Algoritmos em Grafos

Grafos

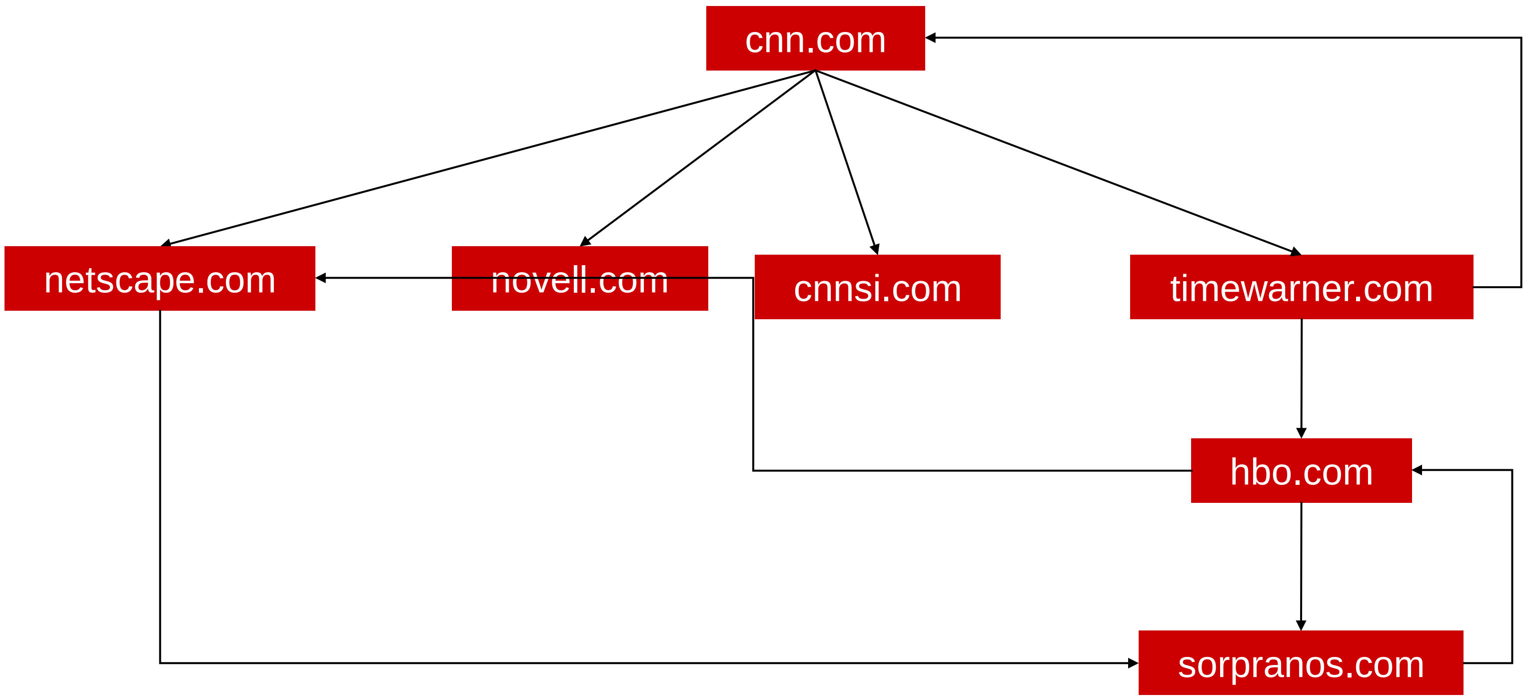
- Grafo $G = (V, E)$
  - V = conjunto de vértices
  - E = conjunto de arestas = subconjunto de $V \times V$
  - Então $|E| = O(|V|^2)$

Que pode ser modelado com grafos?

Grafo	Vértices	Arestas
transporte	interseção de ruas	ruas
comunicação	computadores	cabos
World Wide Web	páginas web	hyperlinks
social	pessoas	relações
rede alimentar	espécies	predador-presa
software	funções	chamadas a funções
escalonamento	tarefas	restrições de precedência
circuitos	portas	cabos
sistemas	estados	transições

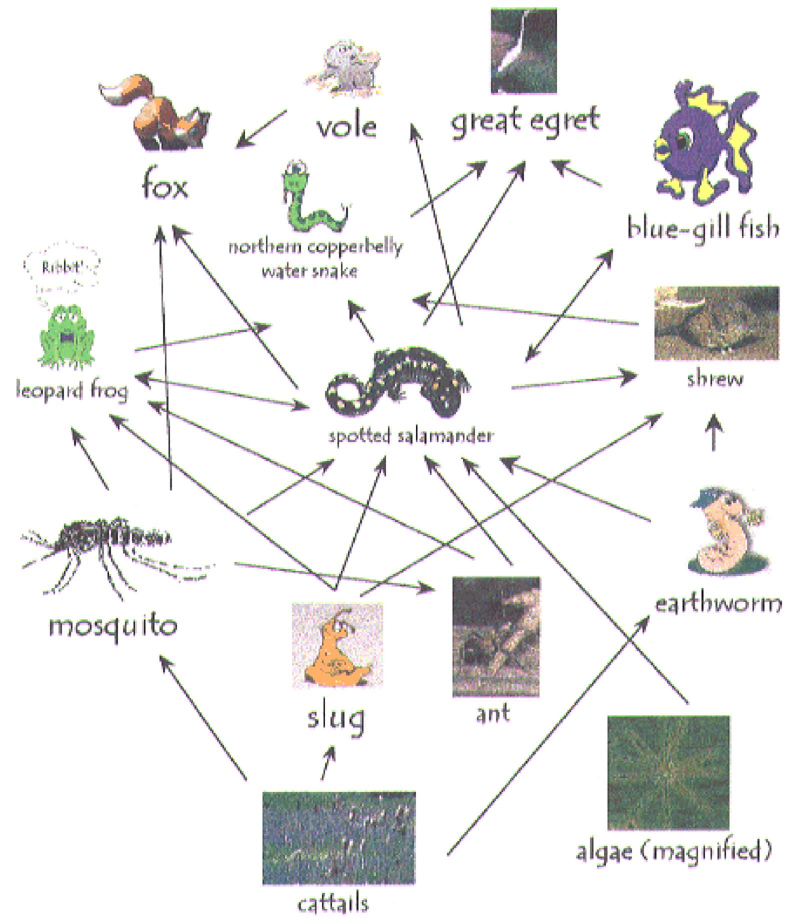
World Wide Web

- GrafoWeb.
  - Vértice: página web.
  - Aresta: hyperlink de uma página a outra.



Rede alimentar

- Grafo de rede alimentar.
  - Vértices = espécies.
  - Arestas = de presa a predador.



Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

Tipos de grafos

- Tipos de grafos:
  - Um grafo conexo tem um caminho de qualquer vértice a qualquer outro.
  - Num grafo não dirigido (não orientado):
    - aresta (u,v) = aresta (v,u)
  - Num grafo dirigido:
    - arco (u,v) vai do vértice u ao vértice v.

Tipos de grafos

- Mais tipos de grafos:
  - Um grafo ponderado associa pesos a suas arestas e/ou vértices
    - Ex, mapa: uma distância associada as arestas
  - Um multigrafo permite multiples arestas entre o mesmo par de vértices.
    - Ex, o grafo de um programa (uma função pode ser chamada várias vezes a partir de uma mesma função)

Grafos

- Tipicamente expressamos a complexidade algorítmica em termos de $|E|$ e $|V|$
  - Se $|E| \approx |V|^2$ o grafo é denso
  - Se $|E| \approx |V|$ o grafo é esparso
  - Se o grafo é conexo $|E| >= |V|-1$
- Algoritmos e estrutura de dados interessantes podem variar dependendo da densidade de grafos

Grafos

- Tipicamente expressamos a complexidade algorítmica em termos de $|E|$ e $|V|$
  – Se $|E| \approx |V|^2$ o grafo é denso
  – Se $|E| \approx |V|$ o grafo é esparso
  – Se o grafo é conexo $|E| >= |V|-1$
- Algoritmos e estrutura de dados interessantes podem variar dependendo da densidade de grafos

Comparando complexidades

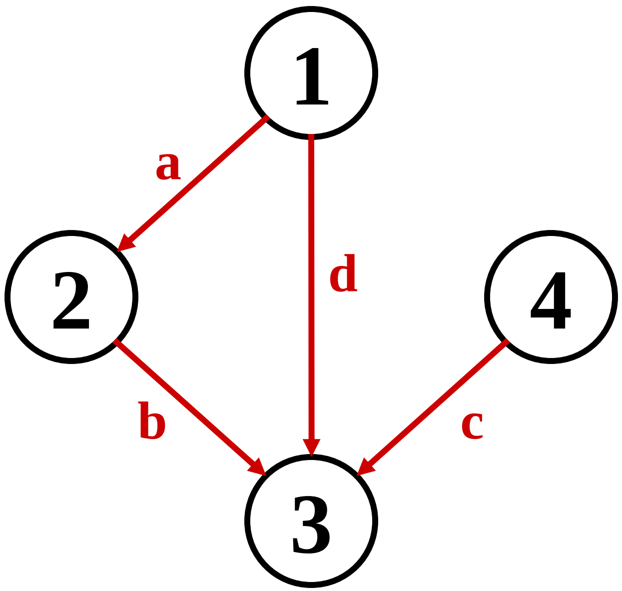
- Compare:
  - $O(EV)$ vs $O(V^3)$
  - $O(V\lg(V))$ vs $O(E)$
  - $O(E\lg(V))$ vs $O(E\lg(E))$
  - $O(EV^2)$ vs $O(V^3)$
  - $O(E+V)$ vs $O(V)$
  - $O(E+V)$ vs $O(E)$

Representação de grafos

- Assumimos $V = \{1, 2, \ldots, n\}$
- Uma matriz de adjacência representa o grafo como uma matriz de n x n A:
  - $A[i, j] = 1$ se a aresta $(i, j) \in E$ (ou peso)
    $= 0$ se a aresta $(i, j) \notin E$

Grafos: Matriz de Adjacência

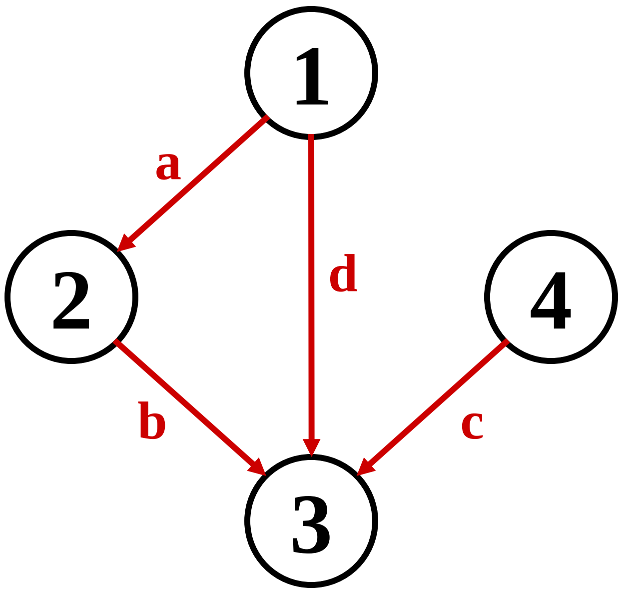
- Exemplo:



A	1	2	3	4
1				
2				
3			??	
4				

Grafos: Matriz de Adjacência

- Examplo:



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

Grafos: Matriz de Adjacência

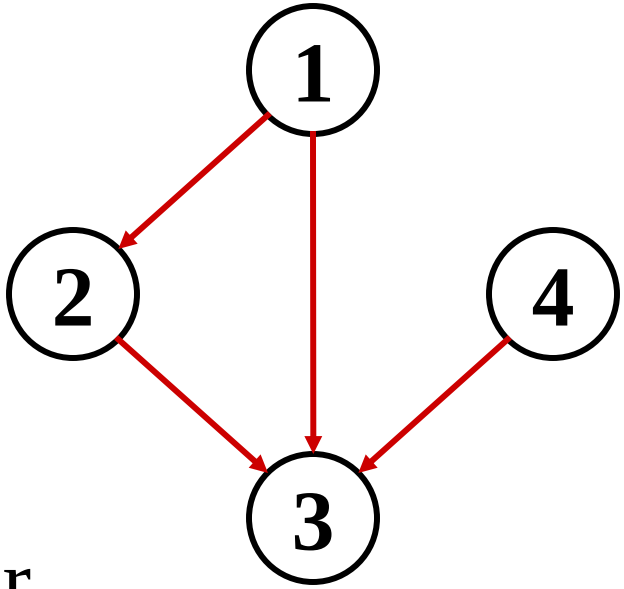
- Quanto espaço precisa uma matriz de adjacência?
- A: $O(V^2)$

Grafos: Matriz de Adjacência

- Ocupa o mesmo espaço para grafos densos ou esparços.
  - Usalmente demasiado espaço para grafos esparços
  - Mas pode ser eficiênte para grafos densos.
- A maioria dos grafos grandes são esparços
  - Por isto a lista de djacência é frequentemente uma representação mais apropriada

Grafos: Lista de Adjacência

- Lista de adjacência: para cada vértice $v \in V$, armazena a lista de vértices adjacentes a v
- Exemplo:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}



- Alternativa: pode armazenar a lista de vértices que chegam a um vértice

Grafos: Lista de Adjacência

- Quanto espaço precisa?
  - O grau de um vértice v = # arestas incidentes
    - Grafos orientados tem grau de entrada e grau de saida
  - Para grafos orientados, # de items na listas de adjacência é

    $$\Sigma \text{ grau de saida}(v) = |E|$$

    $\Theta(V + E)$ espaço (Porque?)
  - Para grafos não orientados

    $$\Sigma \text{ degree}(v) = 2\ |E|$$

  $\Theta(V + E)$ espaço
- Então: listas de adjacências ocupam O(V+E)

Busca em grafos

- Dado: um grafo G = (V, E), orientado ou não.
- Objetivo: percorrer sistematicamente cada vértice e cada aresta.
- Construir uma árvore de busca
  - Escolher um vértice como raiz.
  - Escolher algumas arestas para formar a árvore.
  - Pode construir uma floresta se o grafo não for conectado.

Breadth-First Search

- "Percorre" um grafo, transformando-lo em uma árvore
  - Um vértice de cada vez
  - Expande a fronteira de vértices percorridos segundo a distância ao vértice de início.

Breadth-First Search

- Associa cores aos vértices para guiar o algoritmo
  - Vértices brancos não foram descobertos
    - Todos os vértices começam em branco
  - Vértices cinza foram descobertos mas não processados
    - Podem ter vértices brancos como adjacentes
  - Vértices pretos foram descobertos e processados
    - São adjacentes a vértices cinza e pretos
- Percorre os vértices segundo as listas de adjacências dos vértices cinzas

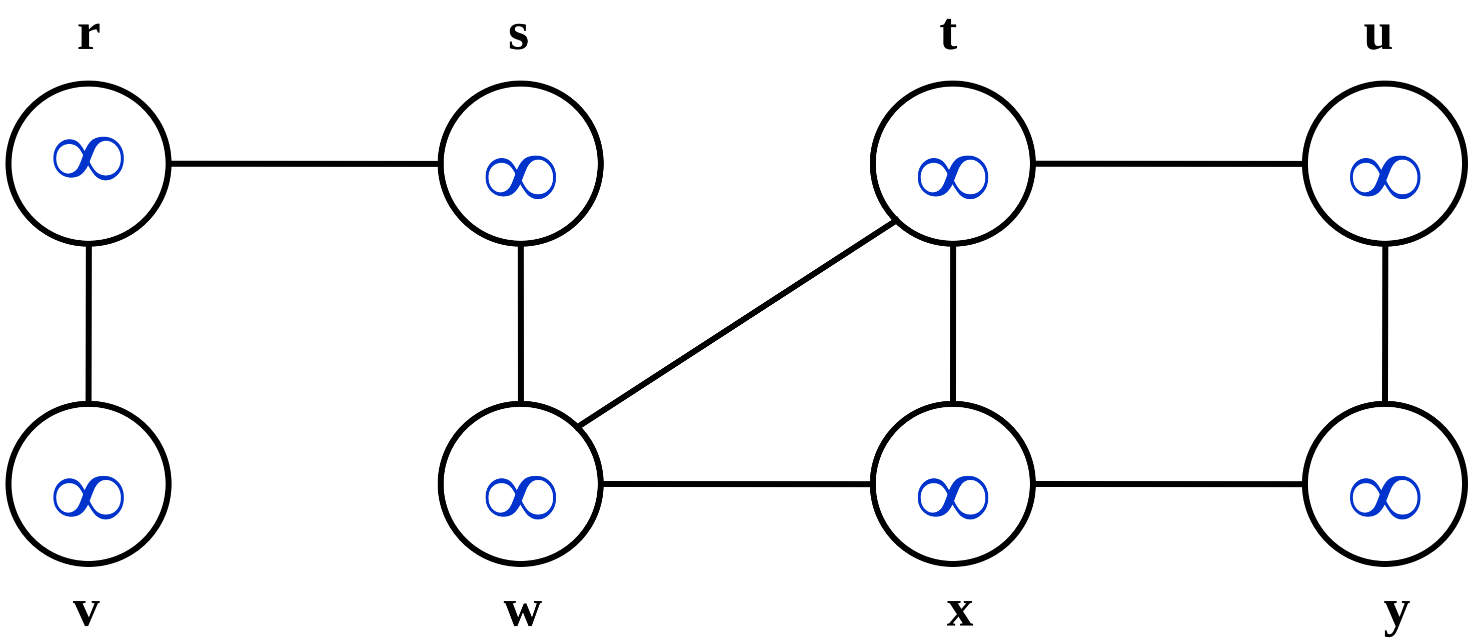
Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};     // Q is a queue; initialize to s
    while (Q not empty) {
        u = RemoveFirst(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

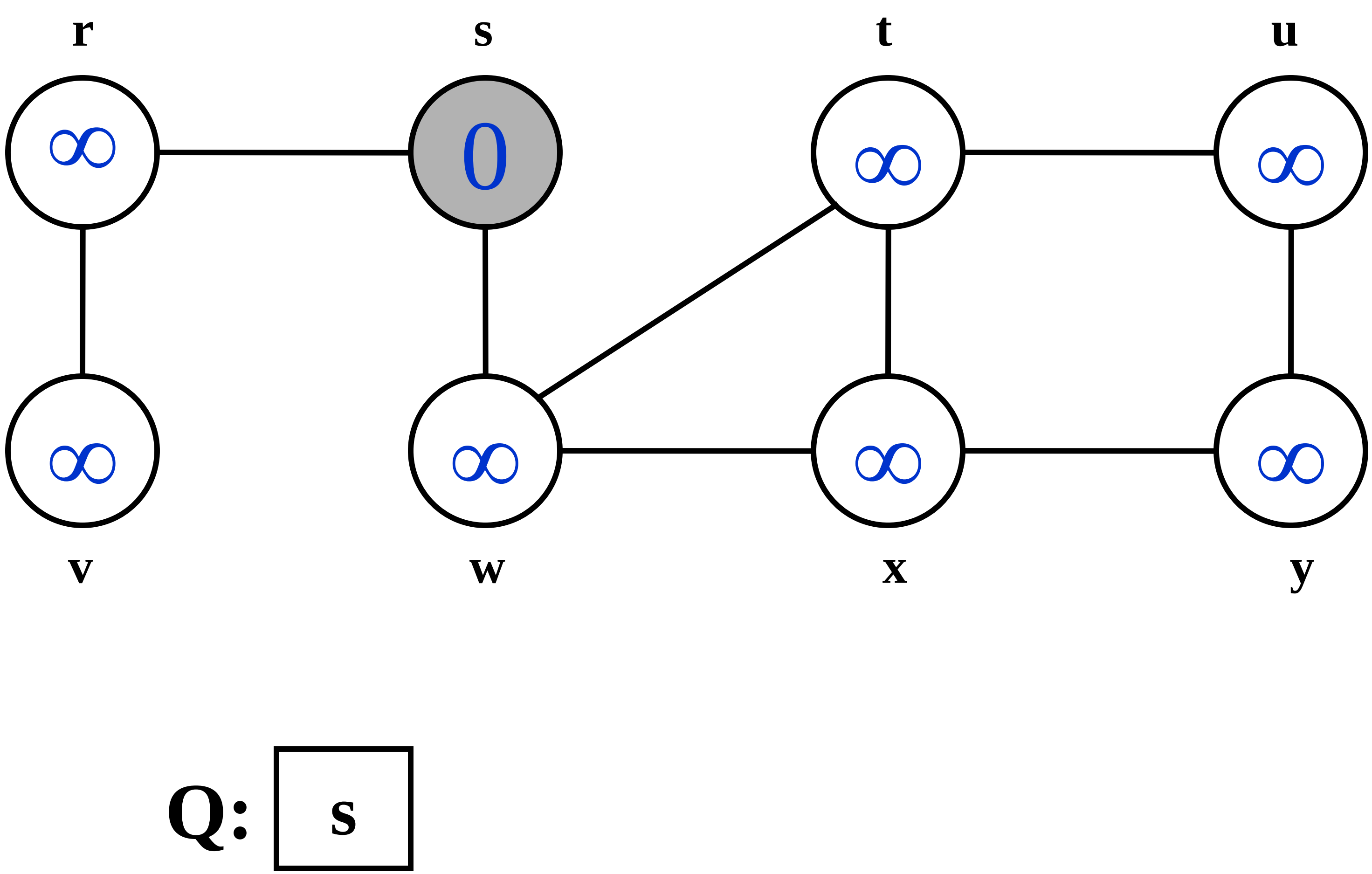
O que `v->d` representa?

