طراحی و تحلیل الگوریتم ها

دکتر امیر لکی زاده

استادیار گروه مهندسی کامپیوتر دانشگاه قم

Graphs

- A graph $G = (V, E)$
  - V = set of vertices
  - E = set of edges = subset of $V \times V$
  - Thus $|E| = O(|V|^2)$

Graph Variations

- Variations:
  - A *connected graph* has a path from every vertex to every other
  - In an *undirected graph:*
    - Edge (u,v) = edge (v,u)
    - No self-loops
  - In a *directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated u→v

Graph Variations

- More variations:
  - A *weighted graph* associates weights with either the edges or the vertices
    - E.g., a road map: edges might be weighted w/ distance
  - A *multigraph* allows multiple edges between the same vertices
    - E.g., the call graph in a program (a function can get called from multiple points in another function)

Graphs

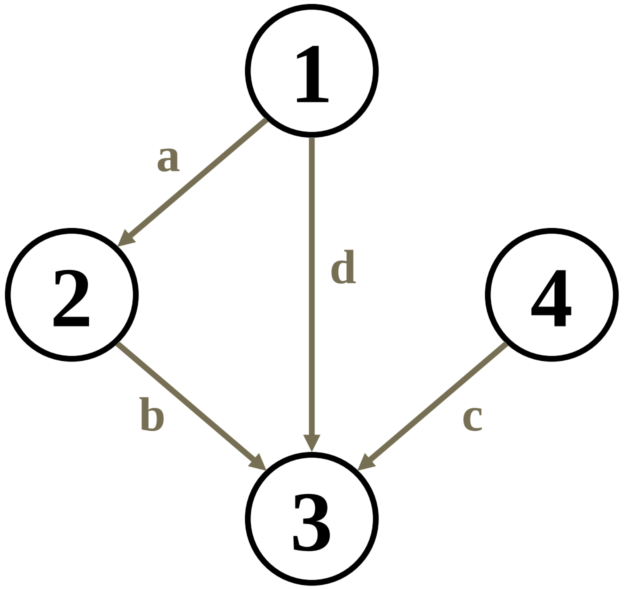
- We will typically express running times in terms of |E| and |V| (often dropping the |'s)
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

Representing Graphs

- Assume V = {1, 2, …, n}
- An *adjacency matrix* represents the graph as a n x n matrix A:
  - A[i, j] = 1 if edge (i, j) $\in$ E (or weight of edge)
    = 0 if edge (i, j) $\notin$ E

Graphs: Adjacency Matrix

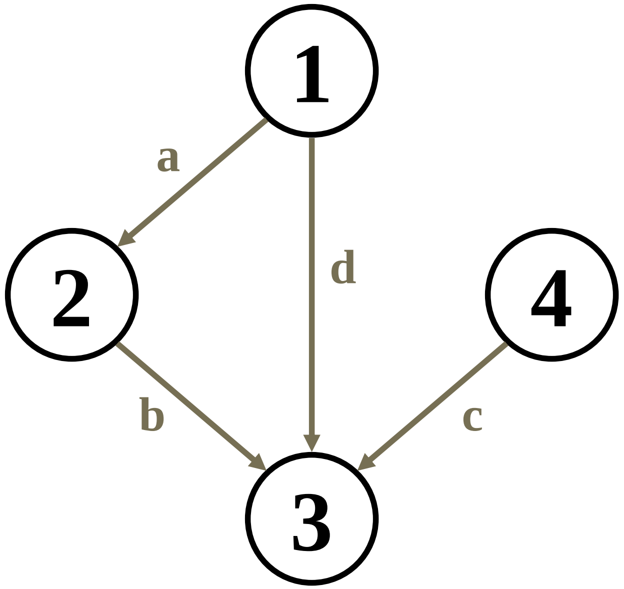
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

Graphs: Adjacency Matrix

- Example:



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

Graphs: Adjacency Matrix

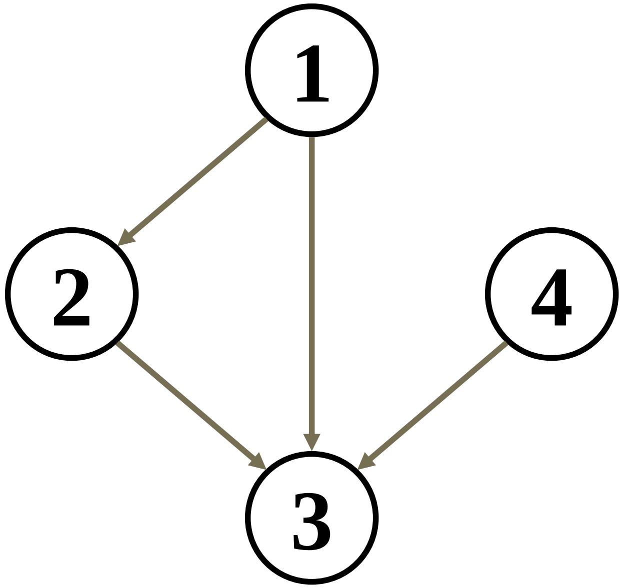
- *How much storage does the adjacency matrix require?*
- A: $O(V^2)$
- *What is the minimum amount of storage needed by an adjacency matrix representation of an undirected graph with 4 vertices?*
- A: 6 bits
  - Undirected graph $\rightarrow$ matrix is symmetric
  - No self-loops $\rightarrow$ don't need diagonal

Graphs: Adjacency Matrix

- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - E.g., planar graphs, in which no edges cross, have $|E| = O(|V|)$ by Euler's formula
  - For this reason the *adjacency list* is often a more appropriate respresentation

Graphs: Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep
  a list of edges coming *into* vertex



Graphs: Adjacency List

- How much storage is required?
  - The *degree* of a vertex v = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    Σ out-degree(v) = |E|
    takes Θ(V + E) storage (*Why?*)
  - For undirected graphs, # items in adj lists is
    Σ degree(v) = 2 |E| (*handshaking lemma*)
    also Θ(V + E) storage
- So: Adjacency lists take O(V+E) storage

Graph Searching

- Given: a graph G = (V, E), directed or undirected
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

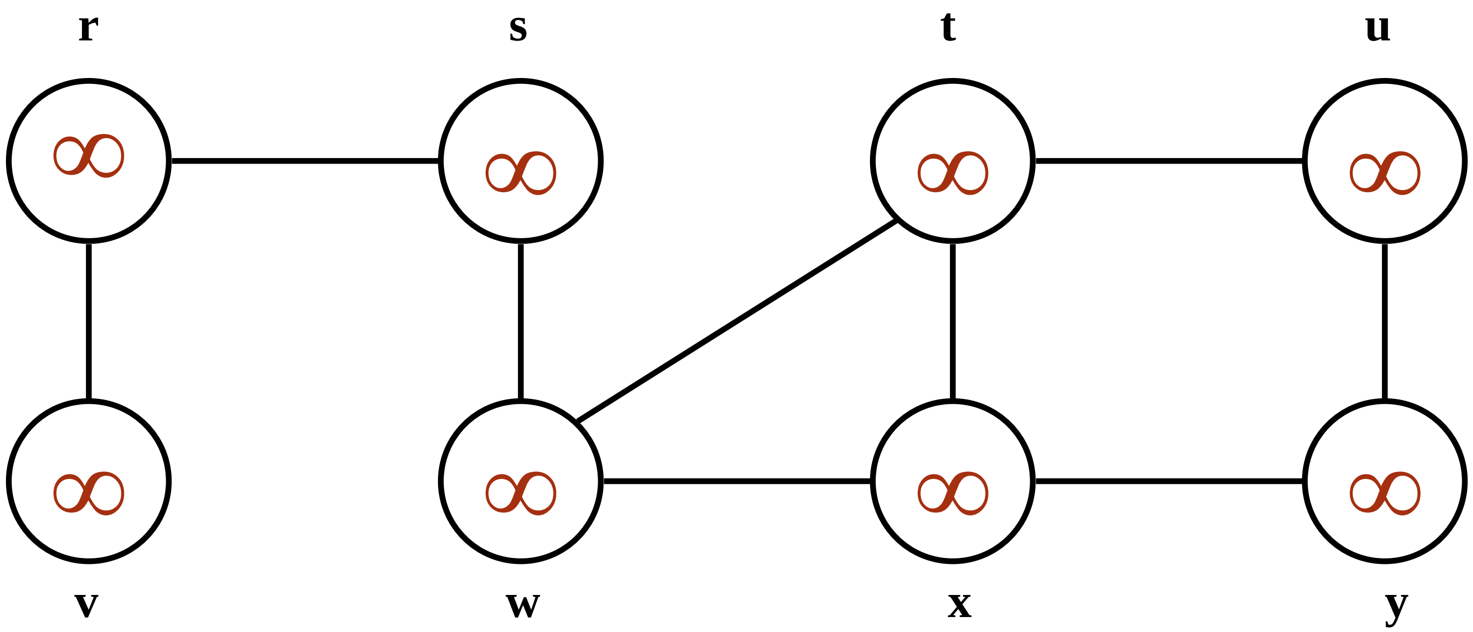
Breadth-First Search

- Again will associate vertex "colors" to guide the algorithm
  - White vertices have not been discovered
    - All vertices start out white
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

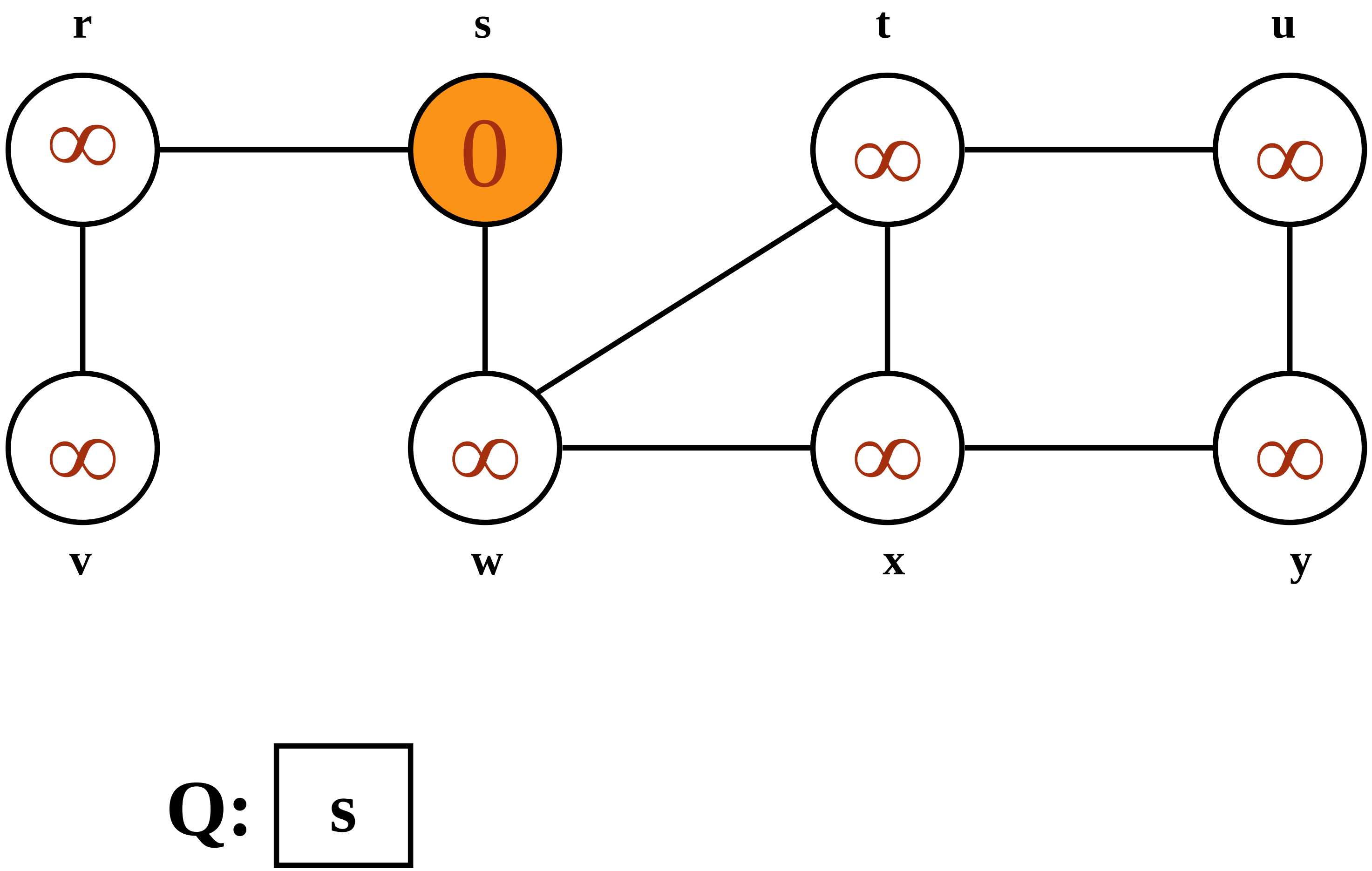
Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};            // Q is a queue (duh); initialize to s
    while (Q not empty) {
        u = RemoveTop(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;
    }
}
```

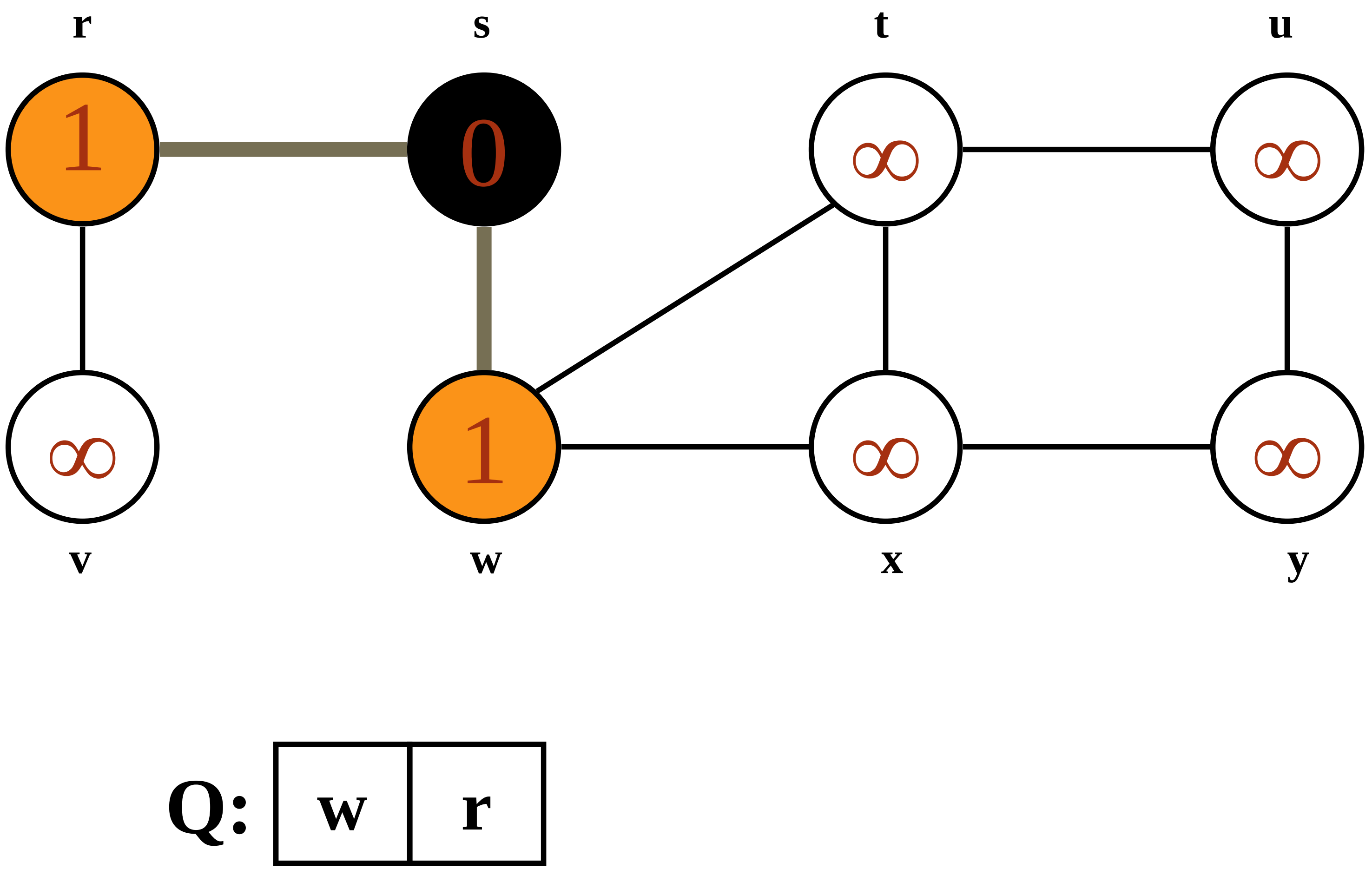
Breadth-First Search: Example



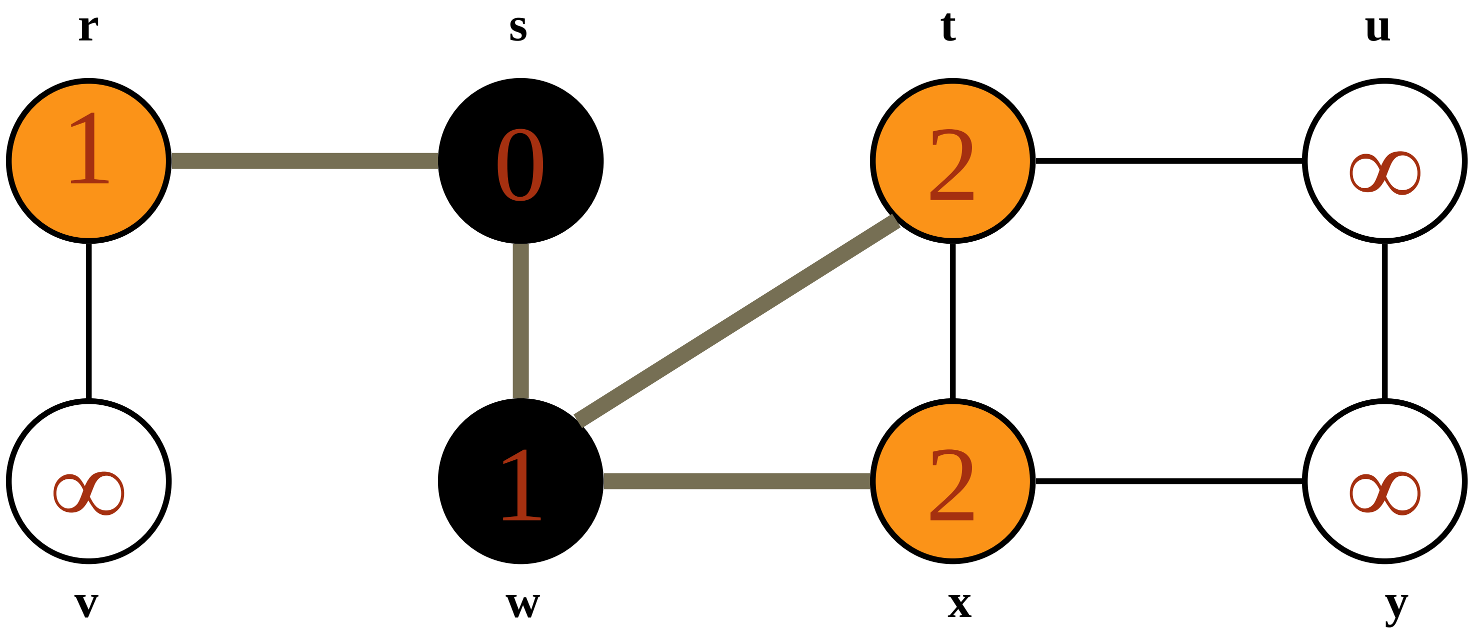
Breadth-First Search: Example



Breadth-First Search: Example

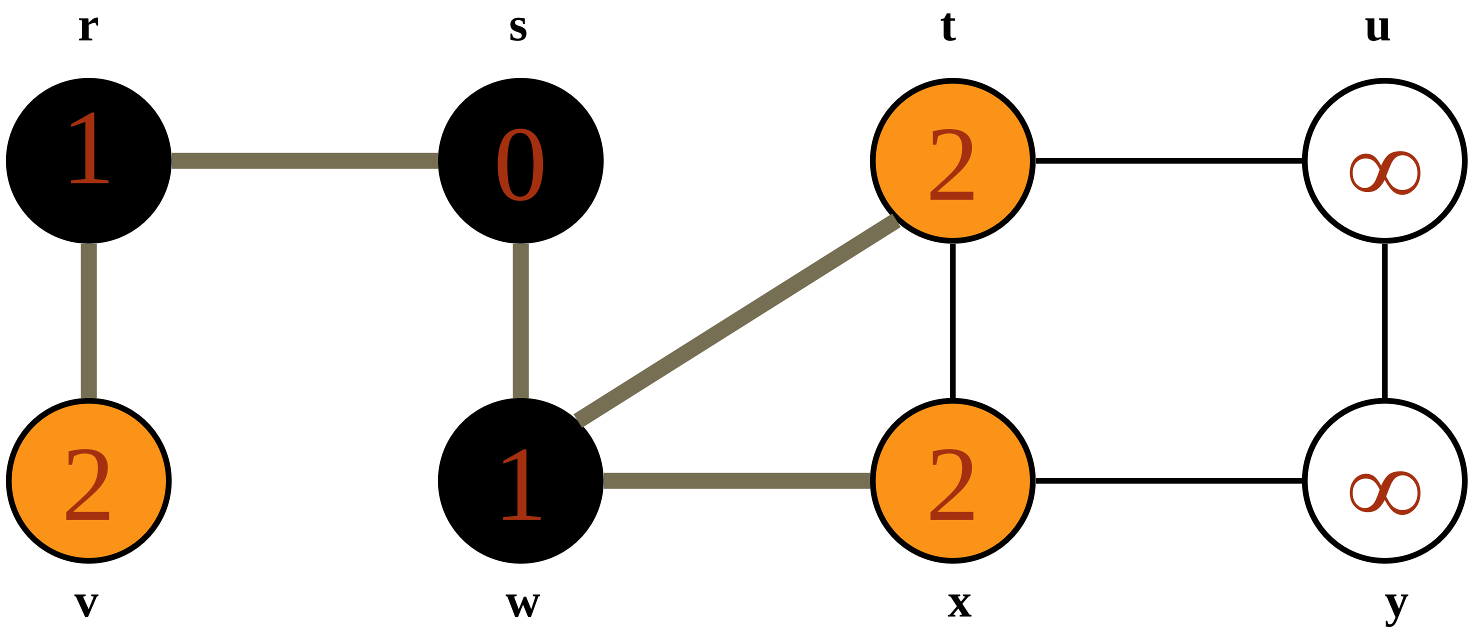


Breadth-First Search: Example



Q: | r | t | x |

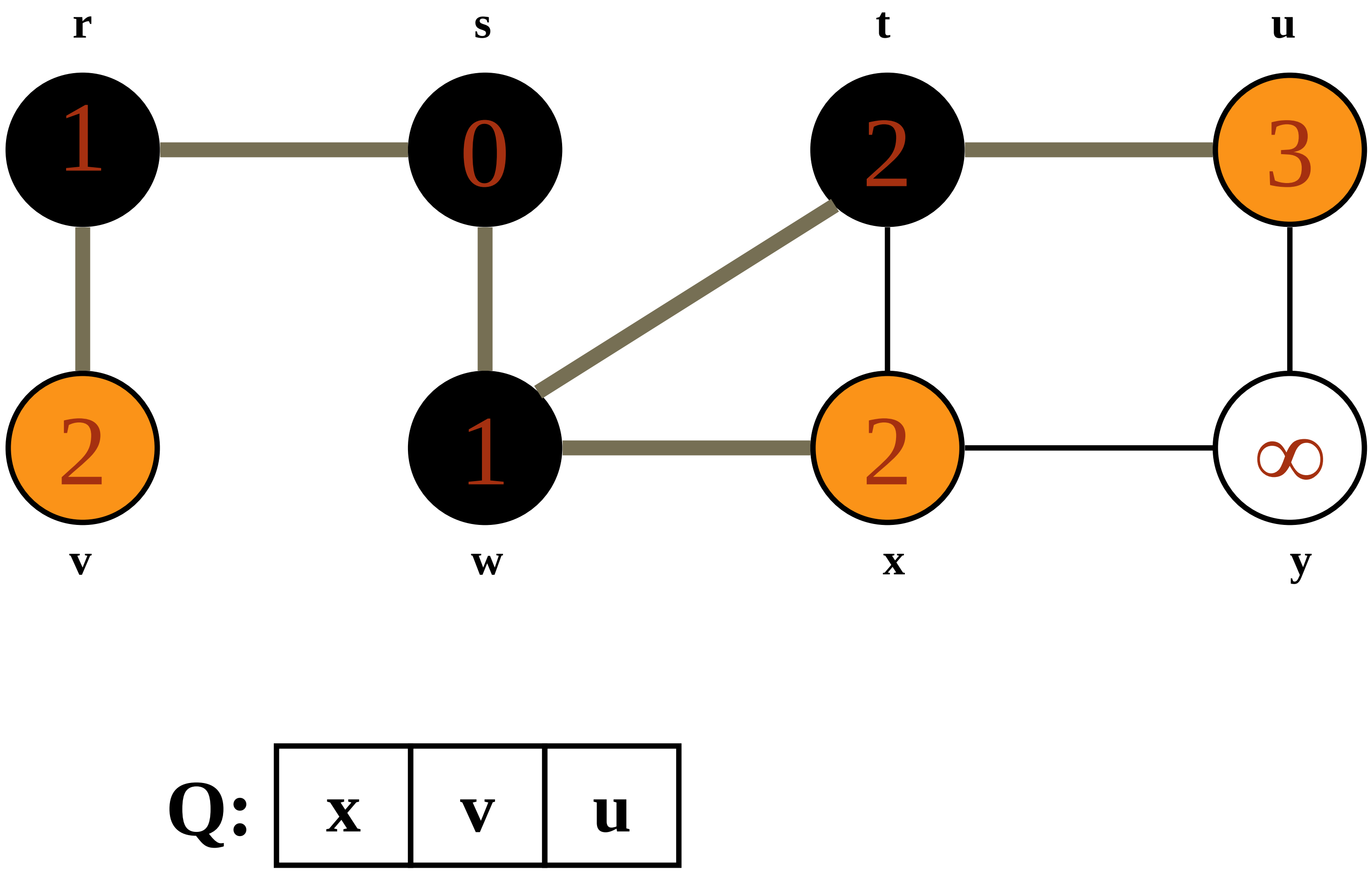
Breadth-First Search: Example



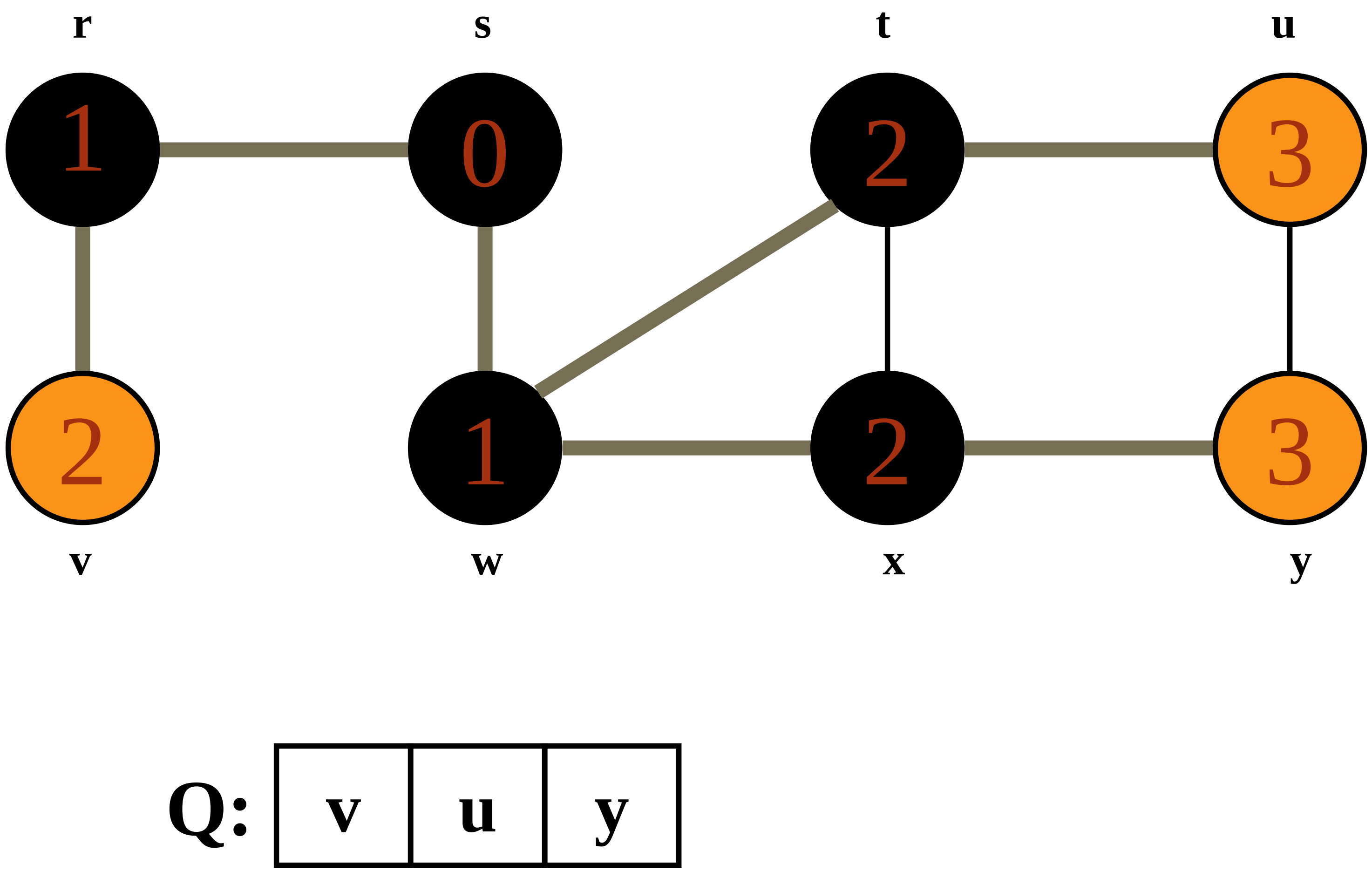
Q:

t	x	v

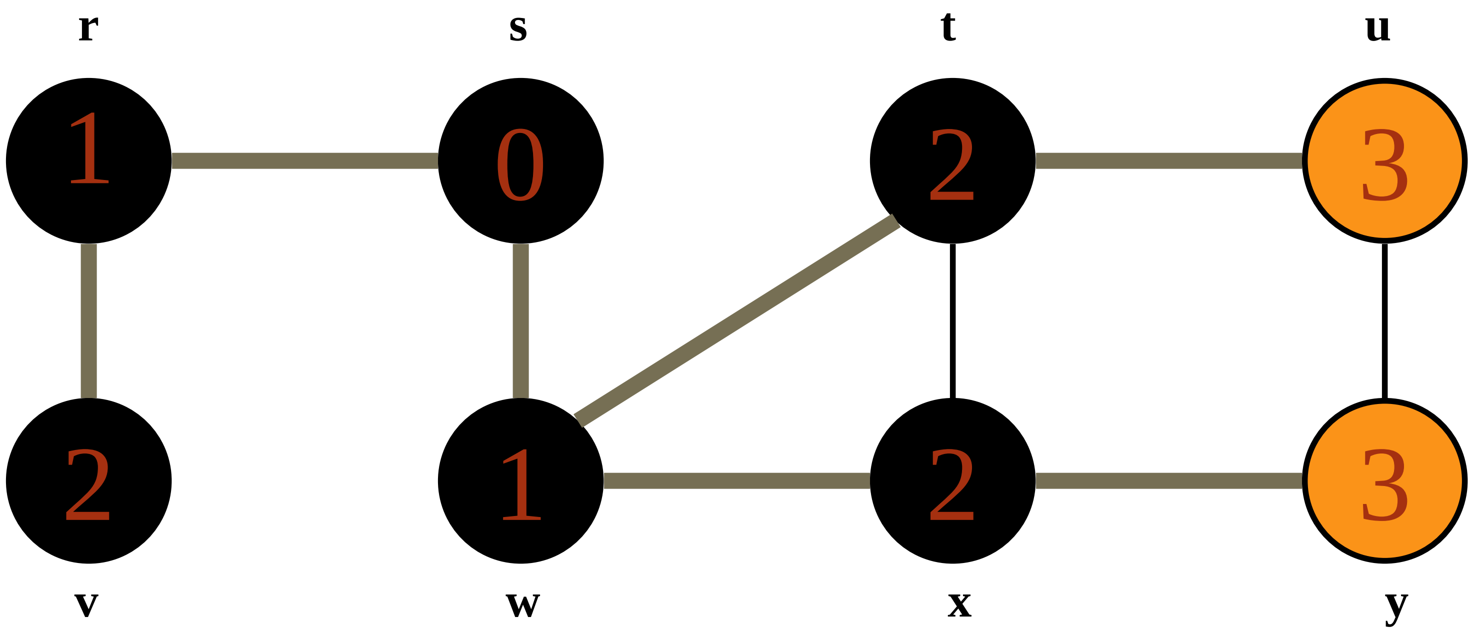
Breadth-First Search: Example



Breadth-First Search: Example

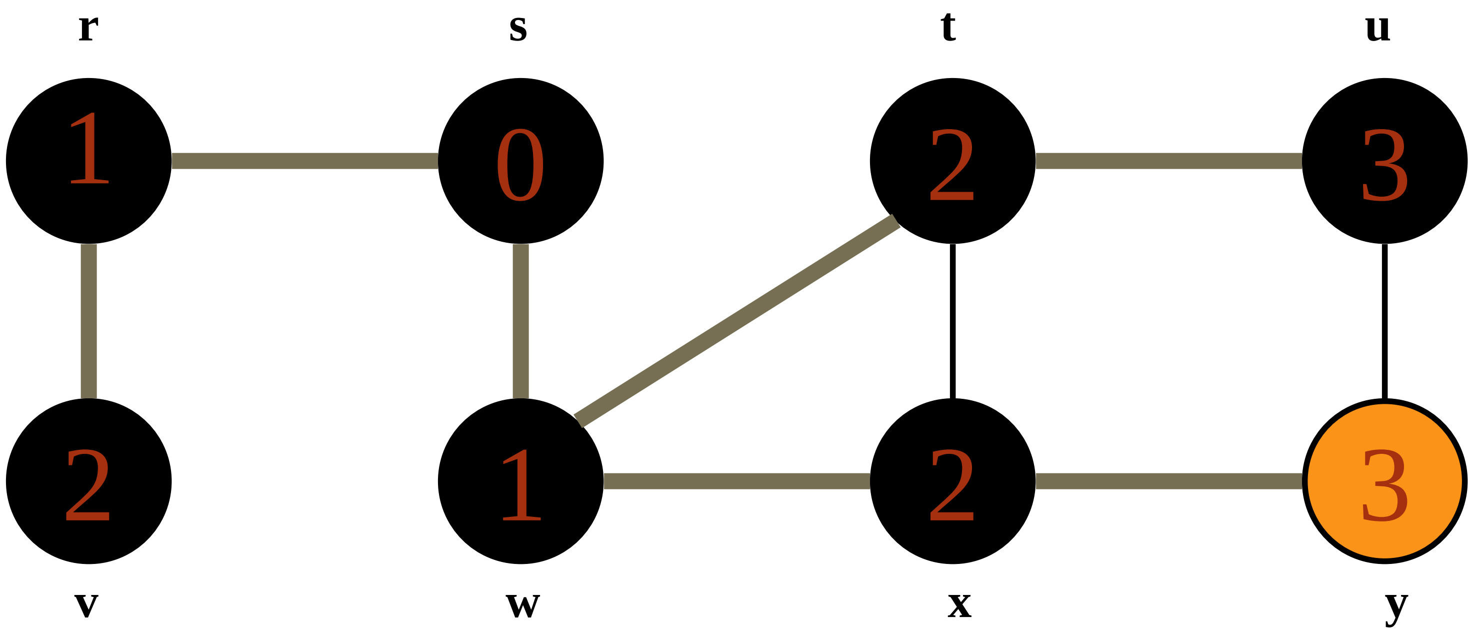


Breadth-First Search: Example



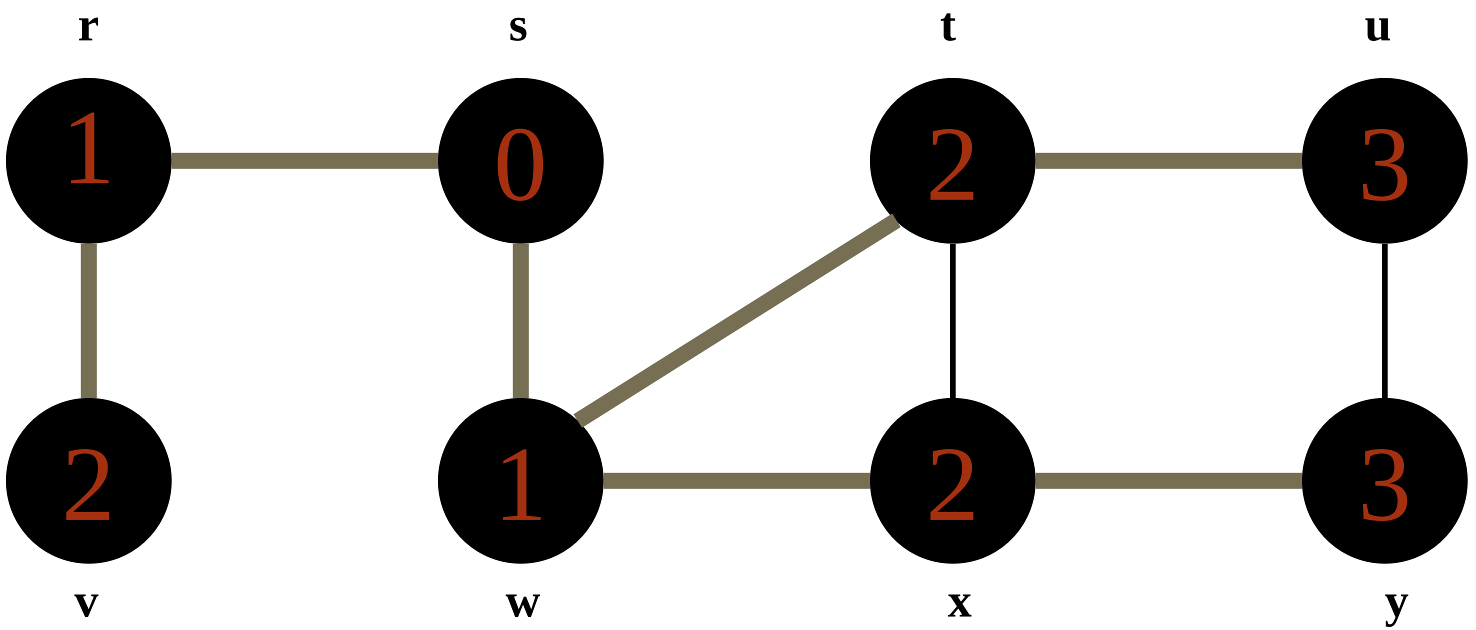
Q: | u | y |

Breadth-First Search: Example



Q: y

Breadth-First Search: Example



Q: Ø

BFS: The Code Again

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = RemoveTop(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;
    }
}
```

