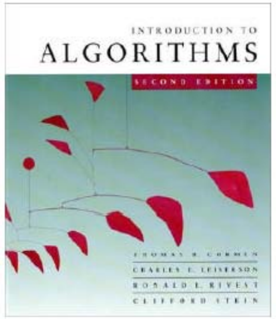
Introduction to Algorithms

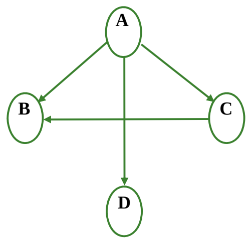


Chapter 22: Elementary Graph Algorithms

Graph Terminology

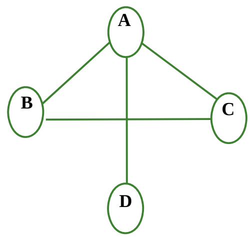
- A graph $G = (V, E)$
  - V = set of vertices
  - E = set of edges
- In an *undirected graph:*
  - *edge*(u, v) = *edge*(v, u)
- In a *directed* graph:
  - *edge*(u,v) goes from vertex u to vertex v, notated $u \rightarrow v$
  - *edge*(u, v) is not the same as *edge*(v, u)

Graph Terminology



Directed graph:

$V = \{A, B, C, D\}$
$E = \{(A,B), (A,C), (A,D), (C,B)\}$

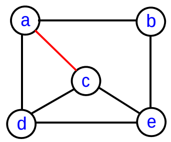


Undirected graph:

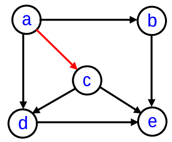
$V = \{A, B, C, D\}$
$E = \{(A,B), (A,C), (A,D), (C,B), (B,A), (C,A), (D,A), (B,C)\}$

Graph Terminology

- ***Adjacent vertices***: connected by an edge
  - Vertex $\boldsymbol{v}$ is adjacent to $\boldsymbol{u}$ if and only if $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$.
  - In an undirected graph with edge $(\boldsymbol{u}, \boldsymbol{v})$, and hence $(\boldsymbol{v}, \boldsymbol{u})$, $\boldsymbol{v}$ is adjacent to $\boldsymbol{u}$ and $\boldsymbol{u}$ is adjacent to $\boldsymbol{v}$.



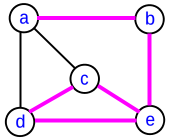
Vertex *a* is adjacent to *c* and vertex *c* is adjacent to *a*



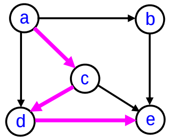
Vertex *c* is adjacent to *a*, but vertex *a* is NOT adjacent to *c*

Graph Terminology

- ***A Path*** in a graph from *u* to *v* is a sequence of edges between vertices $w_0, w_1, \ldots, w_k$, such that $(w_i, w_{i+1}) \in E$, $u = w_0$ and $v = w_k$, for $0 \le i < k$
  - The length of the path is *k*, the number of edges on the path



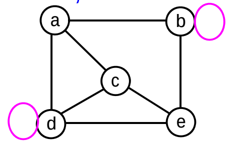
abedce is a path.
cdeb is a path.
bca is NOT a path.



acde is a path.
abec is NOT a path.

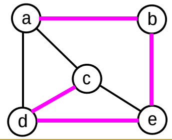
Graph Terminology

- ***Loops***
  - If the graph contains an edge ***(v, v)*** from a vertex to itself, then the path ***v, v*** is sometimes referred to as a ***loop***.



  - The graphs we will consider will generally be loopless.

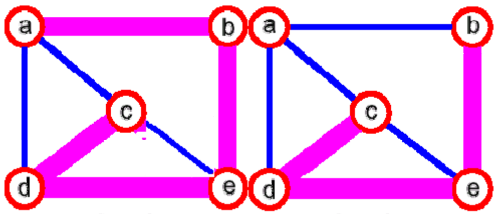
- ***A simple*** path is a path such that ***all vertices are distinct***, except that the first and last could be the same.



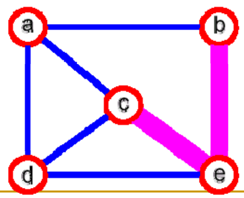
abedc is a simple path.
cdec is a simple path.
abedce is NOT a simple path.

Graph Terminology

- simple path: no repeated vertices



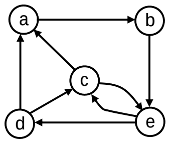
a b e d c   b e d c



b e c

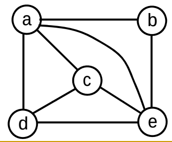
Graph Terminology

- ***Cycles***
  - A ***cycle*** in a **directed graph** is a **path** of length at **least 2** such that the **first** vertex on the path is the same as the **last** one; if the path is **simple**, then the cycle is a ***simple cycle***.



abeda is a simple cycle.
abeceda is a cycle, but is NOT a simple cycle.
abedc is NOT a cycle.

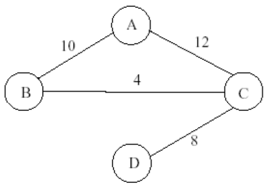
  - A ***cycle*** in a undirected graph
    - A path of length at **least 3** such that the **first** vertex on the path is the same as the **last** one.
    - The edges on the path are **distinct**.



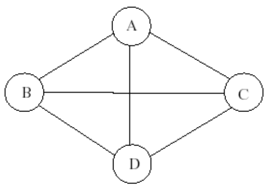
aba is NOT a cycle.
abedceda is NOT a cycle.
abedcea is a cycle, but NOT simple.
abea is a simple cycle.

Graph Terminology

- If each edge in the graph carries a value, then the graph is called ***weighted graph***.
  - A weighted graph is a graph $G = (V, E, W)$, where each edge, $e \in E$ is assigned a real valued weight, $W(e)$.
- A ***complete graph*** is a graph with an edge between every pair of vertices.
  - A graph is called ***complete graph*** if every vertex is adjacent to every other vertex.



