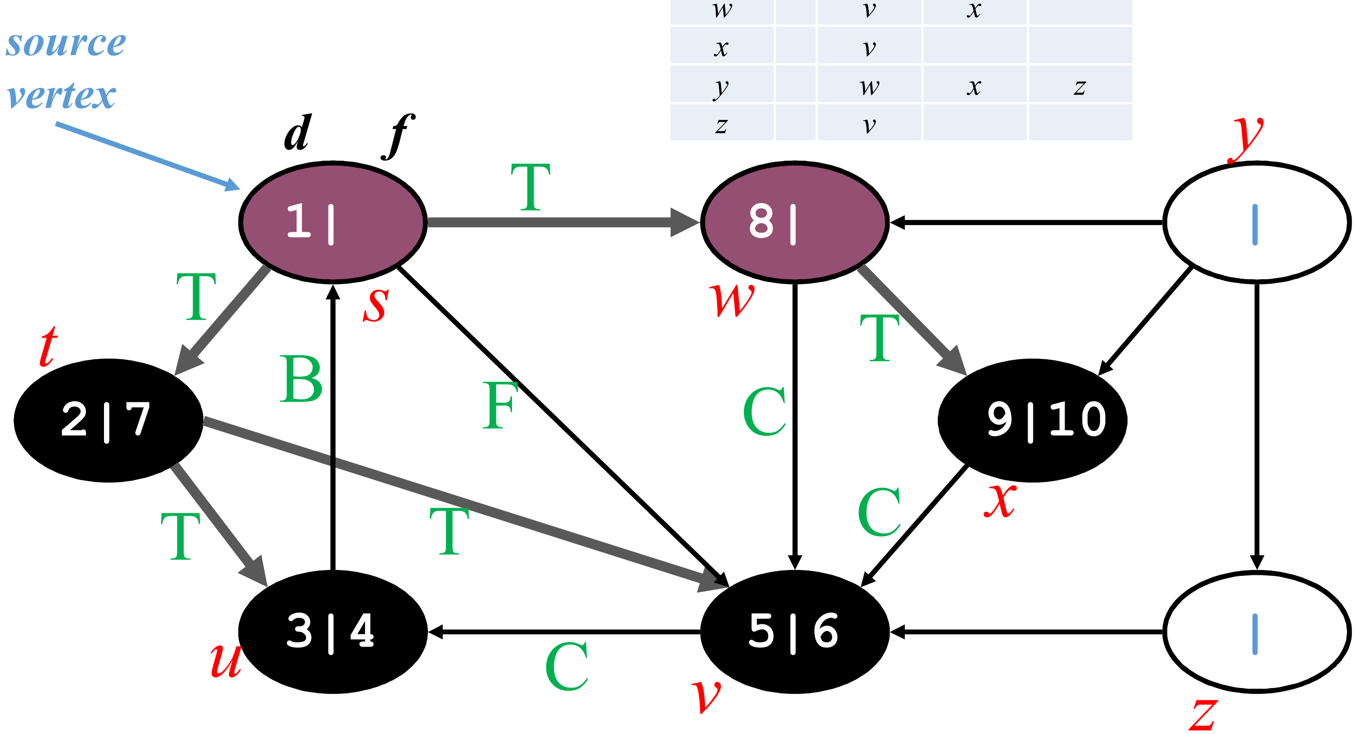
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