Touch every vertex: O(V)

u = every vertex, but only once (Why?)

So v = every vertex that appears in some other vert's adjacency list

What will be the running time?

Total running time: O(V+E)

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
  - Proof given in the book (p. 472-5)
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

Depth-First Search

- *Depth-first search* is another strategy for exploring a graph
  - 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

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?

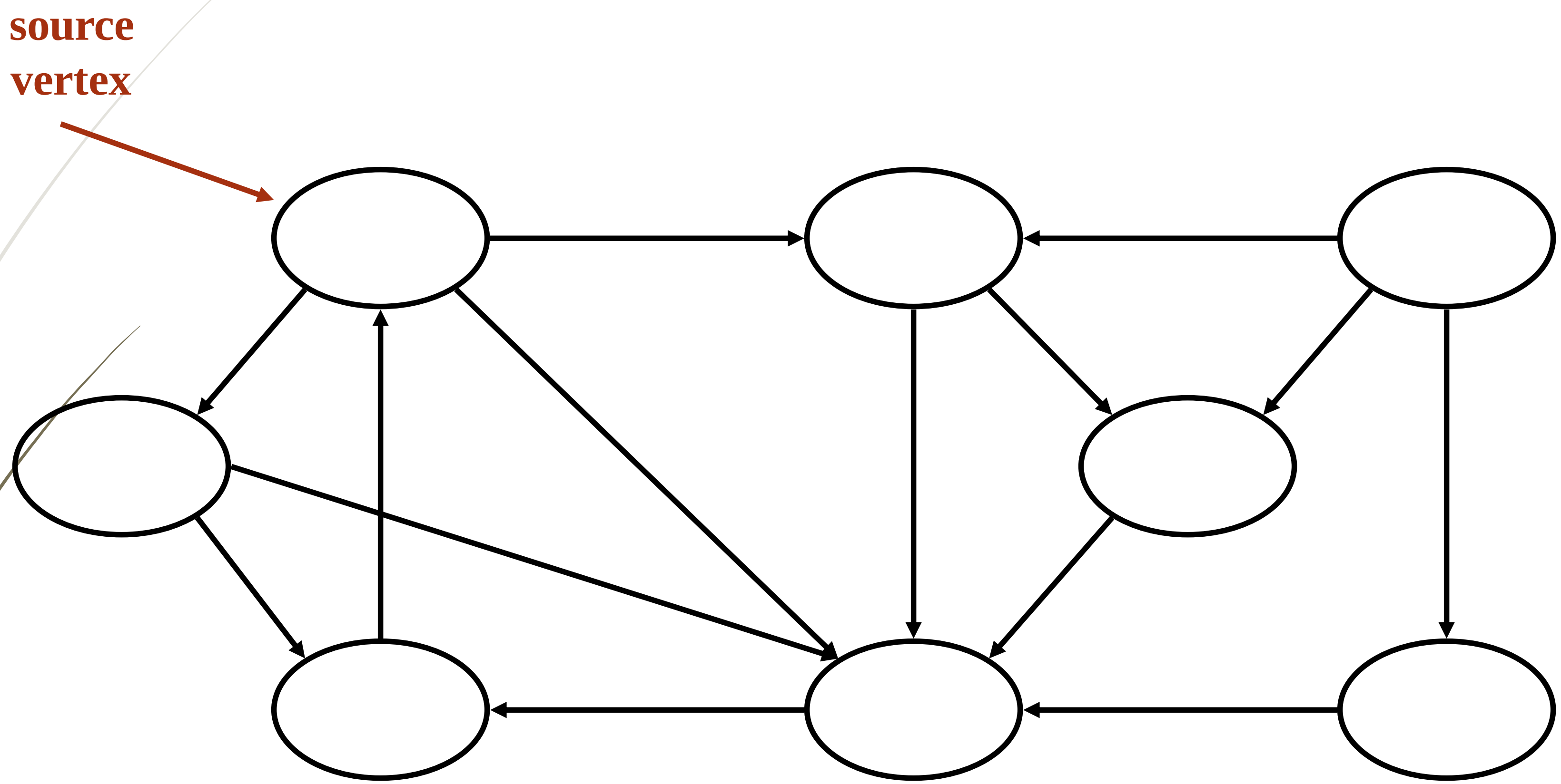
Depth-First Search: The Code

```
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);
   }
}
```

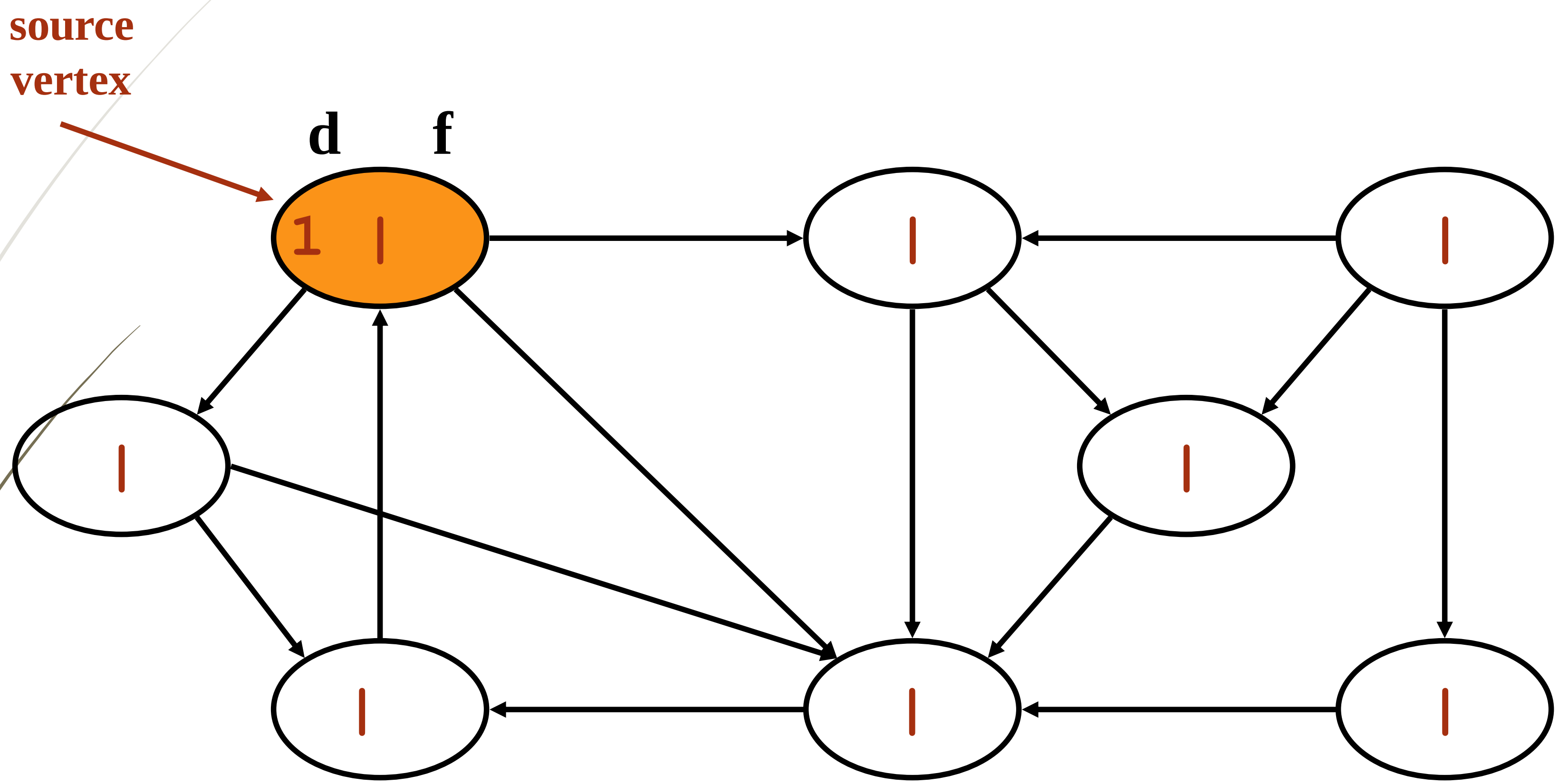
```
DFS_Visit(u)
{
   u->color = GREY;
   time = time+1;
   u->d = time;
   for each v ∈ u->Adj[]
   {
      if (v->color == WHITE)
         DFS_Visit(v);
   }
   u->color = BLACK;
   time = time+1;
   u->f = time;
}
```

So, running time of DFS = O(V+E)