O que `v->p` representa?

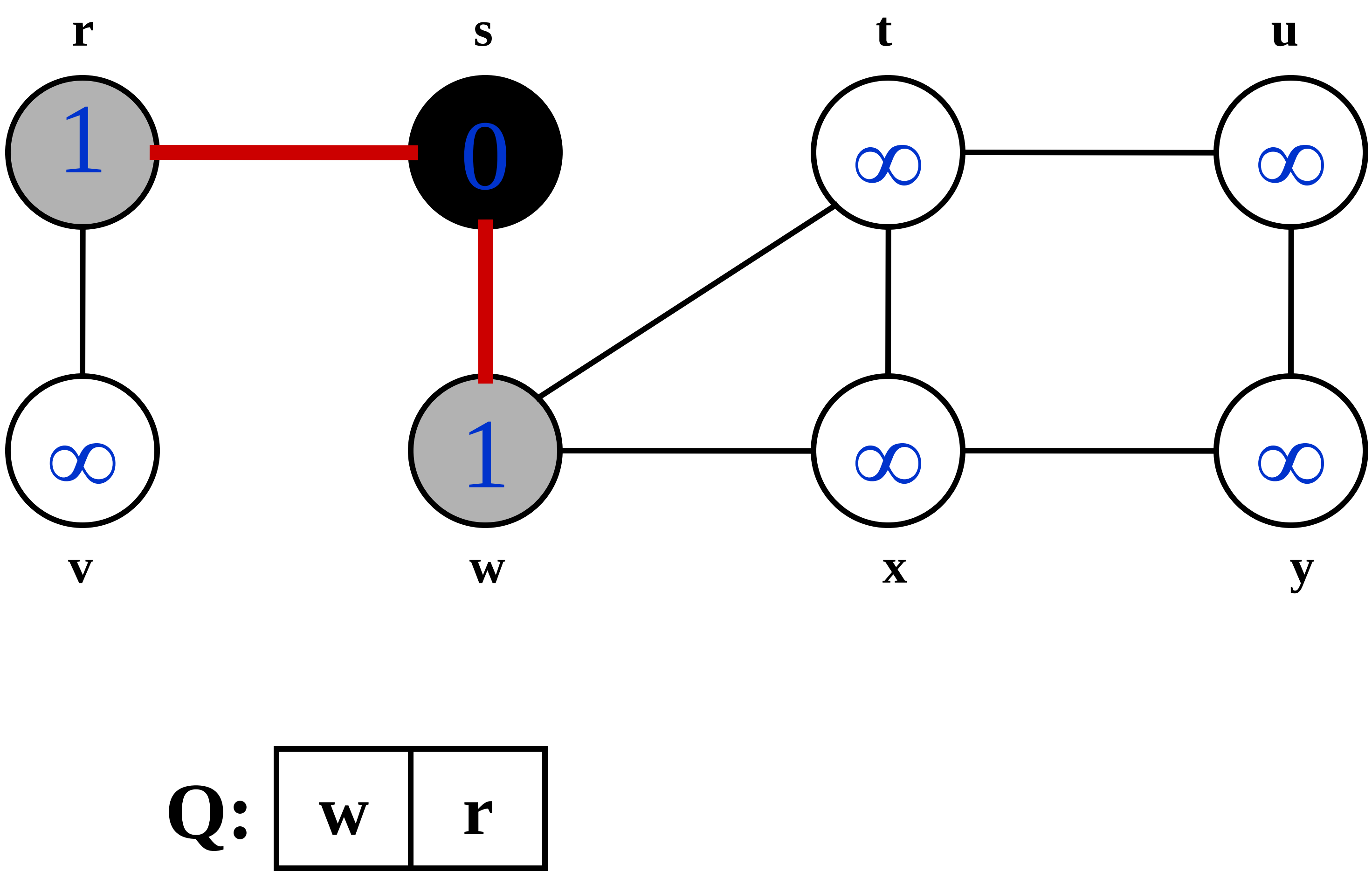
Breadth-First Search: Examplo



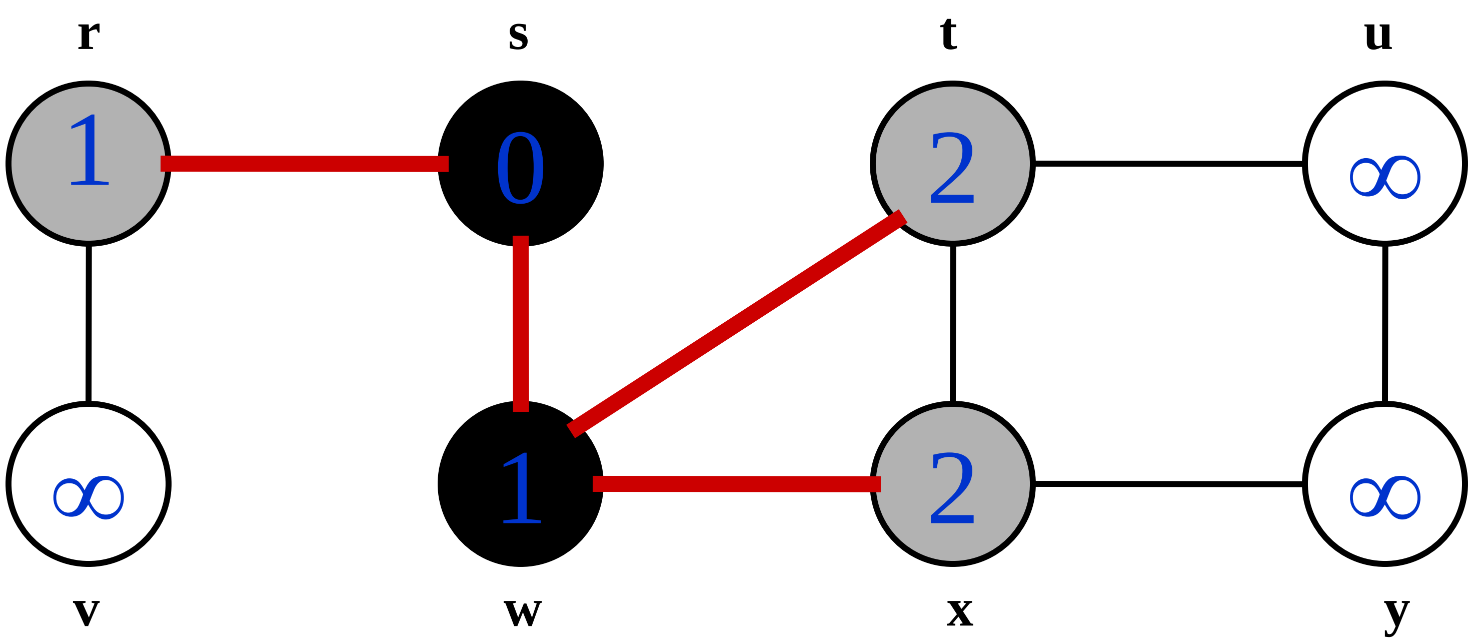
Breadth-First Search: Examplo



Breadth-First Search: Examplo



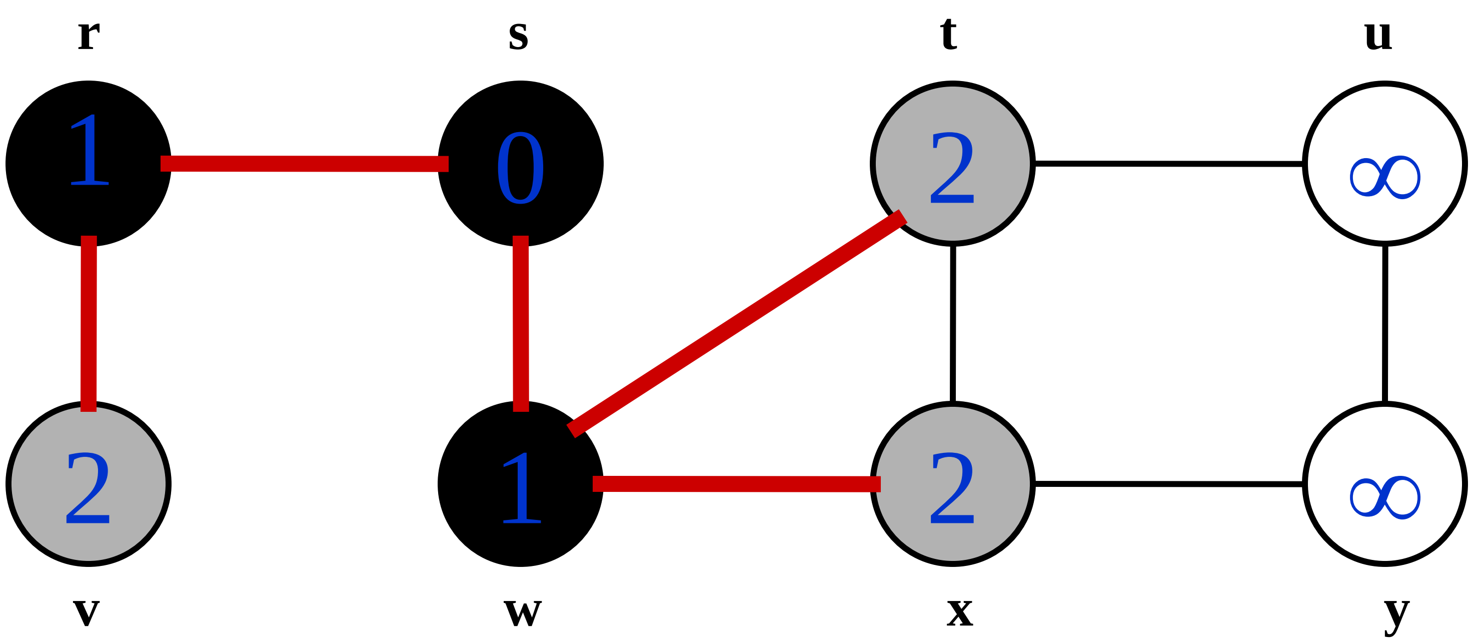
Breadth-First Search: Examplo



Q:

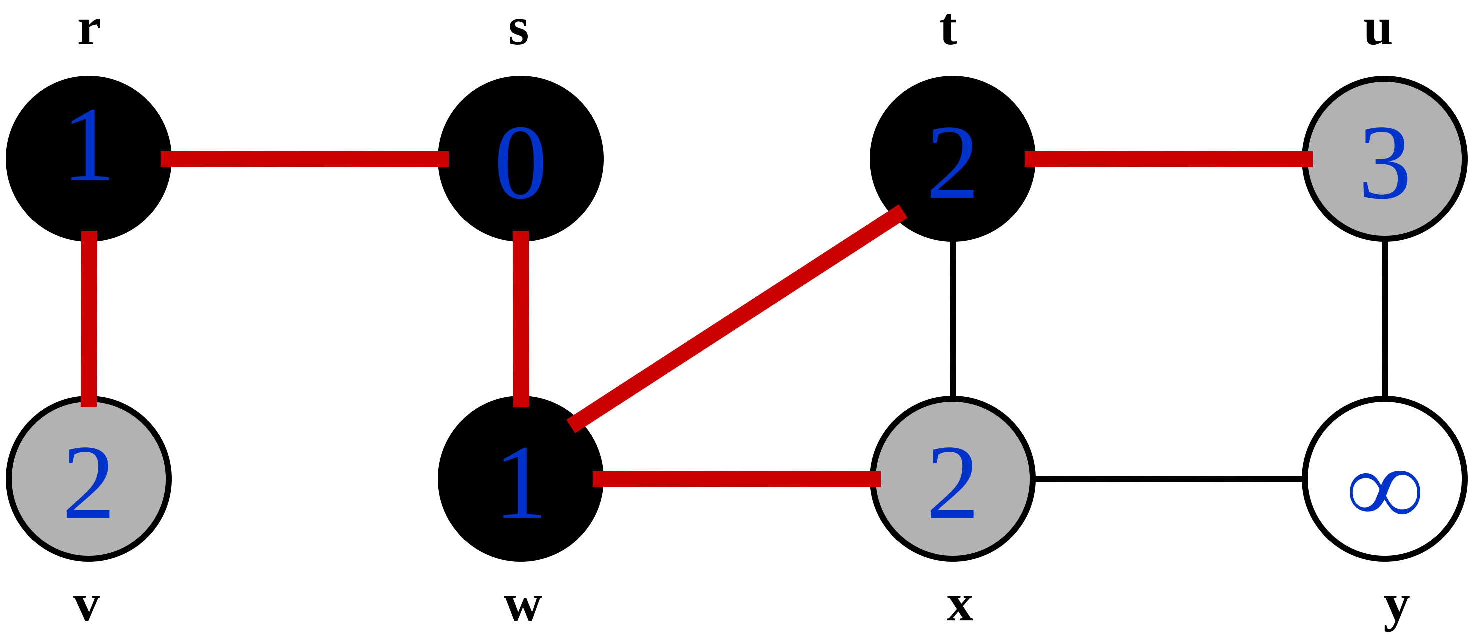
r	t	x

Breadth-First Search: Examplo



Q: | t | x | v |

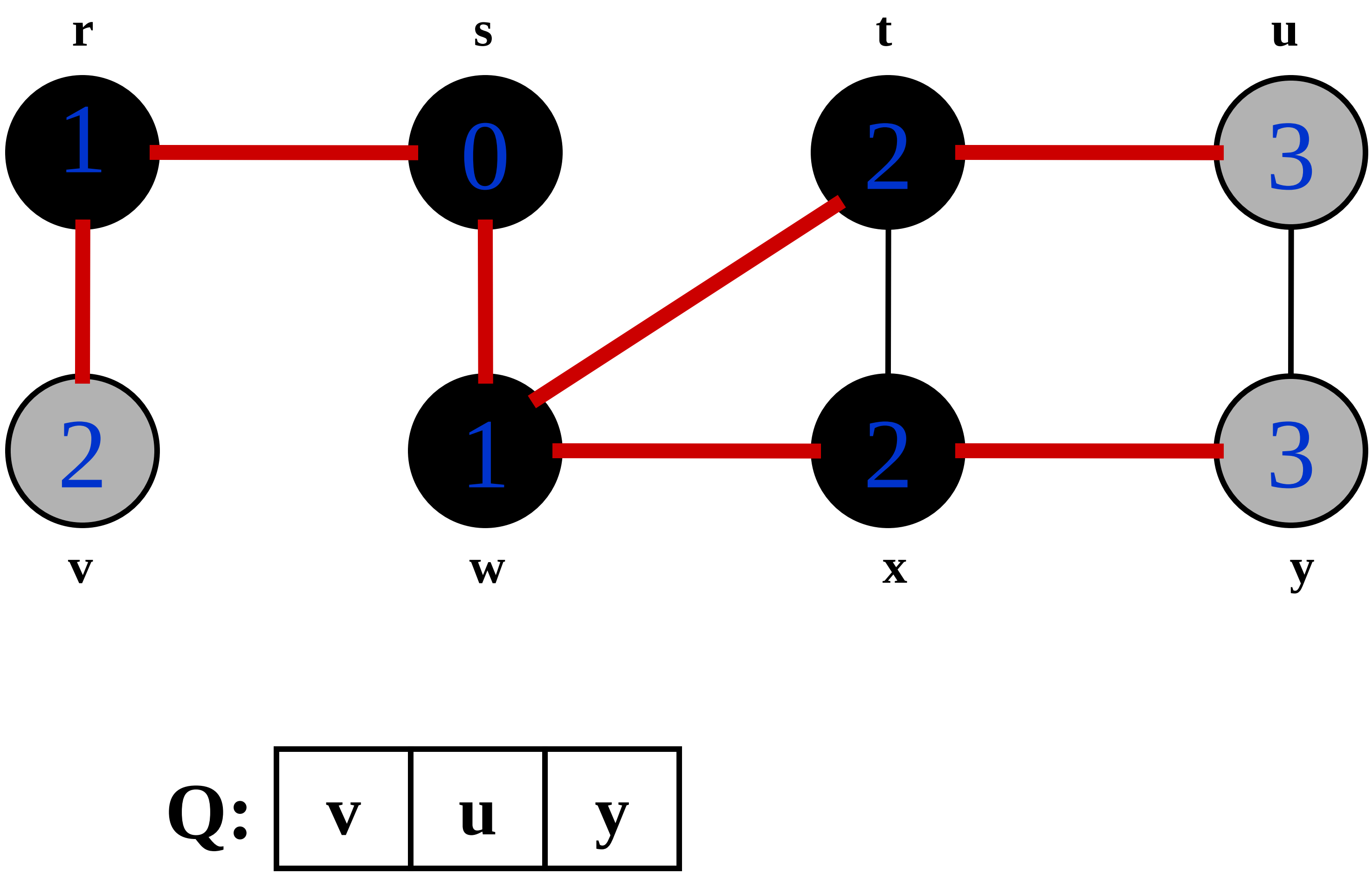
Breadth-First Search: Examplo



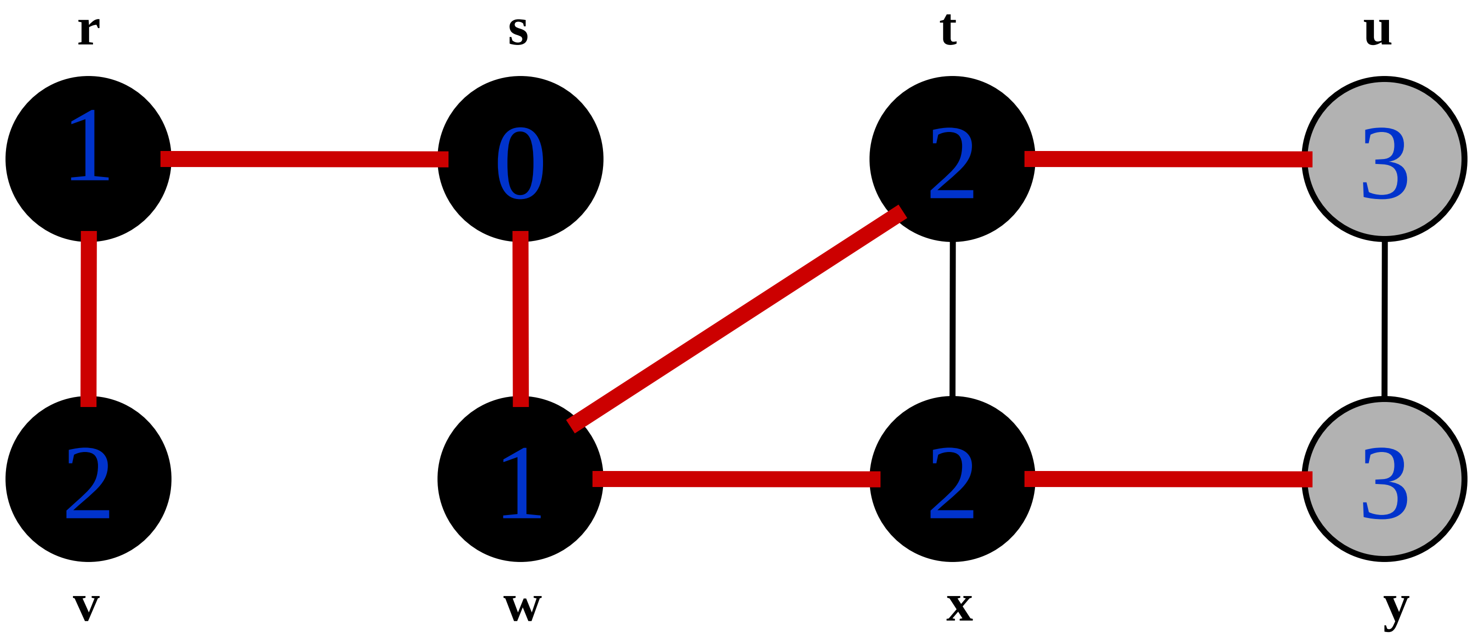
Q:

x	v	u

Breadth-First Search: Examplo

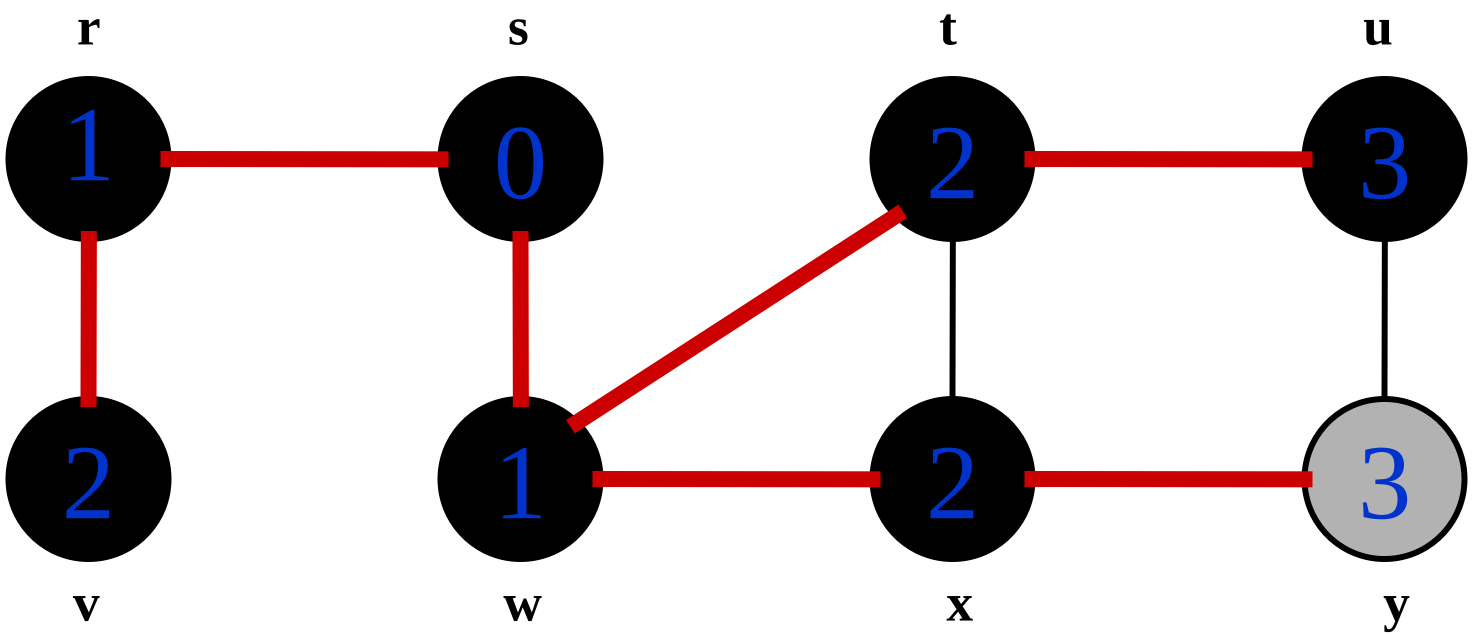


Breadth-First Search: Examplo



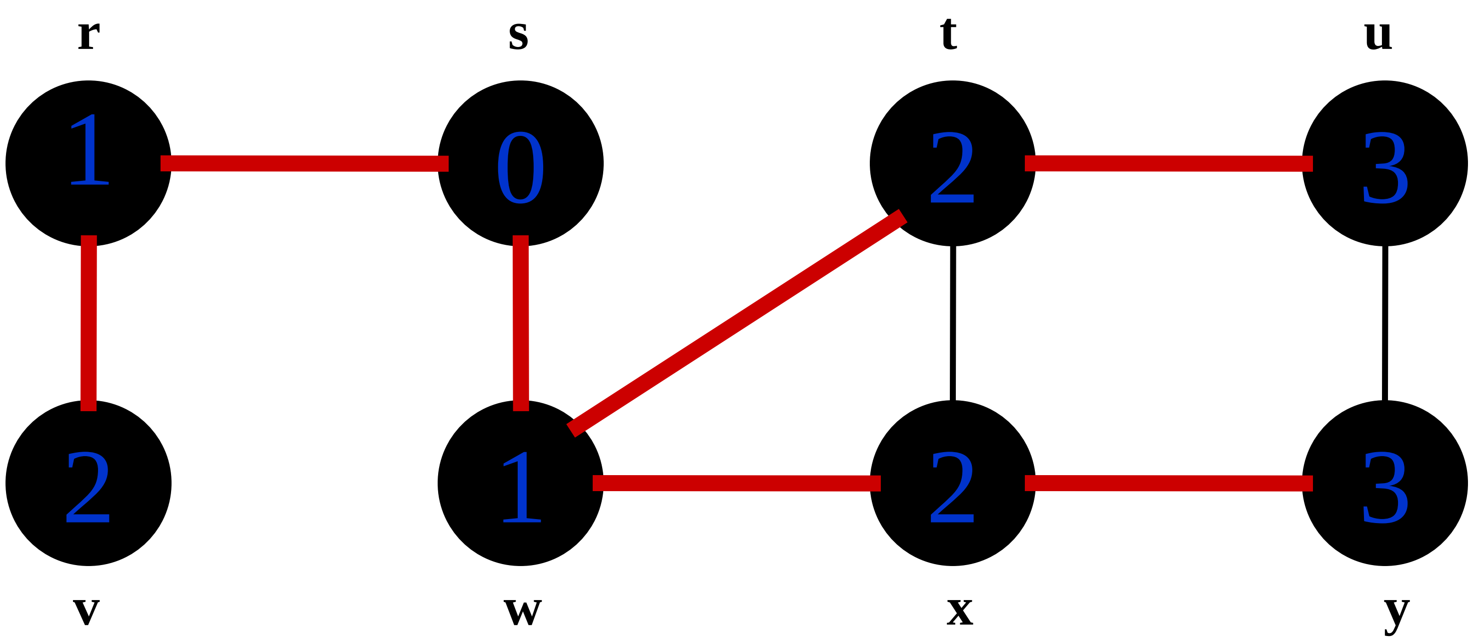
Q: | u | y |

Breadth-First Search: Examplo



Q: $\boxed{y}$

Breadth-First Search: Examplo



Q: Ø

BFS: O Código

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveFirst(Q);
        for each v ∈ u->adj {
            if (v->color == WHITE)
                v->color = GREY;
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
        u->color = BLACK;
    }
}
```

Percorre todos os vértices: O(V)

u = todos os vértices apenas uma vez. (Porque?)

Então v =qualquer vétice que apareça na lista de adjacência de outro vértice

Qual será o tempo de execução?

Tempo de execução: O(V+E)

Breadth-First Search: Propriedades

- BFS computa os caminhos mínimos da fonte a todos os outros vértices do grafo.
  - Distância do caminho mínimo $\delta(s,v)$ = mínimo número de arestas de s a v, ou ∞ se v não e alcançável desde s
- BFS constói uma árvere de busca em largura, na qual os caminhos à raiz representam caminhos mínimos em G
  - Então BFS pode ser usado para computar caminhos mínimos em grafos não ponderados

Depth-First Search

- Depth-first search is another strategy for exploring a graph
  - Explore "deeper" in the graph whenever possible
- Builds a tree over the graph
  - Pick a source vertex to be the root
  - Find ("discover") its children, then their children, etc.

Depth-First Search

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

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What does u->d represent?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What does `u->f` represent?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Will all vertices eventually be colored black?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

What will be the running time?

Depth-First Search: The Code