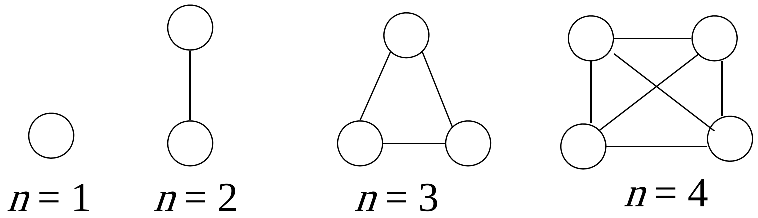
weighted graph



complete graph

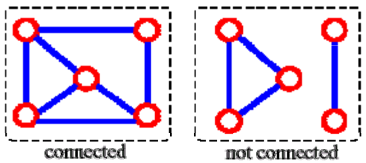
Graph Terminology

- Complete Undirected Graph
  - has all possible edges



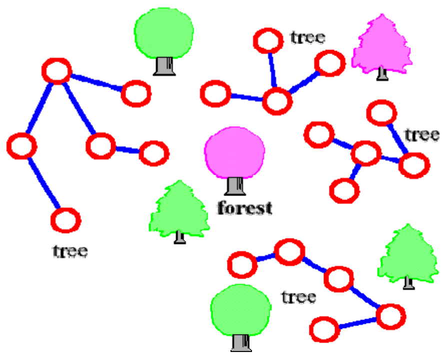
Graph Terminology

- connected graph: any two vertices are connected by some path
  - An undirected graph is ***connected*** if, for every pair of vertices *u* and *v* there is a path from *u* to *v*.

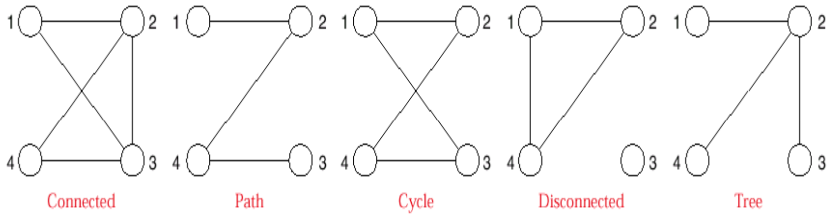


Graph Terminology

- tree - connected graph without cycles
- forest - collection of trees

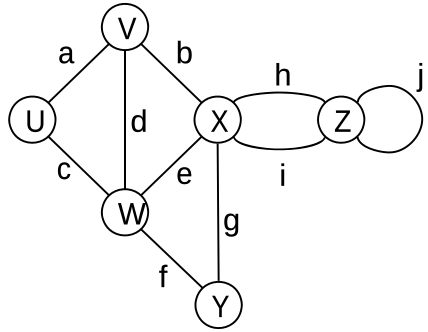


Graph Terminology



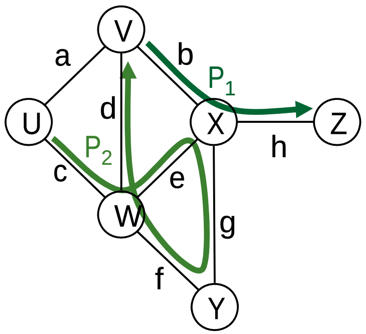
Graph Terminology

- ***End vertices*** (or ***endpoints***) of an ***edge a***
  - **U** and **V** are the endpoints of **a**
- Edges incident on a vertex V
  - **a**, **d**, and **b** are incident on **V**
- **Adjacent vertices**
  - **U** and **V** are adjacent
- **Degree of a vertex X**
  - **X** has degree 5
- **Parallel edges**
  - **h** and **i** are parallel edges
- **Self-loop**
  - **j** is a self-loop



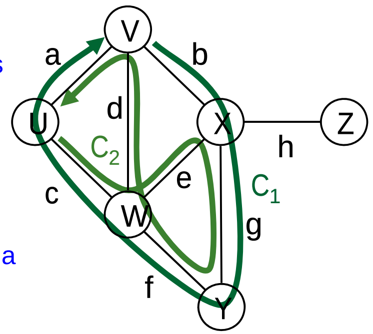
Graph Terminology

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
- **Simple path**
  - path such that all its vertices and edges are **distinct**.
- **Examples**
  - P_1 = (V, X, Z) is a simple path.
  - P_2 = (U, W, X, Y, W, V) is a path that is not simple.



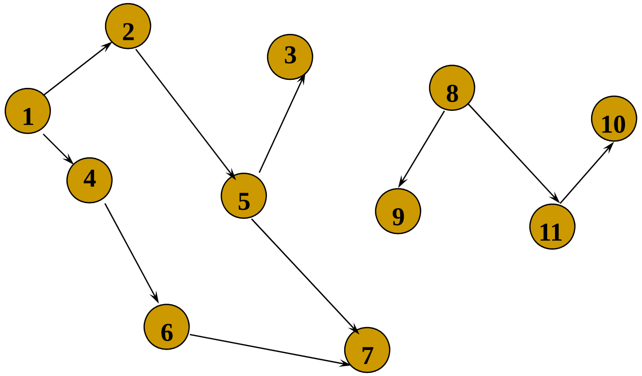
Graph Terminology

- **Cycle**
  - circular sequence of alternating vertices and edges
- Simple cycle
  - cycle such that all its vertices and edges are **distinct**
- Examples
  - C_1 = (V, X, Y, W, U, V) is a simple cycle
  - C_2 = (U, W, X, Y, W, V, U) is a cycle that is not simple



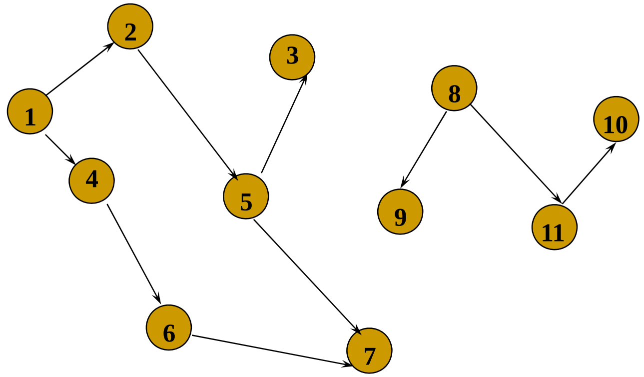
In-Degree of a Vertex

- **in-degree** is number of incoming edges
  - indegree(2) = 1, indegree(8) = 0



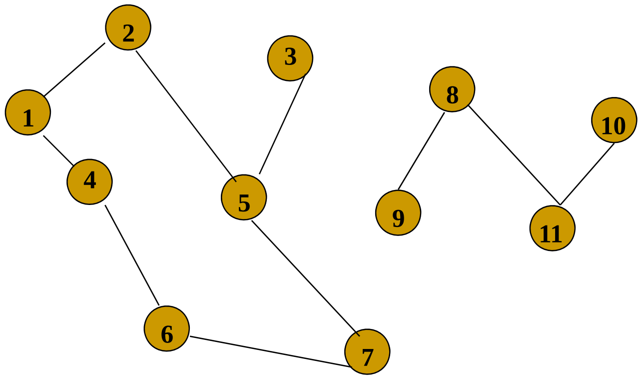
Out-Degree of a Vertex

- **out-degree** is number of outbound edges
  - outdegree(2) = 1, outdegree(8) = 2