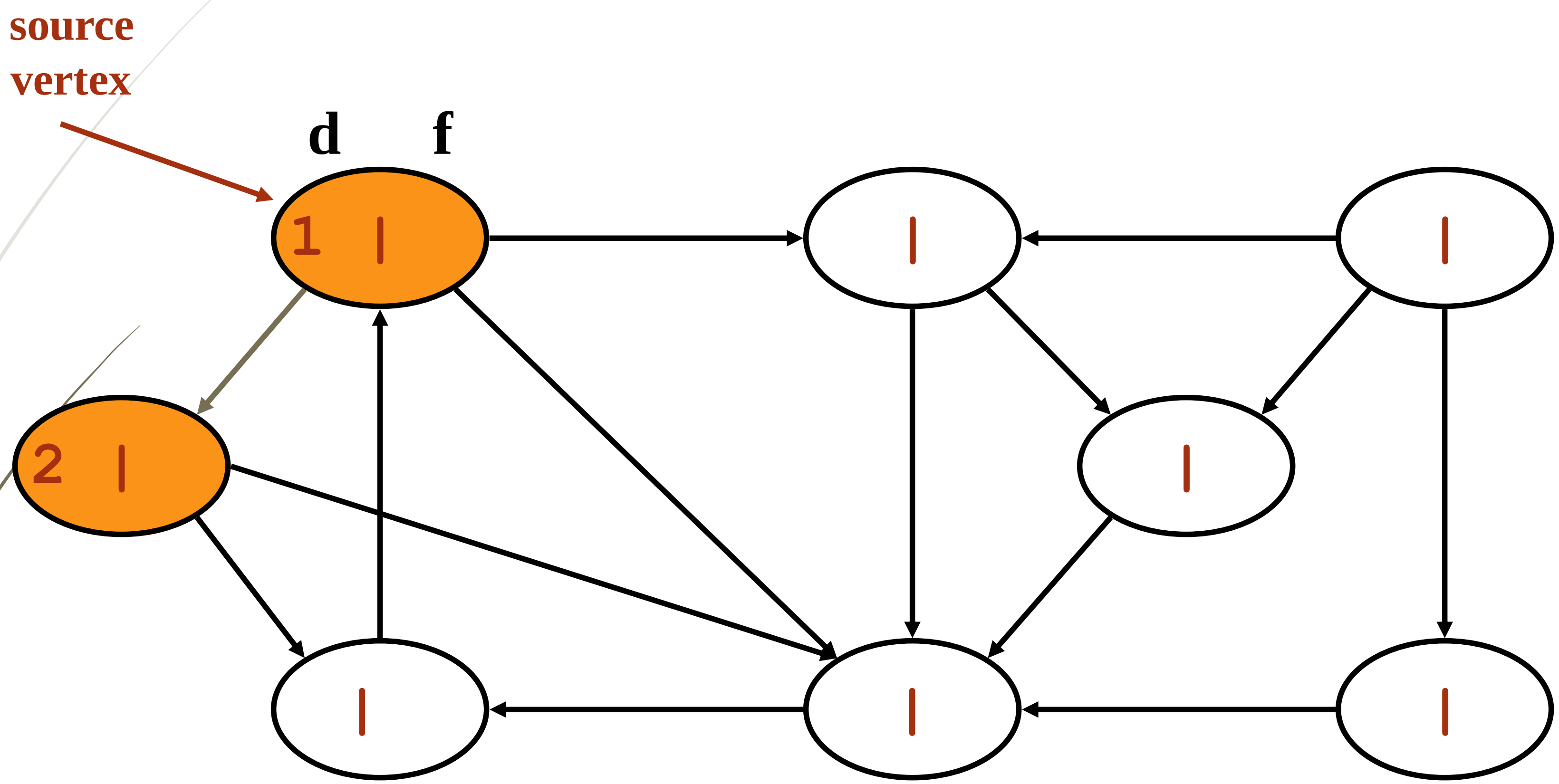
DFS Example



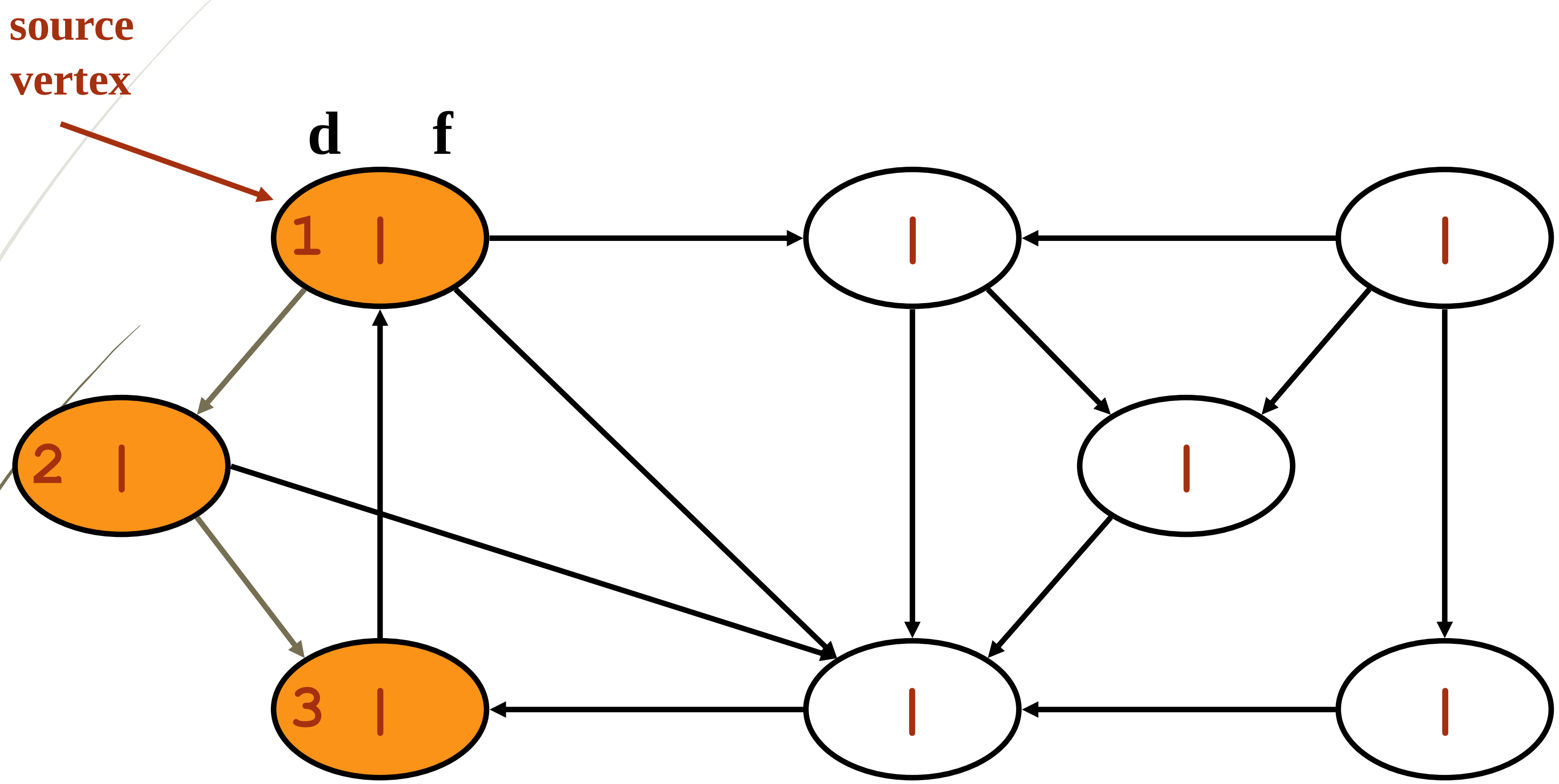
DFS Example



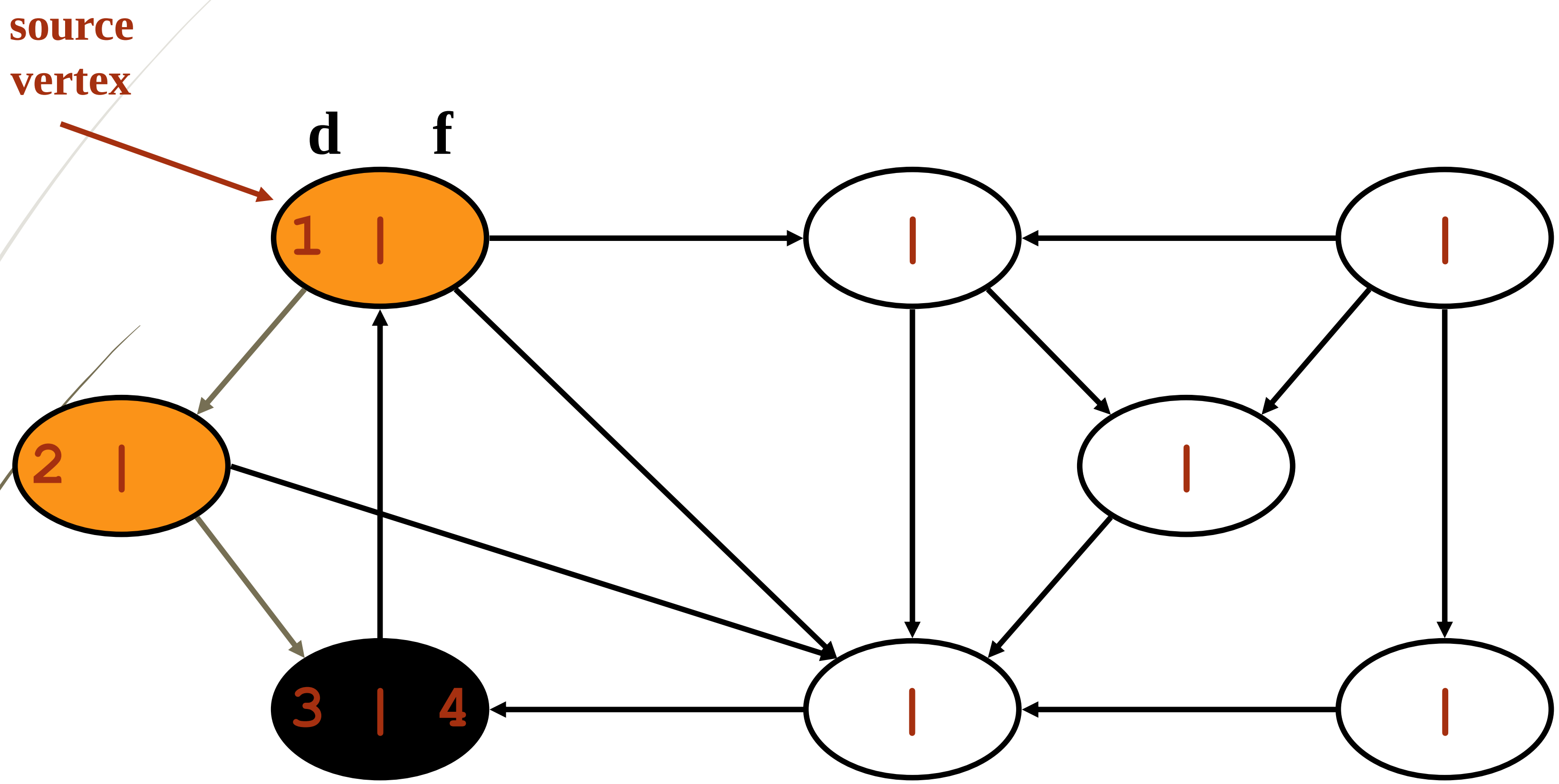
DFS Example



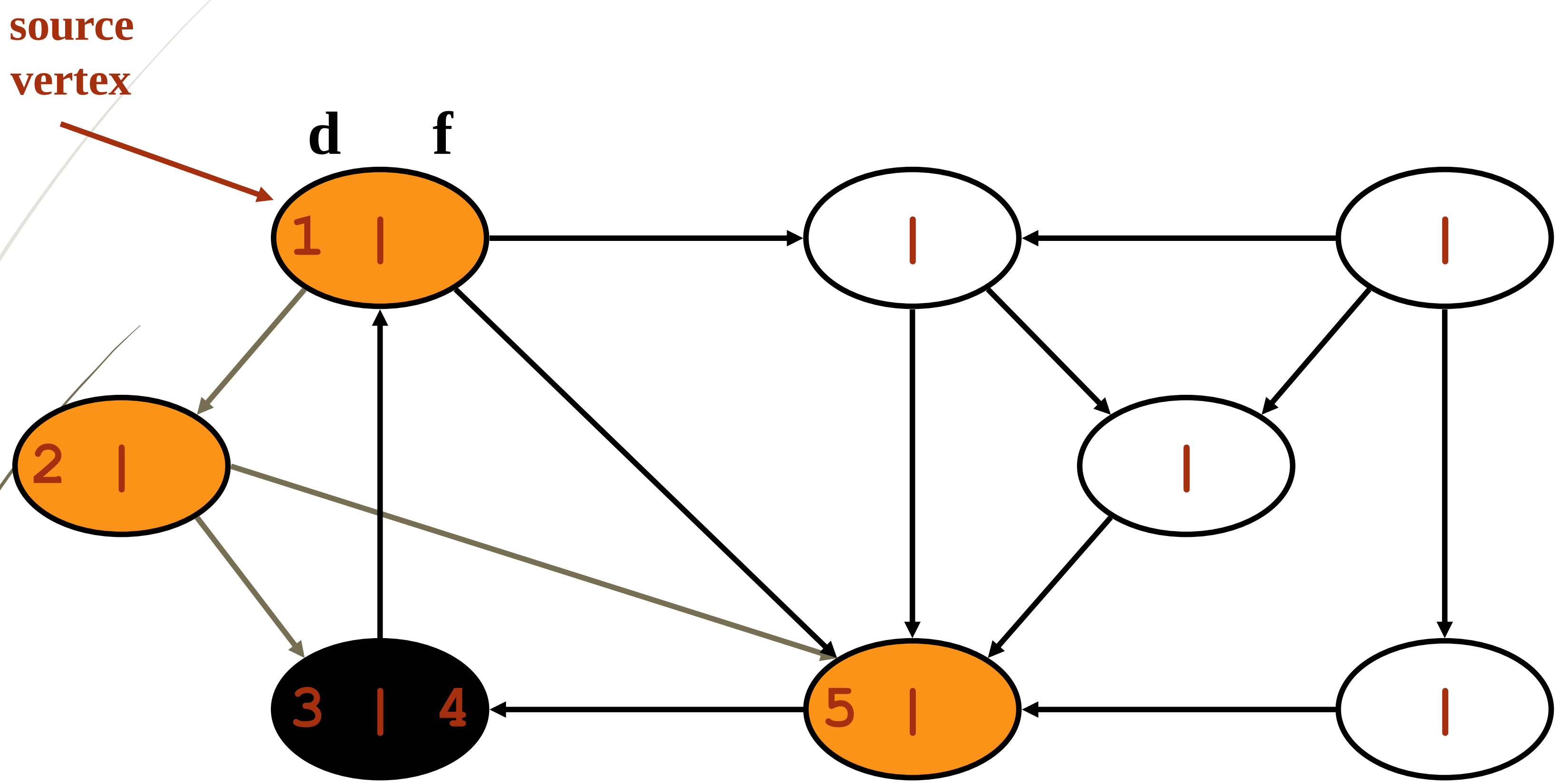
DFS Example



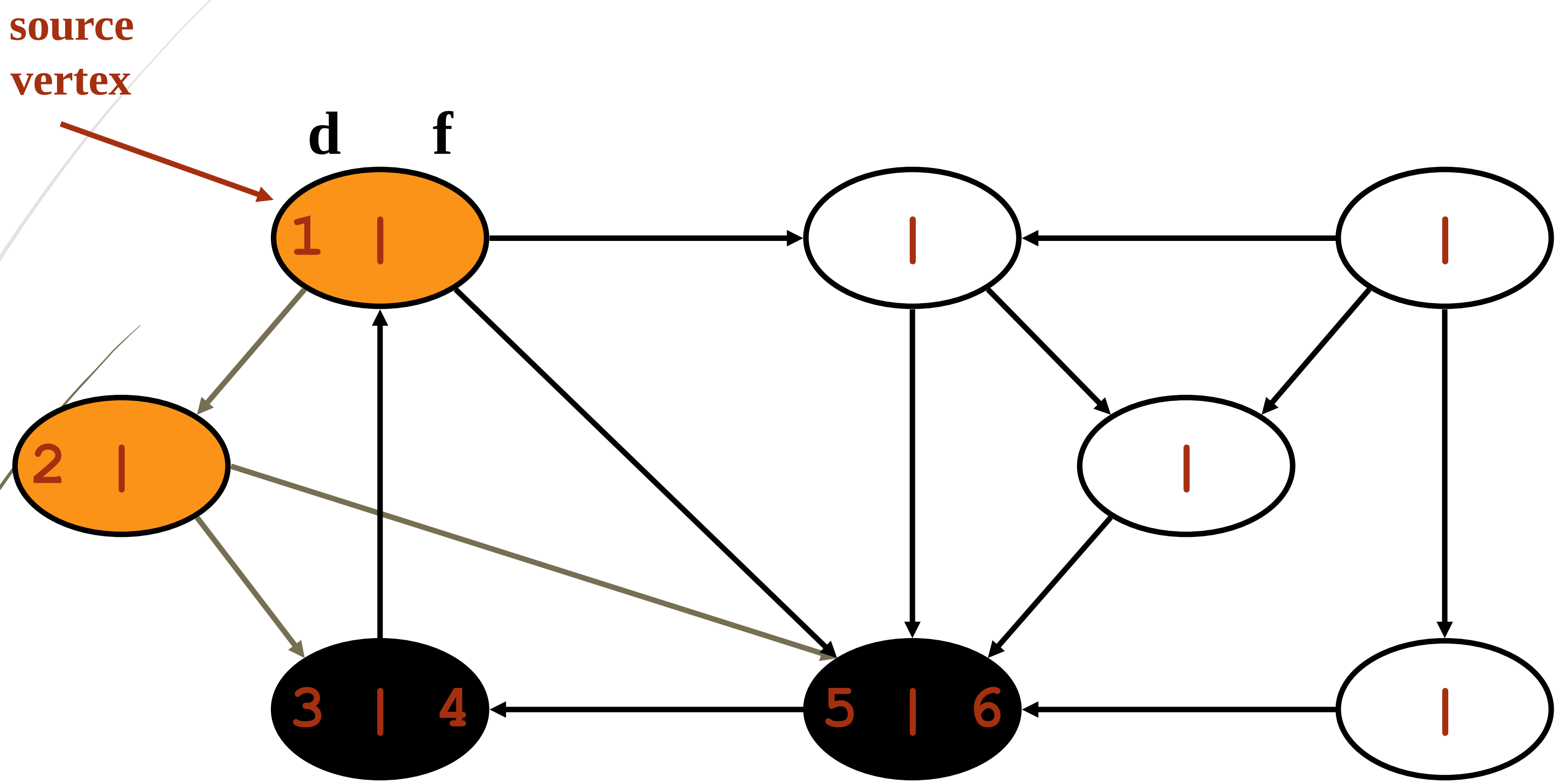
DFS Example



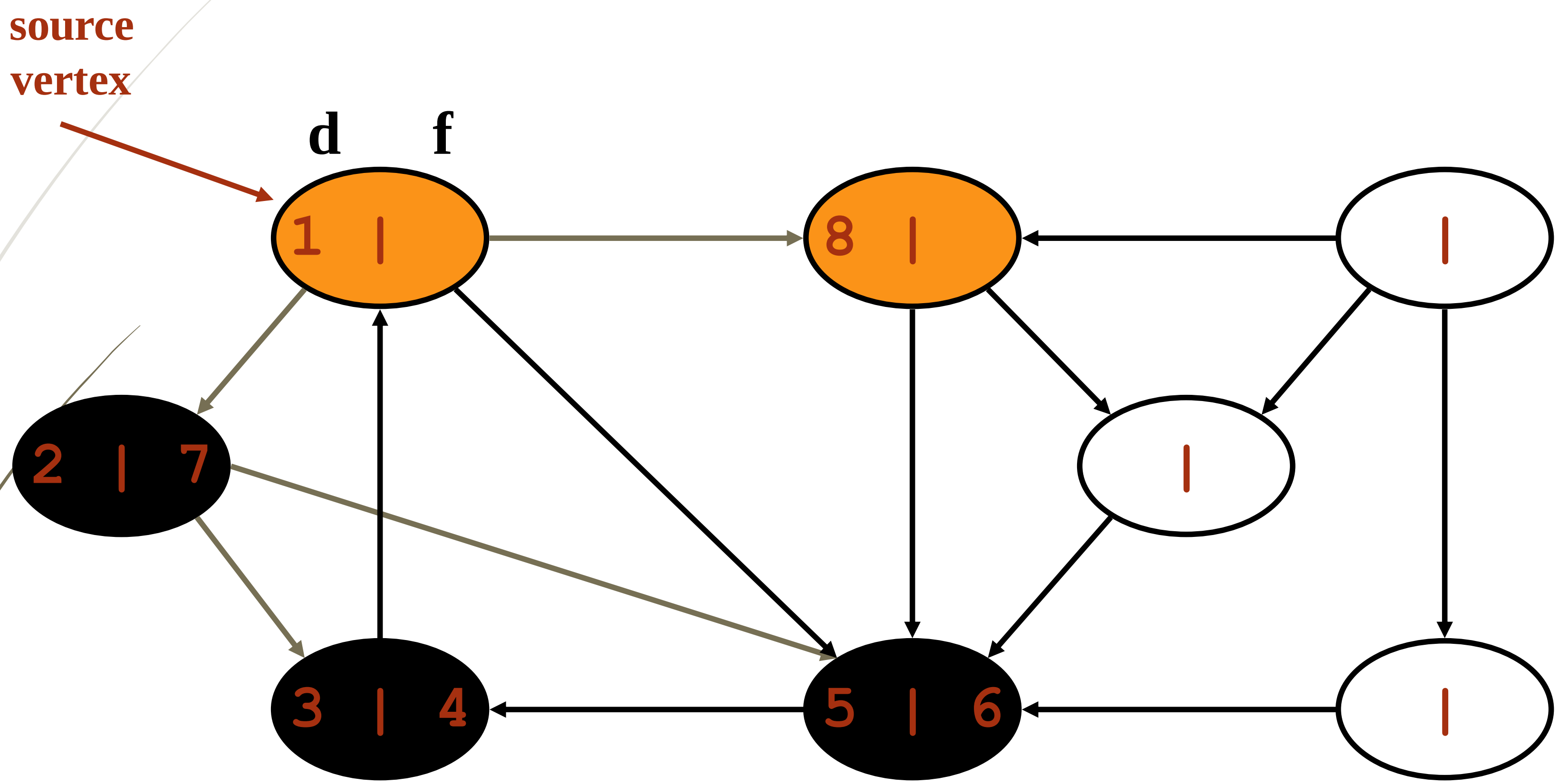
DFS Example



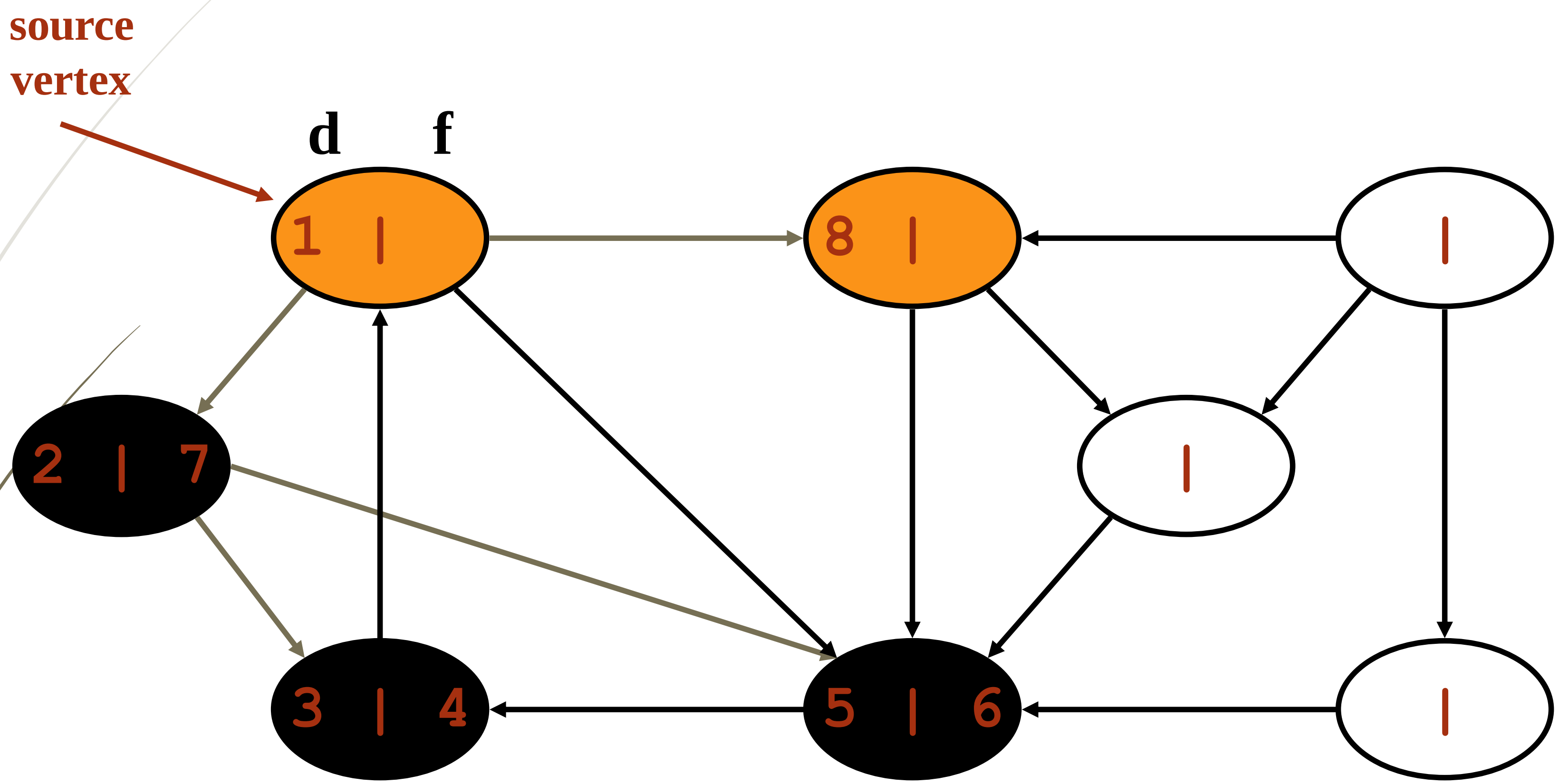
DFS Example



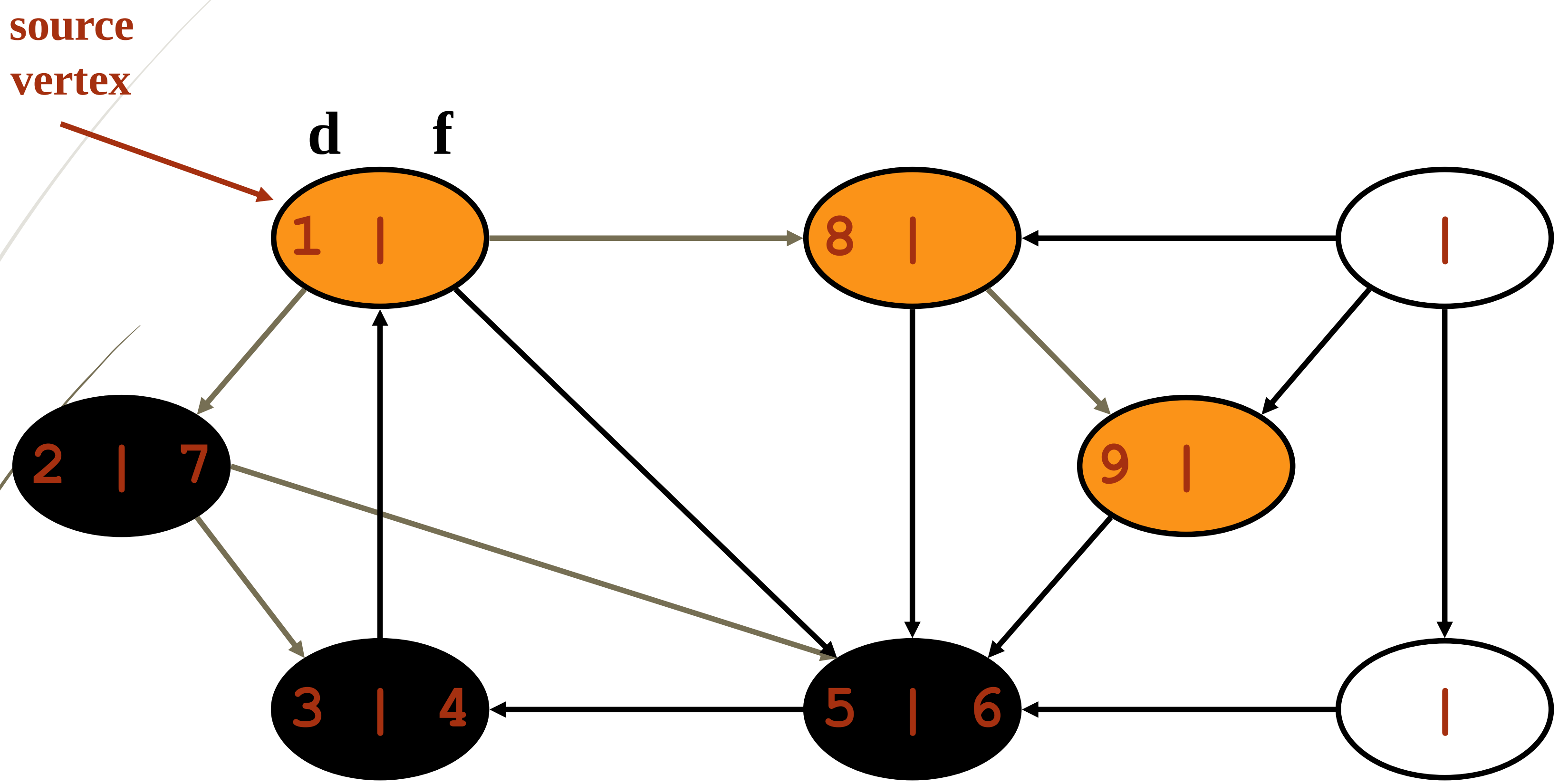
DFS Example



DFS Example

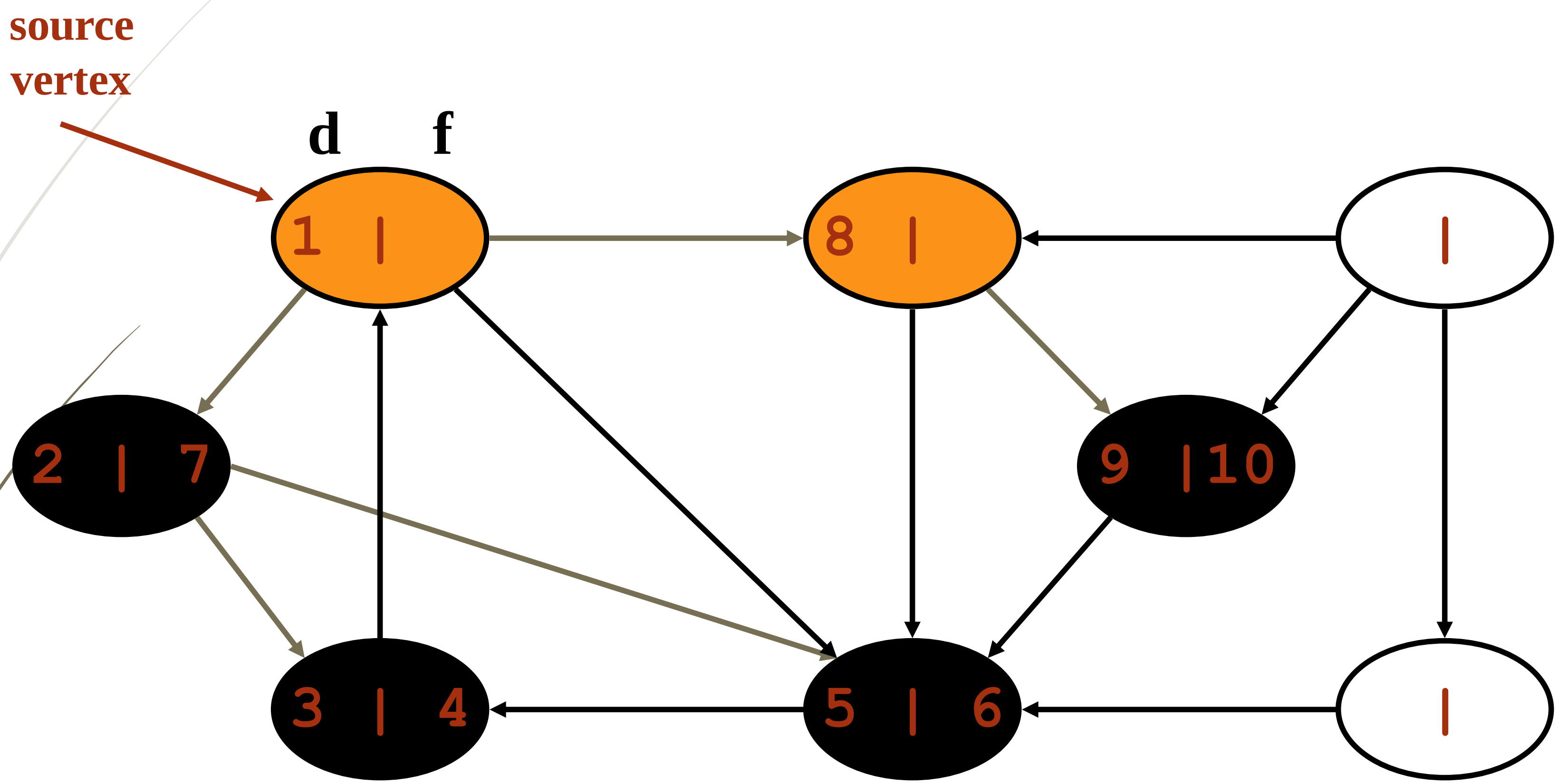


DFS Example

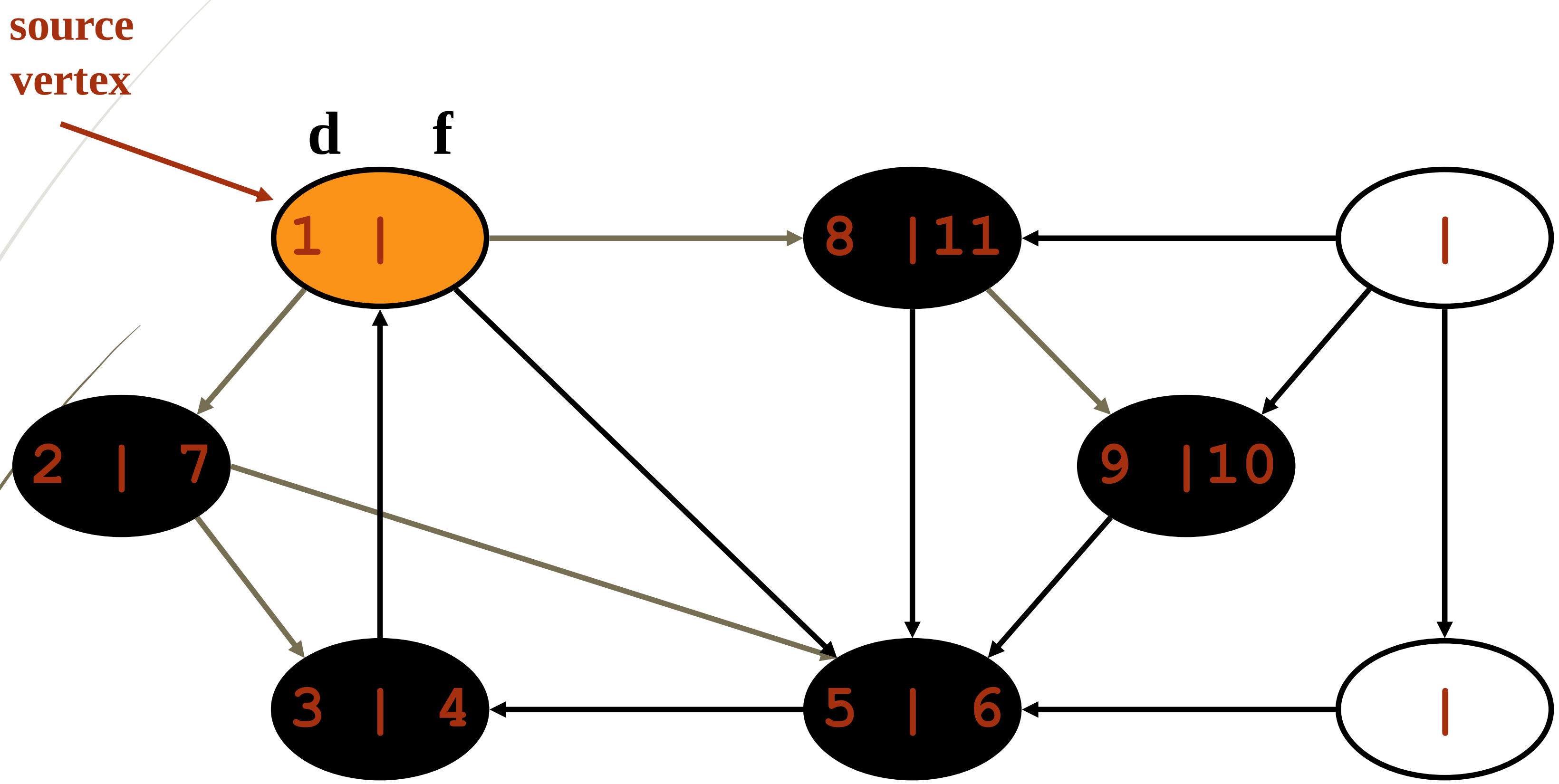


What is the structure of the grey vertices?
What do they represent?