```
DFS(G)
{
    for each vertex u ∈ G->V
    {
        u->color = WHITE;
    }
    time = 0;
    for each vertex u ∈ G->V
    {
        if (u->color == WHITE)
            DFS_Visit(u);
    }
}
```

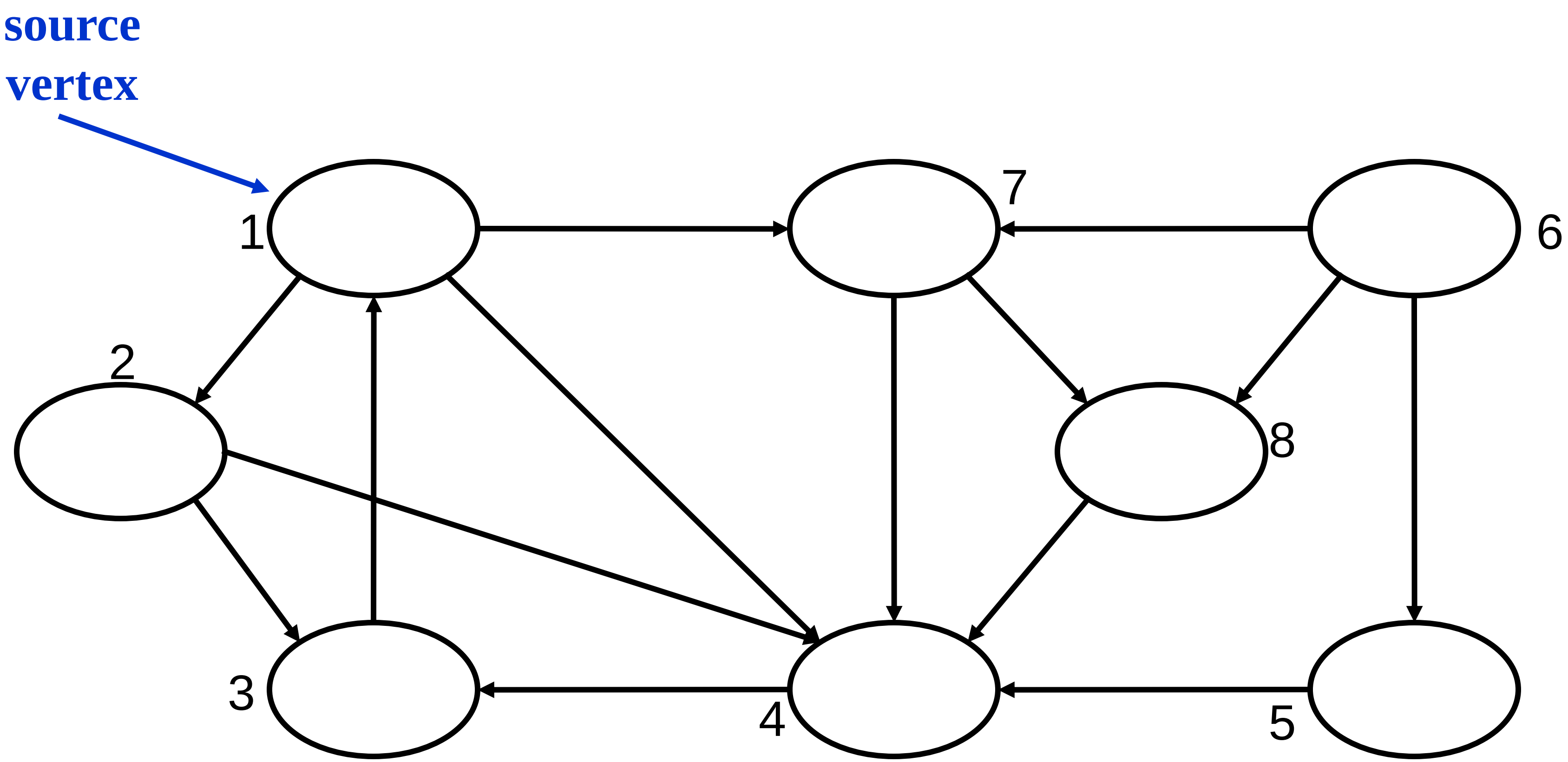
```
DFS_Visit(u)
{
    u->color = GREY;
    time = time+1;
    u->d = time;
    for each v ∈ u->Adj[]
    {
        if (v->color == WHITE)
            DFS_Visit(v);
    }
    u->color = BLACK;
    time = time+1;
    u->f = time;
}
```

Running time: $\Theta(V + E)$ – aggregate analysis

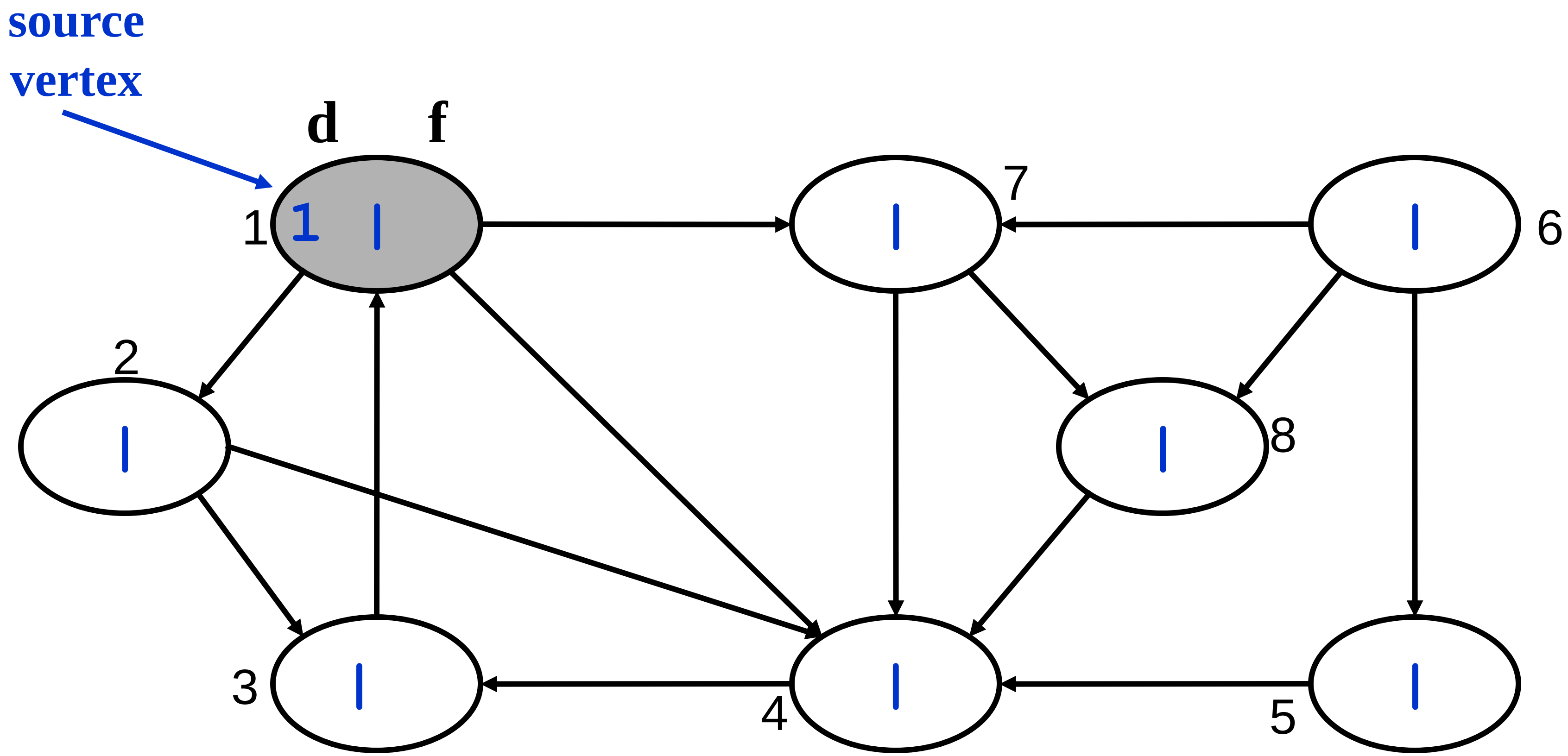
Depth-First Sort Analysis

- This running time argument is an informal example of amortized analysis
  - "Charge" the exploration of edge to the edge:
    - Each loop in DFS_Visit can be attributed to an edge in the graph
    - Runs once/edge if directed graph, twice if undirected
    - Thus loop will run in O(E) time, algorithm O(V+E)
      - Considered linear for graph, b/c adj list requires O(V+E) storage
  - Important to be comfortable with this kind of reasoning and analysis

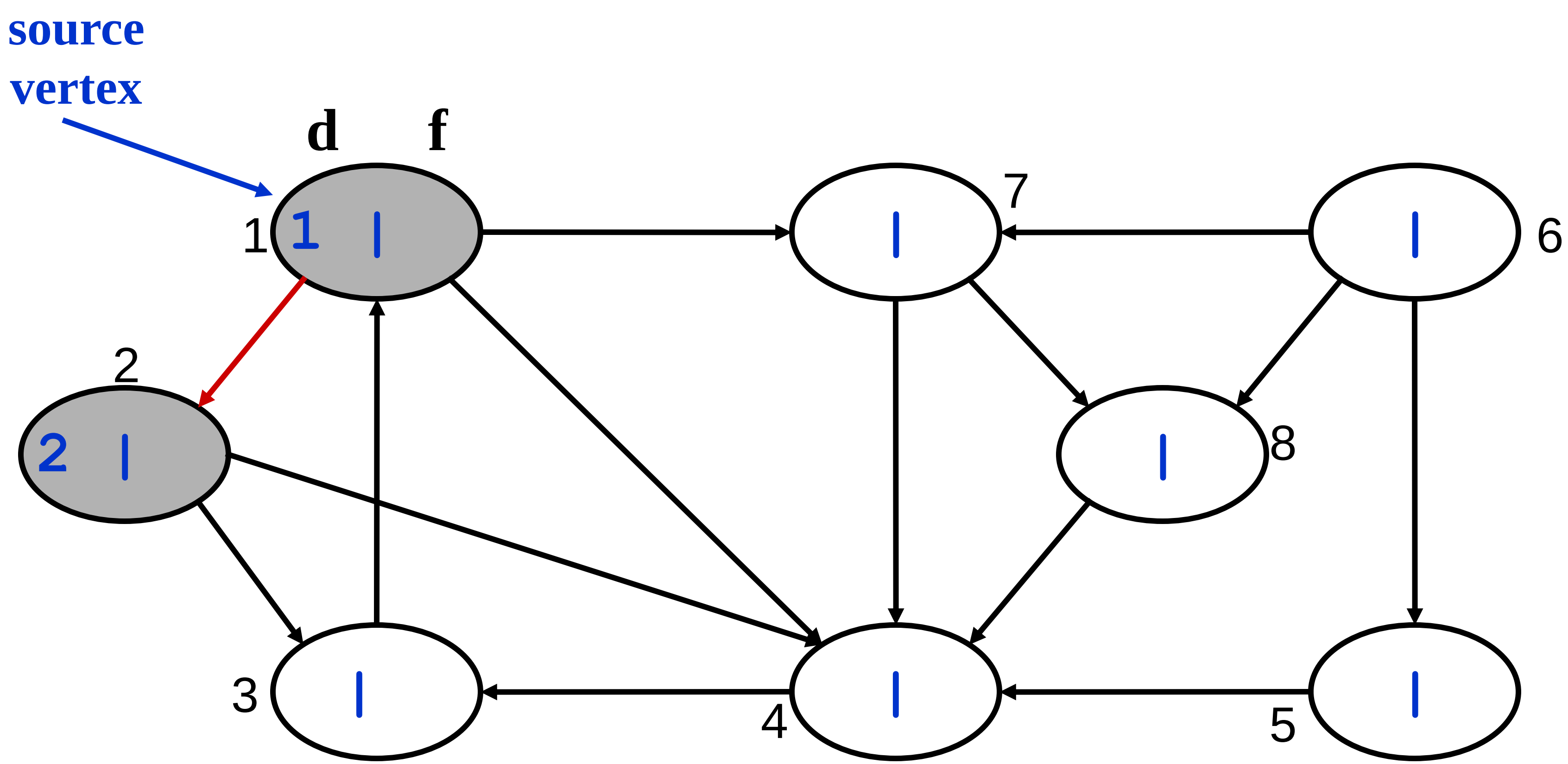
DFS Example



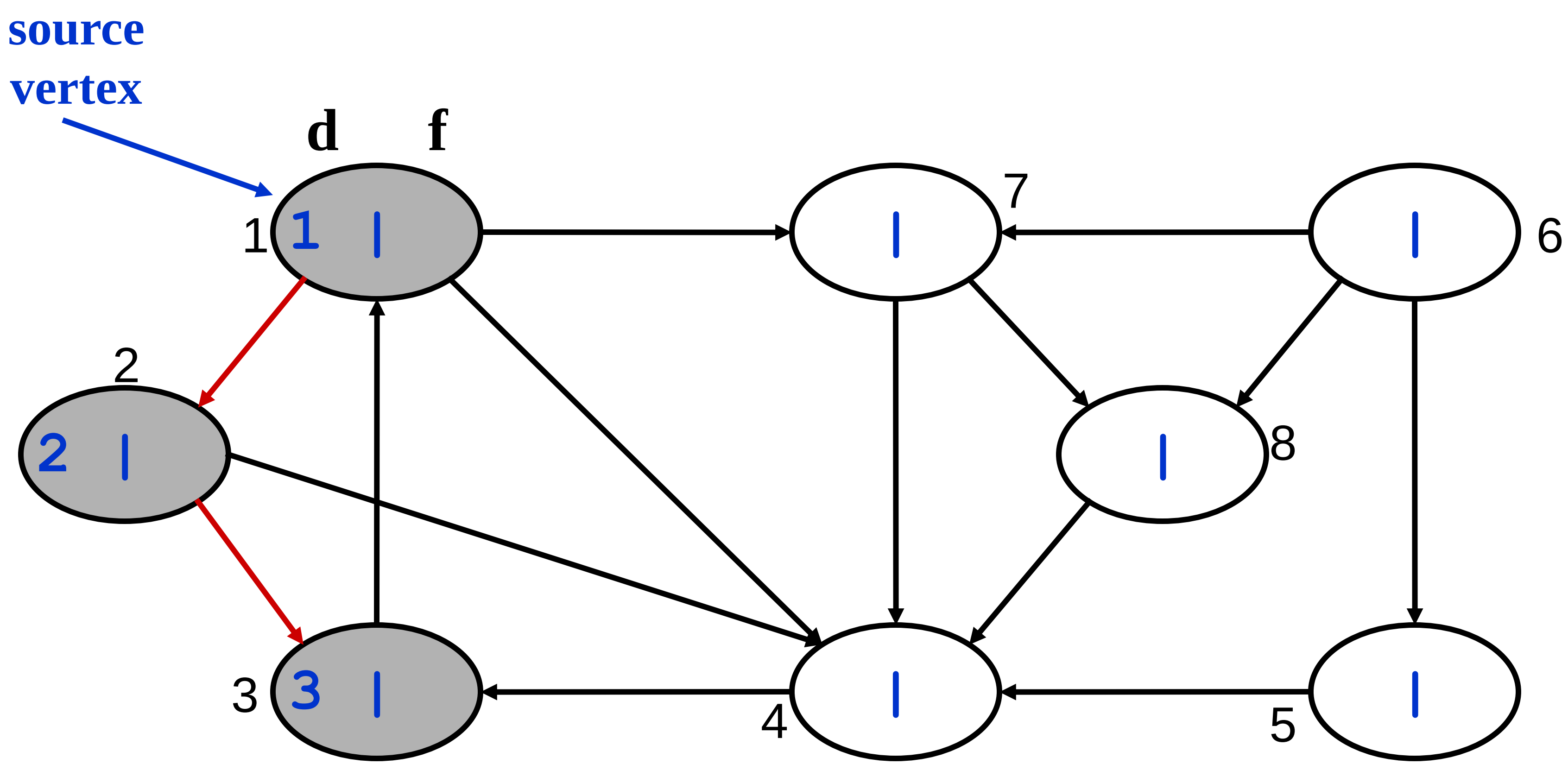
DFS Example



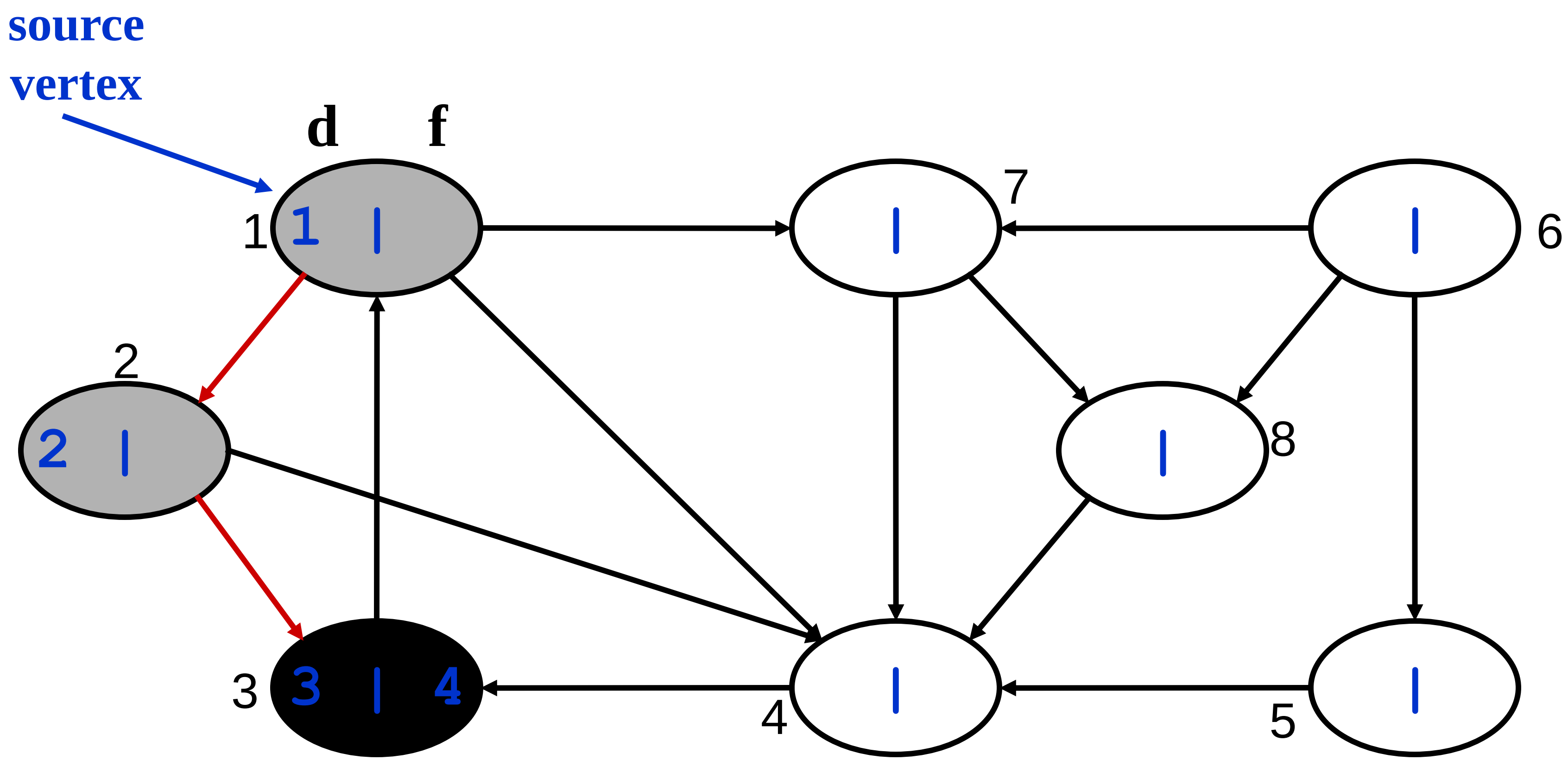
DFS Example



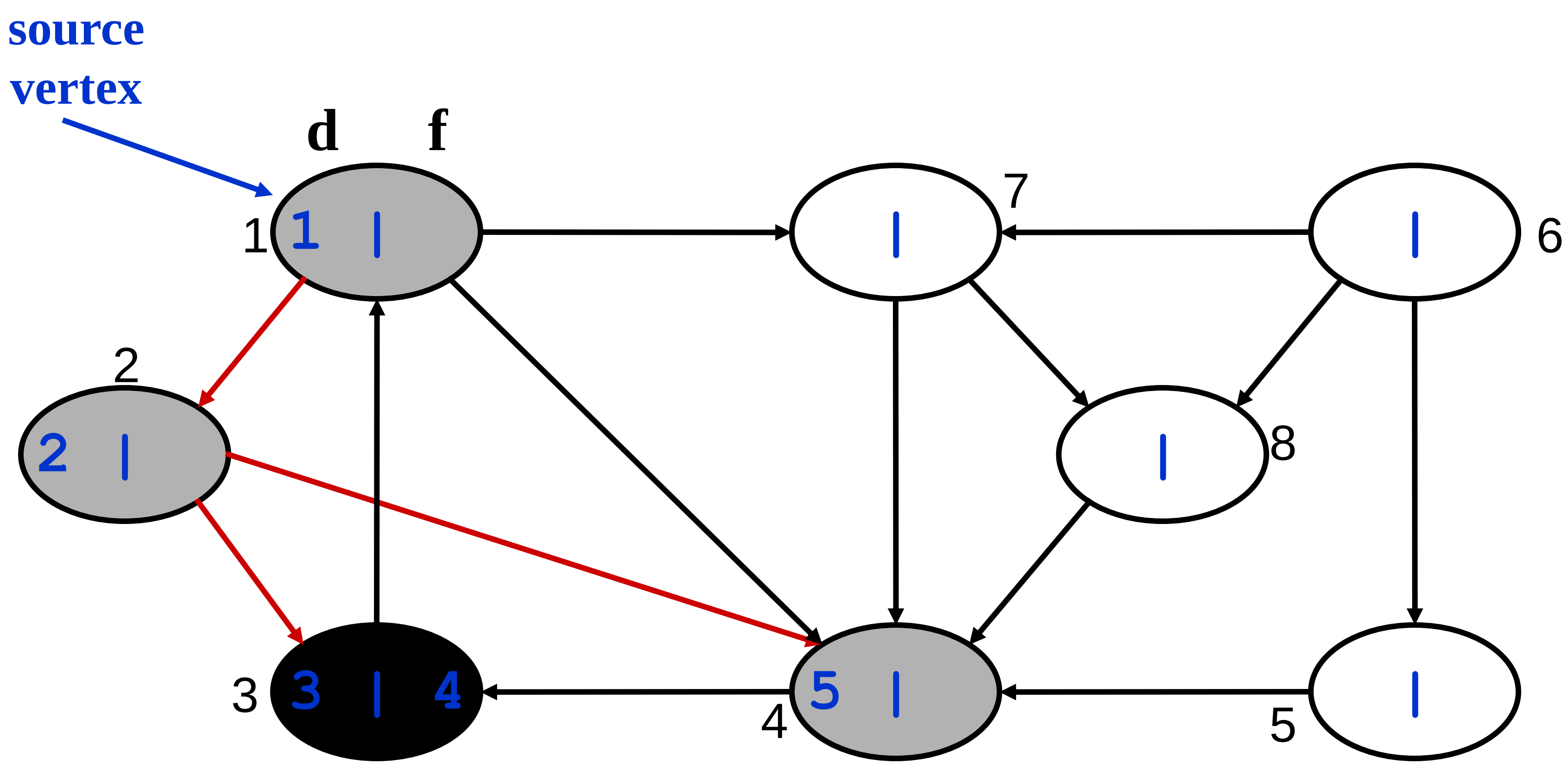
DFS Example



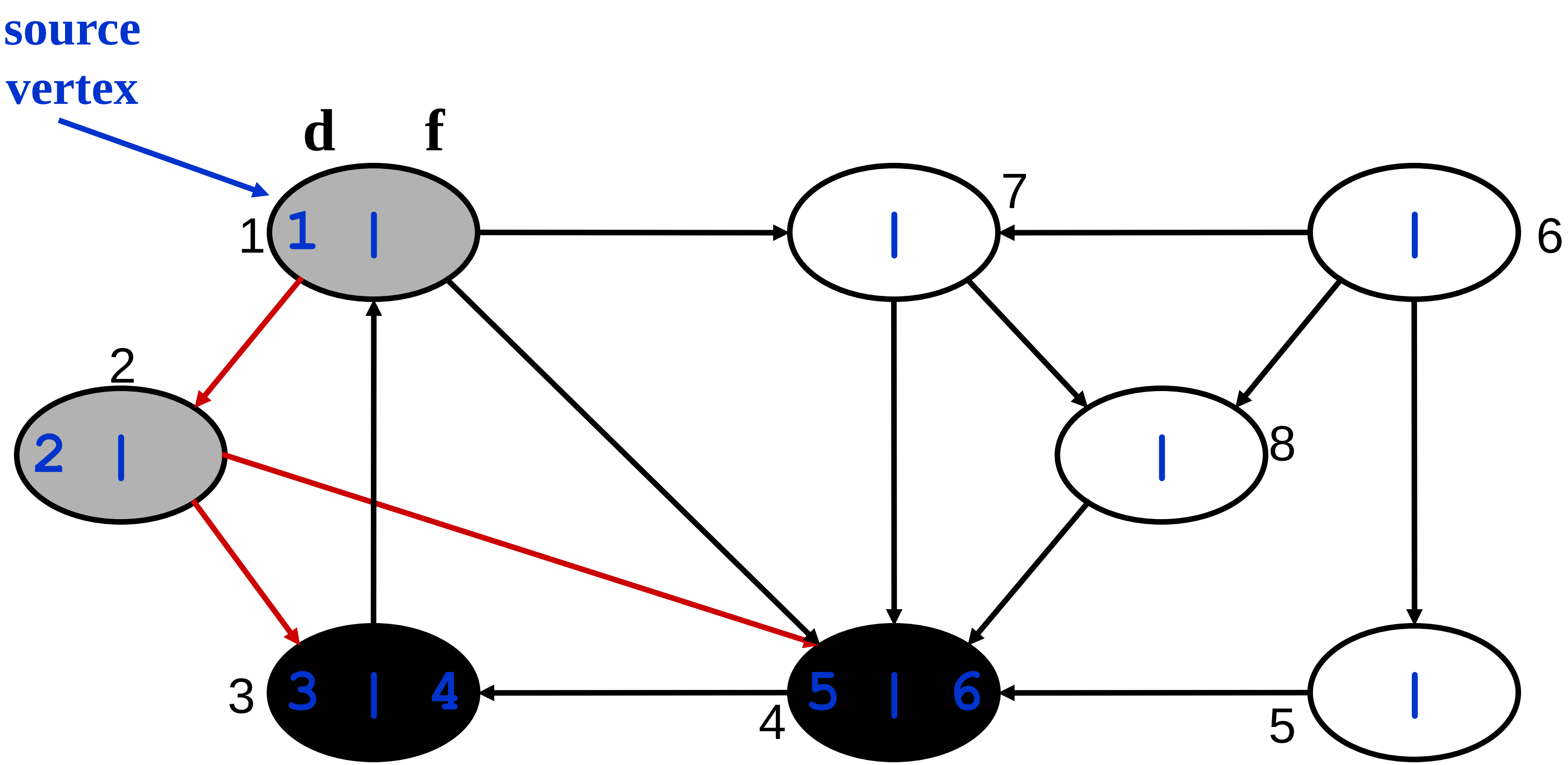
DFS Example



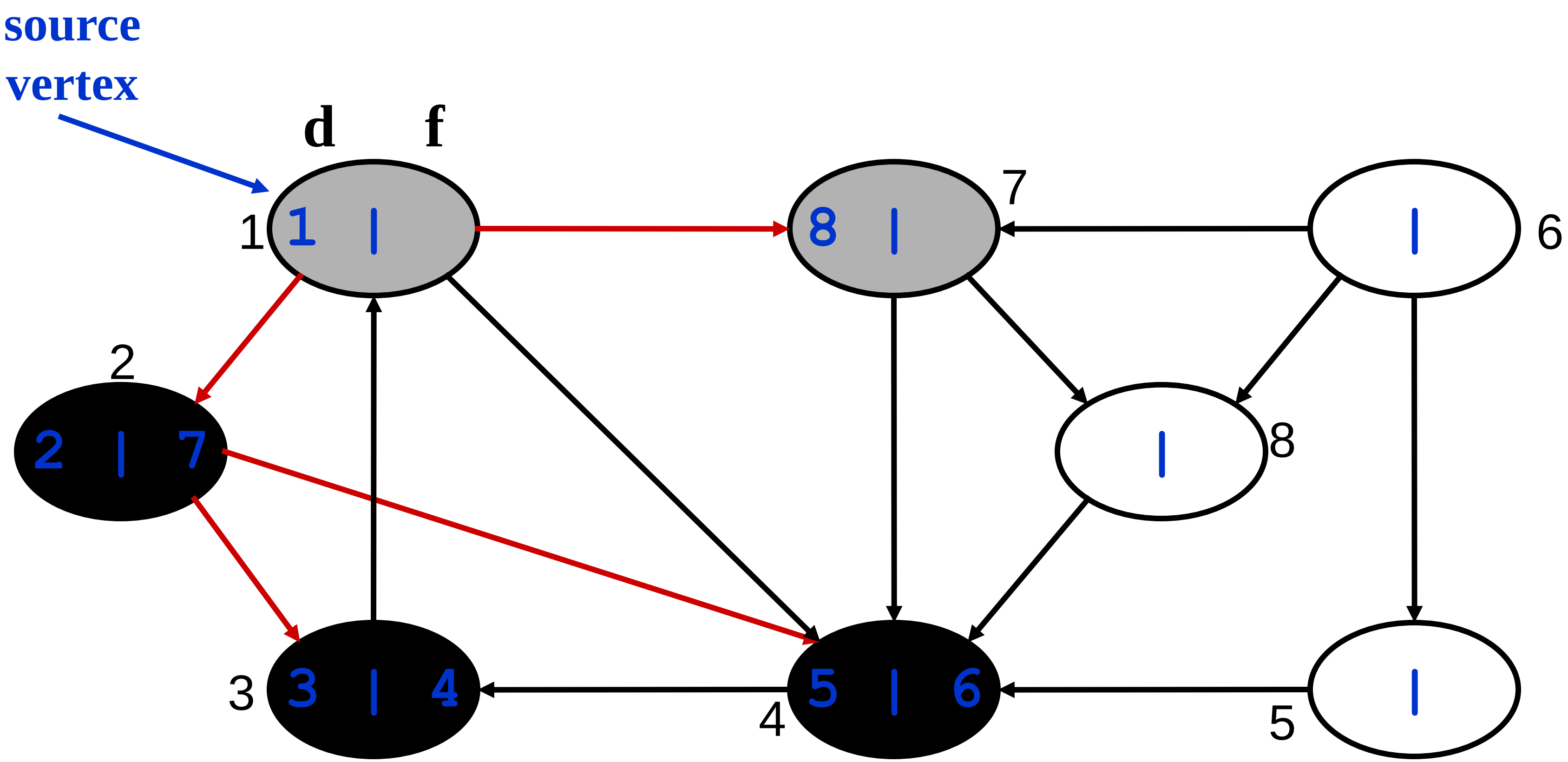
DFS Example



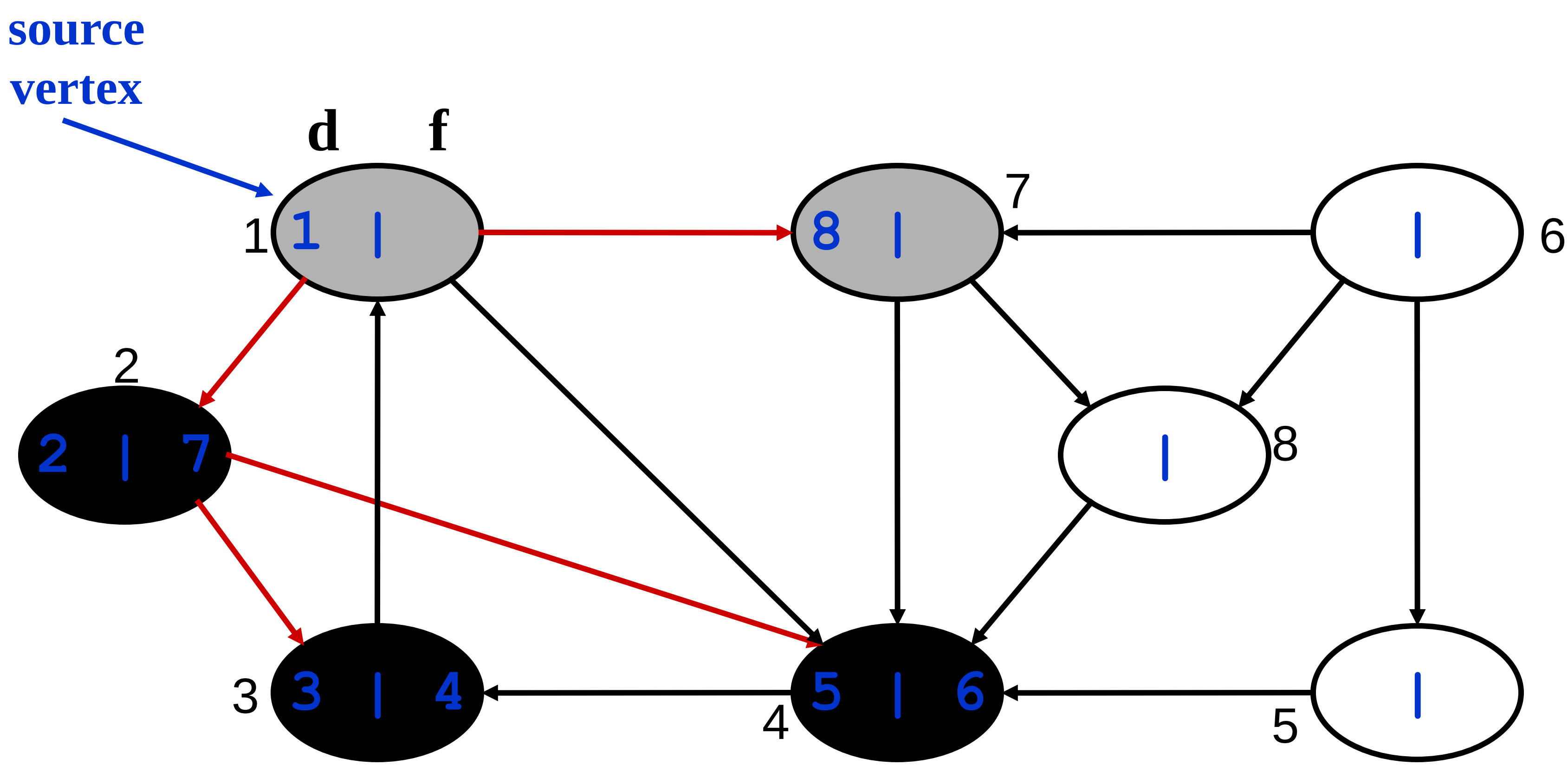