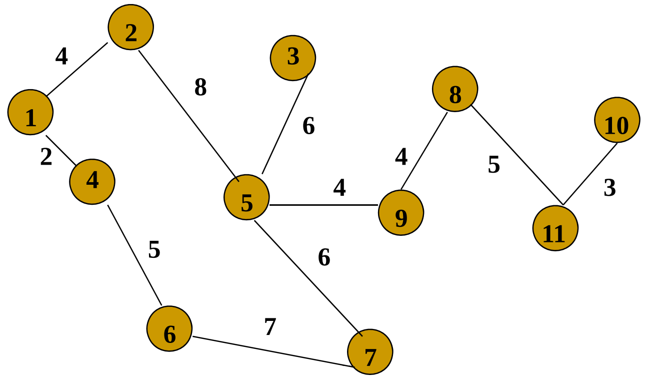
Applications: Communication Network

- *vertex* = city, *edge* = communication link



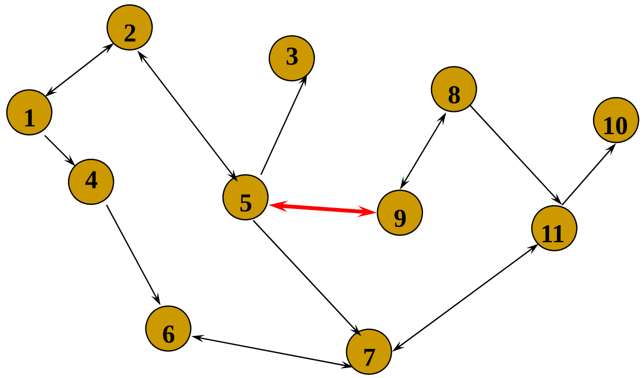
Driving Distance/Time Map

- *vertex* = city,
- *edge* weight = distance/time



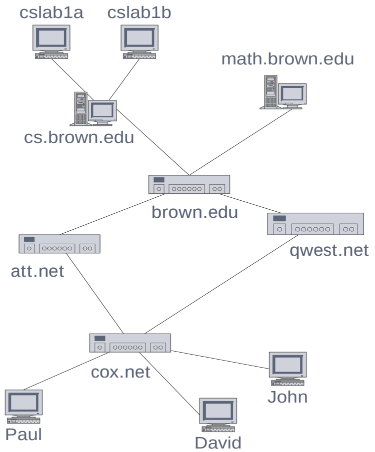
Street Map

- Some streets are one way
- A ***bidirectional*** link represented by 2 directed edge
  - (5, 9) (9, 5)



Computer Networks

- Electronic circuits
  - Printed circuit board
- Computer networks
  - Local area network
  - Internet
  - Web



Graphs

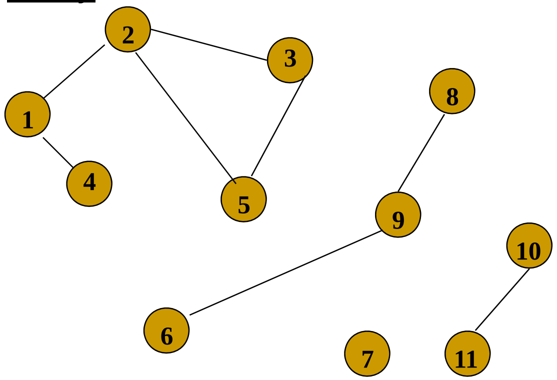
- We will typically express running times in terms of
  - $|V|$ = number of vertices, and
  - $|E|$ = number of edges
  - If $|E| \approx |V|^2$ the graph is ***dense***
  - If $|E| \approx |V|$ the graph is ***sparse***
- If you know you are dealing with dense or sparse graphs, different data structures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

Graph Search Methods

- Many graph problems solved using a search method
  - Path from one vertex to another
  - Is the graph connected?
  - etc.
- Commonly used search methods:
  - Breadth-first search
  - Depth-first search

Graph Search Methods

- A vertex ***u*** is reachable from vertex ***v*** iff there is a path from ***v*** to ***u***.
- A search method starts at a given vertex ***v*** and visits <u>every</u> vertex that is reachable from ***v***.



Breadth-First Search

- Visit start vertex (s) and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.
- All vertices reachable from the start vertex (s) (including the start vertex) are visited.

Breadth-First Search

- Again will associate vertex “colors” to guide the algorithm
  - **White** vertices have not been discovered
    - All vertices start out white
  - **Green** 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 green vertices
- Explore vertices by scanning adjacency list of green vertices

Breadth-First Search

```
BFS(G, s) {
    // initialize vertices;
    for each u ∈ V(G) – {s}{
        do color[u] = WHITE
           d[u] = ∞              // distance from s to u
           p[u] = NIL            // predecessor or parent of u
    }
    color[s] = GREEN
    d[s] = 0
    p[s] = NIL
    Q = Empty;
    Enqueue (Q,s);               // Q is a queue; initialize to s
    while (Q not empty) {
        u = Dequeue(Q);
        for each v ∈ adj[u] {
            if (color[v] == WHITE)
                color[v] = GREEN;
                d[v] = d[u] + 1;
                p[v] = u;
                Enqueue(Q, v);
        }
        color[u] = BLACK;
    }
}
```