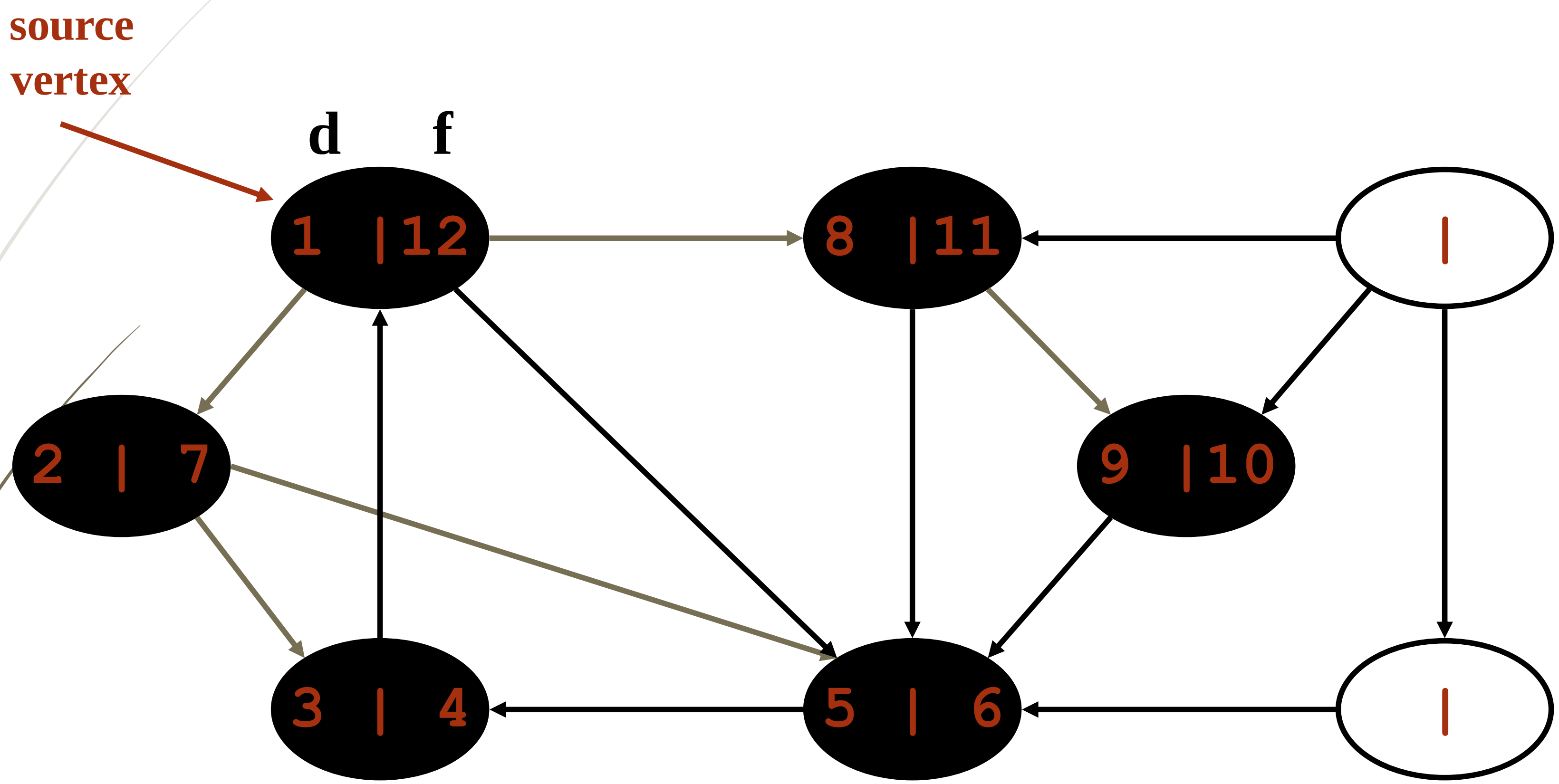
DFS Example



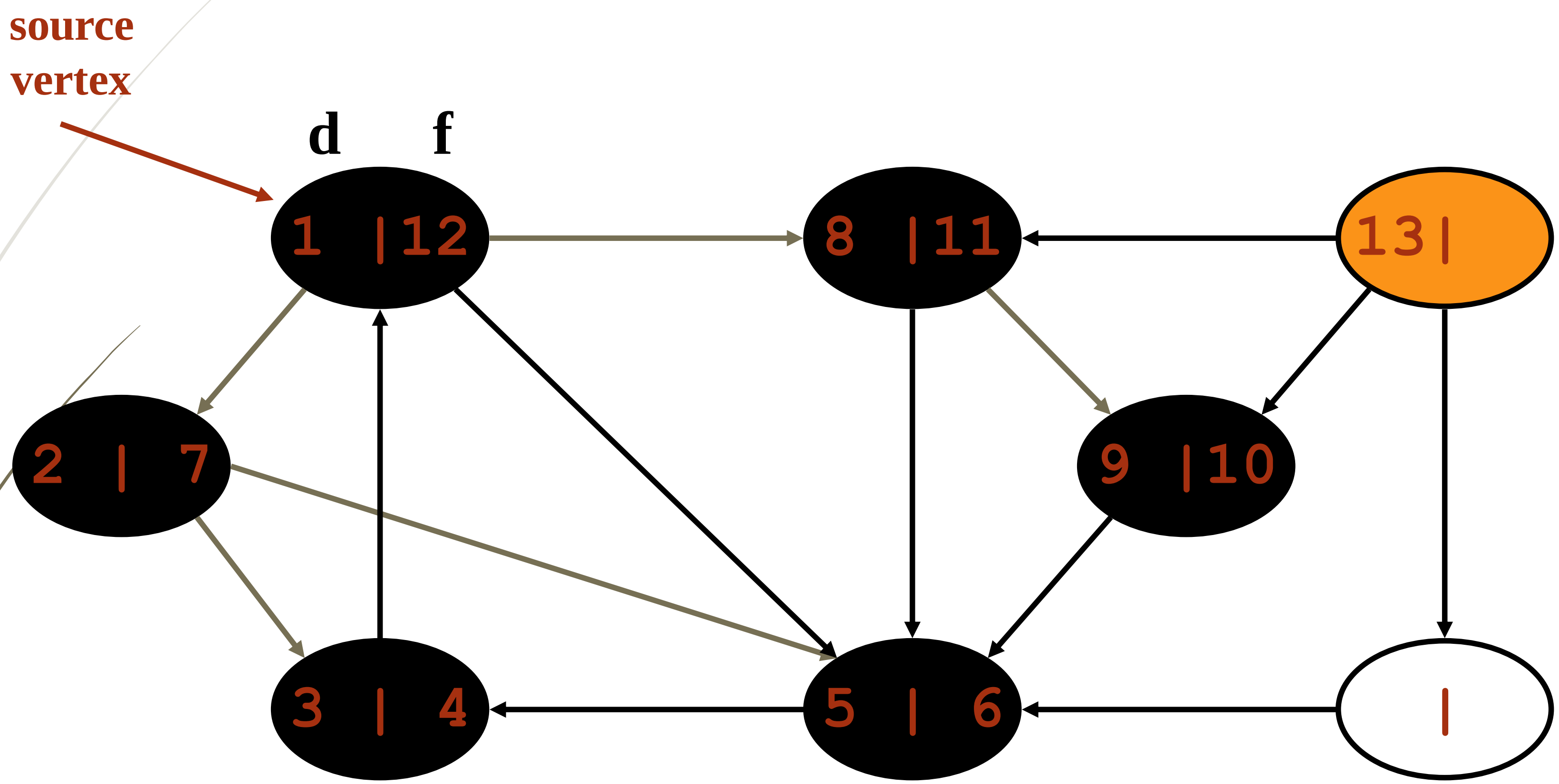
DFS Example



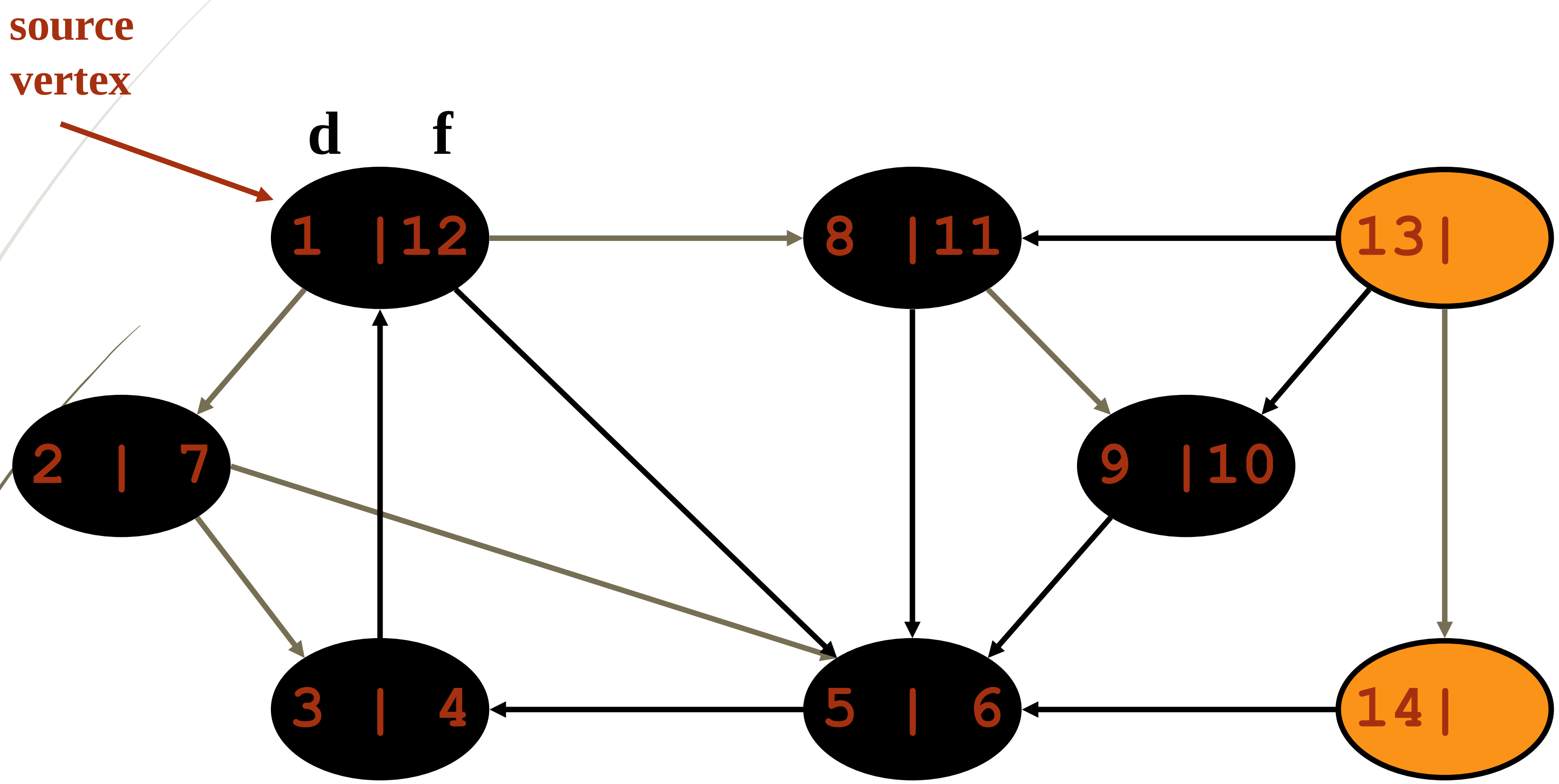
DFS Example



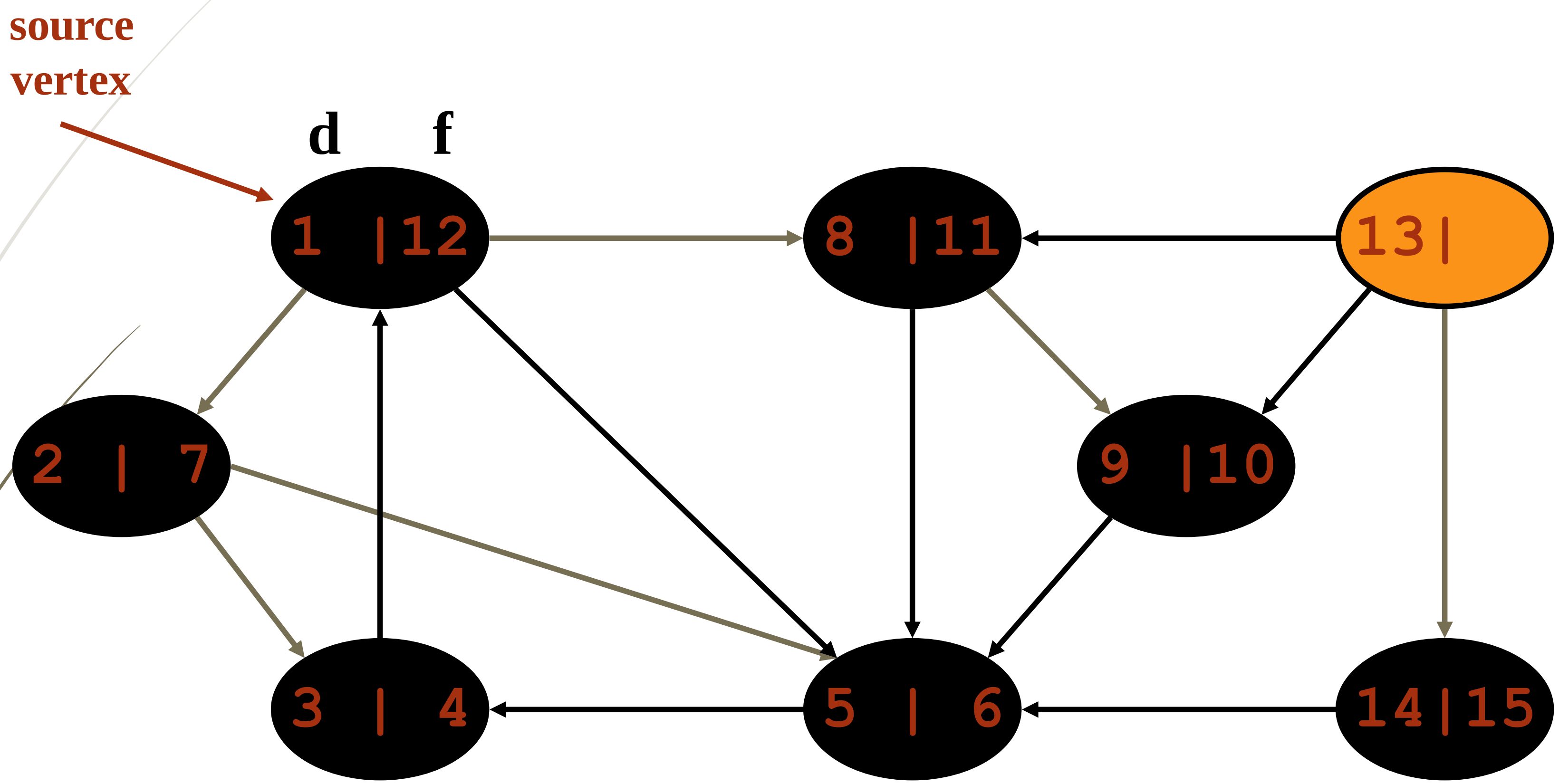
DFS Example



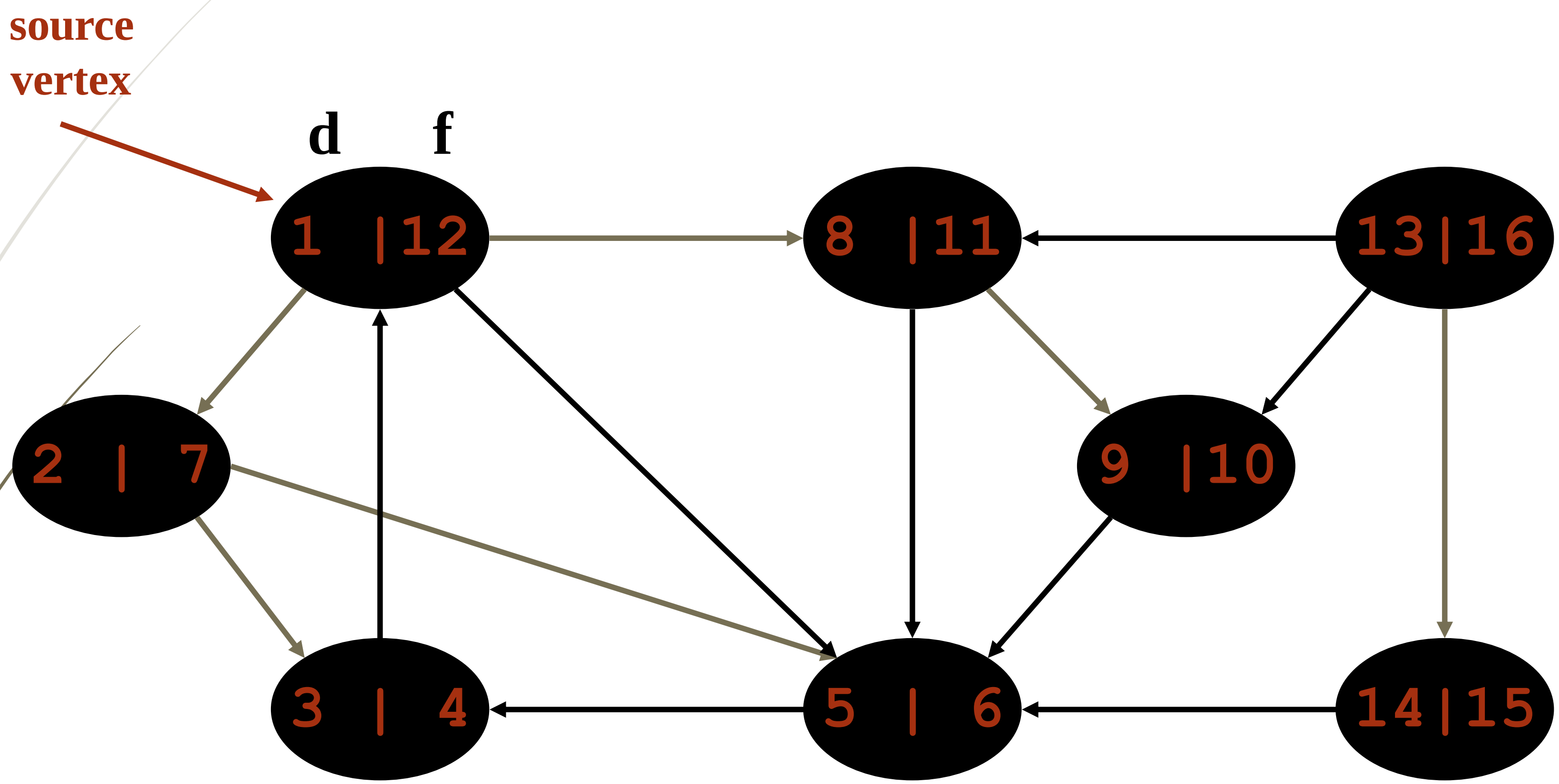
DFS Example



DFS Example



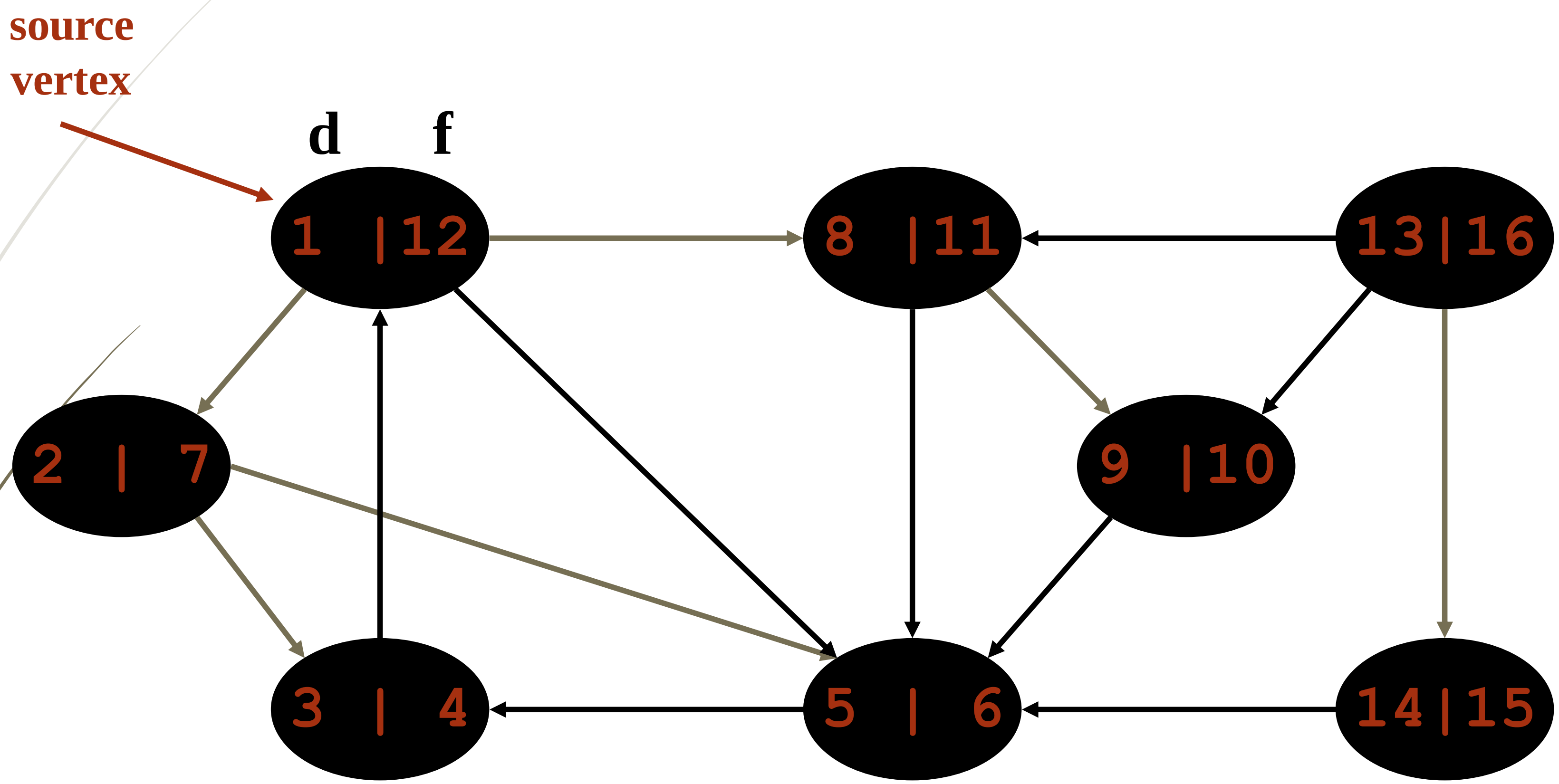
DFS Example



DFS: Kinds of edges

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

DFS Example

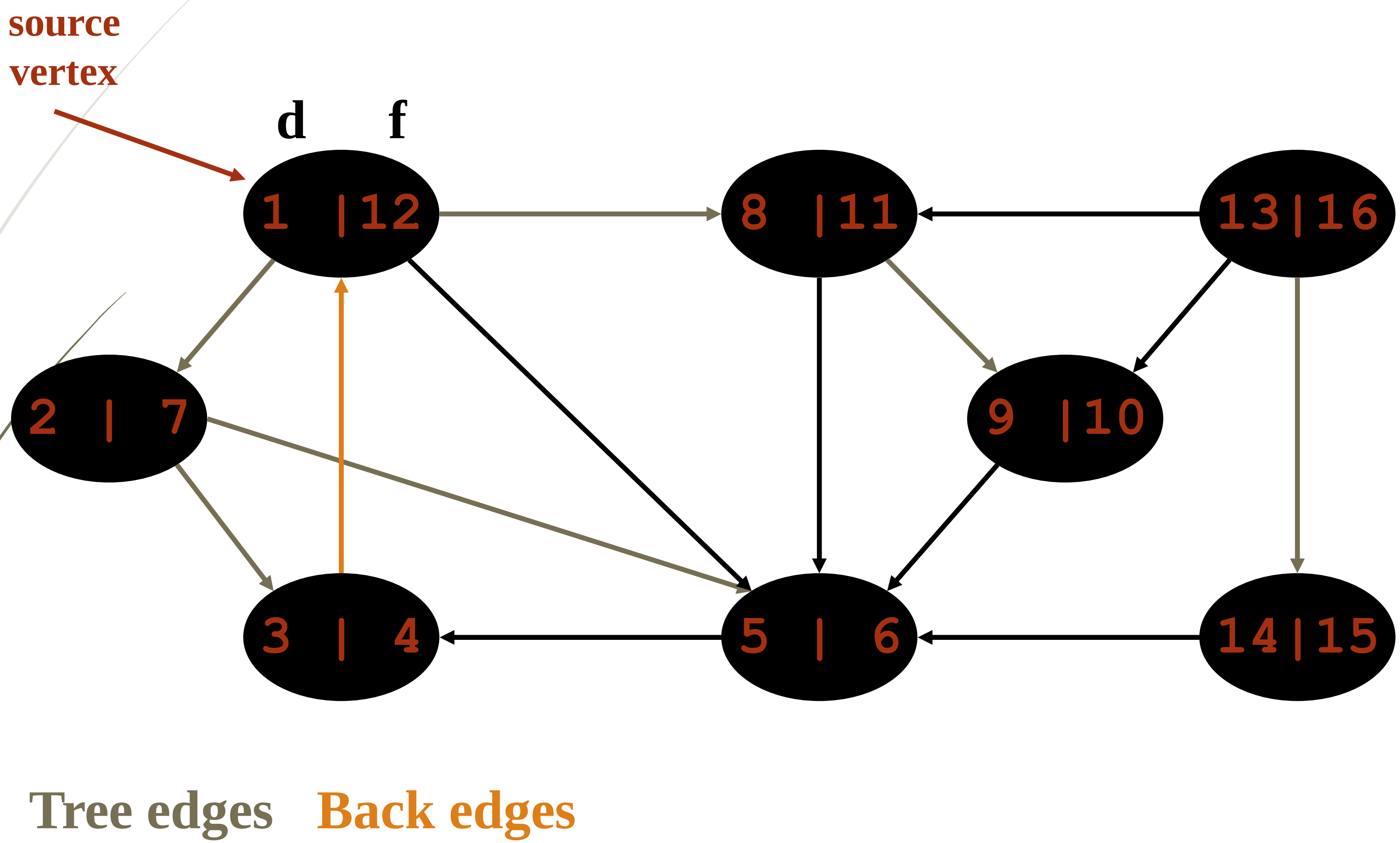


Tree edges

DFS: Kinds of edges

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

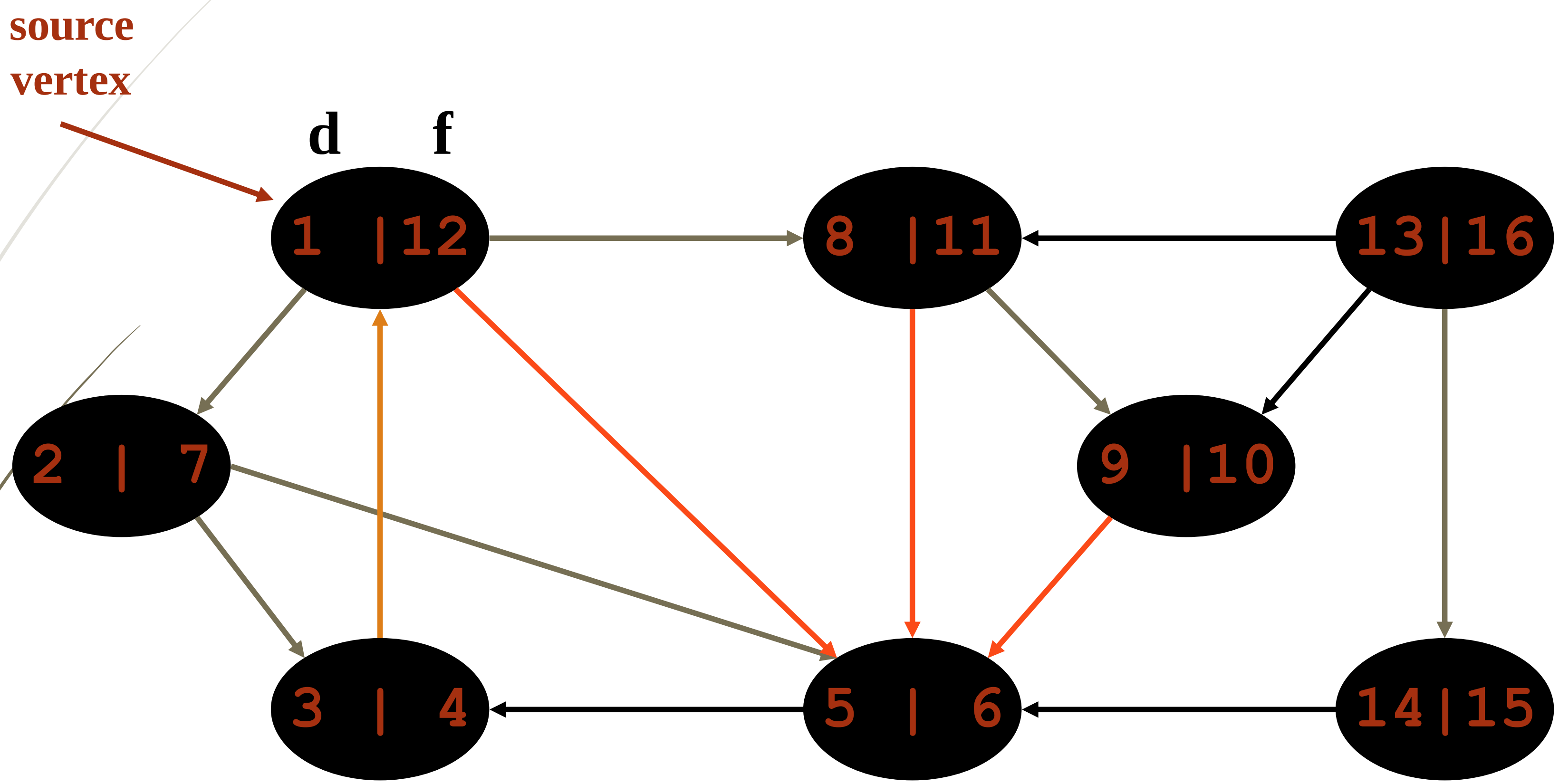
DFS Example


source
vertex
d
f
1 |12
8 |11
13|16
2 | 7
9 |10
3 | 4
5 | 6
14|15
Tree edges
Back edges

DFS: Kinds of edges

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

DFS Example

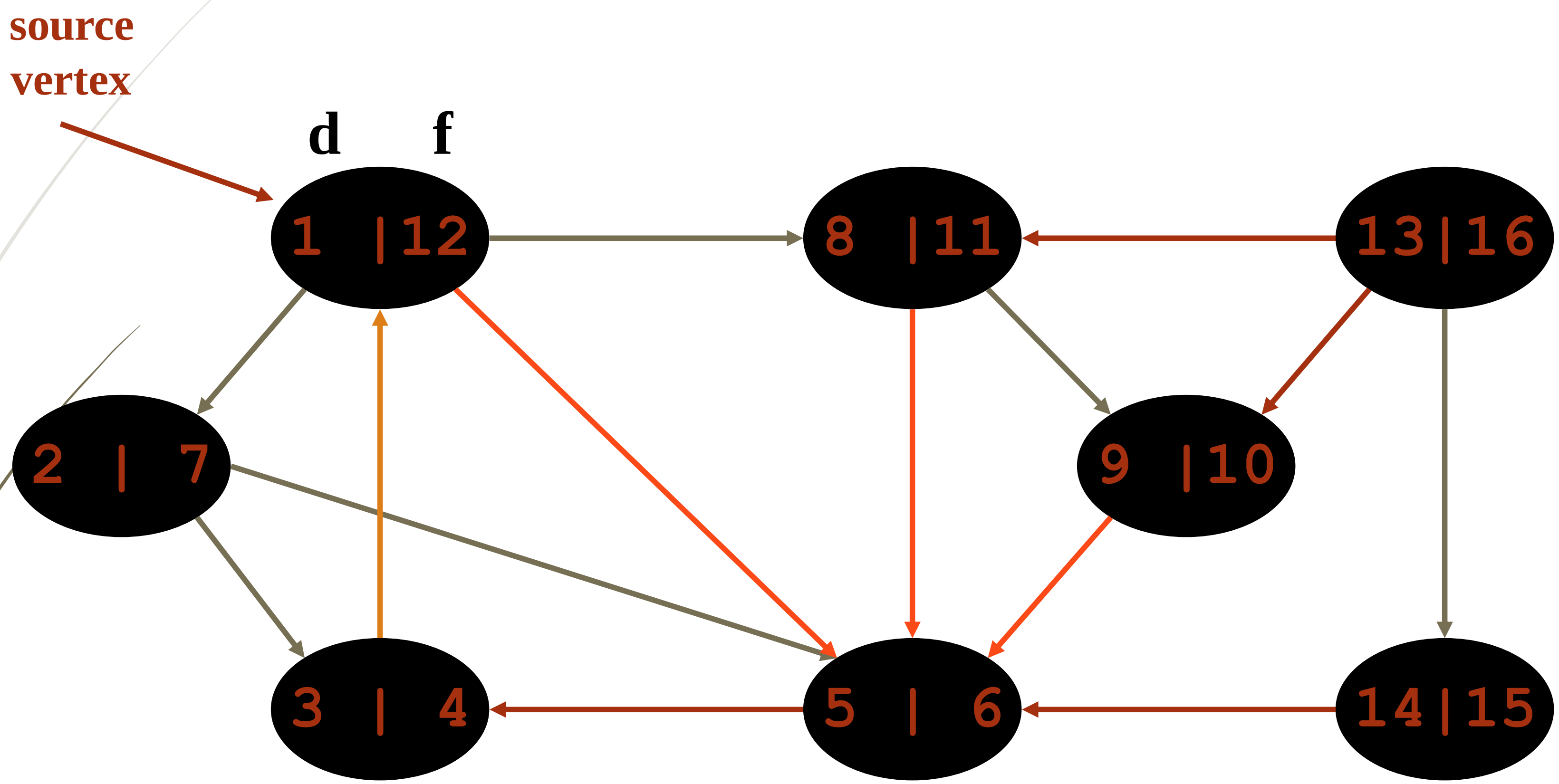


Tree edges Back edges Forward edges

DFS: Kinds of edges

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

DFS Example



Tree edges Back edges Forward edges Cross 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
  - *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 And Graph Cycles

- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
  - If acyclic, no back edges (because a back edge implies a cycle
  - If no back edges, acyclic
    - No back edges implies only tree edges (*Why?*)
    - Only tree edges implies we have a tree or a forest
    - Which by definition is acyclic
- Thus, can run DFS to find whether a graph has a cycle

DFS And Cycles

```
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;
}
```

- *How would you modify the code to detect cycles?*

DFS And Cycles

```
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;
}
```

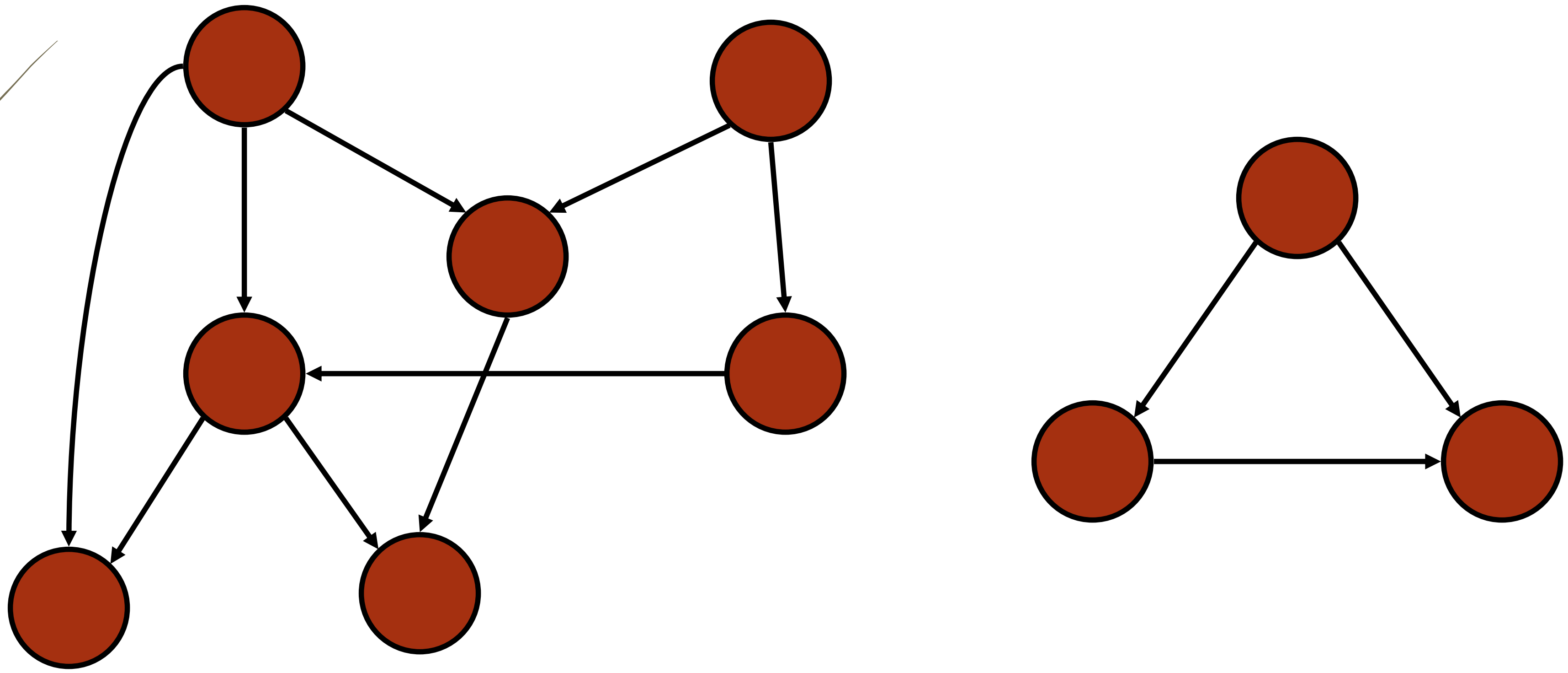
- *What will be the running time?*

DFS And Cycles

- *What will be the running time?*
- A: O(V+E)
- We can actually determine if cycles exist in O(V) time:
  - In an undirected acyclic forest, $|E| \leq |V| - 1$
  - So count the edges: if ever see $|V|$ distinct edges, must have seen a back edge along the way

Directed Acyclic Graphs

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



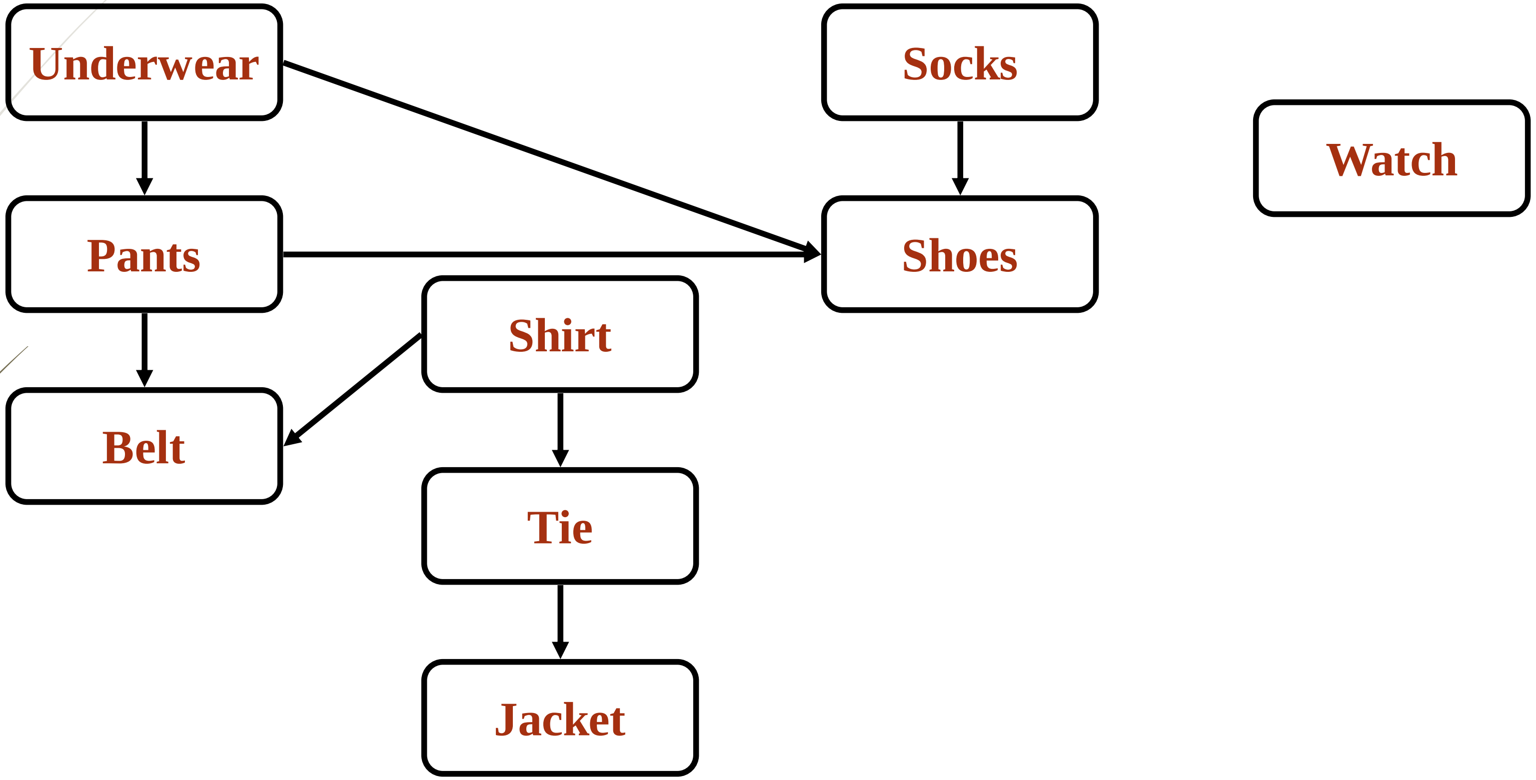
DFS and DAGs

- Argue that a directed graph G is acyclic iff a DFS of G yields no back edges:
  - Forward: if G is acyclic, will be no back edges
    - Trivial: a back edge implies a cycle
  - Backward: if no back edges, G is acyclic
    - Argue contrapositive: G has 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 discovered, whole cycle is white
      - Must visit everything reachable from v before returning from DFS-Visit()
      - So path from u$\rightarrow$v is yellow$\rightarrow$yellow, thus (*u*, *v*) is a back edge