DFS Example



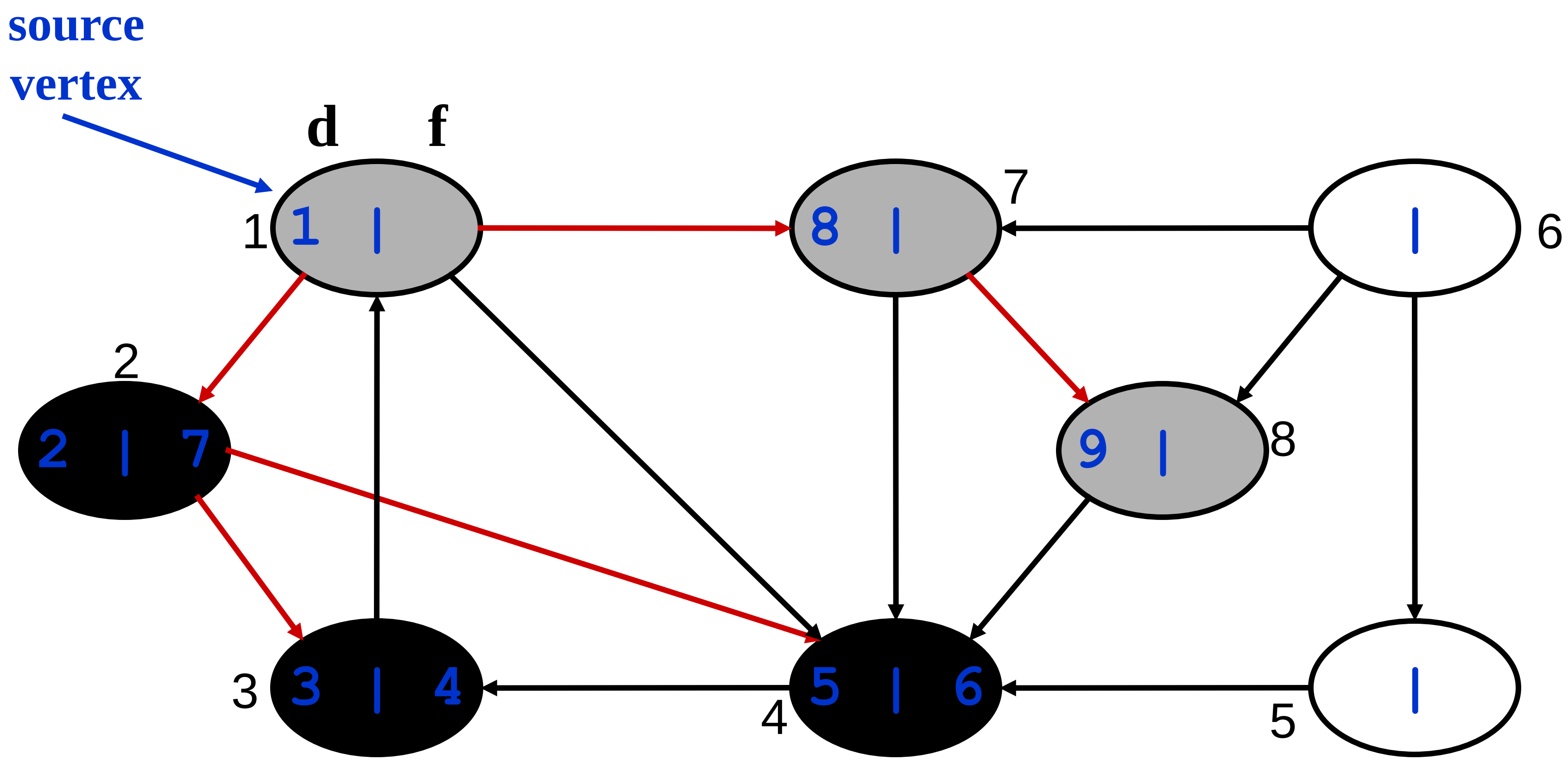
DFS Example



DFS Example

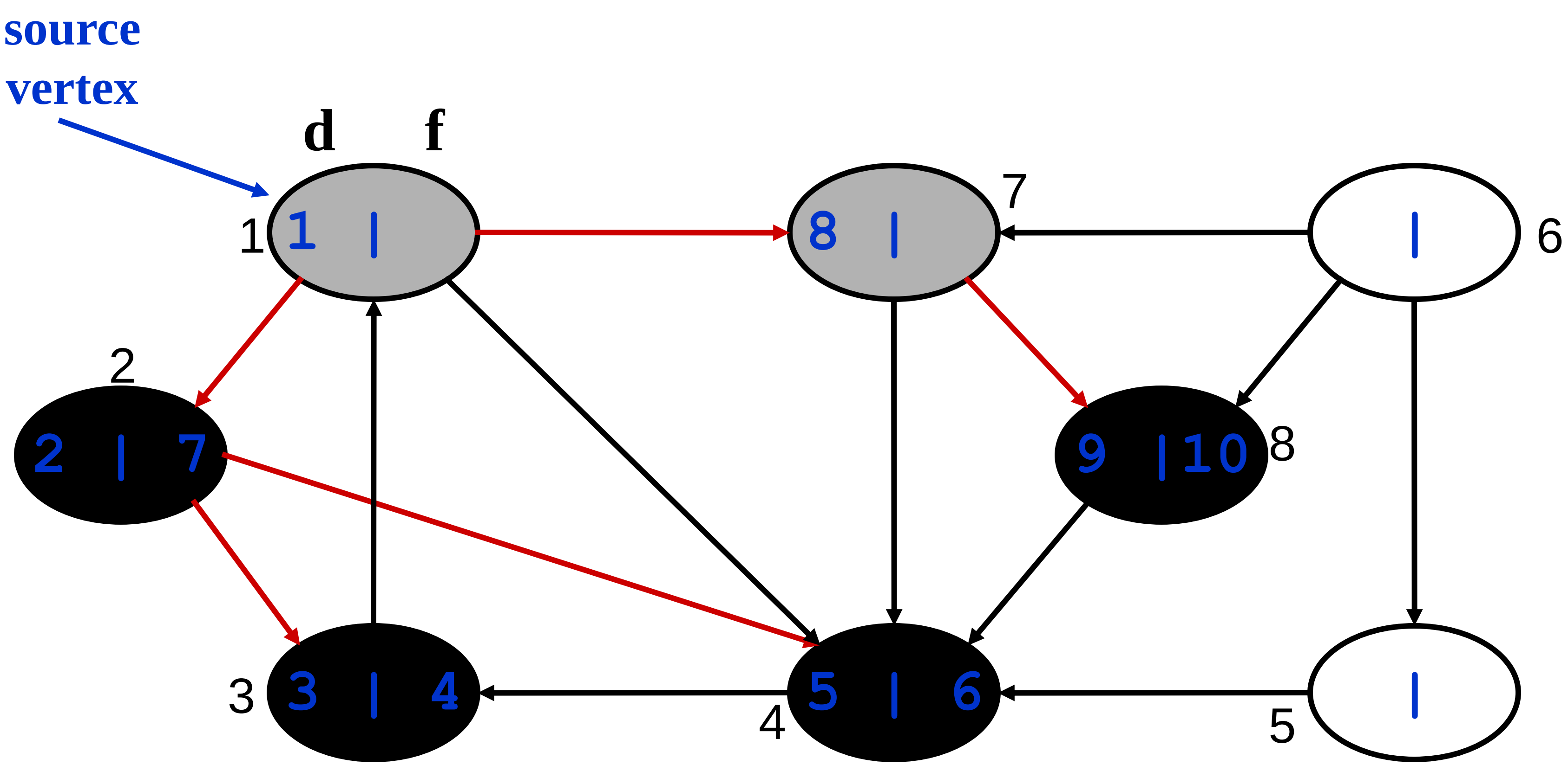


DFS Example

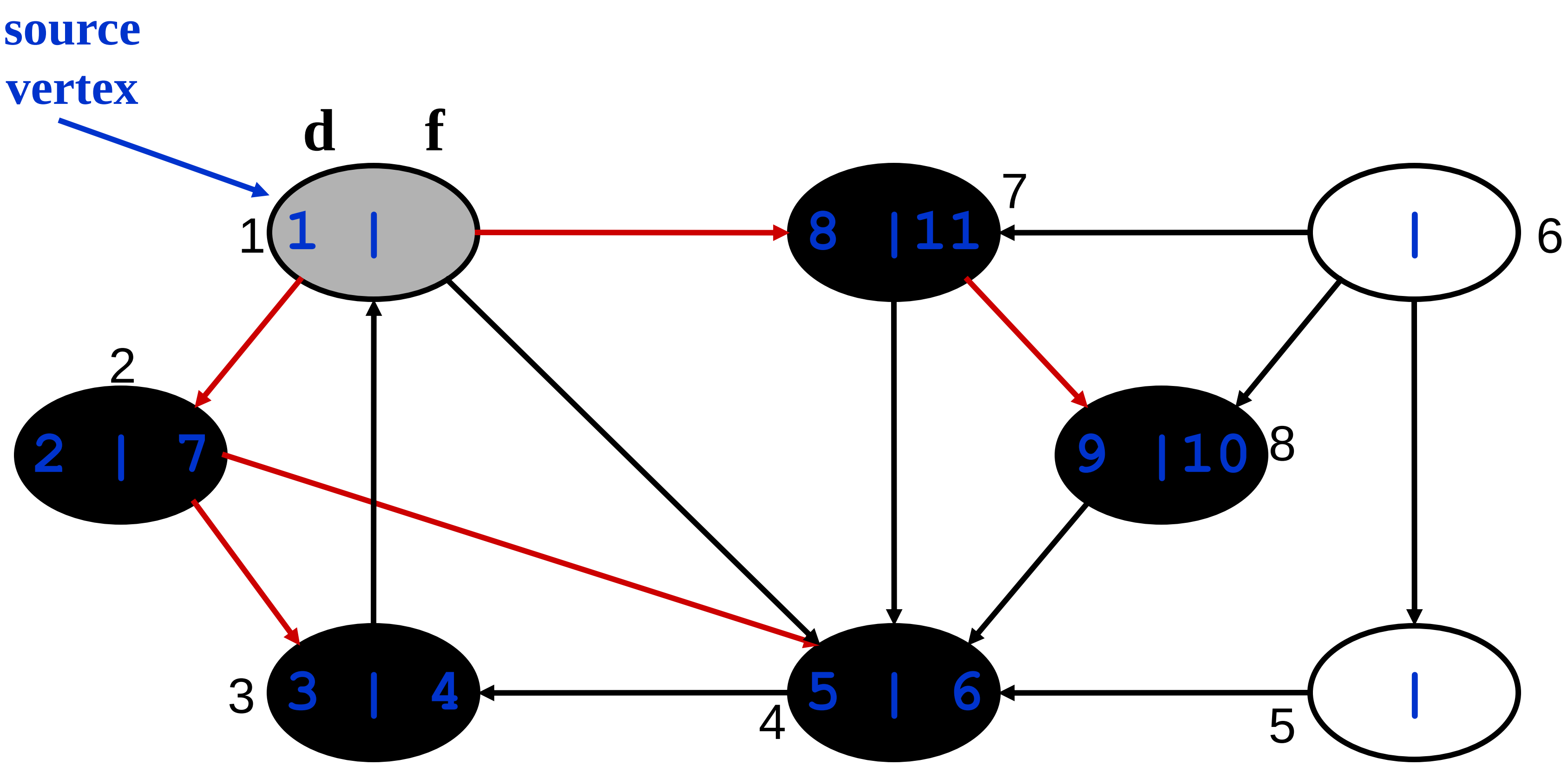


What is the structure of the grey vertices?
What do they represent?

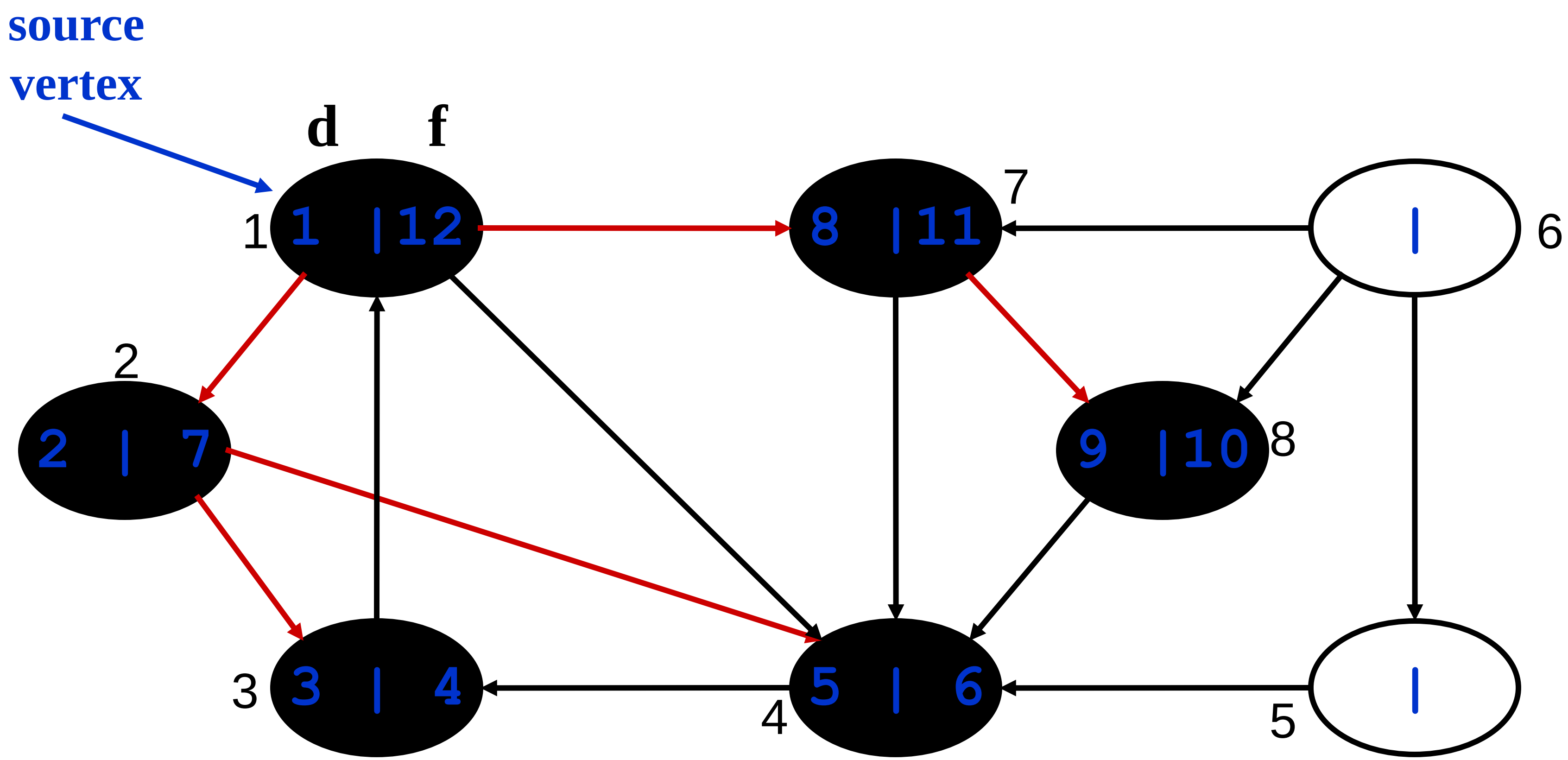
DFS Example



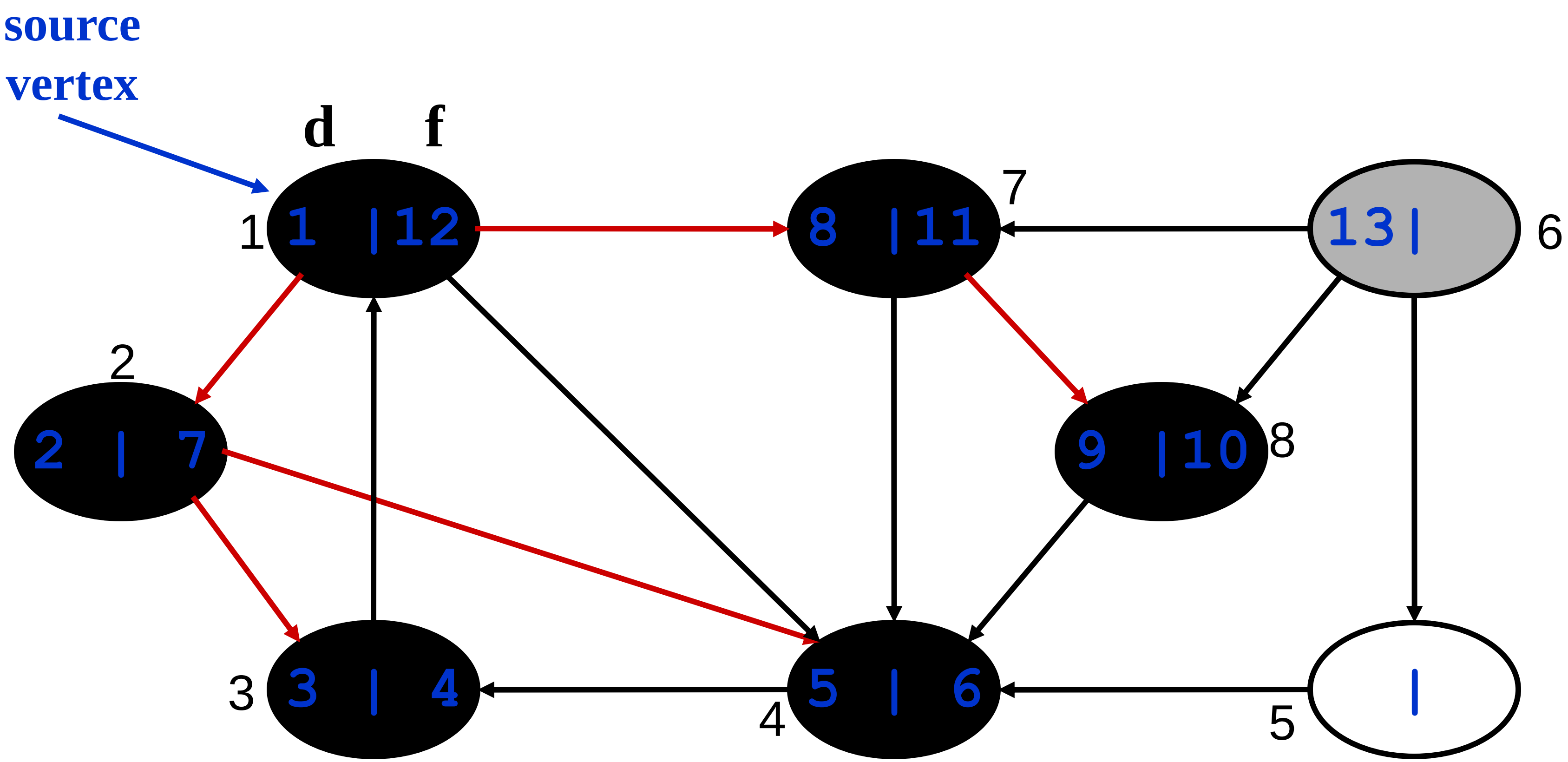
DFS Example



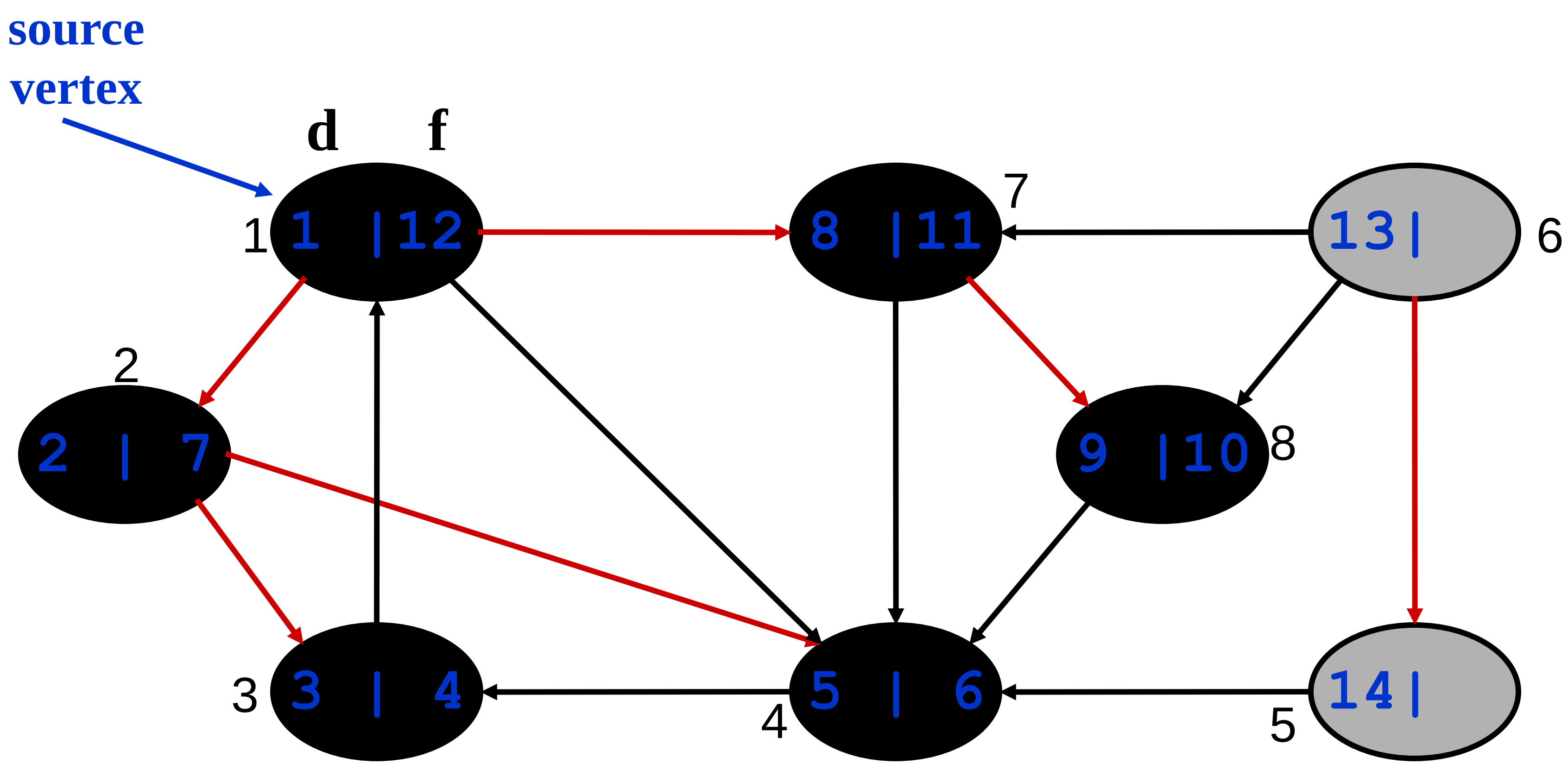
DFS Example



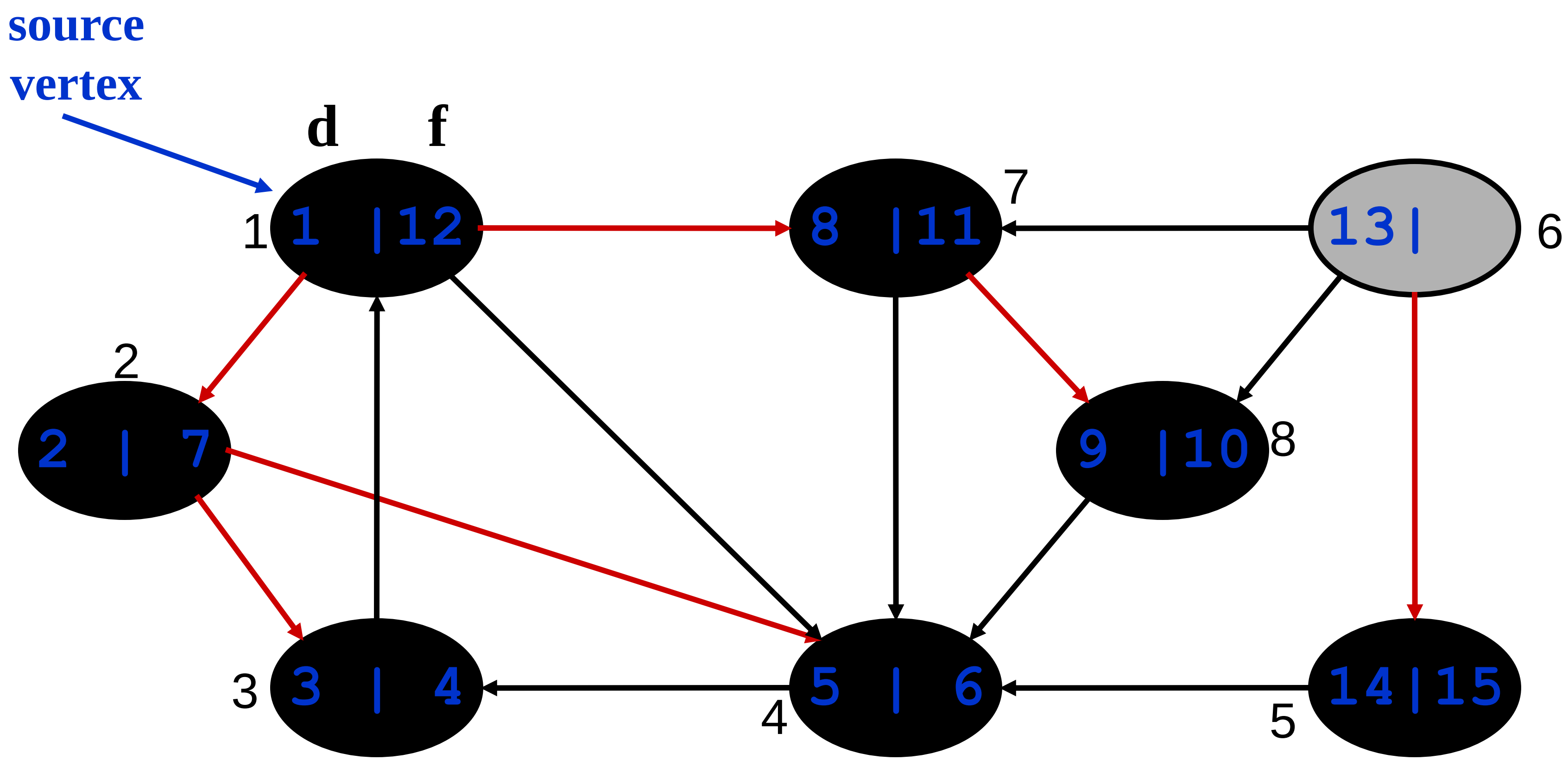
DFS Example



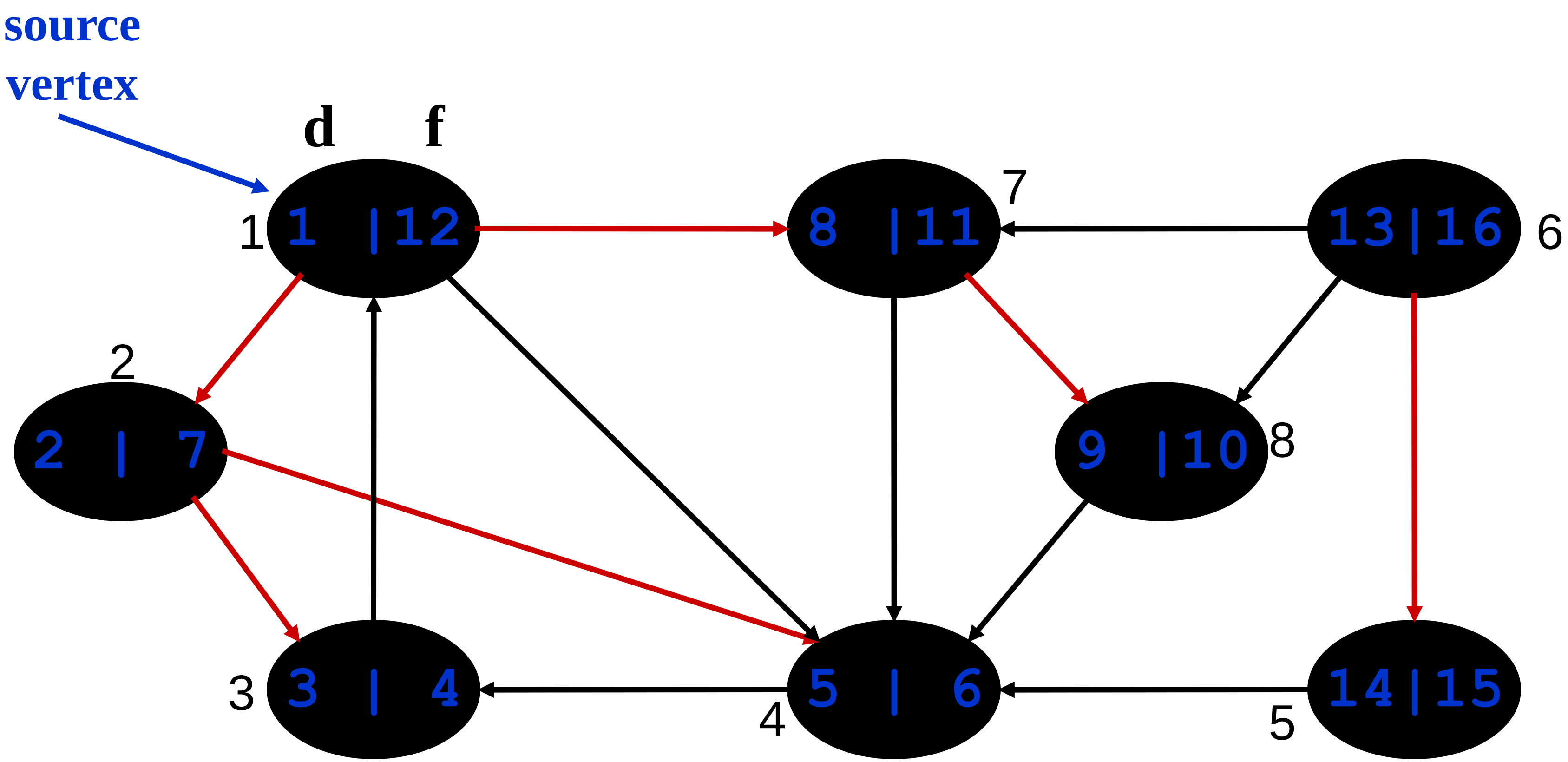
DFS Example



DFS Example



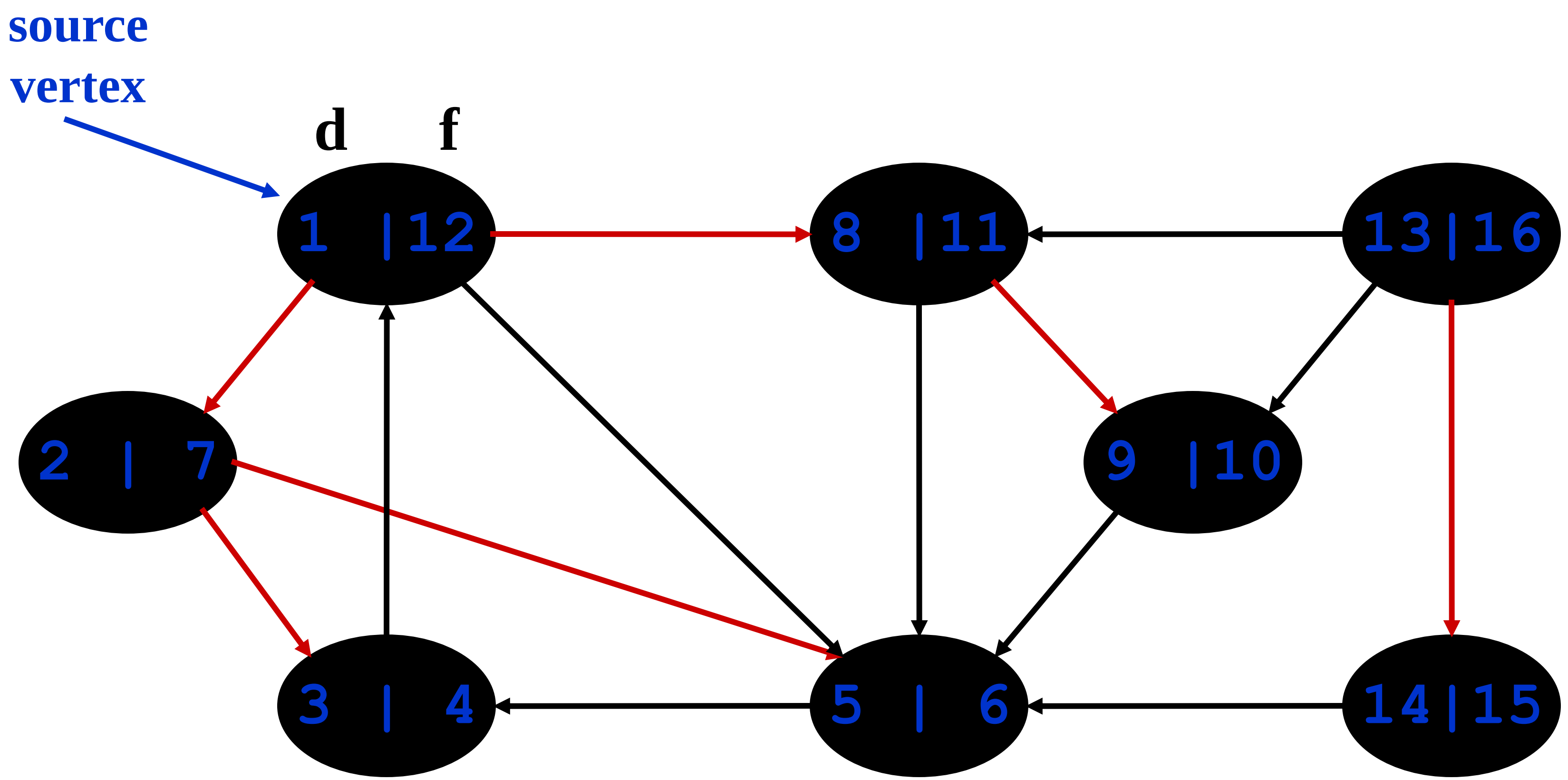
DFS Example



DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new (white) vertex
    - The tree edges form a spanning forest
    - Can tree edges form cycles? Why or why not?

DFS Example

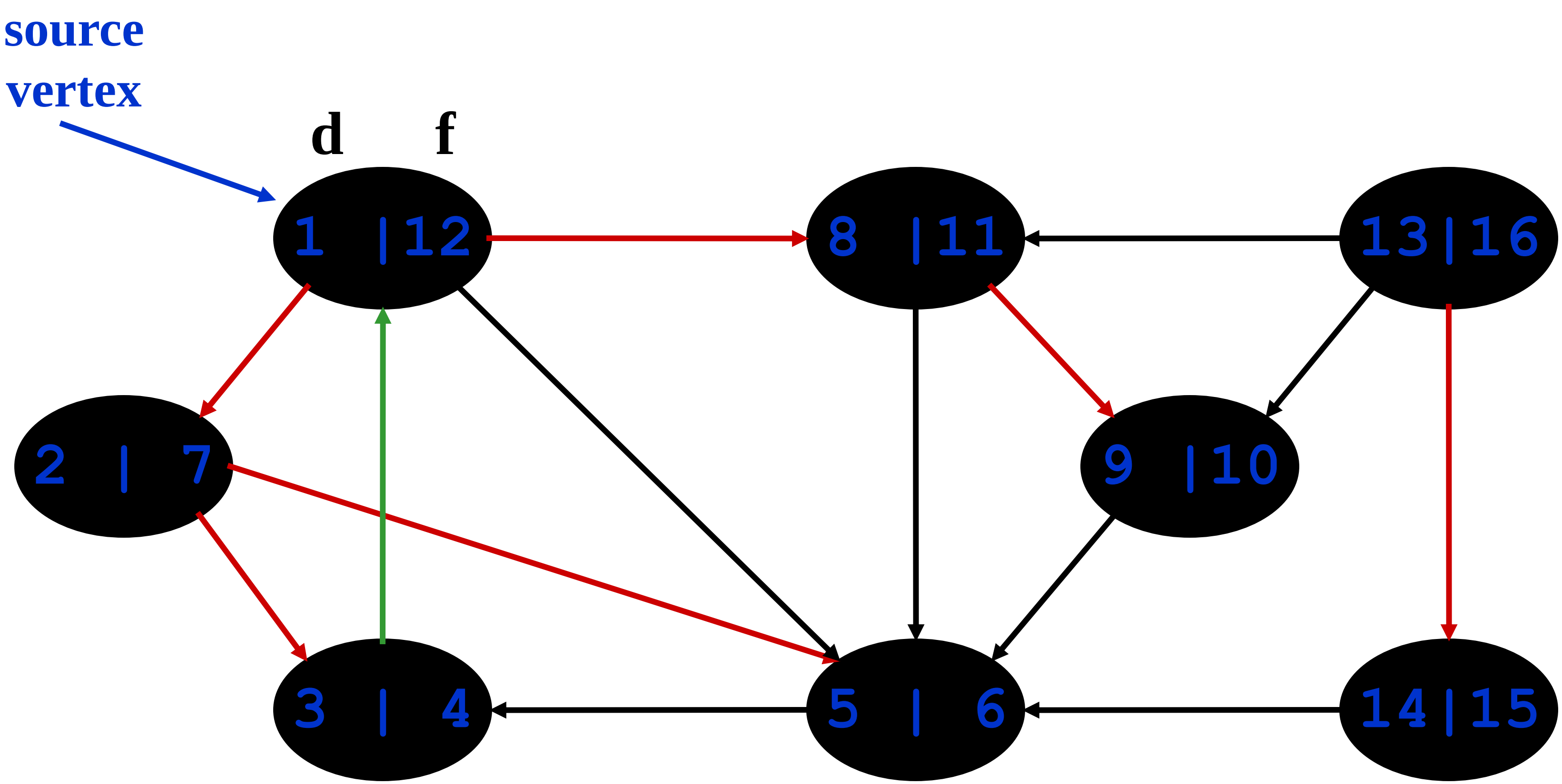


Tree edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new (white) vertex
  - Back edge: from descendent to ancestor
    - Encounter a grey vertex (grey to grey)

DFS Example



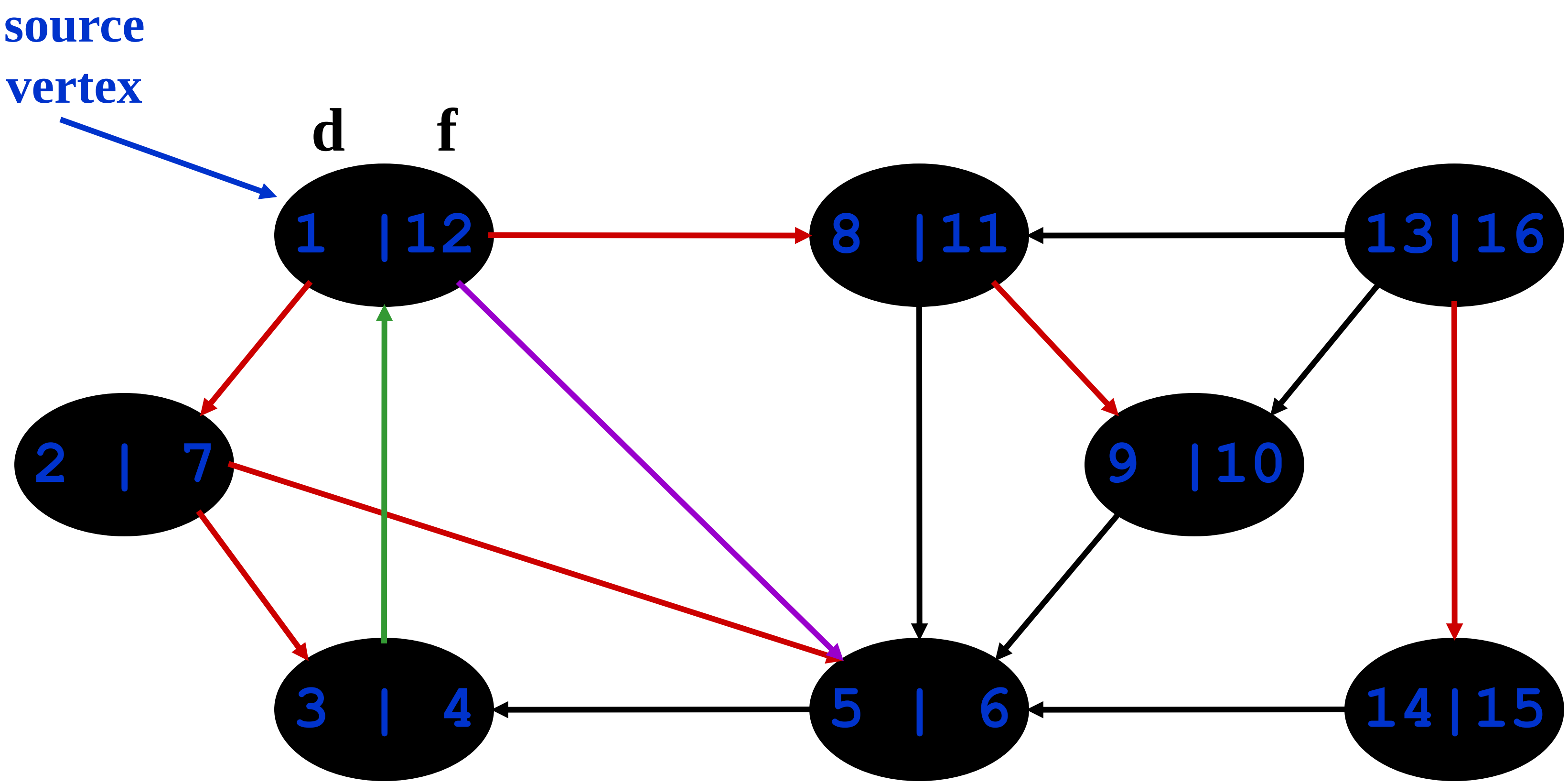
Tree edges **Back edges**

G to W **G to G**

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new (white) vertex
  - Back edge: from descendent to ancestor
  - Forward edge: from ancestor to descendent
    - Not a tree edge, though
    - From grey node to black node

DFS Example

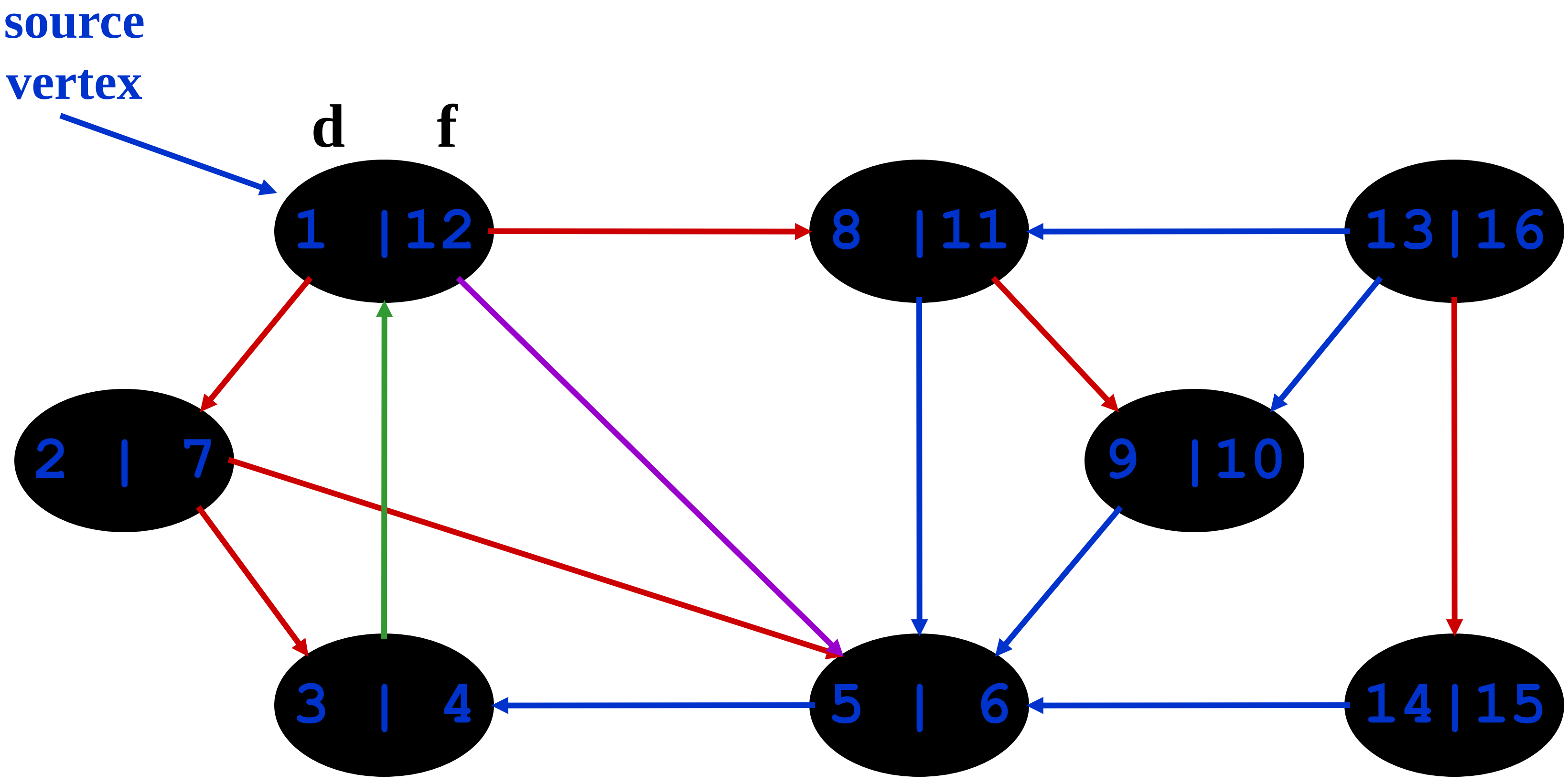


Tree edges	Back edges	Forward edges
G to W	G to G	G to B