What does `d[v]` *represent?*

What does `p[v]` *represent?*

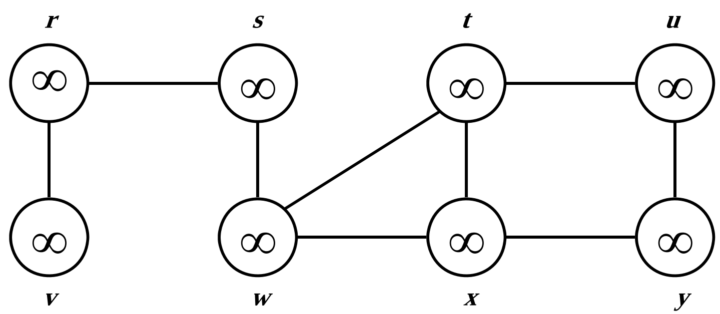
Breadth-First Search

- Lines 1-4 paint every vertex white, set d[u] to be infinity for each vertex (u), and set p[u] the parent of every vertex to be NIL.
- Line 5 paints the source vertex (s) green.
- Line 6 initializes d[s] to 0.
- Line 7 sets the parent of the source to be NIL.
- Lines 8-9 initialize Q to the queue containing just the vertex (s).
- The while loop of lines 10-18 iterates as long as there remain green vertices, which are discovered vertices that have not yet had their adjacency lists fully examined.
  - This while loop maintains the test in line 10, the queue Q consists of the set of the green vertices.

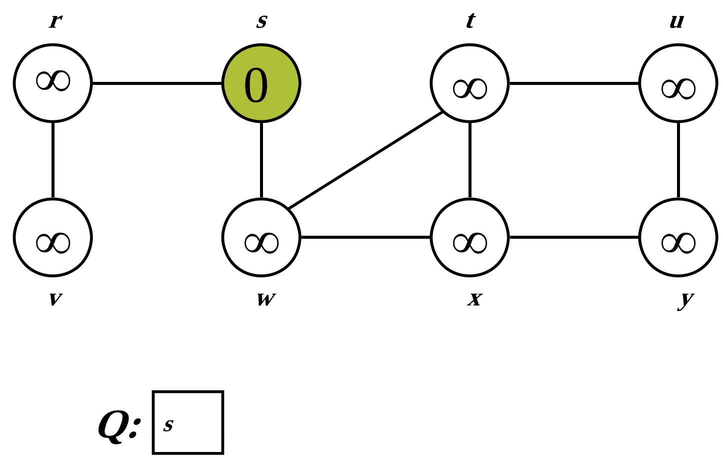
Breadth-First Search

- Prior to the first iteration in line 10, the only green vertex, and the only vertex in Q, is the source vertex (s).
- Line 11 determines the green vertex (u) at the head of the queue Q and removes it from Q.
- The for loop of lines 12-17 considers each vertex (v) in the adjacency list of (u).
- If (v) is white, then it has not yet been discovered, and the algorithm discovers it by executing lines 14-17.
  - It is first greened, and its distance d[v] is set to d[u]+1.
  - Then, u is recorded as its parent.
  - Finally, it is placed at the tail of the queue Q.
- When all the vertices on (u's) adjacency list have been examined, u is blackened in line 18.