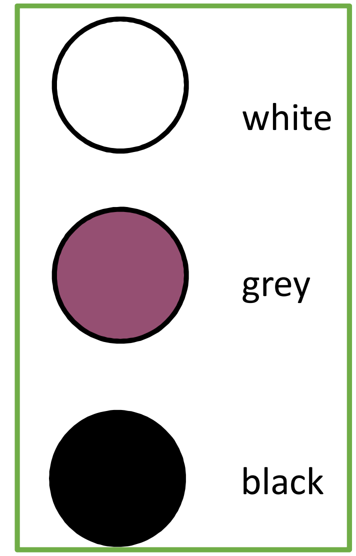




DFS Example

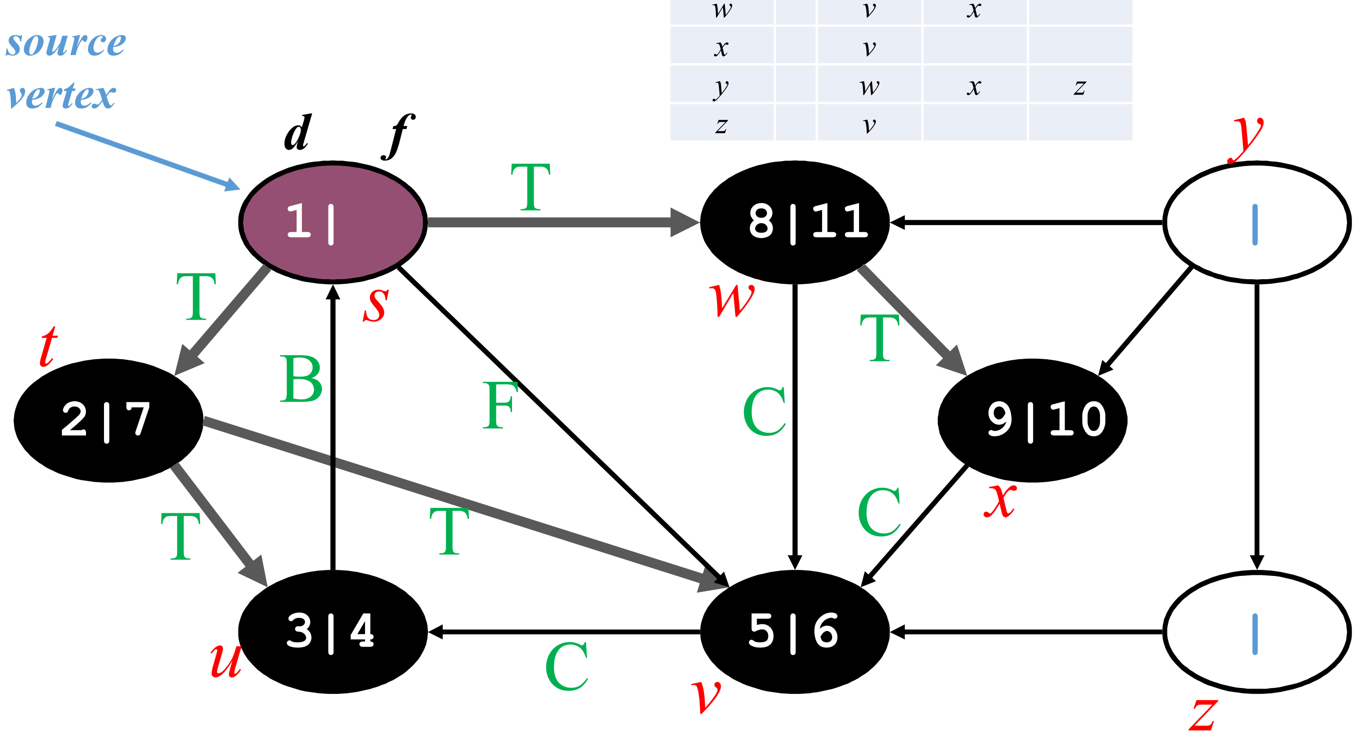
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

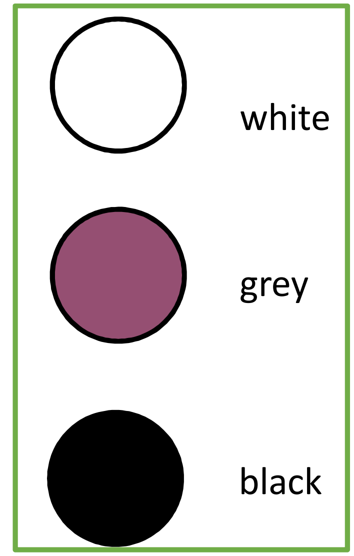




DFS Example

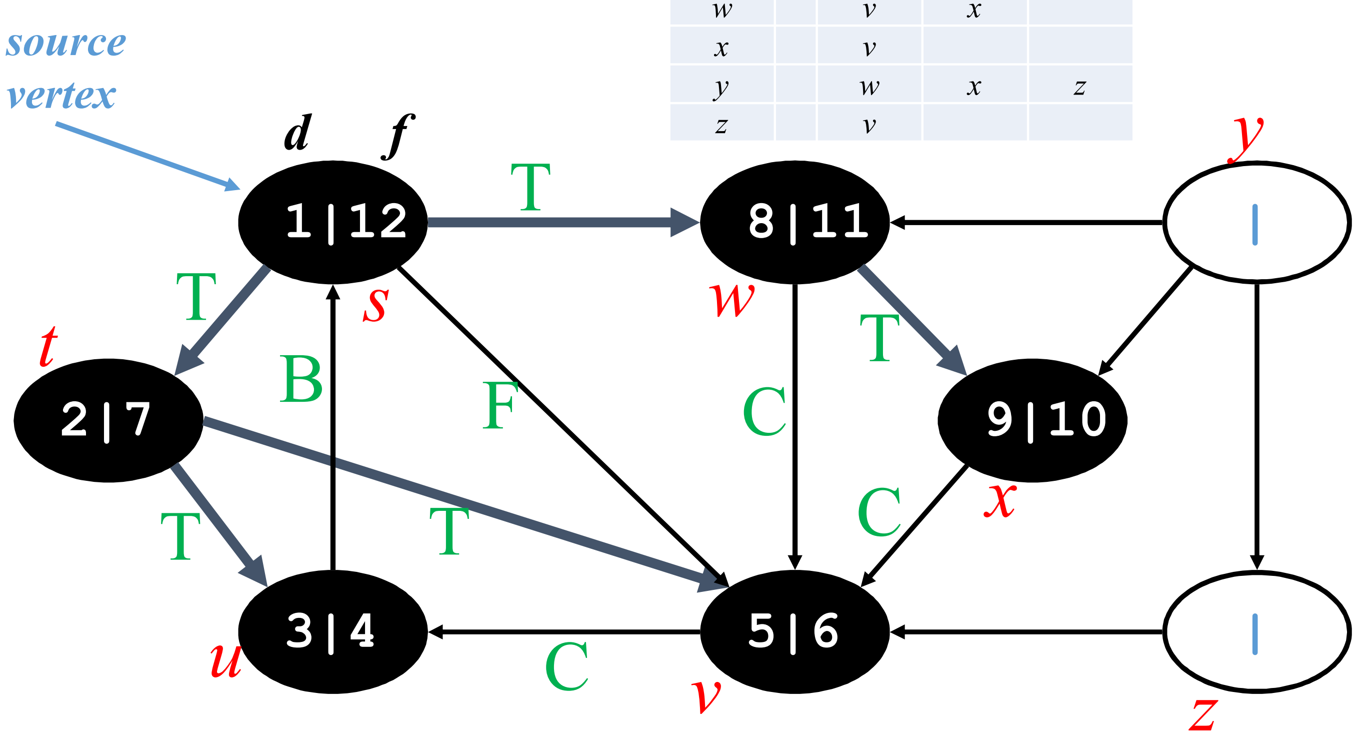
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