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new (white) vertex
  - Back edge: from descendent to ancestor (encounters grey)
  - Forward edge: from ancestor to descendent (encounters black)
  - Cross edge: between a tree or subtrees
    - From a grey node to a black node

DFS Example



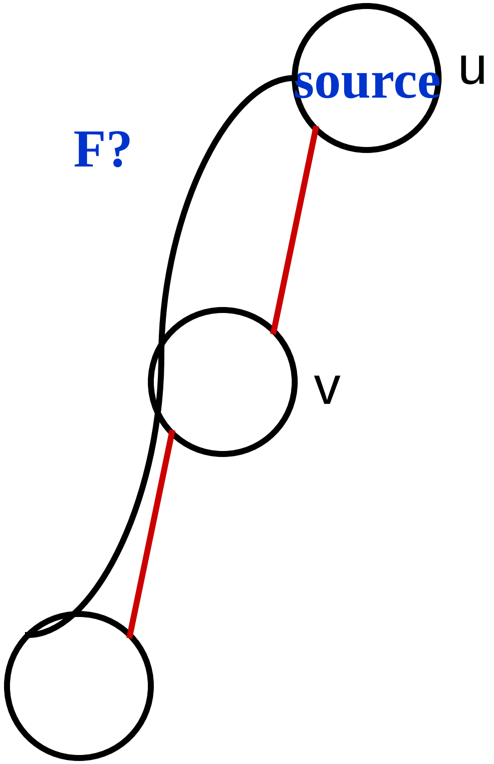
Tree edges G to W **Back edges** G to G **Forward edges** G to B **Cross edges** G to B

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new (white) vertex
  - Back edge: from descendent to ancestor
  - Forward edge: from ancestor to descendent
  - Cross edge: between a tree or subtrees
- Note: tree & back edges are important: most algorithms don't distinguish forward & cross

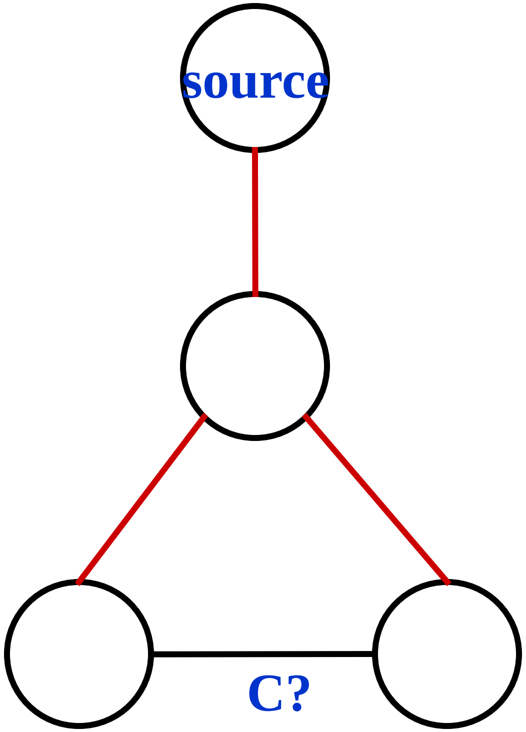
DFS: Kinds Of Edges

- Thm 23.9: If G is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a forward edge
    - But F? edge must actually be a back edge (why?)



DFS: Kinds Of Edges

- Thm 23.9: If G is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a cross edge
    - But C? edge cannot be cross:
    - must be explored from one of the vertices it connects, becoming a tree edge, before other vertex is explored
    - So in fact the picture is wrong…both lower tree edges cannot in fact be tree edges

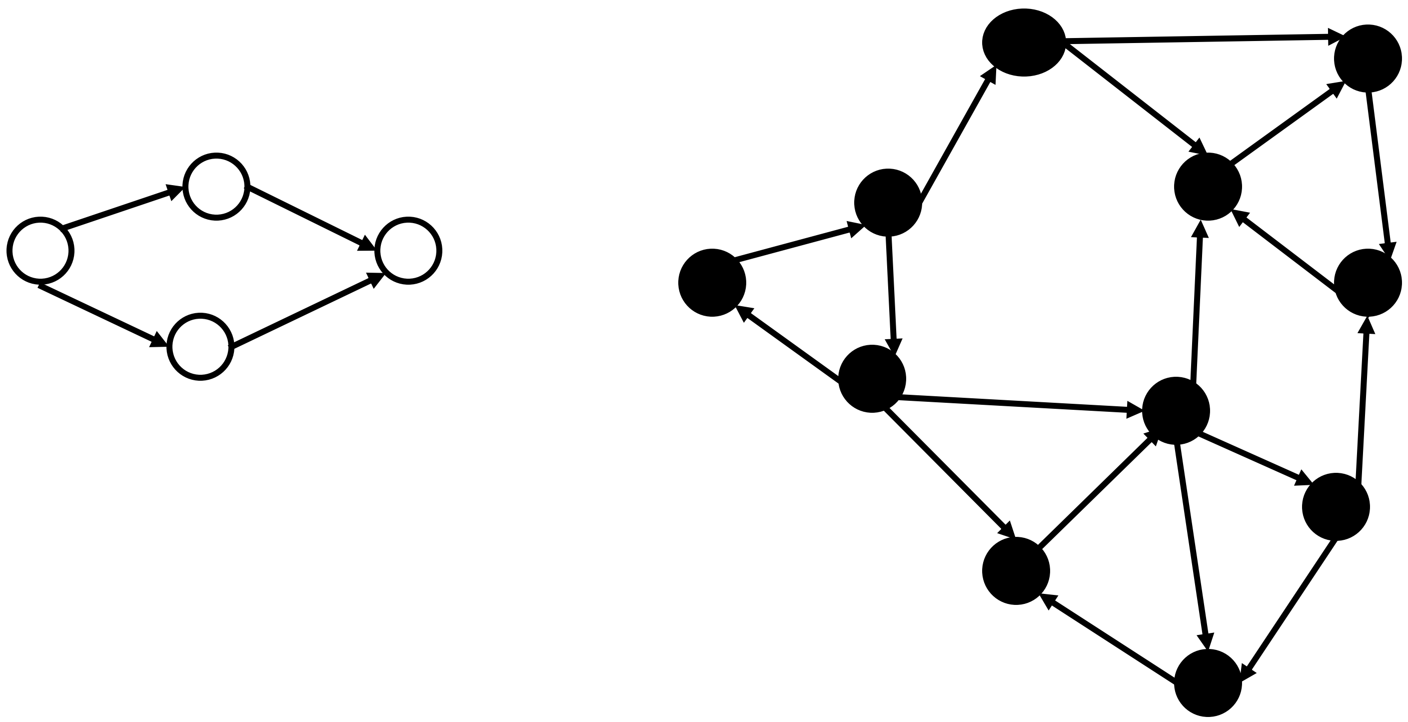


Exercise

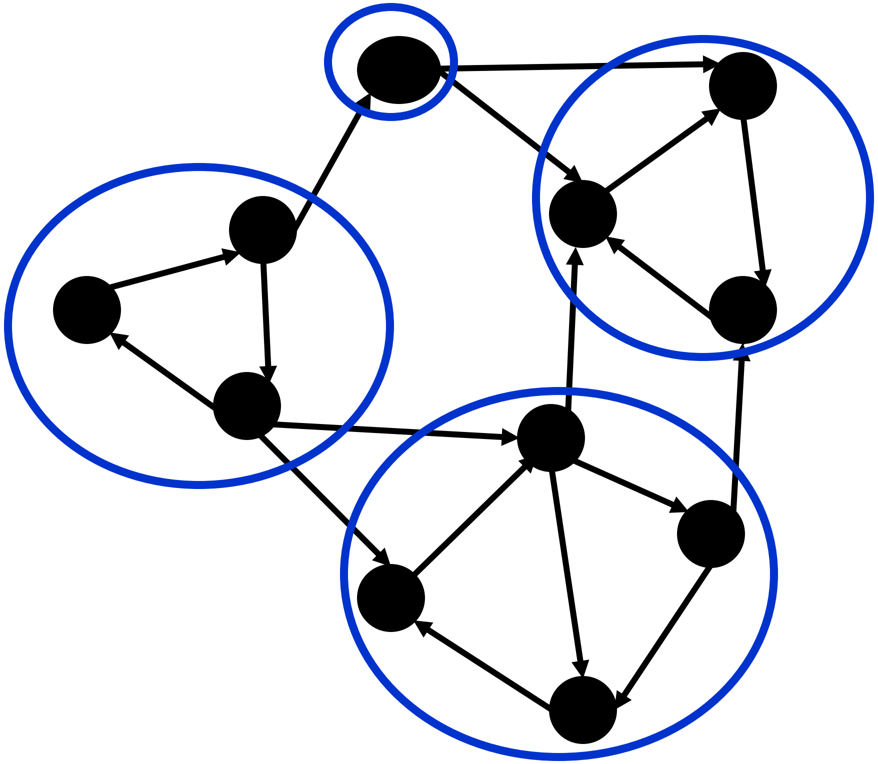
- Given a direct graph G, use DFS to find its strong connected components.