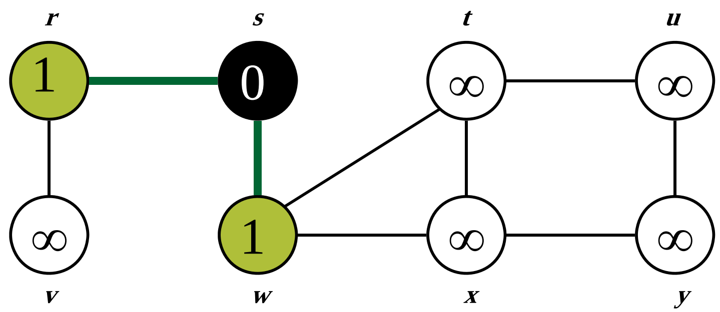
Breadth-First Search: Example



Breadth-First Search: Example

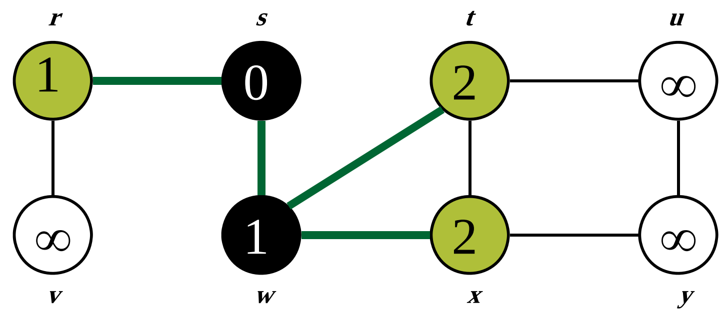


Breadth-First Search: Example



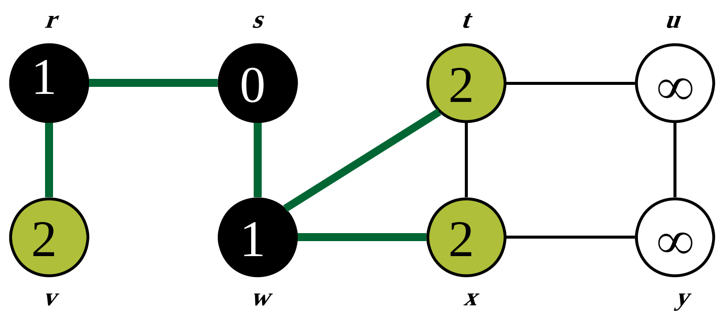
Q: | *w* | *r* |

Breadth-First Search: Example



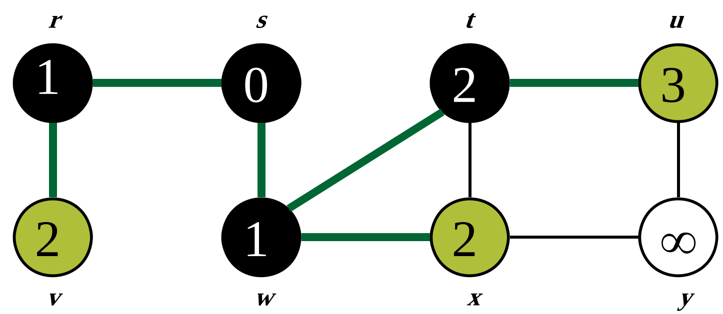
Q: | *r* | *t* | *x* |

Breadth-First Search: Example



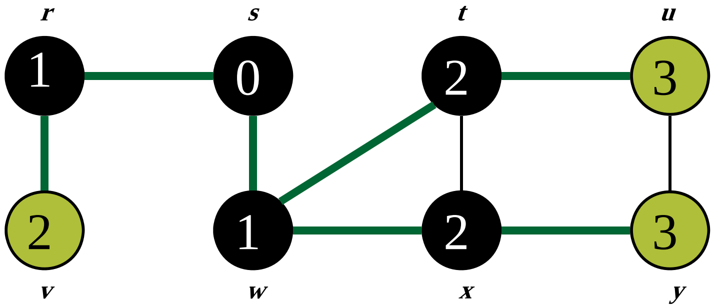
Q: | *t* | *x* | *v* |

Breadth-First Search: Example



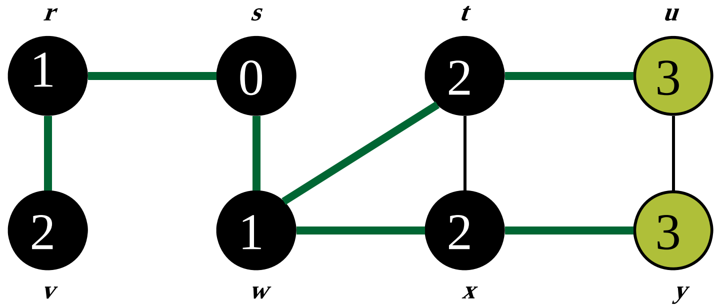
Q: | *x* | *v* | *u* |

Breadth-First Search: Example



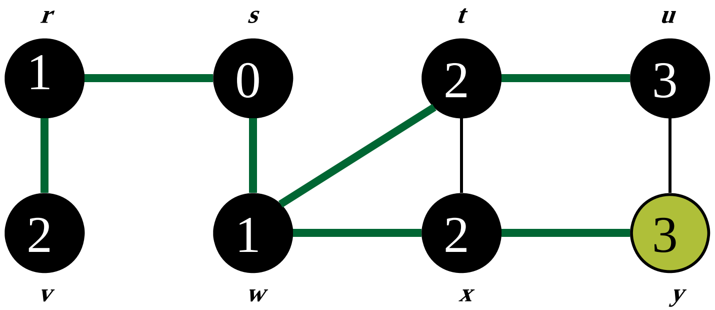
Q: | *v* | *u* | *y* |

Breadth-First Search: Example



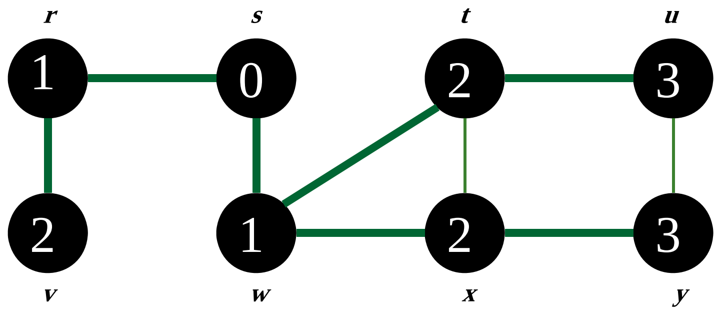
Q: | *u* | *y* |

Breadth-First Search: Example



Q: | *y* |

Breadth-First Search: Example



Q: Ø

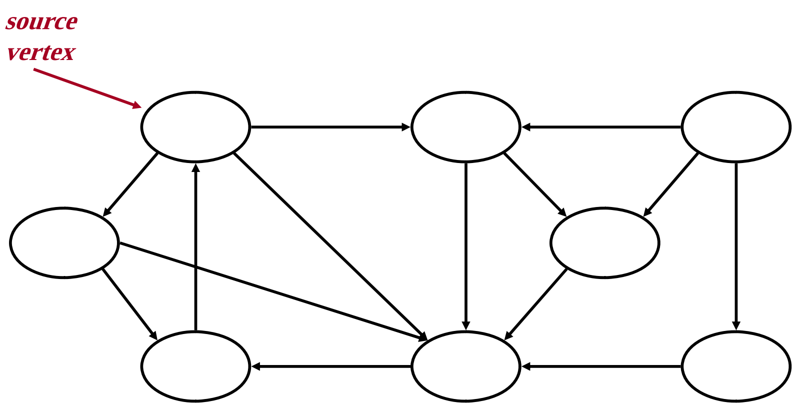
Depth-First Search

- *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

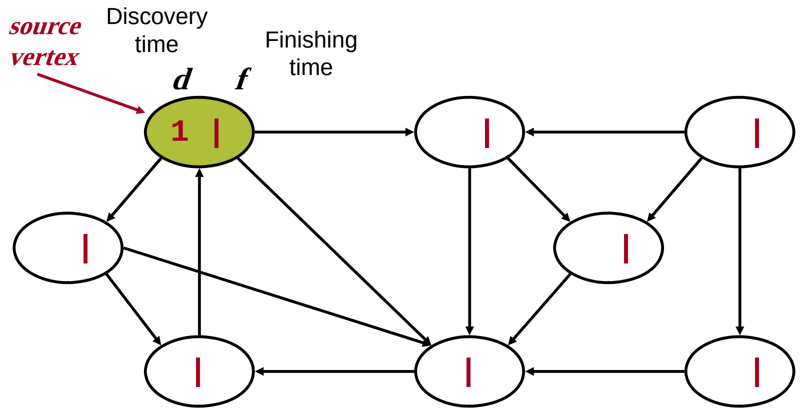
Depth-First Search

- Initialize
  - color all vertices white
- Visit each and every white vertex using DFS-Visit
- Each call to **DFS-Visit(*u*)** **roots a new tree** of the depth-first forest at vertex *u*
- A vertex is **white** if it is undiscovered
- A vertex is **green** if it has been discovered but not all of its edges have been discovered
- A vertex is **black** after all of its adjacent vertices have been discovered (the adj. list was examined completely)