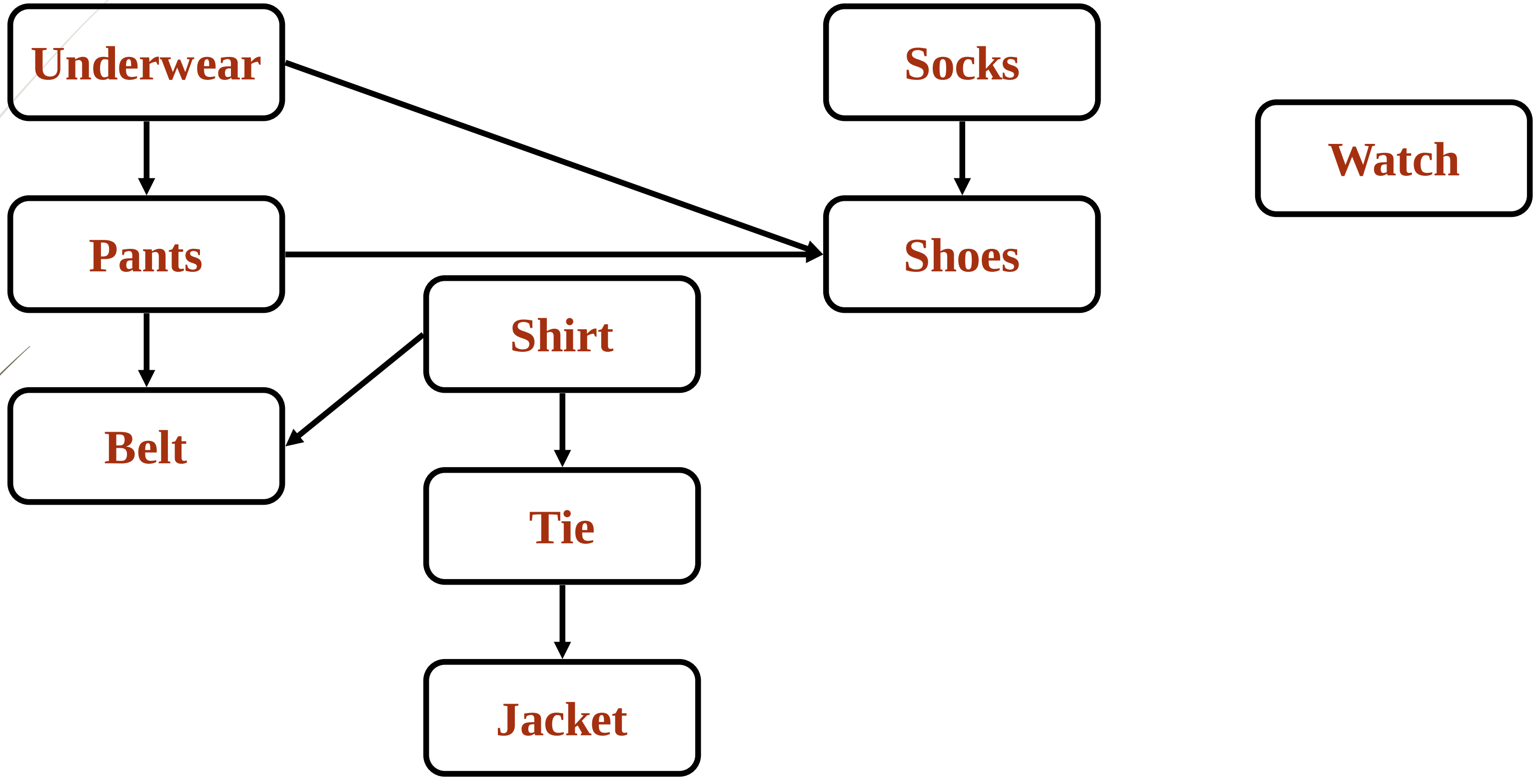
Topological Sort

- *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

Getting Dressed



Getting Dressed



Topological Sort Algorithm

```
Topological-Sort()
{
   Run DFS
   When a vertex is finished, output it
   Vertices are output in reverse topological order
}
```

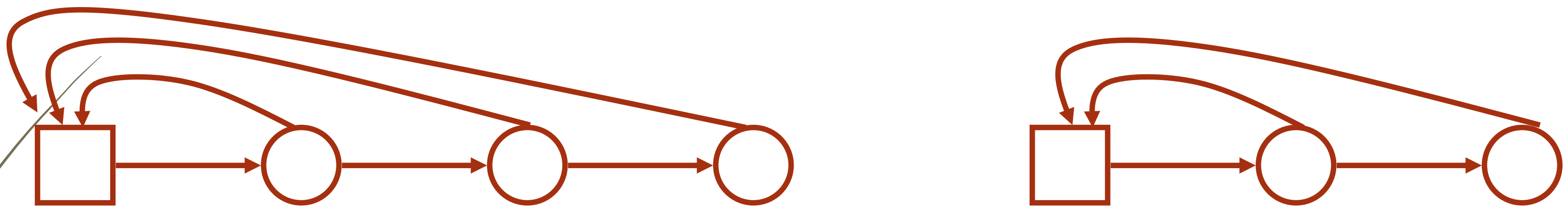
- Time: O(V+E)
- Correctness: Want to prove that
  $(u,v) \in G \Rightarrow u{\rightarrow}f > v{\rightarrow}f$

Disjoint-Set Union Problem

- Want a data structure to support disjoint sets
  - Collection of disjoint sets $S = \{S_i\}$, $S_i \cap S_j = \varnothing$
- Need to support following operations:
  - MakeSet(x): $S = S \mathbf{U} \{\{x\}\}$
  - Union(S_i, S_j): $S = S - \{S_i, S_j\} \mathbf{U} \{S_i \mathbf{U} S_j\}$
  - FindSet(X): return $S_i \in S$ such that $x \in S_i$

Disjoint Set Union

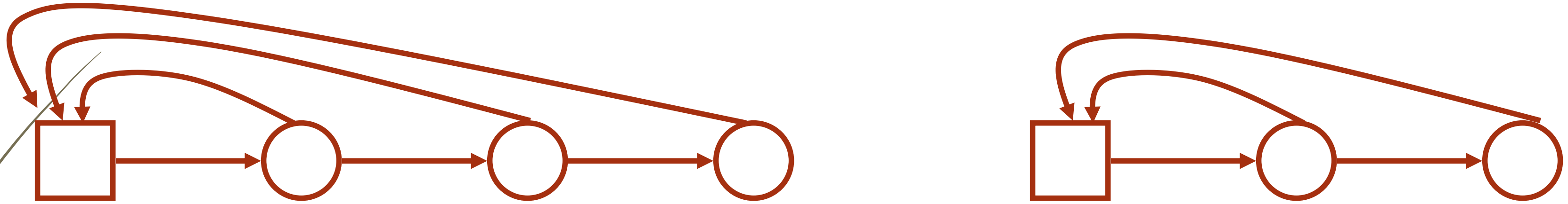
- So how do we implement disjoint-set union?
  - Naïve implementation: use a linked list to represent each set:



- MakeSet(): time
- FindSet(): time
- Union(A,B): “copy” elements of A into B: time

Disjoint Set Union

- So how do we implement disjoint-set union?
  - Naïve implementation: use a linked list to represent each set:



    - MakeSet(): O(1) time
    - FindSet(): O(1) time
    - Union(A,B): "copy" elements of A into B: O(A) time
  - *How long can a single Union() take?*
  - *How long will n Union()'s take?*

Disjoint Set Union

ساختمان داده‌ها برای مجموعه‌های جدا از هم : Data structure for disjoint sets

بررسی ساختمان داده‌های مختلف (لیست پیوندی - ساختمان درختی) برای نمایش

مجموعه‌های متمایز (مجموعه‌هایی با اشتراک تهی)

برای هر مجموعه یکی از عناصر آن به عنوان شاخص (Representive) انتخاب می‌شود به طوری که هر مجموعه با

شاخص آن نامگذاری (مشخص) می‌شود

عملگرها :

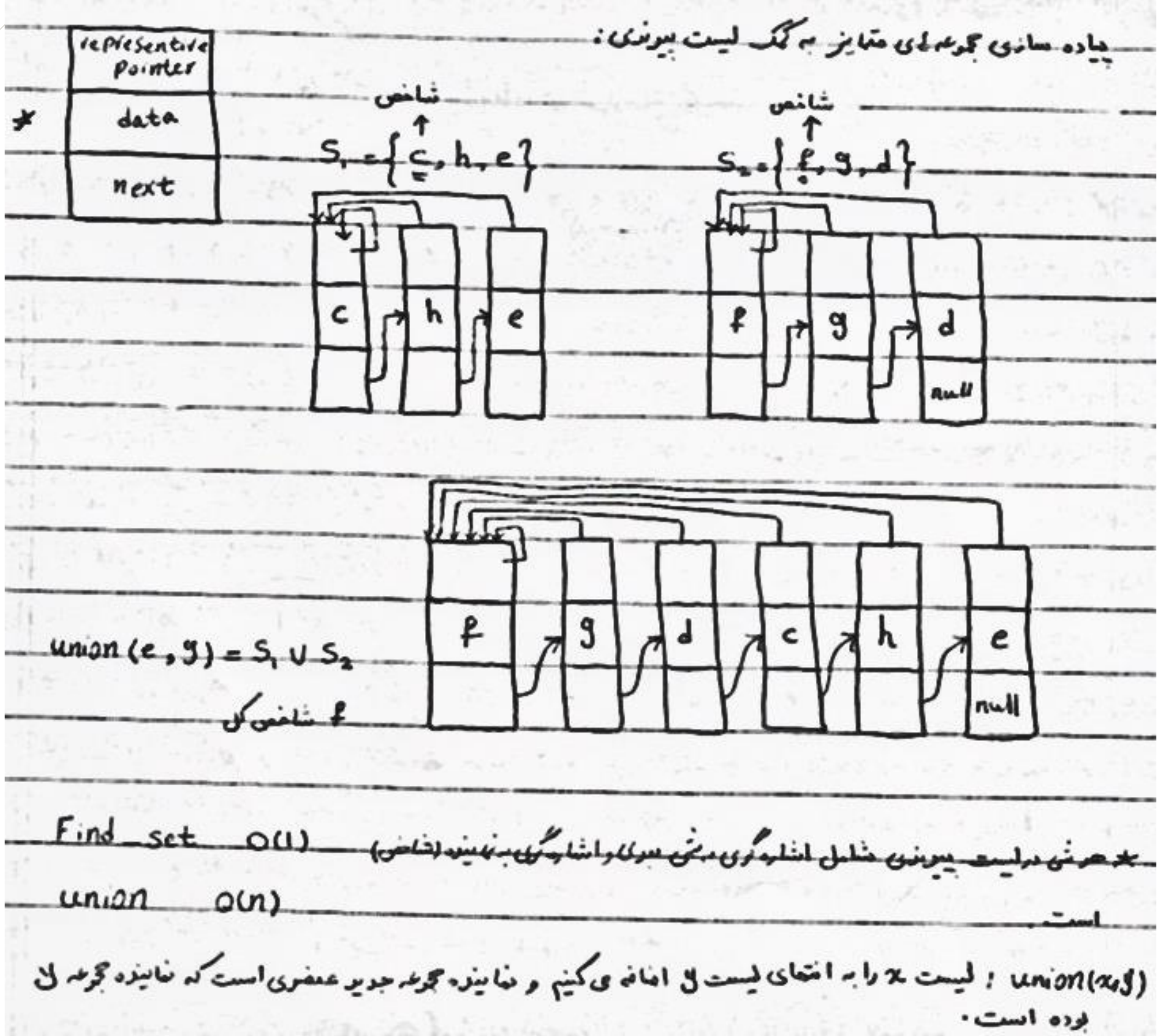
$MakeSet(x)$: یک مجموعه شامل فقط عنصر x تشکیل می‌دهد (x نباید در مجموعه دیگری باشد)

$union(x,y)$: اجتماع مجموعه‌های x (S_x) و y (S_y) ؛ شاخص برای مجموعه $S_x \cup S_y$ یکی از شاخص‌های

S_x یا S_y می‌باشد }

$Find\text{-}Set(x)$: شاخص مجموعه x را بر می‌گرداند —— $O(1)$ (چون فقط اشاره‌گری از خود به شاخص را بر می‌گردانیم)

Disjoint Set Union



Disjoint Set Union

عملیات operation	تعداد اشیای به روز شده number of object updated
$make\_set(x_1)$	1
$make\_set(x_2)$	1
$\vdots$	$\vdots$
$make\_set(x_n)$	1
$union(x_1, x_2)$	1 → 1 لینک تغییر می‌کند
$union(x_2, x_3)$	2 → 2 لینک تغییر می‌کند
$\vdots$	$\vdots$
$union(x_{n-1}, x_n)$	$n-1$

$$\text{تعداد بروز رسانی} = n + \frac{(n-1)(n)}{2}$$

$$\text{تعداد عملیات} = \underbrace{n}_{\sum make\_set} + \underbrace{(n-1)}_{\sum union} = 2n-1$$

$$union \text{ میانگین تعداد بروز رسانی در هر} = \frac{n + \frac{n(n-1)}{2}}{2n-1} = O(n)$$

Disjoint Set Union

پیاده سازی مجموعه های متمایز به کمک ساختار درختی:

۱ـ هر عنصر در یک مجموعه با یک گره نمایش داده می شود.

2ـ هر گره یک اشاره گر به Parent خود دارد. (Parent ریشه خودش است)

3ـ به هر گره یک عدد به عنوان rank اختصاص داده می شود که rank یک حد بالا برای عمق زیردرخت با ریشه گره می باشد.

عملگرها:

make-set : درختی با یک گره ایجاد می کند.

find-set : شاخص را برمی گرداند (ریشه شاخص (بابش) است)

union : ریشه یک درخت به ریشه درخت دیگر اشاره می کند.

پارامترها فقط شاخص ها است

در union استفاده میکنیم

```
make_set(x) {
    Parent[x] ← P[x] ← x
    rank[x] ← 0
}

LINK(x, y) {
    if (rank[x] > rank[y])
        P[y] ← x
    else
        P[x] ← y
        if (rank[x] == rank[y])
            rank[y]++;
}
```

Disjoint Set Union

: LINK(x,y)

```
if (rank[x] > rank[y])  →  P[y] ← x
if (rank[y] > rank[x])  →  P[x] ← y
if (rank[y] == rank[x]) →  P[x] ← y , rank[y]++
```

```
Find_Set(x){
if (x ≠ P[x])
    P[x] ← Find_Set(P[x]);
return (P[x]);
}
```

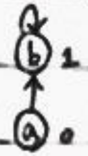
```
union(x,y){
link(find_set(x), find_set(y))
}
```

مثال:

make_set(a)   a 0

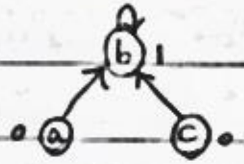
make_set(b)   b 0

union(a,b)

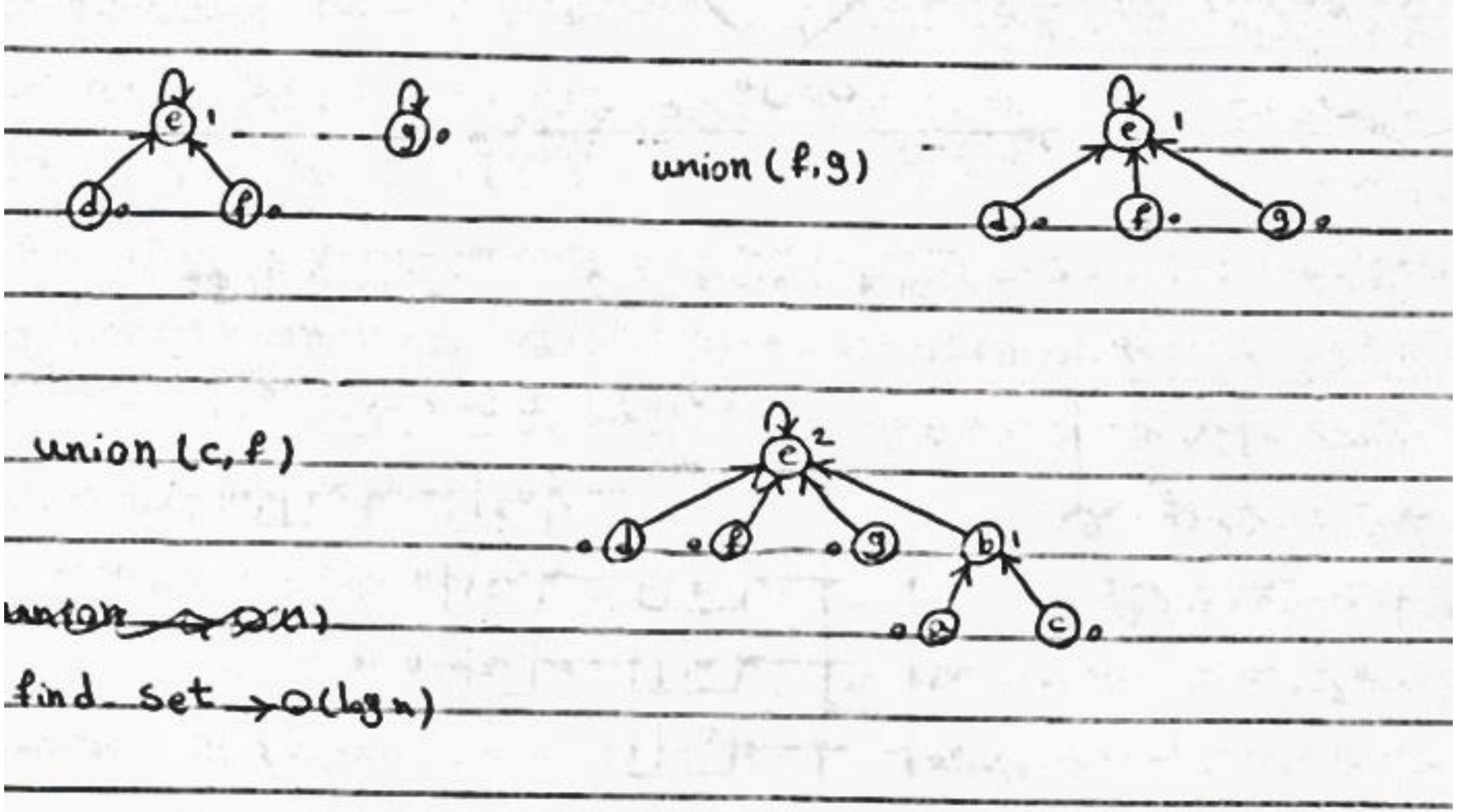


make_set(c)   c 0

union(a,c)



Disjoint Set Union



Disjoint Set Union

قضیه: اگر درخت تشکیل شده توسط عملگر union صفر، تهی دارای m گره باشد. آنگاه حداکثر دارای عمق $\lceil \log m \rceil + 1$ می‌باشد.

اثبات به روش استقرای قوی:

$$if\ (m == 1) \Rightarrow S = \{a\} \quad ⓐ \Rightarrow depth(T) = 1 \le \lceil \log 1 \rceil + 1 = 1$$

فرض می‌کنیم قضیه به ازای $2, 3, \dots, m-1$ برقرار باشد حال برای m ثابت می‌کنیم

اگر درخت T دارای m گره باشد که از union در درخت T_1 با a گره و T_2 با $m-a$ گره تشکیل شده است $0 < a \le m/2$

حال اگر $0 < a < m/2 \rightarrow m-a > a \rightarrow$ حداکثر عمق درخت T_2 بیشتر از حداکثر عمق درخت T_1 است

و لذا عمق درخت T حاصل از union به اندازه عمق درخت T_2 خواهد بود.

طبق فرض

$$depth(T) = depth(T_2) \le \lceil \log(m-a) \rceil + 1 \le \lceil \log m \rceil + 1$$

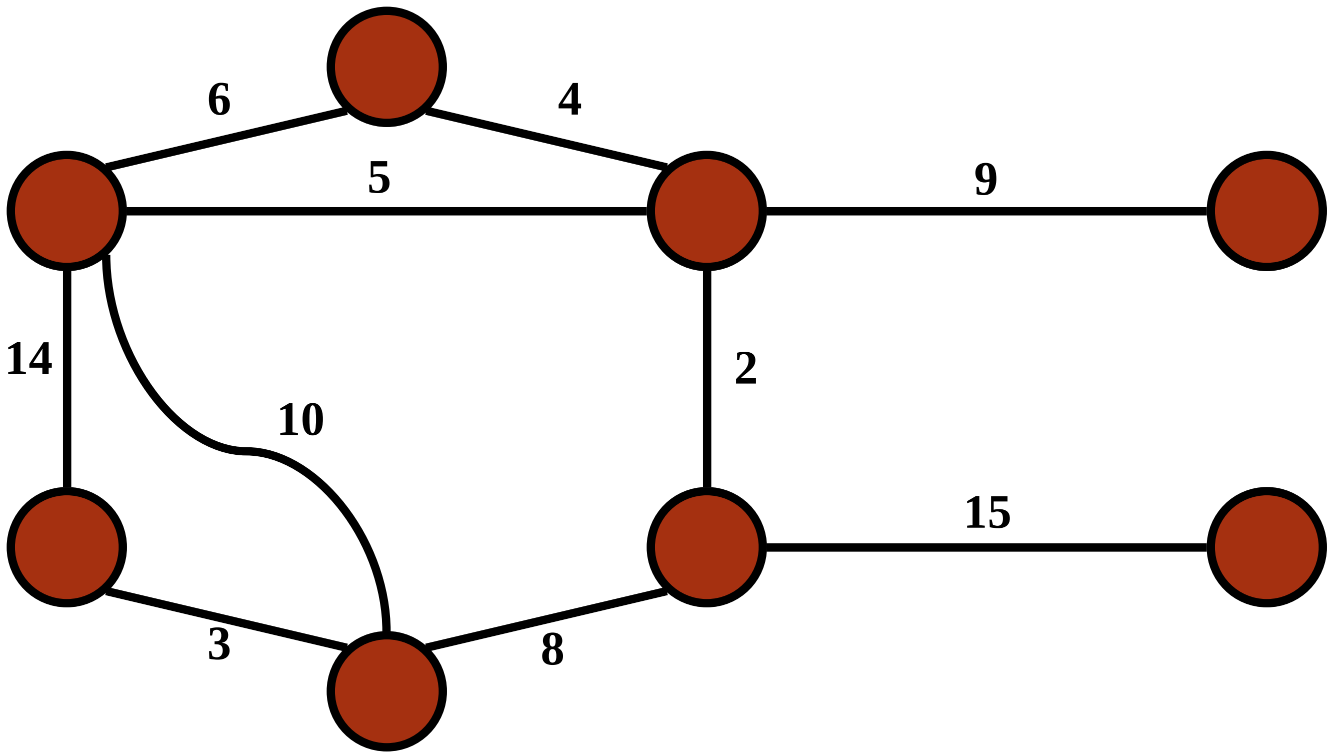
اگر $a = m/2 \rightarrow$ هر دو زیر درخت T_1 و T_2 دارای حداکثر عمق درخت یکسان بوده و عمق درخت T حاصل

از union این دو به اندازه یک واحد از عمق حداکثر T_1 (T_2) بیشتر خواهد بود.

$$depth(T) \le \underbrace{depth(T_1)}_{\text{طبق فرض}} + 1 \le \underbrace{\lceil \log m/2 \rceil + 1}_{\text{طبق فرض}} + 1 = \lceil \log m \rceil + 1$$

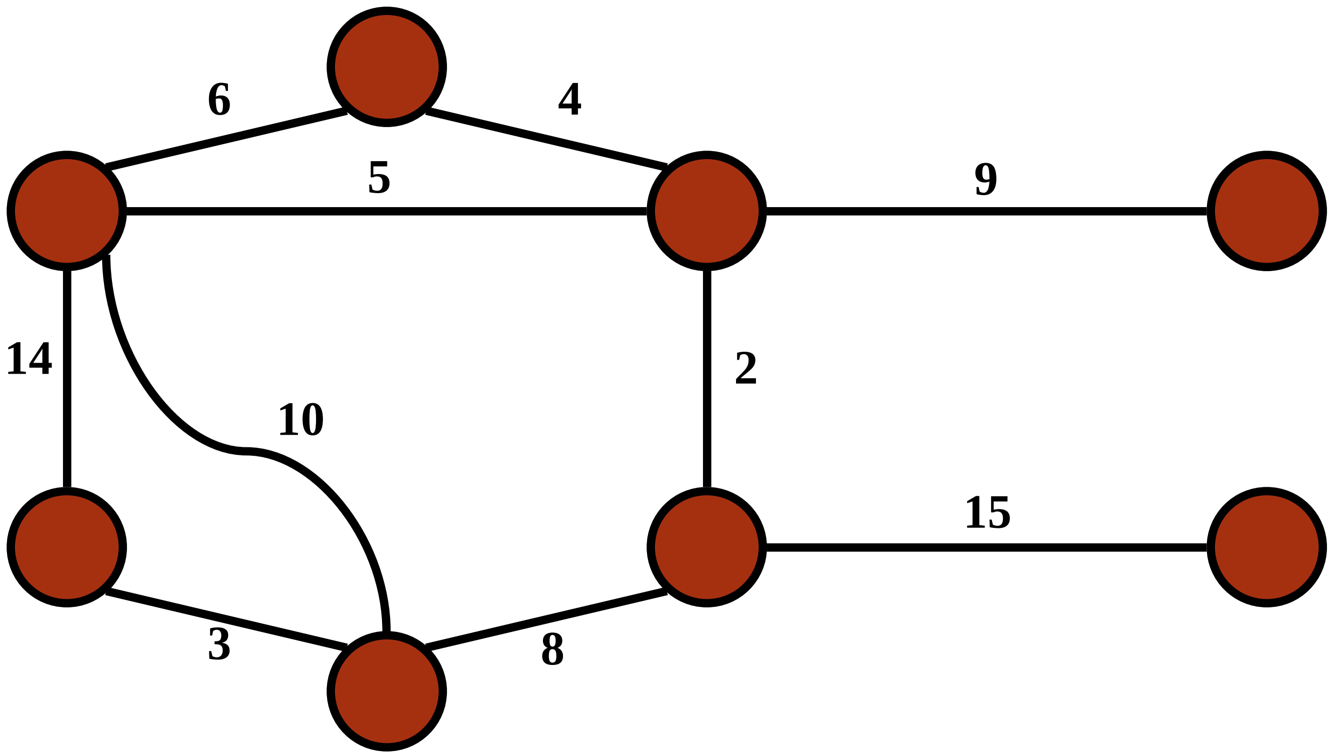
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph:



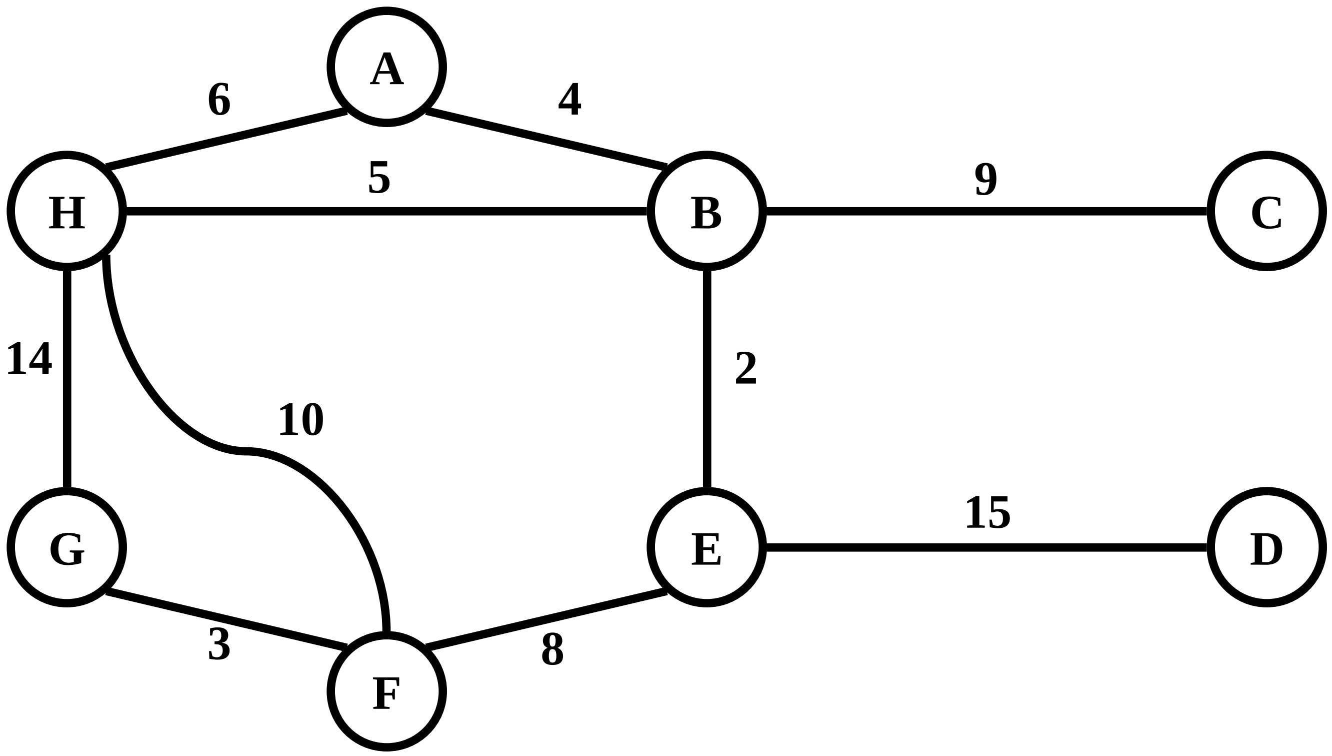
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight



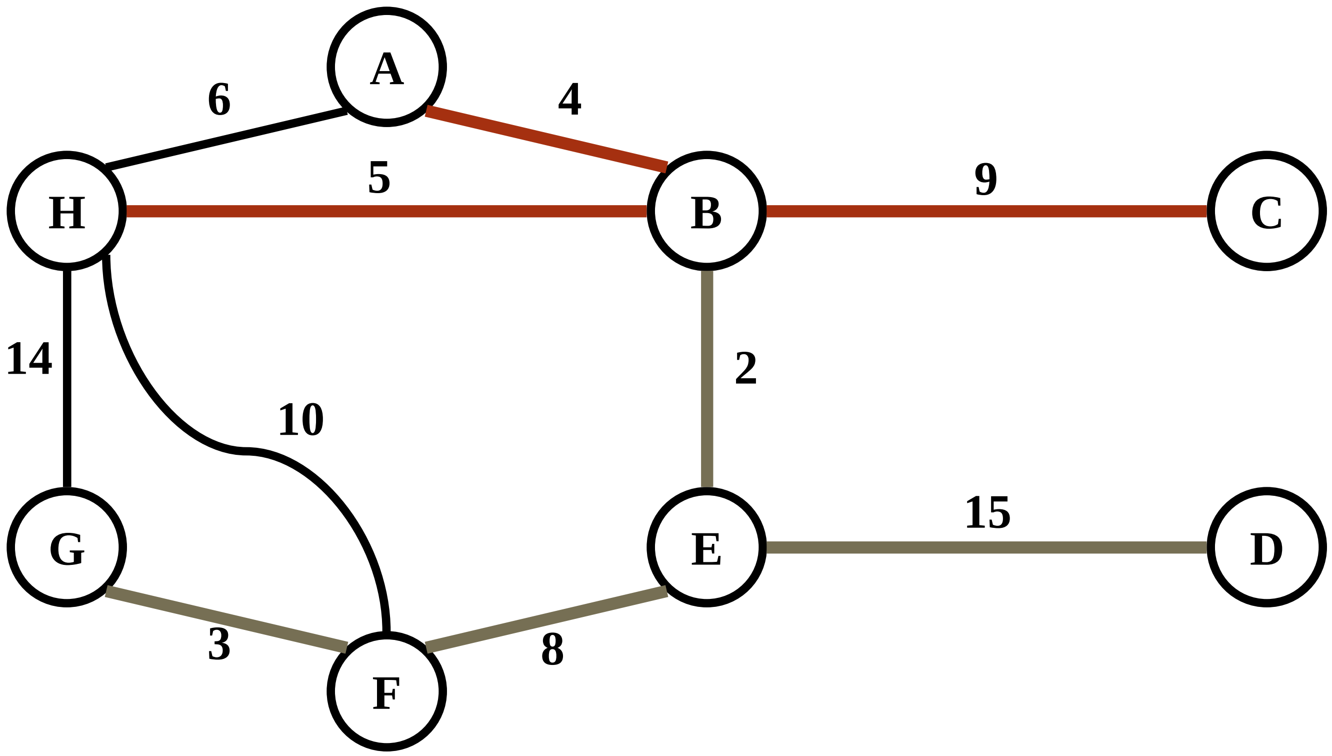
Minimum Spanning Tree

- *Which edges form the minimum spanning tree (MST) of the below graph?*



Minimum Spanning Tree

- Answer:

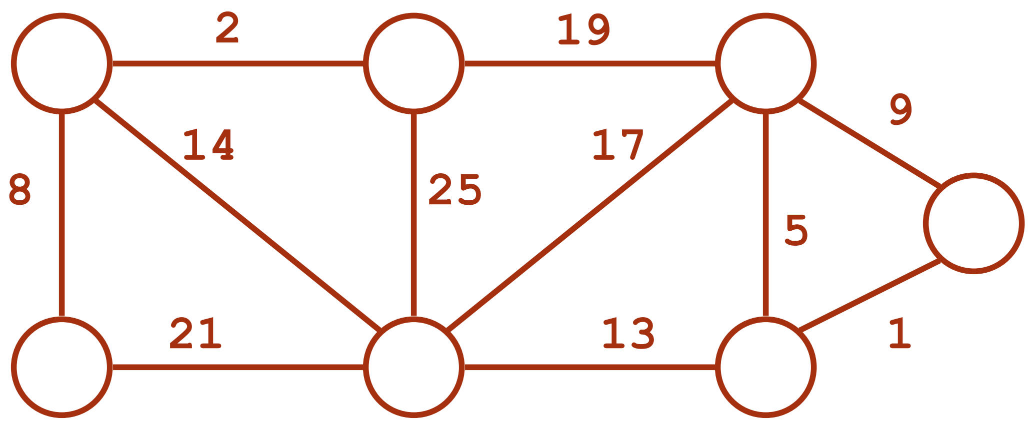


Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

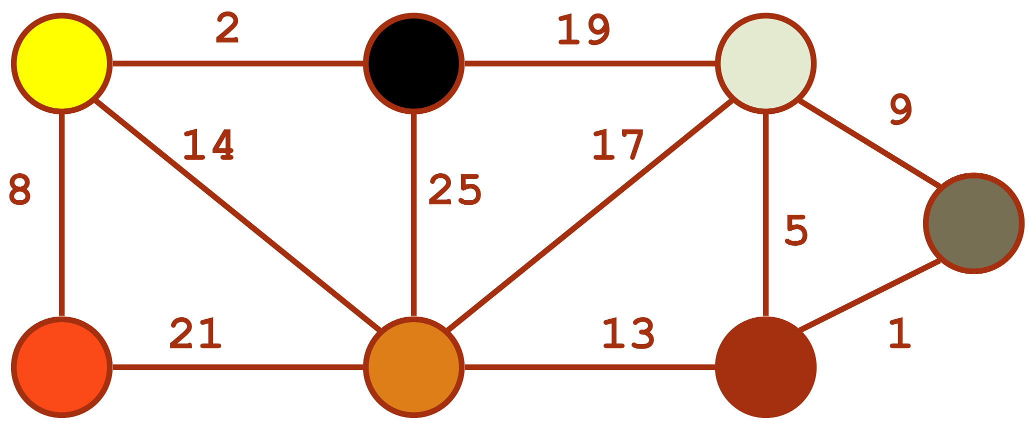



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

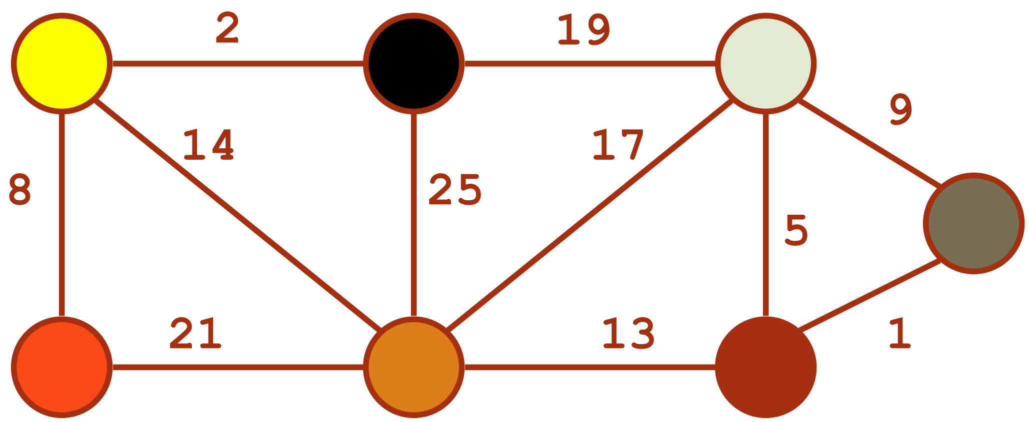
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

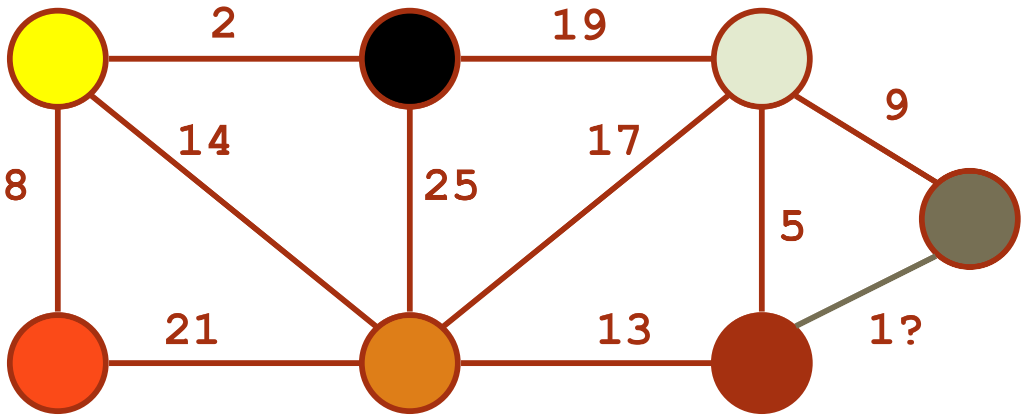
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

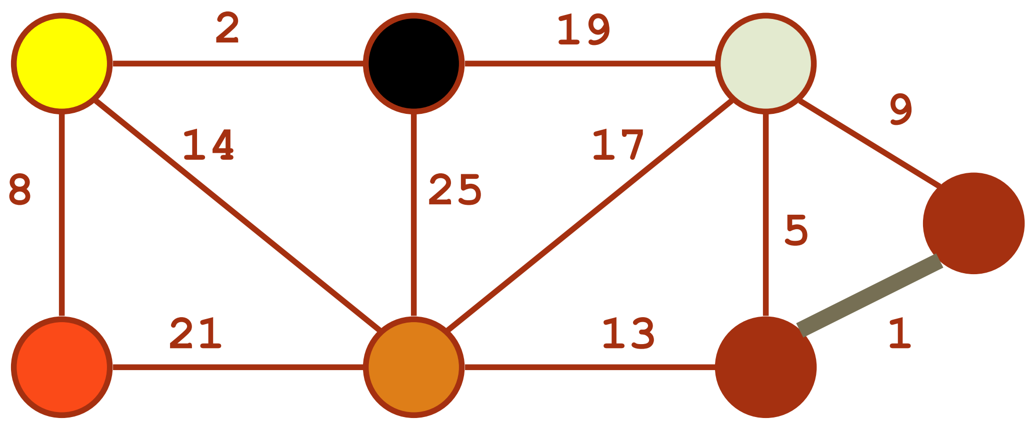



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

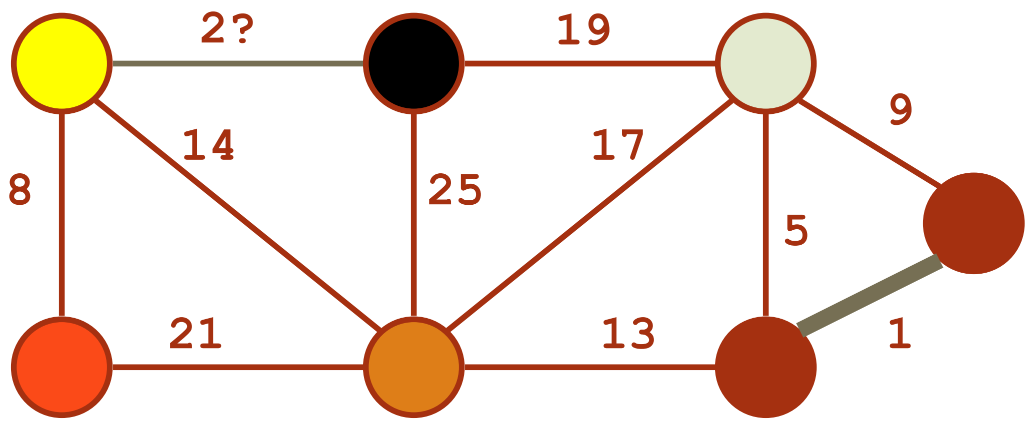
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

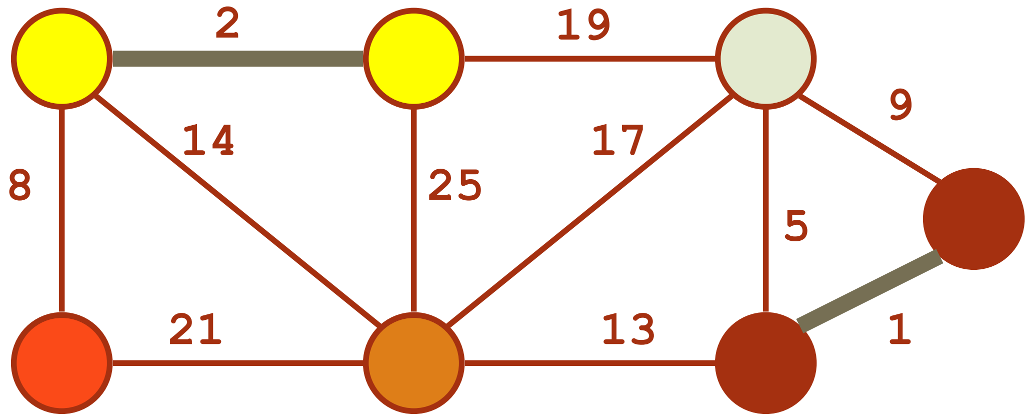
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

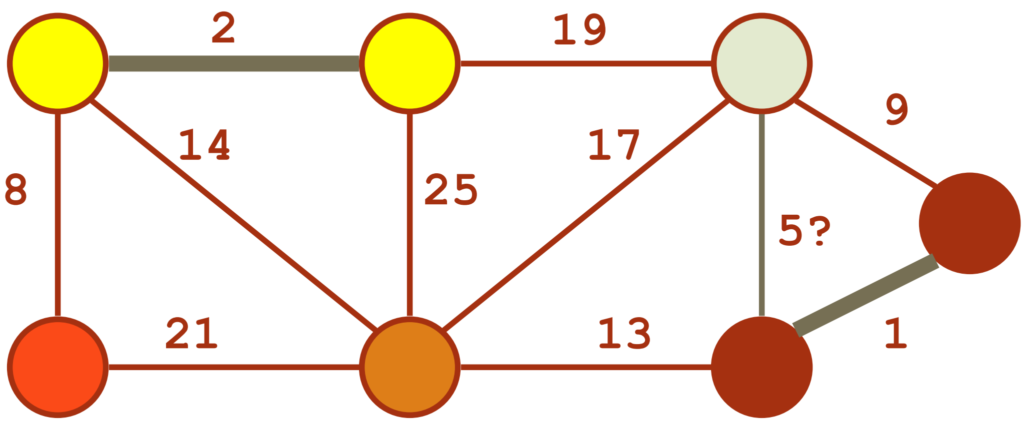
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

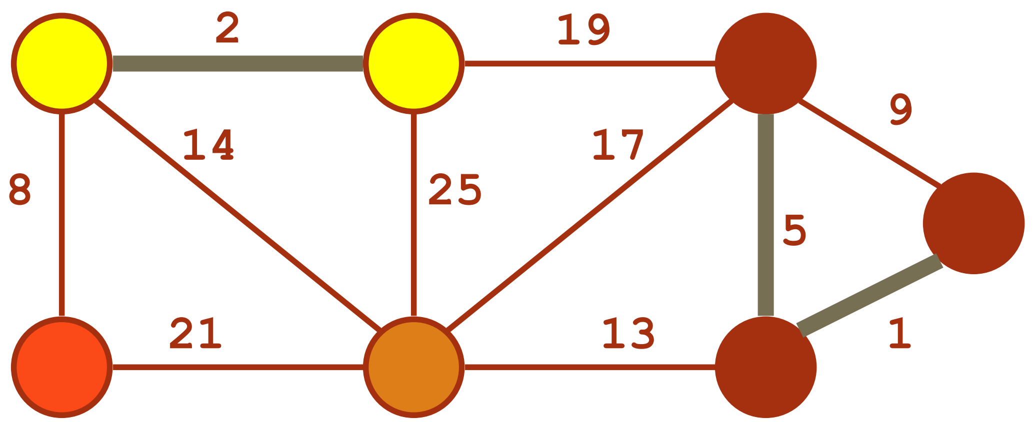



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

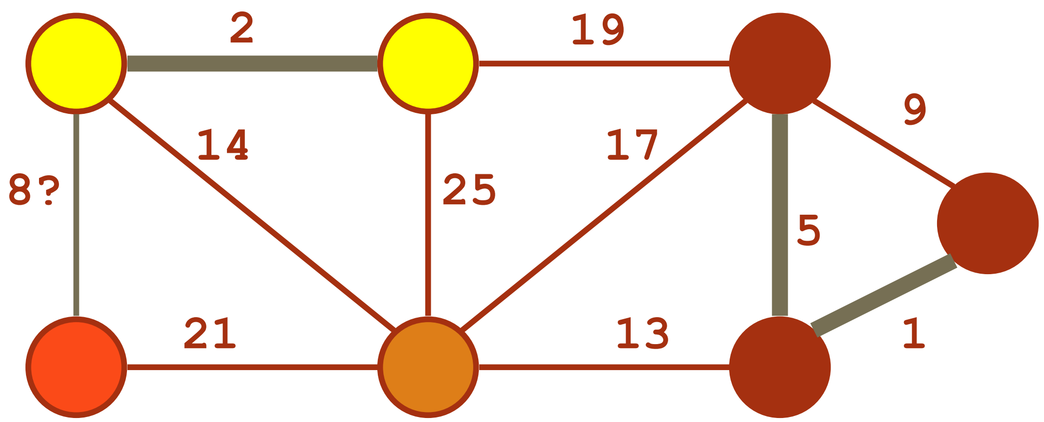
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

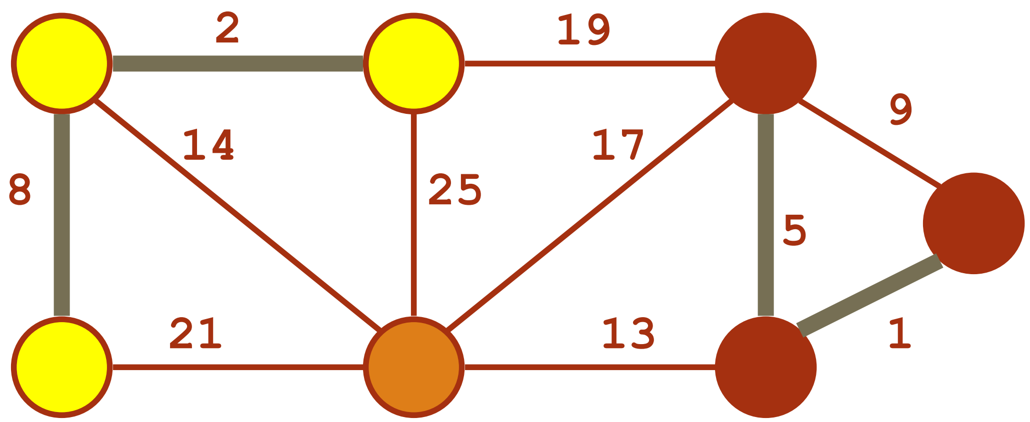
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

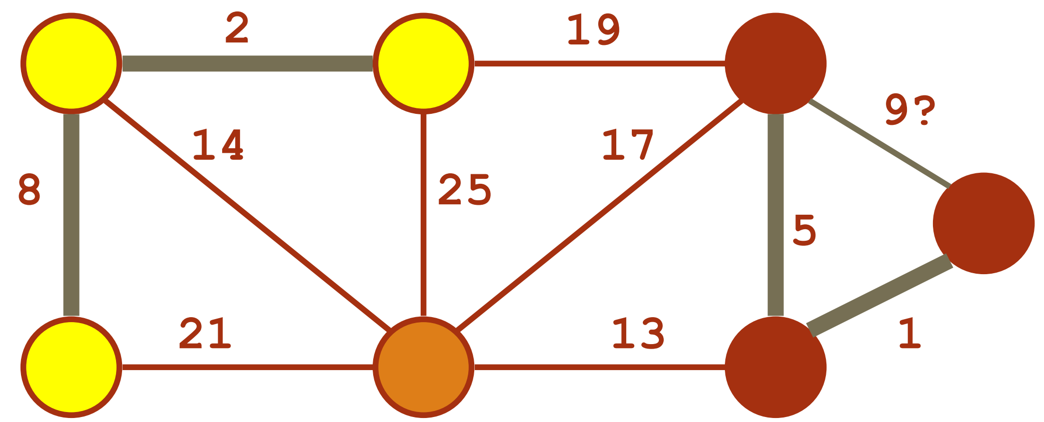



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

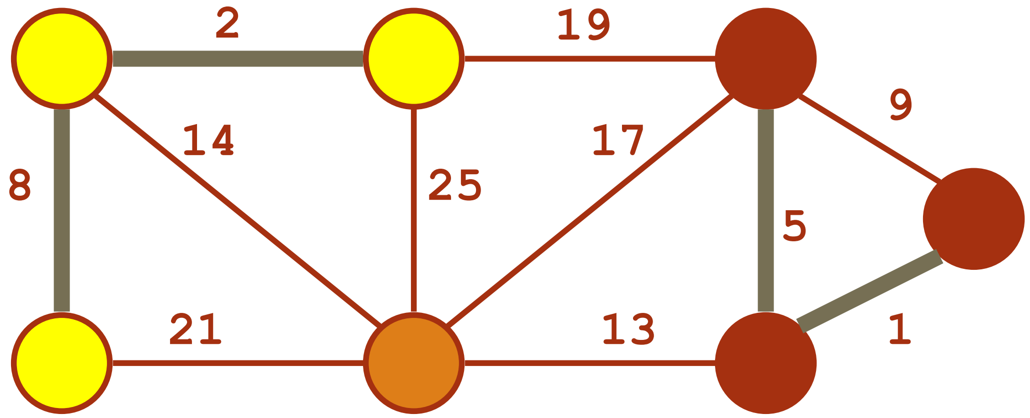
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



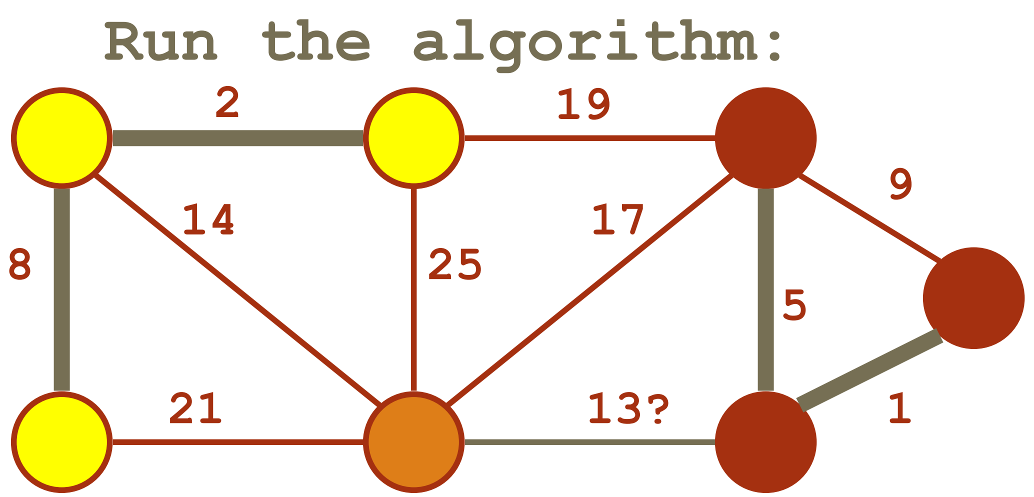
Kruskal's Algorithm

Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

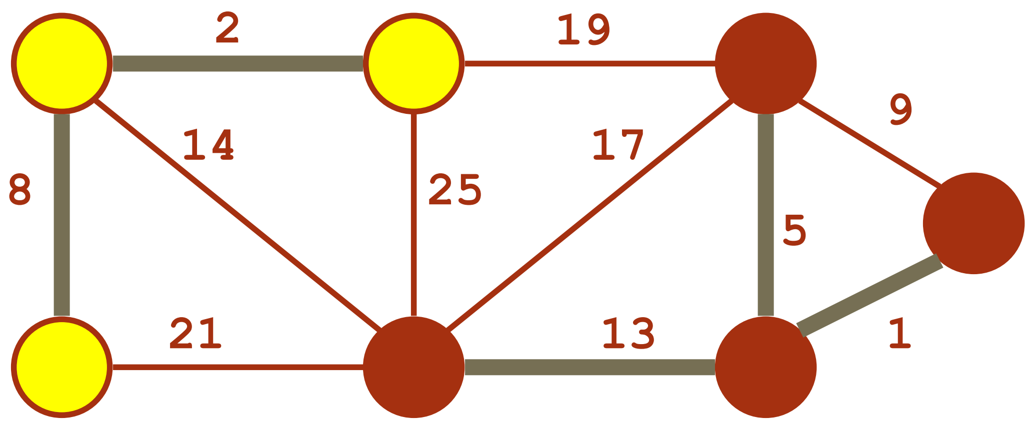



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

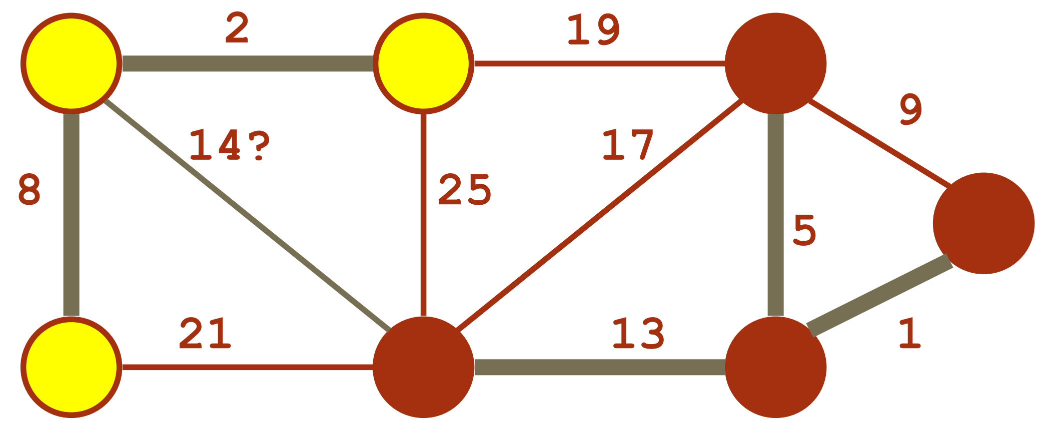
Run the algorithm:



Kruskal's Algorithm

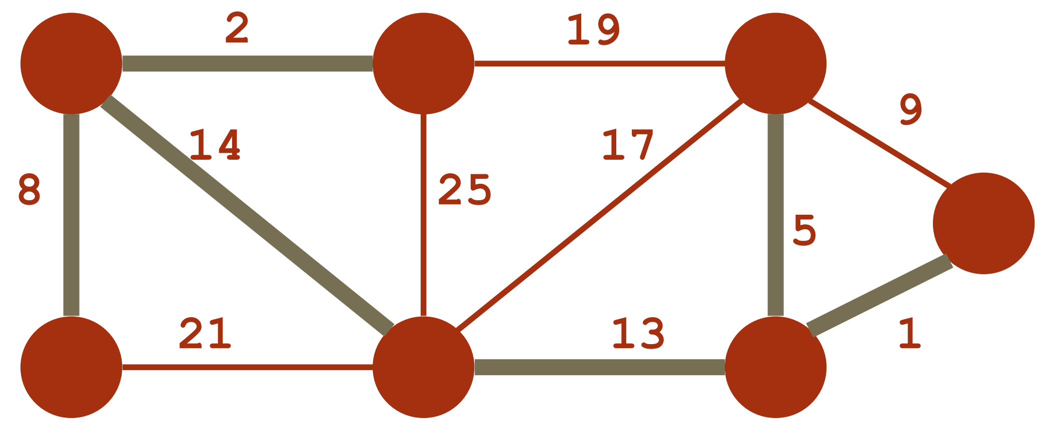
```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

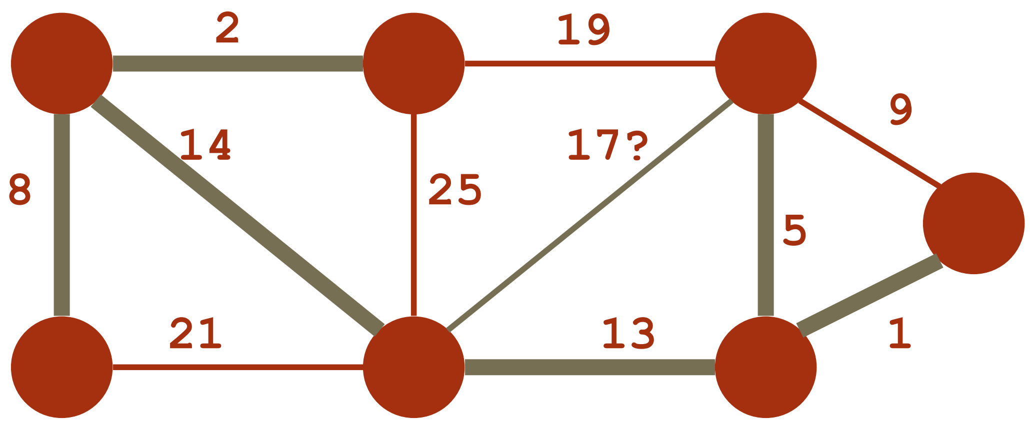
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

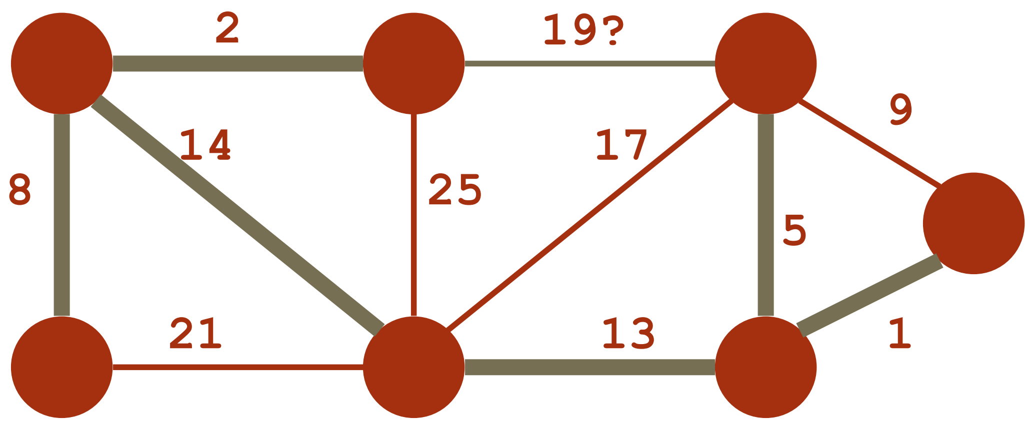
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

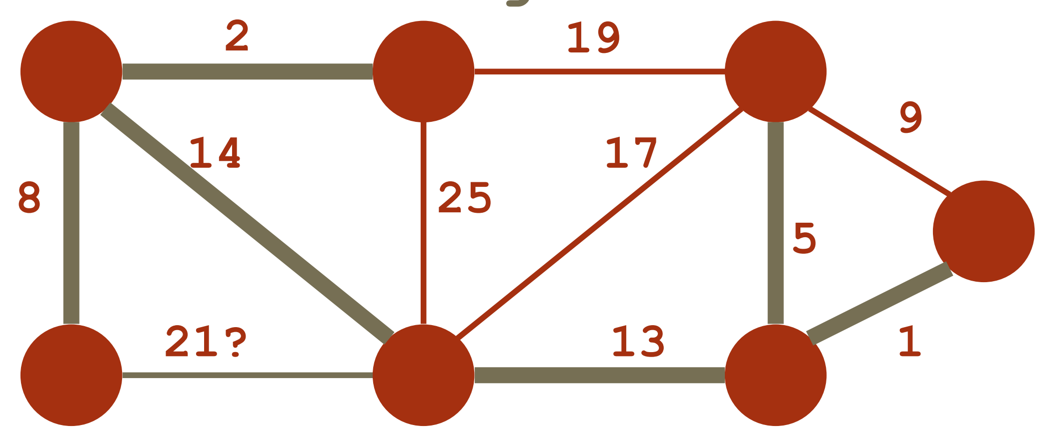
Run the algorithm:




```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

Run the algorithm:

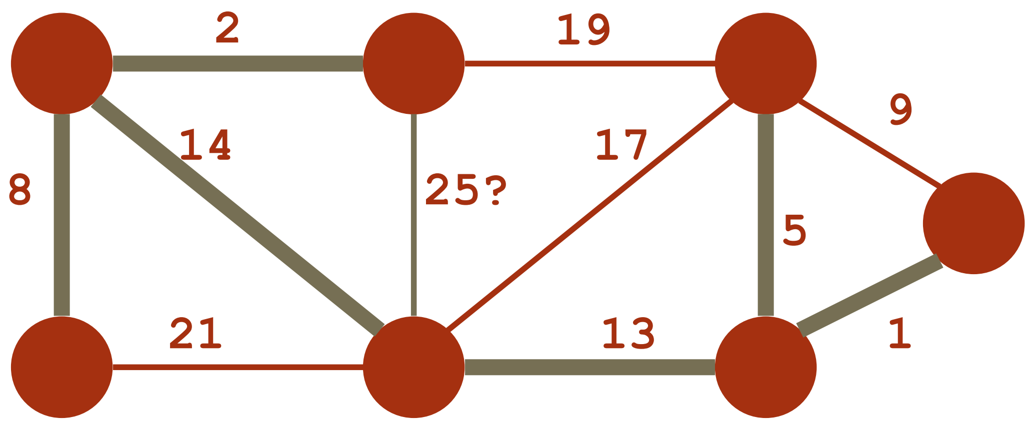



```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

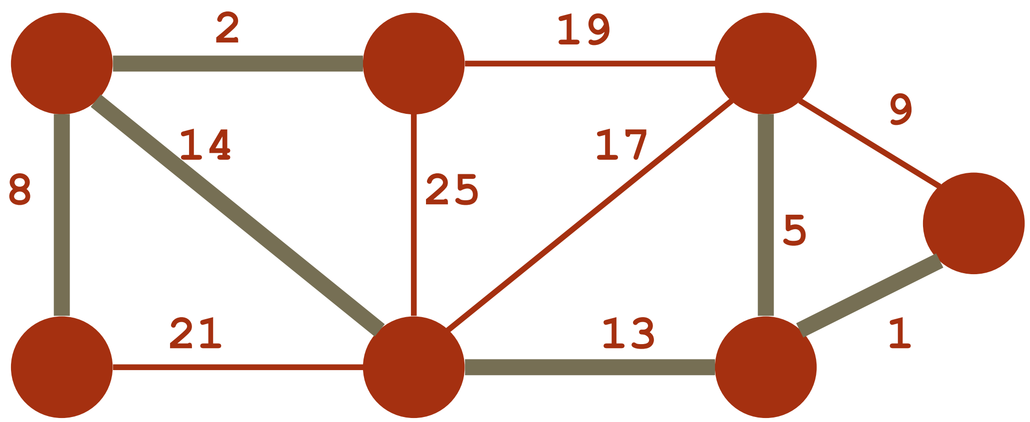
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Kruskal's Algorithm

Run the algorithm:

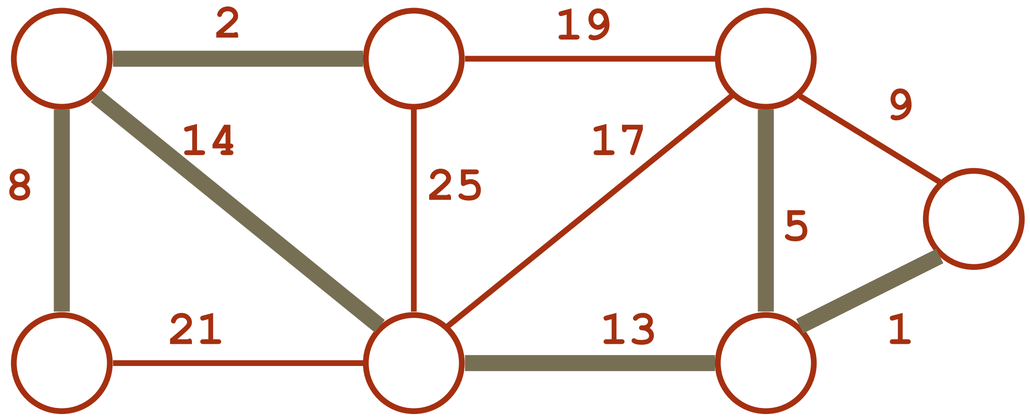



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:



Correctness Of Kruskal's Algorithm

- Sketch of a proof that this algorithm produces an MST for *T*:
  - Assume algorithm is wrong: result is not an MST
  - Then algorithm adds a wrong edge at some point
  - If it adds a wrong edge, there must be a lower weight edge (cut and paste argument)
  - But algorithm chooses lowest weight edge at each step. Contradiction
- Again, important to be comfortable with cut and paste arguments

Kruskal's Algorithm

What will affect the running time?

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

1 Sort

O(V) MakeSet() calls

O(E) FindSet() calls

O(V) Union() calls

(Exactly how many Union()s?)

Kruskal's Algorithm: Running Time

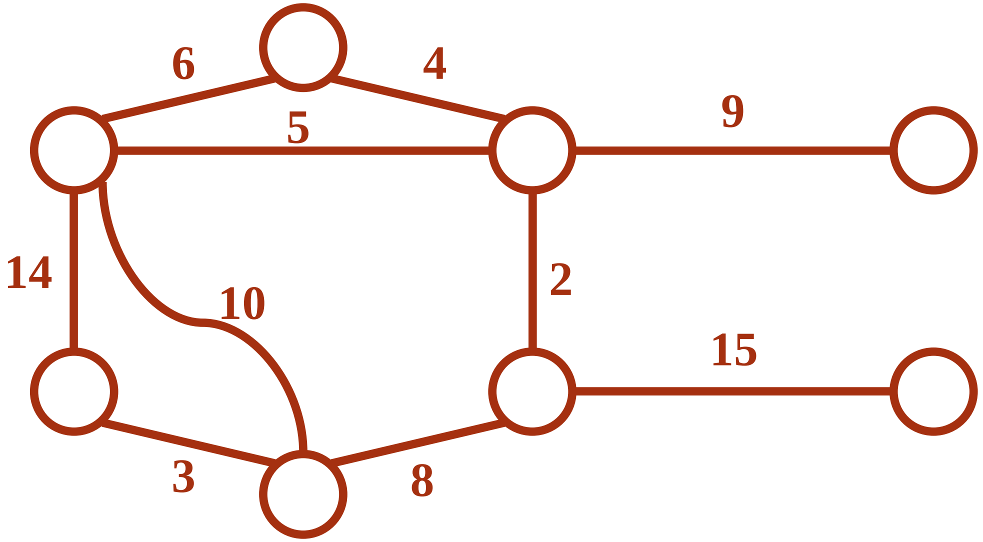
- To summarize:
  - Sort edges: O(E lg E)
  - O(V) MakeSet()'s
  - O(E) FindSet()'s
  - O(V) Union()'s
- Upshot:
  - Best disjoint-set union algorithm makes above 3 operations take $O(E \cdot \alpha(E,V))$, α almost constant
  - Overall thus O(E lg E), almost linear w/o sorting

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Prim's Algorithm

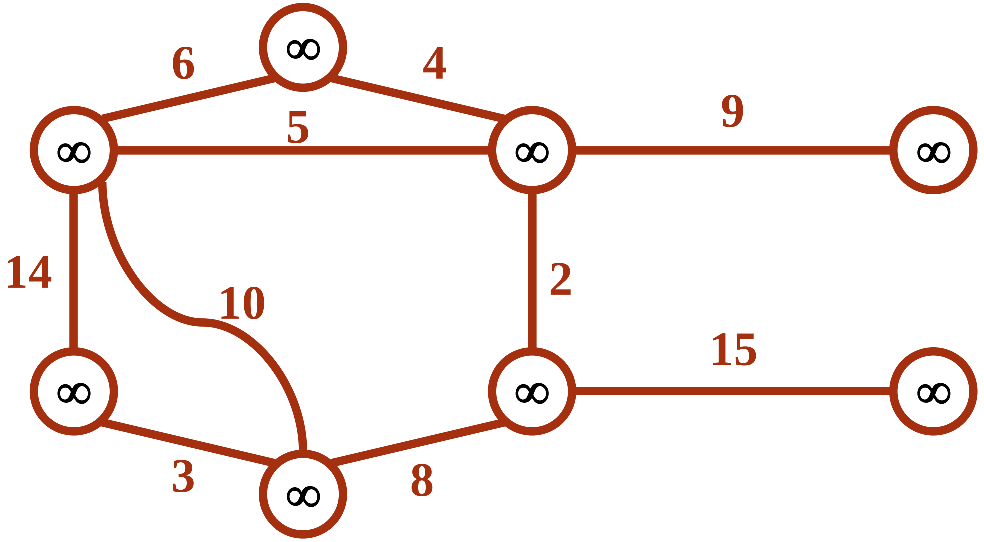
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim's Algorithm

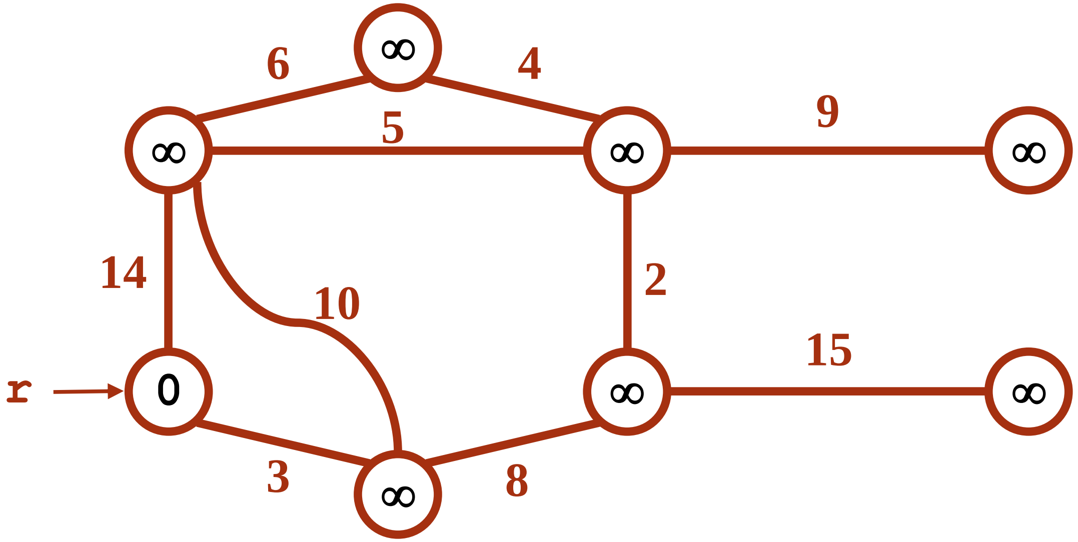
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim's Algorithm

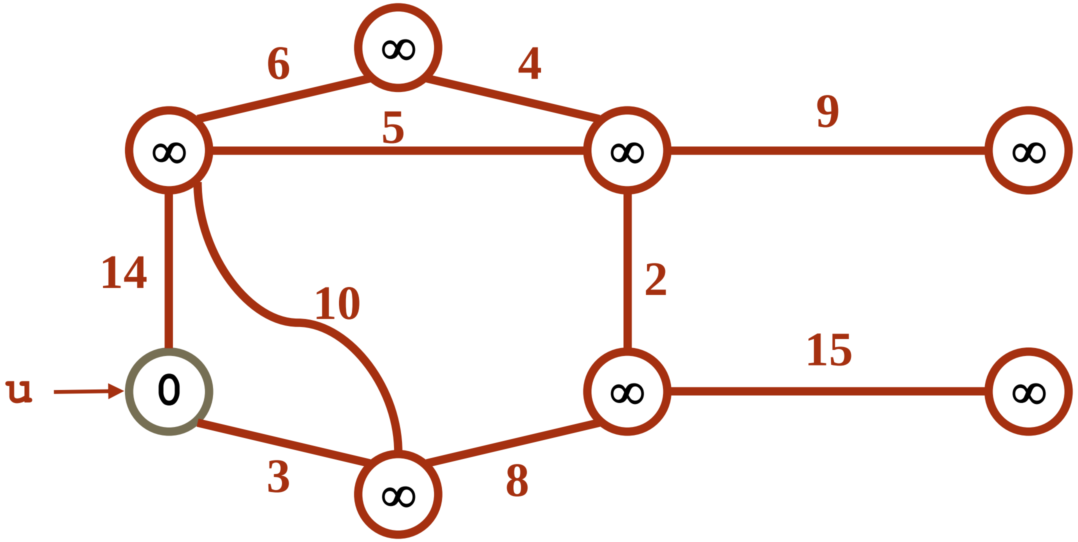
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Pick a start vertex r

Prim's Algorithm

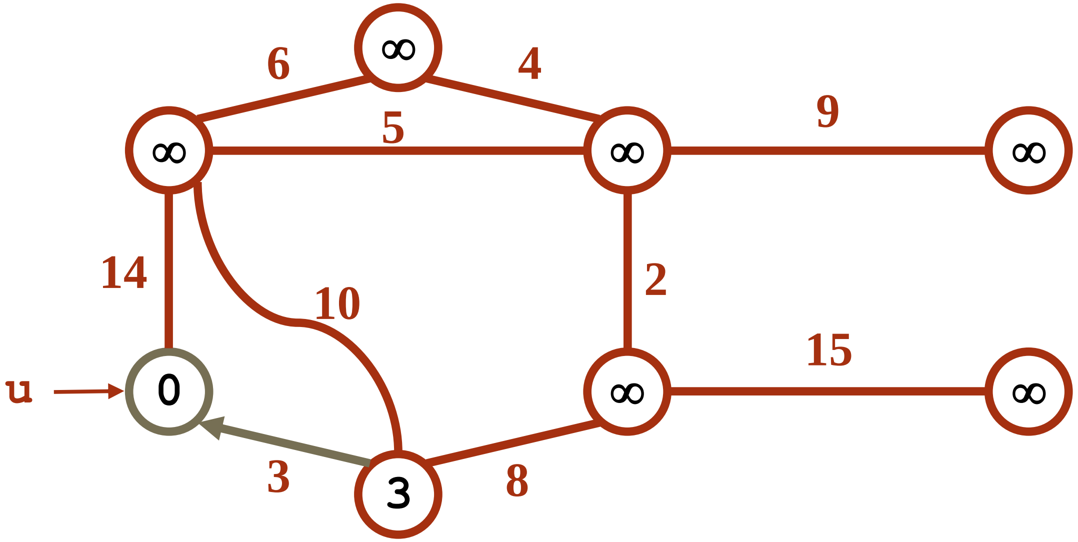
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Red vertices have been removed from Q

Prim's Algorithm

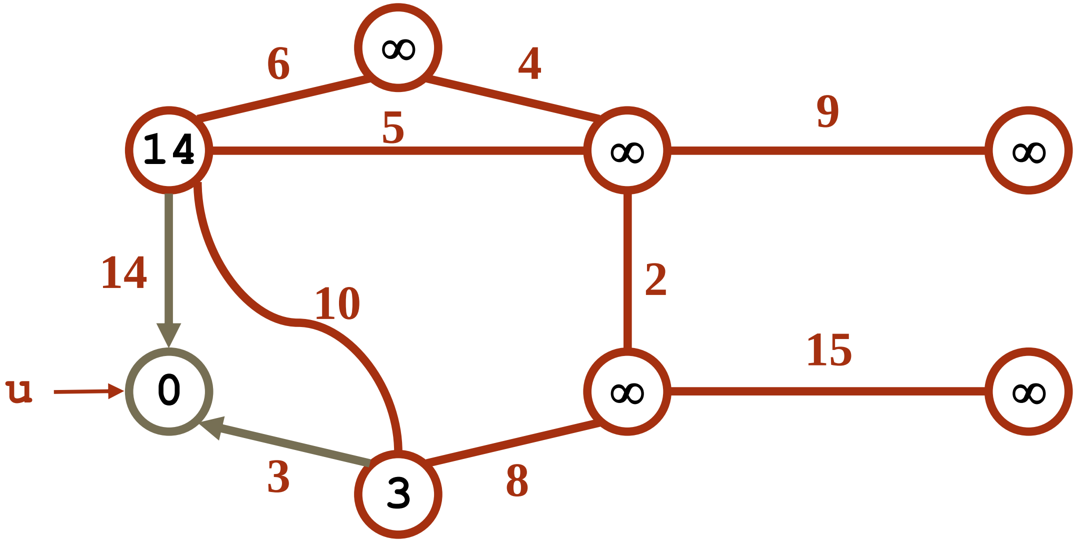
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Red arrows indicate parent pointers

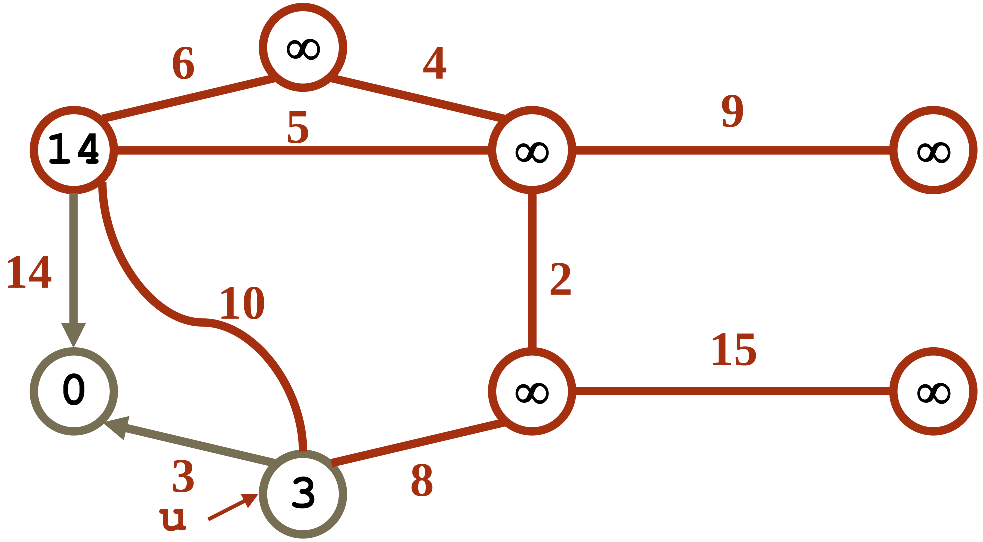
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



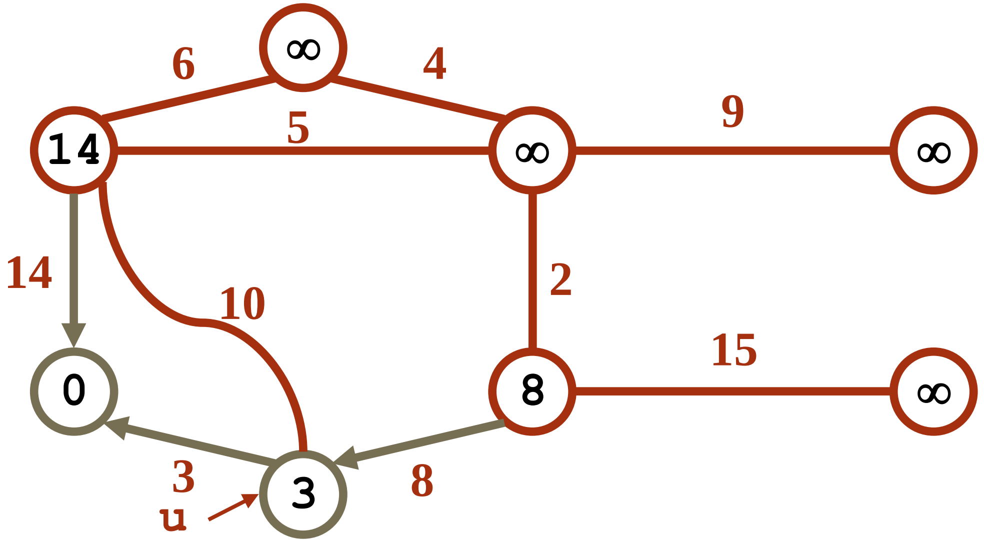
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



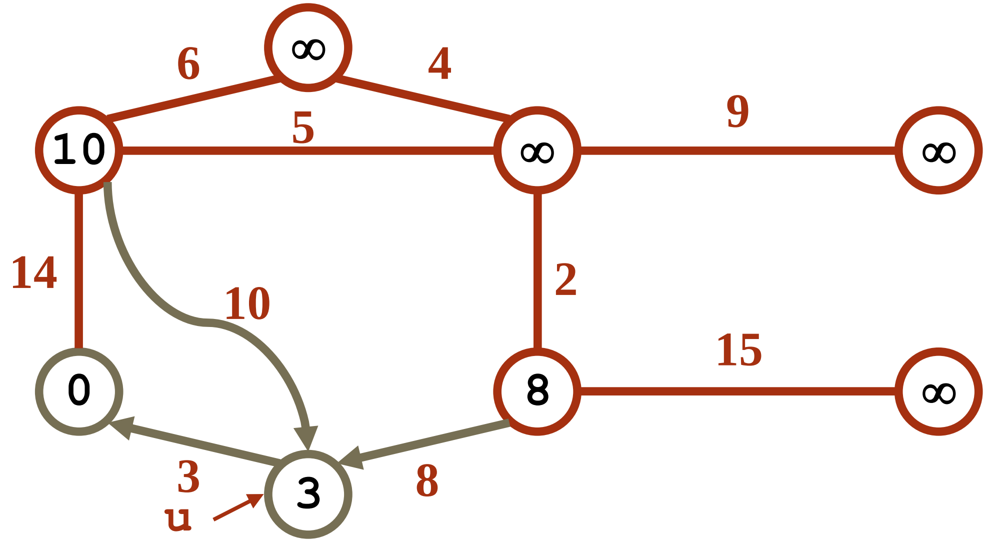
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



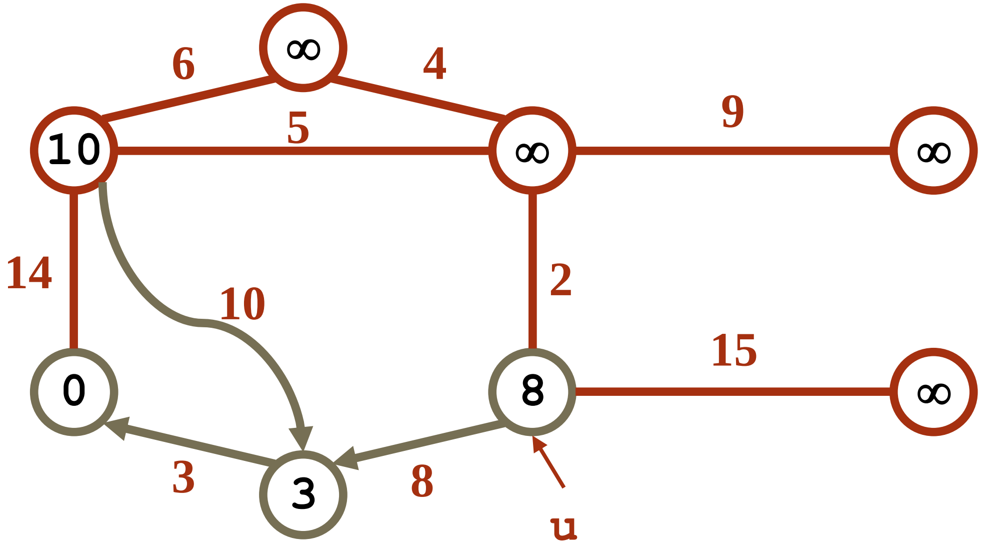
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



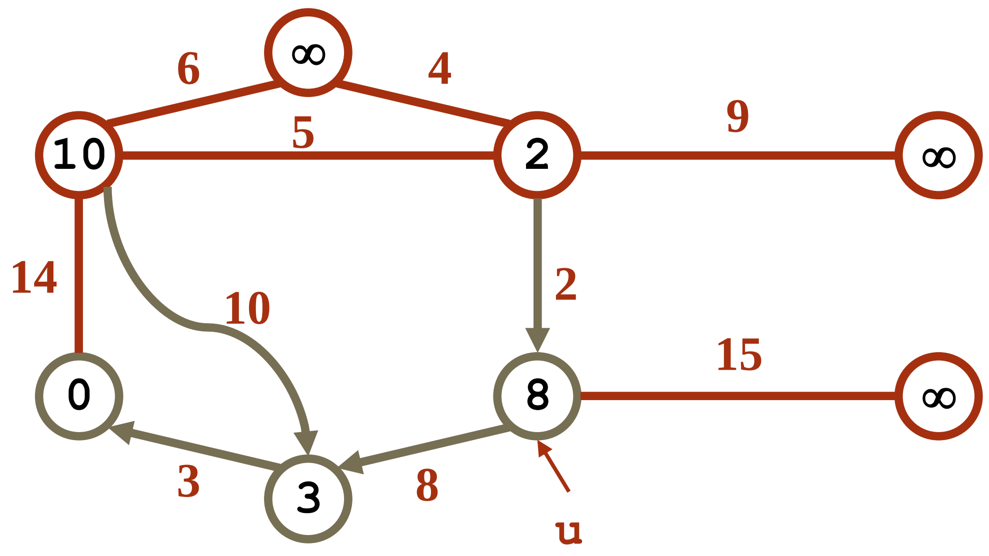
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



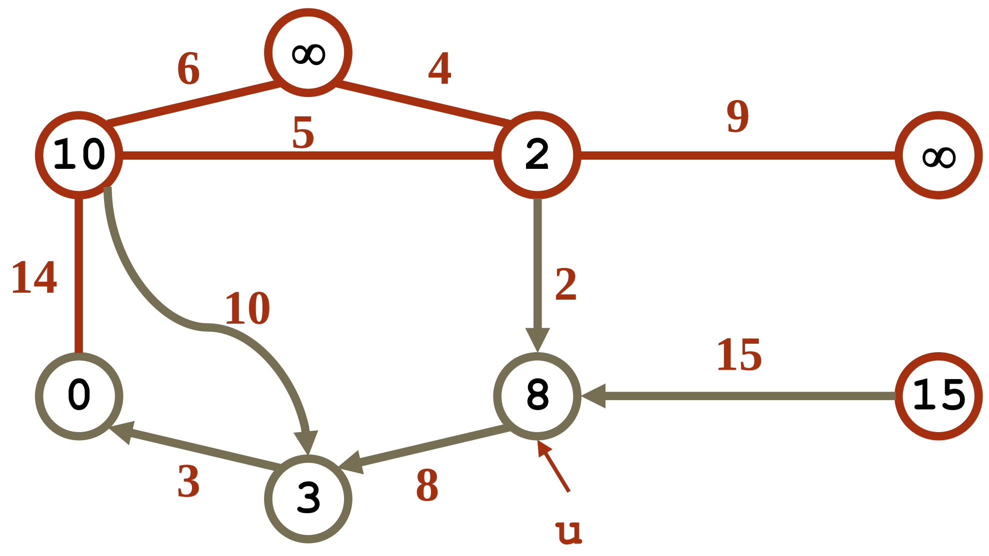
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



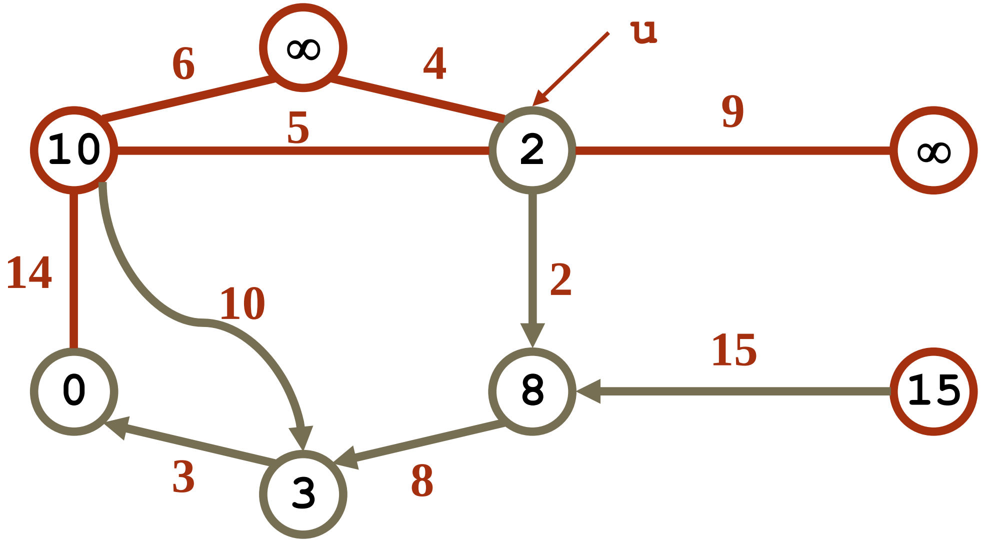
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



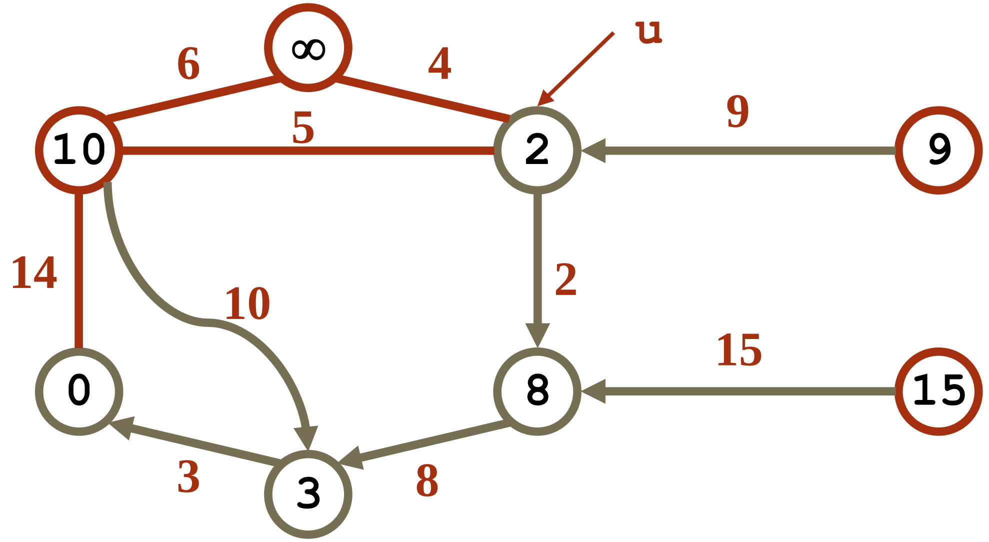
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



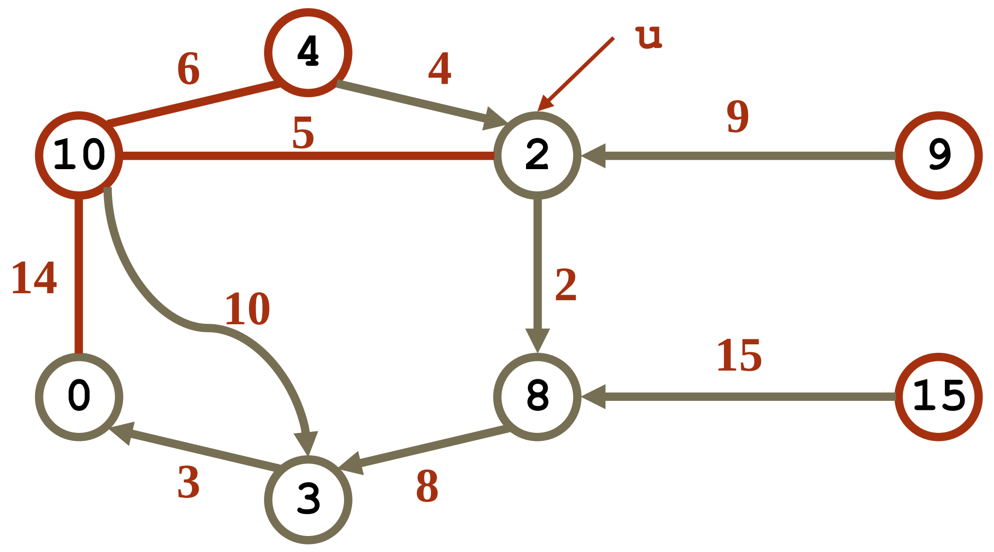
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



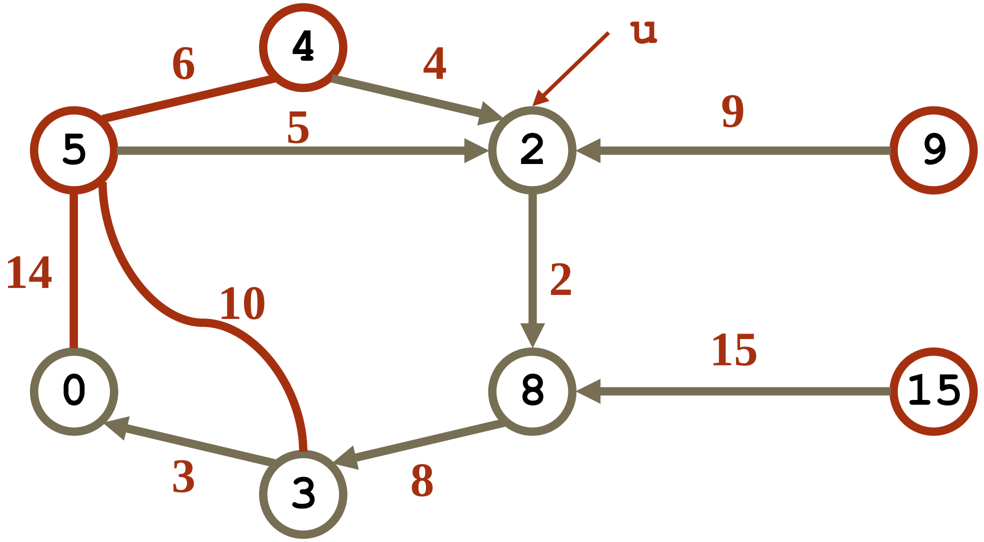
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



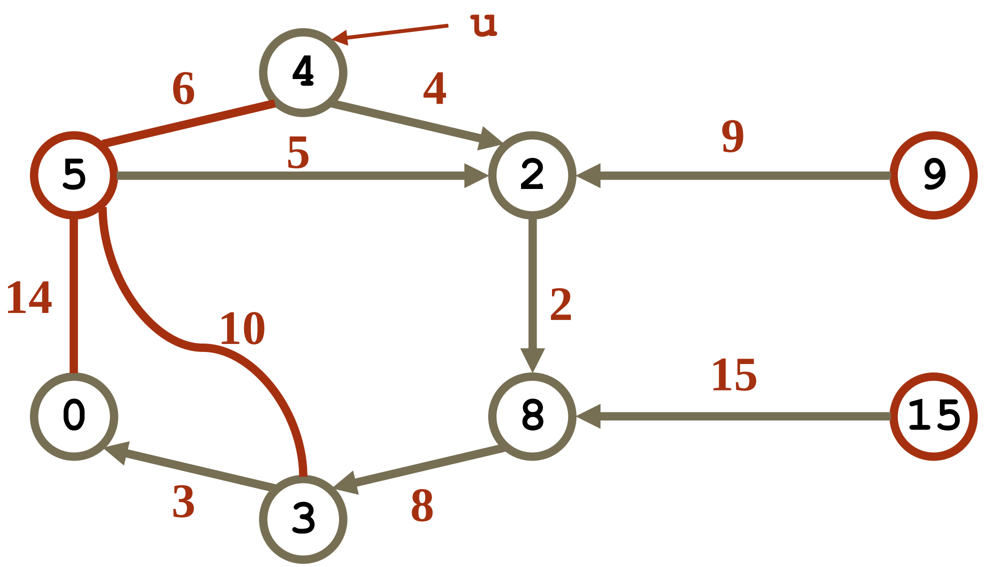
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



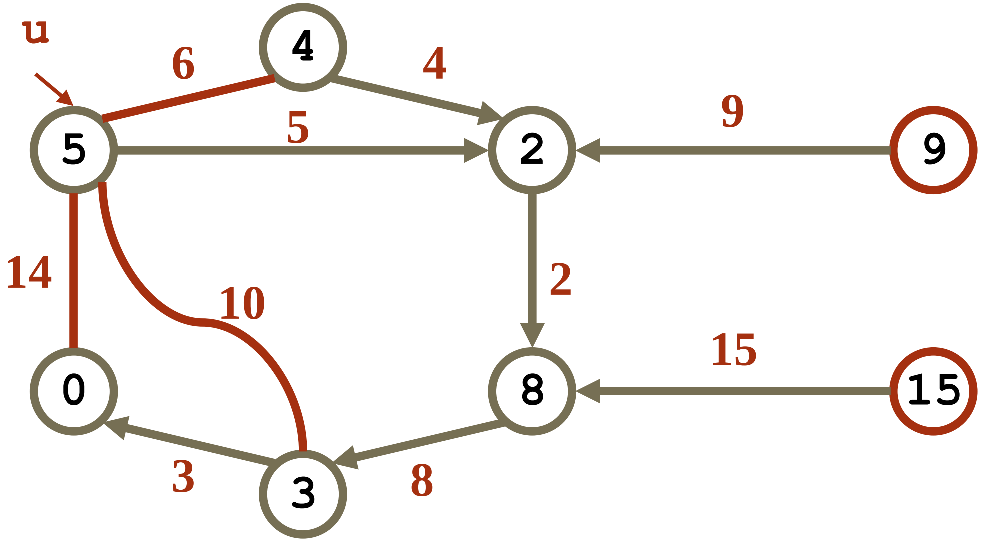
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



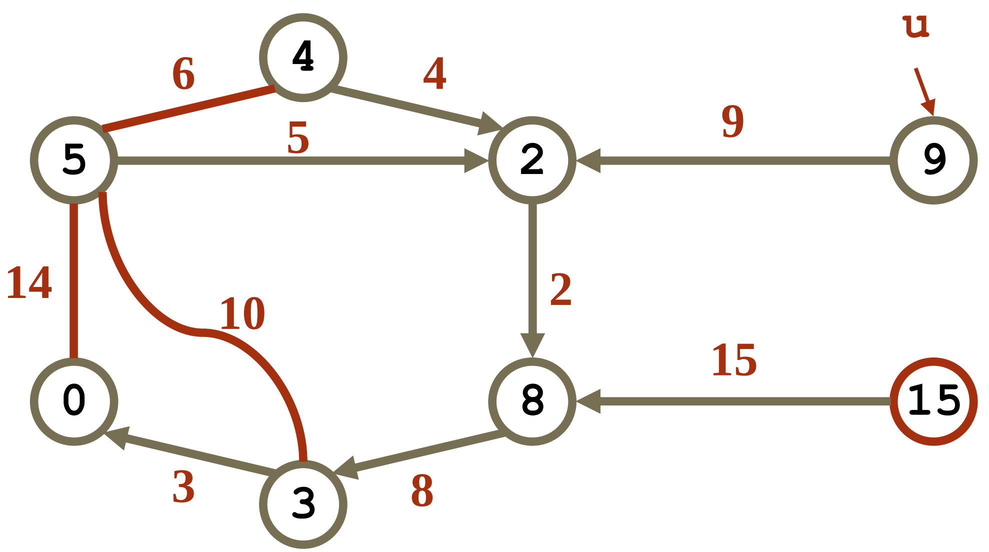
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



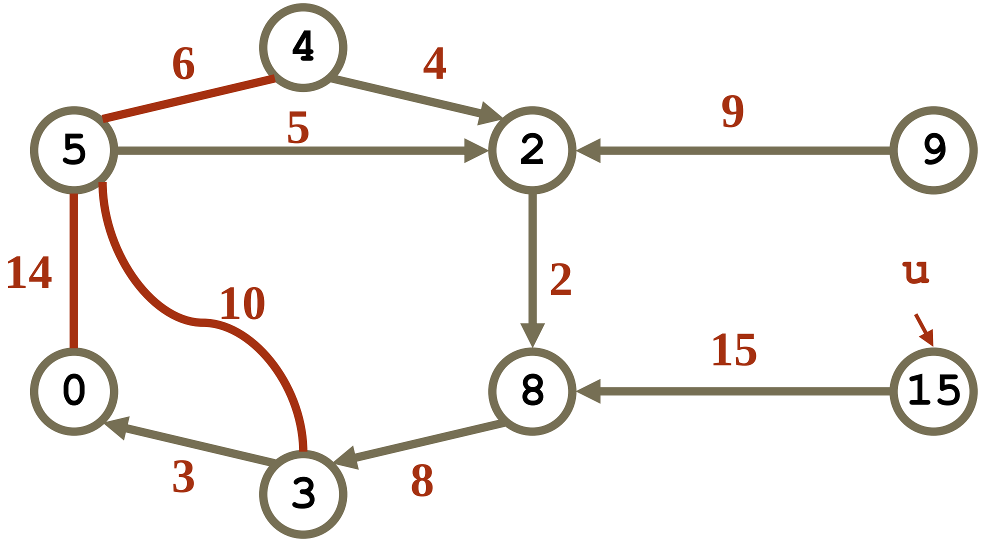
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What is the hidden cost in this code?

Review: Prim’s Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

How often is ExtractMin() called?

How often is DecreaseKey() called?

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What will be the running time?

A: Depends on queue

binary heap: O(E lg V)

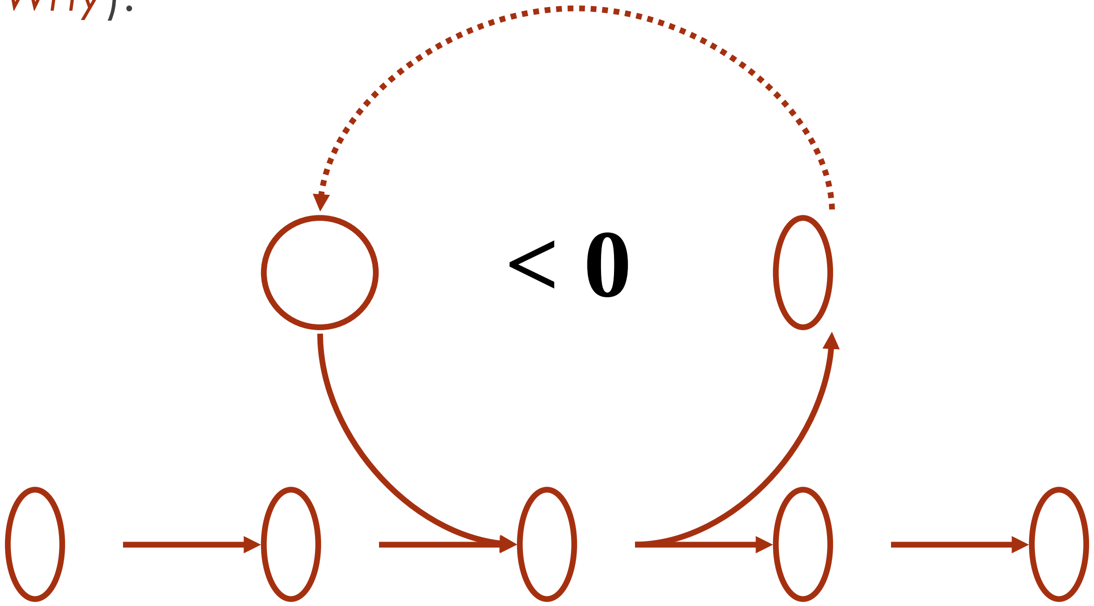
Shortest Path Algorithms

- given a weighted directed graph G, find the minimum-weight path :

- Single source shortest path : Dijkstra, Belman-Ford
- All pairs shortest path : Matrix mult, Floyd-Warshal,…

Shortest Path Properties

- In graphs with negative weight cycles, some shortest paths will not exist (*Why*):



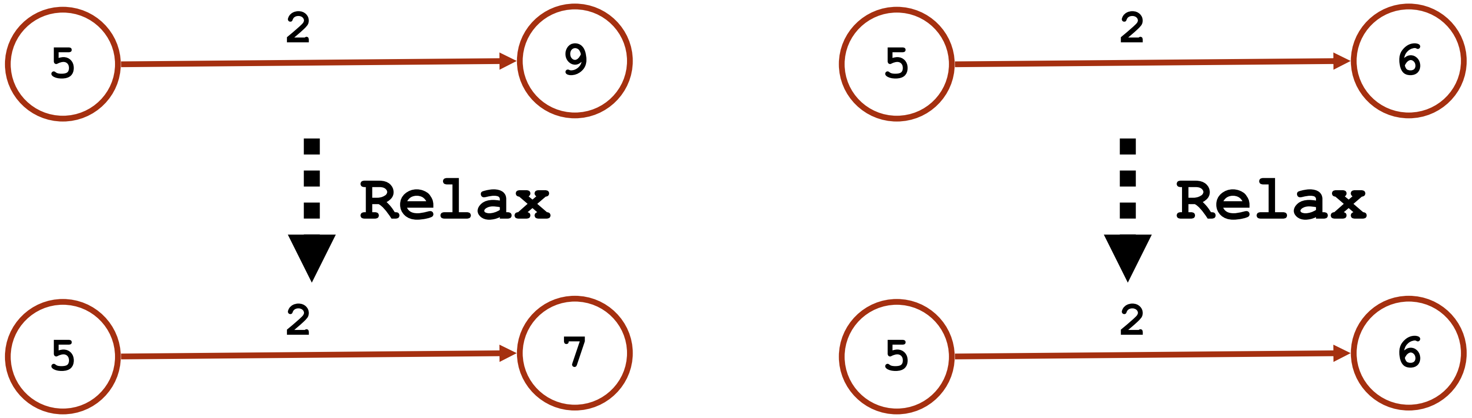
Single-Source Shortest Path

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - "Shortest-path" = minimum weight
  - Weight of path is sum of edges
  - E.g., a road map: what is the shortest path from Chapel Hill to Charlottesville?

Relaxation

- A key technique in shortest path algorithms is *relaxation*
  - Idea: for all v, maintain upper bound d[v] on δ(s,v)