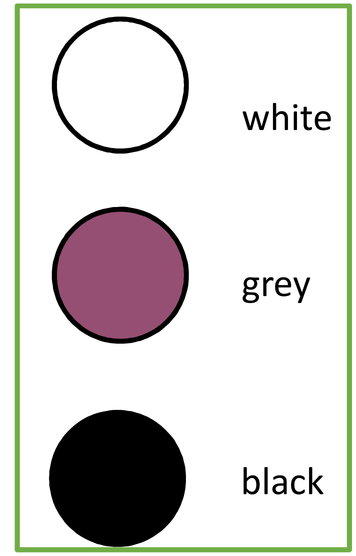




DFS Example

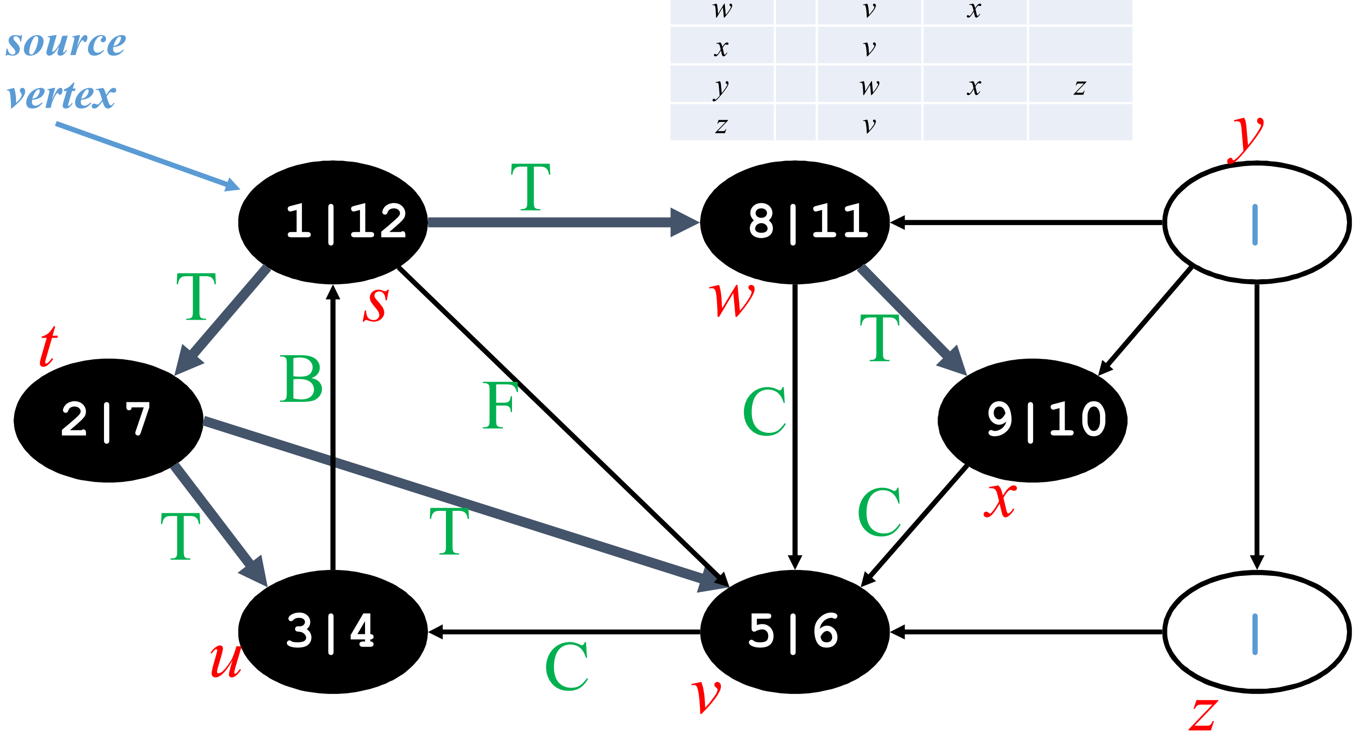
Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

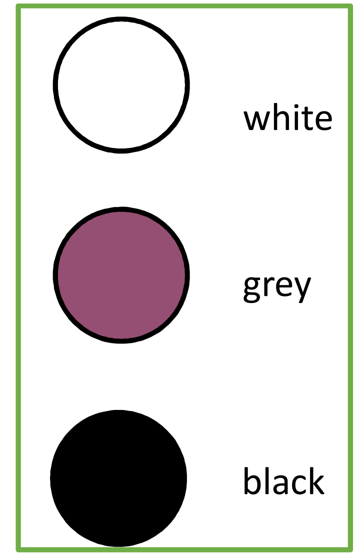




DFS Example

Vertices	Adjacency list		
s	*t*	*v*	*w*
t	*u*	*v*	
u	*s*		
v	*u*		
w	*v*	*x*	
x	*v*		
y	*w*	*x*	*z*
z	*v*		