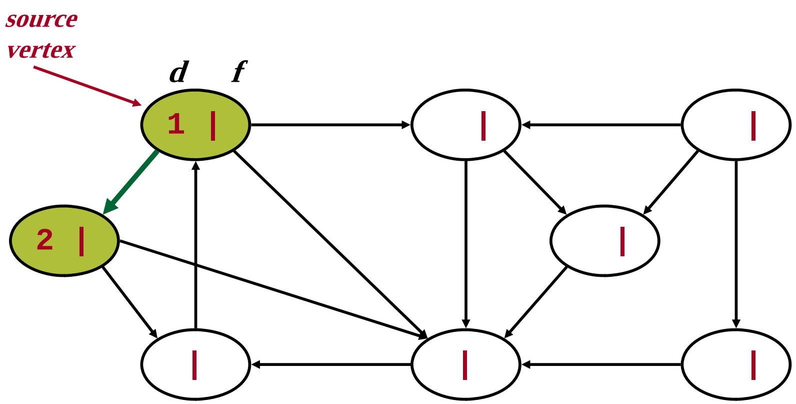
DFS Example



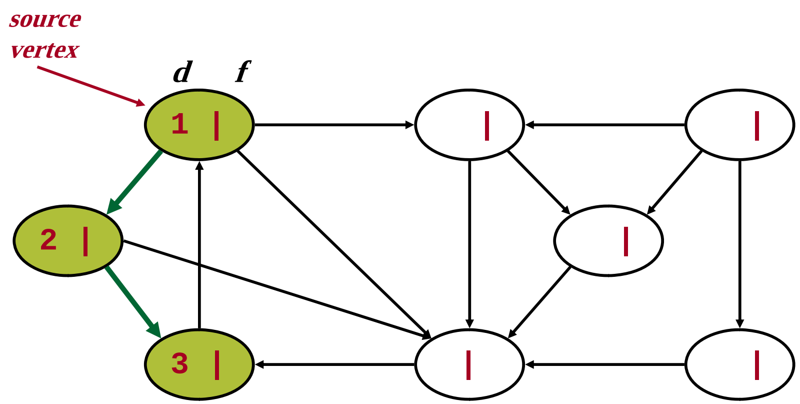
DFS Example



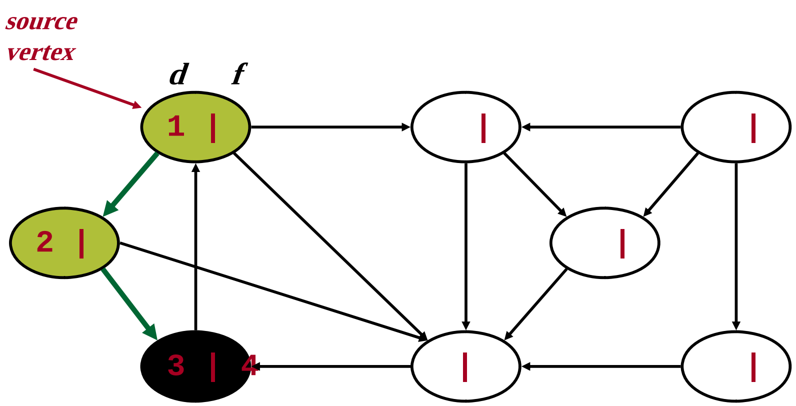
DFS Example



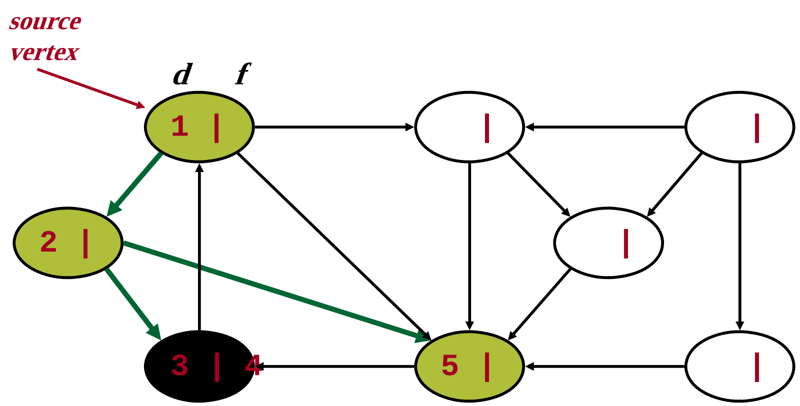
DFS Example



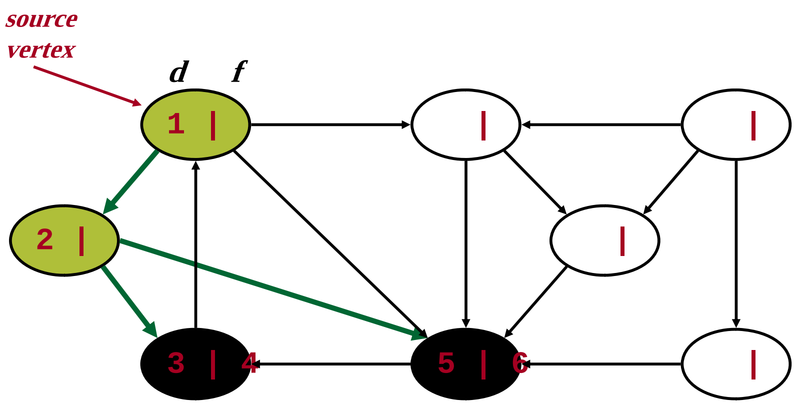
DFS Example



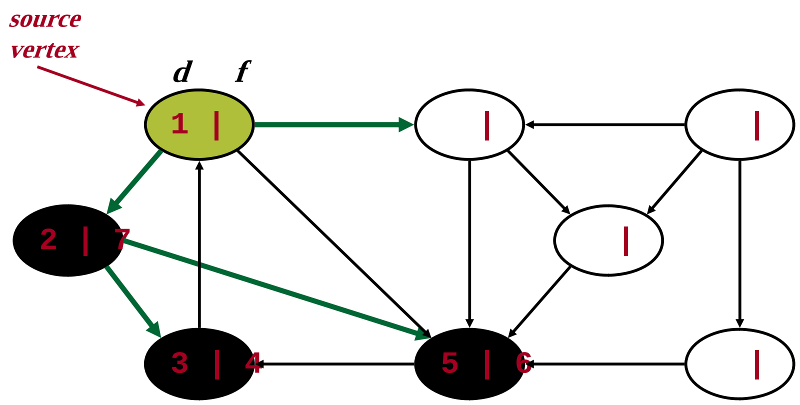
DFS Example