- SCC of a direct graph G=<V,E> is a maximal set of vertices C in V such that each pair u and v in C, u and v are reachable from each other.

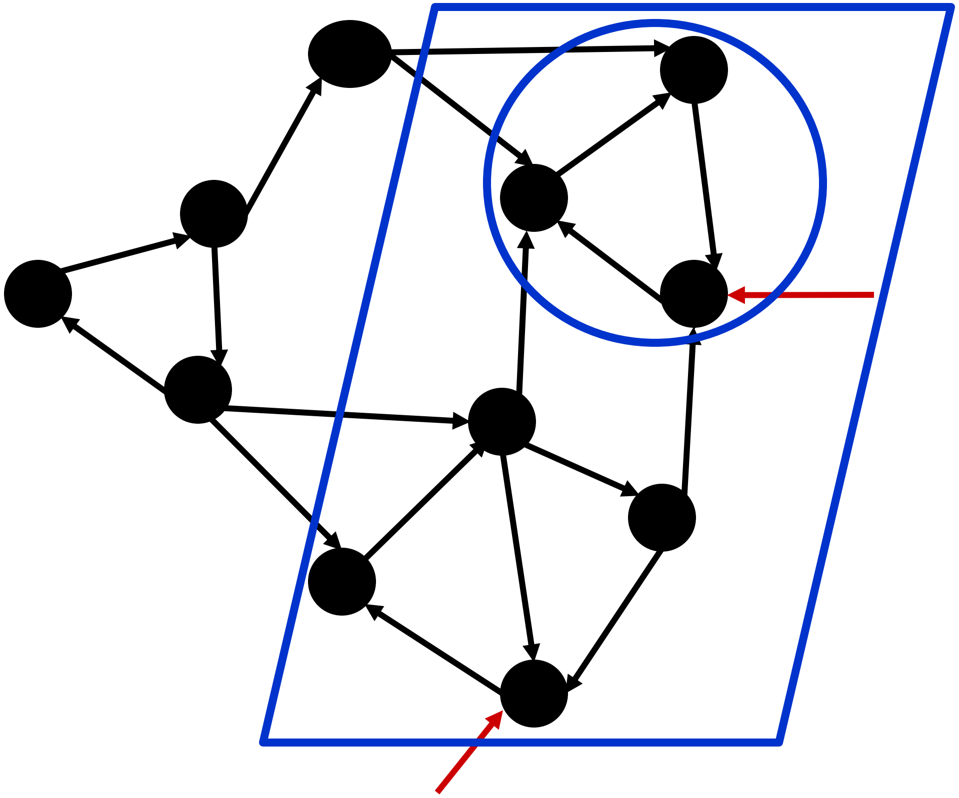
Strong Connected Components



Strong Connected Components



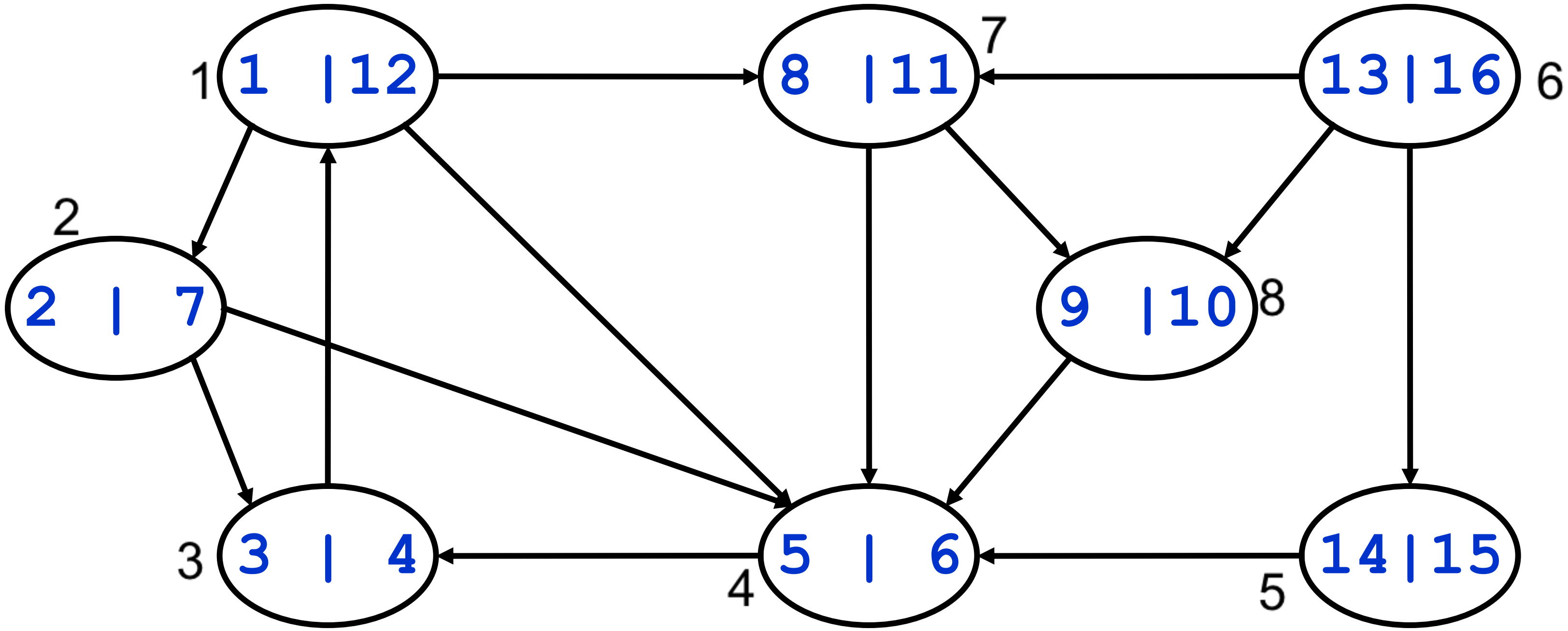
Strong Connected Components



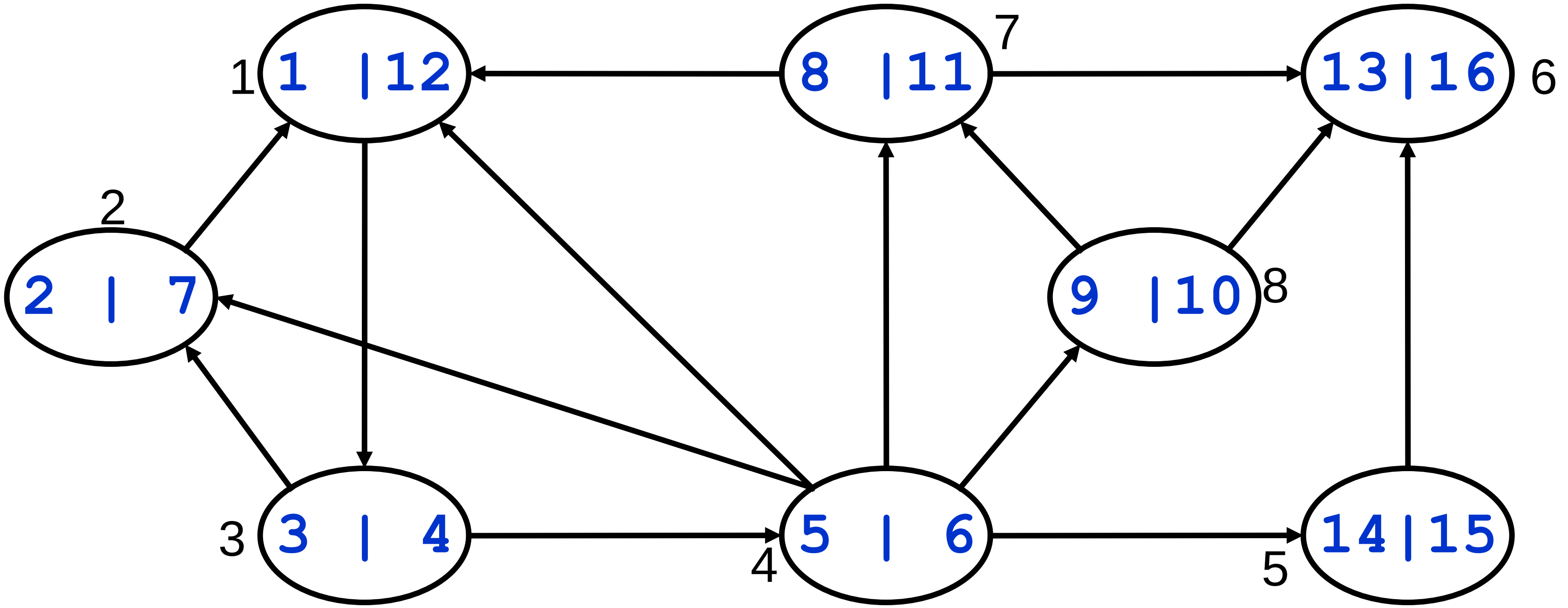
Solution

- Call DFS (G) to compute finish times f of each vertex in G
- Compute G^T
- Call DFS(G^T), but main loop considers finish time f in decreasing order
- Each connected component corresponds to a tree found by DFS in G^T

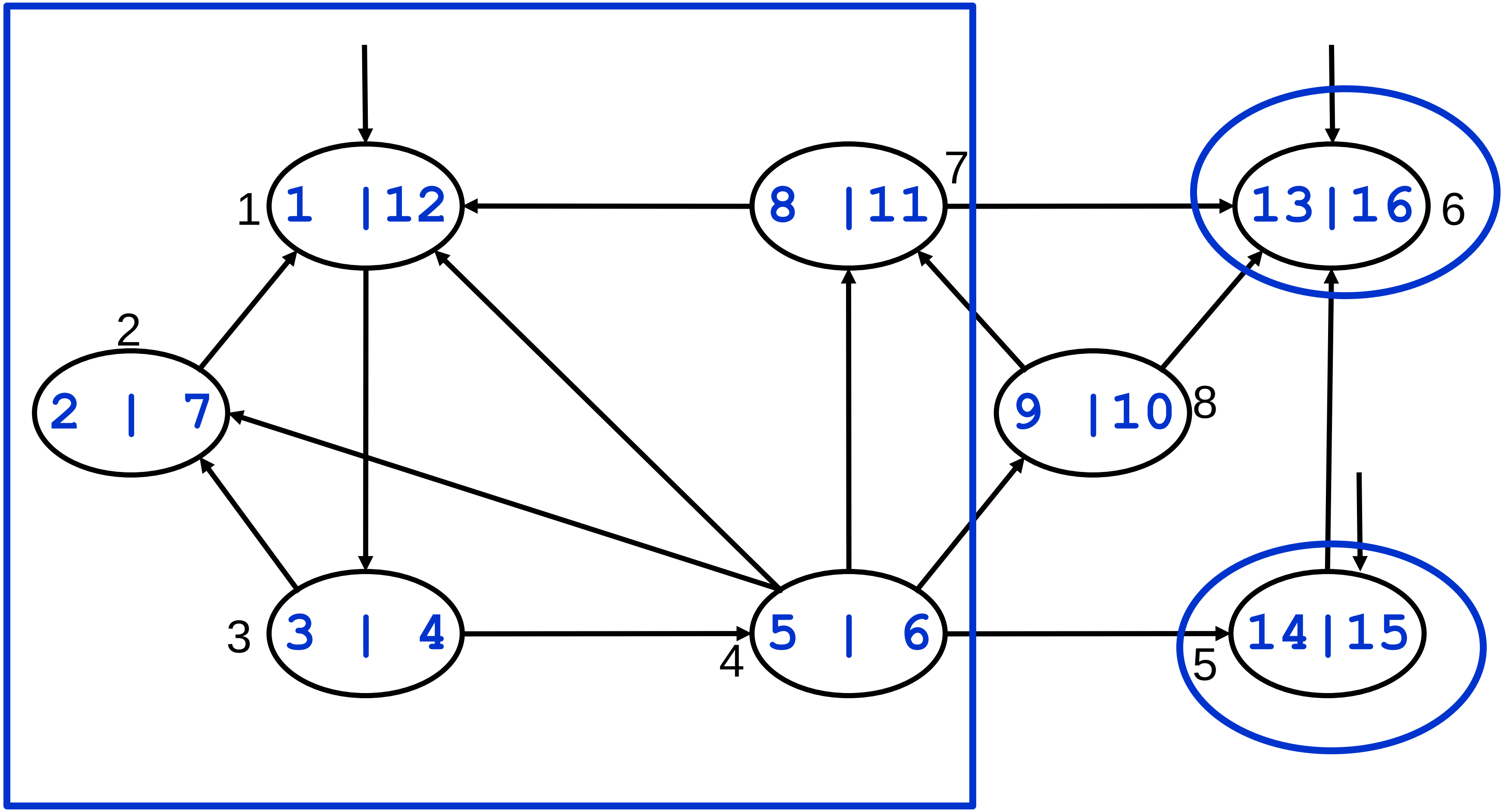
Example



Example



Example



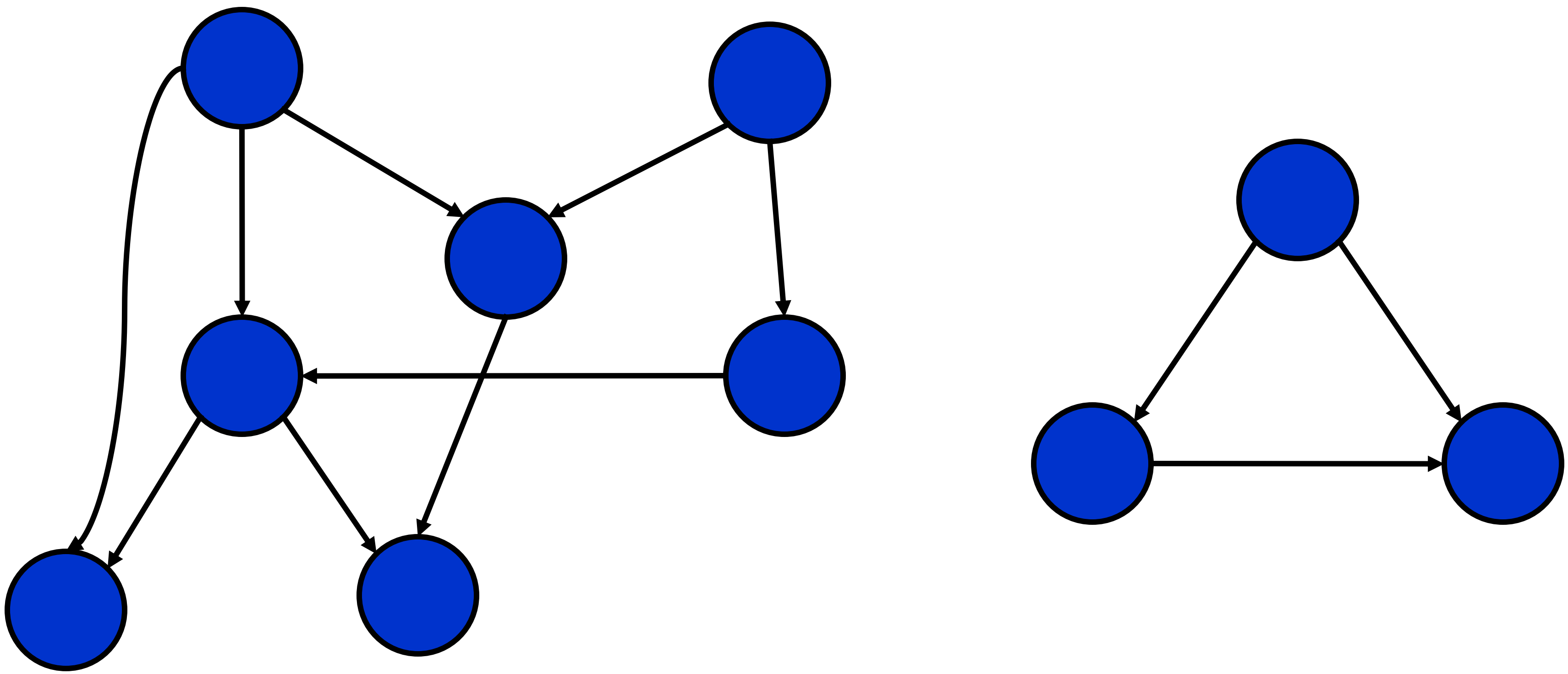
Summary

- Graph represention:
  - Adjancency matrix (dense graphs) – $O(V^2)$
  - Adjancency list (sparse graphs) - $O(V+E)$

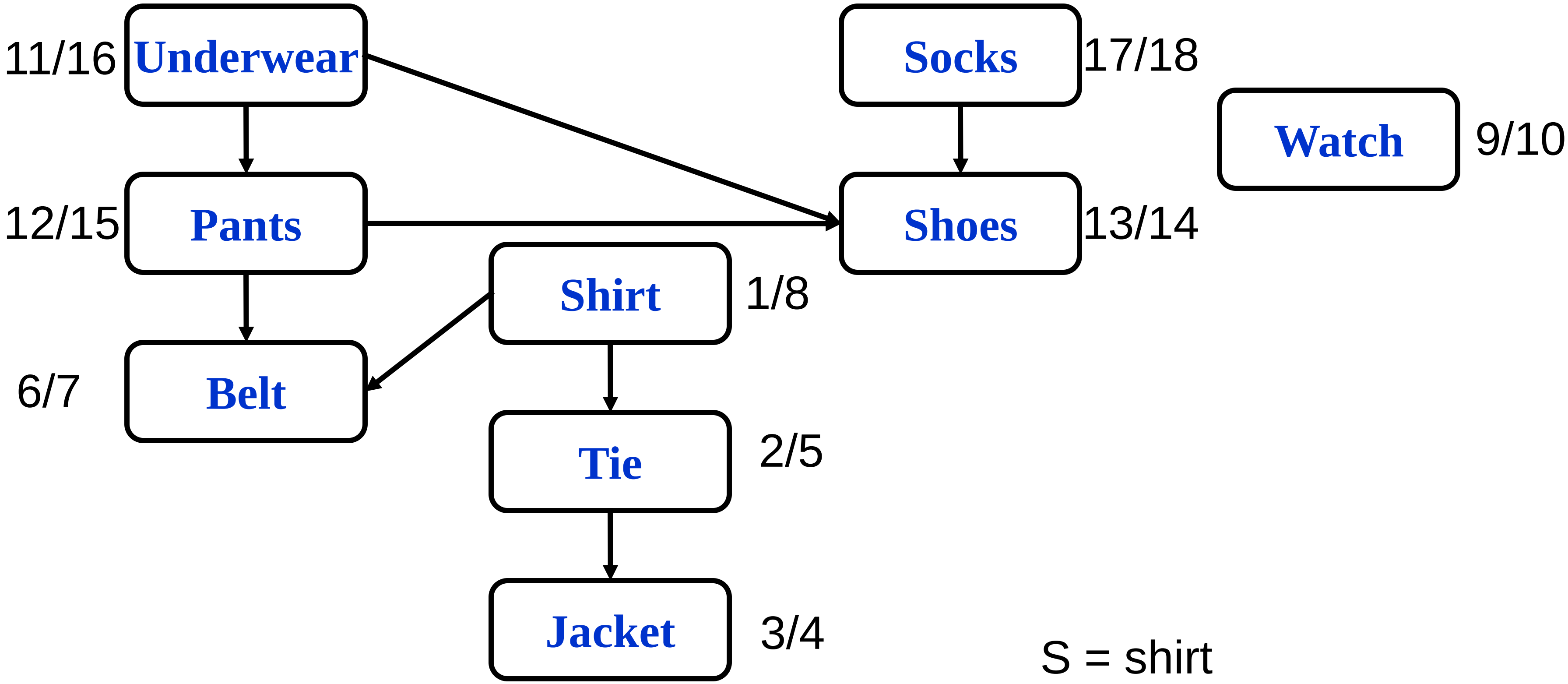
- Algorithms for searching graphs
  - BFS + DFS
  - Both run in **Θ(V + E**) – **aggregate analysis**

Directed Acyclic Graphs

- A directed acyclic graph or DAG is a directed graph with no directed cycles:

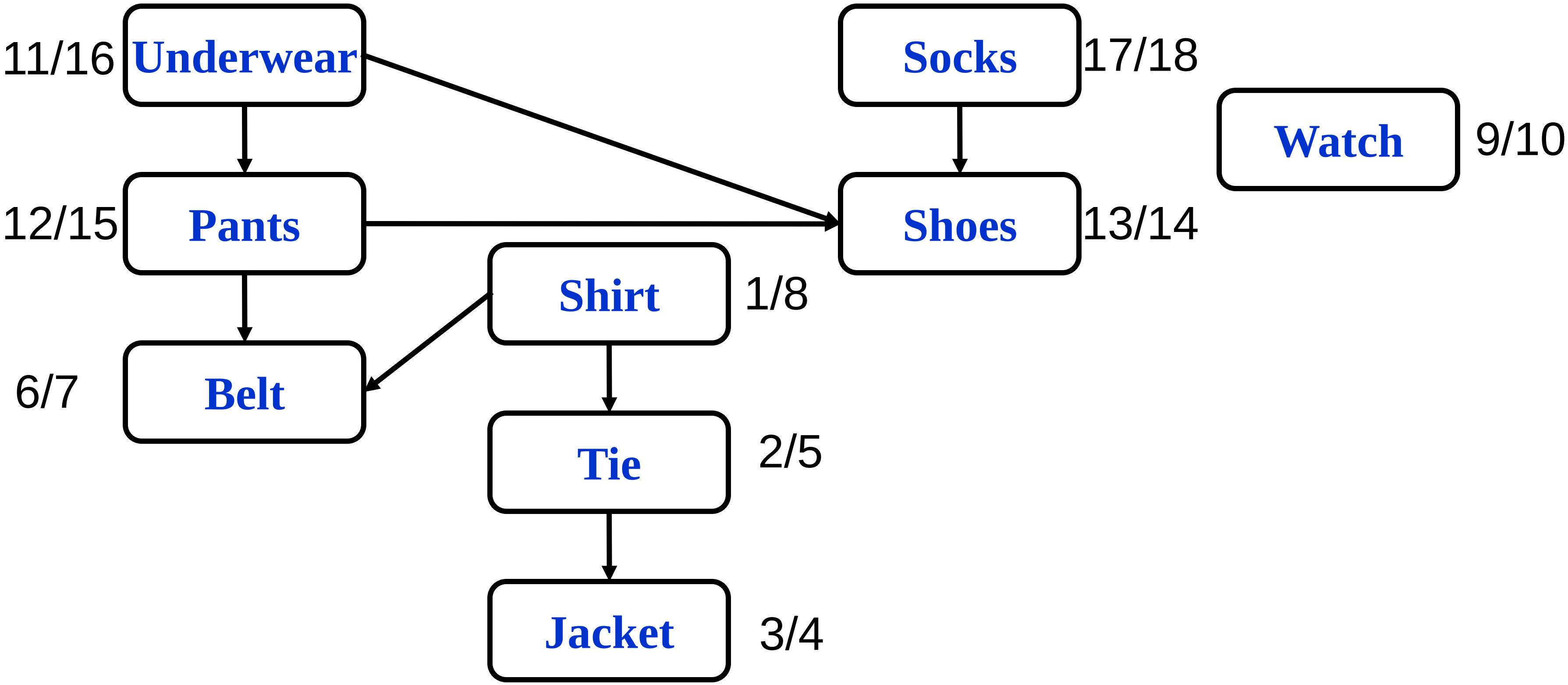


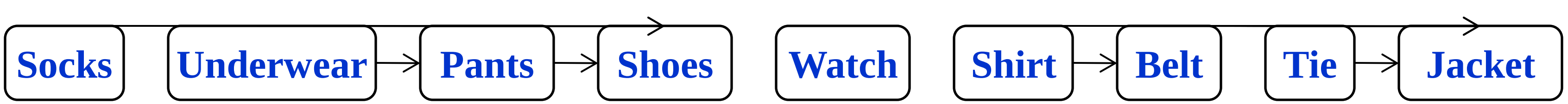
Getting Dressed



S = shirt

Getting Dressed





DFS and DAGs

- A directed graph G is acyclic iff a DFS of G yields no back edges:
  - Forward: if DFS produces a back edge (u,v), v is an ancestor of u. G contains a path from v to u, and the edge (u,v) completes de cycle