We have two WHITE vertices remaining.
They are unreachable from *s*

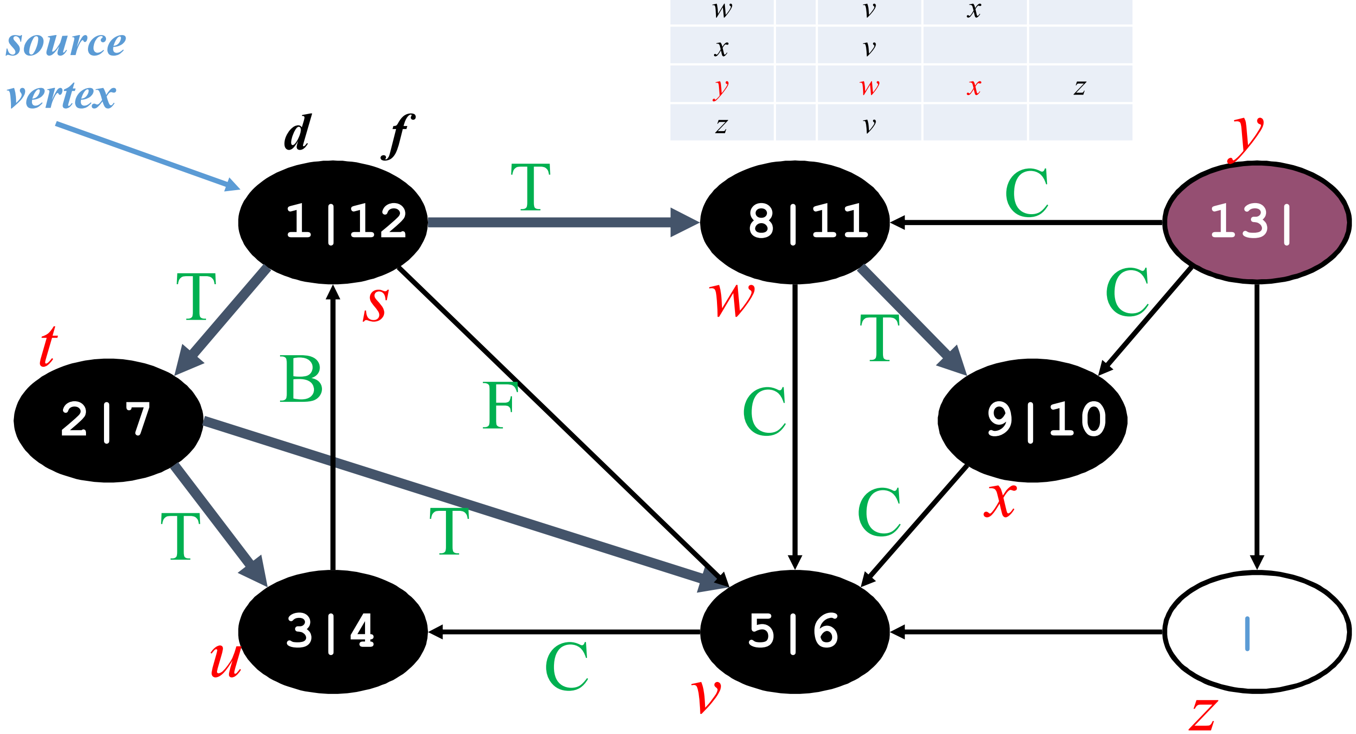
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