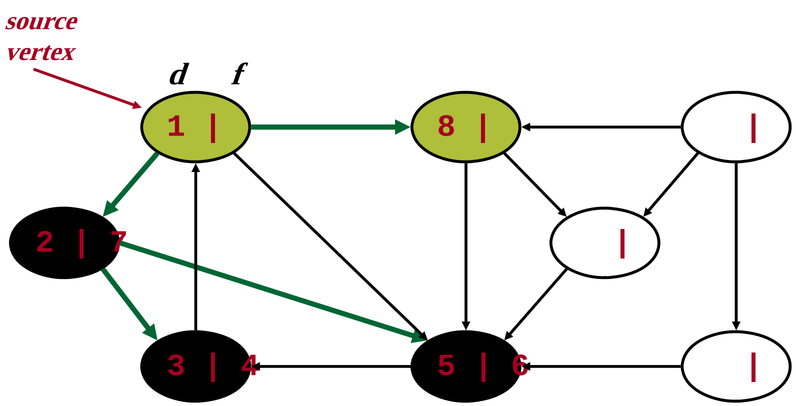
DFS Example



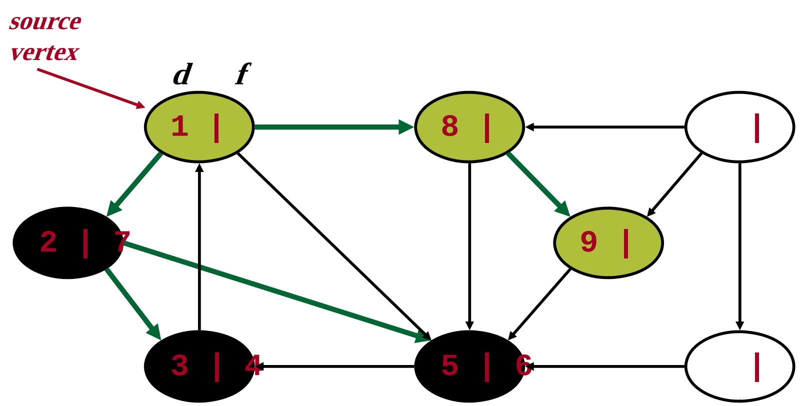
DFS Example



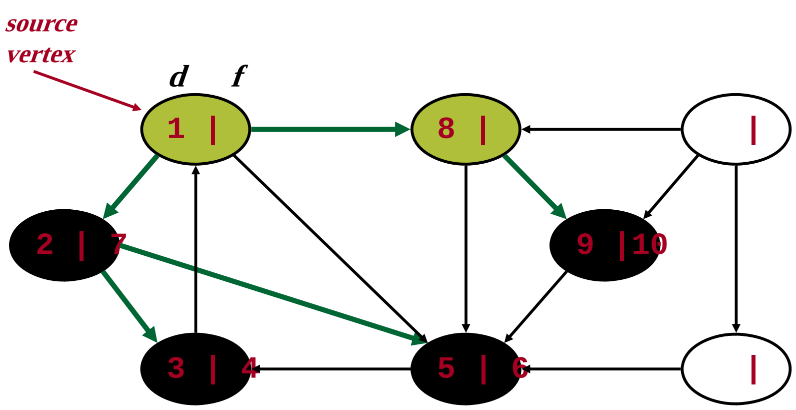
DFS Example



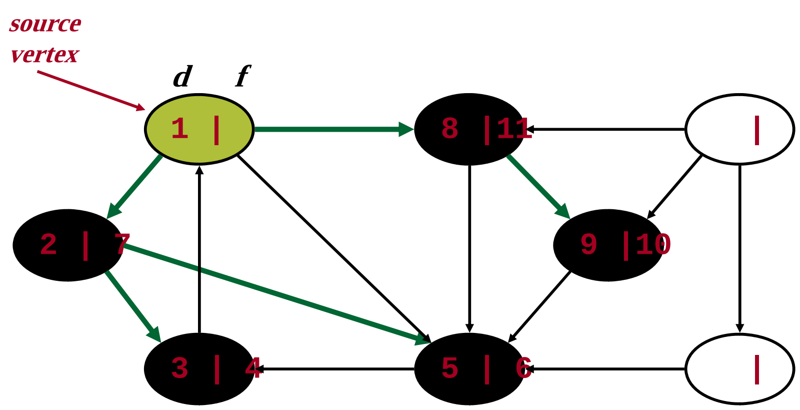
DFS Example



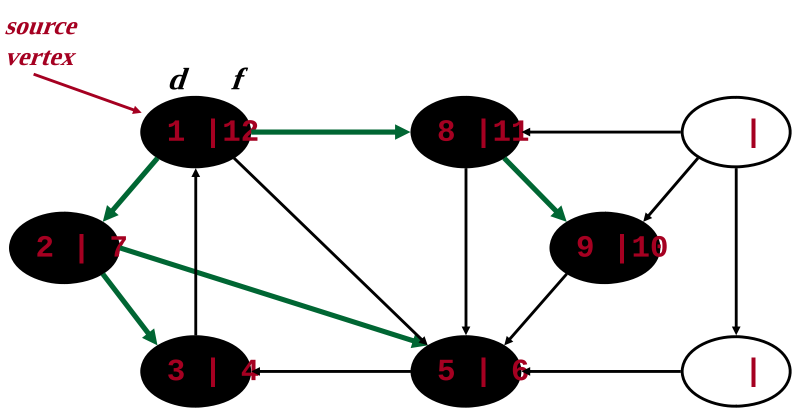
DFS Example



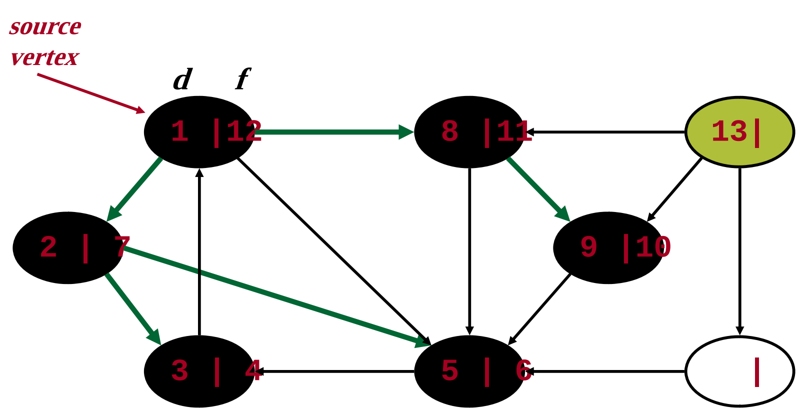