  - Backward: if G contains a cycle $\Rightarrow \exists$ a back edge
    - Let v be the vertex on the cycle first discovered, and u be the predecessor of v on the cycle
      - When v is discovered, whole cycle is white
    - Must visit everything reachable from v before returning from DFS-Visit()
      - So path from $u \rightarrow v$ is grey$\rightarrow$grey, thus (u, v) is a back edge

Topological Sort

- Topological sort of a DAG:
  - Linear ordering of all vertices in graph G such that vertex u comes before vertex v if edge $(u, v) \in G$
- Real-world example: getting dressed

Topological Sort Algorithm

```
Topological-Sort()
{
   Run DFS
   When a vertex is finished, insert it onto
     the front of a linked list
   Return the linked list of vertices
}
```

- Time: O(V+E)
- Correctness: Can be proved using lemma that characterizes Direct Acyclic Graphs (DAGs)

Correctness of Topological Sort

- Topologycal sorting produces a topological sort of a DAG
- Claim: $(u,v) \in G \Rightarrow u \rightarrow f > v \rightarrow f$
  - When (u,v) is explored, u is grey
    - v = grey $\Rightarrow$ (u,v) is back edge. Contradiction to previous lemma
    - v = white $\Rightarrow$ v becomes descendent of u $\Rightarrow v \rightarrow f < u \rightarrow f$ (since must finish v before backtracking and finishing u)
    - v = black $\Rightarrow$ v already finished $\Rightarrow v \rightarrow f < u \rightarrow f$