```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```



Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
          return "no solution";
```

Initialize d[], which will converge to shortest-path value δ

Relaxation: Make |V|-1 passes, relaxing each edge

Test for solution
Under what condition do we get a solution?

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";


Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

What will be the running time?

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```

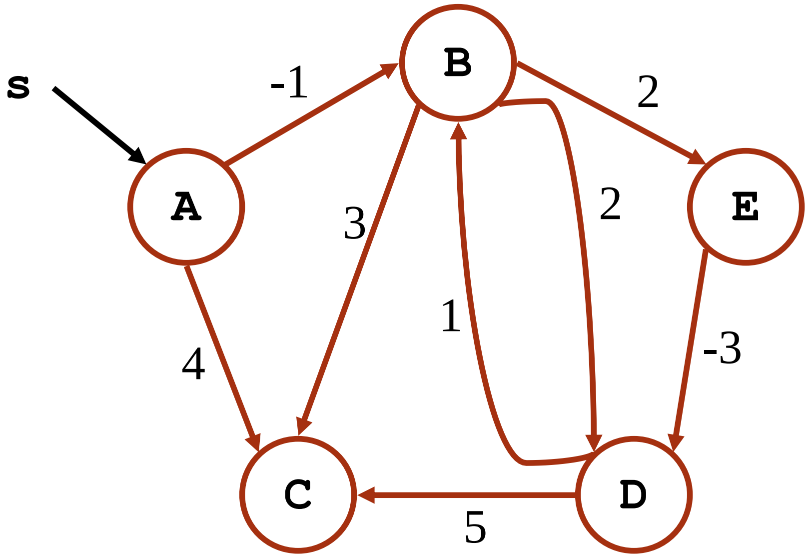
```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

What will be the running time?

A: O(VE)

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```



Ex: work on board

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford

- Note that order in which edges are processed affects how quickly it converges
- Correctness: show $d[v] = \delta(s,v)$ after $|V|-1$ passes
  - Lemma: $d[v] \geq \delta(s,v)$ always
    - Initially true
    - Let v be first vertex for which $d[v] < \delta(s,v)$
    - Let u be the vertex that caused d[v] to change: $d[v] = d[u] + w(u,v)$
    - Then $d[v] < \delta(s,v)$

      $\delta(s,v) \leq \delta(s,u) + w(u,v)$ (*Why?*)

      $\delta(s,u) + w(u,v) \leq d[u] + w(u,v)$ (*Why?*)
    - So $d[v] < d[u] + w(u,v)$. Contradiction.

Bellman-Ford

- Prove: after |V|-1 passes, all *d* values correct
  - Consider shortest path from s to v:
    $s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v$
    - Initially, d[s] = 0 is correct, and doesn't change (*Why?*)
    - After 1 pass through edges, $d[v_1]$ is correct (*Why?*) and doesn't change
    - After 2 passes, $d[v_2]$ is correct and doesn't change
    - …
    - Terminates in |V| - 1 passes: (*Why?*)
    - *What if it doesn't?*

Dijkstra's Algorithm

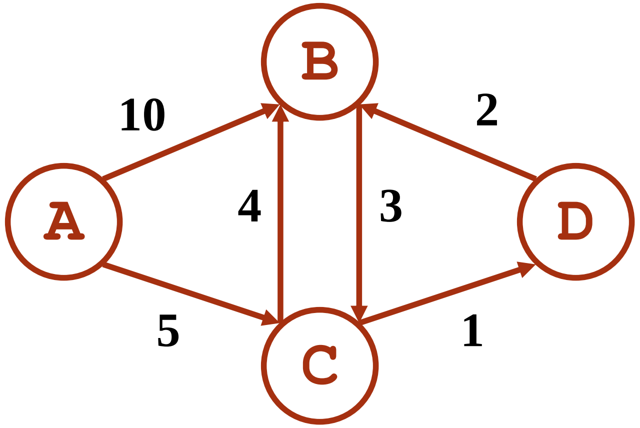
- If no negative edge weights, we can beat BF
- Similar to breadth-first search
  - Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  - Use a priority queue keyed on d[v]

Dijkstra's Algorithm

```
Dijkstra(G)
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0; S = ∅; Q = V;
    while (Q ≠ ∅)
        u = ExtractMin(Q);
        S = S U {u};
        for each v ∈ u->Adj[]
            if (d[v] > d[u]+w(u,v))
                d[v] = d[u]+w(u,v);
```

Relaxation Step

Note: this is really a call to Q->DecreaseKey()



Ex: run the algorithm

Dijkstra's Algorithm

```
Dijkstra(G)
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0; S = ∅; Q = V;
    while (Q ≠ ∅)
        u = ExtractMin(Q);
        S = S U {u};
        for each v ∈ u->Adj[]
            if (d[v] > d[u]+w(u,v))
                d[v] = d[u]+w(u,v);
```

How many times is ExtractMin() called?

How many times is DecraseKey() called?

What will be the total running time?

Dijkstra's Algorithm

```
Dijkstra(G)
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0; S = ∅; Q = V;
    while (Q ≠ ∅)
        u = ExtractMin(Q);
        S = S U {u};
        for each v ∈ u->Adj[]
            if (d[v] > d[u]+w(u,v))
                d[v] = d[u]+w(u,v);
```

How many times is ExtractMin() called?

How many times is DecraseKey() called?

A: O(E lg V) using binary heap for Q

Can acheive O(V lg V + E) with Fibonacci heaps

Dijkstra's Algorithm

```
Dijkstra(G)
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0; S = ∅; Q = V;
    while (Q ≠ ∅)
        u = ExtractMin(Q);
        S = S U{u};
        for each v ∈ u->Adj[]
            if (d[v] > d[u]+w(u,v))
                d[v] = d[u]+w(u,v);
```

Correctness: we must show that when u is removed from Q, it has already converged

All-pairs shortest paths

All –pairs shortest-paths problem

- Problem: Given a directed graph $G=(V, E)$, and a weight function $w: E \rightarrow R$, for each pair of vertices u, v, compute the shortest path weight $\delta(u, v)$, and a shortest path if exists.

- Output:
  - A $V \times V$ matrix $D = (d_{ij})$, where, d_{ij} contains the shortest path weight from vertex i to vertex j. **//Important!**
  - A $V \times V$ matrix $\Pi=(\pi_{ij})$, where, π_{ij} is NIL if either $i=j$ or there is no path from i to j, otherwise π_{ij} is the predecessor of j on some shortest path from i. **// Not covered in class, but in Exercises!**

Methods

1) Application of single-source shortest-path algorithms
2) Direct methods to solve the problem:
    1) Matrix multiplication
    2) Floyd-Warshall algorithm
    3) Johnson's algorithm for sparse graphs
3) Transitive closure (Floyd-Warshall algorithm)

Matrix multiplication

--suppose there are no negative cycles.

- A dynamic programming method:
  - Study structure of an optimal solution
  - Solve the problem recursively
  - Compute the value of an optimal solution in a bottom-up manner
- The operation of each loop is like matrix multiplication.

Matrix multiplication—structure of a shortest path

- Suppose $W = (w_{ij})$ is the adjacency matrix such that

$$w_{ij} = \begin{cases} 0, & \text{if } i = j \\ \text{the weight of edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

Consider a shortest path P from i to j, and suppose that P has at most m edges. Then,

if $i = j$, P has weight 0 and no edges.

If $i \neq j$, we can decompose P into $i \overset{P'}{\leadsto} k \rightarrow j$,

P' is a shortest path from i to k.

Matrix multiplication—recursive solution

Let $l_{ij}^{(m)}$ be the minimum weight of any path from i to j that contains at most m edges.

- $l_{ij}^{(0)} = 0$, if $i = j$, and ∞ otherwise.
- For $m \geq 1$,
- $$l_{ij}^{(m)} = \min_{1 \leq k \leq n}\{l_{ik}^{(m-1)} + w_{kj}\},$$
- The solution is $l_{ij}^{(n-1)}$

Matrix Multiplication

- Solve the problem stage by stage (dynamic programming)
- $L^{(1)} = W$
- $L^{(2)}$
- …
- $L^{(n-1)}$
- where $L^{(m)}$, contains the shortest path weight with path length $\leq m$.

Matrix multiplication (pseudo-code)

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

$n \leftarrow rows[W]$
$L^{(1)} \leftarrow W$
for $m \leftarrow 2$ **to** $n - 1$
    do $L^{(m)} \leftarrow$ EXTEND-SHORTEST-PATHS$(L^{(m-1)}, W)$
return $L^{(n-1)}$

Matrix multiplication (pseudo-code)

EXTEND-SHORTEST-PATHS(L, W)

$n \leftarrow rows[L]$
let $L' = (l'_{ij})$ be an $n \times n$ matrix
for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do $l'_{ij} \leftarrow \infty$
      for $k \leftarrow 1$ **to** n
        do $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

Matrix multiplication (running time)

- $O(n^4)$

Improving the running time:

- No need to compute all the $L^{(m)}$ matrices for $1 \leq m \leq n\text{-}1$.

  We are **interested only in $L^{(n-1)}$**, which is equal to $L^{(m)}$ for all integers $m \geq n\text{-}1$, with assuming that there are no negative cycles.

Improving the running time

Compute the sequence

$L^{(1)} = W,$

$L^{(2)} = W^2 = W \cdot W$,

$L^{(4)} = W^4 = W^2 \cdot W^2,$

$L^{(8)} = W^8 = W^4 \cdot W^4$

...

We need only $\lceil \lg(n\text{-}1) \rceil$ matrix products

➨**Time complexity: O(n^3 lg n)**

Improving running time

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
n ← rows[W]
L^(1) ← W
m ← 1
while m < n − 1
      do L^(2m) ← EXTEND-SHORTEST-PATHS(L^(m), L^(m))
         m ← 2m
return L^(m)
```

Floyd-Warshall algorithm

--suppose there are no negative cycles.

-structure of shortest path

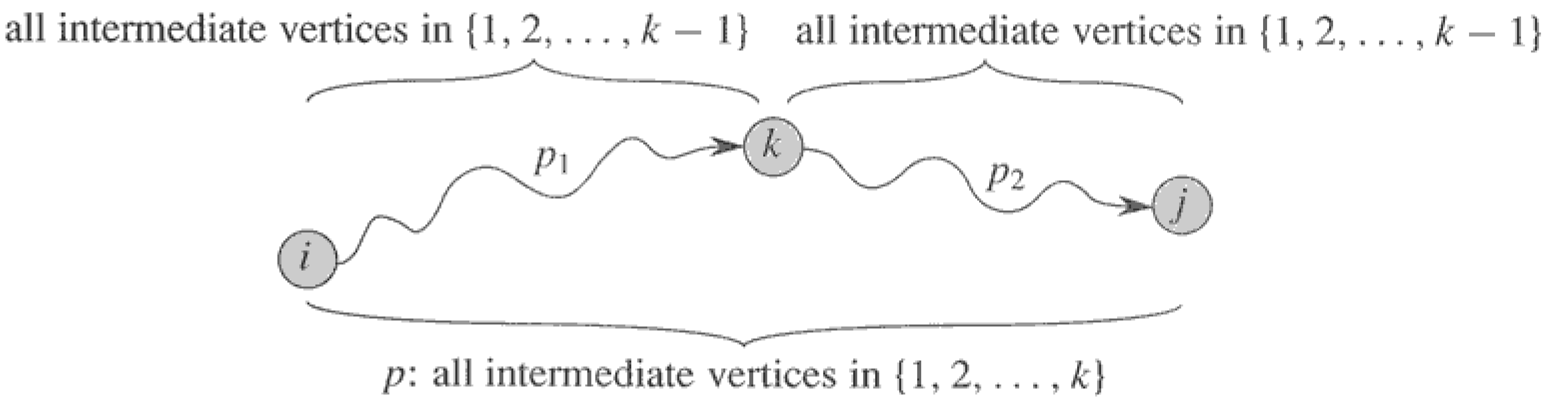


Figure 25.3 Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

Floyd-Warshall algorithm (idea)

- $d_{ij}^{(k)}$: shortest path weight from i to j with **intermediate vertices (excluding *i*, *j*)** from the set $\{1,2,\ldots,k\}$
- Intermediate vertex of a **simple** path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l.

- $d_{ij}^{(0)} = w_{ij}$
  (no intermediate vertices at all)

- How to compute $d_{ij}^{(k)}$ from $D^{(r)}$, for $r < k$

Floyd-Warshall algorithm

-recursive solution

- $d_{ij}^{(0)} = w_{ij}$

  (no intermediate vertices at all)

$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ if $k \geq 1$

Result: $D^{(n)} = (d_{ij}^{(n)})$

(because all intermediate vertices are in the set $\{1, 2, \ldots, n\}$)

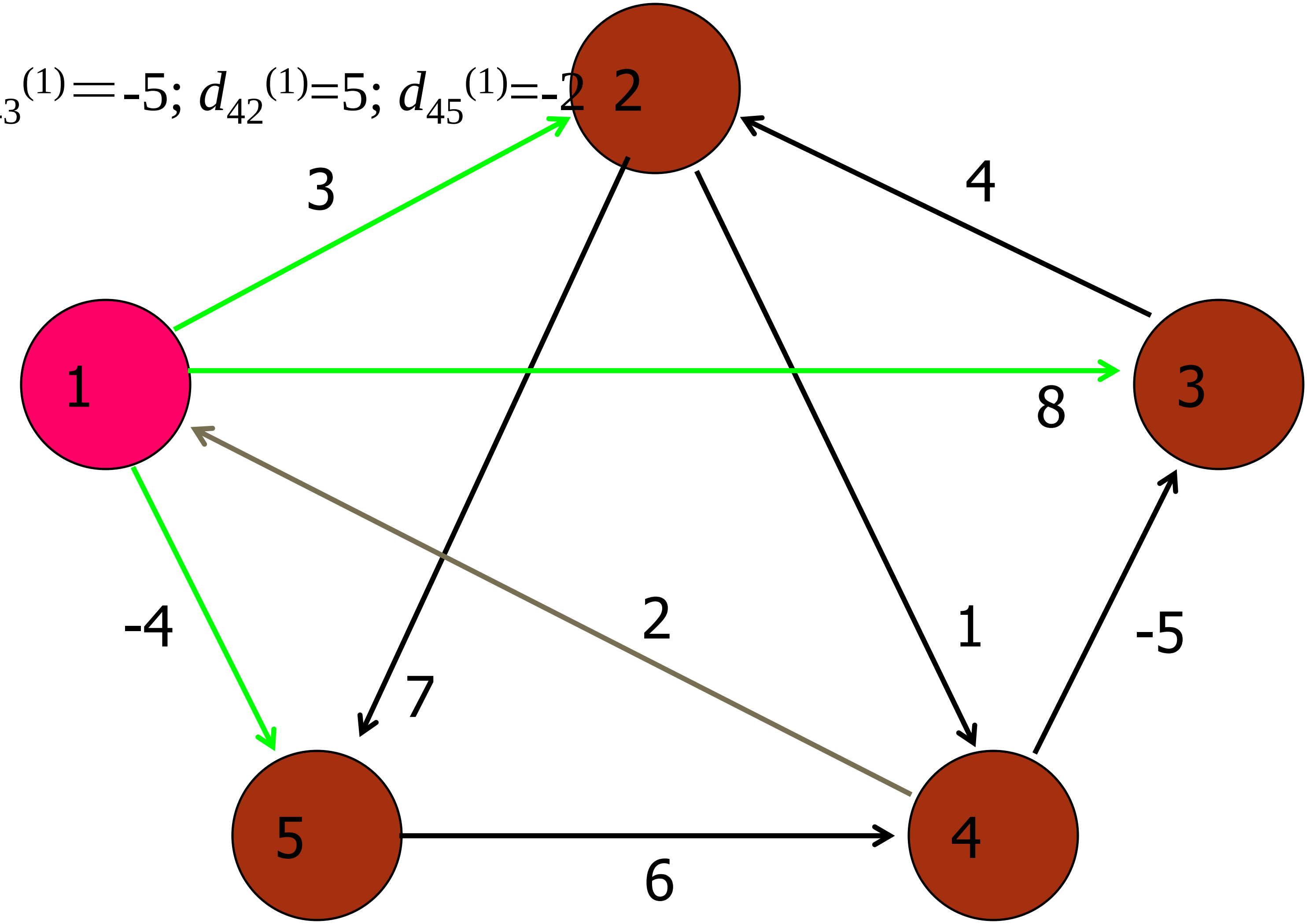
Floyd-Warshall algorithm

-compute shortest-path weights

- Solve the problem stage by stage:
- $D(0)$
- $D(1)$
- $D(2)$
- …
- $D(n)$
- where $D(k)$ contains the shortest path weight with all the **intermediate vertices** from set $\{1,2\ldots,k\}$.

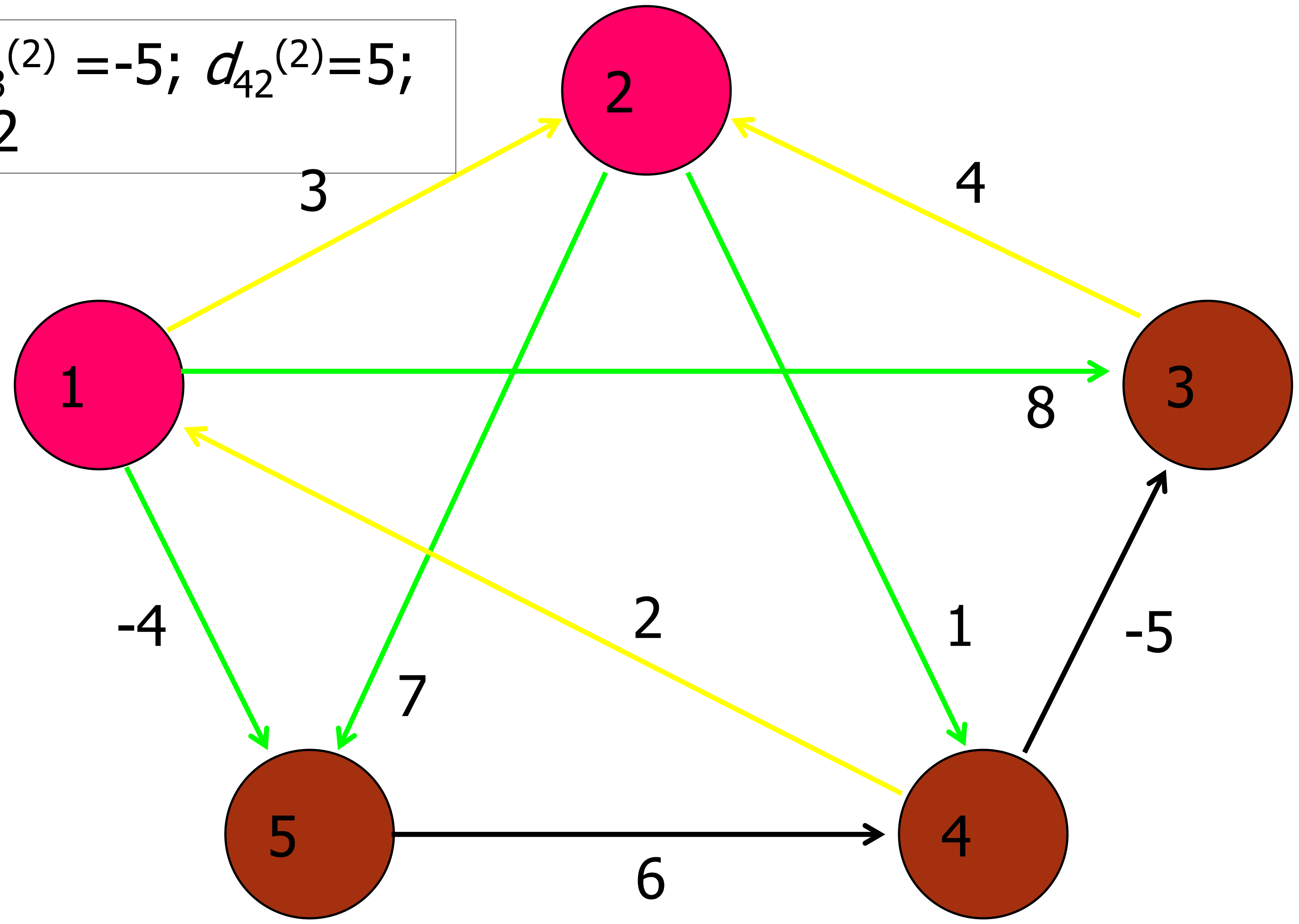
Floyd-Warshall algorithm

k=1: $d_{43}^{(1)}$=-5; $d_{42}^{(1)}$=5; $d_{45}^{(1)}$=-2



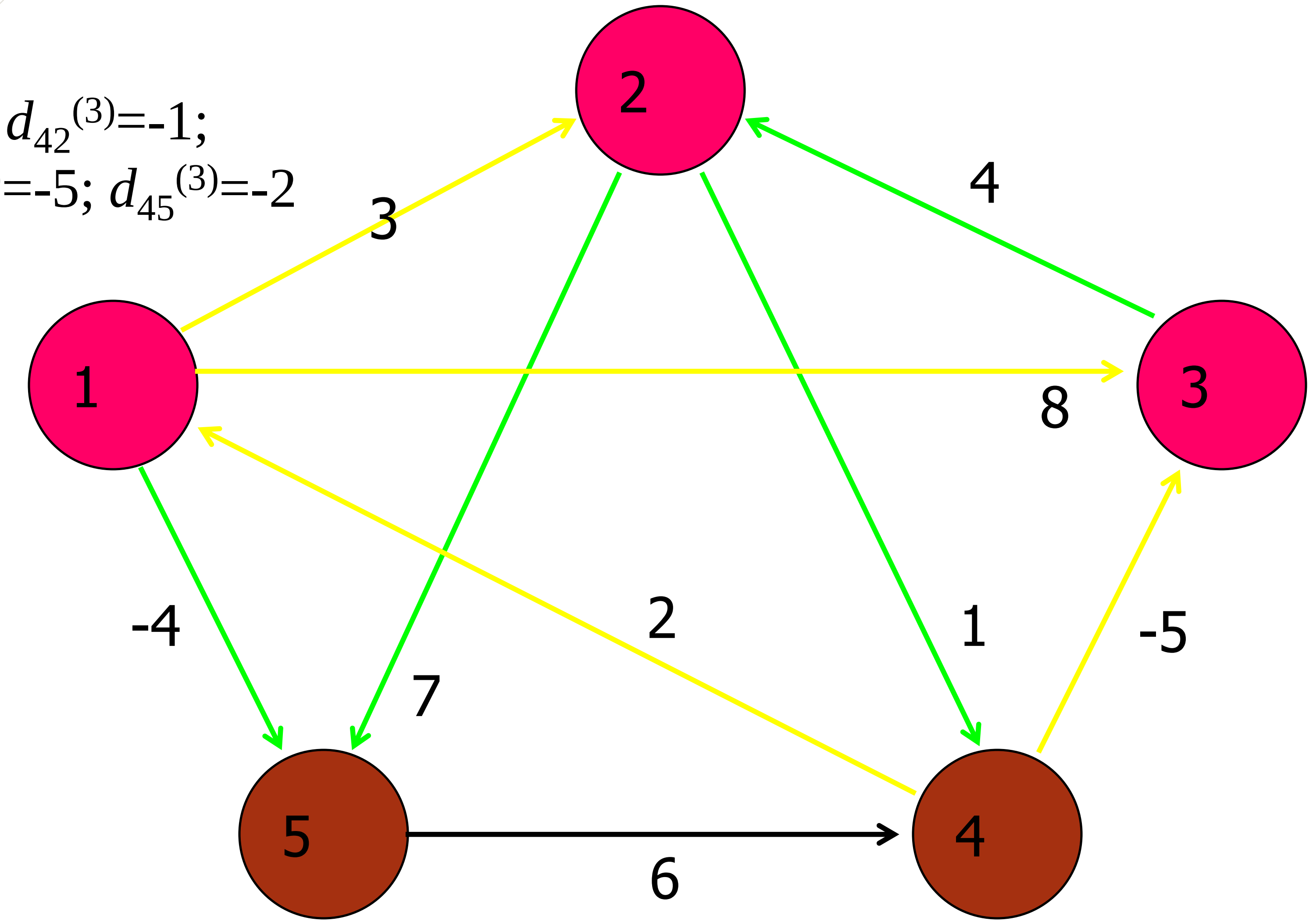
Floyd-Warshall algorithm

k=2: $d_{43}^{(2)}$ =-5; $d_{42}^{(2)}$=5; $d_{45}^{(2)}$=– 2



Floyd-Warshall algorithm

k=3: $d_{42}^{(3)}$=-1;
$d_{43}^{(3)}$=-5; $d_{45}^{(3)}$=-2



Floyd-Warhsall algorithm (pseudo-code)

FLOYD-WARSHALL(W)

$n \leftarrow rows[W]$
$D^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n
    do for $i \leftarrow 1$ **to** n
        do for $j \leftarrow 1$ **to** n
            do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Time complexity: O(n^3)
Space: O(n^3)