DFS Example



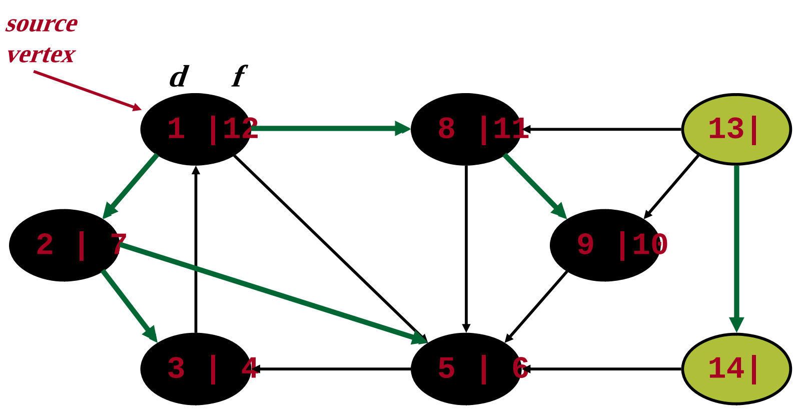
DFS Example



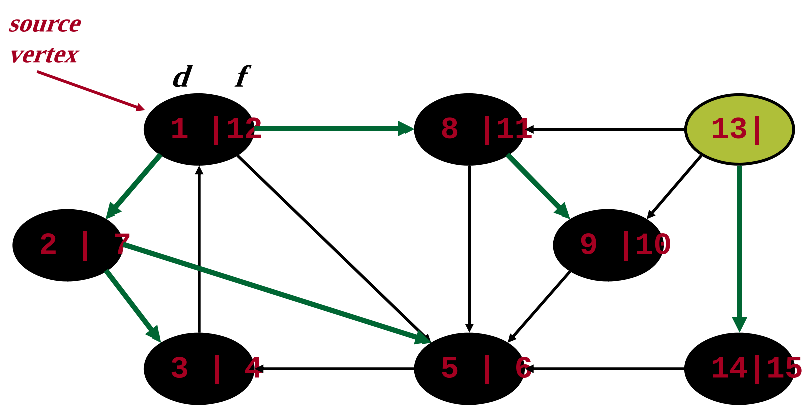
DFS Example



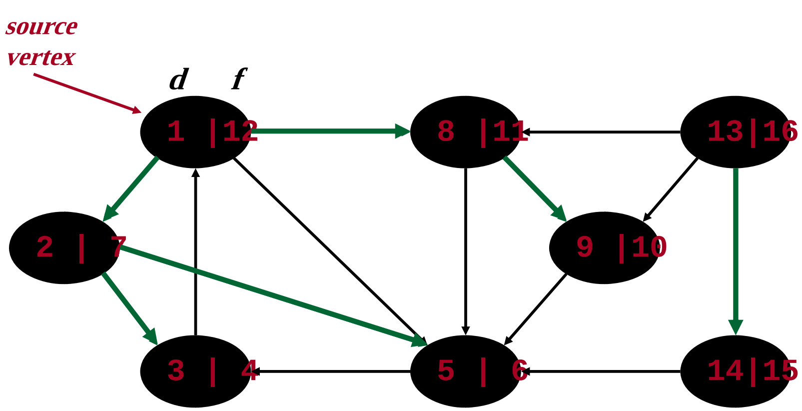
DFS Example



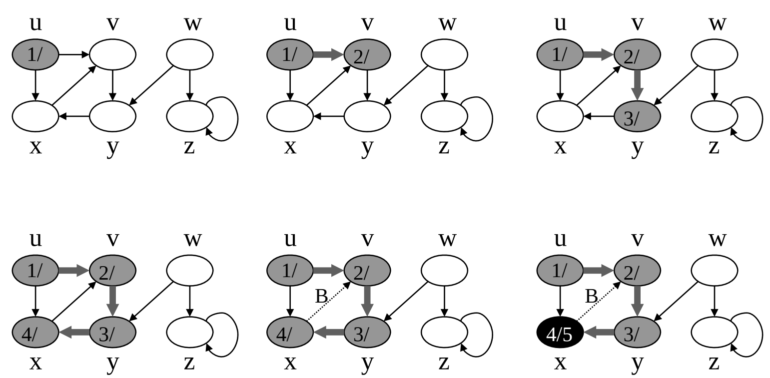
DFS Example



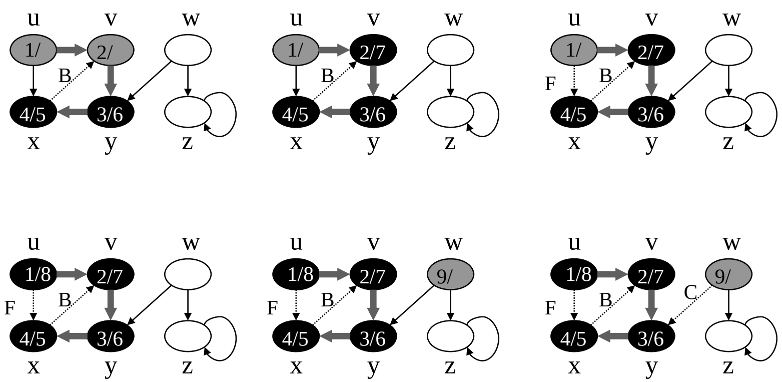
DFS Example



DFS Example



DFS Example



DFS Example