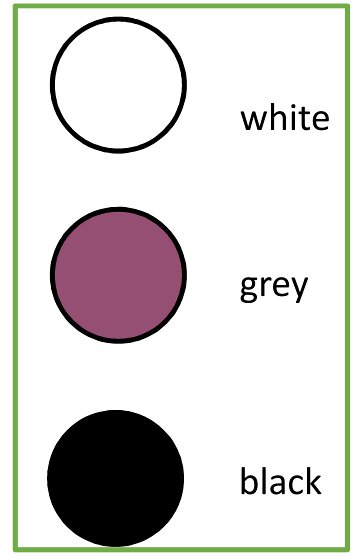
```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

Vertices		**Adjacency list**		
s		t	v	w
t		u	v	
u		s		
v		u		
w		v	x	
x		v		
y		w	x	z
z		v		

DFS Example

Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

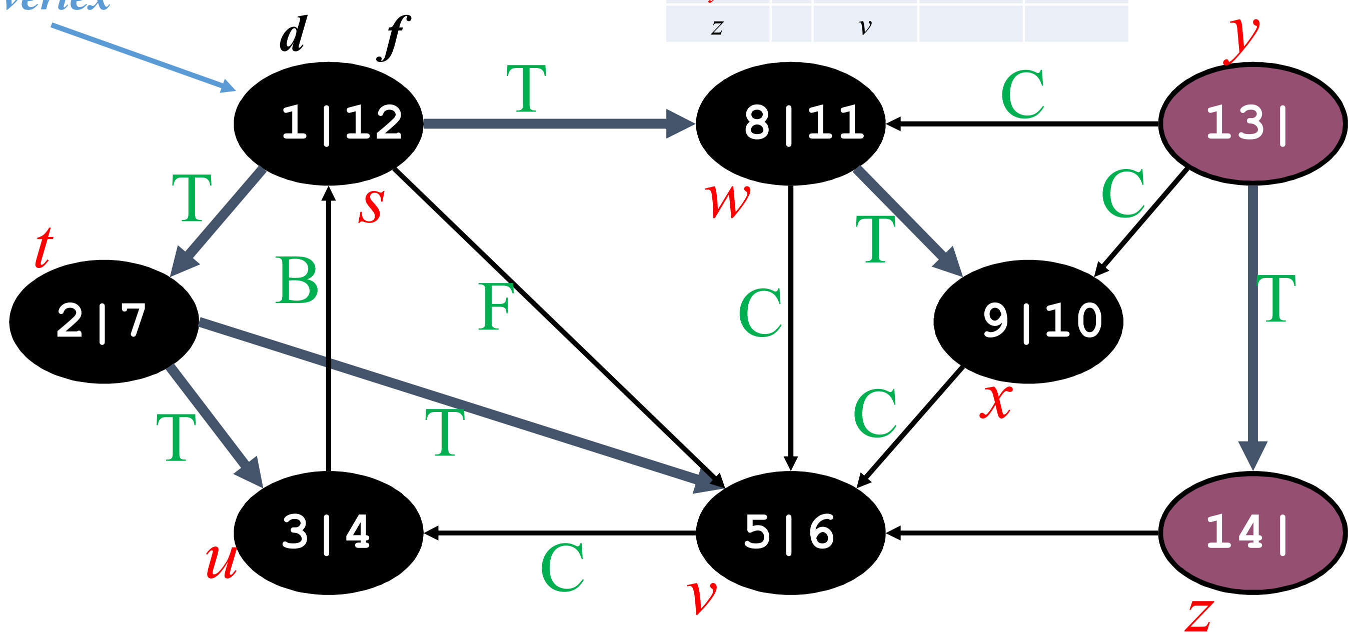


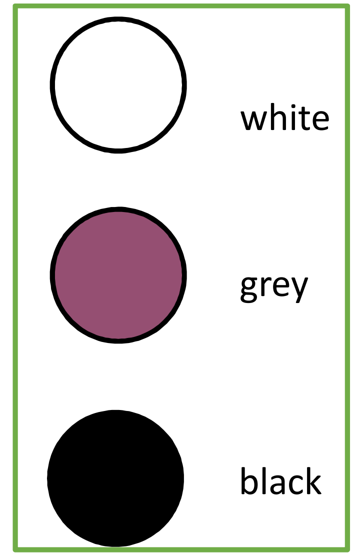


DFS Example

Vertices		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

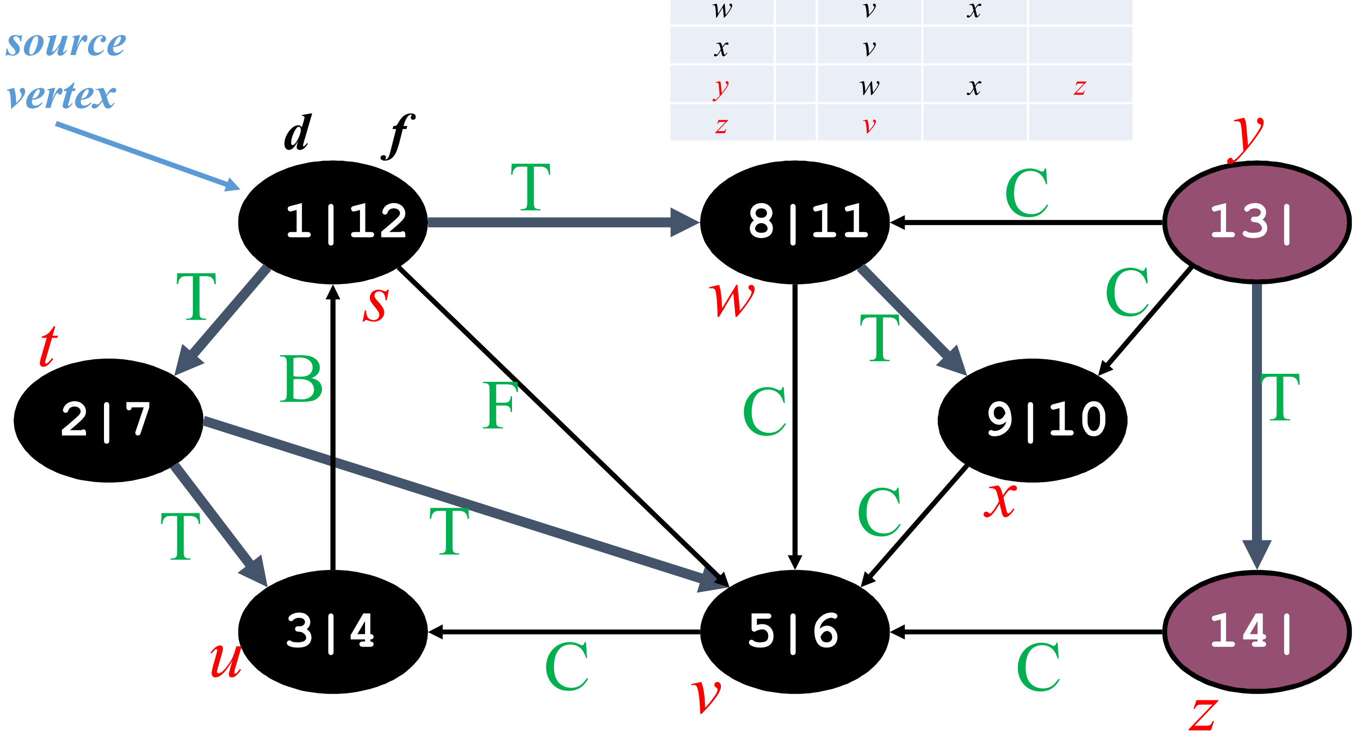
source vertex

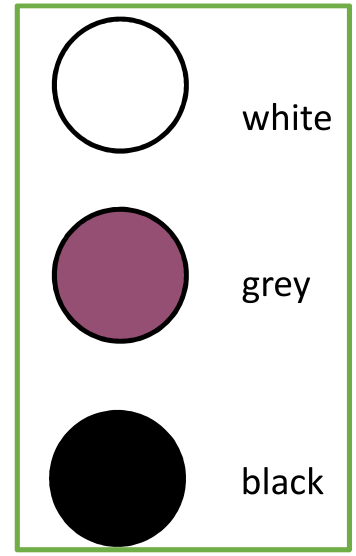




DFS Example

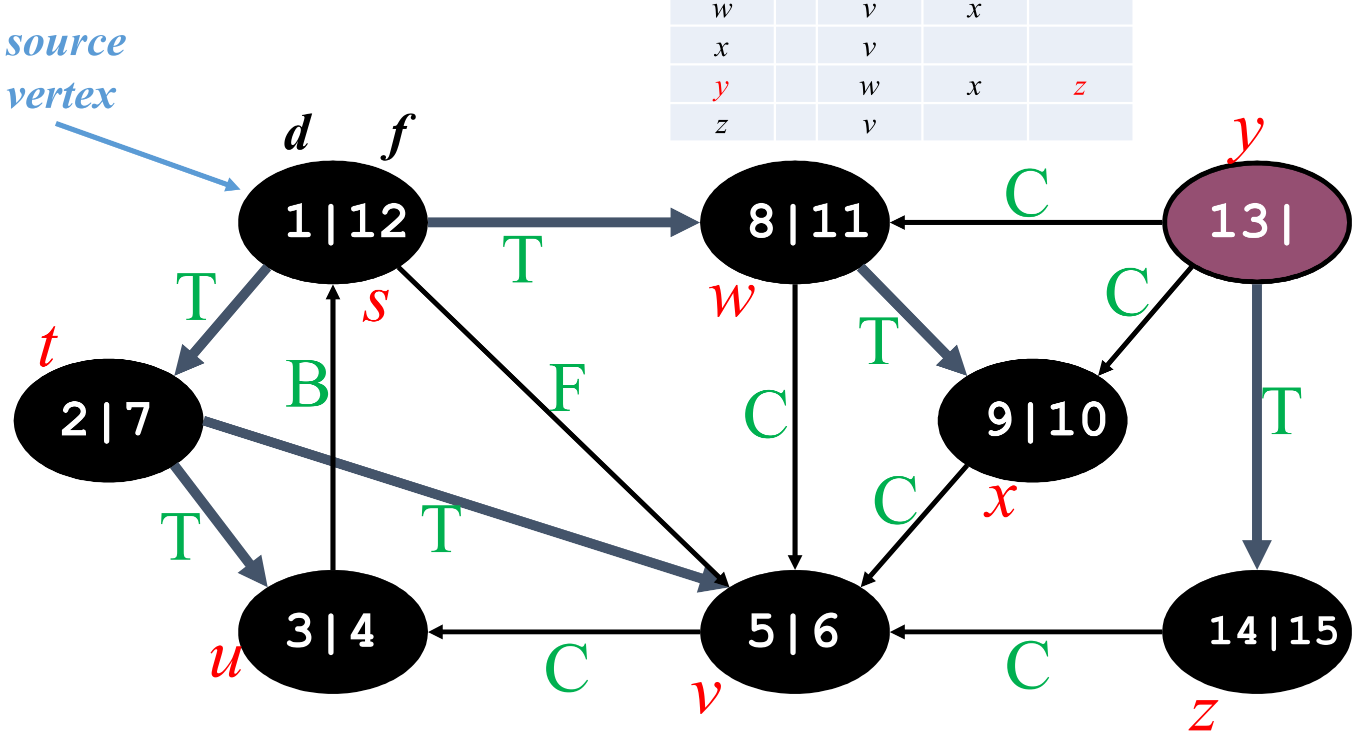
Nodes		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

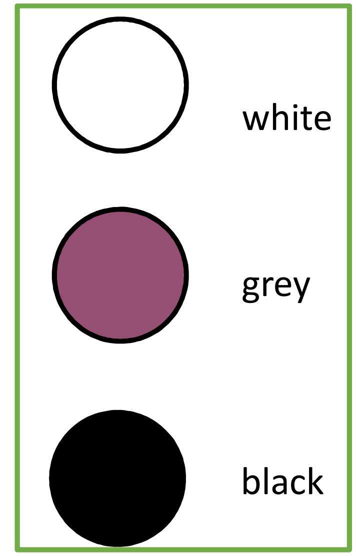




DFS Example

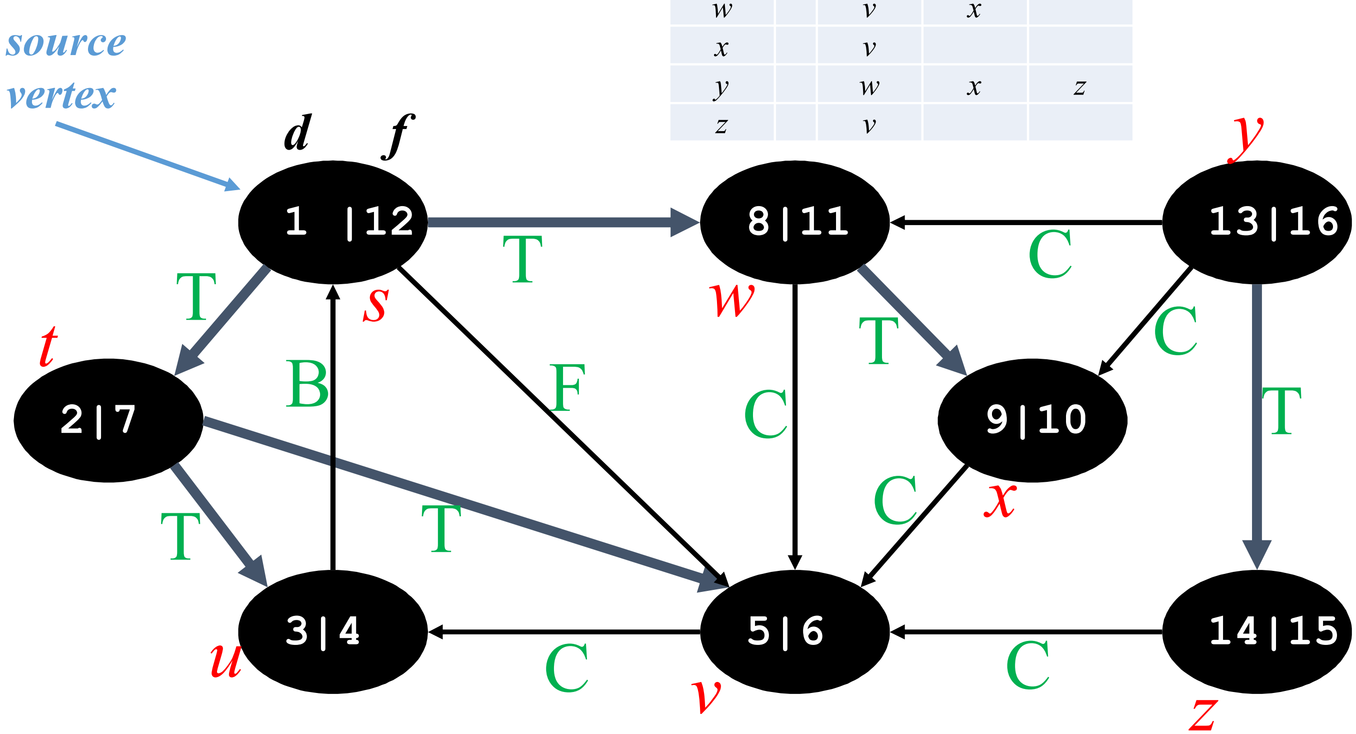
Nodes		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		

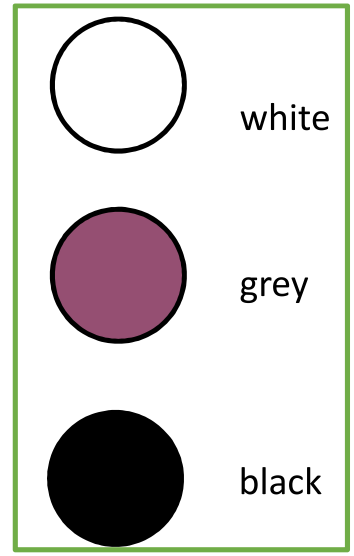




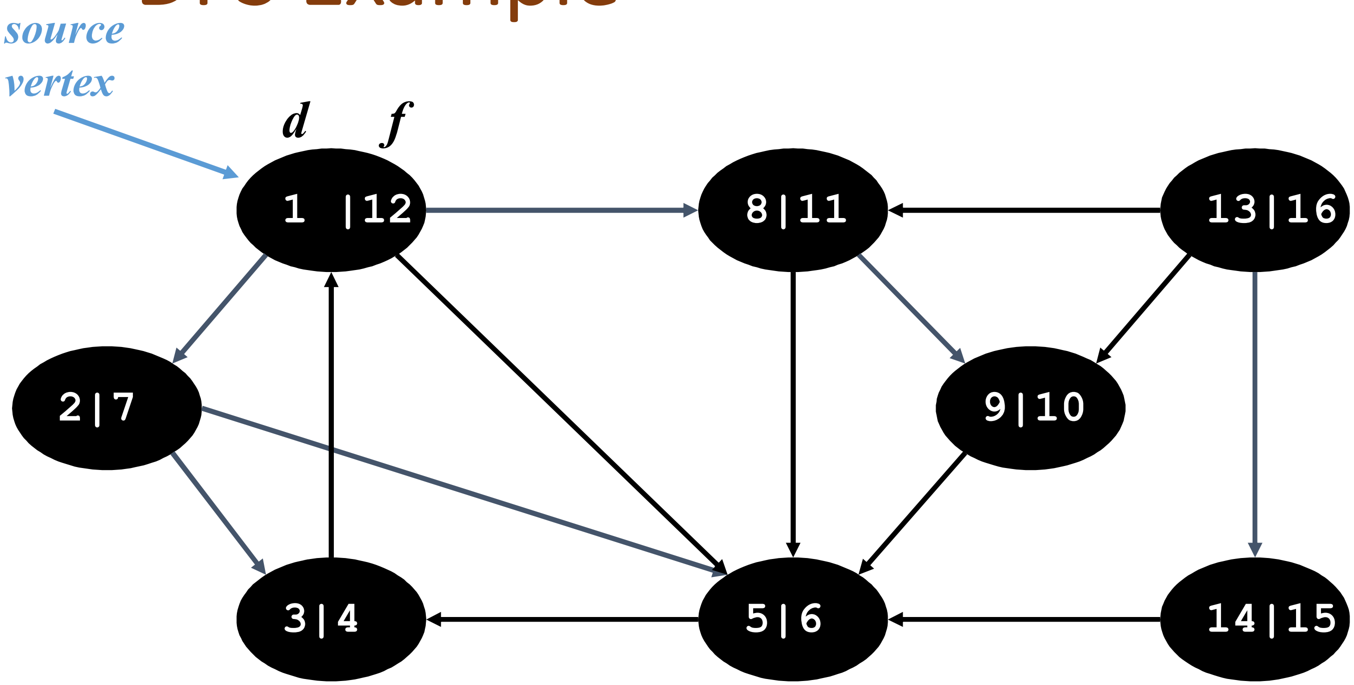
DFS Example

Nodes		Adjacency list		
s		*t*	*v*	*w*
t		*u*	*v*	
u		*s*		
v		*u*		
w		*v*	*x*	
x		*v*		
y		*w*	*x*	*z*
z		*v*		





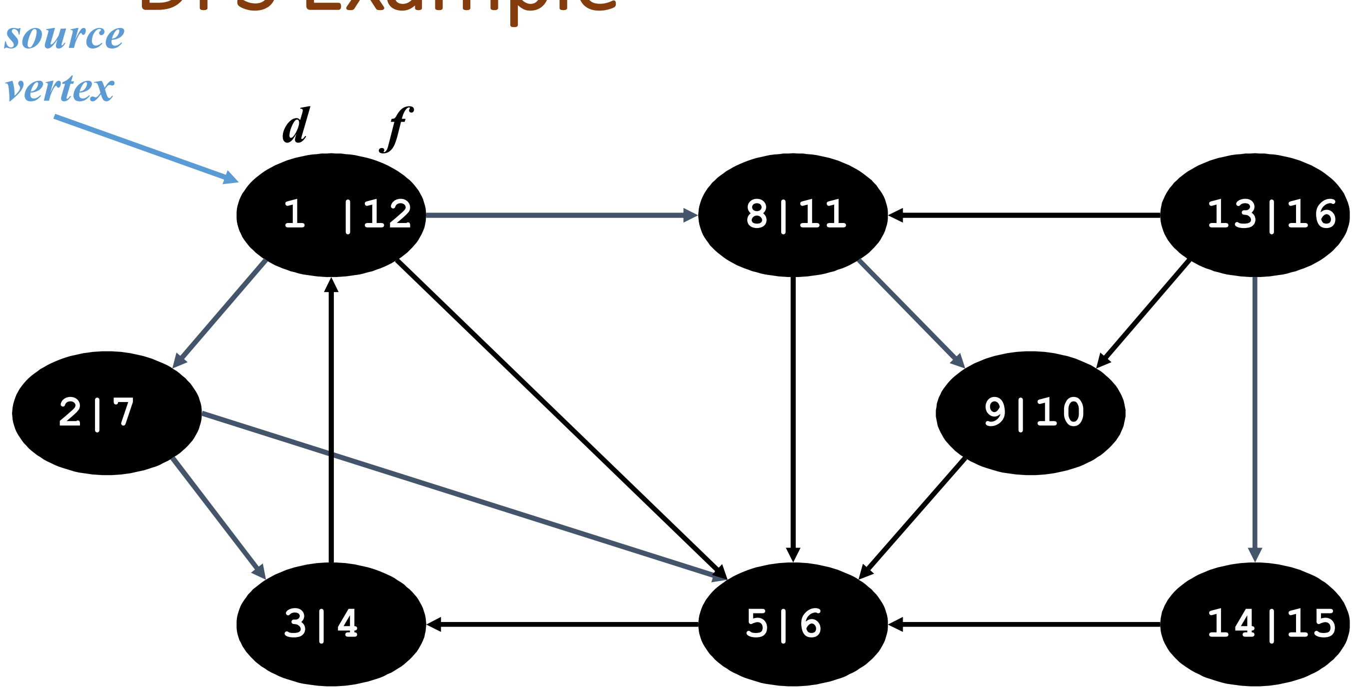
DFS Example



Interesting Facts

- $u.d$ records when vertex u is discovered
- $u.f$ *records* when the processing of vertex u is finished.
- These timestamps are integers between 1 and $2 \times |V|$.
  - Since there is one discovery event and one finishing event for each of the $|V|$ vertices

DFS Example



For every vertex u, we have: $u.d < u.f$ ---(22.2)

Interesting Facts

- $u.d$ records when vertex u is discovered
- $u.f$ *records* when the processing of vertex u is finished.
- These timestamps are integers between 1 and $2 \times |V|$.
  - Since there is one discovery event and one finishing event for each of the $|V|$ vertices

DFS: Properties

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

- $u = v.\pi$ if and only if DFS-VISIT (G, v) is called while searching u's adjacency list and v is white
- v is a descendent of u iff v is discovered WHITE while u is still grey

DFS: Properties

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

- $u = v.\pi$ if and only if DFS-VISIT (G, v) is called while searching u's adjacency list and v is white
- v is a descendent of u iff v is discovered WHITE while u is still grey